Цифровая схемотехника и архитектура компьютера

второе издание

Дэвид М. Хэррис и Сара Л. Хэррис

Издательство
Morgan Kaufman

Спонсор перевода - Imagination Technologies
www.imgtec.com
Переведено командой из компаний и университетов России, Украины, США и Великобритании

This edition of Digital Design and Computer Architecture by David Money Harris and Sarah L Harris is published by arrangement with ELSEVIER INC., a Delaware corporation having its principal place of business at 360 Park Avenue South, New York, NY 10010, USA

Это издание книги Дэвида Мани Харриса и Сары Л. Харрис “Цифровая схемотехника и архитектура компьютера” публикуется по соглашению с ELSEVIER INC., Делавэрской корпорацией, которая осуществляет основную деятельность по адресу 360 Park Avenue South, New York, NY 10010, USA.

ISBN 978-0-12-394424-5 (Original 2nd Edition)

ISBN 978-0-9954839-1-0 (Russian version)

Переведено командой из компаний и университетов России, Украины, США и Великобритании

Имя и фамилия	Организация
Александр Барабанов	доцент Киевского КНУ
Александр Биргер	на пенсии, ранее в Cadence Design Systems
Александр Леденев	Apple, OS X
Александр Телешов	Модуль
Александра Богданова	МИФИ
Алексей Евгеньевич Платунов	профессор ИТМО
Алексей Лавров	аспирант Принстона
Алексей Фрунзе	Imagination Technologies
Андрей Лихолит	eASIC
Андрей Терехов	директор НИИ информационных технологий СПбГУ
Анна Степашкина	Самарский СГАУ
Антон Моисеев	НГТУ им Р.Е.Алексеева, ФИВТ МФТИ
Валерий Казанцев	Synopsys, процессоры ARC
Виктория Ведина	Imagination Technologies
Владимир Рытиков	Apple
Владимир Серяпов	RusBITech
Владимир Хаханов	декан харьковского НУ радиоэлектроники
Григорий Жихарев	МЦСТ
Денис Хартиков	NVidia
Дмитрий Миронов	Runtime Design Automation
Екатерина Степанова	Самарский СГАУ
Иван Графский	выпускник МИФИ
Илья Александрович Кудрявцев	декан Самарского СГАУ

Имя и фамилия	Организация
Константин Евтушенко	Модуль
Константин Петров	НИИСИ РАН
Леонид Брухис	Synopsys, группа по эмуляторам
Леонид Егошин	Imagination Technologies
Линк Джепсон	Imagination Technologies
Максим Горбунов	НИИСИ РАН
Максим Матуско	МИФИ
Максим Парфенов	Marvell Semiconductor, документация
Михаил Барских	НИИСИ РАН
Нина Захарчук	корректор
Павел Валерьевич Кустарев	доцент ИТМО
Петр Чибисов	НИИСИ РАН
Роберт Оуэн	Imagination Technologies
Руслан Тихонов	amperka.ru
Сергей Аряшев	заведующий отделом НИИСИ РАН
Сергей Чураев	AMD, ИТМО
Симон Атанасян	Imagination Technologies
Тимур Палташев	старший менеджер AMD, ИТМО
Эдуард Стародубцев	доцент ИТМО
Юрий Панчул	Imagination Technologies, MIPS processors
Юрий Холопов	кафедра МФТИ в ИТМиВТ
Юрий Шейнин	СПб ГУАП

ПОХВАЛЬНЫЕ ОТЗЫВЫ НА КНИГУ
ЦИФРОВАЯ СХЕМОТЕХНИКА И АРХИТЕКТУРА КОМПЬЮТЕРА

Авторы книги вывели преподавание предмета на качественно иной уровень, создав более доступный для понимания и наглядный учебник, чем "Устройство и проектирование компьютеров" ("Computer Organization and Design"), и описав в нем в деталях, как спроектировать микропроцессор архитектуры MIPS с помощью языков SystemVerilog и VHDL. Текст окажется особенно полезным для студентов, которые в процессе обучения столкнутся с разработкой больших цифровых систем на современных ПЛИС.

Дэвид А. Паттерсон, Калифорнийский Университет в Беркли (прим. переводчика: Дэвид Паттерсон – соавтор вышеупомянутого учебника "Устройство и проектирование компьютеров").

Книга дает свежий взгляд на старую дисциплину. Многие учебники напоминают неухоженные заросли кустарника, но авторы данного учебника сумели отстричь засохшие ветви, сохранив основы и представив их в современном контексте. Эта книга поможет студентам справиться с техническими испытаниями завтрашнего дня.

Джим Френзел, Университет Айдахо.

Книга написана в информативном приятном для чтения стиле. Материал представлен на хорошем уровне для введения в проектирование компьютеров и содержит множество полезных диаграмм. Комбинационные схемы, микроархитектура и системы памяти изложены особенно хорошо.

Джеймс Пинтер-Люк, Колледж им. Дональда Маккенны, Клермонт.

Харрис и Харрис написали очень ясную и легкую для понимания книгу. Упражнения хорошо разработаны, а примеры из реальной практики являются замечательным дополнением. Длинные и вводящие в заблуждение объяснения, часто встречающиеся в подобных книгах, здесь отсутствуют. Очевидно, что авторы посвятили много времени и усилий созданию доступного текста. Я настоятельно рекомендую книгу.

Пейи Чжао, Университет Чепмена.

Харрис и Харрис написали первую книгу, которая успешно совмещает проектирование цифровых систем и архитектуру компьютеров. Книга – долгожданное учебное пособие, в котором подробно рассматривается проектирование цифровых систем и в фантастических деталях объясняется архитектура MIPS. Я настоятельно рекомендую эту книгу.

Джеймс Э. Стайн, Мл., Университет Оклахомы

Это великолепная книга. Авторы органично связывают все важные в проектировании микропроцессоров элементы – транзисторы, схемы, логические элементы, конечные автоматы, память, арифметические блоки – и получают компьютерную архитектуру. Этот текст является незаменимым руководством для понимания, как последовательно разрабатывать сложные системы.

Джеха Ким, Рамбус Инк.

Это очень хорошо написанная книга, которая будет полезна как молодым инженерам, изучающим предмет впервые, так и опытным инженерам, которые смогут использовать ее в качестве справочника. Я настоятельно рекомендую ее.

А. Утку Дирил, Корпорация Энвидиа.

Сведения об авторах

Дэвид Мани Хэррис (David Money Harris) – доцент в колледже им. Харви Мадда (Harvey Mudd College). Он получил ученую степень кандидата наук по электронике в Стэнфордском университете и степень магистра по электронике и информатике в Массачусетском технологическом институте (MIT). Перед Стэнфордом он работал в компании Интел (Intel) в качестве схемотехника и разработчика логики для процессоров Итаниум и Пентиум 2 (Itanium and Pentium II). Впоследствии он работал консультантом в Сан Майкросистемз (Sun Microsystems), Хьюлетт-Паккард (Hewlett-Packard), Эванс энд Сазерленд (Evans & Sutherland) и других компаниях.

Увлечения Дэвида включают в себя преподавание, разработку чипов и активный отдых на природе. В свободное от работы время он занимается пешим туризмом, скалолазанием и альпинизмом. Особенно он любит длинные прогулки с сыном Абрахамом, который родился, когда Дэвид начал работать над этой книгой. Дэвид имеет более десяти патентов и является автором трех других учебников по проектированию чипов, а также двух путеводителей по горам Южной Калифорнии.

Сара Л. Хэррис (Sarah L. Harris) – доцент в колледже им. Харви Мадда (Harvey Mudd College). Она получила степени магистра и кандидата наук по электронике в Стэнфордском университете и степень бакалавра по электронике и вычислительной технике в университете Брайама Янга (BrighamYoung University). Сара также работала в компаниях Хьюлетт-Паккард, Суперкомпьютерном Центре Сан-Диего (San Diego Supercomputer Center), Энвидиа (Nvidia) и исследовательском отделе компании Майкрософт (Microsoft Research) в Пекине.

Интересы Сары не ограничиваются преподаванием,изучением и разработкой новых технологий, она также любит путешествовать, увлекается виндсерфингом, скалолазанием и игрой на гитаре. Среди ее недавних начинаний можно отметить исследования в области интерфейсов, позволяющих проектировать цифровые электрические схемы простыми рисунками от руки, работу в качестве научного корреспондента для филиала Национального Общественного Радио (National Public Radio) и обучение кайтсерфингу. Сара говорит на четырех языках и собирается изучить еще несколько в ближайшем будущем.

Моей семье, Дженнифер, Абрахаму, Сэмюелю и Бенджамину
– DMH

Ивану и Окаану, бросающим вызов логике
– SLH

Оглавление

Предисловие к изданию на русском языке

История развития вычислительной техники в СССР насчитывает практически столько же лет, как и в США, так как разработка быстродействующих компьютеров являлась неотъемлемой частью технологического соперничества двух сверхдержав. Вскоре после разработки первого американского компьютера общего назначения ENIAC (1943-1947), в СССР была разработана МЭСМ (Малая электронная счетная машина, 1947–1950), самый быстрый компьютер в континентальной Европе того времени.

С развалом СССР для вычислительной техники наступили трудные времена, когда всем новым государствам она оказалась практически не нужна. Рыночные реформаторы считали, что все проблемы можно решить закупками на мировом рынке и современная электронная промышленность является излишней обузой для страны в условиях перестройки экономики.

Компьютерная и электронная инженерия оказались невостребованными в отечественной промышленности и устойчиво деградировали с 1990-х

и до середины 2000-х годов, когда руководителям страны стала очевидной невозможность решения проблем информатизации страны за счет импорта без угрозы полной потери технологического суверенитета, совмещенной с возможным подрывом национальной безопасности.

Разработка собственных архитектур электронно-вычислительных машин (ЭВМ) и микропроцессоров были практически остановлены, и немногочисленные выжившие конструкторско-технологические структуры занимались, в основном, клонированием микропроцессоров ведущих мировых производителей в пределах выделенных весьма ограниченных бюджетов. Чудесным исключением можно назвать микропроцессоры с отечественной архитектурой «Эльбрус», которые разрабатывают и развивают ОАО «ИНЭУМ им. И.С. Брука» и фирма «МЦСТ».

Разумеется, это привело к сужению сферы применения современной компьютерной инженерии в рамках этого круга предприятий, которым были доступны лицензии на производство и соответствующие инструментальные средства. Кроме того, такое состояние сказалось на вкладе отечественной электроники в мировую индустрию. Стал актуальным вопрос: есть ли в России электроника?

Если сравнивать с Россией 1913 года, то электроника в стране, безусловно, была. Но если сравнивать с мировой индустрией – то ее практически не было. Общий объем производства электроники в России в 2008 году составлял восемь миллиардов долларов (данные Ассоциации производителей электронной продукции РФ), то есть всего 0,4% от мирового рынка, объем которого составлял более двух триллионов долларов. Население России (142 млн) составляло тогда 2,14% населения планеты, то есть уровень развития электроники был в пять раз ниже «порога самоуважения нации» (2,14/0,4 = 5,35). И это происходило при наличии кадрового корпуса инженеров и работающей высшей инженерной школе.

Начиная с середины 2000-х годов начали издаваться большие учебники по компьютерной архитектуре (архитектуре микропроцессоров), которые являлись переводами популярных американских или европейских университетских учебников. Например, издательство «Питер» уже издало, по крайней мере, два таких учебника:

- *Паттерсон Д., Хеннесси Дж. Архитектура компьютера и проектирование компьютерных систем. 4-е изд. СПб.: Питер, 2012. – ISBN 978-5-459-00291-1;*

- *Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. СПб.: Питер, 2014. – ISBN 978-5-496-00337-7 (а также предыдущие издания).*

К общим недостаткам этих популярных книг относится их описательный характер по отношению к архитектуре микропроцессоров, также как и невозможность спроектировать и построить собственный микропроцессор, базируясь на изучении материала, представленного в этих учебниках. К недостаткам русскоязычных изданий относятся ошибки в переводе специальной терминологии и частая потеря технического смысла в предложениях, также как и относительно небольшой тираж в 2000 экземпляров, обусловленный ограничениями приобретенной издательством лицензии.

Издательство «Техносфера» при поддержке ОАО «ИНЭУМ им. И.С. Брука» готовит перевод еще одного учебника:

Хеннесси Д.Л., Паттерсон Д. Компьютерная архитектура. Количественный подход / перевод с англ. под ред. к.т.н. А.К. Кима.

Это хорошо известный и популярный учебник для старших курсов и магистерских программ, но он начинается именно там, где заканчивается «Архитектура компьютера и проектирование компьютерных систем» Паттерсона и Хеннесси. Понимание и освоение

материала из этого учебника практически невозможно без освоения материала из предшествующих книг.

Отечественная учебная литература по архитектурам ЭВМ и микропроцессоров представлена книгами:

- *Жмакин А.П. Архитектура ЭВМ. 2-е изд. СПб.: БХВ-Петербург, 2010. – ISBN 978-5-9775-0550-5;*
- *Орлов С.А., Цилькер Б.Я. Организация ЭВМ и систем. 2-е изд. СПб.: Питер, 2011. – ISBN: 978-5-49807-862-5.*

Эти учебники в еще большей степени носят описательный характер способов построения различных микропроцессоров на основе зарубежных публикаций и могут рассматриваться как справочники по данному предмету, хотя содержат далеко не всю информацию об архитектурах. Они также непригодны для практической разработки и построения собственного микропроцессора и нуждаются в специальном лабораторном практикуме. Кроме этих книг, имеются многочисленные методические пособия, изданные в разных университетах, курсы и конспекты лекций по предмету, доступные через Интернет. К сожалению, ни один из них не может использоваться в качестве массового учебника для студентов, как младших, так и старших курсов.

Прогресс полупроводниковых технологий и инструментальных средств проектирования цифровых систем в 1990–2000-е годы вывел на первый план языки описания аппаратуры System Verilog и VHDL, которые практически вытеснили традиционное схемотехническое проектирование электронных устройств, включая блоки микропроцессоров. Создание сложных систем на кристалле, объединяющих несколько различных типов микропроцессоров, стало возможным только при использовании средств проектирования, моделирования и верификации ультра-больших интегральных схем, поставляемых фирмами Cadence, Synopsys и Mentor Graphics.

Кроме того, появились доступные по цене средства моделирования и макетирования в виде конструкторских плат, использующих FPGA (Field Programmable Gate Array), называемых в России ПЛИС (Программируемая логическая интегральная схема). Такие платы могут быть приобретены любым учебным центром или даже частным лицом. Используя языки описания аппаратуры и предоставленные производителем FPGA инструментальные средства от вышеупомянутых фирм, любой грамотный студент в состоянии самостоятельно спроектировать и построить сложную цифровую систему, включая микропроцессор.

К сожалению, массовых учебных пособий, позволяющих изучать архитектуру микропроцессоров с практическим уклоном, в том числе и

студентам младших курсов, до последнего времени просто не было. В том числе и в США, где основным учебником оставался вышеупомянутый учебник Паттерсона и Хеннесси «Архитектура компьютера и проектирование компьютерных систем», который впервые был издан в начале 1990-х годов, когда современные инструментальные средства еще не были массово доступны.

Поэтому изданный в 2012 году во второй раз учебник Дэвида Харриса и Сары Харрис «Digital Design and Computer Architecture» стал крайне популярным в США, включая Калифорнийский Университет в Беркли, в котором работает профессором Д. Паттерсон. С одной стороны, он базируется на классическом материале из учебника Паттерсона и Хеннесси, с другой стороны, добавляет все уровни проектирования цифровых блоков на языках описания аппаратуры System Verilog и VHDL, позволяющей практическую реализацию методов построения блоков микропроцессоров на платах FPGA. Кроме того, он содержит интенсивный практикум программирования на языке ассемблера популярного микропроцессора MIPS. Все детали подготовки этой книги и мотивацию вы можете прочитать в предисловии авторов к американскому изданию.

Традиционный подход к русскоязычному изданию такого учебника привел бы к аналогичным результатам, полученным издательством «Питер» – малому тиражу и ограниченной доступности для студентов

младших курсов и техникумов, где эта книга могла бы принести наибольшую пользу. Поэтому, для того, чтобы сделать русскоязычное издание этого учебника массово доступным, его нужно было сделать в электронной версии, загружаемой через Интернет без оплаты или за минимальную плату. Кроме того, нужно было найти спонсора, который бы оплатил стоимость лицензии у издательства Elsevier и подготовку перевода этой книги наряду с необходимой верстской и подготовкой текста к электронному изданию. Решение всех этих вопросов выглядело крайне проблематичным.

Дальнейшие события, связанные с этим проектом, можно отнести к категории мистического совпадения многих случайностей, которые предопределили дальнейшую судьбу проекта. Первым и главным фактором стала личная инициатива Юрия Панчула, разработчика микропроцессоров MIPS, инициатора перевода на русский язык второго издания учебника «Digital Design and Computer Architecture», который он активно пропагандировал в университетах США, России и Украины.

До включения компании MIPS в структуру британской корпорации Imagination Technologies перспективы издания учебника на русском языке были достаточно призрачны ввиду отсутствия значимых бизнес-проектов MIPS в России. Поэтому вторым фактором был приход британских управленцев в подразделение MIPS в составе Imagination Technologies, которые начали интенсивную экспансию на российский

рынок при активной помощи Юрия Панчула и российской компании «Наутех». Сложилась ситуация, когда появилась реальная необходимость в массовом учебнике, в котором в деталях рассматривается архитектура микропроцессоров MIPS и его практическое применение для построения современных систем на кристалле для растущего российского рынка.

Третьим фактором стал приход в Imagination Technologies нового менеджера по мировым образовательным программам Роберта Оуэна (Robert Owen), который предложил спонсировать электронное издание русского перевода учебника и лицензировать право на перевод у издательства Elsevier. Главным условием являлось бесплатное распространение русскоязычной версии учебника с образовательного портала Imagination Technologies.

Четвертым фактором, сделавшим возможным успех проекта, стал энтузиазм и пассионарность русскоязычных инженеров мировой электронной индустрии как в США, так и в России, взявшихся за перевод глав и разделов учебника методом краудсорсинга при активной координации Юрия Панчула. Перевод и редактирование многих глав и разделов, а также полная верификация перевода и синхронизация технических терминов были бы невозможны без активного участия профессоров, доцентов, стажеров и аспирантов из университетов России и Украины, также присоединившихся к проекту.

Пятым и последним фактором можно назвать активную помощь издательских структур и сотрудников Imagination Technologies и корпорации Роснано, которая также активно включилась в проект по изданию учебника в рамках поддержки развития электронной инженерии и разработки наноэлектроники в России.

Надеемся, что этот проект получит дальнейшее продолжение в виде издания бумажной версии учебника и создания портала поддержки развития электронной инженерии в странах русскоязычного мира. Такой портал необходим для консолидации учебных и научных ресурсов, которые помогут университетам и колледжам всегда «быть на острие прогресса» электронных технологий и инструментальных средств.

БЛАГОДАРНОСТИ УЧАСТНИКАМ ПРОЕКТА

Перевод второго издания учебника Дэвида Харриса и Сары Харрис «Digital Design and Computer Architecture» на русский язык был осуществлен в рекордные сроки – всего за четыре месяца. Перевод и редактирование выполнила команда из полусотни энтузиастов, заинтересованных в том, чтобы в русскоязычных странах (России, Украине, Беларуси, Казахстане и других) возникла твердая основа преподавания современной цифровой электроники на основе системного подхода, с одновременным введением в разработку аппаратуры и низкоуровневого программного обеспечения.

В группу переводчиков вошли преподаватели российских и украинских университетов, сотрудники институтов Российской академии наук, инженеры ведущих российских, американских и западноевропейских компаний. Это позволило воспроизвести устойчивую терминологию, пригодную не только для этого проекта, но и для будущих книг по цифровой схемотехнике, языкам описания аппаратуры, компьютерной архитектуре и микроархитектуре, разработке систем на кристалле, использованию программируемых логических интегральных схем и микроконтроллеров.

Инициаторами проекта выступили:

- Юрий Панчул, старший инженер по разработке аппаратуры компании Imagination Technologies, группа разработки микропроцессора MIPS I6400;
- Тимур Палташев, старший менеджер группы компьютерной графики компании Advanced Micro Devices;
- Роберт Оуэн, консультант по образовательным программам, менеджер мировых образовательных программ Imagination Technologies.

Хотелось бы особо отметить следующих участников, которые отличились объемом, скоростью и качеством перевода:

- Валерий Казанцев, старший инженер по применению процессорных ядер ARC компании Synopsys, Санкт-Петербург, Россия;
- Александр Барабанов, доцент кафедры компьютерной инженерии факультета радиофизики, электроники и компьютерных систем Киевского национального университета имени Тараса Шевченко;
- Группа переводчиков в Самарском государственном аэрокосмическом университете имени академика С.П. Королёва

(СГАУ), руководитель группы – декан радиотехнического факультета Илья Александрович Кудрявцев.

Также хотелось бы поблагодарить коллектив компании АНО «eNano», созданной Фондом инфраструктурных образовательных программ ОАО «РОСНАНО», за высококачественную работу по форматированию и верстке книги. «eNano» организовали оперативный и слаженный процесс работы над изданием, который существенно облегчил выход книги в свет. Окончательную версию книги помог создать отдел Creative Services компании Imagination Technologies.

31 декабря 2014 года

Тимур Палташев, Advanced Micro Devices, Sunnyvale, California, USA
Юрий Панчул, Imagination Technologies, Santa Clara, California, USA
Роберт Оуэн, Imagination Technologies, Hertfordshire, United Kingdom

Предисловие

Зачем публиковать еще одно учебное пособие по цифровой схемотехнике и архитектуре компьютера? Несколько дюжин книг по цифровым системам уже опубликованы и активно используются. Также существует несколько хороших изданий по компьютерной архитектуре, в том числе и классические учебники Паттерсона и Хеннесси (Patterson & Hennessy). Предлагаемая книга уникальна тем, что в ней подробно и доступно описан процесс проектирования цифровых систем с точки зрения компьютерной архитектуры, начиная с двоичных цифр «0» и «1» и заканчивая проектированием микропроцессора MIPS.

В течение многих лет в колледже Harvey Mudd мы использовали различные издания книги Паттерсона и Хеннесси «*Computer Organization and Design» (прим. переводчика: четвертое издание этой книги переведено на русский язык в 2012 году издательством «Питер» под названием «Архитектура компьютера и проектирование компьютерных систем*»). В их книге нам особенно нравится описание архитектуры и микроархитектуры MIPS ввиду того, что MIPS является коммерчески успешной микропроцессорной

архитектурой, но при этом остается достаточно простой и позволяет студентам младших курсов самостоятельно построить микропроцессор. Так как этот курс не имеет специальных требований по предварительным знаниям, первая половина семестра посвящена основам проектирования цифровых систем, которые не рассматриваются в книге Паттерсона и Хеннеси. Другие университеты тоже испытывали потребность в учебном пособии, которое сочетало бы цифровую схемотехнику и компьютерную архитектуру. Поэтому мы и взяли на себя обязательство подготовить такую книгу.

Мы считаем, что проектирование микропроцессора является своеобразным обрядом посвящения для студентов инженерных и компьютерных специальностей. Внутренняя работа микропроцессора кажется почти магической для непосвященных, но при подробном объяснении оказывается простой и доступной для понимания. Проектирование цифровых схем само по себе является захватывающим предметом. Программирование на языке ассемблера позволяет понять внутренний язык, на котором говорит микропроцессор. Микроархитектура, в свою очередь, является тем связующим звеном, которое объединяет эти предметы воедино.

Данная книга подходит как для сжатого односеместрового курса «Введение в цифровую схемотехнику и архитектуру компьютера», так и для более углубленного двухсеместрового курса, позволяющего

студентам посвятить больше времени освоению материала и проведению лабораторных работ. Курс может преподаваться «с нуля» и не требует предварительной подготовки. Материал, содержащийся в книге, обычно преподается на втором или третьем курсе университетов, но и заинтересованные первокурсники тоже смогут освоить его.

ОСОБЕННОСТИ КНИГИ

Эта книга содержит ряд особенностей.

Одновременное использование языков SystemVerilog и VHDL

Языки описания аппаратуры (hardware description languages, HDL) находятся в центре современных методов проектирования сложных цифровых систем. К сожалению, разработчики делятся на две примерно равные группы, использующие два разных языка – SystemVerilog и VHDL. Языки описания аппаратуры рассматриваются в **главе 4**, сразу после глав, посвященных проектированию комбинационных и последовательностных логических схем. Затем языки HDL используются в **главе 5** и **главе 7** для разработки цифровых блоков большего размера и процессора целиком. Тем не менее, **Главу 4** можно безболезненно пропустить, если изучение языков HDL не входит в программу.

Эта книга уникальна тем, что использует одновременно и SystemVerilog, и VHDL, что позволяет читателю освоить проектирование цифровых

систем сразу на двух языках. В **главе 4** сначала описываются общие принципы, применимые к обоим языкам, а затем вводится синтаксис и приводятся примеры использования этих языков. Этот двуязычный подход облегчает преподавателю выбор языка HDL, а читателю позволит перейти с одного языка на другой как во время учебы, так и в профессиональной деятельности.

Архитектура и микроархитектура классического процессора MIPS

Главы 6 и 7 посвящены изучению архитектуры MIPS и написаны на основе учебника Паттерсона и Хеннесси «Computer Organization and Design». Архитектура MIPS является идеальной в том смысле, что это реальная архитектура, на которой основаны миллионы выпускаемых ежегодно микросхем, и в то же время она проста для изучения. Кроме того, сотни университетов по всему миру разрабатывают учебные курсы, лабораторные работы и различные инструменты именно для этой архитектуры.

Живые примеры

В **Главах 6, 7 и 8** в качестве примера также рассматривается архитектура, микроархитектура и иерархия памяти процессоров Intel x86. В качестве другого примера в **главе 8** описываются периферийные устройства микроконтроллеров PIC32 компании Microchip. Эти живые примеры показывают, как описанные в данных главах концепции применяются в реальных микросхемах, которые

широко используются в персональных компьютерах и бытовой электронике.

Доступное описание высокопроизводительных архитектур

Глава 7 содержит краткий обзор современных высокопроизводительных микроархитектур: суперскалярной, с внеочередным выполнением команд, многопоточной и многоядерной. Материал изложен в доступной для первокурсников форме и показывает, как можно расширить микроархитектуры, описанные в книге, чтобы получить современный процессор.

Упражнения в конце глав и вопросы для собеседования

Лучшим способом изучения цифровой схемотехники является разработка устройств. В конце каждой главы приведены многочисленные упражнения. За упражнениями следует набор вопросов для собеседования, которые наши коллеги обычно задают студентам, претендующим на работу в отрасли. Эти вопросы предлагают читателю взглянуть на задачи, с которыми соискателям придется столкнуться в ходе собеседования при трудоустройстве. Решения упражнений доступны через вебсайт книги и специальный вебсайт для преподавателей. Более подробная информация приведена в следующем разделе.

МАТЕРИАЛЫ В ИНТЕРНЕТЕ

Дополнительные материалы для этой книги доступны на вебсайте по адресу *textbooks.elsevier.com/9780123944245*. Этот вебсайт доступен всем читателям и содержит:

- Решения нечетных упражнений
- Ссылки на профессиональные средства автоматизированного проектирования (САПР) компаний Altera® и Synopsys®
- Ссылка на QtSpim (также известен как SPIM), симулятор MIPS
- HDL-код для процессора MIPS
- Полезные советы по использованию САПР Altera Quartus II
- Полезные советы по использованию среды разработки Microchip MPLAB IDE
- Слайды лекций в формате PowerPoint
- Образцы учебных и лабораторных материалов для курса
- Список опечаток

Также существует специальный вебсайт для преподавателей, зарегистрировавшихся на *textbooks.elsevier.com*, который содержит:

- Решения всех упражнений
- Ссылки на САПР компаний Altera® и Synopsys® (Synopsys предлагает инструментарий Synplify® Premier университетам, удовлетворяющим определенным требованиям, в упаковке по 50 лицензий. Для получения подробной информации об университетских программах Synopsys зайдите специальный вебсайт для преподавателей)
- Рисунки из текста в форматах JPG и PPT

Дополнительная информация по использованию инструментов Altera, Synopsys, Microchip и QtSpim в вашем курсе приведена в следующем разделе, там же находится информация о материалах для лабораторных работ.

КАК ИСПОЛЬЗОВАТЬ ПРОГРАММНЫЙ ИНСТРУМЕНТАРИЙ В УЧЕБНОМ КУРСЕ

Quartus II Web

Quartus II Web Edition является бесплатной версией профессиональной САПР Quartus ™ II, предназначенной для разработки на FPGA. Это позволяет студентам проектировать цифровые устройства в виде принципиальных схем или на языках SystemVerilog и VHDL. После создания схемы или кода устройства студенты могут симулировать их поведение с использованием САПР ModelSim ™ - Altera Starter Edition, которая доступна вместе с Altera Quartus II Web Edition. Quartus II Web Edition также включает в себя встроенный логический синтезатор, поддерживающий как SystemVerilog, так и VHDL.

Разница между Web Edition и Subscription Edition заключается в том, что Web Edition поддерживает только подмножество наиболее распространенных FPGA производства Altera. Разница между ModelSim - Altera Starter Edition и коммерческих версий ModelSim заключается в том, что Starter Edition искусственно снижает производительность симуляции для проектов, содержащих больше 10 тысяч строк HDL-кода.

Microchip MPLAB IDE

Интегрированная среда разработки Microchip MPLAB является инструментом для программирования микроконтроллеров PIC и доступна для свободного скачивания в сети. MPLAB объединяет написание программы, компиляцию, моделирование и отладку в едином интерфейсе. Она включает в себя компилятор языка C и отладчик, позволяющий студентам разрабатывать C-код и ассемблерные программы, компилировать их, а также загружать и запускать их на микроконтроллере PIC.

Дополнительные инструменты: Synplify Premier и QtSpim

Synplify Premier и QtSpim являются дополнительными инструментами, которые могут быть использованы в этом курсе.

САПР Synplify Premier является средой для синтеза и отладки цифровых систем на FPGA и CPLD. В комплекте есть HDL-Analyst, уникальный графический инструмент анализа HDL-кода, который автоматически генерирует принципиальную схему из исходного HDL-кода и позволяет в реальном времени видеть соответствие определенных конструкций языка и частей принципиальной схемы. Это очень полезно в процессе обучения и отладки.

Synopsys предлагает инструментарий Synplify® Premier университетам, удовлетворяющим определенным требованиям, в упаковке по 50 лицензий. Для получения подробной информации об университетских программах Synopsys зайдите на специальный вебсайт для преподавателей (*textbooks.elsevier.com/9780123944245*).

QtSpim, также известный как SPIM – это симулятор, который исполняет код на языке ассемблера MIPS. Студенты должны сохранить свой ассемблерный код как текстовый файл и запустить его с помощью QtSpim. В QtSpim отображаются команды, содержимое памяти и значения регистров. Ссылки на руководство пользователя и примеры файлов доступны на вебсайте книги (*textbooks.elsevier.com/9780123944245*).

Лабораторные работы

Вебсайт книги содержит ссылки на ряд лабораторных работ, которые охватывают все темы, начиная от проектирования цифровых систем и заканчивая архитектурой компьютера. Из лабораторных работ студенты узнают, как использовать САПР Quartus II для описания своих проектов, их симулирования, синтеза и реализации. Лабораторные работы также включают темы по программированию на языке С и языке ассемблера с использованием среды проектирования MPLAB IDE.

После синтеза студенты могут реализовать свои проекты, используя платы Altera DE2. Эта мощная и относительно недорогая плата доступна для заказа на вебсайте *www.altera.com*. Плата содержит микросхему FPGA, которую можно сконфигурировать для реализации студенческих проектов. Мы предоставляем лабораторные работы, которые описывают, как реализовать различные блоки на плате DE2 с использованием Quartus II Web Edition.

Для выполнения лабораторных работ студенты должны будут загрузить и установить САПР Altera Quartus II Web Edition и Microchip MPLAB IDE. Преподаватели могут также установить эти САПР в учебных. Лабораторные работы включают инструкции по разработке проектов на плате DE2. Этап практической реализации проекта на плате можно пропустить, однако мы считаем, что он имеет большое значение для получения практических навыков.

Мы протестировали лабораторные работы на ОС Windows, но инструменты доступны и для ОС Linux.

ОПЕЧАТКИ

Все опытные программисты знают, что любая программа сложная программа непременно содержит ошибки. Так же происходит и с

книгами. Мы старались выявить и исправить все ошибки и опечатки в этой книге. Тем не менее, некоторые ошибки могли остаться. Список найденных ошибок будет опубликован на вебсайте книги.

Пожалуйста, присылайте найденные ошибки по адресу iup@imgtec.com. Первый человек, который сообщит об ошибке и предоставит исправление, которое мы используем в будущем издании, будет вознагражден премией в $1!

Признательность за поддержку

Прежде всего, мы благодарим Дэвида Паттерсона и Джона Хеннесси за разработку новаторских микроархитектур MIPS, описанных в их учебнике «Computer Organization and Design». В течение многих лет мы использовали различные издания этой книги в преподавании. Мы высоко ценим поддержку, оказанную Дэвидом и Джоном при написании этой книги, а также их великодушное разрешение на использование этих микроархитектур в нашем учебнике.

Отдельное спасибо Дуэйну Бибби (Dwyane Bibby), нашему любимому художнику-оформителю, который долго и упорно трудился, чтобы проиллюстрировать разработку цифровых систем. Мы также высоко ценим энтузиазм Нэйта МакФаддена (Nate McFadden), Тодда Грина (Todd Green), Дэниэла Миллера (Danielle Miller), Робина Дэя (Robyn

Day) и остальных членов команды издательства Morgan Kaufmann, которые сделали возможным появление этой книги.

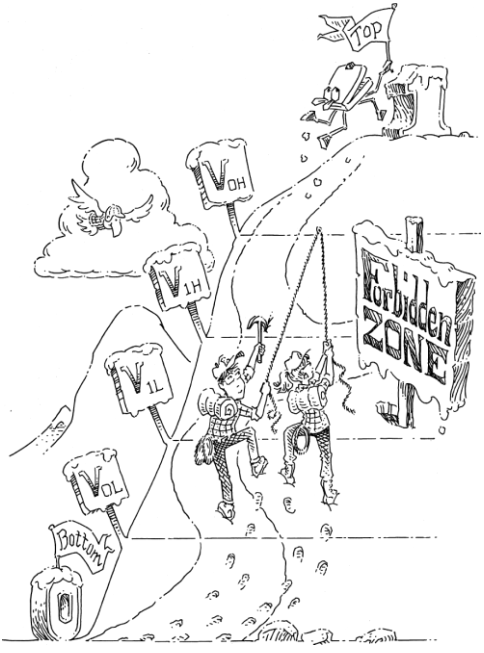
Мы хотели бы поблагодарить Мэтью Уоткинса (Matthew Watkins), который помог написать раздел о гетерогенных многопроцессорных системах в Главе 7. Мы также благодарны Крису Парксу (Chris Parks), Карлу Пирсону (Carl Pearson) и Джонатану Чай (Johnathan Chai), которые проверили коды и разработали оглавление для второго издания.

Огромный вклад в улучшение качества книги внесли многочисленные рецензенты: Джон Барр (John Barr), Джэк Брайнер (Jack V. Briner), Эндрю Браун (Andrew C. Brown), Карл Баумгартнер (Carl Baumgaertner), Утку Дирил (A. Utku Diril), Джим Френцель (Jim Frenzel), Джаэха Ким (Jaeha Kim), Филлип Кинг (Phillip King), Джеймс Пинтер-Лаки (James Pinter-Lucke), Амир Рот (Amir Roth), Джерри Ши (Z. Jerry Shi), Джеймс Стайн (James E. Stine), Люк Тэсье (Luke Teyssier), Пейуй Чжао (Peiyi Zhao), Зак Доддс (Zach Dodds), Натаниэл Гай (Nathaniel Guy), Эшвин Кришна (Aswin Krishna), Волней Педрони (Volnei Pedroni), Карл Ванг (Karl Wang), Рикардо и анонимный рецензент.

Мы также очень признательны нашим студентам в колледже Harvey Mudd, которые дали полезные отзывы на черновики этого учебника. Отдельного упоминания заслуживают Мэтт Вайнер (Matt Weiner), Карл

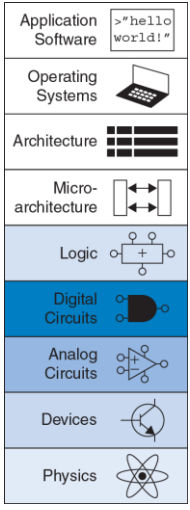
Уолш (Carl Walsh), Эндрю Картер (Andrew Carter), Кейси Шиллинг (Casey Schilling), Элис Клифтон(Alice Clifton), Крис Эйкон (Chris Acon) и Стивен Браунер (Stephen Brawner).

И, конечно же, мы благодарим наши семьи за их любовь и поддержку.


Top
V_{OH}
V_{IH}
V_{IL}
V_{OL}
Bottom
0
Forbidden
ZONE

1

От нуля до единицы



1.1 ПЛАН ИГРЫ

За последние тридцать лет микропроцессоры буквально изменили наш мир до неузнаваемости. Сегодняшний ноутбук обладает большей вычислительной мощностью, чем большой компьютер недавнего прошлого, занимавший целую комнату. Внутри современного автомобиля представительского класса можно обнаружить около пятидесяти микропроцессоров. Именно прогресс в области микропроцессорной техники сделал возможным появление сотовых телефонов и Интернета, значительно продвинул вперед медицину и радикально изменил тактику и стратегию современной войны. Объем продаж мировой полупроводниковой промышленности вырос с 21 миллиарда долларов в 1985 году до 300 миллиардов долларов в 2011 году, причем микропроцессоры составили львиную долю этих продаж. И мы убеждены, что микропроцессоры важны не только с технической, экономической и социальной точек зрения, но и стали одним из самых увлекательных изобретений в истории человечества. Когда вы закончите чтение этой книги, вы будете знать, как спроектировать и построить ваш собственный микропроцессор, а навыки, полученные на этом пути, пригодятся вам для разработки и многих других цифровых систем.

Мы предполагаем, что у вас уже есть базовые знания по теории электричества, некоторый опыт программирования и искреннее

желание понять, что происходит под капотом компьютера. В этой книге основное внимание уделяется разработке цифровых систем, то есть систем, которые используют для своей работы два уровня напряжения, представляющих единицу и нуль. Мы начнем с простейших цифровых логических элементов – вентилей (digital logic gates), которые принимают определенную комбинацию единиц и нулей на входе и трансформируют ее в другую комбинацию единиц и нулей на выходе. После этого мы с вами научимся объединять эти простейшие логические элементы в более сложные модули, такие как сумматоры и блоки памяти. Затем мы перейдем к программированию на языке ассемблера – родном языке микропроцессора. И в завершение, из кирпичиков логических элементов мы с вами соберем полноценный микропроцессор, способный выполнять ваши программы, написанные на языке ассемблера.

Огромным преимуществом цифровых систем над аналоговыми является то, что необходимые для их построения блоки чрезвычайно просты, поскольку оперируют не непрерывными сигналами, а единицами и нулями.

Построение цифровой системы не требует запутанных математических расчетов или глубоких знаний в области физики. Вместо этого, задача, стоящая перед разработчиком цифровых устройств, заключается в том, чтобы собрать сложную работающую систему из этих простых блоков.

Возможно, микропроцессор станет первой спроектированной вами системой, настолько сложной, что ее невозможно целиком удержать в голове. Именно поэтому одной из тем, проходящих красной нитью через эту книгу, является искусство управления сложностью системы.

1.2 ИСКУССТВО УПРАВЛЕНИЯ СЛОЖНОСТЬЮ

Одной из характеристик, отличающих профессионального инженера-электронщика или программиста от дилетанта, является систематический подход к управлению сложностью многоуровневой системы. Современные цифровые системы построены из миллионов и миллиардов транзисторов. Человеческий мозг не в состоянии предсказать поведение подобных систем путем составления уравнений, описывающих движение каждого электрона в каждом транзисторе системы, и последующего решения этой системы уравнений. Для того, чтобы разработать удачный микропроцессор и не утонуть при этом в море избыточной информации, необходимо научиться управлять сложностью разрабатываемой системы.

1.2.1 Абстракция

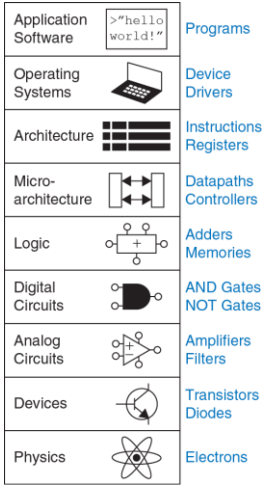


Рис. 1.1 Уровни абстракции электронной вычислительной системы

Критически важный принцип управления сложностью системы – *абстракция*, подразумевающая исключение из рассмотрения тех элементов, которые в данном конкретном случае несущественны для понимания работы этой системы. Любую систему можно рассматривать с различных уровней абстракции. Политику, участвующему в выборах, например, нет нужды учитывать все детали окружающего его мира, ему достаточно абстрактной иерархической модели страны, состоящей из населенных пунктов, областей и федеральных округов. В области может быть несколько населенных пунктов, а федеральный округ включает в себя разные области. Если политик борется за пост президента, то его, скорее всего, интересует то, как проголосует федеральный округ в целом, при этом ему не обязательно знать, какое количество

голосов он наберет в каждом конкретном населенном пункте этого округа. Для политика федеральный округ – это его уровень абстракции. С другой стороны, бюро переписи населения обязано знать количество жителей в каждом городе или поселке страны и потому должно оперировать на самом низком уровне абстракции данной системы – на уровне населенных пунктов.

На **Рис. 1.1** показаны уровни абстракции, типичные для любой электронной компьютерной системы вместе со строительными блоками, характерными для каждого уровня абстракции этой системы. На самом низком уровне абстракции находится физика, изучающая движение электронов. Поведение электронов описывается квантовой механикой и системой уравнений Максвелла.

Рассматриваемая нами современная электронная система состоит из полупроводниковых устройств *(devices)*, таких как транзисторы (а когда-то это были электронные лампы). Каждое такое устройство имеет четко определенные точки соединения с другими подобными устройствами. Эти точки мы будем называть *контактами* (в англоязычной литературе используется термин *terminal*). Любое электронное устройство может быть представлено абстрактной математической моделью, описывающей изменяющуюся во времени взаимозависимость тока и напряжения. Такие же изменения тока и напряжения можно наблюдать на экране осциллографа, если подключить осциллограф к контактам

реального устройства. Данный подход означает, что, если рассматривать систему на уровне устройств, функции которых однозначно определены, то можно не учитывать поведение электронов внутри отдельных устройств этой системы.

Следующий уровень абстракции – это *аналоговые схемы (analog circuits)*, в которых полупроводниковые устройства соединены таким образом, чтобы они образовывали функциональные компоненты, такие как усилители, например. Напряжение на входе и на выходе аналоговой цепи изменяется в непрерывном диапазоне.

В отличие от аналоговых цепей, *цифровые схемы (digital circuits)*, такие как логические вентили, используют два строго ограниченных дискретных уровня напряжения. Один из этих дискретных уровней – это логический нуль, другой – логическая единица. В разделах этой книги, посвященных разработке цифровых схем и устройств, мы будем использовать простейшие цифровые схемы для построения сложных цифровых модулей, таких как сумматоры и блоки памяти.

Микроархитектурный уровень абстракции, или просто *микроархитектура (microarchitecture)*, связывает логический и архитектурный уровни абстракции. Архитектурный уровень абстракции, или *архитектура (architecture)*, описывает компьютер с точки зрения программиста. Например, архитектура Intel x86, используемая

микропроцессорами большинства персональных компьютеров (*ПК*), определяется набором инструкций и регистров (памяти для временного хранения переменных), доступным для использования программистом. Микроархитектура – это соединение простейших цифровых элементов в логические блоки, предназначенные для выполнения команд, определенных какой-то конкретной архитектурой. Отдельно взятая архитектура может быть реализована с использованием различных вариантов микроархитектур с разным соотношением цены, производительности и потребляемой энергии, и такое соотношение зачастую выбирается как баланс между этими тремя факторами. Процессоры Intel Core i7, Intel 80486 и AMD Athlon, например, используют одну и ту же архитектуру x86, но реализованную с использованием трех разных микроархитектурных решений.

Теперь мы перемещаемся в область программного обеспечения. *Операционная система* (*operating system*) управляет операциями нижнего уровня, такими как доступ к жесткому диску или управление памятью. И, наконец, программное обеспечение использует ресурсы операционной системы для решения конкретных задач пользователя.

Именно принцип *абстрагирования от маловажных деталей* позволяет вашей бабушке общаться с внуками в Интернете, не задумываясь о квантовых колебаниях электронов или организации памяти компьютера.

Предмет этой книги – уровни абстракции от цифровых схем до компьютерной архитектуры. Работая на каком-либо из этих уровней абстракции, полезно знать кое-что и об уровнях абстракции, непосредственно сопряженных с тем уровнем, где вы находитесь. Программист, например, не сможет полностью оптимизировать код без понимания архитектуры процессора, который будет выполнять эту программу. Инженер-электронщик, разрабатывающий какой-либо блок микросхемы, не сможет найти компромисс между быстродействием и уровнем потребления энергии транзисторами, ничего не зная о той цифровой схеме, где этот блок будет использоваться. Мы надеемся, что к тому времени, когда вы закончите чтение этой книги, вы сможете выбрать уровень абстракции, необходимый для успешного выполнения любой стоящей перед вами задачи, и оценить влияние ваших инженерных решений на другие уровни абстракции в разрабатываемой вами системе.

Каждая глава этой книги начинается с иконок (см. **Рис. 1.1**), символически изображающих уровни абстракции электронной системы, которые мы перечислили выше. Иконка темно-синего цвета указывает на тот уровень абстракции, которому уделяется главное внимание в этой конкретной главе. Иконки более светлого оттенка синего указывают на другие уровни абстракции, также затронутые в этой главе.

1.2.2 Конструкторская дисциплина

Конструкторская Дисциплина – это преднамеренное ограничение самим конструктором выбора возможных вариантов разработки, что позволяет работать продуктивнее на более высоком уровне абстракции. Использование взаимозаменяемых частей – это, вероятно, самый хорошо знакомый всем нам пример практического применения конструкторской дисциплины. Одним из первых примеров использования взаимозаменяемых деталей и узлов стала унификация при производстве кремнёвых ружей. До начала 19-го века такие ружья производились вручную и в штучном порядке. Высококвалифицированный оружейный мастер тщательно подтачивал и подгонял комплектующие, произведенные несколькими не связанными друг с другом ремесленниками. Конструкторская дисциплина для обеспечения взаимозаменяемости деталей и узлов произвела революцию в оружейной промышленности. Ограничение ассортимента комплектующих деталей до стандартного набора с жестко установленными допусками для каждой детали позволило собирать и ремонтировать ружья гораздо быстрее и использовать при этом менее квалифицированный персонал. Оружейный мастер перестал тратить свое время на разрешение проблем, связанных с нижними уровнями абстракции, такими как доводка какого-то

конкретного ствола или исправление формы отдельного взятого приклада.

В контексте данной книги соблюдение конструкторской дисциплины в виде максимального использования цифровых схем играет очень важную роль. В цифровых схемах используются дискретные значения напряжения, в то время как в аналоговых схемах напряжение изменяется непрерывно. Таким образом, цифровые схемы, которые можно рассматривать как подмножество аналоговых цепей, в некотором смысле уступают по своим характеристикам более широкому классу аналоговых цепей. Однако цифровые цепи гораздо проще проектировать. Ограничивая использование аналоговых схем и по возможности заменяя их цифровыми, мы можем легко объединять отдельные компоненты в сложные системы, которые, в конечном итоге, для большинства приложений превзойдут по своим параметрам системы, построенные на аналоговых цепях. Примером тому могут служить цифровые телевизоры, компакт-диски (CD) и мобильные телефоны, которые уже практически полностью вытеснили своих аналоговых предшественников.

1.2.3 Три базовых принципа

В дополнение к абстрагированию от несущественных деталей и конструкторской дисциплине разработчики электронных систем используют еще три базовых принципа для управления сложностью системы: иерархичность, модульность конструкции и регулярность. Эти принципы применительно как к программному обеспечению, так и к аппаратной части компьютерных систем.

- *Иерархичность* – принцип иерархичности предполагает разделение системы на отдельные модули, а затем последующее разделение каждого такого модуля на фрагменты до уровня, позволяющего легко понять поведение каждого конкретного фрагмента.
- *Модульность* – принцип модульности требует, чтобы каждый модуль в системе имел четко определенную функциональность и набор интерфейсов и мог быть легко и без непредвиденных побочных эффектов соединен с другими модулями системы.
- *Регулярность* – принцип регулярности требует соблюдения единообразия при проектировании отдельных модулей системы. Стандартные модули общего назначения, например, такие как блоки питания, могут использоваться многократно, во много раз снижая количество модулей, необходимых для разработки новой системы.

Капитан Мериуэзер Льюис – один из руководителей знаменитой экспедиции Льюиса и Кларка на северо-запад США, был, пожалуй, одним из самых ранних сторонников взаимозаменяемости. В 1806 году в своем дневнике, касаясь унификации деталей кремнёвых ружей того времени, он написал следующее:

> «Ружья Дрюера и сержанта Прайора одновременно вышли из строя. На ружье Дрюера сломался ударно-спусковой механизм, и мы заменили его на новый. У ружья сержанта Прайора был сломан курковый винт, вместо которого мы поставили запасной курковый винт, заранее изготовленный специально для ударно-спускового механизма этого ружья на мануфактуре Харперс Фейри, где это оружие и было произведено. Если бы не предусмотрительность, заключавшаяся в том, что мы заранее позаботились о запасных частях для ружей, и не мастерство Джона Шилдса, выполнившего всю работу, то большинство ружей нашей экспедиции к этому времени было бы полностью непригодно для какого-либо использования. И я имею полное право записать в своем дневнике, что, к счастью для нас, все наше оружие находится в прекрасном состоянии».

См. История *экспедиции* Льюиса и Кларка в четырех томах под редакцией Элиота Куэса. Первое издание: Харпер, Нью-Йорк, 1893; переиздание: Довер, Нью-Йорк (3 тома), 3:817.

Для иллюстрации трех базовых принципов вновь воспользуемся аналогией из оружейного производства. Нарезное Кремнёвое ружьё

было одним из самых сложных устройств массового применения в начале 19-го века. Используя принцип иерархичности, мы можем разделить его на три главных модуля, как показано на **Рис. 1.2**: ствол, ударно-спусковой механизм и приклад с цевьем.

Ствол – это длинная металлическая труба, через которую при выстреле выбрасывается пуля. Ударно-спусковой механизм производит выстрел. Деревянные приклад и цевье соединяют воедино остальные части ружья и обеспечивают стрелку надежное удержание оружия при выстреле. В свою очередь, ударно-спусковой механизм включает в себя спусковой крючок, курок, кремень, огниво и пороховую полку. Каждый из этих компонентов также может рассматриваться как следующий иерархический уровень и может быть разделен на более мелкие детали.

Принцип модульности требует, чтобы каждый компонент выполнял четко определенную функцию и имел интерфейс. Функция приклада и цевья – служить базой для установки ствола и ударно-спускового механизма. Интерфейс для приклада и цевья – это их длина и расположение крепёжных элементов, таких как винты или шурупы. Ствол ружья, изготовленного с соблюдением принципа модульности конструкции, может быть установлен на приклады и цевья от разных производителей, если все соединяемые части имеют правильную длину и подходящие крепёжные элементы. Функция ствола – разогнать пулю

до необходимой скорости и придать ей вращение, чтобы увеличить точность стрельбы (примечание переводчика: кремнёвые ружья не были нарезными и использовали круглые пули). Принцип модульности требует также, чтобы при соединении модулей не возникало никаких побочных эффектов: конструкция приклада и цевья не должна препятствовать функционированию ствола.

Принцип регулярности учит тому, что взаимозаменяемые детали – это хорошая идея. При соблюдении принципа регулярности поврежденный ствол может быть с легкостью заменен на идентичный. Стволы могут изготовляться на поточной линии с гораздо большей экономической эффективностью, чем в случае штучного производства.

В данной книге мы будем постоянно возвращаться к этим трем базовым принципам: иерархичности, модульности и регулярности.

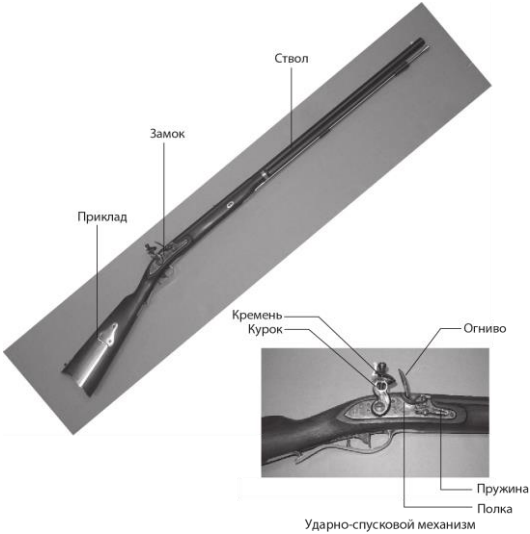
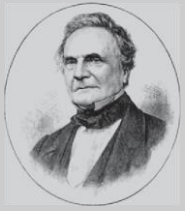


Рис. 1.2 **Кремнёвый ружейный замок**
(Рисунок из Euroarms Italia *www.euroarms.net* © 2006 г.)

1.3 ЦИФРОВАЯ АБСТРАКЦИЯ



Чарльз Бэббидж, 1791–1871

Чарльз Бэббидж родился в 1791 году. Закончил Кембриджский университет и женился на Джорджиане Витмур. Он изобрел Аналитическую Машину – первый в мире механический компьютер. Чарльз Бэббидж также изобрел предохранительную решетку для локомотивов, спидометр и универсальный почтовый тариф. Ученый также очень интересовался отмычками для замков и почему-то ненавидел уличных музыкантов. (Портрет любезно предоставлен Fourmilab Швейцария, *www.fourmilab.ch*).

Большинство физических величин изменяется непрерывно. Например, напряжение в электрическом проводе, частота колебаний или распределение массы – все это параметры, изменяющиеся непрерывно. Цифровые системы, с другой стороны, представляют информацию

в виде дискретно меняющихся переменных с конечным числом строго определённых значений.

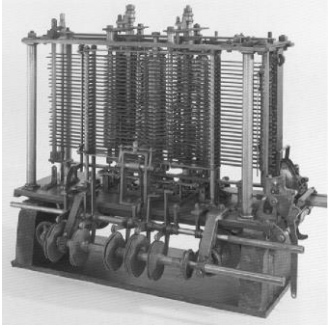


Рис. 1.3 **Аналитическая машина Беббиджа в год его смерти (1871)** (Изображение любезно предоставлено Музеем Науки и общества)

Одной из наиболее ранних цифровых систем стала Аналитическая Машина Чарльза Бэббиджа, которая использовала переменные с десятью дискретными значениями. Начиная с 1834 года и до 1871 года[1] Бэббидж разрабатывал и пытался построить этот механический компьютер. Шестеренки Аналитической Машины могли находится в одном из десяти фиксированных положений, а каждое такое положение было промаркировано от 0 до 9 подобно механическому счетчику пробега автомобиля. **Рис. 1.3** показывает, как выглядел прототип Аналитической Машины. Каждый ряд шестеренок такой машины обрабатывал одну цифру. В своем механическом

[1] А большинству из нас кажется, что обучение в университете – это так долго!

компьютере Бэббидж использовал 25 рядов шестеренок таким образом, чтобы машина обеспечивала вычисления с точностью до 25-го знака.

В отличие от машины Бэббиджа большинство электронных компьютеров использует двоичный (бинарный) код. В случае двоичного кода высокое напряжение – это единица, а низкое напряжение – нуль, поскольку гораздо легче оперировать двумя уровнями напряжения, чем десятью.

Объем информации *D*, передаваемый одной дискретной переменной, которая может находиться в *N* различных состояниях, измеряется в единицах, называемых *битами,* и вычисляется по следующей формуле:

$$D = \log_2 N \text{ bits} \qquad (1.1)$$

Двоичная переменная передает $\log_2 2 = 1$ – один бит информации. Теперь вам, вероятно, понятно, почему единица информации называется битом. *Bit* (*бит*) – это сокращение от английского *binary digit*, что дословно переводится как *двоичный разряд*. Каждая шестеренка в машине Бэббиджа содержит $\log_2 10 = 3{,}322$ бит информации, поскольку она может находиться в одном из $2^{3{,}322} = 10$ уникальных положений. Теоретически непрерывный сигнал может передавать бесконечное количество информации, поскольку может принимать неограниченное

число значений. На практике, однако, шум и ошибки измерения ограничивают информацию, передаваемую большинством непрерывных сигналов, диапазоном от 10 бит до 16 бит. Если же измерение уровня сигнала должно быть произведено очень быстро, то объём передаваемой информации будет еще ниже (в случае 10 бит, например, это будет только 8 бит).

Предмет этой книги – цифровые схемы, использующие двоичные переменные нуль и единицу. Джордж Буль разработал систему логики, использующую двоичные переменные, и эту систему сегодня называют его именем – *Булева логика*. Булевы переменные могут принимать значения *ИСТИНА* (*TRUE*) или *ЛОЖЬ* (*FALSE*). В электронных компьютерах положительное напряжение обычно представляет единицу, а нулевое напряжение представляет нуль. В этой книге мы будем использовать понятия единица (1), ИСТИНА (TRUE) и ВЫСОКОЕ (HIGH) как синонимы. Аналогичным образом мы будем использовать нуль (0), ЛОЖЬ (FALSE), и НИЗКОЕ (LOW) как взаимозаменяемые термины.

Джордж Буль,1815–1864

Джордж Буль родился в семье небогатого ремесленника. Родители Джорджа не могли оплатить его формального образования, поэтому он осваивал математику самоучкой. Несмотря на это, Булю удалось стать преподавателем Королевского колледжа в Ирландии. В 1854 году Джордж Буль написал свою работу Исследование законов мышления, которая впервые ввела в научный оборот двоичные переменные, а также три основных логических оператора И, ИЛИ, НЕ (AND, OR, NOT).

(Портрет любезно предоставлен Американским Физическим Институтом).

Преимущества *цифровой абстракции* заключаются в том, что разработчик цифровой системы может сосредоточиться исключительно на единицах и нулях, полностью игнорируя, каким образом булевы

переменные представлены на физическом уровне. Разработчика не волнует, представлены ли нули и единицы определенными значениями напряжения, вращающимися шестернями или уровнем гидравлической жидкости. Программист может продуктивно работать, не располагая детальной информацией об аппаратном обеспечении компьютера. Однако, понимание того, как работает это аппаратное обеспечение, позволяет программисту гораздо лучше оптимизировать программу для конкретного компьютера.

Как вы могли видеть выше, один-единственный бит не может передать большого количества информации. Поэтому в следующем разделе мы рассмотрим вопрос о том, каким образом набор битов можно использовать для представления десятичных чисел. В последующих главах мы также покажем, как группы битов могут представлять буквы и даже целую программу.

1.4 СИСТЕМЫ СЧИСЛЕНИЯ

Мы все привыкли работать с десятичными числами. Однако, в цифровых системах, построенных на единицах и нулях, использование двоичных или шестнадцатеричных чисел зачастую более удобно. В данном разделе мы рассмотрим системы счисления, использованные в этой книге.

1.4.1 Десятичная система счисления

Еще в начальной школе нас всех научили считать и выполнять различные арифметические операции в *десятичной (decimal)* системе счисления. Такая система использует десять арабских цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 – столько же, сколько у нас пальцев на руках. Числа больше 9 записываются в виде строки цифр. Причем, цифра, находящаяся в каждой последующей позиции такой строки, начиная с крайней правой цифры, имеет «вес», в десять раз превышающий «вес» цифры, находящейся в предыдущей позиции. Именно поэтому десятичную систему счисления называют системой по основанию (*base*) 10. Справаналево «вес» каждой позиции увеличивается следующим образом: 1, 10, 100, 1000 и т.д. Позицию, которую цифра занимает в строке десятичного числа, называют разрядом или декадой.

Чтобы избежать недоразумений при одновременной работе с более чем одной системой счисления, основание системы обычно указывается путем добавления цифры позади и чуть ниже основного числа: 9742_{10}. **Рис. 1.4** показывает, для примера, как десятичное число 9742_{10} может быть записано в виде суммы цифр, составляющих это число, умноженных на «вес» разряда, соответствующего каждой конкретной цифре.

N-разрядное десятичное число может представлять одну из 10^N цифровых комбинаций: 0, 1, 2, 3, … 10^N − 1. Это называется диапазоном *N*-разрядного числа. Десятичное число, состоящее из трех цифр (разрядов), например, представляет одну из 1000 возможных цифровых комбинаций в диапазоне от 0 до 999.

1.4.2 Двоичная система счисления

Одиночный бит может принимать одно из двух значений, 0 или 1. Несколько битов, соединенных в одной строке, образуют *двоичное (binary)* число. Каждая последующая позиция в двоичной строке имеет вдвое больший «вес», чем предыдущая позиция, так что двоичная система счисления – это система по основанию 2. В двоичном числе «вес» каждой позиции увеличивается (так же, как и в десятичном – справа налево) следующим образом: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 и т.д. Работая с двоичными

числами, очень полезно для сохранения времени запомнить значения степеней двойки до 2^{16}.

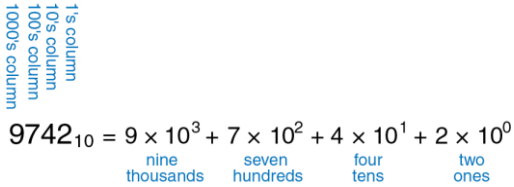


Рис. 1.4 **Представление десятичного числа**

Произвольное *N*-разрядное двоичное число может представлять одну из 2^N цифровых комбинаций: 0, 1, 2, 3, … $2^N - 1$. В **Табл. 1.1** собраны 1-битные, 2-битные, 3-битные, и 4-битные двоичные числа и их десятичные эквиваленты.

Пример 1.1 ПРЕОБРАЗОВАНИЕ ЧИСЕЛ ИЗ ДВОИЧНОЙ СИСТЕМЫ СЧИСЛЕНИЯ В ДЕСЯТИЧНУЮ

Преобразовать двоичное число 10110_2 в десятичное.

Решение: Нужные преобразования представлены на **Рис. 1.5**.

Табл. 1.1 Таблица двоичных чисел и их десятичный эквивалент

1-битные двоичные числа	2-битные двоичные числа	3-битные двоичные числа	4-битные двоичные числа	Десятичные эквиваленты
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

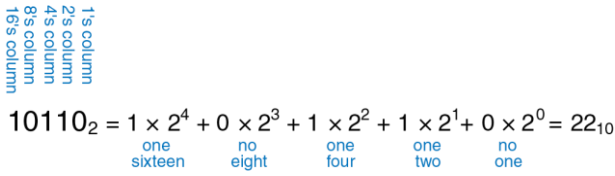


Рис. 1.5 **Преобразование двоичного числа в десятичное число**

Пример 1.2 ПРЕОБРАЗОВАНИЕ ЧИСЕЛ ИЗ ДЕСЯТИЧНОЙ СИСТЕМЫ СЧИСЛЕНИЯ В ДВОИЧНУЮ

Преобразовать десятичное число 84_{10} в двоичное.

Решение: Определите, что должно стоять в каждой позиции двоичного результата: 1 или 0. Вы можете делать это, начиная с левой или правой позиции.

Если начать слева, найдите наибольшую степень 2, меньше или равную заданному числу (в Примере такая степень – это 64). 84 > 64, поэтому ставим 1 в позиции, соответствующей 64. Остается 84–64 = 20, 20 < 32, так что в позиции 32 надо поставить 0, 20 > 16, поэтому в позиции 16 ставим 1. Остается 20–16 = 4. 4 < 8, поэтому 0 в позиции 8. 4 ≥ 4 – ставим 1 в позицию 4. 4–4 = 0, поэтому будут 0 в позициях 2 и 1. Собрав все вместе, получаем $84_{10} = 1010100_2$.

Если начать справа, будем последовательно делить исходное число на 2. Остаток идет в очередную позицию. 84/2 = 42, поэтому 0 в самой правой позиции. 42/2 = 21, 0 во вторую позицию. 21/2 = 10, остаток 1 идет в позицию, соответствующую 4. 10/2 = 5, поэтому 0 в позицию, соответствующую 8. 5/2 = 2,

остаток 1 в позицию 16. 2/2 = 1, 0 в 32 позицию. Наконец, 1/2 = 0 с остатком 1, который идет в позицию 64. Снова, $84_{10} = 1010100_2$

1.4.3 Шестнадцатеричная система счисления

Использование длинных двоичных чисел для записи и выполнения математических расчетов на бумаге утомительно и чревато ошибками. Однако длинное двоичное число можно разбить на группы по четыре бита, каждая из которых представляет одну из 2^4 = 16 цифровых комбинаций. Именно поэтому зачастую бывает удобнее использовать для работы систему счисления по основанию 16, называемую *шестнадцатеричной (hexadecimal)*. Для записи шестнадцатеричных чисел используются цифры от 0 до 9 и буквы от A до F, как показано в **Табл. 1.2** В шестнадцатеричном числе «вес» каждой позиции меняется следующим образом: 1, 16, 16^2 (или 256), 16^3 (или 4096) и т.д.

Интересно, что термин hexadecimal (шестнадцатеричный) введен в научный обиход корпорацией IBM в 1963 году и является комбинацией греческого слова hexi (шесть) и латинского decem (десять). Правильнее было бы использовать латинское же слово sexa (шесть), но термин sexadecimal воспринимался бы несколько неоднозначно.

Табл. 1.2 Шестнадцатеричная система счисления

Шестнадцатеричная цифра	Десятичный эквивалент	Двоичный эквивалент
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Пример 1.3 ПРЕОБРАЗОВАНИЕ ШЕСТНАДЦАТЕРИЧНОГО ЧИСЛА В ДВОИЧНОЕ И ДЕСЯТИЧНОЕ

Преобразовать шестнадцатеричное число $2ED_{16}$ в двоичное и десятичное.

Решение: Преобразование шестнадцатеричного числа в двоичное и обратно – очень простое, так как каждая шестнадцатеричная цифра прямо соответствует 4-разрядному двоичному числу. $2_{16} = 0010_2$, $E_{16} = 1110_2$ и $D_{16} = 1101_2$, так что $2ED_{16} = 001011101101_2$ Преобразование в десятичную систему счисления требует арифметики, показанной на **Рис. 1.6**.

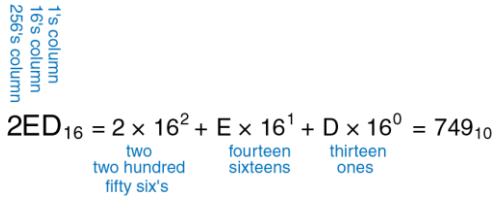


Рис. 1.6 **Преобразование шестнадцатеричного числа в десятичное число**

Пример 1.4 ПРЕОБРАЗОВАНИЕ ДВОИЧНОГО ЧИСЛА В ШЕСТНАДЦАТЕРИЧНОЕ

Преобразовать двоичное число 1111010_2 в шестнадцатеричное.

Решение. Повторим еще раз, это просто. Начинаем справа. 4 наименее значимых бита $1010_2 = A_{16}$. Следующие биты $111_2 = 7_{16}$. Отсюда $1111010_2 = 7A_{16}$.

Пример 1.5 ПРЕОБРАЗОВАНИЕ ДЕСЯТИЧНОГО ЧИСЛА В ШЕСТАНДЦАТЕРИЧНОЕ И ДВОИЧНОЕ

Преобразовать десятичное число 333_{10} в шестнадцатеричное и двоичное.

Решение. Как и в случае преобразования десятичного числа в двоичное, можно начать как слева, так и справа.

Если начать слева, найдите наибольшую степень шестнадцати, меньшую или равную заданному числу (в нашем случае это $16^2 = 256$). Число 256 содержится в числе 333 только один раз, поэтому в позицию с «весом» 256 мы записываем единицу. Остается число 333–256=77. Число 16 содержится в числе 77 четыре раза, поэтому в позицию с «весом» 16 записываем четверку. Остается $77 – 16 \times 4 = 13$. $13_{10} = D_{16}$, поэтому в позицию с «весом» 1 записываем цифру D. Итак, $333_{10} = 14D_{16}$, это число легко преобразовать в двоичное, как мы показали в примере 1.3: $14D_{16} = 101001101_2$.

Если начинать справа, будем повторять деление на 16. Каждый раз остаток идет в очередную колонку. 333/16 = 20 с остатком $13_{10} = D_{16}$, который идет в самую правую позицию. 20/16 = 1 с остатком 4, который идет в позицию с «весом» 16. 1/16 = 0 с остатком 1, который идет в позицию с «весом» 256. В результате опятьполучаем $14D_{16}$.

1.4.4 Байт, полубайт и «весь этот джаз»

Группа из восьми битов называется *байт (byte)*. Байт представляет $2^8 = 256$ цифровых комбинаций. Размер модулей, сохраненных в памяти компьютера, обычно измеряется именно в байтах, а не битах.

Группа из четырех битов (половина байта) называется *полубайт (nibble)*. Полубайт представляет $2^4 = 16$ цифровых комбинаций. Одна шестнадцатеричная цифра занимает один полубайт, а две шестнадцатеричные цифры – один байт. В настоящее время полубайты уже не находят широкого применения, однако этот термин все же стоит знать, да и звучит он забавно (в английском языке *nibble* означает откусывать что-либо маленькими кусочками).

Микропроцессор обрабатывает данные не целиком, а небольшими блоками, называемыми словами. Размер *слова (word)* не является величиной, установленной раз и навсегда, а определяется архитектурой каждого конкретного микропроцессора. На момент написания этой главы (в 2012 году) абсолютное большинство компьютеров использовало 64-битные процессоры. Такие процессоры обрабатывают информацию блоками (словами) длиной 64 бита. А еще не так давно верхом совершенства считались компьютеры, обрабатывающие информацию словами длиной 32 бита. Интересно, что и сегодня наиболее простые микропроцессоры и особенно те, что управляют работой таких бытовых

устройств, как, например, тостеры или микроволновые печи, используют слова длиной 16 бит или даже 8 бит.

В рамках одной группы битов конечный бит, находящийся на одном конце этой группы (обычно правом), называется *наименее значимым битом (least significant bit, lsb*), или просто *младшим битом*, а бит на другом конце группы называется *наиболее значимым битом (most significant bit, msb)*, или *старшим битом*. **Рис. 1.7 (a)** демонстрирует наименее и наиболее значимые биты в случае 6-битного двоичного числа. Аналогичным образом, внутри одного слова можно выделить наименее значимый байт (least significant byte, LSB), или младший байт, и наиболее значимый байт (most significant byte, MSB), или старший байт. **Рис. 1.7 (b)** показывает, как это делается в случае 4-байтного числа, записанного восемью шестнадцатеричными цифрами.

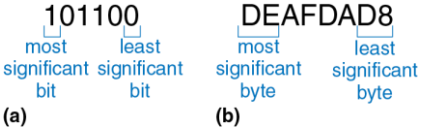


Рис. 1.7 **Наименее и наиболее значимые биты, и байты**

В силу удачного совпадения $2^{10} = 1024 \approx 10^3$. Этот факт позволяет нам использовать приставку *кило* (греческое название тысячи) для сокращенного обозначения 2^{10}. Например, 2^{10} байт – это один килобайт (1 КБ). Подобным же образом, *мега* (греческое название миллиона) обозначает $2^{20} \approx 10^6$, а *гига* (греческое название миллиарда) указывает на $2^{30} \approx 10^9$. Зная, что $2^{10} \approx 1$ тысяча, $2^{20} \approx 1$ миллион, $2^{30} \approx 1$ миллиард и помня значения степеней двойки до 2^9 включительно, будет легко приблизительно рассчитать в уме любую другую степень двух.

Если подходить абсолютно строго к терминологии, то микропроцессором называется такой процессор, все элементы которого размещается на одной микросхеме. До 70-х годов XX века полупроводниковая технология не позволяла разместить процессор целиком на одной микросхеме, поэтому процессоры мощных компьютеров представляли собой набор плат с довольно большим количеством различных микросхем на них. Компания Intel в 1971 году представила первый 4-битный микропроцессор, получивший в качестве названия номер 4004. В наши дни даже самые передовые суперкомпьютеры построены на микропроцессорах, поэтому в этой книге мы будем считать «микропроцессор» и «процессор» тождественными понятиями и использовать оба эти термина как синонимы.

Пример 1.6 ОЦЕНКА СТЕПЕНЕЙ ДВОЙКИ

Найдите приблизительное значение 2^{24} без использования калькулятора.

Решение. Представьте экспоненту как число кратное десяти и остаток.

$2^{24} = 2^{20} \times 2^4$, $2^{20} \approx 1$ миллион. $2^4 = 16$. Итак, $2^{24} \approx 16$ миллионов. На самом деле $2^{24} = 16\,777\,216$, но 16 миллионов – достаточно хорошее приближение для маркетинговых целей.

Так же как 1024 байта называют *килобайтом* (КБ), 1024 бита называют *килобитом* (Кб или кбит). Аналогичным образом, МБ, Мб, ГБ и Гб используются для сокращенного обозначения миллиона и миллиарда байтов и битов. Размеры элементов памяти обычно измеряются в байтах. А вот скорость передачи данных измеряется в битах в секунду. Максимальная скорость передачи данных телефонным модемом, например, составляет 56 килобит в секунду.

1.4.5 Сложение двоичных чисел

Сложение двоичных чисел производится так же, как и сложение десятичных, с той лишь разницей, что двоичное сложение выполнить гораздо проще (см. **Рис. 1.8**). Как и при сложении десятичных чисел, если сумма двух чисел превышает значение, помещающееся в один разряд, мы переносим 1 в следующий разряд. На **Рис. 1.8** для сравнения показано сложение десятичных и двоичных чисел. В крайней

правой колонке на **Рис. 1.8 (a)** складываются числа 7 и 9. Сумма 7 + 9 = 16, что превышает 9, а значит, больше того, что может вместить один десятичный разряд. Поэтому мы записываем в первый разряд 6 (первая колонка), и переносим 10 в следующий разряд (вторая колонка) как 1. Аналогичным же образом при сложении двоичных чисел, если сумма двух чисел превышает 1, мы переносим 2 в следующий разряд как 1. В правой колонке на **Рис. 1.8 (b)**, например, сумма $1 + 1 = 2_{10} = 10_2$, что не может уместиться в одном двоичном разряде. Поэтому мы записываем 0 в первом разряде (первая колонка) и 1 в следующем разряде (вторая колонка). Во второй колонке опять складываются 1 и 1 и еще добавляется 1, перенесенная сюда после сложения чисел в первой колонке. Сумма $1 + 1 + 1 = 3_{10} = 11_2$. Мы записываем 1 в первый разряд (вторая колонка) и снова добавляем 1 в следующий разряд (третья колонка). По очевидной причине бит, добавленный в соседний разряд (колонку), называется битом переноса *(carry bit)*.

```
   11     ← carries →      11
  4277                    1011
+ 5499                  + 0011
  9776                    1110
(a)                     (b)
```

Рис. 1.8 **Примеры сложения с переносом: (a) десятичное (b) двоичное**

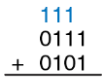


Рис. 1.9 Пример двоичного сложения

Пример 1.7 ДВОИЧНОЕ СЛОЖЕНИЕ

Вычислить $0111_2 + 0101_2$

Решение. На **Рис. 1.9** показано, что сумма равна 1100_2. Переносы выделены синим цветом. Мы можем проверить нашу работу, повторив вычисления в десятичной системе счисления. $0111_2 = 7_{10}$, $0101_2 = 5_{10}$. Сумма равна $12_{10} = 1100_2$

Цифровые системы обычно оперируют числами с заранее определенным и фиксированным количеством разрядов. Ситуацию, когда результат сложения превышает выделенное для него количество разрядов, называют *переполнением (overflow)*. Четырехбитная ячейка памяти, например, может сохранять значения в диапазоне [0, 15]. Такая ячейка переполняется, если результат сложения превышает число 15. В этом случае дополнительный пятый бит отбрасывается, а результат, оставшийся в четырех битах, будет ошибочным. Переполнение можно обнаружить, если следить за переносом бита из наиболее значимого разряда двоичного числа (см. на **Рис. 1.8**), из наиболее левой колонки.

Пример 1.8 СЛОЖЕНИЕ С ПЕРЕПОЛНЕНИЕМ

Вычислить $1101_2 + 0101_2$. Будет ли переполнение?

Решение. На **Рис. 1.10** показано, что сумма равна 10010_2. Результат выходит за границы четырехбитового двоичного числа. Если его нужно запомнить в 4-х битах, наиболее значимый бит пропадет, оставив некорректный результат 0010_2. Если вычисления производятся с числами с пятью или более битами, результат 10010_2 будет корректным.

```
  11 1
  1101
+ 0101
 10010
```

Рис. 1.10 Пример двоичного сложения с переполнением

1.4.6 Знак двоичных чисел

До сих пор мы рассматривали двоичные числа без знака *(unsigned)* – то есть только положительные числа. Часто, однако, для вычислений требуются как положительные, так и отрицательные числа, а это значит, что для знака двоичного числа нам потребуется дополнительный разряд. Существует несколько способов представления двоичных чисел со знаком *(signed)*. Наиболее широко применяются два: *Прямой Код (Sign/Magnitude)* и *Дополнительный Код (Two's Complement)*.

Прямой код

Представление отрицательных двоичных с использованием прямого кода интуитивно покажется вам наиболее привлекательным, поскольку совпадает с привычным способом записи отрицательных чисел, когда сначала идет знак минус, а затем абсолютное значение числа. Двоичное число, состоящее из N битов и записанное в прямом коде, использует наиболее значимый бит для знака, а остальные $N-1$ бита для записи абсолютного значения этого числа. Если наиболее значимый бит 0, то число положительное. Если наиболее значимый бит 1, то число отрицательное.

Пример 1.9 ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В ПРЯМОМ КОДЕ

Запишите числа 5 и –5 как четырехбитовые числа в прямом коде

Решение: Оба числа имеют абсолютную величину $5_{10} = 101_2$. Таким образом, $5_{10} = 0101_2$ и $-5_{10} = 1101_2$.

К сожалению, стандартный способ сложения не работает в случае двоичных чисел со знаком, записанных в прямом коде. Например, складывая $-5_{10} + 5_{10}$ привычным способом, получаем $1101_2 + 0101_2 = 10010_2$. Что, естественно, есть полный абсурд.

Двоичная переменная длиной N битов в прямом коде может представлять число в диапазоне $[-2^{N-1} + 1, 2^{N-1} - 1]$.

Другой несколько странной особенностью прямого кода является наличие +0 и −0, причем оба этих числа соответствуют одному нулю. Нетрудно предположить, что представление одной и той же величины двумя различными способами чревато ошибками.

Дополнительный код

Двоичные числа, записанные с использованием дополнительного кода, и двоичные числа без знака идентичны, за исключением того, что в случае дополнительного кода вес наиболее значимого бита -2^{N-1} вместо 2^{N-1}, как в случае двоичного числа без знака. Дополнительный код гарантирует однозначное представление нуля, допускает сложение чисел по привычной схеме, а значит, избавлен от недостатков прямого кода.

В случае дополнительного кода нулевое значение представлено нулями во всех разрядах двоичного числа: $00\ldots000_2$. Максимальное положительное значение представлено нулем в наиболее значимом разряде и единицами во всех других разрядах двоичного числа: $01\ldots111_2 = 2^{N-1} - 1$. Максимальное отрицательное значение имеет единицу в наиболее значимом разряде и нули во всех остальных разрядах: $10\ldots000_2 = -2^{N-1}$. Отрицательная единица представлена единицами во всех разрядах двоичного числа: $11\ldots111_2$.

Ракета Ариан-5 ценой 7 миллиардов долларов, запущенная 4 июня 1996 года, отклонилась от курса и разрушилась через 40 секунд после запуска. Отказ был вызван тем, что в бортовом компьютере произошло переполнение 16-разрядных регистров, после которого компьютер вышел из строя.

Программное обеспечение Ариан-5 было тщательно протестировано, но на ракете Ариан-4. Однако новая ракета имела двигатели с более высокими скоростными параметрами, которые, будучи переданными бортовому компьютеру, и вызвали переполнение регистров.

(Фото: ESA/CNES/ARIANESPACE- Service Optique CS6)

Обратите внимание на то, что наиболее значимый разряд у всех положительных чисел – это «0», в то время как у отрицательных чисел – это «1», то есть наиболее значимый бит дополнительного кода можно рассматривать как аналог знакового бита прямого кода. Однако на этом сходство кончается, поскольку остальные биты дополнительного кода интерпретируются не так, как биты прямого кода.

В случае дополнительного кода, знак отрицательного двоичного числа изменяется на противоположный путем выполнения специальной операции, называемой *дополнением до двух (taking the two's complement)*. Суть этой операции заключается в том, что инвертируются все биты этого числа, а затем к значению наименее значимого бита прибавляется 1. Подобная операция позволяет найти двоичное представление отрицательного числа или определить его абсолютное значение.

Пример 1.10 ПРЕДСТАВЛЕНИЕ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Найти представление -2_{10} как 4-битового числа в дополнительном коде.

Решение: начните с $+2_{10} = 0010_2$. Для получения -2_{10} инвертируйте биты и добавьте единицу. Инвертируя 0010_2, получим 1101_2. $1101_2 + 1 = 1110_2$. Итак, -2_{10} равно 1110_2.

Пример 1.11 ЗНАЧЕНИЕ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Найти десятичное значение числа 1001_2 в дополнительном коде.

Решение: Число 1001_2 имеет старшую 1, поэтому оно должно быть отрицательным. Чтобы найти его модуль, инвертируем все биты и добавляем 1. Инвертируя 1001_2, получим 0110_2. $0110_2 + 1 = 0111_2 = 7_{10}$. Отсюда, $1001_2 = -7_{10}$.

Неоспоримым преимуществом дополнительного кода является то, что привычный способ сложения работает как в случае положительных, так и отрицательных чисел. Напомним, однако, что при сложении *N*-битных чисел *N*-ный бит (т.е. *N* + 1-й бит результата) не переносится.

Пример 1.12 СЛОЖЕНИЕ ЧИСЕЛ, ПРЕДСТАВЛЕННЫХ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Вычислить (a) $-2_{10} + 1_{10}$ и (b) $-7_{10} + 7_{10}$ с помощью чисел в дополнительном коде

Решение: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. Пятый бит отбрасывается, оставляя правильный 4-битовый результат 0000_2.

Вычитание одного двоичного числа из другого осуществляется путем преобразования вычитаемого в дополнительный код и последующего его сложения с уменьшаемым.

Пример 1.13 ВЫЧИТАНИЕ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Вычислить (a) $5_{10} - 3_{10}$ и (b) $3_{10} - 5_{10}$, используя 4-разрядные числа в дополнительном коде.

Решение:

(a) $3_{10} = 0011_2$. Вычисляя его дополнительный код, получим $-3_{10} = 1101_2$. Теперь сложим $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Отметим, что перенос из наиболее значимой позиции сбрасывается, поскольку результат записывается в четырех битах.

b) Вычисляя дополнительный код от 5_{10}, получим $-5_{10} = 1011_2$.

Теперь сложим $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

Дополнение нуля до двух также производится путем инвертирования всех битов (это дает $11\ldots111_2$) и последующим прибавлением 1, что делает значения всех битов равным 0. При этом перенос наиболее значимого бита игнорируется. В результате, нулевое значение всегда представлено набором только нулевых битов. В отличие от прямого кода дополнительный код не имеет отрицательного нуля. Нуль всегда считается положительным числом, так как его знаковый бит всегда 0.

Так же, как и двоичное число без знака, произвольное *N*-битное число, записанное в дополнительном коде, может принимать одно из 2^N возможных значений. Однако, весь этот диапазон разделен между

положительным и отрицательным числами. Например, 4-битное двоичное число без знака может принимать 16 значений от 0 до 15. В случае дополнительного кода, 4-битное число также принимает 16 значений, но уже от −8 до 7. В общем случае, диапазон *N*-битного числа, записанного в дополнительном коде, охватывает $[-2^{N-1}, 2^{N-1} - 1]$. Легко понять, почему в отрицательном диапазоне оказалось на одно значение больше, чем в положительном – в дополнительном коде отсутствует отрицательный нуль. Максимальное отрицательное число, которое можно записать, используя дополнительный код $10\ldots000_2 = -2^{N-1}$, иногда называют *странным числом (weird number)*. Чтобы дополнить это число до двух, инвертируем все его биты (это даст нам $01\ldots111_2$), прибавим 1 и получим в результате $10\ldots000_2$ – опять это же самое «странное» число. То есть, это единственное отрицательное число, которое не имеет положительной пары.

В случае дополнительного кода сложение двух положительных или отрицательных *N*-битовых чисел может привести к переполнению, если результат будет больше, чем $2^{N-1} - 1$, или меньше, чем -2^{N-1}. Сложение положительного и отрицательного числа, напротив, никогда не приводит к переполнению. В отличие от двоичного числа без знака перенос наиболее значимого бита не является признаком переполнения. Вместо этого индикатором переполнения является

ситуация, когда после сложения двух чисел с одинаковым знаком знаковый бит суммы не совпадает со знаковыми битами слагаемых.

Пример 1.14 СЛОЖЕНИЕ ЧИСЕЛ В ДОПОЛНИТЕЛЬНОМ КОДЕ С ПЕРЕПОЛНЕНИЕМ

Вычислить $4_{10} + 5_{10}$, используя четырехбитные числа в дополнительном коде. Произойдет ли переполнение?

Решение: $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. Результат не помещается в диапазон положительных четырехбитных чисел в дополнительном коде, оказываясь отрицательным. Если бы вычисление выполнялось с пятью или более битами, результат был бы такой $01001_2 = 9_{10}$, что правильно.

В случае необходимости увеличения количества битов произвольного числа, записанного в дополнительном коде, значение знакового бита должно быть скопировано в наиболее значимые разряды модифицированного числа. Эта операция называется знаковым расширением *(sign extension)*. Например, числа 3 и −3 записываются в 4-битном дополнительном коде как 0011 и 1101 соответственно. Если мы увеличиваем число разрядов до семи битов, мы должны скопировать знаковый бит в три наиболее значимых бита модифицированного числа, что дает 0000011 и 1111101.

Сравнение способов представления двоичных чисел

Три наиболее часто использующиеся на практике способа представления двоичных чисел – это двоичные числа без знака, прямой код и дополнительный код. **Табл. 1.3** сравнивает диапазон *N*-битных чисел для каждого из этих трех способов. Преимущества дополнительного кода заключаются в том, что его можно использовать для представления как положительных, так и отрицательных целых чисел, а привычный способ сложения работает для всех чисел, представленных в дополнительном коде. Вычитание осуществляется путем преобразования вычитаемого в отрицательное число (т.е. путем дополнения этого числа до двух) и последующего сложения с уменьшаемым. В дальнейшем в этой книге, если не указано иное, предполагается, что все двоичные числа представлены в дополнительном коде.

Табл. 1.3 Диапазон *N*-битных чисел

Система	Диапазон
Двоичные числа без знака	$[0, 2^N - 1]$
Прямой код	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Дополнительный код	$[-2^{N-1}, 2^{N-1} - 1]$

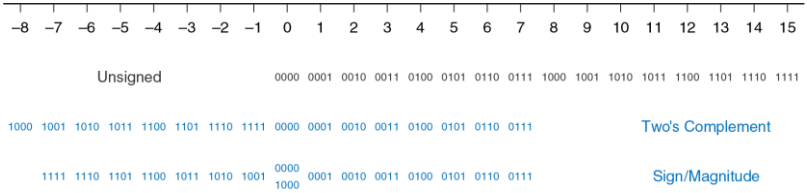


Рис. 1.11 **Числовая шкала и 4-битовое двоичное кодирование**

На **Рис. 1.11** изображена десятичная числовая шкала с соответствующими десятичными и 4-битными двоичными числами, представленными тремя вышеперечисленными способами. Двоичные числа без знака находятся в диапазоне [0, 15] и располагаются в обычном порядке. 4-битные двоичные числа, представленные в дополнительном коде, занимают диапазон [−8, 7]. Причем, положительные числа [0, 7] используют точно такую же кодировку, как и двоичные числа без знака. Отрицательные же числа [–8, –1] кодируются таким образом, что наибольшее двоичное значение каждого такого числа без знака представляет число, наиболее близкое к 0. Обратите внимание на то, что «странное число» 1000 соответствует десятичному значению −8 и не имеет положительной пары. Числа, представленные в прямом коде, занимают диапазон [−7, 7]. При этом, наиболее значимый бит является знаковым. Положительные числа

[0, 7] используют такую же кодировку, как и двоичные числа без знака. Отрицательные числа симметричны положительным, с той лишь разницей, что их знаковый бит имеет значение 1. Нуль представлен двумя значениями 0000 и 1000. В результате того, что два числа соответствуют одному нулю, любое произвольное *N*-разрядное двоичное число в прямом коде может представлять только $2^N - 1$ целых числа.

1.5 ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ

Теперь, когда мы знаем, как использовать бинарные переменные для представления информации, рассмотрим цифровые системы, способные выполнять различные операции с этими переменными. *Логические вентили (logic gates)* – это простейшие цифровые схемы, получающие один или более двоичных сигналов на входе и производящие новый двоичный сигнал на выходе. При графическом изображении логических вентилей для обозначения одного или нескольких входных сигналов и выходного сигнала используются специальные символы. Если смотреть на изображение логического элемента, то входные сигналы обычно размещаются слева (или сверху), а выходные сигналы – справа (или снизу). Разработчики цифровых систем обычно используют первые буквы латинского алфавита для обозначения входных сигналов и латинскую букву *Y* для обозначения

выходного сигнала. Взаимосвязь между входными сигналами и выходным сигналом логического вентиля может быть описана с помощью *таблицы истинности (truth table)* или уравнением Булевой логики. Слева в таблице истинности представлены значения входных сигналов, а справа – значение соответствующего выходного сигнала. Каждая строка в такой таблице соответствует одной из возможных комбинаций входных сигналов. Уравнение Булевой логики – это математическое выражение, описывающее логический элемент с помощью двоичных переменных.

1.5.1 Логический вентиль НЕ

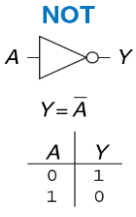


Рис. 1.12 **Вентиль НЕ**

Логический вентиль *НЕ* (*NOT gate)* имеет один вход *A* и один выход *Y*, как показано на **Рис. 1.12**. Причем выходной сигнал Y – это сигнал, обратный входному сигналу A, или, как еще говорят, *инвертированный A (inversed A)*. Если сигнал на входе *A* – это ЛОЖЬ, то сигнал на выходе *Y* будет ИСТИНА. Таблица истинности и уравнение Булевой логики на **Рис. 1.12** суммируют эту связь входного и выходного сигналов. В уравнении Булевой логики линия над обозначением сигнала читается как «не», то есть

математическое выражение $Y = \overline{A}$ произносится как «Y равняется не A». Именно поэтому логический вентиль НЕ также называют *инвертором (inverter)*.

Для обозначения логического вентиля НЕ используют и другие способы записи, включая: $Y = A'$, $Y = \neg A$, $Y = !A$ и $Y = \sim A$. В этой книге мы будем пользоваться исключительно записью $Y = \overline{A}$, однако не удивляйтесь, если в научной и технической литературе вы столкнетесь и с другими обозначениями.

1.5.2 Буфер

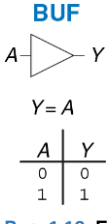


Рис. 1.13 **Буфер**

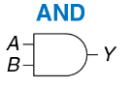
Другим примером логического вентиля с одним входом является *буфер (buffer),* показанный на **Рис. 1.13.**

Буфер просто копирует входной сигнал на выход. Если рассматривать буфер как часть логической схемы, то такой элемент ничем не отличается от простого провода и может показаться бесполезным. Вместе с тем, на аналоговом уровне буфер может обеспечить характеристики, необходимые для нормального функционирования разрабатываемого устройства.

Буфер, например, необходим для передачи большого тока электродвигателю или для быстрой передачи сигнала сразу нескольким логическим элементам. Это еще один пример, доказывающий необходимость рассмотрения любой системы с нескольких уровней абстракции, если мы хотим в полной мере понять эту систему. Рассмотрение буфера только с позиции цифрового уровня абстракции не позволяет нам разглядеть его реальную функцию.

В логических схемах буфер обозначается треугольником. Кружок на выходе логического элемента, в англоязычной литературе часто называемый *пузырем (bubble)*, указывает на инверсию сигнала, как, например, показано на **Рис. 1.12**.

1.5.3 Логический вентиль И



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

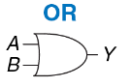
Рис. 1.14 Вентиль И

Логические вентили с двумя входными сигналами гораздо интереснее, чем вентиль НЕ и буфер. Логический вентиль *И (AND gate)*, показанный на **Рис. 1.14**, выдает значение ИСТИНА на выход *Y* исключительно только если оба входных сигнала *A* и *B* имеют значение ИСТИНА. В противном случае выходной сигнал Y имеет значение ЛОЖЬ.

В используемом нами соглашении входные сигналы перечислены в порядке 00, 01, 10, 11, как в случае подсчета в двоичной системе счисления. Уравнение Булевой логики для логического элемента И может быть записано несколькими способами: $Y = A \bullet B$, $Y = AB$, или $Y = A \cap B$. Символ ∩ читается как «пересечение» и больше других нравится специалистам в математической логике. Однако, в этой книге мы предпочитаем использовать выражение $Y = AB$, которое звучит как «*Y* равно *A* и *B*», просто потому, что мы достаточно ленивы, чтобы выбрать то, что короче.

> По словам создателя языка программирования Perl Ларри Уолла, три основных достоинства программиста – это лень, нетерпение и самоуверенность.

1.5.4 Логический вентиль ИЛИ



$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

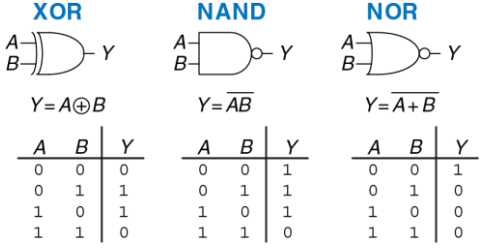
Рис. 1.15 **Вентиль ИЛИ**

Логический вентиль ИЛИ (OR gate), показанный на **Рис. 1.15**, выдаёт значение ИСТИНА на выход *Y*, если хотя бы один из двух входных сигналов *A* или *B* имеет значение ИСТИНА. Уравнение Булевой логики для логического элемента ИЛИ записывается как $Y = A + B$ или $Y = A \cup B$.

Символ $\cup$ читается как «объединение» и опять же больше всего нравится математикам. Разработчики цифровых систем обычно пользуются простым символом +.

Математическое выражение $Y = A + B$ звучит «*Y* равно *A* или *B*».

1.5.5 Другие логические элементы с двумя входными сигналами



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

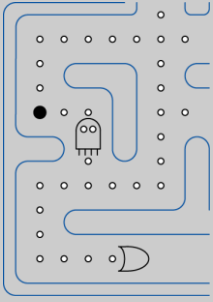
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Рис. 1.16 **Другие логические элементы с двумя входными сигналами**

На **Рис. 1.16** показаны другие широко распространенные логические вентили с двумя входными сигналами. Добавление кружка на выходе любого логического вентиля превращает этот вентиль в ему противоположный – то есть инвертирует его. Таким образом, например, из вентиля *И* получается вентиль *И-НЕ (NAND gate)*. Значение выходного сигнала *Y* вентиля И-НЕ будет ИСТИНА до тех пор, пока оба входных сигнала *A* и *B* не примут значение ИСТИНА. Точно так же из логического вентиля ИЛИ получается вентиль *ИЛИ-НЕ (NOR gate)*. Его выходной сигнал *Y* будет ИСТИНА в том случае, если ни один из входных сигналов, ни *A* ни *B*, не имеет значение ИСТИНА. *Исключающее ИЛИ с количеством входов равным N (N-input XOR gate)*

иногда еще называют элементом контроля по чётности (parity gate). Такой вентиль выдает на выход сигнал ИСТИНА, если нечетное количество входных сигналов имеет значение ИСТИНА. Как и в случае элемента с двумя входными сигналами, комбинации сигналов для элемента с *N* входами перечислены в логической таблице в порядке подсчета в двоичной системе счисления.

Забавный способ запомнить, как обозначается элемент ИЛИ на логических схемах, заключается в том, что графический символ ИЛИ напоминает главный персонаж компьютерной игры Pacman. Причем, широко раскрытая пасть “голодного” ИЛИ находится со стороны входных сигналов и готова проглотить все сигналы ИСТИНА, которые только может найти!



Пример 1.15 Вентиль исключающее ИЛИ-НЕ

На **Рис. 1.17** показаны обозначение и булевское уравнение для вентиля исключающее ИЛИ-НЕ (XNOR) с двумя входами, который выполняет инверсию исключающего ИЛИ. Заполните таблицу истинности.

Решение: На **Рис. 1.18** представлена таблица истинности. Выход исключающего ИЛИ-НЕ есть ИСТИНА, если оба входа имеют значение ЛОЖЬ или оба входа имеют значение ИСТИНА.

A	B	Y
0	0	
0	1	
1	0	
1	1	

Рис. 1.17 Вентиль исключающее ИЛИ-НЕ

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Рис. 1.18 Таблица истинности вентиля исключающее ИЛИ-НЕ

Вентиль исключающее ИЛИ-НЕ с двумя входами иногда называют вентилем равенства, так как его выход есть ИСТИНА, когда входы совпадают.

1.5.6 Логические элементы с количеством входов больше двух

Многие Булевы функции, а значит, и логические вентили, необходимые для их реализации, оперируют тремя и более входными сигналами. Наиболее распространенные из таких вентилей – это И, ИЛИ, Исключающее ИЛИ, И-НЕ, ИЛИ-НЕ и Исключающее ИЛИ-НЕ. Логический вентиль И с количеством входов равным *N* выдает значение ИСТИНА, когда значения на всех *N* входах этого логического вентиля ИСТИНА. Логический вентиль ИЛИ с количеством входов равным *N* выдает ИСТИНА, когда значение хотя бы одного из его входов ИСТИНА.

Пример 1.16 ВЕНТИЛЬ ИЛИ-НЕ С ТРЕМЯ ВХОДАМИ

На **Рис. 1.19** показаны обозначение и булевское уравнение для вентиля ИЛИ-НЕ с тремя входами. Заполните таблицу истинности.

Решение: На **Рис. 1.20** показана таблица истинности. Выход есть ИСТИНА только, если нет ни одного входа со значением ИСТИНА.

$$Y = \overline{A + B + C}$$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Рис. 1.19 Вентиль ИЛИ-НЕ с тремя входами

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Рис. 1.20 Таблица истинности вентиля ИЛИ-НЕ с тремя входами

Пример 1.17 ВЕНТИЛЬ И С ЧЕТЫРЬМЯ ВХОДАМИ

На **Рис. 1.21** показаны обозначение и булевское уравнение для вентиля И с четырьмя входами. Заполните таблицу истинности.

Решение: На **Рис. 1.22** показана таблица истинности. Выход есть ИСТИНА только если все входы имеют значение ИСТИНА.

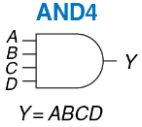


Рис. 1.21 **Вентиль И с четырьмя входами**

A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рис. 1.22 **Таблица истинности вентиля И с четырьмя входами**

1.6 ЗА ПРЕДЕЛАМИ ЦИФРОВОЙ АБСТРАКЦИИ

Цифровая система оперирует дискретными переменными. Однако для представления этих переменных используются непрерывные физические величины, такие как напряжение в электрической цепи, положение шестеренок в механической передаче или уровень жидкости в гидравлическом цилиндре. Задача разработчика цифровой системы – определить, каким образом непрерывно-меняющаяся величина соотносится с конкретным значением дискретной переменной.

Рассмотрим, например, задачу представления двоичного сигнала A напряжением в электрической цепи. Допустим, что напряжение 0 В соответствует значению $A = 0$, а напряжение 5 В соответствует $A = 1$. Однако, реальная цифровая система должна быть устойчива к неизбежному в такой ситуации шуму, так что значение 4,97 В, вероятно, также следует толковать как $A = 1$. А что делать, если напряжение равно 4,3 В? Или 2,8 В? Или 2,500000 В?

1.6.1 Напряжение питания

Предположим, что минимальное напряжение в электронной цифровой системе, называемое также *напряжением земли (ground voltage, или просто ground, или GND)*, составляет 0 В. Самое высокое напряжение в системе поступает от блока питания и, как правило, обозначается V_{DD}.

Транзисторные технологии семидесятых и восьмидесятых годов прошлого века в основном использовали V_{DD} равное 5 В. С переходом на транзисторы меньшего размера, V_{DD} последовательно снижали до 3,3 В, 2,5 В, 1,8 В, 1,5 В, 1,2 В и даже ниже для экономии электроэнергии и во избежание перегрузки транзисторов.

1.6.2 Логические уровни

Отображение непрерывно-меняющейся переменной на различные значения дискретной двоичной переменной выполняется путем определения *логических уровней,* как показано на **Рис. 1.23**. Первый логический элемент в рассматриваемой схеме называется *источник (driver)*, а второй – *приемник (receiver)*. Выходной сигнал источника подключается ко входу приемника. Источник выдает выходной сигнал низкого напряжения (0) в диапазоне от 0 В до V_{OL} или выходной сигнал высокого напряжения (1) в диапазоне от V_{OH} до V_{DD}. Если приемник получает на вход сигнал в диапазоне от 0 до V_{IL}, он рассматривает такой сигнал как нуль. Если приемник получает на вход сигнал в диапазоне от V_{IH} до V_{DD}, он рассматривает такой сигнал как единицу. Если же по какой-либо причине, например, наличия шумов или неисправности одного из элементов схемы, напряжение сигнала на входе приемника падает настолько, что попадает в *запретную зону (forbidden zone)* между V_{IL} и V_{IH}, то поведение этого логического

элемента становится непредсказуемым. V_{OH} и V_{OL} называются соответственно *высоким и низким логическими уровнями выхода (output high and low logic levels)*, а V_{IH} и V_{IL} называются соответственно *высоким и низким логическими уровнями входа (input high and low logic levels)*.

1.6.3 Допускаемые Уровни Шумов

Для того чтобы выходной сигнал источника был правильно интерпретирован на входе приемника, необходимо, чтобы $V_{OL} < V_{IL}$ и $V_{OH} > V_{IH}$. В этом случае, даже если выходной сигнал источника будет загрязнен шумами, приемник по-прежнему сможет правильно определить логический уровень входного сигнала. *Допускаемый уровень шумов (noise margin)* – это то максимальное количество шума, присутствие которого в выходном сигнале источника не мешает приемнику корректно интерпретировать значение полученного сигнала. Как можно увидеть на **Рис. 1.23**, значения *нижнего допускаемого уровня шумов (low noise margin)* и *верхнего допускаемого уровня шумов (high noise margin)* определяются следующим образом:

$$NM_L = V_{IL} - V_{OL} \quad (1.2)$$

$$NM_H = V_{OL} - V_{IH} \quad (1.3)$$

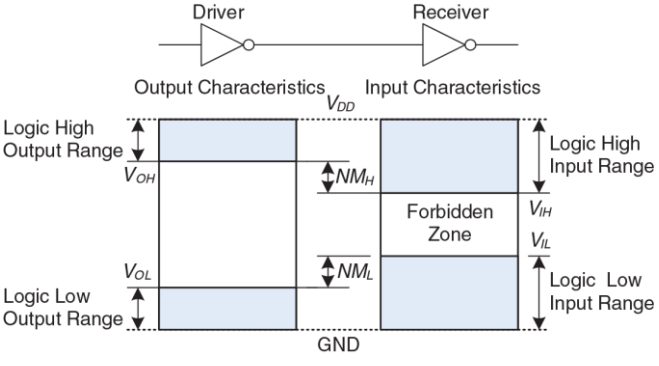


Рис. 1.23 **Логические уровни и уровни шума**

V_{DD} обозначает напряжение стока (*drain*) в транзисторах, построенных на структуре металл-оксид-полупроводник (МОП). Такие транзисторы используются сегодня для создания самых современных микросхем. Напряжение источника питания иногда также обозначают V_{CC}, как напряжение коллектора (*collector*) в биполярных транзисторах более ранних микросхем. Напряжение земли (ground voltage или просто ground) иногда обозначают как V_{SS} потому, что это напряжение на истоке (*source*) МОП-транзистора. Для более подробной информации о том, как функционирует транзистор, смотри **раздел 1.7**.

Пример 1.18 РАСЧЕТ УРОВНЕЙ ШУМА

Рассмотрим схему с инверторами на **Рис. 1.24**. V_{O1} – это напряжение на выходе инвертора I1, а V_{I2} – напряжение на входе инвертора I2. Оба инвертора имеют следующие характеристики: V_{DD} = 5 В, V_{IL} = 1,35 В, V_{IH} = 3,15 В, V_{OL} = 0,33 В, и V_{OH} = 3,84 В. Каковы нижний и верхний уровни шума? Может ли схема корректно обработать уровень шума в 1 В между V_{O1} и V_{I2}?

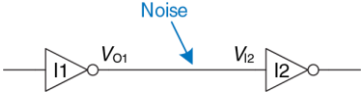


Рис. 1.24 **Схема с инверторами**

Решение: Границы уровня шума инвертора следующие: $NM_L = V_{IL} - V_{OL}$ = (1,35 В − 0,33 В) = 1,02 В, $NM_H = V_{OH} - V_{IH}$ = (3,84 В − 3,15 В) = 0,69 В. Схема может корректно обработать шум в 1В когда на выходе НИЗКИЙ уровень (NML = 1,02 В), но не когда на выходе ВЫСОКИЙ уровень (NM_H = 0,69 *V*). Например, предположим, что инвертор I1 имеет на выходе в наихудшем случае ВЫСОКОЕ значение, $V_{O1} = V_{OH}$ = 3,84 В. Если наличие шума вызовет падение напряжения на 1 В на входе инвертора I2, тогда V_{I2} = (3,84 В − 1 В) = 2,84 В. Это меньше, чем допустимое входное значение ВЫСОКОГО уровня, V_{IH} = 3,15 В, поэтому инвертор I2 может не принять правильное входное значение ВЫСОКОГО уровня.

1.6.4 Передаточная Характеристика

Для понимания предела цифровой абстракции мы должны рассмотреть поведение логических вентилей с аналоговой точки зрения. *Передаточная характеристика (DC transfer characteristics)* какого-либо логического вентиля описывает напряжение на выходе этого элемента как функцию напряжения на его входе, когда входной сигнал изменяется настолько медленно, что выходной сигнал успевает изменяться вслед за ним. Такая характеристика называется передаточной, поскольку описывает взаимосвязь между входным и выходным напряжением.

В случае идеального инвертора переключение будет резким в точке $V_{DD}/2$, как показано на **Рис. 1.25 (а)**. Для $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. Для $V(A) > V_{DD}/2$, $V(Y) = 0$. В этом случае, $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ и $V_{OL} = 0$.

> *DC* указывает на состояние, когда напряжение на входе электронной системы поддерживается постоянным или изменяется так медленно, что остальные параметры системы плавно изменяются вместе с ним. Исторически термин *DC* ведет свое происхождение от понятия постоянный ток (*direct current*) – метод передачи электрической энергии по схеме на расстояние, когда напряжение в линии поддерживается постоянным. В отличие от *DC*, переходная характеристика (*transient response*) схемы – это состояние, когда входное напряжение меняется быстро. Переходные процессы рассматриваются в **разделе 2.9**.

Напряжение при переключении реального инвертора изменяется постепенно между граничными значениями – так, как показано на **Рис. 1.25 (b)**. Если входное напряжение $V(A)$ равно 0, то напряжение на выходе $V(Y) = V_{DD}$. Если $V(A) = V_{DD}$, то $V(Y) = 0$. Однако, переход между этими конечными точками плавный и может находиться правее или левее значения $V_{DD}/2$. В связи с этим, возникает закономерный вопрос, как в этом случае определить логические уровни.

Разумно выбрать в качестве логических уровней те две точки, где наклон передаточной характеристики $dV(Y)/dV(A)$ равен −1. Такие точки называются *граничные коэффициенты передачи (unity gain points)*. Подобный выбор обычно максимизирует допускаемые уровни шумов. При уменьшении V_{IL} V_{OH} увеличивается незначительно. Однако, если V_{IL} растет, V_{OH} падает практически отвесно.

1.6.5 Статическая Дисциплина

Для того, чтобы избежать попадания входных сигналов в запретные зоны, логические вентили должны разрабатываться в соответствии с *принципом статической дисциплины (static discipline)*. Принцип статической дисциплины требует, чтобы при условии наличия логически корректных сигналов на входе каждый элемент системы выдавал логически корректные сигналы на выходе.

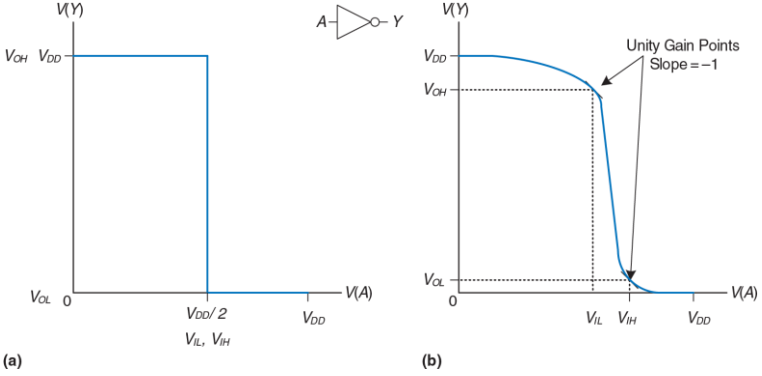


Рис. 1.25 **Передаточные характеристики и уровни шума**

Применение принципа статической дисциплины ограничивает свободу разработчика в выборе аналоговых элементов для построения цифровых систем, однако помогает обеспечить простоту и надежность разрабатываемых цифровых схем. Используя этот принцип, разработчик поднимается с аналогового уровня абстракции на цифровой, что увеличивает производительность проектировщика, избавляя его от рассмотрения излишних деталей.

Выбор V_{DD} и логических уровней может быть произвольным, однако этот выбор должен обеспечить совместимость всех логических вентилей, обменивающихся данными в пределах одной цифровой системы. Поэтому вентили обычно группируются в *семейства логики (logic families)* таким образом, что любой элемент из одного семейства при соединении с любым другим элементом из этого же семейства автоматически обеспечивает соблюдение принципа статической дисциплины. Логические вентили одного семейства соединяются друг с другом так же легко, как и блоки конструктора Лего, поскольку они полностью совместимы по напряжению источника питания и логическим уровням.

Четыре основные семейства логических вентилей доминировали с 70-х по 90-е годы прошлого века – это *ТТЛ – транзисторно-транзисторная логика (Transistor-Transistor Logic*, или *TTL)*, *КМОП – логика, построенная на комплементарной структуре металл-оксид-полупроводник (Complementary Metal-Oxide-Semiconductor Logic*, или *CMOS)*, *НТТЛ – низковольтная транзисторно-транзисторная логика (Low-Voltage Transistor-Transistor Logic*, или *LVTTL*) и *НКМОП низковольтная логика на комплементарной структуре металл-оксид-полупроводник (Low-Voltage Complementary Metal-Oxide-Semiconductor Logic*, или *LVCMOS)*. Логические уровни для всех этих семейств представлены в **Табл. 1.4**. Начиная с 90-х годов прошлого

века, четыре вышеперечисленных семейства распались на большое количество более мелких семейств в связи со все большим распространением устройств, требующих еще более низкого напряжения питания. В **приложении А.6** наиболее распространённые семейства логических вентилей рассматриваются детально.

Табл. 1.4 Семейства логики с уровнями напряжения 5 В и 3,3 В

Семейство логики	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4,75 − 5,25)	0,8	2,0	0,4	2,4
CMOS	5 (4,5 − 6)	1,35	3,15	0,33	3,84
LVTTL	3.3 (3 − 3,6)	0,8	2,0	0,4	2,4
LVCMOS	3.3 (3 − 3,6)	0,9	1,8	0,36	2,7

Пример 1.19 СОВМЕСТИМОСТЬ ЛОГИЧЕСКИХ СЕМЕЙСТВ

Какие из логических семейств из **Табл. 1.4** могут надежно взаимодействовать между собой?

Решение: в **Табл. 1.5** перечислены логические семейства, которые имеют совместимые логические уровни. Заметим, что пятивольтовые логические семейства, такие как TTL и CMOS, могут выдавать на выход ВЫСОКИЙ уровень в 5 В. Если этот пятивольтовый сигнал подается на вход семейству с уровнем 3,3 В, такому как LVTTL или LVCMOS, это может повредить приемник, если в спецификации последнего не указана прямо, что он «5 В-совместимый».

Табл. 1.5 Совместимость логических семейств

		Приемник			
		TTL	CMOS	LVTTL	LVCMOS
Источник	TTL	ДА	НЕТ: $V_{OH} < V_{IH}$	ВОЗМОЖНО[a]	ВОЗМОЖНО[a]
	CMOS	ДА	ДА	ВОЗМОЖНО[a]	ВОЗМОЖНО[a]
	LVTTL	ДА	НЕТ: $V_{OH} < V_{IH}$	ДА	ДА
	LVCMOS	ДА	НЕТ: $V_{OH} < V_{IH}$	ДА	ДА

[a] если сигнал в 5 В ВЫСОКОГО уровня не может повредить вход приемника

1.7 КМОП ТРАНЗИСТОРЫ*

Этот раздел, как и другие разделы, помеченные значком *, являются дополнительными и не являются абсолютно необходимыми для понимания основной темы этой книги.

Аналитическая Машина Беббиджа была механическим устройством с пружинами и шестеренками, а в первых компьютерах использовались реле или вакуумные трубки. Современные компьютеры используют транзисторы, потому что они дешевы, имеют малые размеры и высокую надежность. Транзистор – это переключатель с двумя положениями «включить» и «выключить», контролируемый путем подачи напряжения или тока на управляющую клемму. Существуют два основных типа транзисторов – *биполярные транзисторы (bipolar junction transistors)* и *МОП-транзисторы – металл-оксид-полупроводник- транзисторы* (иногда говорят полевые транзисторы – *metal-oxide-semiconductor field effect transistors*, или *MOSFET*).

В 1958 году Джек Килби из Texas Instruments создал первую интегральную схему, состоявшую из двух транзисторов. В 1959 году Роберт Нойс, работавший тогда в Fairchild Semiconductor, запатентовал метод соединения нескольких транзисторов на одном кремниевом кристалле. В то время один транзистор стоил около 10 американских долларов.

Роберт Нойс, 1927 –1990 Родился в городе Берлингтон штата Айова и получил степень бакалавра в области физики в Гриннеллском колледже, а степень доктора наук в области физики – в Массачусетском Технологическом Институте. Роберта Нойса прозвали “мэром Силиконовой долины” за его обширный вклад в развитие микроэлектроники.

Нойс стал со-основателем Fairchild Semiconductor в 1957 году и корпорации Intel в 1968 году. Он также является одним из изобретателей интегральной микросхемы. Инженеры из групп, возглавляемых Нойсом, в дальнейшем основали целый ряд выдающихся полупроводниковых компаний. (Воспроизводится с разрешения Intel Corporation © 2006 г).

Сегодня, после более чем трех десятилетий беспрецедентного развития полупроводниковой технологии, инженеры могут «упаковать» приблизительно один миллиард полевых МОП-транзисторов на одном квадратном сантиметре кристалла кремния, причем каждый из этих транзисторов будет стоить меньше десяти микроцентов. Плотность размещения транзисторов на кристалле возрастает на порядок, а себестоимость одного транзистора падает каждые восемь лет.

В настоящее время полевые МОП-транзисторы – это те «кирпичики», из которых собираются почти все цифровые системы. В этом разделе мы выйдем за пределы цифровой абстракции и внимательно рассмотрим, как можно построить логические вентили из полевых МОП-транзисторов.

1.7.1 Полупроводники

МОП-транзисторы изготовляются из кремния – элемента, преобладающего в скальной породе и песке. Кремний (Si) – это элемент IV атомной группы, то есть он имеет четыре валентных электрона, может образовывать связи с четырьмя соседними атомами и, таким образом, формировать кристаллическую *решетку (lattice)*. На **Рис. 1.26 (а)**, для простоты, кристаллическая решетка показана в двумерной системе координат, однако полезно помнить, что реальная кристаллическая решетка имеет форму куба. Линия на **Рис. 1.26 (а)**

изображает ковалентную связь. По своей природе, кремний – плохой проводник, потому что все электроны заняты в ковалентных связях. Однако проводимость кремния улучшается, если добавить в него небольшое количество атомов другого вещества, называемого *примесью (dopant)*. Если в качестве примеси используется элемент V атомной группы, например, мышьяк (As), то в каждом атоме примеси окажется дополнительный электрон, не участвующий в образовании ковалентных связей. Этот свободный электрон может легко перемещаться внутри кристаллической решетки. При этом атом мышьяка, потерявший электрон, превращается в положительный ион (As^+), как показано на **Рис. 1.26 (b)**. Электрон имеет *отрицательный заряд (negative charge)*, поэтому мышьяк принято называть примесью *n-типа (n-type dopant)*. Если же в качестве примеси используется элемент III атомной группы, например, бор (B), то в каждом из атомов примеси будет не хватать одного электрона, как показано на **Рис. 1.26 (c)**. Отсутствующий электрон называют *дыркой (hole)*. Электрон из соседнего атома кремния может перейти к атому бора и заполнить недостающую связь. При этом, атом бора, получивший дополнительный электрон, превращается в отрицательный ион (B^-), а в атоме кремния возникает дырка. Таким образом, дырка может мигрировать в кристаллической решетке подобно электрону. Дырка – это всего лишь отсутствие отрицательного заряда, но она ведет себя в полупроводнике как положительно заряженная частица.

Именно поэтому бор называют примесью *p-типа (p-type dopant)*. А поскольку проводимость кремния может меняться на порядки в зависимости от концентрации примеси, кремний называют *полупроводником (semiconductor)*.

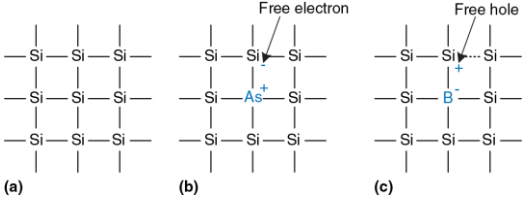


Рис. 1.26 **Кремниевая решетка и атомы примесей**

1.7.2 Диоды

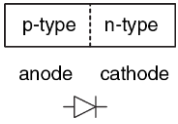


Рис. 1.27 **Структура диода с p-n-соединением и его обозначение**

Диод (diode) – это соединение полупроводника p-типа с полупроводником n-типа, как показано на **Рис. 1.27**. При этом область p-типа называют *анодом (anode)*, а область n-типа называют *катодом (cathode)*. Когда напряжение на аноде превышает напряжение на катоде, *диод открыт (forward biased),* и ток

через него течет от анода к катоду. Если же напряжение на аноде ниже напряжения на катоде, то диод *закрыт (reverse biased),* и ток через диод не течет. Символ диода очень интуитивен и наглядно показывает, что ток через диод может протекать только в одном направлении.

1.7.3 Конденсаторы

Рис. 1.28
Обозначение конденсатора

Конденсатор (capacitor) состоит из двух проводников, отделенных друг от друга изолятором. Если к одному из проводников приложить напряжение V, то через некоторое время этот проводник накопит электрический заряд Q, а другой проводник накопит противоположный электрический заряд −Q. *Емкостью (capacitance)* C конденсатора называется отношение заряда к приложенному напряжению C = Q/V. Емкость прямо пропорциональна размеру проводников и обратно пропорциональна расстоянию между ними. Символ, используемый для обозначения конденсатора, показан на **Рис. 1.28**.

Технические специалисты компании Intel не могут войти в высокочистое помещение, где производятся микросхемы, без защитного комбинезона Gore-Tex, называемом на профессиональном сленге «костюмом кролика» (bunny suit). Наличие такого комбинезона предотвращает от загрязнения кремниевые подложки с микроскопическими транзисторами на них от частиц одежды, кожи или волос. (Воспроизводится с разрешения корпорации Intel©, 2006 год).

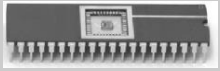
Емкость – это очень важный параметр электрической схемы, поскольку зарядка или разрядка любого проводника требует времени и энергии. Более высокая емкость означает, что электрическая схема будет работать медленнее и потребует для своего функционирования больше энергии. К понятиям скорости и энергии мы будем постоянно возвращаться на протяжении всей этой книги.

1.7.4 n-МОП и p-МОП-транзисторы

Полевой МОП-транзистор представляет собой «сэндвич» из нескольких слоёв проводящих и изолирующих материалов. «Фундамент», с которого начинается построение полевых МОП-транзисторов, – это тонкая круглая кремневая пластина *(wafer)* приблизительно от 15 см до 30 см в диаметре, в русскоязычной литературе называемая подложкой, вафлей или вэйфером. Производственный процесс начинается с пустой подложки. Этот процесс включает заранее определенную последовательность операций, в ходе которой примеси имплантируются в кремний, на подложке выращиваются тонкие пленки кремния и диоксида кремния и наносится слой металла. После каждой операции на подложку в качестве маски наносится определенный *рисунок (pattern)*, чтобы наносимый в ходе следующей операции материал оставался лишь в тех местах, где он необходим. Поскольку размеры одного транзистора – это доли микрона[2], а вся подложка обрабатывается в ходе одного производственного процесса, когда одновременно производятся миллиарды транзисторов, себестоимость одного транзистора существенно снижается. После того, как все операции завершены, подложка нарезается на прямоугольные кристаллы, называемые в англоязычной литературе *chip* или *dice*,

[2] 1 µm = 1 мкм = 10^{-6} м.

причем на каждом из этих прямоугольников размещаются тысячи, миллионы или даже миллиарды транзисторов. Каждый такой кристалл тестируется, а затем помещается в пластиковый или керамический *корпус-упаковку (package)* с металлическими *контактами (pins)* для того, чтобы его можно было установить на монтажной плате.



Корпус с рядом выводов по обеим длинным сторонам (Dual-Inline Package или DIP) с 40 металлическими контактами – по 20 на каждой стороне – содержит внутри небольшой кристалл (на рисунке он практически не виден). Этот кристалл соединяется с ножками контактов золотыми проводами, каждый из которых тоньше человеческого волоса. Фотография Кевина Марра. © Харви колледж.

Сэндвич полевого МОП-транзистора состоит из слоя проводника, называемого затвором *(gate),* наложенного на слой изолятора – диоксида кремния (SiO_2), в свою очередь, наложенного на кремневую пластину, называемую подложкой. Изначально для изготовления затвора использовался тонкий слой металла, отсюда и название этого типа транзисторов – металл-оксид-полупроводник. В современных же технологических процессах в качестве материала затвора используется поликристаллический кремний, поскольку кремний не плавится в ходе

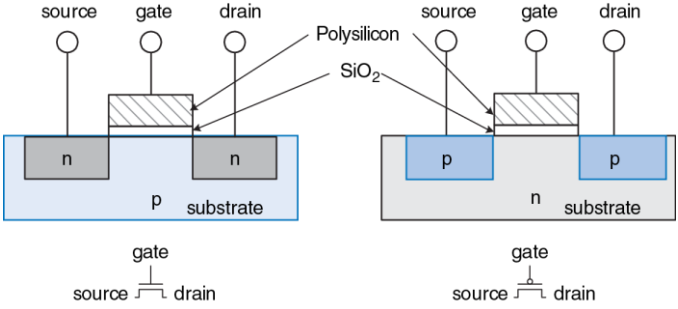
последующей высокотемпературной обработки кристалла. Диоксид кремния – это хорошо известное всем нам стекло, и в полупроводниковой промышленности этот материал часто называют просто оксидом. Слои металл-оксид-полупроводника образуют конденсатор, в котором тонкий слой оксида (или окисла), называемого диэлектриком, изолирует металлическую пластину от полупроводниковой.

> С физической точки зрения исток и сток симметричны. Вместе с тем, мы будем говорить, что электрический заряд перетекает от истока к стоку. В n-МОП-транзисторе электрический заряд переносится электронами, которые двигаются из зоны с отрицательным напряжением в зону с положительным напряжением. В p-МОП-транзисторе заряд переносится дырками, которые двигаются из зоны с положительным напряжением в зону с отрицательным напряжением. Если схематически изобразить транзистор таким образом, чтобы зона максимального положительного напряжения находилась сверху, а зона максимального отрицательного напряжения снизу, то источником (отрицательного) заряда в n-МОП-транзисторе будет нижний вывод, а источником (положительного) заряда в p-МОП-транзисторе будет верхний вывод.

Существуют два вида полевых МОП-транзисторов: n-МОП и p-МОП (по английски n-MOS и p-MOS, что произносится как н-мосс и пи-мосс). На **Рис. 1.29** схематически показано сечение каждого из этих двух типов транзисторов так, как будто мы распилили кристалл и теперь

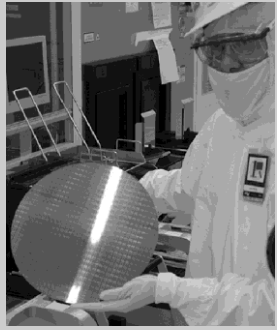
смотрим на транзистор сбоку. В транзисторах n-типа, называемых n-МОП, области, где расположены полупроводниковые примеси n-типа – в свою очередь называемые истоком *(source)* и *стоком (drain)* – находятся рядом с затвором *(gate)*, причем вся эта структура размещается на подложке p-типа. В транзисторах же p-МОП и исток, и сток – это области p-типа, размещенные на подложке n-типа.

Полевой МОП-транзистор ведет себя как переключатель, управляемый приложенным к нему напряжением. В таком транзисторе напряжение перехода создает электрическое поле, включающее или выключающее линию связи между источником и стоком. Термин *полевой транзистор (field effect transistor)* является прямым отражением принципа работы такого устройства. Знакомство с работой полупроводниковых устройств мы начнем с изучения n-МОП-транзистора.



(a) nMOS **(b)** pMOS

Рис. 1.29 **n-МОП и p-МОП-транзисторы**



Технический специалист корпорации Intel держит в руках 12-дюймовый вейфер с несколькими сотнями микропроцессоров на нем. (Воспроизводится с разрешения корпорации Intel©, 2006 год).

Подложка n-МОП транзистора обычно находится под напряжением земли GND, которое является минимальным напряжением в системе. Для начала рассмотрим случай, когда, как показано на **Рис. 1.30 (а)**, напряжение на затворе также равно 0 В. Диоды между истоком или стоком и подложкой находятся в состоянии, называемым обратным смещением *(reverse bias)*, поскольку напряжение на истоке и стоке не является отрицательным. В результате этого канал для движения тока

между истоком и стоком остается закрытым, а транзистор выключенным. Теперь рассмотрим ситуацию, когда напряжение на затворе повышается до V_{DD} – так, как показано на **Рис. 1.30 (b)**. Если приложить положительное напряжение к затвору (верхней пластине конденсатора), то это создает электрическое поле между затвором и подложкой, в результате в зону между истоком и стоком под слоем оксисла формируется избыток электронов. При достаточно высоком напряжении на нижней границе затвора накапливается настолько много электронов, что область с полупроводником p-типа превращается в область с полупроводником n-типа. Такая инвертированная область называется *каналом (channel)*. В этот момент в транзисторе образуется область проводимости от источника n-типа, через каналы n-типа к стоку n-типа, и через этот канал электроны могут беспрепятственно перемещаться от истока к стоку. Транзистор включен. Напряжение перехода, которое требуется для включения транзистора, называется пороговым значением напряжения (*threshold voltage*) V_T и обычно составляет от 0,3 В до 0,7 В.

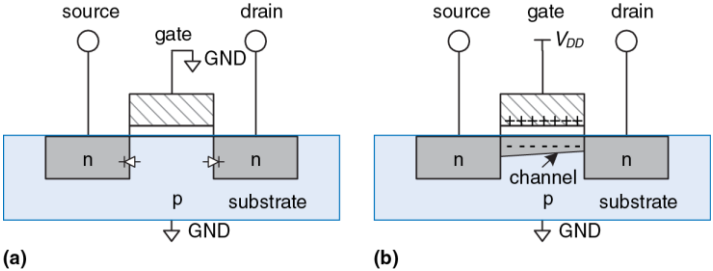
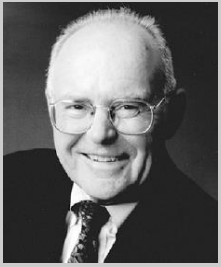


Рис. 1.30 **Работа n-МОП-транзистора**

Транзистор p-МОП работает с точностью до наоборот, как вы, возможно, уже догадались по наличию точки в обозначении этого типа транзистора на **Рис. 1.31**. Подложка p-МОП-транзистора находится под напряжением V_{DD}. Если затвор также находится под напряжением V_{DD}, то p-МОП-транзистор выключен. Если же на затвор подается напряжение земли GND, проводимость канала инвертируется, превращаясь в проводимость p-типа, и транзистор включается.



Гордон Мур, 1929- Гордон Мур родился в Сан-Франциско. Мур получил степень бакалара в области химии в университете штата Калифорния и степень доктора в области химии и физики в Калифорнийском Технологическом Университете (Caltech). В 1968 году Гордон Мур и Роберт Нойс основали корпорацию Intel. В 1965 году Мур заметил, что полупроводниковые технологии развиваются с такой скоростью, что число транзисторов, которое можно разместить на одной микросхеме, удваивается каждый год.

Сегодня эта тенденция известна как закон Мура. Начиная с 1975 года, количество транзисторов на одной микросхеме удваивается каждые два года. Одно из следствий закона Мура гласит, что производительность микропроцессоров удваивается за период от 18 до 24 месяцев. Продажи же полупроводниковых устройств растут по экспоненте. К сожалению, потребление электроэнергии также имеет тенденцию к экспоненциальному росту (воспроизводится с разрешения корпорации Intel ©, 2006 г).

К сожалению, полевые МОП-транзисторы в роли переключателя работают далеко не идеально. В частности, n-МОП-транзисторы хорошо передают 0, но плохо передают 1. Если переход n-МОП-транзистора находится под напряжением V_{DD}, то напряжение на

стоке будет колебаться между 0 и $V_{DD} - V_T$. Аналогичным же образом, p-МОП-транзисторы хорошо передают 1, но плохо передают 0. Однако, как мы увидим в дальнейшем, возможно построить хорошо работающий логический вентиль, используя только те режимы n-МОП- и p-МОП-транзисторов, в которых их работа близка к идеальной.

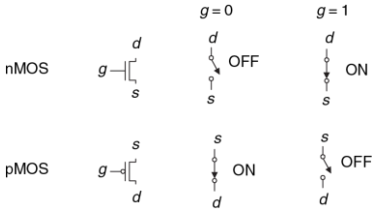


Рис. 1.31 **Модели переключения полевых МОП-транзисторов**

Для изготовления n-МОП-транзистора требуется подложка с проводимостью p-типа, а для изготовления p-МОП-транзисторов необходима подложка n-типа. Для того, чтобы разместить оба типа транзисторов на одном кристалле, производственный процесс, как правило, начинается с подложки p-типа, в который затем имплантируют области для размещения p-МОП-транзисторов n-типа, называемые *колодцами (wells)*. Такой процесс называется *Комплементарным МОП* или *КМОП (Complementary MOS* или *CMOS)*. В настоящее время

КМОП-процесс используется для изготовления подавляющего большинства транзисторов и микросхем.

Подведем итог. КМОП-процесс позволяет разместить МОП-транзисторы n-типа и p-типа, показанные на **Рис. 1.31**, на одном кристалле. Напряжение на затворе (g) управляет током между истоком (s) и стоком (d). Транзисторы n-МОП выключены, когда значение напряжения на переходе соответствует логическому 0, и включены, когда значение напряжения на переходе соответствует логическому 1. Транзисторы p-МОП, напротив, включены, когда значение напряжения на переходе соответствует логическому 0, и выключены, когда значение напряжения на переходе соответствует логическому 1.

1.7.5 Логический вентиль НЕ на КМОП-транзисторах

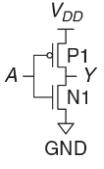


Рис. 1.32 Схема вентиля НЕ

Схема на **Рис. 1.32** демонстрирует, как можно построить логический элемент НЕ, используя КМОП-транзисторы.

На этой схеме треугольник обозначает напряжение земли GND, а горизонтальная линия обозначает напряжение питания V_{DD}.

На всех последующих схемах в этой книге мы не будем использовать буквенные обозначения V_{DD} и GND.

n-МОП-транзистор N1 включен между землей GND и выходным контактом *Y*. В свою очередь, p-МОП-транзистор P1 включен между напряжением питания V_{DD} и выходным контактом *Y*. Напряжение на входном контакте *A* управляет переходами обоих транзисторов.

Если напряжение на *A* равно 0, то транзистор N1 выключен, а транзистор P1 включен. При этом, напряжение на контакте Y равно напряжению питания V_{DD}, а не земли, что соответствует логической единице. В этом случае говорят, что Y «подтянут» к единице (англ.: *pulled up)*. Включенный транзистор P1 хорошо передает логическую единицу (равную напряжению питания), то есть напряжение на контакте *Y* очень близко к V_{DD}. Если же напряжение на контакте A равно логической единице, то транзистор N1 включен, а транзистор P1 выключен, и напряжение на контакте *Y* равно напряжению земли, что соответствует логическому нулю. В этом случае говорят, что *Y* «подтянут» к нулю (англ.: *pulled down)*. Включенный транзистор N1 хорошо передает логический ноль, то есть напряжение на контакте *Y* очень близко к GND. Проверка в таблице истинности на **Рис. 1.12** подтверждает, что мы действительно имеем дело с логическим вентилем НЕ.

1.7.6 Другие логические вентили на КМОП-транзисторах

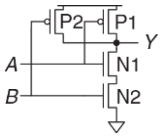


Рис. 1.33 Схема вентиля И-НЕ с двумя входами

На **Рис. 1.33** показана схема для построения с помощью МОП-транзисторов логического элемента И-НЕ с двумя входными контактами. На электронных схемах принято, что если нет никаких дополнительных замечаний или обозначений, то подразумевается, что две линии соединяются друг с другом в том случае, если одна из линий заканчивается в точке пересечения (пересечение в форме буквы Т). Если же обе линии продолжаются за точкой пересечения, то для обозначения контакта этих двух линий в точке пересечения ставится точка. Если точка отсутствует, то это означает, что линии не пересекаются, и одна из линий проходит над другой. На **Рис. 1.33** n-МОП-транзисторы N1 и N2 соединены последовательно. Причем, чтобы замкнуть выходной контакт на землю GND – то есть понизить логический уровень (*pull down*), оба этих транзистора должны быть включены. В то время как p-МОП-транзисторы P1 и P2 соединены параллельно, и только один из них должен быть включен, чтобы соединить выходной контакт с напряжением питания V_{DD} – то есть повысить логический уровень (*pull up*). В **Табл. 1.6** перечислены все возможные состояния для части схемы, понижающей логический уровень (*pull-down network*), для части

схемы, повышающей логический уровень (*pull-up network*), и для выхода. Из **Табл. 1.6** видно, что электрическая схема, показанная на **Рис. 1.33**, действительно работает как логический вентиль И-НЕ. Например, если A равно 1 и B равно 0, то транзистор N1 включен, однако транзистор N2 выключен и блокирует связь контакта *Y* с напряжением земли GND. При этом транзистор P1 выключен, а транзистор P2 включен и соединяет напряжение питания V_{DD} с контактом *Y*. То есть, на контакте *Y* мы имеем 1.

Табл. 1.6 Работа вентиля И-НЕ

A	*B*	Схема понижения логического уровня	Схема повышения логического уровня	*Y*
0	0	ВЫКЛ.	ВКЛ.	1
0	1	ВЫКЛ.	ВКЛ.	1
1	0	ВЫКЛ.	ВКЛ.	1
1	1	ВКЛ.	ВЫКЛ.	0

Рис. 1.34 в обобщенном виде показывает блоки, необходимые для построения любого инвертированного логического вентиля, такого как НЕ, И-НЕ, ИЛИ-НЕ.

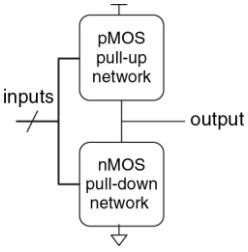


Рис. 1.34 **Общая форма инвертирующего логического вентиля**

Транзисторы n-МОП хорошо передают 0, поэтому схема, понижающая логический уровень (*pull-down network*), составленная из таких транзисторов, помещается между выходным контактом и землей GND для передачи 0 на выход. Транзисторы p-МОП хорошо передают 1, поэтому схема, повышающая логический уровень (*pull-up network*), составленная из таких транзисторов, помещается между выходным контактом и напряжением питания V_{DD} для передачи 1 на выход. Понижающая и повышающая схемы могут состоять из транзисторов, соединенных как параллельно, так и последовательно. Причем при параллельном соединении транзисторов вся схема включена, если включен хотя бы один из транзисторов. При последовательном соединении схема включена, только если оба транзистора включены.

Косая черта на входной линии указывает на то, что этот логический элемент имеет несколько входов.

Опытные разработчики утверждают, что электронные устройства работают, пока они содержат внутри магический дым. Для подтверждения этой теории они ссылаются на наблюдения, в ходе которых было установлено, что если магический дым по каким-то причинам уходит из устройства наружу, то это устройство прекращает функционировать.

Если и понижающую, и повышающую части схемы включить одновременно, то во всей схеме возникнет короткое замыкание между напряжением питания V_{DD} и землей GND. Сигнал на выходном контакте может оказаться в запретной зоне, а транзисторы, потребляющие при этом большое количество энергии, могут перегореть. С другой стороны, если и понижающую и повышающую части схемы одновременно выключить, то выходной сигнал будет отключен и от $V_{DD,}$ и от GND. В этом случае говорят, что выходной сигнал *плавает (floats)*. Его значение, так же как и в случае одновременно включенных схем, не определено. Наличие плавающего сигнала на выходе системы обычно нежелательно, но в **разделе 2.6** мы рассмотрим, как разработчик может использовать такие сигналы.

В правильно функционирующем логическом вентиле в любой момент времени одна из схем должна быть включена, а другая выключена, и напряжение на выходе должно быть или высоким (V_{DD}), или низким (GND). Ни короткое замыкание, ни плавающее значение сигнала не допускается. Чтобы гарантировать это условие, пользуются правилом *дополнения проводимости (conduction complements)*. Если n-МОП-транзисторы в какой-либо цепи соединены последовательно, то p-МОП-транзисторы в этой же цепи должно быть соединены параллельно. Если же n-МОП-транзисторы соединены параллельно, то p-МОП-транзисторы должны соединяться последовательно.

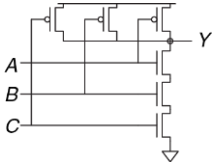


Рис. 1.35 Схема вентиля И-НЕ с тремя входами

Пример 1.20 СХЕМА ВЕНТИЛЯ И-НЕ С ТРЕМЯ ВХОДАМИ

Нарисуйте схему вентиля И-НЕ с тремя входами, используя КМОП-транзисторы.

Решение: Вентиль И-НЕ должен выдать 0 только в том случае, если все входы равны 1. Следовательно, схема, понижающая логический уровень, должна иметь 3 последовательно включенных n-МОП-транзистора. По правилу дополнений (Conduction complements rule) p-МОП-транзисторы должны быть включены параллельно. Такой вентиль показан на **Рис. 1.35**. Вы можете удостовериться в правильности функционирования проверкой таблицы истинности.

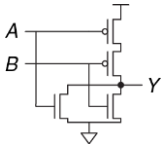


Рис. 1.36 Схема вентиля ИЛИ-НЕ с двумя входами

Пример 1.21 СХЕМА ВЕНТИЛЯ ИЛИ-НЕ С ДВУМЯ ВХОДАМИ

Нарисуйте схему вентиля ИЛИ-НЕ с двумя входами, используя КМОП транзисторы.

Решение: Вентиль ИЛИ-НЕ должен выдавать 0, если хотя бы один из входов равен 1. Следовательно, схема, понижающая логический уровень, должна иметь 2 n-МОП-транзистора, включенных параллельно. По правилу дополнений (Conduction complements rule) p-МОП транзисторы должны быть включены последовательно. Такой вентиль показан на **Рис. 1.36**.

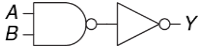


Рис. 1.37 Схема вентиля И с двумя входами

Пример 1.22 СХЕМА ВЕНТИЛЯ И С ДВУМЯ ВХОДАМИ

Нарисуйте схему для вентиля И с двумя входами, используя КМОП транзисторы.

Решение: Схему И невозможно построить на основе одного КМОП-вентиля. Однако, построение И-НЕ и НЕ вентилей – дело довольно простое. Итак, лучший способ построить вентиль И, используя КМОП транзисторы, состоит в том, чтобы использовать И-НЕ, за которым следует НЕ, как показано на **Рис. 1.37**.

1.7.7 Передаточный логический вентиль

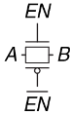


Рис. 1.38
Передаточный вентиль

Иногда разработчику необходим идеальный переключатель, который может одинаково хорошо передавать как 0, так и 1. Вспомним, что n-МОП-транзисторы хорошо передают 0, а p-МОП-транзисторы хорошо передают 1, и параллельное соединение этих двух транзисторов должно хорошо передавать оба этих значения. На **Рис. 1.38** показана такая цепь, называемая передаточным логическим элементом (*transmission gate*), проходным логическим вентилем (*pass gate*) или аналоговым ключом. Выводы этого элемента обозначаются *A* и *B*, поскольку передача сигнала в таком логическом вентиле может идти в двух направлениях, и ни одно из этих направлений не является предпочтительным. Сигналы управления (в англоязычной литературе называемые *enables*), обозначаются *EN* и $\overline{EN}$. Если *EN* равен 0, а $\overline{EN}$ равен 1, то оба транзистора выключены. При этом, весь передаточный логический вентиль выключен, и контакт A не имеет связи с контактом *B*.

Если же *EN* равен 1, а $\overline{EN}$ равен 0, то передаточный логический вентиль включен, и любое логическое значение передается от *A* к *B*.

1.7.8 Псевдо n-МОП-Логика

Построенный по технологии КМОП логический вентиль ИЛИ-НЕ, у которого число входных контактов равно N, использует N параллельно включенных n-МОП-транзисторов и N последовательно включенных p-МОП-транзисторов. Последовательно включенные транзисторы передают сигнал медленнее, чем транзисторы, включенные параллельно, аналогично тому, как сопротивление резисторов, включенных последовательно, будет больше, чем сопротивление резисторов, включенных параллельно. Кроме того, p-МОП-транзисторы передают сигналы медленнее, чем n-МОП-транзисторы, поскольку дырки не могут перемещаться по кристаллической решетке кремния так же быстро, как электроны. В результате, соединенные параллельно n-МОП-транзисторы работают быстро, а соединенные последовательно p-МОП-транзисторы работают медленно, особенно если их много.

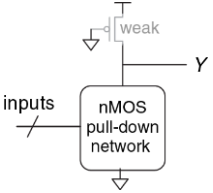


Рис. 1.39 Обобщенный псевдо n-МОП-вентиль

Как показано на **Рис. 1.39**, при использовании *псевдо n-МОП-логики (pseudo-nMOS logic)*, или просто *псевдо-логики*, медленный стек из p-МОП-транзисторов заменяют одним «слабым» p-МОП-транзистором, который всегда находится во включенном состоянии. Такой транзистор часто называют *слабым подтягивающим транзистором (weak pull-up)*. Физические параметры p-МОП-транзистора подбираются таким образом, что этот транзистор до высокого логического уровня (1) выход Y «подтягивает слабо» – то есть только в том случае, когда все n-МОП-транзисторы выключены. Но если при этом хотя бы один из n-МОП-транзисторов включается, то он, превосходя по мощности слабый подтягивающий транзистор, «перетягивает» выход Y настолько близко к напряжению земли GND, что на выходе получается логический 0.

Преимущество псевдо-логики заключается в том, что такую логику можно использовать для создания быстрых ИЛИ-НЕ вентилей с большим количеством входов. Например, на **Рис. 1.40** показан вентиль ИЛИ-НЕ с четырьмя входами, построенный с использованием псевдо-логики.

Логические вентили, использующие псевдо-логику, могут быть очень полезны для построения некоторых видов памяти и логических массивов, описанных в Главе 5. Недостаток псевдо-логики – наличие короткого замыкания между питанием V_{DD} и землей GND, когда сигнал

на выходе – это логический нуль (0). Слабые p-МОП- и n-МОП-транзисторы выключены. При этом, через короткое замыкание постоянно протекает ток, и электрическая энергия от источника питания расходуется впустую. Именно по этой причине псевдо-n-МОП-логика используется ограниченно.

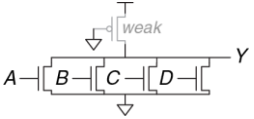


Рис. 1.40 **Псевдо n-МОП-вентиль ИЛИ-НЕ с четырьмя входами**

Термин «псевдо-n-МОП-логика» родился в 70-тые годы прошлого века. Тогда существовал производственный процесс для изготовления только n-МОП-транзисторов. В то время слабые n-МОП-транзисторы использовались для «подтягивания» выходного сигнала до логической единицы (1), поскольку p-МОП-транзисторов просто не было.

1.8 ПОТРЕБЛЯЕМАЯ МОЩНОСТЬ

Потребляемая мощность – это количество энергии, потребляемой системой в единицу времени. Энергопотребление имеет большое значение в цифровых системах. Именно потребляемая мощность определяет время автономной работы без подзарядки батареи любого портативного устройства, такого как сотовый телефон или ноутбук. Не стоит думать, однако, что потребляемая мощность – второстепенный параметр для стационарных устройств. Электричество стоит денег, и к тому же любое устройство может перегреться, если оно потребляет слишком много электроэнергии.

Цифровая система потребляет энергию как в динамическом режиме, когда выполняет какие-либо операции, так и в статическом, когда система находится в состоянии покоя *(idle)*. В динамическом режиме энергия расходуется на зарядку емкостей элементов системы, когда эти элементы переключаются между 0 и 1. И хотя в статическом режиме никаких переключений не происходит, система все равно расходует электрическую энергию.

И сами логические вентили, и проводники, соединяющие эти вентили друг с другом, являются конденсаторами и обладают определенной емкостью. Энергия, получаемая от блока питания, которую необходимо затратить на зарядку емкости C до напряжения $V_{DD,}$ равна $C \times V_{DD}^2$. Если

напряжение на конденсаторе переключается с частотой *f* (т.е. *f* раз в секунду), то конденсатор заряжается *f*/2 раза и разряжается *f*/2 раза в секунду. И, поскольку в процессе разрядки конденсатор не потребляет энергию от источника питания, то получается, что потребление энергии в динамическом режиме можно рассчитать как:

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^{2} f \tag{1.4}$$

Утечка тока в системе происходит, даже если система находится в состоянии покоя. У некоторых типов электронных схем, таких как псевдо n-МОП-логика, рассмотренная в **разделе 1.7.8**, существует путь, соединяющий напряжение питания V_{DD} с землей GND, через который ток протекает постоянно. Суммарная величина тока, протекающего в системе в ее статическом состоянии $I_{DD,}$ называется *током утечки (leakage current)* или *током покоя (quiescent supply current)*. Мощность, потребляемая системой в статическом состоянии, пропорциональна величине тока утечки и может быть рассчитана как:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1.5}$$

Пример 1.23 ПОТРЕБЛЯЕМАЯ МОЩНОСТЬ

Сотовый телефон некоторой модели имеет аккумулятор емкостью 6 Ватт-часов и работает от напряжения 1,2 вольта. Предположим, что во время использования телефон работает на частоте 300 МГц и средняя емкость цифровой схемы телефона в любой конкретный момент составляет 10 нФ (10^{-8} Фарады). При работе телефон также выдает сигнал мощностью 3 Вт на антенну. Когда телефон не используется, динамическая потребляемая мощность падает практически до нуля, так как обработка сигналов отключена. Но телефон также потребляет 40 мА тока покоя в любом случае, работает он или нет. Рассчитайте время, на которое хватит аккумулятора телефона, для случаев:

(а) Если телефон не используется

(б) Если телефон используется непрерывно

Решение: Статическая мощность P_{static} равна (0,040 А)*(1,2 В) = 48 мВт. Если телефон не используется, это единственное потребление мощности, поэтому время жизни аккумулятора равно (6 Ватт-часов)/(0,048 Вт) = 125 часов (примерно 5 дней). В случае, если телефон используется, динамическая мощность $P_{dynamic}$ равна (0,5)*(10^{-8} Ф)*(1,2 В)^2 *(3 * 10^8 Гц) = 2,16 Вт. Общая мощность, являющаяся суммой $P_{dynamic}$, P_{static} и мощности вещания, составит 2,16 Вт + 0,048 Вт + 3 Вт = 5,2 Вт, поэтому время жизни аккумулятора будет равно 6 Ватт-часов/5,2 Вт = 1,15 часа. В этом примере фактическая работа телефона представлена в несколько упрощенном виде, но тем не менее, он иллюстрирует ключевые идеи, касающиеся мощности потребления.

1.9 КРАТКИЙ ОБЗОР ГЛАВЫ 1 И ТОГО, ЧТО НАС ЖДЕТ ВПЕРЕДИ

В этом мире существует 10 видов людей: те, кто знакомы с двоичной системой счисления, и те, кто не знают о ней ничего.

В этой главе мы осветили основные принципы, необходимые для понимания и проектирования сложных электронных систем. И хотя физические величины в реальном мире в большинстве своем аналоговые – то есть изменяются непрерывно, разработчики цифровых систем ограничиваются рассмотрением конечного подмножества дискретных величин непрерывно меняющихся сигналов. В частности, бинарные переменные могут принимать только два значения – 0 и 1, которые еще называются ЛОЖЬ (FALSE) и ИСТИНА (TRUE) или НИЗКОЕ (LOW) и ВЫСОКОЕ (HIGH). Логические вентили определенным образом преобразуют сигналы с одного или нескольких двоичных входов в двоичный сигнал на выходе. Некоторые из наиболее часто используемых логических вентилей перечислены ниже:

- **НЕ**: Имеет на выходе значение ИСТИНА, если сигнал на входе имеет значение ЛОЖЬ.
- **И:** Имеет на выходе значение ИСТИНА, если все сигналы на входе имеют значение ИСТИНА.

- **ИЛИ:** Имеет на выходе значение ИСТИНА, если хотя бы один сигнал на входе имеет значение ИСТИНА.
- **Исключающее ИЛИ:** Имеет на выходе значение ИСТИНА, если нечетное количество сигналов на входе имеет значение ИСТИНА.

Для построения логических вентилей обычно используются транзисторы КМОП, которые, по сути, являются переключателями с электрическим управлением. Транзистор n-МОП включается, если затвор находится под напряжением V_{DD}, что соответствует логической единице. Транзистор p-МОП включается, если затвор находится под напряжением GND, что соответствует логическому нулю.

В **главах 2–5** мы продолжим изучение цифровой логики. В **главе 2** рассматривается *комбинаторная логика (combinational logic*), в которой предполагается, что сигнал на выходе логического вентиля зависит только от состояний на входных контактах этого элемента в данный конкретный момент времени. Те логические вентили, которые мы уже рассмотрели в этой книге, могут служить в качестве примера использования комбинаторной логики. Из **главы 2** вы также поймете, как можно спроектировать схему из нескольких логических вентилей таким образом, чтобы все возможные состояния этой схемы

соответствовали состояниям, заранее описанным в таблице истинности или с помощью уравнения Булевой логики.

Главе 3 описывает *последовательную логику (sequential logic)*. Такая логика уже допускает, что результат на выходе логического вентиля зависит как от текущего состояния на входе, так и от прошлых его состояний. *Регистр (Register)* – это наиболее распространенный элемент последовательной логики, который «запоминает» предыдущее состояние на своем входе. *Конечный автомат (finite state machines)*, построенный на базе регистров и комбинаторной логики, является мощным средством для создания сложных систем на системной основе. В **главе 3** мы также рассмотрим временные соотношения сигналов в цифровой системе, чтобы определить максимально возможную скорость, на которой эта система может нормально работать.

Глава 4 рассматривает *языки описания оборудования (hardware description languages*, или *HDL)*. Языки HDL – родственники обычных языков программирования, но используются они, по большей части, для моделирования и создания аппаратного, а не программного обеспечения. Большинство современных цифровых систем было разработано с использованем HDL. SystemVerilog и VHDL – два наиболее распространенных языка для описания и верификации аппаратуры, и оба они рассматриваются в этой книге. *VHDL (Very high-speed integrated circuits Hardware Description Language)* переводится как

язык для описания и верификации аппаратуры на очень высокоскоростных интегральных схемах.

Глава 5 описывает другие элементы комбинаторной и последовательной логик, такие как *сумматоры (adders)*, *умножители (multipliers)* и *блоки памяти (memories)*.

Глава 6 переходит к компьютерной архитектуре. Она описывает MIPS-процессор – стандартный микропроцессор, использующийся в бытовой электронике, в ряде рабочих станций производства компании Silicon Graphics и в разнообразных системах связи, таких как телевидение, оборудование для локальных сетей и беспроводного интернета. Архитектура MIPS-процессора определяется структурой его регистров, языком ассемблера и набором инструкций. Вы узнаете, как писать программы для MIPS-процессора на языке ассемблера, то есть общаться с этим процессором на его родном языке.

Глава 7 и **глава 8** перекидывают мостик от цифровой логики к компьютерной архитектуре. **Глава 7** исследует микроархитектуру – то есть организацию отдельных строительных блоков, таких как сумматоры и регистры, необходимую для построения работающего процессора. Эта глава научит вас навыкам, необходимым для разработки вашего собственного MIPS-процессора. Более того, в **главе 7** мы рассмотрим три микроархитектуры, иллюстрирующие

различные компромиссы между производительностью процессора и затратами на его производство. Производительность процессоров росла по экспоненте, требуя все более изощренных блоков памяти, чтобы удовлетворить постоянно растущий спрос на данные. **Глава 8** погрузит нас в глубины архитектуры блоков памяти, а также позволить понять, как компьютеры связываются с периферийными устройствами, такими как клавиатура или принтер.

УПРАЖНЕНИЯ

Упражнение 1.1 Объясните не менее трех уровней абстракции, которые используются:

a) Биологами, изучающими работу клеток.

b) Химиками, изучающими состав какого-либо материала.

Ваше объяснение не должно быть длиннее одного абзаца.

Упражнение 1.2 Объясните, как методы иерархичности, модульности и регулярности могут быть использованы:

a) Конструкторами автомобилей.

b) Каким-либо бизнесом для управления ежедневными операциями.

Ваше объяснение не должно быть длиннее одного абзаца.

Упражнение 1.3 Бен Битдидл строит дом (прим. переводчика: Бен Битдидл (Ben Bitdiddle) – персонаж, созданный Стивом Уордом (Steve Ward) в 1970-х годах и с той поры широко используемый в качестве героя сборников задач в Массачусетском технологическом институте (Massachusetts Institute of Technology, MIT) и вне его. Фамилия Бена происходит от термина "bit diddling",

который можно перевести как "битовое жонглирование" – программирование на уровне машинных кодов с манипулированием битами, флагами, полубайтами и другими элементами размером меньше символа). Объясните ему, как он может использовать принципы иерархичности, модульности и регулярности, чтобы сэкономить время и ресурсы.

Упражнение 1.4 Допустим, что напряжение аналогового сигнала в нашей системе меняется в пределах от 0 вольт до 5 вольт. Если мы можем измерить это напряжение с точностью до ±50 милливольт, какое максимальное количество информации в битах этот сигнал может передавать?

Упражнение 1.5 На стене висят старые часы с отломанной минутной стрелкой.

a) Если, используя только часовую стрелку, вы можете определить текущее время с точностью до 15 минут, то сколько битов информации о времени вы можете получить, глядя на эти часы?

b) Если вы будете знать, какая сейчас половина дня – до или после полудня, то сколько дополнительных битов информации о текущем времени вы получите?

Упражнение 1.6 Примерно 4000 лет назад вавилоняне разработали шестидесятеричную (по основанию 60) систему счисления. Сколько битов информации передает одна шестидесятеричная цифра? Как можно записать число 4000_{10}, используя шестидесятеричную систему счисления?

Упражнение 1.7 Как много различных чисел может быть представлено 16 битами?

Упражнение 1.8 Какое максимальное значение может быть представлено 32-разрядным двоичным числом?

Упражнение 1.9 Какое максимальное 16-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже?

a) Двоичное число без знака (unsigned)

b) Дополнительный код (two's complement)

c) Прямой код (sign/magnitude)

Упражнение 1.10 Какое максимальное 32-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже?

a) Двоичное число без знака (unsigned)

b) Дополнительный код (two's complement)

c) Прямой код (sign/magnitude)

Упражнение 1.11 Какое минимальное (наименьшее отрицательное) 16-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже?

a) Двоичное число без знака (unsigned)

b) Дополнительный код (two's complement)

c) Прямой код (sign/magnitude)

Упражнение 1.12 Какое минимальное (наименьшее отрицательное) 32-разрядное двоичное число вы можете представить, используя системы представления двоичных чисел, перечисленные ниже?

a) Двоичное число без знака (unsigned)

b) Дополнительный код (two's complement)

c) Прямой код (sign/magnitude)

Упражнение 1.13 Преобразуйте следующие двоичные числа без знака в десятичные.

a) 1010_2

b) 110110_2

c) 11110000_2

d) 0001000101001112

Упражнение 1.14 Преобразуйте следующие двоичные числа без знака в десятичные.

a) 1110_2

b) 100100_2

c) 11010111_2

d) 011101010100100_2

Упражнение 1.15 Преобразуйте двоичные числа без знака из **упражнения 1.13** в шестнадцатеричные.

Упражнение 1.16 Преобразуйте двоичные числа без знака из **упражнения 1.14** в шестнадцатеричные.

Упражнение 1.17 Преобразуйте следующие шестнадцатеричные числа в десятичные.

a) $A5_{16}$

b) $3B_{16}$

c) $FFFF_{16}$

d) $D0000000_{16}$

Упражнение 1.18 Преобразуйте следующие шестнадцатеричные числа в десятичные.

a) $4E_{16}$

b) $7C_{16}$

c) $ED3A_{16}$

d) $403FB001_{16}$

Упражнение 1.19 Преобразуйте шестнадцатеричные числа из **упражнения 1.17** в двоичные числа без знака.

Упражнение 1.20 Преобразуйте шестнадцатеричные числа из **упражнения 1.18** в двоичные числа без знака.

Упражнение 1.21 Преобразуйте следующие двоичные числа, представленные в дополнительном коде, в десятичные.

a) 1010_2

b) 110110_2

c) 01110000_2

d) 10011111_2

Упражнение 1.22 Преобразуйте следующие двоичные числа, представленные в дополнительном коде, в десятичные.

a) 1110_2

b) 100011_2

c) 01001110_2

d) 10110101_2

Упражнение 1.23 Преобразуйте двоичные числа из **упражнения 1.21** в десятичные, считая, что эти двоичные числа представлены не в дополнительном, а в прямой коде.

Упражнение 1.24 Преобразуйте двоичные числа из **упражнения 1.22** в десятичные, считая, что эти двоичные числа представлены не в дополнительном, а в прямой коде.

Упражнение 1.25 Преобразуйте следующие десятичные числа в двоичные числа без знака.

a) 42_{10}

b) 63_{10}

c) 229_{10}

d) 845_{10}

Упражнение 1.26 Преобразуйте следующие десятичные числа в двоичные числа без знака.

a) 14_{10}

b) 52_{10}

c) 339_{10}

d) 711_{10}

Упражнение 1.27 Преобразуйте десятичные числа из **упражнения 1.25** в шестнадцатеричные.

Упражнение 1.28 Преобразуйте десятичные числа из **упражнения 1.26** в шестнадцатеричные.

Упражнение 1.29 Преобразуйте следующие десятичные числа в 8-битные двоичные числа, представленные в дополнительном коде. Укажите, имеет ли место переполнение.

a) 42_{10}

b) -63_{10}

c) 124_{10}

d) -128_{10}

e) 133_{10}

Упражнение 1.30 Преобразуйте следующие десятичные числа в 8-битные двоичные числа, представленные в дополнительном коде. Укажите, имеет ли место переполнение.

a) 24_{10}

b) -59_{10}

c) 128_{10}

d) -150_{10}

e) 127_{10}

Упражнение 1.31 Преобразуйте десятичные числа из **упражнения 1.29** в 8-битные двоичные числа, представленные в прямом коде.

Упражнение 1.32 Преобразуйте десятичные числа из **упражнения 1.30** в 8-битные двоичные числа, представленные в прямом коде.

Упражнение 1.33 Преобразуйте следующие 4-разрядные двоичные числа, представленные в дополнительном коде, в 8-разрядные двоичные числа, представленные в дополнительном коде:

a) 0101_2

b) 1010_2

Упражнение 1.34 Преобразуйте следующие 4-разрядные двоичные числа, представленные в дополнительном коде, в 8-разрядные двоичные числа, представленные в дополнительном коде:

a) 0111_2

b) 1001_2

Упражнение 1.35 Преобразуйте 4-разрядные двоичные числа из **упражнения 1.33** в 8-разрядные, считая, что это двоичные числа без знака.

Упражнение 1.36 Преобразуйте 4-разрядные двоичные числа из **упражнения 1.34** в 8-разрядные, считая, что это двоичные числа без знака.

Упражнение 1.37 Система счисления по основанию 8 называется *восьмеричной (octal)*. Представьте каждое из чисел в **упражнении 1.25** в восьмеричном виде.

Упражнение 1.38 Система счисления по основанию 8 называется *восьмеричной (octal)*. Представьте каждое из чисел в **упражнении 1.26** в восьмеричном виде.

Упражнение 1.39 Преобразуйте каждое из следующих восьмеричных чисел в двоичное, шестнадцатеричное и десятичное:

a) 42_8

b) 63_8

c) 255_8

d) 3047_8

Упражнение 1.40 Преобразуйте каждое из следующих восьмеричных чисел в двоичное, шестнадцатеричное и десятичное:

a) 23_8

b) 45_8

c) 371_8

d) 2560_8

Упражнение 1.41 Сколько 5-разрядных двоичных чисел, представленных в дополнительном коде, имеют значение большее, чем 0? Сколько – меньшее, чем 0? Каким будет правильный ответ в случае 5-разрядных двоичных чисел, представленных в прямом коде?

Упражнение 1.42 Сколько 7-разрядных двоичных чисел, представленных в дополнительном коде, имеют значение большее, чем 0? Сколько меньшее, чем 0? Каким будет правильный ответ в случае 7-разрядных двоичных чисел, представленных в прямом коде?

Упражнение 1.43 Сколько байтов в 32-битном слове? Сколько полубайтов?

Упражнение 1.44 Сколько байтов в 64-битном слове?

Упражнение 1.45 Если DSL-модем работает со скоростью 768 кбит/сек, сколько байт он может передать за 1 минуту?

Упражнение 1.46 USB3.0 передает данные со скоростью 5 Гбит/сек. Сколько байт USB3.0 может передать за 1 минуту?

Упражнение 1.47 Производители жестких дисков измеряют объемы данных в мегабайтах, что означает 10^6 байт, и гигабайтах, что означает 10^9 байт. Сколько гигабайтов музыки вы можете сохранить на 50-гигабайтном жестком диске?

Упражнение 1.48 Без использования калькулятора рассчитайте приблизительное значение 2^{31}.

Упражнение 1.49 Память процессора Pentium II организована как прямоугольный массив битов, состоящий из 2^8 строк и 2^9 колонок. Без использования калькулятора рассчитайте приблизительное количество битов в этом массиве.

Упражнение 1.50 Нарисуйте цифровую шкалу, аналогичную изображенной на **Рис. 1.11** для 3-битного двоичного числа, представленного в дополнительном коде и прямом коде.

Упражнение 1.51 Нарисуйте цифровую шкалу, аналогичную изображенной на **Рис. 1.11** для 2-битного двоичного числа, представленного в дополнительном коде и прямом коде.

Упражнение 1.52 Сложите следующие двоичные числа без знака:

a) $1001_2 + 0100_2$

b) $1101_2 + 1011_2$

Укажите, если сумма переполняет 4-битный регистр.

Упражнение 1.53 Сложите следующие двоичные числа без знака:

a) $10011001_2 + 01000100_2$

b) $11010010_2 + 10110110_2$

Укажите, если сумма переполняет 8-битный регистр.

Упражнение 1.54 Выполните **упражнение 1.52**, считая, что двоичные числа в этом упражнении представлены в дополнительном коде.

Упражнение 1.55 Выполните **упражнение 1.53**, считая, что двоичные числа в этом упражнении представлены в дополнительном коде.

Упражнение 1.56 Преобразуйте следующие десятичные числа в 6-битные двоичные числа, представленные в дополнительном коде, и сложите их:

a) $16_{10} + 9_{10}$

b) $27_{10} + 31_{10}$

c) $-4_{10} + 19_{10}$

d) $3_{10} + -32_{10}$

e) $-16_{10} + -9_{10}$

f) $-27_{10} + -31_{10}$

Укажите, если сумма переполняет 6-битный регистр.

Упражнение 1.57 Преобразуйте следующие десятичные числа в 6-битные двоичные числа, представленные в дополнительном коде, и сложите их:

a) $7_{10} + 13_{10}$

b) $17_{10} + 25_{10}$

c) $-26_{10} + 8_{10}$

d) $31_{10} + -14_{10}$

e) $-19_{10} + -22_{10}$

f) $-2_{10} + -29_{10}$

Укажите, если сумма переполняет 6-битный регистр.

Упражнение 1.58 Сложите следующие шестнадцатеричные числа без знака:

a) $7_{16} + 9_{16}$

b) $13_{16} + 28_{16}$

a) $AB_{16} + 3E_{16}$

b) $8F_{16} + AD_{16}$

Укажите, если сумма переполняет 8-битный регистр (два шестнадцатеричных числа).

Упражнение 1.59 Сложите следующие шестнадцатеричные числа без знака:

a) $22_{16} + 8_{16}$

b) $73_{16} + 2C_{16}$

c) $7F_{16} + 7F_{16}$

d) $C2_{16} + A4_{16}$

Укажите если сумма переполняет 8-битный регистр (два шестнадцатеричных числа).

Упражнение 1.60 Преобразуйте следующие десятичные числа в 5-разрядные двоичные числа, представленные в дополнительном коде, и вычтите одно число из другого:

a) $9_{10} - 7_{10}$

b) $12_{10} - 15_{10}$

c) $-6_{10} - 11_{10}$

d) $4_{10} - -8_{10}$

Укажите, если разность переполняет 5-битный регистр.

Упражнение 1.61 Преобразуйте следующие десятичные числа в 6-разрядные двоичные числа, представленные в дополнительном коде, и вычтите одно число из другого:

a) $18_{10} - 12_{10}$

b) $30_{10} - 9_{10}$

c) $-28_{10} - 3_{10}$

d) $-16_{10} - 21_{10}$

Укажите, если разность переполняет 6-битный регистр.

Упражнение 1.62 В *N*-битной двоичной системе счисления со *смещением B* (*N*-bit binary number system with *bias B*) положительные и отрицательные числа представляются как значения этих чисел в обычной двоичной системе плюс смещение B. Например, для 5-битной двоичной системы счисления со смещением 15 число 0 представляется как 01111, а число 1 представляется как 10000 и так далее. Системы счисления со смещением иногда используется для выполнения математических операций с плавающей запятой, которые будут рассмотрены в **главе 5**. Ответьте на следующие вопросы применительно к 8-битной системе счисления со смещением 127_{10}:

a) Какое десятичное значение соответствует двоичному числу 10000010_2?

b) Какое двоичное число соответствует значению 0?

c) Как в такой системе будет выглядеть минимальное отрицательное двоичное число, и каким будет его десятичный эквивалент?

d) Как в такой системе будет выглядеть максимальное положительное двоичное число, и каким будет его десятичный эквивалент?

Упражнение 1.63 Нарисуйте цифровую шкалу, аналогичную изображенной на **Рис. 1.11** для 3-битного двоичного числа со смещением равным 3. Что такое система счисления со смещением, объясняется в **упражнении 1.62**.

Упражнение 1.64 В *двоично-кодированной десятичной системе счисления* (*binary-coded decimal* system или *BCD*) 4 бита используются для представления десятичных чисел от 0 до 9. Например, 37_{10} записывается как 00110111_{BCD}.

Ответьте на следующие вопросы применительно к двоично-кодированной десятичной системе счисления:

a) Как будет выглядеть 289_{10} в двоично-кодированной десятичной системе счисления?

b) Как выглядит десятичный эквивалент 100101010001_{BCD}?

c) Как выглядит двоичный эквивалент 01101001_{BCD}?

d) Какие, по-вашему мнению, преимущества имеет двоично-кодированная десятичная система счисления?

Упражнение 1.65 Ответьте на следующие вопросы применительно к двоично-кодированной десятичной системе счисления:

a) Как будет выглядеть 371_{10} в двоично-кодированной десятичной системе счисления?

b) Как выглядит десятичный эквивалент 000110000111_{BCD}?

c) Как выглядит двоичный эквивалент 10010101_{BCD}?

d) Какие, на ваш взгляд, недостатки имеет двоично-кодированная десятичная система счисления по сравнению с двоичной?

Что такое двоично-кодированная десятичная система счисления со смещением, объясняется в **упражнении 1.64**.

Упражнение 1.66 Марсианская летающая тарелка потерпела крушение на кукурузном поле в штате Небраска. Следователи ФБР обнаружили среди обломков руководство по космической навигации с формулами, записанными в марсианской системе счисления. Одна из формул выглядит следующим образом: 325 + 42 = 411. Если эта формула записана без ошибок, сколько пальцев на руке марсианина вы бы ожидали увидеть?

Упражнение 1.67 У Бена Битдидла и Алисы П. Хакер возник спор (прим. переводчика: в англоязычном варианте имя Alyssa P. Hacker созвучно выражению "a LISP hacker", т.е. LISP-хакер (LISP – семейство функциональных языков программирования). Бен утверждает, что у всех целых чисел, которые

больше нуля и кратны шести, есть точно две единицы в двоичном представлении. Алиса не согласна. По ее мнению, все такие числа имеют четное количество единиц в их представлении. Вы согласны с Беном, с Алисой, с обоими или ни с кем из них? Объясните.

Упражнение 1.68 Бен Битдидл и Алиса П. Хакер еще раз спорят. Бен говорит: “я могу получить двоичное дополнение числа путем вычитания 1, а затем инвертируя все биты результата.” Алиса отвечает: “Нет, я могу это сделать путем проверки каждого бита, начиная с наименее значимых. Когда встречу первую 1, инвертирую каждый последующий бит.” Вы согласны с Беном, или с Алисой, или с обоими, или ни с кем? Объясните.

Упражнение 1.69 Напишите программу на вашем любимом языке (например, C, Java, Perl) для преобразования двоичных чисел в десятичные. Пользователь должен ввести без-знаковое двоичное число. Программа должна распечатать его десятичный эквивалент.

Упражнение 1.70 Повторите упражнение 1.69, но для преобразования чисел в системе счисления с произвольной базой b1 в числа в системе счисления с другой базой b2. Поддержите все базы до 16, для цифр больше 9 используйте буквы алфавита. Пользователь должен ввести b1, b2, а затем число в системе счисления с базой b1. Программа должна напечатать эквивалентное число в системе счисления с базой b2.

Упражнение 1.71 Нарисуйте обозначение, логическое уравнение и таблицу истинности для:

a) Вентиля ИЛИ с тремя входами

b) Вентиля исключающее ИЛИ с тремя входами

c) Вентиля исключающее ИЛИ-НЕ с четырьмя входами

Упражнение 1.72 Нарисуйте обозначение, логическое уравнение и таблицу истинности для:

a) Вентиля ИЛИ с четырьмя входами

b) Вентиля исключающее ИЛИ-НЕ с тремя входами

c) Вентиля И-НЕ с пятью входами

Упражнение 1.73 *Мажоритарный вентиль* выдает значение ИСТИНА тогда и только тогда, когда более половины его входов имеют значение ИСТИНА. Заполните таблицу истинности для мажоритарного вентиля, показанного на **Рис. 1.41**.

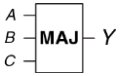


Рис. 1.41 **Мажоритарный вентиль с тремя входами**

Упражнение 1.74 Вентиль И-ИЛИ (AO) с тремя входами, показанный на **Рис. 1.42**, выдает значение ИСТИНА, если входы *A* и *B* имеют значение ИСТИНА или вход *C* имеет значение ИСТИНА. Заполните таблицу истинности для этого вентиля.

Рис. 1.42 **Вентиль И-ИЛИ с тремя входами**

Упражнение 1.75 Вентиль инвертированный ИЛИ-И (OAI) с тремя входами, показанный на **Рис. 1.43**, выдает значение ЛОЖЬ, если вход *C* имеет значение ИСТИНА и входы *A* или *B* имеют значение ИСТИНА. Иначе вентиль выдает значение ИСТИНА. Заполните таблицу истинности для этого вентиля.

Рис. 1.43 **Инвертированный вентиль И-ИЛИ с тремя входами**

Упражнение 1.76 Имеется 16 разных таблиц истинности для булевых функций от двух переменных. Исследуйте эти таблицы, давая каждой одно короткое описательное имя (например, ИЛИ, И-НЕ и так далее).

Упражнение 1.77 Сколько разных таблиц истинности для булевых функций от N переменных?

Упражнение 1.78 Можно ли назначить логические уровни так, чтобы устройство с передаточными характеристиками, показанными на **Рис. 1.44**, могло служить в качестве инвертора? Если да, то какими являются входные и выходные низкие и высокие уровни (V_{IL}, V_{OL}, V_{IH}, и V_{OH}) и уровни шума (N_{ML} и N_{MH})? Если это не так, то объясните почему.

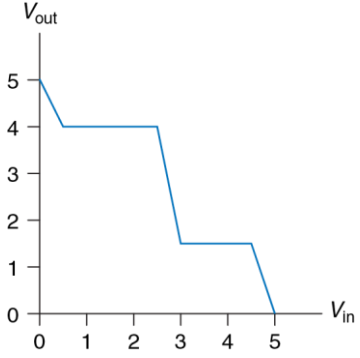


Рис. 1.44 **Передаточные характеристики**

Упражнение 1.79 Повторите **упражнение 1.78** для передаточных характеристик, показанных на **Рис. 1.45**.

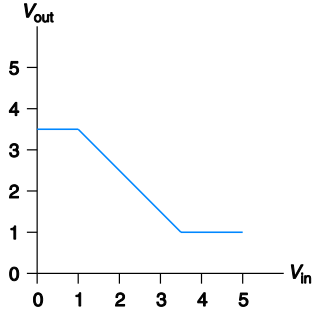


Рис. 1.45 **Передаточные характеристики**

Упражнение 1.80 Можно ли назначить логические уровни так, чтобы устройство с передаточными характеристиками, показанными на **Рис. 1.46**, могло служить в качестве буфера? Если да, то какими являются входные и выходные низкие и высокие уровни (V_{IL}, V_{OL}, V_{IH}, и V_{OH}) и уровни шума (N_{ML} и N_{MH})? Если это не так, то объясните почему.

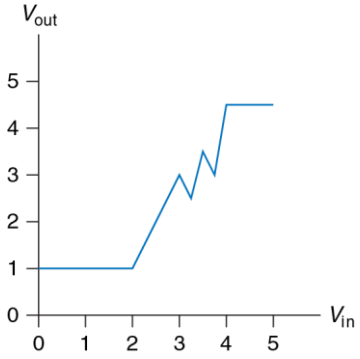


Рис. 1.46 **Передаточные характеристики**

Упражнение 1.81 Бен Битдидл придумал схему с передаточными характеристиками, показанными на **Рис. 1.47**, чтобы использовать ее в качестве буфера. Будет ли эта схема работать? Почему или почему нет? Он утверждает, что она совместима с низковольтными КМОП- и НТТЛ-структурами. Может ли буфер Бена корректно получать входные сигналы от этих логических структур? Может ли ее выход управлять этими логическими структурами? Объясните.

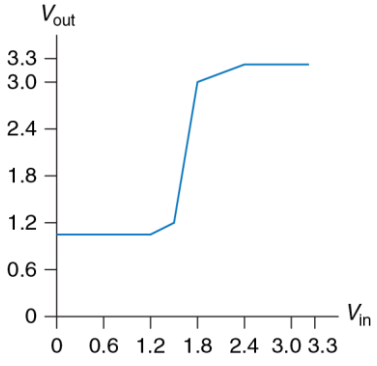


Рис. 1.47 **Передаточные характеристики буфера Бена**

Упражнение 1.82 Во время прогулки по темной аллее Бен Битдидл увидел вентиль с двумя входами и передаточной функцией, показанной на **Рис. 1.48**. Входы обозначены как *A* и *B*, а выходной сигнал – *Y*.

a) Какого типа логический вентиль он увидел?

b) Каковы приблизительные значения высокого и низкого логических уровней?

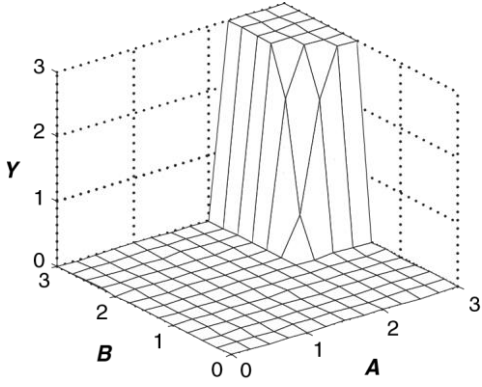


Рис. 1.48 **Передаточные характеристики с двумя входами**

Упражнение 1.83 Повторите **упражнение 1.82** для **Рис. 1.49**.

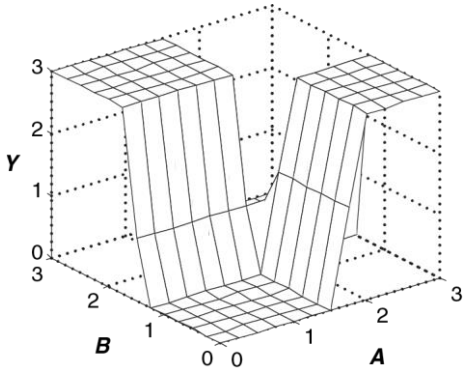


Рис. 1.49 **Передаточные характеристики с двумя входами**

Упражнение 1.84 Сделайте набросок схемы на уровне транзисторов для следующих КМОП-вентилей. Используйте минимальное количество транзисторов.

a) Вентиль И-НЕ с четырьмя входами

b) Вентиль инвертированный ИЛИ-И с тремя входами (см. **упражнение 1.75**)

c) Вентиль И-ИЛИ с тремя входами (см. **упражнение 1.74**)

Упражнение 1.85 Сделайте набросок схемы на уровне транзисторов для следующих КМОП-вентилей. Используйте минимальное количество транзисторов.

a) Вентиль ИЛИ-НЕ с тремя входами

b) Вентиль И с тремя входами

c) Вентиль ИЛИ с двумя входами

Упражнение 1.86 Вентиль меньшинства выдает значение ИСТИНА тогда и только тогда, когда меньше половины его входов имеют значение ИСТИНА. В противном случае он выдает значение ЛОЖЬ. Сделайте набросок схемы на уровне транзисторов для КМОП-вентиля меньшинства. Используйте минимальное количество транзисторов.

Упражнение 1.87 Напишите таблицу истинности для функции вентиля на **Рис. 1.50**. таблица должна иметь два входа *A* и *B*. Как называется эта функция?

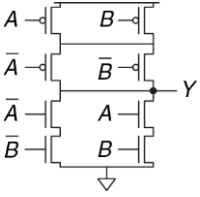


Рис. 1.50 **Таинственная схема**

Упражнение 1.88 Напишите таблицу истинности для функции вентиля на **Рис. 1.51**. таблица должна иметь три входа *A*, *B* и *C*.

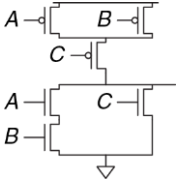


Рис. 1.51 **Таинственная схема**

Упражнение 1.89 Реализуйте следующие вентили с тремя входами, используя только псевдо-n-МОП-логические вентили. Используйте минимальное количество транзисторов.

a) Вентиль ИЛИ-НЕ

b) Вентиль И-НЕ

c) Вентиль И

Упражнение 1.90 Резисторно-транзисторная логика (*РТЛ*) использует n-МОП-транзисторы для выдачи значения НИЗКИЙ (LOW) и резистор с малым сопротивлением для выдачи значения ВЫСОКИЙ (HIGH), когда ни один из путей к заземлению не активен. Вентиль НЕ, построенный с помощью РТЛ, показан на **Рис. 1.52**. Сделайте набросок схемы РТЛ-вентиля ИЛИ-НЕ с тремя входами. Используйте минимальное количество транзисторов.

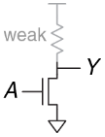


Рис. 1.52 **Вентиль НЕ**

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

Эти вопросы часто задают разработчикам цифровых систем в ходе собеседования при устройстве на работу:

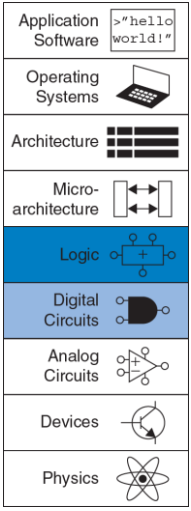
Вопрос 1.1 Сделайте набросок КМОП -схемы на уровне транзисторов для вентиля ИЛИ-НЕ с четырьмя входами.

Вопрос 1.2 Король получил 64 золотые монеты в виде налогов, однако у него есть основания полагать, что одна из них является поддельной. Король поручил Вам выявить поддельную монету. У вас есть весы, на чашки которых можно положить сколько угодно монет на каждой стороне. Сколько раз Вам нужно произвести взвешивание, чтобы найти более легкую фальшивую монету?

Вопрос 1.3 Профессор, преподаватель, студент, занимающийся проектированием цифровых схем, и первокурсник-чемпион по бегу хотят перейти шаткий мост темной ночью. Мост настолько плохой, что безопасно по нему могут одновременно пройти только два человека. У нашей группы всего лишь один фонарик, без него идти страшно, а мост слишком длинный, чтобы перебросить через него фонарик, так что после каждого перехода кто-то должен его перенести обратно к оставшимся людям. Первокурсник может пересечь мост за 1 минуту. Более старший студент может пересечь мост за 2 минуты. Преподаватель может пересечь мост в течение 5-ти минут. Профессор всегда отвлекается, поэтому ему нужно 10 минут, чтобы пересечь мост.
Как организовать переход, чтобы все перешли через мост за кратчайшее время?

Проектирование комбинационной логики

2



2.1 ВВЕДЕНИЕ

В цифровой электронике под *схемой* понимают электрическую цепь, которая обрабатывает дискретные сигналы. Такую схему можно рассматривать как «черный ящик», как показано на **Рис. 2.1**, при этом схема имеет:

- Один или более дискретных *входов*;
- Один или более дискретных *выходов*;
- *Функциональную спецификацию* (*functional specification*), описывающую взаимосвязь между входами и выходами;
- *Временну́ю спецификацию* (*timing specification*), описывающую задержку между изменением сигналов на входе и откликом выходного сигнала.

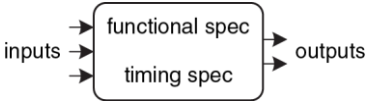


Рис. 2.1 **Схема как «черный ящик» с входами, выходами и спецификациями**

Если заглянуть внутрь такого «черного ящика», мы увидим, что схемы состоят из соединений, также называемых узлами (nodes), и элементов.

Элемент также представляет собой схему с входами, выходами и спецификацией. Соединение – это проводник, напряжение на котором соответствует дискретной переменной. Соединения подразделяются на входы, выходы и внутренние соединения. Входы получают сигналы извне. Выходы отправляют сигналы во внешний мир. Соединения, которые не являются входами или выходами, называются внутренними соединениями. На **Рис. 2.2** показана электронная схема с тремя элементами E1, E2 и E3 и шестью соединениями. Соединения *A*, *B* и *C* – входы, *Y* и *Z* – выходы, а n1 – внутреннее соединение между E1 и E3.

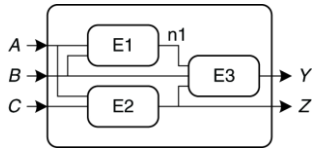


Рис. 2.2 **Элементы и соединения**

Цифровые схемы разделяются на комбинационные (combinational) и последовательностные (sequential). Выходы комбинационных схем зависят только от текущих значений на входах; другими словами, такие схемы комбинируют текущие значения входных сигналов для вычисления значения на выходе. Например, логический элемент – это

комбинационная схема. Выходы последовательностных схем зависят и от текущих, и от предыдущих значений на входах, то есть зависят от последовательности изменения входных сигналов. У комбинационных схем, в отличие от последовательностных схем, память отсутствует. Данная глава посвящена комбинационным схемам, а в **главе 3** мы рассмотрим последовательностные схемы.

Функциональная спецификация комбинационной схемы описывает зависимость значений на выходах от текущих входных значений. Временная спецификация комбинационной схемы состоит из нижней и верхней граничных значений задержки сигнала по пути от входа к выходу. В этой главе мы сначала рассмотрим функциональную спецификацию, а потом вернемся к временной.

На **Рис. 2.3** показана комбинационная схема с двумя входами и одним выходом. Входы *A* и *B* расположены слева, справа изображен выход *Y*. Символ ℄ в прямоугольнике означает, что этот элемент реализован с использованием исключительно комбинационной логики. В этом примере функция *F* определена как «ИЛИ»: $Y = F(A, B) = A + B$.

Другими словами, мы говорим, что выход *Y* – это функция двух входов *A* и *B*, а именно *Y* = *A* ИЛИ *B*. На **Рис. 2.4** показаны два возможных способа построения комбинационной логической схемы, приведенной на **Рис. 2.3**. Как мы неоднократно увидим в этой книге, зачастую

существует множество способов реализации одной и той же функции. Вы сами выбираете, как реализовать требуемую функцию, исходя из имеющихся в распоряжении «строительных блоков», а также ваших проектных ограничений. Эти ограничения часто включают в себя занимаемую на кристалле микросхемы площадь, скорость работы, потребляемую мощность и время разработки.

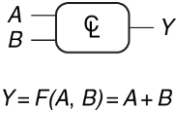


Рис. 2.3 **Комбинационная логическая схема**

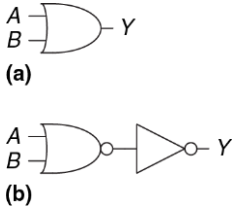


Рис. 2.4 **Два варианта схемы ИЛИ**

На **Рис. 2.5** показана комбинационная схема с несколькими выходами. Данная комбинационная схема называется полным сумматором, мы ещё вернёмся к ней в **разделе 5.2.1**. Два уравнения определяют значения на выходах S и C_{out} как функции входных сигналов A, B и C_{in}.

Для упрощения чертежей мы часто используем перечеркнутую косой чертой линию и число рядом с ней для обозначения *шины* (bus), то есть группы сигналов. Число показывает, сколько сигналов в шине (прим. переводчика: это число обычно называется *шириной шины*). Например, на **Рис. 2.6 (а)** показан блок комбинационной логики с тремя входами и двумя выходами. Если количество разрядов не имеет значения или очевидно из контекста, то косая черта может быть без числа рядом.

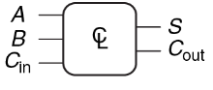


Рис. 2.5 **Многовыходная комбинационная схема**

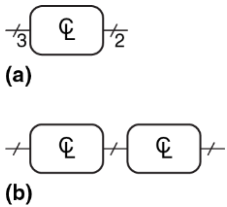


Рис. 2.6 **Обозначение шин на схемах**

На **Рис. 2.6 (b)** показаны два блока комбинационной логики с произвольным числом выходов одного блока, которые являются входами для другого блока.

Правила *комбинационной композиции* говорят нам, как мы можем построить большую комбинационную схему из более маленьких

комбинационных элементов. Схема является комбинационной, если она состоит из соединенных между собой элементов и выполнены следующие условия:

- Каждый элемент схемы сам является комбинационным;
- Каждое соединение схемы является или входом, или подсоединено к одному-единственному выходу другого элемента схемы;
- Схема не содержит циклических путей: каждый путь в схеме проходит через любое соединение не более одного раза.

Правила комбинационной композиции схем являются достаточными, но не строго необходимыми. Некоторые схемы, не подчиняющиеся этим правилам, все же являются комбинационными, поскольку значения их выходов зависят только от текущих значений на входах. Однако бывает довольно сложно определить, являются ли некоторые нетипичные схемы комбинационными или нет, поэтому обычно при разработке комбинационных схем мы ограничиваем себя правилами комбинационной композиции.

Пример 2.1 КОМБИНАЦИОННЫЕ СХЕМЫ

Какие из схем на **Рис. 2.7** являются, согласно правилам комбинационной композиции, комбинационными?

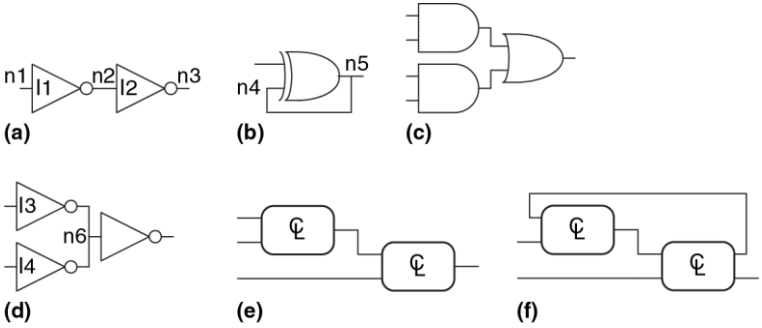


Рис. 2.7 Примеры схем

Решение: Схема (a) – комбинационная. Она составлена из двух комбинационных элементов (инверторы I1 и I2). В ней три соединения: n1, n2 и n3. Соединение n1 – вход схемы и вход для I1; n2 – внутреннее соединение, являющееся выходом для I1 и входом для I2; n3 – выход схемы и выход I2. Схема (b) – это не комбинационная схема, поскольку в ней есть циклический путь: выход элемента «исключающее ИЛИ» подключен к одному из его собственных входов, то есть циклический путь, начинаясь в n4, проходит через «исключающее ИЛИ» к n5, который ведет обратно к n4. Схема (c) – комбинационная, а (d) – не комбинационная, поскольку соединение n6 подключено к выходам двух элементов (I3 и I4). Схема (e) – комбинационная, представляющая собой две комбинационные схемы, соединенные между собой

и образующие более крупную комбинационную схему. Схема (f) не отвечает правилам комбинационной композиции, поскольку в ней есть циклический путь через два элемента. В зависимости от функций этих элементов эта схема может быть, а может и не быть комбинационной.

Большие схемы, такие как микропроцессоры, могут быть очень сложными, поэтому мы будем применять принципы, описанные в **главе 1**, чтобы справится со сложностью. Рассмотрение схемы как «черного ящика» с тщательно определенными интерфейсом и функцией есть применение принципов абстракции и модульности. Построение схемы из более мелких элементов является применением иерархического подхода к разработке. Правила комбинационной композиции суть применение дисциплины.

Функциональная спецификация комбинационной схемы обычно задается в виде таблицы истинности или булева уравнения. В следующих разделах будет описано, как вывести булево уравнение из любой таблицы истинности и как применять булеву алгебру и карты Карно для упрощения уравнений. Мы рассмотрим, как реализовывать эти уравнения, используя логические элементы, и как анализировать скорость работы таких схем.

2.2 БУЛЕВЫ УРАВНЕНИЯ

Булевы уравнения используют переменные, имеющие значение ИСТИНА или ЛОЖЬ, поэтому они идеально подходят для описания цифровой логики. В этом разделе сначала будет приведена терминология, часто используемая в булевых уравнениях, а затем будет показано, как записать булевы уравнения для любой логической функции по её таблице истинности.

2.2.1 Терминология

Дополнение (complement) переменной A – это ее отрицание $\overline{A}$. Переменная или ее дополнение называются *литералом*. Например, A, $\overline{A}$, B и $\overline{B}$ – литералы. Мы будем называть A прямой формой переменной, а $\overline{A}$ – комплементарной формой; «прямая форма» не подразумевает, что значение A равно ИСТИНЕ, а говорит лишь о том, что у A нет черты сверху.

Операция «И» над одним или несколькими литералами называется *конъюнкцией*, *произведением* (product) или *импликантой*. $\overline{A}B$, $A\overline{B}\,\overline{C}$ и B являются импликантами для функции трех переменных. *Минтерм* (minterm, элементарная конъюнктивная форма) – это произведение, включающее все входы функции. $A\overline{B}\,\overline{C}$ – это минтерм для функции трех переменных A, B и C, а $\overline{A}\,B$ – не минтерм, поскольку он не включает в

себя *C*. Аналогично, операция «ИЛИ» над одним или более литералами называется *дизъюнкцией* или суммой. *Макстерм* (maxterm, элементарная дизъюнктивная форма) – это сумма всех входов функции. $A + \overline{B} + C$ является макстермом функции трех переменных *A*, *B* и *C*.

Порядок операций важен при анализе булевых уравнений. Означает ли $Y = A + BC$, что $Y = (A$ ИЛИ $B)$ И C или $Y = A$ ИЛИ $(B$ И $C)$? В булевых уравнениях наибольший приоритет имеет операция НЕ, затем идет И, затем ИЛИ. Как и в обычных уравнениях, произведения вычисляются до вычисления сумм. Таким образом, правильно уравнение читается как $Y = A$ ИЛИ $(B$ И $C)$. **уравнение (2.1)** – ещё один пример, показывающий порядок операций.

$$\overline{A}\,B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \qquad (2.1)$$

2.2.2 Дизъюнктивная форма

Таблица истинности для функции *N* переменных содержит 2^N строк, по одной для каждой возможной комбинации значений входов. Каждой строке в таблице истинности соответствует минтерм, который имеет значение ИСТИНА для этой строки. На **Рис. 2.8** показана таблица истинности функции двух переменных *A* и *B*. В каждой строке показан соответствующий ей минтерм. Например, минтерм для первой строки –

это $\overline{A}\overline{B}$, поскольку $\overline{A}\overline{B}$ имеет значение ИСТИНА тогда, когда $A = 0$ и $B = 0$. Минтермы нумеруют начиная с 0; первая строка соответствует минтерму 0 (m_0), следующая строка – минтерму 1 (m_1), и так далее.

A	*B*	*Y*	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	0	$A\,B$	m_3

Рис. 2.8 **Таблица истинности и минтермы**

Можно написать булево уравнение для любой таблицы истинности путем суммирования всех тех минтермов, для которых выход Y имеет значение ИСТИНА. Например, на **Рис. 2.8** есть только одна строка (минтерм), для которой выход Y имеет значение ИСТИНА, она отмечена синим цветом. Таким образом, $Y = \overline{A}B$. На **Рис. 2.9** показана таблица, в которой выход имеет значение ИСТИНА для нескольких строк. Суммирование отмеченных минтермов дает $Y = \overline{A}\overline{B} + AB$.

Такая сумма минтермов называется *совершенной дизъюнктивной нормальной формой* функции (sum-of-products canonical form). Она представляет собой сумму (операцию «ИЛИ») произведений (операций «И», образующих минтермы). Хотя существует много

способов записать одну и ту же функцию, такую как $Y = \overline{A}\,B + AB$, мы будем записывать минтермы в том же порядке, как в таблице истинности, чтобы всегда получать одно и то же булево выражение для одной и той же таблицы истинности. Совершенная дизъюнктивная нормальная форма также может быть записана через символ суммы Σ. При использовании такого обозначения функция на **Рис. 2.9** будет выглядеть так:

$$F(A, B) = \Sigma(m_1, m_3)$$

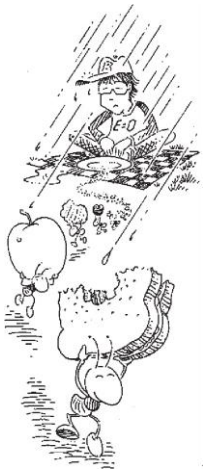
или (2.2)

$$F(A, B) = \Sigma(1, 3)$$

A	*B*	*Y*	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

Рис. 2.9 Таблица истинности с несколькими минтермами, равными ИСТИНЕ

Пример 2.2 ДИЗЪЮНКТИВНАЯ ФОРМА



У Бена Битдидла намечается пикник. Он не обрадуется, если пойдёт дождь или появятся муравьи. Постройте схему, в которой выход будет принимать значение ИСТИНА только в том случае, если Бену пикник понравится.

Решение: Сначала определим входы и выходы. Входами будут переменные *A* и *R*, что означает муравьев (ants) и дождь (rain). Значение A принимает значение ИСТИНА, когда муравьи есть, и ЛОЖЬ, когда муравьев нет. Аналогично, *R* имеет значение ИСТИНА, когда идёт дождь, и ЛОЖЬ, когда Бену светит солнце. Выход *E* (enjoyment, радость) показывает настроение Бена. *E* имеет значение ИСТИНА, когда Бен радуется пикнику, и ЛОЖЬ, когда он страдает. На **Рис. 2.10** показана таблица истинности впечатлений Бена от пикника.

Используя дизъюнктивную форму, запишем уравнение так: $E = A\bar{R}$ или $E = \Sigma(0)$. Мы можем реализовать соответствующую схему, используя два инвертора и двухвходовой элемент И, как показано на **Рис. 2.11 (а)**. Вы могли заметить, что эта таблица является точно такой же, как и таблица для функции «ИЛИ-НЕ», рассмотренной в **разделе 1.5.5**: $E = A$ ИЛИ-НЕ $R = \overline{A+R}$

На **Рис. 2.11 (b)** показана реализация на базе элемента ИЛИ-НЕ. В **разделе 2.3** мы покажем, что выражения $\overline{\overline{A}\overline{R}}$ и $\overline{A+R}$ эквивалентны.

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Рис. 2.10 **Таблица истинности Бена**

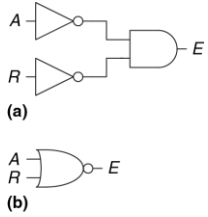


Рис. 2.11 **Схема Бена**

Совершенная дизъюнктивная нормальная форма позволяет записать булево уравнение для любой таблицы истинности с любым количеством переменных. На **Рис. 2.12** показана произвольная таблица истинности для трехвходового элемента. Совершенная дизъюнктивная нормальная форма соответствующей логической функции выглядит так:

$$Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$$

или (2.3)

$$Y = \Sigma(0, 4, 5)$$

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Рис. 2.12 Произвольная таблица истинности с тремя входами

К сожалению, совершенная дизъюнктивная нормальная форма не всегда позволяет получить простое уравнение. В **разделе 2.3** мы покажем, как записать одну и ту же функцию, используя меньшее число членов уравнения.

2.2.3 Конъюнктивная форма

Альтернативный способ выражения булевых функций – это *совершенная конъюнктивная нормальная форма* (products-of-sum

forms). Каждая строка таблицы истинности соответствует макстерму, который имеет значение ЛОЖЬ для этой строки. Например, макстерм для первой строки для двухвходовой таблицы истинности – это (A + B), поскольку (A + B) имеет значение ЛОЖЬ, когда A = 0 и B = 0. Для любой схемы, заданной таблицей истинности, мы можем записать ее булево уравнение как логическое «И» всех макстермов, для которых выход имеет значение ЛОЖЬ. Совершенная конъюнктивная нормальная форма также может быть записана с использованием символа Π.

Пример 2.3 КОНЪЮНКТИВНАЯ ФОРМА

Запишите уравнение в совершенной конъюнктивной нормальной форме для таблицы истинности на **Рис. 2.13**.

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Рис. 2.13 Таблица истинности с макстермами

Решение: Таблица истинности имеет две строки, в которых выход имеет значение ЛОЖЬ. Следовательно, функция может быть записана в конъюнктивной форме так:

Y = (A + B) ($\overline{A}$ + B). Также функция может быть записана как $Y = \Pi(M_0, M_2)$ или $Y = \Pi(0, 2)$. Первый макстерм, (A + B), гарантирует, что Y = 0 для A = 0 и B = 0, так как логическое «И» любого значения и нуля дает нуль. Аналогично, второй макстерм ($\overline{A}$ + B) гарантирует, что Y = 0 для

комбинации $A = 1$ и $B = 0$. На **Рис. 2.13** показана такая же таблица истинности, как и на **Рис. 2.9**, чтобы продемонстрировать, что одна и та же функция может быть записана более чем одним способом.

Аналогично, булево уравнение для пикника Бена (**Рис. 2.10**) может быть записано в совершенной конъюнктивной нормальной форме, если обвести три строки с нулями для того, чтобы получить

$E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$ или $E = \Pi(1, 2, 3)$.

Это не такая красивая запись, как дизъюнктивное уравнение, $E = \overline{A}\,\overline{R}$, но эти два уравнения логически эквивалентны. Дизъюнктивная форма дает более короткое уравнение, когда выход имеет значение ИСТИНА только в нескольких строках таблицы истинности; конъюнктивная же форма проще, когда выход имеет значение ЛОЖЬ только в нескольких строках таблицы истинности.

2.3 БУЛЕВА АЛГЕБРА

В предыдущем разделе мы изучили, как записывать булевы выражения при наличии таблицы истинности. Однако, выражение, получаемое таким способом, не обязательно приводит к простейшему набору логических элементов. Вы можете использовать булеву алгебру для упрощения булевых уравнений точно так же, как вы используете алгебру для упрощения математических уравнений. Правила булевой

алгебры очень похожи на правила обычной алгебры, но в некоторых случаях они проще, потому что переменные имеют только два возможных значения: 0 или 1.

Булева алгебра основана на наборе аксиом, которые мы считаем верными. Аксиомы являются недоказуемыми в том смысле, что определение не может быть доказано. С помощью этих аксиом мы доказываем все теоремы булевой алгебры.

Эти теоремы имеют огромную практическую значимость, потому что с их помощью мы учимся тому, как упрощать логические уравнения, чтобы получать более дешевые и компактные схемы. Аксиомы и теоремы булевой алгебры подчиняются принципу двойственности. Если взаимно заменить символы 0 и 1, а так же взаимно заменить операторы • (И) и + (ИЛИ), то булево выражение останется верным. Мы используем символ «штрих» (’) для обозначения двойственного выражения.

2.3.1 Аксиомы

В **Табл. 2.1** приведены аксиомы булевой алгебры. Эти пять аксиом и двойственные им аксиомы определяют булевы переменные и значения операторов НЕ, И, ИЛИ. Аксиома A1 показывает, что булева переменная B имеет значение 0, если она не имеет значение 1.

Двойственное выражение для этой аксиомы A1’ утверждает, что переменная принимает значение 1, если она не имеет значение 0. Вместе аксиомы A1 и A1’ говорят нам, что мы работает в булевом, то есть бинарном поле, состоящем из значений нулей и единиц. Аксиомы A2 и A2’ определяют операцию НЕ. Аксиомы с A3 по A5 определяют операцию И, а их двойственные аксиомы (A3’ – A5’) определяют операцию ИЛИ.

Табл. 2.1 Аксиомы булевой алгебры

	Аксиома		Двойственная аксиома	Название
A1	$B = 0$ если $B \neq 1$	A1′	$B = 1$ если $B \neq 0$	Бинарное поле
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	НЕ
A3	$0 \cdot 0 = 0$	A3′	$1 + 1 = 1$	И/ИЛИ
A4	$1 \cdot 1 = 1$	A4′	$0 + 0 = 0$	И/ИЛИ
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	И/ИЛИ

2.3.2 Теоремы одной переменной

Теоремы с T1 по T5 в **Табл. 2.2** описывают, как упростить уравнения, содержащие одну переменную.

Теорема *идентичности* T1 утверждает, что для любой булевой переменной B выполнено B И $1 = B$. Двойственная ей теорема говорит о том, что B ИЛИ $0 = B$. В аппаратуре, как показано на **Рис. 2.14**, T1

означает, что если уровень сигнала на одном из входов двухвходового элемента И всегда равен 1, то мы можем удалить этот элемент и заменить его проводом, соединяющим выход этого элемента с входом *B*, значение которого может меняться. Точно так же теорема T1' говорит о том, что если один вход двухвходового элемента ИЛИ всегда равен 0, мы можем заменить этот элемент на провод, соединенный с входом *B*. Как правило, элементы имеют определенную стоимость, энергопотребление и задержку прохождения сигнала, поэтому замена элемента на провод является целесообразной.

Табл. 2.2 Булевы теоремы для одной переменной

	Теорема		Двойственная теорема	Название
T1	$B \cdot 1 = B$	T1′	$B + 0 = B$	Идентичность
T2	$B \cdot 0 = 0$	T2′	$B + 1 = 1$	Нулевой элемент
T3	$B \cdot B = B$	T3′	$B + B = B$	Идемпотентность
T4		$\overline{\overline{B}} = B$		Инволюция
T5	$B \cdot \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Дополнительность

Теорема о *нулевом элементе* T2 говорит, что *B* И 0 всегда равно 0. Следовательно, 0 называют нулевым элементом для операции И, потому что он обнуляет эффект любого другого входа. Двойственная ей теорема говорит о том, что *B* ИЛИ 1 всегда равно 1. Таким образом, 1 – это нулевой элемент для операции ИЛИ. В аппаратуре, как показано

на **Рис. 2.15**, если один вход элемента И равен 0, мы можем заменить элемент И проводом, подключенным к низкому логическому уровню (0). Точно так же, если один из входов элемента ИЛИ равен 1, мы можем заменить элемент ИЛИ на провод, который подключен к высокому логическому уровню (1).

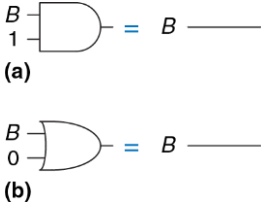


Рис. 2.14 **Теорема идентичности в аппаратуре: (a) T1, (b) T1'**

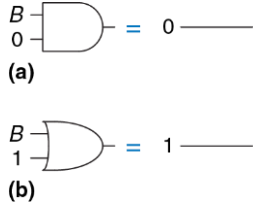


Рис. 2.15 **Теорема о нулевом элементе в аппаратуре: (a) T2, (b) T2'**

Теорема о нулевом элементе приводит к нелепым утверждениям, которые при этом оказываются верными! Эта теорема становится особенно опасной, когда её применяют те, кто делает рекламу: «ВЫ ПОЛУЧИТЕ МИЛЛИОН ДОЛЛАРОВ или мы пришлём вам по почте зубную щётку» (скорее всего, вы получите зубную щётку по почте).

Теорема об *идемпотентности* Т3 утверждает, что операция логического «И» двух равных друг другу переменных имеет значение, равное этой переменной. Аналогичное утверждение верно для операции «ИЛИ» с двумя одинаковыми значениями на входах. Название теоремы происходит от латинских слов «idem» – *тот же, такой же* и «potent» – *сила*. Операции возвращают те же значения, которые вы подаете им на вход. На **Рис. 2.16** показано, как идемпотентность позволяет заменить элемент схемы на провод.

Теорема об *инволюции* Т4 – это забавный способ описания того, что двойное отрицание переменной дает её исходное значение. Два последовательно включенных инвертора логически отменяют друг друга, то есть они эквивалентны проводу, как показано на **Рис. 2.17**. Двойственной ей теоремой является она сама.

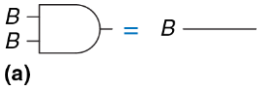


Рис. 2.16 Теорема об идемпотентности в аппаратуре: (a) T3, (b) T3'

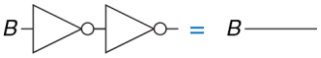
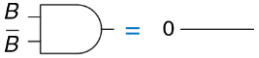
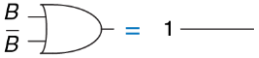


Рис. 2.17 Теорема об инволюции в аппаратуре: T4

Теорема о *дополнительности* T5 (**Рис. 2.18**) утверждает, что операция И над переменной и её инверсным значением дает 0 (потому что одна из них всегда будет равна нулю). И, согласно принципу двойственности, операция ИЛИ над переменной и её инверсным значением всегда дает 1 (так как одна из них всегда будет равна единице).



(a)



(b)

Рис. 2.18 **Теорема инволюции в аппаратуре: (a) T5, (b) T5'**

2.3.3 Теоремы с несколькими переменными

Теоремы с T6 по T12 в **Табл. 2.3** описывают, как упростить уравнения, включающие в себя более одной булевой переменной.

Теоремы T6 о *коммутативности* и T7 об ассоциативности работают так же, как и в традиционной алгебре. В соответствии с принципом коммутативности порядок входов для функций И или ИЛИ не влияет на

значение выхода. Согласно принципу ассоциативности любое группирование входов не влияет на значение выхода.

Теорема о *дистрибутивности* T8 является точно такой же, как и в традиционной алгебре, а двойственная ей теорема T8‘ – нет. Согласно теореме T8 оператор И дистрибутивен относительно операции ИЛИ. T8‘ говорит, что оператор ИЛИ дистрибутивен относительно операции И. В традиционной алгебре оператор умножения дистрибутивен относительно операции сложения, но не наоборот, то есть

$(B + C) \times (B + D) \neq B + (C \times D)$.

Теоремы *поглощения*, *склеивания* и *согласованности* T9 – T11 позволяют нам удалять излишние переменные. Если вы немного подумаете, вы сможете убедиться, что эти теоремы справедливы.

Табл. 2.3 Булевы теоремы для нескольких переменных

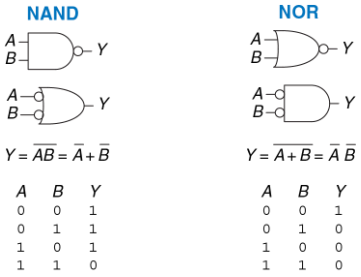
	Теорема		Двойственная теорема	Название
T6	$B \cdot C = C \cdot B$	T6′	$B + C = C + B$	Коммутативность
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	T7′	$(B + C) + D = B + (C + D)$	Ассоциативность
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	T8′	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Дистрибутивность
T9	$B \cdot (B + C) =$ B	T9′	$B + (B \cdot C) = B$	Поглощение
T10	$(B \cdot C) + (B \cdot \overline{C}) = B$	T10′	$(B + C) \cdot (B + \overline{C}) = B$	Склеивание
T11	$(B \cdot$ C$) + (\overline{B} \cdot D) +$ (C $\cdot D) =$	T11′	$(B + C) \cdot (\overline{B} + D) \cdot (C + D) =$	Согласованность

$B \bullet C + \overline{B} \bullet D$	$(B + C) \bullet (\overline{B} + D)$
T12 $\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = \left(\overline{B_0} + \overline{B_1} + \overline{B_2 \ldots}\right)$	T12′ $\overline{B_0 + B_1 + B_2 \ldots} = \left(\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots\right)$ Теорема де Моргана

Теорема *де Моргана* T12 является особенно мощным инструментом при разработке цифровых устройств. Эта теорема поясняет, что дополнение результата умножения всех термов равно сумме дополнений каждого терма. Ана

логично дополнение суммы всех термов равно результату умножения дополнений каждого терма.

В соответствии с теоремой де Моргана, элемент И-НЕ эквивалентен элементу ИЛИ с инвертированными входами. Аналогично, ИЛИ-НЕ эквивалентен элементу И с инвертированными входами. На **Рис. 2.19** показаны эквивалентные по де Моргану элементы И-НЕ и ИЛИ-НЕ. Каждая пара символов, приведенная для каждой функции, называется двойственной. Они логически эквивалентны и взаимозаменяемы.



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Рис. 2.19 **Эквивалентные по де Моргану элементы**

Кружочек на графическом обозначении элементов является обозначением отрицания (инверсии). Интуитивно вы можете представить, что если «вдавить» этот кружочек с одной стороны логического элемента, то он «выскочит» на другой, при этом тип элемента изменится с И на ИЛИ (и наоборот). Это называется «перемещением инверсии». Например, элемент И-НЕ на **Рис. 2.19** состоит из элемента И с отрицанием на выходе. Перемещение инверсии влево приводит к получению элемента ИЛИ с двумя отрицаниями на входах. Базовые правила для перемещения инверсии таковы:

- Перемещение инверсии назад (от выхода) или вперед (от входов) меняет тип элемента с И на ИЛИ и наоборот;
- Перемещение инверсии с выхода назад ко входам приводит к тому, что на всех входах появляется инверсия;
- Перемещение инверсии со всех входов элемента к выходу приводит к появлению инверсии на выходе.

В **разделе 2.5.2** принцип перемещения инверсии используется для анализа схем.

Август де Морган

Август де Морган, умер в 1871 г. Британский математик, родился в Индии. Был слепым на один глаз. Его отец умер, когда ему было 10 лет. Поступил в Тринити Колледж в Кембридже, и был назначен

профессором математики в возрасте 22 лет в только что открытом в то время Лондонском университете. Много писал на различные математические темы, включая логику, алгебру и парадоксы. В честь де Моргана был назван кратер на Луне. Он придумал загадку про год своего рождения: «Мне было X лет в году X^2».

Пример 2.4 ПОЛУЧИТЕ КОНЪЮНКТИВНУЮ ФОРМУ

На **Рис. 2.20** приведена таблица истинности для булевой функции *Y* и её дополнения *Y*. Используя теорему де Моргана, получите конъюнктивную нормальную форму функции *Y* из дизъюнктивной формы *Y*.

Решение: На **Рис. 2.21** обведены минтермы, содержащиеся в функции *Y*. Дизъюнктивная нормальная форма функции Y имеет следующий вид:

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B \tag{2.4}$$

Применяя операцию инверсии к обеим частям уравнения и дважды используя теорему де Моргана, получаем:

$$\overline{\overline{Y}} = (Y) = \overline{\overline{A}\overline{B} + \overline{A}B} = (\overline{\overline{A}\overline{B}})(\overline{\overline{A}B}) = (A+B)(A+\overline{B}) \tag{2.5}$$

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Рис. 2.20 **Таблица истинности, показывающая *Y* и *Y'***

A	B	Y	$\overline{Y}$	minterm
0	0	0	1	$\overline{A}\,\overline{B}$
0	1	0	1	$\overline{A}\,B$
1	0	1	0	$A\,\overline{B}$
1	1	1	0	$A\,B$

Рис. 2.21 **Таблица истинности, показывающая *Y* и *Y'***

2.3.4 Правда обо всем этом

Любопытный читатель может задать вопрос о том, как же доказать правильность теоремы. В булевой алгебре доказательство теорем с конечным числом переменных является простым: нужно показать, что теорема верна для всех возможных значений этих переменных. Этот метод называется *совершенной индукцией* и может быть выполнен с использованием таблицы истинности.

Пример 2.5 ДОКАЗАТЕЛЬСТВО ТЕОРЕМЫ СОГЛАСОВАННОСТИ МЕТОДОМ ПОЛНОГО ПЕРЕБОРА

Докажите теорему согласованности T11 из **Табл. 2.3**.

Решение: проверьте обе части уравнения для всех восьми комбинаций переменных *B*, *C* и *D*. Таблица истинности на **Рис. 2.22** иллюстрирует все эти

комбинации. Поскольку равенство $BC + \overline{B}\,D + CD = BC + \overline{B}\,D$ верно для всех случаев, теорема доказана.

B	C	D	$BC + \overline{B}D + CD$	$BC + \overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Рис. 2.22 **Таблица истинности, доказывающая T11**

2.3.5 Упрощение уравнений

Теоремы булевой алгебры помогают нам упрощать булевы уравнения. Например, возьмём дизъюнктивную форму выражения из таблицы истинности на **Рис. 2.9**: $Y = \overline{A}\,\overline{B} + A\overline{B}$. В соответствии с теоремой T10, уравнение можно упростить до $Y = \overline{B}$. Возможно, это очевидно при взгляде на таблицу истинности. В общем случае может потребоваться несколько шагов для упрощения более сложных уравнений.

Основной принцип упрощения дизъюнктивных уравнений – это комбинирование термов с использованием отношения $PA + P\overline{A} = P$, где P может быть любой импликантой. Насколько может быть упрощено уравнение? По определению уравнение дизъюнктивной формы

является минимизированным, если оно включает в себя минимально возможное количество импликант. Если есть несколько уравнений с одинаковым количеством импликант, минимальным будет то уравнение, в котором меньше литералов.

Импликанта называется простой (prime implicant), если она не может быть объединена с другими импликантами в уравнении для того, чтобы образовать новую импликанту с меньшим количеством литералов. Все импликанты в минимальном уравнении должны быть простыми. Иначе, они могут быть объединены, чтобы уменьшить количество литералов.

Пример 2.6 МИНИМИЗАЦИЯ УРАВНЕНИЯ

Минимизируйте **уравнение (2.3)**: $\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$

Решение: Мы начинаем с исходного уравнения и применяем булевы теоремы шаг за шагом, как показано в **Табл. 2.4**.

Упростили ли мы полностью уравнение на этой стадии? Давайте посмотрим внимательно. В оригинальном уравнении минтермы $\overline{A}\,B\,\overline{C}$ и $A\overline{B}\,\overline{C}$ отличаются только переменной A. Поэтому мы объединяем минтермы и получаем $\overline{B}\;\overline{C}$. Однако, если мы посмотрим на исходное уравнение, мы заметим, что последние два минтерма $A\overline{B}\,\overline{C}$ и $A\overline{B}\,C$ также отличаются одним литералом (C и $\overline{C}$). Таким образом, используя тот же самый метод, мы могли бы объединить эти два минтерма и получить минтерм $A\overline{B}$. Можно сказать, что импликанты $\overline{B}\overline{C}$ и $A\overline{B}$ делят между собой минтерм $A\overline{B}\,\overline{C}$.

Итак, остановились ли мы на упрощении только одной пары минтермов, или мы можем упростить обе? Используя теорему об идемпотентности, мы можем дублировать минтермы столько раз, сколько нам нужно: $B = B + B + B + B\ldots$ Используя этот принцип, мы полностью упрощаем уравнение до его простых импликант, $\overline{B}\overline{C} + A\overline{B}$, как показано в **Табл. 2.5**.

Табл. 2.4 Минимизация выражения

Шаг	Выражение	Объяснение
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,C$	
1	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}\,C$	Т8: дистрибутивность
2	$\overline{B}\,\overline{C}(1) + A\overline{B}\,C$	Т5: дополнительность
3	$\overline{B}\,\overline{C} + A\overline{B}\,C$	Т1: идентичность

Табл. 2.5 Улучшенная минимизация выражения

Шаг	Выражение	Объяснение
	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,C$	
1	$\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,C$	Т3: идемпотентность
2	$\overline{B}\,\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	Т8: дистрибутивность
3	$\overline{B}\,\overline{C}(1) + A\overline{B}(1)$	Т5: дополнительность
4	$\overline{B}\,\overline{C} + A\overline{B}$	Т1: идентичность

Хотя это немного нелогично, расширение импликанты (например, превращение AB в $ABC + AB\overline{C}$) иногда полезно при минимизации уравнений. Делая так, вы можете повторять один из расширенных минтермов для его объединения с другим минтермом.

Вы могли заметить, что полное упрощение булевых уравнений при помощи теорем булевой алгебры может потребовать нескольких попыток,

некоторые из которых будут ошибочными. В **разделе 2.7** описана методика, позволяющая упростить процесс минимизации – карты Карно.

Зачем же трудиться над упрощением булева уравнения, если оно остается логически эквивалентным? Упрощение уменьшает количество элементов, используемых при физическом воплощении функции в аппаратуре, тем самым делая схему меньше, дешевле и, возможно, быстрее. В следующем разделе рассказывается, как воплощать булевы уравнения при помощи логических элементов.

2.4 ОТ ЛОГИКИ К ЛОГИЧЕСКИМ ЭЛЕМЕНТАМ

Принципиальная схема – это изображение цифровой схемы, показывающее элементы и соединяющие их проводники. Например, схема на **Рис. 2.23** показывает возможную аппаратную реализацию нашей любимой логической функции (**уравнение (2.3)**):

$$Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$$

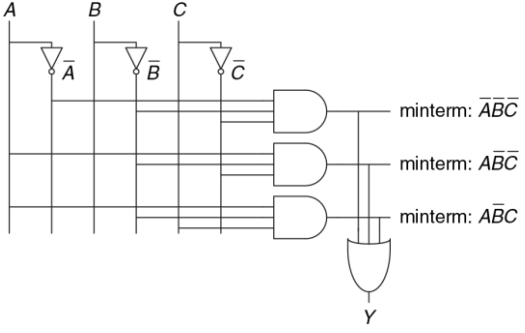


Рис. 2.23 **Схема $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$**

Изображая принципиальные схемы в унифицированном виде, нам становится легче читать их и отлаживать. В большинстве случаев мы будем придерживаться следующих правил:

- Входы изображаются на левой (или верхней) части схемы;
- Выходы изображаются на правой (или нижней) части схемы;
- Всегда, когда это возможно, элементы необходимо изображать слева направо;
- Проводники лучше изображать прямыми линиями, чем линиями с множеством углов (неровные рваные линии отвлекают внимание: приходится следить за тем, куда ведут провода, а не думать о том, что делает схема);
- Проводники всегда должны соединяться в виде буквы «Т»;
- Точка в месте пересечения проводников обозначает их соединение;
- Проводники, пересекающиеся без точки, не имеют соединения друг с другом.

Три последних правила показаны на **Рис. 2.24**.

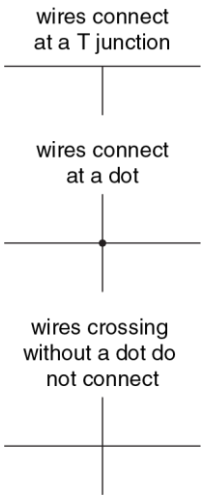


Рис. 2.24
Wireconnections

Любое булево уравнение в дизъюнктивной форме может быть изображено в виде принципиальной схемы с использованием систематического подхода, как показано на **Рис. 2.23**. Сначала нарисуйте вертикальные проводники для входов. Поместите инверторы на соседних вертикальных линиях для получения комплементарных входов, если это необходимо. Нарисуйте горизонтальные линии, ведущие к элементам И, для каждого минтерма. Затем для каждого выхода нарисуйте элемент ИЛИ, соединенный с минтермом, соответствующим этому выходу. Такой стиль изображения называется программируемой логической матрицей (ПЛМ, PLA), потому что инверторы, элементы И и элементы ИЛИ систематически объединены в массивы. Программируемые логические матрицы будут рассмотрены в **разделе 5.6**.

На **Рис. 2.25** показана реализация упрощенного уравнения, которое мы получили при помощи булевой алгебры в Примере 2.6. Заметьте, что упрощенная схема имеет значительно меньше аппаратных элементов,

чем схема на **Рис. 2.23**. Также ее быстродействие может быть выше, поскольку она использует элементы с меньшим количеством входов.

Мы даже можем ещё уменьшить количество элементов (пусть хотя бы на один инвертор), если воспользуемся преимуществом инвертирующих логических элементов. Заметьте, что $\overline{B}\overline{C}$ – это элемент И с инвертированными входами. На **Рис. 2.26** показана схема, которая использует эту оптимизацию для исключения инвертора на входе *C*. Вспомните, что согласно теореме де Моргана логический элемент И с инвертированными входами эквивалентен элементу ИЛИ-НЕ. В зависимости от технологии реализации, использование наименьшего числа элементов или использование элементов определенного типа взамен других может быть выгоднее. Например, в технологии КМОП элементы И-НЕ и ИЛИ-НЕ более предпочтительны, чем И или ИЛИ.

У многих схем имеется несколько выходов, каждый из которых вычисляет независимые булевы функции для входов. Мы можем записать отдельные таблицы истинности для каждого выхода, но часто удобно записать все выходы в одну таблицу истинности и начертить одну схему для всех выходов.

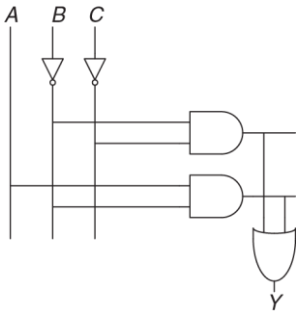


Рис. 2.25 **Схема реализации функции $Y = \bar{B}\bar{C} + A\bar{B}$**

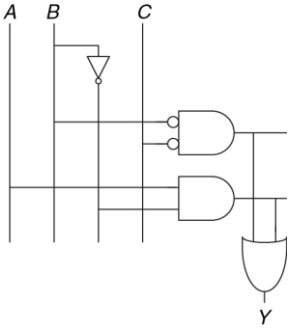


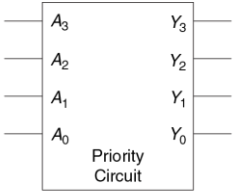
Рис. 2.26 **Схема, использующая меньше элементов**

Пример 2.7 СХЕМЫ С НЕСКОЛЬКИМИ ВЫХОДАМИ

Декан, заведующий кафедрой, аспирант и председатель совета общежития время от времени используют одну аудиторию. К сожалению, иногда аудитория нужна им одновременно, что приводит к катастрофам, как, например, когда встреча декана с пожилыми и уважаемыми членами попечительского совета была запланирована на то же время, что и пивная вечеринка студентов общежития. Алиса Хакер была приглашена для того, чтобы разработать систему резервирования комнаты.

Система имеет четыре входа (A_3, …, A_0) и четыре выхода (Y_3, …, Y_0). Эти сигналы также могут быть записаны в виде $A_{3:0}$ и $Y_{3:0}$. Каждый пользователь активирует свой вход, когда запрашивает аудиторию на следующий день. Система активирует только один выход, подтверждая пользование аудиторией самым высокоприоритетным пользователем. Декан, который оплачивает систему, требует наивысший приоритет (3). Заведующий кафедрой, аспирант и председатель совета общежития имеют приоритеты по убыванию. Запишите таблицу истинности и булевы уравнения для этой системы. Начертите схему, которая будет выполнять эту функцию.

Решение: Данная функция называется четырехвходовой схемой приоритета. Её обозначение и таблица истинности показаны на **Рис. 2.27**. Мы могли бы записать каждый выход в дизъюнктивной форме и упростить уравнения, используя булеву алгебру. Однако достаточно посмотреть на функциональное описание (таблицу истинности), чтобы понять, каковы могут быть упрощенные уравнения: Y_3 имеет значение ИСТИНА всегда, когда подается сигнал A_3, таким образом $Y_3 = A_3$. Y_2 равен ИСТИНЕ, если подан сигнал A_2 и не подан сигнал A_3, таким образом $Y_2 = \overline{A}_3 A_2$. Y_1 имеет значение ИСТИНА, если подан сигнал A_1 и ни на какой из более высокоприоритетных входов сигнал не подан: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$. Y_0 имеет значение ИСТИНА при поданном сигнале A_0 и когда ни один из других выходов не активирован: $Y_1 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$. Схема показана на **Рис. 2.28**. Опытный разработчик часто может реализовать логическую схему, непосредственно глядя в исходные данные. При наличии четко заданной спецификации, просто преобразуйте слова в уравнения, а уравнения в логические элементы схемы.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Рис. 2.27 Схема приоритета

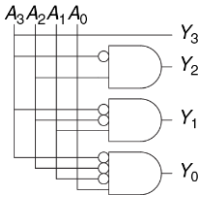


Рис. 2.28 Принципиальная схема

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Рис. 2.29 Таблица истинности схемы приоритета

Символ «X» используется не только для обозначения переменных, чье состояние нам безразлично, но и для обозначения недопустимых состояний сигналов при симуляции логических схем (см. **Раздел 2.6.1**). Старайтесь понять из контекста, о каком варианте использования идет речь. Чтобы избежать такой двусмысленности, некоторые авторы используют символы «D» или «?» для обозначения сигналов, состояние которых нам безразлично.

Обратите внимание, что если в схеме приоритета подается сигнал A_3, то выходы схемы не будут зависеть от того, какие сигналы присутствуют на остальных входах. Мы используем символ X для описания состояния входов, которые нам безразличны, так как не оказывают влияния на выход. На **Рис. 2.29** показано, что таблица

истинности четырехвходовой приоритетной схемы становится гораздо меньше, если убрать значения входов, которыми можно пренебречь. Из этой таблицы истинности мы можем легко получить булевы уравнения в дизъюнктивной форме, опуская входы с X. Значения, которыми можно пренебречь, также могут возникнуть на выходах в таблице истинности, как мы увидим в **разделе 2.7.3**.

2.5 МНОГОУРОВНЕВАЯ КОМБИНАЦИОННАЯ ЛОГИКА

Комбинационная логика, построенная как дизъюнкция конъюнкций (сумма произведений), называется двухуровневой, потому что состоит из литералов, соединенных с элементами И (образующими первый уровень), выходы которых соединены с элементами ИЛИ (образующими второй уровень). Разработчики часто создают схемы с большим числом уровней логических элементов. Такая многоуровневая комбинационная схема может использовать меньше логических элементов, чем ее двухуровневая реализация. Эквивалентные преобразования по законам де Моргана и перемещение инверсии особенно полезны при анализе и разработке многоуровневых схем.

2.5.1 Минимизация аппаратуры

Некоторые логические функции требуют огромного количества аппаратуры, если строить их с использованием двухуровневой логики. Показательный пример – это функция ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) нескольких переменных. Например, рассмотрим построение трехвходового элемента XOR, используя двухуровневую технику, которую мы изучали до сих пор.

Вспомним, что N-входовой XOR выдает на выход значение ИСТИНА, если нечетное число входных операндов имеют значение ИСТИНА. На **Рис. 2.30 (а)** показана таблица истинности трехвходового элемента XOR. В таблице обведены строки, для которых значение выхода будет ИСТИНА. Из таблицы истинности мы понимаем форму логического выражения, соответствующую дизъюнкции конъюнкций (сумме произведений) **уравнения (2.6)**. К сожалению, это выражение невозможно упростить в меньшее количество импликант.

$$Y = \overline{A}\overline{B}C \;\; + \overline{A}B\overline{C} + A\overline{B}\;\overline{C} + ABC \tag{2.6}$$

С другой стороны, A ⊕ B ⊕ C = (A ⊕ B) ⊕ C (если вы сомневаетесь, докажите это самостоятельно с помощью совершенной индукции). Следовательно, трехвходовой элемент XOR можно реализовать каскадом двухвходовых элементов XOR, как показано на **Рис. 2.31**.

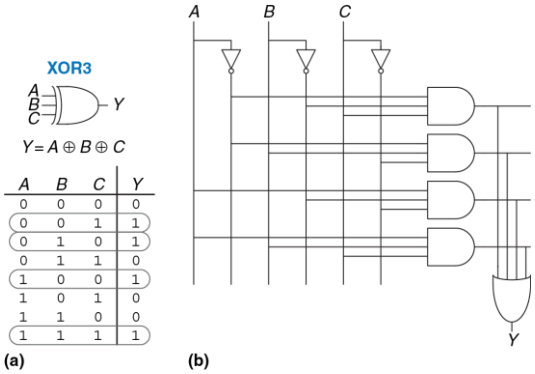


Рис. 2.30 **Трехвходовой элемент XOR: функциональная спецификация (a) и реализация с двумя уровнями логики (b)**

Аналогично, восьмивходовой XOR потребует 128 восьмивходовых элементов И и одного 128-входового элемента ИЛИ для двухуровневой реализации дизъюнкции конъюнкций. Гораздо лучшей альтернативой будет использовать дерево двухвходовых элементов XOR, как показано на **Рис. 2.32**.

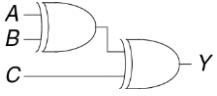


Рис. 2.31 Трехвходовой элемент XOR, собранный из двух двухвходовых элементов XOR

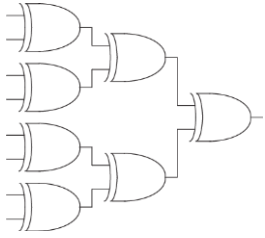


Рис. 2.32 Восьмивходовой элемент XOR, собранный из семи двухвходовых

Выбор наилучшей многоуровневой реализации заданной логической функции – это непростой процесс (выбирать наилучшую многоуровневую реализацию заданной логической функции непросто). Кроме того, «наилучшее» имеет много значений: наименьшее количество элементов, лучшее быстродействие, кратчайшее время разработки, наименьшая стоимость, наименьшее энергопотребление.

В **главе 5** вы увидите, что «наилучшая» схема для одной технологии не обязательно является наилучшей для другой. Например, мы использовали элементы И и ИЛИ, но для КМОП-технологии более эффективны элементы И-НЕ и ИЛИ-НЕ. С опытом, вы увидите, что для

большинства схем вы сможете находить хорошую многоуровневую реализацию, просто рассматривая эти схемы (и действуя по интуиции).

Некоторый опыт вы наработаете, изучая примеры схем остальной части книги. По мере того, как вы учитесь, исследуйте различные варианты разработки и думайте о компромиссах. Сейчас также доступны системы автоматизированного проектирования (САПР), которые позволяют рассматривать огромное пространство возможных многоуровневых реализаций (осуществлять поиск в многомерном пространстве решений) и находить такое, которое наилучшим образом удовлетворяет вашим критериям оптимальности с учетом имеющихся строительных блоков.

2.5.2 Перемещение инверсии

Как вы помните из **раздела 1.7.6** для КМОП-схем лучше подходят элементы И-НЕ и ИЛИ-НЕ, а не И и ИЛИ. Однако чтение уравнений многоуровневых схем с элементами И-НЕ и ИЛИ-НЕ может оказаться довольно трудным. На **Рис. 2.33** показан пример многоуровневой схемы, функция которой не очевидна непосредственно из схемы. Путем перемещения инверсии можно преобразовать подобные схемы так, что инверсия сократится, и функция может стать более понятной. Построенные на принципах из **раздела 2.3.3**, правила для перемещения инверсии таковы:

- Начинайте с выхода цепи и двигайтесь назад к входам.
- Переместите инверсию с общего выхода на входы так, чтобы вы могли читать выражение в терминах выхода (например, Y), а не инвертированного выхода $\overline{Y}$.
- Продвигаясь в обратном направлении, меняйте каждый элемент так, чтобы число инверсий оказалось четным и их можно было сократить. Если текущий элемент имеет входные отрицания, рисуйте предшествующий элемент с выходным отрицанием. Если текущий элемент не имеет входного отрицания, рисуйте предшествующий элемент без выходного отрицания.

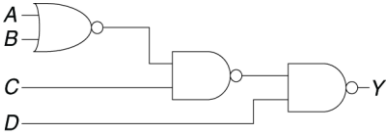


Рис. 2.33 **Многоуровневая схема на элементах И-НЕ и ИЛИ-НЕ**

Рис. 2.34 показывает, как перерисовать схему из **Рис. 2.33**, следуя изложенным правилам. Начинаем с выхода Y. Элемент И-НЕ имеет отрицание на выходе, которое мы хотим устранить. Мы переставляем выходное отрицание «назад», формируя элемент ИЛИ с инверсными

входами, показанный на **Рис. 2.34 (a)**. Двигаясь налево по схеме, мы замечаем, что самый правый элемент теперь имеет входное отрицание, которое может быть отброшено вместе с выходным отрицанием среднего элемента И-НЕ так, что инверсий в этом пути не останется, как показано на **Рис. 2.34 (b)**. Средний элемент не имеет входных инверсий, поэтому мы трансформируем самый левый элемент так, чтобы он не имел выходного отрицания, как показано на **Рис. 2.34 (c)**. Сейчас все отрицания в схеме убраны, за исключением входов, так что функция может быть прочитана в терминах элементов И и ИЛИ с действительными или комплементарными входами: $Y = \bar{A}\bar{B}C + \bar{D}$.

Чтобы подчеркнуть этот последний пункт, на **Рис. 2.35** показана схема, логически эквивалентная схеме на **Рис. 2.34**. Функции внутренних соединений отмечены синим цветом. Поскольку последовательные отрицания могут быть отброшены, мы можем игнорировать инверсии на выходе среднего и на входе самого правого элементов, получив логически эквивалентную схему на **Рис. 2.35**.

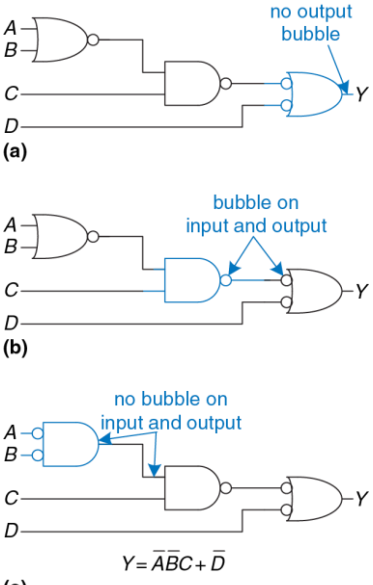


Рис. 2.34 **Схема с удаленными инверсиями**

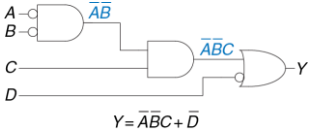


Рис. 2.35 **Логически эквивалентная схема**

Пример 2.8 ПЕРЕМЕЩЕНИЕ ИНВЕРСИИ В КМОП-ЛОГИКЕ

Большинство разработчиков думают в терминах элементов И и ИЛИ, но предположим, что вы хотели бы реализовать схему из **Рис. 2.36** в КМОП-логике, для которой предпочтительны элементы И-НЕ и ИЛИ-НЕ. Используйте перемещение инверсии, чтобы преобразовать схему в элементы И-НЕ, ИЛИ-НЕ и НЕ.

Решение: прямолинейное решение заключается в простой замене каждого элемента И на И-НЕ с инвертором, а каждого элемента ИЛИ – на ИЛИ-НЕ с инвертором, как это показано на **Рис. 2.37**. Такая схема потребует 8 элементов. Заметьте, что инверторы изображены с отрицанием на входе, а не на выходе, чтобы подчеркнуть, что последовательное двойное отрицание не меняет логику работы схемы и может быть отброшено.

Обратите внимание, что отрицания могут быть добавлены на выход элемента и на вход следующего элемента без изменения функции, как показано на **Рис. 2.38 (a)**. Выходной элемент И преобразовывается в элемент И-НЕ и инвертор, как показано на **Рис. 2.38 (b)**. Это решение требует только пять элементов.

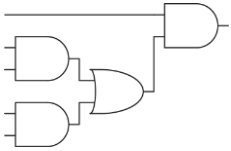


Рис. 2.36 Схема на элементах И и ИЛИ

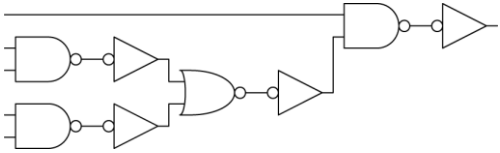


Рис. 2.37 Плохая схема на элементах И-НЕ и ИЛИ-НЕ

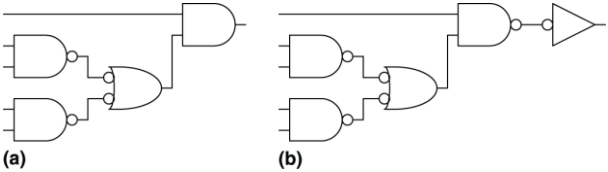


Рис. 2.38 Улучшенная схема на элементах И-НЕ и ИЛИ-НЕ

2.6 ЧТО ЗА X И Z?

Булева алгебра ограничена значениями 0 и 1. Однако реальные схемы могут также иметь недопустимое и плавающее состояния, представляемые символами X и Z соответственно.

2.6.1 Недопустимое значение: X

Символ X обозначает неизвестное логическое значение или недопустимое значение физического напряжения в соединении, не соответствующее уровням логических 0 и 1. Это обычно происходит, если к соединению подключены выходы других элементов схемы, выдающие значения 0 и 1 одновременно. На **Рис. 2.39** показан такой случай, когда выход *Y* подключен к элементам, имеющим на выходе ВЫСОКИЙ и НИЗКИЙ уровни.

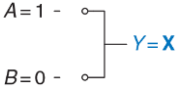


Рис. 2.39 Схема с недопустимым значением на выходе

Эта ситуация, называемая состязанием или конфликтом (contention), считается ошибкой, и её необходимо избегать. Реальное (физическое) напряжение на выходе с конфликтом может быть где-то между нулем и

напряжением питания, в зависимости от соотношения мощностей элементов, выдающих в цепь ВЫСОКОЕ и НИЗКОЕ напряжения. Часто, но не всегда, значение напряжения оказывается в «запрещенной» зоне. Состязание также может стать причиной повышенного потребления энергии конфликтующими элементами, в результате чего схема нагревается и может быть повреждена.

Значение X также иногда используется программами моделирования для обозначения неинициализированного значения. Например, если вы забыли определить входное значение, симулятор присвоит ему значение X для того, чтобы предупредить вас о проблеме.

Как уже упоминалось в **разделе 2.4**, разработчики цифровых схем также используют символ X для обозначения в таблицах истинности безразличных переменных, от которых не зависит состояние выходов. Не путайте эти два смысла. Когда X появляется в таблицах истинности, он показывает, что значение этой переменной может быть и нулем, и единицей. Когда X появляется в схеме, это означает, что цепь имеет неизвестное или запрещенное значение.

2.6.2 Третье состояние: Z

Символ Z указывает, что напряжение в цепи не определяется ни источником ВЫСОКОГО, ни источником НИЗКОГО напряжения. Говорят, что такая цепь отключена, находится в состоянии высокого импеданса или в третьем состоянии. Типично неправильное представление – это что неподключенная, или плавающая цепь имеет значение логического 0. В реальности логическое состояние неподключенной цепи может быть как 0, так и 1, а напряжение на ней может принять некое промежуточное значение в зависимости от истории изменения состояния системы. Неподключенная цепь не обязательно означает наличие ошибки в схеме. Например, какой-нибудь другой элемент схемы может задать цепи допустимый логический уровень именно в тот момент, когда эта цепь влияет на работу схемы.

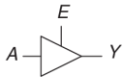
Один из распространенных способов получить неопределенное значение – это забыть подключить вход схемы к источнику напряжения логического уровня или предположить, что неподключенный вход – то же самое, что вход со значением 0. Эта ошибка может привести к тому, что поведение цепи будет хаотичным, так как неопределенные значения на входе могут случайно меняться из 0 в 1. Действительно, касания схемы может быть достаточно, чтобы привести к изменению из-за слабого статического электричества тела. Мы видели схему, которая

корректно работала, только до тех пор, пока студент держал палец на микросхеме.

Буфер с тремя состояниями, показанный на **Рис. 2.40**, имеет три возможных выходных значения: ВЫСОКОЕ (1), НИЗКОЕ (0) и отключенное или плавающее (Z) состояние (прим. переводчика: именно поэтому плавающее состояние называют третьим). Буфер с тремя состояниями имеет вход A, выход Y и сигнал управления E. Когда сигнал разрешения (управления) имеет значение ИСТИНА, буфер с тремя состояниями работает как простой буфер, передавая входное значение на выход. Когда сигнал управления имеет значение ЛОЖЬ, выход буфера переключается в третье состояние и становится плавающим (Z). Буфер с тремя состояниями на **Рис. 2.40** имеет активный высокий уровень. Это значит, что когда сигнал разрешения ВЫСОКИЙ (1), передача разрешена.

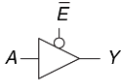
На **Рис. 2.41** показан Буфер с тремя состояниями с активным низким уровнем. Когда сигнал управления НИЗКИЙ (0), передача разрешена. Мы видим, что сигнал имеет активный низкий уровень из-за отрицания, поставленного на его входной цепи. Мы часто обозначаем вход с активным низким уровнем, рисуя черточку (символ отрицания) над его именем ($\bar{E}$), или добавляя букву "b" или "bar" после имени, *Eb* или *Ebar*.

Tristate Buffer



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Рис. 2.40 **Буфер с тремя состояниями**



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Рис. 2.41 **Буфер с тремя состояниями с активным низким уровнем**

Буферы с третьим состоянием обычно используются в шинах, соединяющих несколько микросхем. Например, микропроцессор, видеоконтроллер и Ethernet-контроллер могут нуждаться во взаимодействии с подсистемой памяти в персональном компьютере. Каждая микросхема может подключаться к общей шине памяти, используя буферы с третьим состоянием, как показано на **Рис. 2.42**. При этом только одна микросхема имеет право выставить свой сигнал разрешения, чтобы выдать значение на шину. Выходы других микросхем должны находиться в третьем состоянии, чтобы не стать причиной коллизии с микросхемой, осуществляющей обмен данными с

памятью. Однако, любая микросхема может читать информацию с общей шины в любое время. Такие шины на основе буферов с тремя состояниями когда-то были очень распространенными. Однако, в современных компьютерах высочайшие скорости возможны только при соединении микросхем друг с другом напрямую (point-to-point), а не с помощью общей шины.

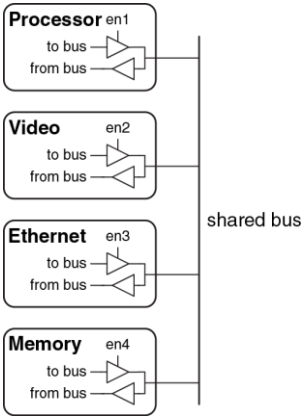


Рис. 2.42 Шина с третьим состоянием, соединяющая несколько микросхем

2.7 КАРТЫ КАРНО

После того, как Вы осуществите несколько преобразований по минимизации булевых уравнений, используя булеву алгебру, Вы поймете, что без соблюдения должной аккуратности, иногда можно получить решение, совершенно отличное от требуемого упрощенного уравнения. Карты Карно представляют собой наглядный метод для упрощения булевых уравнений. Они были изобретены в 1953-м году Морисом Карно, телекоммуникационным инженером из фирмы Bell Labs. Карты Карно очень удобны в случаях, когда уравнение содержит до четырёх переменных. Но, что более важно, они дают понимание сути при манипулировании логическими выражениями.

> Морис Карно родился в 1924 году. Получил степень бакалавра по физике в Городском колледже Нью-Йорка в 1948 году, а в 1952 получил степень доктора философии по физике (Ph.D., аналог степени кандидата наук) в Йельском университете.
>
> С 1952 по 1993 годы работал в Bell Labs и IBM. С 1980 по 1999 год являлся профессором информатики в Политехническом университете Нью-Йорка.

Как мы помним, логическая минимизация осуществляется путем склейки термов. Два терма, включающие в себя импликанту P и два логических значения некоторой переменной A, объединяются, при этом

переменная *A* исключается. Карты Карно позволяют легко находить термы, которые можно склеить, располагая их в виде таблицы.

На **Рис. 2.43** показана таблица истинности и карта Карно для функции трех переменных. Верхняя строка дает 4 возможных значения для переменных *A* и *B*. Левая колонка дает 2 возможных значения переменной *C*. Каждая клетка карты Карно соответствует строке таблицы истинности и содержит значение функции *Y* из этой строки. Например, верхняя левая клетка соответствует первой строке таблицы истинности и показывает, что значение функции *Y* будет равно 1, когда *ABC*=000. Как и каждая строка в таблице истинности, каждая клетка карты Карно представляет собой отдельный минтерм. Для лучшего понимания, на **Рис. 2.43 (c)** показаны минтермы, соответствующие каждой клетке карты Карно.

Каждая клетка, или минтерм, отличается от соседней изменением только одной переменной. Это значит, что соседние клетки различаются только в значении одного литерала, значение которого «истинно» в одной клетке и «ложно» в соседней. Например, клетки, представляющие минтермы $\overline{A}\,\overline{B}\,\overline{C}$ и $\overline{A}\,\overline{B}\,C$ – соседние и различается только в переменной *C*. Вы, наверное, также отметили, что переменные *A* и *B* комбинируются в верхней строке в особом порядке: 00, 01, 11, 10. Этот порядок называется *кодом Грея* (Gray code). В отличие от битового порядка по возрастанию величины (00, 01, 10, 11), в коде Грея

соседние записи отличаются только на один разряд. Например, 01 : 11 отличается только изменением *A* с 0 на 1, тогда как 01 : 10 требует изменения *A* из 1 в 0 и *B* из 0 в 1. Таким образом, обычный последовательный побитный порядок не дает требуемого нам свойства соседних ячеек, который должны различаться только в одной переменной.

> Код Грея был запатентован Фрэнком Греем, исследователем из Bell Labs, в 1953 году (патент США номер 2,632,058). Этот код особенно полезен для электромеханических преобразователей (например, датчиков угла поворота – прим. переводчика), так как он позволяет избавиться от ложных срабатываний. Код грея может быть любой разрядности. Например, трехбитный код Грея выглядит так: 000, 001, 011, 010, 110, 111, 101, 100.
>
> Льюис Кэрролл опубликовал похожую загадку в журнале Vanity Fair в 1879 году.«Правила просты. Даны два слова одинаковой длины. Нужно соединить их цепочкой слов, в которой два соседних слова отличаются лишь одной буквой» – написал он.
>
> Например, слово SHIP можно превратить в слово DOCK так: **SHIP**, SLIP, SLOP, SLOT, SOOT, LOOT, LOOK, LOCK, **DOCK**. Можете ли вы найти более короткую цепочку?

Карты Карно так же «закольцованы». Клетка с самого правого края таблицы является соседней с самой левой, так как они отличаются

только в одной переменной (A). Можно свернуть карту в цилиндр, соединив края, и даже в этом случае соседние клетки также будут отличаться только в одной переменной.

2.7.1 Думайте об овалах

На карте Карно на **Рис. 2.43** содержится только две единицы, что соответствует числу минтермов в уравнении ($\overline{A}\,\overline{B}\,\overline{C}$ и $\overline{A}\,\overline{B}\,C$). Чтение минтермов из карт Карно в точности соответствует чтению дизъюнктивной нормальной формы (ДНФ) из таблицы истинности.

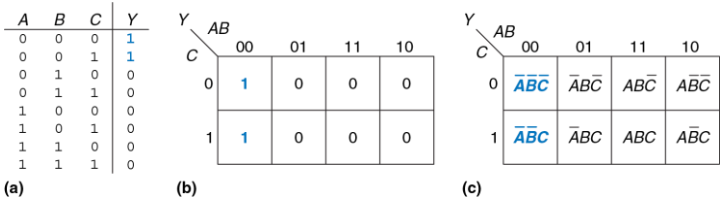


Рис. 2.43 **Функция трех переменных: таблица истинности (a), карта Карно (b), карта Карно с минтермами (c)**

Как и раньше, мы могли бы использовать булеву алгебру для минимизации:

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C = \overline{A}\,\overline{B}\,(\overline{C} + C) = \overline{A}\,\overline{B} \qquad (2.7)$$

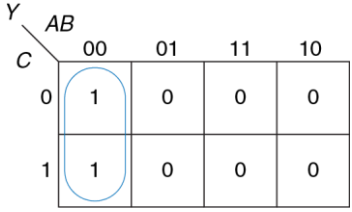


Рис. 2.44 Минимизация при помощи карты Карно

Карты Карно помогают нам делать это упрощение графически, обводя единицы в соседних клетках овалами, как показано на **Рис. 2.44**. Для каждого овала мы пишем соответствующую ему импликанту. Вспомните из **раздела 2.2**, что импликанта является произведением одного или нескольких литералов. Переменные, для которых прямая и комплементарная формы попадают в один овал, исключаются из импликанты. В нашем случае обе формы переменной *C* попадают в овал, так что мы не включаем ее в импликанту. Другими словами, *Y* = ИСТИНА когда *A* = *B* = 0 вне зависимости от *C*. Так что импликантой

будет $\overline{A}\,\overline{B}$: карта Карно дает тот же самый ответ, какой мы получили, используя булеву алгебру.

2.7.2 Логическая минимизация на картах Карно

Карты Карно обеспечивают простой визуальный способ минимизации логических выражений. Просто обведите все прямоугольные блоки с единицами на карте, используя наименьшее возможное число овалов. Каждый овал должен быть максимально большим. Затем прочитайте все импликанты, которые обведены.

Напомним, что формально уравнения булевой алгебры являются минимальными, только когда записаны как сумма наименьшего числа первичных импликант. Каждый овал на карте Карно представляет собой импликанту. Максимально возможный овал является первичной импликантой.

Например, на карте Карно на **Рис. 2.44** $\overline{A}\overline{B}\overline{C}$ и $\overline{A}\overline{B}C$ импликанты, но не первичные. На этой карте только $\overline{A}\,\overline{B}$ является первичной импликантой. Правила для нахождения минимального уравнения из карт Карно следующие:

- Использовать меньше всего овалов, необходимых для покрытия всех 1;

- Все клетки в каждом овале обязаны содержать 1;
- Каждый овал должен охватывать блок, число клеток которого в каждом направлении равно степени двойки (то есть 1, 2 или 4);
- Каждый овал должен быть настолько большим, насколько это возможно;
- Овал может связывать края карты Карно;
- Единица на карте Карно может быть обведена сколько угодно раз, если это позволяет уменьшить число овалов, которые будут использоваться.

Пример 2.9 МИНИМИЗАЦИЯ ФУНКЦИИ ТРЕХ ПЕРЕМЕННЫХ ПРИ ПОМОЩИ КАРТЫ КАРНО

Предположим, у нас есть функция $Y = F(A, B, C)$ с картой Карно, показанной на **Рис. 2.45**. Упростим это выражение, используя карту Карно.

Решение: Обведем единицы на карте Карно, используя наименьшее возможное количество овалов, как показано на **Рис. 2.46**. Каждый овал на карте Карно представляет собой первичную импликанту, а его размер кратен степени двойки (2 × 1 и 2 × 2).

Мы сформируем первичную импликанту для каждого выделенного овала, выписывая только те переменные, которые появляются в нем только в прямой или в комплементарной формах. Например, овал размером 2 × 1 включает в

себя прямую и комплементарную формы переменной B, так что мы не включаем B в первичную импликанту. Однако, в этом овале есть только прямая форма переменной A ($\overline{A}$) и комплементарная форма переменной C ($\overline{C}$), так что мы включаем эти переменные в первичную импликанту A ($\overline{A}$). Подобным же образом овал размером 2 × 2 покрывает все клетки, где B = 0, так что первичная импликанта будет $\overline{B}$.

Обратите внимание, что правая верхняя клетка (минтерм) используется дважды, чтобы сделать овалы первичных импликант как можно большими. Как мы видели в булевой алгебре, это эквивалентно совместному использованию минтерма для уменьшения размера импликанты. Также обратите внимание на то, что овал, покрывающий четыре клетки, оборачивается через края карты Карно.

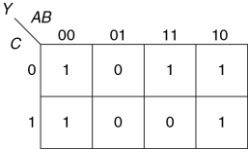


Рис. 2.45 **Карта Карно для примера 2.9**

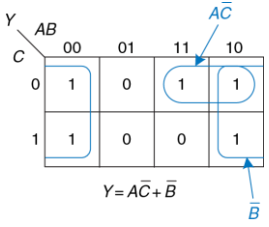


Рис. 2.46 **Решение примера 2.9**

Пример 2.10 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА

Дешифратор семисегментного индикатора получает на вход четырехбитные данные $D[3:0]$ и формирует семь выходов для управления светодиодами для показа цифр от 0 до 9. Семь выходов часто называют сегментами от a до g, или S_a–S_g, как показано на **Рис. 2.47**. Сами цифры показаны на **Рис. 2.48**. Составим таблицу истинности для выходов и используем карты Карно для нахождения логического уравнения для выходов S_a и S_b. При этом предположим, что запрещенные входные значения (10-15) ничего не выводят на индикатор.

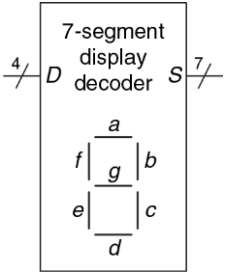


Рис. 2.47 **Семисегментный индикатор**

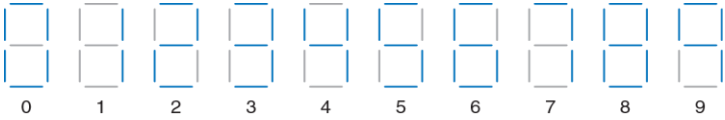


Рис. 2.48 Цифры на семисегментном индикаторе

Решение: Таблица истинности дана в **Табл. 2.6**. Например, вход 0000 должен включать все сегменты, за исключением S_g.

Табл. 2.6 Таблица истинности дешифратора семисегментного индикатора

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
Прочие	0	0	0	0	0	0	0

Каждый из семи выходов является независимой функцией от четырех переменных. Карты Карно для выходов S_a и S_b показаны на **Рис. 2.49**. Помните, что соседние клетки могут отличаться только одной переменной, так что мы промаркируем строки и столбцы в коде Грея: 00, 01, 11, 10. Будьте осторожны и помните этот порядок, когда будете вписывать значения выходов в клетки.

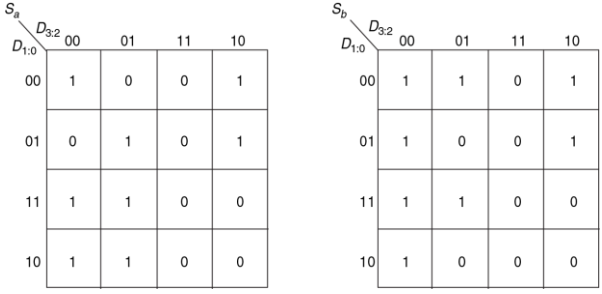


Рис. 2.49 Карты Карно для S_a и S_b

Затем обведем первичные импликанты. При этом используем минимально необходимое количество овалов для покрытия всех единиц. Овалы могут связывать края (вертикальные и горизонтальные), а каждая единица может быть выделена несколько раз. На **Рис. 2.50** показаны первичные импликанты и упрощенные логические уравнения.

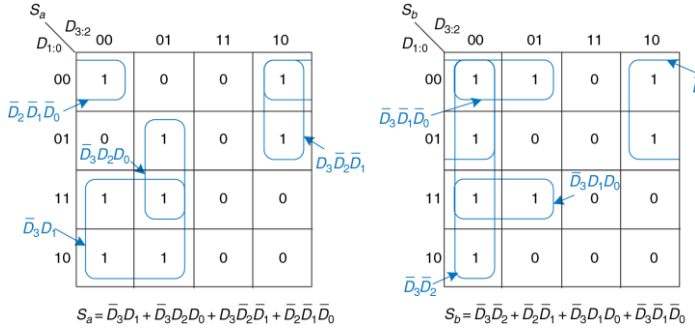


Рис. 2.50 Решение упражнения 2.10

Заметьте, что минимальный набор первичных импликант – не единственно возможный. Например, запись 0000 на карте Карно для S_a может быть выделена вместе с записью 1000, получая минтерм $\bar{D}_2\bar{D}_1\bar{D}_0$. Но вместо этого овал может включать в себя запись 0010, получая минтерм $\bar{D}_3\bar{D}_2\bar{D}_0$, как показано пунктирной линией на **Рис. 2.51**.

Рис. 2.52 иллюстрирует распространенную ошибку, когда не первичная импликанта выбирается для покрытия 1 в левом верхнем углу. Этот минтерм $\bar{D}_3\bar{D}_2\bar{D}_1\bar{D}_0$ дает дизъюнкцию конъюнкций (сумму произведений), которая не минимизирована. Его можно было бы скомбинировать с любым из двух соседних

минтермов для получения овала большего размера, как было сделано на предыдущих двух рисунках.

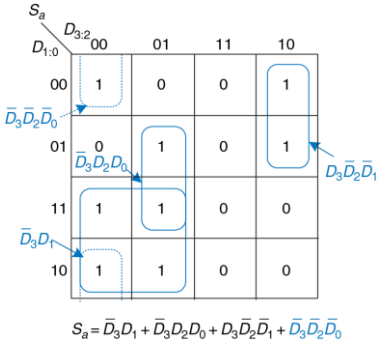


Рис. 2.51 **Альтернативная карта Карно для S_a, использующая другой набор первичных импликант**

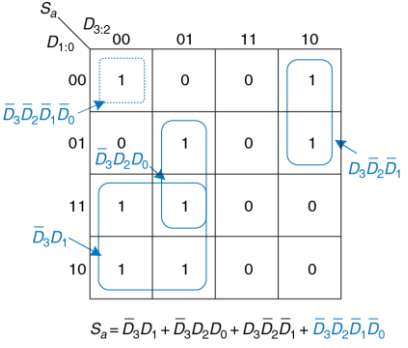


Рис. 2.52 **Карта Карно для S_a, использующая некорректную импликанту**

2.7.3 Безразличные переменные

Вспомните, что безразличные переменные в таблице истинности были введены в **разделе 2.4** для уменьшения числа ее строк в тех случаях, когда соответствующие переменные не влияют на выход. Они обозначаются символом X, который означает, что значение входной переменной может быть или 0, или 1.

Не только входы, но и выходы могут быть безразличными, если состояние выхода не важно или соответствующая комбинация входов никогда не возникает. Такие выходы могут трактоваться или как 0, или как 1, в зависимости от того, как решит разработчик.

В картах Карно безразличные переменные позволяют провести еще большую логическую минимизацию. Их можно включать в овалы, если это помогает покрыть единицы или меньшим количеством овалом, или овалами, большими по размеру, но их можно и не покрывать, если это не помогает минимизации.

Пример 2.11 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА С БЕЗРАЗЛИЧНЫМИ ПЕРЕМЕННЫМИ

Повторим пример 2.10 для случая, когда нас не интересуют значения выходов при запрещенных входных значениях от 10 до 15.

Решение: Карта Карно с безразличными элементами, отмеченными как «X», представлена на **Рис. 2.53**. Поскольку такие элементы могут быть равны как 0, так и 1, мы используем их там, где это поможет покрыть единицы или меньшим количеством овалов, или овалами, большими по размеру. Обведенные значения X трактуются как 1, не обведенные – как 0. Посмотрите, как для сегмента S_a можно выделить овал размером 2 × 2, объединяющий все четыре угла. Используйте клетки с безразличными значениями для упрощения логики.

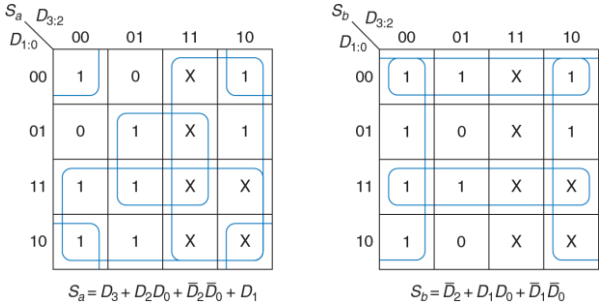


Рис. 2.53 **Карта Карно с безразличными переменными**

2.7.4 Подводя итоги

Булева алгебра и карты Карно – два метода логического упрощения. В конечном счете, целью является нахождение наименее затратного метода реализации конкретной логической функции.

В современной инженерной практике компьютерные программы, называемые синтезаторами логики (logic synthesizers), проводят упрощение схем по описанию логический функций, как мы увидим в **главе 4**. Для больших задач программы логического синтеза намного эффективнее людей. Для маленьких же задач человек с некоторым опытом может найти хорошее решение «на глаз». Никто из авторов книги, тем не менее, никогда не использовал карты Карно в реальной жизни для решения практических задач. Но понимание принципов, лежащих в основе карт Карно, крайне важно. Кроме того, знание карт Карно часто спрашивают на собеседованиях!

2.8 БАЗОВЫЕ КОМБИНАЦИОННЫЕ БЛОКИ

Комбинационные логические элементы часто группируются в «строительные блоки», используемые для создания сложных систем. Это позволяет абстрагироваться от излишней детализации уровня логических элементов и подчеркнуть функцию «строительного блока». Мы уже изучили три таких блока: полный сумматор (см. **раздел 2.1**), схемы приоритета (см. **раздел 2.4**) и дешифратор семисегментного индикатора (см. **раздел 2.7**). Этот раздел представляет два типа блоков, еще более часто используемых при проектировании: мультиплексоры и дешифраторы. В **главе 5** будет рассказано и о других комбинационных «строительных блоках».

2.8.1 Мультиплексоры

Мультиплексоры являются одними из наиболее часто используемых комбинационных схем. Они позволяют выбрать одно выходное значение из нескольких входных в зависимости от значения сигнала выбора.

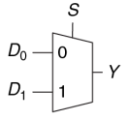
Двухвходовой мультиплексор (2:1)

На **Рис. 2.54** показано условное графическое обозначение и таблица истинности для двухвходового мультиплексора (2:1) с двумя входами данных D_0 и D_1, входом выбора S и одним выходом Y. Мультиплексор

передает на выход один из двух входных сигналов данных, основываясь на сигнале выбора: если $S = 0$, выход $Y = D_0$, и если $S = 1$, то выход $Y = D_1$. S также называют управляющим сигналом, так как он управляет поведением мультиплексора.

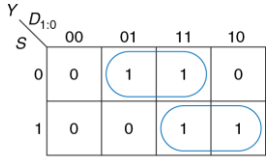
Двухвходовой мультиплексор может быть построен с использованием дизъюнкции конъюнкций (суммы произведений), как показано на **Рис. 2.55**. Логическое выражение для него может быть получено с помощью карт Карно или составлено на основе описания ($Y = 1$ если $S = 0$ И $D_0 = 1$ ИЛИ если $S = 1$ И $D_1 = 1$).

Мультиплексор также может быть построен на буферах с третьим состоянием, как показано на **Рис. 2.56**. Сигналы разрешения буферов с третьим состоянием организованы так, что все время активен только один буфер. Когда $S = 0$, то включен только элемент T0, позволяющий сигналу D_0 передаваться на выход Y. Когда $S = 1$, то активен только элемент T1, передавая на выход сигнал D_1.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Рис. 2.54 Условное обозначение и таблица истинности двухвходового мультиплексора



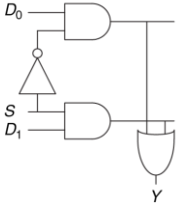


Рис. 2.55 Реализация двухвходового мультиплексора с использованием двухуровневой логики

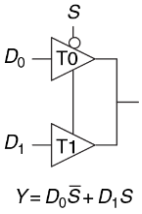


Рис. 2.56 **Мультиплексор на буферах с тремя состояниями**

Строго говоря, соединение двух выходов логических элементов нарушает правила построения комбинационных схем, описанные в **Разделе 2.1**. Однако в этом конкретном случае в любой момент времени только один из этих элементов может подавать сигнал на выход Y, так что такое исключение из правил допустимо.

Многовходовые мультиплексоры

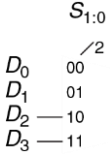


Рис. 2.57
Четырехвходовой мультиплексор

Четырехвходовой мультиплексор (4:1) имеет четыре входа данных и один выход, как показано на **Рис. 2.57**. Для выбора одного из четырех входов данных требуется двухразрядный управляющий сигнал. Четырехвходовой мультиплексор может быть построен с использованием дизъюнкции конъюнкций (суммы произведений), буферов с тремя состояниями или двухвходовых мультиплексоров, как показано на **Рис. 2.58**.

Конъюнкции, подключенные к сигналам разрешения работы буферов с тремя состояниями, могут быть построены с использованием элементов И и инверторов. Они также могут быть сформированы дешифратором, который мы рассмотрим в **разделе 2.8.2**.

Мультиплексоры с большим числом входов, например восьмивходовые или шестнадцативходовые, могут быть построены простым масштабированием методов, показанных на **Рис. 2.58**. В общем случае, мультиплексор N:1 требует $\log_2 N$ управляющих сигналов. Выбор наилучшей реализации, как и прежде, зависит от используемой технологии.

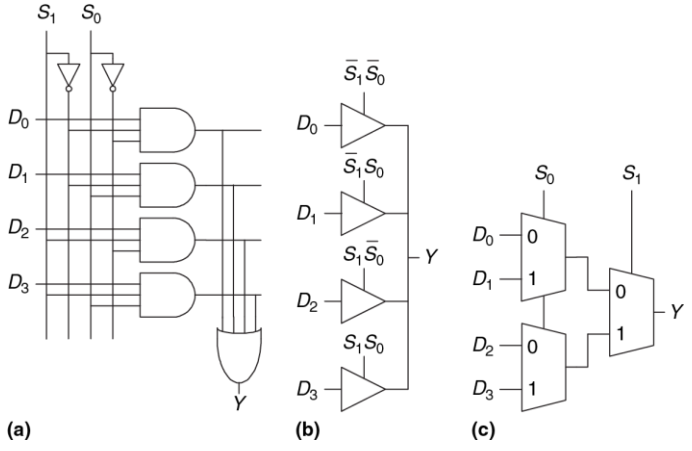


Рис. 2.58 **Реализация четырехвходового мультиплексора: двухуровневая логика (a), буфера с тремя состояниями (b), иерархическая (c)**

Логика на мультиплексорах

Мультиплексоры могут использоваться как таблицы преобразования (lookup tables) для выполнения логических функций. На **Рис. 2.59** показан четырехвходовой мультиплексор, используемый для реализации двухвходового элемента И.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

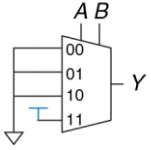


Рис. 2.59 Получение двухвходового элемента И из четырехвходового мультиплексора

Входы *A* и *B* служат управляющими линиями. Входы данных мультиплексора подключены к 0 и 1 согласно соответствующей строке таблицы истинности. Вообще, 2^N-входовой мультиплексор можно запрограммировать для выполнения любой *N*-входовой логической функции, используя 0 и 1 для соответствующих входов данных. Действительно, изменением входных данных мультиплексор может быть перепрограммирован для выполнения различных функций.

Немного смекалки, и мы сможем уменьшить размер мультиплексора наполовину, используя только 2^{N-1}-входовой мультиплексор для выполнения любой *N*-входовой логической функции. Способ заключается в том, чтобы подать один из литералов, так же как 0 и 1, на вход данных мультиплексора.

Для иллюстрации этого принципа на **Рис. 2.60** показаны функции двухвходовых элементов И и ИСКЛЮЧАЮЩЕЕ ИЛИ, реализованных на двухвходовых мультиплексорах. Мы начали с обычной таблицы истинности и затем скомбинировали пары строк, чтобы исключить самую правую входную переменную (*B*), и выразить выход в термах этой переменной. Например, в случае элемента И, когда $A = 0$, то $Y = 0$ вне зависимости от *B*. Когда $A = 1$, то $Y = 0$, если $B = 0$, и $Y = 1$, если $B = 1$, так что $Y = B$. Затем мы используем мультиплексор как таблицу подстановки согласно этой новой уменьшенной таблице истинности.

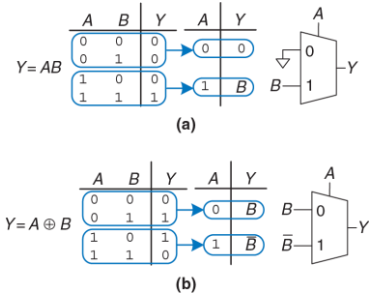


Рис. 2.60 **Реализация логических функций на мультиплексорах**

Пример 2.12 ЛОГИКА С МУЛЬТИПЛЕКСОРАМИ

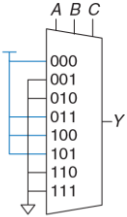
Алисе Хакер необходимо реализовать функцию $Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$ для завершения ее курсового проекта. Когда она посмотрела, какие микросхемы доступны ей в лаборатории, то увидела, что там остался только восьмивходовой мультиплексор. Как ей реализовать эту функцию?

Решение: На **Рис. 2.61** показана схема, разработанная Алисой с использованием одного восьмивходового мультиплексора. Этот мультиплексор выступает в роли таблицы преобразования, где каждая строка таблицы истинности соответствует входу мультиплексора.

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$

(a)



(b)

Рис. 2.61 **Схема Алисы: таблица истинности (a), реализация на восьмивходовом мультиплексоре (b)**

Пример 2.13 ЛОГИКА С МУЛЬТИПЛЕКСОРАМИ, ПОВТОРЕНИЕ

Алиса еще раз включила свою схему перед защитой проекта и сожгла единственный восьмивходовой мультиплексор (она случайно подала напряжение 20 В вместо 5 В после бессонной ночи).

Теперь она просит у своих друзей запасные элементы, и ей дают четырехвходовой мультиплексор и инвертор. Сможет ли она собрать свою схему, используя только эти элементы?

Решение: Алиса уменьшила свою таблицу истинности до четырех строк, сделав выход зависящим от *C*. (Она могла бы также исключить любой из двух других столбцов таблицы истинности, сделав выход зависимым от *A* или *B*). Новая схема показана на **Рис. 2.62**.

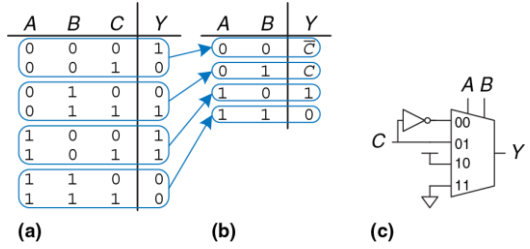
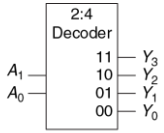


Рис. 2.62 **Новая схема Алисы**

2.8.2 Дешифраторы

В общем случае у дешифратора имеется N входов и 2^N выходов. Он выдает единицу строго на один из выходов в зависимости от набора входных значений. На **Рис. 2.63** показан дешифратор 2:4. Когда $A[1:0]$ = 00, Y_0 = 1. Когда $A[1:0]$ = 01, Y_1 = 1 и так далее. Выходы образуют прямой унитарный код (one-hot code), называемый так потому, что в любое время только один из выходов может принимать высокий уровень.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Рис. 2.63 **Дешифратор 2:4**

Пример 2.14 РЕАЛИЗАЦИЯ ДЕШИФРАТОРА

Реализуйте дешифратор 2:4 на элементах И, ИЛИ и НЕ.

Решение: На **Рис. 2.64** показана реализация дешифратора 2:4, использующая 4 элемента И. Каждый элемент зависит или от действительной, или от комплементарной формы каждого входа. Вообще, дешифратор $N{:}2^N$ может быть построен из 2^N N-входовых элементов И, к которым подходят различные комбинации действительных и комплементарных входов. Каждый выход в дешифраторе представляет собой одиночный минтерм. Например, Y_0 представляет минтерм $\overline{A}_1\overline{A}_0$. Это обстоятельство будет удобно при использовании дешифратора с другими цифровыми базовыми блоками.

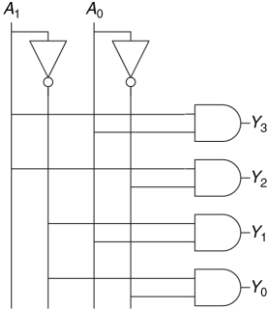


Рис. 2.64 **Реализация дешифратора 2:4**

Логика на дешифраторах

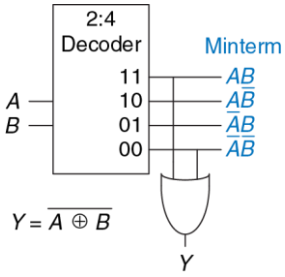


Рис. 2.65 **Реализация логической функции на дешифраторе**

Дешифратор может комбинироваться с элементами ИЛИ для построения логических функций. На **Рис. 2.65** показана двухвходовая функция ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ (XNOR), использующая дешифратор 2:4 и один элемент ИЛИ. Поскольку каждый выход дешифратора представляет одиночный минтерм, функция построена как логическое ИЛИ всех минтермов в этой функции. На **Рис. 2.65** показана функция.

При использовании дешифраторов для реализации логических функций, проще всего выразить функцию таблицей истинности или записать ее в дизъюнктивной нормальной форме. *N*-входовая функция, имеющая *M* единиц в таблице истинности, может быть построена с использованием $N{:}2^N$ дешифратора и *M*-входового элемента ИЛИ, подключенным ко всем минтермам, содержащим единицу в таблице истинности. Эта идея будет применена для создания постоянного запоминающего устройства (ПЗУ) в **разделе 5.5.6**.

2.9 ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ

В предыдущих разделах мы концентрировались в первую очередь на работе схемы, в идеале использующей наименьшее число элементов. Однако, как подтвердит любой опытный разработчик, одна из самых сложных задач в разработке схем – это учет всех ограничений, накладываемых на временные характеристики работы схемы, ведь хорошая схема должна работать предельно быстро и при этом без сбоев.

Изменение выходного значения в ответ на изменение входа занимает время. На **Рис. 2.66** показана задержка между изменением входа буфера и последующим изменением его выхода. Этот рисунок называется временной диаграммой; он изображает переходную характеристику схемы буфера при изменении входа. Переход от НИЗКОГО уровня к ВЫСОКОМУ называется положительным перепадом или фронтом. Аналогично, переход от ВЫСОКОГО уровня к НИЗКОМУ (на рисунке не показан) называется соответственно отрицательным перепадом или срезом. Синяя стрелка показывает, что положительный фронт сигнала Y вызывается положительным фронтом сигнала *A*. Величина задержки измеряется от момента времени, когда входной сигнал *A* достигает уровня 50%, до момента достижения уровня 50% выходным сигналом *Y*. Уровень 50% – это точка, в которой

сигнал находится ровно посередине между НИЗКИМ и ВЫСОКИМ логическими уровнями.

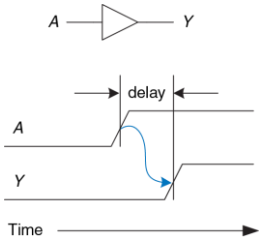


Рис. 2.66 **Задержка схемы**

2.9.1 Задержка распространения и задержка реакции

> Когда разработчики говорят о задержке схемы, они в большинстве случаев имеют в виду наибольшее возможное значение задержки (задержку распространения), если только из контекста не следует другое.

Комбинационная логика характеризуется задержкой распространения (propagation delay) и задержкой реакции, или отклика (contamination delay). Задержка распространения t_{pd} – это максимальное время от начала изменения входа до момента, когда все выходы достигнут

установившихся значений. Задержка реакции t_{cd} – это минимальное время от момента, когда вход изменился, до момента, когда любой из выходов начнет изменять свое значение.

На **Рис. 2.67** синим и серым цветом показаны соответственно задержки распространения и задержка реакции буфера. На рисунке показано, что вход *A* изначально имел или ВЫСОКОЕ, или НИЗКОЕ значение, и оно изменяется на противоположное в определенный момент времени; нас интересует только факт, что оно (значение *A*) изменилось, но не его конкретное значение. В ответ, спустя некоторое время, меняется *Y*. Стрелки показывают, что *Y* может начать меняться через временной интервал t_{cd} после изменения *A*, и что *Y* точно установится в новое значение не позднее, чем через интервал t_{pd}.

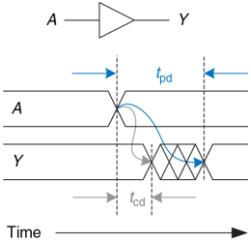


Рис. 2.67 Задержка распространения и задержка реакции

Основные причины задержек в схемах заключаются во времени, требуемом для перезарядки емкостей цепи, а так же в конечной скорости распространения электромагнитных волн в среде. Величины t_{pd} и t_{cd} могут различаться по многим причинам, включающим в себя:

- Разные задержки нарастания и спада сигнала;
- Несколько входов и выходов, одни из которых быстрее чем другие;
- Замедление работы схемы при повышении температуры и ускорение при охлаждении.

Вычисление t_{pd} и t_{cd} требует вникания в нижние уровни абстракций, что выходит за рамки этой книги. Однако, производители обычно предоставляют документацию со спецификацией этих задержек для каждого элемента.

> Задержки в схемах обычно составляют от нескольких пикосекунд (1 пс = 10^{-12} с) до нескольких наносекунд (1 нс = 10^{-9} с). За то время, что вы читали это замечание, прошло несколько триллионов пикосекунд.

Наряду с уже перечисленными факторами, задержки распространения и реакции также определяются *путем*, который проходит сигнал от входа до выхода. На **Рис. 2.68** показана четырехвходовая схема. *Критический путь* (critical path), выделенный синим – это путь от входа *A* или *B* до выхода *Y*. Он соответствует цепи с наибольшей задержкой и

является самым медленным, поскольку входному сигналу нужно пройти три элемента до выхода. Этот путь критический потому, что он ограничивает скорость, с которой работает схема. Самый короткий путь в схеме, показанный серым – путь от входа *D* до выхода *Y*. Это кратчайший и, следовательно, самый быстрый путь в схеме, т.к. входному сигналу до выхода нужно пройти только через один элемент.

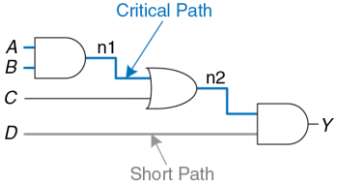


Рис. 2.68 Кратчайшая цепь и цепь с наибольшей задержкой

Задержка распространения комбинационной схемы – это сумма задержек распространения всех элементов в критическом пути. Задержка реакции – сумма задержек реакции всех элементов в кратчайшем пути. Эти задержки показаны на **Рис. 2.69** и описаны следующими уравнениями:

$$t_{pd} = 2t_{pd\_AND} + t_{pd\_OR} \tag{2.8}$$

$$t_{cd} = t_{cd\_AND} \tag{2.9}$$

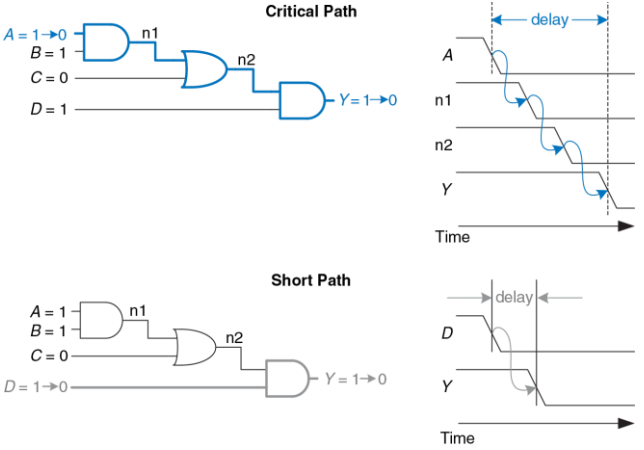


Рис. 2.69 **Временные диаграммы для кратчайшей цепи и цепи с наибольшей задержкой**

Несмотря на то, что мы проигнорировали задержку распространения сигналов по проводам, цифровые схемы на сегодняшний день настолько быстры, что эта задержка может превышать задержку в логических элементах. Связанная со скоростью света задержка распространения сигналов в проводах будет рассмотрена ниже (см. **Приложение А**).

Пример 2.15 НАХОЖДЕНИЕ ЗАДЕРЖЕК

Бену надо найти задержки распространения и отклика схемы, показанной на **Рис. 2.70**. Согласно справочнику каждый элемент имеет задержку распространения 100 пикосекунд (пс) и задержку отклика 60 пс.

Решение: Бен начал с нахождения критического и кратчайшего путей в схеме. Критический путь, выделенный на **Рис. 2.71** синим – это путь от входа *A* или *B* через три элемента до выхода *Y*. Следовательно, t_{pd} – это утроенная задержка распространения одиночного элемента или 300 пс.

Кратчайший путь, выделенный на **Рис. 2.72** серым – это путь от входов *C*, *D* или *E* через два элемента до выхода *Y*. В кратчайшем пути только два элемента, так что t_{cd} равно 120 пс.

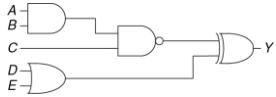


Рис. 2.70 Схема Бена

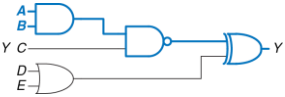


Рис. 2.71 Цепь с наибольшей задержкой

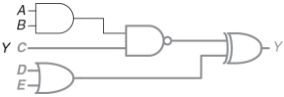


Рис. 2.72 Кратчайшая цепь

Пример 2.16 ВРЕМЕННЫЕ ХАРАКТЕРИСТИКИ МУЛЬТИПЛЕКСОРА: СРАВНЕНИЕ КРИТИЧЕСКИХ ПУТЕЙ

Сравните наихудшие временные характеристики каждой из трех реализаций четырехвходового мультиплексора, показанных на **Рис. 2.58** в **разделе 2.8.1**. Задержки распространения для компонентов перечислены в **Табл. 2.7**. Каким будет критический путь для каждой реализации? Исходя из анализа временных характеристик, какую схему вы предпочтете другим и почему?

Решение: Один из критических путей для каждого из трех вариантов выделен синим на **Рис. 2.73** и **Рис. 2.74**. t_{pd_sy} показывает задержку распространения от управляющего входа S до выхода Y; t_{pd_dy} – от входа данных до выхода Y; t_{pd} – худшее из двух: $\max(t_{pd\_sy}, t_{pd\_dy})$.

Как для двухуровневой логики, так и для реализации на буферах с третьим состоянием, на **Рис. 2.73** критическим является путь от одного из сигналов управления S до выхода Y: $t_{pd} = t_{pd\_sy}$. Эта схема критическая по управлению, поскольку критический путь идет от управляющих сигналов до выхода. Любая дополнительная задержка в сигналах управления добавится непосредственно в

наихудшую задержку. Задержка от *D* до *Y* на **Рис. 2.73 (b)** – всего 50 пс по сравнению с задержкой от *S* до *Y* в 125 пс.

На **Рис. 2.74** показана иерархическая реализация мультиплексора 4:1, использующая два каскада мультиплексоров 2:1. Критический путь в ней от любого входа данных *D* до выхода. Эта схема критическая по данным, поскольку критический путь идет от входа данных до выхода: $t_{pd} = t_{pd\_dy}$.

Если данные приходят на входы задолго до управляющих сигналов, мы должны предпочесть схему с кратчайшей задержкой от управления до выхода (иерархическая схема на **Рис. 2.74**). Аналогично, если управляющие сигналы приходят намного раньше входных данных, мы должны предпочесть схему с кратчайшей задержкой от данных до выхода (реализация на буферах с третьим состоянием на **Рис. 2.73 (b)**.

Наилучший выбор будет зависеть не только от цепи с наибольшей задержкой, но и от потребляемой электроэнергии, стоимости и наличия компонентов.

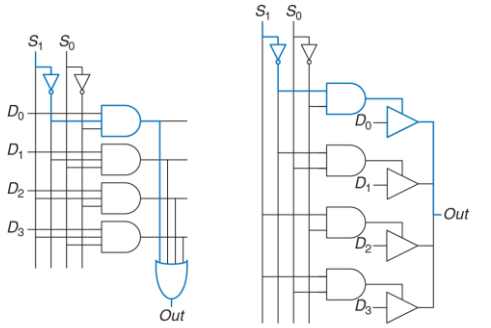
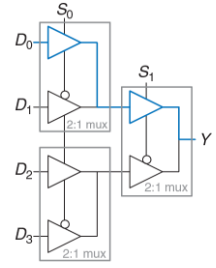


Рис. 2.73 Задержки распространения в четырехвходовом мультиплексоре: двухуровневая логика (a), буфера с тремя состояниями (b)



$t_{pd\_s0y} = t_{pd\_TRLSY} + t_{pd\_TRI\_AY} = \mathbf{85\ ns}$

$t_{pd\_dy} = 2\ t_{pd\_TRI\_AY} = \mathbf{100\ ns}$

Рис. 2.74 Задержки распространения в четырехвходовом мультиплексоре, построенном из двухвходовых

Табл. 2.7 Временные характеристики элементов в схемах мультиплексоров

Элемент	t_{pd} (пс)
НЕ	30
Двухвходовой И	60
Трехвходовой И	80
Четырехвходовой ИЛИ	90
Буфер с тремя состояниями (от *A* до *Y*)	50
Буфер с тремя состояниями (от *E* до *Y*)	35

2.9.2 Импульсные помехи

До сих по мы обсуждали случай, когда одиночное изменение входного сигнала вызывает одиночное изменение выхода. Однако может оказаться, что одиночное изменение на входе вызывает несколько выходных изменений. Это называется *импульсной помехой* или *паразитным импульсом*. Хотя паразитный импульс обычно не вызывает проблем, важно понимать, что он есть, и уметь распознавать его на временных диаграммах. На **Рис. 2.75** показана схема, подверженная паразитным импульсам, и карта Карно для нее.

Логическое уравнение минимизировано корректно, однако посмотрите, что происходит, когда $A = 0$, $C = 1$ и B меняется из 1 в 0. **Рис. 2.76** иллюстрирует этот сценарий. Короткий путь (показан серым) проходит через два элемента: И и ИЛИ. Критический путь (показан синим) проходит через инвертор и два элемента: И и ИЛИ.

Как только B переключится из 1 в 0, n2 (в коротком пути) опустится в 0 до того, как n1 (в критическом пути) сможет установиться в 1. До подъема n1 оба входа элемента ИЛИ будут принимать значение 0, и его выход сбросится в 0. Когда n1 в конце концов поднимется, Y вернется в 1. Как показано на временных диаграммах на **Рис. 2.76**, Y начинается с 1 и заканчивается 1, но на короткое время переключается в 0.

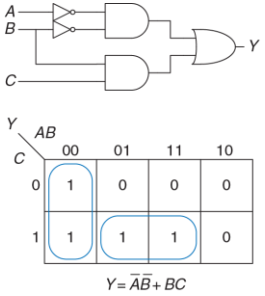


Рис. 2.75 **Схема, подверженная импульсным помехам**

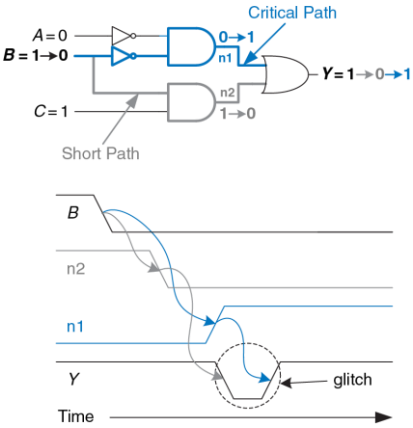


Рис. 2.76 **Временная диаграмма импульсной помехи**

До тех пор, пока мы выдерживаем интервал равный времени задержки распространения, прежде чем использовать значение с выхода, импульсная помеха не представляет проблемы, потому что выход в конце концов установится в правильное значение.

При желании мы можем избежать этого импульса добавлением дополнительного элемента в схему. Это проще понять в термах карты Карно.

На **Рис. 2.77** показано, как изменение входа B при переходе из $ABC = 001$ в $ABC = 011$ приводит к переходу от одной первичной импликанты к другой. Переход через границу двух первичных импликант в карте Карно свидетельствует о возможном появлении импульсной помехи.

Как мы видели на временных диаграммах на **Рис. 2.76**, если схема реализации одной первичной импликанты выключается до того, как может включиться схема другой первичной импликанты, возникнет импульсная помеха. Чтобы исправить это, мы добавили другую цепь, которая охватывает границу первичных импликант, как показано на **Рис. 2.78**. Вы могли бы узнать в этом теорему согласованности, где добавленный терм $\overline{A}C$ — это согласованный или избыточный терм.

На **Рис. 2.79** показана схема, устойчивая к паразитным импульсам. Добавленный элемент И выделен синим. Сейчас переключение B, когда $A = 0$ и $C = 1$, не вызывает паразитного импульса на выходе, поскольку синий элемент И формирует на выходе 1 во время этого перехода.

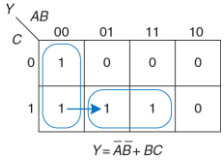


Рис. 2.77 **Переход от одной импликанты к другой**

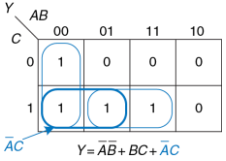


Рис. 2.78 **Карта Карно без импульсных помех**

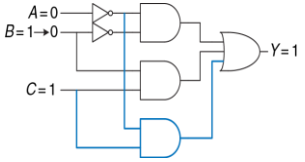


Рис. 2.79 **Схема без импульсных помех**

В общем случае, паразитный импульс может возникать, когда одна переменная пересекает границу между двумя первичными импликантами в карте Карно. Мы можем устранить эти импульсы добавлением избыточных импликант в карту Карно, чтобы покрыть эти границы. Естественно, это будет сделано ценой дополнительных аппаратных затрат.

Однако одновременное переключение нескольких входов также может стать причиной паразитных импульсов. Эти импульсы не могут быть исправлены дополнительными элементами в схеме. Поскольку подавляющее большинство интересующих нас систем имеют одновременные (или почти одновременные) переключения множества входов, возникновение паразитных импульсов в них неизбежно. Хотя

мы показали, как устранить один вид импульсных помех, смысл дискуссии о паразитных импульсах не в том, чтобы устранять их, а в том чтобы знать, что они есть. Это особенно важно, при анализе временных диаграмм в симуляторе или на экране осциллографа.

2.10 РЕЗЮМЕ

Цифровая схема – это модуль с дискретными значениями входов и выходов и спецификацией, описывающей его функциональные и временные характеристики. Эта глава была посвящена комбинационным схемам, выходы которых зависят только от текущих значений на их входах.

Функциональное описание комбинационной схемы может быть задано таблицей истинности или логическим выражением. Логическое выражение для любой таблицы истинности может быть получено в виде совершенной дизъюнктивной нормальной формы или совершенной конъюнктивной нормальной формы. В первом случае функция записывается как дизъюнкция конъюнкций, то есть булева сумма (логическое «ИЛИ») одной или более импликант. Импликанта есть произведение (логическое «И») литералов. Литералы же – это прямая или комплементарная форма входных переменных.

Логические выражения могут быть упрощены, используя правила булевой алгебры. В частности, их можно упростить, объединяя

импликанты, которые отличаются только прямой и комплементарной формами одного из литералов: $PA + P\overline{A} = P$.

Карты Карно – визуальный инструмент для минимизации функций двух–четырех переменных. На практике разработчики обычно могут упростить функции нескольких переменных «в уме», исходя только из своего опыта. Системы автоматизированного проектирования используются для более сложных функций; такие методы и инструменты обсуждаются в **главе 4**.

Логические элементы соединяют для того, чтобы создать комбинационную схему, которая выполняет желаемую функцию. Любая функция в дизъюнктивной нормальной форме может быть построена, используя двухуровневую логику: элемент НЕ образует комплементарную форму входов, элемент И формирует произведения и элемент ИЛИ формирует сумму. В зависимости от функции и доступности базовых элементов, многоуровневая логическая реализация с элементами разных типов может оказаться более эффективной. Например, для КМОП-схем больше подходят элементы И-НЕ и ИЛИ-НЕ, потому что эти элементы могут быть построены напрямую на КМОП-транзисторах без использования дополнительного инвертора. Когда используются элементы И-НЕ и ИЛИ-НЕ, для сокращения числа инверторов полезно применять перемещение инверсии.

Логические элементы комбинируются, чтобы создать более сложные схемы, такие как мультиплексоры, дешифраторы и схемы приоритета. Мультиплексор выбирает один из входов данных, основываясь на входе управления. Дешифратор устанавливает один из выходов в ВЫСОКОЕ значение в соответствии со входами. Приоритетная схема выдает 1 на выход, указывающий на вход с самым высоким приоритетом. Все эти схемы – примеры комбинационных «строительных блоков». В **главе 5** вы познакомитесь с еще большим количеством «строительных блоков», включая различные арифметические схемы. Эти блоки будут широко использованы при создании микропроцессора в **главе 7**.

Временные характеристики комбинационной схемы включают в себя задержки распространения и отклика. Они указывают на наибольшие и наименьшие времена между изменением входа и соответствующим изменением выходов. Вычисление задержки распространения заключается в определении критического пути в схеме и затем в сложении вместе задержек распространения всех элементов на этом пути. Существует множество различных способов реализации сложной комбинационной схемы; эти способы предполагают компромисс между ее скоростью работы и ценой.

В следующей главе будут рассмотрены последовательностные схемы, чьи выходы зависят как от текущих значений входов, так и от всей

предыстории (последовательности) изменений на них. Другими словами, мы рассмотрим схемы, обладающие свойством памяти.

УПРАЖНЕНИЯ

Упражнение 2.1 Запишите логическое выражение в совершенной дизъюнктивной нормальной форме для всех таблиц истинности, приведенных на **Рис. 2.80**.

(a)

A	*B*	*Y*
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Рис. 2.80 **Таблицы истинности для упражнений 2.1 и 2.3**

Упражнение 2.2 Запишите логическое выражение в совершенной дизъюнктивной нормальной форме для всех таблиц истинности, приведенных на **Рис. 2.81**.

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

B	C	A	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

B	C	A	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Рис. 2.81 **Таблицы истинности для упражнений 2.2 и 2.4**

Упражнение 2.3 Запишите логическое выражение в совершенной конъюнктивной нормальной форме для всех таблиц истинности, приведенных на **Рис. 2.80**.

Упражнение 2.4 Запишите логическое выражение в совершенной конъюнктивной нормальной форме для всех таблиц истинности, приведенных на **Рис. 2.81**.

Упражнение 2.5 Минимизируйте все логические выражения, полученные в **упражнении 2.1**.

Упражнение 2.6 Минимизируйте все логические выражения, полученные в **упражнении 2.2**.

Упражнение 2.7 Нарисуйте достаточно простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.5**. Под достаточно простой схемой подразумевается такая, которая состоит из небольшого количества элементов, но при этом ее разработчик не тратит много времени на проверку каждой из возможных реализаций схемы.

Упражнение 2.8 Нарисуйте достаточно простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.6**.

Упражнение 2.9 Повторите **упражнение 2.7**, используя только элементы НЕ, И и ИЛИ.

Упражнение 2.10 Повторите **упражнение 2.8**, используя только элементы НЕ, И и ИЛИ.

Упражнение 2.11 Повторите **упражнение 2.7**, используя только элементы НЕ, И-НЕ и ИЛИ.

Упражнение 2.12 Повторите **упражнение 2.8**, используя только элементы НЕ, И-НЕ и ИЛИ.

Упражнение 2.13 Упростите следующие логические выражения, используя булевы теоремы. Проверьте правильность результатов, используя таблицы истинности или карты Карно.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Упражнение 2.14 Упростите следующие логические выражения, используя булевы теоремы. Проверьте правильность результатов, используя таблицы истинности или карты Карно.

(a) $Y = \overline{A}BC + \overline{A}B\overline{C}$

(b) $Y = \overline{ABC} + A\overline{B}$

(c) $Y = ABC\,\overline{D} + A\overline{BCD} + \overline{(A + B + C + D)}$

Упражнение 2.15 Нарисуйте достаточно простые комбинационные схемы, реализующие выражения, полученные в упражнении **2.13**.

Упражнение 2.16 Нарисуйте достаточно простые комбинационные схемы, реализующие выражения, полученные в **упражнении 2.14**.

Упражнение 2.17 Упростите каждое из следующих логических выражений. Нарисуйте достаточно простые комбинационные схемы, реализующие полученные выражения.

(a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D + \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Упражнение 2.18 Упростите каждое из следующих логических выражений. Нарисуйте достаточно простые комбинационные схемы, реализующие полученные выражения.

(a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$

(b) $Y = \overline{(A + B + C)}D + AD + B$

(c) $Y = ABCD + \overline{A}B\overline{C}D + \overline{(\overline{B} + D)}E$

Упражнение 2.19 Приведите пример таблицы истинности, содержащей от 3 до 5 миллиардов строк, которая может быть реализована схемой, использующей менее 40 двухвходовых логических элементов (но не менее одного).

Упражнение 2.20 Приведите пример схемы с циклическим путем, которая, тем не менее, является комбинационной.

Упражнение 2.21 Алиса Хакер утверждает, что любое логическое выражение может быть записано в виде минимальной дизъюнктивной нормальной формы, то есть в виде булевой суммы простых импликант. Бен Битдидл утверждает, что существуют такие выражения, минимальные формы которых не содержат все простые импликанты. Объясните, почему Алиса права, или приведите контрпример, подтверждающий точку зрения Бена.

Упражнение 2.22 Докажите следующие теоремы, используя совершенную индукцию. Вам не надо доказывать двойственные им теоремы.

a) Теорема об идемпотентности (T3)

b) Теорема дистрибутивности (T8)

c) Теорема склеивания (T10)

Упражнение 2.23 Докажите теорему де Моргана (T12) для трех переменных, используя совершенную индукцию.

Упражнение 2.24 Напишите логические выражения для схемы, показанной на **Рис. 2.82**. Вы не должны минимизировать эти выражения.

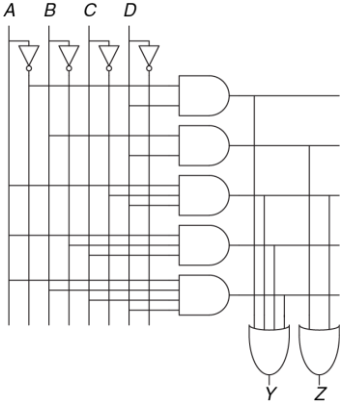


Рис. 2.82 **Принципиальная схема**

Упражнение 2.25 Минимизируйте логические выражения, полученные в **упражнении 2.24**, и нарисуйте усовершенствованную схему, реализующую эти функции.

Упражнение 2.26 Используя элементы, эквивалентные по де Моргану, и метод перемещения инверсии, перерисуйте схему, приведенную на **Рис. 2.83**, чтобы вы могли найти ее логическое выражение «на глаз». Запишите это логическое выражение.

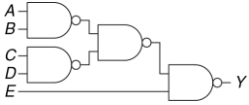


Рис. 2.83 Принципиальная схема

Упражнение 2.27 Повторите **упражнение 2.26** для схемы на **Рис. 2.84**.

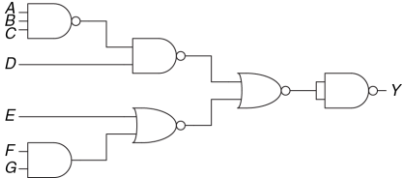


Рис. 2.84 Принципиальная схема

Упражнение 2.28 Найдите минимальное логическое выражение для функции, заданной на **Рис. 2.85**. Не забудьте при этом воспользоваться наличием безразличных значений в таблице истинности.

Упражнение 2.29 Нарисуйте схему, реализующую функцию, полученную в упражнении **2.28**.

A	*B*	*C*	*D*	*Y*
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рис. 2.85 Таблица истинности

Упражнение 2.30 Могут ли в схеме из **упражнения 2.29** появиться потенциальные паразитные импульсы при изменении состояния одного из входов? Если нет, объясните почему. Если да, покажите, как надо изменить схему, чтобы устранить паразитные импульсы.

Упражнение 2.31 Найдите минимальное логическое выражение для функции, заданной на **Рис. 2.86**. Не забудьте при этом воспользоваться наличием безразличных значений в таблице истинности.

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Рис. 2.86 **Таблица истинности**

Упражнение 2.32 Нарисуйте схему, реализующую функцию, полученную в **упражнении 2.31**.

Упражнение 2.33 Бен Битдидл будет наслаждаться пикником в солнечный день, если не будет муравьев. Он также будет наслаждаться пикником в любой день, если увидит колибри, а также в те дни, когда есть муравьи и божьи коровки.

Запишите логическое выражение для его радости (E) в терминах наличия солнца (S), муравьев (A), колибри (H) и божьих коровок (L).

Упражнение 2.34 Завершите проектирование дешифратора семисегментного индикатора для сегментов от S_c до S_g (см. **пример 2.10**):

a) Выведите логическое выражение для выходов от S_c до S_g при условии, что при подаче на вход значения более 9 выход должен быть нулем.

b) Выведите логическое выражение для выходов от S_c до S_g при условии, что при подаче на вход значения более 9 состояние выхода безразлично.

c) Нарисуйте достаточно простую реализацию на уровне логических элементов для случая (b). При необходимости используйте общие логические элементы для нескольких выходов.

Упражнение 2.35 Схема имеет четыре входа и два выхода. На входы $A_{3:0}$ подается число от 0 до 15. Выход P должен быть равен ИСТИНЕ, если число на входе простое (0 и 1 не являются простыми, а 2, 3, 5 и так далее – являются). Выход D должен быть равен ИСТИНЕ, если число делится на 3. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.36 Приоритетный шифратор имеет 2^N входов. Он формирует на N-разрядном выходе номер самого старшего входного бита, который принимает значение ИСТИНА. Он также формирует на выходе NONE значение ИСТИНА, если ни один из входов не принимает значение ИСТИНА. Спроектируйте

восьмивходовой приоритетный шифратор с входом $A_{7:0}$ и выходами $Y_{2:0}$ и NONE. Например, если вход *A* принимает значение 00100000, то выход *Y* должен быть 101, а NONE – 0. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.37 Спроектируйте модифицированный приоритетный шифратор (см. **упражнение 2.36**), который имеет 8-разрядный вход *A*7:0, а также 3-разрядные выходы $Y_{2:0}$ и $Z_{2:0}$. На выходе Y формируется номер самого старшего входного бита, который принимает значение ИСТИНА. На выходе *Z* формируется номер второго по старшинству входного бита, который принимает значение ИСТИНА. *Y* принимает значение 0, если все биты входа – ЛОЖЬ. *Z* принимает значение 0, если только один бит входа – ИСТИНА. Запишите упрощенное логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.38 *M*-битный унарный код числа *k* содержит *k* единиц в младших разрядах и *M*–*k* нулей во всех старших разрядах. Преобразователь бинарного кода в унарный имеет N входов и 2^N–1 выходов. Он формирует (2^N–1)-битный унарный код для числа, установленного на входе. Например, если на входе 110, то на выходе должно быть 0111111. Спроектируйте преобразователь трехбитного бинарного кода в семибитный унарный. Запишите логическое выражение для каждого из выходов и нарисуйте схему.

Упражнение 2.39 Запишите минимизированное логическое выражение для функции, выполняемой схемой, показанной на **Рис. 2.87**.

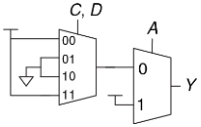


Рис. 2.87 **Схема на мультиплексорах**

Упражнение 2.40 Запишите минимизированное логическое выражение для функции, выполняемой схемой, показанной на **Рис. 2.88**.

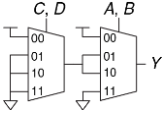


Рис. 2.88 **Схема на мультиплексорах**

Упражнение 2.41 Спроектируйте схему, реализующую функцию, описанную на **Рис. 2.80 (b)**, используя:

a) Восьмивходовой мультиплексор (8:1)

b) Четырехвходовой мультиплексор (4:1) и один инвертор

c) Двухвходовой мультиплексор (2:1) и два любых других логических элемента

Упражнение 2.42 Спроектируйте схему, реализующую функцию из **упражнения 2.17 (a)**, используя:

a) Восьмивходовой мультиплексор (8:1)

b) Четырехвходовой мультиплексор (4:1) без других логических элементов

c) Двухвходовой мультиплексор (2:1), один элемент ИЛИ и один инвертор

Упражнение 2.43 Рассчитайте задержку распространения t_{pd} и задержку реакции t_{cd} для схемы на **Рис. 2.83**. Значения задержек элементов даны в **Табл. 2.8**.

Упражнение 2.44 Рассчитайте задержку распространения и задержку реакции для схемы на **Рис. 2.84**. Значения задержек элементов даны в **Табл. 2.8**.

Упражнение 2.45 Нарисуйте схему для быстродействующего дешифратора 3:8. Значения задержек элементов даны в **Табл. 2.8** (используйте только указанные в таблице элементы). Спроектируйте ваш дешифратор таким образом, чтобы он имел минимальный возможный критический путь, и найдите этот путь. Каковы задержки распространения и реакции?

Табл. 2.8 Значения задержек элементов для Упражнений 2.43–2.47

Элемент	t_{pd} (пс)	t_{cd} (пс)
НЕ	15	10
Двухвходовой И-НЕ	20	15
Трехвходовой И-НЕ	30	25
Двухвходовой ИЛИ-НЕ	30	25
Трехвходовой ИЛИ-НЕ	45	35
Двухвходовой И	30	25
Трехвходовой И	40	30
Двухвходовой ИЛИ	40	30
Трехвходовой ИЛИ	55	45
Двухвходовой Исключающее ИЛИ	60	40

Упражнение 2.46 Измените схему из **упражнения 2.35**, чтобы она была максимально быстродействующей. Используйте только элементы из **Табл. 2.8**. Нарисуйте новую схему и определите критический путь. Каковы задержки распространения и реакции?

Упражнение 2.47 Измените приоритетный дешифратор из **упражнения 2.36**, чтобы он работал максимально быстро. Используйте только элементы из **Табл. 2.8**. Нарисуйте новую схему и определите критический путь. Каковы задержки распространения и реакции?

Упражнение 2.48 Спроектируйте восьмивходовой мультиплексор так, чтобы задержка от входов до выходов была минимальной. Используйте только элементы из **Табл. 2.7**. Нарисуйте схему. Используя значения задержек элементов из таблицы, определите задержку от входов до выходов.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

Здесь представлены примеры вопросов, которые могут быть заданы соискателям при поиске работы в области проектирования цифровых устройств.

Вопрос 2.1 Нарисуйте схему, реализующую функцию «исключающее ИЛИ», используя логические элементы И-НЕ. Какое минимальное количество элементов И-НЕ для этого требуется?

Вопрос 2.2 Спроектируйте схему, которая показывает, содержит ли заданный месяц 31 день. Месяц задается 4-разрядным входом A3:0. Например, значению 0001 на входе соответствует месяц январь, а значению 1100 – декабрь. Выход схемы Y должен принимать значение ИСТИНА только тогда, когда на вход подан номер месяца, в котором 31 день. Напишите упрощенное выражение и нарисуйте схему, используя минимальное количество элементов (подсказка: не забудьте воспользоваться безразличными состояниями).

Вопрос 2.3 Что такое буфер с тремя состояниями? Как и для чего он используется?

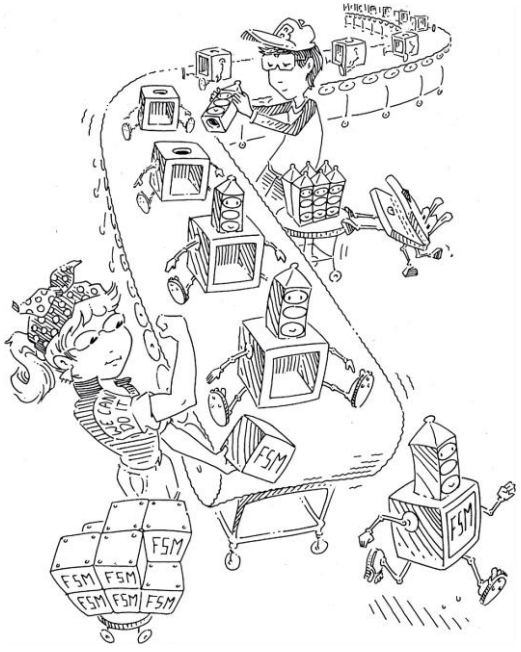
Вопрос 2.4 Элемент или набор элементов является универсальным, если он может быть использован для реализации любой логической функции. Например, набор {И, ИЛИ, НЕ} является универсальным.

a) Является ли элемент И универсальным? Почему?

b) Является ли набор элементов {ИЛИ, НЕ} универсальным? Почему?

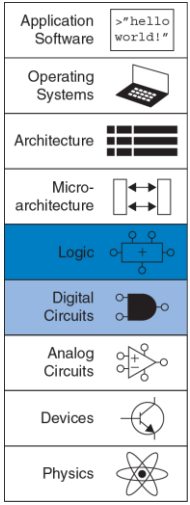
c) Является ли элемент И-НЕ универсальным? Почему?

Вопрос 2.5 Объясните, почему задержка реакции схемы может быть меньше или равна задержке распространения.


FSM
FSM
FSM
FSM
FSM
FSM
FSM

Проектирование последовательностной логики

3



3.1 ВВЕДЕНИЕ

В предыдущей главе мы рассмотрели процесс анализа и проектирования комбинационных логических схем. Значение на выходе комбинационной схемы зависит лишь от значений на входе в текущий момент времени. Мы можем создать оптимизированную схему согласно техническому заданию в виде таблицы истинности или в виде логического выражения.

В этой главе мы будем анализировать и проектировать *последовательностные логические схемы*. Значение на выходе последовательностной логической схемы зависит как от текущих, так и от предыдущих входных значений, следовательно, последовательностные логические схемы обладают памятью. Последовательностные логические схемы могут явно запоминать предыдущие значения определенных входов, а могут «сжимать» предыдущие значения определенных входов в меньшее количество информации, называемое *состоянием системы*. Состояние цифровой последовательностной схемы – набор бит, называемый *переменными состояния*. Эти биты содержат всю информацию о прошлом, необходимую для определения будущего поведения схемы.

Глава начинается с изучения защелок и триггеров. Они являются простыми последовательностными схемами, запоминающими один бит

информации. Вообще говоря, последовательностные схемы достаточно сложно анализировать. С целью упрощения проектирования мы ограничимся только синхронными схемами, состоящими из комбинационной логики и набора триггеров, хранящих информацию о состоянии системы. В главе описываются конечные автоматы, с помощью которых можно легко и просто проектировать последовательностные схемы. Наконец, мы проанализируем быстродействие последовательностных схем и обсудим параллельные вычисления как способ повышения быстродействия.

3.2 ЗАЩЕЛКИ И ТРИГГЕРЫ

Основным блоком для построения памяти является бистабильная ячейка – элемент с двумя устойчивыми состояниями. На **Рис. 3.1 (a)** показана простая бистабильная ячейка, состоящая из пары инверторов, замкнутых в кольцо. Эту схему можно перерисовать так, чтобы рисунок выглядел симметрично (**Рис. 3.1 (b)**). Теперь видно, что инверторы соединены перекрестно, то есть вход I1 соединен с выходом I2 и наоборот. У схемы нет ни одного входа, зато есть два выхода Q и $\overline{Q}$. Анализ этой схемы отличается от анализа комбинационной схемы, так как схема является циклической: Q зависит от $\overline{Q}$, а $\overline{Q}$ зависит от Q.

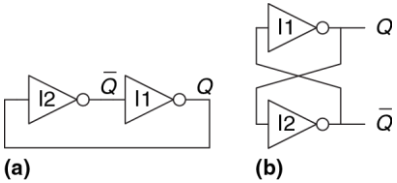


Рис. 3.1 **Перекрестно соединенные инверторы**

Выход последовательностной схемы принято обозначать буквой Q аналогично тому, что выход комбинационной схемы принято обозначать буквой Y.

Рассмотрим два случая: Q=0 и Q=1

- Случай I: Q=0

  Как показано на **Рис. 3.2 (a)**, на вход I2 поступает сигнал Q = 0. I2 инвертирует сигнал и подает на вход I1 сигнал $\bar{Q}$ = 1. Соответственно, на выходе I1 – логический 0. В рассмотренном случае схема находится в *стабильном состоянии*.

- Случай II: Q=1

  Как показано на **Рис. 3.2 (b)**, на вход I2 поступает 1 (Q). I2 инвертирует сигнал и подает на вход I1 0 $\bar{Q}$. Соответственно, на

выходе I1 – логическая 1. В этом случае схема также находится в стабильном состоянии.

Так как инверторы, включенные перекрестно, имеют два стабильных состояния $Q = 0$ и $Q = 1$, то говорят, что схема бистабильна. У схемы есть и третье состояние, когда оба выхода находятся в состоянии между 0 и 1. Такое состояние называется *метастабильным*, и оно будет рассмотрено в **разделе 3.5.4**.

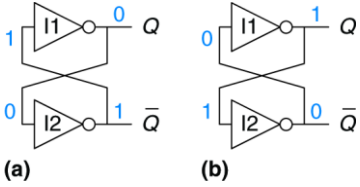


Рис. 3.2 **Бистабильный режим перекрестно соединенных инверторов**

Элемент с N стабильными состояниями хранит $\log_2 N$ бит информации. Таким образом, бистабильная ячейка хранит 1 бит. Состояние перекрестно включенных инверторов содержится в одной переменной состояния Q. Значение Q сообщает нам всю информацию о прошлом, необходимую для определения будущего поведения схемы. В частности, если Q =1, то оно и будет 0 всегда, а если $\overline{Q} = 1$, то оно и останется 1. У схемы есть еще один выход – $\overline{Q}$. Но $\overline{Q}$ не содержит

никакой дополнительной информации, так как если Q известно, то $\overline{Q}$ определено однозначно. С другой стороны, $\overline{Q}$ можно было бы также рассматривать как переменную состояния.

При включении питания исходное состояние последовательностной схемы неизвестно и обычно непредсказуемо. Оно может быть различным всякий раз, когда схему включают.

Несмотря на то, что перекрестно включенные инверторы могут хранить бит информации, они не используются на практике, так как у схемы нет входов, с помощью которых пользователь мог бы контролировать ее состояние. Однако, другие элементы, такие как защелки и триггеры, имеют входы, которые позволяют управлять переменной состояния. Эти схемы рассматриваются в оставшейся части раздела.

3.2.1 RS-триггер

Одной из простейших последовательностных схем является *RS-триггер*, (от англ. Reset и Set), состоящий, как показано на **Рис. 3.3**, из двух перекрестно включенных элементов ИЛИ-НЕ. У защелки есть два входа – R и S и два выхода Q и $\overline{Q}$. Принципы работы RS-триггера и схемы с перекрестно включенными инверторами аналогичны, но состояние защелки контролируются R и S входами, которые сбрасывают и устанавливают выход Q.

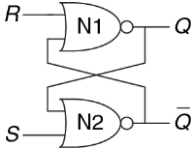


Рис. 3.3 RS-триггер (защелка)

Для того чтобы понять, как работает неизвестная цепь, обычно строят ее таблицу истинности. Вспомним, что на выходе элемента ИЛИ-НЕ появляется логический нуль, если на какой-либо из его входов подана логическая единица. Рассмотрим четыре возможных комбинации *R* и *S*:

- Случай I: *R*=1, *S*=0

  На входе N1 как минимум одна единица – вход *R*, следовательно, выход *Q*=0. Оба входа N2 – в состоянии логического нуля (*Q*=0 и *S*=0), поэтому выход $\overline{Q}$ =1.

- Случай II: *R*=0, *S*=1

  На вход N1 поступает 0 и $\overline{Q}$. Так как мы еще не знаем значения $\overline{Q}$, мы не можем определить значение *Q*. На вход N2 поступает как минимум одна единица *S*, поэтому на выходе $\overline{Q}$ нуль. Теперь можно вернуться к определению состояния выхода элемента N1. Мы знаем, что на обоих его входах 0, следовательно, *Q*=1.

- Случай III: R=1, S=1

  Как на входе N1, так и на входе N2 как минимум по одной единице (R и S), поэтому на выходе каждой защелки – логический 0. Следовательно, Q=0 и $\overline{Q}$ =0.

- Случай IV: R=0, S=0

  На вход N1 поступает 0 и $\overline{Q}$. Так как мы еще не знаем значения $\overline{Q}$, мы не можем определить значение на выходе элемента N1. На вход N2 поступает 0 и Q. Так как мы еще не знаем значения Q, мы не можем определить значение на выходе элемента N2. Кажется, мы зашли в тупик. Этот случай аналогичен случаю с двумя перекрестно включенными инверторами. Мы знаем, что Q должен быть равен либо 0, либо 1. Итак, мы сможем решить проблему, если рассмотрим каждый из этих двух случаев.

- Случай IVa: Q=0

  Так как S и Q равны 0, то на выходе N2 будет логическая 1, $\overline{Q}$ =1, как показано на **Рис. 3.4 (а)**. Теперь на входе N1 есть одна единица – $\overline{Q}$, поэтому на его выходе Q=0, как мы и предполагали.

- Случай IVb: Q=1

  Так как Q = 1, то на выходе N2 будет 0, $\overline{Q}$ =0, как показано на **Рис. 3.4 (b)**. Теперь на обоих входах N1 нули (R и $\overline{Q}$), поэтому на его выходе логическая 1, Q=1, как мы и предполагали.

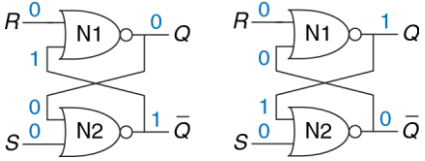


Рис. 3.4 **Бистабильные состояния RS-триггера**

Исходя из сказанного выше, предположим, что у Q есть какое-то определенное значение, установленное до наступления случая IV, которое мы назовем $Q_{пред}$, $Q_{пред}$ может быть либо 0, либо 1. $Q_{пред}$ отражает состояние системы. Когда R и S равны 0, на выходе Q будет сохраняться старое значение $Q_{пред}$, а $\overline{Q}$ будет его булевым дополнением.

Таблица истинности, приведенная на **Рис. 3.5**, иллюстрирует эти четыре случая. Входы R и S отвечают за сброс и установку значений соответственно.

Установить бит означает перевести его в логическую единицу, а сбросить – в логический нуль. Обычно $\overline{Q}$ является булевым дополнением Q. Когда поступает команда сброса R=1, выход Q принимает значение 0, а выход $\overline{Q}$ – противоположное (лог. 1). Когда поступает команда установки бита S=1, выход Q становится единицей, а $\overline{Q}$ – нулем. Если ни на один из входов не поступает логическая единица, на обоих выходах сохраняется предыдущее значение $Q_{пред}$. Подача на входы одновременно R=1 и S=1 не имеет особого смысла, так как это означает, что выход должен быть одновременно и установлен и сброшен, что невозможно. Защелка, не зная, что ей делать, выставляет как на прямом, так и на инверсном выходе логический 0.

Условное обозначение RS-триггера представлено на **Рис. 3.6**. Условные обозначения используются при модульном проектировании схемы с целью абстрагирования от внутренней структуры элемента.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Рис. 3.5 **Таблица истинности RS-триггера**

Рис. 3.6 **Обозначение RS-триггера**

Существует несколько способов построения RS-триггера, таких как использование логических элементов или транзисторов. Тем не менее, любой элемент схемы, специфицированный таблицей истинности на **Рис. 3.5**, обозначается символом на **Рис. 3.6** и называется RS-триггером.

Так же как и перекрестно включенные инверторы, RS-триггер является бистабильным элементом с одним битом состояния, хранящимся в *Q*. Состоянием можно управлять при помощи входов *R* и *S*. Когда на *R* поступает высокий уровень, выход сбрасывается в 0. Когда высокий уровень приходит на *S*, выход устанавливается в 1. Если ни на один вход не пришла логическая единица, триггер сохраняет свое предыдущее состояние, значение выходов не изменяется. Отметим, что вся история сигналов, поданных на вход, может быть сосредоточена в одной переменной состояния *Q*. Не имеет значения, что происходило в прошлом. Все, что нужно, чтобы предсказать будущее поведение RS-триггера, – это знать, было ли последнее изменение состояния триггера сбросом или установкой.

3.2.2 D-защелка

RS-триггер неудобен из-за необычного поведения, если на оба входа триггера одновременно поступает высокий уровень сигнала. Более серьезная проблема состоит в том, что вопросы *ЧТО* и *КОГДА* в контексте изменения состояния триггера объединены его *R* и *S* входами. Подача логической единицы на эти входы определяет не только, *ЧТО* произойдет, но и *КОГДА* это произойдет. Разработка схем упрощается, если эти вопросы *ЧТО* и *КОГДА* разделены. D-триггер-защелка (**Рис. 3.7 (a)**) решает эти проблемы. У триггера есть два входа: вход *данных* D, определяющий, каким будет следующее состояние, и вход *тактового сигнала* CLK, определяющий, когда оно изменится.

Рис. 3.7 **D-триггер-защелка: (a) схема, (b) таблица истинности, (c) обозначение**

Для анализа защелки снова составим таблицу истинности (**Рис. 3.7 (b)**). Сначала рассмотрим внутренние линии $\overline{D}$, *R* и *S*. Если *CLK*=0, то оба сигнала *R* и *S* нулевые, независимо от значения *D*. Если *CLK*=1, на выходе одного элемента И будет единица, а на выходе другого –

нуль. Элемент И, на выходе которого будет 1, определяется входом *D*. Значения *Q* и $\overline{Q}$ определяются *R* и *S* по таблице на **Рис. 3.5**. Заметим, что пока *CLK*=0, *Q* сохраняет предыдущее значение $Q_{пред}$. Если *CLK*=1, *Q*=*D*. Очевидно, что $\overline{Q}$ всегда является булевым дополнением *Q*. В D-защелке исключен случай необычного поведения при одновременно поданных сигналах сброса и установки (*R*=1 и *S*=1).

Таким образом, мы видим, что тактовый сигнал контролирует, КОГДА данные проходят через триггер-защелку. Когда *CLK*=1, защелка «*прозрачна*», т.е. она пропускает данные D на выход *Q*, как если бы он являлся обычным буфером. Когда *CLK*=0, защелка «*непрозрачна*», она не пропускает новые данные с входа D на выход *Q*, а *Q* сохраняет свое значение. D-защелку иногда называют *прозрачным триггером* или *триггером, синхронизируемым уровнем*. Условное обозначение D-защелки представлено на **Рис. 3.7(c)**.

Состояние D-триггера-защелки изменяется непрерывно, пока *CLK*=1. Позже в этой главе мы увидим, что часто удобнее изменять состояние схемы только в определенный момент времени. Следующий раздел – как раз об этом. В нем описывается D-триггер, синхронизируемый фронтом.

Иногда состояние защелки называют «открытым» или «закрытым», а не «прозрачным» или «непрозрачным».

3.2.3 D-Триггер

D-триггер, триггер синхронизируемый фронтом (далее – триггер), может быть построен из двух включенных последовательно D-защелок. Как показано на **Рис. 3.8 (a)**, тактовые сигналы, которые подаются на них, являются булевыми дополнениями друг друга. Первую защелку называют ведущей (master), а вторую – ведомой (slave). Защелки соединены линией N1. Условное обозначение D-триггера приведено на **Рис. 3.8 (b)**. Когда выход $\overline{Q}$ не используется, обозначение может быть упрощено до представленного на **Рис. 3.8 (c)**.

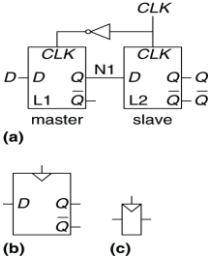


Рис. 3.8 **D-триггер: (a) схема, (b) обозначение, (c) упрощенное обозначение**

Когда *CLK*=0, master-защелка открыта, а slave – закрыта. Следовательно, значение со входа D проходит до линии N1. Когда

CLK=1, master-защелка закрывается, а slave-защелка открывается. Значение с N1 проходит на выход *Q*, но N1 становится отрезанным от *D*. Следовательно, то значение, которое было на входе *D* непосредственно перед переходом *CLK* из 0 в 1, сразу же попадает на выход *Q* после того как тактовый сигнал устанавливается в 1. Во все остальное время *Q* сохраняет свое прежнее значение, так как закрытый триггер постоянно блокирует путь между *D* и *Q*.

Другими словами, *D-триггер копирует значение с D на Q по переднему фронту тактового импульса и помнит это состояние все остальное время*. Перечитайте это определение до тех пор, пока вы его не запомните. Одна из самых распространенных ошибок начинающих разработчиков цифровых схем – они забывают, что такое синхронизация фронтом. Часто передний фронт тактового импульса называют просто фронтом. Вход D определяет новое, будущее состояние триггера. Фронт определяет момент времени, когда состояние будет обновлено.

D-триггер также известен как *MS-триггер*, *master-slave-триггер* и как *триггер, синхронизируемый фронтом*. Треугольник в обозначении указывает на то, что вход синхронизируется фронтом. У многих триггеров выход $\overline{Q}$ отсутствует, и их обычно используют, когда $\overline{Q}$ не нужен.

Пример 3.1 КОЛИЧЕСТВО ТРАНЗИСТОРОВ В ТРИГГЕРЕ

Сколько транзисторов содержится в D-триггере, описанном в этой главе?

Решение: в элементе ИЛИ-НЕ или И-НЕ используется по 4 транзистора. В инверторе используются два транзистора. Элемент И состоит из элементов И-НЕ и НЕ (инвертора), поэтому в нем используется 6 транзисторов. В RS-защелке – два элемента ИЛИ-НЕ или 8 транзисторов. В D-защелке используется RS-защелка, 2 элемента И и один элемент НЕ или 22 транзистора. В D-триггере используются две D-защелки и один элемент НЕ или 46 транзисторов. В **разделе 3.2.7** описываются более эффективные способы реализации триггера на основе КМОП-технологии с использованием проходных ключей.

Различие между триггером и защелкой весьма расплывчатое, оно изменялось с течением времени. В производственных кругах под триггером обычно понимают триггер, синхронизируемый фронтом, или, другими словами, это бистабильный элемент с тактовым входом. Состояние триггера изменяется только по переднему фронту тактового сигнала, то есть когда тактовый сигнал переходит из 0 в 1. Бистабильные элементы, в которых отсутствует синхронизация по фронту, обычно называют защелками.

Употребляя термины «триггер» или «защелка», обычно имеют в виду D-триггер или D-защелку соответственно, потому что именно эти триггеры чаще всего используются на практике.

3.2.4 Регистр

N-разрядный регистр – набор из *N* триггеров с общим тактовым сигналом. Таким образом, все биты регистра обновляются одновременно. Регистр является ключевым блоком при построении большинства последовательностных схем. На **Рис. 3.9** показана схема и обозначение 4-разрядного регистра со входами $D_{3:0}$ и выходами $Q_{3:0}$. $D_{3:0}$ и $Q_{3:0}$ являются 4-разрядными шинами.

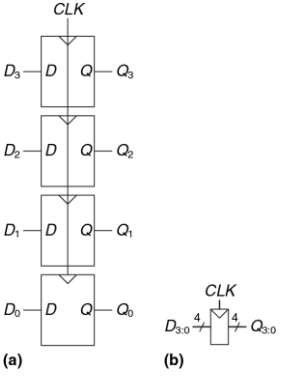


Рис. 3.9 **4-разрядный регистр: (a) схема, (b) обозначение**

3.2.5 Триггер с функцией разрешения

У некоторых триггеров имеется еще один вход, называемый *EN*, или *ENABLE* (разрешить). Этот вход определяет, будут ли данные загружены по фронту или нет. Когда на *EN* подается логическая единица, то такой D-триггер ведет себя так же как и обычный D-триггер. Если же на *EN* поступает логический нуль, то триггер игнорирует тактовый сигнал и сохраняет свое состояние. Такие триггеры полезны, если мы хотим загружать значения в триггер только на протяжении какого-то времени, а не по каждому фронту тактовому импульсу.

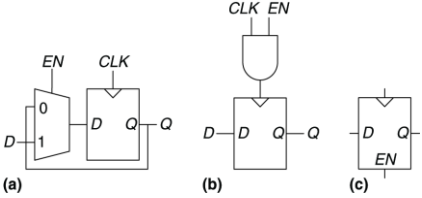


Рис. 3.10 **Триггер с функцией разрешения (a,b) схемы, (c) обозначение**

На **Рис. 3.10** показаны два способа добавления входа разрешения к обычному D-триггеру. На **Рис. 3.10 (a)** входной мультиплексор выбирает, подавать ли данные на вход *D*, если на *EN* логическая единица, или подавать на вход *D* старое значение с выхода *Q*, если

на *EN* подается логический нуль. На **Рис. 3.10 (b)** тактовый сигнал проходит, если *EN* равен единице; импульсы на вход тактового сигнала подаются в обычном режиме. Если на *EN* – логический нуль, то и на *CLK* – также нуль, и триггер сохраняет свое предыдущее состояние. Заметим, что сигнал *EN* не должен изменяться, пока *CLK*=1, во избежание сбоя (выброса) тактового сигнала (переключение в неверное время). Вообще говоря, добавление логических элементов в тракт тактирования – плохая идея. Управление тактированием вносит задержку в тактовый сигнал и может привести к временным ошибкам, о чем будет сказано в **разделе 3.5.3**, то есть делайте так только в том случае, если вы уверены в том, что вы делаете. Обозначение триггера с функцией разрешения представлено на **Рис. 3.10 (c)**.

3.2.6 Триггер с функцией сброса

В *триггере с функцией сброса* добавляется еще один вход, называемый *RESET* (сброс). Когда на *RESET* подан 0, сбрасываемый триггер ведет себя как обычный D-триггер. Когда на RESET подана 1, такой триггер игнорирует вход D и сбрасывает выход в 0. Триггеры с функцией сброса полезны, когда мы хотим ускорить установление определенного состояния (т.е. 0) во всех триггерах системы при первом включении.

Такие триггеры могут сбрасываться как синхронно, так и асинхронно. Синхронно сбрасываемые триггеры сбрасываются только по фронту сигнала CLK. Асинхронно сбрасываемые триггеры сбрасываются сразу же при поступлении логической единицы на вход *RESET*, вне зависимости от тактового сигнала.

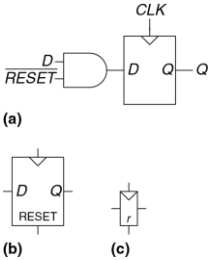


Рис. 3.11 **Синхронно сбрасываемый триггер: (a) схема, (b,c) обозначения**

На **Рис. 3.11 (a)** показано, как построить синхронно сбрасываемый триггер из обычного D-триггера и элемента И. Когда на $\overline{RESET}$ поступает логический нуль, элемент И подает 0 на вход триггера. Когда на $\overline{RESET}$ поступает логическая единица, элемент И пропускает сигнал *D* на вход триггера. В этом примере $\overline{RESET}$ – сигнал с активным

низким уровнем (инверсная логика). Это означает, что сброс происходит, когда на этот вход поступает 0, а не 1. Добавив инвертор, мы могли бы получить схему с активным высоким уровнем (прямая логика). На **Рис. 3.11 (b)** и **Рис. 3.11 (c)** показаны обозначения сбрасываемого триггера с прямым сбросом.

Асинхронно сбрасываемые триггеры требуют изменения своей внутренней структуры и оставлены для самостоятельного разбора (**упражнение 3.13**), однако, и они зачастую доступны разработчикам как стандартный компонент.

Как вы могли бы легко догадаться, иногда используются и триггеры с функцией установки. Когда установлен сигнал SET, в такой триггер загружается логическая 1, и они происходят в синхронном и асинхронном исполнениях. У сбрасываемых и устанавливаемых триггеров также может быть вход ENABLE, и они могут быть сгруппированы в N-разрядные регистры.

3.2.7 Проектирование триггеров и защелок на транзисторном уровне

В **примере 3.1** было показано, что если триггеры построены из логических элементов, то в них используется большое количество транзисторов. Но фундаментальная функция защелки (триггера,

синхронизируемого уровнем) – быть открытой или закрытой, делает ее схожей с ключом. В **разделе 1.7.7** было указано, что использование проходного вентиля – эффективный способ создать КМОП-ключ. Следовательно, мы можем воспользоваться преимуществами проходных ключей с целью уменьшения количества транзисторов.

Как показано на **Рис. 3.12 (а)**, компактная D-защёлка может быть спроектирована с использованием одного проходного ключа. Когда CLK=1, а $\overline{CLK}$ =0 , проходной ключ замкнут, таким образом, D проходит на Q, и защёлка открыта. Когда CLK=0, а $\overline{CLK}$ =1, проходной ключ разомкнут, следовательно, выход Q изолирован от входа D, и защёлка закрыта. Однако такой триггер имеет следующие существенные недостатки:

- **Плавающий потенциал на выходе**: Когда защелка закрыта, значение выхода Q не подтянуто ни к одному логическому уровню. В этом случае узел Q называют *плавающим*, или *динамическим*. Спустя некоторое время, шумы и утечка заряда могут изменить значение выхода Q.
- **Отсутствие буферов**: Отсутствие буферов приводило к некорректной работе нескольких коммерческих микросхем. Случайный выброс, приводящий к появлению на входе D

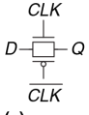
отрицательного напряжения, может включить n-канальный транзистор, открывая защелку, даже если *CLK*=0. Аналогично, выброс на входе *D* выше напряжения питания может открыть p-канальный транзистор, даже если *CLK*=0. Но проходной ключ симметричен, таким образом, он может быть открыт выбросами на выходе *Q*, тем самым влияя на значения входа *D*. Основное правило – ни вход проходного ключа, ни узел состояния последовательностной логической схемы никогда не должны применяться там, где существует вероятность возникновения помех или шумов.

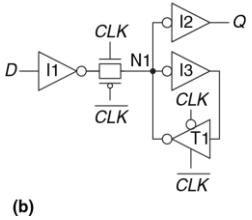
На **Рис. 3.12 (b)** изображена более надежная 12-транзисторная D защелка, используемая в современных коммерческих микросхемах. Хотя она и построена на основе тактируемых проходных ключей, в ней добавлены инвертеры I1 и I2, выполняющие роль входного и выходного буферов. Состояние защелки определяется состоянием узла N1. Инвертер I3 и буфер с тремя состояниями T1 образуют обратную связь, тем самым устраняя эффект плавающего потенциала на N1. Если узел N1 отклонится от стационарного состояния под влиянием помех или шума, то, когда *CLK*=0, буфер T1 вернет его в это состояние.

На **Рис. 3.13** изображен D-триггер, состоящий из двух защелок, управляемых сигналами *CLK* и $\overline{CLK}$. Мы удалили некоторые лишние

инверторы, и теперь для создания триггера требуется лишь 20 транзисторов.

На вход этой схемы поступают оба сигнала: CLK и $\overline{CLK}$. Если $\overline{CLK}$ отсутствует, то ставят инвертор, добавляя тем самым еще два транзистора.





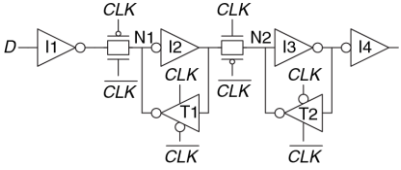


Рис. 3.12 **Схема D-триггера-защелки**

Рис. 3.13 **Схема D-триггера**

3.2.8 Общий обзор

Защелки и триггеры являются фундаментальными функциональными узлами последовательностных логических схем. D-защелка открыта, когда *CLK*=1, тем самым позволяя значению со входа *D* попасть на выход *Q*. D-триггер передает значение с *D* на *Q* только по фронту тактового сигнала. Во всех остальных случаях триггеры и защелки сохраняют свое предыдущее состояние. Регистром называется набор из нескольких D-триггеров с общим тактовым сигналом.

Пример 3.2 СРАВНЕНИЕ ЗАЩЕЛОК И ТРИГГЕРОВ

Бен Битдидл подал сигналы *D* и *CLK*, которые показаны на **Рис. 3.14**, на D-защелку и на D-триггер. Помогите ему определить значение выхода *Q* для каждого устройства.

Решение: На **Рис. 3.15** показаны временные диаграммы выходных сигналов с учетом небольших задержек в триггере и защелке. Стрелки указывают на причину, вызвавшую переключение сигнала на выходе. Исходное значение *Q* неизвестно, это и показано двумя горизонтальными линиями в начале диаграммы. Сначала рассмотрим защёлку. Во время прохождения первого фронта тактового сигнала *CLK* значение *D*=0, поэтому *Q* станет 0. Каждый раз, когда *D* будет изменяться, в то время как *CLK*=1, *Q* также будет изменяться. Если *D* будет изменяться, когда *CLK*=0, изменений на выходе *Q* не будет. Теперь рассмотрим триггер, синхронизируемый фронтом. Значение на выходе *Q*

становится равным значению на входе D по каждому фронту тактового сигнала *CLK*. Во всех других случаях *Q* не изменяется.

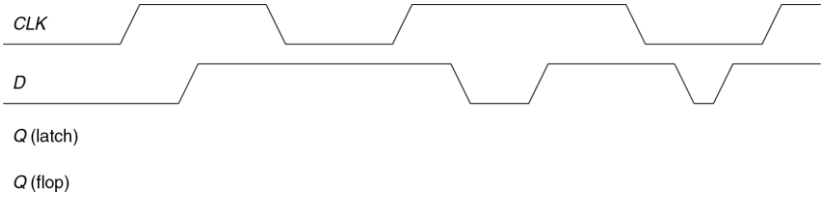


Рис. 3.14 **Исходные временные диаграммы**

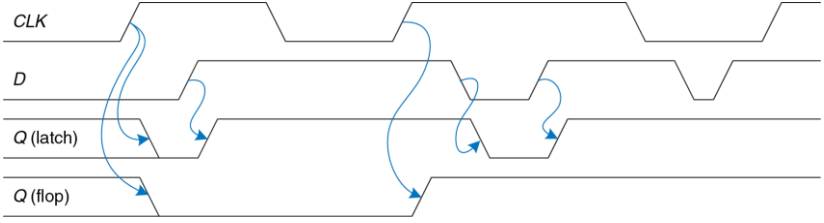


Рис. 3.15 **Решение Примера 3.2**

3.3 ПРОЕКТИРОВАНИЕ СИНХРОННЫХ ЛОГИЧЕСКИХ СХЕМ

Вообще говоря, последовательностные схемы включают в себя все схемы, которые не являются комбинационными, то есть последовательностные схемы – это те, значение выхода которых нельзя однозначно определить, зная лишь текущие значения входов. Поведение некоторых последовательностных схем может быть весьма сложным. Этот раздел начнется с разбора нескольких таких схем. Затем мы введем понятия синхронных последовательностных схем и динамической дисциплины. Ограничив себя рассмотрением только синхронных последовательностных схем, мы сможем сформулировать простые систематические подходы к анализу и проектированию таких схем.

3.3.1 Некоторые проблемные схемы

Пример 3.3 НЕУСТОЙЧИВЫЕ СХЕМЫ

Алиса Хакер столкнулась со схемой, которая состоит из трех инверторов, замкнутых в кольцо, как показано на **Рис. 3.16**. Выход третьего инвертора подается на вход первого. Задержка распространения каждого из инверторов равна 1 нс. Определите, что происходит в такой схеме.

Решение: Предположим, что в начальный момент времени сигнал X равен логическому 0. Тогда Y=1, Z=0, следовательно, X=1, что расходится с нашим

предположением. У этой схемы нет стабильных состояний, поэтому такая схема называется нестабильной или неустойчивой. На **Рис. 3.17** показано поведение схемы. Если сигнал X переходит из 0 в 1 в начальный момент времени, то *Y* перейдет из 1 в 0 в момент времени t=1 нс, а *Z* из 0 в 1 – в t=2 нс, а затем *X* перейдет обратно из 1 в 0 в момент времени t=3 нс. В свою очередь, Y перейдет из 0 в 1 в момент t=4 нс, *Z* перейдет из 1 в 0 во время t=5 нс, а X снова перейдет из 0 в 1 в момент времени t=6 нс, и далее такое поведение схемы будет повторяться. Каждый узел будет колебаться между 0 и 1 с периодом T=6 нс. Такая схема называется кольцевым генератором.

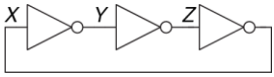


Рис. 3.16 **Кольцо из трех инверторов**

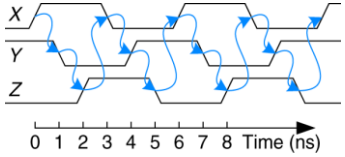


Рис. 3.17 **Временные диаграммы кольцевого генератора**

Период колебаний кольцевого генератора зависит от задержки распространения каждого инвертора. Эта задержка зависит от того, как изготовлен инвертор, от напряжения питания и даже от температуры. Поэтому точно определить период колебаний кольцевого генератора сложно. Иными словами, кольцевой генератор – последовательностная схема без входов и с одним выходом, значения которого периодически изменяются.

Пример 3.4 ГОНКИ В ПОСЛЕДОВАТЕЛЬНОСТНЫХ СХЕМАХ

Бен Битдидл спроектировал новую D-защелку, которая, как он считает, работает лучше, чем изображенная на **Рис. 3.7**, так как в ней используется меньше элементов. Он составил таблицу истинности для выхода Q по данным двух входов D и CLK и предыдущего состояния Q_{prev}. Основываясь на этой таблице, он написал логические уравнения. Для получения Q_{prev} используется обратная связь с выхода Q. Его схема изображена на **Рис. 3.18**. Работает ли его защелка корректно, независимо от задержек каждого элемента?

Решение: На **Рис. 3.19** показано, что схема может работать некорректно из-за появления гонок (англ. race condition), что приводит к сбою в случае, если определенные элементы медленнее других. Пусть $CLK=D=1$. Защелка открыта, пропускает данные и на выходе Q появляется логическая 1. Теперь сигнал CLK переходит из 1 в 0. Триггеру нужно запомнить свое предыдущее значение, т.е. сохранить $Q=1$. Предположим, что задержка распространения инвертора существенно больше задержек элементов И и ИЛИ. В таком случае сигналы N1 и Q перейдут из 1 в 0 ранее, чем сигнал $\overline{CLK}$ станет 1. В этом случае сигнал N2 никогда не примет значение логической единицы, и выходной сигнал схемы Q останется нулевым.

Это пример проекта асинхронной схемы, в которой выходы напрямую связаны обратной связью со входами. Асинхронные схемы не пользуются популярностью из-за непредсказуемости поведения, связанной с быстродействием элементов, когда поведение схемы зависит от того, какой сигнал внутри схемы пройдет быстрее других. Одна схема может работать, при этом другая, кажущаяся

идентичной, собранная из элементов с незначительно отличающимися задержками, может не работать. Или схема может работать только при определенных температурах или напряжениях, при которых задержки соответствуют расчетным. Подобные ошибки проектирования весьма сложно выявлять.

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

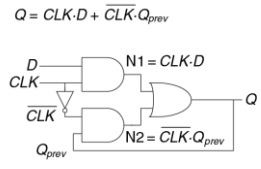


Рис. 3.18 **«Усовершенствованная» D-защелка**

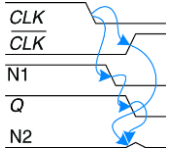


Рис. 3.19 **Временные диаграммы защелки, иллюстрирующие гонки**

3.3.2 Синхронные последовательностные схемы

В предыдущих двух примерах присутствовали циклические пути, в которых выходы напрямую соединены обратной связью со входами. Это скорее последовательностные, чем комбинационные схемы. В комбинационной логике нет циклических путей и нет зависимостей состояния выхода от времени прохождения сигнала. Если на входы комбинационной логической схемы поданы определенные сигналы,

то ее выход спустя некоторое время всегда установится в определенное корректное состояние. Однако, в последовательностных схемах с циклическими путями может появиться нежелательная нестабильность или гонки. Проверка таких схем требует много времени, и многие выдающиеся проектировщики делали подобные ошибки.

Во избежание таких проблем разработчики разрывают циклические пути и добавляют в разрыв регистры. Это превращает схему в набор комбинационной логики и регистров. В регистрах содержится состояние системы, изменяющееся только по фронту тактового импульса. В этом случае говорят, что состояние *синхронизировано* с тактовым сигналом. Если период тактового сигнала достаточно большой, чтобы все входы регистров успели установиться до фронта следующего тактового импульса, то эффекты, связанные с гонками, устраняются. Следование правилу «всегда использовать регистры в обратной связи» приводит нас к формальному определению синхронной последовательностной схемы.

Напомним, что схема (цепь) определяется набором входов и выходов и функциональными и временными параметрами. У последовательностной схемы существует конечный набор дискретных *состояний* $\{S_0, S_1, \ldots, S_{k-1}\}$. У *синхронной последовательностной схемы* есть вход тактового сигнала, передние фронты тактовых импульсов определяют последовательность точек на

временной оси, в которых происходят изменения состояния. Мы часто будем использовать термины «*текущее состояние*» и «*следующее состояние*» для того, чтобы различать состояние системы в настоящем от состояния системы, в которое она перейдет по фронту следующего тактового импульса. Функциональное описание определяет следующее состояние и значение каждого выхода для каждой возможной комбинации текущих состояний и входных сигналов. Временная спецификация состоит из верхней границы t_{pcq} и нижней границы t_{ccq} длительности временного промежутка от переднего фронта тактового импульса до момента изменения *выходного* сигнала, а также из времен *предустановки* и *удержания* t_{setup} и t_{hold}, которые определяет промежуток времени до и после поступления фронта тактового импульса, в течение которого значения на входах не должны изменяться.

> t_{pcq} это задержка распространения тракта вход тактового сигнала – выхода Q (до полного установления нового значения) последовательностной логической схемы. t_{ccq} – это задержка реакции тракта вход тактового сигнала – выхода Q. Эти задержки аналогичны задержкам t_{pd} и t_{cd} в комбинационной логике.

Правила *построения синхронных последовательностных схем* гласят, что схема является синхронной последовательностной схемой, если ее элементы удовлетворяют следующим условиям:

- Каждый элемент схемы является либо регистром, либо комбинационной схемой.
- Как минимум один элемент схемы является регистром.
- Все регистры тактируются единственным тактовым сигналом.
- В каждом циклическом пути присутствует как минимум один регистр.

Такое определение синхронной последовательностной схемы является достаточным, но в то же время слишком строгим. Например, в высокопроизводительных микропроцессорах некоторые регистры могут получать тактовый сигнал с задержкой. Тактовый сигнал также может подаваться через проходной ключ. Это позволяет добиться максимально возможного быстродействия системы. Также в некоторых микропроцессорах вместо регистров используются защелки. Однако это определение подходит ко всем синхронным последовательностным схемам, рассматриваемым в этой книге, и к большинству коммерческих систем.

Последовательностные схемы, не являющиеся синхронными, называют асинхронными.

Триггер является самой простой синхронной последовательностной схемой с двумя состояниями {0,1}. У него есть один вход данных *D*, один вход тактового сигнала *CLK*, один выход *Q*.

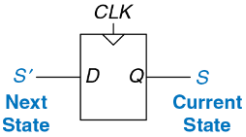


Рис. 3.20 Текущее и следующее состояние триггера

Функциональное описание D-триггера заключается в том, что его следующим состоянием является значение входа *D*, а значение выхода *Q* является текущим состоянием, (См. **Рис. 3.20**).

Мы часто будем обозначать текущее состояние переменной *S*, а следующее состояние – переменной *S*', то есть *S*' (штрих) обозначает следующее состояние, а не инверсию. Временные диаграммы последовательностных схем будут рассмотрены в **разделе 3.5**.

Два других вида синхронных последовательностных схем – конечные автоматы и конвейеры. Они будут рассмотрены позже в этой главе.

Пример 3.5 СИНХРОННЫЕ ПОСЛЕДОВАТЕЛЬНОСТНЫЕ СХЕМЫ

Какие из приведенных на **Рис. 3.21** схем являются последовательностными синхронными схемами?

Решение: Схема (a) является комбинационной, а не последовательностной, так как в ней отсутствуют регистры. (b) – простая последовательностная схема, так как в ней нет обратной связи. (c) не является ни комбинационной, ни последовательностной синхронной схемой, так как она содержит защелку, которая не является ни регистром, ни комбинационной схемой. (d) и (e) – синхронные последовательностные логические схемы; они являются двумя классами конечных автоматов, которые будут обсуждаться в **разделе 3.4**. (f) – ни комбинационная, ни синхронная последовательностная, так как у нее есть циклический путь с выхода комбинационной схемы на ее вход, при этом в тракте обратной связи отсутствует регистр. (g) является синхронной последовательностной схемой в виде конвейера, который мы изучим в **разделе 3.6**. (h) не является, строго говоря, синхронной последовательностной схемой, так как тактовый сигнал второго регистра отличен от первого из-за задержки, возникающей из-за двух инверторов.

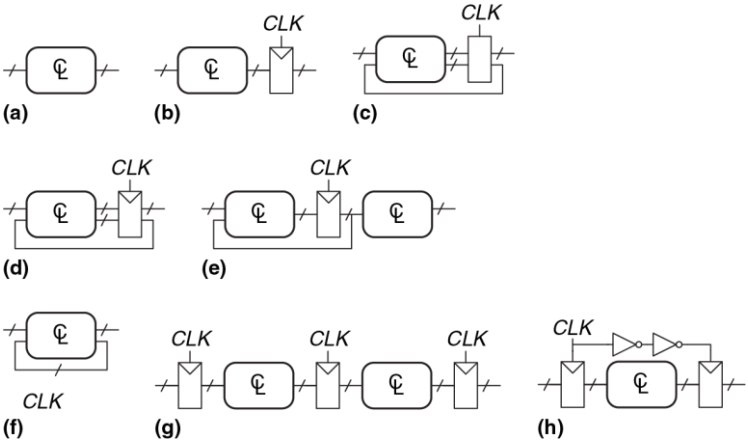


Рис. 3.21 Примеры схем

3.3.3 Синхронные и асинхронные схемы

Теоретически, из-за отсутствия временных ограничений, накладываемых на систему тактирующимися регистрами, при проектировании асинхронных схем разработчик обладает большей свободой, чем при проектировании синхронных. Таким же образом, как аналоговые схемы менее формализованы по сравнению с цифровыми, из-за того, что в аналоговых схемах могут использоваться произвольные напряжения, асинхронные схемы менее формализованы, чем синхронные, так как обратная связь в них может быть любой. Однако, оказывается, что синхронные схемы проектировать и использовать проще, чем асинхронные, так же как цифровые схемы проще проектировать, чем аналоговые. Несмотря на многолетнее научные исследования асинхронных схем, почти все современные цифровые схемы являются синхронными.

Асинхронные схемы иногда используются для связи между собой систем с разными тактовыми сигналами или для считывания значений со входов в произвольное время, так же как аналоговые схемы необходимы для взаимодействия с реальным миром аналоговых (непрерывных) напряжений. Более того, среди разработок в области асинхронных схем есть действительно выдающиеся, некоторые из них могут также улучшить характеристики синхронных схем.

3.4 КОНЕЧНЫЕ АВТОМАТЫ

Последовательностные логические схемы могут быть изображены в форме, представленной на **Рис. 3.22**.

Такие представления *называются конечными автоматами* (*КА*). Они получили свое название из-за того, что схема с *k*-регистрами может находиться в одном из 2^k, то есть в конечном числе, состояний. У КА *M* входов, *N* выходов и *k* бит состояний. На вход КА так же подается тактовый сигнал и, возможно, сигнал сброса. КА состоит из двух блоков комбинационной логики: логики перехода в *следующее состояние* и *выходной логики*, – и из регистра, в котором хранится текущее состояние. По фронту каждого тактового импульса автомат переходит в следующее состояние, которое определяется текущим состоянием и значениями на входах. Существует два основных класса конечных автоматов, которые отличаются своими функциональными описаниями. В *автомате Мура* выходные значения зависят лишь от текущего состояния, в то время как в *автомате Мили* выход зависит как от текущего состояния, так и от входных данных. Конечные автоматы предоставляют систематический способ проектирования синхронных последовательностных схем по заданному функциональному описанию. Этот метод будет описан ниже, а сейчас мы рассмотрим простой пример.

Автоматы Мура и Мили названы в честь своих изобретателей, ученых, разработавших теорию автоматов и математическую базу для них в фирме Bell Labs.

Эдвард Форест Мур (1925–2003) – не путайте с основателем компании Intel Гордоном Муром – опубликовал свою первую статью «Gedanken-experiments on Sequential Machines» («Мысленные эксперименты с последовательностными автоматами») в 1956 году.

Джоржд Мили (1927–2010) опубликовал «Method of Synthesizing Sequential Circuits» (Метод синтеза последовательностных схем) в 1955 году. Впоследствии он написал первую операционную систему для компьютера IBM 704, работая в Bell Labs. Позже он перешел на работу в Гарвардский Университет.

3.4.1 Пример проектирования конечного автомата

Для того чтобы проиллюстрировать процесс проектирования конечного автомата, рассмотрим проблему создания контроллера светофора для загруженного перекрестка в студенческом городке. Студенты-инженеры гуляют по Академической улице, на которой расположены учебные корпуса и общежитие. У них нет времени читать про конечные автоматы, и они не смотрят под ноги во время передвижения. Футболисты носятся между спортзалом и столовой по Беговой улице. Они гоняют мяч туда-сюда и тоже не смотрят под ноги. Несколько

студентов уже получили серьезные травмы на перекрестке, и декан попросил Бена Битдидла установить светофор, пока не произошли инциденты с летальным исходом.

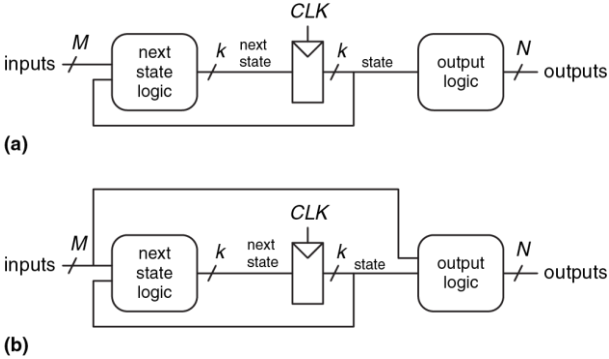


Рис. 3.22 **Конечные автоматы: (a) автомат Мура (b) автомат Мили**

Бен решил справиться с проблемой с помощью конечного автомата. Он установил два датчика движения, T_A и $T_Б$, на Академической и Беговой улицах соответственно. Каждый датчик выдает единицу, если студенты присутствуют на улице и нуль, если никого нет. Он также установил два светофора для управления движением, L_A и $L_Б$. Каждый

светофор получает входной цифровой сигнал, определяющий, каким светом он должен светить: красным, желтым или зеленым. Следовательно, у КА есть два входа, T_A и $T_Б$, и два выхода, L_A и $L_Б$. Перекресток с двумя светофорами и датчиками показан на **Рис. 3.23**. Бен подает тактовые импульсы раз в 5 секунд. По переднему фронту каждого импульса цвет светофора может измениться в зависимости от показаний датчиков движения. Также присутствует кнопка сброса, чтобы техники могли сбрасывать контроллер после подачи питания в известное исходное состояние. На **Рис. 3.24** автомат изображен в виде «черного ящика».

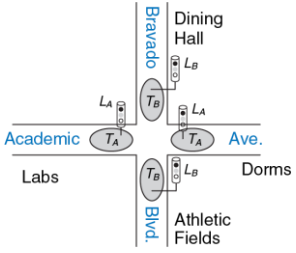


Рис. 3.23 **Карта кампуса**

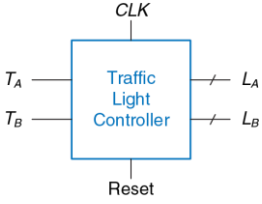


Рис. 3.24 **Конечный автомат как «черный ящик»**

Следующий шаг первокурсника – сделать набросок *диаграммы переходов* (или графа), показанный на **Рис. 3.25**, на котором приведены все возможные состояния системы и переходы между ними.

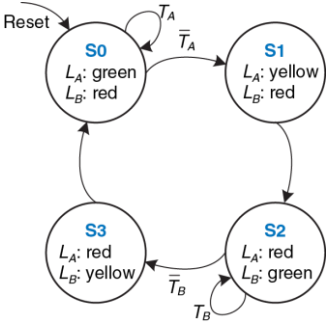


Рис. 3.25 **Таблица переходов**

После сброса светофор горит зеленым на Академической улице и красным – на Беговой. Каждые 5 секунд контроллер анализирует движение и решает, что же делать дальше. Если движение присутствует на Академической улице, то цвет не меняется. Как только Академическая улица освобождается, на ее светофоре 5 секунд горит желтый, затем загорается красный, а на Беговой – зеленый. Аналогично,

зеленый свет на Беговой улице сохраняется до тех пор, пока улица не станет свободной, затем светофор переключается на желтый, а затем – на красный.

Кружки на диаграмме переходов обозначают состояния, а дуги со стрелками между ними – переходы между этими состояниями. Переходы осуществляются по переднему фронту тактового импульса. Мы не будем изображать тактовый сигнал на диаграмме, так как он всегда присутствует в синхронных логических схемах. Более того, тактовый сигнал лишь определяет, когда случится переход, тогда как диаграмма определяет, какой именно переход произойдет. Стрелка, обозначенная как Сброс, указывает на переход извне в состояние S0, отражая то, что система перейдет в это состояние сразу после сброса, независимо от того, в каком она была состоянии до этого. Если присутствует несколько стрелок, выходящих из некоторого состояния, то эти стрелки подписывают, чтобы показать, какой входной сигнал вызвал этот переход. Например, система находится в состоянии S0. Система останется в состоянии S0, если T_A=1, и перейдет в состояние S1, если T_A=0. Если из этого состояния выходит только одна стрелка, это означает, что такой переход произойдет вне зависимости от состояния входов. Например, из состояния S1 система всегда будет переходить в состояние S2, когда L_A – красный, а L_B – зеленый.

На основе этой диаграммы переходов Бен Битдидл записал таблицу переходов (**Табл. 3.1**), которая отражает, каким должно быть следующее состояние *S*', соответствующее текущему состоянию и входным сигналам. Заметим, что в таблице используются символы X, означающие, что следующее состояние не зависит от конкретного входа. Также заметим, что сигнал сброс исключен из этой таблицы. Вместо этого мы использовали сбрасываемые триггеры, которые переходят в состояние S0 сразу после сброса, независимо от данных на входе.

Диаграмма переходов абстрактна в том смысле, что она использует состояния, обозначенные как {S0, S1, S2, S3}, и выходы, обозначенные как {красный, желтый, зеленый}.

Для построения реальной схемы состояниям и выходам должны быть поставлены в соответствие двоичные коды.

Бен выбрал простое кодирование, см. **Табл. 3.2** и **Табл. 3.3**. Каждое состояние и каждое выходное значение закодировано двумя битами: $S_{1:0}$, $L_{A1:0}$ и $L_{B1:0}$.

> Заметим, что состояния обозначаются как S0, S1 и так далее. S_0, S_1 – обозначения с индексами – являются битами двоичного числа, соответствующего некоторому состоянию.

Бен переписывает таблицу переходов, используя двоичное кодирование, как показано в **Табл. 3.4**. Эта таблица является таблицей истинности, определяющей логику следующего состояния. Она определяет следующее состояние *S'* как функцию входов и текущего состояния.

Табл. 3.1 Таблица переходов

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Табл. 3.2 Кодирование состояний

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Табл. 3.3 Кодирование выходов

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Анализ этой таблицы позволяет легко записать булево уравнение для следующего состояния в совершенной дизъюнктивной нормальной форме (СДНФ):

$$S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B$$
$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \quad (3.1)$$

Уравнения могут быть упрощены при помощи карт Карно, но часто это проще сделать в голове, внимательно изучив уравнения. Например, члены $\overline{T}_B$ и T_B в выражении для S_1', очевидно, сокращаются. Следовательно, S_1' сокращается до операции исключающего ИЛИ. **Выражения (3.2)** являются результатом упрощения выражений **(3.1)**.

Табл. 3.4 Таблица переходов с двоичным кодированием

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'	S'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

$$S_1' = S_1 \oplus S_0$$
$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \quad (3.2)$$

Подобным образом Бен записывает таблицу выходов (**Табл. 3.5**), определяя, каким должен быть выход для каждого состояния. Затем он снова составляет и упрощает булевы выражения для выходов. Например, L_{A1} =1 в строках, где истинно S_1=1.

$$L_{A1} = S_1$$
$$L_{A0} = S_1 \overline{S}$$
$$L_{B1} = \overline{S}_1 \quad (3.3)$$
$$L_{B0} = S_1 S_0$$

Табл. 3.5 Таблица выходов

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Наконец, Бен рисует автомат Мура в форме, приведенной на **Рис. 3.22 (a)**. Сначала он рисует 2-разрядный регистр состояний, как показано на **Рис. 3.26 (a)**. По каждому переднему фронту тактового сигнала регистр состояний фиксирует следующее состояние $S'_{1:0}$, и, таким образом, оно становится текущим состоянием $S_{1:0}$ Регистр состояний получает сигнал синхронного или асинхронного сброса для инициализации КА после подачи питания. Затем, основываясь на **уравнениях (3.2)**, Бен рисует схему определения следующего состояния, которые вычисляют следующее состояние по значению на входах и по текущему состоянию. Эта схема показана на **Рис. 3.26 (b)**. Наконец, он по **уравнениям (3.3)** рисует схему (см. **Рис. 3.26 (c)**), которая вычисляет значения на выходах автомата по текущему состоянию.

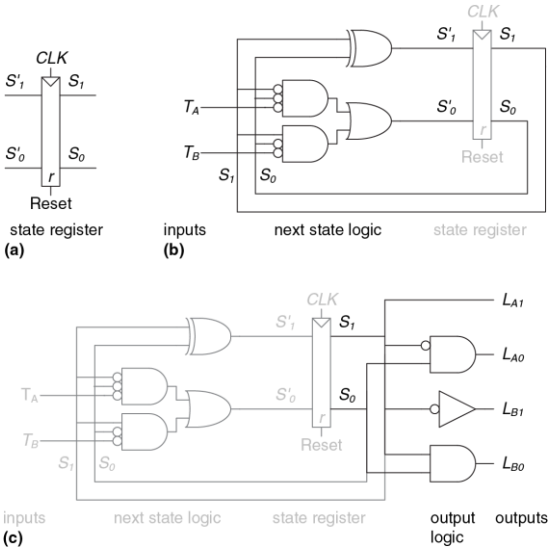


Рис. 3.26 **Схема конечного автомата контроллера светофора**

На **Рис. 3.27** показана временная диаграмма, иллюстрирующая переход контроллера светофора из одного состояния в другое. На диаграмме показаны сигнал *CLK*, *Reset* (Сброс), входы T_A и T_B, следующее состояние *S*', текущее состояние *S* и выходы L_A и L_B. Стрелки показывают причинную связь; например, изменение состояния вызывает изменение выходов, а изменение входов вызывает изменение состояния. Пунктирные линии соответствуют переднему фронту сигнала *CLK*, т.е. времени, когда состояние конечного автомата изменяется.

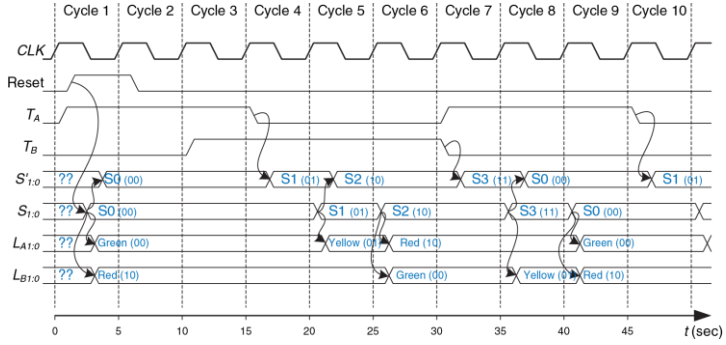


Рис. 3.27 **Временная диаграмма контроллера светофора**

Период тактового сигнала равен 5 секундам, поэтому сигналы светофора могут переключаться максимум раз в 5 секунд. Когда конечный автомат только включен, его состояние неизвестно, это показывают знаки вопроса. Следовательно, система должна быть сброшена для перевода ее в известное состояние. На этой временной диаграмме *S* незамедлительно сбрасывается в S0, подчеркивая то, что используются триггеры с асинхронным сбросом. В состоянии S0 свет L_A зеленый, а свет L_B красный.

> В этой схеме используются несколько элементов И с кружочками на входах. Их можно сделать из элементов И путем подключения инвертора на вход, или же заменить на элементы ИЛИ-НЕ с обычными входами и инверторами или на другие комбинации элементов. Лучший выбор зависит от особенностей используемой технологии.

В данном примере движение на Академической улице начинается сразу же. Следовательно, контроллер остается в состоянии S0, оставляя на светофоре L_A зеленый свет, даже если на Беговой улице кто-то появляется. По прошествии 15 секунд поток на Академической улице рассасывается, и T_A сбрасывается. Контроллер переходит в состояние S1 по фронту соответствующего тактового импульса и зажигает желтый свет на L_A. Еще через 5 секунд контроллер переходит в состояние S2, в котором на L_A загорается красный, а на L_B – зеленый свет. Контроллер остается в состоянии S2 до тех пор, пока Беговая улица не

опустеет. Затем он переходит в состояние S3, зажигая на L_B желтый свет. 5 секунд спустя контроллер переходит в состояние S0, переключая L_B на красный, а L_A – на зеленый свет. Процесс повторяется.

Вопреки ожиданиям, студенты не смотрят на сигналы светофора, и продолжают получать травмы. Декан просит Бена Битдидла и Алису Хакер спроектировать катапульту, чтобы студентов через открытые окна лаборатории и общежития, минуя травмоопасное пересечение. Но это тема для другой книги.

3.4.2 Кодирование состояний

В предыдущем примере кодирование состояний и выходов было выбрано произвольно. Выбор другой кодировки привел бы к другой схеме. Основная проблема заключается в том, как определить кодировку, которая потребует наименьшее количество элементов и приведет к наименьшим задержкам в схеме. К сожалению, простого способа найти самую лучшую кодировку не существует, кроме как перепробовать все возможные, что нерационально в случае, если число состояний велико. Однако зачастую возможно найти хорошую кодировку так, чтобы связанные состояния или выходы имели общие биты. При поиске набора возможных кодировок и выбора наиболее

рациональной из них часто используются системы автоматизированного проектирования (САПР).

Одно из важных решений в кодировании состояний – выбор между двоичным кодированием (00, 01, 10) и прямым кодированием (001, 010, 100), которое также называется кодированием «1 из *N*». При *двоичном кодировании*, как в примере с контроллером светофора, каждому состоянию ставится в соответствие двоичное число (номер этого состояния). Так как K двоичных чисел можно записать в $\log_2 K$ разрядах, системе с *K* состояниями нужно всего $\log_2 K$ битов состояния.

В *прямом кодировании* для каждого состояния используется один бит состояния. По-английски оно называется *one-hot*, потому что только один разряд будет «горячим», то есть только в одном из разрядов содержится логическая единица в любой момент времени. Например, у КА с прямым кодированием и тремя состояниями коды состояний будут 001, 010 и 100. Каждый бит состояния хранится в триггере; таким образом, прямое кодирование требует большего количества триггеров, чем двоичное. Однако при использовании

прямого кодирования схема определения следующего состояния и схема формирования выходных сигналов часто упрощается; таким образом, требуется меньше элементов. Наилучший выбор кодирования зависит от особенностей конкретного автомата.

Пример 3.6 КОДИРОВАНИЕ СОСТОЯНИЙ КОНЕЧНОГО АВТОМАТА

У счетчика с делением на N есть один выход, а входов нет. Выход Y находится в высоком уровне в течение одного периода каждые N периодов тактового сигнала. Другими словами, выход делит тактовую частоту на N.

На **Рис. 3.28** приведена временная диаграмма и диаграмма переходов для счетчика-делителя на 3. Нарисуйте схему такого счетчика с использованием двоичного и прямого кодирований.

Решение: В **Табл. 3.6** и **Табл. 3.7** показаны абстрактные таблицы переходов между состояниями и выхода до кодирования.

Табл. 3.6 Таблица переходов счетчика-делителя на 3

Current State	Next State
S0	S1
S1	S2
S2	S0

Табл. 3.7 Таблица выходов счетчика-делителя на 3

Current State	Output
S0	1
S1	0
S2	0

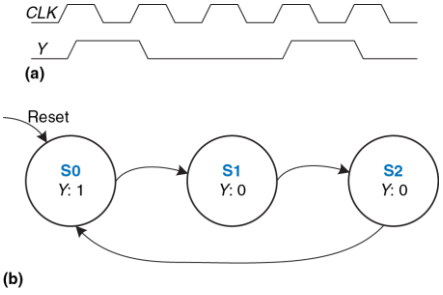


Рис. 3.28 Счетчик-делитель на 3: (a) временная диаграмма, (b) диаграмма переходов

В **Табл. 3.8** сравниваются двоичное и прямое кодирование для трех состояний.

В двоичном кодировании используются два разряда. **Табл. 3.9** является таблицей переходов для этого кодирования. Обратите внимание, что входы отсутствуют; следующее состояние зависит лишь от текущего состояния. Таблицу выхода мы оставим читателю в качестве домашнего задания. Из этих таблиц легко получить выражения для выхода и для следующего состояния:

$$S_1' = \bar{S}_1 S_0$$
$$S_0' = \bar{S}_1 \bar{S}_0 \qquad (3.4)$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

При прямом кодировании используется 3 бита состояния. **Табл. 3.10** – таблица переходов для этого кодирования, а таблицу выхода мы также оставим читателю для самостоятельного выполнения. Выражения для выхода и для следующего состояния будут следующими:

$$\begin{aligned} S_2' &= S_1 \\ S_1' &= S_0 \\ S_0' &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

На **Рис. 3.29** показаны схемы каждого из двух вариантов. Заметим, что «железо» для двоичного кодирования может быть оптимизировано путем использования одного элемента для Y и S_0'. Обратите также внимание на то, что при использовании прямого кодирования для инициализации автомата в состояние S0 в момент сброса необходимо использовать триггеры со входами сброса и установки (resettable and settable). Выбор наилучшей реализации зависит от относительной стоимости элементов (вентилей) и триггеров, но прямое кодирование обычно предпочтительнее в этом конкретном примере.

Еще одной разновидностью прямого кодирования является *one-cold* кодирование, когда бит, соответствующий состоянию системы в текущий момент, сброшен, в то время как остальные биты – установлены: 110, 101, 011.

Табл. 3.8 Двоичное и прямое кодирование счетчика-делителя на 3

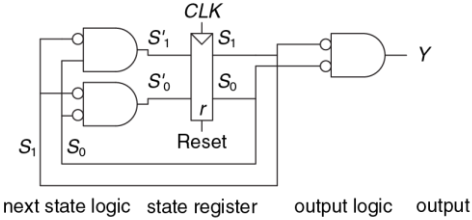
State	One-Hot Encoding			Binary Encoding	
	S_2	S_1	S_0	S'_1	S'_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Табл. 3.9 Таблица переходов с двоичным кодированием

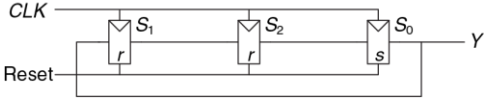
Current State		Next State	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Табл. 3.10 **Таблица переходов с прямым кодированием**

Current State			Next State		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1



(a)



(b)

Рис. 3.29 **Схемы счетчика-делителя на 3 с двоичным (a) и прямым (b) кодированием**

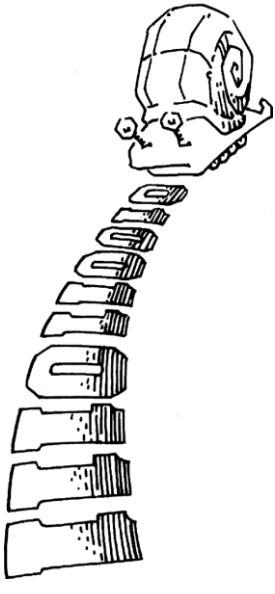
3.4.3 Автоматы Мура и Мили

До сих пор мы рассматривали примеры автоматов Мура, выход в которых зависит лишь от состояния системы. Поэтому на диаграммах переходов для автоматов Мура значения выходов пишутся внутри кружков. Вспомним, что автоматы Мили очень похожи на автоматы Мура, но значения на их выходах могут зависеть он значений на входах таким же образом, как они зависят от текущего состояния системы. Поэтому на диаграммах переходов для автоматов Миля значения выходов пишутся над стрелками. В блоке комбинационной логики, который вычисляет выходные значения, используются значения текущего состояния и входов, как показано на **Рис. 3.32 (b)**.

> Простым способом запомнить разницу между двумя типами конечных автоматов состояний является тот факт, что у автомата Мура обычно больше (Moore – more) состояний, чем у автомата Мили, решающего ту же задачу.

Пример 3.7 СРАВНЕНИЕ АВТОМАТОВ МУРА И МИЛИ

У Алисы есть улитка-робот с автоматом с «мозгами» в виде конечного автомата. Улитка ползает слева направо по перфоленте (перфорированные бумажные ленты активно использовались в вычислительной технике в 80-х гг), содержащей последовательность нулей и единиц. По каждому тактовому импульсу улитка переползает на следующий бит.



Улитка улыбается, если последовательность из двух последних бит, через которые она переползла, равна 01. Спроектируйте автомат, определяющий, когда улитке нужно улыбнуться. На вход *A* поступает значение бита под считывающим устройством улитки. На выходе *Y* устанавливается логическая единица, когда улитка улыбается. Сравните реализации на автоматах Мура и Мили. Нарисуйте временные диаграммы для каждого автомата; изобразите на них вход, состояние и выход; улитка проползает последовательность 0100110111.

Решение: Для автомата Мура требуется три состояния, как показано на **Рис. 3.30 (a)**. Убедитесь в том, что диаграмма переходов изображена верно, в частности, поясните, почему присутствует стрелка из S2 в S1, когда на входе 0.

В отличие от автомата Мура, автомату Мили требуется всего два состояния, что проиллюстрировано **Рис. 3.30 (b)**. Каждая стрелка подписана по принципу *A*/*Y*. *A* – это значение входа, которое вызвало переход, а *Y* – это соответствующий выходной сигнал.

В **Табл. 3.11** и **Табл. 3.1** показана диаграмма переходов и таблица состояний выхода для автомата Мура. Автомату Мура потребуется, как минимум, два бита состояния. Давайте будем использовать двоичное кодирование: S0=00, S1=01, S2=10. **Табл. 3.13** и **Табл. 3.14** являются результатом переписывания **Табл. 3.11** и **Табл. 3.1** с таким кодированием.

Следовательно, значение следующего состояния и значение выхода для этого состояния ни на что не влияют (X) (не показано в таблицах). Мы пользуемся тем, что это состояние нам безразлично, для упрощения выражений.

Далее составим по этим таблицам выражения для следующего состояния и для выхода. Заметим, что эти выражения упрощены с учетом того, что состояния 11 не существует.

$$\begin{aligned} S_1^{'} &= S_0 A \\ S_0^{'} &= \overline{A} \end{aligned} \tag{3.8}$$

$$Y = S_1 \tag{3.9}$$

Табл. 3.15 – сводная таблица переходов и выхода для автомата Мили. Автомату Мили необходим только один бит состояния. Будем использовать двоичное кодирование: S0=0 и S1=1. Перепишем **Табл. 3.15** в **Табл. 3.16**, используя такое кодирование.

По этим таблицам составим выражения для следующего состояния и для выхода.

$$S_0' = \overline{A} \tag{3.10}$$

$$Y = S_0 A \tag{3.11}$$

Схемы автоматов Мили и Мура представлены на **Рис. 3.31**. Временные диаграммы для каждого из автоматов представлены на **Рис. 3.32**.

Каждый из автоматов проходит через разную последовательность состояний. Более того, выход автомата Мили опережает выход автомата Мура на один период, так как он реагирует на вход, а не ждет изменения состояния. Если на выходе автомата Мили поставить триггер, добавив тем самым задержку, то по временным параметрам такая конструкция станет эквивалентной автомату Мура. Когда будете выбирать тип автомата для вашего проекта, подумайте, в какой момент вы хотите увидеть реакцию выходов.

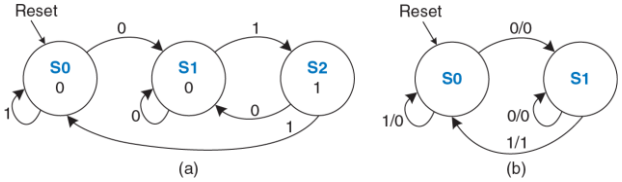


Рис. 3.30 Диаграммы переходов КА: (a) автомат Мура, (b) автомат Мили

Табл. 3.11 Таблица переходов автомата Мура

Current State *S*	Input *A*	Next State *S′*
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Табл. 3.12 Таблица выходов автомата Мура

Current State *S*	Output *Y*
S0	0
S1	0
S2	1

Табл. 3.13 Таблица переходов автомата Мура с кодированием состояний

Current State S_1	Current State S_0	Input A	Next State S'_1	Next State S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Табл. 3.14 Таблица выходов автомата Мура с кодированием состояний

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Табл. 3.15 Таблица переходов и выходов автомата Мили

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Табл. 3.16 Таблица переходов и выходов автомата Мили с кодированием состояний

Current State S_0	Input A	Next State $S'0$	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	0

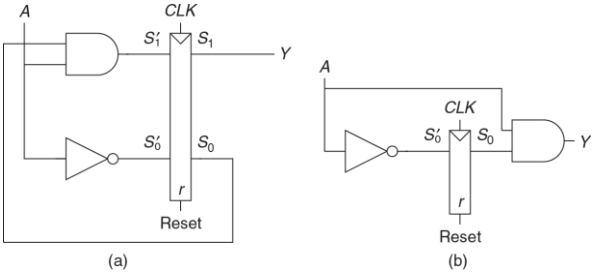


Рис. 3.31 Схемы КА: (a) Мура, (b) Мили

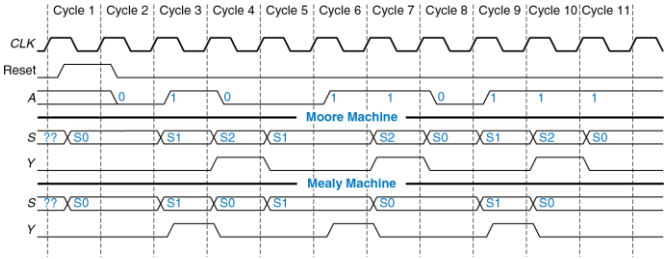


Рис. 3.32 Временные диаграммы автомата Мура и автомата Мили

3.4.4 Декомпозиция конечных автоматов

Проектирование сложных конечных автоматов часто упрощается, если их можно разбить на несколько более простых автоматов, взаимодействующих друг с другом таким образом, что выход одних автоматов является входом других. Такое применение принципов иерархической организации и модульного проектирования называется *декомпозицией* конечных автоматов.

Пример 3.8 МОДУЛЬНЫЕ И НЕМОДУЛЬНЫЕ КОНЕЧНЫЕ АВТОМАТЫ

Модифицируйте контроллер светофора из **Раздела 3.4.1** так, чтобы в нем появился режим «парада». В этом режиме светофор на Беговой улице остается зеленым, когда команда и зрители идут на футбольные игры разрозненными группами. У контроллера появляются еще два входа: P и R. Получая сигнал P, контроллер хотя бы на один цикл входит в режим парада, а получая сигнал P, – хотя бы на один цикл выходит из этого режима. Находясь в режиме парада, контроллер проходит свою обычную последовательность переключений до тех пор, пока L_B не станет зеленым, а затем остается в этом состоянии до тех пор, пока режим парада не закончится.

Сначала нарисуем диаграмму переходов для одного-единственного КА, как показано на **Рис. 3.33 (a)**. Затем нарисуем диаграмму переходов для двух взаимодействующих КА, как показано на **Рис. 3.33 (b)**. Автомат выбора режима выставляет выход M в единицу, когда он переходит в режим парада. Автомат световых сигналов управляет светофорами в зависимости от M и датчиков движения T_A и T_B.

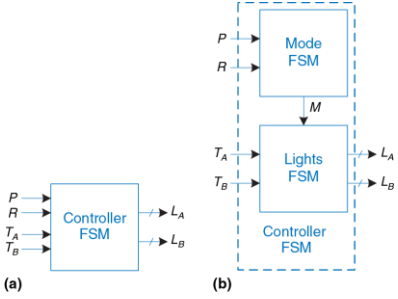


Рис. 3.33 (a) Немодульная и (b) модульная модель КА модифицированного контроллера светофора

Решение: На **Рис. 3.34 (a)** представлена реализация с одним-единственным автоматом. Состояния S0-S3 отвечают за нормальный режим работы, а состояния S4-S7 – за режим парада. Две половины диаграммы практически идентичны, за исключением того, что в режиме парада КА остается в состоянии S6, включая зеленый свет на Беговой улице. Входы *P* и *R* управляют переходами между этими двумя половинами. Такой автомат слишком сложный и тяжелый в разработке. На **Рис. 3.34 (b)** представлена модульная реализация КА. У КА выбора режима будет всего два состояния: когда светофор в нормальном и когда – в парадном режиме. Автомат световых сигналов модифицирован таким образом, чтобы оставаться в состоянии S2, пока *M*=1.

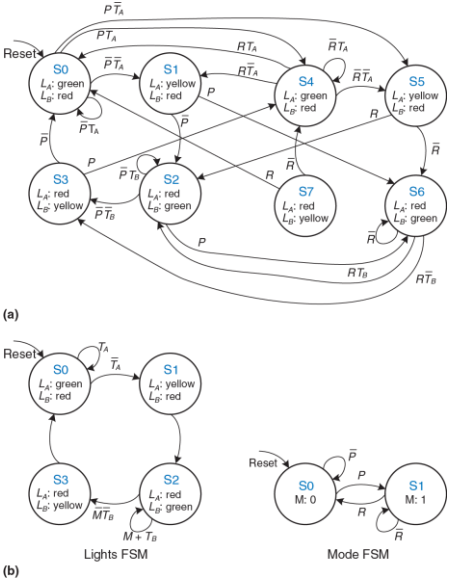


Рис. 3.34 **Диаграммы переходов: (a) немодульная, (b) модульная**

3.4.5 Восстановление конечных автоматов по электрической схеме

Восстановление конечных автоматов по электрической схеме практически является процессом, обратным проектированию КА. Этот процесс необходим, например, при рассмотрении проекта с неполной документацией или для реверсивного проектирования чьей-то системы.

- Проанализируйте схему, возможные состояния входов, выходов и регистра состояний.
- Составьте выражения для следующего состояния и для выходов.
- Составьте таблицу выходов и таблицу переходов.
- Вычеркните из таблицы переходов состояния, в которые система никогда не попадает.
- Присвойте имя каждому используемому набору бит-состояний.
- Перепишите таблицы выходов и переходов, используя эти обозначения.
- Нарисуйте диаграмму переходов.
- Опишите словами то, что делает автомат.

На последнем шаге не бойтесь развернуто описывать цели и функции автомата, чтобы избежать простого переформулирования каждого перехода из диаграммы переходов.

Пример 3.9 ВОССТАНОВЛЕНИЕ КА ПО ЕГО СХЕМЕ

Алиса Хакер приехала домой, но в ее кодовом замке заменили проводку, и ее старый код больше не работает. К замку приколот лист бумаги со схемой, которая приведена на **Рис. 3.35**.

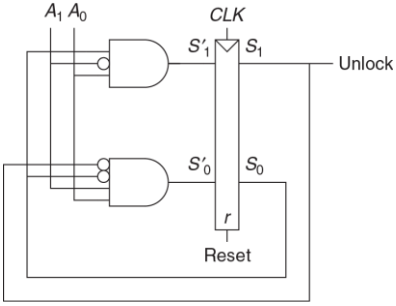


Рис. 3.35 **Схема автомата из Примера 3.9**

Алиса полагает, что схема может быть конечным автоматом и решает восстановить диаграмму переходов, чтобы узнать, поможет ли ей это попасть внутрь.

Решение: Алиса начинает изучать схему. Входом является $A_{1:0}$, а выходом – разблокировка. Биты состояний уже обозначены на **Рис. 3.35**. Это автомат Мура, так как выходы зависят только от битов состояния. Прямо по схеме она записывает выражения для следующего состояния и для выхода:

$$\begin{aligned} S_1' &= S_0 \overline{A}_1 A_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 A_1 A_0 \\ Unlock &= S_1 \end{aligned} \tag{3.12}$$

Затем она составляет таблицы переходов и выхода (**Табл. 3.17**, **Табл. 3.18**) по написанным уравнениям. Сначала Алиса расставляет единицы (последние два столбца таблицы) по **выражениям (3.12)**, а в остальных местах пишет нули.

Табл. 3.17 Таблица следующих состояний, восстановленная по схеме на Рис. 3.35

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Табл. 3.18 Таблица выходов, восстановленная по схеме на Рис. 3.35

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	1
1	1	1

Алиса сокращает таблицу путем вычеркивания неиспользуемых состояний и путем комбинирования строк, используя при этом безразличные разряды. Состояние $S_{1:0}$=11 нигде не встречается в **Табл. 3.17** как возможное следующее состояние, поэтому строки с этим состоянием можно вычеркнуть. Для текущего состояния $S_{1:0}$=10 следующее состояние всегда $S_{1:0}$=00, независимо от входов. **Табл. 3.19** и **Табл. 3.20** являются результатом сокращения.

Табл. 3.19 Сокращенная таблица следующих состояний

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0		1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Табл. 3.20 Сокращенная таблица выходов

Current State		Output
S_1	S_0	*Unlock*
0	0	0
0	1	0
1	0	1

Она присваивает имена для каждой комбинации битов состояний: S0 это $S_{1:0}$=00, S1 это $S_{1:0}$=01, а S2 это $S_{1:0}$=10. Алиса переписывает **Табл. 3.19** и **Табл. 3.20** в **Табл. 3.21** и **Табл. 3.22**, используя эти обозначения.

Табл. 3.21 Символьная таблица следующих состояний

Current State *S*	Input *A*	Next State *S'*
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Табл. 3.22 Символьная таблица выходов

Current State *S*	Output *Unlock*
S0	0
S1	0
S2	1

По **Табл. 3.21** и **Табл. 3.22** она рисует диаграмму переходов, которая представлена на **Рис. 3.36**. Изучив ее, она приходит к выводу, что конечный автомат разблокирует дверь после обнаружения поданных на вход $A_{1:0}$ трёх единиц. Затем дверь снова блокируется. Алиса пробует ввести этот код, и дверь открывается!

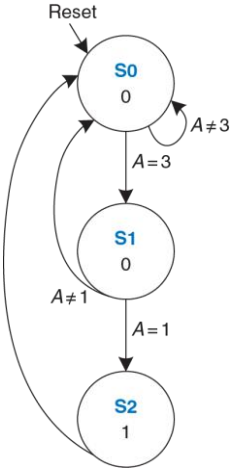


Рис. 3.36 Диаграмма переходов полученного КА

3.4.6 Обзор конечных автоматов

Конечные автоматы являются мощным инструментом для систематического проектирования последовательностных схем по техническому заданию. Используйте следующую последовательность действий для создания КА:

- Определите входы и выходы.
- Нарисуйте диаграмму переходов.
- Для автомата Мура:
  - Составьте таблицу переходов.
  - Составьте таблицу выходов.
- Для автомата Мили:
  - Составьте объединенную таблицу выходов и переходов.
- Выберите метод кодирования состояний – выбранный метод повлияет на схемотехническую реализацию.
- Составьте булевы выражения для следующего состояния и для выходной комбинационной схемы.
- Нарисуйте принципиальную схему.

Мы неоднократно будем использовать КА для создания сложных цифровых систем на протяжении всей этой книги.

3.5 СИНХРОНИЗАЦИЯ ПОСЛЕДОВАТЕЛЬНОСТНЫХ СХЕМ

Вспомните, что триггер копирует сигнал с *D*-входа на *Q*-выход по переднему фронту тактового сигнала. Этот процесс называется *фиксацией* (*sampling*) *D*-сигнала по фронту тактового импульса. Поведение триггера корректно, если сигнал на *D*-входе стабилен (равен 0 или 1 и не изменяется) в течение переднего фронта тактового сигнала. Но что произойдет, если сигнал *D* не будет стабилен во время изменения тактового сигнала?

Эта ситуация аналогична той, которая возникает при спуске затвора фотокамеры. Представьте, что вы пытаетесь снять прыжок лягушки с плавающего листа кувшинки в озеро. Если вы нажмете на спуск перед прыжком, то на фотографии вы увидите лягушку на листе кувшинки. Если вы нажмете на спуск после прыжка, то на фотографии будет рябь на воде. Но если вы нажмете на спуск во время прыжка, то на фотографии вы увидите смазанное

изображение вытянутой вдоль направления прыжка лягушки. Одной их характеристик фотокамеры является *апертурное время*, в течение которого фотографируемый объект должен быть неподвижен, чтобы на фотографии сформировалось его резкое изображение. Аналогично, последовательностный элемент имеет апертурное время до и после фронта тактового сигнала, в течение которого его информационные входные сигналы должны быть стабильными, чтобы на выходе триггера сформировался корректный сигнал.

Часть апертурного времени последовательностного элемента до фронта тактового импульса называется временем *предустановки* (*setup time*), после фронта – *временем удержания* (*hold time*). Подобно статической дисциплине, которая разрешает использование логических уровней только за пределами запретной зоны, динамическая дисциплина позволяет использовать только те сигналы, которые изменяются вне апертурного времени. При выполнении требований динамической дисциплины мы можем оперировать дискретными единицами времени, которые называются тактовыми циклами, аналогично тому, как мы оперируем с дискретными логическими уровнями 1 и 0. Сигнал может изменяться и осциллировать в течение некоторого ограниченного промежутка времени. При выполнении требований динамической дисциплины важно лишь его значение в конце цикла тактового сигнала, когда он уже принял стабильное

значение. Следовательно, для описания сигнала *A* можно использовать его величину *A*[*n*] в конце *n*-го цикла тактового импульса, где *n* – целое число, вместо его величины *A*(*t*) в произвольный времени *t*, где *t* – действительное число.

Период тактовых импульсов должен быть достаточно большим, чтобы переходные процессы всех сигналов успели завершиться. Это требование ограничивает быстродействие всей системы. В реальных системах тактовые импульсы поступают на входы триггеров неодновременно. Этот разброс по времени, который называется расфазировкой или разбросом фаз тактового сигнала (clock skew), заставляет разработчиков дополнительно увеличивать период тактовых сигналов.

Иногда невозможно удовлетворить требованиям динамической дисциплины, особенно в устройствах сопряжения цифровой системы с реальным миром. Например, рассмотрим схему, к входу которой подключена кнопка. Обезьяна может нажать на кнопку как раз во время фронта тактового импульса. Это может привести к возникновению явления, которое называется метастабильностью, при этом триггер оказывается в промежуточном состоянии между 0 и 1, причем переход в корректное логическое состояние (0 или 1) может происходить бесконечно долго. Решением проблемы асинхронных входов является использование синхронизатора, на выходе которого некорректный

логический уровень может появиться с очень малой (но не нулевой) вероятностью.

Эти идеи будут детально рассмотрены в оставшейся части раздела.

3.5.1 Динамическая дисциплина

До сих пор мы рассматривали функциональные спецификации последовательностных схем. Вспомните, что синхронные последовательностные схемы, такие так триггеры или конечные автоматы, имеют также и временную спецификацию, пример которой показан на **Рис. 3.37**.

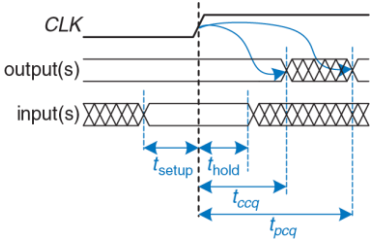


Рис. 3.37 Временная спецификация синхронной последовательностной схемы

После перехода 0→1 тактового сигнала (переднего фронта тактового импульса) выход (или выходы) схемы могут начать изменяться не ранее, чем через время tccq (задержка реакции clock-to-*Q*, contamination delay clock-to-*Q*[3]), и должны принять стационарное значение не позднее чем через время t_{pcq} (задержка распространения clk-to-*Q*, (propagation delay clock-to-*Q*). Эти величины представляют собой наименьшую и наибольшую задержки схемы, соответственно. Для того, чтобы фиксация была выполнена корректно, информационный вход (или входы) схемы должен быть стабильным в течение некоторого *времени предустановки* (*setup time*) t_{setup} перед передним фронтом тактового сигнала и не должны изменяться в течение *времени удержания* (*hold time*) t_{hold} после переднего фронта тактового сигнала. Сумма времен *предустановки* и удержания называется *апертурным временем* схемы, это общее время, в течение которого информационный входной сигнал должен быть стабилен для его фиксации на выходе.

Динамическая дисциплина требует, чтобы входы синхронной последовательностной схемы были стабильны в течение времени предустановки до и времени удержания после фронта тактового импульса. Выполнение этих требований гарантирует, что в процессе

[3] В российской, да и зарубежной нормативно-технической документации чаще всего используется только задержка распространения (propagation delay), но указываются ее минимальное и максимальное значения

фиксации значения информационного входа триггером он не будет изменяться. Поскольку мы будем рассматривать только установившиеся значения входных сигналов в моменты времени, когда они фиксируются, мы может считать сигналы дискретными как по уровню, так и по времени.

3.5.2 Временные характеристики системы

Периодом тактового сигнала или *длительностью цикла синхронизации,* T_c, называется промежуток времени между передними фронтами последовательных тактовых импульсов. Обратная величина, $f_c = 1/T_c$, называется *тактовой частотой*. Увеличение тактовой частоты без изменения остальных параметров схемы приводит к увеличению ее производительности. Частота измеряется в Герцах (Гц), или в циклах за одну секунду: 1 мегагерц (МГц) = 10^6 Гц, and 1 гигагерц (ГГц) = 10^9 Гц.

За тридцать лет, прошедших со времени, когда семья одного из авторов купила компьютер Apple II+, до момента написания книги, тактовая частота микропроцессора увеличилась с 1 МГц до нескольких ГГц, более чем в тысячу раз. Это увеличение быстродействия компьютеров частично объясняет революционные изменения, которые благодаря им произошли в обществе.

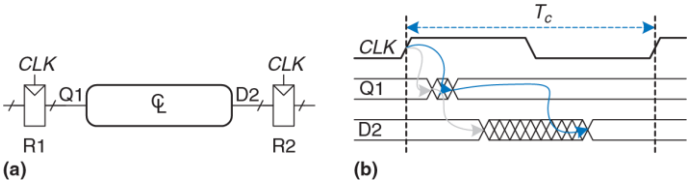


Рис. 3.38 **Тракт между регистрами и временная диаграмма**

На **Рис. 3.38 (а)** показана характерная структура тракта обработки информации синхронной последовательностной схемы, для которой мы рассчитаем период тактового сигнала. По переднему фронту тактового импульса на выходе регистра R1 формируется выходной сигнал (или сигналы) *Q*1. Эти сигналы поступают на вход блока комбинационной логики, выходные сигналы этого блока поступают на вход (или входы) *D*2 регистра R2. Как показано на **Рис. 3.38 (b)**, выходной сигнал блока может начать изменяться не ранее окончания времени реакции после завершения изменения его входного сигнала и принимает окончательное значение спустя максимальное время задержки распространения от момента установления входного сигнала. Серые стрелки показывают минимальную задержку с учетом R1 и комбинационной логики, а синие – максимальную задержку распространения в тракте регистр R1 – комбинационная логика.

Мы проанализируем временные ограничения с учетом времен предустановки и удержания второго регистра, R2.

Ограничение времени предустановки

На Рис. 3.39 на временной диаграмме приведена только максимальная задержка в тракте обработке информации, которая обозначена синими стрелками.

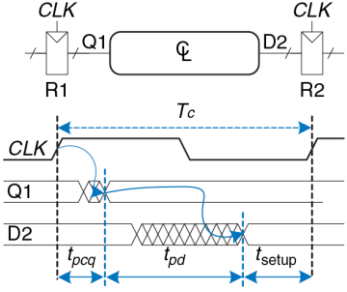


Рис. 3.39 Максимальная задержка для ограничения времени предустановки

Для выполнения ограничения по времени предустановки регистра R2, сигнал *D*2 должен установиться не позднее, чем за время предустановки до фронта следующего тактового импульса. Таким

образом, мы можем получить выражение для минимальной длительности периода синхросигнала:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \tag{3.13}$$

При проектировании коммерческих продуктов период тактового сигнала будущего изделия часто задается из соображений конкурентоспособности руководителем отдела разработок или отделом маркетинга. Более того, задержка распространения сигнала триггером от фронта тактового сигнала до выхода (Clock-to-Q) и время предустановки t_{pcq} и t_{setup} определены производителем. Следовательно, неравенство **(3.13)** следует преобразовать для определения максимальной задержки распространения комбинационной схемы, поскольку обычно именно это – единственный параметр, который может изменять проектировщик:

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}) \tag{3.14}$$

Слагаемое в скобках, $t_{pcq} + t_{\text{setup}}$, называется *потерями на упорядочение* (*sequencing overhead*). В идеальном случае весь период тактового сигнала может быть затрачен на вычисления в комбинационной логике (время t_{pd}). Однако, потери на упорядочение в триггерах уменьшают это время. Неравенство **(3.14)** называется *ограничением времени предустановки* или *ограничением максимальной задержки*, поскольку

оно зависит от времени предустановки и ограничивает максимальную задержку распространения в комбинационной логической схеме.

Если задержка распространения в комбинационной схеме слишком велика, то вход *D*2 может не успеть принять свое установившееся состояние ко времени, когда регистр R2 ожидает стабильности и фиксирует его. Таким образом, R2 может зафиксировать некорректный результат или даже логический уровень в запретной зоне. В таком случае схема будет работать некорректно. Проблему можно решить увеличением периода тактового сигнала или пересмотром комбинационной схемы с целью добиться меньшей задержки распространения.

Ограничение времени удержания

Регистр R2 на **Рис. 3.38 (а)** имеет также *ограничение времени удержания*. Его вход, *D*2, не должен изменяться в течение некоторого времени t_{hold} после переднего фронта тактового импульса.

В соответствии с **Рис. 3.40**, *D*2 может измениться через $t_{ccq} + t_{cd}$ после переднего фронта тактового импульса. Следовательно, можно записать:

$$t_{ccq} + t_{cd} \geq t_{hold} \tag{3.15}$$

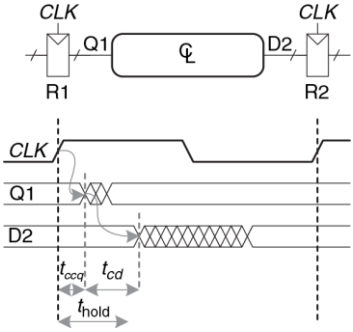


Рис. 3.40 **Минимальная задержка для ограничения времени удержания**

Как и ранее, характеристики используемого в схеме триггера t_{ccq} и t_{hold} обычно находятся вне влияния разработчика схемы. После простых преобразований мы можем записать неравенство для минимальной задержки комбинационной логической схемы:

$$t_{cd} \geq t_{hold} - t_{ccq} \tag{3.16}$$

Неравенство **(3.16)** также называется *ограничением времени удержания* или *ограничением минимальной задержки*, потому что оно ограничивает минимальную задержку комбинационной схемы.

Мы предполагаем, что при соединении логических элементов между собой временные проблемы синхронизации не возникают. В частности, мы считаем, что при непосредственном последовательном соединении двух триггеров, как показано на **Рис. 3.41**, проблемы, обусловленные временем удержания, не возникают.

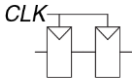


Рис. 3.41 Непосредственное последовательное соединение триггеров

В этом случае, вследствие отсутствия комбинационной логики между триггерами, $t_{cd} = 0$. При такой подстановке неравенство **(3.16)** сводится к требованию:

$$t_{\text{hold}} \leq t_{ccq} \tag{3.17}$$

Иными словами, время удержания надежного триггера должно быть меньше, чем его задержка реакции. Часто триггеры проектируются так, что $t_{\text{hold}} = 0$, следовательно, неравенство **(3.17)** всегда выполняется. В этой книге, если не указано обратное, мы будем считать такое предположение истинным- и игнорировать ограничение времени удержания.

Тем не менее, ограничения времени удержания критически важны. Если они нарушаются, то единственным решением является увеличение задержки реакции комбинационной схемы, что требует ее перепроектирования. Такие нарушения, в отличие от нарушений ограничений времени предустановки, не могут быть исправлены изменением периода тактового сигнала. Перепроектирование интегральной микросхемы и производство ее исправленного варианта занимает несколько месяцев и требует затрат в несколько миллионов долларов при современных технологиях, поэтому к *нарушениям ограничения времени удержания* нужно относиться крайне серьезно.

Заключение

Последовательностные схемы имеют ограничения времен предустановки и удержания, которые устанавливают максимальную и минимальную задержки в комбинационной логической схеме между триггерами. Современные триггеры обычно спроектированы так, что минимальная задержка в комбинационной логике равна нулю, то есть триггеры могут быть размещены непосредственно друг за другом. Максимальная задержка ограничивает число последовательных логических элементов, включенных один за другим в критическом пути быстродействующей схемы.

Пример 3.10 ВРЕМЕННОЙ АНАЛИЗ

Бен Битдидл спроектировал схему, которая показана на **Рис. 3.42**. В соответствии со спецификацией компонентов, которые он использует, задержка реакции тактовый вход-выход триггеров равна 30 пс, а задержка распространения – 80 пс. Они имеют время предустановки 50 пс. и время удержания – 60 пс. У логических элементов задержка распространения равна 40 пс, задержка реакции – 25 пс. Помогите Бену определить максимальную тактовую частоту его схемы и выяснить, могут ли происходить нарушения ограничения времени удержания в ней. Этот процесс называется временным анализом.

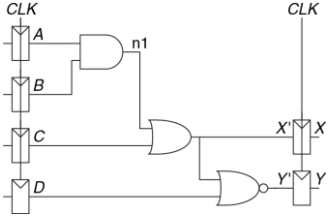


Рис. 3.42 **Пример схемы для временного анализа**

Решение: На **Рис. 3.43 (а)** приведены временные диаграммы сигналов, которые показывают, когда они могут изменяться. Сигнал на входы *A* – *D* поступает с регистров, поэтому они могут измениться через короткое время после переднего фронта сигнала *CLK*.

Критический путь возникает, когда $B = 1$, $C = 0$, $D = 0$ и A изменяется от 0 к 1, что приводит к переключению n1 в 1, X' в 1, Y' в 0, как показано на **Рис. 3.43 (b)**. В этот путь входят задержки трех логических элементов. Для оценки задержки в критическом пути будем считать, что задержка каждого элемента равна задержке распространения. Сигнал Y′ должен установиться ранее следующего переднего фронта *CLK*. Следовательно, минимальная длительность цикла равна

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250\text{ps} \qquad (3.18)$$

Максимальная тактовая частота равна $f_c = 1/T_c = 4$ ГГц.

Короткий (по времени прохождения сигналом) путь возникает, когда $A = 0$ и C переключается в 1, как показано на **Рис. 3.43 (c)**.

Для короткого пути будем считать, что каждый логический элемент переключается сразу после завершения задержки реакции. Этот путь включает в себя только один элемент, поэтому переключение может наступить через $t_{ccq} + t_{cd} = 30 + 25 = 55$ пс. Однако, следует помнить, что время удержания триггера равно 60 пс, это означает, что сигнал X′ обязательно должен быть стабильным в течение 60 пс после переднего фронта тактового сигнала *CLK*, чтобы триггер смог надежно зафиксировать величину сигнала X′. В этом случае в течение первого переднего фронта *CLK* $X' = 0$, то есть триггер должен зафиксировать 0. Однако, поскольку X' не поддерживается стабильным в течение времени удержания, действительное значение X будет непредсказуемым. В этой схеме происходит нарушение ограничений времени удержания, и ее поведение непредсказуемо при любой тактовой частоте.

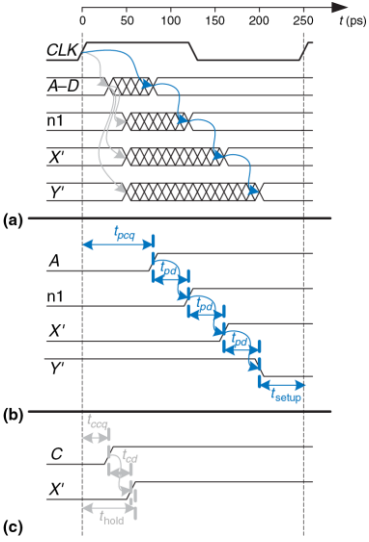


Рис. 3.43 **Временная диаграмма: (a) общий случай, (b) критический путь, (c) короткий путь**

Пример 3.11 ИСПРАВЛЕНИЕ НАРУШЕНИЙ ВРЕМЕНИ УДЕРЖАНИЯ

Алиса Хакер предлагает исправить схему Бена путем добавления буферных элементов, которые будут замедлять прохождение сигнала через короткий путь, как показано на **Рис. 3.44**. Буфера имеют такую же задержку, как и остальные логические вентили. Определите максимальную тактовую частоту и проверьте, будут ли возникать проблемы, связанные со временем удержания.

Решение: На **Рис. 3.45** приведены временные диаграммы, которые показывают, когда сигналы могут изменяться. Критический путь от A до Y не изменился, потому что он не проходит через буферы. Следовательно, максимальная тактовая частота равна, как и ранее, 4ГГц. Однако время прохождения сигнала через короткий путь будет увеличено на величину минимальной задержки буферов. Теперь X' не изменится в течение $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ пс после фронта тактового сигнала. Таким образом, X' будет стабилен в течение времени удержания 60 пс, то есть схема будет работать правильно.

В этом примере аномально большое время удержания было использовано только для демонстрации сути проблем, связанных со временем удержания. Большинство триггеров спроектированы так, что $t_{\text{hold}} < t_{ccq}$, это позволяет избежать таких проблем. Однако, в некоторых высокопроизводительных микропроцессорах, включая Pentium 4, вместо триггеров используется элемент, который называется импульсная защелка (pulsed latch). Импульсная защелка ведет себя подобно обычному триггеру, но имеет малую задержку тактовый вход-выход и большое время удержания. Добавление буферов позволяет часто, но не всегда, устранить проблемы, связанные с ограничением времени удержания, без увеличения времени прохождения сигнала по критическому пути.

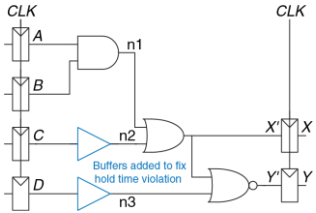


Рис. 3.44 Исправленная схема, в которой отсутствуют нарушения ограничения времени удержания

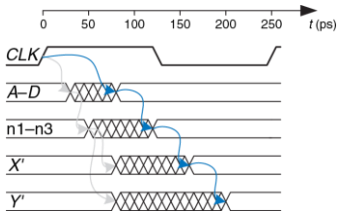


Рис. 3.45 Временная диаграмма схемы с буферами, в которой отсутствуют нарушения ограничения времени удержания

3.5.3 Расфазировка тактовых сигналов

В предыдущих разделах предполагалось, что тактовые импульсы поступают на все регистры в одно и то же время. В действительности существует некоторый разброс этого времени. Эта неодновременность фронтов называется *расфазировкой*.

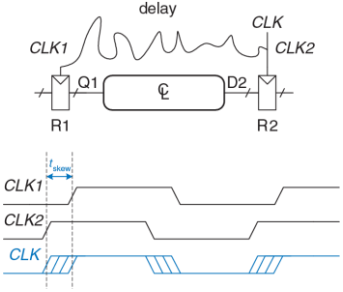


Рис. 3.46 Расфазировка тактовых сигналов, обусловленная задержками в межсоединениях

Например, длина проводников, по которым тактовые сигналы поступают на разные регистры, может быть разной, это приводит к разным временам задержки, как показано на **Рис. 3.46**.

Шум также приводит к различным задержкам. Стробирование тактовых сигналов, которое было описано в **разделе 3.2.5**, приводит к их дополнительной задержке. Если в схеме используются стробированные и нестробированные тактовые сигналы, то между ними будет существенное рассогласование. На **Рис. 3.46** сигнал *CLK*2 будет опережать по времени сигнал *CLK*1 из-за сложного пути тактового сигнала между регистрами. Если трассировка цепи тактового сигнала будет выполнена по-другому, ситуация может быть противоположной, *CLK*2 будет отставать от сигнала *CLK*1. При выполнении временного анализа мы рассматриваем наихудший случай, что позволяет гарантировать, что схема будет работать при всех условиях.

Учет расфазировки изменяет временную диаграмму, которая была показана на **Рис. 3.38**, модифицированная диаграмма приведена на **Рис. 3.47**.

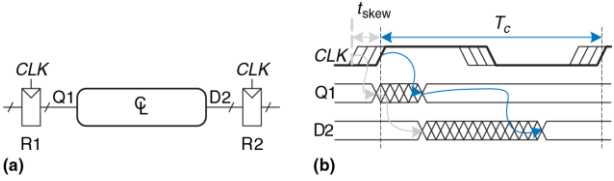


Рис. 3.47 **Временная диаграмма с учетом расфазировки тактовых импульсов**

Жирной линией показана максимальная задержка тактового сигнала, тонкие линии показывают, что синхросигнал, может появиться на t_{skew}.раньше.

Вначале рассмотрим ограничение времени предустановки, соответствующие диаграммы приведены на **Рис. 3.48**.

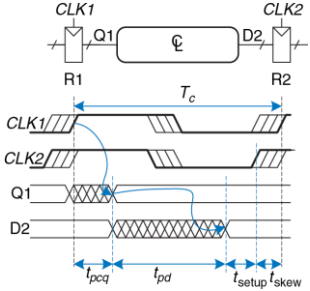


Рис. 3.48 Ограничение времени предустановки с учетом расфазировки тактовых импульсов

В худшем случае на регистр R1 поступает тактовый сигнал с наибольшей задержкой, а на R2 – с наименьшей, что оставляет минимальное время для прохождения данных через комбинационную схему между регистрами.

На вход регистра R2 данные поступают через регистр R1 и комбинационную логику, они должны прийти к стационарному состоянию перед началом их фиксации регистром R2. Следовательно, можно сделать вывод, что

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \tag{3.19}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \tag{3.20}$$

Далее мы рассмотрим ограничение времени удержания (см. **Рис. 3.49**). В худшем случае на регистр R1 поступает тактовый сигнал с наименьшей задержкой, а на R2 – с наибольшей. Данные могут быстро пройти через регистр R1 и комбинационную логику, но должны поступить на вход регистра R2 не ранее окончания времени удержания после переднего фронта тактового импульса.

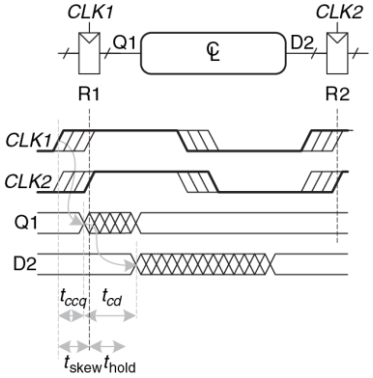


Рис. 3.49 Ограничение времени удержания с учетом расфазировки тактовых импульсов

Таким образом, можно записать:

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} \quad (3.21)$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \quad (3.22)$$

В итоге, расфазировка тактовых импульсов приводит к эффективному увеличению как времени предустановки, так и времени удержания. Это, в свою очередь, приводит к росту потерь на упорядочение и уменьшает время, доступное для обработки данных комбинационной схемой. Даже если $t_{hold} = 0$, пара последовательно соединенных триггеров будет нарушать неравенство **(3.22)**, если if $t_{skew} > t_{ccq}$. Чтобы предотвратить такие серьезные нарушения ограничений времени удержания, проектировщик должен ограничивать расфазировку тактовых сигналов. Иногда триггеры специально проектируются медленными (время t_{ccq} велико), чтобы избежать проблем времени удержания, даже если расфазировка тактовых сигналов существенна.

Пример 3.12 ВРЕМЕННОЙ АНАЛИЗ РАСФАЗИРОВКИ ТАКТОВЫХ

Выполните задание **примера 3.10** в предположении, что в системе есть разфазировка тактовых импульсов величиной 50 пс.

Решение: Критический путь остается без изменений, но эффективное время предустановки увеличивается из-за расфазировки. Следовательно, минимальный период тактового сигнала равен

$$T_c \geq t_{pcq} + 3t_{pd} + t_{setup} + t_{skew} = 80 + 3.40 + 50 + 50 = 300\text{ps} \quad (3.23)$$

Максимальная частота тактового сигнала будет $f_c = 1/T_c = 3.33$ ГГц.

Короткий путь также остается без изменений, время прохождения сигнала по нему равно 55 пс. Эффективное время удержания увеличивается на величину расфазировки до 60 + 50 = 110 пс, что существенно больше 55 пс. Следовательно, в схеме будет нарушено ограничение времени удержания, и она будет некорректно работать при любой частоте тактового сигнала. Напомним, что в этой схеме ограничение времени удержания было нарушено и без расфазировки. Расфазировка тактовых сигналов только ухудшила ситуацию.

Пример 3.13 ИСПРАВЛЕНИЕ НАРУШЕНИЯ ОГРАНИЧЕНИЯ ВРЕМЕНИ

Повторите упражнение из **примера 3.11** в предположении, что в системе есть расфазировка тактовых импульсов величиной 50 пс.

Решение: Критический путь не изменяется, поэтому максимальная тактовая частота остается равной 3.33 ГГц. Время прохождения сигнала по короткому пути увеличивается до 80 пс. Это все еще меньше, чем $t_{hold} + t_{skew} = 110$ пс, следовательно, в схеме нарушаются ограничения времени удержания. Чтобы решать проблему, в схему следует добавить еще несколько буферов. Поскольку они входят в критический путь, то максимальная тактовая частота уменьшится. В качестве альтернативы, можно рассмотреть использование других триггеров с меньшим временем удержания.

3.5.4 Метастабильность

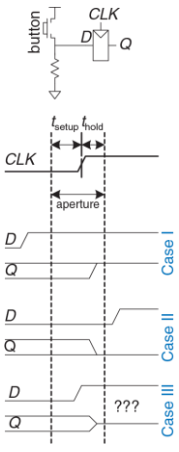


Рис. 3.50 Входной сигнал, который изменяется до, после или в течение апертурного времени

Как было указано ранее, не всегда можно гарантировать, что вход последовательностной схемы будет стабилен в течение апертурного времени, особенно если входной сигнал поступает от внешнего асинхронного источника. Рассмотрим кнопку, подсоединенную к входу триггера, как показано на **Рис. 3.50**. Когда кнопка не нажата, $D = 0$. Когда кнопка нажата, $D = 1$. Обезьяна может нажимать кнопку в любой произвольный момент времени по отношению к фронту тактового сигнала. Мы хотим знать сигнал на выходе Q после переднего фронта сигнала *CLK*. В случае I, когда кнопка нажимается задолго до фронта *CLK*, $Q = 1$. В случае II, кнопка нажимается только намного позже фронта *CLK*, $Q = 0$. Но в случае III, когда кнопка нажимается в промежутке, который охватывает время предустановки перед фронтом тактового импульса и время удержания после него, входной сигнал нарушает динамическую дисциплину и выход будет неопределенным.

Метастабильное состояние

Когда состояние информационного входа триггера изменяется в течение апертурного времени, на его выходе Q может на некоторое время появиться напряжение в диапазоне от 0 до V_{DD}, то есть в запретной зоне. Такое состояние называется *метастабильным*. Со временем выход триггера перейдет в *стабильное состояние* 0 или 1. Однако *время разрешения*, необходимое для достижения стабильного состояния, не ограничивается.

Метастабильное состояние триггера подобно состоянию шарика на вершине между двумя впадинами, как показано на **Рис. 3.51**. Положения во впадинах являются стабильными, поскольку шарик будет находиться в них неограниченно долго при отсутствии внешнего возмущения. Положение на вершине возвышенности называется метастабильным, потому что шарик будет находиться в нем только при условии идеальной балансировки. Но, поскольку в мире нет ничего совершенного, со временем шарик скатится в одну из впадин. Необходимое для этого время зависит от степени первоначальной

балансировки шарика. Каждое бистабильное устройство имеет метастабильное состояние между двумя стабильными.

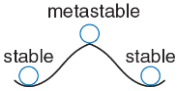


Рис. 3.51 **Стабильное и метастабильное состояния**

Время разрешения

Если вход триггера изменяется в произвольный момент цикла тактового сигнала, то время разрешения, t_{res}, необходимое для перехода в стабильное состояние, также является случайной величиной. Если вход изменяется вне апертурного времени, то $t_{res} = t_{pcq}$. Но если произойдет изменение входа в апертурное время, t_{res} может быть существенно больше.

Теоретическое и экспериментальное рассмотрение (см. **раздел 3.5.6**) показывает, что вероятность того, что время разрешения превышает некоторое время t, экспоненциально падает с ростом t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3.24}$$

где T_c – период тактового сигнала, T_0 и τ – характеристики триггера. Выражение справедливо, только если t намного больше, чем t_{pcq}.

Интуитивно понятно, что отношение T_0/T_c описывает вероятность того, что вход изменится в неудачное время (то есть в апертурное время); эта вероятность уменьшается с ростом периода тактового сигнала T_c. τ – временная константа, которая показывает, насколько быстро триггер выходит из метастабильного состояния; она связана с задержкой в перекрестно соединенных вентилях триггера.

Таким образом, если вход бистабильного устройства, такого как триггер, изменяется в течении апертурного времени, его выход может некоторое время находиться в метастабильном состоянии, прежде чем перейти в стабильное состояние 0 или 1. Время перехода в стабильное состояние не ограничено, потому что для любого конечного времени t вероятность того, что триггер все еще находится в метастабильном состоянии, не равна нулю. Однако, эта вероятность экспоненциально падает с ростом t. Следовательно, если подождать достаточно долго, намного больше, чем t_{pcq}, то с весьма высокой вероятностью можно ожидать того, что триггер достигнет корректного логического состояния.

3.5.5 Синхронизаторы

Наличие асинхронных входов цифровой системы, которые принимают информацию из внешнего мира, неизбежно. Например, сигналы, которые формирует человек, асинхронны. Такие асинхронные входы, если к ним относиться небрежно, могут привести к появлению метастабильных состояний в системе, что приведет к ее непредсказуемым отказам, которые крайне сложно отследить и исправить. При наличии асинхронных входов- проектировщик системы должен обеспечить достаточно малую вероятность появления метастабильных напряжений. Смысл слова «достаточно» зависит от контекста. Для сотового телефона, вероятно, один отказ за 10 лет допустим, потому что пользователь может всегда выключить и включить телефон, если он «зависнет». Для медицинского прибора более предпочтительным является один отказ за предполагаемое время существования вселенной (10^{10} лет). Чтобы гарантировать корректность логических уровней, все асинхронные входы должны пройти через *синхронизаторы*.

Синхронизатор, как показано на **Рис. 3.52**, является устройством, на вход которого поступает асинхронный сигнал *D* и тактовый сигнал *CLK*. За ограниченное время он формирует выходной сигнал *Q*, который с очень высокой вероятностью имеет корректный логический уровень. Если вход *D* стабилен в течение апертурного времени, то выход *Q*

должен принять значение входа. Если *D* изменяется в течение апертурного времени, то *Q* может принять значение 0 или 1, но не должен быть метастабильным.

На **Рис. 3.1** показано, как из двух триггеров можно построить простой синхронизатор. Триггер F1 фиксирует значение входного сигнала *D* по переднему фронту тактового сигнала *CLK*. Если *D* изменяется в апертурное время, его выход D2 на некоторое время может стать метастабильным. Если период тактового сигнала достаточно велик, то с высокой вероятностью до конца периода D2 придет к корректному логическому уровню. Триггер F2 затем фиксирует D2, который теперь стабилен, и формирует корректный выходной сигнал.

Мы говорим о сбое синхронизатора, если его выход *Q* станет метастабильным. Это может произойти, если *D*2 не успеет прийти к корректному состоянию до начала времени предустановки триггера F2, то есть когда $t_{res} > T_c - t_{\text{setup}}$. В соответствии с **выражением (3.1)**, вероятность сбоя для одиночного изменения входа в произвольное время равна:

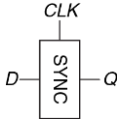


Рис. 3.52 **Символ синхронизатора**

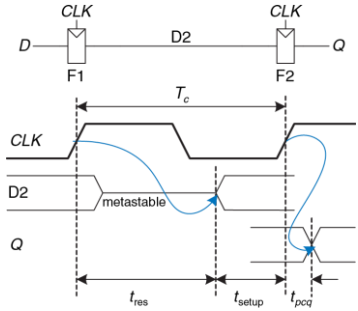


Рис. 3.53 **Простой синхронизатор**

$$P(\text{failture}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}} \qquad (3.25)$$

Вероятность сбоя, *P*(failure), есть вероятность того, что выход *Q* будет метастабильным после однократного изменения входа *D*. Если *D* изменяется один раз за секунду, то вероятность сбоя за одну секунду

будет просто *P*(failure). Однако, если *D* изменяется *N* раз за секунду, то вероятность ошибки за секунду будет в *N* раз большей:

$$P(\text{failure})/\text{sec} = N\frac{T_0}{T_c}e^{-\frac{T_c - t_{setup}}{\tau}} \tag{3.26}$$

Надежность системы обычно измеряют *средним временем наработки на отказ* (*mean time between failures*, *MTBF*). Как понятно из названия, MTBF – это среднее время между отказами системы. Эта величина обратна вероятности сбоя системы за любую заданную секунду:

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{-\frac{T_c - t_{setup}}{\tau}}}{NT_0} \tag{3.27}$$

Выражение **(3.27)** показывает, что MTBF растет экспоненциально с ростом времени ожидания синхронизатора, T_c. Для большинства систем синхронизатор, который ожидает один период тактового сигнала, обеспечивает достаточную величину MTBF. В высокоскоростных системах может понадобиться ожидание на большее количество периодов тактового сигнала.

Пример 3.14 Синхронизатор для входа конечного автомата

Конечный автомат, который управляет работой светофора, (см. **раздел 3.4.1**) принимает асинхронные входные сигналы от датчиков дорожного движения. Предположим, что для обеспечения стабильности входов используются синхронизаторы. В среднем за одну секунду датчик срабатывает 0.2 раза. Триггер в синхронизаторе имеет следующие характеристики: τ= 200 пс, T_0 = 150 пс, и t_{setup} = 500 пс. Каким должен быть период синхронизатора, чтобы среднее время наработки на отказ (MTBF) превышало 1 год?

Решение: 1 год ≈ $\pi \times 10^7$ секунд.

$$\pi \times 10^7 = \frac{T_c e^{-\frac{T_c - 500\times10^{-12}}{200^{-12}}}}{(0.2)(150\times10^{-12})} \tag{3.28}$$

Для нахождения искомого периода нужно решить **уравнение (3.27)** которое не имеет решения в аналитическом виде. Однако его достаточно просто решить методом проб и ошибок. В электронной таблице можно попробовать несколько величин T_c и посчитать MTBF, пока не будет найдена величина T_c, которая даст MTBF близкое к 1 году: T_c = 3.036 нс

3.5.6 Вычисление времени разрешения

Выражение (3.24) можно получить, используя базовые знания курсов теории цепей, дифференциальных уравнений и теории вероятностей. Этот раздел можно пропустить, если вы не интересуетесь выводом этого выражения или если вы слабо знакомы с элементарной математикой.

Выход триггера будет метастабильным спустя некоторое время t, если триггер пытается зафиксировать изменяющийся вход (что приводит к возникновению метастабильного состояния) и выход не успевает прийти к корректному уровню в течение этого времени после фронта тактового сигнала. Символически это можно выразить так:

$$P(tres > t) = P(\text{samples changing input}) \times P(\text{unresolved}) \qquad (3.29)$$

Оба вероятностные сомножителя будут рассмотрены отдельно. Как показано на **Рис. 3.54**, асинхронный входной сигнал переходит из состояния 0 в состояние 1 в течение некоторого времени t_{switch}. Вероятность того, что вход изменится в течение апертурного времени, равна

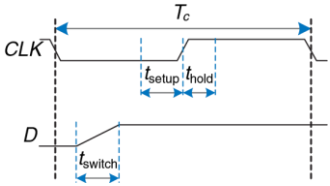


Рис. 3.54 **Временная диаграмма входного сигнала**

Как показано на **Рис. 3.54**, асинхронный входной сигнал переходит из состояния 0 в состояние 1 в течение некоторого времени t_{switch}. Вероятность того, что вход изменится в течение апертурного времени, равна

$$P_{(\text{samples changing input})} = \frac{t_{switch} + t_{setup} + t_{hold}}{T_c} \quad (3.30)$$

Если триггер уже перешел в метастабильное состояние с вероятностью P (samples changing input), то время, необходимое для разрешения метастабильности, зависит от внутренней структуры схемы. Это время определяет вероятность P (unresolved) – вероятность того, что триггер не успевает перейти в корректное состояние (0 или 1) за время t. В этом разделе будет проанализирована простая модель бистабильного прибора и сделана оценка этой вероятности.

Для построения бистабильного прибора используется запоминающее устройство с положительной обратной связью. На **Рис. 3.55 (a)** показана реализация такой обратной связи с использованием двух инверторов; поведение такой схемы является репрезентативным для большинства бистабильных элементов. Пара инверторов ведет себя аналогично буферу. Для построения модели можно считать, что буфер имеет симметричную передаточную характеристику на постоянном токе, которая показана на **Рис. 3.55 (b)**, наклон характеристики равен G.

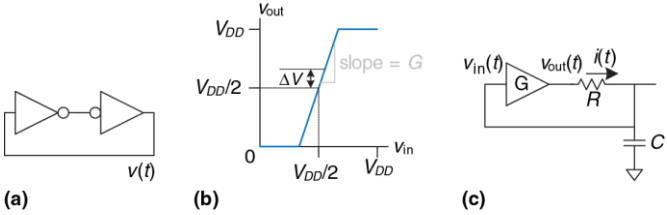


Рис. 3.55 **Схемная модель бистабильного устройства**

Выходной ток буфера ограничен, этот факт можно промоделировать его выходным сопротивлением, R. Все реальные схемы имеют также некоторую емкость C, которую нужно перезаряжать при изменении состояния схемы. Процесс зарядки конденсатора через резистор не позволяет буферу переключиться мгновенно, характерное время этого

процесса равно RC. Таким образом, полная модель схемы показана на **Рис. 3.55 (c)**, где $v_{out}(t)$ – напряжение, определяющее состояние бистабильной схемы.

Состояние схемы, при котором $\upsilon_{out}(t) = \upsilon_{in}(t) = V_{DD}/2$, является метастабильным; если схема стартует точно с этого состояния, то при отсутствии шума она будет находиться в нем неопределенно долго. Поскольку все напряжения являются непрерывными величинами, то вероятность того, что работа схемы начнется точно в точке метастабильности, исчезающе мала. Однако работа схемы может начаться в нулевой момент времени около точки метастабильности, когда $\upsilon_{out}(0) = V_{DD}/2 + \Delta V$, где ΔV–малое отклонение. В таком случае положительная обратная связь в конце-концов приведет $\upsilon_{out}(t)$ к V_{DD}, если $\Delta V > 0$, или к 0, если $\Delta V < 0$. Время, необходимое для достижения V_{DD} или 0, является временем разрешения бистабильного прибора.

Передаточная характеристика буфера по постоянному току нелинейна, но в окрестности точки метастабильности она имеет форму, близкую к линейной. Более точно, если $\upsilon_{in}(t) = V_{DD}/2 + \Delta V/G$, то $\upsilon_{out}(t) = V_{DD}/2 + \Delta V$, для малых ΔV. Ток через резистор равен $i(t) = (\upsilon_{out}(t) - \upsilon_{in}(t))/R$. Конденсатор заряжается со скоростью $dv_{in}(t)/dt = i(t)/C$. Объединяя эти два выражения, можно найти уравнение для выходного напряжения.

$$\frac{d\upsilon_{\text{out}}(t)}{dt} = \frac{(G-1)}{RC}\left[d\upsilon_{\text{out}} - \frac{V_{DD}}{2}\right] \tag{3.31}$$

Это линейное дифференциальное уравнение первого порядка. Решая его с начальным условием $v_{\text{out}}(0) = V_{DD}/2 + \Delta V$, можно найти зависимость выходного напряжения от времени:

$$\upsilon_{\text{out}}(t) = \frac{V_{DD}}{2} + \Delta V e^{-\frac{(G-1)t}{RC}} \tag{3.32}$$

На **Рис. 3.56** приведены графики $\upsilon_{\text{out}}(t)$ для разных начальных точек. Напряжение $\upsilon_{\text{out}}(t)$ экспоненциально удаляется от метастабильной точки $V_{DD}/2$, пока не достигнет предела V_{DD} или 0. Выход в конце-концов приходит в корректное логическое состояние 0 или 1. Время, необходимое для этого, зависит от отклонения начального напряжения (ΔV) от точки метастабильности ($V_{DD}/2$).

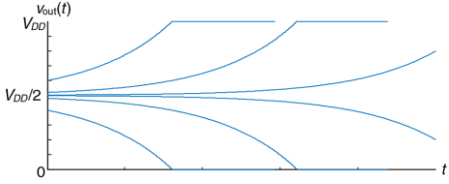


Рис. 3.56 **Временная диаграмма процесса перехода в корректное состояние**

Если в **уравнение (3.32)** подставить $\upsilon_{out}(t_{res}) = V_{DD}$ или 0, то можно найти время разрешения t_{res}:

$$|\Delta V| e^{-\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2} \tag{3.33}$$

$$t_{res} = \frac{RC}{G-1} \ln \frac{V_{DD}}{2|\Delta V|} \tag{3.34}$$

Таким образом, время разрешения экспоненциально возрастает, если бистабильное устройство имеет большое сопротивление или емкость, которые не позволяют выходному напряжению изменяться быстро. Оно уменьшается, если бистабильное устройство имеет большое усиление, G. Время разрешения также логарифмически возрастает при

приближении начальных условий схемы к точке метастабильности ($\Delta V \to 0$)

Обозначим τ через $\frac{RC}{G-1}$. Из **уравнения (3.34)** можно получить значение начального отклонения, которое соответствует некоторому заданному времени разрешения t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-t_{res}/\tau} \qquad (3.35)$$

Предположим, что бистабильное устройство пытается зафиксировать входной сигнал во время его изменения. На его вход поступает напряжение $\upsilon_{in}(0)$, которое предполагается равномерно распределенным в интервале от 0 до V_{DD}. Вероятность того, что выход не достигнет корректного значения через время t_{res}, зависит от вероятности того, что начальное отклонение будет достаточно малым. Более точно, начальное отклонение υ_{out} должно быть меньше, чем $\Delta V_{res}/G$. Тогда вероятность того, что входной сигнал бистабильного устройства имеет достаточно малое отклонение, равна

$$P_{(\text{unresolved})} = P\left(\left|\upsilon_{\text{in}}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{GV_{DD}} \tag{3.36}$$

Таким образом, вероятность того, что время разрешения превосходит некоторую заданную величину *t*, задается следующим выражением:

$$P(t_{res} > t) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{GT_c} e^{-t/\tau} \tag{3.37}$$

Обратите внимание на то, что выражения **(3.37)** и **(3.24)** имеют одинаковый вид, если $T_0 = (t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}})/G$ и $\tau = RC/(G - 1)$. Итак, мы вывели выражение **(3.24)** и показали, как величины T_0 и τ зависят от физических свойств бистабильного устройства.

3.6 ПАРАЛЛЕЛИЗМ

Скорость обработки информации системой характеризуется задержкой и пропускной способностью информации передачи информации через нее. Мы определим *токен* (*token*) как группу входов, которая обрабатывается для того, чтобы получить группу выходов. Это название связано с методом визуализации передачи данных внутри системы путем размещения в схеме токенов или маркеров и их передвижением по схеме вместе с обрабатываемыми данными. *Задержка,* или *латентность* (*latency*) системы – время, которое необходимо для прохождения одного токена через всю систему с ее входа на выход. *Пропускная способность* (*throughput*) – количество токенов, которое может быть обработано системой в единицу времени.

Пример 3.15 ПРОПУСКНАЯ СПОСОБНОСТЬ И ЗАДЕРЖКА ПРИ ПРИГОТОВЛЕНИИ ПЕЧЕНЬЯ

Бену нужно быстро подготовиться к вечеринке с молоком и печеньем, посвященной введению в эксплуатацию его светофора. За 5 минут он сворачивает печенья и укладывает их на противень. В течение 15 минут печенья выпекаются в печи. После окончания выпекания он начинает готовить следующий противень. Какая пропускная способность и задержка выпекания Беном противня печенья?

Решение: В этом примере противень является токеном. Задержка равна 1/3 часа на противень. Пропускная способность – 3 противня в час.

Достаточно легко понять, что пропускная способность может быть увеличена путем обработки нескольких токенов в одно и то же время. Это называется *параллелизмом* и используется в двух формах: пространственной и временной. В *пространственном параллелизме* используется несколько копий аппаратных блоков, так что в одно и то же время можно выполнять несколько задач. *Временной параллелизм* предполагает разбиение задачи на несколько стадий (или ступеней), как это происходит на сборочном конвейере. Несколько задач могут быть распределены по ступеням. Хотя все задачи должны пройти по всем ступеням, разные задачи в любой заданный момент времени будут находиться на своей ступени, так что несколько задач могут одновременно обрабатываться на разных ступенях. Временной

параллелизм часто называется *конвейеризацией*. Пространственный параллелизм часто называют просто параллелизмом, но мы будем избегать этого названия из-за его неоднозначности.

Пример 3.16 Параллелизм при приготовлении печенья

К Бену Битдидлу на вечеринку придут сотни друзей, и ему нужно печь печенье быстрее. Он собирается использовать пространственный и/или временной параллелизм.

Пространственный параллелизм: Бен просит Алису Хакер помочь ему. У нее есть собственная печь и противень

Временной параллелизм: Бену дали второй противень. Как только он ставит один противень в печь, он начинает сворачивать печенье на другом противне, а не ожидает окончания выпекания печенья на первом противне.

Какая будет задержка и пропускная способность при использовании пространственного параллелизма? Временного? При использовании обоих видов параллелизма?

Решение: Задержка – это время, необходимое для завершения одной задачи от начала до конца. Во всех случаях задержка равна 1/3 часа. Если в начале у Бена не было печенья, то задержка – это время, необходимое для производства первого противня.

Пропускная способность – это количество противней с печеньем, которое производится за один час. При использовании пространственного параллелизма

и Бен, и Алиса делают по одному противню каждые 20 минут. Следовательно, пропускная способность удваивается и составляет 6 противней в/час. При использовании временного параллелизма Бен ставит новый противень в печь каждые 15 минут, пропускная способность равна 4 противня/час. Это показано на **Рис. 3.57**.

Если Бен и Алиса используют оба метода, они могут выпечь 8 противней в/час.

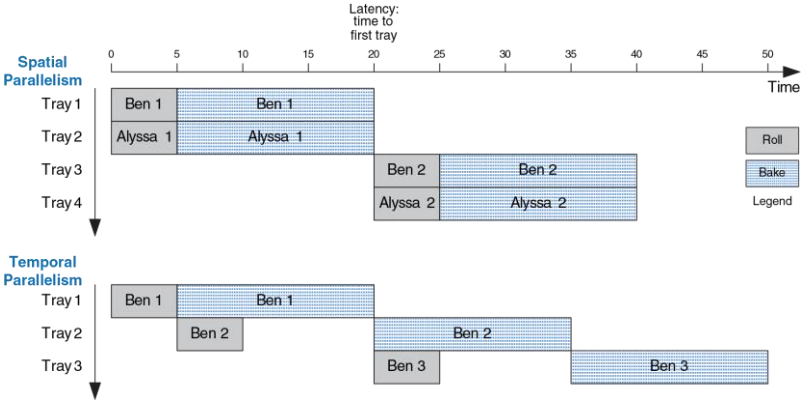


Рис. 3.57 **Пространственный и временной параллелизм при приготовлении печенья**

Рассмотрим систему с задержкой L. Если в системе отсутствует параллелизм, то пропускная способность будет $1/L$. В системе с пространственным параллелизмом, которая содержит N копий аппаратных блоков, пропускная способность будет N/L. В системе с временным параллелизмом задача в идеальном случае разбивается на N стадий или ступеней одинаковой длины. В этом случае пропускная способность будет также равна N/L, причем необходим только один экземпляр аппаратного блока. Однако, как показывает пример приготовления печенья, часто создание N ступеней одной и той же продолжительности обработки невозможно. Если самая длинная ступень имеет задержку L_1, то пропускная способность конвейеризированной системы будет равна $1/L_1$.

Конвейеризация (временной параллелизм) особенно привлекательна, поскольку она увеличивает скорость схемы без увеличения аппаратных затрат. Вместо этого, регистры, установленные между блоками комбинационной логики, разделяют ее на короткие ступени, которые могут работать на более высокой тактовой частоте. Регистры не позволяют токенам, находящимся в одной ступени, догонять и разрушать токены, которые находятся в следующей стадии обработки.

На **Рис. 3.58** приведен пример схемы, в которой отсутствует конвейеризация. Она состоит из четырех блоков логики, которые расположены между двумя регистрами. Критический путь проходит

через блоки 2, 3 и 4. Предположим, что регистр имеет задержку распространения на тактовый вход-выход 0.3 нс и время удержания 0.2 нс. Тогда время минимальный период тактового сигнала равен $Tc = 0.3 + 3 + 2 + 4 + 0.2 = 9.5$ нс. Схема имеет задержку 9.5 нс и пропускную способность 1/9.5 ns = 105 МГц.

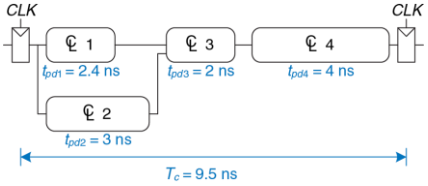


Рис. 3.58 **Схема без конвейеризации**

На **Рис. 3.59** показана эта же самая схема, разделенная с помощью дополнительных регистров между блоками 3 и 4, на 2-ступенчатый конвейер. Первая ступень имеет минимальный период тактового сигнала 0.3 + 3 + 2 + 0.2 = 5.5 нс. Минимальный период для второй ступени равен 0.3 + 4 + 0.2 = 4.5 нс. Тактовый сигнал должен быть достаточно медленным для того, чтобы работали все ступени. Следовательно, T_c = 5.5 ns. Задержка равна двум периодам тактового сигнала или 11 нс. Пропускная способность равна 1/5.5 ns = 182 MHz. Этот пример показывает, что в реальных схемах конвейеризация с

двумя ступенями почти удваивает пропускную способность и немного увеличивает задержку. Для сравнения, идеальная конвейеризация точно удвоила бы пропускную способность и не ухудшила задержку. Несоответствие возникает потому, что реальную схему невозможно разделить на две абсолютно равные части и также потому, что конвейерные регистры вносят дополнительные потери на упорядочение.

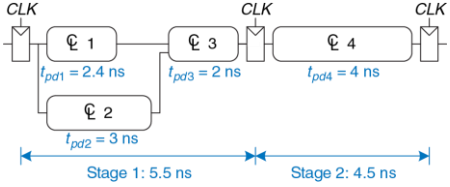


Рис. 3.59 Схема с двухступенчатым конвейером

На **Рис. 3.60** показан еще один вариант той же схемы, в котором используется трехступенчатый конвейер. Обратите внимание, что в схеме необходимо на два регистра больше, они сохраняют результаты блоков 1 и 2 в конце первой ступени конвейера. Время цикла ограничивается теперь третьей ступенью и равно 4.5 нс. Задержка равна трем циклам или 13.5 нс. Пропускная способность равна 1/4.5 ns = 222 МГц. Как и в прошлом варианте схемы, добавление еще

одной ступени конвейера улучшает пропускную способность за счет небольшого увеличения задержки.

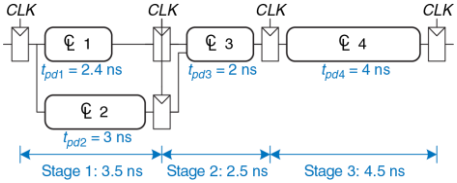


Рис. 3.60 **Схема с трехступенчатым конвейером**

Хотя эти рассмотренные подходы весьма эффективны, они не могут быть использованы во всех ситуациях. Использование параллелизма ограничивается *взаимозависимостями* (*dependencies*) реальных задач. Если текущая задача зависит от результатов предыдущей задачи, а не только от своих предыдущих шагов, то выполнение задачи не может быть начато до завершения предыдущей задачи. Например, если Бен Битдидл хочет проверить, достаточно ли вкусны печенья из первого противня, перед приготовлением второго, он имеет взаимозависимость, которая препятствует использованию конвейера или параллелизму. Параллелизм – один из самых важных методов проектирования высокопроизводительных цифровых систем. Конвейеризация будет

далее обсуждаться в разделе 7, там же будут показаны примеры обработки взаимозависимостей.

3.7 РЕЗЮМЕ

Эта глава была посвящена рассмотрению методов анализа и проектирования последовательностной логики. В отличие от комбинационной логики, выходные сигналы которой зависят только от текущего состояния входных сигналов, выходные сигналы последовательностной логики зависят как от текущего, так и от предыдущего состояния входных сигналов. Другими словами, последовательностная логика помнит информацию о входных сигналах в предыдущие моменты времени. Эта память называется состоянием логики.

> Любой, кто сможет изобрести логику, выходы которой зависят от будущих входов, станет фантастически богатым!

Последовательностные схемы могут быть сложны для анализа, и их легко неправильно спроектировать, поэтому мы ограничимся использованием небольшого числа тщательно спроектированных аппаратных блоков. Наиболее важным элементом для наших целей является триггер, который принимает тактовый сигнал и входной сигнал D и формирует выходной сигнал Q. По переднему фронту

тактового импульса триггер копирует вход *D* на выход *Q*, в противном случае он сохраняет старое состояние *Q*. Группа триггеров с общим тактовым сигналом называется регистром. На триггеры могут также поступать управляющие сигналы сброса или разрешения.

Хотя существует множество форм последовательностных схем, мы ограничимся использованием синхронных последовательностных схем, поскольку их просто разрабатывать. Синхронные последовательностные схемы состоят из блоков комбинационной логики, разделенных тактируемыми регистрами. Состояние схемы сохраняется в регистрах и обновляется только по фронтам тактового сигнала.

Один из эффективных подходов к проектированию последовательностных схем основывается на использовании конечных автоматов. Для проектирования конечного автомата сначала следует определить его входы и выходы, затем сделать эскиз диаграммы переходов с указанием состояний и условий переходов между ними. Затем для всех состояний автомата нужно выбрать кодировку и на основе диаграммы создать таблицу переходов между состояниями и таблицу выходов, которые показывают следующее состояние и выходной сигнал при заданном текущем состоянии и входном сигнале. По этим таблицам проектируется комбинационная логическая схема,

которая определяет следующее состояние и выходной сигнал, и создается эскиз схемы.

Синхронные последовательностные схемы имеют временную спецификацию, которая включает в себя задержки распространения и реакции тракта тактовый вход-выход, t_{pcq} и t_{ccq}, а также времена предустановки и удержания, t_{setup} и t_{hold}. Для корректной работы таких схем их входы должны быть стабильными в течение апертурного времени, которое состоит из времени предустановки перед передним фронтом тактового импульса и времени удержания после него. Минимальный период T_c тактового сигнала системы равен сумме задержек распространения комбинационной логики, t_{pd}, и задержек t_{pcq} + t_{setup} в регистрах. Для корректной работы схемы задержка реакции регистров и комбинационной логики должна быть больше, чем t_{hold}. Несмотря на распространенное заблуждение, время удержания не влияет на величину минимального периода тактового сигнала .

Общая производительность системы измеряется задержкой и пропускной способностью. Задержка – это время, необходимое для прохождения одного токена с входа системы на ее выход. Пропускная способность – количество токенов, которое система может обработать в единицу времени. Параллелизм увеличивает пропускную способность системы.

УПРАЖНЕНИЯ

Упражнение 3.1 Временные диаграммы входных сигналов RS-триггера-защёлки показаны на **Рис. 3.61**. Нарисуйте временную диаграмму значений выхода *Q*.

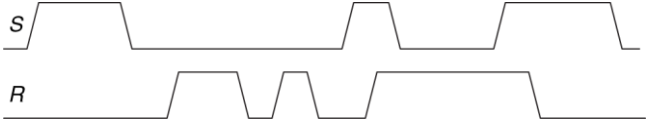


Рис. 3.61 Временные диаграммы входов RS-триггера-защёлки для упражнения 3.1

Упражнение 3.2 Временные диаграммы входных сигналов RS-триггера-защёлки показаны на **Рис. 3.62**. Нарисуйте временную диаграмму значений выхода *Q*.

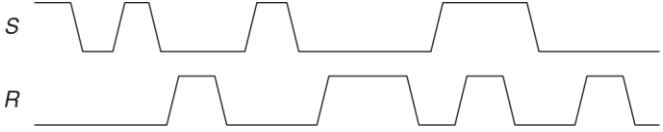


Рис. 3.62 Временные диаграммы входов RS-триггера-защёлки для упражнения 3.2

Упражнение 3.3 Временные диаграммы входных сигналов D-триггера-защёлки показаны на **Рис. 3.63**. Нарисуйте временную диаграмму значений выхода *Q*.

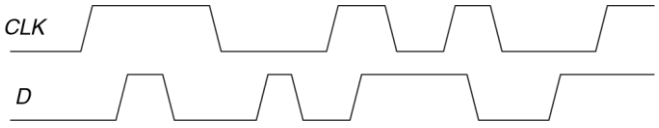


Рис. 3.63 **Временные диаграммы входов D-триггера для упражнений 3.3 и 3.5**

Упражнение 3.4 Временные диаграммы входных сигналов D-триггера-защёлки показаны на **Рис. 3.64**. Нарисуйте временную диаграмму значений выхода *Q*.

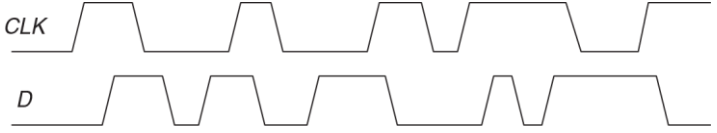


Рис. 3.64 **Временные диаграммы входов D-триггера для упражнений 3.4 и 3.6**

Упражнение 3.5 На **Рис. 3.63** показаны временные диаграммы входов D-триггера (синхронизируемого фронтом). Нарисуйте временную диаграмму значений выхода *Q*.

Упражнение 3.6 На **Рис. 3.64** показаны временные диаграммы входов D-триггера (синхронизируемого фронтом). Нарисуйте временную диаграмму значений выхода *Q*.

Упражнение 3.7 Является ли схема, изображенная на **Рис. 3.65**, комбинационной или последовательностной? Объясните взаимосвязь входов с выходами. Как называется такая схема?.

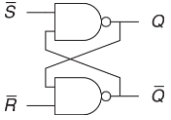


Рис. 3.65 **Исследуемая схема**

Упражнение 3.8 Является ли схема, изображенная на **Рис. 3.66**, комбинационной или последовательностной? Объясните взаимосвязь входов с выходами. Как называется такая схема?

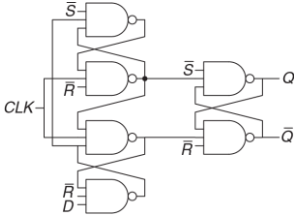


Рис. 3.66 **Исследуемая схема**

Упражнение 3.9 *T-триггер* (от англ. toggle – переключать) имеет один вход, *CLK*, и один выход, *Q*. По каждому фронту тактового сигнала значение на выходе триггера изменяется на противоположное. Нарисуйте схему T-триггера, используя D-триггер и инвертор.

Упражнение 3.10 На вход JK-триггера поступают тактовый сигнал *CLK* и входные данные *J* и *K*. Триггер синхронизируется по фронту тактового сигнала. В случае, если *J* и *K* равны нулю, то на выходе *Q* сохраняется старое значение. Если *J*=1, *K*=0, то *Q* устанавливается в 1. Если *J*=0, *K*=1, то *Q* сбрасывается в 0. Если *J*=1, *K*=1, то *Q* принимает противоположное значение.

a) Постройте JK-триггер, используя D-триггер и комбинационную логику.

b) Постройте D-триггер, используя JK-триггер и комбинационную логику.

c) Постройте T-триггер (см. **упражнение 3.9**), используя JK-триггер.

Упражнение 3.11 Схема, изображенная на **Рис. 3.67**, называется *C-элементом Мюллера*. Объясните взаимосвязь входов с выходами.

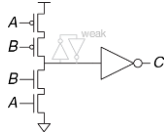


Рис. 3.67 **C-элемент Мюллера**

Упражнение 3.12 Спроектируйте D-защёлку с асинхронным сбросом, используя логические элементы.

Упражнение 3.13 Спроектируйте D-триггер с асинхронным сбросом, используя логические элементы.

Упражнение 3.14 Спроектируйте синхронно устанавливаемый D-триггер, используя логические элементы.

Упражнение 3.15 Спроектируйте асинхронно устанавливаемый D-триггер, используя логические элементы.

Упражнение 3.16 Кольцевой генератор состоит из N инверторов, замкнутых в кольцо. У каждого инвертора есть минимальная tcd и максимальная t_{pd} задержки. Определите диапазон частот, в котором может работать кольцевой генератор при условии, что N нечётно.

Упражнение 3.17 Почему число N из **упражнения 3.16** должно быть нечётным?

Упражнение 3.18 Какие из схем на **Рис. 3.68** являются синхронными последовательностными? Ответ поясните.

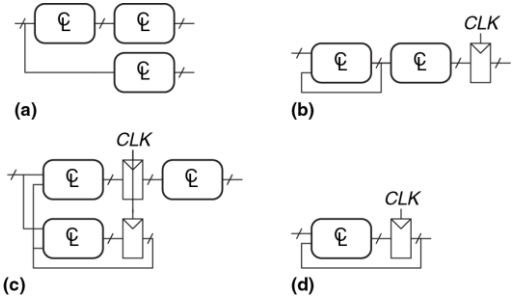


Рис. 3.68 **Схемы**

Упражнение 3.19 Вы проектируете контроллер лифта для 25-этажного здания. У контроллера есть два входа: ВВЕРХ и ВНИЗ. Выходными данными является номер этажа, на котором находится лифт. 13-й этаж отсутствует. Чему равно минимальное количество битов для хранения состояния в контроллере?

Упражнение 3.20 Вы проектируете конечный автомат для отслеживания настроение четырёх студентов, работающих в лаборатории по проектированию цифровых схем. У студентов может быть следующие настроения: СЧАСТЛИВЫЙ (если схема работает), ГРУСТНЫЙ (если схема сгорела), ЗАНЯТЫЙ (работает над схемой), ЗАГРУЖЕННЫЙ (думает над схемой), СПЯЩИЙ (спит на рабочем

месте). Сколько состояний будет у вашего автомата? Какое минимальное количество битов состояний необходимо для кодирования состояния автомата?

Упражнение 3.21 Как бы Вы разделили конечный автомат из **упражнения 3.20** на несколько менее сложных автоматов? Сколько состояний было бы у каждого такого простого автомата? Какое минимальное количество бит необходимо для такого модульного проекта?

Упражнение 3.22 Опишите словами, что делает автомат на **Рис. 3.69**. Заполните таблицу переходов и таблицу выходов, используя двоичное кодирование. Составьте булевы выражения для следующего состояния и для выхода и нарисуйте схему этого конечного автомата.

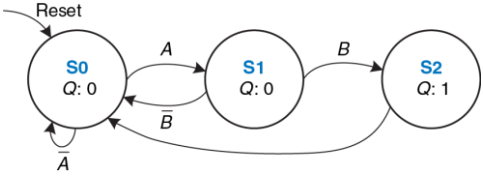


Рис. 3.69 **Диаграмма переходов**

Упражнение 3.23 Опишите словами, что делает автомат на **Рис. 3.70**. Заполните таблицу переходов и таблицу выходов, используя двоичное

кодирование. Составьте булевы выражения для следующего состояния и для выхода и нарисуйте схему этого конечного автомата.

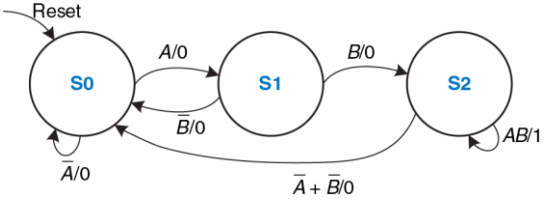


Рис. 3.70 **Диаграмма переходов**

Упражнение 3.24 На пересечениии Академической и Беговой улиц все еще случаются происшествия. Футболисты устремляются на перекрёсток, как только на их светофоре загорается зелёный свет, и сталкиваются с зазевавшимися ботаниками. Последние вступают на перекресток всё еще на зеленый свет. Усовершенствуйте светофор из **раздела 3.4.1** так, чтобы на обеих улицах горел красный свет в течении 5 секунд до того как какой-либо из светофоров станет зелёным. Нарисуйте диаграмму переходов автомата Мура, кодирование состояний, таблицу переходов, таблицу выходов, выражения для выходов и для следующего состояния и схему конечного автомата.

Упражнение 3.25 У улитки Алисы из **раздела 3.4.3** есть дочка с автоматом Мили. Улитка-дочка улыбается, когда она проходит последовательность 1101 или 1110.

Нарисуйте диаграмму переходов для этой весёлой улитки, используя как можно меньше состояний. Выберите кодирование состояний и составьте общую таблицу переходов и выходов. Составьте выражения для выхода и для следующего состояния и нарисуйте схему автомата.

Упражнение 3.26 Вас уговорили спроектировать автомат с прохладительными напитками для офиса. Расходы на напитки частично покрывает профсоюз, поэтому они стоят всего по 5 рублей. Автомат принимает монеты в 1, 2 и 5 рублей. Как только покупатель внесет необходимую сумму, автомат выдаст напиток и сдаст сдачу. Спроектируйте конечный автомат для автомата с прохладительными напитками. Входами автомата являются 1, 2 и 5 рублей, а именно, какая из этих монет вставлена. Предположим, что по каждому тактовому сигналу вставляется только одна монета. Автомат имеет выходы: налить газировку, вернуть 1 рубль, вернуть 2 рубля, вернуть 2 по 2 рубля. Как только в автомате набирается 5 рублей (или больше), он выставляет сигнал «НАЛИТЬ ГАЗИРОВКУ», а также сигналы, возврата соответствующей сдачи. Затем автомат должен быть готов опять принимать монеты.

Упражнение 3.27 У кода Грея есть полезное свойство: коды соседних чисел отличаются друг от друга только в одном разряде. В **Табл. 3.23** представлен 3-разрядный код Грея, представляющий числовую последовательность от 0 до 7. Спроектируйте 3-разрядный автомат счетчика в коде Грея по модулю 8. У автомата нет входов, но есть 3 выхода. (Счётчик по модулю N считает от 0 до N-1, затем цикл повторяется. Например, в часах используется счётчик по модулю 60 для того чтобы считать минуты и секунды от 0 до 59.) После сброса

на счетчике должно быть 000. По каждому фронту тактового сигнала счетчик должен переходить к следующему коду Грея. По достижении кода 100 счётчик должен опять перейти к коду 000.

Табл. 3.23 3-разрядный код Грея

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Упражнение 3.28 Усовершенствуйте свой автомат счетчика в коде Грея из **упражнения 3.27** так, чтобы от мог считать как вверх, так и вниз. У счётчика появится вход ВВЕРХ. Если ВВЕРХ=1, то счётчик будет переходить к следующему коду, а если ВВЕРХ=0 – то к предыдущему.

Упражнение 3.29 Ваша компания, Детекторама, хочет спроектировать конечный автомат с двумя входами *A* и *B* и одним выходом *Z*. Выход в *n*–ом цикле, Z_n, и

является или результатом логического И или логического ИЛИ текущего A_n и предыдущего A_{n-1} значений на входе, в зависимости от сигнала B_n.

$$Z_n=A_nA_{n-1} \qquad \text{если } B_n=0$$

$$Z_n=A_n+A_{n-1} \qquad \text{если } B_n=1$$

a) Нарисуйте временную диаграмму для Z по данным диаграммам A и B, изображенным на **Рис. 3.71**

b) Этот автомат является автоматом Мура или автоматом Мили?

c) Спроектируйте конечный автомат. Составьте диаграмму переходов, закодированную таблицу переходов, выражения для выходов и следующего состояния и нарисуйте схему.

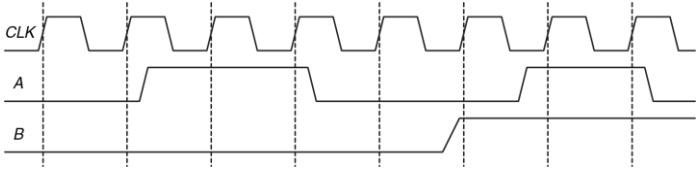


Рис. 3.71 **Входные временные диаграммы конечного автомата**

Упражнение 3.30 Спроектируйте конечный автомат с одним входом A и двумя выходами X и Y. На выходе X должна появиться 1, если 1 поступали на вход как

минимум 3 цикла (необязательно подряд), а на Y должна появиться 1, если X=1 как минимум 2 цикла подряд. Составьте диаграмму переходов, закодированную таблицу переходов, выражения для выходов и следующего состояния и нарисуйте схему.

Упражнение 3.31 Проанализируйте конечный автомат, показанный на **Рис. 3.72**. Составьте таблицу переходов и выходов, а также диаграмму состояний. Опишите словами, что делает этот автомат.

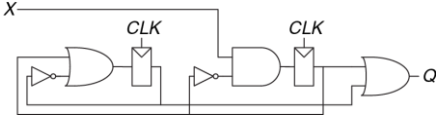


Рис. 3.72 **Схема конечного автомата**

Упражнение 3.32 Повторите **упражнение 3.31** со схемой, показанной на **Рис. 3.73**. Напомним, что входы регистров *s* и *r* отвечают за установку (set) и сброс (reset) соответственно.

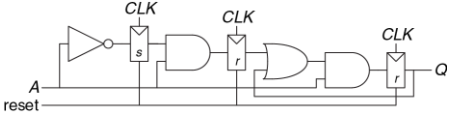


Рис. 3.73 **Схема конечного автомата**

Упражнение 3.33 Бен Битдидл спроектировал схему вычисления функции XOR с четырьмя входами и регистрами на входе и выходе (см. **Рис. 3.74**). Каждый двухвходовый элемент XOR имеет задержку распространения 100 пс. и задержку реакции 55 пс. Время предустановки триггеров равно 60 пс., время удержания – 20 пс., максимальная задержка тактовый сигнал-выход равна 70 пс., минимальная – 50 пс.

a) Чему будет равна максимальная рабочая частота схемы при отсутствии расфазировки тактовых импульсов?

b) Какая расфазировка тактовых импульсов допустима, если схема должна работать на частоте 2 ГГц?

c) Какая расфазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?

d) Алиса Хакер утверждает, что она может перепроектировать комбинационную логическую схему с целью повышения ее скорости и устойчивости к расфазировке тактовых импульсов. В ее улучшенной схеме также используется три двухвходовых элемента XOR, но они по-другому соединены между собой. Какую схему она спроектировала? Какая нее будет максимальная частота без расфазировки тактовых импульсов? Какая расфазировка тактовых импульсов допустима до возникновения нарушений ограничений времени удержания?

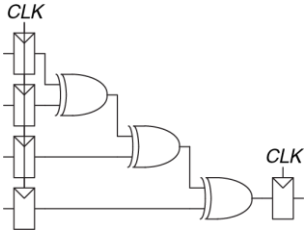


Рис. 3.74 **Схема вычисления функции XOR с регистрами на входе и выходе**

Упражнение 3.34 В рамках разработки сверхбыстродействующего двухразрядного процессора RePentium вам поручено проектирование сумматора. Как показано на **Рис. 3.75**, сумматор состоит из двух полных сумматоров, выход переноса первого сумматора подсоединен ко входу переноса второго. На входе и выходе сумматора находятся регистры, сумматор должен выполнить суммирование за один период тактового сигнала. Задержки распространения полных сумматоров равны: по тракту вход C_{in} – выходы C_{out} и *Sum* (*S*) – 20 пс, по тракту входы *A* и *B* – выход C_{out} – 25 пс., по тракту входы *A* и *B* – выход *S* – 30 пс. Полные сумматоры имеют задержки реакции: по тракту вход C_{in} – любой выход – 15 пс., по тракту входы *A* и *B* – любой выход – 22 пс. Время предустановки триггеров равно 30 пс., время удержания – 10 пс., задержки тракта тактовый сигнал-выход: распространения 10 пс., реакции 21 пс.

a) Чему будет равна максимальная рабочая частота схемы при отсутствии расфазировки тактовых импульсов?

b) Какая расфазировка тактовых импульсов допустима, если схема должна работать на частоте 8 ГГц?

c) Какая расфазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?

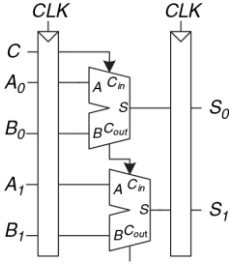


Рис. 3.75 Схема двухразрядного сумматора

Упражнение 3.35 В логических матрицах, программируемых пользователем, (*field programmable gate array* (*FPGA*)) для создания комбинационных логических схем используются конфигурируемые логические блоки (*configurable logic blocks* (*CLBs*)), а не логические элементы.

В матрицах Xilinx Spartan 3 задержки распространения и реакции каждого CLB равно 0.61 и 0.30 нс., соответсвенно. Они также содержат триггеры, задержки распространения и реакции которых равны 0.72 и 0.50 нс., а времена предустановки и удержания – 0.53 и 0 нс., соответственно.

a) Если вы проектируете систему, которая должна работать на частоте 40 МГц, сколько последовательно соединенных CLB можно разместить между двумя триггерами? При ответе можно считать, что расфазировка тактовых импульсов и задержка в соединениях между CLB отсутствует.

b) Предположим, что все пути между триггерами проходят через, по крайней мере, один CLB. Какая рафазировка тактовых импульсов допустима до возникновения в схеме нарушений ограничений времени удержания?

Упражнение 3.36 Для построения синхронизатора используются два триггера с t_{setup} = 50 пс., T_0 = 20 пс., and τ = 30 ps. Асинхронный вход изменяется 10^8 раз за секунду. Чему равен минимальный период синхронизатора, при котором среднее время между отказами (MTBF) достигнет 100 лет?

Упражнение 3.37 Вам необходимо построить синхронизатор, который принимает асинхронные входные сигналы, среднее время между отказами (MTBF) должно быть не менее 50 лет. Тактовая частота системы равна 1 ГГц, триггеры имеют следующие параметры τ = 100 пс., T_0 = 110 пс., and t_{setup} = 70 пс. На вход синхронизатора каждые 2 секунды поступает новый асинхронный сигнал. Чему равна вероятность отказа, которая соответствует заданному среднему время

между отказами (MTBF)? Сколько периодов тактового сигнала следует выждать перед считыванием зафиксированного входного сигнала для достижения этой вероятности.

Упражнение 3.38 Вы столкнулись со своим напарником по лабораторным работам в коридоре, когда он шел навстречу вам. Оба вы отступили в одну сторону и все еще находитесь на пути друг друга. Затем вы оба отступили в другую сторону и продолжаете мешать друг другу пройти. Далее вы оба решили чуть подождать, в надежде, что встречный отступит в сторону, и вы разойдетесь. Вы можете промоделировать эту ситуацию как метастабильную и применить к ней ту же теорию, которая была развита для синхронизаторов и триггеров. Предположим, вы создаете математическую модель своего поведения и поведения своего напарника. Состояние, в котором вы мешаете проходу друг друга, можно трактовать как метастабильное. Вероятность того, что вы остаетесь в этом состоянии после t секунд равна $e^{-\frac{t}{\tau}}$, величина τ описывает скорость вашей реакции, сегодня из-за недосыпания ваш разум затуманен и τ=20 с.

a) Через какое время с вероятностью 99% метастабильность будет разрешена (то есть, вы сможете обойти друг друга)?

b) Вы не только не выспались, но и сильно проголодались. Ситуация крайне серьезная, вы умрете от голода, если не попадете в кафетерий через 3 минуты. Какая вероятность того, что ваш напарник по лабораторным работам должен будет доставить вас в морг?

Упражнение 3.39 Вы построили синхронизатор с использованием триггеров с T_0 = 20 пс. и τ = 30 пс. Ваш начальник поручил вам увеличить среднее время между отказами (MTBF) в 10 раз. Насколько вам нужно увеличить период тактового сигнала?

Упражнение 3.40 Бен Битдидл изобрел новый улучшенный синхронизатор, который по его заявлениям подавляет метастабильность за единственный период. Схема «улучшенного синхронизатора» показана на **Рис. 3.76**. Бен поясняет, что схема в блоке *M* представляет собой аналоговый «детектор метастабильности», который выдаёт сигнал высокого логического уровня, если напряжение на его входе попадает в запретную зону между V_{IL} и V_{IH}. Детектор метастабильности проверяет, не появился ли на выходе *D*2 первого триггера метастабильный сигнал. Если он действительно появился, то «детектор метастабильности» асинхронно сбрасывает триггер и на его выходе появляется корректный логический сигнал 0. Второй триггер фиксирует сигнал *D*2 и на его выходе *Q* всегда будет корректный логический уровень. Алиса Хакер говорит Бену, что схема не будет работать как заявлено, поскольку устранение метастабильности так же невозможно, как и построение вечного двигателя. Кто их них прав? Покажите где ошибка Бена или почему Алиса ошибается.

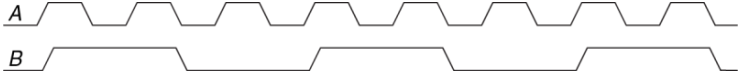


Рис. 3.76 **“Новый улучшенный” синхронизатор**

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

В этом разделе представлены типовые вопросы, которые могут быть заданы соискателям при поиске работы в области проектирования цифровых систем.

Вопрос 3.1 Нарисуйте диаграмму конечного автомата, который детектирует поступление на вход последовательности 01010.

Вопрос 3.2 Спроектируйте конечный автомат, который принимает последовательность битов (один бит за раз), и выполняет над ними операцию дополнения до 2.Он имеет два входа, *Start* и *A*, и один выход *Q*. Двоичное число произвольной длины подается на вход *A*, начиная младшего разряда. Соответствующий выходной бит появляется на том же цикле на выходе *Q*. Вход *Start* устанавливается на один цикл для инициализации конечного автомата перед поступлением младшего бита.

Вопрос 3.3 Чем отличается триггер-защелка и триггер, тактируемый фронтом импульса? Когда следует использовать каждый из них?

Вопрос 3.4 Спроектируйте конечный автомат, который выполняет функции пятиразрядного счетчика.

Вопрос 3.5 Спроектируйте схему детектирования фронта сигнала. Ее выход должен принимать значение 1 в течение одного периода после перехода входного сигнала из состояния 0 в 1.

Вопрос 3.6 Опишите концепцию конвейеризации и методы ее использования.

Вопрос 3.7 Поясните ситуацию, когда временя удержания триггера отрицательно.

Вопрос 3.8 Спроектируйте схему, которая принимает сигнал *A* (см. **Рис. 3.77**) и формирует на выходе сигнал *B*.

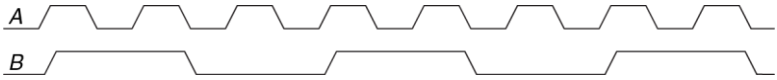


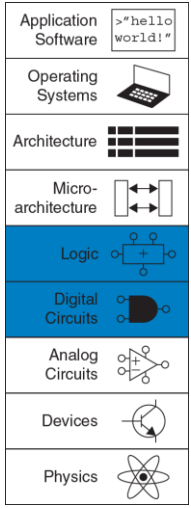
Рис. 3.77 **Формы сигналов**

Вопрос 3.9 Рассмотрим блок комбинационной логики между двумя регистрами. Поясните временные ограничения, которым такой блок должен удовлетворять. Если поставить буфер на тактовом входе второго триггера станут ли ограничения времени предустановки мягче или жестче?


HDL Dictionary
Verilog
VHDL
a & b
a ^ b
~(a+b)
a and b
a xor b
a nor b
Hardware

Языки описания аппаратуры

4



4.1 ВВЕДЕНИЕ

До сих пор мы рассматривали разработку комбинационных и последовательностных цифровых схем на уровне схемотехники. Процесс поиска наилучшего набора логических элементов для выполнения данной логической функции трудоемок и чреват ошибками, так как требует упрощения логических таблиц или выражений и перевода конечных автоматов в вентили вручную. В 1990-е годы разработчики обнаружили, что их производительность труда резко возрастала, если они работали на более высоком уровне абстракции, определяя только логическую функцию и предоставляя создание оптимизированных логических элементов *системе автоматического проектирования* (САПР). Два основных *языка описания аппаратуры* (Hardware Description Language, HDL) – SystemVerilog и VHDL.

SystemVerilog и VHDL построены на похожих принципах, но их синтаксис весьма различается. Их обсуждение в этой главе разделено на две колонки для сравнения, где SystemVerilog будет слева, а VHDL – справа. При первом чтении сосредоточьтесь на одном из языков. Как только вы разберетесь с одним, при необходимости вы сможете быстро усвоить другой. В последующих главах показана аппаратура и в схематическом виде и в форме HDL-модели. Если вы предпочтете пропустить эту главу и не изучать языки описания цифровой аппаратуры, вы тем не менее сможете постичь принципы архитектуры

микропроцессоров на уровне схем. Однако, подавляющее большинство коммерческих систем сейчас строится с использованием языков описания цифровой аппаратуры, а не на уровне схемотехники. Если вы когда-либо в вашей карьере собираетесь заниматься разработкой цифровых схем, мы настоятельно рекомендуем вам выучить один из языков описания аппаратуры.

4.1.1 Модули

Блок цифровой аппаратуры, имеющий входы и выходы, называется *модулем*. Логический элемент “И”, мультиплексор и схема приоритетов являются примерами модулей цифровой аппаратуры. Есть два общепринятых типа описания функциональности модуля – поведенческий и структурный. Поведенческая модель описывает, что модуль делает. Структурная модель описывает то, как построен модуль из простых элементов, с применением принципа иерархии. Код на SystemVerilog и VHDL из **примера 4.1** показывает поведенческое описание модуля, который рассчитывает булеву функцию из **примера 2.6**. На обоих языках модуль назван sillyfunction и имеет 3 входа, a, b и c и один выход y, и, как и следовало ожидать, следует принципу модульности. Он имеет полностью определенный интерфейс, состоящий из его входов и выходов, и выполняет определенную функцию. Конкретный способ, которым модуль был описан, неважен

для тех, кто будет использовать модуль в будущем, поскольку модуль выполняет свою функцию.

Пример 4.1 КОМБИНАЦИОННАЯ ЛОГИКА

SystemVerilog

```
module sillyfunction(input logic a, b, c, output logic y);
  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b & c;
endmodule
```

Модуль на SystemVerilog начинается с имени модуля и списка входов и выходов. Оператор `assign` описывает комбинационную логику. Тильда (`~`) означает НЕ, амперсанд (`&`) – И, а вертикальная черта (`|`) – ИЛИ.

Сигналы типа `logic`, как входы и выходы в примере – логические переменные, принимающие значения 0 или 1. Они также могут принимать плавающее и неопределенное значения – это обсуждается в **разделе 4.2.8**.

Тип `logic` появился в SystemVerilog. Он введен для замены типа `reg`, бывшего постоянным источником затруднений в Verilog. Тип `logic` стоит использовать везде, кроме описания сигналов с несколькими источниками. Такие сигналы называются цепями (net) и будут объяснены в **разделе 4.7**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
 end;
 architecture synth of sillyfunction is
 begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);
 end;
```

Код на VHDL состоит из трех частей: объявления используемых библиотек и внешних объектов (`library`, `use`), объявления интерфейса объекта (`entity`) и его внутренней структуры (`architecture`).

Конструкция для объявления используемых внешних объектов будет рассматриваться в **разделе 4.7.2**. В объявлении интерфейса указывается имя модуля и перечисляются его входы и выходы. Блок `architecture` определяет, что модуль делает.

У сигналов в VHDL, в том числе входов и выходов, должен быть указан тип. Цифровые сигналы стоит объявлять как `STD_LOGIC`. Сигналы этого типа принимают значения '`0`' или '`1`', а также плавающее и неопределенное значения, которые будут описаны в **разделе 4.2.9**. Тип `STD_LOGIC` определен в библиотеке `IEEE.STD_LOGIC_1164`, поэтому библиотеку объявлять обязательно. VHDL не определяет соотношение приоритетов операций AND и OR, поэтому при записи логических выражений нужно всегда использовать скобки.

4.1.2 Происхождение языков SystemVerilog и VHDL

Примерно в половине вузов, где преподают цифровую схемотехнику, читают VHDL, а в оставшейся половине – Verilog. В промышленности склоняются к SystemVerilog, но много компаний еще используют VHDL, поэтому многим разработчикам нужно владеть обоими языками. По сравнению с SystemVerilog, VHDL более многословный и громоздкий, чем можно было бы ожидать от языка, разработанного комитетом («Верблюд – это лошадь, разработанная комитетом» – американская шутка. – прим. перев.)

SystemVerilog

Верилог был разработан компанией Gateway Design Automation в 1984 году как фирменный язык для симуляции логических схем. В 1989 году Gateway приобрела компания Cadence, и Verilog стал открытым стандартом в 1990 году под управлением сообщества Open Verilog International. Язык стал стандартом IEEE *(прим. переводчика: институт инженеров по электротехнике и электронике (IEEE) – профессиональное сообщество, ответственное за многие компьютерные стандарты, например, Wi-Fi (802.11), Ethernet (802.3), и чисел с плавающей точкой (754))*.в 1995 году. В 2005 году язык был расширен для упорядочивания и лучшей поддержки моделирования и верификации систем. Эти расширения были объединены в единый стандарт, который сейчас называется SystemVerilog(стандарт IEEE 1800-2009). Файлы языка SystemVerilog обычно имеют расширение .sv.

Термин “баг” существовал еще до изобретения компьютера. В 1878 году Томас Эдисон называл багами “огрехи и затруднения” в своих изобретениях. Первый настоящий компьютерный баг был молью, попавшей между контактами реле электромеханического компьютера Harvard Mark II в 1947 году. Ее нашла Грейс Хоппер, которая зарегистрировала этот случай в рабочем журнале, приклеив моль и прокомментировав: “впервые обнаружен настоящий баг”.

(Источник: запись в регистрационном журнале Исторического центра Военно-морского флота, Флот США, фото № NII 96566-KN)

VHDL

Аббревиатура VHDL расшифровывается как VHSIC Hardware Description Language. VHSIC, в свою очередь, происходит от сокращения Very High Speed Integrated Circuits – названия программы министерства обороны США. Разработка VHDL был начата в 1981 году министерством обороны для описания структуры и функциональности электронных схем. За основу для разработки был

взят язык программирования ADA. Изначальной целью языка была документация, но затем он был быстро адаптирован для симуляции и синтеза. IEEE стандартизировал его в 1987 году, и после этого язык обновлялся несколько раз. Эта глава основана на редакции VHDL 2008 года (стандарт IEEE 1076-2008), которая упорядочивает язык во многих аспектах. На момент написания не все функции стандарта VHDL 2008 года поддерживаются в САПР; эта глава только используют те функции, которые поддерживаются в Synplicity, Altera Quartus и Modelsim. Файл языка VHDL имеет расширение .vhd .

На обоих языках можно полностью описать любую электронную систему, но у каждого языка есть свои особенности. Лучше использовать язык, который уже распространен в вашей организации или тот, которого требуют ваши клиенты. Большинство САПР сейчас позволяют смешивать языки, поэтому разные модули могут быть написаны на разных языках.

4.1.3 Симуляция и Синтез

Две основные цели HDL – логическая симуляция и синтез. Во время симуляции на входы модуля подаются некоторые воздействия и проверяются выходы, чтобы убедиться, что модуль функционирует корректно. Во время синтеза текстовое описание модуля преобразуется в логические элементы.

Симуляция

Люди регулярно совершают ошибки. Ошибки в цифровой аппаратуре называют багами. Ясно, что устранение багов в цифровой системе очень важно, особенно когда от правильной работы аппаратуры зависят чьи-то жизни. Тестирование системы в лаборатории весьма трудоёмко. Исследовать причины ошибок в лаборатории может быть очень сложно, так как наблюдать можно только сигналы, подключенные к контактам чипа, а то, что происходит внутри чипа, напрямую наблюдать невозможно. Исправление ошибок уже после того, как система была выпущена, может быть очень дорого. Например, исправление одной ошибки в новейших интегральных микросхемах стоит больше миллиона долларов и занимает несколько месяцев. Печально известный баг в команде деления с плавающей точкой (FDIV) в процессоре Pentium вынудил корпорацию Intel отозвать чипы после того, как они были поставлены заказчикам, что стоило им 475 миллионов долларов. Логическая симуляция необходима для тестирования системы до того, как она будет выпущена.

Рис. 4.1 показывает графики сигналов из симуляции предыдущего модуля sillyfunction, демонстрирующие, что модуль работает корректно. *(Прим. переводчика: симуляция была проведена в программе ModelSim PE Student Edition версии 10.0c. Modelsim был выбран, так как он используется коммерчески и имеет студенческую версию с*

возможностью бесплатной симуляции до 10 тыс. строк кода). Y есть лог.1, когда a,b,c есть 000, 100, или 101, как и указано в логическом выражении.

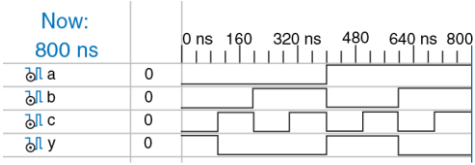


Рис. 4.1 **Графики сигналов**

Синтез

Логический синтез преобразует код на HDL в нетлист, описывающий цифровую аппаратуру (т.е. логические элементы и соединяющие их проводники). Логический синтезатор может выполнять оптимизацию для сокращения количества необходимых элементов. Нетлист может быть текстовым файлом или нарисован в виде схемы, чтобы было легче визуализировать систему. **Рис. 4.2** показывает результаты синтеза модуля sillyfunction *(прим. переводчика: синтез был сделан с помощью программы Synplify Premier от Synplicity. Этот САПР был выбран, так как он является лидирующим коммерческим продуктом для синтеза HDL в программируемые логические интегральные*

схемы (смотри **раздел 5.6.2***) и так как он доступен по цене и дешев для использования в университетах).* Обратите внимание, что , как три трехвходовых элемента И упрощены в 2 двухвходовых элемента И, как мы обнаружили в **примере 2.6**, используя булеву алгебру.

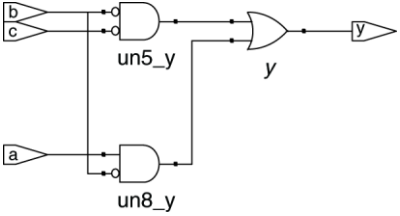


Рис. 4.2 **Схема sillyfunction**

Описание схем на HDL напоминает программный код. Однако вы должны помнить, что ваш код предназначен для описания аппаратуры. SystemVerilog и VHDL – сложные языки со множеством операторов. Не все из них могут быть синтезированы в аппаратуре: например, оператор вывода результатов на экран во время симуляции не превращается в цифровую схему. Так как наша основная задача – создание цифровой схемы, мы акцентируем свое внимание на синтезируемом подмножестве языков. Точнее, мы будем делить код на HDL на синтезируемые модули и среду тестирования. Синтезируемые модули

описывают цифровую схему. Среда тестирования содержит код, который подает воздействия на входы модуля и проверяет правильность значений его выходов, а также выводит несоответствия между ожидаемыми и действительными значениями. Код среды тестирования предназначается только для симуляции и не может быть синтезирован.

Одна из главных ошибок начинающих заключается в том, что они думают о коде на HDL как о компьютерной программе, а не как о подспорье для описания цифровой аппаратуры. Если вы не представляете, хотя бы примерно, во что должен синтезироваться ваш код на HDL, то, скорее всего, результат вам не понравится. Ваша цифровая схема может получиться гораздо больше, чем нужно, или может оказаться, что ваш код симулируется правильно, но не может быть реализован в аппаратуре. Вместо этого, вы должны думать над вашей разработкой в понятиях комбинационной логики, регистров и конечных автоматов. Нарисуйте эти блоки на бумаге и покажите, как они будут подключены до того, как вы начнете писать код.

По нашему опыту, лучший способ выучить HDL – на примерах. В HDL есть определенные способы описания разных типов логики; эти способы называются идиомами. В этой главе мы научим вас, как писать идиомы для блоков каждого типа логики и затем как сложить блоки вместе, чтобы получить работающую систему. Когда вам понадобится

описать аппаратуру определенного типа, посмотрите на похожий пример и адаптируйте его под свои цели. Мы не будем пытаться строго описывать весь синтаксис HDL, так как это скучно и потому что это ведет к представлению о HDL как о языках программирования, а не как о подспорье для разработки аппаратуры. Если вам понадобится дополнительная информация об особенностях языков, то обратитесь к спецификациям VHDL и SystemVerilog, изданным IEEE, или многочисленным сухим, но исчерпывающим учебникам. (См. рекомендованный список литературы в конце книги).

4.2 КОМБИНАЦИОННАЯ ЛОГИКА

Помните, что мы тренируемся проектировать синхронные последовательностные схемы, которые состоят из комбинационной логики и регистров. Состояние выходов комбинационной схемы зависит только от входных сигналов. В этом разделе описано, как создавать поведенческие модели комбинационной логики с использованием HDL.

4.2.1 Битовые операторы

Битовые операторы манипулируют однобитовыми сигналами или многоразрядными шинами. Так модуль inv в **примере 4.2** описывает 4 инвертора, подключенные к четырехразрядным шинам.

Пример 4.2 ИНВЕРТОРЫ

SystemVerilog

```
module inv(input logic [3:0] a,
           output logic [3:0] y);
  assign y = ~a;
endmodule
```

`a[3:0]` представляет собой 4-битную шину. Биты, от старшего к младшему, записываются так: `a[3]`, `a[2]`, `a[1]` и `a[0]`. Такой порядок битов называется little-endian, т.к. младший бит имеет наименьший битовый номер. Мы могли бы назвать шину `a[4:1]`, и тогда `a[4]` был бы старшим. Или мы могли бы написать

a[0:3], и тогда порядок битов от старшего к младшему был бы следующим: `a[0]`, `a[1]`, `a[2]` и `a[3]`. Такой порядок битов называется big-endian.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of inv is
begin
  y <= not a;
end;
```

В VHDL для определения шин типа STD_LOGIC используется `STD_LOGIC_VECTOR`. `STD_LOGIC_VECTOR(3 downto 0)` представляет собой 4-битную шину. Биты от старшего к младшему: `a(3)`, `a(2)`, `a(1)` и `a(0)`. Такой порядок битов называется little-endian , т.к. младший бит имеет наименьший битовый номер. Мы могли бы объявить шину как `STD_LOGIC_VECTOR (4 downto 1)`, и тогда 4-ый бит был бы старшим. Или мы могли бы записать `STD_LOGIC_VECTOR(0 to 3)`, тогда порядок битов от старшего к младшему был бы следующим: `a(0)`, `a(1)`, `a(2)` и `a(3)`. Такой порядок битов называется big-endian.

Порядок следования разрядов шины является чисто условным. (См. происхождение термина в панели ссылок **раздела 6.2.2**) Действительно, и в этом примере порядок битов неважен, т.к. для

набора инверторов не имеет значения, где какой бит находится. Порядок битов имеет значение только для некоторых операторов, например, оператора сложения, в которых сумма из одного столбца переносится в другой. Любой порядок является приемлемым, если он используется последовательно. Мы будем постоянно использовать порядок битов слева направо от старшего к младшему, `[N - 1:0]` на языке SystemVerilog и `(N - 1 downto 0)` на языке VHDL для *N*-разрядной шины.

После каждого примера кода в этой главе приводится схема, созданная из кода SystemVerilog инструментом синтеза Synplify Premier. **Рис. 4.3** показывает, что модуль inv синтезируется в виде блока из 4 инверторов, обозначенных символом инвертора с надписью `y[3:0]`. Блок инверторов соединен с четырехбитными входной и выходной шинами. Подобная же аппаратная реализация получается из синтезированного VHDL- кода.

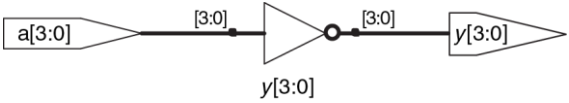


Рис. 4.3 Синтезированная схема модуля inv

Модуль `gates` в **примере 4.3** описывает битовые операции, которые выполняются на четырехбитных шинах для других основных логических функций.

Пример 4.3 ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ

SystemVerilog

```
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  /*five different two-input logic
    gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a |b;      // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a |b);  // NOR
endmodule
```

Символы `~`, `^` и `|` – это примеры операторов в языке SystemVerilog, тогда как `a`, `b` и `y1` являются операндами. Комбинация операторов и операндов, таких как `a & b` или `~(a | b)` называется выражением. Полная команда, такая как `assign y4 = ~(a&b);`называется оператором. `Assign out = in1 op in2;` называется оператором непрерывного присваивания. Он заканчивается точкой с запятой. Когда в операторе непрерывного присваивания входные значения справа от знака «=» меняются, результат слева от знака «=» вычисляется заново. Таким образом, непрерывное присваивание описывает комбинационную логику.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity gates is
port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:  out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of gates is
begin
  -- five different two-input logic gates
  -- acting on 4-bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

НЕ, исключающее ИЛИ и ИЛИ – это примеры операторов в языке VHDL, тогда как `a`, `b` и `y1` являются операндами. Комбинация операторов и операндов, таких как `a and b` или `a nor b`,называется выражением. Полная команда, например, `y4 = a nand b`; называется оператором. `out= in1 op in2`; называется оператором одновременного присваивания сигнала. Операторы присваивания в VHDL заканчиваются точкой с запятой. Когда в операторе одновременного присваивания сигнала входные значения справа от знака «=» изменяются, результат слева от знака «=» вычисляется заново. Таким образом, оператор одновременного присваивания сигнала описывает комбинационную логику.

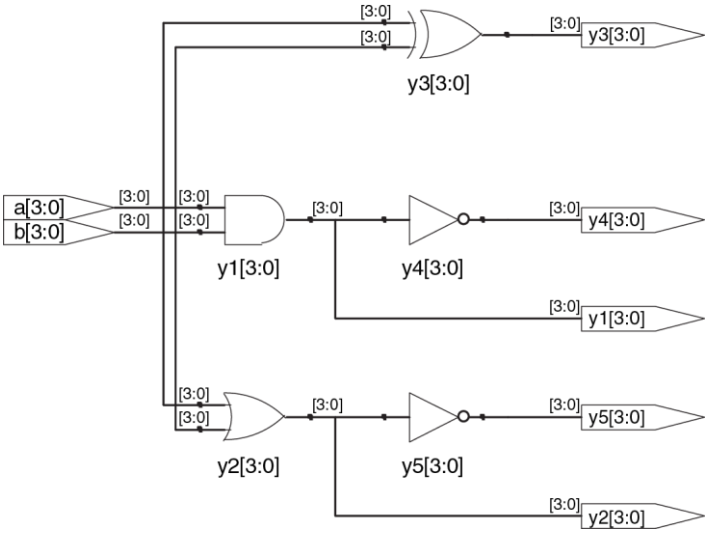


Рис. 4.4 **Синтезированная схема модуля gates**

4.2.2 Комментарии и пробелы

Пример с модулем gates демонстрирует, как оформлять комментарии. Языки SystemVerilog и VHDL не имеют особых требований к использованию свободного пространства (например, пробелы, табуляция и разрывы строк). Тем не менее, надлежащие отступы и использование пустых строк помогают сделать читаемыми необычные разработки. Будьте последовательны в использовании прописных букв и подчеркиваний в именах сигналов и модулей. В этом тексте используются только строчные буквы. Имена сигналов и модулей не должны начинаться с цифр.

SystemVerilog

Комментарии в языке SystemVerilog схожи с комментариями языков C или Java. Комментарии, начинающиеся с «`/*`», могут занимать несколько строк, до следующего знака «`*/`». Комментарии, начинающиеся с «`//`», продолжаются до конца строки.

SystemVerilog чувствителен к регистру символов (прописным и строчным буквам). `y1` и `Y1` в SystemVerilog – это разные сигналы, однако, использование множества сигналов, отличающихся только регистром символов, вносит путаницу.

VHDL

Комментарии, начинающиеся с «/*», могут занимать несколько строк до следующего знака «*/». Комментарии, начинающиеся с «--», продолжаются до конца строки.
VHDL не чувствителен к регистру символов. В VHDL `y1` и `Y1` – это один и тот же сигнал. Однако, другие программы, открывающие ваш файл, могут оказаться чувствительны к регистру символов, что приводит к неприятным ошибкам, если вы смешиваете прописные и строчные буквы.

4.2.3 Операторы сокращения

Операторы сокращения соответствуют многовходовым элементам, работающим на одной шине. **Пример 4.4** описывает восьмивходовый вентиль И с входами a_7, a_6, ..., a_0. Аналогичные операторы сокращения существуют для вентилей ИЛИ, исключающее ИЛИ, И-НЕ, ИЛИ-НЕ и исключающее ИЛИ с инверсией. Запомните, что многовходовый вентиль исключающее ИЛИ осуществляет функцию контроля четности, возвращая значение ИСТИНА, если нечетное количество входов имеют состояние ИСТИНА.

Пример 4.4 ВОСЬМИВХОДОВЫЙ ВЕНТИЛЬ И

SystemVerilog

```
module and8(input logic [7:0] a,
            output logic      y);
  assign y = &a;
  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
Endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity and8 is
  port(a: in STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;
architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --   a(3) and a(2) and a(1) and a(0);
end;
```

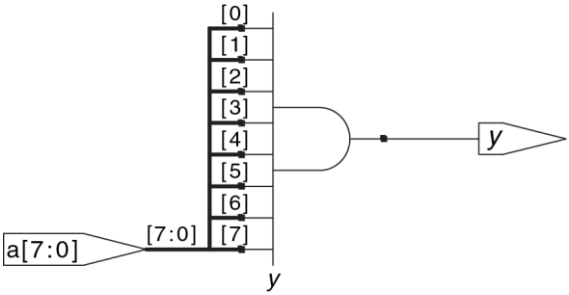


Рис. 4.5 **Синтезированная схема модуля and8**

4.2.4 Условное присваивание

Операторы условного присваивания выбирают определенный выход среди других, исходя из состояния входа, называемого УСЛОВИЕ. В **примере 4.5** показан двухвходовой мультиплексор, использующий условное присваивание.

Пример 4.5 ДВУХВХОДОВОЙ МУЛЬТИПЛЕКСОР

SystemVerilog

Условный оператор `?:` выбирает между вторым и третьим выражениями, руководствуясь первым выражением. Первое выражение называется условие (condition). Если условие принимает значение 1, то оператор выбирает второе выражение. Если условие принимает значение 0, то оператор выбирает третье выражение. `?:` особенно полезен для описания мультиплексоров, т.к. на основании состояния первого входа, он выбирает между двумя другими. Следующий код демонстрирует программную реализацию двухвходового мультиплексора с 4-битными входами и выходами с использованием условного оператора.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic s,
            output logic [3:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Если `s` равно 1, то `y` = `d1`, иначе `y` = `d0`. Оператор `?:` также называют тернарным оператором, так как он имеет три входа. С такой же целью он используется в языках C и Java.

VHDL

Условное присваивание сигнала осуществляет разные операции, зависящие от некоторых условий. Они особенно полезны для описания мультиплексоров.

Например, двухвходовой мультиплексор может использовать условное присваивание сигнала для выбора одного из двух 4-битных входов.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
  port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
       s:      in STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

Условное присваивание сигнала устанавливает `y` в `d1`, если s имеет значение 1. В противном случае он устанавливает y в `d0`. Обратите внимание, что в версиях VHDL до 2008, нужно было писать `when s = '1'`, а не `when s`.

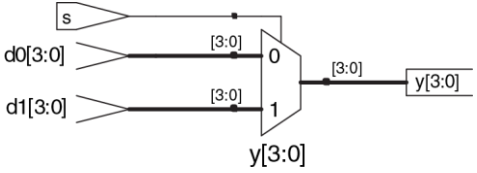


Рис. 4.6 **Синтезированная схема модуля `mux2`**

В **примере 4.6** продемонстрирован четырехвходовой мультиплексор, работающий по тому же принципу, что и мультиплексор из **примера 4.5**.

На **Рис. 4.7** изображена схема, созданная с помощью SynplifyPremier. Программное обеспечение использует обозначение мультиплексора, отличающееся от того, которое до сих пор приводилось в тексте. Мультиплексор имеет многоразрядные входы данных `(d)` и одиночные входы разрешения `(e)`. Когда один из входов активирован, соответствующие данные отправляются на выход. Например, когда `s[1] = s[0] = 0`, нижний вентиль И- `un1_s_5`, формирует 1, активируя нижний вход мультиплексора, в результате выбирается `d0[3:0]`.

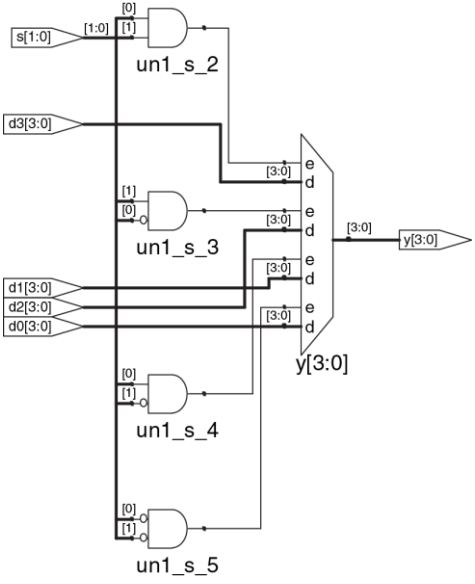


Рис. 4.7 **Синтезированная схема модуля mux4**

Пример 4.6 ЧЕТЫРЕХВХОДОВОЙ МУЛЬТИПЛЕКСОР

SystemVerilog

Четырехвходовой мультиплексор может выбрать один из четырех входов с помощью вложенных условных операторов.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);
  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
Endmodule
```

Если `s[1]` принимает значение 1, тогда мультиплексор выбирает первое выражение, `(s[0] ? d3 : d2)`. Это выражение в свою очередь выбирает или `d3`, или `d2` на основе `s[0] ( y = d3`, если `s[0]` имеет значение 1 и d2, если `s[0]` имеет значение 0). Если `s[1]` имеет значение 0, тогда мультиплексор подобным образом выбирает второе выражение, которое дает или `d1`, или `d0` в зависимости от `s[0]`.

VHDL

Четырехвходовой мультиплексор может выбрать один из четырех входов с помощью нескольких условий else в операторе условного присваивания сигнала.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL также поддерживает операторы выборочного присваивания сигнала для обеспечения более краткой записи, когда выбирается одна из нескольких возможностей. Это аналогично использованию операции `switch/case` вместо нескольких операций `if/else` в некоторых языках программирования. Четырехвходовой мультиплексор может быть переписан с использованием выборочного присваивания сигнала следующим образом:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

4.2.5 Внутренние переменные

Часто бывает удобно разбить сложную функцию на несколько промежуточных. Например, полный сумматор, который будет описан в **разделе 5.2.1**, представляет собой схему с тремя входами и двумя выходами, определяемыми следующими уравнениями:

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in} \quad (4.1)$$

Если мы введем промежуточные сигналы P и G,

$$P = A \oplus B$$
$$G + AB \quad (4.2)$$

мы сможем переписать уравнения для полного сумматора в виде:

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in} \quad (4.3)$$

Вы можете проверить это, заполнив таблицу истинности, чтобы убедиться, что это правильно.

Переменные P и G называются *внутренними*, потому что они не являются ни входами, ни выходами, они используются только внутри модуля. Они подобны локальным переменным в языках программирования. **Пример 4.7** показывает, как эти переменные используются в HDL.

Операции присваивания в HDL (`assign` в языке SystemVerilog и `=` в VHDL) происходят параллельно. Это отличается от традиционных языков программирования, таких как C или Java, в которых операторы оцениваются в том порядке, в котором они записаны. В традиционных языках важно, что выражение $S = P \oplus C_{in}$ следует за выражением $P=A \oplus B$, поскольку операторы выполняются последовательно. В HDL порядок записи не имеет значения. Подобно аппаратным средствам, операторы присваивания HDL выполняются в момент, когда входы и сигналы с правой стороны выражения меняют свое значение

независимо от порядка, в котором операторы присваивания появляются в модуле.

Пример 4.7 ПОЛНЫЙ СУММАТОР

SystemVerilog

В языке SystemVerilog внутренние сигналы обычно объявляются как logic.

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g |(p & cin);
endmodule
```

VHDL

В VHDL сигналы обычно используют для представления внутренних переменных, значения которых определяются одновременными операторами присваивания, таких как `p = a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
```

```
       s, cout:   out STD_LOGIC);
end;
architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;
  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

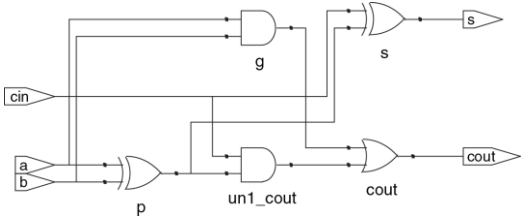


Рис. 4.8 Синтезированная схема модуля fulladder

4.2.6 Приоритет

Обратите внимание, что мы использовали скобки в вычислении cout в **примере 4.7**, чтобы определить порядок операций: $C_{out} = G + (P \bullet C_{in})$, а не $Cout = (G + P) \bullet Ci_n$. Если мы не используем скобки, порядок операций определяется по умолчанию. **Пример 4.8** определяет приоритет операторов от высшего к низшему для каждого языка. Таблицы включают арифметические операции, операции сдвига и операции сравнения, которые будут рассмотрены в **главе 5**.

Пример 4.8 ПРИОРИТЕТ ОПЕРАТОРОВ

SystemVerilog

Табл. 4.1

	Операция	Значение
Высший	`~`	Побитовое отрицание (НЕ)
	`*, /, %`	Умножение, деление, остаток
	`+, -`	Сложение, вычитание
	`<<, >>`	Сдвиг влево/вправо
	`<<<, >>>`	Арифметический сдвиг влево/вправо
	`<, <=, >, >=`	Сравнение на больше-меньше

	Операция	Значение
Низший	`==, !=`	Сравнение на равенство
	`&, ~&`	И, И-НЕ
	`^, ~^`	Исключающее ИЛИ, исключающее ИЛИ-НЕ
	`|, ~|`	ИЛИ, ИЛИ-НЕ
	`?:`	Условный оператор

Система приоритета операторов для SystemVerilog подобна системам, принятым в других языках программирования. В частности, И имеет приоритет над ИЛИ. Можно пользоваться приоритетом операторов, чтобы исключить использование круглых скобок.

```
assign cout = g |p & cin;
```

VHDL

Табл. 4.2

	Операция	Значение
Высший	`not`	НЕ
	`*, /, mod, rem`	Умножение, деление, модуль, остаток
	`+, -`	Сложение, вычитание
	`rol, ror,` `srl, sll`	Циклический сдвиг влево/вправо Логический сдвиг влево/вправо
Низший	`<, <=, >, >=`	Сравнение на больше-меньше
	`=, /=`	Сравнение на равенство
	`and, or, nand, nor, xor, xnor`	Логические операции

В VHDL умножение имеет приоритет над сложением. Однако, в отличие от SystemVerilog, здесь все логические операторы (`and`, `or` и т.д.) имеют одинаковый приоритет. Поэтому скобки необходимы; в противном случае `cout = g or p and cin` будет интерпретироваться слева направо как `cout = (g or p) and cin`.

4.2.7 Числа

Числа указываются в двоичной, восьмеричной, десятичной или шестнадцатеричной системе счисления (основания 2, 8, 10 и 16 соответственно). Размер, т.е. количество бит, может быть также указан, свободные разряды заполняются нулями. Подчеркивания в числах игнорируются и могут быть полезными, лишь когда требуется разбить длинное число на более читаемые фрагменты. **Пример 4.9** объясняет, как числа записываются в каждом из языков.

Пример 4.9 ЧИСЛА

SystemVerilog

Формат для объявления констант – `N'Bvalue`, где `N` – размер в битах, `B` – буква, указывающая на основание и `value` – значение. Например, `9'h25` определяет 9-битное число со значением $25_{16} = 37_{10} = 000100101_2$. SystemVerilog поддерживает 'b для основания 2, `'o` – для основания 8, `'d` – для основания 10 и `'h` – для основания 16. Если основание опущено, то по умолчанию оно равно 10. Если не указан размер, то предполагается, что число имеет столько же бит, сколько и выражение, в котором оно используется. Недостающие старшие разряды дополняются нулями автоматически до полного размера. Например, если `w` – 6-битная шина, то assign `w = 'b11` присваивает w значение 000011. Лучшей практикой является явное указание размера. Исключением является то, что `'0` и `'1` служат конструкциями SystemVerilog для заполнения шины нулями или единицами соответственно.

Табл. 4.3

Запись	Число бит	Основание	Значение	Представление
`3'b101`	3	2	5	101
`'b11`	?	2	3	0000011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	000101010

VHDL

В VHDL числа STD_LOGIC записываются в бинарном коде и заключаются в одинарные кавычки: `'0'` and `'1'` указывают на логические уровни 0 и 1. Формат объявления констант типа STD_LOGIC_VECTOR следующий: `NB"value"`, где `N` – размер в битах, `B` – буква, указывающая на основание и `value` – значение. Например, `9X"25"` определяет 9-битное число со значением $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 поддерживает `B` для основания 2, `O` – для основания 8, `D` – для основания 10 и `X` – для основания 16.

Если основание опущено, то по умолчанию оно равно 10. Если размер не указан, то предполагается, что число имеет размер, соответствующий числу битов

значения. По состоянию на октябрь 2011 SynplifyPremier от Synopsys не поддерживает указание размера.
`others =‘0’` и `others =‘1’` – конструкции VHDL с заполнением всех битов нулями или единицами соответственно.

Табл. 4.4

Запись	Число бит	Основание	Значение	Представление
`3B"101"`	3	2	5	101
`B"11"`	2	2	3	11
`8B"11"`	8	2	3	00000011
`8B"1010_1011"`	8	2	171	10101011
`3D"6"`	3	10	6	110
`6O"42"`	6	8	34	100010
`8X"AB"`	8	16	171	10101011
`"101"`	3	2	5	101
`B"101"`	3	2	5	101
`X"AB"`	8	16	171	10101011

4.2.8 Z-состояние и X-состояние

В HDL z-состояние используется для указания на плавающее значение. Использование z-состояния, в частности, полезно для описания буфера с тремя состояниями, состояние выхода которого является плавающим, когда на вход разрешения подан 0. Вспомните из **Раздела 2.6.2**, что шина может управляться несколькими буферами с тремя состояниями, только один из которых должен быть активен. **Пример 4.10** демонстрирует программную реализацию тристабильного буфера. Если этот буфер активирован, то состояние на выходе будет таким же, как и на входе. Если буфер не активирован, то состояние на выходе назначается плавающим значением (`z`).

Пример 4.10 ТРИСТАБИЛЬНЫЙ БУФЕР

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
  assign y = en ? a : 4'bz;
endmodule
```

Обратите внимание, что y объявляется как `tri`, а не `logic`. Сигналы типа `logic` могут иметь только один драйвер. Тристабильные шины могут иметь несколько драйверов, поэтому они должны объявляться, как net. Два применяемых типа net

в SystemVerilog имеют названия `tri` и `trireg`. Обычно только один драйвер в сети активен в конкретный момент времени, и сеть принимает задаваемые им значения. Если ни один из драйверов не активирован, то `tri` плавает (`z`), в то время как `trireg` сохраняет предыдущее значение. Если для входа или выхода тип не указан, то предполагается, что тип – `tri`. Также обратите внимание, что выход модуля типа `tri` может использоваться как вход типа `logic` для других модулей. В дальнейшем цепи с несколькими драйверами будут рассматриваться в **разделе 4.7**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of tristate is
begin
  y <= a when en else “ZZZZ”;
end;
```

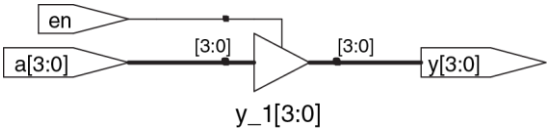


Рис. 4.9 Синтезированная схема модуля `tristate`

Также в HDL используют `x` для указания недействительного логического уровня. Если на шину одновременно попадает 0 и 1 с двух активных тристабильных буферов (или других элементов), то в результате получаем x, что указывает на конфликт. Если все тристабильные буферы, управляющие шиной, одновременно находятся в состоянии OFF, то на шине будет плавающее состояние, на что указывает z. В начале моделирования состояния узлов, таких как выходы триггеров, инициализируются неизвестным состоянием (x в SystemVerilog и u − в VHDL) . Это помогает отслеживать ошибки, которые появляются, если вы забыли сбросить триггер перед тем, как использовать его выход.

Если логический элемент получает плавающее значение на входе, то он может сформировать x на выходе, когда у него не получается определить правильное выходное значение. Если элемент получает на входе недействительное или неинициализированное значение, то на выходе он может сформировать x. **Пример 4.11** показывает, как в

SystemVerilog и VHDL комбинируют эти различные значения сигналов в логических элементах. X- или u-состояния при моделировании практически всегда означают ошибки или плохой стиль программирования. В синтезированной цепи это соответствует плавающему входу элемента, неинициализированному состоянию или конфликту. X или u могут быть случайно интерпретированы схемой как 0 или 1, что приведет к непредсказуемому поведению программы.

Пример 4.11 ТАБЛИЦЫ ИСТИННОСТИ С НЕОПРЕДЕЛЕННЫМИ И ПЛАВАЮЩИМИ ВХОДАМИ

SystemVerilog

Сигналы в SystemVerilog могут принимать значения `0`, `1`, `z` и `x`. Константы SystemVerilog, начинающиеся с `z` или `x`, при необходимости дополняются символами `z` или `x` в старших разрядах (вместо нулей) для достижения необходимой длины.

Табл. 4.5 демонстрирует таблицу истинности для вентиля И, используя все четыре возможные значения сигнала. Обратите внимание, что вентиль может иногда определять выход, несмотря на неизвестное состояние некоторых входов. Например, `0&z` возвращает `0`, потому что на выходе вентиля И всегда `0`, если какой-то из входов имеет состояние `0`. В противном случае, плавающее или некорректное состояние на входах приводит к недействительным состояниям на выходах, отображающимся в SystemVerilog как `x`.

Табл. 4.5

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

VHDL

Сигналы типа STD_LOGIC в VHDL могут принимать значения `'0'`, `'1'`, `'z'`, `'x'` и `'u'`.

Табл. 4.6 демонстрирует таблицу истинности для вентиля И, используя пять возможных значений сигнала. Обратите внимание, что вентиль может иногда определять выход, несмотря на неизвестные состояния некоторых входов. Например, `'0'` and `'z'` возвращает `'0'`, т.к. на выходе вентиля И всегда `'0'`, если какой-то из входов имеет состояние `'0'`. В противном случае, плавающее или некорректное состояние на входах приводит к недействительным состояниям на выходах, отображающимся в VHDL как `'x'`. Неинициализированные состояния входов приводят к неинициализированным состояниям сигналов на выходах, отображающимся в VHDL как `'u'`.

Табл. 4.6

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

Пример 4.12 МАНИПУЛЯЦИИ С БИТАМИ

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Оператор `{}` используется для объединения шин. `{3{d[0]}}` указывает на три копии `d[0]`. Не путайте 3-битную двоичную константу `3'b101` с шиной с именем b. Обратите внимание, что определение длины 3-битной константы имеет решающее значение; в противном случае в середине у могло бы появиться неизвестное количество нулей. Если бы размерность y превышала 9 бит, то нули были бы помещены в старших битах.

VHDL

```
y <=(c(2 downto 1), d(0), d(0), d(0), c(0), 3B“101”);
```

Оператор агрегирования используется для объединения шин. y должен быть 9-битным сигналом типа:

```
STD_LOGIC_VECTOR.
```

Другой пример демонстрирует возможности оператора агрегирования в VHDL. Предположим, что `z` – это 8-битный сигнал типа:
`STD_LOGIC_VECTOR`, тогда при выполнении операции агрегирования

```
z <= (“10”, 4 => ‘1’, 2 downto 1 =>‘1’, others =>‘0’)
```

`z` получит значение 10010110. “10” переходит в старшую пару битов. 1 также помещается в 4-й бит, и биты 2 и 1. Все остальные биты равны 0.

4.2.9 Манипуляция битами

Часто программистам приходится работать с фрагментом шины или сцеплять (объединять) сигналы для формирования шин. Эти операции называются манипуляциями битами. В **примере 4.12** y задается 9-ти битной переменной $c_2c_1d_0d_0d_0c_0101$ с использованием манипуляций битами.

4.2.10 Задержки

Операторы в HDL могут быть связаны с задержками, указанными в произвольных единицах. В процессе моделирования они помогают предсказать, насколько быстро будет работать схема (если вы укажете соответствующие задержки), также при отладке они помогают понять причину и следствие (устанавливать источник плохого результата сложно, если в результате моделирования все сигналы меняются одновременно). Эти задержки игнорируются в процессе синтеза; задержка элемента, сгенерированного синтезатором, зависит от значений t_{pd} и t_{cd}, а не от чисел в HDL-коде.

В **примере 4.13** добавлена задержка к первоначальной функции из **примера 4.1**, $y = \overline{abc} + a\overline{bc} + a\overline{b}c$. Предполагается, что инвертор имеет задержку 1 нс, трехвходовый элемент И имеет задержку 2 нс, а трехвходовый элемент ИЛИ – задержку 4 нс. **Рис. 4.1** показывает результаты моделирования с задержкой сигнала y 7нс относительно входов. Обратите внимание, что у неизвестно в начале моделирования.

Пример 4.13 ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ С ЗАДЕРЖКАМИ

SystemVerilog

```
`timescale 1ns/1ps
module example(input  logic a, b, c,
               output logic y);
```

```
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 |n2 | n3;
endmodule
```

Файлы SystemVerilog могут включать указание на шкалу времени, определяющее значение каждого промежутка времени. Это выражение имеет вид `‘timescale unit/precision`. В этом файле каждая единица времени равна 1нс, а моделирование проводится с точностью 1пс. Если в файле нет указания на шкалу времени, то единица времени и точность используются по умолчанию (обычно оба параметра равны 1нс). В SystemVerilog символ `#` используется для указания количества единиц задержки. Он может содержаться в операции `assign`, а так же в неблокирующих(<=) и блокирующих (=) присваиваниях, которые будут рассмотрены в **разделе 4.5.4**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;
architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
```

```
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

В VHDL заявление `after` используется для обозначения задержек. Единицы в этом случае определяются в наносекундах.

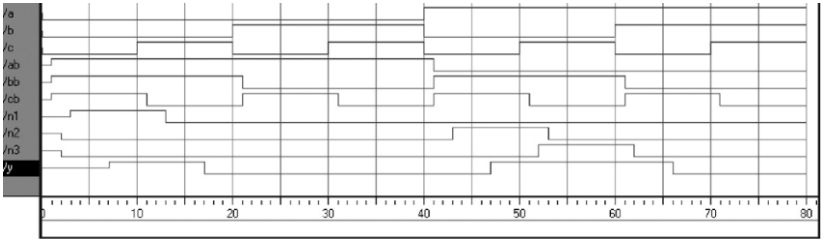


Рис. 4.10 **Пример моделирования сигналов с задержками (симулятор ModelSim)**

4.3 СТРУКТУРНОЕ МОДЕЛИРОВАНИЕ

В предыдущей главе обсуждалось поведенческое моделирование, описывающее модуль с точки зрения отношений между входами и выходами. Эта глава изучает структурное моделирование, описывающее модуль с точки зрения того, как он составлен из более простых модулей.

Например, **HDL-Пример 4.14** показывает, как собирается четырехвходовой мультиплексор из трех двухвходовых мультиплексоров. Каждая копия двухвходового мультиплексора называется экземпляром. Множество экземпляров одного модуля различаются отдельными названиями, в данном примере это lowmux, highmux и finalmux. Это пример системы, в которой двухвходовый мультиплексор повторно используется много раз.

Пример 4.14 HDL СТРУКТУРНАЯ МОДЕЛЬ ЧЕТЫРЕХВХОДОВОГО МУЛЬТИПЛЕКСОРА.

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);
  logic [3:0] low, high;
  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
```

```
  mux2 finalmux(low, high, s[1], y);
endmodule
```

Три экземпляра модуля `mux2` называются `lowmux`, `highmux` и `finalmux`. Модуль `mux2` должен быть где-нибудь объявлен в SystemVerilog-коде – (см. HDL-**примеры 4.5**, **4.15** или **4.34**)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:  mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

В архитектуре в первую очередь должны быть объявлены порты `mux2` при помощи оператора объявления компонента. Это позволяет инструментам VHDL проверить, что компонент, который вы хотите использовать, имеет те же порты, что и интерфейс, который был объявлен где-то еще в другом операторе интерфейса. Это позволяет предотвратить ошибки, вызванные изменением интерфейса, но не самого объекта. Однако объявление компонента делает VHDL-код довольно громоздким. Обратите внимание, что эта архитектура модуля `mux4` была названа `struct`, тогда как архитектуры модулей с поведенческими описаниями из **раздела 4.2** назывались `synth`. VHDL позволяет иметь множество архитектур (реализаций) одного интерфейса; архитектуры различаются по имени. Сами имена не имеют значения для инструментов САПР, но `struct` и `synth` являются общепринятыми. Синтезируемый VHDL-код, как правило, содержит только одну архитектуру для каждого интерфейса, так что мы не будем обсуждать VHDL-синтаксис, используемый для настройки того, какую архитектуру выбирать, когда определено множество из них.

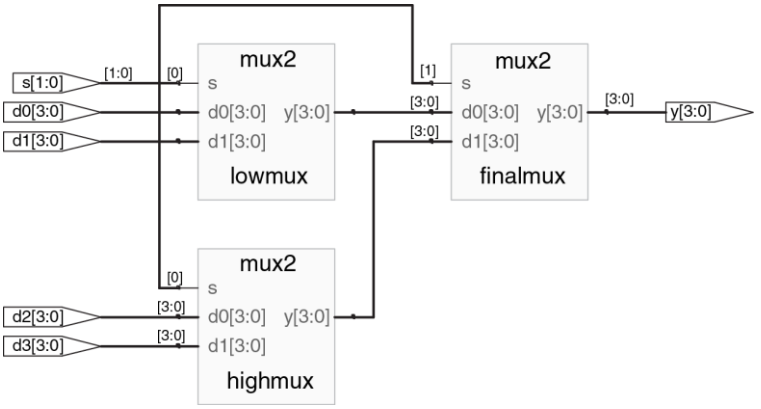


Рис. 4.11 **Синтезированная схема модуля `mux4`**

В **примере 4.15** HDL используется структурное моделирование для создания двухвходового мультиплексора из пары буферов с тремя состояниями. Однако построение логики из таких буферов не рекомендуется.

Пример 4.15 HDL СТРУКТУРНАЯ МОДЕЛЬ ДВУХВХОДОВОГО МУЛЬТИПЛЕКСОРА

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);
  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

В языке SystemVerilog выражения, такие как `~s`, разрешены в списке портов экземпляра. Допустимы выражения любой сложности, но это не поощряется, ибо они делают код сложным для чтения.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
          y: out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
```

```
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

В языке VHDL такие выражения, как `not s`, не разрешены в карте портов экземпляра. Таким образом, `sbar` должен быть определен как отдельный сигнал.

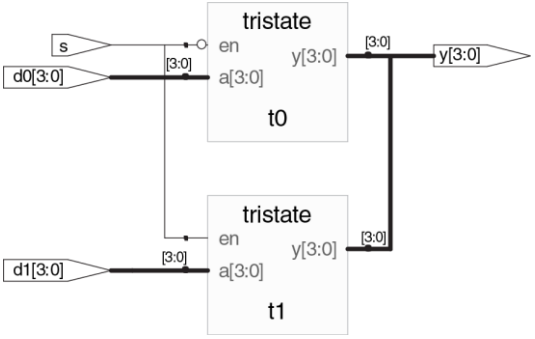


Рис. 4.12 **Синтезированная схема модуля `mux2`**

Пример 4.16 HDL показывает, как модули могут получать доступ к части шины. Двухвходовый мультиплексор разрядностью 8-бит построен с помощью двух четырехбитных двухвходовых мультиплексоров, объявленных ранее и работающих с младшим и старшим полубайтами.

В целом, сложные системы создаются иерархически. Система описывается структурно с помощью включения в неё основных компонентов. Каждый из этих компонентов описывается структурно из своих строительных блоков и так далее рекурсивно до тех пор, пока дело не дойдет до частей, достаточно простых для поведенческого описания. Хорошим стилем является стремление избежать (или по крайне мере минимизировать) смешения структурных и поведенческих описаний внутри одного модуля.

Пример 4.16 ОБРАЩЕНИЕ К ЧАСТЯМ ШИН

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);
  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin
  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

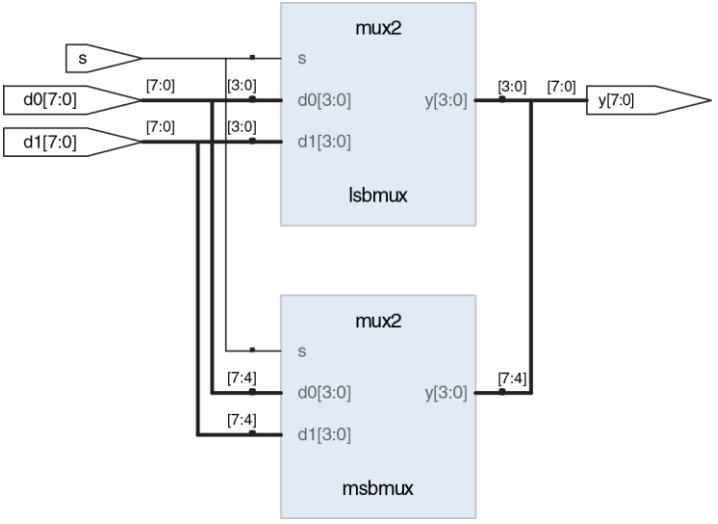


Рис. 4.13 **Синтезированная схема модуля mux2_8**

4.4 ПОСЛЕДОВАТЕЛЬНОСТНАЯ ЛОГИКА

Синтезаторы HDL распознают определенные идиомы и превращают их в конкретные последовательные схемы. Код, написанный в ином стиле, может правильно симулироваться, но в синтезированной схеме могут оказаться как грубые, так и труднораспознаваемые ошибки. В этом разделе представлены идиомы, рекомендованные для описания регистров и защелок.

4.4.1 Регистры

Подавляющее большинство современных коммерческих систем построено на регистрах, использующих срабатывающие по переднему фронту тактового импульса D-триггеры. В **примере 4.17** показана идиома для такого триггера.

Сигналы, значения которым присвоены в операторах `always` языка SystemVerilog и операторах `process` языка VHDL, сохраняют свое состояние, пока не случится событие из списка чувствительности оператора, приводящее к изменению их значения. Поэтому код, использующий эти операторы с соответствующими списками чувствительности, может описывать последовательные схемы с памятью. Например, у триггера в списке чувствительности есть только

сигнал `clk`, и потому триггер хранит старое значение `q` до следующего переднего фронта `clk`, даже если входной сигнал `d` изменился раньше.

В отличие от них, оператор непрерывного присваивания SystemVerilog (`assign`) и оператор одновременного присваивания VHDL (`<=`) перевычисляются каждый раз, когда изменяется какая-либо из переменных в правой части, поэтому эти операторы могут описать только комбинационную логику *(с помощью этих операторов можно описывать и логику, сохраняющую состояние, например,* `assign q = clk ? d : q`*; но делать это не рекомендуется – прим. переводчика)*.

Пример 4.17 РЕГИСТР

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);
  always_ff @(posedge clk)
    q <= d;
endmodule
```

В общем случае, оператор `always` языка SystemVerilog имеет вид

```
always @(sensitivity list)
  statement;
```

Оператор выполняется, только когда случается событие, заданное в списке чувствительности. В этом примере оператором является `q <= d` (читается "q принимает значение d"). Таким образом, триггер копирует d в q по переднему фронту тактового сигнала, а в остальное время значение q остается неизменным. Учтите, что список чувствительности также иногда называют списком стимулов.

`<=` называется неблокирующим присваиванием. Пока считайте его обычным присваиванием `=`; мы вернемся к трудноуловимой разнице между ними в **разделе 4.5.4**. Заметьте, что внутри оператора always неблокирующее присваивание `<=` используется вместо `assign`.

Как мы увидим в последующих разделах, операторы `always` можно использовать для создания триггеров, защелок или комбинационной логики в зависимости от списка чувствительности и оператора. Из-за подобной гибкости языка при синтезе аппаратных блоков можно непреднамеренно получить нежелательную конфигурацию. Во избежание таких ошибок в SystemVerilog введены операторы `always_ff`, `always_latch` и `always_comb`. Оператор `always_ff` ведет себя так же, как `always`, но используется только тогда, когда подразумевается синтез триггеров, и позволяет инструментальной среде в противном случае выдавать предупреждение.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

Оператор `process` языка VHDL имеет вид:

```
process(sensitivity list) begin
  statement;
end process;
```

Оператор выполняется, когда изменяется какая-либо из переменных из списка чувствительности. В этом примере оператор if проверяет, было ли изменение передним фронтом тактового сигнала (такта) clk. Если да, то q <= d (читается "q принимает значение d"). Таким образом, триггер копирует `d` в `q` по переднему фронту сигнала clk, а в остальное время значение `q` остается неизменным.

Другой вариант идиомы VHDL для записи триггера:

```
process(clk) begin
  if clk'event and clk = '1' then
     q <= d;
  end if;
end process;
```

`rising_edge(clk)` is synonymous with `clk'event and clk = '1'`.

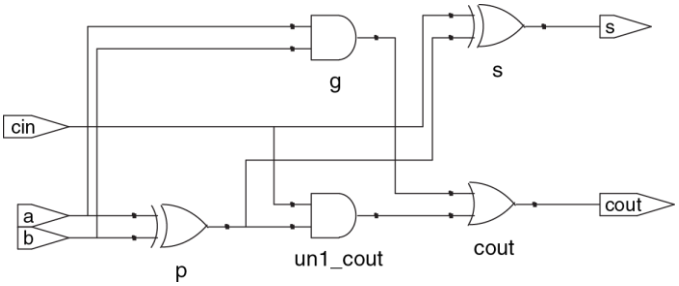


Рис. 4.14 **Синтезированная схема модуля `flop`**

4.4.2 Регистры со сбросом

В начале симуляции или сразу после подачи питания на схему значения на выходе триггеров или регистров неизвестны, что обозначается как значение `x` в SystemVerilog или как `u` в VHDL. На практике полезно использовать регистры со сбросом, чтобы при включении можно было привести систему в предопределенное состояние. Сброс может быть синхронным или асинхронным. Помните, что асинхронный сброс происходит немедленно, в отличие от синхронного, который сбрасывает выходной сигнал только по следующему переднему фронту такта. В **примере 4.18** показаны идиомы для триггеров с асинхронным и синхронным сбросом. Имейте в виду, что отличить синхронный и асинхронный сброс на принципиальной схеме может быть непросто. Например, Synplify Premier помещает на схемах асинхронный сброс на нижней стороне триггера, а синхронный − на левой.

Пример 4.18 РЕГИСТР СО СБРОСОМ

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);
  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);
  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

Сигналы в списке чувствительности оператора `always` разделяются запятой или словом `or`. Заметьте, что у триггера с асинхронным сбросом в списке чувствительности `posedge reset` есть, а у триггера с синхронным сбросом этого сигнала нет. Поэтому триггер с асинхронным сбросом реагирует на передний фронт `reset` немедленно, а с синхронным – только по переднему фронту такта.

В примере у обоих модулей одно и то же имя `flopr`, поэтому в схеме можно использовать либо один модуль, либо другой.

VHDL

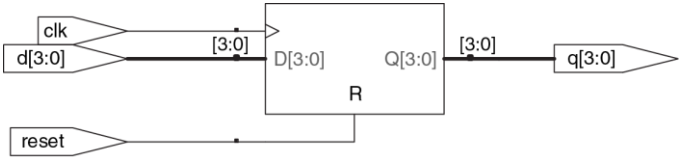
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port(clk, reset: in STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;
```

```
architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= “0000”;
      else q <= d;
      end if;
    end if;
  end process;
end;
```

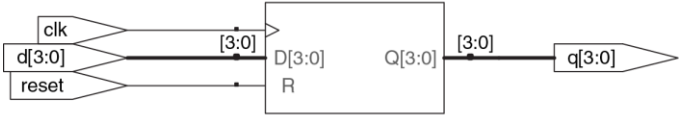
Сигналы в списке чувствительности оператора `process` разделяются запятой. Заметьте, что у триггера с асинхронным сбросом в списке чувствительности `reset` есть, а у триггера с синхронным сбросом − нет. Поэтому триггер с асинхронным сбросом реагирует на передний фронт `reset` немедленно, а с синхронным− только по переднему фронту такта.

Помните, что состояние триггера инициализируется как `'u'` при старте симуляции VHDL. Как уже упоминалось, имя архитектуры (в данном примере `synchronous` или `asynchronous`) игнорируется инструментальной средой, но помогает людям, читающим код.

Так как обе архитектуры описывают один и тот же объект flopr, в схеме можно использовать либо одну архитектуру, либо другую.



(a)



(b)

Рис. 4.15 Синтезированная схема модуля `flopr`, (а) с асинхронным сбросом, (б) с синхронным сбросом

4.4.3 Регистры с сигналом разрешения

Регистры с сигналом разрешения реагируют на тактовый импульс только при условии подачи активного уровня на линию разрешения.

В **примере 4.19** показан регистр с условием `en` и асинхронным сбросом `reset`, сохраняющий предыдущее значение, если оба сигнала имеют значение FALSE.

Пример 4.19 РЕГИСТР С УСЛОВИЕМ И СБРОСОМ

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

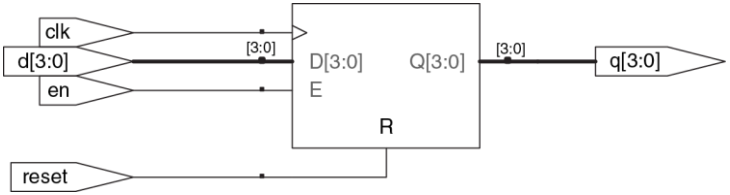


Рис. 4.16 **Синтезированная схема модуля `flopenr`**

4.4.4 Группы регистров

Один оператор `always/process` можно использовать для описания нескольких элементов аппаратуры. Рассмотрим, например, синхронизатор из **раздела 3.5.5**, состоящий из двух последовательных триггеров, показанный на **Рис. 4.17** и описанный в **примере 4.20**. По переднему фронту `clk`, `d` копируется в `n1`, и в то же время `n1` копируется в `q`.

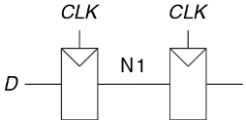


Рис. 4.17 Схема синхронизатора

Пример 4.20 СИНХРОНИЗАТОР

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Обратите внимание на конструкцию `begin/end`. Они обрамляют группу из нескольких операторов, находящихся внутри оператора `always`, наподобие скобок `{}` в C или Java. Конструкция `begin/end` не была нужна в примере `flopr`, потому что `if/else` считается одним оператором.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;
architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
        n1 <= d;
        q <= n1;
    end if;
  end process;
end;
```

Переменная `n1` должна быть декларирована как `signal`, так как она используется внутри модуля в качестве сигнала для соединения логических элементов

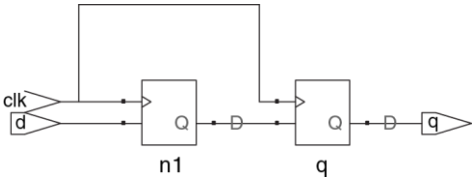


Рис. 4.18 **Синтезированная схема модуля `sync`**

4.4.5 Защелки

Возвращаясь к **разделу 3.2.2**, вспомним, что D-защелка открыта при высоком уровне тактового сигнала, т. е. пропускает сигнал данных с входа на выход. Защелка закрывается, когда уровень становится низким, сохраняя свое значение. Фрагмент кода в **примере 4.21** показывает идиому для D-защелки.

Не все программы-синтезаторы хорошо справляются с защелками. Если вы не уверены, что ваш синтезатор их поддерживает, или нет особых причин использовать именно защелки, пользуйтесь вместо них триггерами, работающими по фронту сигнала. Также нужно следить, чтобы в коде на HDL не было конструкций, приводящих к появлению нежелательных защелок, что легко может произойти в результате

невнимательности. Многие программы синтеза предупреждают, когда создают защелку; и если вы ее не ждали, то ищите ошибку в своем коде. А если вы не знаете, нужна ли вам в схеме защелка или нет, то это скорее всего значит, что вы пишете на HDL, как на обычном языке программирования, и у вас впереди могут быть большие проблемы.

Пример 4.21 D-ЗАЩЕЛКА

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` is в данном случае эквивалентно `always @(clk, d)` и оптимально для описания защелки на SystemVerilog. Оператор `always_latch` вычисляется при каждом изменении `clk` или `d`.
При высоком уровне `clk` переменная `q` принимает значение `d`, т.е. этот код описывает защелку, активную по высокому уровню.
В противном случае `q` сохраняет свое значение. SystemVerilog может выдавать предупреждение, если оператор `always_latch` не описывает реальную защелку.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

В списке чувствительности есть и `clk`, и `d`, так что `process` вычисляется каждый раз, когда `clk` или `d` изменяются. При высоком уровне `clk` переменная q принимает значение d.

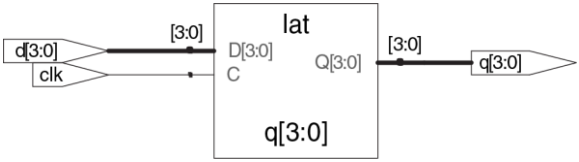


Рис. 4.19 **Синтезированная схема модуля `latch`**

4.5 И СНОВА КОМБИНАЦИОННАЯ ЛОГИКА

В **разделе 4.2** мы использовали операторы присваивания для поведенческого описания комбинационной логики. Операторы `always` языка SystemVerilog и операторы `process` языка VHDL используются для описания последовательных схем, потому что они сохраняют состояние переменных, если не было указано их изменить. Однако, эти операторы можно использовать и для поведенческого описания комбинационной логики, если список чувствительности написан так, чтобы отвечать на любое изменение входных сигналов, и тело оператора определяет значение выходного сигнала при любой комбинации значений входов. Код на HDL в **примере 4.22** использует операторы `always/process` для описания группы из 4-х инверторов (синтезированную схему см. на **Рис. 4.3**).

Пример 4.22 ИНВЕРТОР с помощью `always/process`

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic[3:0] y);
  always_comb
    y = ~a;
endmodule
```

Оператор `always_comb` исполняет выражения внутри оператора `always` каждый раз, когда изменяется любой из сигналов в правой части `<=` или `=` оператора `always`. В данном случае это эквивалентно `always @(a)`, но гораздо надежнее, так как позволяет избегать ошибок в случае переименования или добавления сигналов в оператор `always`.
Если код внутри оператора `always_comb` не является комбинационной логикой, то тогда SystemVerilog будет выдавать предупреждение. Оператор `always_comb` эквивалентен `always @(*)`, но является более предпочтительным в SystemVerilog.Равенство = в операторе always называется блокирующим присваиванием, в отличие от неблокирующего присваивания `<=`. В SystemVerilog хорошей практикой является использование блокирующих присваиваний для комбинационной логики и неблокирующих – для последовательной. Это будет далее обсуждаться в **разделе 4.5.4**.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
```

```
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

Оператор `process(all)` исполняет все выражения внутри `process`, как только изменяется любой из сигналов оператора process. Это эквивалентно `process(a)`, но существенно лучше, так как позволяет избежать ошибок при переименовании или добавлении новых сигналов.
Операторы `begin` и `end process` обязательны в VHDL, даже если `process` содержит только одно присваивание.

В обоих языках можно использовать блокирующие и неблокирующие присваивания в операторах `always/process`. Внутри одного оператора блокирующие присваивания выполняются в том порядке, в котором они написаны, в точности как в обычном языке программирования, а обновление значений переменных в левой части неблокирующих присваиваний выполняется "одновременно", после того, как вычислены значения всех правых частей неблокирующих присваиваний.

Код в **примере 4.23** описывает полный сумматор, в котором использованы промежуточные сигналы `p` и `g` для вычисления `s` и `cout`. В результате получается та же схема, что и на **Рис. 4.8**, но с использованием операторов `always/process` вместо операторов присваивания.

Эти два примера не очень удачны для демонстрации использования `always/process` для комбинационной логики – в них больше строк кода, чем в эквивалентных примерах на HDL **4.2** и **4.7** с использованием операторов присваивания. Однако для моделирования более сложной комбинационной логики удобно пользоваться операторами `case` и `if`, которые допускаются лишь внутри операторов `always/process`. Их мы рассмотрим в следующих разделах.

SystemVerilog

В операторе `always` знак равенства = означает блокирующее присваивание, а <= означает неблокирующее (также известное как одновременное) присваивание. Не путайте эти два присваивания с непрерывным присваиванием с помощью оператора `assign`. Операторы `assign` должны использоваться вне операторов `always` и тоже вычисляются одновременно.

VHDL

В операторе process := означает блокирующее присваивание, а <= означает неблокирующее (также известное как одновременное) присваивание. Неблокирующие присваивания применяются к выходам и к сигналам. Блокирующие присваивания применяются к переменным, объявленным в операторах `process` (см. код в **примере 4.23**). Символ **<=** может использоваться и за пределами операторов `process`, где тоже исполняется одновременно.

Пример 4.23 ПОЛНЫЙ СУММАТОР с помощью `always/process`

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;
  always_comb
    begin
      p = a ^ b;              // blocking
      g = a & b;              // blocking
      s = p ^ cin;            // blocking
      cout = g |(p & cin);    // blocking
    end
endmodule
```

Здесь эквивалентом `always_comb` было бы `always @(a, b, cin)`, но `always_comb` лучше, поскольку позволяет избежать ошибок, связанных с недостающими в списке чувствительности сигналами.

По причинам, которые мы обсудим в **разделе 4.5.4**, для комбинационной логики лучше использовать блокирующие присваивания. В этом примере они использованы для вычисления вначале `p`, затем `g`, `s`, и `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;
architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Здесь эквивалентом оператора `process(all)` был бы `process(a, b, cin)`, но `process(all)` лучше, поскольку позволяет избежать ошибок, связанных с недостающими в списке чувствительности сигналами.

По причинам, которые мы обсудим в **разделе 4.5.4**, для промежуточных переменных в комбинационной логике лучше использовать блокирующие присваивания. В этом примере они использованы для вычисления новых значений `p` и `g`, необходимых для последующего вычисления `s` и `cout`.

Так как `p` и `g` упоминаются в левой части операторов блокирующего присваивания (`:=`) в операторе `process`, то они должны быть объявлены как

`variable`, а не как `signal`. Объявление переменных пишется перед `begin` того процесса, в котором эти переменные используются.

4.5.1 Операторы case

Вот более удачный пример использования операторов `always/process` для комбинационной логики − дешифратор для семисегментного индикатора, выполненный с использованием оператора `case`, который можно писать лишь внутри оператора `always/process`.

Как вы могли заметить из **примера 2.10** дешифратора семисегментного индикатора, процесс разработки больших блоков комбинационной логики утомителен и чреват ошибками. Языки описания аппаратуры облегчают этот процесс, позволяя определять функциональность на более высоком уровне абстракции, и затем автоматически синтезировать ее в вентили. В коде **примера 4.24** используется оператор `case` для описания дешифратора семисегментного индикатора по таблице истинности. Оператор `case` выполняет различные действия в зависимости от значения его входных данных. Он синтезируется в комбинационную логику, если все возможные сочетания входных данных определены; в противном случае получится последовательная логика, т.к. выход сохранит свое предшествующее значение в неопределенных случаях.

Пример 4.24 ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО ИНДИКАТОРА

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                    abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
   endcase
endmodule
```

Оператор `case` проверяет значение `data`; если `data` равно 0, выполнится действие после двоеточия, т. е. установка `segments` в 1111110. Аналогично проверяются другие значения `data` вплоть до 9 (обратите внимание, что по умолчанию система счисления десятичная).Условие `default` (по умолчанию) – удобный способ определить выход для всех случаев, не упомянутых явно,

тем самым гарантируя в результате комбинационную логику. В SystemVerilog операторы `case` обязаны находиться внутри операторов `always`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;
architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                          abcdefg
      when X"0" =>   segments <= "1111110";
      when X"1" =>   segments <= "0110000";
      when X"2" =>   segments <= "1101101";
      when X"3" =>   segments <= "1111001";
      when X"4" =>   segments <= "0110011";
      when X"5" =>   segments <= "1011011";
      when X"6" =>   segments <= "1011111";
      when X"7" =>   segments <= "1110000";
      when X"8" =>   segments <= "1111111";
      when X"9" =>   segments <= "1110011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

Оператор `case` проверяет значение `data`; если `data` равно 0, выполнится действие после `=>`, т. е. установка `segments` в 1111110. Аналогично проверяются другие значения data вплоть до 9 (обратите внимание на использование X для обозначения шестнадцатеричных чисел). Условие `others` (другие) – удобный способ определить выход для всех случаев, не упомянутых явно, тем самым гарантируя в результате комбинационную логику.
В отличие от SystemVerilog, в VHDL допускаются присваивания сигналам с выбором (см. **пример 4.6**), которые по сути похожи на операторы `case`, но могут встречаться и за пределами операторов `process`, так что поводов использовать операторы `process` для описания комбинационной логики в VHDL меньше.

Рис. 4.20 **Синтезированная схема модуля `sevenseg`**

Synplify Premier синтезирует дешифратор семисегментного индикатора как постоянную память (ПЗУ), содержащую 7 выходов для каждой из 16 возможных комбинаций входов. ПЗУ обсуждаются в **разделе 5.5.6**.

Если бы условие `default` или `others` не было упомянуто в операторе `case`, то дешифратор сохранял бы предыдущее значение выхода,

когда вход находится в диапазоне 10-15. Для аппаратуры такое поведение было бы странно.

Обычные дешифраторы тоже часто записываются с помощью операторов `case`. В коде **примера 4.25** представлен дешифратор 3:8.

Пример 4.25 ДЕШИФРАТОР 3:8

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

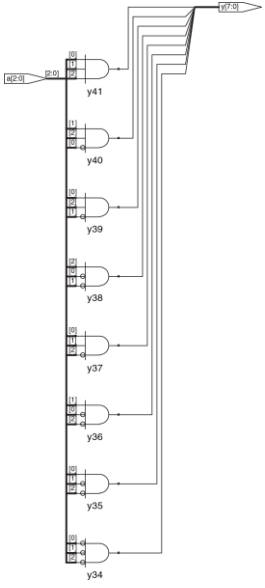


Рис. 4.21 Синтезированная схема модуля `decoder3_8`

Строго говоря, условие `default` в данном случае не нужно для синтеза, поскольку перечислены все возможные сочетания входов, но оно полезно для симуляции на случай, если какой-либо из входов равен `x` или `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000" =>  y <= "00000001";
      when "001" =>  y <= "00000010";
      when "010" =>  y <= "00000100";
      when "011" =>  y <= "00001000";
      when "100" =>  y <= "00010000";
      when "101" =>  y <= "00100000";
      when "110" =>  y <= "01000000";
      when "111" =>  y <= "10000000";
      when others => y <= "XXXXXXXX";
    end case;
  end process;
end;
```

Строго говоря, условие `others` в данном случае не нужно для синтеза, поскольку перечислены все возможные сочетания входов, но оно полезно для симуляции на случай, если какой-либо из входов равен `x`, `z` или `u`.

4.5.2 Операторы if

Операторы `always/process` могут содержать также операторы `if`, за которыми может следовать оператор `else`. Если все возможные сочетания входов обработаны, то оператор синтезируется в комбинационную логику, иначе – в последовательную (например, защелка в **разделе 4.4.5**).

В **примере 4.26** используются операторы `if` для описания схемы приоритетов, определенной в **разделе 2.4**. Вспомним, что *N*-входовая схема приоритетов устанавливает в значение TRUE тот из выходов, который соответствует наиболее приоритетному входу, равному TRUE.

Пример 4.26 СХЕМА ПРИОРИТЕТОВ

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y <= 4'b1000;
    else if (a[2]) y <= 4'b0100;
    else if (a[1]) y <= 4'b0010;
    else if (a[0]) y <= 4'b0001;
    else           y <= 4'b0000;
endmodule
```

В SystemVerilog операторы `if` обязаны быть внутри операторов `always`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if   a(3) then y <= "1000";
    elsifa(2) then y <= "0100";
    elsifa(1) then y <= "0010";
    elsifa(0) then y <= "0001";
    else           y <= "0000";
    end if;
  end process;
end;
```

В отличие от SystemVerilog, в VHDL есть операторы условного присваивания (см. **пример 4.6**), которые по сути похожи на операторы `if`, но могут встречаться и за пределами операторов `process`, так что поводов использовать процессы для описания комбинационной логики в VHDL меньше.

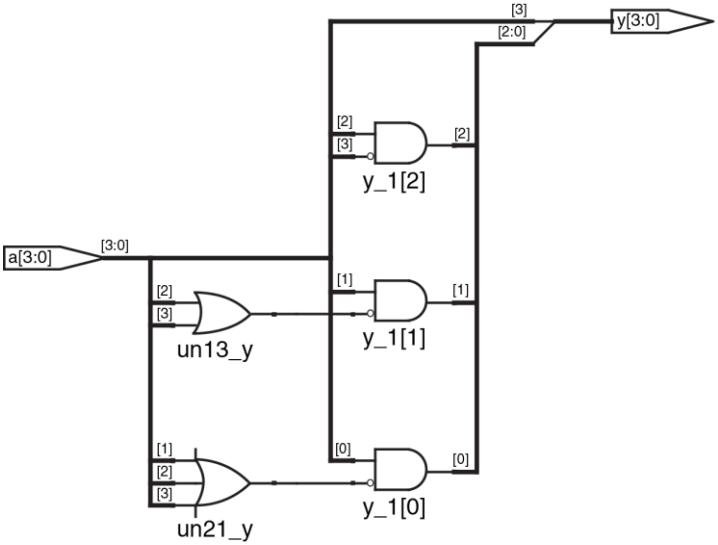


Рис. 4.22 **Синтезированная схема модуля `priorityckt`**

4.5.3 Таблицы истинности с незначащими битами

Как показано в разделе 2.7.3, в таблицах истинности могут быть незначащие биты ради упрощения логики. В коде примера 4.27 показано, как описать приоритетную схему с незначащими битами.

Synplify Premier синтезирует схему для этого модуля, показанную на Рис. 4.23, которая слегка отличается от схемы приоритетов на Рис. 4.24, но они логически эквивалентны.

Пример 4.27 СХЕМА ПРИОРИТЕТОВ С НЕЗНАЧАЩИМИ БИТАМИ

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y <= 4'b1000;
      4'b01??: y <= 4'b0100;
      4'b001?: y <= 4'b0010;
      4'b0001: y <= 4'b0001;
      default: y <= 4'b0000;
   endcase
endmodule
```

Оператор `casez` работает так же, как и case, но еще и распознает знак «?» как незначащий бит.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture dontcare of priority_casez is
begin
  process(all) begin
    casez a is
      when "1---" =>y <= "1000";
      when "01--" =>y <= "0100";
      when "001-" =>y <= "0010";
      when "0001" =>y <= "0001";
      when others =>y <= "0000";
    end casez;
  end process;
end;
```

Оператор `casez` работает так же, как и `case`, но еще и распознает знак «-» как незначащий бит.

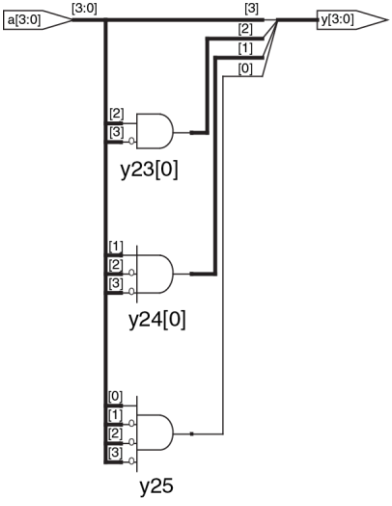


Рис. 4.23 **Синтезированная схема модуля priority_casez**

4.5.4 Блокирующие и неблокирующие присваивания

В кратком руководстве ниже объясняется, когда и как использовать тот или иной тип присваивания. Если ему не следовать, то можно написать код, который, возможно, будет работать в режиме симуляции, но будет синтезироваться в некорректную схему. Далее в этом разделе для любознательного читателя объясняются принципы, лежащие в основе данного руководства.

КРАТКОЕ РУКОВОДСТВО ПО БЛОКИРУЮЩИМ И НЕБЛОКИРУЮЩИМ ПРИСВАИВАНИЯМ

SystemVerilog

1. Используйте `always_ff @(posedge clk)` и неблокирующие присваивания для моделирования последовательной логики.

```
always_ff @(posedge clk)
begin
n1 <= d; // неблокирующее
q <= n1; // неблокирующее

end
```

2. Используйте непрерывные присваивания для моделирования простой комбинационной логики.

```
assign y = s ? d1 : d0;
```

3. Используйте `always_comb` и блокирующие присваивания для моделирования более сложной комбинационной логики, когда удобнее использовать оператор `always`.

```
always_comb
begin
p = a ^ b; // блокирующее
g = a & b; // блокирующее
s = p ^ cin;
cout = g | (p & cin);
end
```

4. Не присваивайте значение одному и тому же сигналу в разных операторах `always` или непрерывных присваиваниях.

VHDL

1. Используйте `process(clk)` и неблокирующие присваивания для моделирования синхронной последовательной логики.

```
process(clk) begin
if rising_edge(clk) then
n1 <= d; -- nonblocking
q <= n1; -- nonblocking
end if;
end process;
```

2. Используйте одновременные присваивания вне операторов `process` для моделирования простой комбинационной логики.

```
y <= d0 when s = '0' else d1;
```

3. Используйте `process(all)` для моделирования более сложной комбинационной логики, если оператор `process` удобнее. Пользуйтесь блокирующими присваиваниями для локальных переменных.

```
process(all)
variable p, g: STD_LOGIC;
begin
p := a xor b; -- блокирующее
g := a and b; -- блокирующее
s <= p xor cin;
cout <= g or (p and cin);
end process;
```

4. Не присваивайте значение одной и той же переменной в разных операторах `process` или одновременных присваиваниях.

Комбинационная логика*

Полный сумматор в коде **примера 4.23** корректно смоделирован с использованием блокирующих присваиваний. В этом разделе мы рассмотрим, как он работает и чем он отличается от модели, использующей неблокирующие присваивания.

Представьте, что значения `a`, `b` и `cin` первоначально равны 0. Значения `p`, `g`, `s` и `cout` будут тоже равны 0. В какой-то момент `a` изменяется на 1, активируя оператор `always/process`. Четыре блокирующих присваивания выполняются в показанном ниже порядке (В случае VHDL присваивания `s` и `cout` выполняются одновременно). Заметьте, что `p` и `g` получают свои новые значения до `s` и `cout` благодаря блокирующим присваиваниям. Это важно, потому что мы хотим вычислять `s` и `cout`, пользуясь новыми значениями `p` и `g`.

1. `p` ← 1 ⊕ 0 = 1

2. `g` ← 1 • 0 = 0

3. `s` ← 1 ⊕ 0 = 1

4. `cout` ← 0 + 1 • 0 = 0

Пример 4.28 иллюстрирует использование неблокирующих присваиваний.

Рассмотрим тот же случай, когда `a` из 0 становится 1, в то время как `b` и `cin` равны 0. Четыре неблокирующих присваивания выполняются одновременно:

`p` ← 1 ⊕ 0 = 1 `g` ← 1 • 0 = 0 `s` ← 0 ⊕ 0 = 0 `cout` ← 0 + 0 • 0 = 0

Пример 4.28 ПОЛНЫЙ СУММАТОР С НЕБЛОКИРУЮЩИМИ ПРИСВАИВАНИЯМИ

SystemVerilog

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;
  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking
      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule;
```

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;
architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
```

```
  process(all) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Так как `p` и `g` упоминаются в левой части неблокирующих присваиваний в операторе `process`, они должны быть объявлены как `signal`, а не как `variable`. Объявление сигналов записываются перед `begin` в `architecture`, а не в `process`.

Таким образом, `s` вычисляется одновременно с `p`, и потому использует старое значение `p`. Из-за этого `s` остается равным 0, а не становится 1. Однако, `p` изменяется с 0 на 1. Это изменение вызывает исполнение оператора `always/process` во второй раз:

`p` ← 1 ⊕ 0 = 1 `g` ← 1 • 0 = 0 `s` ← 1 ⊕ 0 = 1 `cout` ← 0 + 1 • 0 = 0

На этот раз `p` уже равно 1, и `s`, как и следует, становится равным 1. Неблокирующие присваивания в конце-концов приходят к правильному ответу, но оператору `always/process` приходится выполняться дважды. От этого симуляция получается медленнее, хотя код и синтезируется в ту же схему.

Еще один недостаток неблокирующих присваиваний для моделирования комбинационной логики – при симуляции может получиться неверный результат, если забыть упомянуть промежуточные переменные в списке чувствительности *(при использовании* `always_comb` *и* `process(all)` *для комбинационной логики этот недостаток неактуален – прим. Переводчика)*.

Хуже того, некоторые синтезаторы создадут правильную схему, даже если неверный список чувствительности приводит к неверной симуляции. Это ведет к несовпадению результатов симуляции и реального поведения аппаратуры.

SystemVerilog

Если бы список чувствительности оператора `always` в коде **примера 4.28** был написан как `always@(a, b, cin)`, а не как `always_comb`, оператор не выполнился бы повторно, когда изменяется `p` или `g`. В этом случае `s` ошибочно остался бы равным 0 вместо 1.

VHDL

Если бы список чувствительности оператора `process` в коде **примера 4.28** был записан как `process(a, b, cin)` а не как `process(all)`, оператор не выполнился бы повторно, когда изменяется `p` или `g`. В этом случае `s` ошибочно остался бы равным 0 вместо 1.

Последовательная логика*

Синхронизатор в коде **примера 4.20** корректно смоделирован с использованием неблокирующих присваиваний. По переднему фронту тактового сигнала `d` копируется в `n1` в то же время, как `n1` копируется в `q`, так что код, как и следует, описывает два регистра. Например, пусть первоначально `d`=0, `n1`=1 и `q`=0. По переднему фронту тактового сигнала одновременно выполняются два присваивания, так что после прохождения фронта `n1`=0 и `q`=1:

`n1` ← `d` = 0 `q` ← `n1` = 1

В коде **примера 4.29** делается попытка описать тот же модуль с помощью блокирующих присваиваний. По переднему фронту `clk`, `d` копируется в `n1`. Затем это новое значение `n1` копируется в `q`, в результате чего значение `d` ошибочно оказывается и в `n1`, и в `q`. Присваивания выполняются одно за другим, так что после фронта сигнала, `q` = `n1` = 0.

1. `n1` ← `d` = 0

2. `q` ← `n1` = 0

Оттого, что переменная `n1` не видна окружающему миру и не влияет на поведение `q`, синтезатор ликвидирует ее в процессе оптимизации, как показано на **Рис. 4.24**.

Пример 4.29 ПЛОХОЙ СИНХРОНИЗАТОР С БЛОКИРУЮЩИМИ ПРИСВАИВАНИЯМИ

SystemVerilog

```
// Bad implementation of a synchronizer using blocking // assignments
module syncbad(input  logic clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation of a synchronizer using blocking -- assignment
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;
architecture bad of syncbad is
begin
  process(clk)
```

```
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

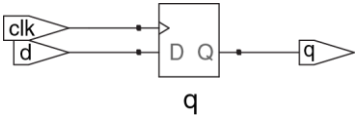


Рис. 4.24 Синтезированная схема для syncbad

Мораль этой иллюстрации такова: для моделирования последовательной логики в операторах `always/process` следует пользоваться исключительно неблокирующими присваиваниями. С помощью разных хитростей, например, изменения порядка присваиваний, можно добиться правильной работы блокирующих присваиваний, но они не дают никаких преимуществ, а лишь привносят риск нежелательного поведения. Некоторые последовательные схемы не будут работать с использованием блокирующих присваиваний независимо от их порядка *(авторы предлагают взять на веру, что не*

стоит использовать в SystemVerilog блокирующее присваивание для последовательной логики, даже если оно в операторе `always` *единственное. Это связано с особенностями алгоритмов симуляции SystemVerilog, в подробности которых мы не будем вдаваться – прим. переводчика)*.

4.6 КОНЕЧНЫЕ АВТОМАТЫ

Как вы помните, конечный автомат (КА) состоит из регистра состояния и двух блоков комбинационной логики для вычисления следующего состояния и выхода при заданных в текущем состоянии и информации на входе, как показано на **Рис. 3.22**. Описания конечных автоматов на HDL, соответственно, состоят из трех частей, моделирующих регистр состояния, логику следующего состояния и логику выхода.

Пример 4.30 КОНЕЧНЫЙ АВТОМАТ, ДЕЛЯЩИЙ НА 3

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;
  // state register
  always_ff @(posedge clk, posedge reset)
```

```
    if (reset) state <= S0;
    else       state <= nextstate;
  // next state logic
  always_comb
    case (state)
      S0:      nextstate <= S1;
      S1:      nextstate <= S2;
      S2:      nextstate <= S0;
      default: nextstate <= S0;
    endcase
  // output logic
  assign y = (state = = S0);
endmodule
```

Оператор `typedef` определяет значение `statetype` как двухбитный `logic тип` с тремя возможными значениями: `S0`, `S1` или `S2`. `state` и `nextstate` – сигналы типа `statetype`.

Константам перечисления, упомянутым в определении типа, по умолчанию присваиваются порядковые значения: `S0=00`, `S1=01`, и `S2=10`. Они могут быть явно изменены пользователем, но программа-синтезатор рассматривает их как рекомендацию, а не как требование. Например, следующий фрагмент кодирует состояния трехбитным унарным (1-hot) кодом:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100}
statetype;
```

Обратите внимание на оператор `case`, определяющий функцию переходов. Из-за того, что логика для следующего состояния должна быть комбинационной, условие default (значения по умолчанию) является обязательным, даже несмотря на то, что состояния `2'b11` не бывает.
Выход y равен 1, когда автомат находится в состоянии `S0`. Результат операции сравнения на равенство `a == b` равен 1, когда `a` равно `b`, и 0 в противном случае. Операция сравнения на неравенство `a != b`, наоборот, дает 1, когда `a` не равно `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;
  -- next state logic
```

```
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;
  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

В этом примере определяется новый тип перечисления данных `statetype` с тремя возможными значениями: `S0`, `S1` и `S2`. `state` и `nextstate` – сигналы типа `statetype`. Благодаря использованию перечисления, а не явно задаваемых кодов состояний, VHDL позволяет синтезатору выбрать оптимальный код для состояний.
Выход `y`, равен 1, когда `state` равно `S0`. Операция сравнения на неравенство записывается как `/=`. Чтобы получить на выходе 1, когда состояние отлично от `S0`, замените сравнение на `state /= S0`.

В **примере 4.30** описывается КА деления на 3 из **раздела 3.4.2**. Для инициализации КА используется асинхронный сброс. Регистр состояния использует стандартную идиому для триггеров. Логика формирования следующего состояния и выхода является комбинационной.

Программа-синтезатор Synplify Premier порождает лишь блочную диаграмму и диаграмму переходов для автомата; она не показывает логические элементы или входы и выходы на узлах и дугах, поэтому следует проверить по диаграмме, правильно ли вы определили КА в HDL коде.

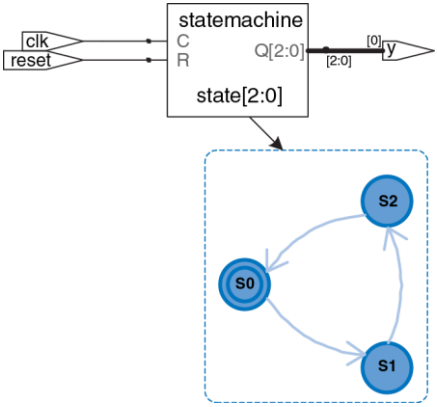


Рис. 4.25 Синтезированная схема модуля divideby3fsm

Диаграмма переходов на **Рис. 4.25** для КА деления на 3 аналогична диаграмме на **Рис. 3.28 (b)**. Двойной кружок означает, что автомат сбрасывается в состояние S0. Реализация автомата на уровне вентилей была показана в **разделе 3.4.2**.

Заметьте, что состояния обозначены константами перечисления, а не двоичными значениями. Благодаря этому код становится более читабельным и его легче изменять.

Если по какой-либо причине мы захотим, чтобы выход был равен 1 в состояниях S0 и S1, выходная логика изменится следующим образом:

SystemVerilog

```
// выходная логика
assign y = (state== S0 | state== S1);
```

VHDL

```
-- выходная логика
y <= '1' when (state = S0 or state = S1) else '0';
```

Следующие два примера описывают КА распознавателя – битового шаблона улитки из **раздела 3.4.3**. В коде показано, как использовать операторы `case` и `if` для обработки следующего состояния и выходной логики, зависящей и от входа, и от текущего состояния. В автомате Мура (**пример 4.31**) выход зависит только от текущего состояния, а в автомате Мили (**пример 4.32**) выход зависит и от текущего состояния, и от входов.

Пример 4.31 АВТОМАТ МУРА ДЛЯ РАСПОЗНАВАНИЯ ПАТТЕРНА

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;
  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;
  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase
  // output logic
  assign y = (state = = S2);
endmodule
```

Заметьте, что неблокирующие присваивания (<=) используются в регистре состояния для описания последовательной логики, а для комбинационной логики следующего состояния используются блокирующие присваивания (=).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;
architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then        state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;
  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
```

```
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;
  --output logic
  y <= '1' when state = S2 else '0';
end;
```

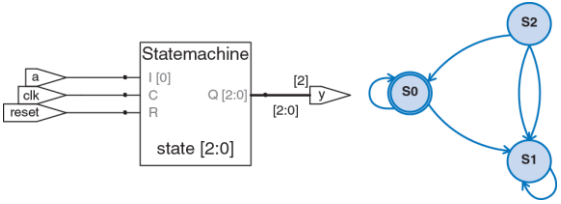


Рис. 4.26 Синтезированная схема модуля `patternMoore`

Пример 4.32 АВТОМАТ МИЛИ ДЛЯ РАСПОЗНАВАНИЯ БИТОВОГО ШАБЛОНА

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);
  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;
  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;
  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default: nextstate = S0;
    endcase
  // output logic
  assign y = (a & state = = S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;
architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then               state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;
  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
```

```
      when others =>
                  nextstate <= S0;
    end case;
  end process;
  -- output logic
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

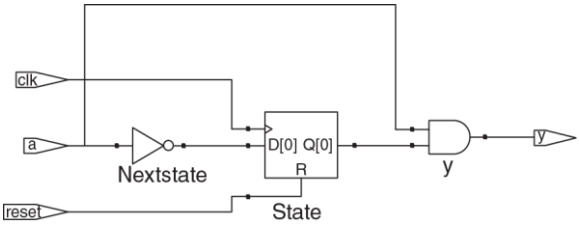


Рис. 4.27 Синтезированная схема модуля `patternMealy`

4.7 ТИПЫ ДАННЫХ*

В этом разделе более подробно рассматриваются особенности типов данных в SystemVerilog и VHDL.

4.7.1 SystemVerilog

В предшественнике SystemVerilog, языке Verilog, в основном, использовались два типа: `reg` и `wire`. Несмотря на свое название, сигнал типа `reg` не обязан соответствовать регистру, и эта путаница затрудняла изучение языка. Во избежание этой путаницы в SystemVerilog добавлен тип `logic`, который и используется в этой книге. В этом разделе подробно рассказывается о типах `reg` и `wire` для тех, кому предстоит читать старый код на языке Verilog.

В Verilog, если сигнал встречается в левой части оператора `<=` или `=` в `always`-блоке, он должен быть объявлен как `reg`, в противном случае – как `wire`. Поэтому сигнал типа `reg` может быть выходом триггера, защелки или комбинационной логики, в зависимости от списка чувствительности и оператора внутри `always`-блока.

У входных и выходных портов по умолчанию – тип `wire`, если их тип не объявлен как `reg`. Ниже показано, как триггер описывается на обычном Verilog. Обратите внимание, что сигналы `clk` и `d` – типа `wire` по

умолчанию, а `q` явно объявлен как `reg`, потому что он встречается в левой части оператора <= в `always`-блоке.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);
  always @(posedge clk)
    q <= d;
endmodule
```

Тип `logic`, добавленный в SystemVerilog – это синоним типа `reg`, но его название избавлено от нежелательных ассоциаций с триггером. Кроме того, в SystemVerilog ослаблены ограничения в части использования операторов `assign` и в иерархических назначениях портов, так что сигналы типа `logic` могут быть использованы вне блоков `always` – там, где традиционно требовались бы сигналы типа `wire`. Таким образом, подавляющее большинство сигналов в SystemVerilog может быть типа `logic`. Исключение – сигнал с несколькими источниками, например, тристабильная высокоимпедансная шина с тремя состояниями, который должен быть объявлен как цепь (net), как показано в коде **примера 4.10**. Благодаря этому правилу, когда сигнал типа `logic` по ошибке подключен к нескольким источникам, SystemVerilog выдает сообщение об ошибке

уже во время компиляции, а не присваивает ему значение `x` во время симуляции.

Наиболее распространенные типы цепей – `wire` и `tri`. Эти два типа – синонимы, но `wire` традиционно используется, когда источник (driver) один, а tri – когда их несколько. В SystemVerilog в типе `wire` нет необходимости: для сигналов с одним источником `logic` предпочтительнее.

Когда у всех активных источников цепи типа `tri` одно и то же значение, она получает это значение. Если все источники неактивны, цепь отключена (`floats`) (`z`). Если у активных источников разные значения (`0`, `1`, `x`), то цепь находится в состоянии конфликта (in contention) (`x`).

Есть и другие типы цепей, значения которых определяются по-другому при неактивных источниках или в случае конфликта. Эти типы используются редко, но могут встречаться там же, где и тип `tri` (например, для цепей с несколькими источниками). Они описаны в **Табл. 4.7**.

Табл. 4.7 Определение значения цепей

Тип цепи	Значение при неактивных источниках	Значение при конфликте источников
tri	z	x
trireg	предыдущее значение	x
triand	z	0, если есть хоть один 0
trior	z	1, если есть хоть одна 1
tri0	0	x
tri1	1	x

4.7.2 VHDL

В отличие от SystemVerilog, язык VHDL – со строгой типизацией, что защищает пользователя от некоторых ошибок, но временами он неуклюж.

Несмотря на то, что тип `STD_LOGIC` принципиально важен, он не встроен в язык VHDL, а является частью библиотеки `IEEE.STD_LOGIC_1164`. Из-за этого в каждом файле должны быть операторы подключения библиотеки, что можно было видеть выше в примерах.

Кроме того, в `IEEE.STD_LOGIC_1164` отсутствуют базовые операции типа сложения, сравнения, сдвигов и преобразования в целые из данных типа `STD_LOGIC_VECTOR`. Их, в конце-концов, добавили в стандарте VHDL 2008 в библиотеку `IEEE.NUMERIC_STD_UNSIGNED`.

В VHDL также есть тип `BOOLEAN` с двумя значениями: `true` и `false`. Значения типа `BOOLEAN` возвращаются операциями сравнения (например, сравнения на равенство, `s = '0'`) и используются в условных операторах, как `when` и `if`. Казалось бы, `BOOLEAN true` должно быть эквивалентно `STD_LOGIC '1'`, а `BOOLEAN false` должно значить то же, что и `STD_LOGIC '0'`, но эти типы не были взаимозаменяемы вплоть до VHDL 2008. Например, в старом коде на VHDL приходилось писать

```
y <= d1 when (s = '1') else d0;
```

а в VHDL 2008, где оператор `when` автоматически преобразует `s` из `STD_LOGIC` в `BOOLEAN`, уже можно писать просто

```
y <= d1 when s else d0;
```

Но и в VHDL 2008 всё еще нужно писать

```
q <= '1' when (state = S2) else '0';
```

а не

```
q <= (state = S2);
```

потому что (`state = S2`) возвращает результат типа `BOOELAN`, который не может быть присвоен сигналу типа `STD_LOGIC`.

Хотя мы не объявляем никаких сигналов типа `BOOLEAN`, они автоматически выводятся из сравнений и используются в условных операторах. Аналогично, в VHDL есть тип `INTEGER` для представления целых чисел со знаком. Сигналы типа `INTEGER` могут принимать значения как минимум от $-(2^{31}-1)$ до $2^{31}-1$. В качестве индексов массивов нужно использовать целые числа. Например, в операторе

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2 и 3 – целые, служащие индексами для выбора битов сигнала a. Для индексации нельзя использовать сигнал типа `STD_LOGIC` или `STD_LOGIC_VECTOR`, поэтому нужно преобразовать его в `INTEGER`, как показано ниже в примере восьмивходового мультиплексора, выбирающего один бит из вектора с помощью трехбитного индекса. Функция `TO_INTEGER`, определенная в библиотеке `IEEE.NUMERIC_STD_UNSIGNED`, преобразует из `STD_LOGIC_VECTOR` в неотрицательные значения `INTEGER`.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mux8 is
  port(d: inSTD_LOGIC_VECTOR(7 downto 0);
       s: inSTD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;
architecture synth of mux8 is
begin
  y <= d(TO_INTEGER(s));
end;
```

VHDL также строг в отношении портов типа `out`: их можно использовать исключительно в качестве выходов. Например, следующий пример двух- и трехвходового вентиля И некорректен, так как `v` – выход, но используется также для вычисления `w`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v, w: out STD_LOGIC);
end;
architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

Для решения этой проблемы в VHDL есть отдельный тип порта: `buffer`. Сигнал, подключенный к такому порту, ведет себя как выход, но также может быть использован внутри модуля. Вот исправленный текст объявления интерфейса:

```
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v: buffer   STD_LOGIC;
       w: out      STD_LOGIC);
 end;
```

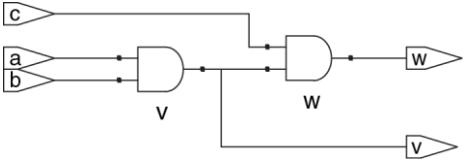


Рис. 4.28 Синтезированная схема модуля `and23`

В Verilog и SystemVerilog этого ограничения никогда не было, поэтому в них не нужны буферные порты. В VHDL 2008 это ограничение тоже снято, но на момент написания это нововведение не поддерживалось программой Synplify.

В результате многих операций, таких как сложение, вычитание или операции булевой логики, получается одно и то же битовое представление результата, будь он со знаком или без знака. В отличие от них, сравнения на больше-меньше, умножение и арифметические сдвиги вправо выполняются для чисел в дополнительном коде со знаком и двоичных чисел без знака по-разному. Эти операции рассматриваются в **главе 5**. В коде **примера 4.33** показано, как обозначаются сигналы, представляющие числа со знаком.

Пример 4.33 БЕЗЗНАКОВЫЙ УМНОЖИТЕЛЬ (a) И УМНОЖИТЕЛЬ СО ЗНАКОМ (b)

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y = a *b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a *b;
endmodule
```

В SystemVerilog сигналы понимаются как беззнаковые по умолчанию. Добавление модификатора `signed` (например, `logic signed [3:0] a`), приводит к тому, что сигнал рассматривается как число со знаком.

VHDL

```
-- 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

В VHDL для выполнения арифметических операций и операций сравнения над `STD_LOGIC_VECTOR` используется библиотека `NUMERIC_STD_UNSIGNED`. При этом векторы считаются беззнаковыми.

```
Use IEEE.NUMERIC_STD_UNSIGNED.all
```

В VHDL также определены типы данных `UNSIGNED` и `SIGNED`, (в библиотеке `IEEE.NUMERIC_STD`), но их рассмотрение выходит за рамки этой главы.

4.8 ПАРАМЕТРИЗОВАННЫЕ МОДУЛИ*

До сих пор у модулей в наших примерах входы и выходы были фиксированной ширины. Например, нам понадобилось определить два разных модуля для двухвходового мультиплексора с четырехбитными и восьмибитными входами, но в языках описания аппаратуры HDL можно описывать и параметризованные модули с портами переменной ширины.

В коде **примера 4.34** объявляется параметризованный двухвходовой мультиплексор с шириной входов, равной по умолчанию восьми битам, который затем используется для создания четырехвходовых мультиплексоров с восьмибитными и двенадцатибитными входами.

Пример 4.34 ПАРАМЕТРИЗОВАННЫЕ *N*-БИТНЫЕ ДВУХВХОДОВЫЕ МУЛЬТИПЛЕКСОРЫ

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input logic [width-1:0] d0, d1,
    input logic             s,
    output logic [width-1:0]y);
   assign y = s ? d1 : d0;
endmodule
```

В SystemVerilog возможна конструкция `#(parameter …)` перед списком входов и выходов для определения параметров модуля. В примере выше оператор `parameter` состоит из параметра по имени `width` со значением по умолчанию, равным 8. Число бит входов и выходов может зависеть от параметра.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);
  logic [7:0] low, hi;
  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

8-битный четырехвходовой мультиплексор состоит из трех экземпляров двухвходового мультиплексора с шириной входов, установленной по умолчанию. В отличие от него, в 12-битном четырехвходовом мультиплексоре `mux4_12` понадобится переопределить ширину входов с помощью конструкции `#( )` перед именем экземпляра (instance):

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]s,
               output logic [11:0] y);
  logic [11:0] low, hi;
  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Не путайте использование знака # для обозначения задержек с использованием #(...) при объявлении и переопределении параметров.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in STD_LOGIC_VECTOR(width-1 downto 0);
    s: in STD_LOGIC;
    y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

Оператор `generic` состоит из указания значения 8 по умолчанию для `width` типа целое.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(1 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture struct of mux4_8 is
```

```
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux: mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

8-битный четырехвходовой мультиплексор, `mux4_8`, включает три мультиплексора 2:1 с шириной по умолчанию.
В отличие от него, в 12-битном четырехвходовом мультиплексоре `mux4_12` понадобится переопределить ширину по умолчанию с помощью `generic map`:

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

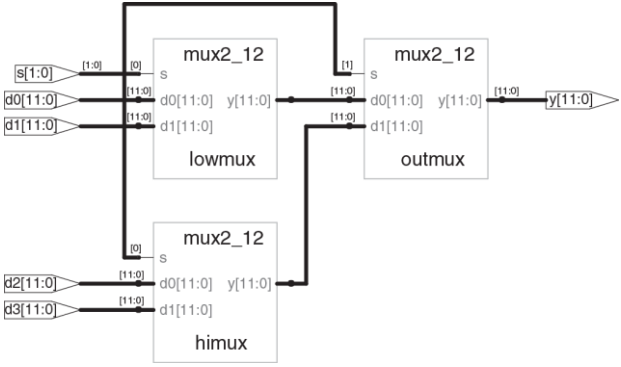


Рис. 4.29 Синтезированная схема модуля `mux4_12`

Пример 4.35 ПАРАМЕТРИЗОВАННЫЙ ДЕШИФРАТОР N:2^N

SystemVerilog

```
module decoder

  #(parameter N = 3)
  (input logic [N-1:0]     a,
   output logic [2**N-1:0] y);
```

```
  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

2**N означает 2^N

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;
entity decoder is
  generic(N: integer := 3);
  port(a: in STD_LOGIC_VECTOR(N-1 downto 0);
       y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;
architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

2**N означает 2^N

В коде **примера 4.35** показан дешифратор, который является еще более удачным примером параметризованного модуля. Широкий дешифратор $N{:}2^N$ довольно утомительно описывать с помощью оператора `case`, но это легко сделать с помощью параметризованного модуля, который просто устанавливает нужный бит в 1. Иначе говоря, в дешифраторе использовано блокирующее присваивание для установки всех битов в 0, а затем нужный бит изменяется в 1.

В языках описания аппаратуры также предусмотрен оператор `generate` для получения разного количества аппаратуры в зависимости от значения параметра. В операторе `generate` допускаются циклы `for` и операторы `if` для определения количества и свойства желаемой аппаратуры. В коде **примера 4.36** демонстрируется, как использовать операторы `generate` для получения *N*-входовой функции И из каскада двухвходовых вентилей И. Конечно, для этой конкретной цели лучше подошла бы операция редукции, но этот пример иллюстрирует общий принцип использования оператора `generate`.

Используйте операторы `generate` с осторожностью – из-за них легко можно непреднамеренно получить очень большую схему!

Пример 4.36 ПАРАМЕТРИЗОВАННЫЙ N-ВХОДОВОЙ ВЕНТИЛЬ И

SystemVerilog

```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);
  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0] = a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i] = a[i] & x[i-1];
    end
  endgenerate

  assign y = x[width-1];
endmodule
```

Оператор `for` проходит по `i`=1, 2, ..., `width-1` для получения множества последовательных вентилей И. После `begin` в цикле `for` внутри `generate` должно быть двоеточие и произвольная метка (в данном случае `forloop`). *(Обратите также внимание на объявление переменной цикла* `i` *как* `genvar`*. – прим. переводчика.)*

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
   signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
   x(0) <= a(0);
   gen: for i in 1 to width-1 generate
     x(i) <= a(i) and x(i-1);
   end generate;
   y <= x(width-1);
end;
```

Переменную цикла `generate` объявлять не нужно.

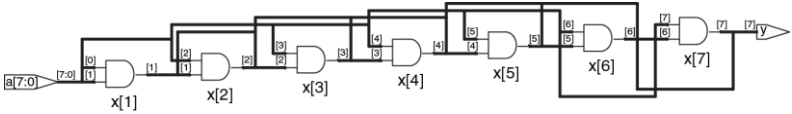


Рис. 4.30 **Синтезированная схема модуля `andN`**

4.9 СРЕДА ТЕСТИРОВАНИЯ

Среда тестирования – это модуль на HDL, который используется для тестирования другого модуля, называемого тестируемое устройство (device under test, DUT). (Некоторые программы разработки называют тестируемый модуль unit under test, UUT.) Среда тестирования содержит операторы для генерации значений, подаваемых на входы DUT и, в идеале, также и для проверки, что на выходе получаются правильные значения. Наборы входных и желаемых выходных значений называются тестовыми векторами.

Попробуем протестировать модуль `sillyfunction` из **раздела 4.1.1**, вычисляющий $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. Это простой модуль, поэтому можно проделать исчерпывающее тестирование, подавая на входы все восемь возможных тестовых векторов.

В **примере 4.37** показана простая среда тестирования. Она включает в себя тестируемый блок DUT, затем подает значения векторов на его входы. Блокирующие присваивания и задержки нужны для приложения значений в желаемом порядке. Пользователь должен просмотреть результаты симуляции и проверить правильность результатов. Среды тестирования симулируются так же, как и другие модули, но не являются синтезируемыми.

Пример 4.37 СРЕДА ТЕСТИРОВАНИЯ

SystemVerilog

```
module testbench1();
  logic a, b, c, y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

Оператор `initial` выполняет содержащиеся в нем операторы при начале симуляции. В данном случае он подает на входы набор 000 и ждет 10 единиц времени. Затем он подает 001 и ждет еще 10 единиц времени, и так далее, пока не будут поданы все восемь возможных наборов. Операторы `initial` должны использоваться только в средах тестирования для симуляции, а не в модулях, из которых будет синтезирована аппаратура. В аппаратуре нет способа магическим образом исполнить при включении последовательность шагов.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench1 is -- no inputs or outputs
end;
architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);
  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

Оператор `process` подает на входы набор 000 и ждет 10 нс. Затем он подает 001 и ждет еще 10 нс, и так далее, пока не будут поданы все восемь возможных наборов.
Наконец, процесс входит в вечное ожидание, иначе его выполнение началось бы заново, и он стал бы подавать тестовые векторы повторно.

Проверять правильность выходов вручную утомительно и чревато ошибками, да и тестировать в уме относительно легко, когда схема свежа в памяти. Однако если придется внести в нее поправки через несколько недель, то определять впоследствии, какое значение нужно считать правильным, будет гораздо труднее. Гораздо лучше написать среду тестирования с самопроверкой, показанную в коде **примера 4.38**.

Пример 4.38 СРЕДА ТЕСТИРОВАНИЯ С САМОПРОВЕРКОЙ

SystemVerilog

```
module testbench2();
  logic a, b, c, y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1; #10;
    assert (y === 0) else $error("001 failed.");
```

```
    b = 1; c = 0; #10;
    assert (y === 0) else $error("010 failed.");
    c = 1; #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1; #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0; #10;
    assert (y === 0) else $error("110 failed.");
    c = 1; #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

Оператор `assert` в SystemVerilog проверяет, истинно ли указанное условие. Если нет, то выполняется оператор `else`. Системная процедура `$error` в операторе `else` печатает сообщение об ошибке с указанием нарушенного условия. Операторы `assert` игнорируются при синтезе.

В SystemVerilog сравнение с помощью `==` и `!=` работает для сигналов, которые не принимают значения x и z. Среда тестирования использует операторы `===` и `!==` для сравнений на равенство и неравенство соответственно, потому что эти операторы работают также и с операндами, значения которых могут быть `x` или `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench2 is -- no inputs or outputs
end;
architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);
  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                      wait for 10 ns;
      assert y = '1' report "101 failed.";
```

```
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

Оператор `assert` проверяет условие и печатает сообщение, указанное после `report`, если условие не выполнено. Оператор имеет смысл только при симуляции, не при синтезе.

Писать код для каждого тестового вектора тоже становится утомительно, особенно для модулей, требующих большого количества тестовых векторов. Еще лучше держать тестовые вектора в отдельном файле. Тогда среда тестирования будет просто читать их из файла, подавать входной вектор на входы DUT, проверять, что значения выходов совпадают с выходным вектором, и повторять, пока не будет достигнут конец файла.

В **примере 4.39** показана такая среда тестирования. Она генерирует тактовый сигнал с помощью оператора `always/process` без списка чувствительности, поэтому этот оператор выполняется как бесконечный цикл. В начале симуляции среда читает тестовые векторы из текстового файла и выставляет `reset` в течение двух тактов. Хотя тактовый сигнал и сброс не нужны для тестирования комбинационной логики, они упомянуты, потому что они будут важны для тестирования последовательных устройств.

Вот `example.tv` – файл, в котором находятся входы и ожидаемый выход в двоичном виде:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Пример 4.39 СРЕДА ТЕСТИРОВАНИЯ С ФАЙЛОМ ТЕСТОВЫХ ВЕКТОРОВ

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // generate clock
  always
   begin
     clk = 1; #5; clk = 0; #5;
   end
  // at start of test, load vectors
  // and pulse reset
```

```
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end
  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end
  // check results on falling edge of clk
  always @(negedge clk)
   if (~reset) begin // skip during reset
     if (y !== yexpected) begin // check result
        $display("Error: inputs = %b", {a, b, c});
        $display(" outputs = %b (%b expected)", y, yexpected);
        errors = errors + 1;
     end
     vectornum = vectornum + 1;
     if (testvectors[vectornum] === 4'bx) begin
       $display("%d tests completed with %d errors", vectornum, errors);
       $finish;
     end
    end
endmodule
```

`$readmemb` читает файл с двоичными числами в массив `testvectors`. `$readmemh` – аналогична, но читает файл с шестнадцатеричными числами.

Следующий блок кода ждет одну единицу времени после переднего фронта тактового сигнала (во избежание путаницы, если тактовый сигнал и данные меняются одновременно), затем устанавливает три входа (`a`, `b` и `c`) и ожидаемый выход (`yexpected`) согласно четырем битам в текущем тестовом векторе.
Среда сравнивает полученный выход, `y`, с ожидаемым выходом, `yexpected`, и печатает сообщение об ошибке, если они не совпадают. `%b` и `%d` означают печать значений в двоичном и десятичном виде, соответственно. `$display` – это системная процедура печати в окно симулятора. Например, `$display("%b %b", y, yexpected);` печатает два значения, `y` и `yexpected`, в двоичном виде. `%h` печатает в шестнадцатеричном виде.
Этот процесс повторяется, пока в массиве `testvectors` не закончатся прочитанные из файла тестовые вектора. `$finish` завершает симуляцию.
Обратите внимание, что хотя модуль на SystemVerilog предусматривает вплоть до 10001 тестовых векторов, симуляция завершится после подачи восьми векторов из файла.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;
entity testbench3 is -- no inputs or outputs
end;
architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
```

```
  signal a, b, c, y: STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);
  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;
  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;
  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop
      -- change vectors on rising edge
      wait until rising_edge(clk);
```

```
      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
      read(L, vector_out);
      (a, b, c) <= vector_in(2 downto 0) after 1 ns;
      y_expected <= vector_out after 1 ns;
      -- -- check results on falling edge
      wait until falling_edge(clk);
      if y /= y_expected then
         report "Error: y = " & std_logic'image(y);
         errors := errors + 1;
       end if;
       vectornum := vectornum + 1;
     end loop;
     -- summarize results at end of simulation
     if (errors = 0) then
      report "NO ERRORS -- " &
             integer'image(vectornum) &
             " tests completed successfully."
             severity failure;
    else
      report integer'image(vectornum) &
             " tests completed, errors = " &
             integer'image(errors)
             severity failure;
    end if;
  end process;
end;
```

Код на VHDL использует команды чтения из файла, рассмотрение которых не входит в эту главу, но дает понять, как выглядит среда тестирования с самопроверкой на VHDL.

Новые значения входов подаются по переднему фронту тактового сигнала, а выход проверяется по заднему фронту. Сообщения об ошибках выдаются в момент возникновения. В конце симуляции среда тестирования печатает итог: общее количество тестовых векторов и количество обнаруженных ошибок.

Среда в HDL **примере 4.39** избыточна для такой простой схемы. Однако ее легко изменить для тестирования более сложных схем, заменив файл `example.tv`, включив в среду другое тестируемое устройство и изменив несколько строк кода для установки входов и проверки выходов.

4.10 РЕЗЮМЕ

Языки описания аппаратуры (HDL) – очень важные инструменты разработчиков современной цифровой электроники. Изучив SystemVerilog или VHDL, вы сможете разрабатывать цифровые системы гораздо быстрее, чем при традиционном черчении принципиальных схем. Цикл отладки тоже обычно гораздо короче, так как изменения заключаются в редактировании текста, а не утомительном переподключении проводов на схеме. Однако с использованием HDL цикл отладки может быть и гораздо дольше, если вы плохо представляете себе, какую аппаратуру описывает ваш код.

Языки описания аппаратуры используются и для симуляции, и для синтеза. Логическая симуляция – мощный способ протестировать систему на компьютере, перед тем как она превратится в аппаратуру. Симуляторы позволяют вам проверить те значения сигналов в системе, которые могут быть недоступны для измерения на реальной электрической схеме. Логический синтез превращает код на HDL в цифровые логические схемы.

Самое важное, что вам нужно помнить при написании кода на HDL – это то, что вы описываете настоящую аппаратуру, а не пишете программу для компьютера. Начинающие часто совершают ошибку, создавая код на HDL, не продумав, какую именно аппаратуру они хотят получить.

Если вы не знаете, какая аппаратура выйдет из кода, вы, скорее всего, не достигнете нужного результата. Поэтому начинайте с эскиза блочной диаграммы системы, определяя, какие ее части – комбинационная логика, какие – последовательные схемы или конечные автоматы, и т. д. Затем пишите код для каждой части на HDL, используя правильные конструкции для нужного типа аппаратуры.

УПРАЖНЕНИЯ

Упражнения в этом разделе можно выполнять на языке, который вам больше нравится. Если у вас есть симулятор, протестируйте, что вы написали. Выведите значения сигналов и объясните, как они доказывают, что схема работает правильно. Если у вас есть синтезатор, синтезируйте схему. Напечатайте полученную принципиальную схему и объясните, почему она удовлетворяет ожиданиям.

Упражнение 4.1 Нарисуйте диаграмму схемы, описанной программой ниже. Упростите схему, добившись минимума вентилей.

SystemVerilog

```
module exercise1(input logic a, b, c,
                 output logic y, z);
  assign y = a & b & c |a & b & ~c | a & ~b & c;
  assign z = a & b |~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
```

```
end;
architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or
       (a and not b and c);
  z <= (a and b) or (not a and not b);
```

Упражнение 4.2 Нарисуйте диаграмму схемы, описанной программой ниже. Упростите схему, добившись минимума вентилей.

SystemVerilog

```
module exercise2(input  logic[3:0] a,
                 output logic [1:0] y);
  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
```

```
end;
architecture synth of exercise2 is
begin
  process(all) begin
    if    a(0) then y <= "11";
    elsif a(1) then y <= "10";
    elsif a(2) then y <= "01";
    elsif a(3) then y <= "00";
    else            y <= a(1 downto 0);
    end if;
end process;
```

Упражнение 4.3 Напишите модуль на HDL, вычисляющий четырехвходовую функцию XOR (исключающее ИЛИ). Вход обозначьте $a_{3:0}$, выход – `y`.

Упражнение 4.4 Напишите среду тестирования с самопроверкой для **упражнения 4.3**. Создайте файл, содержащий все 16 вариантов входов. Просимулируйте схему и убедитесь, что она работает. Внесите ошибку в файл с тестовыми векторами и убедитесь, что среда тестирования сообщает о несовпадении результатов.

Упражнение 4.5 Напишите на HDL модуль `minority` с тремя входами, `a`, `b`, и `c`, и одним выходом, `y`, принимающим значение TRUE, если не менее двух входов равны FALSE.

Упражнение 4.6 Напишите на HDL модуль для управления семисегментным индикатором шестнадцатеричных цифр. Должны поддерживаться не только цифры 0-9, но и A, B, C, D, E и F.

Упражнение 4.7 Напишите среду тестирования с самопроверкой для **упражнения 4.6**. Создайте файл, содержащий все 16 вариантов входов. Просимулируйте схему и убедитесь, что она работает. Внесите ошибку в файл с тестовыми векторами и убедитесь, что среда тестирования сообщает о несовпадении результатов.

Упражнение 4.8 Напишите восьмивходовой мультиплексор с именем `mux8`, входами $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, и выходом `y`.

Упражнение 4.9 Напишите структурный модуль для вычисления логической функции `y=ab~ + b~c~ + a~bc` с помощью мультиплексорной логики. Используйте мультиплексор из **упражнения 4.8**.

Упражнение 4.10 Повторите **упражнение 4.9** с помощью четырехвходового мультиплексора и любого количества вентилей НЕ.

Упражнение 4.11 В **разделе 4.5.4** было замечено, что синхронизатор можно описать с помощью блокирующих присваиваний в правильном порядке. Придумайте простую последовательную схему, которую нельзя правильно описать с помощью блокирующих присваиваний, независимо от их порядка.

Упражнение 4.12 Напишите модуль на HDL для схемы приоритетов с восемью входами.

Упражнение 4.13 Напишите модуль на HDL для дешифратора 2:4.

Упражнение 4.14 Напишите модуль на HDL для дешифратора 6:64 с помощью трех экземпляров дешифратора 2:4 из **упражнения 4.13** и нескольких трехвходовых вентилей И.

Упражнение 4.15 Напишите модуль на HDL, реализующий логические выражения из **упражнения 2.13**.

Упражнение 4.16 Напишите модуль на HDL, реализующий схему из **упражнения 2.26**.

Упражнение 4.17 Напишите модуль на HDL, реализующий схему из **упражнения 2.27**.

Упражнение 4.18 Напишите модуль на HDL, реализующий логическую функцию из **упражнения 2.28**. Обратите особое внимание на то, как обходиться с незначащими битами.

Упражнение 4.19 Напишите модуль на HDL, реализующий функции из **упражнения 2.35**.

Упражнение 4.20 Напишите модуль на HDL, реализующий кодер с приоритетами из **упражнения 2.36**.

Упражнение 4.21 Напишите модуль на HDL, реализующий модифицированный кодер с приоритетами из **упражнения 2.37**.

Упражнение 4.22 Напишите модуль на HDL, реализующий преобразователь из бинарного в унарный код из **упражнения 2.38**.

Упражнение 4.23 Напишите модуль на HDL, реализующий функцию дней в месяце из **вопроса 2.2**.

Упражнение 4.24 Нарисуйте диаграмму состояний конечного автомата, описанного нижеследующим кодом на HDL:

SystemVerilog

```
module fsm2(input  logicclk, reset,
            input  logica, b,
            output logicy);
  logic [1:0] state, nextstate;
  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;
    always_comb
      case (state)
       S0: if (a ^ b) nextstate = S1;
           else       nextstate = SO;
       S1: if (a & b) nextstate = S2;
           else       nextstate = SO;
       S2: if (a | b) nextstate = S3;
           else       nextstate = SO;
       S3: if (a | b) nextstate = S3;
           else      nextstate = SO;
    endcase
  assign y = (state = = S1) |(state = = S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;
architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;
  process(all) begin
    case state is
      when S0 =>if (a xor b) then
                    nextstate <= S1;
                else nextstate <= S0;
                end if;
      when S1 =>if (a and b) then
                    nextstate <= S2;
                else nextstate <= S0;
                end if;
      when S2 =>if (a or b) then
                    nextstate <= S3;
```

```
                else nextstate <= S0;
                end if;
      when S3 =>if (a or b) then
                    nextstate <= S3;
                else nextstate <= S0;
                end if;
   end case;
  end process;
  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

Упражнение 4.25 Нарисуйте диаграмму состояний конечного автомата, описанного нижеследующим кодом на HDL. Автоматы подобного типа используются для предсказания переходов в некоторых микропроцессорах.

SystemVerilog

```
module fsm1(input logic clk, reset,
            input logic taken, back,
            output logic predicttaken);
  logic [4:0] state, nextstate;
  parameter S0 = 5'b00001;
  parameter SI = 5'b00010;
  parameter S2 = 5'b00100;
  parameter S3 = 5'b01000;
  parameter S4 = 5'b10000;
```

```
  always_ff @(posedge clk, posedge reset)
   if (reset) state <= S2;
   else       state <= nextstate;
  always_comb
    case (state)
      S0: if (taken) nextstate = S1;
          else       nextstate = S0;
      S1: if (taken) nextstate = S2;
          else       nextstate = S0;
      S2: if (taken) nextstate = S3;
          else       nextstate = S1;
      S3: if (taken) nextstate = S4;
          else       nextstate = S2;
      S4: if (taken) nextstate = S4;
          else       nextstate = S3;
      default:       nextstate = S2;
    endcase
  assign predicttaken = (state = = S4) |
                        (state = = S3) |
                        (state = = S2 && back);
Endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164. all;
entity fsm1 is
  port(clk, reset:    in  STD_LOGIC;
       taken, back:   in  STD_LOGIC;
       predicttaken:  out STD_LOGIC);
```

```
end;
architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;
process(all) begin
   case state is
    when S0 => if taken then
                    nextstate <= S1;
               else nextstate <= S0;
               end if;
    when S1 => if taken then
                    nextstate => S2;
               else nextstate <= S0;
               end if;
    when S2 => if taken then
                    nextstate <= S3;
               else nextstate <= S1;
               end if;
    when S3 => if taken then
                    nextstate <= S4;
               else nextstate <= S2;
               end if;
    when S4 => if taken then
```

```
                    nextstate <= S4;
                else nextstate <= S3;
                end if;
    when others =>nextstate <= S2;
  end case;
end process;
-- output logic
predicttaken <= '1' when
                ((state = S4) or (state = S3) or
                (state = S2 and back = '1'))
  else '0';
end;
```

Упражнение 4.26 Напишите модуль на HDL для SR-защелки.

Упражнение 4.27 Напишите модуль на HDL для JK-триггера со входами clk, J и K, и выходом Q. По переднему фронту тактового сигнала Q сохраняет предыдущее состояние, если J = K = 0, становится равным 1, если J = 1, сбрасывается в 0, если K = 1, и инвертируется, если J = K = 1.

Упражнение 4.28 Напишите модуль на HDL для защелки на рис. 3.18. Используйте один оператор присваивания для каждого вентиля. Задайте задержку 1 (или 1 нс) для каждого вентиля. Просимулируйте защелку и убедитесь, что она работает правильно. Затем увеличьте задержку у инвертора.

Насколько большой может быть задержка, прежде чем защелка сломается из-за гонки сигналов?

Упражнение 4.29 Напишите модуль на HDL для контроллера светофора из раздела 4.3.1.

Упражнение 4.30 Напишите три модуля на HDL для факторизованного контроллера светофора с режимом парада из **примера 3.8**. Назовите эти модули controller, mode и lights, и назовите их входы-выходы как на **Рис. 3.33 (b)**.

Упражнение 4.31 Напишите модуль на HDL, описывающий схему на **Рис. 3.42**.

Упражнение 4.32 Напишите модуль на HDL для конечного автомата с диаграммой состояний, изображенной на **Рис. 3.69** из **упражнения 3.22**.

Упражнение 4.33 Напишите модуль на HDL для конечного автомата с диаграммой состояний, изображенной на **Рис. 3.70** из **упражнения 3.23**.

Упражнение 4.34 Напишите модуль на HDL для улучшенного контроллера светофора из **упражнения 3.24**.

Упражнение 4.35 Напишите модуль на HDL для (непонятно!) из **упражнения 3.25**.

Упражнение 4.36 Напишите модуль на HDL для дозатора напитков из **упражнения 3.26**.

Упражнение 4.37 Напишите модуль на HDL для счетчика в коде Грея из **упражнения 3.27**.

Упражнение 4.38 Напишите модуль на HDL для счетчика в коде Грея ВВЕРХ/ВНИЗ из **упражнения 3.28**.

Упражнение 4.39 Напишите модуль на HDL для конечного автомата из **упражнения 3.29**.

Упражнение 4.40 Напишите модуль на HDL для конечного автомата из **упражнения 3.30**.

Упражнение 4.41 Напишите модуль на HDL для последовательного вычисления противоположного значения из **вопроса 3.2**.

Упражнение 4.42 Напишите модуль на HDL для схемы из **упражнения 3.31**.

Упражнение 4.43 Напишите модуль на HDL для схемы из **упражнения 3.32**.

Упражнение 4.44 Напишите модуль на HDL для схемы из **упражнения 3.33**.

Упражнение 4.45 Напишите модуль на HDL для схемы из **упражнения 3.34**, при желании с использованием полного сумматора из **раздела 4.2.5**.

Упражнения по SystemVerilog

Упражнение 4.46 Что значит, когда в SystemVerilog сигнал объявлен как `tri`?

Упражнение 4.47 Перепишите модуль `syncbad` из **примера 4.29**. Используйте неблокирующие присваивания, но измените код так, чтобы получился правильный синхронизатор с двумя триггерами.

Упражнение 4.48 Рассмотрите следующие два модуля на SystemVerilog. Функционально одинаковы ли они? Нарисуйте аппаратуру, которую означает каждый из них.

```
module code1(input  logic clk, a, b, c,
             output logic y);
  logic x;
  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2 (input logic a, b, c, clk,
              output logic y);
  logic x;
```

```
    always_ff @(posedge clk) begin
      y <= x | c;
      x <= a & b;
    end
  endmodule
```

Упражнение 4.49 Повторите **упражнение 4.48**, если в каждом присваивании <= заменено на =.

Упражнение 4.50 В нижеследущих модулях на SystemVerilog показаны типичные ошибки, замеченные авторами у студентов при выполнении лабораторных работ. Объясните ошибку в каждом модуле и укажите, как ее исправить.

```
(a) module latch(input logic      clk,
                 input logic[3:0] d,
                 output reg [3:0] q);
      always @(clk)
        if (clk) q <= d;
    endmodule

(b) module gates(input  logic [3:0] a, b,
                 output logic [3:0] y1, y2, y3, y4, y5);
      always @(a)
        begin
          y1 = a & b;
          y2 = a |b;
          y3 = a ^ b;
          y4 = ~(a & b);
```

```
            y5 = ~(a |b);
          end
      endmodule

(c) module mux2(input  logic [3:0] d0, d1,
                input  logic   s,
                output logic [3:0] y);
        always @(posedge s)
          if (s) y <= d1;
          else y <= d0;
      endmodule

(d) module twoflops(input  logic clk,
                    input  logic d0, d1,
                    output logic q0, q1);
        always @(posedge clk)
          q1 = d1;
          q0 = d0;
      endmodule

(e) module FSM(input  logic clk,
               input  logic a,
               output logic out1, out2);
        logic state;
        // next state logic and register (sequential)
        always_ff @(posedge clk)
          if (state == 0) begin
              if (a) state <= 1;
          end else begin
              if (~a) state <= 0;
```

```
          end
        always_comb // output logic (combinational)
          if (state == 0) out1 = 1;
          else            out2 = 1;
      endmodule

(f) module priority(input  logic [3:0] a,
                    output logic [3:0] y);
        always_comb
          if      (a[3]) y = 4'b1000;
          else if (a[2]) y = 4'b0100;
          else if (a[1]) y = 4'b0010;
          else if (a[0]) y = 4'b0001;
      endmodule

(g) module divideby3FSM(input  logicclk,
                        input  logicreset,
                        output logicout);
        logic [1:0] state, nextstate;
        parameter S0 = 2'b00;
        parameter S1 = 2'b01;
        parameter S2 = 2'b10;
        // State Register
        always_ff @(posedge clk, posedge reset)
          if (reset) state <= S0;
          else       state <= nextstate;
        // Next State Logic
        always @(state)
          case (state)
            S0: nextstate = S1;
```

```
            S1: nextstate = S2;
            S2: nextstate = S0;
          endcase
        // Output Logic
        assign out = (state == S2);
      endmodule

(h) module mux2tri(input  logic [3:0] d0, d1,
                   input  logic       s,
                   output tri [3:0] y);
        tristate t0(d0, s, y);
        tristate t1(d1, s, y);
      endmodule

(i) module floprsen(input  logic       clk,
                    input  logic       reset,
                    input  logic       set,
                    input  logic [3:0] d,
                    output logic [3:0] q);
        always_ff @(posedge clk, posedge reset)
          if (reset) q <= 0;
          else       q <= d;
        always @(set)
          if (set)  q <= 1;
      endmodule

(j) module and3(input  logic a, b, c,
                output logic y);
        logic tmp;
        always @(a, b, c)
```

```
        begin
          tmp <= a & b;
          y <= tmp & c;
        end
      endmodule
```

Упражнение 4.51 Зачем в VHDL надо писать

```
q <= '1' when state = S0 else '0';
```

а не просто

```
q <= (state = S0);
```

Упражнение 4.52 В каждом из нижеследующих модулей на VHDL есть ошибка. Для краткости показаны лишь описания архитектуры; считайте, что объявление библиотеки и объявление интерфейса правильны.

Объясните ошибку и укажите, как ее исправить.

```
(a) architecture synth of latch is
    begin
      process(clk) begin
        if clk = '1' then q <= d;
         end if;
       end process;
     end;
```

```
(b) architecture proc of gates is
      begin
        process(a) begin
          Y1 <= a and b;
          y2 <= a or b;
          y3 <= a xor b;
          y4 <= a nand b;
          y5 <= a nor b;
        end process;
      end;

(c) architecture synth of flop is
      begin
        process(clk)
          if rising_edge(clk) then
            q <= d;
      end;

(d) architecture synth of priority is
      begin
        process(all) begin
          if  a(3) then y <= "1000";
          elsif a(2) then y <= "0100";
          elsif a(1) then y <= "0010";
          elsif a(0) then y <= "0001";
          end if;
        end process;
      end;

(e) architecture synth of divideby3FSM is
```

```
        type statetype is (S0, S1, S2);
        signal state, nextstate: statetype;
      begin
        process(clk, reset) begin
          if reset then state <= S0;
          elsif rising_edge(clk) then
            state <= nextstate;
          end if;
        end process;
        process(state) begin
          case state is
            when S0 =>nextstate <= S1;
            when S1 =>nextstate <= S2;
            when S2 =>nextstate <= S0;
          end case;
        end process;
        q <= '1' when state = S0 else '0';
      end;

(f) architecture struct of mux2 is
        component tristate
          port(a: in   STD_LOGIC_VECTOR(3 downto 0);
               en: in  STD_LOGIC;
               y:  out STD_LOGIC_VECTOR(3 downto 0));
        end component;
      begin
        t0: tristate port map(d0, s, y);
        t1: tristate port map(d1, s, y);
      end;
```

```
(g) architecture asynchronous of floprs is
      begin
        process(clk, reset) begin
          if reset then
            q <= '0';
          elsif rising_edge(clk) then
            q <= d;
          end if;
        end process;
        process(set) begin
          if set then
            q <= '1';
          end if;
        end process;
      end;
```

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

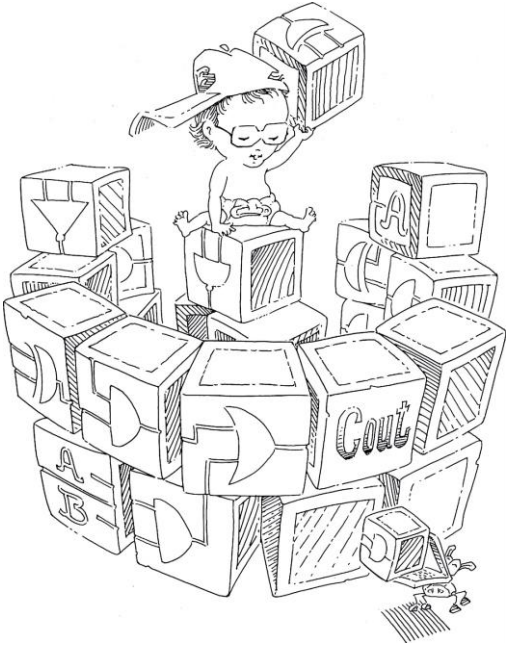
Эти вопросы задавались на собеседованиях по приему на работу, связанную с проектированием цифровых систем.

Вопрос 4.1 Напишите строку на HDL, реализующую управление 32-битной шиной data сигналом sel, получая 32-битный сигнал result. Если sel истинно, result = data, иначе все биты result – нули.

Вопрос 4.2 Объясните разницу между блокирующими и неблокирующими присваиваниями в SystemVerilog. Приведите примеры.

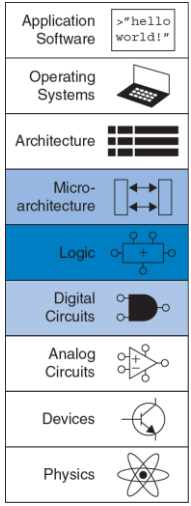
Вопрос 4.3 Что делает этот оператор SystemVerilog:

```
result = | (data[15:0] & 16'hC820);
```


A
Cout
A
B

Цифровые функциональные узлы

5



5.1 ВВЕДЕНИЕ

В предыдущих главах мы познакомились с проектирование комбинационных и последовательностных схем с использованием логических выражений, схем и языков описания аппаратуры. В этой главе мы рассмотрим более сложные комбинационные и последовательностные функциональные узлы, используемые в цифровых системах. Такие узлы включают в себя арифметические схемы, счетчики, схемы сдвига, матрицы памяти и матрицы логических элементов. Эти функциональные узлы полезны не только сами по себе, но и как демонстрация принципов иерархичности, модульности и регулярности. Функциональные узлы иерархически собраны из нескольких простейших компонент, таких как логические вентили, мультиплексоры и декодеры. Каждый функциональный узел имеет четко определенный интерфейс и может рассматриваться как черный ящик, когда не так важна его базовая реализация. Регулярная структура каждого функционального узла может расширяться до любого размера. В **главе 7** подобные функциональные узлы будут использоваться для создания микропроцессора.

5.2 АРИФМЕТИЧЕСКИЕ СХЕМЫ

Арифметические схемы являются основным функциональным узлом любого компьютера. Компьютеры и цифровые схемы выполняют множество арифметических операций: сложение, вычитание, сравнение, сдвиги, умножение и деление. В этой главе будет описана аппаратная реализация всех перечисленных операций.

5.2.1 Сложение

Сложение – одна из самых распространенных операций в цифровых системах. Для начала мы рассмотрим сложение двух одноразрядных двоичных чисел. Затем мы расширим эту процедуру до N-разрядных чисел. Сумматоры демонстрируют компромисс между скоростью и сложностью реализации.

Полусумматор

Вначале спроектируем одноразрядный *полусумматор* (*half adder*). Как показано на **Рис. 5.1**, полусумматор имеет два входа (A и B) и два выхода (S и C_{out}). S – это сумма A и B. Если и A, и B равны 1, то выход S должен стать равным 2, такое число не может быть представлено в виде одного двоичного разряда. В этом случае результат указывается вместе с переносом C_{out} в следующий разряд. Полусумматор может

быть построен из элементов XOR («исключающее ИЛИ») и AND («логическое И»).

В многоразрядном сумматоре выход C_{out} подсоединяется ко входу переноса следующего разряда. Например, на **Рис. 5.2** бит переноса показан синим цветом, он является выходом C_{out} одноразрядного сумматора 1-го разряда и входом C_{in} сумматора следующего разряда. Однако в полусумматоре нет входа переноса C_{in} для связи с выходом C_{out} предыдущего разряда. В *полном сумматоре*, рассматриваемом в следующем разделе, такой вход есть.

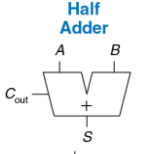
Полный сумматор

Как показано на **Рис. 5.3**, *полный сумматор* (*full adder*), описанный в **разделе 2.1**, имеет вход переноса C_{in},. На рисунке также приведены уравнения для S и C_{out}.

Сумматор с распространяющимся переносом

N-разрядный сумматор складывает 2 N-разрядных числа (A и B), а также входной перенос C_{in}, и формирует N-разрядный результат S и выходной перенос C_{out}. Такой сумматор называется *сумматором с распространяющимся переносом* (*carry propagate adder, CPA*), так как выходной перенос одного разряда переходит в следующий разряд. Условное обозначение такого сумматора показано на **Рис. 5.4**. Оно аналогично обозначению полного сумматора за исключением того, что

входы/выходы *A*, *B*, *S* являются шинами, а не отдельными разрядами. Самыми распространенными реализациями СРА являются: сумматоры с последовательным переносом (ripple-carry adders), с ускоренным переносом (carry-lookahead adders) и префиксные сумматоры (prefix adders).



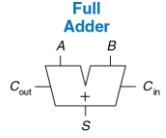
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

Рис. 5.1 **Одноразрядный полусумматор**

```
  1
 0001
+0101
 0110
```

Рис. 5.2 **Бит переноса**



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Рис. 5.3 **Одноразрядный полный сумматор**

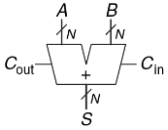


Рис. 5.4 **Сумматор с распространяющимся переносом**

Сумматоры с последовательным переносом

Самый простой способ реализации *N*-разрядного сумматора – это объединение в цепь *N* полных сумматоров. Выход C_{out} некоторого разряда будет поступать на вход C_{in} следующего разряда и т. д. (см. **Рис. 5.5** для 32-разрядного сумматора).

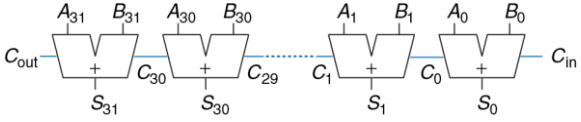


Рис. 5.5 **32-разрядный сумматор с последовательным переносом**

Такая схема называется *сумматором с последовательным переносом* (*ripple-carry adder*). При ее проектировании используется принцип модульности и регулярности: модуль полного сумматора многократно используется для формирования большей системы. Такой сумматор имеет недостаток: его скорость падает при увеличении числа

разрядов N. S_{31} зависит от C_{30}, который зависит от C_{29}, который в свою очередь зависит от C_{28} и т. д. до C_{in} (см. **Рис. 5.5**). Перенос проходит через всю цепь. Задержка такого сумматора (t_{ripple}) увеличивается вместе с количеством разрядов, как показано в **уравнении (5.1)**, где t_{FA} – это задержка полного сумматора.

$$t_{ripple} = Nt_{FA} \quad (5.1)$$

Сумматоры с ускоренным переносом

Главной причиной того, что большие сумматоры с последовательным переносом работают медленно, является то, что сигнал переноса должен пройти через все биты сумматора. Сумматоры с ускоренным переносом (*carry-lookahead* adder, *CLA*) – это другой тип сумматоров с распространяющимся переносом, который решает эту проблему путем разделения сумматора на *блоки* и реализуя схему так, чтобы определить выходной перенос блока как только стал известен его входной перенос. Таким образом, мы смотрим вперед через блоки и не ждем прохождения переноса через все полные сумматоры внутри блока. К примеру, 32-разрядный сумматор может быть разделен на 8 4-разрядных сумматоров.

Сумматоры с ускоренным переносом используют сигналы *генерации* (*G*) и *распространения* (*P*), которые описывают, как блок (или разряд)

определяет выход переноса. *i*-ый разряд сумматора генерирует перенос, если он выдает перенос на своем выходе, независимо от наличия переноса на входе. *i*-ый разряд сумматора генерирует C_i в том случае, если и A_i, и B_i равны 1. Таким образом, сигнал генерации G_i можно вычислить как $G_i = A_iB_i$. Разряд называется *распространяющим*, если выходной сигнал переноса появляется при наличии входного переноса. Разряд будет распространять входной сигнал переноса, C_{i-1} , если либо A_i, либо B_i равны 1. Таким образом, $P_i = A_i + B_i$. Используя эти определения, мы можем переписать логику формирования сигнала переноса для определенного разряда. Разряд i сумматора будет формировать выходной сигнал переноса C_i, если он или генерирует перенос G_i или распространяет входной перенос P_i C_{i-1}. В виде уравнения это можно записать следующим образом:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1} \qquad (5.2)$$

Определения сигналов генерации и распространения относятся и к многоразрядным блокам. Блок называется генерирующим перенос, если он создает выходной перенос независимо от входного сигнала переноса данного блока. Блок называется распространяющим перенос, если выходной перенос возникает при поступлении входного переноса. $G_{i:j}$ и $P_{i:j}$ определяются, как сигналы генерации и распространения для блоков, охватывающих разряды с i до j.

Обычно в электронных схемах сигналы распространяются слева направо. Арифметические схемы нарушают эти правила, так как перенос идет справа налево (от младшего разряда к старшему).

В течение многих лет люди используют множество способов для выполнения арифметических действий. Дети считают на пальцах (и некоторые взрослые, кстати, тоже). Китайцы и вавилоняне изобрели счеты еще в 2400 г. до н.э. Логарифмические линейки, придуманные в 1630 году, использовались вплоть до 1970-ых, затем стали входить в обиход ручные инженерные калькуляторы. Сегодня компьютеры и цифровые калькуляторы используются повсеместно. Что придумают дальше?

Блок генерирует перенос, если самый старший разряд генерирует перенос или если старший разряд распространяет перенос, сгенерированный предыдущим разрядок и т. д. Например, логика блока генерации для блока, охватывающего разряды от 0 до 3, будет следующей:

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)) \quad (5.3)$$

Блок распространяет перенос, если все входящие в него разряды этот перенос распространяют. Логика распространения для блока, охватывающего разряды с 0 до 3:

$$P_{3:0} = P_3 P_2 P_1 P_0 \quad (5.4)$$

При помощи блоковых сигналов генерации и распространения можно быстро определить выходной перенос блока C_i, используя его входной перенос C_j.

$$C_i = G_{i:j} + P_{i:j} C_j \quad (5.5)$$

На **Рис. 5.6 (a)** изображен 32-разрядный сумматор с ускоренным переносом, состоящий из 8 4-разрядных блоков. Каждый блок содержит 4-разрядный сумматор с последовательным переносом и схему ускоренного переноса, которая определяет выходной перенос блока по входному, которая показана на **Рис. 5.6 (b)**. На рисунке не показаны

элементы «И» и «ИЛИ» необходимые для вычисления одноразрядных сигналов генерации и распространения G_i и P_i по A_i и B_i. Сумматор с ускоренным переносом демонстрирует модульность и регулярность.

Все блоки сумматора одновременно вычисляют однобитовые и блоковые сигналы генерации и распространения. Критический путь начинается с вычисления G_0 и $G_{3:0}$ в первом блоке сумматора. Сигнал C_{in} затем распространяется по направлению к C_{out} через логические элементы И/ИЛИ всех блоков. Для большого сумматора это происходит гораздо быстрее, чем распространение переноса через каждый последующий разряд сумматора. И, наконец, критический путь через последний блок содержит небольшой сумматор с последовательным переносом. Таким образом, *N*-разрядный сумматор, разделенный на *k*-разрядные блоки, имеет задержку

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k–1)t_{AND\_OR} + k_{tFA} \quad (5.6)$$

где t_{pg} – задержка отдельных логических элементов генерации/распространения (одиночных логических элементов И/ИЛИ) при генерации *P* и *G*. t_{pg_block} является задержкой формирования сигналов генерации/распространения $P_{i:j}$ и $G_{i:j}$ для *k* -разрядного блока, а t_{AND_OR} является задержкой тракта C_{in} – C_{out}, в который входит логика И/ИЛИ *k*–разрядного CLA-блока. При $N > 16$ такой сумматор работает гораздо

быстрее, чем сумматор с последовательным переносом. Однако задержка сумматора по-прежнему линейно возрастает с ростом *N*.

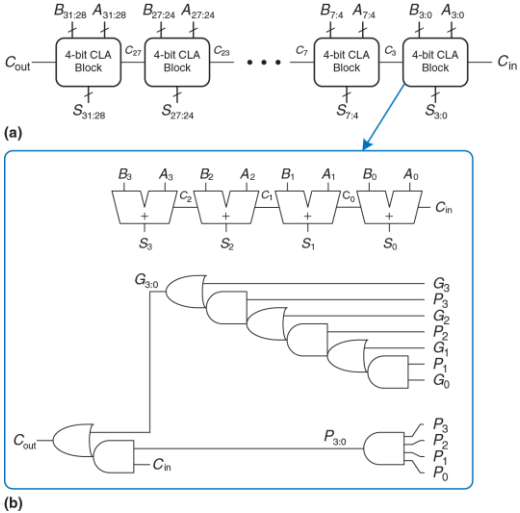


Рис. 5.6 **(a) 32-разрядный сумматор с ускоренным переносом и (b) его 4-х битный блок**

Пример 5.1 ЗАДЕРЖКИ СУММАТОРОВ С ПОСЛЕДОВАТЕЛЬНЫМ И УСКОРЕННЫМ ПЕРЕНОСАМИ

Сравним задержки 32-разрядного сумматора с последовательным переносом и 32-разрядного сумматора с ускоренным переносом, который состоит из 4-разрядных блоков. Предположим, что задержка каждого двухвходового логического элемента составляет 100 пс, а задержка полного сумматора – 300 пс.

Решение: В соответствии с **формулой (5.1)**, задержка распространения 32-разрядного сумматора с последовательным переносом равна 32x300 пс = 9.6 нс.

У сумматора с ускоренным переносом t_{pg} = 100 пс, t_{pg_block} = 6 x 100 пс = 600 пс, и t_{AND_OR} = 2 x100 пс = 200 пс. В соответствии с **уравнением (5.6)** задержка распространения 32-х разрядного сумматора с ускоренным переносом, состоящего из 4-х разрядных блоков, равна 100 пс + 600пс + (32/4-1) x 200 пс + (4 x 300) пс = 3.3 нс, что почти в три раза меньше, чем у сумматора с последовательным переносом.

Префиксный сумматор

Префиксный сумматор развивает логику генерации и распространения сумматора с ускоренным переносом для еще более быстрого выполнения операции сложения. Сначала он вычисляет *G* и *P* для пар разрядов, далее для блоков из 4-х разрядов, затем для блоков из 8-ми, 16-ти и т. д. разрядов, пока сигнал генерации не будет известен для каждого разряда. Сумма определяется всеми сигналами генерации.

Иначе говоря, стратегия префиксного сумматора заключается в вычислении входного сигнала переноса C_{i-1} для каждого разряда так быстро, насколько это возможно. Затем по формуле вычисляется сумма:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5.7}$$

Определим разряд i =-1 для вычисления C_{in}: $G_{-1} = C_{in}$ и P_{-1} =0. Следовательно, $C_{i-1} = G_{i-1:-1}$, так как выходной сигнал переноса i-1-го разряда будет активным, если блок, охватывающий разряды от i-1 до -1, генерирует перенос. Полученный перенос генерируется или в разряде i-1, или в предыдущем разряде и затем распространяется дальше. Следовательно, мы можем переписать **уравнение (5.7)**, как

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5.8}$$

Первые компьютеры использовали сумматоры с ускоренным переносом, так как компоненты стоили очень дорого, а такие сумматоры используют меньше аппаратных ресурсов. Практически все современные компьютеры используют префиксные сумматоры в критических путях, так как транзисторы стали дешевле, а быстродействие – один из важнейших показателей.

Таким образом, основной проблемой является быстрое вычисление всех блоковых сигналов генерации $G_{-1:-1}$, $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, . . . , $G_{N-2:-1}$.

Эти сигналы вместе с $P_{-1:-1}$, $P_{0:-1}$, $P_{1:-1}$, $P_{2:-1}$, . . . , $P_{N-2:-1}$ называют *префиксными.*

На **Рис. 5.7** показан 16-разрядный префиксный сумматор. Его работа начинается с предварительного формирования сигналов P_i и G_i для всех разрядов A_i и B_i с использованием элементов И и ИЛИ. Затем используется $\log_2 N = 4$ уровня черных ячеек для формирования префиксов $G_{i:j}$ и $P_{i:j}$. Черная ячейка принимает входы из верхней части блока, охватывающего биты $i:k$, и из нижней части блока, охватывающего биты $k-1:j$. Затем эти части объединяются для формирования сигналов генерации и распространения всего блока, охватывающего биты $i:j$. Используя **уравнения (5.9)** и **(5.10)**, получим

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{5.9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{5.10}$$

Другими словами, блок, охватывающий биты $i:j$, будет генерировать сигнал переноса, если верхняя часть генерирует перенос или если она распространяет перенос, сгенерированный в нижней части. Блок будет распространять перенос, если и верхняя, и нижняя части распространяют его. В итоге, префиксный сумматор вычисляет сумму на основе **уравнения (5.8)**.

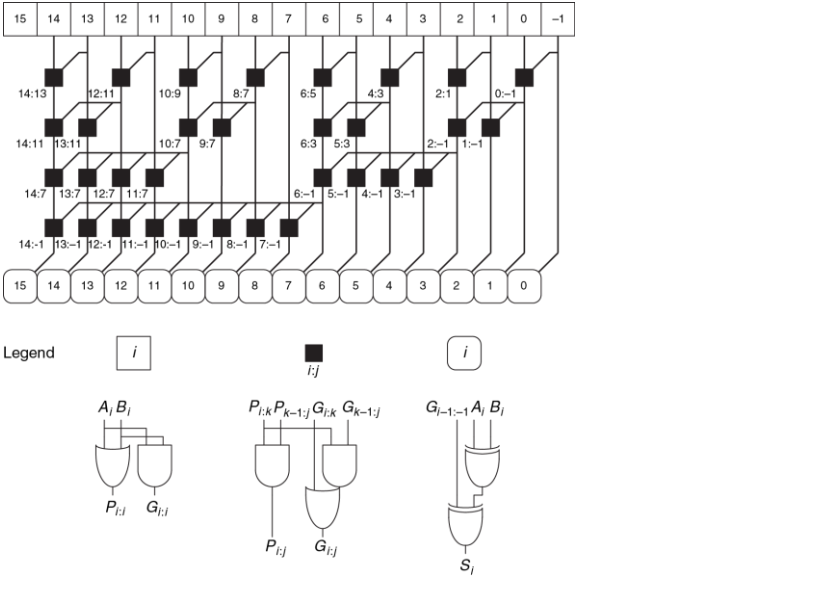


Рис. 5.7 **16-разрядный префиксный сумматор**

Таким образом, задержка префиксного сумматора достигает значения, которое растет с числом разрядов сумматора логарифмически, а не линейно. Ускорение значительно, особенно для сумматоров, имеющих 32 и более разрядов. Такой сумматор использует существенно больше аппаратных средств, чем простой сумматор с ускоренным переносом. Сеть черных ячеек называется *префиксным деревом.*

Основной принцип использования префиксного дерева, при котором время вычислений растет логарифмически с ростом числа входов, является мощной технологией. При некотором умении этот принцип может быть применен для многих других схем (см. например, **упражнение 5.7**).

Критический путь N-разрядного префиксного сумматора включает в себя предварительное вычисление P_i и G_i, за которым следует $\log_2 N$ каскадов черных ячеек для получения всех префиксов. Затем сигналы $G_{i\text{-}1:\text{-}1}$ обрабатываются финальными элементами «исключающее ИЛИ» в нижней части схемы для получения сигнала S_i. Задержка N-разрядного префиксного сумматора равна

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \qquad (5.11)$$

где t_{pg_prefix} – задержка черной префиксной ячейки.

Пример 5.2 ЗАДЕРЖКА ПРЕФИКСНОГО СУММАТОРА

Вычислим задержку 32-разрядного префиксного сумматора. Допустим, что задержка каждого двухвходового логического элемента равна 100 пс.

Решение: Задержка распространения каждой черной префиксной ячейки равна t_{pg_prefix} = 200 пс (задержки 2-х логических элементов). Таким образом, используя **уравнение (5.11)**, задержка распространения 32-разрядного префиксного сумматора равна 100 пс + $\log_2(32)$ + 200 пс + 100 пс + 1.2 нс, что примерно в 3 раза меньше, чем у сумматора с ускоренным переносом из **примера 5.1**. В действительности, выгода не такая большая, но префиксные сумматоры действительно работают существенно быстрее, чем любые альтернативные.

Заключение

В этом разделе были рассмотрены полусумматор, полный сумматор и три типа сумматоров с распространяющимся переносом: сумматоры с последовательным переносом, ускоренным переносом и префиксный сумматор. Быстрые сумматоры используют больше аппаратных средств и, следовательно, являются более дорогостоящими и энергозатратными. Все это должно быть учтено при выборе нужного сумматора в процессе проектирования.

Языки описания аппаратуры предоставляют возможность использования операции + для определения сумматора с распространяющимся переносом. Современные средства синтеза выбирают из множества возможных реализаций проекта самую

дешевую и простую, которая удовлетворяет требованиям по скорости. Это очень упрощает работу проектировщика. В **примере 5.1 HDL** с помощью языков описания аппаратуры описан сумматор с распространяющимся переносом, имеющий вход и выход переноса.

Пример 5.1 HDL HDL СУММАТОР

SystemVerilog

```
module adder #(parameter N = 8)
              (input logic [N-1:0]  a, b,
               input logic          cin,
               output logic [N-1:0] s,
               output logic         cout);
   assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity adder is
   generic(N: integer := 8);
   port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
        cin:  in  STD_LOGIC;
        s:    out STD_LOGIC_VECTOR(N-1 downto 0);
        cout: out STD_LOGIC);
   end;
   architecture synth of adder is
     signal result: STD_LOGIC_VECTOR(N downto 0);
   begin
     result <= ("0" & a) + ("0" & b) + cin;
     s      <= result(N-1 downto 0);
     cout   <= result(N);
   end;
```

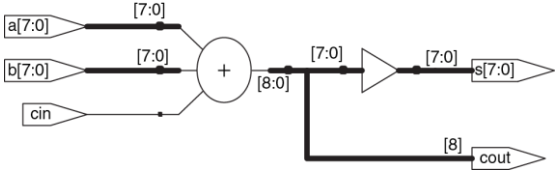


Рис. 5.8 **Синтезированный сумматор**

5.2.2 Вычитание

В **параграфе 1.4.6** было показано, что сумматоры могут складывать положительные и отрицательные числа, используя представление числа в дополнительном коде. Вычитание производится почти также просто: меняется знак второго числа, затем числа складываются. Изменение знака числа в дополнительном коде производится путем инверсии битов и прибавления 1.

Для вычисления $Y = A - B$ вначале создается дополнительный код числа B: инвертируются разряды B и прибавляется 1; $-B = \overline{B} + 1$. Полученное значение складывается с A. Эта сумма может быть получена одним сумматором с распространяющимся переносом путем сложения $A + \overline{B}$ при $C_{in} = 1$. На **Рис. 5.9** показано условное обозначение устройства вычитания и базовая аппаратная реализация для вычисления $Y = A - B$.

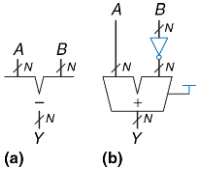


Рис. 5.9 **Устройство вычитания: (a) условное обозначение, (b) реализация**

Пример 5.2 HDL описывает операцию вычитания.

Пример 5.2 HDL УСТРОЙСТВО ВЫЧИТАНИЯ

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic [N-1:0] y);
    assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity subtractor is
   generic(N: integer := 8);
   port(a, b:  in  STD_LOGIC_VECTOR(N-1 downto 0);
        y:     out STD_LOGIC_VECTOR(N-1 downto 0));
end;
architecture synth of subtractor is
begin
  y <= a - b;
end;
```

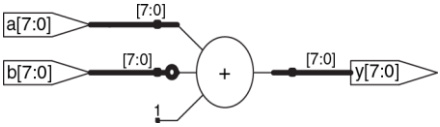


Рис. 5.10 **Синтезированное устройство вычитания**

5.2.3 Компараторы

Компараторы определяют, являются ли два двоичных числа равными или одно из них больше/меньше другого. Компаратор получает два *N*-разрядных двоичных числа *A* и *B*. Существует 2 типа компараторов.

Компаратор равенства выдает один выходной сигнал, показывая, равны ли *A* и *B* (*A*==*B*). *Компаратор величины* выдает один и более выходных сигналов, показывая отношение величин *A* и *B*.

Компаратор равенства имеет простую аппаратную реализацию. На **Рис. 5.11** показано обозначение и реализация 4-разрядного компаратора равенства. Сначала, с помощью логических элементов XNOR, он проверяет, являются ли соответствующие разряды *A* и *B* равными. Значения будут равными, если все соответствующие разряды равны.

Как показано на **Рис. 5.12**, компаратор величины вычисляет *A*-*B* и анализирует знак (самый старший разряд) результата. Если результат отрицательный (самый старший разряд = 1), то *A* меньше *B*. В противном случае *A* больше или равно *B*.

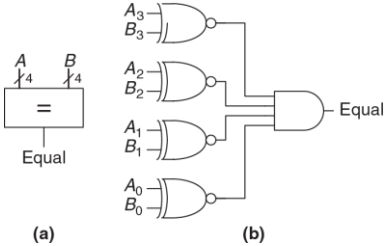


Рис. 5.11 **4-разрядный компаратор равенства: (a) условное обозначение, (b) реализация**

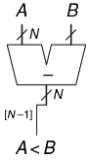


Рис. 5.12 **N-разрядный компаратор величины**

Пример 5.3 HDL показывает использование этих двух типов компараторов.

Пример 5.3 HDL КОМПАРАТОРЫ

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);
  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a <b);
  assign lte = (a <= b);
  assign gt  = (a >b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity comparators is
  generic(N: integer : = 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;
architecture synth of comparator is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
```

```
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

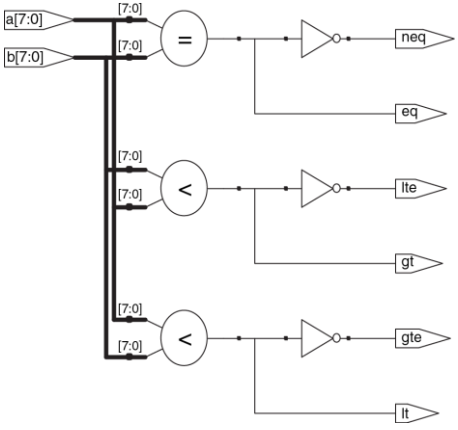


Рис. 5.13 **Синтезированный компаратор**

5.2.4 АЛУ

Арифметико-логическое устройство (*АЛУ, Arithmetic/Logical Unit* (*ALU*) объединяет различные арифметические и логические операции в одном узле. Например, типичное АЛУ может выполнять сложение, вычитание, сравнение величин, операции «И» и «ИЛИ». АЛУ входит в ядро большинства компьютерных систем.

На **Рис. 5.14** показано условное обозначение *N*-разрядного АЛУ с *N*-разрядными входами и выходами. В АЛУ поступает управляющий сигнал *F*, который определяет, какую функцию нужно выполнить. Обычно сигналы управления показывают голубым цветом, чтобы отличать их от сигналов данных. В **Табл. 5.1** перечислены типичные функции, которые выполняет АЛУ. Функция SLT используется для сравнения по значению и будет рассмотрена чуть позже в этом же разделе.

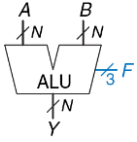


Рис. 5.14 **Условное обозначение арифметико-логического устройства (АЛУ)**

Табл. 5.1 Операции АЛУ

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

На **Рис. 5.15** показана реализация узла АЛУ. Он состоит из *N*-битного сумматора, *N* двухвходовых логических элементов И и двухвходовых логических элементов ИЛИ. Также он содержит инверторы и мультиплексор для инверсии битов *B*, когда управляющий сигнал F_2 активен. Мультиплексор с организацией 4:1 выбирает необходимую функцию исходя из сигналов управления $F_{1:0}$.

Говоря точнее, арифметические и логические блоки АЛУ оперируют с *A* и *BB*. *BB* равно *B* или $\overline{B}$, в зависимости от F_2. Если $F_{1:0}$ = 00, то выходной мультиплексор выбирает операцию *A* AND *BB*. Если $F_{1:0}$ = 01, то АЛУ вычисляет *A* OR *BB*. Если $F_{1:0}$ = 10, то АЛУ выполнит сложение или вычитание. Заметьте, что F_2 также является входом переноса

сумматора. При использовании дополнительного кода $\overline{B} + 1 = -B$. Если $F_2 = 0$, то АЛУ вычисляет $A + B$, если $F_2 = 1$, то будет вычислена разница $A + \overline{B} + 1 = A - B$.

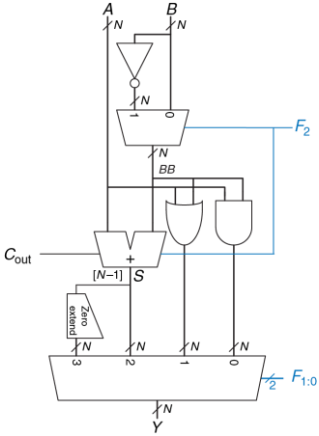


Рис. 5.15 **N-разрядное АЛУ**

Когда $F_{2:0}$ = 111 АЛУ выполняет операцию SLT *(set if less than – установить, если меньше, чем)*. Когда $A < B$, Y = 1. В противном случае Y = 0. Таким образом, Y устанавливается в 1, если A меньше B.

Операция SLT выполняется путем вычисления $S = A – B$. Если S отрицательно (т.е. установлен знаковый бит), то A меньше B. *Модуль дополнения нулями* создает N-разрядный выходной сигнал, объединяя однобитовый вход с нулем в самых старших разрядах. Бит знака (разряд N-1) переменной S будет входом для модуля установки нуля.

Пример 5.3 УСТАНОВИТЬ, ЕСЛИ МЕНЬШЕ

Построим 32-разрядный АЛУ с операцией SLT. Допустим, $A = 25_{10}$, $B = 32_{10}$. Найдем сигнал управления и выходной сигнал Y.

Решение: Так как $A < B$, то Y должен быть равен 1. Для SLT, $F_{2:0}$ = 111. Вместе с сигналом F_2 = 1 это сконфигурирует модуль сумматора как вычитающее устройство с выходным сигналом S: $25_{10} – 32_{10} = -7_{10} = 1111…1001_2$. Учитывая $F_{1:0}$ = 11, выходной сигнал мультиплексора будет $Y = S_{31} = 1$.

Некоторые АЛУ имеют специальные выходные сигналы, называемые *флагами*, которые показывают информацию о выходе АЛУ. Например, *флаг переполнения* показывает, что результат работы сумматора переполнился. *Флаг нуля* показывает, что выход АЛУ установился в 0.

Описание *N*-разрядного АЛУ на HDL оставлено для **упражнения 5.9**. Существует много вариаций базового АЛУ, которые выполняют такие функции как XOR или сравнение на равенство.

5.2.5 Схемы сдвига и циклического сдвига

Схемы *сдвига и схемы циклического сдвига* перемещают биты и, следовательно, умножают или делят число на степень 2. В соответствии с названием, схемы сдвига передвигают разряды двоичного числа влево или вправо на определенное число позиций. Существует несколько видов таких схем:

- **Логические схемы сдвига** сдвигают число влево (LSL) или вправо (LSR) и заполняют пустые разряды нулями.

  Например, 11001 LSR 2 = 00110; 11001 LSL 2 = 00100

- **Арифметические схемы сдвига** действуют так же, как и логические, но при сдвиге вправо они заполняют наиболее значащие разряды значением знакового бита исходного числа. Это необходимо при умножении и делении чисел со знаком. (смотри **разделы 5.2.6** и **5.2.7**). Арифметический сдвиг влево (ASL) работает так же, как и логический (LSL).

  Например: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100

- **Схемы циклического сдвига** сдвигают число по кругу так, что пустые места заполняются разрядами, которые выдвинуты из другого конца.

  Например: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111

N-разрядная схема сдвига может быть построена из *N* мультиплексоров *N*:1. Вход сдвигается на 0 – *N*-1 разрядов в зависимости от значения log_2N линий выбора. На **Рис. 5.16** показаны условное обозначение и аппаратная реализация 4-разрядной схемы сдвига. Операторы <<, >> и >>> обычно обозначают сдвиг влево, логический сдвиг вправо и арифметический сдвиг вправо соответственно. В зависимости от значения 2-разрядной величины сдвига $shamt_{1:0}$, на выход *Y* поступает входной сигнал *A*, сдвинутый на 0 – 3 разряда. Для всех схем сдвига, если $shamt_{1:0}$ = 00, то *Y* = *A*. В **упражнении 5.14** рассматривается разработка схем циклического сдвига.

Сдвиг влево – это частный случай умножения. Сдвиг влево на *N* бит умножает число на 2*N*. Например, $000011_2 << 4 = 110000_2$ равносильно $3_{10} \times 2^4 = 48_{10}$.

Арифметический сдвиг вправо – это специальный случай деления. Арифметический сдвиг вправо на *N* бит делит число на 2^N. К примеру, $11100_2 >>> 2 = 11111_2$ равносильно $-4_{10}/2^2 = -1_{10}$.

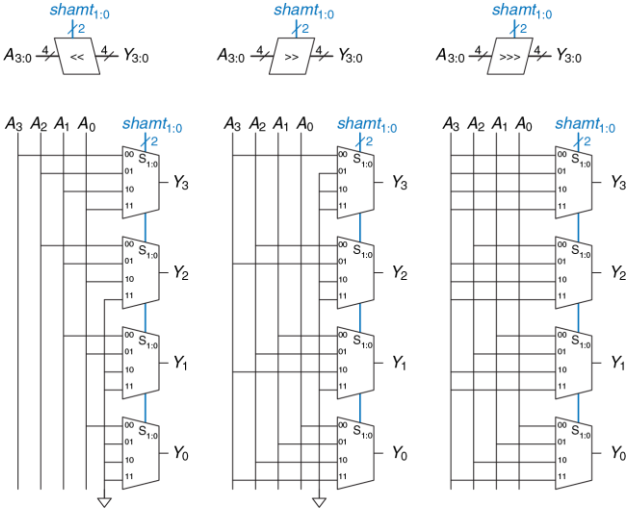


Рис. 5.16 **4-разрядные схемы сдвига: (a) сдвиг влево, (b) логический сдвиг вправо, (c) арифметический сдвиг вправо**

5.2.6 Умножение

Умножение беззнаковых двоичных чисел подобно десятичному умножению, однако оно оперирует только с единицами и нулями. На **Рис. 5.17** сравнивается умножение двоичных и десятичных чисел.

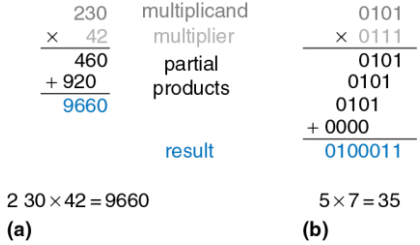


Рис. 5.17 **Умножение: (a) десятичное, (b) двоичное**

В обоих случаях *частичные произведения* формируются путем умножения отдельных разрядов множителя на всё множимое. Сдвинутые частичные произведения затем складываются, и мы получаем результат.

В общем случае, умножитель *N*x*N* перемножает два *N*-разрядных числа и порождает 2*N*-разрядный результат. Частичные произведения при двоичном умножении равны или множимому, или нулю. Умножение

одного разряда двоичных чисел равносильно операции И, поэтому для формирования частичных произведений используются логические элементы И.

На **Рис. 5.18** показаны условное обозначение, функциональное описание и аппаратная реализация умножителя 4x4. Умножитель получает множимое и множитель A и B и вычисляет произведение P. На **Рис. 5.18 (b)** показано, как формируются частичные произведения. Каждое частичное произведение равно результату операций И, аргументами которых являются отдельные разряды множителя (B_3, B_2, B_1, или B_0) и все разряды множимого (A_3, A_2, A_1, A_0). Для N-разрядных операндов будет существовать N частичных произведений и N-1 каскадов (стадий) одноразрядных сумматоров. Например, для умножителя 4x4 частичное произведение первого ряда – это B_0 AND (A_3, A_2, A_1, A_0). Это частичное произведение прибавляется к сдвинутому второму частичному произведению B_1 AND (A_3, A_2, A_1, A_0). Следующие ряды логических элементов И и сумматоров формируют и добавляют оставшиеся частичные произведения.

Описание умножителя на HDL приведено в **примере 5.4 HDL**. Так же как и для сумматоров, существует множество реализаций умножителей с различными компромиссами между скоростью и стоимостью. Инструментальные средства синтеза могут выбирать наиболее подходящую реализацию по заданным временным ограничениям.

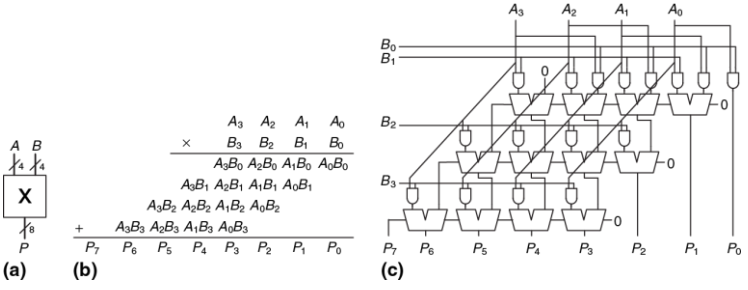


Рис. 5.18 Умножитель 4 × 4: (a) условное обозначение, (b) функции, (c) реализация

Пример 5.4 HDL УМНОЖИТЕЛЬ

SystemVerilog

```
module multiplier #(parameter N = 8)
                   (input logic [N-1:0] a, b,
                    output logic [2*N-1:0] y);
   assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity multiplier is
  generic(N: integer := 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       y:    out STD_LOGIC_VECTOR(2*N-1 downto 0));
end;
architecture synth of multiplier is
begin
  y <= a * b;
end;
```

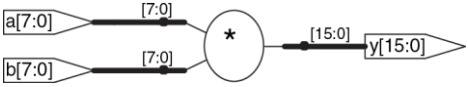


Рис. 5.19 Синтезированный умножитель

5.2.7 Деление

Двоичное деление *N*-разрядных беззнаковых чисел в диапазоне [0, 2^{N-1}] может быть выполнено с использованием следующего алгоритма:

```
R' = 0
for i = N-1 to 0
   R = {R' << 1, Ai}
   D = R - B
   if D < 0 then  Qi = 0, R' = R  // R < B
   else           Qi = 1, R' = D  // R ≥ B
R = R'
```

Частичный остаток *R* инициализируется 0. Наиболее значимый разряд делимого *A* затем становится наименее значимым разрядом *R*. Делитель *B* многократно вычитается из частичного остатка и определяется знак разницы *D*. Если она отрицательна (т.е. знаковый разряд равен 1), то разряд частного Q_i равен 0, и разница отбрасывается. В противном случае – Q_i равен 1 и частичный остаток обновляется, он становится равным разнице *D*. Затем частичный остаток удваивается (сдвигается влево на один разряд), и процесс повторяется. Результат удовлетворяет условию $A/B = Q + R/B$.

На **Рис. 5.20** показана схема 4-разрядной матрицы деления.

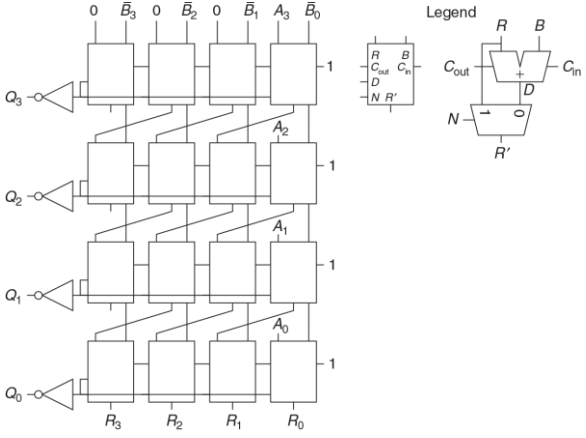


Рис. 5.20 **Матрица деления**

Схема вычисляет A/B и на выход выдает частное Q и остаток R. На вставке показаны условное обозначение и схемы каждого блока в матрице деления. Сигнал N показывает, является ли результат $R–B$ отрицательным. Это определяется по выходному сигналу переноса C_{out} самого левого блока в ряду, который является знаком разницы.

Задержка *N*-разрядной матрицы деления увеличивается пропорционально N^2, так как перенос должен пройти через все *N* каскадов в ряду перед тем, как определится знак и мультиплексор выберет *R* или *D*. Это повторяется для всех *N* рядов. Деление – очень медленная и дорогая операция в аппаратной реализации, поэтому ее следует использовать как можно реже.

5.2.8 Дополнительная литература

Компьютерная арифметика может быть предметом целой книги. Учебник Эрцеговича и Ланга (Ercegovac & Lang) «*Digital Arithmetic*» – отличный обзор всей области. «*CMOS VLSI Design*» Весте и Харриса (Weste & Harris) охватывает проектирование высокопроизводительных схем для арифметических операций.

5.3 ПРЕДСТАВЛЕНИЕ ЧИСЕЛ

Компьютер работает как с целыми, так и с дробными числами. До настоящего момента мы рассматривали только представления знаковых и беззнаковых целых чисел, которые были описаны в **параграфе 1.4**. В данном разделе вводится представление чисел с фиксированной и с плавающей точкой, с помощью которого можно представить рациональные числа. Числа с фиксированной точкой – это аналог десятичных чисел; некоторые биты представляют целую часть, а оставшиеся – дробную. Числа с плавающей точкой являются аналогом экспоненциального представления числа с мантиссой и порядком (прим. переводчика: в англоязычных странах в качестве разделителя целой и дробной частей чисел используется точка, а не запятая. Так сложилось, что в современной русскоязычной технической литературе термин «точка» используется чаще, поэтому и мы будем его использовать. В некоторых других книгах используется термин «запятая»).

5.3.1 Числа с фиксированной точкой

Представление «с фиксированной точкой» подразумевает двоичную запятую между битами целой и дробной части, аналогично десятичной точке между целой и дробной частями обычного десятичного числа.

Например, на **Рис. 5.21 (a)** показано число с фиксированной точкой с 4-мя битами целой части и 4-мя дробной.

На **Рис. 5.21 (b)** голубым цветом показана двоичная запятая, а на **Рис. 5.21 (c)** изображено эквивалентное десятичное число.

Знаковые числа с фиксированной точкой можно использовать как в прямом, так и в дополнительном коде.

На **Рис. 5.22** показаны оба представления числа –2.375 с фиксированной запятой с использованием 4-х целых бит и 4-х дробных бит. Неявная двоичная запятая для ясности изображена голубым цветом.

(a) 01101100

(b) 0110.1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

Рис. 5.21 Представление числа 6.75 с фиксированной точкой с четырьмя битами целой части и четырьмя дробной

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Рис. 5.22 Представление числа –2.375 с фиксированной точкой: (a) абсолютное значение, (b) прямой код, (c) дополнительный код

В прямом коде знаковый бит используется для указания знака. Дополнительный код двоичного числа получается инверсией битов

абсолютного значения и добавления 1 к младшему разряду. В этом примере младший разряд соответствует 2^{-4}.

Как и все представления двоичных чисел, числа с фиксированной точкой являются лишь набором бит. Не существует способа узнать о существовании двоичной точки кроме как из соглашения между людьми, интерпретирующими число.

Пример 5.4 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ С ЧИСЛАМИ С ФИКСИРОВАННОЙ ТОЧКОЙ

Вычислим выражение 0.75 ±0.625, используя числа с фиксированной точкой.

Решение: Сначала преобразуем 0.625, абсолютное значение второго числа, в стандартное представление двоичного числа с фиксированной точкой. $0.625 \geq 2^{-1}$, следовательно, ставим 1 в разряд 2^{-1}, оставляя $0.625 – 0.5 = 0.125$. Так как $0.125 < 2^{-2}$, то ставим 0 в разряд 2^{-2}. Так как $0.125 \geq 2^{-3}$, то ставим 1 в разряд 2^{-3}, оставляя $0.125 – 0.125 = 0$. Таким образом, в разряде 2^{-4} будет 0. Таким образом, $0.62510 = 0000.1010_2$.

На **Рис. 5.23** показано преобразование числа –0.625 в двоичное представление в дополнительном коде. На **Рис. 5.24** показано сложение чисел с фиксированной запятой и, для сравнения, десятичный эквивалент.

Заметьте, что первый единичный бит в двоичном представлении числа с фиксированной точкой на **Рис. 5.24 (а)** отброшен в 8-битовом результате.

0000.1010	Binary Magnitude
1111.0101	One's Complement
+ 1	Add 1
1111.0110	Two's Complement

Рис. 5.23 **Представление числа в дополнительном коде**

0000.1100	0.75
+ 1111.0110	+ (–0.625)
10000.0010	0.125
(a)	**(b)**

Рис. 5.24 **Сложение: (a) двоичных чисел с фиксированной точкой, (b) десятичный эквивалент**

Для корректных вычислений с использованием чисел с фиксированной точкой используется двоичное представление в дополнительном коде.

5.3.2 Числа с плавающей точкой

$\pm M \times B^E$

Рис. 5.25 **Числа с плавающей точкой**

Числа с плавающей точкой соответствуют экспоненциальному представлению. В этом представлении преодолены ограничения наличия только фиксированного количества целых и дробных бит, поэтому оно позволяет представлять очень большие и очень маленькие числа. Как и в экспоненциальном представлении, числа с плавающей запятой имеют знак, мантиссу (M), основание (B) и порядок (E), что показано на **Рис. 5.25**.

К примеру, число $4.1*10^3$ является десятичным экспоненциальным представлением числа 4100. Мантиссой является 4.1, основание

равно 10, а порядок равен 3. Десятичная запятая «переплывает» на позицию правее самого значимого (старшего) разряда. У чисел с плавающей точкой основание будет равно 2, а мантисса будет двоичным числом. 32 бита используются для представления 1 знакового бита, 8 бит порядка и 23 бит мантиссы.

Пример 5.5 32-БИТНОЕ ЧИСЛО С ПЛАВАЮЩЕЙ ТОЧКОЙ

Найдите представление десятичного числа 228 в виде числа с плавающей точкой.

Решение: Для начала преобразуем десятичное число в двоичное:

$228_{10} = 11100100_2 = 1.11001_2 * 2^7$. На **Рис. 5.26** показано 32-битное кодирование, которое далее для эффективности будет модифицировано. Знаковый бит положительный, равен 0, 8 бит порядка дают значение 7, а оставшиеся 23 бита – это мантисса.

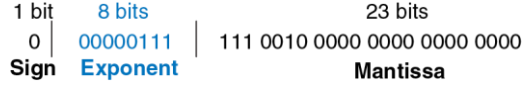


Рис. 5.26 **32-разрядной кодирование числа с плавающей точкой: версия 1**

В двоичных числах с плавающей точкой первый бит мантиссы (слева от точки) всегда равен 1, и поэтому его можно не сохранять.

Это называется неявная старшая единица. На **Рис. 5.27** изображено модифицированное представление:

$228_{10} = 11100100_2 * 2^0 = 1.11001_2 * 2^7$. Неявная старшая единица не входит в 23 бита мантиссы. Сохраняются только дробные биты. Это освобождает дополнительный бит для полезных данных.

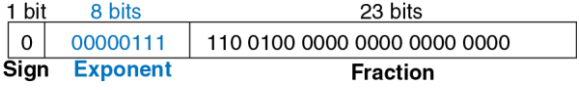


Рис. 5.27 Кодирование числа с плавающей точкой: версия 2

Сделаем последнюю модификацию представления порядка. Порядок должен представлять, как положительный показатель степени, так и отрицательный. Для этого в формате с плавающей точкой используется смещенный порядок, который представляет собой первоначальный порядок плюс постоянное смещение. 32-битное представление с плавающей точкой использует смещение 127. Например, для порядка 7, смещенный порядок будет выглядеть так: $7 + 127 = 134 = 10000110_2$, для порядка -4 смещенный порядок равен- $4 + 127 = 123 = 01111011_2$.

На **Рис. 5.28** показано представление числа $1.11001_2 * 2^7$ в формате с плавающей точкой с неявной старшей единицей и смещенным порядком 134 (7 + 127). Это представление соответствует стандарту IEEE 754.

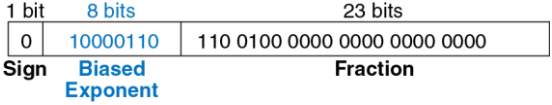


Рис. 5.28 Представление числа с плавающей точкой по стандарту IEEE 754

Особые случаи: 0, ±∝ и NaN

Стандарт IEEE для чисел с плавающей точкой включает в себя особые случаи представления таких чисел, как 0, бесконечность и недопустимые результаты. К примеру, представить число 0 в виде числа с плавающей точкой невозможно из-за наличия неявной старшей единицы. Для этих случаев зарезервированы специальные коды: порядок состоит только из нулей или единиц. В **Табл. 5.2** показано обозначение 0, ±∝, NaN. Как и в знаковых числах, плавающая запятая имеет и положительный и отрицательный 0. NaN используется для чисел, которые не существуют, например корень из -1 и $\log_2(-5)$.

Табл. 5.2 Обозначение 0, ±∞ и NaN в соответствии со стандартом IEEE 754

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∝	0	11111111	00000000000000000000000
±∝	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Очевидно, что существует много разумных способов представления чисел с плавающей точкой. Много лет производители компьютеров использовали несовместимые форматы. Результат от одного компьютера не мог быть непосредственно интерпретирован другим. Институт инженеров электротехники и электроники (Institute of Electricaland Electronics Engineers, IEEE) решил эту проблему, определив в 1985 году стандарт IEEE754. Сейчас этот формат используется повсеместно, и он будет обсуждаться в данном разделе.

Форматы одинарной и двойной точности

Ранее мы рассматривали 32-битные числа с плавающей точкой. Такой формат также называют форматом одинарной точности. Стандарт IEEE 754 также определяет 64-битные числа двойной точности, которые позволяют представить большой диапазон чисел с большой точностью. В **Табл. 5.3** приведено число бит, используемых в полях разных форматов.

Табл. 5.3 Числа с плавающей точкой с одинарной и двойной точностью

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

Если исключить специальные случаи, упомянутые ранее, обычные числа одинарной точности охватывают диапазон от $\pm 1.175494 * 10^{-38}$ до

±3.402824 * 10^{38}. Их точность составляет около 7 десятичных разрядов, так как $2^{-24} \approx 10^{-7}$. Числа с двойной точностью охватывают диапазон от ±2.22507385850720 * 10^{-308} до ±1.79769313486232 * 10^{308} и имеют точность около 15 десятичных разрядов.

Некоторые числа нельзя точно представить в виде числа с плавающей точкой, как, например, 1.7. Однако, когда вы вводите 1.7 на калькуляторе, вы видите точно 1.7, не 1.69999… Для этого большинство приложений, как например калькулятор и различные финансовые программы, используют двоично-десятичный формат (BCD) или формат с основанием 10. Числа в таком формате кодируют каждый десятичный разряд с помощью 4 бит со значением от 0 до 9. Например, число 1.7 в формате BCD с четырьмя целыми и четырьмя дробными битами представляет собой 0001.0111. Конечно, не все так просто. Ценой является усложнение арифметических схем и неполное использование кодировки (не используются кодировки A-F), следовательно, снижается эффективность. Таким образом, для ресурсоемких приложений числа в формате с плавающей точкой гораздо эффективнее.

Округление

Арифметические результаты, которые выходят за пределы доступной точности, необходимо округлять до наиболее близких чисел. Существуют следующие способы округления: округление в меньшую сторону (1), округление в большую сторону (2), округление до нуля (3) и

округление к ближайшему числу (4). По умолчанию принято округление к ближайшему числу. В этом случае, если два числа находятся на одинаковом расстоянии, то выбирается то, у которого будет ноль в младшем разряде дробной части.

Напомним, что число переполняется, когда его величина слишком велика для какого-либо представления. Аналогично, число является изчезающе малым, когда оно слишком мало для представления. При округлении (4) переполненные числа округляются до $\pm\infty$, а изчезающе малые округляются до нуля.

Сложение чисел с плавающей точкой

Сложение чисел с плавающей точкой не такая простая операция, как в случае представления чисел в дополнительном коде. Для выполнения сложения двух таких чисел необходимо выполнить следующие шаги:

1. Выделить биты порядка и мантиссы.
2. Присоединить неявную старшую единицу к мантиссе.
3. Сравнить порядки.
4. При необходимости сдвинуть мантиссу числа, имеющего меньший порядок.
5. Сложить мантиссы.

6. При необходимости нормализовать мантиссу и порядок.
7. Округлить результат.
8. Собрать обратно порядок и мантиссу в итоговое число с плавающей точкой.

На **Рис. 5.29** показан процесс сложения чисел с плавающей точкой 7.875 ($1.11111 * 2^2$) и 0.1875 ($1.1 * 2^{-3}$). Результат равен 8.0625 ($1.0000001 * 2^3$). После извлечения мантиссы и порядка, присоединения неявной старшей единицы (шаги 1 и 2), порядки сравнивается путем вычитания меньшего порядка из большего. Результатом будет число бит, на которое необходимо сдвинуть мантиссу меньшего число вправо (шаг 4) для выравнивания двоичной точки (т.е. чтобы сделать порядки равными). Выровненные значения складываются. Так как мантисса суммы больше или равна 2.0, результат нужно нормализовать, сдвинув его вправо на 1 бит и увеличить порядок на 1. В этом пример результат точный и никаких округлений не требуется. Он сохраняется в формате с плавающей точкой, после удаления неявной старшей единицы мантиссы и добавления знакового бита.

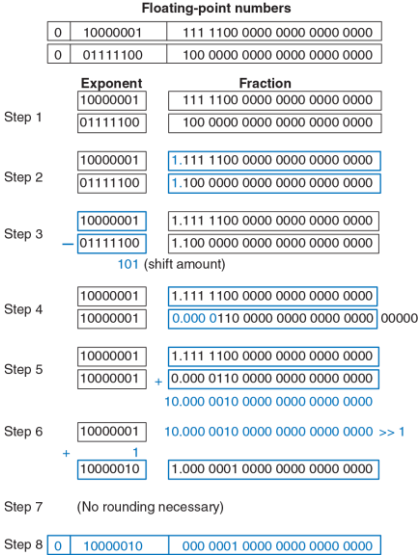


Рис. 5.29 Сложение чисел с плавающей точкой

Вычисления при использовании чисел в формате с плавающей точкой обычно выполняются с помощью специальных аппаратных средств для увеличения скорости. Такая аппаратура называется FPU (floating-point unit), она, как правило, отличается от CPU (central processing unit). Печально известный баг FDIV (floating-point division) в FPU процессора Pentium стоил компании Intel $475 миллионов, которые она вынуждена была потратить на отзыв и замену дефектных микросхем. Ошибка произошла только потому, что не была правильно загружена таблица преобразования.

5.4 ФУНКЦИОНАЛЬНЫЕ УЗЛЫ ПОСЛЕДОВАТЕЛЬНОСТНОЙ ЛОГИКИ

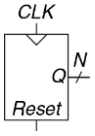
В этом разделе будут рассмотрены функциональные узлы последовательностной логики – счетчики и сдвигающие регистры.

5.4.1 Счетчики

N-разрядный двоичный счетчик, который показан на **Рис. 5.30**, представляет собой последовательностную арифметическую схему, которая имеет входы тактового сигнала, сброса и *N*-разрядный выход *Q*. Сигнал сброса инициализирует выходы нулевым значением. Выход счетчик последовательно принимает все 2^N возможные значения *N*-разрядного двоичного числа, переход к следующему значению происходит по переднему фронту тактового импульса.

На **Рис. 5.31** показан *N*-разрядный счетчик, который состоит из сумматора и регистра, имеющего вход сброса. На каждом цикле счетчик добавляет 1 к величине, которая хранится в регистре. В **примере 5.5 HDL** на языках HDL описан двоичный счетчик с асинхронным сбросом.

Счетчики других типов, например, реверсивные, рассмотрены в **упражнениях 5.43–5.46**.



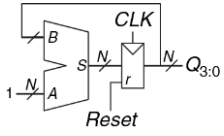


Рис. 5.30 Условное обозначение счетчика **Рис. 5.31** N-битовый счетчик

Пример 5.5 HDL СЧЕТЧИК

SystemVerilog

```
module counter #(parameter N = 8)
                (input logic clk,
                 input logic reset,
                 output logic [N-1:0] q);
   always_ff @(posedge clk, posedge reset)
     if (reset)q <= 0;
     else      q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity counter is
  generic(N: integer := 8);
  port(clk, reset: in STD_LOGIC;
       q:          out STD_LOGIC_VECTOR(N-1 downto 0));
```

```
end;
architecture synth of counter is
begin
  process(clk, reset) begin
    if reset then               q <= (OTHERS => '0');
    elsif rising_edge(clk) then q <= q + '1';
    end if;
  end process;
end;
```

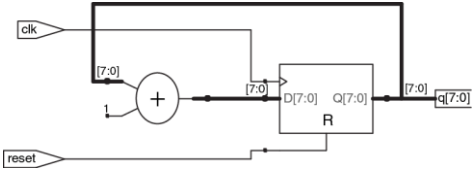


Рис. 5.32 **Синтезированный счетчик**

5.4.2 Сдвигающие регистры

На **Рис. 5.33** показан сдвигающий регистр, который имеет вход тактового сигнала, последовательный вход S_{in}, последовательный выход S_{in} и N параллельных выходов $Q_{N-1:0}$. По каждому переднему фронту тактового импульса в первый триггер регистра записывается

новый бит со входа S_{in} а содержимое следующих триггеров сдвигается вперед. Последний бит регистра можно считать с выхода S_{out}.

Сдвигающий регистр можно рассматривать как последовательно-параллельный преобразователь. На вход S_{in} поступают последовательные данные (по одному биту за раз). После N циклов последние N значений входного сигнала можно параллельно считать с выхода Q.

> Не следует путать сдвигающие регистры и схемы сдвига, которые были рассмотрены в **разделе 5.2.5**. Сдвигающий регистр является последовательностной схемой, в которую по каждому фронту тактового сигнала поступает новый бит. Схема сдвига является комбинационной схемой, которая сдвигает биты входного сигнала на указанную величину.

Как показано на **Рис. 5.34**, сдвигающий регистр может быть построен из N последовательно соединенных триггеров. Некоторые сдвигающие регистры имеют сигнал сброса для инициализации всех триггеров.

В *параллельно-последовательный преобразователь* параллельно загружается N бит, которые затем последовательно (по одному биты за раз) поступают на выход, Схемотехника параллельно-последовательного преобразователя и сдвигающего регистра подобны. Сдвигающий регистр можно модифицировать для выполнения как

последовательно-параллельного, так и параллельно-последовательного преобразования, если к нему добавить параллельный вход $D_{N-1:0}$ и сигнал управления *Load*, как показано на **Рис. 5.35**. Когда вход *Load* активирован, во все триггеры параллельно загружаются данные со входа *D*. В противном случае сдвигающий регистр выполняет обычный сдвиг. В **примере 5.6 HDL** сдвигающий регистр описан на языках HDL.

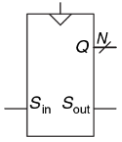


Рис. 5.33 Условное обозначение сдвигающего регистра

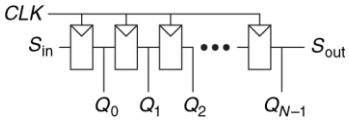


Рис. 5.34 Схема сдвигающего регистра

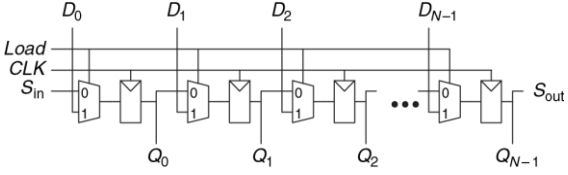


Рис. 5.35 Сдвигающий регистр с параллельной загрузкой

Пример 5.6 HDL СДВИГАЮЩИЙ РЕГИСТР С ПАРАЛЛЕЛЬНОЙ ЗАГРУЗКОЙ

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic         clk,
                  input  logic         reset, load,
                  input  logic         sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic         sout);
  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};
  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(N-1 downto 0);
       q:          out STD_LOGIC_VECTOR(N-1 downto 0);
       sout:       out STD_LOGIC);
 end;
 architecture synth of shiftreg is
```

```
 begin
  process(clk, reset) begin
    if reset = '1' then  q <= (OTHERS => '0');
    elsif rising_edge(clk) then
      if load then       q <= d;
      else               q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;
  sout <= q(N-1);
end;
```

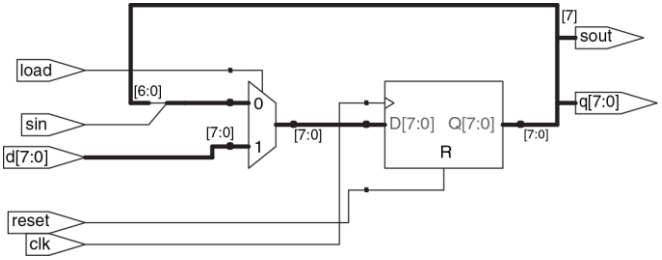


Рис. 5.36 **Сдвигающий регистр с параллельной загрузкой**

Сканирующие цепочки*

Часто для тестирования последовательностных схем используются *сканирующие цепочки* (*scan chains*), в которых используются сдвигающие регистры. Тестирование комбинационных схем относительно просто. На вход схемы подают специально подобранные входные сигналы, которые называются тестовыми векторами, а значения выходных сигналов сравнивают с ожидаемыми результатами. Тестирование последовательностных схем гораздо сложнее, поскольку их состояние зависит от предыстории входных сигналов. Если начальное состояние схемы зафиксировано, то для достижения интересующего состояния может потребоваться большое количество тестовых векторов. Например, для проверки корректности работы старшего разряда 32-битового счетчика необходимо сбросить счетчик, а затем подать на него 2^{31} (около двух миллиардов) тактовых импульсов!

Для решения этой проблемы желательно иметь возможность непосредственно наблюдать и изменять все состояния схемы. Это достигается введением специального тестового режима, в котором содержимое всех триггеров может быть считано или изменено надлежащим образом. Большинство реальных систем содержит чрезвычайно много триггеров, поэтому невозможно выделить специальные контакты для чтения и изменения их содержимого. Вместо

этого все триггеры системы соединены между собой в один огромный сдвигающий регистр, который называется сканирующей цепочкой. При нормальной работе триггеры получают данные со своих информационных входов *D*, а сканирование отключено. В тестовом режиме происходит последовательный сдвиг содержимого всех триггеров, которые входят в сканирующую цепочку, их старое содержимое поступает на выход S_{out}, а новое загружается через вход S_{in}. В состав *сканируемого триггера* (*scannable flip-flop*) кроме собственно триггера входит мультиплексор загрузки. На **Рис. 5.37** приведена схема и графическое обозначение сканируемого триггера и показано, как они соединяются последовательно для создания *N*-битового сканируемого регистра.

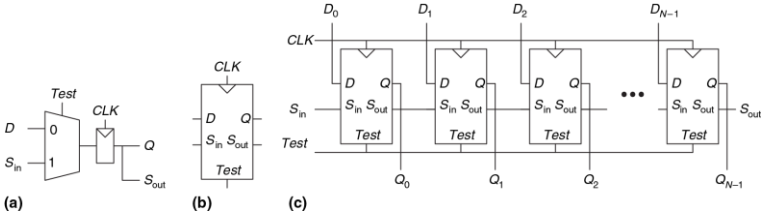


Рис. 5.37 Сканируемый триггер: (a) схема, (b) условное обозначение, and (c) *N*-битовый сканируемый регистр

Например, работу старшего разряда 32-битового счетчика можно протестировать следующим образом: в тестовом режиме он переводится в состояние 011111…111, затем выполняется один цикл счета в нормальном режиме, после этого в тестовом режиме считывается состояние счетчика, которое должно быть 100000…000. Эта последовательность действий требует только 32 + 1 + 32 = 65 циклов.

5.5 МАТРИЦЫ ПАМЯТИ

В предыдущих разделах мы познакомились с арифметическими и последовательностными схемами, которые используются для обработки данных. Для хранения этих данных и результатов работы схем в цифровых системах необходимы *запоминающие устройства* (*memories*). Регистр, состоящий из нескольких триггеров, является таким запоминающим устройством, предназначенным для хранения небольших объемов данных. В этом разделе мы рассмотрим *матрицы памяти*, которые позволяют эффективно хранить большие объемы данных.

Вначале мы познакомимся с общими характеристиками всех типов матриц памяти. Затем рассмотрим три типа матриц памяти: динамическое оперативное запоминающее устройство (ОЗУ) (DRAM,

динамическая память с произвольным доступом), статическое оперативное запоминающее устройство (SRAM, статическая память с произвольным доступом), постоянное запоминающее устройство (ПЗУ) (ROM, память только для чтения). Эти типы матриц отличаются способом хранения данных. Далее будут кратко проанализированы аппаратные затраты для создания матрицы памяти и их быстродействие. В конце раздела мы рассмотрим использование матриц памяти для выполнения функций комбинационной логики и способы их описания с помощью языков описания аппаратуры (HDL).

5.5.1 Обзор

На **Рис. 5.38** показано графические обозначение обобщенной матрицы памяти. Память организована как двумерная матрица запоминающих элементов. Содержимое памяти записывается и считывается по строкам. Эта строка выбирается *адресом* (*Address*). Записанные или считанные значения называются *данными* (*Data*). Матрица с N-битным адресом и M-битными данными имеет 2^N строк и M столбцов. Каждая строка данных называется *словом*. Таким образом, матрица содержит 2^N M-битных слов.

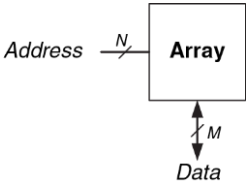


Рис. 5.38 **Условное обозначение обобщенной матрицы памяти**

На **Рис. 5.39** показана матрица памяти, адрес которой состоит из двух бит, а данные – из трех. Два адресных бита выбирают одну из четырех строк (слов данных) матрицы. Ширина каждого слова данных равна трем битам. На **Рис. 5.39 (b)** приведен пример возможного содержимого матрицы памяти. *Глубина* матрицы равна количеству ее строк, а ее *ширина* – количеству столбцов, которое также называется размером слова. Размер матрицы равен произведению количества столбцов на количество строк. На **Рис. 5.39** показана матрица 4-слова × 3-бит, или просто 4 × 3. Обозначение матрицы 1024-слов × 32-бит показано на **Рис. 5.40**. Общий размер этой матрицы равен 32 килобита (Kb).

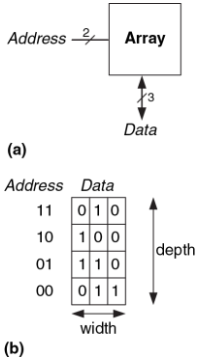


Рис. 5.39 **Матрица памяти 4 × 3: (a) условное обозначение, (b) функция**

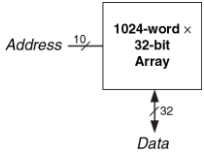


Рис. 5.40 **Матрица 32 Kb: depth = 2^{10} = 1024 words, ширина = 32 бит**

Запоминающие элементы

Матрицы памяти представляют собой набор запоминающих элементов, каждый из которых хранит один бит данных. На **Рис. 5.41** показано, что каждый запоминающий элемент соединен с линией слов (линией выборки слов) и линией битов (линией записи-считывания). При любой

комбинации адресных битов активируется только одна линия выборки слов и, тем самым, разрешается доступ к элементам соответствующей строки. Когда линия выборки слов некоторой строки активна, элементы этой строки могут выдавать данные на линии записи-считывания или принимать данные с этих линий. В противном случае запоминающие элементы отсоединены от линии записи-считывания. Для разных типов памяти схемы запоминающих элементов будут разными.

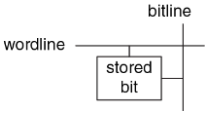


Рис. 5.41 **Запоминающий элемент**

При чтении битов линия записи-считывания вначале находится в отключенном состоянии (Z). Затем включается линия выборки слов, и запоминающие элементы выдают хранимое значение на линию записи-считывания. При записи информации в запоминающий элемент сигнал на линию записи-считывания поступает со специального усилителя записи-считывания, имеющего небольшое выходное сопротивление. Затем включается линия выборки слов, и линии записи-считывания соединяются с запоминающими элементами. Сигнал с линии записи-считывания подавляет содержимое запоминающего элемента, и в нее записывается новая информация.

Организация

На **Рис. 5.42** показана внутренняя организация матрицы памяти 4 × 3. Конечно, реальные запоминающие устройства имеют намного больший объем, но поведение малых матриц памяти может быть экстраполировано на поведение больших. В этом примере матрица хранит данные, которые приведены на **Рис. 5.39 (b)**.

При чтении содержимого памяти активируется линия выборки слов, и с запоминающих элементов соответствующей строки на линии записи-считывания поступает напряжение высокого или низкого логического уровня. При записи на линии записи-считывания с помощью усилителя записи-считывания подаются данные, которые будут сохранены в элементах строки, а затем активируется соответствующая линия выборки слов. Например, для чтения данных по адресу 10 линии записи-считывания остаются в отключенном состоянии, декодер активирует вторую линию выборки слов ($wordline_2$), и данные, которые хранятся в этой строке (100), считываются с линий записи-считывания (*Data*). Для записи значения 001 по адресу 11 на линии записи-считывания с усилителя записи-считывания поступает величина 001, затем активируется третья линия выборки слов ($wordline_3$), и новое значение сохраняется в запоминающих элементах.

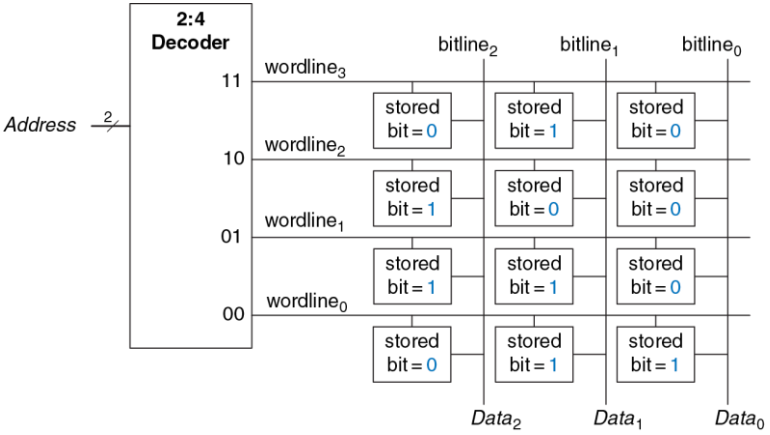


Рис. 5.42 **Матрица памяти 4 × 3**

Порты памяти

Память всех типов имеет один или несколько *портов* (*ports*). Через порты осуществляется доступ к содержимому памяти по некоторому адресу для чтения, записи или чтения-записи. В предыдущем примере была рассмотрена однопортовая память.

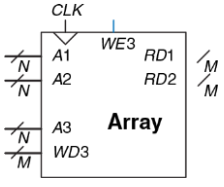


Рис. 5.43 Трехпортовая память

Многопортовая память обеспечивает одновременный доступ к содержимому по нескольким адресам. На **Рис. 5.43** показана трехпортовая память с двумя портами для чтения и одним для записи. Порт 1 считывает данные, которые хранятся по адресу *A*1, и выдает их на выход *RD*1. Порт 2 выдает информацию, хранимую по адресу *A*2, на выход *RD*2. Порт 3 позволяет записать данные, поданные на вход *WD*3, в элемент по адресу *A*3, запись информации осуществляется по переднему фронту тактового импульса при активном сигнале *WE*3.

Типы памяти

Матрицы памяти характеризуются размером (глубины × ширина), количеством и типом портов. Память всех типов хранит данные в матрице запоминающих элементов, но способ хранения битов различный.

Запоминающие устройства классифицируются по способу хранения битов. Запоминающие устройства делятся на два больших класса: оперативные запоминающие устройства (ОЗУ) (RAM, память с произвольным доступом) и постоянные запоминающие устройства (ПЗУ) (ROM, память только для чтения). ОЗУ является

энергозависимым, то есть, при отключении питания информация, которая хранилась в ОЗУ, утрачивается. ПЗУ энергонезависимо, оно сохраняет свои данные даже при отсутствии питания.

Разделение запоминающих устройств на два больших класса – ОЗУ и ПЗУ – возникло на заре компьютерной эры и сейчас устарело и не отражает реальную ситуацию. В ОЗУ время доступа ко всем данным одинаково. Напротив, в запоминающих устройствах с последовательным доступом, таких как память на магнитной ленте, доступ к «ближним» данным происходит намного быстрее, чем доступ к «дальним» (например, тем, которые хранятся на противоположном конце магнитной ленты). Исторически ПЗУ называется постоянным, поскольку данные из такого устройства можно было считывать, но нельзя было записывать в него. Тем, не менее, в современные ПЗУ данные могут быть записаны! Главное отличие, на которое следует обратить внимание, состоит в том, что ОЗУ энергозависимо, а ПЗУ энергонезависимо.

Роберт Деннард, 1932–

Динамическое ОЗУ было изобретено в 1966 году на фирме IBM Робертом Деннардом. Хотя многие относились скептически к принципу работы динамического ОЗУ, с середины 1970-х годов динамическая память используется практически во всех компьютерах. По его утверждению, Деннард мало занимался творческой работой до прихода в IBM, где руководство поручило ему задокументировать свои идеи и оформить на них патенты. После 1965 года он получил 35 патентов в области полупроводниковой техники и микроэлектроники. (Фотография любезно предоставлена IBM).

Основными классами ОЗУ являются динамическое оперативное запоминающее устройство (динамическая память, *DRAM*) и статическое оперативное запоминающее устройство (статическая память, *SRAM*). Динамическая память сохраняет данные в виде заряда конденсаторов, а статическая – в виде состояния бистабильной схемы, состоящей из двух перекрестно соединенных инверторов. Существует много разновидностей ПЗУ, которые отличаются методами записи и считывания информации. Разные типы запоминающих устройств будут рассмотрены в следующих разделах.

5.5.2 Динамическое ОЗУ (DRAM)

В *Динамическом ОЗУ* (*DRAM*) битовым значениям соответствует наличие и отсутствие заряда конденсатора. На **Рис. 5.44** показан запоминающий элемент динамического ОЗУ. Значение бита сохраняется в конденсаторе. *N*-канальный МОП-транзистор (nMOS) является ключом, который может подключить конденсатор к линии записи-считывания или отключить его. Когда линия выборки слов активна, транзистор включается, и хранимые биты передаются на линию записи-считывания или наоборот, происходит запись новой информации в элемент.

Как показано на **Рис. 5.45 (a)**, когда конденсатор заряжен до V_{DD}, хранимый бит равен 1; когда он разряжен до нуля (**Рис. 5.45 (b)**),

хранимый бит равен 0. Узел конденсатора будет динамическим, поскольку он фактически не управляется транзистором, подсоединенным к V_{DD} или GND.

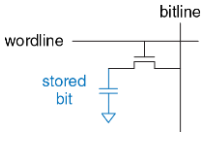


Рис. 5.44 Запоминающий элемент динамического ОЗУ

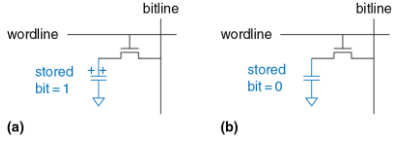


Рис. 5.45 Хранение данных в динамическим ОЗУ

При чтении данные передаются от конденсатора на линию записи-считывания. При записи данные поступают с линии записи-считывания на конденсатор. Чтение уничтожает данные, которые хранились в конденсаторе, поэтому после каждого чтения данные должны быть восстановлены (перезаписаны). Даже если из динамического ОЗУ не нужно считывать данные, из-за саморазряда конденсаторов они должны регенерироваться (считываться и перезаписываться) каждые несколько миллисекунд.

5.5.3 Статическое ОЗУ (SRAM)

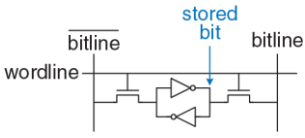


Рис. 5.46 Запоминающий элемент статического ОЗУ

Статическое ОЗУ (*SRAM*) называется статическим, поскольку в нем отсутствует необходимость регенерации хранимых данных. На **Рис. 5.46** показан запоминающий элемент статического ОЗУ. Данные хранятся в бистабильной схеме, состоящей их двух перекрестно соединенных инверторов, подобной тем, которые были рассмотрены в **разделе 3.2**.

Каждая запоминающий элемент имеет два выхода, *bitline* и $\overline{bitline}$. Когда линия выборки слов активна, оба n-канальных МОП-транзистора открываются и данные могут быть записаны в элемент или считаны из него. В отличие от динамического ОЗУ, перекрестно соединенные инверторы возвращают запоминающий элемент в равновесное состояние, если она из него выйдет вследствие случайных отклонений.

5.5.4 Площадь и задержки

Триггеры, статические и динамические ОЗУ являются энергозависимыми запоминающими устройствами, но они различаются временными характеристиками и площадью кристалла, необходимой

для хранения одного бита. В **Табл. 5.4** приведено сравнение этих трех типов энергозависимой памяти. Данные, хранимые в триггере, непосредственно доступны на его выходе. Но схема триггера состоит, по крайней мере, из 20-ти транзисторов. В общем случае, чем больше транзисторов используется в приборе, тем большую площадь он занимает, потребляет больше энергии и стоит дороже. Задержка в динамическом ОЗУ больше, чем в статическом ОЗУ, потому что в нем линия записи-считывания фактически не управляется транзистором. Задержка динамического ОЗУ ограничивается относительно медленной передачей заряда из конденсатора на линию считывания-записи. Из-за необходимости выполнения периодической регенерации и регенерации после чтения динамическое ОЗУ имеет меньшую пропускную способность, чем статическое. Современные разновидности динамического ОЗУ, такие как *синхронное динамическое ОЗУ (SDRAM)* и *синхронное динамическое ОЗУ с удвоенной скоростью обмена (DDR SDRAM* или коротко *DDR)* были разработаны для преодоления этой проблемы. В синхронном динамическом ОЗУ используется тактовый сигнал для конвейеризации доступа к памяти. В синхронном динамическом ОЗУ с удвоенной скоростью обмена передача данных происходит как по переднему, так и по заднему фронту тактового импульса, что удваивает пропускную способность при заданной частоте тактового сигнала. Синхронное динамическое ОЗУ с удвоенной скоростью обмена было впервые стандартизировано в 2000 году и

работало на частотах от 100 до 200 МГц. В более новых стандартах, DDR2, DDR3 и DDR4, тактовая частота была увеличена и к 2012 году она превысила 1 ГГц.

Табл. 5.4 Сравнение типов памяти

Тип памяти	Количество транзисторов в запоминающем элементе	Задержка
Триггер	~20	Малая
Статическое ОЗУ	6	Средняя
Динамическое ОЗУ	1	Большая

Задержка памяти и ее пропускная способность также зависят от размера памяти; при прочих равных условиях память большего объема, как правило, работает медленнее, чем меньшего. Выбор лучшего типа памяти для конкретного проекта зависит от требований к быстродействию, цене и энергопотреблению.

5.5.5 Регистровые файлы

Цифровые системы часто используют несколько регистров для хранения временных переменных. Такие группы регистров, которые называются *регистровыми файлами*, обычно реализуются в виде небольших многопортовых матриц статического ОЗУ, поскольку они более компактны, чем матрицы триггеров.

На **Рис. 5.47** показан трехпортовый регистровый файл, состоящий из 32 регистров по 32 бита каждый, который построен на основе трехпортовой памяти, подобной показанной на **Рис. 5.44**. Регистровый файл имеет два порта для чтения (*A1/RD1* и *A2/RD2*) и один порт для записи (*A3/WD3*). Пятиразрядные адреса *A1*, *A2*, и *A3* обеспечивают доступ к любому из 2^5 = 32 регистров. Таким образом, одновременно можно записывать информацию в два регистра и считывать из одного.

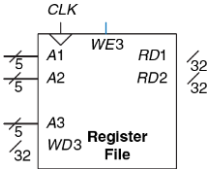


Рис. 5.47 **Регистровый файл 32 × 32 register с двумя портами чтения и одним портом записи**

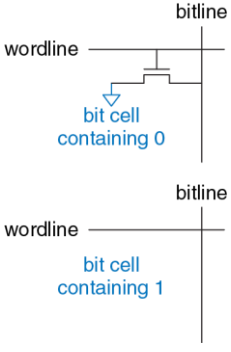
5.5.6 Постоянное Запоминающее Устройство

В *постоянном запоминающем устройстве* (*ПЗУ*, *ROM*) хранимым битовым значениям соответствует наличие или отсутствие транзистора. На **Рис. 5.48** показан простой запоминающий элемент ПЗУ. При чтении информации из элемента на линию записи-считывания от внешнего источника подается уровень слабой логической 1. Затем активируется

линия выборки слов. Если в элементе есть транзистор, он открывается и устанавливает на линии записи-считывания уровень логического 0. Когда транзистор отсутствует, на линии записи-считывания остается уровень логической 1. Обратите внимание на то, что ПЗУ является комбинационной схемой и не имеет состояния, которое может быть потеряно при отключении питания.

Содержимое ПЗУ может быть показано с помощью точечной нотации. На **Рис. 5.49** приведена точечная нотация для ПЗУ 4-слова × 3-бит, которая содержит данные **Рис. 5.39**. Наличие точки на пересечении строки (линии выборки слов) и столбца (линии записи-считывания) показывает, что хранимый бит равен 1. Например, на верхней линии выборки слов есть только одна точка на ее пересечении с $Data_1$, следовательно, по адресу 11 хранится значение 010.

Концептуально ПЗУ может быть построено с использованием двухуровневой логики, состоящей из группы логических элементов И, за которой следует группа элементов ИЛИ. Элементы И порождают все возможные минтермы и, следовательно, формируют декодер. На **Рис. 5.50** показано ПЗУ **Рис. 5.49**, построенное с использованием декодера и элементов ИЛИ. Каждая точка на **Рис. 5.49** соответствует соединению строки и входа элемента ИЛИ на **Рис. 5.50**.



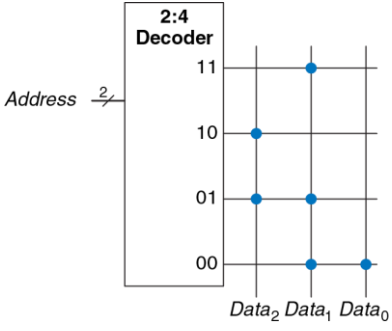


Рис. 5.48 **Запоминающие элементы ПЗУ, содержащие 0 и 1**

Рис. 5.49 **ПЗУ 4 × 3: точечная нотация**

Для выходных битов данных с одной точкой, таких как $Data_0$, элемент ИЛИ не нужен. Такое представление ПЗУ показывает, что с помощью ПЗУ можно реализовать произвольную двухуровневую логическую функцию. Реальные ПЗУ состоят из транзисторов, а не логических элементов, что позволяет уменьшить их размер и стоимость. В **разделе 5.6.3** реализация ПЗУ на уровне транзисторов будет рассмотрена детально.

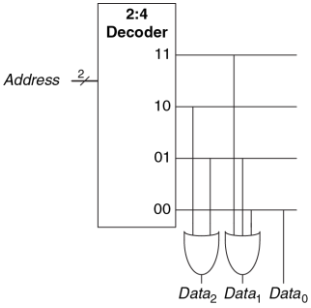


Рис. 5.50 **Реализация ПЗУ 4 × 3 с использованием логических элементов**

Содержимое запоминающих элементов ПЗУ, которое показано на **Рис. 5.48**, определяется при его изготовлении наличием или отсутствием транзистора в каждой ячейке. В *программируемом ПЗУ* (*PROM* , *ППЗУ*) транзисторы размещены во всех элементах, но в них есть возможность управлять соединением этих транзисторов с землей.

На **Рис. 5.51** показан запоминающий элемент *ПЗУ, программируемого плавкими перемычками,* (*fuse-programmable ROM*). Пользователь может программировать ПЗУ, подавая высокое напряжение на некоторые перемычки и, тем самым, пережигая их. Если перемычка присутствует, то транзистор соединен с землей, и элемент хранит 0. Если перемычка

разрушена, то транзистор отсоединен от земли и элемент хранит 1. Такое ПЗУ также называют однократно программируемым ПЗУ, поскольку после пережигания перемычки ее невозможно восстановить.

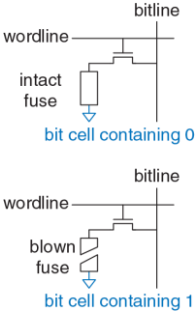


Рис. 5.51 **Запоминающий элемент ПЗУ, программируемого перемычками**

В перепрограммируемых ПЗУ реализован механизм обратимого соединения-разъединения транзисторов с землей. В стираемых программируемых ПЗУ (*СППЗУ*, *Erasable PROMs*, *EPROM*) n-МОП-транзисторы и перемычки заменены *транзисторами с плавающим затвором* (*floating-gate transistor*). Плавающий затвор не соединен физически ни с какими другими проводниками. Когда на транзистор

подается достаточно высокое напряжение, электроны туннелируют через изолятор на плавающий затвор, транзистор включается и соединяет линию выборки слов и линию битов (выход декодера). Когда *СППЗУ* облучают ультрафиолетовым излучением в течение примерно получаса, электроны выбрасываются с плавающего затвора, и транзистор выключается. Эти действия называются *программированием* и *стиранием*, соответственно. В *электрически стираемом программируемом ПЗУ (ЭСППЗУ*, *electrically erasable PROM, EEPROM*) и *флэш-памяти* (*flash memory*) используется аналогичный принцип, однако ультрафиолетовое излучение не используется, поскольку на кристалле присутствует специальная схема стирания. В ЭСППЗУ запоминающие элементы можно стирать индивидуально, в флэш-памяти стирание происходит большими блоками, она дешевле, поскольку в ней используется меньшее количество стирающих схем. В 2012 году стоимость флэш-памяти составляла примерно $1 за 1 Гб, и она продолжала падать примерно на 30 – 40% за год. Флэш-память стала очень популярной для сохранения больших объемов данных в переносных устройствах с батарейным питанием, таких как камеры и музыкальные проигрыватели.

Таким образом, современные ПЗУ не являются постоянными в строгом значении слова: они могут программироваться, т.е. информация в них может записываться. Различие между ОЗУ и ПЗУ состоит в том, что

запись в ПЗУ требует больше времени, и они являются энергонезависимыми.

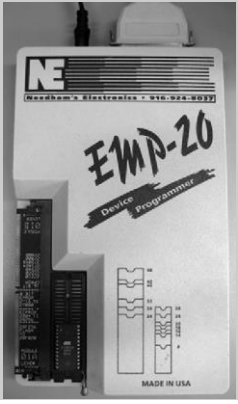
Фуджио Масуока, 1944–

Получил степень Ph.D. в области электротехники в университете Тохоку, Япония. Занимался разработкой запоминающих устройств и быстродействующих схем на фирме Toshiba с 1971 по 1995 год. Изобрел флэш-память в конце 1970-х годов при выполнении самостоятельного любительского проекта по ночам и выходным. Флэш-память получила свое имя из-за того, что процесс стирания памяти напоминает работу вспышки (flash) камеры. Toshiba запоздала с коммерческой реализацией идеи флэш-памяти; первенство принадлежит фирме Intel, которая предложила коммерческие изделия в 1988 году. Рынок флэш-памяти растет на $25 миллиардов за год.

Доктор Масуока позже присоединился к факультету университета Тохоку и работает над созданием трехмерного транзистора.

Из-за быстрого падения цены накопители на основе флэш-памяти с разъемом USB заменили компакт-диски и дискеты.

Программируемые ПЗУ можно конфигурировать с помощью специального прибора – программатора, подобного показанному ниже. Прибор подсоединяется к компьютеру, который задает тип ПЗУ и данные, которые должны быть запрограммированы. Программатор пережигает перемычки или инжектирует заряд в плавающие затворы ПЗУ. Процесс программирования иногда называют прожиганием ПЗУ.



5.5.7 Реализация логических функций с использованием матриц памяти

Хотя основным применением матриц памяти является хранение данных, они также могут использоваться для реализации комбинационных логических функций. Например, выход $Data_2$ ПЗУ, которое показано на **Рис. 5.49**, представляет собой функцию XOR двух входов *Address*.

Аналогично, $Data_0$ есть функция NAND двух входов. Матрица памяти размерностью 2^N-слов × *M*-бит может реализовать произвольную логическую функцию с *N*-входами и *M*-выходами. Например, ПЗУ на **Рис. 5.49** реализует три функции двух аргументов.

Матрицы памяти, которые реализуют логические функции, называются таблицами преобразований (*lookup tables*, *LUT*). На **Рис. 5.52** показана матрица памяти 4-слова × 1-бит, которая используется как таблица преобразования для реализации функции $Y = AB$. При использовании памяти для выполнения логической функции для заданной комбинации входов (адреса) в ней происходит поиск соответствующего значения выхода. Каждый адрес соответствует строке в таблице истинности, а каждый хранимый бит – значению выходного сигнала.

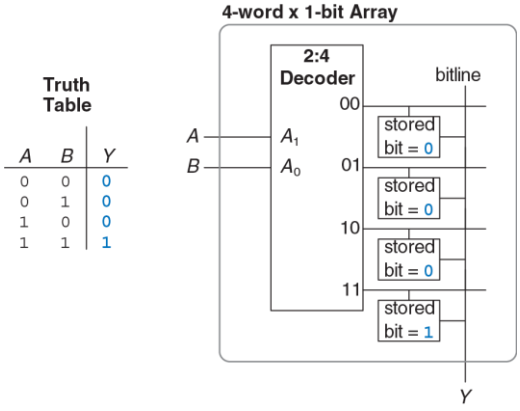


Рис. 5.52 **Матрица 4-слова × 1-бит с использованием таблицы преобразования**

5.5.8 Языки описания аппаратуры и память

В **примере 5.7 HDL** на языках HDL описано ОЗУ размерностью 2^N-слов × M-бит. У этого ОЗУ есть синхронный вход разрешения записи. Другими словами, запись в память происходит по переднему фронту тактового импульса, если сигнал разрешения записи (write enable) *we*

находится в активном состоянии. Чтение происходит немедленно. Непосредственно после включения питания содержимое ОЗУ непредсказуемо.

Пример 5.7 HDL ОЗУ

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);
  logic [M-1:0] mem [2**N-1:0];
  always_ff @(posedge clk)
    if (we) mem [adr] <= din;
  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;
entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
       we:   in  STD_LOGIC;
       adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
       din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
```

```
       dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;
architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
   if rising_edge(clk) then
    if we then mem(TO_INTEGER(adr)) <= din;
    end if;
   end if;
  end process;
  dout <= mem(TO_INTEGER(adr));
end;
```

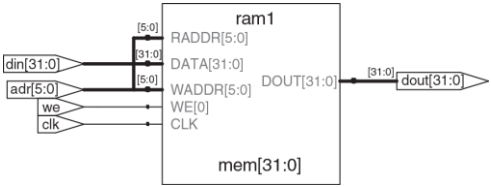


Рис. 5.53 **Synthesized ram**

В **примере 5.8 HDL** на языках HDL описано ПЗУ размерностью 4-слова × 3-бит. Содержимое ПЗУ задается в операторе case. ПЗУ насколько мало, что может быть синтезировано в виде набора логических элементов, а не матрицы. Обратите внимание на то, что декодер семисегментного кода из **примера 4.24** на языке HDL был синтезирован в виде ПЗУ на **Рис. 4.20**.

Пример 5.8 HDL ПЗУ

SystemVerilog

```
module rom(input logic  [1:0] adr,
           output logic [2:0] dout):
  always_comb
    case(adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;
architecture synth of rom is
begin
  process(all) begin
    case adr is
      when “00” => dout <= “011”;
      when “01” => dout <= “110”;
      when “10” => dout <= “100”;
      when “11” => dout <= “010”;
    end case;
  end process;
end;
```

5.6 МАТРИЦЫ ЛОГИЧЕСКИХ ЭЛЕМЕНТОВ

Логические элементы, как и запоминающие элементы, могут быть организованы в регулярные матрицы. Если соединения между логическими элементами программируемы, такие матрицы можно сконфигурировать для реализации произвольной логической функции, причем при этом не надо будет изменять соединения между микросхемами на плате. Регулярная структура упрощает проектирование. Матрицы логических элементов производятся в больших количествах, что обеспечивает их малую стоимость. Существует программное обеспечение, позволяющее перенести проекты цифровых устройств в такие матрицы. Большинство матриц логических элементов реконфигурируемо, что позволяет изменить проект без замены аппаратного обеспечения. Реконфигурируемость очень ценна при разработке и полезна при эксплуатации изделия, поскольку оно может быть обновлено путем простой загрузки новой конфигурации.

В этом разделе будут рассмотрены два типа матриц логических элементов: программируемая логическая матрица (programmable logic arrays, ПЛМ, PLA) и программируемая пользователем матрица логических элементов (field programmable gate arrays, ППМЛЭ, FPGA). В программируемой логической матрице (ПЛМ, PLA), которая представляет собой более старую технологию, можно реализовать

только комбинационные логические функции. Программируемая пользователем матрица логических элементов (FPGA) позволяет создавать как комбинационные, так и последовательностные схемы.

5.6.1 Программируемые логические матрицы

Программируемые логические матрицы (*ПЛМ*, *PLA*) позволяют реализовать двухуровневые комбинационные логические схемы, заданные совершенной дизъюнктивной нормальной формой (СДНФ). На **Рис. 5.54** показано, что ПЛМ состоит из матрицы И, за которой следует матрица ИЛМ. Входы (в прямой и инверсной форме) поступают на матрицу И, которая создает импликанты, которые, в свою очередь, объединяются функциями ИЛИ и формируют выходной сигнал матрицы. ПЛМ размерности $M \times N \times P$-бит имеет M входов, N импликантов и P выходов.

На **Рис. 5.55** приведена точечная нотация ПЛМ 3 × 3 × 2-бит, которая реализовывает функции $X = \overline{A}\,\overline{B}\,C + AB\overline{C}$ и $Y = A\overline{B}$. Каждая строка в матрице И формирует импликант. Точки в строках матрицы И показывают, какие литералы формируют импликант.

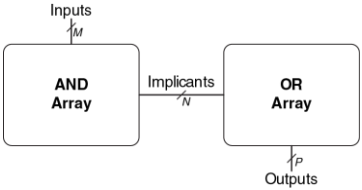


Рис. 5.54 Программируемая логическая матрица (PLA) *M* × *N* × *P*-бит

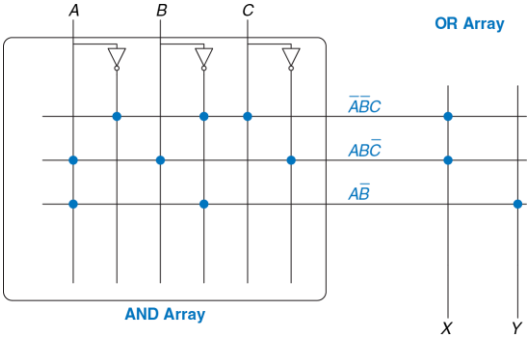


Рис. 5.55 Программируемая логическая матрица (PLA) 3 × 3 × 2-бит: точечная нотация

Матрица И на **Рис. 5.55** формирует три импликанта: $\overline{A}\overline{B}C + AB\overline{C}$ и $A\overline{B}$. Точки в матрице ИЛИ показывают, какие импликанты входят в выходную функцию.

На **Рис. 5.56** проиллюстрировано, как ПЛМ может быть построена с использованием двухуровневой логики. Альтернативная реализация будет рассмотрена в **разделе 5.6.3**.

ПЗУ можно рассматривать как разновидность ПЛМ. ПЗУ с организацией 2^M-слов × N-бит представляет собой ПЛМ-размерности M × 2^M × N-бит. Декодер выполняет функции матрицы И и создает все 2^M минтермов. Массив запоминающих элементов выполняет функции матрицы ИЛИ и определяет выходные сигналы. Если функция зависит не от всех 2^M минтермов, то, весьма вероятно, реализация с ПЛМ будет более компактной, чем с ПЗУ. Например, для выполнения функций ПЛМ размерности 3 × 3 × 2-бит, которая показана на **Рис. 5.55** и **Рис. 5.56**, потребуется ПЗУ 8-слов × 2-бит.

В *простых программируемых логических устройствах* (*ППЛУ, SPLD*) базовые матрицы И и ИЛИ ПЛМ дополнены регистрами и дополнительными схемами. Однако в настоящее время ППЛУ и ПЛМ в основном вытеснены программируемыми пользователем матрицами логических элементов (ППМЛЭ, FPGA), которые более гибки и эффективны при создании больших систем.

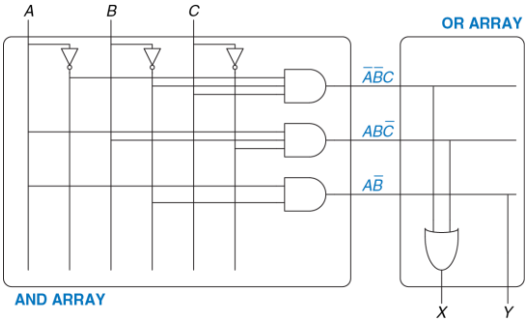


Рис. 5.56 **Реализация программируемой логической матрицы (PLA) 3 × 3 × 2-бит с использованием двухуровневой логики**

5.6.2 Программируемые пользователем матрицы логических элементов

Программируемы пользователем матрицы логических элементов (*ППМЛЭ*, *FPGA*) представляют собой матрицу реконфигурируемых элементов. С использованием специального программного обеспечения пользователь может описать свой проект на языке описания аппаратуры или в виде схемы, а затем реализовать его в FPGA. В ряде отношений матрицы FPGA мощнее и гибче, чем ПЛМ.

В FPGA возможно реализовать как комбинационные, так и последовательностные схемы. В них можно реализовывать многоуровневые логические схемы, тогда как в ПЛМ могут быть реализованы только двухуровневые схемы. В современные FPGA интегрированы другие полезные узлы, такие как умножители, высокоскоростные устройства ввода/вывода, ЦАП, АЦП, большие ОЗУ и процессоры.

FPGA используются во многих потребительских продуктах, таких как автомобили, медицинское оборудование, устройства обработки медиа-информации. Например, в системах навигации, круиз-контроля, звуковоспроизведения автомобилей Mercedes Benz S-класса используется более десяти FPGA и PLD фирмы Xilinx. FPGA позволяют быстрее выводить изделия на рынок и упрощают отладку и добавление новых возможностей на поздних этапах жизненного цикла продукта.

FPGA представляет собой матрицу конфигурируемых *логических элементов* (*logic elements*, *ЛЭ*, *LE*), которые также называются *конфигурируемыми логическими блоками* (*configurable logic blocks*, *КЛБ*,*CLB*). Каждый ЛЭ можно сконфигурировать для выполнения функций некторой комбинационной или последовательностной схемы. На **Рис. 5.57** приведена обобщенная структура FPGA. ЛЭ окружены *элементами ввода/вывода* (*input/output elements*, *IOE, ЭВВ*), которые предназначены для организации обмена информацией между FPGA и прочими компонентами системы. Элементы ввода/вывода соединяют входы и выходы логических элементов с контактами корпуса микросхемы. Логические элементы могут быть соединены между собой и с элементами ввода/вывода с помощью программируемых каналов трассировки.

Лидерами на рынке FPGA являются фирмы Altera Corp. и Xilinx, Inc. На **Рис. 5.58** показан один логический элемент FPGA фирмы Altera Cyclone IV, производство которой началось в 2009 году. Основными компонентами логического элемента является четырехвходовая таблица преобразования (LUT) и однобитовый регистр. Логический элемент также содержит конфигурируемые мультиплексоры, предназначенные для коммутации сигналов в логическом элементе. При программировании FPGA устанавливается содержимое таблиц

преобразования (LUT) и определяются входные сигналы мультиплексоров, которые проходят на их выходы.

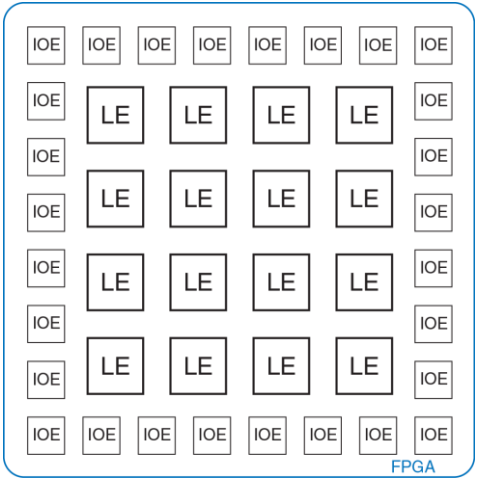


Рис. 5.57 **Обобщенная структура FPGA**

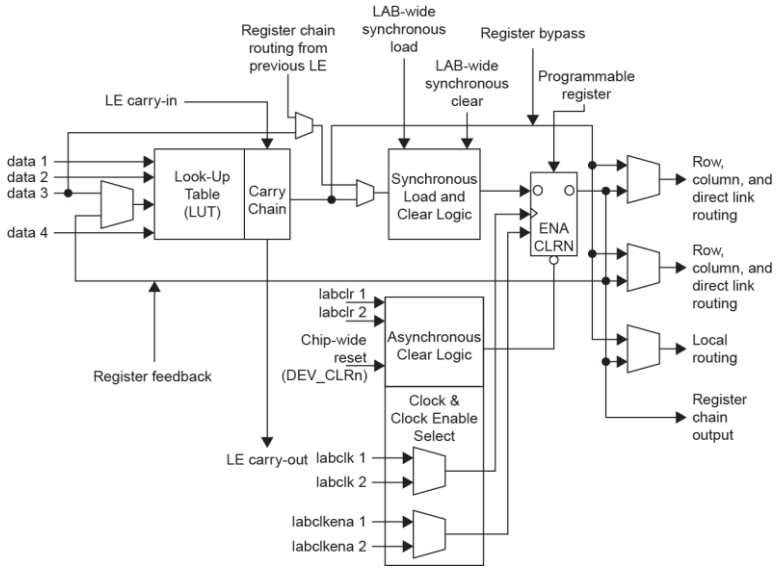


Рис. 5.58 **Cyclone IV Logic Element (LE)**
(Воспроизведено с разрешения из Altera Cyclone™ IV Handbook © 2010 Altera Corporation.)

Логический элемент FPGA Cyclone IV содержит одну четырехвходовую таблицу преобразования (LUT) и один триггер. Путем загрузки соответствующих значений в LUT, она может быть сконфигурирована для реализации произвольной логической функции четырех (или менее) аргументов. Также при конфигурировании FPGA сигналам выбора, которые определяют, как мультиплексоры будут коммутировать каналы передачи данных в пределах логического элемента (LE) и между ним и соседними логическими элементами (LE) или элементами ввода/вывода (IOE), присваиваются необходимые значения. Например, в зависимости от конфигурации мультиплексора на один из входов LUT некоторого LE может поступать сигнал или с его входа *data 3* или с выхода регистра этого же LE. На остальные три входа LUT сигналы всегда поступают со входов LE *data 1*, *data 2* и *data 4*. В зависимости от трассировки внешних соединений сигнал на входы *data 1-4* поступает с IOE или выходов других LE. Выход LUT может поступать либо непосредственно на выход LE при реализации комбинационной логической схемы, либо через триггер при создании последовательностной схемы. Сигнал на вход триггера может поступать с выхода LUT этого же LE, входа *data 3* или с выхода регистра предыдущего LE. Кроме того, в LE входит ряд вспомогательных схем: дополнительные мультиплексоры для трассировки, схемы управления сигналами разрешения и сброса триггера, схемы, позволяющие реализовать сумматор с

последовательным переносом. В FPGA фирмы Altera группы из 16 LE объедены в блок логических матриц (*logic array block*, *LAB*), для передачи данных между LE одного блока существуют специальные локальные соединения.

Таким образом, в LE FPGA Cyclone IV можно реализовать одну функцию четырех (или менее) входов, причем она может быть комбинационной или последовательностной, то есть иметь на выходе триггер. FPGA других производителей организованы немного по-другому, но принцип построения остается общим. Например, в FPGA фирмы Xilinx седьмой серии вместо четырехвходовой LUT используется шестивходовая.

При разработке конфигурации FPGA проектировщик вначале создает схемное описание проекта или описание на HDL. Затем происходит синтез проекта. Пакет синтеза схем определяет, как следует сконфигурировать LUT, мультиплексоры и каналы трассировки для реализации заданных функций. Эта конфигурационная информация загружается в FPGA. Так как FPGA Cyclone IV сохраняют конфигурационную информацию в статическом ОЗУ, они могут быть легко перепрограммированы. Содержимое статического ОЗУ FPGA может быть загружено с компьютера (в лабораторных условиях) или при включении питания из специальной микросхемы ЭСППЗУ (EEPROM). Некоторые производители встраивают ЭСППЗУ

непосредственно в микросхему FPGA или используют для конфигурирования FPGA однократнопрограммируемые перемычки.

Пример 5.6 ПОСТРОЕНИЕ ФУНКЦИЙ С ИСПОЛЬЗОВАНИЕМ ЛОГИЧЕСКИХ ЭЛЕМЕНТОВ

Поясните, как следует сконфигурировать один или несколько логических элементов (LE) FPGA Cyclone IV для реализации следующих функций:

(a) $X = \overline{A}\,\overline{B}\,C + AB\overline{C}$ и $Y = A\overline{B}$ (b) $Y = JKLMPQR$; (c) счетчик по основанию 3 с двоичной кодировкой состояния (см. **Рис. 3.29 (a)**). При необходимости вы можете показать связи между логическими элементами.

Решение: (a) Для реализации функций следует сконфигурировать два логических элемента. Как показано на **Рис. 5.59**, первая таблица преобразования (LUT) вычисляет *X*, вторая – *Y*. На входы data 1, data 2 и data 3 первой таблицы преобразования подаются сигналы *A*, *B* и *C* (эти соединения устанавливаются трассировочными каналами), вход data 4 не используется, но на него нужно подать какое-либо значение, например 0. Во второй таблице преобразования на входы data 1 и data 2 подаются сигналы *A* и *B*; остальные входы не используются, и на них подан 0. Выходной мультиплексор сконфигурирован для подачи на выход комбинационного сигнала с таблиц преобразования, таким образом на выходе формируются требуемые сигналы *X* и *Y*. В общем случае, один логический элемент позволяет вычислить произвольную функцию четырех (или менее) аргументов.

(b) Таблица преобразования (LUT) первого логического элемента (LE) должна быть сконфигурирована для вычисления $X = JKLM$, а второго – $Y = XPQR$.

Выходные мультиплексоры должны выбирать комбинационные выходы *X* и *Y* каждого логического элемента (LE). Эта конфигурация показана на **Рис. 5.60**. Трассировочные каналы между логическими элементами (LE), которые показаны синими пунктирными линиями, соединяют выход первого логического элемента со входом второго. В общем случае, группа логических элементов (LE) позволяет вычислить аналогичным образом функцию *N*-входных переменных.

(c) Конечный автомат имеет два бита для хранения состояния ($S_{1:0}$) и один выход (*Y*). Следующее состояние зависит от двух битов текущего состояния. Как показано на **Рис. 5.61**, для определения следующего состояния по текущему используется два логических элемента (LE). Два триггера, по одному из каждого логического элемента (LE), хранят это состояние. У триггеров есть вход сброса, который может быть соединен с внешним сигналом Reset. Синими пунктирными линиями показан тракт подачи сигнала через трассировочные каналы и мультиплексоры на входах data 3 с выходных регистров назад на входы таблиц преобразования (LUT). В общем случае для вычисления выхода *Y* может понадобиться дополнительный логический элемент (LE). Однако, в данном случае $Y = S_0$, то есть *Y* поступает с выхода первого логического элемента (LE). Таким образом, весь конечный автомат реализован на двух логических элементах (LE). В общем случае, для реализации конечного автомата необходимо по крайней мере по одному логическому элементу (LE) для каждого бита состояния; если логика определения выхода или следующего состояния слишком сложна для одной таблицы преобразования (LUT), то могут потребоваться дополнительные логические элементы (LE).

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

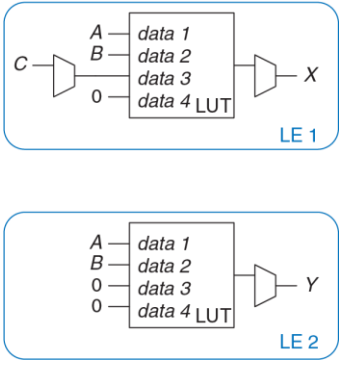


Рис. 5.59 Конфигурация логического элемента (LE) для реализации двух функций, имеющих до четырех входов

(*J*) *data 1*	(*K*) *data 2*	(*L*) *data 3*	(*M*) *data 4*	(*X*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*P*) *data 1*	(*Q*) *data 2*	(*R*) *data 3*	(*X*) *data 4*	(*Y*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

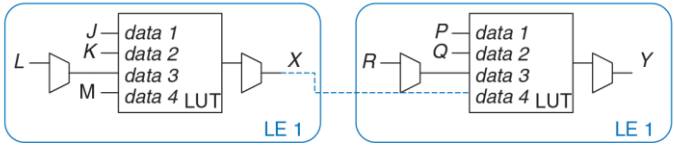


Рис. 5.60 **Конфигурация логических элементов (LE) для реализации одной функции, имеющей более четырех входов**

data 1	data 2	(S_0) data 3	(S_1) data 4	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	data 2	(S_1) data 3	(S_0) data 4	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0

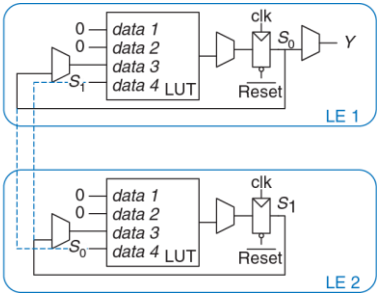


Рис. 5.61 **Конфигурация логических элементов (LE) для реализации конечного автомата с двухбитовым состоянием**

Пример 5.7 ЗАДЕРЖКА В ЛОГИЧЕСКОМ ЭЛЕМЕНТЕ

Алиса разрабатывает конечный автомат, который должен работать на частоте 200 МГц. Она использует FPGA Cyclone IV GX со следующими характеристиками: t_{LE} = 381 пс на LE, t_{setup} = 76 пс и t_{pcq} = 199 пс для всех триггеров. Задержка в соединении между LE равна 246 пс. Время удержания триггеров положим равным 0. Какое максимальное количество LE можно использовать в ее проекте?

Решение: Для определения максимальной задержки распространения в комбинационной логической схеме Алиса использует **неравенство (3.13)**:

$t_{pd} \le T_c - (t_{pcq} + t_{\text{setup}})$.

Таким образом, t_{pd} = 5 нс −(0.199 нс + 0.076 нс), то есть $t_{pd} \le 4.725$ нс. Задержка в каждом логическом элементе (LE) в сумме с задержкой в соединении логических элементов ($t_{LE+wire}$) равна 381 пс + 246 пс = 627 пс. Максимальное количество (N) логических элементов (LE) можно определить из условия $Nt_{LE+wire} \le 4.725$ нс. Таком образом, N = 7

5.6.3 Схемотехника матриц

Для минимизации размеров и цены в ПЗУ и ПЛМ вместо традиционных логических элементов часто используются псевдо-n-МОП (pseudo-nMOS) или динамические (см. **раздел 1.7.8**) схемы.

Во многих ПЗУ и ПЛМ вместо псевдо-n-МОП (pseudo-nMOS) используются динамические схемы. В динамических элементах p-МОП-транзистор включен не все время, что позволяет снижать энергопотребление, когда его состояние не имеет значения и он выключен. Во всех прочих отношениях (при проектировании и использовании) динамические и псевдо-n-МОП-матрицы памяти аналогичны.

На **Рис. 5.62 (a)** представлена точечная нотация для ПЗУ 4 × 3-бит, которое реализует следующие функции: $X = A \oplus B$, $Y = \overline{A} + B$ и $Z = \overline{A}\,\overline{B}$. Это те же функции, которые были представлены на **Рис. 5.49**, причем адресные входы были переобозначены как *A* и *B*, а выходы – *X*, *Y* и *Z*. Реализация с псевдо-n-МОП-элементами показана на **Рис. 5.62 (b)**. Выход каждого декодера соединен с затворами n-МОП-транзисторов его строки. Как известно, в псевдо-n-МОП-схемах выход связан с цепью питания p-МОП-транзистором с большим сопротивлением канала. Выход имеет высокий потенциал, только если n-МОП-транзистор, который связывает его с землей, закрыт. Эти транзисторы расположены на всех пересечениях, где точка *отсутствует*. Для сравнения на **Рис. 5.62 (b)** сохранены точки точечной нотации, которая была показана на **Рис. 5.62 (a)**. p-МОП-транзисторы устанавливают высокий логический уровень всех линий слов, на которых n-МОП-транзисторы отсутствуют. Например, когда *AB* = 11, линия слов 11 имеет высокий потенциал, соединенные с ней n-МОП-транзисторы открываются и на выходах *X* и *Z* устанавливают низкое напряжение. На пересечении линии выход *Y* и линии 11 n-МОП-транзистор отсутствует, следовательно, на этом выходе сохраняется высокое напряжение.

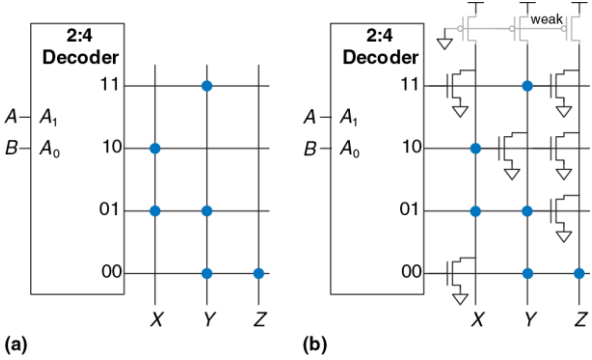


Рис. 5.62 **Реализация ПЗУ: (a) точечная нотация, (b) псевдо-n-МОП-схема**

ПЛМ также могут быть реализованы с использованием псевдо-n-МОП-схем. На **Рис. 5.63** показана такая реализация ПЛМ, которая была изображена на **Рис. 5.55**. n-МОП-транзисторы, обеспечивающие низкий уровень, расположены на *неотмеченных точками* пересечениях матрицы И и в отмеченных строках матрицы ИЛИ. Столбцы матрицы ИЛИ поступают на выход через инверторы. Для сравнения синие точки с точечной нотации (см. **Рис. 5.55**) показаны на **Рис. 5.63**.

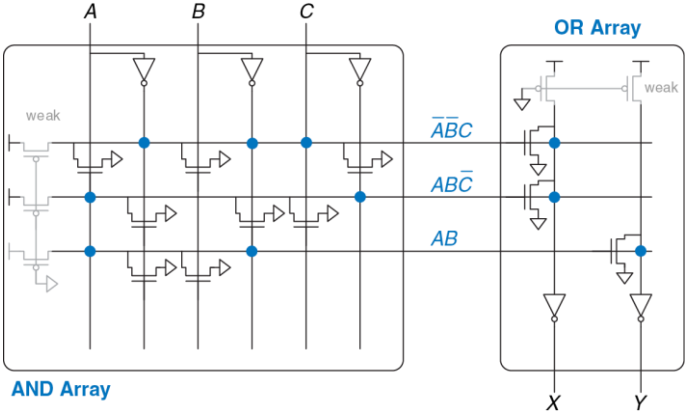


Рис. 5.63 **Реализация ПЛМ 3 × 3 × 2-бит с использованием псевдо-n-МОП-схем**

5.7 РЕЗЮМЕ

В этой главе были рассмотрены функциональные узлы, которые используются во многих цифровых системах. В число таких функциональных узлов входят арифметические схемы: сумматоры, блоки вычитания, умножители, делители, схемы сдвига, последовательностные схемы: счетчики, сдвиговые регистры, логические матрицы и запоминающие устройства. В этой главе также были рассмотрены представления дробных чисел с плавающей и фиксированной точкой. В **главе 7** эти функциональные узлы будут использоваться для построения микропроцессора.

Большое количество арифметических схем строятся с использованием сумматоров. Полусумматор имеет два однобитовых входа *A* и *B* и два выхода – сумма и перенос. В полном сумматоре ко входам полусумматора добавляется вход переноса. *N* полных сумматоров можно соединить последовательно и, тем самым, создать параллельный сумматор, который складывает два *N*-битовых числа. Такой сумматор также называют сумматором с последовательным переносом. Более быстрые параллельные сумматоры можно создать с использованием технологий группового ускоренного и префиксного переноса.

В блоке вычитания знак второго операнда инвертируется, а затем выполняется операция сложения. Схема сравнения вычитает одно число из другого, а результат сравнения определяется по знаку разницы. В умножителе элементы И формируют частичные произведения, а затем они складываются с помощью полных сумматоров. В схеме деления делитель многократно вычитается из частичного остатка, и по знаку разницы определяются двоичные разряды частного. В счетчике для хранения состояния используется регистр, а для его увеличения – сумматор.

Дробные числа представляются в формах с плавающей или с фиксированной точкой. Представление с фиксированной точкой аналогично десятичному, а с плавающей – экспоненциальному. Для обработки чисел с фиксированной точкой используются обычные арифметические схемы, а числа с плавающей точкой требуют использования более сложных схем, которые выделяют и обрабатывают знак, порядок и мантиссу.

Запоминающие устройства большого объема организованы в виде матрицы слов. Запоминающие устройства имеют один или более портов для чтения и/или записи слов. Содержимое энергозависимой памяти, такой как статическое или динамическое ОЗУ, утрачивается при выключении питания схемы. Статическое ОЗУ быстрее, чем динамическое, но использует больше транзисторов. Регистровый файл

представляет собой небольшое многопортовое статическое ОЗУ. Содержимое энергонезависимой памяти, которая называется постоянным запоминающим устройством (ПЗУ), сохраняется неограниченно долго при отсутствии питания. Несмотря на название, содержимое большинства современных ПЗУ может быть изменено.

Логические элементы также могут быть организованы в виде матриц. Для выполнения функций комбинационной логики могут использоваться матрицы памяти, в которых хранится таблица преобразования. ПЛМ состоит из соединенных между собой конфигурируемых матриц И и ИЛИ, в ПЛМ могут быть реализованы только комбинационные схемы. FPGA содержит большое количество небольших таблиц преобразования и регистров и позволяет реализовывать как комбинационные, так и последовательностные схемы. Содержимое таблиц преобразования и их межсоединение может быть сконфигурировано для выполнения любой логической функции. Современные FPGA могут быть легко перепрограммированы, содержат большое количество конфигурируемых логических элементов, весьма дешевы, что позволяет на их основе создавать сложные цифровые системы. Они широко используются как в коммерческих мало- и среднесерийных изделиях, так и в образовательных проектах.

УПРАЖНЕНИЯ

Упражнение 5.1 Чему будет равна задержка следующих 64-разрядных сумматоров? Задержка любого двухвходового логического элемента равна 150 пс, а полного сумматора – 450 пс.

a) сумматор с последовательным переносом

b) сумматор с ускоренным переносом, состоящий из 4-битовых блоков

c) префиксный сумматор

Упражнение 5.2 Спроектируйте два сумматора с распространяющимся переносом: 64-хразрядный сумматор с последовательным переносом и 64-разрядный сумматор с ускоренным переносом, состоящий из 4-битовых блоков. Используйте только двухвходовые логические элементы. Каждый такой элемент имеет площадь 15 мкм^2, задержку 50 пс и полную емкость 20 фФ. Статической мощностью можно пренебречь.

a) Сравните площадь, задержку и потребляемую мощность сумматоров, работающих на частоте 100 МГц при напряжении питания 1.2 В.

b) Обсудите компромисс между мощностью, площадью и задержкой.

Упражнение 5.3 Объясните, почему проектировщик может использовать сумматор с последовательным переносом, а не сумматор с ускоренным переносом.

Упражнение 5.4 Спроектируйте 16-разрядный префиксный сумматор, показанный на **Рис. 5.7**, с использованием языков описания аппаратуры. Промоделируйте и протестируйте свой модуль и покажите, что он работает корректно.

Упражнение 5.5 В префиксной сети, показанной на **Рис. 5.7**, для вычисления всех префиксов используются черные ячейки. Сигналы распространения некоторых блоков на самом деле не нужны. Спроектируйте «серую ячейку», которая получает сигналы G и P для битов $i{:}k$ и $k{-}1{:}j$, но вычисляет только $G_{i:j}$, а не $P_{i:j}$. Перерисуйте префиксную сеть, в которой везде, где возможно, черные ячейки будут заменены на серые.

Упражнение 5.6 Префиксная сеть, показанная на **Рис. 5.7**, – не единственный способ вычисления всех префиксов с логарифмической задержкой. Сеть Когге-Стоуна является другой распространенной префиксной сетью, которая выполняет те же функции с использованием иного соединения черных ячеек. Исследуйте сумматор Когге-Стоуна и нарисуйте схему, подобную показанной на **Рис. 5.7**, на которой черные ячейки будут формировать сумматор Когге-Стоуна.

Упражнение 5.7 Вспомните, что *N*-входовый приоритетный шифратор имеет $\log_2 N$ выходов, на которых формируется двоичное число, соответствующее номеру самого старшего входа, на который подана логическая 1 (см. **упражнение 2.36**).

a) Спроектируйте *N*-входовый приоритетный шифратор у которого задержка увеличивается логарифмически с ростом *N*. Нарисуйте схему шифратора и рассчитайте его задержку, исходя из задержек отдельных логических элементов.

b) Опишете ваш проект на языке описания аппаратуры. Промоделируйте и протестируйте свой модуль и покажите, что он работает корректно.

Упражнение 5.8 Спроектируйте следующие компараторы 32-разрядных чисел. Нарисуйте схемы.

a) не равно

b) больше, чем

c) меньше или равно

Упражнение 5.9 Спроектируйте 32-разрядное АЛУ, показанное на **Рис. 5.15**, с использованием вашего любимого языка описания аппаратуры. Модуль верхнего уровня может быть или структурным или поведенческим.

Упражнение 5.10 Добавьте выход *Overflow* к 32-разрядному АЛУ из **упражнения 5.9**. Этот выход принимает значение логической 1, если сумматор переполняется, в противном случае значение на выходе 0.

a) Запишите булево уравнения для выхода *Overflow*.

b) Нарисуйте схему, формирующую сигнал переполнения.

c) Спроектируйте модифицированное АЛУ с использованием языка описания аппаратуры.

Упражнение 5.11 Добавьте выход *Zero* к 32-разрядному АЛУ из **упражнения 5.9**. Выход принимает значение логической 1, когда *Y* == 0.

Упражнение 5.12 Напишите код среды тестирования для 32-разрядного АЛУ из **упражнений 5.9**, **5.10**, **5.11** и протестируйте АЛУ. Разработайте все необходимые файлы с тестовыми векторами. Для убеждения скептиков, обязательно детально протестируйте поведение схемы при «неудобных» данных.

Упражнение 5.13 Спроектируйте схему сдвига, которая сдвигает 32-битовый вход влево на два бита. Выход также состоит из 32-х битов. Поясните работу вашего проекта словами и нарисуйте его схему. Реализуйте ваш проект с использованием вашего любимого языка описания аппаратуры.

Упражнение 5.14 Разработайте 4-битовую схему циклического сдвига влево и вправо. Нарисуйте схему вашего проекта. Реализуйте ваш проект с использованием вашего любимого языка описания аппаратуры.

Упражнение 5.15 Спроектируйте 8-битовую схему сдвига влево с использованием только 24 мультиплексоров 2:1. На вход схемы поступает 8-битовый входной сигнал и 3-битовая величина сдвига, $shamt_{2:0}$. На выходе схемы формируется 8-битовый сигнал Y. Нарисуйте принципиальную схему.

Упражнение 5.16 Поясните, как можно построить любую N-битовую схему сдвига или циклического сдвига используя всего $N\log_2 N$ мультиплексоров 2:1.

Упражнение 5.17 Двухуровневая схема сдвига, показанная на **Рис. 5.64**, может выполнять любую N-битовую операцию сдвига или цикличного сдвига. Она сдвигает $2N$-битовый вход вправо на k бит. N младших бит результата поступают на выход Y. Старшие N бит входа обозначены через B, младшие N бит – C. При соответствующим выборе B, C, и k двухуровневая схема сдвига может выполнять любой сдвиг или циклический сдвиг. Поясните, как B, C, и k связаны с A, $shamt$ и N для выполнения:

a) логического сдвига A вправо на $shamt$

b) арифметического сдвига A вправо на $shamt$

c) сдвига A влево на $shamt$

d) циклического сдвига *A* вправо на *shamt*

e) циклического сдвига *A* влево на *shamt*

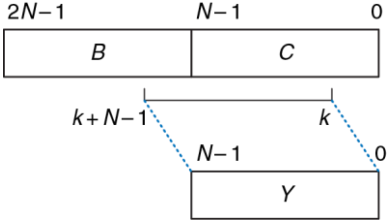


Рис. 5.64 **Двухуровневая схема сдвига**

Упражнение 5.18 Найдите критический путь и время прохождения сигнала по нему для умножителя 4 × 4, показанного на **Рис. 5.18**, считая известными задержки элемента И (t_{AND}) и сумматора (t_{FA}). Чему будет равна задержка аналогичного умножителя *N* × *N*?

Упражнение 5.19 Найдите критический путь и время прохождения сигнала по нему для схемы деления 4 × 4, показанной на **Рис. 5.20**, считая известными задержки мультиплексора 2:1 (t_{MUX}), сумматора (t_{FA}) м инвертора (t_{INV}). Чему будет равна задержка аналогичной схемы деления *N* × *N*?

Упражнение 5.20 Спроектируйте умножитель, который работает с числами, представленными в дополнительном коде.

Упражнение 5.21 Модуль расширения знака увеличивает количество разрядов числа, представленного в дополнительном коде, с M до N ($N > M$) путем копирования самого старшего разряда входа в старшие разряды выхода (см. **раздел 1.4.6**). Модуль имеет M-разрядный вход A м N-разрядный выход Y. Нарисуйте схему модуля расширения знака с 4-разрядным входом и 8-разрядным выходом. Опишите ваш проект на языке описания аппаратуры.

Упражнение 5.22 Модуль дополнения нулями увеличивает количество разрядов беззнакового числа с M до N ($N > M$) путем присвоения старшим разрядам выхода нулевого значения. Нарисуйте схему модуля дополнения нулями с 4-разрядным входом и 8-разрядным выходом. Опишите ваш проект на языке описания аппаратуры.

Упражнение 5.23 Посчитайте $111001.000_2/001100.000_2$ в двоичной системе счисления, используя стандартный школьный алгоритм деления. Продемонстрируйте свою работу.

Упражнение 5.24 Числа какого диапазона можно представить с использованием следующих форматов?

а) 24-битовое беззнаковое число с фиксированной точкой с 12 битами целой части и 12 дробной

b) 24- битовое число в прямом коде с фиксированной точкой с 12 битами целой части и 12 дробной

c) 24- битовое число в дополнительном коде с фиксированной точкой с 12 битами целой части и 12 дробной

Упражнение 5.25 Представьте следующие десятичные числа в 16-разрядном двоичном формате в прямом коде с 8 битами целой части и 8 дробной. Выразите ваш ответ в шестнадцатеричной системе.

a) -13.5625

b) 42.3125

c) -17.15625

Упражнение 5.26 Представьте следующие десятичные числа в 12-разрядном двоичном формате в прямом коде с 6 битами целой части и 6 дробной. Выразите ваш ответ в шестнадцатеричной системе.

a) -30.5

b) 16.25

c) -8.078125

Упражнение 5.27 Представьте десятичные числа из **упражнения 5.25** в 16-разрядном двоичном формате в дополнительном коде с 8 битами целой части и 8 дробной. Выразите ваш ответ в шестнадцатеричной системе.

Упражнение 5.28 Представьте десятичные числа из **упражнения 5.26** в 12-разрядном двоичном формате в дополнительном коде с 6 битами целой части и 6 дробной. Выразите ваш ответ в шестнадцатеричной системе.

Упражнение 5.29 Представьте десятичные числа из **упражнения 5.25** в формате с плавающей точкой и одинарной точностью в соответствии со стандартом IEEE 754. Выразите ваш ответ в шестнадцатеричной системе.

Упражнение 5.30 Представьте десятичные числа из **упражнения 5.26** в формате с плавающей точкой и одинарной точностью в соответствии со стандартом IEEE 754. Выразите ваш ответ в шестнадцатеричной системе.

Упражнение 5.31 Преобразуйте следующие двоичные числа с фиксированной точкой, заданные в дополнительном коде, в десятичные. Для простоты двоичная запятая в этом примере показана явно.

a) 0101.1000

b) 1111.1111

c) 1000.0000

Упражнение 5.32 Повторите **упражнение 5.31** для следующих двоичных чисел с фиксированной точкой, заданных в дополнительном коде.

a) 011101.10101

b) 100110.11010

c) 101000.00100

Упражнение 5.33 При сложении двух чисел с плавающей точкой мантисса числа с меньшим порядком сдвигается. Зачем это делается? Поясните словами и приведите пример, подтверждающий ваше объяснение.

Упражнение 5.34 Сложите следующие числа, заданные в формате с плавающей точкой и одинарной точностью в соответствии со стандартом IEEE 754.

a) C0123456 + 81C564B7

b) D0B10301 + D1B43203

c) 5EF10324 + 5E039020

Упражнение 5.35 Сложите следующие числа, заданные в формате с плавающей точкой и одинарной точностью в соответствии со стандартом IEEE 754.

a) C0D20004 + 72407020

b) C0D20004 + 40DC0004

c) (5FBE4000 + 3FF80000) + DFDE4000

(Почему полученные результаты парадоксальны? Поясните.)

Упражнение 5.36 Модифицируйте процедуру сложения чисел с плавающей точкой, описанную в **разделе 5.3.2**, для выполнения вычислений, как с положительными, так и с отрицательными числами.

Упражнение 5.37 Рассмотрим числа, заданные в формате с плавающей точкой и одинарной точностью в соответствии со стандартом IEEE 754.

a) Сколько чисел можно представить в таком формате? Особые случаи $\pm\infty$ или NaN не нужно учитывать.

b) Сколько дополнительных чисел можно представить, если не вводить в рассмотрение особые случаи $\pm\infty$ или NaN?

c) Поясните, почему для $\pm\infty$ and NaN выделено специальное представление.

Упражнение 5.38 Рассмотрим следующие десятичные числа: 245 и 0.0625.

a) Запишите эти числа в формате с плавающей точкой и одинарной точностью. Выразите ваш ответ в шестнадцатеричной системе.

b) Выполните сравнение величин двух 32-разрядных чисел, полученных в задании (a). Другими словами, интерпретируйте два 32-разрядные числа, как числа в дополнительном коде и сравните их. Будет ли сравнение таких целых чисел давать корректный результат?

c) Вы решили предложить новый формат с плавающей точкой и одинарной точностью. Единственное отличие от стандарта IEEE 754 чисел с плавающей точкой и одинарной точностью состоит в том, что вы предлагаете для порядка использовать дополнительный код, а не смещение. Запишите два числа в соответствии с вашим новым стандартом. Выразите ваш ответ в шестнадцатеричной системе.

d) Будет ли целочисленное сравнение работать с новым форматом из задания (c)?

e) Почему использование алгоритма сравнения целых чисел для чисел с плавающей точкой удобно?

Упражнение 5.39 Спроектируйте сумматор чисел с плавающей точкой и одинарной точностью с использованием вашего любимого языка описания аппаратуры. Перед написанием кода нарисуйте схему вашего проекта. Промоделируйте и протестируйте ваш сумматор, чтобы доказать скептикам, что он работает корректно. Вы можете ограничиться использованием только положительных чисел и округление выполнять до нуля (выполнять усечение). Также вы можете не рассматривать особые случаи, приведенные в **Табл. 5.2**.

Упражнение 5.40 В этом упражнении вам нужно будет спроектировать 32-битовый умножитель с плавающей точкой. Умножитель имеет два 32-битовых входа для чисел с плавающей точкой и один 32-битовый выход. Вы можете ограничиться использованием только положительных чисел и округление выполнять до нуля (выполнять усечение). Также вы можете не рассматривать особые случаи, приведенные в **Табл. 5.2**.

a) Опишите последовательность шагов, необходимых умножения 32-битовых чисел с плавающей точкой.

b) Нарисуйте схему 32-битового умножителя с плавающей точкой.

c) Опишите 32-битовый умножитель с плавающей точкой на языке описания аппаратуры. Промоделируйте и протестируйте ваш умножитель, чтобы доказать скептикам, что он работает корректно.

Упражнение 5.41 В этом упражнении вам нужно будет спроектировать 32-битовый префиксный сумматор.

a) Нарисуйте схему вашего проекта.

b) Спроектируйте 32-битовый префиксный сумматор с использованием языка описания аппаратуры. Промоделируйте и протестируйте ваш сумматор и покажите, что он работает корректно.

c) Чему будет равна задержка 32-битового префиксного сумматора, спроектированного в задании (a)? Задержка каждого двухвходового логического элемента равна 100 пс.

d) Спроектируйте конвейерную версию 32-битового префиксного сумматора, нарисуйте его схему. Насколько быстро будет быстро работать конвейерный префиксный сумматор? Потери на упорядочение ($t_{pcq} + t_{setup}$) равны 80 пс. Спроектируйте сумматор так, чтобы он имел максимально возможное быстродействие.

e) Спроектируйте 32-битовый конвейерный префиксный сумматор с использованием языка описания аппаратуры.

Упражнение 5.42 Инкрементор к *N*-разрядному числу прибавляет 1. Постройте 8-разрядный инктрементор с использованием полусумматоров.

Упражнение 5.43 Постройте 32-разрядный синхронный *реверсивный счетчик* (*Up/Down counter*). Он имеет входы *Reset* и *Up*. Когда вход *Reset* установлен в 1, все выходы сбрасываются в 0. В противном случае, если *Up* = 1, счетчик считает вверх, а когда *Up* = 0 – вниз.

Упражнение 5.44 Спроектируйте 32-разрядный счетчик, состояние которого увеличивается на 4 по каждому фронту тактового импульса. Счетчик имеет входы сброса и тактовых импульсов. После сброса все выходы счетчика устанавливаются в 0.

Упражнение 5.45 Измените счетчик из **упражнения 5.44** так, чтобы в зависимости от сигнала управления *Load*, счетчик либо увеличивал свое состояние на 4 или загружал новое 32-разрядное значение *D*. Когда *Load* = 1, счетчик загружает новое значение, поданное на вход *D*.

Упражнение 5.46 *N*-разрядный *счетчик Джонсона* (*Johnson counter*) состоит из *N*-разрядного сдвигающего регистра, имеющего сигнал сброса. Выход сдвигающего регистра (S_{out}) инвертируется и подается назад на его вход (S_{in}). Когда счетчик сбрасывается., все его рарзяди принимают нулевое значение.

a) Найдите последовательность значений на $Q_{3:0}$, которая появляется на выходе 4-разрядного счетчика Джонсона непосредственно после сброса.

b) Через сколько циклов последовательность на выходе *N*-разрядного счетчика Джонсона будет повторяться? Поясните.

c) Спроектируйте десятичный счетчик с использованием 5-разрядного счетчика Джонсона, десяти элементов И, и инверторов. Десятичный счетчик имеет входы тактового сигнала и сброса, и выход $Y_{9:0}$ с прямым кодированием «1 из 10». После сброса активируется выход Y_0. После каждого цикла активируется следующий выход. После десяти циклов состояние счетчика повторяется. Нарисуйте схему десятичного счетчика.

d) Какие преимущества может иметь счетчик Джонсона по сравнению с обычными счетчиками?

Упражнение 5.47 Создайте HDL-описание 4-битового сканируемого регистра, подобного показанному на **Рис. 5.37**. Промоделируйте и протестируйте свой HDL-модуль и покажите, что он работает корректно.

Упражнение 5.48 Английский язык имеет весьма большую избыточность, что позволяет восстановить искаженную передачу. Двоичные данные также могут быть переданы с избыточностью, которая может использоваться для исправления ошибок. Например, число 0 будет закодировано как 00000, а число 1 – как 11111. Данные передаются через зашумленный канал, который может инвертировать один или два бита. Приемник может восстановить исходные данные, если в посылке, соответствующей 0, будет, по крайней мере, три (из пяти) битов 0, аналогично для 1 будет не менее трех битов 1.

a) Предложите кодировку для передачи двухбитовых блоков 00, 01, 10 и 11 с использованием пяти битов, которая позволяет исправлять все однобитные ошибки. Подсказка: кодировка 00000 и 11111 для 00 и 11, соответственно, не будет работать.

b) Спроектируйте схему, которая будет принимать пятибитовый блок кодированных данных и декодировать его в двухбитовый блок (00, 01, 10, or 11), даже если один бит был искажен при передаче.

c) Предположим, вы хотите использовать альтернативную пятибитовую кодировку. Как можно реализовать этот проект для обеспечения возможности изменения кодировки без замены аппаратного обеспечения?

Упражнение 5.48 Флэш ЭСППЗУ, или просто флэш-память, является относительно недавним изобретением, которое революционно изменило рынок потребительской электроники. Изучите и поясните, как работает флэш-память. Для пояснения принципа работы плавающего затвора используйте диаграммы. Поясните, как происходит запись информации в память. Оформите ссылки на использованные ресурсы.

Упражнение 5.49 Участники проекта по исследованию внеземной жизни обнаружили, что на дне озера Моно живут инопланетяне. Для классификации инопланетян по возможным планетам происхождения на основе данных NASA (зелёный ил коричневый цвет кожи, слизистость, уродство) нужно создать цифровую схему. Детальные консультации с внеземными биологами привели к следующим заключениям:

- Если инопланетянин 1) зеленый и слизкий или 2) уродлив, коричневый и слизкий, то он может быть марсианином.
- Если существо 1) уродливое, коричневое и слизкое или 2) зеленое и не уродливое и не слизкое – оно может быть с Венеры.
- Если существо 1) коричневое и не уродливое и ни слизкое или 2) зеленое и слизкое – оно может быть с Юпитера.

Обратите внимание на то, что эти исследования все еще не совсем точны: например, форма жизни с пятнами зеленого и коричневого цвета, слизкая, но не уродливая, может быть с Марса или Юпитера.

a) Запрограммируйте 4 × 4 × 3 ПЛМ для идентификации пришельца. Вы можете использовать точечную нотацию.

b) Запрограммируйте 16 × 3 ПЗУ для идентификации пришельца. Вы можете использовать точечную нотацию.

c) Реализуйте свой проект на HDL.

Упражнение 5.50 Реализуйте следующие функции с использованием одного 16 × 3 ПЗУ. Для описания содержимого памяти используйте точечную нотацию.

a) $X = AB + B\bar{C}D + \overline{AB}$

b) $Y = AB + BD$

c) $Z = A + B + C + D$

Упражнение 5.51 Реализуйте функции из **упражнения 5.50** с использованием 4 × 8 × 3 ПЛМ. Вы можете использовать точечную нотацию.

Упражнение 5.52 Определите размер ПЗУ, которое можно использовать для программирования следующих комбинационных схем. Является ли использование ПЗУ для реализации этих функций хорошим проектным решением? Поясните, почему да или почему нет.

a) 16-битный сумматор/вычитатель с C_{in} и C_{out}

b) умножитель 8 × 8

c) 16-битный приоритетный шифратор (см. **упражнение 2.36**)

Упражнение 5.53 На **Рис. 5.65** показан ряд схем, в которых используется ПЗУ. Можно ли схему в столбце I заменить схемой со столбца II той же строки при условии надлежащего программирования ПЗУ?

Упражнение 5.54 Сколько логических элементов (LE) FPGA Cyclone IV необходимо для реализации указанных ниже функций? Покажите, как для этого нужно сконфигурировать один или несколько логических элементов.
При разработке конфигурации не следует пользоваться программами синтеза.

a) комбинационная функция из **упражнения 2.13 (c)**

b) комбинационная функция из **упражнения 2.17 (c)**

c) функция с двумя выходами из **упражнения 2.24**

d) функция из **упражнения 2.35**

e) четырехвходовый приоритетный шифратор (см. **упражнение 2.36**)

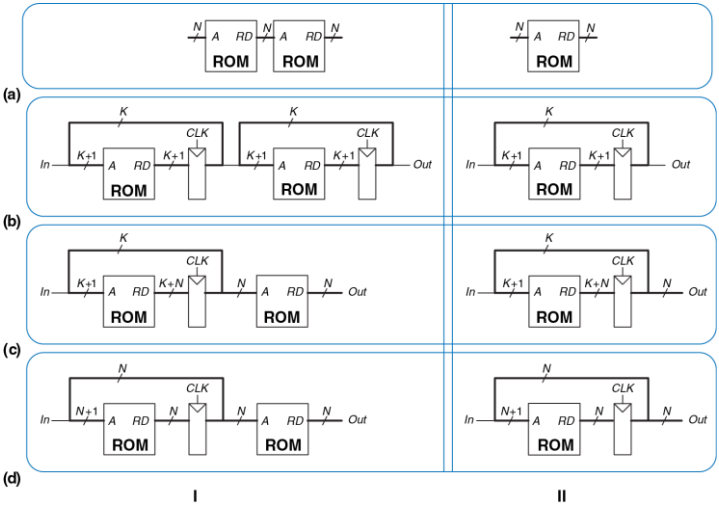


Рис. 5.65 Схемы на основе ПЗУ

Упражнение 5.55 Повторите **упражнение 5.54** для следующих функций:

a) восьмивходовый приоритетный шифратор (см. **упражнение 2.36**)

b) 3:8 декодер

c) четырехбитовый сумматор с последовательным переносом (без входа и выхода переноса)

d) конечный автомат из **упражнения 3.22**

e) счетчик, выход которого представлен в коде Грея, из **упражнения 3.27**

Упражнение 5.56 На **Рис. 5.58** показан логический элемент FPGA Cyclone IV LE. В **Табл. 5.5** приведены его временные параметры.

a) Какое минимальное количество логических элементов FPGA Cyclone IV необходимо для реализации показанного на **рисунке 3.26** конечного автомата?

b) Чему равна максимальная тактовая частота, на которой этот конечный автомат будет стабильно работать при отсутствии расфазировки тактовых импульсов?

c) Чему равна максимальная тактовая частота, на которой этот конечный автомат будет надежно работать, если максимальная расфазировка тактовых импульсов равна 3 нс?

Табл. 5.5 Временные характеристики Cyclone IV

Наименование	Величина (пс)
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (одного LE)	381
t_{wire} (между LE)	246
t_{skew}	0

Упражнение 5.57 Повторите **упражнение 5.56** для конечного автомата, который показан на **Рис. 3.31 (b)**.

Упражнение 5.58 Вы собираетесь использовать FPGA для реализации сортировщика леденцов M&M. В машине будет цветовой сенсор и мотор, который отправляет красные леденцы в одну банку, а зеленые – в другую. Проект будет реализован как конечный автомат с использованием FPGA Cyclone IV. Временные характеристики FPGA приведены в **Табл. 5.1**. Вы хотите, чтобы ваш конечный автомат работал на частоте 100 МГц. Какое максимально количество логических элементов (LE) может входить в критический путь? Чему равна максимальная частота, на которой будет работать конечный автомат?

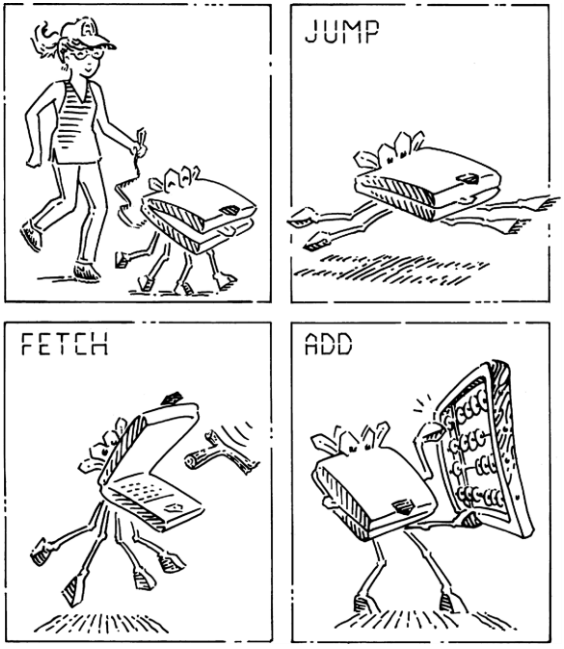
ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

В этом разделе представлены типовые вопросы, которые могут быть заданы соискателям при поиске работы в области проектирования цифровых систем.

Вопрос 5.1 Чему равен наибольший возможный результат перемножения двух беззнаковых *N*-битовых чисел?

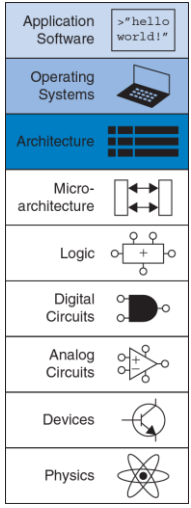
Вопрос 5.2 В *двоично-десятичном* (*BCD*) представлении для каждого десятичного разряда используется четыре бита. Например, 42_{10} будет представлено как 01000010_{BCD}. Поясните, почему процессор может использовать двоично-десятичное представление.

Вопрос 5.3 Спроектируйте сумматор, который будет складывать два беззнаковых 8-битовых числа в двоично-десятичном представлении (см. **вопрос 5.2**). Нарисуйте схему и создайте HDL-описание вашего сумматора. Сумматор имеет входные сигналы A, B и C_{in}, выходные – S и C_{out}. Сигналы C_{in} и C_{out} представляют собой однобитовый вход и выход переноса, A, B и S – 8-битовые числа в двоично-десятичном представлении.


JUMP
FETCH
ADD

Архитектура

6



6.1 ПРЕДИСЛОВИЕ

В предыдущих главах мы познакомились с принципами разработки цифровых устройств и основными цифровыми строительными блоками. В этой главе мы поднимемся на несколько уровней абстракции выше и определим *архитектуру* компьютера. Архитектура – это то, как видит компьютер программист. Она определена набором команд (языком) и местом нахождения операндов (регистры и память). Существует множество разных архитектур, таких как: x86, MIPS, SPARC и PowerPC.

Чтобы понять архитектуру любого компьютера, нужно в первую очередь выучить его язык. Слова в языке компьютера называются «инструкциями» или «командами», а словарный запас компьютера – «системой команд» (*примечание переводчика: иногда говорят, что команда – это двоичное представление слов на языке компьютера, то есть представление на уровне машинных кодов, а инструкция – это понятное человеку символьное представление этих слов на любом языке, включая язык ассемблера; в этой книге мы будем считать слова «инструкция» и «команда» синонимами*).

Даже сложные приложения, такие как редакторы текста и электронные таблицы, в конечном итоге состоят из последовательности таких простых команд, как сложение, вычитание и переход. Инструкция компьютера определяет операцию, которую нужно исполнить, и её

операнды. Операнды – это входные данные, с которыми производится операция, и получаемые результаты. Операнды могут находиться в памяти, в регистрах или внутри самой инструкции.

Аппаратное обеспечение компьютера «понимает» только нули и единицы, поэтому инструкции закодированы двоичными числами в формате, который называется *машинным языком*. Так же как мы используем буквы и прочие символы на письме для представления речи в виде, удобном для хранения, передачи и иных манипуляций, компьютеры используют двоичные числа, чтобы кодировать машинный язык. Микропроцессоры – это цифровые системы, которые читают и выполняют команды машинного языка. Для людей чтение и написание компьютерных программ на машинном языке представляется нудным и утомительным, поэтому мы предпочитаем представлять инструкции в символическом формате, который называется *языком ассемблера*. Почти все архитектуры определяют основные инструкции, такие как сложение, вычитание и переход, которые работают с ячейками памяти или регистрами. Как только вы изучили один набор инструкций, выучить другие становится довольно легко.

С какой архитектуры лучше всего начать изучение предмета?

Изучение коммерчески успешных архитектур, например, Intel x86, имеет практический смысл потому, что зная их, можно писать программы для настоящих компьютеров. К сожалению, многие из этих архитектур полны исторических нагромождений и причуд, накопленных за годы не всегда согласованной работы разных команд инженеров. Из-за этого такие архитектуры сложно понимать и воплощать в жизнь.

Многие книги учат воображаемым архитектурам, специально упрощенным для того, чтобы продемонстрировать ключевые концепции.

Мы же последуем примеру Дэвида Паттерсона и Джона Хеннесси, авторов книги «Архитектура компьютера и проектирование компьютерных систем», и сфокусируемся на архитектуре MIPS. В мире произведены сотни миллионов микропроцессоров MIPS, что говорит о бесспорной практической важности этой архитектуры (прим. переводчика: согласно официальным данным от Imagination Technologies, лицензиара ядер и архитектуры MIPS, число произведенных процессоров MIPS превышает 3,5 миллиарда, причем за один только 2014 финансовый год – 728 миллионов). При этом она остаётся достаточно ясной и имеет минимальное количество странностей. В конце этой главы мы вкратце рассмотрим архитектуру x86 и сравним её с архитектурой MIPS.

Архитектура компьютера не определяет структуру аппаратного обеспечения, которое её реализует. Зачастую существуют разные аппаратные реализации одной и той же архитектуры. Например, компании Intel и Advanced Micro Devices (AMD) производят разные микропроцессоры, которые относятся к архитектуре x86. Все они могут выполнять одни и те же программы, но при этом в их основе лежит разное аппаратное обеспечение, поэтому эти процессоры имеют разное соотношение производительности, цены и энергопотребления. Некоторые микропроцессоры оптимизированы для работы в высокопроизводительных серверах, другие оптимизированы для долгой работы батареи в ноутбуках. Взаимное расположение регистров, памяти, АЛУ и других строительных блоков, из которых состоит микропроцессор, называют *микроархитектурой*, она будет предметом **главы 7**. Нередко у одной и той же архитектуры может быть большое количество разных микроархитектур.

В этой книге мы представим вам архитектуру MIPS, которая была разработана Джоном Хеннесси и его коллегами в Стэнфорде в 1980-е годы.[4] Процессоры MIPS среди прочих использовались компаниями

[4] Архитектура MIPS, описанная в этой и следующей главах, основана на ранней версии архитектуры MIPS I, а также содержит небольшие упрощения, принятые в академической среде для обучения студентов. Главным упрощением является игнорирование слотов задержанного выполнения (см. раздел 6.4.2). Тем не менее, студенческие программы

Silicon Graphics, Nintendo и Cisco. Мы начнём описание архитектуры MIPS с рассказа об основных инструкциях, расположении операндов и форматах машинного языка. Далее мы расскажем про инструкции, которые используются для реализации общих конструкций программирования. К таким конструкциям относятся ветвления, циклы, работа с массивами и вызовы процедур.

В этой главе мы покажем, как архитектура MIPS формировалась из желания разработчиков следовать четырем простым принципам, сформулированным Паттерсоном и Хеннесси: (1) для простоты придерживайтесь единообразия; (2) типичный сценарий должен быть

остаются совместимыми с реальными процессорами, так как учебные симуляторы SPIM и MARS MIPS Simulator содержат режимы как со слотами, так и без них, а используемые и студентами, и профессиональными разработчиками ассемблеры, включая GNU Toolchain, по умолчанию позволяют программисту писать программы так, как будто слотов задержанного выполнения не существует. Ассемблер автоматически переставляет и добавляет инструкции с учетом слотов при необходимости. Кроме того, карта памяти, приведенная в этой главе (см. раздел 6.6.1), не является единственно возможной, просто в силу простоты ее часто используют, работая с учебными симуляторами. Наконец, более поздние версии архитектуры (MIPS32 и 64-битная MIPS64) содержат дополнительные инструкции, в результате чего некоторые псевдокоманды, например, циклический сдвиг, стали реальными командами процессора (см. раздел 6.7.1) – прим. переводчика.

быстрым; (3) чем меньше, тем быстрее; (4) хорошая разработка требует хороших компромиссов.

6.2 ЯЗЫК АССЕМБЛЕРА

Язык ассемблера – это удобное для восприятия человеком представление родного языка компьютера. Каждая инструкция языка ассемблера задаёт операцию, которую необходимо выполнить, а также операнды, которые будут использованы во время выполнения. Далее мы познакомим вас с простыми арифметическими инструкциями и покажем, как эти операции пишутся на языке ассемблера. Затем мы определим операнды для инструкций MIPS: регистры, ячейки памяти и константы.

В этой главе предполагается, что вы уже имеете некоторое знакомство с высокоуровневыми языками программирования, такими как C, C++ или Java (эти языки практически равнозначны для большинства примеров в этой главе, но там, где они отличаются, мы будем использовать C). В Приложении C приведено введение в язык C для тех, у кого мало или совсем нет опыта программирования на этих языках.

6.2.1 Инструкции

Наиболее частая операция, выполняемая компьютером, – это сложение. В **примере кода 6.1** показан код, который складывает переменные `b` и `c` и записывает результат в переменную `a`. Каждый пример сначала написан на языке высокого уровня (используется синтаксис C, C++ и Java), а затем переписан на языке ассемблера MIPS.

Первая часть инструкции ассемблера, `add`, называется *мнемоникой* и определяет, какую операцию нужно выполнить. Операция осуществляется над `b` и `c`, *операндами-источниками*, а результат записывается в `a`, *операнд-назначение.* (Прим. переводчика: иногда операнды-источники называют просто *операндами*, а операнд-назначение – *результатом*).

Пример кода 6.1 СЛОЖЕНИЕ

Код на языке высокого уровня

```
a = b + c;
```

Код на языке ассемблера MIPS

```
add a, b, c
```

Мнемоника происходит от греческого слова **μιμνΕσκεστηαι**. Мнемоники языка ассемблера запомнить проще, чем наборы нулей и единиц машинного языка, представляющих ту же операцию.

Пример кода 6.2 демонстрирует, что вычитание похоже на сложение. Формат инструкции такой же, как у инструкции `add`, только операция называется `sub`. Как мы с вами увидим дальше, подобное сходство не ограничивается лишь этими двумя инструкциями. Единообразный формат для команд является примером первого принципа хорошей разработки:

Пример кода 6.2 ВЫЧИТАНИЕ

Код на языке высокого уровня

```
a = b - c;
```

Код на языке ассемблера MIPS

```
sub a, b, c
```

> **Первое правило хорошей разработки:** для простоты придерживайтесь единообразия

Инструкции с одинаковым количеством операндов – в нашем случае с двумя операндами-источниками и одним операндом-назначением (то есть с двумя операндами и одним результатом) – проще

закодировать и выполнять на аппаратном уровне. Более сложный высокоуровневый код преобразуется во множество инструкций MIPS, как показано в **примере кода 6.3**.

В примерах на языках высокого уровня однострочные комментарии начинаются с символов `//` и продолжаются до конца строки. Многострочные комментарии начинаются с `/*` и завершаются `*/`. В языке ассемблера MIPS используются только однострочные комментарии. Они начинаются с `#` и продолжаются до конца строки (прим. переводчика: современные ассемблеры MIPS вызывают препроцессор языка C перед непосредственным ассемблированием, что позволяет использовать в ассемблерном коде комментарии в стиле языка C и директивы препроцессора C, такие как `#include` и `#define`). В программе на языке ассемблера в **примере кода 6.3** используется временная переменная `t` для хранения промежуточного результата.

Пример кода 6.3 БОЛЕЕ СЛОЖНЫЙ КОД

Код на языке высокого уровня

```
a = b + c - d; // single-line comment
               /* multiple-line
                  comment */
```

Код на языке ассемблера MIPS

```
sub t, c, d     # t = c - d
add a, b, t     # a = b + t
```

Использование нескольких инструкций ассемблера для выполнения более сложных операций является иллюстрацией второго принципа хорошей разработки компьютерной архитектуры:

> **Второе правило хорошей разработки:** Типичный сценарий должен быть быстрым

При использовании системы команд MIPS типичный сценарий становится быстрым потому, что она включает в себя только простые и постоянно используемые команды. Количество команд специально оставляют небольшим, чтобы аппаратное обеспечение для их поддержки было простым и быстрым. Более сложные операции, используемые не так часто, выполняются при помощи последовательности нескольких простых команд. По этой причине MIPS относится к компьютерным архитектурам *с сокращенным набором команд* (англ.: *reduced instruction set computer, RISC*). Архитектуры с большим количеством сложных инструкций, такие как архитектура x86 от Intel, называются компьютерами со сложным набором команд (англ.: *complex instruction set computer, CISC*). Например, x86 определяет инструкцию «перемещение строки», которая копирует строку

(последовательность символов) из одной части памяти в другую. Такая операция требует большого количества, вплоть до нескольких сотен, простых инструкций на RISC-машине. С другой стороны, реализация сложных инструкций в архитектуре CISC требует дополнительного аппаратного обеспечения и увеличивает накладные расходы, которые замедляют простые инструкции.

Архитектура RISC использует небольшое множество различных команд, что уменьшает сложность аппаратного обеспечения и размер инструкций. Например, код операции в системе команд, состоящей из 64 простых инструкций, потребует $\log_2 64 = 6$ бит, а в системе команд из 256 сложных инструкций потребует уже $\log_2 256 = 8$ бит. В CISC-машинах сложные команды, даже если они используются очень редко, увеличивают накладные расходы на выполнение всех инструкций, включая и самые простые.

6.2.2 Операнды: регистры, память и константы

Инструкция работает с операндами. В **примере кода 6.1** переменные `a`, `b` и `c` являются операндами. Но компьютеры оперируют нулями и единицами, а не именами переменных. Инструкция должна знать место, откуда она сможет брать двоичные данные. Операнды могут находиться в регистрах или памяти, а еще они могут быть *константами*, записанными в теле самой инструкции. Компьютеры

используют различные места для хранения операндов, чтобы повысить скорость исполнения и/или более эффективно размещать данные. Обращение к операндам-константам или операндам, находящимся в регистрах, происходит быстро, но они могут вместить лишь небольшое количество данных. Остальные данные хранятся в ёмкой, но медленной памяти. Архитектуру MIPS называют 32-битной потому, что она оперирует 32-битными данными (в некоторых коммерческих продуктах архитектура MIPS была расширена до 64 бит, но в этой книге мы будем рассматривать только 32-битный вариант).

Регистры

Чтобы команды могли быстро выполняться, они должны быстро получать доступ к операндам. Но чтение операндов из памяти занимает много времени, поэтому большинство архитектур предоставляют небольшое количество регистров для хранения наиболее часто используемых операндов. Архитектура MIPS использует 32 регистра, которые называют *набором регистров* или *регистровым файлом*. Чем меньше количество регистров, тем быстрее к ним доступ. Это приводит нас к третьему правилу хорошей разработки компьютерной архитектуры:

> **Третье правило хорошей разработки:** Чем меньше, тем быстрее

Найти необходимую информацию получится гораздо быстрее в небольшом количестве тематически подобранных книг, лежащих на столе, а не в многочисленных книгах, находящихся на полках в библиотеке. То же самое и с чтением данных из регистров и памяти. Прочитать данные из небольшого набора регистров (например, из 32 регистров) можно гораздо быстрее, чем из 1000 регистров или из большой памяти. Небольшие регистровые файлы обычно состоят из маленького массива памяти SRAM (см. **раздел 5.5.3**). Такой массив использует небольшой дешифратор адреса, подключенный битовыми линиями к относительно малому количеству ячеек памяти, благодаря чему цепи с наибольшей задержкой получаются короче, чем при доступе к большой памяти.

В **примере кода 6.4** показана инструкция `add` с регистровыми операндами. Имена регистров MIPS начинаются со знака `$`. Переменные `a`, `b` и `c` произвольно размещены в регистрах `$s0`, `$s1` и `$s2`. Имя `$s1` произносят как «регистр s1» или «доллар s1». Инструкция складывает 32-битные значения, хранящиеся в `$s1 (b)` и `$s2 (c)` и записывает 32-битный результат в `$s0 (a)`.

Пример кода 6.4 РЕГИСТРОВЫЕ ОПЕРАНДЫ

Код на языке высокого уровня

```
a = b + c
```

Код на языке ассемблера MIPS

```
# $s0 = a, $s1 = b, $s2 = c
  add $s0, $s1, $s2       # a = b + c
```

MIPS обычно хранит переменные в 18 из 32 регистров: `$s0`–`$s7` и `$t0`–`$t9`. Регистры, имена которых начинаются на `$s`, называют *сохраняемыми* (англ.: *saved*) регистрами. В соответствии с соглашением об использовании регистров MIPS эти регистры используются для размещения в них переменных таких, как `a`, `b` и `c`. Сохраняемые регистры имеют особое значение в контексте вызова процедур (см. **раздел 6.4.6**). Регистры, имена которых начинаются с `$t`, называют *временными* (англ.: temporary) регистрами. Они используются для хранения временных переменных. **пример кода 6.5** показывает написанный на ассемблере MIPS код, использующий временный регистр `$t0` для вычисленного промежуточного значения `c – d`.

Пример кода 6.5 ВРЕМЕННЫЕ ОПЕРАНДЫ

Код на языке высокого уровня

```
a = b + c - d;
```

Код на языке ассемблера MIPS

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d
  sub $t0, $s2, $s3       # t = c - d
  add $s0, $s1, $t0       # a = b + t
```

Пример 6.1 ТРАНСЛЯЦИЯ КОДА ИЗ ЯЗЫКА ВЫСОКОГО УРОВНЯ В ЯЗЫК АССЕМБЛЕРА

Транслируйте приведенный ниже код, написанный на языке высокого уровня, в код на языке ассемблера (прим. переводчика: трансляцией называется процесс преобразования программы, написанной на одном языке программирования, в программу на другом языке). Считайте, что переменные `a`, `b` и `c` находятся в регистрах `$s0`–`$s2`, а `f`, `g`, `h`, `i` и `j` – в регистрах `$s3`–`$s7`.

```
a = b - c;
f = (g + h) - (i + j);
```

Решение: Программа использует четыре ассемблерные инструкции

```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h
# $s6 = i, $s7 = j
  sub $s0, $s1, $s2  # a = b - c
  add $t0, $s4, $s5  # $t0 = g + h
  add $t1, $s6, $s7  # $t1 = i + j
  sub $s3, $t0, $t1  # f = (g + h) - (i + j)
```

Набор регистров

В архитектуре MIPS определено 32 регистра общего назначения. У каждого регистра есть имя и порядковый номер от 0 до 31. В **Табл. 6.1** перечислены имена, порядковые номера и роли каждого регистра. `$0` всегда содержит нуль потому, что эта константа часто используется в компьютерных программах. Мы уже обсудили регистры

$s и $t. Про остальные регистры мы расскажем по мере повествования в этой главе.

Табл. 6.1 Набор регистров MIPS

Название	Номер	Назначение
$0	0	Константный нуль
$at	1	Временный регистр для нужд ассемблера
$v0–$v1	2–3	Возвращаемые функциями значения
$a0–$a3	4–7	Аргументы функций
$t0–$t7	8–15	Временные переменные
$s0–$s7	16–23	Сохраняемые переменные
$t8–$t9	24–25	Временные переменные
$k0–$k1	26–27	Временные переменные операционной системы (ОС)
$gp	28	Глобальный указатель (англ.: global pointer)
$sp	29	Указатель стека (англ.: stack pointer)
$fp	30	Указатель кадра стека (англ.: frame pointer)
$ra	31	Регистр адреса возврата из функции

Память

Если бы операнды хранились только в регистрах, то мы могли бы писать лишь простые программы, содержащие не более 32 переменных. Поэтому данные также можно хранить в памяти. По сравнению с регистровым файлом память имеет много места для хранения данных, но доступ к ней занимает больше времени. По этой причине часто используемые переменные хранятся в регистрах. Комбинируя память и регистры, программа может получать доступ к большим объёмам данных достаточно быстро. Как было описано в **разделе 5.5**, память устроена как массив слов с данными. Архитектура MIPS использует 32 битные адреса памяти и 32-битные слова с данными.

MIPS использует побайтовую адресацию памяти. Это значит, что каждый байт памяти имеет уникальный адрес. Однако, для лучшего понимания, мы сначала покажем пословную адресацию памяти, а потом расскажем про побайтовую адресацию памяти в MIPS.

На **Рис. 6.1** изображен массив памяти с *пословной адресацией*. Видно, что каждое 32-битное слово данных (англ.: word) имеет уникальный 32-битный адрес (англ.: word address). И 32-битные адреса слов, и 32-битные значения (англ.: data) на **Рис. 6.1** записаны в шестнадцатеричной системе счисления. Как видно из рисунка, значение 0xF2F1AC07 хранится в памяти по адресу 1 (шестнадцатеричные числа часто записываются с префиксом 0x). При графическом изображении

памяти традиционно размещают меньшие адреса внизу, а большие – наверху.

MIPS использует команду *загрузить слово* (*англ.: load word*), `lw`, для чтения слова данных из памяти в регистр. В **примере кода 6.6** показана загрузка слова, находящегося в памяти по адресу 1, в регистр `$s3`.

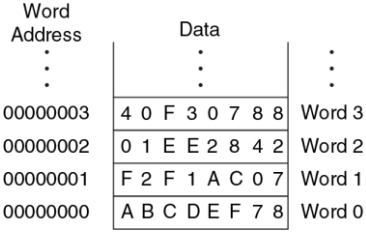


Рис. 6.1 **Память с пословной адресацией**

Инструкция `lw` определяет *эффективный адрес* в памяти как сумму *базового адреса* и *смещения*. Базовый адрес (записан в скобках в инструкции) является регистром. Смещение (записано перед скобками) является константой. В **примере кода 6.6** базовый адрес – это регистр `$0`, содержащий значение 0, а сдвиг – это 1, поэтому инструкция `lw` читает значение из памяти по адресу (`$0` + 1) = 1. После

выполнения команды загрузки слова (`lw`) в `$s3` появляется значение 0xF2F1AC07, которое находилось в памяти по адресу 1, как было показано на **Рис. 6.1**.

Пример кода 6.6 ЗАГРУЗКА СЛОВА ИЗ ПАМЯТИ С ПОСЛОВНОЙ АДРЕСАЦИЕЙ

Код на языке ассемблера

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw  $s3, 1($0)      # read memory word 1 into $s
```

Аналогичным образом MIPS использует инструкцию *сохранить слово* (*англ.: store word*), `sw`, для записи данных из регистра в память. В **примере кода 6.7** программа записывает значение регистра `$s7` в слово памяти с адресом 5. Эти примеры для простоты использовали `$0` в качестве базового адреса, но на самом деле для указания базового адреса можно использовать любой регистр.

Пример кода 6.7 ЗАПИСЬ СЛОВА В ПАМЯТЬ С ПОСЛОВНОЙ АДРЕСАЦИЕЙ

Код на языке ассемблера

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw  $s7, 5($0)          # write $s7 to memory word 5
```

В двух предыдущих примерах показана компьютерная архитектура с пословной адресацией памяти. Но модель памяти MIPS имеет не пословную, а побайтовую адресацию, при которой каждый байт данных имеет уникальный адрес.

Так как 32-битное слово состоит из четырёх 8-битных байтов, то адрес каждого слова (англ.: word address) кратен четырём, как показано на **Рис. 6.2**, где 32-битные адреса и значения слов тоже представлены в шестнадцатеричной системе счисления.

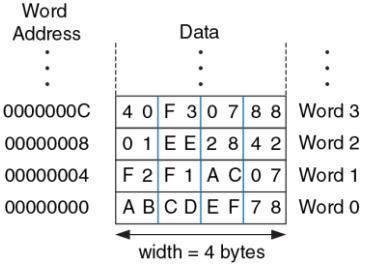


Рис. 6.2 **Память с побайтовой адресацией**

В **примере кода 6.8** показано, как читать и записывать слова в память MIPS, адресуемую побайтно. Адрес слова – это порядковый номер слова, умноженный на четыре. Программа на ассемблере MIPS читает

слова 0, 2 и 3 и записывает слова 1, 8 и 100. Смещение может быть записано в десятичной или шестнадцатеричной системе счисления (прим. переводчика: в этом примере смещение 400 и порядковый номер 100 – десятичные числа).

Пример кода 6.8 ДОСТУП К ПАМЯТИ С ПОБАЙТОВОЙ АДРЕСАЦИЕЙ

Код на языке ассемблера MIPS

```
lw  $s0, 0($0)        # read data word 0 (0xABCDEF78) into $s0
lw  $s1, 8($0)        # read data word 2 (0x01EE2842) into $s1
lw  $s2, 0xC($0)      # read data word 3 (0x40F30788) into $s2
sw  $s3, 4($0)        # write $s3 to data word 1
sw  $s4, 0x20($0)     # write $s4 to data word 8
sw  $s5, 400($0)      # write $s5 to data word 100
```

Архитектура MIPS также включает инструкции `lb` и `sb`, которые загружают и сохраняют отдельные байты, а не слова. Они похожи на инструкции `lw` и `sw` и будут рассмотрены далее в **разделе 6.4.5**.

Память с побайтовой адресацией может быть организована с *прямым порядком* следования байтов (*от младшего к старшему; англ.: little-endian*) или с *обратным порядком* (*от старшего к младшему; англ.: big-endian*), как показано на **Рис. 6.3**. В обоих случаях самый старший байт (англ.: most significant byte, MSB) находится слева, а самый младший байт (англ.: least significant byte, LSB) – справа. В машинах с

прямым порядком следования байты пронумерованы от 0, начиная с самого младшего байта. В машинах с обратным порядком следования байты пронумерованы от 0, начиная с самого старшего байта. Пословная адресация одинакова в обеих моделях, то есть один и тот же адрес слова указывает на одни и те же четыре байта. Отличаются только адреса байтов (англ.: byte addresses) внутри слова.

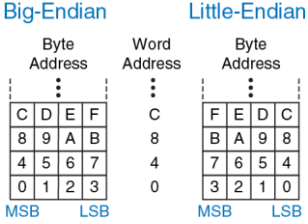


Рис. 6.3 **Адресация данных с прямым и обратным порядком байтов**

Пример 6.2 ПАМЯТЬ С ПРЯМЫМ И ОБРАТНЫМ ПОРЯДКОМ БАЙТОВ

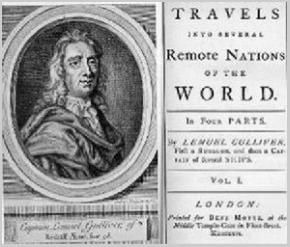
Предположим, что регистр `$s0` вначале содержит значение 0x23456789. Какое значение будет содержать этот регистр после того, как приведенная ниже программа выполнится на машине с прямым порядком байтов? А с обратным порядком? Инструкция `lb $s0, 1($0)` загружает из памяти с байтовым адресом

`(1+$0) = 1` байт данных в самый младший байт регистра `$s0`. Команда `lb` будет рассмотрена в **разделе 6.4.5**.

```
sw  $s0, 0($0)
lb  $s0, 1($0)
```

Решение: на **Рис. 6.4** показано, как машины с прямым и обратным порядком сохраняют значение 0x23456789 в слово 0 в памяти. После выполнения команды загрузки байта `lb $s0, 1($0)` регистр `$s0` будет содержать 0x00000067 на машине с прямым порядком байтов и 0x00000045 на машине с обратным порядком.

Рис. 6.4 **Расположение данных в памяти с прямым и обратным порядком байтов**



Происхождение английских терминов little-endian (прямой порядок следования байтов) и big-endian (обратный порядок) восходит к произведению «Путешествия Гулливера» Джонатана Свифта, впервые опубликованному в 1726 году под псевдонимом Исаака Бикерстаффа. В его рассказах король лилипутов требовал от граждан (остроконечников, англ.: Little-Endians) разбивать яйцо с острого конца. Тупоконечники (англ.: Big-Endians) были повстанцами, разбивавшими яйца с тупого конца. Термины были впервые применены к компьютерным архитектурам Дэнни Коэном в его статье «О священных войнах и мольбе о мире», опубликованной в день дурака 1 апреля 1980 (USC/ISI IEN 137).

(Фотография из коллекции Бразертона любезно предоставлена библиотекой Лидского университета).

Процессор PowerPC компании IBM, который ранее использовался в компьютерах Apple Macintosh, имеет обратный порядок следования байтов. Архитектура x86 компании Intel, которая используется в персональных компьютерах, имеет прямой порядок следования байтов. Некоторые процессоры MIPS используют прямой порядок, другие – обратный.[5] Выбор порядка следования байтов абсолютно произволен, но он ведёт к проблемам при обмене данными между компьютерами с разным порядком байтов. В примерах этой книги в тех случаях, когда порядок байтов имеет значение, мы будем использовать прямой порядок.

В архитектуре MIPS адреса слов для команд `lw` и `sw` должны быть *выровнены по словам (англ.: word aligned)*, то есть адреса должны делиться на 4 без остатка. Таким образом, инструкция `lw $s0, 7($0)` является некорректной. Некоторые архитектуры, такие как x86, позволяют производить операции чтения и записи данных по невыровненным адресам, но MIPS требует строгого выравнивания из соображений простоты аппаратной реализации. Разумеется, адреса

[5] SPIM, симулятор MIPS, прилагающийся к этой книге, использует порядок байтов той машины, на которой он запущен. Например, когда SPIM запущен на машине с архитектурой Intel x86, то используется прямой порядок. На старых компьютерах Apple Macintosh или Sun SPARC использовался бы обратный порядок.

байтов в командах загрузки и сохранения байтов (`lb` и `sb`) не обязательно должны быть выровнены по словам.

Константы/Непосредственные операнды

Команды загрузки и сохранения слова (`lw` и `sw`) также демонстрируют использование *констант* в командах MIPS. Эти константы называют *непосредственными операндами* (англ.: *immediate*), потому что их значения находятся непосредственно внутри команды и не требуют обращения к регистрам или памяти. Еще одна часто используемая команда MIPS, использующая непосредственный операнд – это команда сложения с константой `addi` (англ.: add immediate), которая прибавляет константу к значению регистра, как показано в **примере кода 6.9**.

Пример кода 6.9 НЕПОСРЕДСТВЕННЫЕ ОПЕРАНДЫ

Код на языке высокого уровня

```
a = a + 4;
b = a - 12;
```

Код на языке ассемблера MIPS

```
# $s0 = a, $s1 = b
  addi  $s0, $s0, 4       # a = a + 4
  addi  $s1, $s0, -12     # b = a - 12
```

Константа, находящаяся внутри команды, является 16-битным числом, представленным в дополнительном коде, и может принимать значения из диапазона [–32,768; 32,767]. Так как вычитание эквивалентно сложению с отрицательным числом, то для простоты реализации в архитектуре MIPS отсутствует команда `subi`.

Вспомните, что инструкции `add` и `sub` используют в качестве операндов три регистра, а инструкции `lw`, `sw` и `addi` – два регистра и константу. Так как форматы инструкций отличаются, то получается, что команды `lw` и `sw` нарушают первое правило хорошей разработки, гласящее, что для простоты нужно придерживаться единообразия. Однако этот случай позволяет нам представить последнее правило хорошей разработки:

> **Четвертое правило хорошей разработки:** она требует хороших компромиссов

Единый формат инструкций будет простым, но негибким. В системе команд MIPS в качестве компромисса поддерживается три формата инструкций. Первый формат используется для инструкций типа `add` и `sub`, у которых есть три регистровых операнда. Второй формат – для инструкций типа `lw` и `addi`, у которых есть два регистровых операнда и 16-битная константа (непосредственный операнд). Третий формат мы рассмотрим позже – он предназначен для инструкций, которым нужна

26-битная константа и не нужны регистровые операнды. В следующем разделе мы обсудим эти три формата инструкций MIPS и покажем, как они кодируются в двоичный формат.

6.3 МАШИННЫЙ ЯЗЫК

Язык ассемблера удобен для чтения человеком, но цифровые схемы понимают только нули и единицы. Поэтому программу, написанную на языке ассемблера, переводят из последовательности мнемоник в последовательность нулей и единиц, которую называют *машинным языком*.

Для простоты нужно придерживаться единообразия, и наиболее единообразным представлением команд в машинном языке было бы такое, где каждая команда занимала бы ровно одно слово памяти. Длина команд в архитектуре MIPS составляет 32 бита, при этом некоторые из них используют только часть из этих бит. И хотя можно было бы сделать длину команд переменной, это излишне усложнило бы архитектуру.

Для простоты можно было бы также определить единый формат для всех инструкций, но, как уже говорилось, такой подход обернулся бы серьезными ограничениями. В архитектуре MIPS в качестве компромисса используются три формата инструкций: типа *R*, типа *I* и

типа *J*. Небольшое количество форматов обеспечивает определенное единообразие между всеми тремя типами и, как следствие, более простую аппаратную реализацию. При этом разные форматы позволяют учитывать различные потребности инструкций, как, например, необходимость хранить большие константы внутри инструкций. Инструкции типа *R* используют три регистровых операнда. Инструкции типа *I* используют два регистровых операнда и 16-битную константу. Инструкции типа *J* (англ.: jump – прыжок, переход) используют 26-битную константой. В этом разделе мы представим все три формата, но подробное обсуждение инструкций типа *J* оставим до **раздела 6.4.2**.

6.3.1 Инструкции типа R

Название типа *R* является сокращением от *регистрового типа* (англ. *register-type*). Инструкции типа *R* используют три регистра в качестве операндов: два регистра-источника и один регистр-назначение. На **Рис. 6.5** показан машинный формат команды типа *R*. 32-битная команда состоит из шести полей: `op`, `rs`, `rt`, `rd`, `shamt` и `funct`. Каждое поле состоит из пяти или шести бит.

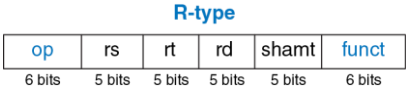


Рис. 6.5 Формат команды типа *R*

Операция, выполняемая командой, закодирована двумя полями, отмеченными синим цветом: полем `op` (также называемым `opcode` или кодом операции) и полем `funct` (также называемым функцией). У всех команд типа *R* поле `opcode` равно нулю. Операция, выполняемая этими командами, определяется исключительно полем `funct`. Например, поля `opcode` и `funct` у инструкции `add` равны 0 (000000_2) и 32 (100000_2) соответственно. Аналогично, у команды `sub` поля `opcode` и `funct` равны 0 и 34.

Операнды закодированы тремя полями: `rs`, `rt` и `rd`. Поля содержат номера регистров, приведенные в **Табл. 6.1**. Например, `$s0` – это регистр с номером 16. Регистры `rs` и `rt` являются регистрами-источниками, а `rd` – регистром-назначением (или регистром результата).

Название поля `rs` является сокращением от «регистр-источник» (англ. register source). Название поля `rt` – следующее за `rs` в алфавитном порядке, обычно обозначает второй регистр-источник.

Пятое поле, `shamt`, используется только для операций сдвига. В таких командах двоичное значение, хранимое в 5-битном поле `shamt`, задаёт величину сдвига (англ.: *shift amount*). У всех остальных команд типа *R* поле shamt равно 0.

На **Рис. 6.6** показан машинный код для двух инструкций типа *R* – `add` и `sub`. Обратите внимание на то, что в инструкции на языке ассемблера регистр-назначение идёт первым, а в команде на машинном языке он третий. Например, в ассемблерной инструкции `add $s0, $s1, $s2` поле `rs = $s1` (17), поле `rt = $s2` (18), а поле `rd = $s0` (16).

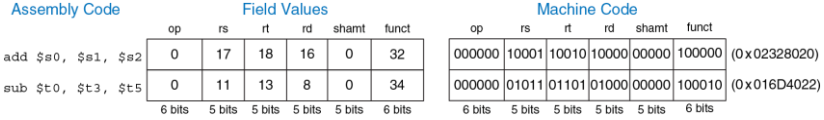


Рис. 6.6 Машинный код для инструкций типа *R*

В **Табл. B.1** и **Табл. B.2** приложения B перечислены значения поля `opcode` для всех команд, встречающихся в этой книге, и значения поля `funct` для команд типа *R*.

Пример 6.3 ТРАНСЛЯЦИЯ С ЯЗЫКА АССЕМБЛЕРА В МАШИННЫЙ ЯЗЫК

Транслируйте приведенную ниже ассемблерную инструкцию в машинный язык.

```
add $t0, $s4, $s5
```

Решение: как показано в **Табл. 6.1**, номера регистров `$t0`, `$s4` и `$s5` равны 8, 20 и 21 соответственно. Согласно **Табл. B.1** и **Табл. B.2**, поле `opcode` для команды `add` равно нулю, а поле `funct` равно 32. Поля и получившийся машинный код показаны на **Рис. 6.7**. Простейший способ получить шестнадцатеричный машинный код – это сначала записать машинный код в двоичном виде, а затем поделить его на группы по четыре бита, которые заменить соответствующими шестнадцатеричными цифрами (показанными на рисунке синим цветом). Таким образом, для этой инструкции машинный код равен 0x02954020.

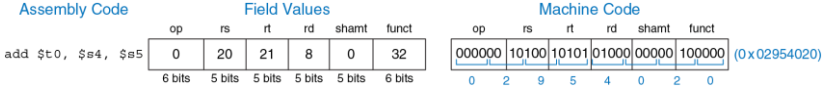


Рис. 6.7 **Машинный код для инструкции типа *R* из примера 6.3**

6.3.2 Инструкции типа I

Название типа *I* является сокращением от *непосредственного типа* (англ. *immediate-type*). Инструкции типа *I* используют в качестве операндов два регистра и один непосредственный операнд (константу).

На **Рис. 6.8** показан формат машинной команды типа *I*. 32-битная команда состоит из четырёх полей: `op`, `rs`, `rt` и `imm`. Первые три поля (`op`, `rs` и `rt`) аналогичны таким же полям в командах типа *R*. Поле `imm` (сокр. от англ. *immediate*) содержит 16-битную константу.

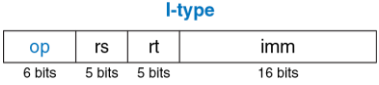


Рис. 6.8 Формат команды типа *I*

Операция определяется исключительно полем `opcode`, отмеченным синим цветом. Операнды заданы в трёх полях: `rs`, `rt` и `imm`. Поля `rs` и `imm` всегда используются как операнды-источники. Поле `rt` в некоторых командах (например, `addi` и `lw`) содержит номер регистра-назначения, в других (например, `sw`) – номер регистра-источника.

На **Рис. 6.9** приведено несколько примеров кодирования инструкций типа *I*. Вспомним, что отрицательные значения констант записывают в виде 16-битных чисел, представленных в дополнительном коде. В инструкции на языке ассемблера поле `rs` указывают первым в том случае, когда оно содержит номер регистра-назначения, но в команде машинного языка оно всегда является вторым по счету.

Инструкции типа *I* содержат 16-битную константу `imm`, но константы участвуют в 32-битных операциях. Например, инструкция `lw` добавляет 16-битное смещение к 32-битному базовому адресу. Что же произойдёт в верхних 16 битах? 16-битные константы сначала будут расширены до 32 бит следующим образом: у неотрицательных констант верхние 16 бит будут заполнены нулями, а у отрицательных констант они будут заполнены единицами. Из **раздела 1.4.6** мы помним, что этот приём называется *расширением знака*. N-битное число расширяется знаком до *M*-битного числа (*M* > *N*) путём копирования знакового (старшего) бита N-битного числа во все старшие биты *M*-битного числа. Расширение знака у числа, представленного в дополнительном коде, не меняет его значения. После расширения константы будет выполнена основная операция инструкции.

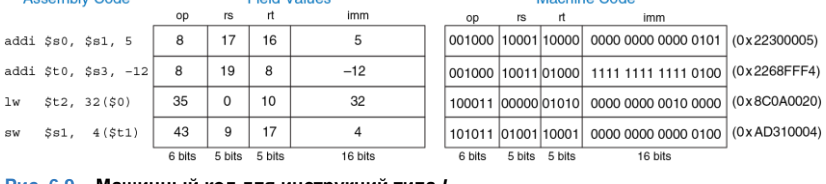


Рис. 6.9 Машинный код для инструкций типа *I*

Большинство команд MIPS производят расширение знака у непосредственных операндов. Например, `addi`, `lw` и `sw` производят расширение знака при использовании как положительных, так и отрицательных констант. Исключением из правила являются логические операции (`andi`, `ori`, `xori`), которые вместо расширения знака делают *дополнение нулями*, заполняя верхнюю половину теперь уже 32-битной константы нулями. Логические операции будут обсуждаться далее в **разделе 6.4.1**.

Пример 6.4 ТРАНСЛЯЦИЯ ИНСТРУКЦИЙ ТИПА I В МАШИННЫЙ КОД

Транслируйте приведенную ниже инструкцию в машинный код.

```
lw $s3,-24($s4)
```

Решение: согласно **Табл. 6.1**, номера регистров `$s3` и `$s4` равны 19 и 20 соответственно. Как показано в **Табл. B.1**, код операции (`opcode`) команды `lw` равен. Поле `rs` указывает на регистр `$s`4, содержащий базовый адрес, а поле `rt` указывает на регистр-назначение `$s3`. Поле imm, хранящее непосредственный операнд, содержит смещение, равное −24. Поля и получившийся машинный код показаны на **Рис. 6.10**.

Рис. 6.10 **Машинный код для инструкции типа *I* из примера 6.4**

6.3.3 Инструкции типа J

Название типа *J* является сокращением от английского слова *прыжок* (англ.: *jump*). Этот формат используется только для инструкций безусловного перехода и ветвления (см. **раздел 6.4.2**).

Как показано на **Рис. 6.11**, в формате команд этого типа определён только один 26-битный операнд `addr`.

Рис. 6.11 **Формат команды типа *J***

Как и другие команды, команды типа *J* начинаются с 6-битного поля кода операции (`opcode`). Оставшиеся биты используются для указания адреса перехода (`addr`). Дальнейшее обсуждение и примеры машинных кодов для этих команд приведены в **разделах 6.4.2** и **6.5**.

6.3.4 Расшифровываем машинные коды

Чтобы понимать язык машины, нужно уметь расшифровывать поля каждой 32-битной команды. Для разных команд определены разные форматы, но во всех форматах команды начинаются с 6-битного поля `opcode`. Если оно равно 0, то это команда типа *R*, иначе это команда типа *I* или *J*.

Пример 6.5 ТРАНСЛЯЦИЯ МАШИННЫХ КОДОВ НА ЯЗЫК АССЕМБЛЕРА

Транслируйте приведенные ниже машинные коды на язык ассемблера.

```
0x2237FFF1
0x02F34022
```

Решение: как показано на **Рис. 6.12**, сначала запишем машинные коды в двоичном коде и посмотрим на шесть самых старших бит, чтобы выяснить код операции (`opcode`) каждой из команд. Код операции скажет нам, как нужно интерпретировать оставшиеся биты. Коды операций равны 001000_2 (8_{10}) и 000000_2 (0_{10}), что соответствует команде `addi` и команде типа R соответственно. Поле `funct` команды типа R равно 100010_2 (34_{10}), что соответствует команде `sub`. На **Рис. 6.12** также показан код на языке ассемблера, эквивалентный этим двум машинным командам.

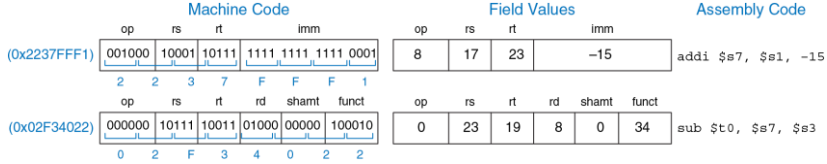


Рис. 6.12 **Трансляция машинного кода на язык ассемблера**

6.3.5 Могущество хранимой программы

Программа, написанная на машинном языке – это последовательность чисел (в архитектуре MIPS – 32-битных чисел), представляющих инструкции. Как и любые другие двоичные числа, эти инструкции можно хранить в памяти. Этот подход называется концепцией *хранимой программы* (англ.: *stored program concept*), и в нём заключается главная причина могущества компьютеров. Запуск новой программы не требует больших затрат времени и усилий на изменение или реконфигурацию аппаратного обеспечения; всё, что для этого необходимо – записать новую программу в память. Хранимые программы, в отличие от жестко зафиксированного аппаратного обеспечения, выполняющего лишь строго определенные функции, позволяют осуществлять *вычисления общего назначения* (англ.: *general purpose computing*). Используя этот подход, компьютер может выполнять любые приложения, начиная от

простого калькулятора и заканчивая текстовыми процессорами и проигрывателями видео, просто меняя хранимую программу.

В хранимой программе команды считываются, или *выбираются* (англ.: *fetch*) из памяти и выполняются процессором. Даже большие и сложные программы превращаются в последовательность операций чтения из памяти и выполнения команд.

На **Рис. 6.13** показано, как машинные команды хранятся в памяти. В программах для MIPS команды обычно хранятся, начиная с адреса 0x00400000 *(прим. переводчика: здесь и далее, в* **разделе 6.6***, авторы подразумевают типичное расположение команд и данных программ в памяти симулятора SPIM).* Вспомним, что адресация памяти в архитектуре MIPS побайтовая, поэтому 32-битные (4-байтовые) адреса команд кратны четырём байтам, а не одному.

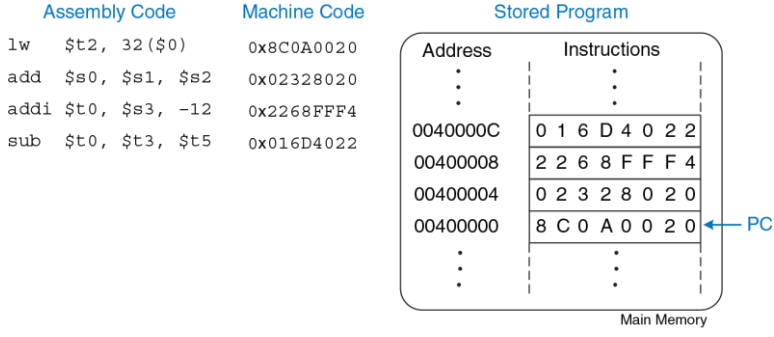


Рис. 6.13 **Хранимая программа**

Чтобы запустить, или *выполнить*, хранимую программу, процессор последовательно выбирает ее команды из памяти. Далее выбранные команды расшифровываются (*дешифруются*) и выполняются аппаратным обеспечением. Адрес текущей команды хранится в 32-битном регистре, который называют *счётчиком команд* (англ.: *program counter*, `PC`). Счётчик команд – это отдельный регистр, он не связан с 32 регистрами общего назначения, перечисленными в **Табл. 6.1**.

Для того чтобы выполнить код, показанный на **Рис. 6.1**, операционная система загружает в счётчик команд значение 0x00400000. Процессор читает из памяти по этому адресу команду 0x8C0A0020 и выполняет ее. Затем процессор увеличивает значение счётчика команд на 4 (оно становится равным 0x00400004), выбирает из памяти и выполняет новую команду, после чего процесс повторяется.

Архитектурное состояние (англ.: *architectural state*) микропроцессора содержит состояние программы. Архитектурное состояние процессоров MIPS включает в себя содержимое регистрового файла и счётчика команд. Если операционная система в какой-либо момент выполнения программы сохранит архитектурное состояние, то сможет эту программу прервать, сделать что-то ещё, а потом восстановить архитектурное состояние, после чего прерванная программа продолжит выполняться, даже не узнав, что её вообще прерывали (*прим. переводчика: строго говоря, архитектурное состояние также включает в себя и содержимое памяти, где расположены команды и данные выполняющейся программы; поскольку память обычно рассматривается отдельно от процессора, то авторы не включили ее в архитектурное состояние процессора*). Архитектурное состояние будет играть важную роль, когда мы приступим к созданию микропроцессора в **главе 7**.

6.4 ПРОГРАММИРОВАНИЕ

Языки программирования, подобные C и Java, называют языками программирования высокого уровня потому, что они предоставляют программисту возможность разрабатывать программы, используя абстракции более высокого уровня, чем те, что имеются в языке ассемблера. Большинство языков программирования высокого уровня используют весьма общие программные конструкции, такие как арифметические и логические операции, операторы `if/else`, циклы `for` и `while`, индексирование массивов и вызовы функций. В Приложении C приведено больше примеров таких конструкций в языке C. В этом разделе мы узнаем, как можно реализовать такие высокоуровневые конструкции на ассемблере MIPS.

6.4.1 Арифметические/логические инструкции

В архитектуре MIPS определены разнообразные арифметические и логические инструкции. Сейчас мы кратко с ними ознакомимся, поскольку они пригодятся нам в дальнейшем для построения высокоуровневых программных конструкций.

Ада Лавлейс, 1815–1852

Написала первую компьютерную программу. Программа предназначалась для вычисления чисел Бернулли на аналитической машине Чарльза Бэббиджа. Была единственным законнорожденным ребёнком поэта лорда Байрона.

Логические инструкции

В архитектуре MIPS имеются логические операции `AND`, `OR`, `XOR` и `NOR`. Соответствующие им одноименные инструкции типа *R* производят

побитовые операции над значениями двух регистров-источников и помещают результат в регистр-назначение. На **Рис. 6.14** продемонстрированы примеры выполнения этих операций с двумя исходными значениями, 0xFFFF0000 и 0x46A1F0B7. На рисунке показаны значения, попадающие в регистр-назначение `rd` после того, как инструкции выполнены.

Инструкция `and` полезна для *наложения маски* (англ. *masking*) на биты, т.е. для обнуления ненужных битов. На **Рис. 6.14** показана операция 0xFFFF0000 AND 0x46A1F0B7 = 0x46A10000. Инструкция `and` маскирует два младших байта и помещает два старших незамаскированных байта со значением 0x46A1 из регистра `$s2` в регистр `$s3`. Маска может быть наложена на любое подмножество битов регистра.

Инструкцию `or` хорошо использовать для объединения битов из двух регистров. Например, в результате операции 0x347A0000 `OR` 0x000072FC = 0x347A72FC мы получили комбинацию двух значений.

В архитектуре MIPS не определена операция инвертирования битов NOT, но так как A NOR `$0` = NOT A, то инструкцию NOR можно использовать в качестве замены (прим. переводчика: инструкция NOR сначала делает операцию OR с исходными значениями, затем побитно инвертирует результат этой операции).

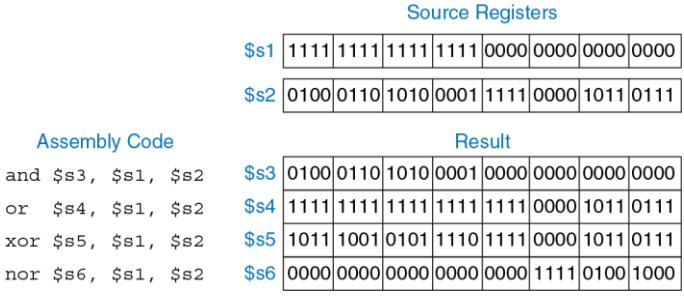


Рис. 6.14 **Логические операции**

Логические инструкции также могут работать с непосредственными операндами. Это такие инструкции типа *I*, как `andi`, `ori` и `xori`. Инструкция `nori` не определена потому, что редко нужна и легко может быть заменена уже имеющимися инструкциями. В **упражнении 6.16** читателю предлагается реализовать эквивалент инструкции `nori`. На **Рис. 6.15** продемонстрированы примеры выполнения инструкций `andi`, `ori` и `xori`.

На рисунке показаны значения регистра-источника и непосредственного операнда, а также содержимое регистра-назначения `rt` по завершении выполнения инструкций. Поскольку эти инструкции работают с

32-битным значением регистра и 16-битной константой, то сначала они дополняют константу до 32 бит нулями.

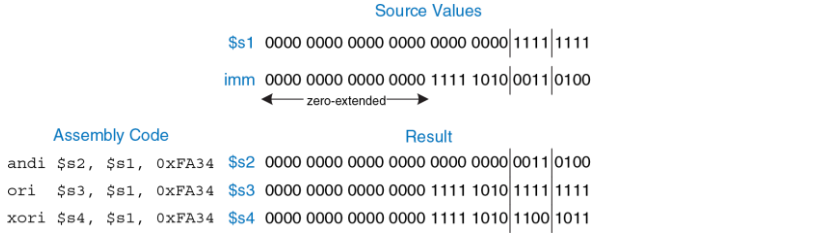


Рис. 6.15 Логические операции над непосредственными операндами

Инструкции сдвига сдвигают значение в регистре влево или вправо на любое заданное количество бит, вплоть до 31. Операции сдвига фактически умножают или делят сдвигаемые значения на степени двойки. В архитектуре MIPS существуют следующие инструкции сдвига: `sll` (логический сдвиг влево, англ.: *shift left logical*), `srl` (логический сдвиг вправо, англ.: *shift right logical*) и `sra` (арифметический сдвиг вправо, англ.: *shift right arithmetic*).

Как уже обсуждалось в **разделе 5.2.5**, сдвиги влево всегда заполняют освобождающиеся младшие биты нулями. Вместе с тем, сдвиги вправо могут быть как логическими (в освобождающиеся старшие биты

задвигаются нули), так и арифметическими (освобождающиеся старшие биты заполняются значением знакового бита). На **Рис. 6.16** показан машинный код инструкций `sll`, `srl` и `sra`. В регистре `rt` (т.е. `$s1`) хранится 32-битное значение, которое нужно сдвигать, поле `shamt` задаёт величину сдвига (4). Результат сдвига помещается в регистр `rd`.

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`sll $t0, $s1, 4`	0	0	17	8	4	0	000000	00000	10001	01000	00100	000000	(0x00114100)
`srl $s2, $s1, 4`	0	0	17	18	4	2	000000	00000	10001	10010	00100	000010	(0x00119102)
`sra $s3, $s1, 4`	0	0	17	19	4	3	000000	00000	10001	10011	00100	000011	(0x00119903)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Рис. 6.16 Машинные коды инструкций сдвига типа *R*

Рис. 6.17 иллюстрирует пример работы инструкций сдвига `sll`, `srl` и `sra`. Сдвиг значения влево на *N* битов эквивалентен умножению на 2^N. Аналогично, арифметический сдвиг значения вправо на *N* битов эквивалентен делению на 2^N, как уже обсуждалось в **разделе 5.2.5** (прим. переводчика: арифметический сдвиг отрицательных чисел вправо производит деление на 2^N с округлением результата в сторону минус бесконечности, т.е. –2 SRA 1 = –1 и –1 SRA 1 = –1. В остальных случаях сдвиг вправо происходит с округлением результата в сторону нуля, т.е. 1 SRA 1 = 0).

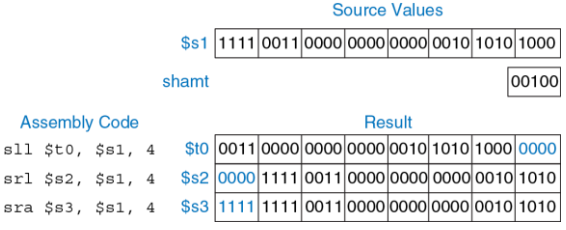


Рис. 6.17 Операции сдвига

В архитектуре MIPS также представлены инструкции так называемого переменного сдвига: `sllv` (логический переменный сдвиг влево, англ.: *shift left logical variable*), `srlv` (логический переменный сдвиг вправо, англ.: *shift right logical variable*) и `srav` (арифметический переменный сдвиг вправо, англ.: *shift right arithmetic variable*). На **Рис. 6.18** показан машинный код этих инструкций.

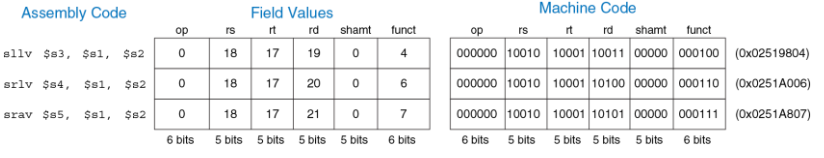


Рис. 6.18 Машинный код инструкций переменного сдвига

Ассемблерные инструкции переменного сдвига имеют форму `sllv rd, rt, rs`. Операнды `rt` и `rs` следуют в обратном порядке по сравнению с большинством инструкций типа *R*. Регистр `rt` (`$s1`) содержит сдвигаемое значение, а пять младших бит поля `rs` (`$s2`) определяют величину сдвига. Результат сдвига, как и ранее, помещается в регистр `rd`. Поле `shamt` не используется и должно быть равно нулю. **Рис. 6.19** иллюстрирует работу инструкций переменного сдвига.

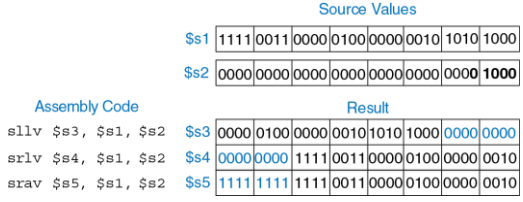


Рис. 6.19 **Операции переменного сдвига**

Загрузка констант

Как показано в **примере кода 6.10**, инструкцию `addi` удобно использовать для присвоения переменным значений 16-битных констант. Для присвоения переменным значений 32-битных констант следует использовать инструкцию `lui` (англ.: load upper immediate), которая загружает константу в старшие 16 бит регистра и обнуляет

младшие 16 битов, а также инструкцию `ori` для загрузки константы в младшие 16 бит без изменения старших, как показано в **примере кода 6.11**.

> Тип данных `int` в языке C обычно совпадает по размеру со словом памяти и хранит целые числа в дополнительном коде. Архитектура MIPS использует 32-битные слова, поэтому тип данных `int` подходит для целых чисел в диапазоне $[-2^{31}, 2^{31}-1]$.

Пример кода 6.10 16-БИТНАЯ КОНСТАНТА

Код на языке высокого уровня

```
int a = 0x4f3c;
```

Код на языке ассемблера MIPS

```
# $s0 = a
  addi $s0, $0, 0x4f3c     # a = 0x4f3c
```

Пример кода 6.11 32-БИТНАЯ КОНСТАНТА

Код на языке высокого уровня

```
int a = 0x6d5e4f3c;
```

Код на языке ассемблера MIPS

```
# $s0 = a
  lui $s0, 0x6d5e         # a = 0x6d5e0000
  ori $s0, $s0, 0x4f3c    # a = 0x6d5e4f3c
```

Инструкции умножения и деления*

Умножение и деление отличаются от других арифметических операций. Умножение двух 32-битных чисел даёт 64-битное произведение. Деление двух 32-битных чисел даёт 32-битное частное и 32-битный остаток.

В архитектуре MIPS определено два регистра специального назначения `hi` и `lo`, в которые сохраняются результаты умножения и деления. Инструкция `mult $s0, $s1` умножает значения из регистров `$s0` и `$s1`. Старшие 32 бита произведения помещаются в регистр `hi`, а младшие – в регистр `lo`. Аналогично, инструкция `div $s0, $s1` вычисляет значение `$s0/$s1`. Частное помещается в `lo`, а остаток – в `hi`.

В архитектуре MIPS есть и другая команда умножения, которая помещает 32-битный результат в регистр общего назначения. Инструкция `mul $s1, $s2, $s3` умножает значения из `$s2` и `$s3` и сохраняет 32-битный результат в `$s1` (*прим. переводчика: при*

использовании инструкции mul старшие 32 бита произведения нигде не сохраняются).

Регистры hi и lo не входят в число обычных 32 регистров общего назначения MIPS, поэтому для работы с ними требуются специальные инструкции. Инструкция mfhi `$s2` (пересылка из регистра `hi`, от англ. move from `hi`) копирует значение из регистра hi в `$s2`. Инструкция mflo `$s3` (пересылка из регистра `lo`, от англ. move from lo) копирует значение из регистра lo в `$s3`. Технически регистры `hi` и `lo` являются частью архитектурного состояния, однако мы обычно не обращаем на них внимания в этой книге.

6.4.2 Переходы

Преимуществом компьютера над калькулятором является способность принимать решения. Компьютер выполняет разные задачи в зависимости от входных данных. Например, операторы `if/else`, операторы `switch/case`, циклы `while` и `for` выполняют те или иные части кода в зависимости от результата проверки некоторых условий.

Для последовательного выполнения инструкций счетчик команд увеличивается на 4 после каждой из них. Инструкции переходов изменяют счетчик программы для того, чтобы пропустить некоторые участки кода или повторить предыдущий код. Инструкции условных

переходов, также называемые инструкциями ветвления (англ.: *branch*), проверяют какое-либо условие и осуществляют переход только в том случае, если проверка возвращает ИСТИНУ. Инструкции безусловного перехода (англ.: *jump*) осуществляют переход всегда.

Условные переходы

Система команд MIPS содержит две основные инструкции условного перехода: *ветвление при равенстве* (`beq`, от англ. branch if equal) и *ветвление при неравенстве* (`bne`, от англ. branch if not equal). Инструкция `beq` осуществляет переход, когда содержимое двух регистров равно, а `bne` осуществляет переход, если оно не равно. **пример кода 6.12** демонстрирует использование инструкции `beq`.

Пример кода 6.12 УСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `beq`

Код на языке ассемблера MIPS

```
   addi  $s0, $0, 4         # $s0 = 0 + 4 = 4
   addi  $s1, $0, 1         # $s1 = 0 + 1 = 1
   sll   $s1, $s1, 2        # $s1 = 1 << 2 = 4
   beq   $s0, $s1, target   # $s0 = = $s1, so branch is taken
   addi  $s1, $s1, 1        # not executed
   sub   $s1, $s1, $s0      # not executed
target:
   add   $s1, $s1, $s0      # $s1 = 4 + 4 = 8
```

Когда программа в **примере кода 6.12** достигает инструкции ветвления при равенстве (`beq`), значение в регистре `$s0` равно значению в `$s1`, поэтому осуществляется переход. Таким образом, следующей выполненной инструкцией будет инструкция add, располагающаяся сразу за меткой с именем target. Две инструкции, расположенные между инструкцией ветвления и меткой, не выполняются.[6]

Метки в ассемблерном коде являются ссылками на инструкции программы. Когда ассемблерный код транслируется в машинный, метки заменяются соответствующими адресами инструкций (см. **раздел 6.5**). Определяя новую метку непосредственно перед инструкцией, на которую она будет ссылаться, мы ставим двоеточие после имени метки. Имена меток не могут совпадать с зарезервированными словами, в частности, с именами инструкций. Большинство программистов делают отступы из пробелов или символов табуляции перед инструкциями, но не делают их перед метками, что позволяет визуально выделить метки среди остального кода.

---

[6] На практике, из-за конвейеризации (обсуждаемой в главе 7) в процессорах MIPS реализовано так называемое *отложенное ветвление*. Это значит, что инструкция, расположенная сразу за условным или безусловным переходом, выполняется всегда вне зависимости от того, осуществляется ли переход или нет. Говорят, что такая инструкция находится в *слоте задержанного выполнения* (англ.: *branch delay slot*). Эта особенность не учитывается в ассемблерном коде, приведенном в этой главе.

Пример кода 6.13 демонстрирует использование инструкции ветвления при неравенстве (`bne`). В этом случае ветвление не осуществляется потому, что `$s0` равен `$s1`, и процессор продолжает выполнять код, расположенный сразу после инструкции `bne`. В этом фрагменте кода выполняются все инструкции.

Пример кода 6.13 УСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `bne`

Код на языке ассемблера MIPS

```
   addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
   addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
   s11   $s1, $s1, 2         # $s1 = 1 << 2 = 4
   bne   $s0, $s1, target    # $s0 = = $s1, so branch is not taken
   addi  $s1, $s1, 1         # $s1 = 4 + 1 = 5
   sub   $s1, $s1, $s0       # $s1 = 5 - 4 = 1
target:
   add   $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Безусловные переходы

Для безусловных переходов программа может использовать инструкции трёх типов: обычный *безусловный переход* (`j`, от англ. *jump*), *безусловный переход с возвратом* (`jal`, от англ. *jump and link*) и *безусловный переход по регистру* (`jr`, от англ. *jump register*). Безусловный переход (`j`) осуществляет переход к инструкции по

указанной метке. Безусловный переход с возвратом (`jal`) похож на `j`, но дополнительно сохраняет адрес возврата и используется при вызове функций, о чём будет подробнее рассказано в **разделе 6.4.6**. Безусловный переход по регистру (`jr`) осуществляет переход к инструкции, адрес которой хранится в одном из регистров процессора.

По окончании выполнения инструкции `j target` в **примере кода 6.14** программа продолжит исполнение с инструкции `add`, расположенной непосредственно за меткой `target`. Ни одна инструкция между `j` и меткой не будет выполнена.

> Инструкции `j` и `jal` являются инструкциями типа *J*. Инструкция `jr` является инструкцией типа *R*, но использует только операнд `rs`.

Пример кода 6.14 БЕЗУСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `j`

Код на языке ассемблера MIPS

```
   addi  $s0, $0, 4      # $s0 = 4
   addi  $s1, $0, 1      # $s1 = 1
   j     target          # jump to target
   addi  $s1, $s1, 1     # not executed
   sub   $s1, $s1, $s0   # not executed
target:
   add   $s1, $s1, $s0   # $s1 = 1 + 4 = 5
```

Пример кода 6.15 демонстрирует использование инструкции безусловного перехода по регистру (`jr`). В этом примере адреса инструкций указаны слева от каждой из них. Инструкция `jr $s0` осуществляет переход по адресу 0x00002010, взятому из регистра `$s0`.

Пример кода 6.15 БЕЗУСЛОВНЫЙ ПЕРЕХОД С ИСПОЛЬЗОВАНИЕМ `jr`

Код на языке ассемблера MIPS

```
0x00002000 addi $s0,$0, 0x2010  # $s0 = 0x2010
0x00002004 jr   $s0             # jump to 0x00002010
0x00002008 addi $s1,$0, 1       # not executed
0x0000200c sra  $s1,$s1, 2      # not executed
0x00002010 lw   $s3,44($s1)     # executed after jr instruction
```

6.4.3 Условные операторы

Операторы `if`, `if/else` и `switch/case` являются условными операторами, которые часто используются в языках высокого уровня. Каждый из этих операторов при выполнении определённого условия исполняет участок кода, состоящий, в свою очередь, из одного или нескольких операторов. В этом разделе показано, как перевести эти высокоуровневые конструкции на язык ассемблера MIPS.

Оператор `if`

Оператор `if` выполняет участок кода, называемый *блоком «если»* (англ.: if block), только если выполнено заданное условие. **Пример кода 6.16** демонстрирует, как перевести выражение с оператором `if` на язык ассемблера MIPS.

Пример кода 6.16 ОПЕРАТОР `if`

Код на языке высокого уровня

```
if (i == j)
  f = g + h;
f = f - i;
```

Код на языке ассемблера MIPS

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, L1    # if i != j, skip if block
  add $s0, $s1, $s2   # if block: f = g + h
L1:
sub $s0, $s0, $s3   # f = f - i
```

Код на языке ассемблера для оператора `if` проверяет условие, противоположное условию, заданному на языке высокого уровня. В **примере кода 6.16** код на языке высокого уровня проверяет условие `i == j`, а ассемблерный код проверяет условие `i != j`. Инструкция

`bne` осуществляет ветвление, пропуская блок если, когда `i != j`. Если `i == j`, то ветвление не осуществляется и *блок «если»*, как и ожидалось, выполняется.

Операторы `if/else`

Операторы `if/else` выполняют один из двух участков кода в зависимости от условия. Когда выполнено условие выражения `if`, выполняется *блок «если»*. В противном случае выполняется *блок «иначе»* (англ.: else block). **Пример кода 6.17** демонстрирует пример оператора `if/else`.

Пример кода 6.17 ОПЕРАТОРЫ `if/else`

Код на языке высокого уровня

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

Код на языке ассемблера MIPS

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else     # if i != j, branch to else
  add $s0, $s1, $s2      # if block: f = g + h
  j   L2                 # skip past the else block
else:
  sub $s0, $s0, $s3      # else block: f = f - i
L2:
```

Как и в случае оператора `if`, ассемблерный код для оператора `if/else` проверяет условие, противоположное условию, заданному в коде на языке высокого уровня. Так, в **примере кода 6.17** код высокого уровня проверяет условие `i == j`. Ассемблерный код проверяет противоположное условие (`i != j`). Если противоположное условие истинно, то инструкция `bne` пропускает *блок если* и выполняет *блок иначе*. В противном случае, *блок «если»* выполняется и завершается инструкцией перехода (`j`) для перехода на участок после *блока «если»* (прим. переводчика: компиляторы языков высокого уровня могут менять местами *блоки «если»* и *блоки «иначе»*, а значит и генерировать машинный код с проверкой оригинального, а не противоположного условия).

Операторы `switch/case`*

Операторы `switch/case` выполняют один из нескольких участков кода в зависимости от того, какое из данных условий удовлетворяется. Если ни одно из условий не удовлетворяется, то выполняется блок `default`. Оператор `case` аналогичен последовательности *вложенных* операторов `if/else`. **Пример кода 6.18** демонстрирует два фрагмента на языке высокого уровня с одной и той же функциональностью: они вычисляют размер комиссии при выдаче денег из банкомата для сумм 20, 50 и 100 долларов. Сумма задаётся в переменной `amount` *(прим. переводчика: следует отметить, что разные высокоуровневые конструкции могут быть реализованы одинаково на ассемблере MIPS, и наоборот, одна и та же высокоуровневая конструкция может быть реализована на ассемблере по-разному. Например, оператор* `switch` *можно реализовать на языке ассемблера, создав массив с адресами переходов, который индексируется аргументом* `switch`*).*

Пример кода 6.18 ОПЕРАТОРЫ `switch/case`

Код на языке высокого уровня

```
switch (amount) {
  case 20:  fee = 2; break;
  case 50:  fee = 3; break;
  case 100: fee = 5; break;
  default:  fee = 0;
}
// equivalent function using if/else statements
  if      (amount = = 20)  fee = 2;
  else if (amount = = 50)  fee = 3;
  else if (amount = = 100) fee = 5;
else                       fee = 0;
```

Код на языке ассемблера MIPS

```
# $s0 = amount, $s1 = fee
case20:
  addi $t0, $0, 20       # $t0 = 20
  bne  $s0, $t0, case50  # amount = = 20? if not,
                         # skip to case50
  addi $s1, $0, 2        # if so, fee = 2
  j    done              # and break out of case
case50:
  addi $t0, $0, 50       # $t0 = 50
  bne  $s0, $t0, case100 # amount = = 50? if not,
                         # skip to case100
  addi $s1, $0, 3        # if so, fee = 3
```

```
  j    done              # and break out of case
case100:
  addi $t0, $0, 100      # $t0 = 100
  bne  $s0, $t0, default # amount = = 100? if not,
                         # skip to default
  addi $s1, $0, 5        # if so, fee = 5
  j    done              # and break out of case
default:
  add  $s1, $0, $0       # fee = 0
done:
```

6.4.4 Зацикливаемся

Циклы многократно выполняют участок кода в зависимости от условия. Циклы `for` и циклы `while` являются обычными конструкциями для организации циклов в языках высокого уровня. В этом разделе будет показано, как перевести их на язык ассемблера MIPS.

Циклы `while`

Циклы `while` многократно выполняют участок кода до тех пор, пока условие не станет ложным. В **примере кода 6.19** цикл `while` ищет значение `x` такое, что $2^x = 128$. Цикл выполнится семь раз, прежде чем достигнет условия `pow` = 128.

Пример кода 6.19 ЦИКЛ `while`

Код на языке высокого уровня

```
int pow = 1;
int x = 0;
while (pow != 128)
{
  pow = pow * 2;
  x = x + 1;
}
```

Код на языке ассемблера MIPS

```
# $s0 = pow, $s1 = x
  addi $s0, $0, 1      # pow = 1
  addi $s1, $0, 0      # x = 0
  addi $t0, $0, 128    # t0 = 128 for comparison
while:
  beq  $s0, $t0, done # if pow = = 128, exit while loop
  sll  $s0, $s0, 1     # pow = pow * 2
  addi $s1, $s1, 1     # x = x + 1
  j    while
done:
```

В ассемблерном коде в цикле `while` проверяется условие, противоположное условию, использованному на языке высокого уровня, аналогично тому, как это делается для оператора `if/else`. Если это

противоположное условие истинно, то цикл `while` завершается. В **примере кода 6.19** оператор цикла `while` сравнивает значение переменной `pow` со значением 128 и завершает цикл, если они равны. В противном случае происходит удвоение `pow` (используя сдвиг влево), увеличение `x` на 1 и переход обратно на начало цикла `while`.

Циклы `do/while` аналогичны циклам `while`, за исключением того, что `do/while` выполняют тело цикла один раз до первой проверки условия. Они выглядят следующим образом:

```
do
  оператор
while (условие);
```

Циклы `for`

Циклы `for`, как и циклы `while`, многократно выполняют участок кода до тех пор, пока условие цикла не станет ложным. Однако циклы `for` добавляют поддержку *счетчика цикла*, который обычно хранит количество выполненных итераций цикла. Обычно цикл `for` выглядит следующим образом:

```
for (инициализация; условие; операция цикла)
оператор
```

Код инициализации выполняется до того, как цикл `for` начнется. Условие проверяется в начале каждой итерации. Если условие не выполнено, цикл завершается. Операция цикла выполняется в конце каждой итерации.

Пример кода 6.20 складывает целые числа от 0 до 9. Счетчик итераций цикла, в данном случае `i`, инициализируется нулем и увеличивается на единицу в конце каждой итерации. Условие `i != 10` проверяется в начале каждой итерации. Итерация цикла `for` выполняется только тогда, когда условие истинно, т.е. когда значение `i` не равно 10, иначе цикл завершается. В нашем случае цикл `for` выполняется 10 раз. Циклы `for` могут быть реализованы и при помощи циклов `while`, но цикл `for` часто удобнее.

Пример кода 6.20 ЦИКЛ `for`

Код на языке высокого уровня

```
int sum = 0;
for (i = 0; i != 10; i = i + 1) {
  sum = sum + i ;
}
// equivalent to the following while loop
int sum = 0;
int i = 0;
while (i != 10) {
```

```
  sum = sum + i;
  i = i + 1;
}
```

Код на языке ассемблера MIPS

```
# $s0 = i, $s1 = sum
  add  $s1, $0, $0   # sum = 0
  addi $s0, $0, 0    # i = 0
  addi $t0, $0, 10   # $t0 = 10
for:
  beq  $s0, $t0, done # if i = = 10, branch to done
  add  $s1, $s1, $s0  # sum = sum + i
  addi $s0, $s0, 1    # increment i
  j    for
done:
```

Сравнение по величине

До сих пор в рассмотренных примерах мы использовали инструкции `beq` и `bne` для сравнений и ветвлений на основе равенства или неравенства значений. Архитектура MIPS содержит инструкцию *установить, если меньше* (`slt`, от англ. set less than) для сравнения значений по величине. Инструкция `slt` устанавливает регистр `rd` в 1, если значение регистра `rs` меньше значения регистра `rt`. Иначе, в регистр `rd` записывается 0.

Пример 6.6 ЦИКЛЫ С ИСПОЛЬЗОВАНИЕМ ИНСТРУКЦИИ `SLT`

Приведенный ниже код на языке высокого уровня складывает все степени двойки от 1 до 100. Транслируйте его на язык ассемблера.

```
// high-level code
int sum = 0;
for (i = 1; i < 101; i = i * 2)
sum = sum + i;
```

Решение: Код на языке ассемблера использует инструкцию *установить*, *если меньше* (`slt`) для выполнения сравнения в цикле `for`.

```
# MIPS assembly code
# $s0 = i, $s1 = sum
  addi $s1, $0, 0       # sum = 0
  addi $s0, $0, 1       # i = 1
  addi $t0, $0, 101     # $t0 = 101
loop:
  slt  $t1, $s0, $t0    # if (i < 101) $t1 = 1, else $t1 = 0
  beq  $t1, $0, done    # if $t1 == 0 (i >= 101), branch to done
  add  $s1, $s1, $s0    # sum = sum + i
  sll  $s0, $s0, 1      # i = i * 2
  j    loop
done:
```

В **упражнении 6.17** вам будет предложено использовать инструкцию `slt` и для других операций сравнения, таких как больше, больше или равно, меньше или равно.

6.4.5 Массивы

Массивы удобно использовать для доступа к большому количеству однородных или сходных данных. Массив располагается в ячейках памяти со строго последовательными адресами и занимает непрерывный участок памяти. Каждый массив состоит из последовательности элементов одинакового размера, и каждый элемент массива имеет порядковый номер, называемый индексом. В этом разделе показано, как получать доступ к элементам массива в памяти.

Индексация массивов

На **Рис. 6.20** показан массив из пяти элементов – целых чисел. Элементы имеют индексы от 0 до 4. Массив хранится в оперативной памяти, начиная с *базового адреса* 0x10007000. Базовый адрес определяет адрес самого первого элемента массива, `array[0]`.

В **примере кода 6.21** первые два элемента массива `array` умножаются на 8 и помещаются обратно в массив.

Пример кода 6.21 ДОСТУП К МАССИВУ

Код на языке высокого уровня

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

Код на языке ассемблера MIPS

```
# $s0 = base address of array
  lui $s0, 0x1000          # $s0 = 0x10000000
  ori $s0, $s0, 0x7000     # $s0 = 0x10007000
  lw  $t1, 0($s0)          # $t1 = array[0]
  sll $t1, $t1, 3          # $t1 = $t1 << 3 = $t1 * 8
  sw  $t1, 0($s0)          # array[0] = $t1
  lw  $t1, 4($s0)          # $t1 = array[1]
  sll $t1, $t1, 3          # $t1 = $t1 << 3 = $t1 * 8
  sw  $t1, 4($s0)          # array[1] = $t1
```

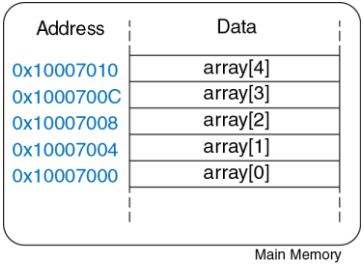


Рис. 6.20 **Массив из пяти элементов**

Первым шагом при доступе к элементам массива является загрузка базового адреса массива в регистр. В **примере кода 6.21** базовый

адрес загружается в `$s0`. Вспомним, что инструкции *загрузки константы в старшие 16 бит* (`lui`) и *побитовое логическое «ИЛИ» с константой* (`ori`) можно использовать для загрузки 32-битной константы в регистр.

Из **примера кода 6.21** также становится понятно, почему инструкция `lw` вычисляет эффективный адрес путем сложения базового адреса и смещения. Базовый адрес указывает на начало массива, а смещение можно использовать для доступа к последующим его элементам. Так, например, элемент `array[1]` хранится в памяти по адресу 0x10007004, т.е. на одно слово дальше, чем `array[0]`, поэтому доступ к нему осуществляется со смещением 4 от базового адреса *(прим. переводчика: строго говоря, если адрес элемента массива известен наперёд, и он меньше 0x00008000, например, 20, то можно не загружать базовый адрес массива в регистр, а сразу произвести его чтение или запись, используя регистр* `$0`*:* `lw $t1, 20($0)` *или* `sw $t1, 20($0)`*).*

Вы могли заметить, что для манипуляций с двумя элементами массива в **примере кода 6.21** по существу используется один и тот же код, разница состоит лишь в значениях индексов этих элементов. Дублирование кода не представляет проблемы, когда мы работаем с двумя элементами массива, но такой подход становится крайне

неэффективным, когда необходим доступ ко многим элементам массива.

В **примере кода 6.22** использован цикл `for` для умножения на 8 всех элементов массива, состоящего из 1000 элементов и находящегося в памяти по базовому адресу 0x23B8F000.

Пример кода 6.22 ДОСТУП К МАССИВУ С ПОМОЩЬЮ ЦИКЛА `for`

Код на языке высокого уровня

```
int i;
int array[1000];
for (i = 0; i < 1000; i = i + 1)
  array[i] = array[i] * 8;
```

Код на языке ассемблера MIPS

```
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8       # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000  # $s0 = 0x23B8F000
  addi $s1, $0, 0        # i = 0
  addi $t2, $0, 1000     # $t2 = 1000
loop:
  slt  $t0, $s1, $t2     # i < 1000?
  beq  $t0, $0, done     # if not, then done
  sll  $t0, $s1, 2       # $t0 = i*4 (byte offset)
  add  $t0, $t0, $s0     # address of array[i]
```

```
  lw   $t1, 0($t0)        # $t1 = array[i]
  sll  $t1, $t1, 3       # $t1 = array[i] * 8
  sw   $t1, 0($t0)       # array[i] = array[i] * 8
  addi $s1, $s1, 1       # i = i + 1
  j    loop              # repeat
done:
```

На **Рис. 6.21** показан массив, состоящий из 1000 элементов. В качестве индекса массива теперь используются не константы, а переменная `i`, поэтому мы не можем использовать непосредственные операнды в качестве смещения для инструкции `lw`. Вместо этого мы вычисляем адрес `i`-го элемента. Вспомним, что каждый элемент нашего массива – это четырёхбайтовое слово. Поскольку память адресуется побайтно, то смещение `i`-го элемента относительно базового адреса массива будет равно `i`*4. Сумма этого смещения и базового адреса массива и даст нам адрес `i`-го элемента в памяти. Для вычисления `i`*4 можно воспользоваться инструкцией сдвига влево на 2 бита. Проиллюстрированный здесь подход применим для массивов любого размера.

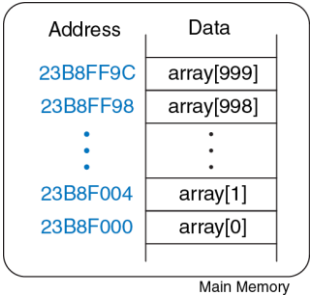


Рис. 6.21 **Размещение `array[1000]` в памяти**

Байты и символы

Так как на англоязычной клавиатуре менее 128 символов, то символы английского языка обычно хранятся не в целых машинных словах, а в восьмибитовых байтах, каждый из которых способен хранить до 256 различных значений. Язык C использует тип данных `char` для представления байтов или символов *(прим. переводчика: тип* `char` *в языке C определен как целочисленный тип данных размером не менее восьми битов. На практике встречаются системы, где размер байта и, соответственно, типа* `char` *больше, чем восемь бит. Во избежание путаницы с размером байта иногда используют*

термин октет, означающий ровно восемь бит. Тип `char` *в языке C может представлять либо знаковые, либо беззнаковые целые числа. Компилятор C вправе реализовать* `char` *и так, и этак. Чтобы избавиться от неоднозначности, используйте вместо типа* `char` *либо* `signed char`*, либо* `unsigned char`*).*

> Другие языки программирования, такие как Java, используют иные способы кодирования символов, в частности, Юникод (Unicode), также известный как Уникод. В первых версиях стандарта Юникод для кодов символов отводилось 16 бит, что позволяло поддерживать диакритические знаки (ударения, умляуты и прочие) и разнообразные языки, в том числе азиатские. В современной версии Юникода определено более ста тысяч различных символов, и 16 бит уже не достаточно для кода произвольного символа. Это вынуждает отводить на каждый символ Юникода 32 бита памяти или использовать одно из представлений с переменной длиной, например, UTF-16. Чтобы узнать больше о Юникоде, посетите www.unicode.org.

В ранних компьютерах отсутствовало однозначное соответствие между байтами и символами английского языка, поэтому текстовый обмен между компьютерами был затруднителен. В 1963 году Американская ассоциация по стандартизации опубликовала Американский стандартный код для обмена информацией (англ.: American Standard

Code for Information Interchange, сокр. ASCII), в котором каждому символу было назначено уникальное значение байта.

В **Табл. 6.2** приведены коды для всех печатных символов. Значения ASCII приведены в шестнадцатеричной форме. Буквы верхнего и нижнего регистров отличаются на 0x20 (32).

В архитектуре MIPS определены инструкции для загрузки и сохранения байтов, необходимые для работы с индивидуальными байтами или символами данных: *загрузка байта без знака* (`lbu`, от англ. load byte unsigned), *загрузка байта с расширением знака* (`lb`, от англ. load byte) и *сохранение байта* (`sb`, от англ. store byte). Все три инструкции показаны на **Рис. 6.22**.

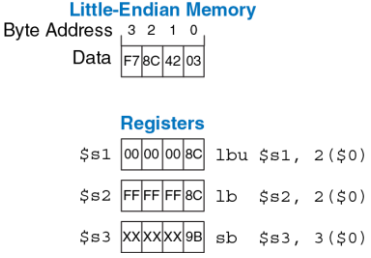


Рис. 6.22 **Инструкции для загрузки и сохранения байтов**

Табл. 6.2 Кодировка ASCII

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	−	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Коды ASCII развились из более ранних форм символьных кодировок. В 1838 году телеграфы начали использовать Азбуку Морзе, то есть последовательность точек и тире, для передачи символов. В современной Азбуке Морзе буквы A, B, C и D представляются как « . – », « – . . . », « – . – . » и « – . . » соответственно. Количество и порядок точек и тире отличаются для каждой буквы, и часто встречающиеся буквы имеют более короткие коды, что повышает компактность кодировки. В 1874 году Жан Морис Эмиль Бодо изобрёл 5-битный код, названный Азбукой Бодо. В усовершенствованной Азбуке Бодо-Мюррея буквы A, B, C и D были представлены как 00011, 11001, 01110 и 01001. Однако 32 возможных вариантов этого 5-битного кода было недостаточно для всех английских символов, а 7-битной кодировки было достаточно. Таким образом, с развитием электронных средств связи 7-битная кодировка ASCII стала стандартом. На практике под символы ASCII обычно отводятся целые байты, а кодировку ASCII зачастую расширяют до восьми бит, что позволяет закодировать в одном байте 128 дополнительных символов, например, символов другого языка.

Инструкция *загрузки байта без знака* (`lbu`) загружает байт из памяти в младший байт 32-битного регистра, обнуляя при этом остальную его часть. Эта операция называется *дополнением нулями* (англ.: zero-extension). Инструкция *загрузки байта с расширением знака* (`lb`) загружает байт из памяти в младший байт 32-битного регистра, а оставшиеся биты регистра заполняет знаковым битом байта,

т.е. битом 7. Эта операция называется *расширением знака* (англ.: sign-extension). Инструкция *сохранения байта* (`sb`) записывает младший байт 32-битного регистра в байт памяти по указанному адресу. На **Рис. 6.22** инструкция `lbu` загружает байт из памяти по адресу 2 в младший байт регистра `$s1` и заполняет оставшиеся биты регистра нулём. Инструкция `lb` загружает тот же байт в тот же регистр, заполняя оставшиеся биты регистра единицами. Инструкция `sb` сохраняет младший байт `$s3` в байт памяти по адресу 3 и заменяет значение 0xF7 значением 0x9B. Старшие байты регистра `$s3` игнорируются.

Пример 6.7 ИСПОЛЬЗОВАНИЕ `lb` И `sb` ДЛЯ ДОСТУПА К МАССИВУ СИМВОЛОВ

Приведенный ниже код на языке программирования высокого уровня преобразует буквы, находящиеся в массиве символов из 10 элементов, из строчных в прописные путем вычитания 32 из каждого элемента массива. Транслируйте этот код на язык ассемблера MIPS. Не забудьте, что адреса соседних элементов массива теперь отличаются на один, а не на четыре байта. Считайте, что регистр `$s0` уже содержит базовый адрес массива `chararray`.

```
// high-level code
char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
chararray[i] = chararray[i] - 32;
```

Решение:

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i
       addi  $s1, $0, 0       # i = 0
       addi  $t0, $0, 10      # $t0 = 10
loop:
       beq   $t0, $s1, done   # if i = = 10, exit loop
       add   $t1, $s1, $s0    # $t1 = address of chararray[i]
       lb    $t2, 0($t1)      # $t2 = array[i]
       addi  $t2, $t2, -32    # convert to upper case: $t2 = $t2 - 32
       sb    $t2, 0($t1)      # store new value in array:
                              # chararray[i] = $t2
       addi  $s1, $s1, 1      # i = i+1
       j     loop             # repeat
done:
```

Последовательность символов называют *строкой* (англ.: string). У строк переменная длина, поэтому языки программирования должны предоставлять какой-нибудь способ определения либо длины, либо конца строки. В языке Си на окончание строки указывает *нуль-символ* (0x00). Например, на **Рис. 6.23** показана строка “Hello!” (0x48 65 6C 6C 6F 21 00), хранимая в памяти. Длина строки составляет семь байтов, строка занимает адреса с 0x1522FFF0 по 0x1522FFF6 включительно. Первый символ строки (H = 0x48) хранится по наименьшему адресу (0x1522FFF0).

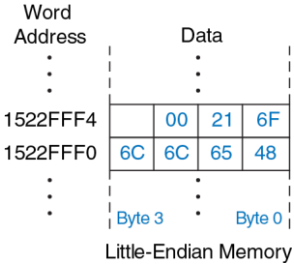


Рис. 6.23 **Строка “Hello!”, расположенная в памяти**

6.4.6 Вызовы функций

В языках высокого уровня обычно используют *функции,* или процедуры, для повторного использования часто выполняемого кода и для того, чтобы сделать программу модульной и читаемой. У функций есть входные параметры, называемые *аргументами*, и выходной результат, называемый *возвращаемым значением*. Функции должны вычислять возвращаемое значение, не вызывая неожиданных побочных эффектов.

Когда одна функция вызывает другую, *вызывающая* функция и *вызываемая* функция должны прийти к соглашению о том, где размещать аргументы и возвращаемое значение. Следуя

соглашениям, принятым в архитектуре MIPS, вызывающая функция размещает до четырёх аргументов в регистры `$a0`–`$a3` перед тем, как произвести вызов, а вызываемая функция помещает возвращаемое значение в регистры `$v0`–`$v1` перед тем, как завершить работу. Следуя этому соглашению, обе функции знают, где искать аргументы и куда возвращать значение, даже если вызывающая и вызываемая функции были написаны разными людьми.

Вызываемая функция не должна вмешиваться в работу вызывающей функции. Вкратце, это означает, что вызываемая функция должна знать, куда передать управление после завершения работы, и она не должна портить значения любых регистров или памяти, которые нужны вызывающей функции. Вызывающая функция сохраняет *адрес возврата* (англ.: return address) в регистре адреса возврата (`$ra`) в тот момент, когда она передаёт управление вызываемой функции путем выполнения инструкции *безусловного перехода с возвратом* (`jal`). Вызываемая функция не должна изменять архитектурное состояние и содержимое памяти, от которых зависит вызывающая функция. В частности, вызываемая функция должна оставить неизменным содержимое сохраняемых регистров `$s0`–`$s7`, регистра `$ra` и *стека* – участка памяти, используемого для хранения временных переменных (прим. переводчика: иными словами, если вызываемая функция хочет

изменить эти регистры, то она должна сохранить их значения в каком-нибудь другом месте и восстановить эти значения перед выходом).

В этом разделе мы покажем, как вызывать функции и возвращаться из них, продемонстрируем, как функции получают доступ к входным аргументам и возвращают значение, а также то, как они используют стек для хранения временных переменных.

Вызовы и возвраты из функций

Архитектура MIPS использует инструкцию *безусловного перехода с возвратом* (`jal`) для вызова функции и инструкцию *безусловного перехода по регистру* (`jr`) для возврата из функции. **Пример кода 6.23** демонстрирует главную функцию `main`, вызывающую функцию `simple`. Здесь функция `main` является вызывающей, а `simple` – вызываемой. Функция `simple` не получает входных аргументов и ничего не возвращает, она просто передаёт управление обратно вызывающей функции. В **примере кода 6.23** слева от каждой инструкции MIPS приведены их адреса в шестнадцатеричном формате.

Пример кода 6.23 ВЫЗОВ ФУНКЦИИ `simple`

Код на языке высокого уровня

```
int main() {
  simple();
  …
}
// void means the function returns no value
void simple() {
  return;
}
```

Код на языке ассемблера MIPS

```
0x00400200 main:    jal simple # call function
0x00400204          …
0x00401020 simple:  jr $ra     # return
```

Инструкции безусловного перехода с возвратом (`jal`) и безусловного перехода по регистру (`jr $ra`) – две необходимые для вызова функций инструкции. Инструкция `jal` выполняет две операции: сохраняет адрес *следующей* за ней инструкции в регистре адреса возврата (`$ra`) и выполняет переход по адресу первой инструкции вызываемой функции.

В **примере кода 6.23** функция `main` вызывает функцию `simple`, выполняя инструкцию `jal`. Инструкция `jal` выполняет переход на

метку simple и сохраняет значение 0x00400204 в регистре $ra. Функция simple немедленно завершается, выполняя инструкцию jr $ra, то есть осуществляет переход к инструкции по адресу, находящемуся в регистре $ra. После этого функция main продолжает выполняться с этого адреса (0x00400204).

Входные аргументы и возвращаемые значения

В **примере кода 6.23** функция simple не очень-то полезна, потому что она не получает входных значений от вызывающей функции (main) и ничего не возвращает. По соглашениям, принятым в архитектуре MIPS, функции используют регистры $a0–$a3 для входных аргументов и регистры $v0–$v1 для возвращаемого значения. В **примере кода 6.24** функция diffofsums вызывается с четырьмя аргументами и возвращает один результат.

Следуя соглашениям MIPS, вызывающая функция main помещает аргументы функции слева направо в регистры входных значений $a0–$a3. Вызываемая функция diffofsums размещает возвращаемое значение в регистре возвращаемых значений $v0.

Функция, возвращающая 64-битное значение, например, число с плавающей точкой двойной точности, использует оба регистра $v0 и $v1. Если функции нужно передать более четырех аргументов,

то дополнительные аргументы размещаются в стеке, который мы обсудим далее.

Пример кода 6.24 содержит неочевидные ошибки. В **примерах кода 6.25** и **6.26** приведены улучшенные версии программы.

Пример кода 6.24 ВЫЗОВ ФУНКЦИИ С АРГУМЕНТАМИ И ВОЗВРАЩАЕМЫМ ЗНАЧЕНИЕМ

Код на языке высокого уровня

```
int main()
{
  int y;
  …
  y = diffofsums(2, 3, 4, 5);
  …
}
int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;
}
```

Код на языке ассемблера MIPS

```
# $s0 = y
main:
  …
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call function
  add  $s0, $v0, $0  # y = returned value
  …
# $s0 = result
diffofsums:
  add  $t0, $a0, $a1 # $t0 = f + g
  add  $t1, $a2, $a3 # $t1 = h + i
  sub  $s0, $t0, $t1 # result = (f + g) - (h + i)
  add  $v0, $s0, $0  # put return value in $v0
  jr   $ra           # return to caller
```

Стек

Стек (англ.: stack) – это участок памяти для хранения локальных переменных функции. Стек расширяется (занимает больше памяти), если процессору нужно больше места, и сужается (занимает меньше памяти), если процессору больше не нужны сохранённые там переменные. Перед тем, как объяснить, как функции используют стек для хранения временных переменных, мы объясним, как стек работает.

Стек является очередью, работающей в режиме «*последним пришёл – первым ушёл*» (LIFO, от англ. last-in-first-out). Как и в стопке тарелок, последний элемент, помещенный (англ.: push) на стек (верхняя тарелка), будет первым элементом, который с него снимут (извлекут, англ.: pop). Каждая функция может выделить память на стеке для хранения локальных переменных, и она же должна освободить её перед возвратом. *Верхушка стека* (англ.: top of the stack) – это память, которая была выделена последней. Так же, как стопка тарелок растёт вверх в пространстве, стек в архитектуре MIPS увеличивается в памяти. Стек расширяется в сторону младших адресов по мере выделения нового места в памяти для программы (функции).

На **Рис. 6.24** изображен стек. *Регистр указателя стека* (`$sp`, от англ. stack pointer) – это специальный регистр, который указывает на верхушку стека. Указатель (англ.: pointer) – причудливое имя для обычного адреса памяти. Он указывает на данные, то есть

предоставляет их адрес. Например, на **Рис. 6.24 (a)** указатель стека `$sp` содержит адрес 0x7FFFFFFC и указывает на значение 0x12345678. Регистр $sp указывает на верхушку стека – наименьший адрес памяти, доступной на стеке. Таким образом, на **Рис. 6.24 (a)** стек не может использовать память ниже слова с адресом 0x7FFFFFFC.

Указатель стека (`$sp`) изначально равен большему адресу памяти, после чего его значение по необходимости уменьшается для увеличения доступного программе места. На **Рис. 6.24 (b)** изображен стек, расширяющийся для того, чтобы выделить два дополнительных слова данных для хранения временных переменных. Для этого значение регистра `$sp` уменьшается на 8 и становится равным 0x7FFFFFF4. Два дополнительных слова данных 0xAABBCCDD и 0x11223344 временно размещаются на стеке.

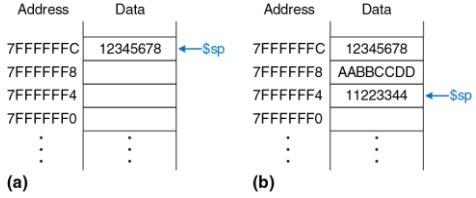

Address
Data
7FFFFFFC
12345678
$sp
7FFFFFF8
7FFFFFF4
7FFFFFF0
(a)
Address
Data
7FFFFFFC
12345678
7FFFFFF8
AABBCCDD
7FFFFFF4
11223344
$sp
7FFFFFF0
(b)

Рис. 6.24 Стек

Одно из важных применений стека – сохранение и восстановление значений регистров, используемых внутри функции. Вспомним, что функция должна производить вычисления и возвращать значения, но не должна приводить к неожиданным побочным эффектам. В частности, она не должна менять значения никаких регистров кроме регистра `$v0`, содержащего возвращаемое значение. В **примере кода 6.24** функция diffofsums нарушает это правило, потому что она изменяет регистры `$t0`, `$t1` и `$s0`. Если бы функция main использовала регистры `$t0`, `$t1` или `$s0` до вызова diffofsums, то содержимое этих регистров было бы повреждено вызовом этой функции.

Чтобы решить эту проблему, функция сохраняет значения регистров на стеке перед тем, как изменить их, и восстанавливает их из стека перед тем, как завершиться. А именно, совершает следующие шаги:

1. Выделяет пространство на стеке для сохранения значений одного или нескольких регистров.
2. Сохраняет значения регистров на стек.
3. Выполняет функцию, используя регистры.
4. Восстанавливает исходные значения регистров из стека.
5. Освобождает пространство на стеке.

В **примере кода 6.25** приведена улучшенная версия функции `diffofsums`, которая сохраняет и восстанавливает регистры `$t0`, `$t1` и `$s0`.

Пример кода 6.25 ФУНКЦИЯ, СОХРАНЯЮЩАЯ РЕГИСТРЫ НА СТЕКЕ

Код на языке ассемблера MIPS

```
# $s0 = result
diffofsums:
  addi  $sp, $sp, -12 # make space on stack to store three registers
  sw    $s0, 8($sp)   # save $s0 on stack
  sw    $t0, 4($sp)   # save $t0 on stack
  sw    $t1, 0($sp)   # save $t1 on stack
  add   $t0, $a0, $a1 # $t0 = f + g
  add   $t1, $a2, $a3 # $t1 = h + i
  sub   $s0, $t0, $t1 # result = (f + g) - (h + i)
  add   $v0, $s0, $0  # put return value in $v0
  lw    $t1, 0($sp)   # restore $t1 from stack
  lw    $t0, 4($sp)   # restore $t0 from stack
  lw    $s0, 8($sp)   # restore $s0 from stack
  addi  $sp, $sp, 12  # deallocate stack space
  jr    $ra           # return to caller
```

На **Рис. 6.25** показан стек до, во время и после вызова функции `diffofsums` из **примера кода 6.25**.

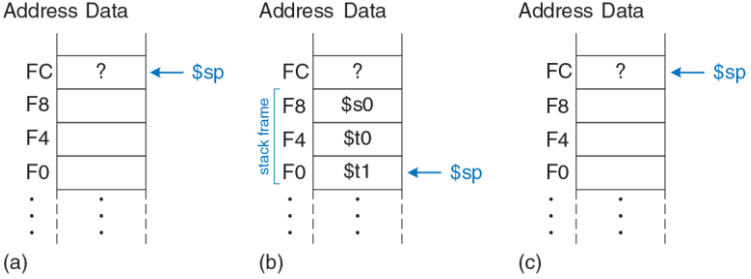


Рис. 6.25 Стек до (a), во время (b) и после (c) вызова функции `diffofsums`

Функция diffofsums выделяет пространство на стеке для трёх слов, уменьшая указатель стека `$sp` на 12. Затем она сохраняет текущие значения `$s0`, `$t0` и `$t1` в выделенном пространстве. Дальше выполняется остальная часть функции, которая меняет значения этих трёх регистров. В завершение, функция `diffofsums` восстанавливает значения регистров `$s0`, `$t0` и `$t1` из стека, освобождает пространство на стеке и возвращается в `main`. Когда функция выполняет возврат, в регистре `$v0` находится её результат, но другие побочные эффекты

отсутствуют: в регистрах `$s0`, `$t0`, `$t1` и `$sp` находятся те же значения, которые там и были до вызова функции.

Оберегаемые регистры

В **примере кода 6.26** предполагается, что временные регистры `$t0` и `$t1` следует сохранять и восстанавливать. Если вызывающая функция не использует эти регистры, то усилия по сохранению и восстановлению их значений тратятся впустую. Чтобы избежать этих издержек, в архитектуре MIPS регистры разделены на две категории: *оберегаемые* (англ.: preserved) и *необерегаемые* (англ.: nonpreserved). Оберегаемые регистры включают `$s0`–`$s7` (отсюда их название: *сохраняемые, англ.: saved*). Необерегаемые регистры включают `$t0`–`$t9` (отсюда их название: *временные, англ.: temporary*). Функция должна сохранять и восстанавливать любые оберегаемые регистры, с которыми она собирается работать, но может свободно менять значения необерегаемых регистров.

В **примере кода 6.26** показана улучшенная версия функции diffofsums, которая сохраняет на стеке только регистр `$s0`. Регистры `$t0` и `$t1` являются необерегаемыми регистрами, поэтому их сохранять не обязательно.

Пример кода 6.26 ФУНКЦИЯ, СОХРАНЯЮЩАЯ ОБЕРЕГАЕМЫЕ РЕГИСТРЫ НА СТЕКЕ

Код на языке ассемблера MIPS

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to store one register
  sw   $s0, 0($sp)    # save $s0 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr   $ra            # return to caller
```

Вспомним, что когда одна функция вызывает другую, то первая называется *вызывающей функцией*, а вторая – *вызываемой*. Вызываемая функция должна сохранять и восстанавливать любые оберегаемые регистры, которые собирается использовать, но может свободно изменять любые необеpегаемые регистры. Следовательно, если вызывающая функция держит актуальные данные в необеpегаемых регистрах, она должна сохранять необеpегаемые регистры перед тем, как вызывать другую функцию, а затем их восстанавливать. По этой причине оберегаемые регистры также

называют *сохраняемыми вызываемой функцией*, а необеregаемые регистры называют *сохраняемыми вызывающей функцией*.

В **Табл. 6.3** приведены все оберегаемые регистры. Регистры `$s0`–`$s7` обычно используют для хранения локальных переменных внутри функции, поэтому они должны быть сохранены. Регистр `$ra` также следует сохранять, чтобы функция знала, куда возвращаться. Регистры `$t0`–`$t9` используют для хранения временных результатов перед тем, как присвоить эти значения локальным переменным. Вычисления, использующие временные результаты, обычно завершаются до того, как вызывается функция, поэтому эти регистры не оберегаются, а необходимость сохранять их в вызывающей функции возникает крайне редко. Регистры `$a0`–`$a3` часто перезаписываются в процессе вызова функции, поэтому вызывающая функция должна сохранять их, если эти значения могут понадобиться ей после завершения вызванной функции. Регистры `$v0`–`$v1`, очевидно, не следует оберегать, потому что в них вызываемая функция возвращает свой результат.

Табл. 6.3 Оберегаемые и необерегаемые регистры

Оберегаемые	Необерегаемые
Сохраняемые регистры: `$s0`–`$s7`	Временные регистры: `$t0`–`$t9`
Адрес возврата: `$ra`	Регистры аргументов: `$a0`–`$a3`
Указатель стека: `$sp`	Возвращаемые значения: `$v0`–`$v1`
Содержимое стека выше указателя стека	Стек ниже указателя стека

Стек выше указателя стека автоматически остаётся в сохранности, если только вызываемая функция не осуществляет запись в память по адресам выше `$sp`. При таком подходе она не меняет *кадры стека* (англ.: stack frames) других функций. Сам указатель стека остаётся в сохранности потому, что вызываемая функция перед завершением работы освобождает свой кадр стека, прибавляя к `$sp` то же значение, которое вычла из него в начале.

Рекурсивные вызовы функций

Функция, которая не вызывает другие функции, называется *листовой*, или *терминальной*, функцией (англ.: leaf function); пример – функция `diffofsums`. Функция, которая вызывает другие функции, называется *нелистовой* (или, соответственно, *нетерминальной*, англ.: nonleaf function). Как было замечено ранее, нелистовые функции устроены более сложно, потому что перед вызовом других функций им

приходится сохранять необеregаемые регистры на стеке и затем восстанавливать эти регистры. А именно, вызывающая функция сохраняет любые необеregаемые регистры (`$t0`–`$t9` и `$a0`–`$a3`), значения которых будут нужны ей после вызова. Вызываемая функция сохраняет любые оберегаемые регистры (`$s0`–`$s7` и `$ra`), которые собирается изменять.

Рекурсивная функция – это нелистовая функция, вызывающая сама себя. Функция вычисления факториала может быть реализована в виде рекурсивной функции. Вспомним, что

factorial(n) = n × (n – 1) × (n – 2) ×… × 2 × 1. Функция `factorial` в рекурсивном представлении выглядит как *factorial*(n) = n × f*actorial*(n – 1). Факториал от 1 – это просто 1. В **примере кода 6.27** показана функция `factorial`, записанная в рекурсивном виде. Для удобства предполагаем, что программа начинается с адреса 0x90.

Пример кода 6.27 РЕКУРСИВНЫЙ ВЫЗОВ ФУНКЦИИ `factorial`

Код на языке высокого уровня

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

Код на языке ассемблера MIPS

```
0x90  factorial: addi $sp, $sp, -8  # make room on stack
0x94             sw   $a0, 4($sp)   # store $a0
0x98             sw   $ra, 0($sp)   # store $ra
0x9C             addi $t0, $0, 2    # $t0 = 2
OxAO             slt  $t0, $a0, $t0 # n <= 1 ?
0xA4             beq  $t0, $0, else # no: goto else
0xA8             addi $v0, $0, 1    # yes: return 1
OxAC             addi $sp, $sp, 8   # restore $sp
OxBO             jr   $ra           # return
0xB4     else:   addi $a0, $a0, -1  # n = n - 1
0xB8             jal  factorial     # recursive call
OxBC             Iw   $ra, 0($sp)   # restore $ra
OxCO             Iw   $a0, 4($sp)   # restore $a0
0xC4             addi $sp, $sp, 8   # restore $sp
0xC8             mul  $v0, $a0, $v0 # n * factorial(n-1)
OxCC             jr   $ra           # return
```

Функция `factorial` изменяет регистры `$a0` и `$ra`, поэтому она сохраняет их на стеке. Затем она проверяет условие `n` < 2. Если условие выполнено, она помещает значение 1 в регистр `$v0`, восстанавливает указатель стека и возвращается к вызывающей функции. В этом случае нет необходимости восстанавливать регистры `$ra` и `$a0` потому, что они не менялись. Если `n` > 1, функция рекурсивно вызывает f`actorial(n-1)`. Затем она восстанавливает

значение n (`$a0`) и адрес возврата (`$ra`) из стека, выполняет умножение и возвращает результат. Инструкция умножения (`mul $v0, $a0, $v0`) умножает `$a0` и `$v0` и помещает результат в `$v0`.

На **Рис. 6.26** показан стек в процессе выполнения функции `factorial(3)`. Мы предполагаем, что `$sp` изначально указывает на 0xFC, как показано на **Рис. 6.26 (a)**. Функция создаёт кадр стека из двух слов для хранения `$a0` и `$ra`. При первом вызове `factorial` сохраняет `$a0` (содержащий `n` = 3) по адресу 0xF8 и `$ra` по адресу 0xF4, как показано на **Рис. 6.26 (b)**. Затем функция меняет `$a0` на `n` = 2 и рекурсивно вызывает `factorial(2)`, при этом значение `$ra` меняется на 0xBC. При втором вызове функция сохраняет `$a0` (содержащий `n = 2`) по адресу 0xF0 и $ra по адресу 0xEC. В этот раз мы знаем, что `$ra` содержит 0xBC. Затем функция меняет `$a0` на `n` = 1 и рекурсивно вызывает `factorial(1)`. При третьем вызове она сохраняет `$a0` (содержащий `n` = 1) по адресу 0xE8 и `$ra` по адресу 0xE4. На этот раз `$ra` опять содержит 0xBC. Третий вызов `factorial` возвращает значение 1 в регистре `$v0` и освобождает кадр стека перед тем, как вернуться ко второму вызову. Второй вызов восстанавливает значение `n`=2, восстанавливает значение 0xBC в `$ra` (так получилось, что он уже содержит это значение), освобождает кадр стека и возвращает `$v0` = 2 × 1 = 2 первому вызову. Первый вызов

восстанавливает `n` = 3, восстанавливает адрес возврата к вызывающей функции в `$ra`, освобождает кадр стека и возвращает `$v0` = 3 × 2 = 6. На **Рис. 6.26 (c)** показан стек после завершения рекурсивно вызванных функций. Когда функция factorial возвращает управление вызвавшей ее функции, указатель стека находится в своём изначальном положении (0xFC), содержимое стека выше значения указателя стека не менялось и все оберегаемые регистры содержат свои изначальные значения. Регистр `$v0` содержит результат вычисления (6).

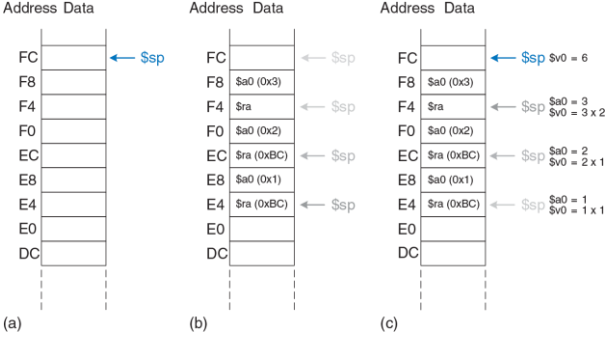


Рис. 6.26 Стек до (a), после последнего рекурсивного вызова (b) и после завершения (c) вызова функции `factorial` при `n` = 3

Дополнительные аргументы и локальные переменные*

У функций могут быть более четырёх аргументов и локальных переменных. Стек используют для хранения этих временных значений. Следуя соглашениям архитектуры MIPS, если у функции больше четырёх аргументов, то первые четыре из них, как обычно, передаются через регистры для аргументов. Дополнительные аргументы передаются на стеке, прямо над указателем стека `$sp`. *Вызывающая* функция должна расширить стек для дополнительных аргументов. На **Рис. 6.27 (a)** показан стек вызывающей функции при вызове функции, принимающей более четырёх аргументов.

Функция также может объявлять локальные переменные или массивы. С *локальными* переменными, объявленными внутри функции, может работать только сама эта функция. Локальные переменные хранятся в регистрах `$s0`–`$s7`; если локальных переменных слишком много, их можно хранить в кадре стека функции. В частности, локальные массивы хранятся в стеке.

На **Рис. 6.27 (b)** показана структура стека вызываемой функции. Кадр стека содержит аргументы самой функции, адрес возврата и любые оберегаемые регистры, которые функция может менять. Он также содержит локальные массивы и любые дополнительные локальные переменные. Если у вызываемой функции более четырёх аргументов, она находит их в кадре стека вызывающей функции. Доступ к

дополнительным аргументам – это единственный случай, когда функции позволено читать данные из стека за пределами собственного кадра.

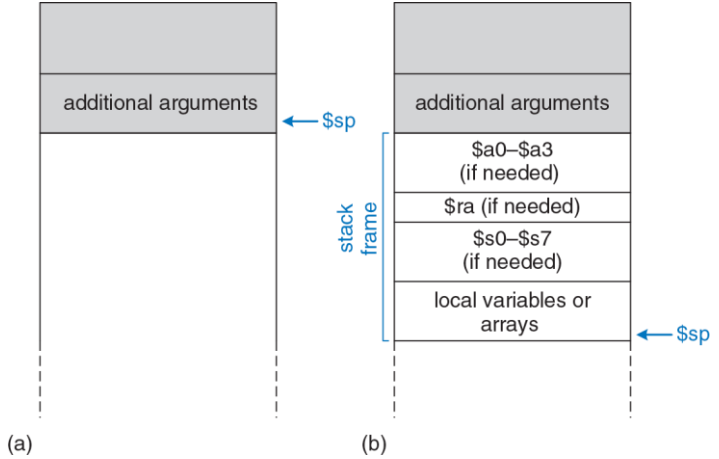


Рис. 6.27 Использование стека перед вызовом (a) и после вызова (b)

6.5 РЕЖИМЫ АДРЕСАЦИИ

В архитектуре MIPS используются пять *режимов адресации*: регистровый (англ.: register-only), непосредственный (англ.: immediate), базовый (англ.: base), относительно счетчика команд (англ.: PC-relative) и псевдопрямой (англ.: pseudo-direct). Первые три режима (регистровый, непосредственный и базовый) определяют способы чтения и записи операндов. Последние два (режим адресации относительно счетчика команд и псевдопрямой режим) определяют способы записи счётчика команд (англ.: program counter, `PC`).

Регистровая адресация

При *регистровой адресации* регистры используются для всех операндов-источников и операндов-назначений (иными словами – для всех операндов и результата). Все инструкции типа *R* используют именно такой режим адресации.

Непосредственная адресация

При *непосредственной адресации* в качестве операндов наряду с регистрами используют 16-битные константы (непосредственные операнды). Этот режим адресации используют некоторые инструкции типа *I*, такие как сложение с константой (`addi`) и загрузка константы в старшие 16 бит регистра (`lui`).

Базовая адресация

Инструкции для доступа в память, такие как загрузка слова (`lw`) и сохранение слова (`sw`), используют *базовую адресацию*. Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре `rs` и 16-битного смещения с расширенным знаком, являющегося непосредственным операндом.

Адресация относительно счетчика команд

Инструкции условного перехода, или ветвления, используют *адресацию относительно счетчика команд* для определения нового значения счетчика команд в том случае, если нужно осуществить переход. Смещение со знаком прибавляется к счетчику команд (PC) для определения нового значения PC, поэтому тот адрес, куда будет осуществлен переход, называют адресом *относительно* счетчика команд.

В **примере кода 6.28** показана часть функции `factorial` из **примера кода 6.27**.

Пример кода 6.28 ВЫЧИСЛЕНИЕ ЦЕЛЕВОГО АДРЕСА ВЕТВЛЕНИЯ

Код на языке ассемблера MIPS

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xBO          jr   $ra
0xB4    else: addi $a0, $a0, -1
0xB8          jal  factorial
```

На **Рис. 6.28** показан машинный код для инструкции `beq`. *Целевой адрес ветвления* (англ.: *branch target address, BTA*) – это адрес инструкции, которая будет выполнена следующей в том случае, если случится ветвление. На **Рис. 6.28** у инструкции `beq` целевой адрес ветвления равен 0xB4 – это адрес инструкции с меткой `else`.

Рис. 6.28 **Машинный код для инструкции `beq`**

16-битный непосредственный операнд задаёт число инструкций между целевым адресом ветвления и инструкцией, находящейся сразу *после* инструкции перехода (т.е. инструкции по адресу PC + 4). В нашем

случае непосредственный операнд равен 3 потому, что целевой адрес ветвления (0xB4) расположен через 3 инструкции после инструкции с адресом PC + 4 (0xA8).

Процессор вычисляет целевой адрес ветвления, выполняя знаковое расширение 16-битного непосредственного операнда до 32 бит и умножая полученное значение на 4 (чтобы преобразовать слова в байты), после чего добавляя это произведение к значению PC + 4.

Пример 6.8 АДРЕСАЦИЯ ОТНОСИТЕЛЬНО СЧЕТЧИКА КОМАНД

Вычислите значение непосредственного операнда и запишите машинный код для инструкции `bne` в программе, приведенной ниже.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Решение:

На **Рис. 6.29** показан машинный код инструкции `bne`. Ее целевой адрес ветвления равен 0x40 и находится на 6 инструкций раньше, чем значение PC + 4 (0x58), поэтому непосредственный операнд в этом случае равен –6.

Рис. 6.29 Машинный код для инструкции `bne` из примера 6.8

Псевдопрямая адресация

При *прямой адресации* адрес перехода задаётся внутри инструкции. Инструкции безусловного перехода `j` и `jal` в идеале могли бы использовать прямую адресацию для определения 32-битного *целевого адреса перехода* (англ.: *jump target address, JTA*), указывающего адрес инструкции, которая будет выполнена следующей.

К сожалению, в формате инструкций типа *J* нет достаточного количества бит для того, чтобы задать полный 32-битный адрес перехода. Шесть старших бит инструкции занимает код операции (поле `opcode`), поэтому для адреса перехода остаётся только 26 бит. К счастью, два младших бита адреса перехода ($JTA_{1:0}$) всегда должны быть равны нулю, потому что все инструкции выровнены по словам. Следующие 26 бит адреса перехода ($JTA_{27:2}$) берутся из поля `addr` инструкции. Четыре старших бита адреса перехода ($JTA_{31:28}$) берутся из четырёх старших бит значения PC + 4. Такой способ адресации называется *псевдопрямым*.

В **примере кода 6.29** показана инструкция `jal`, использующая псевдопрямую адресацию. Целевой адрес безусловного перехода этой инструкции равен 0x004000A0, а ее машинный код показан на **Рис. 6.30**. Четыре старших и два младших бита целевого адреса перехода отбрасываются, а оставшиеся биты помещаются в 26-битное поле адреса (`addr`).

Пример кода 6.29 ВЫЧИСЛЕНИЕ ЦЕЛЕВОГО АДРЕСА ПЕРЕХОДА

Код на языке ассемблера MIPS

```
0x0040005C      jal   sum
...
0x004000A0 sum: add   $v0, $a0, $a1
```

Процессор вычисляет целевой адрес перехода инструкции типа *J*, добавляя два нуля в конец и четыре старших бита значения PC + 4 в начало 26-битного поля `addr`.

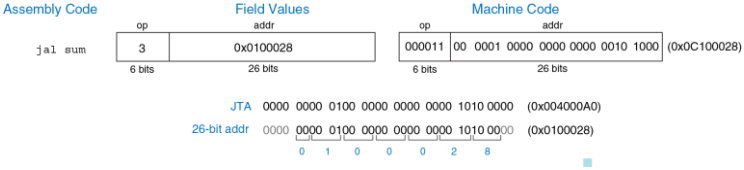


Рис. 6.30 Машинный код инструкции `jal`

Так как четыре старших бита адреса перехода берутся из значения PC + 4, то длина такого перехода ограничена. Ограничения длины перехода для инструкций ветвления и перехода (т.е. условного и безусловного переходов) исследуются в **упражнениях 6.29–6.32**. Все инструкции типа *J* (`j` и `jal`) используют псевдопрямую адресацию.

Обратите внимание, что инструкция безусловного перехода по регистру (`jr`) не является инструкцией типа *J*. Она является инструкцией типа *R* и выполняет переход по 32-битному адресу, находящемуся в регистре `rs` (прим. переводчика: то же самое относится и к инструкции безусловного перехода по регистру с возвратом `jalr` – см. **Приложение B**).

6.6 КАМЕРА, МОТОР! КОМПИЛИРУЕМ, АССЕМБЛИРУЕМ И ЗАГРУЖАЕМ

Ранее мы показали, как небольшие фрагменты кода, написанного на языке высокого уровня, транслируются в ассемблерный и машинный коды. В этом разделе мы рассмотрим, как происходит компиляция и ассемблирование целой программы, написанной на языке высокого уровня, и покажем, как загрузить ее в память компьютера для выполнения.

Мы начнем с рассмотрения карты памяти MIPS, описывающей расположение кода, данных и стека в памяти. Затем на примере покажем этапы выполнения кода программы.

6.6.1 Карта памяти

Так как архитектура MIPS использует 32-битные адреса, то размер адресного пространства составляет 2^{32} байта = 4 гигабайта (Гбайт). Адреса слов кратны 4 и располагаются в промежутке от 0 до 0xFFFFFFFC. На **Рис. 6.31** изображена карта памяти MIPS. Адресное пространство разделено на четыре части, или сегмента: сегмент кода, сегмент глобальных данных, сегмент динамических данных и зарезервированный сегмент. Эти сегменты рассматриваются в следующих разделах.

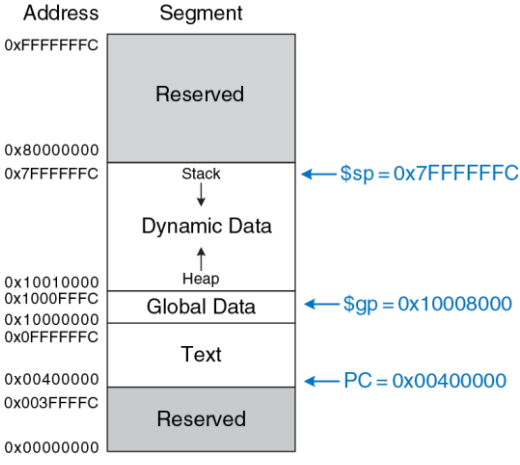


Рис. 6.31 **Карта памяти MIPS**

Грейс Хоппер, 1906-1992

Окончила Йельский университет со степенью доктора философии по математике (англ.: Ph.D., западный аналог степени кандидата математических наук). Во время работы на корпорацию Remington Rand разработала первый компилятор. Сыграла важную роль в разработке языка программирования COBOL. Будучи офицером ВМФ, получила множество наград, в том числе медаль за победу во Второй мировой войне и медаль за службу национальной обороне.

Сегмент кода

Сегмент кода (англ.: text segment) содержит машинные команды исполняемой программы. Его размер достаточен для размещения почти 256 Мбайт кода. Обратите внимание, что четыре старших бита адреса в сегменте кода всегда равны нулю, что позволяет использовать инструкцию `j` для перехода по любому адресу в программе.

Сегмент глобальных данных

Сегмент глобальных данных (англ.: global data segment) содержит глобальные переменные, которые, в отличие от локальных переменных, находятся в области видимости всех функций программы. Глобальные переменные инициализируются при загрузке программы, но до начала ее выполнения. В программе на языке Си эти переменные объявляются вне функции `main` и доступны для всех функций. Размер этого сегмента позволяет разместить 64 Кбайт глобальных переменных.

Доступ к глобальным переменным осуществляется при помощи глобального указателя (`$gp`), который инициализируется значением 0x100080000. В отличие от указателя стека (`$sp`), `$gp` не меняется во время выполнения программы. Любая глобальная переменная доступна при помощи 16-битного положительного или отрицательного смещения относительно `$gp`. Во время ассемблирования смещение

уже известно, так что для доступа к глобальным переменным можно использовать режим базовой адресации с константными смещениями.

Сегмент динамических данных

Сегмент динамических данных (англ.: dynamic data segment) содержит стек и кучу. В момент запуска программы этот сегмент не содержит данных – они динамически выделяются и освобождаются в нем в процессе выполнения программы. Сегмент динамических данных – это самый большой сегмент памяти, используемый программой; его размер составляет почти 2 Гбайт.

Как обсуждалось в **разделе 6.4.6**, стек используется для сохранения и восстановления регистров, используемых функциями, а также хранения локальных переменных, таких как массивы. Стек растет вниз от верхней границы сегмента динамических данных (0x7FFFFFFC), а доступ к кадрам стека осуществляется в режиме очереди LIFO («последним пришел – первым ушел»).

Куча (англ.: heap) хранит блоки памяти, динамически выделяемые программе во время работы. В языке C выделение памяти осуществляется функцией `malloc`; в C++ и Java для этого служит функция `new`. Как и в случае кучи одежды на полу комнаты в общежитии, данные, находящиеся в куче, можно использовать и

выбрасывать в произвольном порядке. Куча растет вверх от нижней границы сегмента динамических данных.

Если стек и куча прорастут друг в друга, данные программы могут быть повреждены. Функция выделения памяти стремится избежать этой ситуации. Она возвращает *ошибку нехватки памяти* (англ.: out-of-memory error), если свободной памяти недостаточно для размещения новых динамических данных.

Зарезервированный сегмент

Зарезервированный сегмент используется операционной системой и не может использоваться непосредственно программой. Часть зарезервированной памяти используется для прерываний (см. **раздел 7.7**) и для отображения устройств ввода-вывода в адресное пространство (см. **раздел 8.5**).

6.6.2 Трансляция и запуск программы

На **Рис. 6.32** показаны этапы, необходимые для трансляции в машинный язык и начала выполнения программы, написанной на языке высокого уровня. Высокоуровневый код компилируется в код на языке ассемблера, который затем ассемблируется в машинный код и сохраняется в виде объектного файла. *Компоновщик,* также называемый *редактором связей* или *линкером* (англ.: linker), объединяет полученный объектный код с объектным кодом библиотек и других файлов, в результате чего получается готовая к исполнению программа. На практике, большинство компиляторных пакетов выполняют все три шага: компиляцию, ассемблирование и компоновку. Наконец, загрузчик загружает программу в память и запускает ее. В оставшейся части этого раздела мы более подробно рассмотрим эти этапы на примере простой программы.

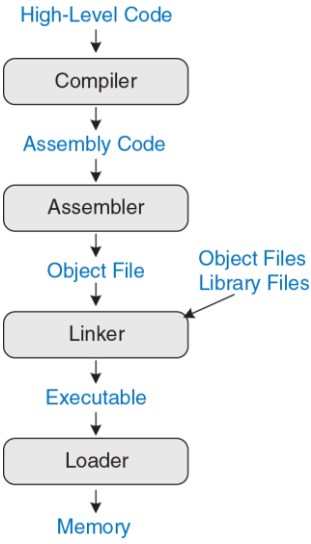


Рис. 6.32 **Этапы трансляции и запуска программы**

Пример кода 6.30 КОМПИЛЯЦИЯ ПРОГРАММЫ

Код на языке высокого уровня

```
int f, g, y; // global variables
int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
int sum(int a, int b) {
  return (a + b);
}
```

Код на языке ассемблера MIPS

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4  # make stack frame
  sw   $ra, 0($sp)   # store $ra on stack
  addi $a0, $0, 2    # $a0 = 2
  sw   $a0, f        # f = 2
  addi $a1, $0, 3    # $a1 = 3
  sw   $a1, g        # g = 3
```

```
  jal  sum           # call sum function
  sw   $v0, y        # y = sum(f, g)
  lw   $ra, 0($sp)   # restore $ra from stack
  addi $sp, $sp, 4   # restore stack pointer
  jr   $ra           # return to operating system
sum:
  add  $v0, $a0, $a1 # $v0 = a + b
  jr   $ra           # return to caller
```

Этап 1: Компиляция

Компилятор транслирует код высокого уровня в код на языке ассемблера. В **примере кода 6.30** показана простая программа на языке высокого уровня, содержащая три глобальные переменные и две функции, а также ассемблерный код, сгенерированный типичным компилятором. Ключевые слова `.data` и `.text` – это ассемблерные директивы, указывающие на начало сегментов данных и кода соответственно. Для обозначения глобальных переменных `f`, `g` и `y` используются метки. Места для их хранения будут определены ассемблером. На данный момент они остаются в коде в виде символов.

Этап 2: Трансляция

Ассемблер транслирует код на языке ассемблера в объектный файл, содержащий код на машинном языке. Для трансляции кода ассемблер делает два прохода. Во время первого прохода ассемблер назначает

адреса командам и находит все символы, такие как метки и имена глобальных переменных. После первого прохода ассемблера код выглядит следующим образом:

```
0x00400000 main: addi $sp, $sp, -4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

Имена и адреса символов хранятся в *таблице символов*. Таблица символов для нашего примера приведена в **Табл. 6.4**. Адреса символов заполняются после первого прохода, когда адреса меток уже известны. Глобальным переменным присваиваются адреса из сегмента глобальных данных, начиная с адреса 0x10000000.

Во время второго прохода ассемблер генерирует машинный код. Адреса глобальных переменных и меток берутся из таблицы символов.

Код на машинном языке и таблица символов сохраняются в объектном файле.

Табл. 6.4 Таблица символов

Символ	Адрес
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Этап 3: Компоновка

Большие программы обычно содержат много файлов. Если программист изменяет один из этих файлов, то перекомпилировать и заново транслировать все остальные файлы выходит довольно затратно. Например, программы часто вызывают функции из библиотечных файлов, которые почти никогда не меняются, а соответствующие объектные файлы не нуждаются в обновлении.

Работа компоновщика заключается в том, чтобы объединить все объектные файлы в один-единственный файл с машинным кодом, который называется исполняемым файлом. Компоновщик перемещает данные и команды в объектных файлах так, чтобы они не наслаивались

друг на друга. Он использует информацию из таблицы символов для коррекции адресов перемещаемых глобальных переменных и меток.

В нашем примере только один объектный файл, поэтому никакого перемещения не требуется. На **Рис. 6.33** показан полученный исполняемый файл. Он состоит из трех секций: заголовка, сегмента кода и сегмента данных. Заголовок исполняемого файла содержит информацию о размерах сегмента кода (т.е. об объеме кода) и размерах сегмента данных (т.е. о количестве глобально объявленных данных). Все размеры приведены в байтах. Команды в сегмент кода приведены в том же порядке, в котором они расположены в памяти.

На рисунке рядом с машинным кодом показаны команды в виде, удобном для восприятия и интерпретации человеком. Исполняемый файл содержит только машинные команды. Сегмент данных задает адреса всех глобальных переменных. Доступ к глобальным переменным осуществляется при помощи базовой адресации относительно адреса, определяемого глобальным указателем `$gp`. Например, первая команда `sw $a0, 0x8000 ($gp)` присваивает значение 2 глобальной переменной `f`, которая размещена в памяти по адресу 0x10000000. Помните, что смещение 0x8000 – это 16-битное число со знаком, которое после знакового расширения до 32 битов прибавляется к базовому адресу, находящемуся в регистре `$gp`. Таким

образом, `$gp` + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000 – это адрес переменной `f`.

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, –4
sw   $ra, 0($sp)
addi $a0, $0, 2
sw   $a0, 0x8000($gp)
addi $a1, $0, 3
sw   $a1, 0x8004($gp)
jal  0x0040002C
sw   $v0, 0x8008($gp)
lw   $ra, 0($sp)
addi $sp, $sp, –4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Рис. 6.33 Исполняемый файл

Этап 4: Загрузка

Операционная система загружает программу, считывая сегмент кода исполняемого файла с устройства хранения данных (обычно это жесткий диск) в сегмент кода памяти. Операционная система сначала присваивает регистру `$gp` значение 0x10008000, равное середине сегмента глобальных данных, затем присваивает регистру `$sp` значение 0x7FFFFFFC, равное верхней границе сегмента динамических данных, после чего выполняет команду `jal` 0x00400000 для перехода к началу программы. На **Рис. 6.34** показана карта памяти в начале выполнения программы.

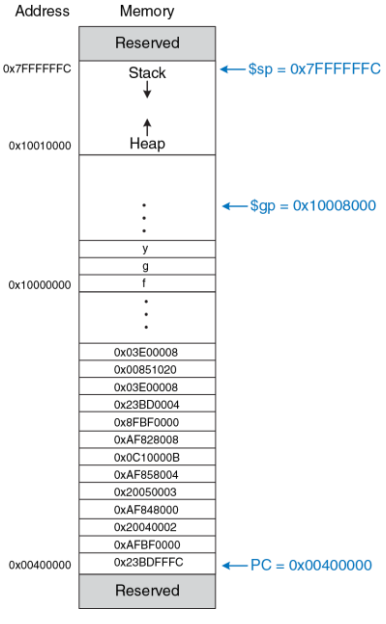


Рис. 6.34 Исполняемый файл, загруженный в память

6.7 ДОБАВОЧНЫЕ СВЕДЕНИЯ*

В этом разделе рассматривается несколько дополнительных тем, которые не вписываются в другие места этой главы. Они включают в себя псевдокоманды, исключения, знаковые и беззнаковые арифметические команды и команды арифметики с плавающей точкой.

6.7.1 Псевдокоманды

Если какая-либо команда не доступна в наборе команд MIPS, то это, вероятно, связано с тем, что та же операция может быть выполнена одной или несколькими существующими командами MIPS. Помните, что MIPS является компьютером с сокращенным набором команд (RISC), поэтому размер команд и сложность аппаратного обеспечения минимизированы путем использования лишь небольшого количества команд.

Тем не менее, MIPS определяет псевдокоманды, которые на самом деле не являются частью набора команд, но часто используются программистами и компиляторами. При преобразовании в машинный код псевдокоманды транслируются в одну или несколько команд MIPS.

Табл. 6.5 содержит примеры псевдокоманд и соответствующих им команд MIPS. Например, вызов псевдокоманды `li` загружает в регистр 32-битную константу, используя комбинацию команд `lui` и `ori`.

Псевдокоманда «нет операции» (`nop`) не выполняет никаких действий. После ее выполнения счетчик команд увеличивается на 4, но никакие другие регистры и содержимое памяти не изменяются. Машинный код команды `nop` – 0x00000000.

Табл. 6.5 Псевдокоманды

Псевдокоманда	Соответствующие команды MIPS
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s2, $s1`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Некоторым псевдокомандам требуется временный регистр для промежуточных вычислений. Например, псевдокоманда `beq $t2,` $\texttt{imm}_{15:0}$`, Loop` сравнивает `$t2` с 16-битным непосредственным операндом. Для хранения непосредственного операнда этой псевдокоманде нужен дополнительный временный регистр. Ассемблер использует для этих целей временный регистр `$at`. В **Табл. 6.6** показано, как ассемблер использует `$at` для преобразования псевдокоманд в существующие команды MIPS.

Табл. 6.6 Псевдокоманды, использующие регистр `$at`

Псевдокоманда	Соответствующие команды MIPS
`beq $t2, imm15:0, Loop`	`addi $at, $0, imm15:0` `beq $t2, $at, Loop`

В **упражнениях 6.38** и **6.39** вам будет предложено реализовать и другие псевдокоманды, такие как *циклический сдвиг влево* (`rol`) или *циклический сдвиг вправо* (`ror`).

6.7.2 Исключения

Исключение (англ.: exception) выглядит как незапланированный вызов функции, которая переходит по новому адресу. Исключения могут быть вызваны как аппаратным обеспечением, так и программой. Например, процессор может получить уведомление о том, что пользователь нажал клавишу на клавиатуре. В этом случае процессор может приостановить выполнение программы, определить нажатую клавишу и сохранить информацию об этом, после чего возобновить выполнение прерванной программы. Исключения, вызванные устройствами ввода-вывода, такими как клавиатура, часто называют *прерываниями* (англ.: interrupt). С другой стороны, ошибка может возникнуть в самой программе, например, из-за использования неопределенной команды. В этом

случае программа совершает переход к коду операционной системы (ОС), который может завершить выполнение программы-нарушителя. Исключения, возникающие в программах, иногда называют ловушками (англ.: traps). Другими причинами исключений могут быть деление на ноль, попытка чтения несуществующей памяти, аппаратные сбои, точка останова отладчика (англ.: debugger breakpoint) и арифметическое переполнение (см. **раздел 6.7.3**).

Процессор запоминает причину исключения и значение счетчика команд в момент его возникновения, а затем передает управление функции обработчика исключений (англ.: exception handler). Обработчик исключений – это код, являющийся, как правило, частью операционной системы, который выясняет причину возникновения исключения и реагирует соответствующим образом (например, чтением клавиатуры в ответ на прерывание). Затем управление возвращается программе, выполнявшейся до возникновения исключения. В процессорах MIPS обработчик исключений всегда находится по адресу 0x80000180 (прим. переводчика: в более поздних версиях архитектуры MIPS были введены более гибкие схемы обработки исключений). Когда возникает исключение, процессор всегда переходит по этому адресу, независимо от причины исключения.

В архитектуре MIPS для записи причины исключения используется регистр специального назначения, называемый *регистром причины*

исключения (англ.: Cause register). При возникновении исключения в этот регистр записывается соответствующий ему код, показанный в **Табл. 6.7**. Обработчик исключений читает регистр причины, чтобы определить, как именно нужно обрабатывать исключение. В некоторых других архитектурах вместо регистра причины используются индивидуальные обработчики для каждого исключения.

Табл. 6.7 Коды причины исключения

Исключение	Код, записываемый в регистр причины
Аппаратное прерывание	0x00000000
Системный вызов (system call)	0x00000020
Точка останова (breakpoint) или деление на 0	0x00000024
Неопределенная команда	0x00000028
Арифметическое переполнение	0x00000030

Архитектура MIPS также использует регистр специального назначения EPC (англ.: Exception Program Counter) для хранения значения счетчика команд прерванной программы на время обработки исключения. После обработки исключения процессор возвращается к выполнению команды по адресу, сохраненному в EPC. Этот процесс аналогичен использованию $ra для хранения старого значения счетчика команд во время выполнения команды jal.

Регистр EPC и регистр причины исключения не являются частью регистрового файла MIPS. Команда загрузки из регистра сопроцессора 0 (`mfc0`) позволяет скопировать содержимое этих и других регистров специального назначения в один из регистров общего назначения. Сопроцессор 0 является той частью процессора MIPS, которая выполняет системные функции, то есть обрабатывает исключения и проводит диагностику процессора. Например, команда `mfc0 $t0, Cause` копирует регистр причины исключения в `$t0`.

Команды `syscall` и `break` вызывают срабатывание ловушек (программных исключений) для выполнения системных вызовов (англ.: system calls) и использования точек останова при отладке. Обработчик исключений использует регистр EPC для поиска команды, вызвавшей исключение, и может определить тип системного вызова или точки останова, анализируя ее поля.

Подводя итоги, возникновение исключения приводит к передаче управления обработчику исключений. Обработчик исключений сохраняет регистры на стек, а затем использует команду `mcf0` для определения причины исключения и выполнения соответствующих действий. Когда обработка исключения закончена, то регистры восстанавливаются из стека, а адрес возврата копируется из регистра

EPC в регистр $k0 при помощи mfc0, после чего при помощи команды jr $k0 происходит возврат к прерванной программе.

> Регистры $k0 и $k1 включены в набор регистров MIPS. Они зарезервированы операционной системой для обработки исключений. Их не нужно сохранять и восстанавливать при возникновении исключений.

6.7.3 Команды для чисел со знаком и без знака

Напомним, что двоичное число может быть со знаком или без знака. Для представления чисел со знаком архитектура MIPS использует дополнительный код. Некоторые команды MIPS имеют две версии – одну для чисел со знаком и вторую для чисел без знака. Примером таких команд являются команды сложения и вычитания, умножения и деления, команды сравнения и команды загрузки части слова.

Сложение и вычитание

Вне зависимости от того, со знаком число или без знака, сложение и вычитание всегда выполняются одинаково. Однако интерпретация результатов при этом отличается.

Как упоминалось в **разделе 1.4.6**, если произвести сложение двух больших чисел со знаком, то можно получить результат с некорректным знаком. Например, если сложить два огромных положительных числа,

то получится отрицательный результат: 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE = −2. Аналогично, сложив два огромных отрицательных числа, получим положительное: 0x80000001 + 0x80000001 = 0x00000002. Такая ситуация называется арифметическим переполнением.

Язык C игнорирует арифметическое переполнение, но другие языки, такие как Фортран, требуют, чтобы программа была осведомлена о возникновении такой ситуации. Как упоминалось в **разделе 6.7.2**, при возникновении арифметического переполнения в процессоре MIPS случается исключение. Программа может решить, что делать с переполнением (например, можно повторить вычисления с большей точностью, чтобы избежать переполнения), а затем вернуться туда, где она была прервана.

MIPS поддерживает сложение и вычитание чисел со знаком и без знака. Для чисел со знаком предназначены команды `add`, `addi` и `sub`. Для чисел без знака используются команды `addu`, `addiu` и `subu`. Эти версии команд идентичны за исключением того, что версии для чисел без знака не вызывают исключение при переполнении, а версии для чисел со знаком вызывают. Поскольку программы на языке C игнорируют исключения, то фактически используются только версии команд для чисел без знака.

Умножение и деление

Результаты операций умножения и деления зависят от того, учитывают они знак или нет. Например, если интерпретировать 0xFFFFFFFF как число без знака, то оно будет представлять собой очень большую величину, но как знаковое число оно имеет значение –1. Следовательно, произведение 0xFFFFFFFF × 0xFFFFFFFF будет равно 0xFFFFFFFE00000001, если используются беззнаковые числа, и 0x0000000000000001 при использовании чисел со знаком.

Таким образом, команды умножения и деления также существуют в двух версиях – для знаковых (`mult` и `div`) и беззнаковых (`multu` и `divu`) чисел соответственно.

Команды сравнения

Команды сравнения могут сравнивать либо значения двух регистров (команда `slt`), либо значение регистра с непосредственным операндом (`slti`). У этих команд тоже есть версии для чисел со знаком (`slt` и `slti`) и без знака (`sltu` и `sltiu`). При сравнении чисел со знаком число 0x80000000 – наименьшее возможное (самое большое по модулю отрицательное число, которое можно представить в дополнительном коде). Но если воспринимать его как беззнаковое число, то 0x80000000 больше, чем 0x7FFFFFFF, но меньше, чем 0x80000001, потому что все беззнаковые числа положительные.

Будьте осторожны, потому что команда `sltiu` сначала знаково расширяет непосредственный операнд, а потом уже рассматривает его как беззнаковое число. Например, команда `sltiu $s0, $s1, 0x8042` сравнит значение из регистра `$s1` с числом 0xFFFF8042, считая, что это большое положительное число.

Команды загрузки

Как было сказано в **разделе 6.4.5**, команды загрузки байтов имеют версии для чисел со знаком (`lb`) и без знака (`lbu`). Команда `lb` загружает байт данных из памяти в младший байт регистра и заполняет оставшиеся биты регистра знаковым битом загруженного байта. Команда `lbu` заполняет оставшиеся биты нулями. Аналогично работают и команды загрузки полуслова (`lh` и `lhu`), которые загружают по два байта в младшую часть 32-битного регистра и заполняют оставшиеся биты либо знаковым битом, либо нулями.

6.7.4 Команды для работы с числами с плавающей точкой

В архитектуре MIPS предусмотрено использование опционального сопроцессора для чисел с плавающей точкой (англ.: floating point numbers), который называется Сопроцессором 1. В ранних реализациях процессоров MIPS этот сопроцессор размещался в отдельной микросхеме, которую мог приобрести каждый, кому требовалась

высокая производительность при работе с числами с плавающей точкой. В большинстве современных реализаций процессор MIPS и его Сопроцессор 1 находятся на одном кристалле.

Архитектура MIPS предусматривает наличие тридцати двух 32-битных регистров для чисел с плавающей точкой (`$f0`–`$f31`). Это не те же самые регистры, которые мы использовали до сих пор. MIPS поддерживает числа с плавающей точкой одинарной и двойной точности в IEEE-формате (примечание переводчика: имеется в виду стандарт IEEE 754). Числа двойной точности (64-битные) размещаются в парах 32-битных регистров, так что для операций с такими числами доступны только 16 четных регистров (`$f0`, `$f2`, `$f4`, `...`, `$f30`). В соответствии с соглашением, принятым в архитектуре MIPS, у этих регистров разное назначение, показанное в **Табл. 6.8**.

Табл. 6.8 Набор регистров с плавающей точкой

Название	Номер	Назначение
`$fv0–$fv1`	0, 2	Значение, возвращаемое функцией
`$ft0–$ft3`	4, 6, 8, 10	Временные переменные
`$fa0–$fa1`	12, 14	Аргументы функции
`$ft4–$ft5`	16, 18	Временные переменные
`$fs0–$fs5`	20, 22, 24, 26, 28, 30	Сохраняемые переменные

Код операции у всех команд с плавающей точкой равен 17 (10001_2). Для определения типа команды в них должны присутствовать поля `funct` и `cop` (от англ. coprocessor).

На **Рис. 6.35** показан формат команд типа *F*, используемый для чисел с плавающей точкой. Поле `cop` равно 16 (10000_2) для команд одинарной точности и 17 (10001_2) для команд двойной точности. Аналогично командам типа R, у команд типа *F* есть два операнда-источника (`fs` и `ft`) и один операнд-назначение (`fd`).

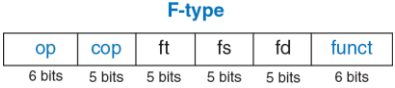


Рис. 6.35 **Формат команд типа *F***

Команды одинарной и двойной точности различаются суффиксом в мнемонике (`.s` и `.d` соответственно). Команды с плавающей точкой включают сложение (`add.s`, `add.d`), вычитание (`sub.s`, `sub.d`), умножение (`mul.s`, `mul.d`), деление (`div.s`, `div.d`), изменение знака (`neg.s`, `neg.d`) и вычисление модуля (`abs.s`, `abs.d`).

В тех случаях, когда ветвление зависит от условия, вычисляемого с использованием чисел с плавающей точкой, оно делится на две части. Сначала команда сравнения устанавливает или сбрасывает

специальный флаг условия `fpcond` (от англ. floating point condition flag). После этого команда ветвления проверяет этот флаг и в зависимости от его состояния осуществляет переход. Команды сравнения включают команды проверки на равенство (`c.seq.s/c.seq.d`), проверки на то, что один операнд меньше другого (`c.lt.s/c.lt.d`) или меньше или равен другому (`c.le.s/c.le.d`). Команды ветвления `bc1f` и `bc1t` осуществляют переход, если флаг `fpcond` имеет значение ЛОЖЬ или ИСТИНА соответственно. Другие варианты сравнений, такие как проверка на неравенство или на то, что один операнд больше другого, или больше или равен другому, осуществляются при помощи команд с суффиксами `seq`, `lt`, `le` и последующей командой `bc1f`.

Регистры с плавающей точкой загружаются из памяти и записываются в память при помощи команд `lwc1` и `swc1` соответственно. Эти команды перемещают по 32 бита, так что для работы с числами двойной точности необходимо использовать по две такие команды.

6.8 ЖИВОЙ ПРИМЕР: АРХИТЕКТУРА X86*

Практически все персональные компьютеры используют процессоры с архитектурой x86. Архитектура x86, также называемая IA-32 – это 32-разрядная архитектура, изначально разработанная компанией Intel. Компания AMD также продает x86-совместимые микропроцессоры.

Архитектура x86 имеет долгую и запутанную историю, которая берет начало в 1978 году, когда Intel объявила о разработке 16-битного микропроцессора 8086. Компания IBM выбрала 8086 и его брата 8088 для своих первых персональных компьютеров (ПК). В 1985 году Intel представила 32-разрядный микропроцессор 80386, который был обратно совместим с 8086 и мог запускать программы, разработанные для более ранних ПК. Процессорные архитектуры, совместимые с 80386, называют x86-совместимыми архитектурами. Процессоры Pentium, Core и Athlon – наиболее известные x86-совместимые процессоры. В **разделе 7.9** эволюция этих процессоров будет описана более подробно.

Различные группы разработчиков в Intel и AMD на протяжении многих лет добавляли множество новых команд и возможностей в устаревшую архитектуру. В результате она выглядит гораздо менее элегантно, чем MIPS. Как объясняют Паттерсон и Хеннесси: «эта архитектура похожа на лоскутное одеяло, ее сложно понять и невозможно полюбить».

Тем не менее, совместимость программного обеспечения гораздо важнее технической элегантности, так что x86 является де-факто стандартом для ПК на протяжении более чем двух десятилетий. Каждый год продается свыше 100 миллионов x86-совместимых микропроцессоров. Это огромный рынок, оправдывающий ежегодные затраты на улучшение этих процессоров, превышающие пять миллиардов долларов.

Архитектура x86 является примером CISC-архитектуры (от англ.: Complex Instruction Set Computer – компьютер с полным набором команд). В отличие команд в RISC-архитектурах, таких как MIPS, каждая CISC-команда способна произвести больше работы. Из-за этого программы для CISC-архитектур обычно состоят из меньшего количества команд. Кодировки команд были подобраны так, чтобы обеспечивать наибольшую компактность кода – это требовалось в те времена, когда стоимость оперативной памяти было гораздо выше, чем сейчас. Команды имеют переменную длину, которая зачастую меньше 32 бит. Недостаток же состоит в том, что сложные команды трудно дешифровать, к тому же они, как правило, работают медленнее.

В этом разделе мы ознакомимся с архитектурой x86. Наша цель состоит не в том, чтобы сделать вас программистом на языке ассемблера x86, а, скорее, в том, чтобы проиллюстрировать некоторые сходства и различия между x86 и MIPS. Мы считаем, что это

интересно – посмотреть, как работает архитектура x86. Тем не менее, материалы из этого раздела не требуются, чтобы понять оставшуюся часть книги. Основные отличия между x86 и MIPS приведены в Табл. 6.9.

Табл. 6.9 **Основные отличия между MIPS и x86**

Характеристики	MIPS	x86
Количество регистров	32, общего назначения	8, некоторые ограничения по использованию
Количество операндов	3 (2 источника, 1 назначение)	2 (1 источник, 1 источник/назначение)
Расположение операндов	Регистры или непосредственные операнды	Регистры, непосредственные операнды или память
Размер операнда	32 бита	8, 16 или 32 бита
Коды условий	Нет	Да
Типы команд	Простые	Простые и сложные
Размер команд	Фиксированный, 4 байта	Переменный, 1–15 байтов

6.8.1 Регистры x86

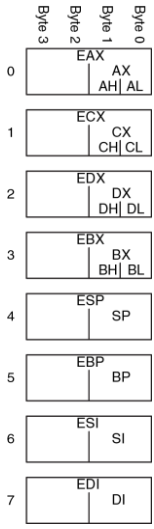


Рис. 6.36 Регистры архитектуры x86

У микропроцессора 8086 было восемь 16-битных регистров. Некоторые из них позволяли осуществлять доступ отдельно к старшим и младшим восьми битам. Когда была представлена 32-битная архитектура 80386, регистры были просто расширены до 32 бит. Эти регистры называются `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` и `EDI`. Для обеспечения обратной совместимости была оставлена возможность получить отдельный доступ к их младшим 16 битам, а для некоторых регистров – и к двум младшим байтам, как показано на **Рис. 6.36**.

Эти восемь регистров можно, за некоторым исключением, считать регистрами общего назначения. Некоторые команды не могут использовать некоторые из них. Другие команды всегда записывают результат в определенные регистры. Также как регистр `$sp` в MIPS, регистр `ESP`, как правило, используется как указатель стека.

Счетчик команд в архитектуре x86 называется `EIP` (от англ. extended instruction pointer). Аналогично

счетчику команд в архитектуре MIPS, он увеличивается при переходе от одной команды к другой, а также может быть изменен командами ветвлений, безусловных переходов и вызова функций.

6.8.2 Операнды x86

Команды MIPS всегда производят действия либо с регистрами, либо с непосредственными операндами. Для перемещения данных между памятью и регистрами необходимы явные команды загрузки и сохранения. Команды x86, напротив, могут работать как с регистрами и непосредственными операндами, так и с внешней памятью. Это частично компенсирует небольшой набор регистров.

Команды MIPS обычно содержат три операнда: два операнда-источника и один операнд-назначение. Команды x86 содержат только два операнда: операнд-источник и операнд-источник/назначение. Следовательно, команда x86 всегда записывает результат на место одного из операндов. В **Табл. 6.10** перечислены поддерживаемые комбинации расположения операндов в командах x86. Как видно, возможны любые комбинации, исключая память–память.

Табл. 6.10 Расположение операндов

Источник/ Назначение	Источник	Пример	Выполняемая функция
Регистр	Регистр	`add EAX, EBX`	`EAX <- EAX + EBX`
Регистр	Непосредственный операнд	`add EAX, 42`	`EAX <- EAX + 42`
Регистр	Память	`add EAX, [20]`	`EAX <- EAX + Mem[20]`
Память	Регистр	`add [20], EAX`	`Mem[20] <- Mem[20] + EAX`
Память	Непосредственный операнд	`add [20], 42`	`Mem[20] <- Mem[20] + 42`

Аналогично MIPS, архитектура x86 имеет 32-битное пространство памяти с побайтовой адресацией. Однако x86 поддерживает намного больше различных режимов адресации памяти. Расположение ячейки памяти задается при помощи комбинации регистра базового адреса, смещения и регистра масштабируемого индекса (см. Табл. 6.11).

Табл. 6.11 Режимы адресации памяти

Пример	Назначение	Комментарий
`add EAX, [20]`	`EAX <- EAX + Mem[20]`	Смещение (displacement)
`add EAX, [ESP]`	`EAX <- EAX + Mem[ESP]`	Базовая адресация
`add EAX, [EDX+40]`	`EAX <- EAX + Mem[EDX+40]`	Базовая адресация + смещение
`add EAX, [60+EDI*4]`	`EAX <- EAX + Mem[60+EDI*4]`	Смещение + масштабируемый индекс
`add EAX, [EDX+80+EDI*2]`	`EAX <- EAX + Mem[EDX+80+EDI*2]`	Базовая адресация + смещение + масштабируемый индекс

Смещение может иметь 8-, 16- или 32-битное значение. Регистр масштабируемого индекса может быть умножен на 1, 2, 4 или 8. Режим базовой адресации со смещением аналогичен режиму относительной адресации в MIPS, используемому для команд загрузки и сохранения. Масштабируемый индекс обеспечивает простой способ доступа к массивам и структурам с 2-, 4- или 8-байтовыми элементами без необходимости использовать команды для явного расчета адресов.

В то время как MIPS всегда оперирует с 32-битными словами данных, команды x86 могут использовать 8-, 16- или 32-битные данные. Это проиллюстрировано в Табл. 6.12.

Табл. 6.12 **Инструкции, использующие 8-, 16- или 32-битные операнды**

Пример	Назначение	Размер операндов
`add AH, BL`	`AH <- AH + BL`	8 бит
`add AX, -1`	`AX <- AX + 0xFFFF`	16 бит
`add EAX, EDX`	`EAX <- EAX + EDX`	32 бита

6.8.3 Флаги состояния

Как и большинство архитектур CISC, x86 использует флаги состояния (также называемые кодами условий) для принятия решений о ветвлениях и отслеживания переносов и арифметических переполнений. В архитектуре x86 используется 32-битный регистр `EFLAGS`, в котором хранятся флаги состояния. Назначение некоторых битов из регистра `EFLAGS` приведено в Табл. 6.13. Оставшиеся биты используются операционной системой.

Табл. 6.13 Некоторые биты регистра EFLAGS

Название	Назначение
CF (Carry Flag, флаг переноса)	Показывает, что при выполнении последней арифметической операции результат вышел за пределы разрядной сетки. Указывает на то, что произошло переполнение при беззнаковых вычислениях. Также используется как флаг переноса при работе с числами, разрядность которых превышает разрядность архитектуры
ZF (Zero Flag, флаг нуля)	Показывает, что результат последней операции равен нулю
SF (Sign Flag, флаг знака)	Показывает, что результат последней операции был отрицательным (старший бит результата равен 1).
OF (Overflow Flag, флаг переполнения)	Показывает, что произошло переполнение при вычислениях со знаковыми числами в дополнительном коде

Архитектурное состояние процессора x86 включает в себя EFLAGS, а также восемь регистров и EIP.

6.8.4 Команды x86

Архитектура x86 имеет большую, чем у MIPS, систему команд. В Табл. 6.14 показаны некоторые из команд общего назначения. Система команд x86 также включает команды обработки чисел с плавающей точкой и коротких векторов упакованных данных. Операнд-назначение обозначен как `D` (регистр или ячейка памяти), а операнд-источник обозначен как `S` (регистр, непосредственный операнд или ячейка памяти).

Табл. 6.14 Некоторые инструкции x86

Инструкция	Назначение	Функция
`ADD/SUB`	Сложение/вычитание	`D = D + S / D = D − S`
`ADDC`	Сложение с переносом	`D = D + S + CF`
`INC/DEC`	Увеличение/уменьшение	`D = D + 1 / D = D − 1`
`CMP`	Сравнение	Установить флаги по результатам `D − S`
`NEG`	Инверсия	`D = −D`
`AND/OR/XOR`	Логическое «И/ИЛИ/ИСКЛЮЧАЮЩЕЕ ИЛИ»	`D = D операция S`
`NOT`	Логическое НЕ	$D = \overline{D}$
`IMUL/MUL`	Знаковое/беззнаковое умножение	`EDX:EAX = EAX ×D`

Инструкция	Назначение	Функция
IDIV/DIV	Знаковое/беззнаковое деление	EDX:EAX/D EAX = частное; EDX = остаток
SAR/SHR	Арифметический/логический сдвиг вправо	D = D >>> S / D = D >> S
SAL/SHL	Сдвиг влево	D = D << S
ROR/ROL	Циклический сдвиг вправо/влево	Циклически сдвинуть D на S разрядов
RCR/RCL	Циклический сдвиг вправо/влево через бит переноса	Циклически сдвинуть CF и D на S разрядов
BT	Проверка бита	CF = D[S] (бит номер S из D)
BTR/BTS	Проверить бит и сбросить/установить его	CF = D[S]; D[S] = 0 / 1
TEST	Установить флаги по результатам проверки бит	Установить флаги по результатам D И S
MOV	Скопировать операнд	D = S
PUSH	Поместить на стек	ESP = ESP −4; Mem[ESP] = S
POP	Прочитать из стека	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	Сбросить/установить флаг переноса	CF = 0 / 1

Инструкция	Назначение	Функция
JMP	Безусловный переход	Переход по относительному адресу: EIP = EIP + S Переход по абсолютному адресу: EIP = S
Jcc	Ветвление (условный переход)	Если установлен флаг, то EIP = EIP + S
LOOP	Проверка условия цикла	ECX = ECX – 1 Если ECX ≠ 0, то EIP = EIP + imm
CALL	Вызов функции	ESP = ESP – 4; MEM[ESP] = EIP; EIP = S
RET	Возврат из функции	EIP = MEM[ESP]; ESP = ESP + 4

Обратите внимание, что некоторые команды всегда производят действия только с определенными регистрами. Например, умножение двух 32-битных чисел всегда использует в качестве одного из источников EAX и всегда записывает 64-битный результат в EDX и EAX. Команда LOOP всегда хранит счетчик итераций цикла в ECX, а команды PUSH, POP, CALL и RET используют указатель вершины стека ESP.

Команды условного перехода проверяют значения флагов и, если выполнено соответствующее условие, осуществляют ветвление.

Эти команды имеют много разновидностей. Например, команда JZ осуществляет переход том в случае, когда флаг нуля (ZF) равен 1, а команда JNZ – когда ZF равен 0. Команды перехода обычно следуют за командами, которые устанавливают флаги, такими как команда сравнения (CMP). В **Табл. 6.15** перечислены некоторые команды условных переходов и то, как на них воздействуют флаги, предварительно установленные командами сравнения.

Табл. 6.15 Некоторые условия ветвлений

Инструкция	Назначение	Действие после CMP D, S
JZ/JE	Ветвление, если ZF = 1	Ветвление, если D = S
JNZ/JNE	Ветвление, если ZF = 0	Ветвление, если D ≠ S
JGE	Ветвление, если SF = OF	Ветвление, если D ≥ S
JG	Ветвление, если SF = OF и ZF = 0	Ветвление, если D > S
JLE	Ветвление, если SF ≠ OF и ZF = 1	Ветвление, если D ≤ S
JL	Ветвление, если SF ≠ OF	Ветвление, если D <S
JC/JB	Ветвление, если CF = 1	
JNC	Ветвление, если CF = 0	
JO	Ветвление, если OF = 1	
JNO	Ветвление, если OF = 0	
JS	Ветвление, если SF = 1	
JNS	Ветвление, если SF = 0	

6.8.5 Кодировка команд x86

Кодировка команд x86 – это тяжелое наследие десятилетий постепенных изменений. В отличие от MIPS, где команды всегда имеют длину 32 бита, длина команды x86 может составлять от 1 до 15 байтов, как показано на Рис. 6.37.[7]

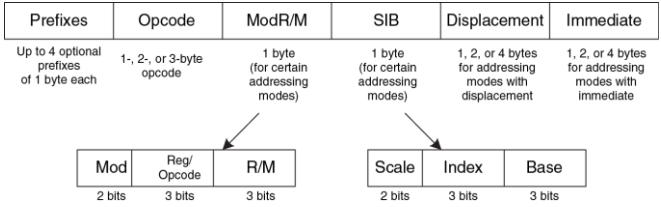


Рис. 6.37 Кодировка команд x86

Код операции (`opcode`) может составлять 1, 2 или 3 байта. Далее следуют четыре дополнительных поля: `ModR/M`, `SIB`, `Displacement` и `Immediate`. Поле `ModR/M` определяет режим адресации. Поле `SIB` определяет коэффициент масштабирования (scale), индексный (index) и базовый (base) регистры в некоторых режимах адресации. Поле

[7] Если использовать все необязательные поля, то можно собрать команду длиной 17 байтов, но x86 имеет ограничение на длину корректной команды, равное 15 байтам.

`Displacement` содержит 1-, 2- или 4-байтовое смещение, используемое в соответствующих режимах адресации. Поле `Immediate` содержит 1-, 2- или 4-байтовую константу для команд, использующих непосредственный операнд. Более того, команда может иметь до четырех однобайтных префиксов, изменяющих ее поведение.

Однобайтовое поле `ModR/M` использует 2-битное поле режима `Mod` и 3-битное поле `R/M` для задания режима адресации одного из операндов. Операнд может находиться в одном из восьми регистров, либо его можно указать при помощи одного из 24 режимов адресации памяти. Из-за ошибок в кодировке регистры `ESP` и `EBP` не могут использоваться как базовый или индексный регистры в некоторых режимах адресации. В поле `Reg` указывается регистр, используемый в качестве второго операнда. Для некоторых команд, не имеющих второго операнда, поле `Reg` используется для хранения трех дополнительных бит кода операции.

В режимах адресации, использующих регистр масштабируемого индекса, байт `SIB` определяет индексный регистр и коэффициент масштабирования (1, 2, 4 или 8). Если при адресации используются базовый адрес и индекс, то `SIB` также определяет регистр базового адреса.

Архитектура MIPS позволяет точно определить тип команды по полям кода операции (`opcode`) и `funct`. Архитектура x86 использует разное количество бит для определения разных команд. Часто используемые команды имеют меньший размер, что уменьшает среднюю длину команд в программе. Некоторые команды могут иметь несколько кодов операций. Например, команда `add AL, imm8` выполняет 8-битное сложение регистра `AL` и непосредственного операнда. Эта команда представляется в виде однобайтового кода операции (0x04) и однобайтового непосредственного операнда. Регистр `A` (`AL`, `AX` или `EAX`) называется *аккумулятором*. С другой стороны, команда `add D, imm8` производит 8-битное сложение непосредственного операнда с операндом `D` и записывает результат в `D`, причем `D` может быть регистром или ячейкой памяти. Эта команда состоит из однобайтового кода операции (0x08), одного или более байтов, определяющих местонахождение `D`, и однобайтового непосредственного операнда `imm8`. Как видите, многие команды имеют более короткие кодировки в том случае, если их результат сохраняется в аккумулятор.

В оригинальном процессоре 8086 в коде операции указывалась разрядность операндов (8 или 16 бит). Когда в процессор 80386 добавили 32-битные операнды, то свободных кодов операции, которые позволили бы добавить новый размер операндов, уже не осталось, поэтому команды, использующие 16-битные и 32-битные операнды,

имеют одинаковые коды операции. Чтобы различать их, используют дополнительный бит в дескрипторе сегмента кода, который устанавливается операционной системой и указывает процессору, какую команду он должен выполнить. Для обратной совместимости с программами, написанными для 8086, этот бит устанавливается в нуль, после чего все операнды по умолчанию считаются 16-битными. Если же этот бит равен единице, то используются 32-битные операнды. Более того, программист может изменить форму конкретной команды при помощи префикса: если перед кодом операции добавить префикс 0x66, то будет использоваться альтернативный размер операндов (16 битов в 32-битном режиме или 32 бита в 16-битном режиме).

6.8.6 Другие особенности x86

В процессор 80286 был добавлен механизм сегментации для разделения памяти на сегменты размером до 64 Кбайт. Когда операционная система включала сегментацию, то все адреса вычислялись относительно начала сегмента. Процессор проверял адреса и при выходе за пределы сегмента формировал сигнал ошибки, тем самым предотвращая доступ программ за пределы своего сегмента. Сегментация оказалась неудобна для программистов и не используется в современных версиях ОС Windows.

Архитектура x86 поддерживает команды, работающие с цепочками (последовательностями или строками) байтов или слов. Эти команды реализуют операции копирования, сравнения и поиска определенного значения. В современных процессорах такие команды, как правило, работают медленнее, чем последовательность простых команд, делающих то же самое, поэтому их лучше избегать.

Как мы уже упоминали ранее, префикс 0x66 используется для выбора 16-битных или 32-битных операндов. Другие префиксы используются для захвата внешней шины (это необходимо для обеспечения атомарного доступа к переменным в общей памяти в многопроцессорных системах), предсказания ветвлений или повторения команды при обработке цепочки байтов или слов.

В середине 1990 годов Intel и Hewlett-Packard совместно разработали новую 64-битную архитектуру под названием IA-64. Она была разработана с чистого листа, использовала результаты исследований в области компьютерной архитектуры, полученные за двадцать лет, прошедших с момента появления 8086, и обеспечивала 64-битное адресное пространство. Тем не менее, IA-64 до сих пор не стала успешной на рынке. Большинство компьютеров, которым необходимо большое адресное пространство, используют 64-битные расширения x86.

6.8.7 Оглядываясь назад

В этом разделе мы рассмотрели основные отличия RISC-архитектуры MIPS от CISC-архитектуры x86. Архитектура x86 позволяет писать более короткие программы, потому что ее сложные команды эквивалентны нескольким простым командам MIPS и вдобавок закодированы так, чтобы занимать минимум места в памяти. Однако архитектура x86 – это мешанина из всевозможных решений и функциональности, накопленных за годы разработки. Некоторые из них давно не несут никакой пользы, но приходится сохранять их для обратной совместимости со старыми программами. У этой архитектуры слишком мало регистров, ее команды сложно дешифровать, а набор команд трудно объяснить. Несмотря на эти недостатки, x86 остается доминирующей архитектурой для персональных компьютеров потому, что невозможно переоценить важность совместимости программного обеспечения, и потому, что огромный рынок оправдывает затраты на разработку все более быстрых x86-совместимых микропроцессоров.

6.9 РЕЗЮМЕ

Чтобы командовать компьютером, нужно разговаривать на его языке. Архитектура компьютера определяет, как именно нужно это делать. На сегодняшний день в мире широко используется большое количество разных архитектур, но если вы хорошо поймете одну из них, то изучить остальные будет довольно просто. При изучении новой архитектуры вы должны задать следующие главные вопросы:

- Какова длина слова данных?
- Какие регистры доступны?
- Как организована память?
- Какие есть инструкции?

Архитектура MIPS является 32-битной потому, что она работает с 32-битными данными (примечание переводчика: помимо 32-битного варианта архитектуры MIPS, называемого MIPS32, существует и 64-битный вариант – MIPS64. Архитектура MIPS64 позволяет выполнять программы, написанные для MIPS32 и работающие в пользовательском режиме, без изменений). В архитектуре MIPS определено 32 регистра общего назначения. В принципе, почти любой регистр можно использовать для любой цели. Тем не менее,

существуют соглашения, по которым определенные регистры зарезервированы для конкретных целей. Это сделано для того, чтобы облегчить процесс программирования и для того, чтобы функции, написанные разными программистами, могли легко между собой взаимодействовать. Например, регистр 0 (`$0`) всегда содержит константу 0, регистр `$ra` содержит адрес возврата после выполнения инструкции `jal`, а регистры `$a0`–`$a3` и `$v0`–`$v1` хранят аргументы и возвращаемое значение функции. В архитектуре MIPS память адресуется побайтово и использует 32-битные адреса. Карта памяти была описана в **разделе 6.6.1**. Каждая инструкция имеет длину 32 бита и выровнена в памяти по границе 4-байтного слова. В этой главе мы обсудили лишь наиболее часто используемые инструкции MIPS.

Важность определения компьютерной архитектуры заключается в том, что программа, написанная для выбранной архитектуры, будет работать на совершенно разных реализациях этой архитектуры. Например, программы, написанные для процессора Intel Pentium в 1993 году, будут в общем случае работать (причем работать значительно быстрее) на процессорах Intel Xeon или AMD Phenom в 2012 году.

В первой половине этой книги мы узнали о схемных и логических уровнях абстракции. В этой главе мы прыгнули на архитектурный уровень. В следующей главе мы изучим микроархитектуру – способ

организации цифровых строительных блоков, с помощью которых создаётся аппаратная реализация архитектуры процессора. Микроархитектура – это мост между электрическими схемами и программированием. По нашему мнению, изучение микроархитектуры является одним из наиболее захватывающих занятий для инженера: вы узнаете, как создать собственный микропроцессор!

УПРАЖНЕНИЯ

Упражнение 6.1 Приведите три примера из архитектуры MIPS для каждого из принципов хорошей разработки: (1) для простоты придерживайтесь единообразия, (2) типичный сценарий должен быть быстрым, (3) чем меньше, тем быстрее, (4) хорошая разработка требует хороших компромиссов. Поясните, как каждый из ваших примеров иллюстрирует соответствующий принцип.

Упражнение 6.2 Архитектура MIPS содержит набор 32-битных регистров. Можно ли создать компьютерную архитектуру без регистров? Если можно, кратко опишите такую архитектуру и её систему команд. Каковы преимущества и недостатки будут у этой архитектуры по сравнению с архитектурой MIPS?

Упражнение 6.3 Представьте себе 32-битное слово, хранящееся в адресуемой побайтово памяти и имеющее порядковый номер 42.

a) Каков байтовый адрес у слова памяти с порядковым номером 42?

b) Каковы все байтовые адреса, занимаемые этим словом памяти?

c) Предположим, что в этом слове хранится значение 0xFF223344. Изобразите графически, как это число хранится в байтах слова в случаях прямого и обратного порядка следования байтов. Ваш рисунок должен быть похож на **Рис. 6.4**. Отметьте байтовые адреса всех байтов данных.

Упражнение 6.4 Повторите **упражнение 6.3** для 32-битного слова, имеющего порядковый номер 15 в памяти.

Упражнение 6.5 Объясните, как использовать следующую программу, чтобы определить, является ли порядок следования байтов прямым или обратным:

```
li   $t0, 0xABCD9876
sw   $t0, 100($0)
lb   $s5, 101($0)
```

Упражнение 6.6 Используя кодировку ASCII, запишите следующие строки в виде последовательностей шестнадцатеричных значений символов этих строк *(прим. переводчика: вам, вероятно, потребуется транслитерировать ваше имя, используя латинский алфавит):*

a) SOS

b) Cool!

c) (ваше собственное имя)

Упражнение 6.7 Повторите **упражнение 6.6** для следующих строк:

a) howdy

b) ions

c) To the rescue!

Упражнение 6.8 Покажите, как строки из **упражнения 6.6** хранятся в адресуемой побайтово памяти, начиная с адреса 0x1000100C: (a) на машине с прямым порядком следования байтов и (b) на машине с обратным порядком следования байтов. Ваш рисунок должен быть похож на **Рис. 6.4**. Отметьте байтовые адреса всех байтов данных в обоих случаях.

Упражнение 6.9 Повторите **упражнение 6.8** для строк из **упражнения 6.7**.

Упражнение 6.10 Преобразуйте следующий код из языка ассемблера MIPS в машинный язык. Запишите инструкции в шестнадцатеричном формате.

```
add  $t0, $s0, $s1
lw   $t0, 0x20($t7)
addi $s0, $0, -10
```

Упражнение 6.11 Повторите **упражнение 6.10** для следующего кода:

```
addi $s0, $0, 73
sw   $t1, -7($t2)
sub  $t1, $s7, $s2
```

Упражнение 6.12 Рассмотрим инструкции типа *I*.

a) Какие инструкции из **упражнения 6.10** являются инструкциями типа *I*?

b) Для каждой инструкции типа *I* из **упражнения 6.10** примените расширение знака к непосредственному 16-битному операнду так, чтобы получилось 32-битное число.

Упражнение 6.13 Повторите **упражнение 6.12** для инструкций из **упражнения 6.11**.

Упражнение 6.14 Преобразуйте следующую программу из машинного языка в программу на языке ассемблера MIPS. Цифрами слева показаны адреса инструкций в памяти. Цифры справа – это инструкции по соответствующим адресам. Объясните, что делает эта программа, предполагая, что перед началом её выполнения `$a0` содержит некое положительное число `n`, а после завершения программы в `$v0` получается некоторый результат. Также напишите эту программу на языке высокого уровня (например, C).

```
0x00400000 0x20080000
```

```
0x00400004 0x20090001
0x00400008 0x0089502A
0x0040000C 0x15400003
0x00400010 0x01094020
0x00400014 0x21290002
0x00400018 0x08100002
0x0040001C 0x01001020
0x00400020 0x03E00008
```

Упражнение 6.15 Повторите **упражнение 6.14** для приведенного ниже машинного кода. Используйте значения `$a0` и `$a1` как входные данные. В начале программы `$a0` содержит некоторое 32-битное число, а `$a1` – адрес некоторого массива из 32 символов (типа `char`).

```
0x00400000 0x2008001F
0x00400004 0x01044806
0x00400008 0x31290001
0x0040000C 0x0009482A
0x00400010 0xA0A90000
0x00400014 0x20A50001
0x00400018 0x2108FFFF
0x0040001C 0x0501FFF9
0x00400020 0x03E00008
```

Упражнение 6.16 В архитектуре MIPS имеется инструкция `nor`, но отсутствует её вариант с непосредственным операндом `nori`. Тем не менее, команда `nori` может быть реализована существующими инструкциями. Напишите на языке ассемблера код со следующей функциональностью: `$t0` = `$t1` NOR 0xF234. Используйте наименьшее возможное число инструкций.

Упражнение 6.17 Реализуйте следующие фрагменты кода высокого уровня на языке ассемблера MIPS, используя инструкцию `slt`. Значения целочисленных переменных `g` и `h` хранятся в регистрах `$s0` и `$s1` соответственно.

```
(a)   if (g > h)
          g = g + h;
       else
          g = g - h;

(b)   if (g >= h)
          g = g + 1;
      else
          h = h - 1;

(c)   if (g <= h)
          g = 0;
      else
          h = 0;
```

Упражнение 6.18 Напишите функцию на языке высокого уровня (например, C), имеющую следующий прототип: `int find42(int array[], int size)`. Здесь `array` задаёт базовый адрес некоторого массива целых чисел, а `size` содержит число элементов в этом массиве. Функция должна возвращать порядковый номер первого элемента массива, содержащего значение 42. Если в массиве нет числа 42, то функция должна вернуть –1.

Упражнение 6.19 Функция `strcpy` копирует строку символов, расположенную в памяти по адресу `src`, в новое место с адресом `dst`.

> Эта простая функция копирования строки имеет один весьма серьезный недостаток: она не может узнать, зарезервировано ли достаточно места в памяти по адресу `dst`, чтобы скопировать туда исходную строку. Если компьютерный взломщик может заставить программу выполнить функцию `strcpy` с чрезмерно длинной строкой, находящейся по адресу `src`, то `strcpy` может изменить важные данные и даже инструкции в памяти программы, располагающиеся за зарезервированным участком памяти. Ловко модифицированный код может «захватить» компьютер и подчинить его действия взломщику. Это так называемая *атака переполнения буфера*. Она используется вредоносными программами, в частности печально известным «червём» Blaster, который причинил ущерба приблизительно на 525 млн. долларов в 2003 году.

```
// C code
 void strcpy(char dst[], char src[]) {
   int i = 0;
   do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(Прим. переводчика: здесь приведена некорректная реализация функции `strcpy`*, необоснованно ограничивающая длину строки диапазоном неотрицательных чисел типа* `int`*.)*

a) Реализуйте приведенную выше функцию `strcpy` на языке ассемблера MIPS. Используйте `$s0` для `i`.

b) Изобразите стек до вызова, во время и после вызова функции `strcpy`. Считайте, что перед вызовом `strcpy` `$sp` = 0x7FFFFF00.

Упражнение 6.20 Реализуйте функцию из **упражнения 6.18** на языке ассемблера MIPS.

Упражнение 6.21 Рассмотрим приведенный ниже код на языке ассемблера MIPS. Функции `func1`, `func2` и `func3` – нелистовые (нетерминальные) функции, а `func4` – листовая (терминальная). Полный код функций не показан, но в комментариях указаны регистры, используемые каждой из них.

```
0x00401000   func1:...              # func1 uses $s0-$s1
0x00401020          jal func2
 …
0x00401100   func2:...              # func2 uses $s2-$s7
0x0040117C          jal func3
 …
0x00401400   func3:...              # func3 uses $s1-$s3
0x00401704          jal func4
 …
0x00403008   func4:...              # func4 uses no preserved
0x00403118          jr $ra          # registers
```

a) Сколько слов занимает кадр стека у каждой из этих функций?

b) Изобразите стек после вызова `func4`. Укажите, какие регистры хранятся в стеке и где именно. Отметьте каждый из кадров стека. Там, где это возможно, подпишите значения, сохранённые в стеке.

Упражнение 6.22 Каждое число в последовательности Фибоначчи является суммой двух предыдущих чисел. В **Табл. 6.16** перечислены первые числа последовательности *fib*(*n*).

Табл. 6.16 Последовательность Фибоначчи

n	1	2	3	4	5	6	7	8	9	10	11	…
fib(*n*)	1	1	2	3	5	8	13	21	34	55	89	…

a) Чему равны значения *fib*(*n*) для *n* = 0 и *n* = –1?

b) Напишите функцию с именем `fib` на языке высокого уровня (например, C). Функция должна возвращать число Фибоначчи для любого неотрицательного значения *n*. Подсказка: используйте цикл. Прокомментируйте ваш код.

c) Преобразуйте функцию, написанную в части (b), в код на ассемблере MIPS. После каждой строки кода добавьте строку комментария, поясняющего, что она делает. Протестируйте код для случая *fib*(9) в симуляторе SPIM (чтобы узнать, как установить симулятор SPIM, см. Предисловие).

Упражнение 6.23 Обратимся к **примеру кода 6.27**. В этом упражнении предположим, что функция `factorial` вызывается с аргументом `n` = 5.

a) Чему будет равен регистр `$v0`, когда функция `factorial` завершится и управление будет возвращено вызвавшей ее функции?

b) Предположим, что инструкции, сохраняющие и восстанавливающие `$ra`, расположенные по адресам 0x98 и 0xBC, были убраны (например, заменены на `nop`). В этом случае программа (1) войдет в бесконечный цикл, но не завершится аварийно; (2) завершится аварийно (произойдет переполнение стека или счетчик команд выйдет за пределы программы); (3) вернет неправильное значение в `$v0` (если да, то какое?); (4) продолжит работать правильно, несмотря на изменения?

c) Повторите часть (b), когда будут удалены инструкции по следующим адресам:

   (i) 0x94 и 0xC0 (инструкции, которые сохраняют и восстанавливают `$a0`)

   (ii) 0x90 и 0xC4 (инструкции, которые сохраняют и восстанавливают `$sp`). Примечание: метка `factorial` не удаляется.

   (iii) 0xAC (инструкция, которая восстанавливает `$sp`)

Упражнение 6.24 Бен Битдидл попытался вычислить функцию $f(a,b) = 2a + 3b$ для положительного значения b, но переусердствовал с вызовами функций и рекурсией и написал вот такой код:

```
// high-level code for functions f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
```

```
}
int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x -1);
}
```

После этого Бен транслировал эти две функции на язык ассемблера. Он также написал функцию `test`, которая вызывает функцию *f*(5,3).

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j; f2: $a0 = x, $s0 = k
0x00400000   test: addi $a0, $0, 5    # $a0 = 5 (a = 5)
0x00400004         addi $a1, $0, 3    # $a1 = 3 (b = 3)
0x00400008         jal  f             # call f(5, 3)
0x0040000C   loop: j    loop          # and loop forever
0x00400010   f:    addi $sp, $sp, -16 # make room on the stack
                                      # for $s0, $a0, $a1, and $ra
0x00400014         sw   $a1, 12($sp)  # save $a1 (b)
0x00400018         sw   $a0, 8($sp)   # save $a0 (a)
0x0040001C         sw   $ra, 4($sp)   # save $ra
0x00400020         sw   $s0, 0($sp)   # save $s0
0x00400024         add  $s0, $a0, $0  # $s0 = $a0 (j = a)
0x00400028         add  $a0, $a1, $0  # place b as argument for f2
0x0040002C         jal  f2            # call f2(b)
0x00400030         lw   $a0, 8($sp)   # restore $a0 (a) after call
0x00400034         lw   $a1, 12($sp)  # restore $a1 (b) after call
0x00400038         add  $v0, $v0, $s0 # $v0 = f2(b) + j
```

```
0x0040003C          add  $v0, $v0, $a0 # $v0 = (f2(b) + j) + a
0x00400040          lw   $s0, 0($sp)   # restore $s0
0x00400044          lw   $ra, 4($sp)   # restore $ra
0x00400048          addi $sp, $sp, 16  # restore $sp (stack pointer)
0x0040004C          jr   $ra           # return to point of call
0x00400050    f2:   addi $sp, $sp, -12 # make room on the stack for
                                       # $s0, $a0, and $ra
0x00400054          sw   $a0, 8($sp)   # save $a0 (x)
0x00400058          sw   $ra, 4($sp)   # save return address
0x0040005C          sw   $s0, 0($sp)   # save $s0
0x00400060          addi $s0, $0, 3    # k = 3
0x00400064          bne  $a0, $0, else # x = 0?
0x00400068          addi $v0, $0, 0    # yes: return value should be 0
0x0040006C          j    done          # and clean up
0x00400070    else: addi $a0, $a0, -1  # no: $a0 = $a0 - 1 (x = x - 1)
0x00400074          jal  f2            # call f2(x -1)
0x00400078          lw   $a0, 8($sp)   # restore $a0 (x)
0x0040007C          add  $v0, $v0, $s0 # $v0 = f2(x - 1) + k
0x00400080    done: lw   $s0, 0($sp)   # restore $s0
0x00400084          lw   $ra, 4($sp)   # restore $ra
0x00400088          addi $sp, $sp, 12  # restore $sp
0x0040008C          jr   $ra           # return to point of call
```

Вам может быть полезно изобразить стек по примеру **Рис. 6.26**. Это поможет вам ответить на следующие вопросы:

a) Если код выполнится, начиная с метки `test`, то какое значение окажется в регистре `$v0`, когда программа дойдет до метки `loop`? Правильно ли программа вычислит 2*a* + 3*b*?

b) Предположим, что Бен удалил инструкции по адресам 0x0040001C и 0x00400044, которые сохраняют и восстанавливают значение регистра `$ra`. В этом случае программа (1) войдет в бесконечный цикл, но не остановится; (2) завершится аварийно (произойдет переполнение стека или счетчик команд выйдет за пределы программы); (3) вернет неправильное значение в `$v0` (если да, то какое?); (4) будет работать правильно, несмотря на изменения?

c) Повторите часть (b), когда будут удалены инструкции по следующим адресам. Обратите внимание, что удаляются только инструкции, но не метки:

(i) 0x00400018 и 0x00400030 (инструкции, сохраняющие и восстанавливающие `$a0`)

(ii) 0x00400014 и 0x00400034 (инструкции, сохраняющие и восстанавливающие `$a1`)

(iii) 0x00400020 и 0x00400040 (инструкции, сохраняющие и восстанавливающие `$s0`)

(iv) 0x00400050 и 0x00400088 (инструкции, сохраняющие и восстанавливающие `$sp`)

(v) 0x0040005C и 0x00400080 (инструкции, сохраняющие и восстанавливающие `$s0`)

(vi) 0x00400058 и 0x00400084 (инструкции, сохраняющие и восстанавливающие `$ra`)

(vii) 0x00400054 и 0x00400078 (инструкции, сохраняющие и восстанавливающие `$a0`)

Упражнение 6.25 Переведите приведенные ниже инструкции `beq`, `j` и `jal` в машинный код. Адреса инструкций указаны слева от каждой из них:

```
(a)
0x00401000        beq $t0, $s1, Loop
0x00401004        ...
0x00401008        ...
0x0040100C  Loop: ...

(b)
0x00401000        beq $t7, $s4, done
...               ...
0x00402040  done: ...

(c)
0x0040310C  back: ...
...               ...
0x00405000         beq $t9, $s7, back

(d)
0x00403000         jal func
...                ...
```

```
0x0041147C  func:  ...

(e)
0x00403004  back:  ...
...                ...
0x0040400C  j      back
```

Упражнение 6.26 Рассмотрим следующий фрагмент кода на языке ассемблера MIPS. Числа слева от каждой инструкции указывают ее адрес:

```
0x00400028       add  $a0, $a1, $0
0x0040002C       jal  f2
0x00400030 f1:   jr   $ra
0x00400034 f2:   sw   $s0,0($s2)
0x00400038       bne  $a0,$0, else
0x0040003C       j    f1
0x00400040 else: addi $a0, $a0, -1
0x00400044       j    f2
```

a) Транслируйте последовательность инструкций в машинный код в шестнадцатеричном формате.

b) Сделайте список режимов адресации, которые были использованы для каждой строки кода

Упражнение 6.27 Рассмотрим следующий фрагмент кода на C:

```
// C code
void setArray(int num) {
  int i;
  int array[10];
  for (i = 0; i < 10; i = i + 1) {
    array[i] = compare(num, i);
  }
}
int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
  else
    return 0;
}
int sub(int a, int b) {
  return a - b;
}
```

a) Перепишите этот фрагмент кода на языке ассемблера MIPS. Используйте регистр `$s0` для хранения переменной `i`. Следите за тем, чтобы правильно работать с указателем стека. Массив хранится в стеке функции `setArray` (см. **раздел 6.4.6**).

b) Предположим, что первой вызванной функцией будет `setArray`. Нарисуйте состояние стека перед вызовом `setArray` и во время каждого

последующего вызова. Укажите имена регистров и переменных, хранящихся в стеке. Отметьте расположение `$sp` и каждого кадра стека.

c) Как бы работал ваш код, если бы вы забыли сохранить в стеке регистр `$ra`?

Упражнение 6.28 Рассмотрим следующий фрагмент кода на C:

```
// C code
int f(int n, int k) {
  int b;
  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n - 1, k + 1);
   return b * k;
}
```

a) Транслируйте функцию `f` на язык ассемблера MIPS. Обратите особое внимание на правильность сохранения и восстановления регистров между вызовами функций, а также на использование конвенций MIPS по сохранению регистров. Тщательно комментируйте ваш код. Вы можете использовать инструкцию `mul`. Функция начинается с адреса 0x00400100. Храните локальную переменную `b` в регистре `$s0`.

b) Пошагово выполните функцию из пункта (a) для случая `f(2, 4)`. Изобразите стек, как на **Рис. 6.26 (c)**. Подпишите имена и значения регистров, хранящихся в каждом слове стека. Проследите за значением указателя стека (`$sp`). Четко обозначьте каждый кадр стека. Для вас также может быть

полезно отследить значения в `$a0`, `$a1`, `$v0` и `$s0` в процессе выполнения программы. Предположим, что при вызове `f` значение `$s0` = 0xABCD, а `$ra` = 0x400004. Каким будет конечный результат в регистре `$v0`?

Упражнение 6.29 Каков диапазон адресов, по которым инструкции ветвления, такие как `beq` и `bne`, могут выполнять переходы в MIPS? Дайте ответ в виде количества инструкций относительно адреса инструкции ветвления.

Упражнение 6.30 Ответы на приведенные ниже вопросы позволят вам лучше понять работу инструкции безусловного перехода, `j`. Дайте ответы в виде количества инструкций относительно адреса инструкции безусловного перехода.

a) Как далеко вперед (то есть по направлению к большим адресам) может перейти команда безусловного перехода (`j`) в наихудшем случае? Наихудший случай – это когда переход не может быть осуществлен далеко. Объясните словами, используя по необходимости примеры.

b) Как далеко вперед может перейти команда безусловного перехода (`j`) в наилучшем случае? Наилучший случай – это когда переход может быть осуществлен дальше всего. Поясните ответ.

c) Как далеко назад (то есть по направлению к меньшим адресам) может перейти команда безусловного перехода (`j`) в наихудшем случае? Поясните ответ.

d) Как далеко назад может перейти команда безусловного перехода (`j`) в наилучшем случае? Поясните ответ.

Упражнение 6.31 Объясните, почему выгодно иметь большое поле адреса (`addr`) в машинном формате команд безусловного перехода `j` и `jal`

Упражнение 6.32 Напишите код на языке ассемблера, который переходит к инструкции, отстоящей на 64 мегаинструкции от начала этого кода. Напомним, что 1 мегаинструкция = 2^{20} инструкций = 1,048,576 инструкций. Предположим, что ваш код начинается с адреса 0x00400000. Используйте минимальное количество инструкций.

Упражнение 6.33 Напишите функцию на языке высокого уровня, которая берет массив 32- разрядных целых чисел из 10 элементов, использующих прямой порядок следования байтов (от младшего к старшему, little-endian) и преобразует его в формат с обратным порядком (от старшего к младшего, big-endian). Перепишите код на языке ассемблера MIPS. Прокомментируйте весь ваш код и используйте минимальное количество инструкций.

Упражнение 6.34 Рассмотрим две строки: `string1` и `string2`.

a) Напишите код на языке высокого уровня для функции под названием concat, которая соединяет их (склеивает их вместе): `void concat(char string1[], char string2[], char stringconcat[])`. Заметьте, что эта

функция не возвращает значения (т.е. тип возвращаемого значения равен `void`). Результат объединения `string1` и `string2` помещается в строку в `stringconcat`. Предполагается, что массив символов `stringconcat` является достаточно большим, чтобы вместить результат.

b) Перепишите код из части (a) на языке ассемблера MIPS.

Упражнение 6.35 Напишите программу на ассемблере MIPS, которая складывает два положительных числа с плавающей точкой одинарной точности, которые хранятся в регистрах `$s0` и `$s1`. Не используйте специальные инструкции для работы с плавающей точкой. В этом упражнении вам не нужно беспокоиться о кодах значений, зарезервированных для специальных целей (например, 0, NaN и т.д.), а также о возможных переполнениях или потере точности. Используйте симулятор SPIM для тестирования кода. Вам нужно будет вручную установить значения `$s0` и `$s1`, чтобы протестировать код. Продемонстрируйте, что ваш код работает надежно.

Упражнение 6.36 Покажите, как приведенная ниже программа MIPS загружается и выполняется в памяти.

```
#  MIPS assembly code
main:
   addi  $sp, $sp, -4
   sw    $ra, 0($sp)
   lw    $a0, x
   lw    $a1, y
```

```
   jal   diff
   lw    $ra, 0($sp)
   addi  $sp, $sp, 4
   jr    $ra
diff:
   sub   $v0, $a0, $a1
   jr    $ra
```

a) Сначала отметьте рядом с каждой инструкцией ее адрес.

b) Нарисуйте таблицы символов для меток и их адресов.

c) Сконвертируйте все инструкции в машинный код.

d) Укажите размер сегментов данных (data) и кода (text) в байтах.

e) Нарисуйте карту памяти и укажите, где хранятся данные и команды.

Упражнение 6.37 Повторите **упражнение 6.36** для следующего кода

```
# MIPS assembly code
main:
   addi  $sp, $sp, -4
   sw    $ra, 0($sp)
   addi  $t0, $0, 15
   sw    $t0, a
   addi  $a1, $0, 27
   sw    $a1, b
   lw    $a0, a
```

```
   jal   greater
   lw    $ra, 0($sp)
   addi  $sp, $sp, 4
   jr    $ra
greater:
   slt   $v0, $a1, $a0
   jr    $ra
```

Упражнение 6.38 Покажите инструкции MIPS, которые реализуют следующие псевдокоманды. Вы можете использовать регистр `$at` для хранения временных данных, но вам запрещается портить содержимое (затирать) другие регистры.

a) `addi $t0, $s2,` $\texttt{imm}_{31:0}$ (прим. переводчика: под $\texttt{imm}_{31:0}$ подразумевается 32-битный непосредственный операнд)

b) `lw $t5,` $\texttt{imm}_{31:0}$`($s0)`

c) `rol $t0, $t1, 5` (циклически сдвинуть `$t1` влево на 5 разрядов и поместить результат в `$t0`)

d) `ror $s4, $t6, 31` (циклически сдвинуть `$t6` вправо на 31 разряд и поместить результат в `$s4`)

Упражнение 6.39 Повторите **упражнение 6.38** для следующих псевдокоманд:

a) `beq  $t1,` $\texttt{imm}_{31:0}$`, L`

b) `ble  $t3, $t5, L`

c) `bgt  $t3, $t5, L`

d) `bge  $t3, $t5, L`

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

Приведенные ниже вопросы обычно задают на собеседованиях на вакансии разработчиков цифровой аппаратуры (но эти вопросы относятся и к любым языкам ассемблера).

Вопрос 6.1 Напишите программу на ассемблере MIPS, которая меняет местами содержимое двух регистров (`$t0` и `$t1`). Программа не должна использовать другие регистры.

Вопрос 6.2 Предположим, что у вас есть массив из положительных и отрицательных целых чисел. Напишите программу на ассемблере MIPS, которая находит подмножество массива с максимальной суммой. Адрес массива и количество элементов хранятся в регистрах `$a0` и `$a1` соответственно. Программа должна поместить найденное подмножество массива, начиная с адреса, находящегося в регистре `$a2`. Код должен работать максимально быстро.

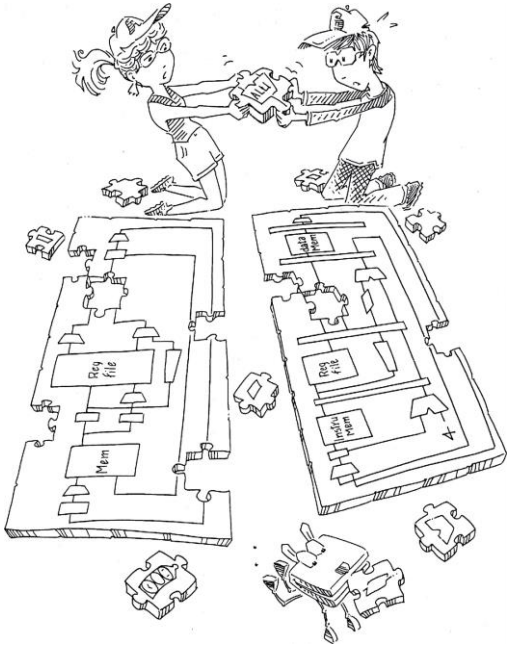
Вопрос 6.3 Дан массив, хранящий строку C. Строка содержит предложение. Придумайте алгоритм, который запишет слова в предложении в обратном порядке и сохранит результат обратно в этот массив. Реализуйте ваш алгоритм на языке ассемблера MIPS.

Вопрос 6.4 Придумайте алгоритм подсчета количества единиц в 32-битном числе. Реализуйте ваш алгоритм на языке ассемблера MIPS.

Вопрос 6.5 Напишите программу на языке ассемблера MIPS, меняющую порядок битов в регистре на обратный. Используйте как можно меньше инструкций. Исходное значение хранится в регистре `$t3`.

Вопрос 6.6 Напишите программу на языке ассемблера MIPS, проверяющую, произошло ли переполнение при сложении регистров `$t2` и `$t3`. Используйте как можно меньше инструкций.

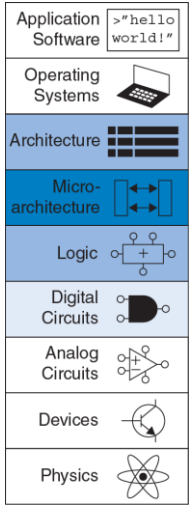
Вопрос 6.7 Придумайте алгоритм, который проверяет, является ли заданная строка палиндромом (палиндром – это слово, которое читается в обоих направлениях одинаково, например, “wow” или “racecar”). Напишите программу на языке ассемблера MIPS, реализующую этот алгоритм.


Reg File
Mem
Reg File
Mem

Микроархитектура

Глава написана с помощью Мэтью Уоткинса (Matthew Watkins)

7



7.1 ВВЕДЕНИЕ

Из этой главы вы узнаете, как собрать собственную версию, а точнее – три версии процессора MIPS, отличающиеся между собой разным соотношением производительности, цены и сложности.

Для непосвященных создание микропроцессора выглядит как черная магия. На самом деле это относительно просто, и к этому моменту вы уже знаете все, что нужно.

В частности, вы изучили разработку комбинационных и последовательностных схем по заданным функциональным и временным ограничениям. Вы познакомились с построением арифметических схем и блоков памяти. Также вы изучили архитектуру MIPS, описывающую регистры, команды и память так, как видит их программист.

Эта глава посвящена микроархитектуре, которая является связующим звеном между логическими схемами и архитектурой. Микроархитектура описывает, как именно в процессоре расположены и соединены друг с другом регистры, АЛУ, конечные автоматы, блоки памяти и другие блоки, необходимые для реализации архитектуры. У каждой архитектуры, включая MIPS, может быть много различных микроархитектур, обеспечивающих разное соотношение производительности, цены и сложности. Все они смогут выполнять одни

и те же программы, но их внутреннее устройство может очень сильно отличаться. В этой главе мы разработаем три различные микроархитектуры, чтобы проиллюстрировать компромиссы, на которые приходится идти разработчику.

Эта глава в значительной степени опирается на классические разработки MIPS-процессоров, описанные Дэвидом Паттерсоном и Джоном Хеннесси в их учебнике *Архитектура компьютера и проектирование компьютерных систем* (*Computer Organization and Design*). Дэвид и Джон великодушно поделились этими элегантными разработками, которые являются отличным примером реальной коммерческой архитектуры и в то же время относительно просты и легки для понимания.

7.1.1 Архитектурное состояние и система команд

Напомним, что компьютерная архитектура определяется набором команд и *архитектурным состоянием.* Архитектурное состояние процессора MIPS определяется содержимым счетчика команд (program counter, PC) и 32 видимых программисту регистров, поэтому любой процессор, реализующий архитектуру MIPS, вне зависимости от его микроархитектуры обязан иметь счетчик команд и ровно 32 регистра. Зная текущее архитектурное состояние, процессор точно знает, какую операцию над какими данными надо выполнить для получения нового

архитектурного состояния. У некоторых микроархитектур есть также и неархитектурное (т.е. невидимое программисту) состояние, которое используется или для упрощения логики, или для улучшения производительности. Когда мы столкнемся с необходимостью добавить неархитектурное состояние, мы обратим на это ваше внимание.

Дэвид Паттерсон был первым в семье, кто окончил университет (Калифорнийский Университет в Лос-Анджелесе в 1969 году). С 1977 года он является профессором информатики в Калифорнийском Университете в Беркли. Он является одним из изобретателей архитектуры RISC. В 1984 он разработал архитектуру SPARC, которая использовалась компанией Sun Microsystems (прим. переводчика: Sun Microsystems была куплена корпорацией Oracle в 2010 году). Также он отец технологий RAID (избыточный массив независимых дисков) и NOW (сеть рабочих станций).

Джон Хеннесси – президент Стэнфордского Университета, где с 1977 года является профессором информатики и вычислительной техники. Он так же является одним из изобретателей RISC. В 1984 году в Стэнфорде он разработал архитектуру MIPS и основал MIPS Computer Systems. По состоянию на 2004 год было продано более 300 миллионов процессоров MIPS.

В свободное время эти двое достойных подражания джентльменов расслабляются, сочиняя учебники.

Чтобы микроархитектура оставалась простой для понимания, мы рассмотрим только небольшое подмножество набора команд MIPS, а именно:

- Арифметические и логические команды типа R: `add`, `sub`, `and`, `or`, `slt`
- Команды доступа в память: `lw`, `sw`
- Команды условного перехода: `beq`

После того, как мы разработаем микроархитектуру, поддерживающую эти команды, мы расширим ее, добавив команды `addi` и `j`. Это подмножество команд было выбрано потому, что его достаточно для написания множества интересных программ. Как только вы поймете, как реализовать эти команды в аппаратуре, вы сможете добавить и другие.

7.1.2 Процесс разработки

Мы разделим нашу микроархитектуру на две взаимодействующих части: тракт данных и устройство управления. Тракт данных работает со словами данных. Он содержит такие блоки, как память, регистры, АЛУ и мультиплексоры. MIPS – это 32-битная архитектура, так что мы будем использовать 32-битный тракт данных. Устройство управления

получает текущую команду из тракта данных и в ответ говорит ему, как именно выполнять эту команду. В частности, устройство управления генерирует адресные сигналы для мультиплексоров, сигналы разрешения работы для регистров и сигналы разрешения записи в память.

Хороший способ разработки сложной системы – начать с элементов, которые хранят ее состояние. Эти элементы включают память для хранения команд и данных и блоки для хранения архитектурного состояния, т.е. счетчик команд и видимые программисту регистры (прим. переводчика: в зависимости от контекста под состоянием процессора может пониматься как его чисто архитектурное состояние, так и архитектурное состояние плюс содержимое памяти). Затем между этими элементами нужно расположить комбинационные схемы, вычисляющие новое состояние на основе текущего состояния. Команда читается из той части памяти, где находится программа; команды загрузки и сохранения затем читают или пишут данные в другую часть памяти. Поэтому зачастую бывает удобно разделить память на две меньшие по размеру части, чтобы одна содержала команды, а другая – данные. На **Рис. 7.1** показаны четыре вышеупомянутых элемента: счётчик команд, регистровый файл, память команд и память данных.

На **Рис. 7.1** самые толстые линии используются для 32-битных шин данных. Линии потоньше используются для шин меньшей разрядности,

таких как пятибитная шина адреса регистрового файла. Самые тонкие синие линии используются для управляющих сигналов, таких как сигнал разрешения записи в регистровый файл. Мы будем использовать линии разной толщины и дальше, чтобы избежать загромождения диаграмм указанием разрядности шин. Кстати, у элементов, хранящих состояние системы, обычно есть сигнал сброса, который устанавливает их в известное состояние в момент включения. Опять же, мы не будем показывать сигналы сброса на диаграммах.

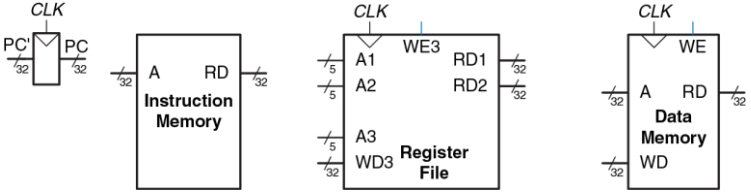


Рис. 7.1 **Элементы, хранящие состояние процессора MIPS**

Счетчик команд – это обычный 32-битный регистр. Его выход (*PC*) содержит адрес текущей команды. Его вход (*PC'*) содержит адрес следующей команды.

Память команд имеет один порт чтения[8]. На адресный вход A подается 32-битный адрес команды, после чего на выходе *RD* появляется 32-битное число (т.е. команда), прочитанное из памяти по этому адресу.

Регистровый файл, содержащий 32 элемента по 32 бита каждый, имеет два порта чтения и один порт записи данных. Порты чтения имеют пятибитные входы адреса *A1* и *A2*, каждый из которых определяет один из $2^5 = 32$ регистров в качестве источника данных для команды. Каждый из портов читает 32-битное значение из регистра и подает его на выходы *RD1* и *RD2* соответственно. Порт записи получает пятибитный адрес регистра на адресный вход *A3*, 32-битное число на вход данных *WD*, сигнал разрешения записи *WE3* и тактовый сигнал. Если сигнал разрешения записи равен единице, то регистровый файл записывает данные в указанный регистр по положительному фронту тактового сигнала.

Память данных имеет один порт чтения/записи. Если сигнал разрешения записи *WE* равен единице, то данные со входа *WD*

[8] Это упрощение сделано для того, чтобы можно было считать память команд памятью только для чтения (ROM); в большинстве реальных процессоров память команд должна быть доступна и для записи, чтобы операционная система могла загружать в нее новые программы. Многотактная микроархитектура, описанная в Разделе 7.4, более реалистична в этом плане, так как содержит общую память команд и данных, доступную как для чтения, так и для записи.

записываются в ячейку памяти с адресом *A* по положительному фронту тактового сигнала. Если же сигнал разрешения записи равен нулю, то данные из ячейки с адресом A подаются на выход *RD*.

Сброс счетчика команд

Из всех регистров процессора как минимум счетчик команд (PC) должен иметь сигнал сброса, который проинициализирует его в момент включения процессора. При получении сигнала сброса процессор MIPS инициализирует PC значением 0xBFC00000; как только сигнал сброса снят, процессор начинает выполнять код по этому адресу. Этот код загружает операционную систему (ОС). Затем ОС загружает в память по адресу 0x00400000 пользовательскую программу и передает ей управление. В этой главе для простоты мы будем считать, что счетчик команд инициализируется значением 0x00000000, поэтому будем размещать наши программы в памяти так, чтобы они начинались именно с этого адреса.

Чтение из памяти команд, регистрового файла и памяти данных происходит асинхронно, то есть независимо от тактового сигнала. Другими словами, сразу же после изменения значения на адресном входе на выходе *RD* появляются новые данные. Это происходит не мгновенно, так как существует задержка распространения сигнала, однако тактовый сигнал для чтения не требуется. Запись же производится исключительно по положительному фронту тактового

сигнала. Таким образом, состояние системы изменяется только по фронту тактового сигнала. Адрес, данные и сигнал разрешения записи должны стать корректными за некоторое время до прихода фронта тактового сигнала (время предустановки, setup) и ни в коем случае не должны изменяться до тех пор, пока не пройдет некоторое время после прихода фронта (время удержания, hold).

В связи с тем, что элементы памяти изменяют свои значения только по положительному фронту тактового сигнала, они являются синхронными последовательностными схемами. Микропроцессор строится из тактируемых элементов памяти и комбинационной логики, поэтому он тоже является синхронной последовательностной схемой. На самом деле, процессор можно рассматривать как гигантский конечный автомат или как несколько более простых и взаимодействующих между собой конечных автоматов.

7.1.3 Микроархитектуры MIPS

В этой главе мы разработаем три микроархитектуры для процессорной архитектуры MIPS: однотактную, многотактную и конвейерную. Они различаются способом соединения элементов памяти, а также наличием или отсутствием неархитектурного состояния.

Однотактная микроархитектура выполняет всю команду за один такт. Ее принцип работы легко объяснить, а устройство управления довольно простое. Из-за того, что все действия выполняются за один такт, эта микроархитектура не требует никакого неархитектурного состояния. Однако длительность такта ограничена самой медленной командой.

Многотактная микроархитектура выполняет команду за несколько более коротких тактов. Простым командам нужно меньше тактов, чем сложным. Вдобавок, многотактная микроархитектура уменьшает количество необходимой аппаратуры путем повторного использования таких «дорогих» блоков, как сумматоры и блоки памяти. Например, при выполнении команды один и тот же сумматор на разных тактах может быть использован для разных целей. Повторное использование блоков достигается путем добавления в многотактный процессор нескольких неархитектурных регистров для сохранения промежуточных результатов. Многотактный процессор выполняет только одну команду за раз, и каждая команда занимает несколько тактов.

Конвейерная микроархитектура – результат применения принципа конвейерной обработки к однотактной микроархитектуре. Вследствие этого она позволяет выполнять несколько команд одновременно, значительно улучшая пропускную способность процессора. Конвейерная микроархитектура требует дополнительной логики для

разрешения конфликтов между одновременно выполняемыми в конвейере командами. Она также требует несколько неархитектурных регистров, расположенных между стадиями конвейера. Тем не менее, эта дополнительная логика и регистры того стоят – в наши дни все коммерческие высокопроизводительные процессоры используют конвейеры. Мы изучим детали и компромиссы этих трех микроархитектур в следующих разделах. В конце главы мы упомянем дополнительные способы увеличения производительности, используемые в современных высокопроизводительных процессорах.

7.2 АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ

Как мы уже упоминали ранее, у каждой процессорной архитектуры может быть много различных микроархитектур, обеспечивающих разное соотношение цены и производительности. Цена зависит от количества логических элементов и от технологии производства микросхемы. Прогресс в КМОП-технологиях позволяет размещать все больше и больше транзисторов на кристалле за те же деньги, что активно используется для производства новых процессоров с еще большей производительностью. Точный расчет цены невозможен без детального знания конкретной технологии производства, но в целом, чем больше логических элементов и памяти, тем больше цена. В этом разделе мы познакомимся с основами анализа производительности.

Производительность компьютерной системы можно измерить множеством способов, и маркетологи стараются выбрать именно те из них, которые позволяют их компьютерам выглядеть в наилучшем свете вне зависимости от того, имеют эти измерения какое-либо отношение к реальной жизни или нет. Например, компании Intel и AMD продают процессоры с одинаковой архитектурой x86. В конце 1990-х и начале 2000-х годов в рекламе процессоров Intel Pentium III и Pentium 4 широко использовалась их тактовая частота, так как Intel могла производить процессоры с большей частотой, чем ее конкуренты. Однако AMD продавала процессоры Athlon, которые выполняли программы быстрее, чем процессоры Intel с такой же тактовой частотой. Как же быть пользователю?

Единственный по-настоящему честный способ узнать производительность компьютера – измерить время выполнения вашей программы. Чем быстрее компьютер выполнит ее, тем выше его производительность. Еще один хороший способ – измерить время выполнения не одной, а нескольких программ, похожих на те, которые вы планируете запускать; это особенно важно, если ваша программа еще не написана или измерения проводит кто-то, у кого ее нет. Такие программы называются тестовым набором (benchmark), а полученные времена обычно публикуются, чтобы было ясно, насколько быстр

компьютер. Время выполнения программы в секундах можно вычислить по **формуле (7.1)**

$$ExecutionTime = (\#\ instructions)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right) \quad (7.1)$$

Количество команд в программе зависит от архитектуры процессора. У некоторых архитектур могут быть очень сложные команды, каждая из которых выполняет множество действий, что уменьшает общее количество команд в программе. Однако такие команды зачастую медленнее выполняются логическими схемами процессора. Количество команд также сильно зависит от смекалки программиста. В этой главе мы будем подразумевать, что количество команд в программах одинаково для всех реализаций архитектуры MIPS, то есть не зависит от микроархитектуры.

Количество тактов на команду, часто называемое *CPI* (cycles per instruction) – это среднее количество тактов процессора, необходимых для выполнения команды. Это соотношение обратно пропорционально производительности, измеряемой в командах на такт (*IPC* – instructions per cycle). У разных микроархитектур разное CPI. В этой главе мы будем считать, что процессор работает с идеальной подсистемой памяти, которая никак не влияет на CPI. В **главе 8** мы рассмотрим

случаи, когда процессору иногда приходится ждать ответа из памяти, что увеличивает CPI.

Число секунд на такт – это период T_c тактового сигнала, который зависит от имеющей наибольшую задержку цепи, соединяющей логические элементы внутри процессора (critical path). У разных микроархитектур период тактового сигнала может сильно отличаться. Он зависит, в том числе, и от выбранных разработчиками способов реализации аппаратных блоков. Например, сумматор с ускоренным переносом работает быстрее, чем сумматор с последовательным переносом. До сих пор улучшение технологий производства удваивало скорость переключения транзисторов каждые четыре-шесть лет, так что процессор, произведенный сегодня, работает гораздо быстрее, чем процессор с точно такой же микроархитектурой и аппаратными блоками, но произведенный десять лет назад.

Главная задача, стоящая перед разработчиком микроархитектуры – создать такой процессор, который обеспечивал бы наименьшее возможное время выполнения программ, в то же время удовлетворяя ограничениям по цене и/или энергопотреблению. Так как решения, принятые на микроархитектурном уровне, влияют и на CPI, и на T_c, и в свою очередь зависят от выбранных аппаратных блоков и схемотехнических решений, то выбор наилучшего варианта требует очень внимательного анализа.

Существует много других факторов, которые влияют на общую производительность компьютера. Например, производительность жестких дисков, памяти, графической или сетевой подсистем может быть настолько плохой, что производительность процессора на их фоне не будет иметь абсолютно никакого значения. Даже самый быстрый в мире процессор не поможет вам загружать вебсайты побыстрее, если вы подключены к Интернету через обычную телефонную линию. Эти факторы выходят за рамки этой книги и рассматривать их мы не будем.

7.3 ОДНОТАКТНЫЙ ПРОЦЕССОР

Сначала мы разработаем микроархитектуру MIPS, которая выполняет команды за один такт. Начнем с конструирования тракта данных путем соединения показанных на **Рис. 7.1** элементов, хранящих состояние процессора, при помощи комбинационной логики, которая и будет выполнять разные команды. Управляющие сигналы нужны, чтобы указывать, как именно тракт данных должен выполнять команду, находящуюся в нем в текущий момент времени. Устройство управления содержит комбинационную логику, которая формирует необходимые управляющие сигналы в зависимости от того, какая команда выполняется в данный момент. В заключение мы оценим производительность такого процессора.

7.3.1 Однотактный тракт данных

В этом разделе мы шаг за шагом создадим однотактный тракт данных, используя элементы, показанные на **Рис. 7.1**. Новые элементы и цепи будем выделять черным (или синим, в случае управляющих сигналов), а уже рассмотренные элементы будем перекрашивать серым.

Счетчик команд (Program Counter, PC) содержит адрес команды, которую надо выполнить. Первым шагом нам надо прочитать эту команду из памяти команд. Как показано на **Рис. 7.2**, счетчик команд напрямую подключен к адресному входу памяти команд. Команда, прочитанная, или выбранная (fetched), из памяти команд – это 32-битная команда, отмеченная на рисунке как Instr.

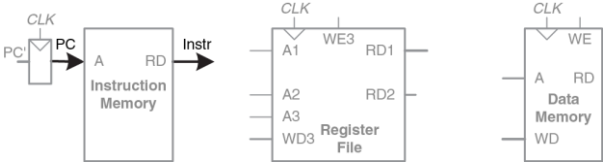


Рис. 7.2 Выборка команды из памяти

Дальнейшие действия процессора будут зависеть от того, какая именно команда была выбрана. Для начала давайте создадим тракт данных

для команды `lw`, после чего подумаем, как расширить его так, чтобы он мог выполнять и другие команды.

Для команды `lw` следующим шагом мы должны прочитать регистр операнда (source register), содержащий так называемый базовый адрес. Номер этого регистра указан в поле `rs` ($Instr_{25:21}$). Эти пять битов подключены к адресному входу первого порта (*A1*) регистрового файла, как показано на **Рис. 7.3**. Значение, прочитанное из регистрового файла, появляется на его выходе *RD1.*

Команде `lw` также требуется смещение (offset) – число, которое будет прибавлено к базовому адресу. Смещение передается как непосредственный операнд (immediate), то есть находится непосредственно в поле $Instr_{15:0}$, занимая младшие 16 бит. Так как 16-битное число может быть как положительным, так и отрицательным, то над ним должна быть выполнена операция знакового расширения до 32 бит, как показано на **Рис. 7.4**. Полученное 32-битное расширенное значение называется *SignImm*. Как вы помните из **раздела 1.4.6**, знаковое расширение заключается в том, что знаковый бит (он же старший бит) расширяемого числа просто копируется во все старшие биты расширенного числа, а именно $SignImm_{15:0} = Instr_{15:0}$ и $SignImm_{31:16} = Instr_{15}$.

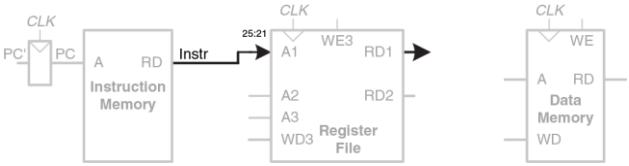


Рис. 7.3 **Чтение операнда из регистрового файла**

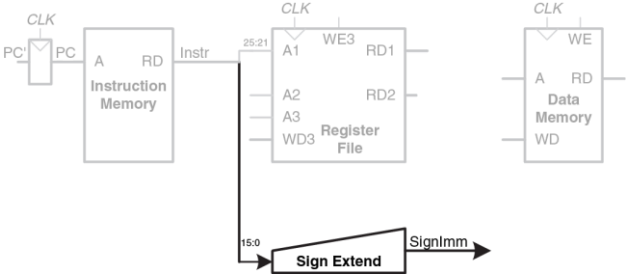


Рис. 7.4 **Знаковое расширение непосредственного операнда**

Процессор должен добавить смещение к базовому адресу, чтобы получить адрес, по которому будет произведено чтение из памяти. Для суммирования мы добавляем в тракт данных АЛУ (ALU), как показано на **Рис. 7.5**. АЛУ получает на входы два операнда, *SrcA* и *SrcB.*

SrcA подается прямо из регистрового файла, а *SrcB* – из смещения со знаковым расширением. Трехбитный управляющий сигнал *ALUControl* говорит АЛУ, какую операцию надо выполнить. На выходе из АЛУ получается 32-битный результат *ALUResult* и флаг нуля (Zero flag), который устанавливается в единицу, если *ALUResult* равен нулю. Для команды lw сигнал *ALUControl* должен быть равен 010 – в этом случае смещение будет прибавлено к базовому адресу. Далее *ALUResult* подается на адресный вход памяти данных, как показано на **Рис. 7.5**.

Значение, прочитанное из памяти данных, попадает на шину *ReadData*, после чего записывается обратно в регистровый файл в конце такта, как показано на **Рис. 7.6**. Третий порт регистрового файла – это порт записи.

Регистр результата (destination register), в который команда lw запишет прочитанное из памяти значение, определяется полем rt ($Instr_{20:16}$), которое подключено к адресному входу третьего порта (*A3*) регистрового файла. Шина *ReadData* подключена ко входу данных третьего порта (*WD3*). Управляющий сигнал *RegWrite,* в свою очередь, соединен со входом разрешения записи третьего порта (*WE3*) и активен во время выполнения команды lw, чтобы прочитанное значение было записано в регистровый файл. Сама запись происходит по положительному фронту тактового сигнала, которым заканчивается такт процессора.

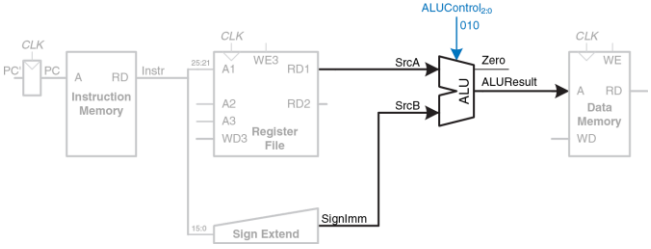


Рис. 7.5 Вычисление адреса данных в памяти

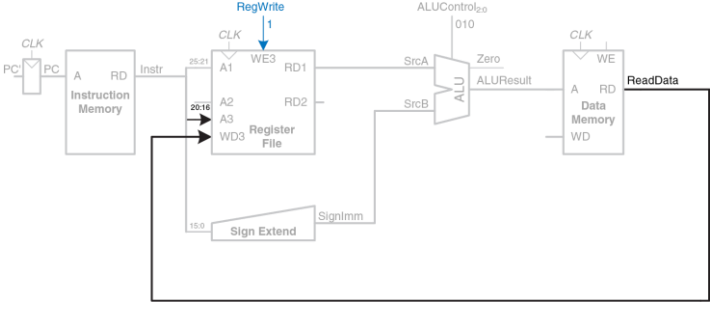


Рис. 7.6 Запись в регистровый файл

Одновременно с выполнением команды процессор должен вычислить адрес следующей команды, *PC'*. Так как команды 32-битные, то есть четырехбайтные, то адрес следующей команды равен *PC* + 4. На **Рис. 7.7** показано, что для этого нам нужен еще один сумматор. Новый адрес записывается в счетчик команд в момент прихода положительного фронта тактового сигнала. На этом создание тракта данных завершено.

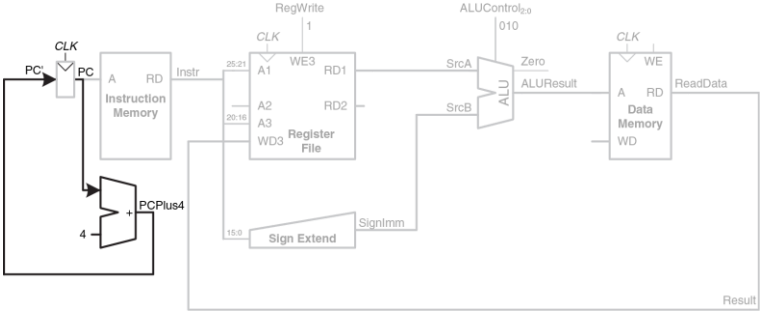


Рис. 7.7 **Определение адреса следующей команды**

Теперь давайте улучшим тракт данных, чтобы он мог выполнять еще и команду `sw`. Как и команда `lw`, `sw` читает базовый адрес из первого

порта регистрового файла и знаково расширяет смещение, передаваемое как непосредственный операнд. АЛУ складывает базовый адрес со смещением, чтобы получить адрес в памяти. Все эти функции уже реализованы в тракте данных.

Команда `sw`, в отличие от `lw`, читает из регистрового файла еще один регистр и записывает его содержимое в память данных, как показано на **Рис. 7.8**. Номер регистра указывается в поле `rt` ($Instr_{20:16}$). Эти пять бит подключены ко второму порту (*A2*) регистрового файла. Прочитанное значение появляется на выходе *RD2* и попадает на вход записи в память данных. Вход разрешения записи (*WE*) управляется сигналом *MemWrite*. Для команды sw сигнал *MemWrite* = 1, чтобы данные были записаны в память; *ALUControl* = 010, чтобы базовый адрес был просуммирован со смещением; и *RegWrite* = 0, потому что команда ничего не пишет в регистровый файл. Заметьте, что *ReadData* читается из памяти в любом случае, но прочитанное значение игнорируется, так как *RegWrite* = 0.

Теперь добавим поддержку команд типа R – add, sub, and, or, и slt. Все эти команды читают два регистра из регистрового файла, выполняют над ними некие операции в АЛУ и записывают результат обратно в третий регистр. Единственное различие этих команд – в типе операции. Таким образом, все они могут быть выполнены одной и той же

аппаратурой, используя только лишь разные значения управляющего сигнала *ALUControl*.

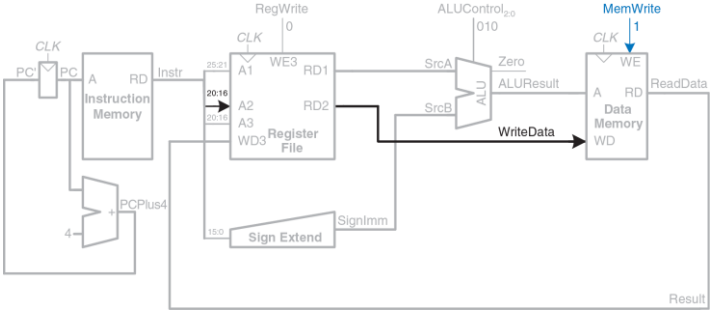


Рис. 7.8 Запись командой sw данных в память

Обновленный тракт данных с поддержкой команд типа R показан на **Рис. 7.9**. Содержимое двух регистров читается из регистрового файла и подается на входы АЛУ. На **Рис. 7.8** операнд *SrcB* всегда был равен непосредственному значению *SignImm* с расширением знака. Теперь же нам пришлось добавить мультиплексор, чтобы была возможность подать на вход АЛУ или *SignImm*, или выход *RD2* регистрового файла. Мультиплексор управляется новым сигналом *ALUSrc. ALUSrc* равен

нулю для команд типа R – в этом случае на вход АЛУ подается значение из регистрового файла. Для команд `lw` и `sw` *ALUSrc* равен единице, и на вход АЛУ подается *SignImm*. Добавление мультиплексоров в тракт данных – очень удобный способ выбрать один из нескольких источников данных и подать его на вход какого-либо блока процессора. Чтобы закончить с командами типа R, мы воспользуемся этим способом еще дважды.

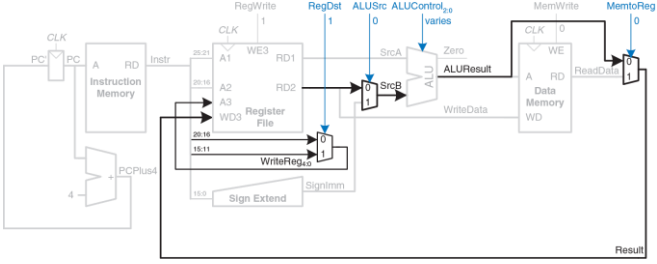


Рис. 7.9 Изменения в тракте данных для поддержки команд типа R

На **Рис. 7.8** порт записи регистрового файла был всегда подключен к памяти данных. Однако команды типа R должны записывать в регистровый файл значение *ALUResult*. Чтобы выбирать между *ReadData* и *ALUResult*, мы добавляем еще один мультиплексор, выход

которого будем называть *Result*. Этот мультиплексор управляется еще одним новым сигналом – *MemtoReg. MemtoReg* равен нулю для команд типа R – в этом случае *Result* принимает значение *ALUResult*. Для команды `lw` *MemtoReg* равен единице, а *Result* принимает значение *ReadData*. Для команды sw значение *MemtoReg* не играет никакой роли, так как sw ничего в регистровый файл не пишет.

Аналогично, на **Рис. 7.8** номер регистра в регистровом файле, куда нужно было записать данные, определялся полем `rt` ($Instr_{20:16}$). Для команд типа R, однако, номер регистра задается полем rd ($Instr_{15:11}$), поэтому мы добавляем третий мультиплексор, чтобы присваивать сигналу *WriteReg* значение из нужного поля. Этот мультиплексор управляется сигналом *RegDst. RegDst* равен единице для команд типа R – в этом случае *WriteReg* принимает значение поля rd ($Instr_{15:11}$). Для команды `lw` *RegDst* равен нулю, а *WriteReg* принимает значение поля `rt` ($Instr_{20:16}$). Для команды `sw` значение RegDst не играет никакой роли, так как sw ничего в регистровый файл не пишет.

Наконец, давайте добавим поддержку команды `beq`. Эта команда сравнивает два регистра, и, если они равны, то добавляет смещение к счетчику команд PC, выполняя, таким образом, условный переход. Напомним, что смещение – это положительное или отрицательное число, передаваемое как непосредственный операнд в поле $Instr_{15:0}$.

Смещение указывает, сколько команд нужно перепрыгнуть, прежде чем продолжить выполнение программы. Таким образом, над значением непосредственного операнда надо выполнить операцию знакового расширения, поле чего умножить его на четыре, чтобы получить новое значение счетчика команд: $PC' = PC + 4 + SignImm \times 4$

Изменения, которые нужно внести в тракт данных, показаны на **Рис. 7.10**. Новое значение счетчика команд на случай выполненного условного перехода (*PCBranch*) вычисляется путем сдвига *SignImm* влево на два разряда и последующего сложения с *PCPlus4*. Сдвиг влево на два разряда – это легкий способ умножения на четыре, так как сдвиг на константное число разрядов не требует никаких логических элементов, а требует только переподключения сигналов. Два регистра сравниваются путем вычитания одного из другого в АЛУ. Если *ALUResult* равен нулю, о чем сигнализирует флаг нуля, то регистры равны. Нужно добавить мультиплексор, чтобы выбрать, какое именно значение присвоить *PC'* – *PCPlus4* или *PCBranch. PCBranch* используется тогда, когда выполняется команда условного перехода и установлен флаг нуля. Таким образом, сигнал *Branch* равен единице для команды beq и нулю для прочих команд. Для beq сигналы *ALUControl* = 110, что означает, что АЛУ должно вычитать. *ALUSrc* = 0, чтобы операнд *SrcB* был прочитан из регистрового файла. *RegWrite* и *MemWrite* равны нулю, так как команда условного перехода ничего не

пишет ни в регистровый файл, ни в память. Значения *RegDst* и *MemtoReg* нас не интересуют, потому что в регистровый файл ничего не пишется.

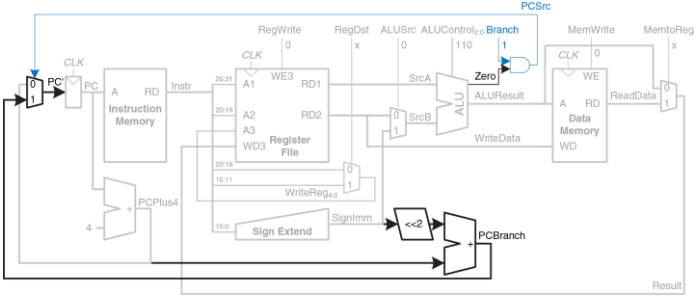


Рис. 7.10 Изменения в тракте данных для поддержки команды beq

На этом разработка тракта данных однотактного процессора завершена. Мы рассмотрели не только устройство процессора, но и сам процесс разработки, во время которого мы выбирали элементы памяти и соединяли их при помощи все усложняющейся комбинационной логики. В следующем разделе мы рассмотрим, как формировать управляющие сигналы, настраивающие тракт данных на выполнение той или иной команды.

7.3.2 Однотактное устройство управления

Устройство управления формирует управляющие сигналы на основе полей `opcode` и `funct`, присутствующих в каждой команде ($Instr_{31:26}$ и $Instr_{5:0}$ соответственно). На **Рис. 7.11** показан однотактный процессор MIPS с устройством управления, подключенным к тракту данных.

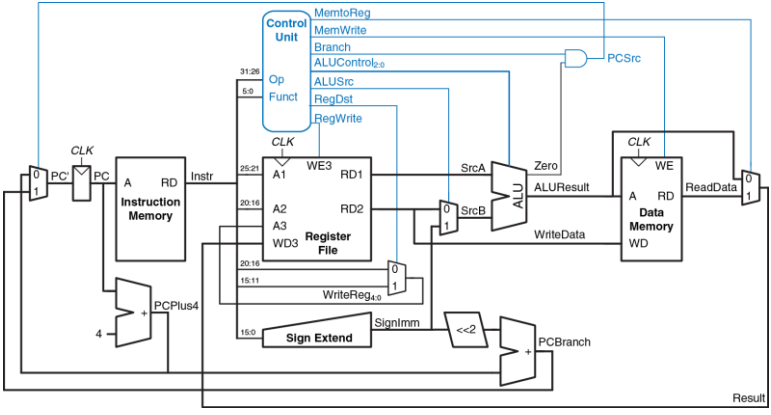


Рис. 7.11 Законченный однотактный процессор MIPS

Большая часть информации для устройства управления берется из поля `opcode`, но команды типа R также используют и поле `funct` для определения операции в АЛУ. Таким образом, для упрощения разработки мы поделим устройство управления на две части, содержащие комбинационную логику, как показано на **Рис. 7.12**.

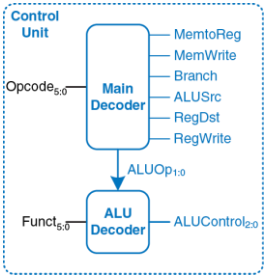


Рис. 7.12 Внутренняя структура устройства управления

Основной дешифратор вычисляет значение большинства выходов на основе поля `opcode`. Он также формирует двухбитный сигнал *ALUOp*. Дешифратор АЛУ (ALU decoder) использует *ALUOp* совместно с полем `funct` для вычисления состояния *ALUControl*. Расшифровка значений сигнала *ALUOp* приведена в **Табл. 7.1**.

Табл. 7.1 Расшифровка ALUOp

ALUOp	Функция
00	Сложение
01	Вычитание
10	Определяется полем `funct`
11	Не используется

Таблица истинности для дешифратора АЛУ приведена в **Табл. 7.2**. Значения всех трех битов сигнала *ALUControl* были приведены в **Табл. 5.1**. Так как сигнал *ALUOp* никогда не равен 11, то для упрощения логики мы можем использовать неопределенные значения X1 и 1X вместо 01 и 10.

Когда сигнал *ALUOp* равен 00 или 01, АЛУ должно складывать или вычитать соответственно. Когда он равен 10, то значение *ALUControl* должно определяться полем `funct`. Заметьте, что для команд типа R, которые мы добавили, первые два бита поля `funct` всегда равны 10, так что мы можем их проигнорировать для упрощения дешифратора.

Управляющие сигналы для всех команд уже были описаны, когда мы создавали тракт данных. Таблица истинности для основного

дешифратора, показывающая зависимость управляющих сигналов от значения `opcode`, приведена в **Табл. 7.3**.

Табл. 7.2 Таблица истинности дешифратора АЛУ

ALUOp	funct	ALUControl
00	X	010 (сложение)
X1	X	110 (вычитание)
1X	100000 (`add`)	010 (сложение)
1X	100010 (`sub`)	110 (вычитание)
1X	100100 (`and`)	000 (логическое «И»)
1X	100101 (`or`)	001 (логическое «ИЛИ»)
1X	101010 (`slt`)	111 (установить, если меньше)

Для всех команд типа R основной дешифратор формирует одинаковые сигналы; эти команды отличаются только сигналами, сформированными дешифратором АЛУ. Для команд, которые не пишут в регистровый файл (например, `sw` или `beq`), управляющие сигналы *RegDst* и *MemtoReg* могут принимать любое состояние, то есть являются неопределенными (X); адрес и данные, приходящие на порт записи регистрового файла, не имеют никакого значения, так как *RegWrite* равен нулю.

Для создания дешифратора вы можете использовать любой известный вам метод синтеза комбинационных схем.

Табл. 7.3 Таблица истинности основного дешифратора

Команда	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
Команды типа R	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Пример 7.1 ФУНКЦИОНИРОВАНИЕ ОДНОТАКТНОГО ПРОЦЕССОРА

Определите значения управляющих сигналов, а также части тракта данных, которые задействованы при выполнении команды `or`.

Решение: на **Рис. 7.13** показаны управляющие сигналы и пути движения данных во время выполнения команды `or`. Счетчик команд указывает на ячейку памяти, из которой выбирается команда.

Прохождение данных через регистровый файл и АЛУ показано синей пунктирной линией. Из регистрового файла читаются два операнда; их номера регистров задаются полями $Instr_{25:21}$ и $Instr_{20:16}$. На вход *SrcB* нужно подать значение, прочитанное из второго порта регистрового файла, а не *SignImm*, так что *ALUSrc*

должен быть равен нулю. Команда or – это команда типа R, то есть *ALUOp* равен 10, поэтому значение сигнала *ALUControl*, которое для команды or должно быть равно 001, будет вычислено на основе поля `funct`. Сигнал Result формируется в АЛУ, поэтому *MemtoReg* должен быть равен нулю. Результат записывается в регистровый файл, поэтому RegWrite будет равен единице. Команда ничего не пишет в память, так что *MemWrite* равен нулю.

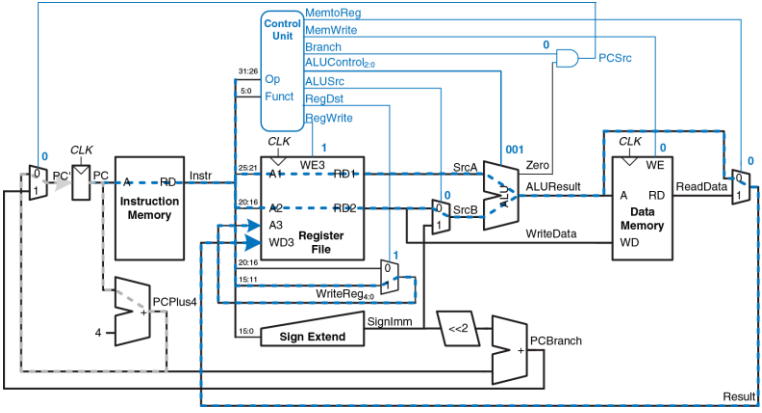


Рис. 7.13 Управляющие сигналы и пути движения данных при выполнении команды `or`

Запись в регистр результата также показана синей пунктирной линией. Номер этого регистра задается в поле `rd` ($Instr_{15:11}$), так что сигнал *RegDst* должен быть равен единице.

Запись нового значения в счетчик команд показана серой пунктирной линией. Так как команда or не является командой условного перехода, то сигнал Branch равен нулю и, соответственно, *PCSrc* тоже равен нулю. В результате счетчик команд получает новое значение из *PCPlus4*.

Важно иметь в виду, что по цепям, не отмеченным пунктиром, тоже передаются какие-то сигналы и данные, однако для этой конкретной команды совершенно не имеет значения, что они из себя представляют. Например, происходит расширение знака непосредственного операнда, а данные читаются из памяти, но это не оказывает никакого влияния на будущее состояние системы.

7.3.3 Дополнительные команды

Мы рассмотрели лишь малое подмножество полной системы команд MIPS. Добавим команды `addi` и `j`, чтобы проиллюстрировать процесс увеличения числа поддерживаемых команд. Кроме того, это даст нам систему команд, достаточную для написания множества интересных программ. Мы увидим, что поддержка некоторых новых команд зачастую заключается всего лишь в усложнении основного дешифратора, тогда как для других команд могут понадобиться дополнительные аппаратные блоки в тракте данных

Пример 7.2 КОМАНДА `addi`

Команда сложения с непосредственным операндом (`addi`) складывает значение одного из регистров со значением, хранящимся непосредственно в поле команды, и записывает результат в другой регистр. В тракте данных уже есть вся необходимая функциональность для выполнения этой команды. Определите, какие изменения необходимо внести в устройство управления, чтобы добавить поддержку команды `addi`.

Решение: все, что нужно сделать – это добавить новую строку в таблицу истинности основного дешифратора и заполнить ее значениями управляющих сигналов для команды `addi`, как показано в **Табл. 7.4**. Так как результат должен быть записан в регистровый файл, то *RegWrite* должен быть равен единице. Номер регистра результата указывается в поле `rt` команды Instr, так что *RegDst* равен нулю. На вход *SrcB* подается непосредственный операнд, так что *ALUSrc* равен единице. Так как команда `addi` не является командой условного перехода, а также не пишет в память, то сигналы *Branch* и *MemWrite* равны нулю. Результат формируется в АЛУ, а не читается из памяти, так что *MemtoReg* тоже равен нулю. Наконец, АЛУ должно выполнить сложение, так что сигнал *ALUOp* должен быть равен 00.

Табл. 7.4 Таблица истинности основного дешифратора с поддержкой `addi`

Команда	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
Команды типа R	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	001000	1	0	1	0	0	0	00

Пример 7.3 КОМАНДА `j`

Команда безусловного перехода `j` записывает новое значение в счетчик команд. Два младших бита нового значения всегда равны нулю, потому что длина любой команды равна четырем байтам, следовательно, ее адрес всегда кратен четырем. Следующие 26 бит процессор возьмет из поля адреса перехода (jump address field) $Instr_{25:0}$. Четыре оставшихся старших бита будут равны четырем старшим битам предыдущего значения счетчика команд.

Разработанный нами тракт данных не может вычислить значение *PC'* описанным образом. Определите, какие изменения нужно внести в тракт данных и в устройство управления, чтобы добавить поддержку команды `j`.

Решение: во-первых, надо добавить логику вычисления нового значения счетчика команд (*PC'*) для команды `j`, а также мультиплексор, чтобы можно было выбрать это значение. Как показано на **Рис. 7.14**, новый мультиплексор будет управляться сигналом *Jump*.

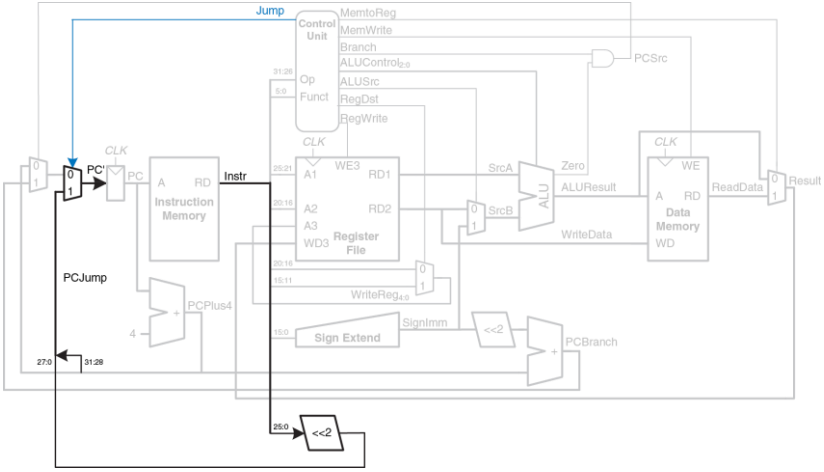


Рис. 7.14 Изменения в тракте данных для поддержки команды `j`

Потом в таблицу истинности основного дешифратора для команды `j` нужно добавить новую строку, а для управляющего сигнала *Jump* – новый столбец, как показано в **Табл. 7.5**. Управляющий сигнал *Jump* равен единице для команды `j` и нулю для всех остальных команд. Так как команда `j` не пишет ни в регистровый файл, ни в память, то сигналы *RegWrite* и *MemWrite* равны нулю, поэтому нам совершенно все равно, какие вычисления будут происходить в тракте данных. Таким образом, сигналы *RegDst*, *ALUSrc*, *Branch*, *MemtoReg* и *ALUOp* могут быть равны чему угодно.

Табл. 7.5 Таблица истинности основного дешифратора с поддержкой `j`

Команда	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
Команды типа R	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`addi`	001000	1	0	1	0	0	0	00	0
`j`	000010	0	X	X	X	0	X	XX	1

7.3.4 Анализ производительности

Каждая команда в однотактном процессоре выполняется ровно за один такт, так что CPI равно единице. Цепь с наибольшей задержкой (critical path) для команды `lw` показана на **Рис. 7.15** синей пунктирной линией. Она начинается там, где в счетчик команд по положительному фронту тактового сигнала записывается новое значение. Затем обновленное значение PC используется для выборки следующей команды. Потом процессор читает *SrcA* из регистрового файла, одновременно с этим знаково расширяя смещение, которое через мультиплексор *ALUSrc* подается на вход АЛУ как операнд *SrcB*. АЛУ складывает операнды *SrcA* и *SrcB*, вычисляя адрес в памяти (effective address). По этому адресу производится чтение из памяти данных. Мультиплексор *MemtoReg* выбирает *ReadData*. Наконец, сигнал *Result* должен стать стабильным на входе регистрового файла до того, как придет следующий положительный фронт тактового сигнала, иначе будет записано неверное значение. Таким образом, минимальная длительность одного такта равна:

$$T_c = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, t_{sext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \quad (7.2)$$

В большинстве технологий производства микросхем доступ к АЛУ, памяти и регистровым файлам занимает гораздо больше времени, чем

прочие операции. Таким образом, мы можем приближенно посчитать длительность одного такта как:

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup} \quad (7.3)$$

Численное значение длительности такта зависит от конкретной технологии.

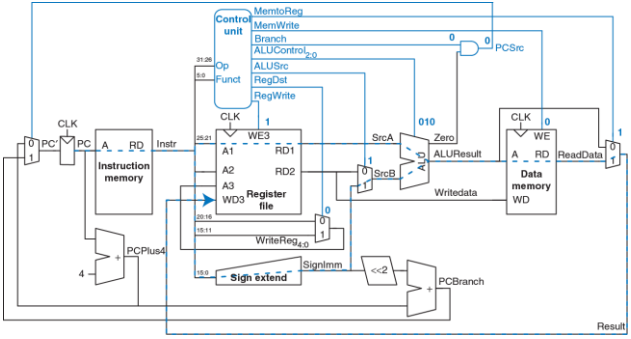


Рис. 7.15 Цепь с наибольшей задержкой для команды lw

У других команд цепи с наибольшей задержкой могут быть короче. Например, командам типа R не нужно обращаться к памяти данных.

Тем не менее, раз уж мы разрабатываем синхронные последовательностные схемы, то период тактового сигнала всегда должен определяться самой медленной командой.

Пример 7.4 ПРОИЗВОДИТЕЛЬНОСТЬ ОДНОТАКТНОГО ПРОЦЕССОРА

Бен Битдидл подумывает построить однотактный процессор MIPS по 65-нм КМОП-техпроцессу. Он выяснил, что задержки логических элементов такие же, как в **Табл. 7.6**. Помогите ему вычислить время выполнения программы, состоящей из 100 миллиардов команд.

Решение: согласно **уравнению (7.3)**, длительность такта однотактного процессора равна:

T_{c1} = 30 + 2(250) + 150 + 200 + 25 + 20 = 925 пс.

Мы использовали индекс “1”, чтобы можно было отличить однотактный процессор от других процессоров, которые мы разработаем позже. Согласно Уравнению 7.1, общее время выполнения программы составит:

T_1 = (100 × 10^9 команд) (1 такт/команду) (925 × 10^{-12} с/такт) = 92,5 с.

Табл. 7.6 Задержки элементов

Элемент	Параметр	Задержка (пс)
Задержка распространения clk-to-Q в регистре	t_{pcq}	30
Время предустановки регистра	t_{setup}	20
Мультиплексор	t_{mux}	25
АЛУ	t_{ALU}	200
Чтение из памяти	t_{mem}	250
Чтение из регистрового файла	t_{RFread}	150
Время предустановки регистрового файла (register file setup)	tRF_{setup}	20

7.4 МНОГОТАКТНЫЙ ПРОЦЕССОР

У однотактного процессора три основных проблемы. Во-первых, период его тактового сигнала должен быть достаточно большим, чтобы успела выполниться самая медленная команда (`lw`), несмотря на то, что большинство остальных команд гораздо быстрее. Во-вторых, ему нужно три сумматора (один для АЛУ и два для вычисления нового значения счетчика команд); сумматоры, особенно быстрые, требуют множества логических элементов, что делает их относительно дорогими схемами. В-третьих, ему требуется отдельная память команд и данных, что зачастую нереально. В большинстве компьютеров используют общую память для команд и данных, доступную для чтения и записи.

Один из способов решить эти проблемы – использовать многотактный процессор, в котором выполнение каждой команды разбивается на несколько этапов. На каждом этапе процессор может читать или писать данные в память или регистровый файл, или выполнять какую-нибудь операцию в АЛУ. У разных команд в этом случае будет разное количество этапов, так что простые команды смогут выполняться быстрее, чем сложные. Понадобится только один сумматор; на разных этапах он может использоваться для разных целей. Также процессор сможет обходиться общей памятью для команд и данных. Команды будут выбираться на первом этапе, а чтение или запись данных будут происходить на одном из последующих этапов.

Мы будем разрабатывать многотактный процессор тем же способом, что и однотактный. Сначала сконструируем тракт данных, соединяя при помощи комбинационной логики блоки памяти и блоки, хранящие архитектурное состояние процессора. Однако, помимо этого, мы добавим и другие блоки для хранения информации о промежуточном (неархитектурном) состоянии между этапами. После этого займемся устройством управления. Так как теперь оно должно формировать разные управляющие сигналы в зависимости от текущего этапа выполнения команды, то вместо комбинационных схем нам понадобится конечный автомат. После этого посмотрим, как добавлять поддержку новых команд. Напоследок мы снова оценим производительность и сравним ее с производительностью однотактного процессора.

7.4.1 Многотактный тракт данных

Как и прежде, в основу нашей разработки мы положим показанные на **Рис. 7.16** элементы, хранящие состояние – память и архитектурное состояние процессора. В однотактном процессоре мы использовали раздельную память команд и данных, потому что нужно было за один и тот же такт и читать из памяти команд, и обращаться к памяти данных. Теперь мы будем использовать общую память, хранящую и команды, и данные. Это более реалистичный сценарий, и теперь он возможен

благодаря тому, что мы можем выбирать команду на одном такте, а обращаться к памяти данных на другом. Счетчик команд и регистровый файл при этом изменений не претерпели. Шаг за шагом мы будем добавлять новые компоненты, нужные для каждого их этапов выполнения команды. Новые элементы и цепи будем выделять черным (или синим, в случае управляющих сигналов), а уже рассмотренные элементы будем перекрашивать серым.

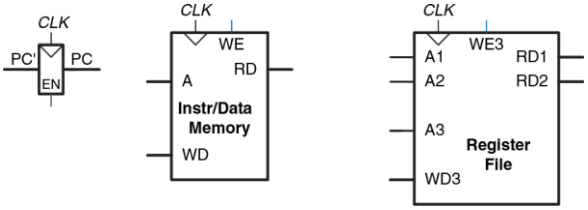


Рис. 7.16 **Общая память команд и данных и хранящие архитектурное состояние элементы**

Счетчик команд содержит адрес команды, которая должна быть выполнена следующей. Соответственно, первым делом надо прочитать ее из памяти команд. Как показано на **Рис. 7.17**, счетчик команд напрямую подсоединен к адресному входу памяти команд. Прочитанная из памяти команда сохраняется во временный (неархитектурный) регистр команд (Instruction Register), так что мы сможем использовать ее в следующих тактах. Сигнал разрешения записи в регистр команд назовем *IRWrite* и будем использовать его, когда потребуется обновить находящуюся в регистре команду.

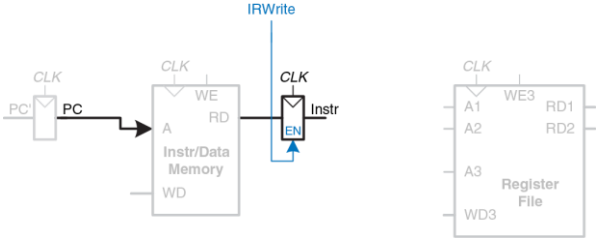


Рис. 7.17 **Выборка команды из памяти**

Как и в случае с однотактным процессором, начнем с команды `lw`, после чего будем добавлять новые компоненты, необходимые для поддержки других команд. Для команды `lw` следующим шагом будет

чтение регистра, содержащего базовый адрес. Номер регистра указывается в поле `rs` ($Instr_{25:21}$) и подается на адресный вход первого порта (*A1*) регистрового файла, как показано на **Рис. 7.18**. Значение, прочитанное из регистрового файла, появляется на его выходе *RD1*, после чего сохраняется во временный регистр *A*.

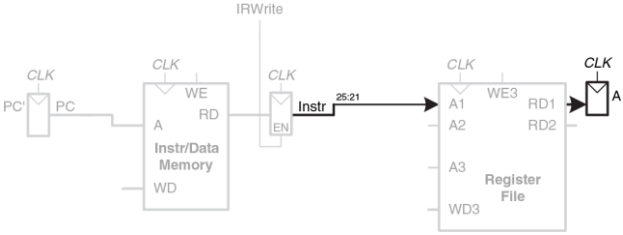


Рис. 7.18 Чтение операнда из регистрового файла

Команде `lw` также нужно смещение, которое представляет собой непосредственный операнд (immediate), находящийся в одном из полей команды, $Instr_{15:0}$. Над этим операндом выполняется операция знакового расширения, в результате чего получается 32-битное число *SignImm*, как показано на **Рис. 7.19**. Мы могли бы сохранить *SignImm* в еще один временный регистр, но так как *SignImm* – это выход комбинационной схемы, вход которой зависит исключительно от *Instr*,

а *Instr* не будет меняться все то время, пока команда выполняется, то нет смысла добавлять еще один временный регистр для хранения константного значения.

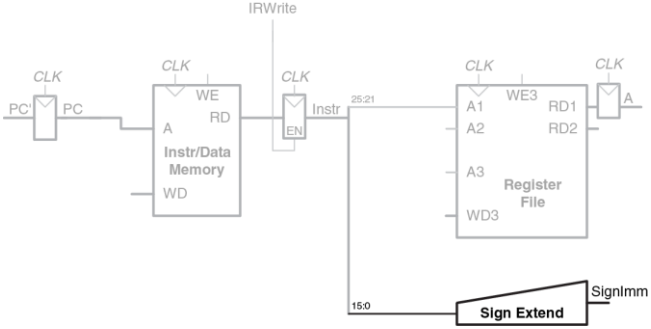


Рис. 7.19 Знаковое расширение непосредственного операнда

Адрес, по которому мы должны читать из памяти, получается путем сложения базового адреса и смещения. Для сложения мы используем АЛУ, как показано на **Рис. 7.20**. Чтобы АЛУ выполнило сложение, управляющий сигнал *ALUControl* должен быть равен 010. *ALUResult* сохраняется во временном регистре *ALUOut*.

Следующим шагом мы должны прочитать данные из памяти, используя только что вычисленный адрес. Для этого перед адресным входом памяти необходимо добавить мультиплексор, чтобы в качестве адреса *Adr* можно было использовать либо PC, либо *ALUOut*, как показано на **Рис. 7.21**.

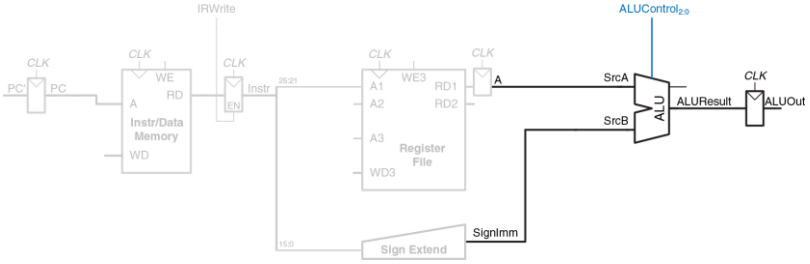


Рис. 7.20 Сложение базового адреса и смещения

Этот мультиплексор управляется сигналом *IorD*, показывающим, должен ли быть подан адрес команды или адрес данных. Прочитанные из памяти данные сохраняются во временном регистре *Data*. Заметьте, что мультиплексор адреса позволяет нам повторно использовать память во время выполнения команды lw. Сначала в качестве адреса мы используем PC, что позволяет выбрать команду. Затем в качестве адреса мы используем *ALUOut* и читаем данные. Таким образом, управляющий сигнал *IorD* должен принимать разные значения на разных этапах выполнения команды.

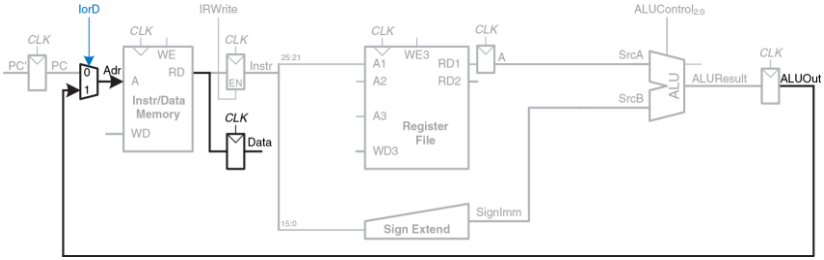


Рис. 7.21 Загрузка данных из памяти

В **разделе 7.4.2** мы создадим конечный автомат, который будет формировать требуемую последовательность управляющих сигналов.

Наконец, данные должны быть записаны в регистровый файл, как показано на **Рис. 7.22**. Номер регистра результата определяется полем `rt` (*Instr*$_{20:16}$).

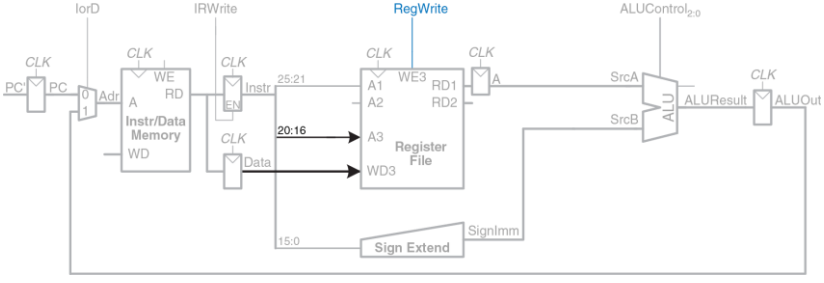


Рис. 7.22 Запись данных в регистровый файл

За то время, пока выполняются все вышеперечисленные операции, процессор должен увеличить счетчик команд на четыре. В однотактном процессоре для этого нам потребовался отдельный сумматор. В многотактном процессоре мы можем использовать уже имеющееся АЛУ на одном из первых этапов, пока оно еще не используется.

Для этого понадобится добавить пару мультиплексоров, которые позволят подавать на входы АЛУ содержимое счетчика команд PC и константу 4, как показано на **Рис. 7.23**.

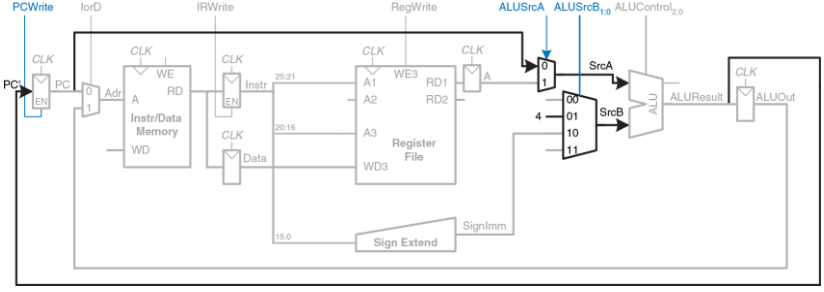


Рис. 7.23 Увеличение счетчика команд на четыре

Двухвходовой мультиплексор (мультиплексор 2:1), управляемый сигналом *ALUSrcA*, подает на *SrcA* либо PC, либо регистр *A*. Четырехвходовой мультиплексор (мультиплексор 4:1), управляемый сигналом *ALUSrcB*, подает на *SrcB* либо константу 4, либо *SignImm*. Оставшиеся два входа мультиплексора нам понадобятся позже, когда мы будем добавлять новые команды (нумерация входов мультиплексоров может быть любой). Для того, чтобы обновить счетчик

команд, АЛУ складывает *SrcA* (*PC*) и *SrcB* (4) и записывает полученный результат в счетчик команд. Управляющий сигнал *PCWrite* разрешает запись в счетчик команд только на тех тактах, где это необходимо.

На этом создание тракта данных для команды `lw` завершено. Теперь добавим поддержку команды `sw`. Как и команда `lw`, `sw` читает базовый адрес из первого порта регистрового файла и выполняет знаковое расширение непосредственного операнда, после чего АЛУ складывает их, получая адрес для записи в память. Все эти функции уже есть в тракте данных.

Единственное отличие `sw` – это то, что мы должны прочитать еще один регистр из регистрового файла и записать его содержимое в память, как показано на **Рис. 7.24**. Номер регистра указан в поле `rt` ($Instr_{20:16}$), которое подключено ко второму порту регистрового файла. Прочитанное значение сохраняется во временном регистре *B*. На следующем этапе оно подается на порт записи данных (*WD*) памяти. Новый управляющий сигнал *MemWrite* показывает, когда именно данные должны быть записаны в память.

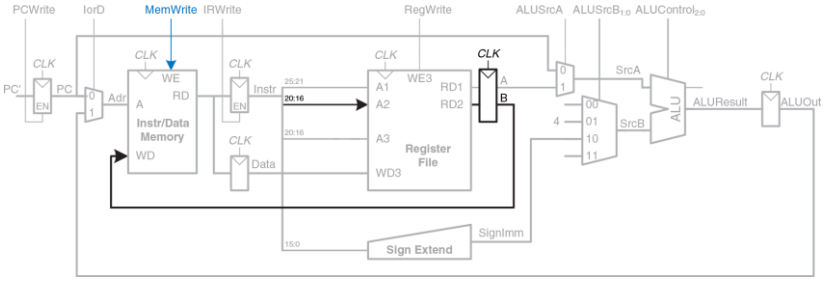


Рис. 7.24 Изменения в тракте данных для поддержки команды sw

В случае команд типа R сразу после выборки команды из памяти из регистрового файла читаются два операнда. Сигнал $ALUSrcB_{1:0}$, управляющий мультиплексором *SrcB*, позволяет выбрать регистр *B* в качестве второго операнда АЛУ, как показано на **Рис. 7.25**. АЛУ выполняет требуемую операцию и сохраняет результат в *ALUOut*. На следующем этапе *ALUOut* записывается обратно в регистр, номер которого указан в поле `rd` ($Instr_{15:11}$). Для этого нам потребуются два дополнительных мультиплексора. Мультиплексор *MemtoReg* подает на *WD3* либо *ALUOut* (для команд типа R), либо *Data* (для `lw`). Мультиплексор *RegDst* позволяет выбрать, в каком поле команды находится номер регистра, куда будет записан результат – `rt` или `rd`.

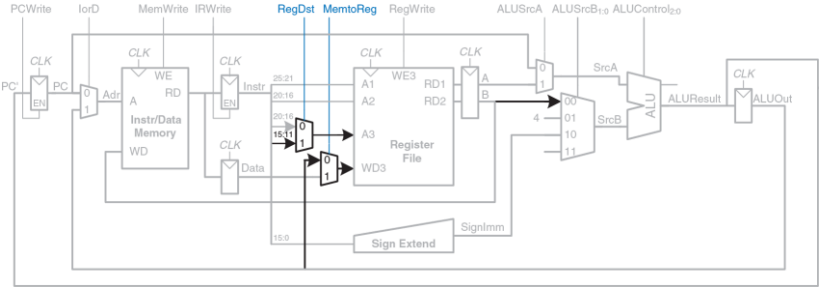


Рис. 7.25 Изменения в тракте данных для поддержки команд типа R

В случае команды `beq` сразу после выборки команды из памяти из регистрового файла читаются два операнда. Для того чтобы определить, равны они или нет, АЛУ вычитает один из другого и в том случае, если результат равен нулю, устанавливает флаг нуля (*Zero* flag). Одновременно с этим тракт данных должен вычислить следующее значение счетчика команд на тот случай, если условный переход будет все-таки выполнен: *PC'* = *PC* + 4 + *SignImm* × 4. В однотактном процессоре для этого использовался еще один сумматор. Теперь же мы можем использовать АЛУ в третий раз и сэкономить еще несколько логических элементов. Таким образом, сначала АЛУ вычислит значение *PC* + 4 и запишет его в счетчик команд, также как и для всех прочих

команд. Затем АЛУ использует обновленное значение PC для вычисления *PC* + *SignImm* × 4. Для умножения *SignImm* на четыре мы, как и прежде, просто сдвинем его влево на два разряда, как показано на **Рис. 7.26**. Мультиплексор *SrcB* выбирает сдвинутое значение, после чего оно суммируется с PC, в результате чего получается адрес перехода, который и сохраняется в *ALUOut*. Еще один новый мультиплексор, управляемый сигналом *PCSrc*, выбирает один из сигналов и подает его на *PC'*. Счетчик команд обновляется, если установлен сигнал *PCWrite*, или если выполнен условный переход. Управляющий сигнал *Branch* показывает, является ли выполняющаяся команда командой условного перехода beq. Условный переход будет выполнен, если установлен флаг нуля. Таким образом, сигнал разрешения записи в счетчик команд *PCEn* равен единице или тогда, когда установлен сигнал *PCWrite*, или когда сигнал *Branch* установлен одновременно со флагом нуля.

На этом разработка многотактного тракта данных процессора MIPS закончена. Процесс разработки был очень похож на тот, который мы использовали для однотактного процессора, когда методически добавляли блок за блоком между элементами, хранящими состояние процессора. Главное же отличие заключалось в том, что каждая команда выполнялась в несколько этапов. Нам потребовались невидимые программисту временные (неархитектурные) регистры,

чтобы сохранять результаты каждого из этих этапов. За счет этого мы смогли повторно использовать одно и то же АЛУ, что позволило избавиться от нескольких сумматоров. Таким же образом мы смогли поместить команды и данные в общую память. В следующем разделе мы создадим конечный автомат, который будет формировать управляющие сигналы для каждого этапа в нужной последовательности.

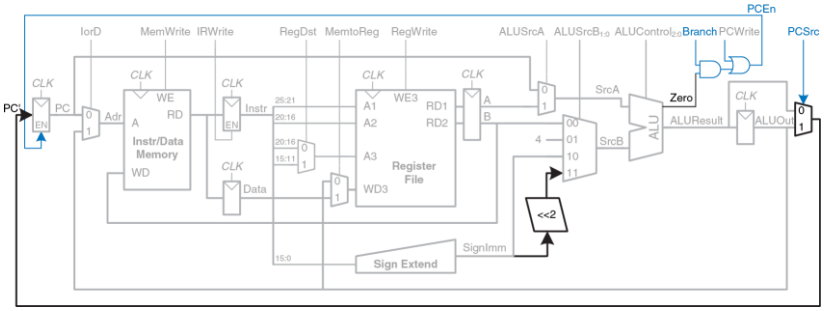


Рис. 7.26 Изменения в тракте данных для поддержки команды beq

7.4.2 Многотактное устройство управления

Как и в однотактном процессоре, устройство управления формирует управляющие сигналы в зависимости от полей opcode и funct команды ($Instr_{31:26}$ и $Instr_{5:0}$ соответственно). На **Рис. 7.27** показан многотактный процессор MIPS с устройством управления, подключенным к тракту данных. Тракт данных показан черным цветом, а устройство управления – синим.

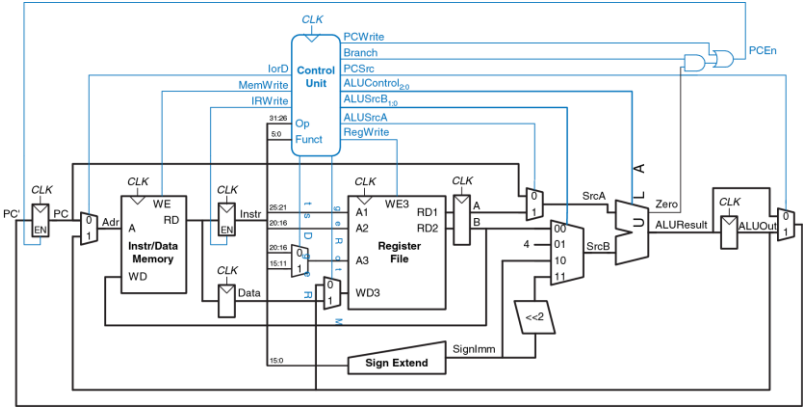


Рис. 7.27 Законченный многотактный процессор MIPS

Как и в однотактном процессоре, устройство управления поделено на две части, как показано на **Рис. 7.28**. Дешифратор АЛУ остается точно таким же, как и раньше; его таблица истинности приведена в **Табл. 7.2**.

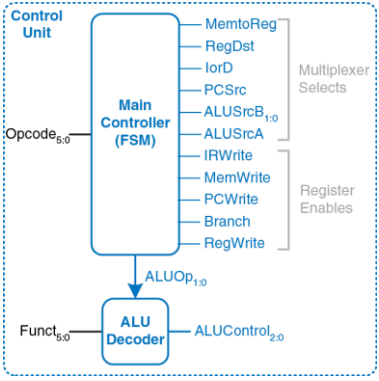


Рис. 7.28 Изменения в тракте данных для поддержки команд типа R

А вот вместо основного дешифратора нам понадобится конечный автомат, так как управляющие сигналы теперь будут зависеть не только от того, какая команда выполняется в данный момент, но и от того, на каком этапе выполнения она находится. Будем называть этот автомат *управляющим автоматом*. Оставшуюся часть раздела мы посвятим разработке его диаграммы состояний.

Управляющий автомат формирует сигналы управления мультиплексорами и сигналы разрешения записи в регистры тракта данных. Сигналы для мультиплексоров называются *MemtoReg*, *RegDst*, *IorD*, *PCSrc*, *ALUSrcB* и *ALUSrcA*. Сигналы разрешения записи называются *IRWrite*, *MemWrite*, *PCWrite*, *Branch* и *RegWrite*.

Чтобы сделать диаграмму состояний более удобочитаемой, мы будем указывать только те управляющие сигналы, которые имеют смысл на конкретном этапе выполнения команды. Сигналы управления мультиплексорами будем указывать лишь тогда, когда они действительно используются. Сигналы разрешения записи будем указывать, только если они не равны нулю.

Первым этапом каждой команды является чтение из памяти по адресу, находящемуся в счетчике команд, то есть выборка команды из памяти. В это состояние управляющий автомат переходит по сигналу сброса (reset). Чтобы адрес для чтения был взят из счетчика команд, *IorD* должен быть равен нулю. Чтобы прочитанное значение попало в регистр команд (IR), *IRWrite* устанавливается в единицу. Одновременно с этим счетчик команд должен быть увеличен на четыре – после этого он будет указывать на следующую команду. Так как АЛУ в этот момент свободно, то процессор может использовать его для вычисления PC + 4 одновременно с выборкой команды из памяти. *ALUSrcA* = 0, поэтому на первый вход АЛУ (*SrcA)* подается PC. *ALUSrcB* = 01, поэтому на второй

вход АЛУ (*SrcB)* подается константа 4. *ALUOp* = 00, так что дешифратор АЛУ устанавливает сигнал *ALUControl* равным 010, чтобы АЛУ выполнило именно сложение. Чтобы содержимое счетчика команд обновилось, нужно, чтобы *PCSrc* был равен нулю, а *PCWrite* – единице. Все эти управляющие сигналы показаны на **Рис. 7.29**.

Рис. 7.29 Этап выборки команды

Даные на этом этапе проходят через тракт так, как показано на **Рис. 7.30**. Выборка команды показана синим пунктиром, а увеличение счетчика команд – серым.

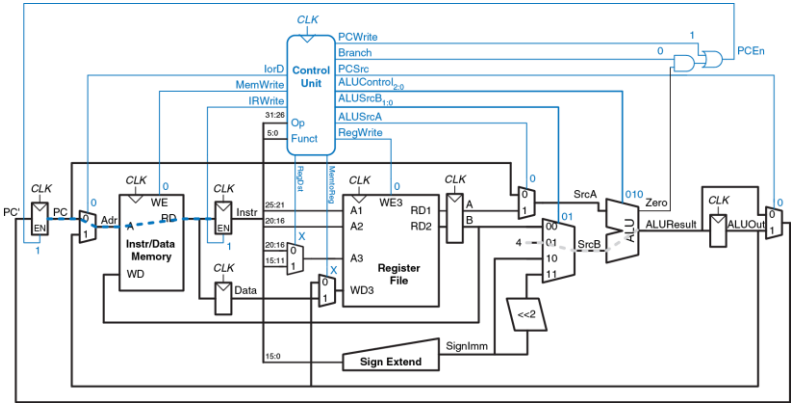


Рис. 7.30 Пути движения данных на этапе выборки команды

На следующем этапе происходит чтение из регистрового файла и дешифрация команды. Из регистрового файла всегда читаются два регистра, номера которых указаны в полях `rs` и `rt` команды *Instr*. Одновременно с этим происходит знаковое расширение непосредственного операнда. Дешифрация команды заключается в том, что процессор смотрит в поле `opcode`, чтобы понять, что от него требуется. Для дешифрации никакие управляющие сигналы не нужны, однако управляющий автомат должен подождать один такт, чтобы она завершились. Новое состояние выделено на **Рис. 7.31** синим цветом. Данные на этом этапе проходят через тракт так, как показано на **Рис. 7.32**.

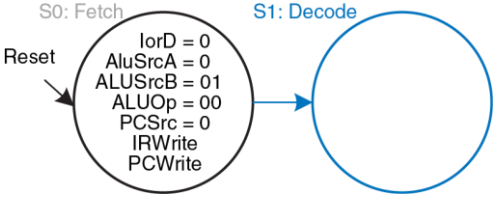


Рис. 7.31 **Этап дешифрации команды**

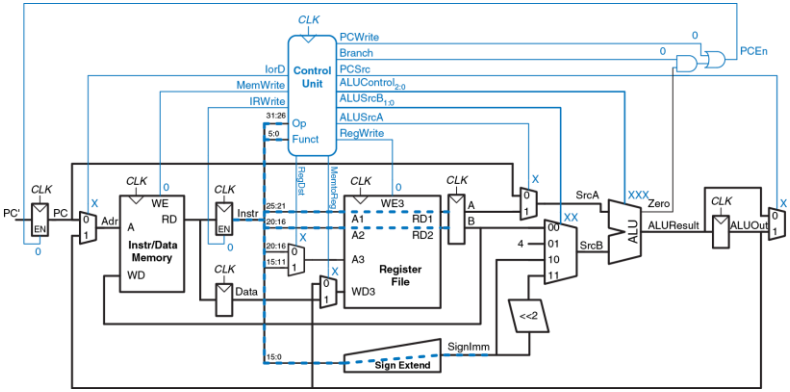


Рис. 7.32 Пути движения данных на этапе дешифрации команды

Теперь, в зависимости от поля `opcode`, управляющий автомат должен перейти в одно из нескольких состояний. Если выполняется команда загрузки или сохранения данных (`lw` или `sw`), процессор вычисляет адрес в памяти путем сложения базового адреса и непосредственного операнда с расширенным знаком. *ALUSrcA* = 1, чтобы в качестве первого операнда был выбран регистр *A*. *ALUSrcB* = 10, чтобы в качестве второго операнда был выбран сигнал *SignImm*. *ALUOp* = 00,

поэтому АЛУ выполнит сложение. Полученный адрес сохраняется в регистр *ALUOut* для использования на следующем этапе. Новое состояние выделено на **Рис. 7.33** синим цветом. Данные на этом этапе проходят через тракт так, как показано на **Рис. 7.34**.

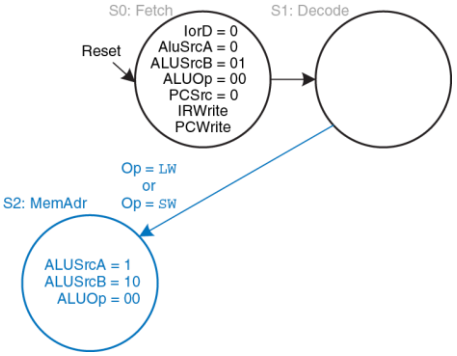


Рис. 7.33 Вычисление адреса в памяти данных

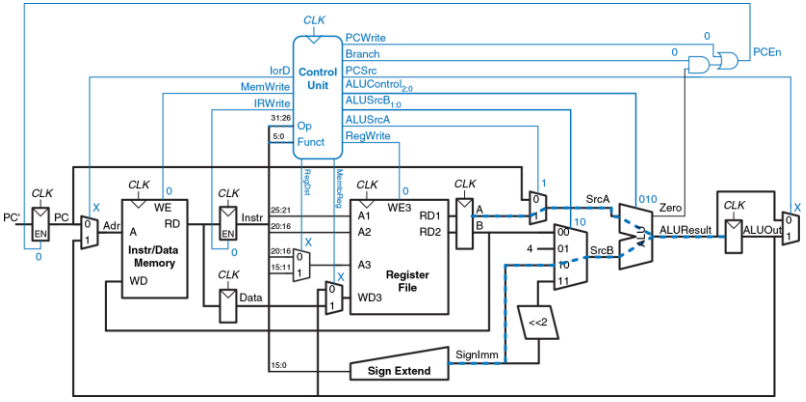


Рис. 7.34 Пути движения данных при вычислении адреса

Далее, в случае команды `lw` многотактный процессор должен прочитать данные из памяти и сохранить их в регистровый файл. Эти два этапа показаны на **Рис. 7.35**.

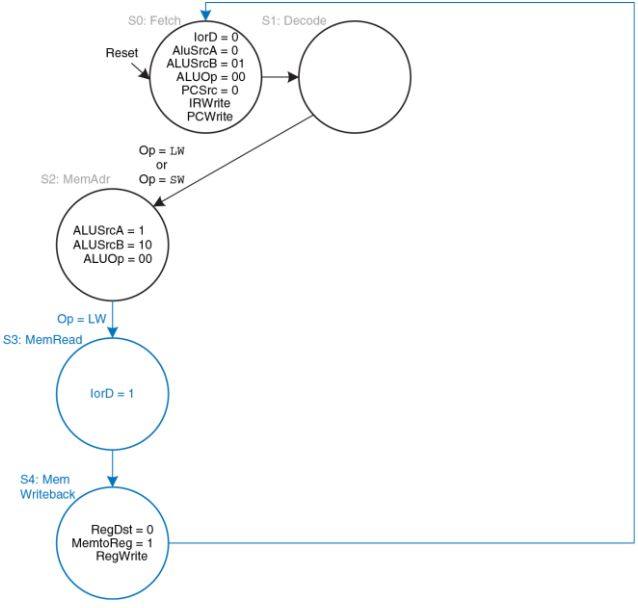


Рис. 7.35 **Этапы чтения данных из памяти**

Для чтения данных из памяти сигнал *IorD* должен быть равен единице – в этом случае адрес будет взят из регистра *ALUOut*. Прочитанное значение сохраняется в регистр Data на этапе S3. На следующем этапе (S4), содержимое *Data* записывается в регистровый файл, для чего сигнал *MemtoReg* устанавливается в единицу, а *RegDst* – в ноль. Для завершения записи управляющий автомат устанавливает сигнал *RegWrite.* Наконец, автомат возвращается в первоначальное состояние S0, чтобы выбрать из памяти следующую команду. Для этого и последующих этапов постарайтесь нарисовать диаграмму состояний самостоятельно.

В случае команды sw значение, прочитанное в состоянии S2 из второго порта регистрового файла, должно быть записано в память. Для этого добавим состояние S5, в котором *IorD* = 1, чтобы в качестве адреса использовалось содержимое регистра *ALUOut,* полученное в предыдущем состоянии. Для записи в память также потребуется установить сигнал *MemWrite*. После этого управляющий автомат должен вернуться в состояние S0, чтобы выбрать из памяти следующую команду. Соответствующая диаграмма состояний приведена на **Рис. 7.36**.

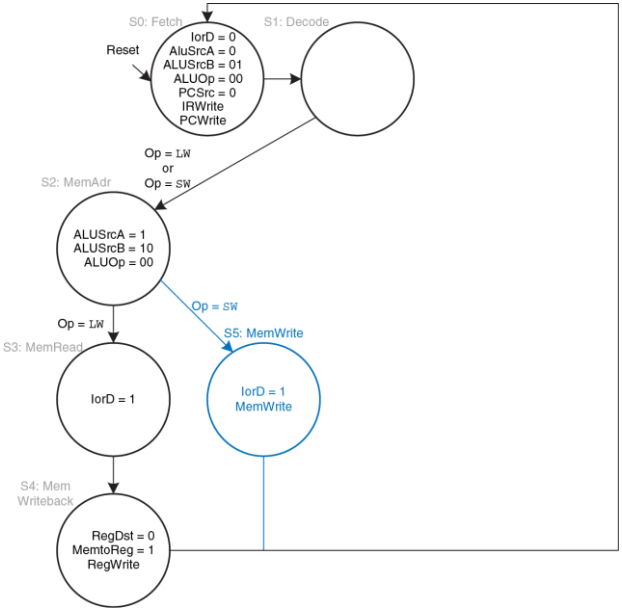


Рис. 7.36 Этап записи в память данных

В случае команд типа R многотактный процессор должен посчитать результат в АЛУ и записать его в память. На **Рис. 7.37** показаны два новых состояния. В состоянии S6 процессор выбирает в качестве операндов регистры *A* и *B* (*ALUSrcA* = 1, *ALUSrcB* = 00), после чего АЛУ производит над ними вычисления согласно содержимому поля funct (*ALUOp* = 10 для всех команд типа R). После этого *ALUResult* сохраняется в *ALUOut*. В состоянии S7 *ALUOut* записывается обратно в регистровый файл. Сигнал *RegDst* устанавливается равным единице, так как номер регистра задан в поле rd. *MemtoReg* = 0, так как данные для записи (*WD3)* идут из *ALUOut.* Наконец, для записи в регистровый файл *RegWrite* устанавливается в единицу.

Для выполнения команды beq процессор должен вычислить адрес перехода и сравнить два регистра, чтобы определить, нужно ли перейти по этому адресу. Для этого нужно два раза использовать АЛУ, что, казалось бы, подразумевает добавление еще двух новых состояний. Заметьте, однако, что в состоянии S1 АЛУ не используется, поэтому процессор вполне может воспользоваться этим для вычисления адреса перехода путем сложения счетчика команд, который в состоянии S1 уже равен *PC* + 4, с умноженным на четыре сигналом *SignImm*, как показано на **Рис. 7.38**.

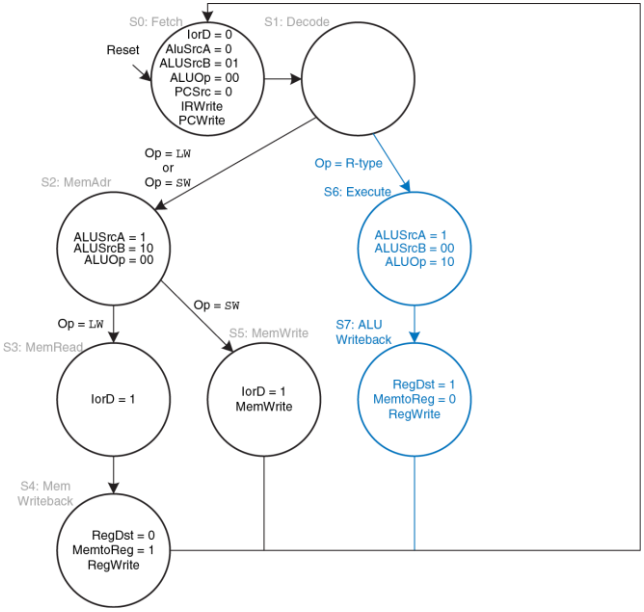


Рис. 7.37 Этапы вычисления результатов команд типа R

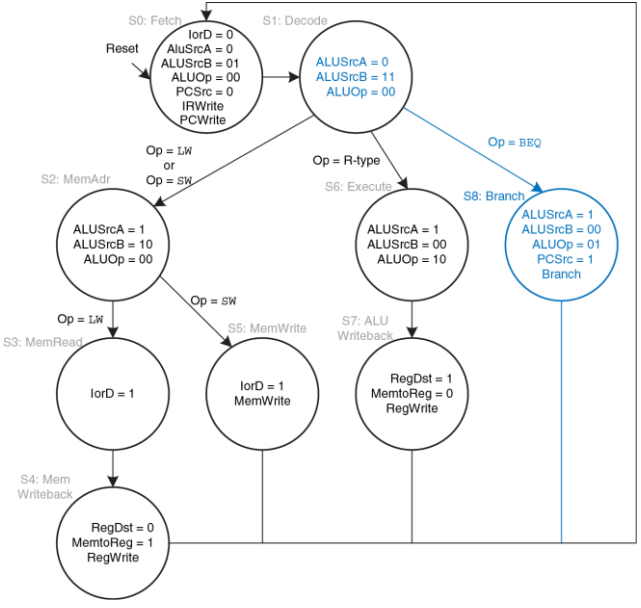


Рис. 7.38 Этапы выполнения команды `beq`

Для этого *ALUSrcA* = 0, *ALUSrcB* = 11 и *ALUOp* = 00. Полученный адрес перехода сохраняется в регистре *ALUOut*. Если бы выполнялась не beq, а другая команда, то адрес перехода, который вычислил процессор, был бы не нужен, однако и никакого вреда от этого бы не было.

В состоянии S8 процессор сравнивает два регистра, вычитая один из другого и проверяя результат на равенство нулю. Если получился ноль, то процессор выполняет переход по только что вычисленному адресу. Регистр A подается на вход АЛУ, когда *ALUSrcA* = 1, регистр B – когда *ALUSrcB* = 00. Для того чтобы АЛУ вычитало, *ALUOp* = 01. *PCSrc* = 1, чтобы адрес перехода был получен из *ALUOut*, а *Branch* = 1, чтобы он был записан в счетчик команд, если результат вычитания равен нулю.[9]

Полная диаграмма состояний управляющего автомата многотактного процессора показана на **Рис. 7.39**. Превращение этой диаграммы в логическую схему – простая, но утомительная задача, которую можно выполнить одним из способов, описанных в **главе 3**, либо закодировать ее на языке описания аппаратуры и синтезировать, как описано в **главе 4**.

[9] Теперь мы видим, почему необходим мультиплексор PCSrc, подающий на вход записи счетчика команд или ALUResult (в состоянии S0), или ALUOut (в состоянии S8).

Рис. 7.39 Полная диаграмма состояний управляющего автомата

7.4.3 Дополнительные команды

В разделе 7.3.3 мы добавили в тракт данных однотактного процессора поддержку команд `addi` и `j`. Давайте теперь сделаем то же самое для многотактного процессора. Следующие два примера иллюстрируют этот процесс.

Пример 7.5 КОМАНДА `addi`

Модифицируйте многотактный процессор, чтобы добавить поддержку команды `addi`.

Решение: в тракте данных уже есть вся необходимая функциональность для сложения регистров и непосредственных операндов, так что нам нужно всего лишь добавить несколько новых состояний в управляющий автомат, как показано на Рис. 7.40. Новые состояния похожи на те, которые используются для команд типа R. В состоянии S9 регистр *A* складывается с *SignImm* (*ALUSrc*A = 1, *ALUSrcB* = 10, *ALUOp* = 00), а результат (*ALUResult*) сохраняется в *ALUOut*. В состоянии S10 ALUOut записывается в регистр, номер которого указан в поле `rt` команды *Instr* (*RegDst* = 0, *MemtoRe*g = 0, сигнал *RegWrite* активен). Внимательный читатель мог заметить, что состояния S2 и S9 одинаковы и могли бы быть объединены в одно.

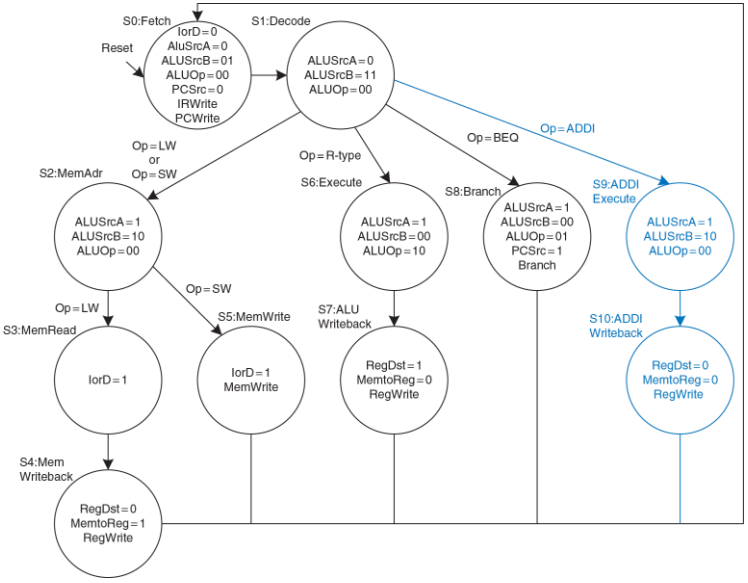


Рис. 7.40 **Диаграмма состояний управляющего автомата для команды `addi`**

Пример 7.6 КОМАНДА `j`

Модифицируйте многотактный процессор, чтобы добавить поддержку команды `j`.

Решение: во-первых, мы должны добавить в тракт данных логику вычисления значения счетчика команд для команды `j`. После этого надо будет добавить еще одно новое состояние в управляющий автомат.

На **Рис. 7.41** показан модифицированный тракт данных. Адрес безусловного перехода вычисляется путем сдвига 26-битного поля `addr` на два разряда влево, что дает нам 28 бит адреса. Оставшиеся четыре старших бита копируются из старших битов счетчика команд (PC). Получившийся адрес нужно подать на мультиплексор *PCSrc*, для чего число его входов данных нужно увеличить с двух до трех.

На **Рис. 7.42** показана модифицированная диаграмма состояний управляющего автомата. В новом состоянии S11 в счетчик команд будет записано значение *PC'*, равное *PCJump* (*PCSrc* = 10). Заметьте, что из-за того, что управляющий сигнал *PCSrc* стал двухбитовым, пришлось также изменить состояния S0 и S8.

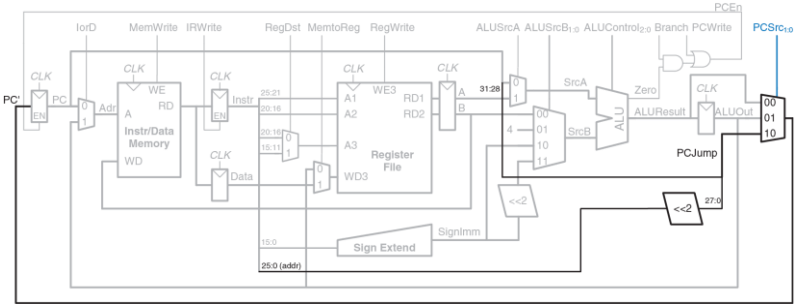


Рис. 7.41 **Изменения в тракте данных для поддержки команды j**

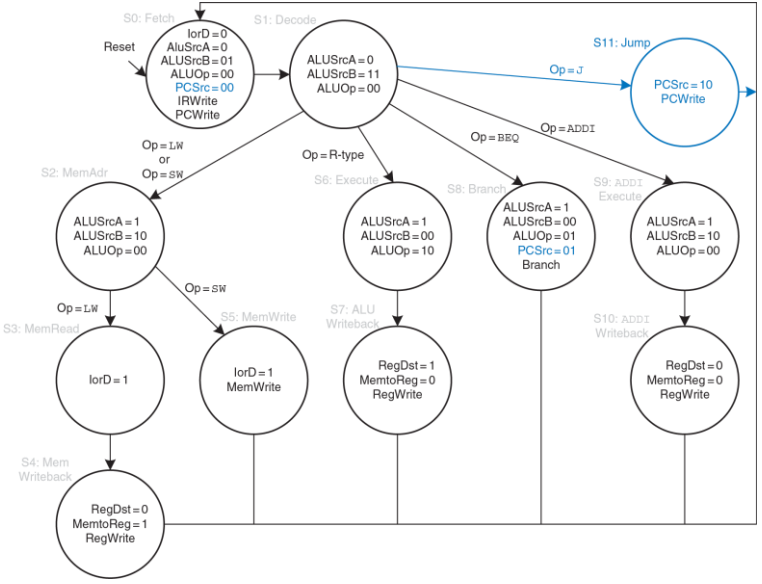


Рис. 7.42 **Диаграмма состояний управляющего автомата для команды `j`**

7.4.4 Оценка производительности

Время выполнения команды зависит от требуемого количества тактов и их длительности. В отличие от однотактного процессора, который выполняет все команды за один такт, многотактному процессору для выполнения разных команд требуется разное число тактов. Однако поскольку он выполняет меньшее количество действий за такт, то его такты гораздо короче, чем у однотактного.

Для команд `beq` и `j` многотактному процессору нужно три такта, для команд `sw`, `addi` и команд типа R – четыре, а для команды `lw` – пять. Суммарное число тактов на команду (CPI) будет зависеть от частоты использования каждой из команд.

Мы разработали многотактный процессор так, чтобы на каждом такте он выполнял либо арифметическую операцию в АЛУ, либо операцию доступа к памяти, либо операцию доступа к регистровому файлу. Давайте предположим, что доступ к регистровому файлу быстрее, чем к памяти, и что запись в память быстрее, чем чтение. При внимательном изучении тракта данных оказывается, что, скорее всего цепью с максимальной задержкой, определяющей минимальную длительность такта, будет одна из этих двух:

$$T_c = t_{pcq} + t_{mux} + max(t_{ALU} + t_{mux}, t_{mem}) + t_{\text{setup}} \quad (7.4)$$

Численное значение длительности такта будет зависеть от конкретной технологии производства.

Пример 7.7 CPI МНОГОТАКТНОГО ПРОЦЕССОРА

Тестовый набор SPECINT2000 содержит примерно 25% команд загрузки, 10% команд сохранения, 11% команд условного перехода, 2% команд безусловного перехода и 52% команд типа R[10]. Определите среднее число тактов на команду (cycles per instruction, CPI) для этого тестового набора.

Решение: Среднее CPI можно вычислить, перемножив число тактов, требуемое для выполнения команды каждого типа, на относительное количество соответствующих команд и просуммировав полученные значения. Для этого тестового набора среднее CPI = (0,11 + 0,02)(3) + (0,52 + 0,10)(4) + (0,25)(5) = 4,12. Если бы все команды выполнялись одинаковое время, то CPI было бы равно пяти.

Пример 7.8 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл задумался о том, стоит ли ему разрабатывать многотактный процессор вместо однотактного. В обоих случаях он планирует использовать 65-нм КМОП техпроцесс. Задержки, характерные для этого техпроцесса, даны в **Табл. 7.6**. Помогите ему сравнить время выполнения тестового набора SPECINT2000, состоящего из 100 миллиардов команд, для обоих процессоров (см. **пример 7.7**).

[10] Данные из книги «Архитектура компьютера и проектирование компьютерных систем», Паттерсон, Хеннесси, изд. Питер, 2012.

Решение: согласно **уравнению (7.4)**, длительность такта многотактного процессора равна:

T_{c2} = 30 + 25 + 250 + 20 = 325 пс.

Используя значение CPI, рассчитанное в **примере 7.7** и равное 4,12, общее время выполнения программы многотактным процессором равно:

T_2 = (100 × 10^9 команд)(4,12 тактов/команду) (325 × 10^{-12} с/такт) = 133,9 с.

Как мы выяснили в **примере 7.4**, длительность такта однотактного процессора Tc1 = 925 пс, его CPI равно 1, а общее время выполнения программы равно 92,5 с.

При разработке многотактного процессора мы хотели избежать того, чтобы все команды выполнялись с той же скоростью, что и самая медленная. К сожалению, этот пример показывает, что для данных значений CPI и задержек элементов многотактный процессор медленнее, чем однотактный. Фундаментальная проблема оказалась в том, что, несмотря на разбиение самой медленной команды (`lw`) на пять этапов, длительность такта в многотактном процессоре уменьшилась вовсе не в пять раз. Одной из причин явилось то, что не все этапы стали одинаковой длины. Другой причиной стали накладные расходы, связанные со временными регистрами – задержка в 50 пс, равная сумме времени предустановки и времени срабатывания регистра, теперь добавляется не к общему времени выполнения команды, а к каждому этапу по отдельности. Инженеры выяснили, что довольно трудно использовать тот факт, что одни вычисления могут происходить быстрее, чем другие, если разница во времени вычисления мала.

В сравнении с однотактным процессором многотактный процессор, скорее всего, окажется меньшего размера, так как нет необходимости в двух дополнительных сумматорах, а память команд и данных объединена в один блок. Взамен ему требуется пять неархитектурных (временных) регистров и несколько дополнительных мультиплексоров.

7.5 КОНВЕЙЕРНЫЙ ПРОЦЕССОР

Конвейеризация, о которой мы говорили в **разделе 3.6** – это мощное средство увеличения пропускной способности цифровой системы. Мы разработаем конвейерный процессор, разделив однотактный процессор на пять стадий. Таким образом, пять команд смогут выполняться одновременно, по одной в каждой из стадий. Так как каждая стадия содержит только одну пятую от всей логики процессора, то частота тактового сигнала может быть почти в пять раз выше. В идеальном случае латентность команд не изменится, а пропускная способность вырастет в пять раз. Микропроцессоры выполняют миллионы и миллиарды команд в секунду, так что производительность важнее, чем латентность. Конвейеризация требует определенных накладных расходов, так что в реальной жизни пропускная способность будет ниже, чем в идеальном случае, но в любом случае у конвейеризации так много преимуществ, а обходится она так дешево, что все современные высокопроизводительные микропроцессоры – конвейерные.

Чтение и запись в память и регистровый файл, а также использование АЛУ, обычно привносят наибольшие задержки в процессоре. Мы поделим конвейер на стадии таким образом, чтобы каждая из них включала ровно одну из этих операций. Стадии мы назовем *Fetch (выборка)*, *Decode (дешифрация)*, *Execute (выполнение)*, *Memory*

(доступ к памяти) и *Writeback (запись результатов)*. Они похожи на пять этапов выполнения команды `lw` в многотактном процессоре. В стадии *Fetch* процессор читает команду из памяти команд. В стадии *Decode* процессор читает операнды из регистрового файла и дешифрует команду, чтобы установить управляющие сигналы. В стадии *Execute* процессор выполняет вычисления в АЛУ. В стадии *Memory* процессор читает или пишет в память данных. Наконец, в стадии *Writeback* процессор, если нужно, записывает результат в регистровый файл.

На **Рис. 7.43** приведена временная диаграмма для сравнения однотактного и конвейерного процессоров. По горизонтальной оси отложено время, а по вертикальной – команды. Значения задержек логических элементов на диаграмме взяты из **Табл. 7.6**; задержками мультиплексоров и регистров мы пренебрежем. В однотактном процессоре, показанном на **Рис. 7.43 (а)**, первая команда выбирается из памяти в момент времени 0; далее процессор читает операнды из регистрового файла; далее АЛУ выполняет необходимые вычисления. Наконец, происходит доступ к памяти данных, и результат записывается в регистровый файл через 950пс. Выполнение второй команды начинается после того, как закончена первая. Таким образом, как видно из диаграммы, однотактный процессор обеспечивает латентность команд, равную 250 + 150 + 200 + 250 + 100 = 950 пс,

и пропускную способность, равную одной команде за 950 пс (1,05 миллиарда команд в секунду).

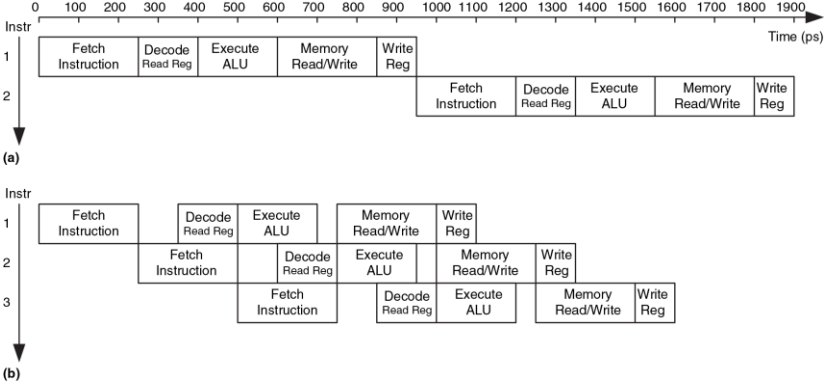


Рис. 7.43 Временная диаграмма однотактного (a) и конвейерного (b) процессоров

В конвейерном процессоре, приведенном на **Рис. 7.43 (b)**, длина стадии равна 250 пс и определяется самой медленной стадией – стадией доступа к памяти (Fetch или Memory). В начальный момент времени первая команда выбирается из памяти. Через 250 пс, первая команда попадает в стадию Decode, а вторая команда выбирается из памяти.

Когда прошло 500 пс, первая команда начинает выполняться, вторая попадает в стадию Decode, а третья выбирается. И так далее, пока все команды не завершатся. Латентность команд составляет 1250 пс. Пропускная способность – одна команда за 250 пс (четыре миллиарда команд в секунду). Латентность команд в конвейерном процессоре немного больше, чем в однотактном, потому что стадии конвейера не идеально сбалансированы, то есть не содержат абсолютно одинаковое количество логики. Аналогично, пропускная способность процессора с пятистадийным конвейером не в пять раз больше, чем у однотактного. Тем не менее, увеличение пропускной способности весьма значительно.

На **Рис. 7.44** показано абстрактное представление работающего конвейера, иллюстрирующее продвижение команд по конвейеру. Каждая стадия изображена картинкой, содержащей главный компонент стадии – память команд (instruction memory, IM), чтение регистрового файла (register file, RF), АЛУ, память данных (DM) и запись в регистровый файл (writeback). Если смотреть на строки, то можно узнать, на каком такте команда находится в той или иной стадии. Например, команда sub выбирается из памяти на третьем такте и выполняется на пятом. Если смотреть на столбцы, то можно узнать, чем заняты стадии конвейера на конкретном такте. Например, на шестом такте команда or выбирается из памяти команд, регистр `$s1` читается из регистрового файла, АЛУ вычисляет логическое И над

содержимым регистров `$t5` и `$t6`, память данных не используется и простаивает, а регистровый файл записывает результат сложения в `$s3`. Стадии закрашены серым, если они используются. Например, память данных используется командами `lw` на четвертом такте и `sw` на восьмом. Память команд и АЛУ используются на каждом такте. Результат записывается в регистровый файл всеми командами, кроме `sw`. В конвейерном процессоре значения записываются в регистровый файл в первой части такта, а читаются во второй, что так же отмечено на рисунке. В этом случае данные могут быть записаны и затем прочитаны обратно за один и тот же такт.

Главная проблема в конвейерных системах – разрешение конфликтов (*hazards*), которые возникают, когда результаты одной из команд требуются для выполнения последующей команды до того, как первая завершится. Например, если бы команда `add` на **Рис. 7.44** использовала регистр `$s2` вместо `$t2`, то возник бы конфликт, потому что регистр `$s2` еще не был бы записан командой `lw` в тот момент, когда команда `add` должна была его прочитать. В этом разделе мы рассмотрим пересылку данных через байпас (*bypassing, или forwarding*), приостановки (*stalls*) и сбросы (*flushes*) конвейера в качестве методов разрешения конфликтов. После этого мы повторно оценим производительность, приняв во внимание накладные расходы на

организацию конвейерной обработки (sequencing) и влияние конфликтов.

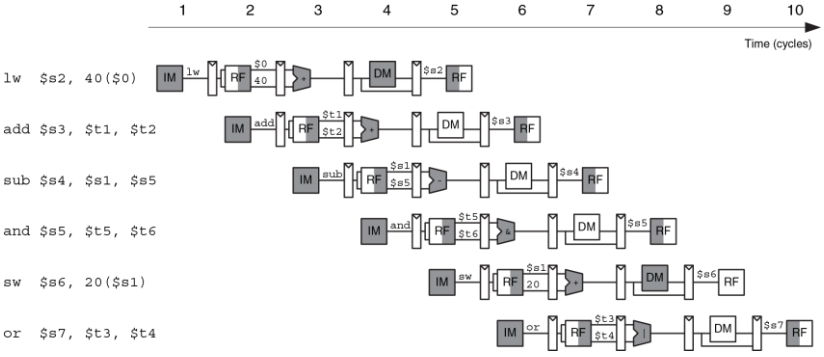


Рис. 7.44 Абстрактное представление работающего конвейера

7.5.1 Конвейерный тракт данных

Конвейерный тракт данных можно получить, порезав однотактный тракт данных на пять стадий, разделенных регистрами (pipeline registers). На **Рис. 7.45 (a)** показан однотактный тракт данных, растянутый таким образом, чтобы оставить место для регистров между стадиями.

На **Рис. 7.45 (b)** показан конвейерный тракт данных, поделенный на пять стадий путем вставки в него четырех регистров. Названия стадий и границы между ними показаны синим цветом. Ко всем сигналам добавлен суффикс (F, D, E, M или W), показывающий, к какой стадии они относятся.

Регистровый файл – особенный в том смысле, что процессор читает из него в стадии Decode, а пишет в стадии Writeback. Поэтому, несмотря на то, что на рисунке он находится в стадии Decode, но адрес и данные для записи приходят из стадии Writeback. Эта обратная связь будет приводить к конфликтам конвейера, которые мы рассмотрим в **разделе 7.5.3**. В конвейерном процессоре значения записываются в регистровый файл по отрицательному фронту тактового сигнала *CLK*, когда значение на его входе *WD3* уже стабильно.

Одна маленькая, но чрезвычайно важная проблема организации конвейерной обработки данных – это то, что все сигналы, относящиеся к конкретной команде, должны обязательно продвигаться по конвейеру одновременно друг с другом, в унисон. На **Рис. 7.45 (b)** есть связанная с этим ошибка. Можете ли вы ее найти?

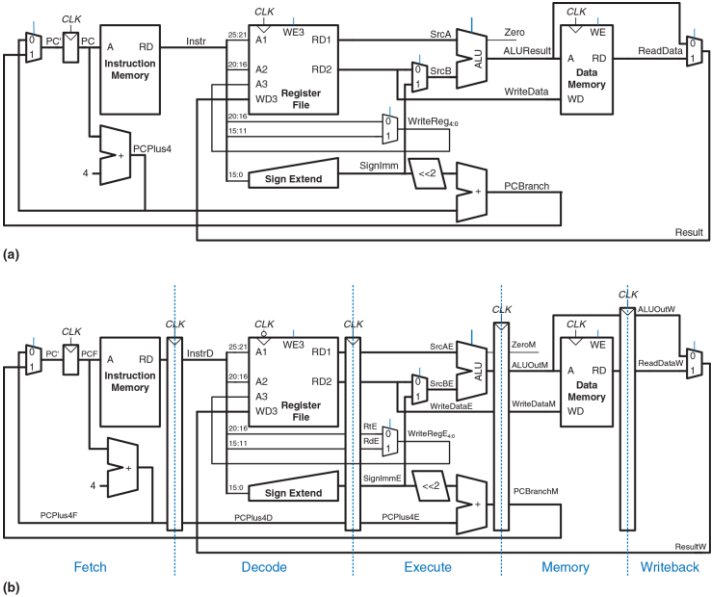


Рис. 7.45 **Однотактный (a) и конвейерный (b) тракты данных**

Ошибка – в логике записи в регистровый файл, которая происходит в стадии Writeback. В регистровый файл записывается значение *ResultW* из стадии Writeback, но в качестве адреса используется сигнал *WriteRegE* из стадии Execute. На диаграмме, приведенной на **Рис. 7.44**, на пятом такте результат команды `lw` был бы ошибочно записан в регистр `$s4`, а не в `$s2`.

На **Рис. 7.46** показан исправленный тракт данных. Сигнал *WriteReg* теперь проходит через два дополнительных регистра в стадиях Memory и Writeback, то есть остается синхронным с остальными сигналами команды. Теперь *WriteRegW* и *ResultW* подаются на входы регистрового файла в стадии Writeback одновременно.

Внимательный читатель мог заметить, что в логике сигнала *PC'* тоже есть проблемы, потому что этот сигнал может понадобиться изменить одновременно в стадиях Fetch и Memory (используя сигналы *PCPlus4F* или *PCBranchM* соответственно). В **разделе 7.5.3** мы покажем, как разрешить этот конфликт.

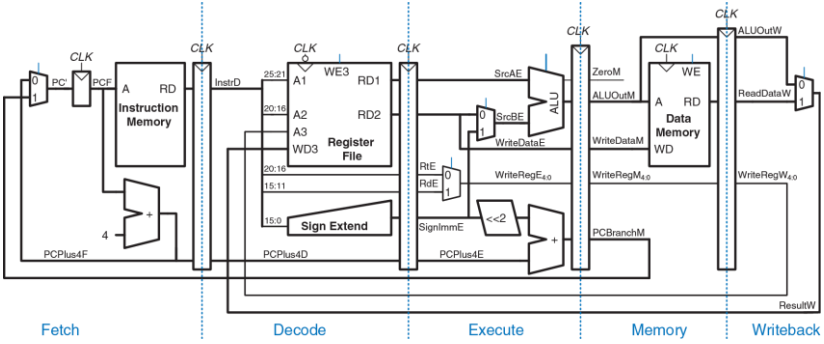


Рис. 7.46 **Исправленный тракт данных**

7.5.2 Конвейерное устройство управления

Конвейерный процессор использует те же управляющие сигналы, что и однотактный процессор, поэтому использует такое же устройство управления. В стадии Decode оно, в зависимости от полей `opcode` и `funct` команды, формирует управляющие сигналы, как было показано в **разделе 7.3.2**. Эти управляющие сигналы должны быть конвейеризированы точно так же, как и тракт данных, чтобы оставаться синхронными с командой, перемещающейся из одной стадии в другую.

Полностью конвейерный процессор с устройством управления показан на **Рис. 7.47**. Аналогично сигналу *WriteReg* на **Рис. 7.46**, сигнал *RegWrite*, пройдя через несколько регистров, обязательно должен дойти до стадии Writeback перед тем, как попасть на вход регистрового файла.

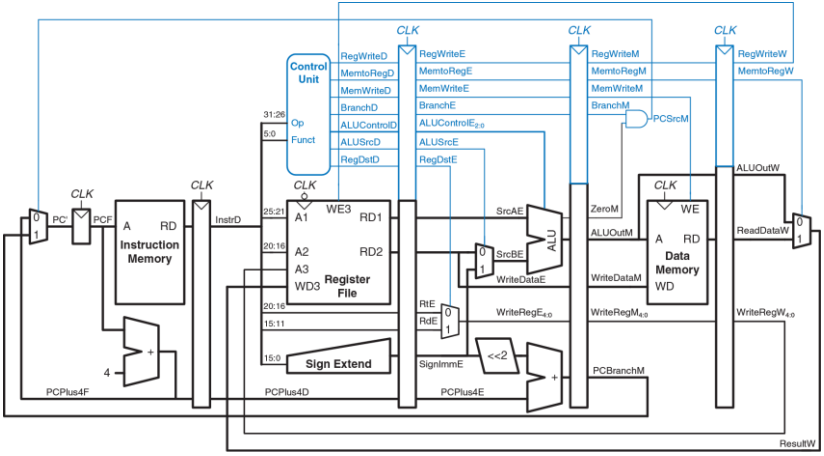


Рис. 7.47 Конвейерный процессор с устройством управления

7.5.3 Конфликты

В конвейерном процессоре выполняются несколько команд одновременно. Когда одна из них зависит от результатов другой, еще не завершенной команды, то говорят, что произошел конфликт (hazard) в конвейере.

Процессор может читать и писать в регистровый файл за один такт. Запись происходит в первой части такта, а чтение – во второй, так что значение в регистр можно записать и затем прочитать обратно за один такт, и это не приведет к конфликту.

На Рис. 7.48 показан конфликт, который возникает, когда одна команда пишет в регистр (`$s0`), а следующая команда читает из него. Это называется конфликтом чтения после записи (*read after write, RAW*). Команда `add` записывает результат в `$s0` в первой части пятого такта, однако команда `and` читает `$s0` на третьем такте, то есть получает неверное значение. Команда `or` читает `$s0` на четвертом такте, и тоже получает неверное значение. Команда `sub` читает `$s0` во второй половине пятого такта, то есть наконец-то получает корректное значение, которое было записано в регистр в первой половине пятого такта. Все последующие команды также прочитают корректное значение из `$s0`. Как видно из диаграммы, конфликт в конвейере возникает тогда, когда команда записывает значение в регистр и хотя

бы одна из следующих двух команд читает его. Если не принять мер, конвейер вычислит неправильный результат.

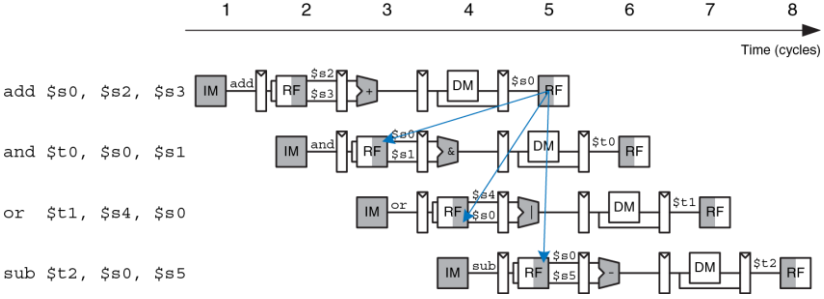


Рис. 7.48 Конфликты в конвейере

Однако при ближайшем рассмотрении оказывается, что результат команды `add` вычисляется в АЛУ на третьем такте, а команде `and` он требуется лишь на четвертом. В принципе, мы могли бы переслать результат выполнения первой команды второй до того, как он будет записан в регистровый файл, разрешив конфликт чтения после записи без необходимости приостанавливать конвейер. Часть тракта данных, обеспечивающая такую пересылку, называется *байпасом* (bypass). В некоторых других случаях, которые мы рассмотрим далее, конвейер

все-таки придется приостанавливать, чтобы дать процессору время вычислить требуемый результат до того, как он понадобится последующим командам. В любом случае, чтобы программы выполнялась корректно, несмотря на конвейеризацию, мы должны что-то предпринять для разрешения конфликтов.

Конфликты можно разделить на конфликты данных (data hazards) и конфликты управления (control hazards). Конфликт данных происходит, когда команда пытается прочитать из регистра значение, которое еще не было записано предыдущей командой. Конфликт управления происходит, когда процессор выбирает из памяти следующую команду до того, как стало ясно, какую именно команду надо выбрать. В оставшейся части этого раздела мы добавим в процессор блок разрешения конфликтов (hazard unit), который будет выявлять и разрешать конфликты таким образом, чтобы процессор выполнял программы корректно.

Разрешение конфликтов пересылкой через байпас

Некоторые конфликты данных можно разрешить путем *пересылки результата через байпас* (bypassing, также используется термин forwarding) из стадий Memory или Writeback в ожидающую этот результат команду, находящуюся в стадии Execute. Чтобы организовать байпас, понадобится добавить мультиплексоры перед АЛУ. Теперь операнд можно получить либо из регистрового файла, либо напрямую

из стадий Memory или Writeback, как показано на **Рис. 7.49**. Таким образом, на четвертом такте `$s0` пересылается через байпас из стадии Memory, где находится команда `add`, в стадию Execute, где находится команда `and`, которой нужен результат выполнения `add`. На пятом такте `$s0` пересылается из стадии Writeback, где теперь находится команда `add`, в стадию Execute, где находится ожидающая ее результата команда `or`.

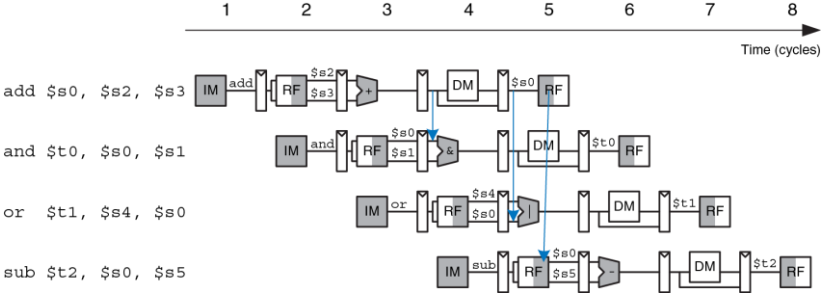


Рис. 7.49 Пересылка данных через байпас

Пересылка данных через байпас необходима, если номер любого из регистров операндов команды, находящейся в стадии Execute, равен

номеру регистра результата команды, находящейся в стадии Memory или Writeback. На **Рис. 7.50** показан модифицированный конвейерный процессор с байпасом.

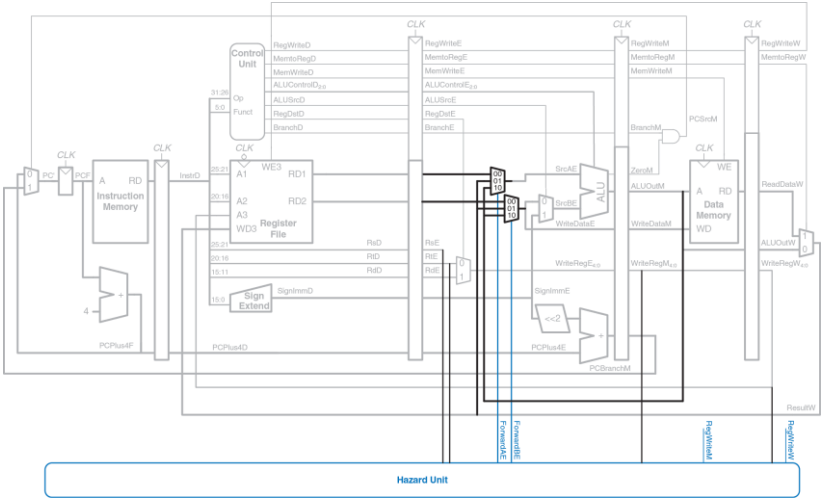


Рис. 7.50 Разрешение конфликтов в конвейере при помощи байпаса

У него есть блок обнаружения конфликтов и два новых мультиплексора. Блок обнаружения конфликтов получает на свой вход номера регистров, хранящих операнды команды, находящейся в стадии Execute, а также номера регистров результатов команд, находящихся в стадиях Memory и Writeback. Также ему необходимы сигналы *RegWrite* из стадий Memory и Writeback. Эти сигналы показывают, нужно ли на самом деле писать результат в регистр или нет (например, команды `sw` и `beq` не записывают свои результаты в регистровый файл, поэтому их результаты пересылать не нужно). Заметьте, что на рисунке сигналы *RegWrite*, подключенные к блоку обнаружения конфликтов, изображены как короткие линии с названиями сигналов, как бы висящие в воздухе. Это сделано для того, чтобы не засорять рисунок длинными линиями, соединяющими управляющие сигналы вверху и блок обнаружения конфликтов внизу. Вместо этого предполагается, что все линии с одинаковыми названиями сигналов соединены между собой.

Блок обнаружения конфликтов управляет мультиплексорами байпаса (forwarding multiplexers), которые определяют, взять ли операнды из регистрового файла или переслать их напрямую из стадии Memory или Writeback. Если в одной из этих стадий происходит запись результата в регистр и номер этого регистра совпадает с номером регистра операнда следующей команды, то используется байпас. Регистр `$0` – исключение, он всегда содержит ноль, поэтому его нельзя пересылать.

Если номера регистров результатов в стадиях Memory и Writeback одинаковы, то приоритет отдается стадии Memory, так как она содержит более новую команду. Ниже приведена функция, определяющая логику пересылки данных в операнд *SrcA*. Логика для операнда *SrcB* (*ForwardBE*) точно такая же, за исключением того, что она проверяет поле `rt`, а не `rs`.

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
                                ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
                                ForwardAE = 01
else                            ForwardAE = 00
```

Разрешение конфликтов данных приостановками конвейера

Пересылка данных через байпас может разрешить конфликт при чтении после записи, только если результат вычисляется в стадии Execute, потому что только в этом случае его можно сразу переслать в стадию Execute следующей команды. К сожалению, команда `lw` не может прочитать данные раньше, чем в конце стадии Memory, поэтому ее результат нельзя переслать в стадию Execute следующей команды.

В этом случае мы будем говорить, что латентность команды `lw` равна двум тактам, потому что зависимая команда не может использовать результат `lw` раньше, чем через два такта. Эта проблема показана на

Рис. 7.51. Команда `lw` получает данные из памяти в конце четвертого такта. Однако команде `and` эти данные требуются в качестве операнда уже в самом начале четвертого такта. Пересылка данных через байпас не поможет разрешить этот конфликт.

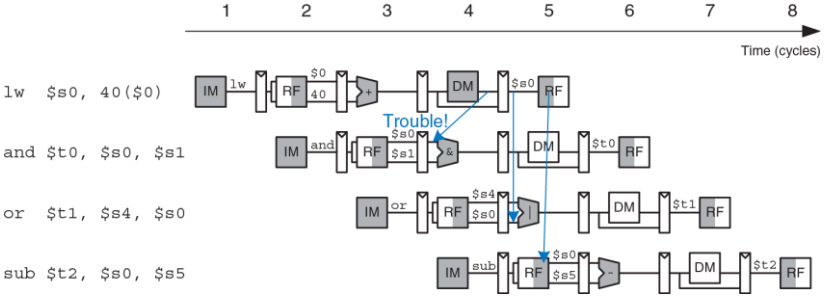


Рис. 7.51 Проблема при пересылке результата команды `lw` через байпас

Альтернативное решение – приостановить конвейер, задержав все операции до тех пор, пока данные не станут доступны. На **Рис. 7.52** показана приостановка зависимой команды (`and`) в стадии Decode. Команда `and` попадает в эту стадию на третьем такте и остается там на четвертом такте. Следующая команда (`or`) должна, соответственно,

оставаться в стадии Fetch в течение третьего и четвертого тактов, так как стадия Decode занята.

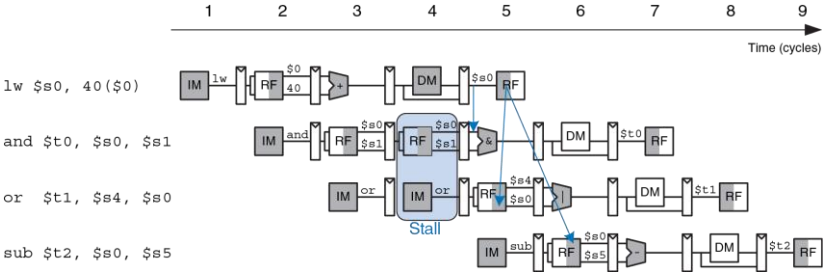


Рис. 7.52 Разрешение конфликта приостановкой конвейера

На пятом такте результат команды `lw` можно через байпас переслать из стадии Writeback в стадию Execute, где будет находиться команда `and`. На этом же такте операнд `$s0` команды or может быть прочитан прямо из регистрового файла, без какой-либо пересылки данных.

Заметьте, что теперь стадия Execute на четвертом такте не используется. Аналогично, стадия Memory не используется на пятом такте, а Writeback – на шестом. Эта неиспользуемая стадия, проходящая по конвейеру, называется пузырьком (*bubble*), и ведет себя

так же, как команда `nop`. Пузырек получается путем обнуления всех управляющих сигналов стадии Execute на время приостановки стадии Decode, так что он не приводит ни к каким изменениям архитектурного состояния.

Подводя итог, стадию конвейера можно приостановить, если запретить обновление регистра, находящегося между этой и предыдущей стадиями. Как только какая-либо стадия приостановлена, все предыдущие стадии тоже должны быть приостановлены, чтобы ни одна из команд не пропала. Регистр, находящийся сразу после приостановленной стадии, должен быть очищен, чтобы «мусор» не попал в конвейер. Приостановки конвейера ухудшают производительность, поэтому должны использоваться только при необходимости.

На **Рис. 7.53** показан модифицированный процессор, который умеет приостанавливать конвейер для разрешения конфликтов данных, возникающих при выполнении команды `lw`. Блок управления конфликтами смотрит, какая команда находится в стадии Execute. Если это lw, а номер регистра результата (`rtE`) совпадает с номером любого из регистров операндов команды, находящейся в стадии Decode (`rsD` или `rtD`), то стадия Decode должна быть приостановлена до тех пор, пока операнд не будет прочитан из памяти.

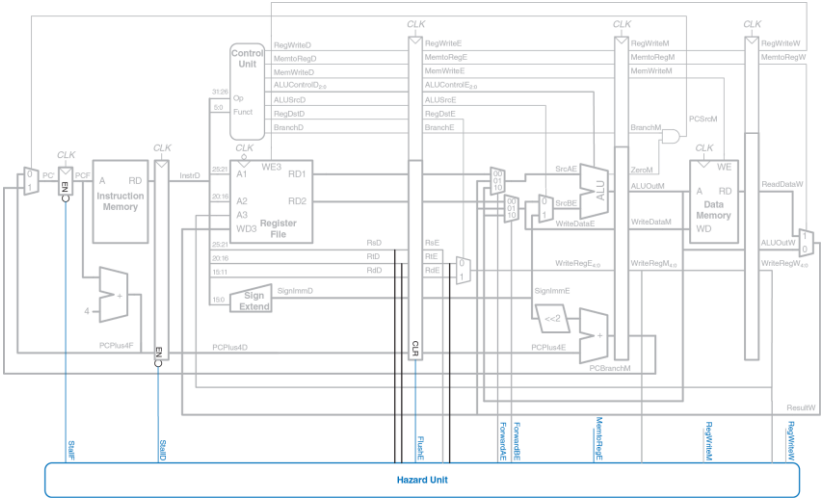


Рис. 7.53 Разрешение конфликтов в конвейере при помощи приостановок

Для приостановки стадий Fetch и Decode нужно добавить вход разрешения работы (*EN*) временным регистрам, расположенным перед этими стадиями, а также вход синхронного сброса (*CLR*) временному

регистру, расположенному перед стадией Execute. Когда возникает необходимость приостановить конвейер из-за команды lw, устанавливаются сигналы *StallD* и *StallF*, запрещающие временным регистрам перед стадиями Decode и Fetch изменять их старое значение. Также устанавливается сигнал *FlushE*, очищающий содержимое временного регистра перед стадией Execute, что приводит к появлению пузырька.[11]

Для команды lw всегда устанавливается сигнал *MemtoReg.* Таким образом, логика формирования сигналов приостановки (stall) и очистки (flush) выглядит так:

```
lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE
StallF = StallD = FlushE = lwstall
```

Разрешение конфликтов управления

Выполнение команды `beq` приводит к конфликту управления: конвейерный процессор не знает, какую команду выбрать следующей, поскольку в этот момент еще не ясно, нужно ли будет выполнить условный переход или нет.

[11] Строго говоря, очищать требуется только номера регистров (RsE, RtE и RdE), а так же управляющие сигналы, которые могут привести к изменению содержимого памяти или архитектурного состояния (RegWrite, MemWrite и Branch). Все остальные сигналы и данные могут быть равны чему угодно, они ни на что не повлияют.

Один из способов разрешить этот конфликт – приостановить конвейер до тех пор, пока не будет принято нужное решение (т.е. до тех пор, пока не будет вычислен сигнал *PCSrc*). Решение принимается в стадии Memory, так что для каждой команды условного перехода придется приостанавливать конвейер на три такта. Такое решение повлекло бы весьма плачевные последствия для производительности системы.

Есть и альтернативный способ – предсказать, будет ли выполнен условный переход или нет, и начать выполнять команды, основываясь на этом. Как только условие перехода будет вычислено, процессор может прервать эти команды, если предсказание было неверным. Предположим, что мы предсказали, что условный переход не будет выполнен, и продолжили выполнять команды в порядке следования. Если окажется, что переход должен был быть выполнен, то конвейер должен быть очищен (flushed) от трех команд, идущих сразу за командой перехода, путем очистки соответствующих временных регистров конвейера. Зря потраченные в этом случае такты называются простоем из-за неправильно предсказанного перехода (*branch misprediction penalty*).

На **Рис. 7.54** показано, что происходит в конвейере, если выполнен условный переход из адреса 20 по адресу 64. Условие перехода вычисляется на четвертом такте; к этому моменту процессор уже выбрал команды `and`, `or` и `sub` из адресов 24, 28, и 2C соответственно.

Конвейер должен быть очищен от этих команд, после чего на пятом такте будет выбрана команда `slt` по адресу 64. Мы добились небольшого улучшения, однако очистка трех команд в случае выполненного перехода все еще существенно ухудшает производительность.

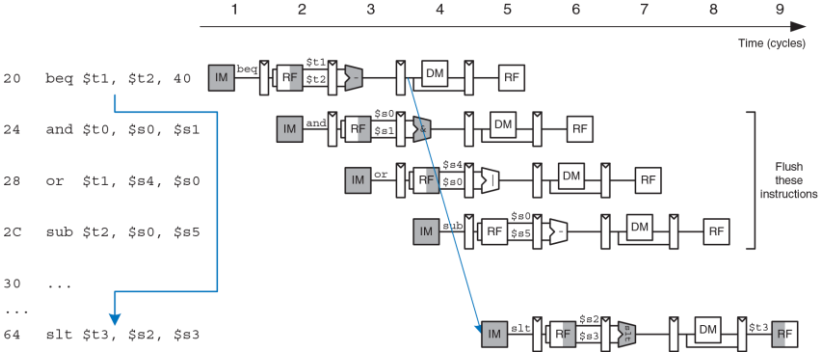


Рис. 7.54 Очистка конвейера при выполненном условном переходе

Мы могли бы уменьшить простой из-за неправильно предсказанного перехода, если бы могли вычислить условие перехода пораньше. Так как вычисление условия заключается в определении равенства двух

регистров, то мы могли бы использовать отдельный компаратор – это гораздо быстрее, чем вычитать два числа и проверять результат на равенство нулю. Если компаратор достаточно быстр, то можно перенести его в стадию Decode, чтобы прочитанные из регистрового файла операнды тут же сравнивались, а результат сравнения использовался для вычисления нового значения счетчика команд к концу стадии Decode.

На **Рис. 7.55** показано функционирование конвейера с ранним вычислением условий перехода, происходящим на втором такте. На третьем такте процессор очищает конвейер от команды `and` и выбирает из памяти команду `slt`, то есть простой из-за неправильно предсказанного перехода уменьшился с трех тактов до одного.

На **Рис. 7.56** показан конвейерный процессор с ранним вычислением условий переходов, способный разрешать конфликты управления. В стадию Decode добавлен компаратор, туда же перенесен и логический элемент И, используемый для получения сигнала *PCSrc*, так что теперь *PCSrc* формируется в стадии Decode, а не Memory, как раньше. Чтобы адрес перехода *PCBranch* был вычислен вовремя, соответствующий сумматор тоже должен быть перенесен в стадию Decode. Для того чтобы очистить стадию Decode от ошибочно выбранной команды в случае неправильно предсказанного перехода, нужно добавить вход синхронного сброса для временного регистра,

находящегося между стадиями Fetch и Decode, и подключить его к сигналу *PCSrcD*.

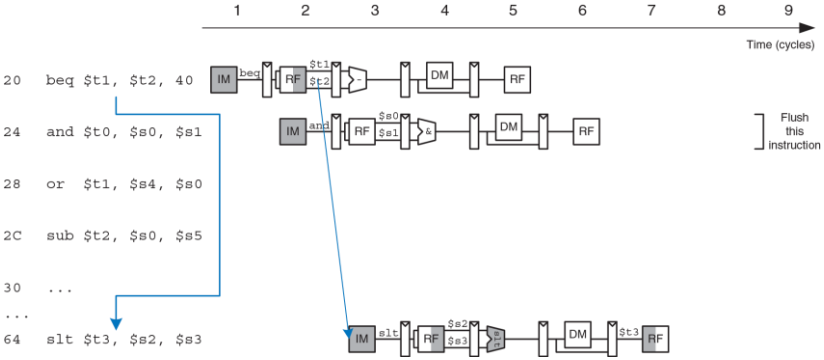


Рис. 7.55 Раннее вычисление условия перехода

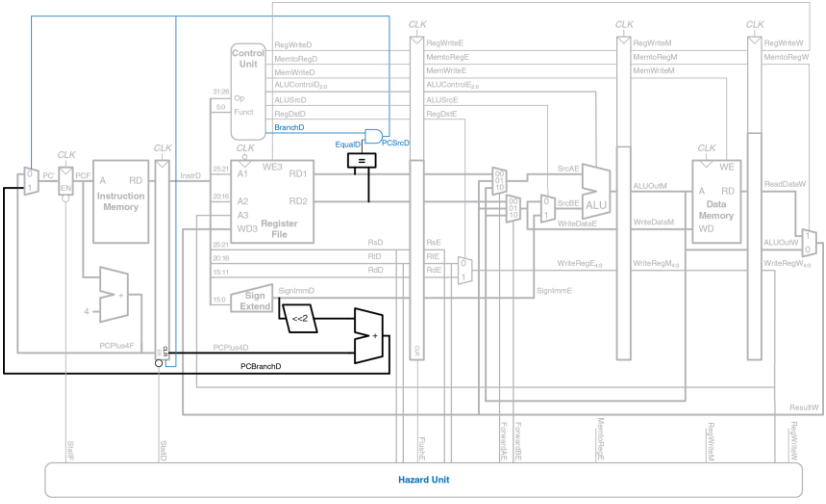


Рис. 7.56 Разрешение конфликтов управления в конвейере

К сожалению, раннее вычисление условия переходов приводит к новому конфликту данных при чтении после записи. Если один из операндов, используемых для вычисления условия перехода,

изменяется предыдущей командой и еще не записан обратно в регистровый файл, то при выполнении команды перехода из регистрового файла будет прочитано некорректное значение. Как и раньше, мы можем разрешить этот конфликт при помощи байпаса или приостановить конвейер, если данные не готовы.

На **Рис. 7.57** показаны изменения, которые необходимо внести, чтобы разрешить эту зависимость в стадии Decode. Если результат предыдущей команды находится в стадии Writeback, он будет записан в регистровый файл в первой части такта и прочитан во второй, так что конфликта не будет. Если предыдущая команда была арифметической и ее результат находится в стадии Memory, то можно переслать его при помощи байпаса в компаратор через два новых мультиплексора. Если результат арифметической команды находится в стадии Execute или результат команды `lw` – в стадии Memory, то стадия Decode должна быть приостановлена до тех пор, пока не будет готов результат.

Сигналы, управляющие логикой пересылки данных в стадию Decode, выглядят так:

```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

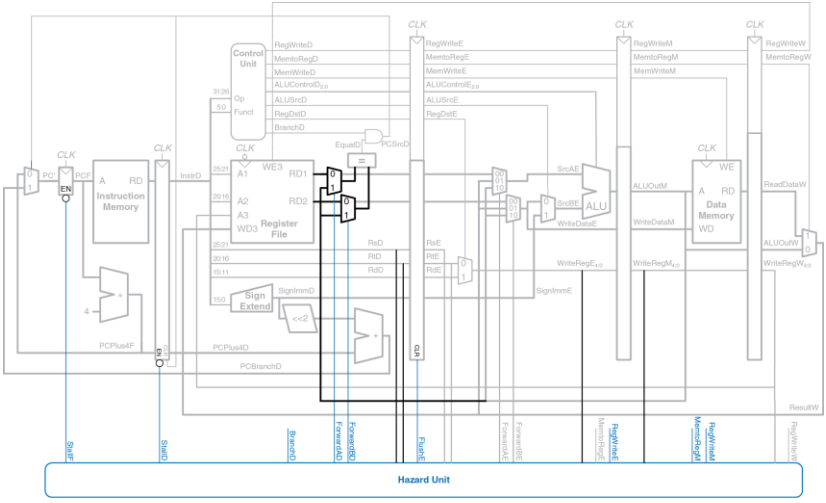


Рис. 7.57 Устранение RAW-конфликта при раннем вычислении условия перехода

Сигнал, управляющий приостановкой конвейера при выполнении команды условного перехода, показан ниже. Процессор должен вычислить условие перехода в стадии Decode. Если любой из

операндов этой команды зависит от арифметической команды, находящейся в стадии Execute, или от команды `lw`, находящейся в стадии Memory, то процессор приостанавливается до тех пор, пока операнды не будут готовы.

```
branchstall =
  BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
                            OR
BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

Теперь процессор будет приостановлен как в случае конфликта данных при чтении из памяти, так и в случае конфликта управления:

```
StallF = StallD = FlushE = lwstall OR branchstall
```

Подводя итоги

Конфликты при чтении данных после записи (RAW data hazards) случаются, когда одна из команд зависит от результата предыдущей команды, еще не записанного в регистровый файл. Такие конфликты можно разрешить при помощи байпаса, если результат вычислен достаточно рано; в противном случае потребуется приостановка конвейера до тех пор, пока он не будет готов. Конфликты управления возникают, когда нужно выбрать из памяти команду, а решение о том, какую именно, еще не принято. Эти конфликты разрешаются путем предсказания того, какая именно команда должна быть выбрана, и очисткой конвейера от ошибочно выбранных команд в случае, если

предсказание не сбылось. Количество команд, от которых придется очистить конвейер в случае неправильно предсказанного перехода, минимизируется путем перемещения логики вычисления условия переходов в начало конвейера. Как вы могли заметить, при разработке конвейерных процессоров нужно понимать, как различные команды могут взаимодействовать между собой, а так же заранее обнаружить все возможные конфликты. На **Рис. 7.58** показан конвейерный процессор, способный разрешить все имеющиеся конфликты.

7.5.4 Дополнительные команды

Добавление новых команд в конвейерный процессор весьма похоже на их добавление в однотактный процессор. Однако новые команды могут приводить к новым конфликтам, которые надо определять и разрешать.

В частности, для поддержки команд `addi` и `j` в конвейерном процессоре требуется модифицировать устройство управления в точности так же, как было описано в **разделе 7.3.3**, а также добавить в тракт данных мультиплексор для безусловных переходов вдобавок к уже имеющемуся мультиплексору для условных переходов. Как и в случае условных переходов, безусловный переход происходит в стадии Decode, так что стадия Fetch должна быть очищена от следующей команды. Разработку логики очистки оставим как **упражнение 7.35**.

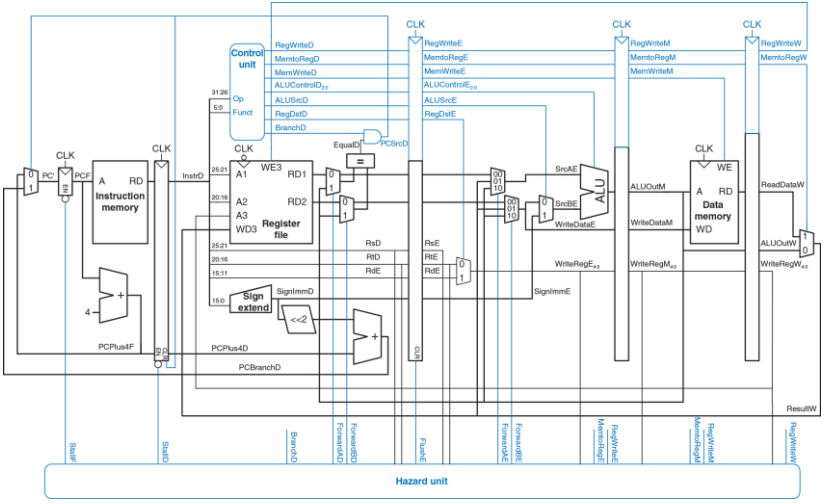


Рис. 7.58 **Конвейерный процессор, разрешающий все конфликты**

7.5.5 Оценка производительности

В идеальном случае у конвейерного процессора количество тактов на команду должно быть равно единице (CPI = 1), потому что на каждом такте процессор начинает выполнять новую команду. Однако приостановки и очистки конвейера приводят к тому, что некоторые такты пропадают, так что в реальной жизни CPI немного больше, при этом оно зависит от выполняемой программы.

Пример 7.9 CPI КОНВЕЙЕРНОГО ПРОЦЕССОРА

Тестовый набор SPECINT2000, рассмотренный в **примере 7.7**, содержит примерно 25% команд загрузки, 10% команд сохранения, 11% команд условного перехода, 2% команд безусловного перехода и 52% команд типа R. Предположим, что в 40% случаев результат команды загрузки требуется непосредственно следующей за ней команде, что приводит к приостановке конвейера. Также предположим, что четверть всех условных переходов предсказываются неверно, что приводит к очистке конвейера. Будем также считать, что при выполнении команды безусловного перехода конвейер всегда очищается от следующей непосредственно за ней команды (прим. переводчика: не забывайте, что к тому моменту, когда станет ясно, что выбрана команда безусловного перехода, конвейер по инерции уже успеет выбрать из памяти следующую команду, которую, очевидно, выполнять не следует). Прочими конфликтами пренебрежем. Требуется вычислить среднее число тактов на команду (cycles per instruction, CPI) в конвейерном процессоре.

Решение: Среднее CPI можно вычислить, перемножив число тактов, требуемое для выполнения команды каждого типа, на относительное количество соответствующих команд и просуммировав полученные значения (прим. переводчика: в данном случае под временем выполнения команды имеется в виду число тактов, через которое может быть запущена на выполнение следующая команда. Если команда выполняется за один такт, то следующая за ней команда может быть запущена уже на следующем такте). Команды загрузки данных выполняются за один такт, если нет конфликтов, и за два такта, если конвейер должен быть приостановлен для разрешения конфликта, так что их CPI = (0,6)(1) + (0,4)(2) = 1,4. Команды условного перехода выполняются за один такт, если переход предсказан корректно, и за два такта в противно случае, так что их CPI = (0,75)(1) + (0,25)(2) = 1,25. У команд безусловного перехода CPI всегда равен двум. CPI всех остальных команд равен единице. Таким образом, для этого тестового набора среднее CPI = (0,25)(1,4) + (0,1)(1) + (0,11)(1,25) + (0,02)(2) + (0,52)(1) = 1,15.

Минимальную длительность такта мы можем определить, оценив наиболее длинную цепь в каждой из пяти стадий, показанных на **Рис. 7.58**. Если принять во внимание, что значение в регистровый файл записывается в стадии Writeback в первой части такта, а читается в стадии Decode во второй части такта, то получится, что длительность такта в стадиях Decode и Writeback равна удвоенному времени чтения или записи соответственно, включая все накладные расходы на мультиплексирование данных и тому подобное.

$$T_c = max \left(\begin{array}{cc} t_{pcq}+t_{mem}+t_{\text{setup}} & \text{Fetch} \\ 2(t_{RF\text{read}}+t_{mux}+t_{\text{eq}}+t_{\text{AND}}+T_{\text{mux}}+t_{\text{setup}}) & \text{Decode} \\ t_{pcq}+t_{\text{mux}}+t_{\text{mux}}+t_{\text{ALU}}+t_{\text{setu}}p & \text{Execute} \\ t_{pcq}+t_{\text{memwrite}}+t_{\text{setup}} & \text{Memory} \\ (t_{pcq}+t_{\text{mux}}+t_{RF\text{write}}) & \text{Writeback} \end{array} \right) \quad (7.5)$$

Пример 7.10 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

Бен Битдидл хочет сравнить производительность конвейерного процессора с производительностью однотактного и многотактного процессоров, рассмотренной в **примере 7.8**. Большая часть задержек элементов уже была приведена в **Табл. 7.6**. Задержка компаратора составляет 40 пс, логического элемента «И» – 15 пс, время записи в регистровый файл составляет 100 пс, а записи в память – 220 пс. Помогите Бену сравнить время выполнения 100 миллиардов команд из тестового набора SPECINT2000 на каждом из этих процессоров.

Решение: согласно **формуле (7.5)**, длительность такта конвейерного процессора равна

T_{c3} = max[30 + 250 + 20, 2(150 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 + 20, 30 + 220 + 20, 2(30 + 25 + 100)] = 550 пс. Согласно **формуле (7.1)**, общее время выполнения равно

T_3 = (100 × 10^9 команд)(1,15 тактов/команду)(550 × 10^{-12} с/такт) = 63,3 с. Время выполнения для однотактного процессора было равно 92,5 с, а для многотактного – 133,9 с.

Таким образом, конвейерный процессор значительно быстрее прочих. Однако о пятикратном преимуществе над однотактным процессором, которое мы надеялись получить из-за пятистадийного конвейера, нет и речи. Конфликты в конвейере вносят свою лепту в увеличение CPI; другая причина в том, что для организации конвейера нам потребовалось добавить временные (неархитектурные) регистры между стадиями, поэтому накладные расходы из-за использования регистров, равные сумме времени предустановки и времени задержки (clk-to-Q), теперь добавляются к каждой стадии, а не к общему времени выполнения команды. То, что больший процент длительности такта уходит на накладные расходы, ограничивает преимущества конвейерной обработки.

Внимательный читатель мог заметить, что длительность стадии Decode значительно больше, чем остальных стадий, так как и чтение из регистрового файла, и сравнение двух регистров для вычисления условия перехода должны выполняться за половину такта. Похоже, что идея перенести компаратор в стадию Decode была не самой лучшей. Если бы вычисление условия перехода происходило в стадии Execute, то CPI, конечно, немного увеличился бы, так как неправильное предсказание перехода приводило бы к необходимости очистить конвейер от двух неправильно выбранных команд вместо одной. Тем не менее, длительность такта стала бы существенно меньше, так что в результате производительность процессора все равно бы выросла.

Основные компоненты конвейерного процессора те же, что и у однотактного, однако ему вдобавок требуется значительное количество временных регистров между стадиями, мультиплексоров и логики разрешения конфликтов.

7.6 ПИШЕМ ПРОЦЕССОР НА HDL*

В этом разделе приведен HDL-код однотактного процессора MIPS, который поддерживает все рассмотренные в этой главе команды, включая `addi` и `j`. Этот код послужит примером хорошего стиля описания систем умеренной сложности. Написание HDL-кода для многотактного и конвейерного процессоров мы оставим читателю в качестве **упражнений 7.25** и **7.40**.

В этом разделе мы будем считать, что и память команд, и память данных находятся снаружи процессора и подсоединены к нему с помощью шин адреса и данных. Это более реалистичный сценарий, потому что большинство процессоров используют внешнюю память. Этот подход также демонстрирует то, как процессор может взаимодействовать с внешним миром.

Процессор состоит из тракта данных и устройства управления. Устройство управления, в свою очередь, состоит из основного

дешифратора и дешифратора АЛУ. На **Рис. 7.59** показана диаграмма однотактного процессора, подсоединенного к внешней памяти.

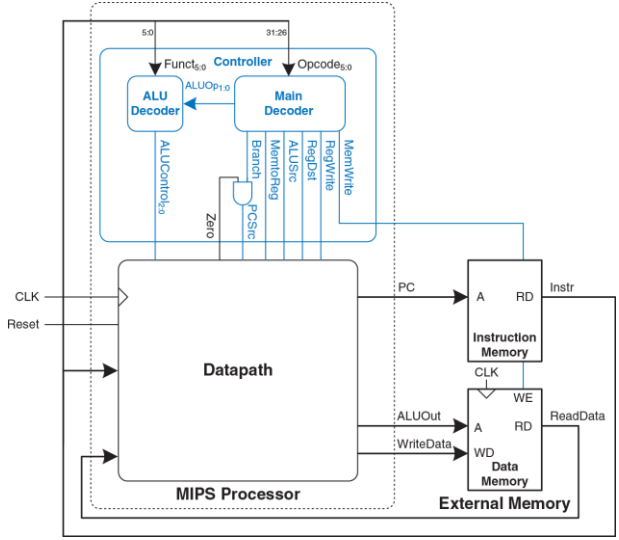


Рис. 7.59 **Подключение однотактового процессора ко внешней памяти**

HDL-код поделен на несколько частей, каждая из которых описана в отдельном разделе. **Раздел 7.6.1** содержит описание тракта данных и устройства управления. **Раздел 7.6.2** содержит универсальные строительные блоки, такие как регистры и мультиплексоры, которые используются для любой микроархитектуры. **Раздел 7.6.3** демонстрирует код тестового окружения и внешней памяти. HDL-код доступен в виде файлов для скачивания на web-сайте книги (см. **Предисловие**).

7.6.1 Однотактный процессор

Основные модули однотактного процессора MIPS приведены ниже.

Пример HDL-кода 7.1 ОДНОТАКТНЫЙ ПРОЦЕССОР MIPS

SystemVerilog

```
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata);

  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero;
  logic [2:0] alucontrol;
```

```
  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mips is -- single cycle MIPS processor
  port(clk, reset:        in  STD_LOGIC;
       pc:                out STD_LOGIC_VECTOR(31 downto 0);
       instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       memwrite:          out STD_LOGIC;
       aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
         zero:               in  STD_LOGIC;
         memtoreg, memwrite: out STD_LOGIC;
         pcsrc, alusrc:      out STD_LOGIC;
```

```
        regdst, regwrite:   out STD_LOGIC;
        jump:               out STD_LOGIC;
        alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset:        in     STD_LOGIC;
        memtoreg, pcsrc:    in     STD_LOGIC;
        alusrc, regdst:     in     STD_LOGIC;
        regwrite, jump:     in     STD_LOGIC;
        alucontrol:         in     STD_LOGIC_VECTOR(2 downto 0);
        zero:               out    STD_LOGIC;
        pc:                 buffer STD_LOGIC_VECTOR(31 downto 0);
        instr:              in     STD_LOGIC_VECTOR(31 downto 0);
        aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
        readdata:           in     STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal memtoreg, alusrc, regdst, regwrite, jump, pcsrc: STD_LOGIC;
  signal zero: STD_LOGIC;
  signal alucontrol: STD_LOGIC_VECTOR(2 downto 0);
begin
  cont: controller port map(instr(31 downto 26),
                            instr(5 downto 0), zero, memtoreg,
                            memwrite, pcsrc, alusrc, regdst,
                            regwrite, jump, alucontrol);
  dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc,
                        regdst, regwrite, jump, alucontrol,
                        zero, pc, instr, aluout, writedata,
                        readdata);
  end;
```

Пример HDL-кода 7.2 УСТРОЙСТВО УПРАВЛЕНИЯ

SystemVerilog

```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity controller is -- single cycle control decoder
  port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
       zero:               in  STD_LOGIC;
       memtoreg, memwrite: out STD_LOGIC;
```

```
      pcsrc, alusrc:      out STD_LOGIC;
      regdst, regwrite:   out STD_LOGIC;
      jump:               out STD_LOGIC;
      alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(op:                in  STD_LOGIC_VECTOR(5 downto 0);
         memtoreg, memwrite: out STD_LOGIC;
         branch, alusrc:     out STD_LOGIC;
         regdst, regwrite:   out STD_LOGIC;
         jump:               out STD_LOGIC;
         aluop:              out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component aludec
    port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
         aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
         alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  signal aluop:  STD_LOGIC_VECTOR(1 downto 0);
  signal branch: STD_LOGIC;
begin
md:maindec port map(op, memtoreg, memwrite, branch,
                    alusrc, regdst, regwrite, jump, aluop);
  ad: aludec port map(funct, aluop, alucontrol);
  pcsrc <= branch and zero;
end;
```

Пример HDL-кода 7.3 ОСНОВНОЙ ДЕШИФРАТОР

SystemVerilog

```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);
  logic [8:0] controls;
  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;
  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity maindec is -- main control decoder
  port(op:                in  STD_LOGIC_VECTOR(5 downto 0);
       memtoreg, memwrite: out STD_LOGIC;
       branch, alusrc:     out STD_LOGIC;
       regdst, regwrite:   out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(8 downto 0);
begin
  process(all) begin
    case op is
      when "000000" =>controls <= "110000010"; -- RTYPE
      when "100011" =>controls <= "101001000"; -- LW
      when "101011" =>controls <= "001010000"; -- SW
      when "000100" =>controls <= "000100001"; -- BEQ
      when "001000" =>controls <= "101000000"; -- ADDI
      when "000010" =>controls <= "000000100"; -- J
      when others   =>controls <= "---------"; -- illegal op
    end case;
  end process;
  (regwrite, regdst, alusrc, branch, memwrite,
   memtoreg, jump, aluop(1 downto 0)) <= controls;
end;
```

Пример HDL-кода 7.4 ДЕШИФРАТОР АЛУ

SystemVerilog

```
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);
  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110; // sub (for beq)
      default: case(funct)         // R-type instructions
          6'b100000: alucontrol <= 3'b010; // add
          6'b100010: alucontrol <= 3'b110; // sub
          6'b100100: alucontrol <= 3'b000; // and
          6'b100101: alucontrol <= 3'b001; // or
          6'b101010: alucontrol <= 3'b111; // slt
          default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity aludec is -- ALU control decoder
  port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
       aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;
architecture behave of aludec is
begin
  process(all) begin
    case aluop is
      when "00" =>alucontrol <= "010"; -- add (for lw/sw/addi)
      when "01" =>alucontrol <= "110"; -- sub (for beq)
      when others =>case funct is      -- R-type instructions
        when "100000" =>alucontrol <= "010"; -- add
        when "100010" =>alucontrol <= "110"; -- sub
        when "100100" =>alucontrol <= "000"; -- and
        when "100101" =>alucontrol <= "001"; -- or
        when "101010" =>alucontrol <= "111"; -- slt
        when others   =>alucontrol <= "---"; -- ???
      end case;
    end case;
  end process;
end;
```

Пример HDL-кода 7.5 ТРАКТ ДАННЫХ

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [2:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);
  logic [4:0] writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;
  // next PC   logic
  Flopr #(32)   pcreg(clk, reset, pcnext, pc);
  adder         pcadd1(pc, 32'b100, pcplus4);
  sl2           immsh(signimm, signimmsh);
  adder         pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)    pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)    pcmux(pcnextbr, {pcplus4[31:28],
                      instr[25:0], 2'b00}, jump, pcnext);
  // register file logic
  Regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata);
```

```
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0], signimm);
  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb, alucontrol, aluout, zero);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
entity datapath is -- MIPS datapath
  port(clk, reset:        in     STD_LOGIC;
       memtoreg, pcsrc:   in     STD_LOGIC;
       alusrc, regdst:    in     STD_LOGIC;
       regwrite, jump:    in     STD_LOGIC;
       alucontrol:        in     STD_LOGIC_VECTOR(2 downto 0);
       zero:              out    STD_LOGIC;
       pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
       instr:             in     STD_LOGIC_VECTOR(31 downto 0);
       aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in     STD_LOGIC_VECTOR(31 downto 0));
end;
architecture struct of datapath is

  component alu
    port(a, b:       in     STD_LOGIC_VECTOR(31 downto 0);
         alucontrol: in     STD_LOGIC_VECTOR(2 downto 0);
```

```
        result:     buffer STD_LOGIC_VECTOR(31 downto 0);
        zero:       out    STD_LOGIC);
  end component;
  component regfile
    port(clk:           in  STD_LOGIC;
         we3:           in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
           s:    in  STD_LOGIC;
```

```
          y:    out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal  writereg:          STD_LOGIC_VECTOR(4 downto 0);
  signal  pcjump, pcnext,
          pcnextbr, pcplus4,
          pcbranch:          STD_LOGIC_VECTOR(31 downto 0);
  signal signimm, signimmsh:STD_LOGIC_VECTOR(31 downto 0);
  signal srca, srcb, result:STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch, pcsrc,
  pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump, pcnext);
  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                       instr(20 downto 16), writereg, result, srca,
                       writedata);
  wrmux: mux2 generic map(5) port map(instr(20 downto 16),
                                      instr(15 downto 11),
                                      regdst, writereg);
resmux: mux2 generic map(32) port map(aluout, readdata,
                                        memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);
  -- ALU logic
srcbmux: mux2 generic map(32) port map(writedata, signimm,
```

```
                                        alusrc, srcb);
  mainalu: alu port map(srca, srcb, alucontrol, aluout, zero);
end;
```

7.6.2 Универсальные строительные блоки

Этот раздел содержит универсальные строительные блоки, которые могут быть полезны для реализации любой микроархитектуры MIPS. Эти блоки включают регистровый файл, сумматор, блок сдвига влево, блок расширения знака, триггер с сигналом сброса и мультиплексор. Написание HDL-кода для АЛУ оставлено читателю в качестве **упражнения 5.9**.

Пример HDL-кода 7.6 РЕГИСТРОВЫЙ ФАЙЛ

SystemVerilog

```
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);
  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clk
```

```
  // register 0 hardwired to 0
  // note: for pipelined processor, write third port
  // on falling edge of clk

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
       wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end;

  architecture behave of regfile is
    type ramtype is array (31 downto 0) of STD_LOGIC_VECTOR(31
                           downto 0);

    signal mem: ramtype;
  begin
```

```
    -- three-ported register file
    -- read two ports combinationally
    -- write third port on rising edge of clk
    -- register 0 hardwired to 0
    -- note: for pipelined processor, write third port
    -- on falling edge of clk
  process(clk) begin
    if rising_edge(clk) then
       if we3 = '1' then mem(to_integer(wa3)) <= wd3;
       end if;
    end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 0) then rd1 <= X"00000000";
        -- register 0 holds 0
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

Пример HDL-кода 7.7 СУММАТОР

SystemVerilog

```
module adder(input  logic [31:0] a, b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

Пример HDL-кода 7.8 БЛОК СДВИГА ВЛЕВО (УМНОЖЕНИЕ НА 4)

SystemVerilog

```
module sl2(input  logic [31:0] a,
           output logic [31:0] y);
  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sl2 is -- shift left by 2
  port(a: in  STD_LOGIC_VECTOR(31 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of sl2 is
begin
  y <= a(29 downto 0) & "00";
end;
```

Пример HDL-кода 7.9 БЛОК РАСШИРЕНИЯ ЗНАКА

SystemVerilog

```
module signext(input  logic [15:0] a,
               output logic [31:0] y);
  assign y = {{16{a[15]}}, a};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity signext is -- sign extender
  port(a: in  STD_LOGIC_VECTOR(15 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of signext is
begin
  y <= X"ffff" & a when a(15) else X"0000" & a;
end;
```

Пример HDL-кода 7.10 ТРИГГЕР С СИНХРОННЫМ СБРОСОМ

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               Input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopr is -- flip-flop with synchronous reset
  generic (width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

Пример HDL-кода 7.11 МУЛЬТИПЛЕКСОР 2:1

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is -- two-input multiplexer
  generic(width: integer := 8);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;
```

7.6.3 Тестовое окружение

Тестовое окружение загружает программу в память команд. Программа, представленная на **Рис. 7.60**, использует все команды процессора, выполняя такие вычисления, которые приводят к правильному результату только тогда, когда все команды работают правильно. В случае успешного выполнения программа запишет значение 7 по адресу 84; маловероятно, что это произойдет в случае ошибок в аппаратуре. Это – пример целенаправленного тестирования (ad hoc testing).

HDL-код тестового окружения, модуля верхнего уровня процессора MIPS (top-level module), а также блоков внешней памяти приведен ниже. Каждый блок памяти содержит 64 слова.

```
# mipstest.asm
# David_Harris@hmc.edu , Sarah_Harris@hmc.edu 31 March 2012
#
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#          Assembly              Description              Address  Machine
main:   addi $2, $0, 5       # initialize $2 = 5          0        20020005
        addi $3, $0, 12      # initialize $3 = 12         4        2003000c
        addi $7, $3, -9      # initialize $7 = 3          8        2067fff7
        or   $4, $7, $2      # $4 = (3 OR 5) = 7          c        00e22025
        and  $5, $3, $4      # $5 = (12 AND 7) = 4        10       00642824
        add  $5, $5, $4      # $5 = 4 + 7 = 11            14       00a42820
        beq  $5, $7, end     # shouldn't be taken         18       10a7000a
        slt  $4, $3, $4      # $4 = 12 < 7 = 0            1c       0064202a
        beq  $4, $0, around  # should be taken            20       10800001
        addi $5, $0, 0       # shouldn't happen           24       20050000
around: slt  $4, $7, $2      # $4 = 3 < 5 = 1             28       00e2202a
        add  $7, $4, $5      # $7 = 1 + 11 = 12           2c       00853820
        sub  $7, $7, $2      # $7 = 12 - 5 = 7            30       00e23822
        sw   $7, 68($3)      # [80] = 7                   34       ac670044
        lw   $2, 80($0)      # $2 = [80] = 7              38       8c020050
        j    end             # should be taken            3c       08000011
        addi $2, $0, 1       # shouldn't happen           40       20020001
end:    sw   $2, 84($0)      # write mem[84] = 7          44       ac020054
```

Рис. 7.60 Ассемблерный и машинный код тестовой программы

Рис. 7.61 `memfile.dat`

```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

Пример HDL-кода 7.12 ТЕСТОВОЕ ОКРУЖЕНИЕ ПРОЦЕССОРА MIPS

SystemVerilog

```
module testbench();

  logic  clk;
  logic  reset;
  logic [31:0] writedata, dataadr;
  logic        memwrite;
  // instantiate device to be tested
  top dut (clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end

  // check results
  always @(negedge clk)
   begin
    if (memwrite) begin
       if (dataadr = = = 84 & writedata = = = 7) begin
```

```
          $display("Simulation succeeded");
          $stop;
       end else if (dataadr != = 80) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is
end;
architecture test of testbench is
  component top
    port(clk, reset:          in  STD_LOGIC;
         writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
         memwrite:           out STD_LOGIC);
  end component;
  signal writedata, dataadr:   STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, memwrite: STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: top port map(clk, reset, writedata, dataadr, memwrite);
```

```
  -- Generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 84 at end of program
  process(clk) begin
    if (clk'event and clk = '0' and memwrite = '1') then
      if (to_integer(dataadr) = 84 and to_integer
          (writedata) = 7) then
        report "NO ERRORS: Simulation succeeded" severity failure;
      elsif (dataadr /= 80) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

Пример HDL-кода 7.13 МОДУЛЬ ВЕРХНЕГО УРОВНЯ ПРОЦЕССОРА MIPS

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic        memwrite);

  logic [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is -- top-level design for testing
  port(clk, reset:         in     STD_LOGIC;
       writedata, dataadr: buffer STD_LOGIC_VECTOR(31 downto 0);
       memwrite:           buffer STD_LOGIC);
end;

architecture test of top is
  component mips
```

```
    port(clk, reset:      in STD_LOGIC;
         pc: out STD_LOGIC_VECTOR(31 downto 0);
         instr: in STD_LOGIC_VECTOR(31 downto 0);
         memwrite: out STD_LOGIC;
         aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
         readdata: in STD_LOGIC_VECTOR(31 downto 0));
end component;
component imem
  port(a:  in  STD_LOGIC_VECTOR(5 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component dmem
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:      out STD_LOGIC_VECTOR(31 downto 0));
end component;
signal pc, instr,
       readdata: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  mips1: mips port map(clk, reset, pc, instr, memwrite,
                       dataadr, writedata, readdata);
  imem1: imem port map(pc(7 downto 2), instr);
  dmem1: dmem port map(clk, memwrite, dataadr, writedata, readdata);
end;
```

Пример HDL-кода 7.14 ПАМЯТЬ ДАННЫХ

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is -- data memory
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR (31 downto 0);
       rd:      out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of dmem is
begin
```

```
  process is
    type ramtype is array (63 downto 0) of
                          STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if rising_edge(clk) then
          if (we = '1') then mem (to_integer(a(7 downto 2))):=wd;
          end if;
      end if;
      rd <= mem (to_integer(a (7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

Пример HDL-кода 7.15 ПАМЯТЬ КОМАНД

SystemVerilog

```
module imem(input  logic [5:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat", RAM);
  assign rd = RAM[a]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity imem is -- instruction memory
  port(a:in STD_LOGIC_VECTOR(5 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
  process is
    file mem_file: TEXT;
    variable L: line;
```

```
  variable ch: character;
  variable i, index, result: integer;
  type ramtype is array (63 downto 0) of
                        STD_LOGIC_VECTOR(31 downto 0);
  variable mem: ramtype;
begin
  -- initialize memory from file
  for i in 0 to 63 loop -- set all contents low
    mem(i) := (others =>'0');
  end loop;
  index := 0;
  FILE_OPEN (mem_file, "C:/docs/DDCA2e/hdl/memfile.dat",
             READ_MODE);
  while not endfile(mem_file) loop
    readline(mem_file, L);
    result := 0;
    for i in 1 to 8 loop
      read (L, ch);
      if '0' <= ch and ch <= '9' then
          result := character'pos(ch) - character'pos('0');
      elsif 'a' <= ch and ch <= 'f' then
      result := character'pos(ch) - character'pos('a')+10;
      else report "Format error on line" & integer'
           image(index) severity error;
      end if;
      mem(index)(35-i*4 downto 32-i*4) :=
          to_std_logic_vector(result,4);
    end loop;
    index := index + 1;
  end loop;
```

```
    -- read memory
    loop
      rd <= mem(to_integer(a));
      wait on a;
    end loop;
  end process;
end;
```

7.7 ИСКЛЮЧЕНИЯ*

Раздел 6.7.2 вводит понятие исключений, которые вызывают незапланированные изменения в порядке следования команд в программе. В этом разделе мы расширим наш многотактный процессор, чтобы он поддерживал два вида исключений: неопределенную команду и арифметическое переполнение. Поддержка исключений в прочих микроархитектурах основана на тех же принципах.

Как описано в **разделе 6.7.2**, когда случается исключение, процессор копирует счетчик команд в регистр EPC (Exception Program Counter) и записывает специальный код в регистр причины исключения (Cause register), сохраняя, таким образом, информацию о причине исключения. Код причины, равный 0x28, означает неопределенную команду, а 0x30 – арифметическое переполнение (см. **Табл. 6.7**). Затем процессор выполняет переход к обработчику исключения, который располагается

по адресу 0x80000180. Обработчик исключения – это программный код, который исполняется в ответ на возникновение исключения. Этот обработчик является частью операционной системы.

Как мы уже обсуждали в **разделе 6.7.2**, регистры причины исключения и EPC являются частью Сопроцессора 0 – той частью процессора MIPS, которая выполняет системные функции. Сопроцессор 0 может содержать до 32 регистров специального назначения, включая регистры причины исключения и EPC. Обработчик исключения может использовать команду mfc0 (прочитать из Сопроцессора 0), чтобы скопировать эти регистры специального назначения в один из регистров общего назначения в регистровом файле; внутри Сопроцессора 0 регистр причины исключения расположен под номером 13, а EPC – под номером 14.

Для того чтобы обрабатывать исключения, мы должны добавить регистры причины исключения и EPC в тракт данных и расширить мультиплексор PCSrc так, чтобы можно было выбирать еще и адрес обработчика исключений, как показано на **Рис. 7.62**. Эти два новых регистра имеют сигналы разрешения записи (EPCWrite и CauseWrite соответственно), чтобы при возникновении исключения была возможность сохранить счетчик команд (PC) и регистр причины исключения. Код причины формируется мультиплексором, который выбирает нужное значение для случившегося исключения. АЛУ также

должно формировать сигнал переполнения, как мы уже обсудили в **разделе 5.2.4**.[12]

Чтобы добавить поддержку команды mfc0, мы должны добавить механизм, который позволит выбирать регистры Сопроцессора 0 и записывать их в регистровый файл, как показано на **Рис. 7.63**. Поле $Instr_{15:11}$ используется командой mfc0 для того, чтобы указать номер регистра сопроцессора; на этом рисунке присутствуют только два регистра – регистры причины исключения и EPC. Мы также должны добавить дополнительный вход для мультиплексора MemtoReg, чтобы выбирать значение, поступившее от Сопроцессора 0.

[12] Строго говоря, АЛУ должно формировать сигнал переполнения только для команд `add` и `sub`.

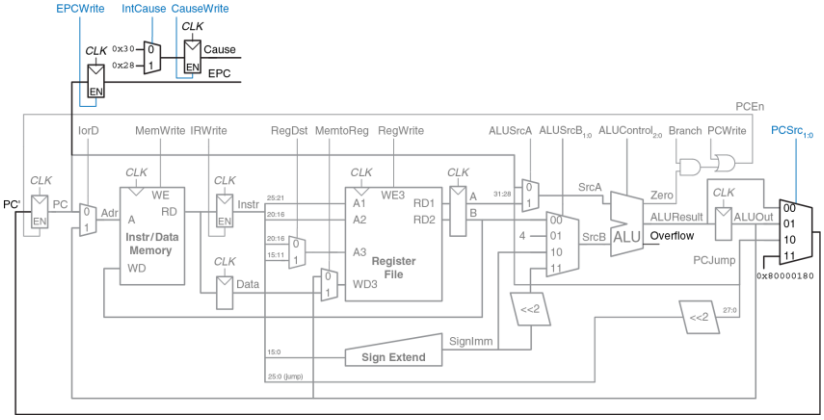


Рис. 7.62 Тракт данных с поддержкой исключений

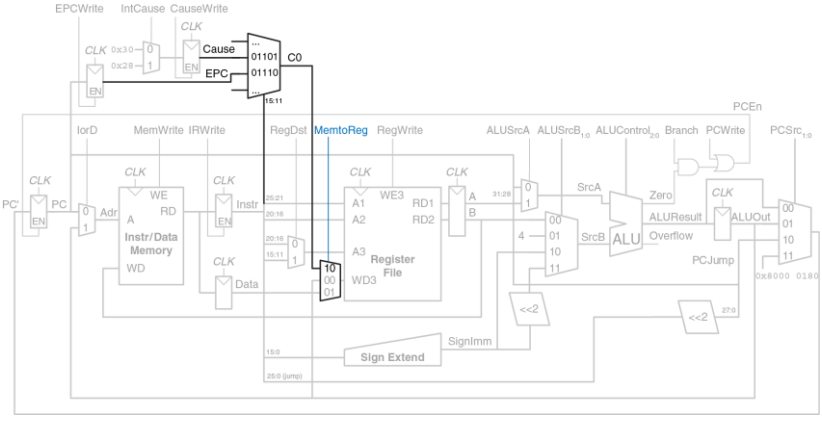


Рис. 7.63 Тракт данных с поддержкой команды `mfc0`

Модифицированное устройство управления показано на **Рис. 7.64**. Оно получает сигнал переполнения от АЛУ и формирует три новых управляющих сигнала: сигнал записи в регистр EPC, сигнал записи в регистр причины исключения и сигнал выбора регистра причины исключения. Также оно включает три новых состояния – два для

поддержки новых исключений (по одному на каждое) и еще одно – для команды `mfc0`.

Если устройство управления получает неопределенную команду (команду, которую неизвестно как выполнять), то оно переходит в состояние S12, сохраняет счетчик команд в регистр EPC, записывает 0x28 в регистр причины исключения и выполняет безусловный переход к обработчику исключения. Аналогично, при обнаружении арифметического переполнения в командах `add` или `sub`, устройство управления переходит в состояние S13, сохраняет счетчик команд в регистр EPC, записывает 0x30 в регистр причины исключения и передает управление обработчику исключения. Обратите внимание, что когда случается исключение, то выполнение команды прерывается, а в регистровый файл ничего не пишется. Когда процессор дешифровал очередную команду и понял, что это `mfc0`, он переходит в состояние S14 и записывает соответствующий регистр Сопроцессора 0 в регистровый файл.

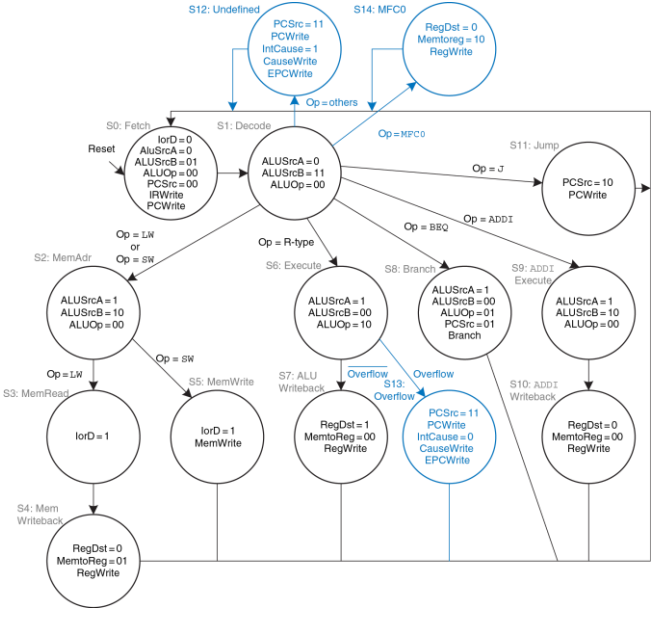


Рис. 7.64 **Устройство управления с поддержкой исключений и команды `mfc0`**

7.8 УЛУЧШЕННЫЕ МИКРОАРХИТЕКТУРЫ*

Высокопроизводительные микропроцессоры используют большое разнообразие приемов для того, чтобы выполнять программы быстрее. Вспомним, что время, требуемое для выполнения программы, пропорционально периоду тактового сигнала, а также среднему количеству тактов на команду (clock cycles per instruction – CPI). Таким образом, чтобы увеличить производительность, необходимо либо ускорить тактовый сигнал, либо снизить CPI. Этот раздел представляет собой обзор некоторых способов достичь этого. Реализация этих способов является довольно сложной, поэтому в этом курсе мы сфокусируемся только на общих концепциях. Книга Хеннесси и Паттерсона *Архитектура Компьютера* (John L. Hennessy, David A. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach. The Morgan Kaufmann Series in Computer Architecture. Sep 30, 2011) является наиболее авторитетным источником для тех, кто захочет полностью изучить все детали.

Каждые 2–3 года развитие технологий производства КМОП уменьшает размеры каждого транзистора на 30% в каждом измерении, тем самым удваивая количество транзисторов, которое можно разместить на кристалле. Технологический процесс характеризуется линейным разрешением (feature size), определяющим размеры самого маленького транзистора, который можно надежно произвести с помощью этого

процесса. С уменьшением размера транзисторы работают быстрее и, как правило, потребляют меньше электроэнергии. Таким образом, даже если не менять микроархитектуру, частота тактового сигнала может увеличиться просто потому, что все логические элементы стали быстрее. Кроме того, чем меньше по размеру транзисторы, тем больше их поместится на кристалле. Разработчики микроархитектуры используют дополнительные транзисторы, чтобы строить более сложные процессоры или помещать больше процессоров на кристалл. К сожалению, увеличение количества транзисторов и скорости, на которой они работают, вызывает увеличение энергопотребления. На настоящий момент энергопотребление стало одной из главных забот разработчиков микропроцессоров (см. **раздел 1.8**). Перед ними встает непростая задача добиться компромисса между скоростью, энергопотреблением и ценой микросхем, некоторые из которых содержат миллиарды транзисторов и являются одними из самых сложных систем, когда-либо созданных человечеством.

7.8.1 Длинные конвейеры

Помимо улучшения технологического процесса, простейший способ увеличить тактовую частоту процессора – поделить конвейер на большее количество стадий. Каждая стадия в этом случае содержит меньше логики и, следовательно, может работать быстрее. В начале

этой главы мы рассмотрели классический пятистадийный конвейер, но в наши дни нередко можно увидеть конвейеры в 10–20 стадий.

Максимальное количество стадий конвейера ограничено конфликтами в конвейере, накладными расходами на временные регистры (т.е. их временем предустановки и удержания), стоимостью разработки и производства. Более длинные конвейеры приводят к большему числу зависимостей внутри процессора. Некоторые из зависимостей могут быть разрешены при помощи байпаса (bypassing или forwarding), т.е. передачи результата из более поздних стадий конвейера в более ранние напрямую, минуя регистровый файл. Другие зависимости требуют приостановок (stalls), которые увеличивают CPI. Регистры, которые находятся между стадиями конвейера, приводят к накладным расходам из-за их времени предустановки (setup time) и задержки распространения (clk-to-Q delay), а также из-за сдвига фазы тактового сигнала (clock skew), вследствие чего добавление каждой последующей стадии дает все меньший прирост производительности. Наконец, добавление большего количества стадий увеличивает стоимость разработки и производства процессора, так как приходится добавлять регистры между стадиями и логику для разрешения более сложных конфликтов.

Пример 7.11 ДЛИННЫЕ КОНВЕЙЕРЫ

Давайте построим конвейерный процессор, поделив однотактный процессор на N стадий ($N \geq 5$). Задержка распространения сигнала через комбинационную логику однотактного процессора составляет 875 пс. К этому нужно добавить 50 пс, представляющие собой накладные расходы на использование регистров, окружающих эту логику, и равные сумме времени предустановки (setup) и времени задержки (clk-to-Q). Предположим, что комбинационная логика может быть поделена на произвольное число стадий, а логика обнаружения конфликтов не увеличивает ее задержку. CPI пятистадийного конвейера из **примера 7.9** равен 1,15. Также предположим, что каждая дополнительная стадия увеличивает CPI на 0,1 из-за неправильного предсказания переходов и прочих конфликтов. При каком числе стадий процессор будет выполнять программы быстрее всего?

Решение: если мы разделим задержку комбинационной логики, равную 875 пс, на N стадий и учтем, что для каждой стадии нужно добавить накладные расходы на использование временного регистра, равные 50 пс, то длительность такта будет равна $T_c = (875/N + 50)$ пс. CPI будет равно $1{,}15 + 0{,}1(N - 5)$. Время выполнения одной команды равно произведению длительности такта на CPI. На **Рис. 7.65** приведен график зависимости длительности такта и длительности выполнения команды (Instruction time) от числа стадий. Минимальная длительность выполнения команды равна 227 пс при числе стадий $N = 11$. Это всего лишь чуть-чуть лучше, чем 245 пс на команду, которые можно достичь, используя шестистадийный конвейер.

В конце 1990 и начале 2000 годов усилиями маркетологов считалось, что чем выше частота микропроцессора ($1/T_c$), тем он лучше. Это привело к тому, что разработчики стали использовать очень длинные конвейеры (от 20 до 31 стадии в Pentium 4), чтобы максимально увеличить частоту, даже если преимущества в производительности были сомнительными. Энергопотребление микропроцессора пропорционально частоте тактового сигнала, а также увеличивается из-за увеличивающегося числа регистров между стадиями, поэтому сейчас, когда энергопотребление так важно, число стадий в микропроцессорах уменьшается.

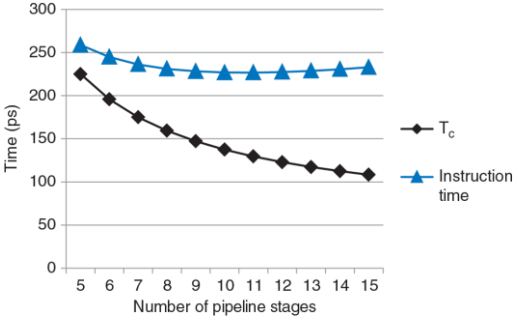


Рис. 7.65 Зависимость длительности такта и длительности выполнения команды от числа стадий

7.8.2 Предсказание условных переходов

Теоретически CPI идеального конвейерного процессора должно быть равно единице. Одной из основных причин более высокого CPI является простой процессора из-за неправильно предсказанных условных переходов (branch misprediction penalty). С увеличением длины конвейера необходимость перехода выясняется во все более поздних стадиях конвейера. Таким образом, простой из-за неправильно предсказанных переходов становится все больше и больше, так как конвейер должен быть очищен (flushed) от всех команд, выбранных после неправильно предсказанного перехода. Чтобы разрешить/устранить/уменьшить влияние эту проблему, большинство конвейерных процессоров используют *предсказатель условных переходов (branch predictor)*, позволяющий с высокой вероятностью угадать, стоит ли осуществлять переход. Вспомним, что наш конвейер из **раздела 7.5.3** всегда просто предсказывал, что переход осуществлять не стоит.

Некоторые условные переходы происходят, когда программа доходит до конца цикла (т.е. в операторах for или while) и переходит к его началу для новой итерации. Циклы часто выполняются много раз, поэтому такие условные переходы назад как правило исполняются. Простейший метод предсказания условных переходов – это проверить направление перехода и считать, что переход назад всегда будет выполнен. Такой

метод называется *статическим предсказанием переходов (static branch prediction)*, потому что он не зависит от истории выполнения команд программы.

Переходы вперед трудно предсказать без детального понимания конкретной программы. Из-за этого большинство процессоров используют *динамические предсказатели переходов (dynamic branch predictors)*, которые используют историю выполнения программы для того, чтобы предсказать, нужно ли выполнить переход. Динамические предсказатели переходов содержат таблицу с последними несколькими сотнями (или тысячами) команд условного перехода, выполненными процессором. Эта таблица, которую иногда называют *буфером целевых адресов ветвлений (branch target buffer)*, содержит адреса переходов и информацию о том, был ли переход выполнен.

Чтобы проиллюстрировать работу динамического предсказателя переходов, рассмотрим код цикла из **примера 6.20**. Цикл повторяется 10 раз, причем команда `beq` для выхода из цикла выполняется только на последней итерации.

```
   add   $s1, $0,  $0     # sum = 0
   addi  $s0, $0,  0      # i = 0
   addi  $t0, $0,  10     # $t0 = 10
for:
   beq   $s0, $t0, done   # if i == 10, branch to done
   add   $s1, $s1, $s0    # sum = sum + i
   addi  $s0, $s0, 1      # increment i
   j     for
done:
```

Однобитный динамический предсказатель переходов (one-bit dynamic branch predictor) запоминает, был ли переход выполнен в прошлый раз, и предсказывает, что в следующий раз произойдет то же самое. Пока цикл повторяется, предсказатель помнит, что в прошлый раз команда `beq` не была выполнена, и предсказывает, что она не будет выполнена и в следующий раз. Это предсказание остается правильным вплоть до последней итерации, на которой переход все-таки выполняется.

К сожалению, если цикл запустить снова, то предсказатель переходов будет помнить, что в последний раз условный переход был выполнен. Поэтому на первой итерации запущенного заново цикла предсказатель ошибется – неправильно предскажет, что переход нужно выполнить. Итого, однобитный предсказатель переходов ошибется и на первой, и на последней итерации этого цикла.

Двухбитный динамический предсказатель переходов решает эту проблему, используя четыре состояния: переход точно выполнится, переход скорее выполнится, переход скорее не выполнится и переход точно не выполнится *(strongly taken, weakly taken, weakly not taken, strongly not taken)*, как показано на **Рис. 7.66**.

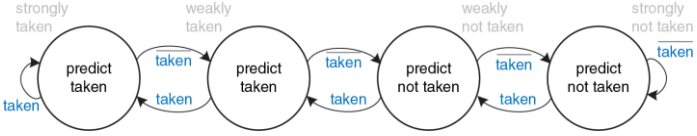


Рис. 7.66 **Диаграмма состояний двухбитного предсказателя переходов**

Пока цикл повторяется, предсказатель переходит в состояние «точно не выполнится» и остается в нем, предсказывая, что условный переход не будет выполнен и в следующий раз. Это предсказание остается верным вплоть до последней итерации, на которой условный переход выполняется и переводит предсказатель в состояние «скорее не выполнится». Когда цикл начнется снова, предсказатель переходов правильно предскажет, что переход не должен быть выполнен, и снова перейдет в состояние «точно не выполнится». Резюмируя, двухбитный предсказатель переходов неправильно предсказывает только переход на последней итерации цикла.

Как легко представить, предсказатели переходов могут следить и за большей историей выполнения программы, тем самым повышая точность предсказаний. Для типичных программ точность хороших предсказателей переходов превышает 90%.

Предсказатель переходов работает на стадии выборки команд из памяти (стадия Fetch) конвейера и используется процессором, чтобы определить, какую команду выбирать на следующем такте. Когда предсказатель говорит, что условный переход будет выполнен, процессор выбирает следующую команду из ячейки памяти, адрес которой находится в таблице адресов переходов (branch target buffer). В этой таблице удобно держать адреса назначения как условных, так и безусловных переходов, чтобы избежать очистки конвейера от напрасно выбранных команд, следующих за командой безусловного перехода.

7.8.3 Суперскалярный процессор

Скалярный процессор (scalar processor) в каждый момент времени осуществляет вычисления только над одной порцией данных (примечание переводчика: здесь и далее авторы имеют в виду число команд, одновременно находящихся в стадии Execute, а не во всем конвейере целиком). *Векторный процессор (vector processor)* работает над несколькими порциями данных одновременно, но использует для этого только одну команду. *Суперскалярный процессор (superscalar processor)* запускает на выполнение (issues) несколько команд одновременно, каждая из которых оперирует над одной порцией данных.

Конвейерный процессор MIPS, который мы разработали в этой главе, является скалярным. Векторные процессоры часто использовались для суперкомпьютеров 1980-х и 1990-х годов, так как позволяли эффективно обрабатывать длинные вектора (небольшие массивы) данных, часто встречающиеся в научных вычислениях. Современные высокопроизводительные процессоры являются суперскалярными, так как запуск на выполнение нескольких независимых команд одновременно является более гибким подходом, чем подход, применяемый в векторных процессорах.

При этом современные процессоры также включают в себя аппаратные расширения для работы с короткими векторами данных, которые повсеместно применяются в графических и мультимедийных

приложениях. Такие расширения называются SIMD-блоками (*Single Instruction Multiple Data*) и позволяют использовать одиночный поток команд для обработки множественного потока данных (примечание переводчика: хотя процессор MIPS, описанный в этой книге, содержит простой скалярный конвейер, но существуют и суперскалярные реализации архитектуры MIPS, и реализации с векторными сопроцессорами, и реализации с SIMD-расширениями).

Тракт данных *суперскалярного процессора* содержит несколько копий функциональных блоков, что позволяет ему выполнять несколько команд одновременно. На **Рис. 7.67** показана диаграмма двухканального (2-way) суперскалярного процессора, который осуществляет выборку и выполнение двух команд за такт. Тракт данных выбирает из памяти команд две команды за раз. Он содержит регистровый файл с шестью портами, чтобы читать четыре операнда и записывать назад два результата на каждом такте. Тракт данных также содержит два АЛУ и двухпортовую память данных, чтобы выполнять две команды одновременно.

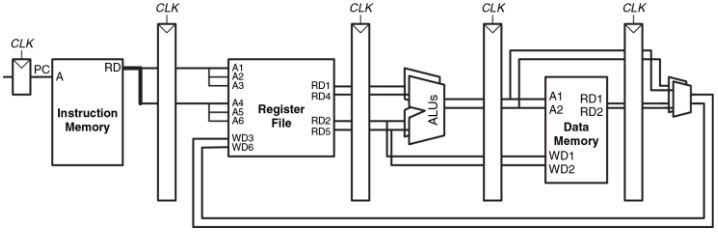


Рис. 7.67 **Тракт данных суперскалярного процессора**

На **Рис. 7.68** показана диаграмма двухканального суперскалярного процессора, который выполняет две инструкции на каждом такте. В этом случае CPI процессора равен 0,5. Разработчики зачастую используют величину, обратную CPI – количество команд на такт (instructions per cycle, IPC), которое для этого процессора равно 2,0.

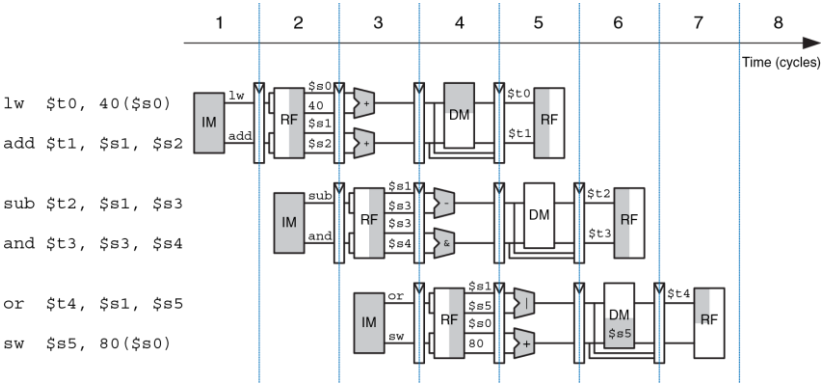


Рис. 7.68 **Работающий суперскалярный конвейер**

Выполнять много команд одновременно довольно трудно из-за зависимостей между ними. Рассмотрим **Рис. 7.69**, на котором показана диаграмма конвейера, выполняющего программу с зависимостями между данными. Зависимости в коде показаны голубым цветом. Команда `add` зависит от `$t0`, значение которого изменяет команда `lw`, поэтому эти команды нельзя запускать на выполнение одновременно. На самом деле, команда `add` приостанавливается еще на один такт,

чтобы `lw` могла переслать через байпас прочитанное из памяти значение `$t0` команде `add` на пятом такте. Оставшиеся конфликты (между `sub` и `and` из-за `$t0` и между `or` и `sw` из-за `$t3`) разрешаются при помощи пересылки результатов через байпас. Эта программа, показанная ниже, выполняется за пять тактов, а IPC равно 1,17.

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Вспомните, что у параллелизма две формы – временная и пространственная. Конвейерная реализация – это пример временного параллелизма, а наличие нескольких экземпляров одних и тех же функциональных блоков – пример пространственного. Суперскалярные процессоры используют обе формы параллелизма, чтобы достичь производительности, значительно превосходящей производительность наших однотактного и многотактного процессоров.

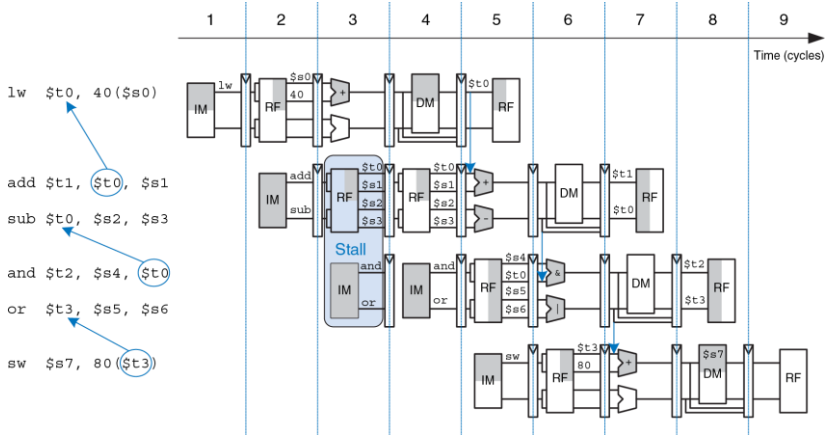


Рис. 7.69 Программа с зависимостями между командами

Коммерческие суперскалярные процессоры могут быть трех-, четырех- или даже шестиканальными. Им приходится обрабатывать и конфликты управления, вызываемые, например, условными переходами, и конфликты данных. К сожалению, в реальных программах встречается много зависимостей, поэтому суперскалярные процессоры

с большим количеством каналов редко могут использовать все свои функциональные блоки полностью. Более того, большое количество функциональных блоков и сложности с организацией байпаса требуют множества дополнительных логических элементов и потребляют огромное количество электроэнергии.

7.8.4 Процессор с внеочередным выполнением команд

Чтобы справиться с проблемой зависимостей, процессор с внеочередным выполнением команд (out-of-order processor) заранее просматривает большое количество команд, которые, по его мнению, надо будет скоро выполнить, чтобы как можно быстрее обнаружить и запустить на выполнение те команды, которые не зависят друг от друга. Команды могут выполняться не в том в порядке, в котором они находятся в программе, но только при условии, что процессор учитывает все зависимости, что позволит программе выдать ожидаемый результат.

Рассмотрим выполнение той же программы, которую мы привели на **Рис. 7.69**, на двухканальном суперскалярном процессоре с внеочередным выполнением команд. За один такт процессор может запускать до двух команд из любой части программы при условии, что он соблюдает все зависимости. На **Рис. 7.70** показаны зависимости данных и работа процессора.

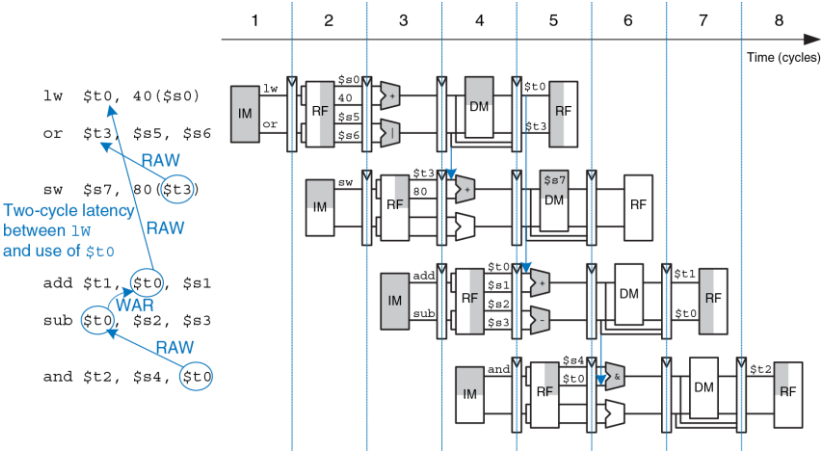


Рис. 7.70 **Внеочередное выполнение команд, зависящих друг от друга**

Чуть позже мы обсудим классификацию зависимостей (RAW и WAR). Ниже описаны ограничения на запуск команд:

- Такт 1
  - Команда `lw` запускается на выполнение.
  - Команды `add`, `sub` и `and` зависят от `lw`, так как используют `$t0`, поэтому их пока запустить нельзя. А вот команда `or` не зависит от `lw`, поэтому она тоже запускается на выполнение.
- Такт 2
  - Вспомните, что между запуском команды `lw` и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому `add` запустить пока нельзя, так как она зависит от `$t0`. Команда `sub` записывает результат в `$t0`, поэтому ее нельзя запускать перед `add`, иначе `add` получит неверное значение `$t0`. Команда `and` зависит от `sub`.
  - Запускается только команда `sw`.

- Такт 3
  - На третьем такте значение в `$t0` становится корректным, поэтому запускается `add`. Команда `sub` тоже запускается на выполнение, потому что `add` прочитает `$t0` раньше, чем `sub` изменит его.
- Такт 4
  - Запускается команда `and`. Значение `$t0` пересылается от `sub` к `and` через байпас.

Таким образом, процессор с внеочередным выполнением команд запускает шесть команд за четыре такта, то есть его IPC равно 1,5.

Зависимость `add` от `lw` из-за использования `$t0` называется конфликтом чтения после записи, или RAW-конфликтом *(Read-After-Write, RAW)*. Команда `add` не имеет права читать `$t0` до тех пор, пока `lw` в него не запишет. Мы уже встречались с таким типом зависимости, когда рассматривали конвейерный процессор, и знаем, как с ней справляться. Такая зависимость по своей природе ограничивает скорость выполнения программы, даже если у процессора есть бесконечно много функциональных блоков для выполнения команд. Аналогично, зависимость `sw` от `or` из-за использования `$t3` и

зависимость `and` от `sub` из-за использования `$t0` являются RAW-зависимостями.

Зависимость между `sub` и `add` из-за использования `$t0` называется конфликтом записи после чтения, или WAR-конфликтом (*Write-After-Read, WAR)*, или *антизависимостью (antidependence)*. Команда sub не имеет права писать в `$t0` до того, как add прочитает его. Это необходимо, чтобы команда `add` получила правильное значение в соответствии с исходным порядком команд в программе. WAR-конфликты не могут возникнуть в обычной реализации конвейера MIPS, но могут возникнуть в процессоре с внеочередным выполнением команд, если он попытается запустить зависимую команду (в данном случае `sub`) слишком рано.

В отличие от RAW-конфликтов, WAR-конфликты не являются неизбежными во время работы программы. Это просто следствие решения программиста (или компилятора) использовать один и тот же регистр для двух несвязанных друг с другом команд. Если бы команда `sub` записывала бы результат в `$t4`, а не в `$t0`, то зависимость исчезла бы и можно было бы запустить `sub` перед `add`.

Третий тип конфликта, не показанный в программе, называется конфликтом записи после записи, или WAW-конфликтом *(Write-After-Write, WAW)*. Его еще называют *зависимостью вывода (output dependency)* или *ложной зависимостью (false dependency)*.

WAW-конфликт случается, когда команда пытается писать в регистр после того, как в него уже записала следующая по ходу программы команда. В результате этого конфликта в регистр будет записано неверное значение. Например, в программе ниже две команды, `add` и `sub`, пишут в `$t0`. Согласно порядку команд в программе, окончательное значение в `$t0` должна записать `sub`. Если бы процессор с внеочередным выполнением команд попытался выполнить `sub` перед `add`, то произошел бы WAW-конфликт.

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

Как и WAR-конфликты, WAW- конфликты также не являются неизбежной ситуацией. Они, опять же, возникают из-за решения программиста (или компилятора) использовать один и тот же регистр для двух несвязанных друг с другом команд. Если команда `sub` была запущена первой, процессор мог бы устранить WAW- конфликт, запретив команде `add` записывать куда-либо свой результат. Этот прием называют «раздавливанием» *(squashing)* команды `add`.[13]

[13] Вы можете спросить, зачем вообще нужно запускать команду `add`? Дело в том, что процессоры с внеочередным выполнением команд обязаны гарантировать, что во время выполнения программы произойдут все те же исключения, которые произошли бы, если бы все команды выполнялись в исходном порядке. Команда `add` может потенциально

Процессоры с внеочередным выполнением команд используют специальную таблицу, чтобы отслеживать команды, ожидающие запуска. Эта таблица, иногда называемая таблом готовности (*scoreboard),* содержит информацию о зависимостях тех команд, которые процессор только собирается запустить. Размер таблицы определяет, сколько команд одновременно могут являться кандидатами на запуск. На каждом такте процессор сверяется с таблицей и запускает столько команд, сколько он может, с учетом зависимостей и количества доступных функциональных блоков (АЛУ, портов памяти и т.д.)

Параллелизм на уровне команд (*instruction level parallelism, ILP*) – это число команд конкретной программы, которые могут одновременно выполняться на определенной микроархитектуре. Теоретические исследования показали, что ILP для микроархитектур с внеочередным выполнением команд, при условии идеального предсказания переходов и огромного количества функциональных блоков, может быть довольно высоким. К сожалению, на практике даже шестиканальные суперскалярные процессоры с внеочередным выполнением команд редко достигают ILP, превышающего 2 или 3.

вызвать исключение из-за арифметического переполнения, поэтому ее необходимо запустить для того, чтобы проверить, произойдет переполнение или нет, даже если ее результат будет не нужен.

7.8.5 Переименование регистров

Для того, чтобы устранить WAR-конфликты, процессоры с внеочередным выполнением команд используют прием, который называется *переименованием регистров (register renaming)*. Для реализации переименования регистров в процессор добавляют так называемые неархитектурные (невидимые программисту) *регистры переименования (renaming registers)*. Например, в процессор MIPS может быть добавлено 20 регистров переименования, называемых `$r0`-`$r19`. Программист не может использовать эти регистры напрямую, поскольку они не являются частью архитектуры. Но при этом процессор может использовать их для устранения конфликтов.

Например, в предыдущем разделе был показан WAR-конфликт между командами `sub` и `add`, который случился из-за повторного использования `$t0`. Процессор с внеочередным выполнением команд может *переименовать* `$t0` в `$r0` для команды `sub`. После этого `sub` можно выполнить быстрее, потому что у `$r0` нет зависимости от команды `add`. У процессора есть таблица с информацией о том, какие регистры были переименованы, поэтому он может соответствующим образом переименовать регистры и в последующих зависимых командах. В этом примере `$t0` необходимо переименовать в `$r0` еще в команде `and`, потому что она использует результат команды `sub`.

На **Рис. 7.71** показана та же программа, что и на **Рис. 7.69**, но выполняемая на процессоре с переименованием регистров. Чтобы устранить WAR-конфликт, регистр `$t0` был переименован в `$r0` в командах `sub` и `and`. Ниже описаны ограничения на запуск команд.

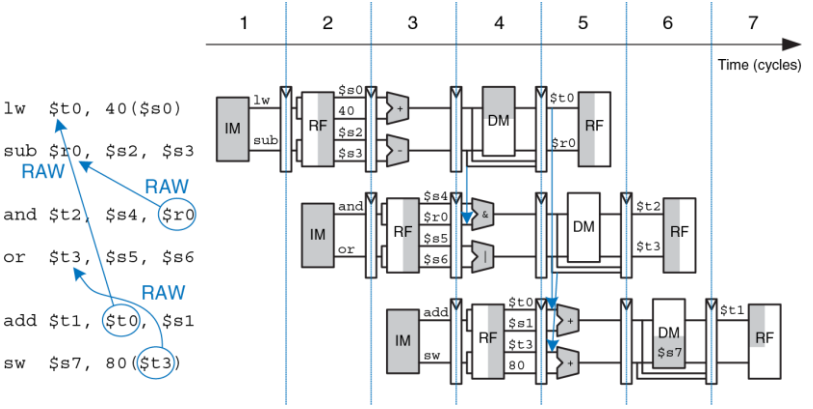


Рис. 7.71 Внеочередное выполнение команд с переименованием регистров

- Такт 1
  - Команда `lw` запускается на выполнение.
  - Команда `add` зависит от `lw` из-за использования `$t0`, поэтому пока что запустить ее нельзя. Однако команда `sub` теперь независима, так как ее регистр результата переименован в `$r0`, поэтому `sub` также запускается на выполнение.
- Такт 2
  - Вспомните, что между запуском команды `lw` и моментом, когда ее результат может быть использован зависимой от нее командой, существует задержка в два такта. Поэтому `add` запустить пока нельзя, так как она зависит от `$t0`. Команда `and` зависит от уже запущенной на выполнение `sub`, поэтому ее тоже можно запускать – значение `$r0` будет передано от `sub` к `and` через байпас.
  - У команды `or` нет зависимостей, поэтому она тоже запускается.
- Такт 3

– На третьем такте значение в `$t0` становится корректным, поэтому запускается `add`. Значение в `$t3` также становится корректным, поэтому запускается и `sw`.

Таким образом, процессор с внеочередным выполнением команд и переименованием регистров запускает шесть команд за три такта, то есть его IPC равно 2.

7.8.6 SIMD

Одиночный поток команд, множественный поток данных (single instruction multiple data, SIMD) – это способ обработки данных, при котором одна команда обрабатывает множество блоков данных параллельно. Типичным примером использования SIMD, особенно при обработке данных для компьютерной графики, является выполнение арифметической операции над несколькими небольшими независимыми блоками данных. Такая обработка также называется *пакованной арифметикой (packed arithmetic)*.

Например, 32-битный микропроцессор может упаковывать четыре 8-битных элемента (блока) данных в одно 32-битное слово. Команды сложения и вычитания упакованных чисел (*packed add* и *packed sub* соответственно) обрабатывают все четыре элемента данных одновременно. На **Рис. 7.72** показано, как команда сложения

упакованных чисел суммирует четыре пары 8-битных чисел и получает четыре результата. 32-битное слово можно было бы разделить и на два 16-битных элемента. Для выполнения команд пакованной арифметики необходимо модифицировать его АЛУ так, чтобы можно было отключать сигналы переноса между элементами данных. То есть арифметическое переполнение при выполнении $a_0 + b_0$ не должно влиять на результат выполнения $a_1 + b_1$.

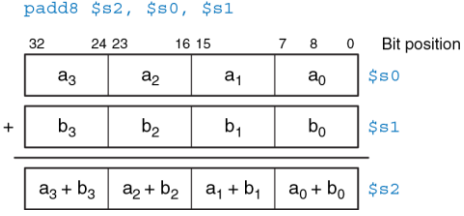


Рис. 7.72 Пакованная арифметика: четыре одновременных операции

Короткие элементы данных часто используются в приложениях компьютерной графики. Например, пиксел на цифровом фотоснимке может использовать по восемь бит для хранения красного, зеленого и синего компонентов цвета. При использовании 32-битных машинных слов для обработки этих компонентов верхние 24 бита будут потрачены зря. Если же мы упакуем компоненты четырех смежных пикселей в

одно 32-битное слово, то обработка может выполняться в четыре раза быстрее.

SIMD-команды для 64-битных архитектур еще более эффективны, так как в одно 64-битное слово они могут упаковать восемь 8-битных, четыре 16-битных или два 32-битных элемента (примечание переводчика: современная версия архитектуры MIPS включает MIPS® SIMD Architecture, которая позволяет использовать 32 128-битных регистра, в каждый из которых можно упаковать шестнадцать 8-битных, восемь 16-битных, четыре 32-битных или два 64-битных элемента). SIMD-команды также используются для вычислений с плавающей точкой. Например, четыре 32-битных значения с плавающей точкой одинарной точности можно упаковать в одно 128-битное слово.

7.8.7 Многопоточность

Так как параллелизм на уровне команд (instruction level parallelism, ILP) у реальных программ, как правило, довольно низок, добавление все новых и новых функциональных блоков к суперскалярному процессору или процессору с внеочередным выполнением команд приводит ко все меньшему эффекту. Еще одной проблемой является то, что основная память гораздо медленнее, чем процессор (мы рассмотрим это в **главе 8**). Большинство команд загрузки и сохранения данных работают со значительно более быстрой и маленькой памятью, которая

называется *кэш-памятью (cache memory)*. К сожалению, если нужных команд или данных в кэше нет, то процессор может быть приостановлен на 100 и более тактов, ожидая получения информации из основной памяти. Многопоточность – это способ загрузить работой процессор с большим количеством функциональных блоков, даже если у программы низкий ILP или она приостановлена на время ожидания данных из памяти.

Для того, чтобы объяснить суть многопоточности, нам надо определить несколько новых терминов. Программа, которая выполняется на компьютере, называется *процессом (process)*. Компьютеры могут выполнять несколько процессов одновременно. Например, на своем ПК вы можете слушать музыку и сидеть в интернете, одновременно запустив антивирус. Каждый процесс состоит из одного или более *потоков команд (threads),* которые тоже выполняются одновременно. Например, в текстовом редакторе один поток может обрабатывать набор текста пользователем, второй поток в это время может проверять орфографию, третий – печатать документ на принтере. При такой организации пользователю не нужно ждать, пока закончится печатать, чтобы снова набирать текст. Степень, до которой процесс можно разделить на несколько одновременно выполняющихся потоков, определяет уровень параллелизма на уровне потоков (thread level parallelism, TLP).

В обычном процессоре одновременная работа потоков – это только иллюзия. Реально же потоки выполняются процессором по очереди под управлением операционной системы. Когда «смена» одного потока подходит к концу, ОС сохраняет его архитектурное состояние, загружает из памяти архитектурное состояние следующего потока и передает ему управление. Эта процедура называется *переключением контекста* (*context switching)*. До тех пор, пока процессор переключается между потоками достаточно быстро, пользователю кажется, что все потоки выполняются одновременно.

У многопоточного процессора есть несколько копий архитектурного состояния, вследствие чего несколько потоков могут быть активны одновременно (примечание переводчика: то, что несколько процессов могут быть активны одновременно, не означает, что все они могут выполняться одновременно. Обычно в каждый момент времени выполнятся только один из них. Существуют микроархитектуры, объединяющие многопоточность и суперскалярность – вот они могут выполнять несколько команд из разных потоков одновременно на одном ядре). Например если мы расширим наш процессор MIPS так, чтобы у него было четыре счетчика команд и 128 регистров, то одновременно могут быть доступны четыре потока. Если один из них приостанавливается в ожидании данных из основной памяти, то процессор немедленно переключает контекст на другой поток. Это

переключение происходит безо всяких накладных расходов, так как счетчик команд и регистры уже доступны и их не надо отдельно загружать. Более того, если один из потоков не может в полной мере использовать все функциональные блоки процессора из-за недостаточного уровня параллелизма, то другой поток может запустить на исполнение команды, использующие незанятые блоки.

Многопоточность не улучшает производительность отдельного потока, потому что она не повышает ILP. Тем не менее, она улучшает общую пропускную способность процессора, так как несколько потоков могут более полно использовать те ресурсы процессора, которые бы не использовались при выполнении одного-единственного потока. Многопоточность относительно легко реализовать, так как требуется добавить только копии счетчика команд и регистрового файла. Функциональные блоки и память копировать не надо.

7.8.8 Симметричные мультипроцессоры

Многопроцессорная система (multiprocessor system), или просто *мультипроцессор*, состоит из нескольких процессоров и аппаратуры для соединения их между собой. Наиболее часто встречающаяся форма многопроцессорности в компьютерных системах – это *гомогенная многопроцессорность (homogeneous multiprocessing),* также называемая *симметричной многопроцессорностью (symmetric*

multiprocessing, SMP), при которой два или более одинаковых процессоров подключены к общей основной памяти.

> Ученые, которые ищут признаки инопланетного разума, используют самый большой в мире кластер (cluster) мультипроцессоров для анализа данных от радиотелескопов с целью поиска шаблонов, которые могли бы быть признаками жизни в других солнечных системах. Этот кластер состоит из персональных компьютеров, принадлежащих более чем 3,8 миллионам волонтеров во всем мире. Когда компьютер из кластера простаивает, он получает блок данных из централизованного сервера, анализирует эти данные и отсылает результаты обратно на сервер. Вы тоже можете пожертвовать кластеру время простоя вашего компьютера, зайдя на вебсайт setiathome.berkeley.edu.

Процессоры в мультипроцессорной системе могут быть как несколькими отдельными микросхемами, так и несколькими *ядрами (cores)* внутри одной микросхемы. Современным процессорам доступно неимоверное количество транзисторов. Если использовать эти транзисторы только для того, чтобы увеличивать длину конвейера или добавлять функциональные блоки в суперскалярный процессор, то в определенный момент рост производительности станет незначительным, а энергопотребление зашкалит. Примерно в 2005 году разработчики компьютерных микроархитектур дружно принялись размещать по нескольких копий процессора на одном кристалле; эти

копии и называются ядрами (примечание переводчика: в 2014 году передовые процессорные компании разрабатывают процессоры с десятками и даже сотнями ядер на одном кристалле. Такие ядра трудно сделать когерентными между собой и с общей памятью, но и некогерентный мультипроцессор может быть полезен для приложений вроде независимой обработки большого количества сетевых пакетов).

Мультипроцессоры можно использовать или для того, чтобы выполнять больше потоков одновременно, или для того, чтобы быстрее выполнять один конкретный поток. Выполнять больше потоков одновременно довольно просто – эти потоки можно просто распределить между процессорами. К сожалению, пользователям персональных компьютеров обычно не нужно выполнять много потоков – им нужно выполнять всего несколько потоков, но при этом максимально быстро. Ускорение одного потока при помощи мультипроцессора – далеко не тривиальная задача. Чтобы достичь этого, программист должен разделить один медленный поток на несколько меньших потоков, которые можно будет запустить на разных процессорах. Все еще больше усложняется, если процессорам нужно обмениваться данными. На сегодняшний день эффективное использование большого количества процессорных ядер – одна из главных проблем, стоящих перед разработчиками компьютеров и программистами.

Другие формы многопроцессорности включают в себя *гетерогенную многопроцессорность (heterogeneous multiprocessing)* и кластеры. Гетерогенные мультипроцессоры, которые также называют *асимметричными мультипроцессорами (asymmetric multiprocessors),* используют разные специализированные микропроцессоры для разных задач и будут рассмотрены в следующем разделе. В случае *кластерной многопроцессорности (clustered multiprocessing)* у каждого процессора есть своя отдельная подсистема памяти. Кластером также называют группу обычных персональных компьютеров, соединенных вместе сетью и выполняющих специализированные программы, позволяющие компьютерам вместе решать масштабную задачу.

7.8.9 Гетерогенные мультипроцессоры

У гомогенных (симметричных) мультипроцессоров, описанных в **разделе 7.8.8**, есть несколько преимуществ. Их довольно легко разрабатывать, так как спроектированный один раз процессор можно затем просто скопировать необходимое число раз, чтобы увеличить общую производительность системы. Их также довольно легко программировать, потому что любая программа может выполняться на любом процессоре в системе и при этом обеспечивать примерно одинаковую производительность.

К сожалению, нет никакой гарантии, что с добавлением все большего и большего количества ядер производительность системы продолжит улучшаться. По состоянию на 2012 год пользовательские программы, запускаемые на домашних компьютерах, использовали в среднем 2–3 потока. У типичного пользователя, к тому же, обычно запущена всего пара таких программ одновременно. И хотя этого достаточно, чтобы загрузить двух- или четырехядерную систему, но добавление большего числа ядер будет приводить ко все менее заметным результатам до тех пор, пока программы не начнут использовать параллелизм более широко. Кроме этого, так как процессоры общего назначения разрабатываются с целью обеспечивать хорошую среднюю производительность на широком классе задач, то они, как правило, являются далеко не самым энергоэффективным способом выполнения той или иной конкретной операции. Энергоэффективность особенно важна в мобильных системах, где энергопотребление сильно ограничено.

Гетерогенные мультипроцессоры (heterogeneous multiprocessors), также называемые асимметричными, призваны решить эти проблемы путем использования различных типов процессорных ядер и/или специализированной аппаратуры в одной системе. Каждое приложение использует те ресурсы, которые позволяют достичь или наилучшей производительности, или наилучшего соотношения производительности и энергопотребления для этого конкретного

приложения. Так как в наши дни разработчики могут использовать сколько угодно транзисторов, то никого особенно не заботит, что не каждое приложение сможет использовать все имеющиеся в наличии аппаратные блоки. Гетерогенные системы могут принимать разные формы. Они могут включать в себя ядра с различными микроархитектурами, имеющие разное соотношение энергопотребления, производительности и занимаемой на кристалле площади. Например, система может включать в себя как простые ядра с последовательным выполнением команд (in-order execution), так и более сложные суперскалярные ядра с внеочередным выполнением команд. Приложения, которые могут эффективно использовать более высокопроизводительные, но менее энергоэффективные ядра, получают такую возможность, в то время как другие приложения, которые не могут эффективно использовать дополнительную вычислительную мощность суперскалярных ядер, будут использовать более энергоэффективные ядра с последовательным выполнением команд.

В такой системе ядра могут использовать как одну и ту же систему команд, что позволяет запускать приложения на любом из ядер, так и различные системы команд, что открывает дополнительные возможности по приспособлению ядер к поставленной задаче. Примером второго варианта является процессор Cell Broadband Engine (или просто Cell) от IBM. Cell включает в себя одно двухканальное

суперскалярное ядро общего назначения с последовательным выполнением команд (dual-issue in-order processor) с архитектурой IBM Power, которое называется power processor element (PPE), а также восемь синергетических процессорных ядер (synergistic processor elements, SPEs). Ядра SPE используют новую систему команд *SPU ISA*, которая была разработана для улучшения энергоэффективности при выполнении интенсивных математических вычислений. Хотя Cell позволяет использовать различные парадигмы программирования, но идея разработчиков заключается в том, чтобы PPE занимался задачами управления и контроля, такими как разделение нагрузки между несколькими SPE, в то время как сами SPE выполняли бы большую часть вычислений. Гетерогенная природа процессора Cell позволяет ему обеспечивать гораздо более высокую производительность при заданном энергопотреблении и площади кристалла, чем у обычных процессоров с архитектурой Power.

Другие гетерогенные системы могут включать смесь традиционных процессорных ядер и специализированной аппаратуры. Одним из первых подобных примеров стали сопроцессоры для вычислений с плавающей точкой. Раньше в микропроцессорах не хватало места на кристалле для блока вычислений с плавающей точкой, а пользователи, которым нужна была более высокая производительность для вычислений с плавающей точкой, могли добавить отдельную микросхему, реализующую их аппаратную поддержку. В современных

микропроцессорах может быть одно или даже несколько блоков вычислений с плавающей точкой на одном кристалле, а разработчики начинают добавлять другие типы специализированной аппаратуры. И у AMD, и у Intel есть процессоры, в которых на одном кристалле в дополнение к традиционным ядрам x86 размещены графические ускорители (*graphics processing unit*, *GPU*) и даже FPGA. Впервые процессор и графический ускоритель были размещены на одном кристалле в линейках процессоров AMD Fusion и Intel Sandy Bridge. Микросхемы Intel серии E600 (Stellarton), выпущенные в начале 2011 года, объединили на одном кристалле процессор Atom и Altera FPGA. Микросхема внутри сотового телефона содержит как обычный процессор, отвечающий за взаимодействие с пользователем (управление телефонной книгой, просмотр вебсайтов и компьютерные игры), так и процессор цифровой обработки сигналов (*digital signal processor*, *DSP*) со специализированными командами для дешифровки в реальном времени данных, передаваемых по каналам беспроводной связи. В системах с ограниченными возможностями энергопотребления такая интегрированная, специализированная аппаратура обеспечивает лучшее соотношение между производительностью и энергопотреблением, чем выполнение тех же операций на процессорном ядре общего назначения.

Система команд синергетического процессора

Система команд синергетического процессора (Synergistic Processor Unit, SPU) разработана для того, чтобы на определенных видах нагрузок обеспечивать тот же уровень производительности, что и процессоры общего назначения, но используя в два раза меньшую площадь на кристалле и энергопотребление. В область применения системы команд SPU входит графика, обработка потоков данных и другие нагрузки, требующие высокопроизводительных вычислений, таких как физика компьютерных игр и другие виды моделирования систем. Для достижения поставленных целей по производительности, энергопотреблению и занимаемой площади на кристалле эта система команд включает в себя несколько особенностей, в том числе 128-битное SIMD-расширение, программно-управляемую память (software-managed memory) и большой регистровый файл. При использовании программно-управляемой памяти программист должен в явном виде пересылать данные между внешней и локальной памятью, тогда как в традиционных процессорах за это отвечает кэш-память. Программно-управляемая память, если ее правильно использовать, может одновременно помочь уменьшить энергопотребление и улучшить производительность, так как необходимые данные будут пересылаться из внешней памяти в локальную заблаговременно и только тогда, когда они действительно нужны. Большой регистровый файл поможет избежать нехватки регистров без сложного механизма переименования регистров.

У гетерогенных систем есть и недостатки. Они сложнее и в разработке, и в программировании, так как требуется не только разработать разнообразные аппаратные блоки, но и решить, когда и как наилучшим образом использовать различные ресурсы системы. В конечном итоге, и у гомогенных, и у гетерогенных систем есть свои ниши. Гомогенные мультипроцессоры подходят, например, для больших центров обработки данных, где нет недостатка в задачах с высоким параллелизмом на уровне потоков. Гетерогенные системы хороши в случае более разнообразной вычислительной нагрузки и ограниченного параллелизма.

7.9 ЖИВОЙ ПРИМЕР: МИКРОАРХИТЕКТУРА X86

В **разделе 6.8** мы рассмотрели архитектуру x86, используемую практически во всех ПК. Пришло время взглянуть на эволюцию микроархитектур процессоров x86 – от простых ко все более быстрым и сложным. Те же принципы, которые мы применяли к микроархитектуре MIPS, были использованы и в x86.

Intel разработала первый однокристальный микропроцессор, четырехбитный 4004, в 1971 году с целью использования в качестве универсального контроллера для калькуляторов. Он содержал 2300 транзисторов на кристалле кремния площадью 12 мм^2, был изготовлен по 10 мкм технологическому процессу и работал с тактовой частотой 750 кГц. Фотография чипа, сделанная под микроскопом, показана на **Рис. 7.73**.

На ней видны столбцы, содержащие по четыре однотипных структуры, как и следовало ожидать от четырехбитного микропроцессора. На периферии видны соединительные провода (bond wires), которые используются для соединения кристалла микропроцессора с его корпусом и печатной платой.

Успех процессора 4004 вдохновил Intel на создание восьмибитного микропроцессора 8008, а затем и 8080, который, в свою очередь, превратился в 16-битные процессоры 8086 (1978 год) и 80286

(1982 год). В 1985 году Intel представила процессор 80386, который расширил архитектуру 8086 и превратил ее в 32-битную, ознаменовав появление архитектуры x86.

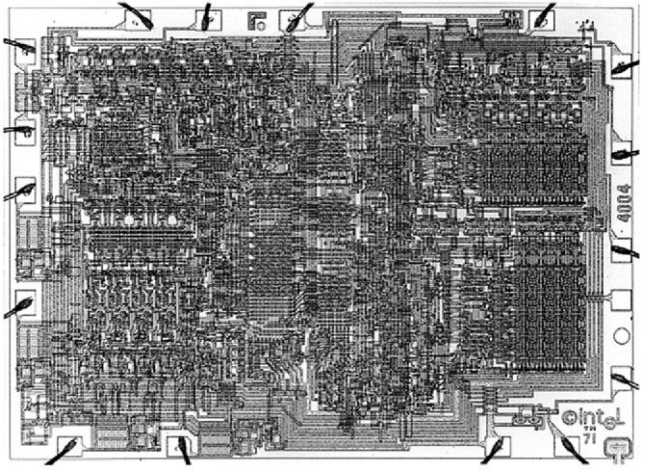


Рис. 7.73 **Микропроцессор Intel 4004**

В Табл. 7.7 приведены данные об основных микропроцессорах Intel x86. За 40 лет, прошедшие после создания 4004, линейные размеры транзисторов уменьшились в 160 раз, число транзисторов на кристалле увеличилось на пять порядков, а рабочая частота выросла почти на четыре порядка. Ни в одной другой области техники никогда не было такого удивительного прогресса в столь короткий промежуток времени.

Табл. 7.7 Эволюция микропроцессоров Intel x86

Процессор	Год	Техпроцесс (мкм)	Число транзисторов, млн	Частота (МГц)	Микроархитектура
80386	1985	1.5–1.0	0,275	16–25	Многотактная
80486	1989	1.0–0.6	1,2	25–100	Конвейерная
Pentium	1993	0.8–0.35	3,2–4,5	60–300	Суперскалярная
Pentium II	1997	0.35–0.25	7,5	233–450	Внеочередное выполнение
Pentium III	1999	0.25–0.18	9,5–28	450–1400	Внеочередное выполнение
Pentium 4	2001	0.18–0.09	42–178	1400–3730	Внеочередное выполнение
Pentium M	2003	0.13–0.09	77–140	900–2130	Внеочередное выполнение
Core Duo	2005	0.065	152	1500–2160	Двухядерная
Core 2 Duo	2006	0.065–0.045	167–410	1800–3300	Двухядерная
Core i3-i7	2009	0.045–0.032	382–731+	2530–3460	Многоядерная

Intel 80386 представлял собой многотактный процессор. На фотографии кристалла на **Рис. 7.74** отмечены его основные компоненты. Слева отчетливо виден 32-битный тракт данных (data path). Каждый столбец обрабатывает один бит данных.

Программируемая логическая матрица, содержащая *микрокод* (microcode PLA), используется устройством управления (controller) для того, чтобы определить порядок перехода между состояниями управляющего автомата. Блок управления памятью (memory management unit) в правом верхнем углу управляет доступом ко внешней памяти.

У процессора Intel 80486, изображённого на **Рис. 7.76**, производительность значительно улучшилась благодаря использованию конвейерной обработки. Снова отчетливо видны тракт данных, устройство управдения и программируемая логическая матрица с микрокодом.

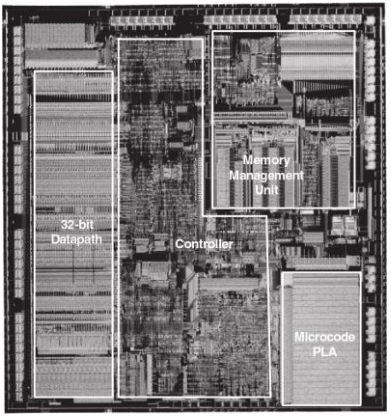


Рис. 7.74 Микропроцессор 80386

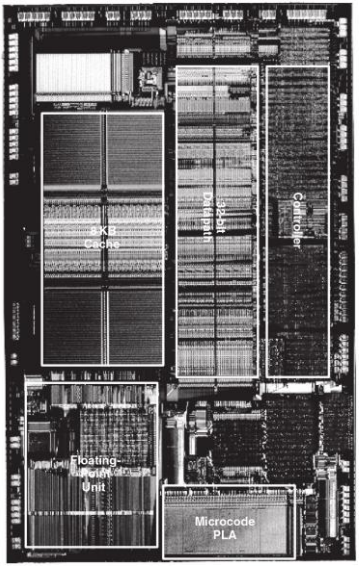


Рис. 7.75 Микропроцессор 80486

Процессор Pentium, изображенный на **Рис. 7.76**, использовал суперскалярную архитектуру и мог запускать на выполнение две команды одновременно. Intel использовала название Pentium вместо 80586, так как компания AMD стала серьезным конкурентом, продавая аналоги процессоров 80486. Intel хотела зарегистрировать название «586» в качестве торговой марки, чтобы никто больше не смог заниматься производством процессоров с таким названием, но оказалось, что зарегистрировать цифры в маркировке микросхемы в качестве торговой марки нельзя, поэтому было принято решение назвать новые процессоры «Pentium». Pentium использовал раздельные кэши команд и данных. Он также использовал предсказание переходов для уменьшения потерь производительности из-за команд условного перехода.

Процессоры Pentium Pro, Pentium II и Pentium III имели общую микроархитектуру под кодовым названием P6, обеспечивавшую внеочередное выполнение команд. Сложные команды из системы команд x86 разбивались на одну или несколько микроопераций (micro-ops), похожих на команды MIPS. Микрооперации затем выполнялись на быстром вычислительном ядре, имевшем 11-стадийный конвейер и обеспечивавшем внеочередное выполнение команд. На **Рис. 7.78** изображен Pentium III. 32-битный тракт данных назван целочисленным функциональным блоком (integer execution unit, IEU). Тракт данных для

операций над числами с плавающей запятой назван блоком вычислений с плавающей запятой (floating point unit, FPU).

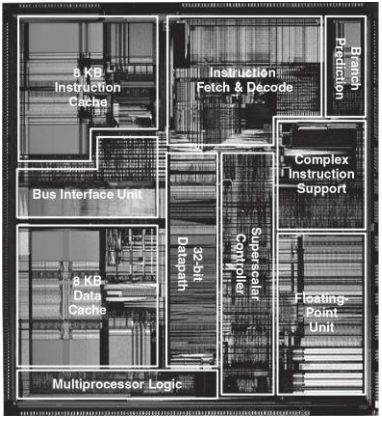


Рис. 7.76 Микропроцессор Pentium

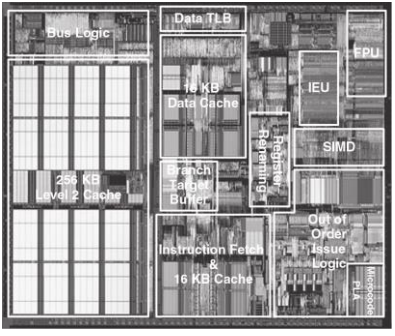


Рис. 7.77 Микропроцессор Pentium III

В процессоре также имелся функциональный блок SIMD для пакетной арифметики с целыми числами и числами с плавающей запятой. Площадь кристалла, потраченная на организацию внеочередного

запуска команд на выполнение, оказалась больше, чем площадь, потраченная на само выполнение команд. Кэши команд и данных были увеличены до 16 Кб каждый. На кристалл Pentium III также была добавлена более большая, но в то же время более медленная кэш-память второго уровня размером 256 Кб.

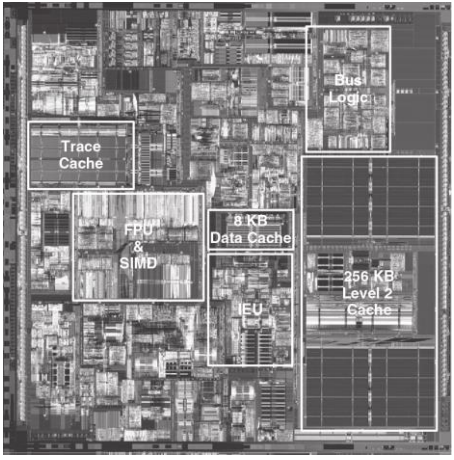


Рис. 7.78 **Микропроцессор Pentium 4**

К концу 1990-х годов процессоры позиционировались на рынке в зависимости от их тактовой частоты. Pentium 4 был очередным процессором со внеочередным выполнением команд, но использовал конвейер очень большой длины для достижения экстремально высоких тактовых частот. Первоначально конвейер имел 20 стадий, а в более поздних версиях – 31 стадию, что позволило достичь частоты, превышающей 3 ГГц. Кристалл, изображенный на **Рис. 7.79**, содержит от 42 до 178 млн. транзисторов (в зависимости от размера кэш-памяти), поэтому на фотографии трудно разглядеть даже основные функциональные блоки. Дешифрация трех команд из набора команд x86 за один такт оказалась невозможной на такой высокой тактовой частоте, поскольку коды операций команд сложны, а сами команды имеют переменную длину. Вместо этого процессор проводил предварительное разбиение команд на более простые микрооперации и сохранял их в *кэш последовательностей микроопераций (trace cache)*. Более поздние версии Pentium 4 также использовали многопоточность для увеличения пропускной способности процессора при выполнениее нескольких потоков.

Использование в Pentium 4 длинных конвейеров и высокой тактовой частоты привело к чрезвычайно высокому энергопотреблению, иногда превышающем 100 Вт. Это оказалось просто недопустимо для

ноутбуков; для обычных ПК это привело к удорожанию систем охлаждения.

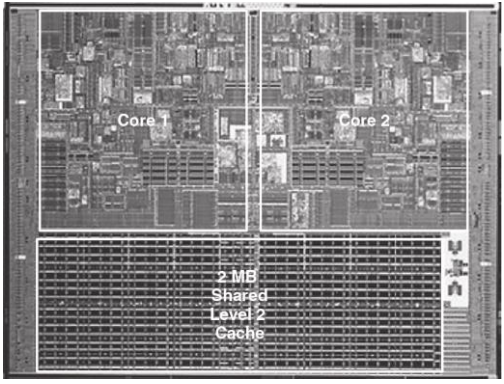


Рис. 7.79 Микропроцессор Core Duo

Intel обнаружила, что предыдущая микроархитектура P6 могла достичь сопоставимой производительности при гораздо более низкой тактовой частоте и мощности. Процессор Pentium M использовал усовершенствованную версию этой микроархитектуры с внеочередным выполнением команд, в которой размер кэш-памяти команд и данных

был увеличен до 32 Кбайт, а размер кэш-памяти второго уровня составлял от одного до двух Мбайт.

Процессор Core Duo представлял собой многоядерный процессор, содержащий два ядра Pentium M, подключенных к общей кэш-памяти второго уровня размером 2 МБ. На **Рис. 7.80** трудно различить отдельные функциональные блоки, но два ядра и большой кэш отчетливо видны.

In В 2009 году Intel представила новую микроархитектуру под кодовым названием Nehalem, которая является модернизированной версией микроархитектуры Core.

Эти процессоры, включая Core i3, i5 и i7, имеют расширенный набор команд с поддержкой 64-битных команд. У них может быть от двух до шести ядер, кэш-память третьего уровня размером 3–15 МБ и встроенный контроллер памяти. В некоторые модели встроены графические процессоры, называемые также графическими ускорителями. Часть моделей поддерживает технологию TurboBoost, позволяющую повысить производительность однопоточного кода путем отключения неиспользуемых процессорных ядер и повышения напряжения и тактовой частоты активного ядра. Некоторые модели поддерживают гиперпоточность (hyperthreading) – так Intel называет многопоточность с двумя активными потоками, которая, с точки зрения

пользователя, удваивает количество ядер. На **Рис. 7.80** изображен кристалл Core i7 с четырьмя ядрами и 8 МБ общей кэш-памяти третьего уровня (L3).

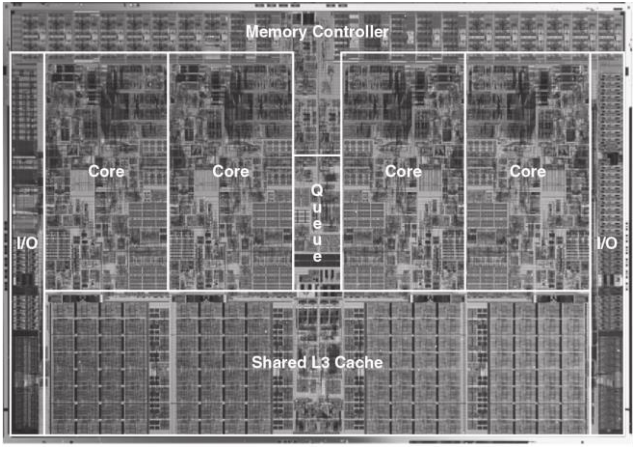


Рис. 7.80 **Микропроцессор Core i7**
(источник: http://www.intel.com/pressroom/archive/releases/2008/20081117comp_sm.htm, приведено с разрешения Intel)

7.10 РЕЗЮМЕ

В этой главе мы рассмотрели три способа построения процессоров MIPS, отличающихся разным соотношением цены и стоимости. Мы считаем это сродни магии – как может столь сложное на вид устройство, как микропроцессор, на самом деле быть столь простым, что его схема занимает всего полстраницы? Более того, принцип его работы, такой таинственный для непосвященных, на поверку оказывается довольно очевидным.

Чтобы собрать микроархитектурный пазл, нам понадобились знания почти изо всех разделов, рассмотренных в этой книге. Мы разрабатывали комбинационные и последовательностные схемы так, как было описано в **главе 2** и **главе 3**, применяли строительные блоки, рассмотренные в **главе 5**. Мы воплощали в жизнь архитектуру MIPS, описанную в **главе 6**. Используя методы, описанные в **главе 4**, мы смогли описать микроархитектуру всего на нескольких страницах HDL-кода.

Разработка различных вариантов микроархитектур также потребовала от нас применения принципов управления сложностью. Микроархитектурная абстракция образует связь между логическим и архитектурным уровнями абстракции и представляет собой истинную суть этой книги о проектировании цифровых систем и компьютерной

архитектуре. Мы также использовали абстракции уровня блочных диаграмм и языков описания аппаратуры, чтобы в сжатой форме описывать взаимное расположение компонентов. При разработке микроархитектур мы широко применяли принципы повторяемости и модульности, повторно используя библиотеки часто используемых строительных блоков, таких как АЛУ, блоки памяти, мультиплексоры и регистры. Мы активно использовали иерархическую организацию, разделив микроархитектуру на тракт данных и устройство управления, которые мы создали из функциональных блоков, а те, в свою очередь – из логических элементов, а логические элементы – из транзисторов, как было описано в первых пяти главах.

В этой главе мы сравнили однотактную, многотактную и конвейерную микроархитектуры процессора MIPS. Все они реализуют одно и то же подмножество набора команд MIPS и имеют одинкаовое архитектурное состояние. Однотактный процессор наиболее прост, а каждая его команда выполняется за один такт, т.е. его CPI равен 1.

Многотактный процессор выполняет команды за переменное число более коротких этапов. Таким образом он может оспользовать одно единственное АЛУ вместо нескольких сумматоров. Однако ему требуется несколько неархитектурных регистров, чтобы хранить промежуточные результаты вычислений между этапами. В теории многотактный процессор мог бы быть быстрее, так как не все команды

выполняются за одинаковое время. На практике, однако, он обычно медленнее, так как длительность его такта ограничена длительностью самого медленного этапа, на которую, в свою очередь, негативно влияют накладные расходы, связанные с использованием неархитектурных регистров.

Конвейерный процессор разделяет однотактный процессор на пять относительно быстрых стадий. Для этого между его стадиями добавляют временные регистры, что позволяет изолировать друг от друга пять одновременно выполняющихся команд. В идеальном мире его CPI был бы равен 1, но конфликты в конвейере приводят к необходимости периодически приостанавливать и очищать его, что увеличивает CPI. Логика, необходимая для разрешения конфликтов, также увеличивает сложность процессора. Период тактового сигнала мог бы быть в пять раз меньше, чем у однотактоного процессора, но на практике он далеко не так мал, потому что ограничен скоростью работы самой медленной стадии, а так же накладными расходами из-за добавленных между стадиями регистров. Тем не менее, конвейерная обработка обеспечивает существенное увеличение производительности, поэтому она используется во всех современных высокопроизводительных микропроцессорах.

Хотя микроархитектуры, рассмотренные в этой главе, реализуют только ограниченное подмножество аржитектуры MIPS, мы показали,

что добавление новых команд требует внесения весьма простых и понятных изменений в тракт данных и устройство управления. Поддержка исключений также требует лишь незначительных изменений.

Существенным ограничением этой главы является то, что мы считали подсистему памяти идеальной, обеспечивающей быстрый доступ и способность хранить всю программу и все данные целиком. В реальности же большая и быстрая память чрезмерно дорога. В следующей главе мы покажем, как получить большинство преимуществ, характерных для большой и быстрой памяти, имея только небольшую, но быструю память, хранящую лишь самую часто используемую информацию, а также медленную, но большую память, хранящую все остальное.

УПРАЖНЕНИЯ

Упражнение 7.1 Предположим, что один из перечисленных ниже управляющих сигналов в однотактном процессоре MIPS неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему?

a) *RegWrite*

b) $ALUOp_1$

c) *MemWrite*

Упражнение 7.2 Повторите **упражнение 7.1** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 7.3 Модифицируйте однотактовый процессор MIPS так, чтобы он поддерживал одну из перечисленных ниже команд. Описание команд вы можете найти ниже, см. **Приложение B**. Сделайте копию **Рис. 7.11** и отметьте необходимые изменения в тракте данных. Назовите новые управляющие сигналы. Сделайте копию **Табл. 7.8** и покажите все необходимые изменения в основном дешифраторе. Опишите любые другие необходимые изменения.

a) `sll`

b) `lui`

c) `slti`

d) `blez`

Табл. 7.8 Таблица истинности основного дешифратора

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Упражнение 7.4 Повторите **упражнение 7.3** для следующих команд.

a) `jal`

b) `lh`

c) `jr`

d) `srl`

Упражнение 7.5 Во многих процессорных архитектурах есть команда загрузки с приращением адреса (load with postincrement), которая обновляет индексный регистр таким образом, чтобы после завершения загрузки он указывал на

следующее слово в памяти. Ассемблерная инструкция `lwinc $rt, imm($rs)` эквивалентна следующим двум:

```
lw $rt, imm($rs)
addi $rs, $rs, 4
```

Повторите **упражнение 7.3** для команды lwinc. Возможно ли добавить эту команду без каких-либо изменений в регистровом файле?

Упражнение 7.6 Добавьте в однотактовый процессор MIPS блок вычислений с плавающей точкой одинарной точности, выполняющий команды `add.s`, `sub.s` и `mul.s`. Считайте, что сумматор и умножитель чисел с плавающей точкой у вас есть. Объясните, какие изменения нужно внести в тракт данных и в устройство управления.

Упражнение 7.7 Ваша подруга – гуру схемотехники – предложила переделать один из блоков однотактового процессора MIPS так, чтобы задержка этого блока уменьшилась вдвое. Используя значения задержек из **Табл. 7.6**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Какова в этом случае будет длительность такта процессора?

Упражнение 7.8 Взгляните на задержки в **Табл. 7.6**. Бен Битдиддл разработал префиксный сумматор, уменьшающий задержку АЛУ на 20 пс. Считая, что все остальные задержки остаются неизменными, определите новую длительность

такта процессора MIPS. Определите, сколько времени займет выполнение тестовой программы, содержащей 100 миллиардов команд.

Упражнение 7.9 Предположим, что один из перечисленных ниже управляющих сигналов в многотактном процессоре MIPS неисправен и постоянно равен нулю, даже когда должен быть равен единице (stuck-at-0 fault). Какие команды перестанут корректно работать? Почему?

a) *MemtoReg*

b) $ALUOp_0$

c) *PCSrc*

Упражнение 7.10 Повторите **упражнение 7.9** для случая, когда неисправный сигнал постоянно равен единице (stuck-at-1 fault).

Упражнение 7.11 Измените HDL-код однотактового процессора MIPS, приведенный в **разделе 7.6.1**, добавив поддержку одной из команд из **упражнения 7.3**. Расширьте тестовое окружение, приведенное в **разделе 7.6.3**, чтобы убедиться, что новая команда работает корректно.

Упражнение 7.12 Повторите **упражнение 7.11** для команд из **упражнения 7.4**.

Упражнение 7.13 Модифицируйте многотактовый процессор MIPS так, чтобы он поддерживал одну из перечисленных ниже команд. Описание команд вы можете найти в **Приложение B**. Сделайте копию **Рис. 7.27** и отметьте необходимые изменения в тракте данных. Назовите новые управляющие сигналы. Сделайте копию **Рис. 7.39** и покажите все необходимые изменения в управляющем автомате. Опишите любые другие необходимые изменения.

a) `srlv`

b) `ori`

c) `xori`

d) `jr`

Упражнение 7.14 Повторите **упражнение 7.13** для следующих команд.

a) `bne`

b) `lb`

c) `lbu`

d) `andi`

Упражнение 7.15 Повторите **упражнение 7.5** для многотакного процессора MIPS. Опишите все необходимые изменения в многотактном тракте данных и

управляющем автомате. Возможно ли добавить эту команду без каких-либо изменений в регистровом файле?

Упражнение 7.16 Повторите **упражнение 7.6** для многотактного процессора MIPS.

Упражнение 7.17 Предположим, что сумматор и умножитель чисел с плавающей точкой из **упражнения 7.16** выполняют операции за два такта. Другими словами, если операнды подаются на их входы в начале первого такта, то результат появляется на выходе только на втором такте. Как в этом случае изменится ваше решение **упражнения 7.16**?

Упражнение 7.18 Ваша подруга – гуру схемотехники – предложила переделать один из блоков многотактового процессора MIPS так, чтобы задержка этого блока существенно уменьшилась. Используя значения задержек из **Табл. 7.6**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Учтите, что попытки сделать его быстрее, чем необходимо – напрасная трата времени вашей подруги. Какова будет длительность такта процессора?

Упражнение 7.19 Повторите **упражнение 7.8** для многотактного процессора. Считайте, что процентное соотношение разных типов команд такое же, как в **примере 7.7**.

Упражнение 7.20 Предположим, что задержки компонентов многотактного процессора MIPS такие же, как в **Табл. 7.6**. Алиса П. Хакер разработала новый регистровый файл, который потребляет на 40% меньше электроэнергии, но при этом работает в два раза медленнее. Стоит ли ей для своего многотактного процессора выбрать более медленный, но меньше потребляющий регистровый файл?

Упражнение 7.21 Корпорация «Голиаф» заявила, что запатентовала трехпортовый регистровый файл. Вместо того, чтобы судиться с ней, Бен Битдиддл разработал новый регистровый файл, у которого всего один порт чтения/записи (как у объединенной памяти команд и данных). Переделайте многотактовый тракт данных и устройство управления так, чтобы они использовали новый регистровый файл.

Упражнение 7.22 Чему равно CPI переделанного многотактного процессора MIPS из **упражнения 7.21**? Считайте, что процентное соотношение разных типов команд такое же, как в **примере 7.7**.

Упражнение 7.23 Сколько тактов потребуется, чтобы выполнить следующую программу на многотактном процессоре MIPS? Чему равно CPI для этой программы?

```
      addi $s0, $0, done  # result = 5

   while:
```

```
    beq $s0, $0, done   # if result > 0, execute while block
    addi $s0, $s0, -1   # while block: result = result-1
    j    while

done:
```

Упражнение 7.24 Повторите **упражнение 7.23** для следующей программы.

```
  add   $s0, $0, $0     # i = 0
  add   $s1, $0, $0     # sum = 0
  addi  $t0, $0, 10     # $t0 = 10

loop:
  slt   $t1, $s0, $t0   # if (i Б 10), $t1 = 1, else $t1 = 0
  beq   $t1, $0,  done  # if $t1 == 0 (i Ю = 10), branch to done
  add   $s1, $s1, $s0   # sum = sum + i
  addi  $s0, $s0, 1     # increment i
  j loop
done:
```

Упражнение 7.25 Напишите HDL-код многотактного процессора MIPS. Процессор должен быть совместим с модулем верхнего уровня, приведенным ниже. Модуль mem используется для хранения и команд, и данных. Протестируйте ваш процессор, используя тестовое окружение из **раздела 7.6.3**.

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, adr,
           output logic        memwrite);
```

```
  logic [31:0] readdata;
  // instantiate processor and memories
  mips mips(clk, reset, adr, writedata, memwrite, readdata);
  mem mem(clk, memwrite, adr, writedata, readdata);
endmodule

module mem(input  logic        clk, we,
           input  logic [31:0] a, wd,
           output logic [31:0] rd);
   logic [31:0] RAM[63:0];
initial
   begin
    $readmemh("memfile.dat", RAM);
   end
  assign rd = RAM[a[31:2]]; // word aligned
  always @(posedge clk)
    if (we)
     RAM[a[31:2]] <= wd;
Endmodule
```

Упражнение 7.26 Добавьте в HDL-код многотактного процессора MIPS из **упражнения 7.25** поддержку одной из новых команд из **упражнения 7.13**. Расширьте тестовое окружение, чтобы убедиться, что новая команда работает корректно.

Упражнение 7.27 Повторите **упражнение 7.26** для одной из команд из **упражнения 7.14**.

Упражнение 7.28 Конвейерный процессор MIPS выполняет приведенную ниже программу. Какие регистры он читает и в какие регистры пишет на пятом такте?

```
addi $s1, $s2, 5
sub  $t0, $t1, $t2
lw   $t3, 15($s1)
sw   $t5, 72($t0)
or   $t2, $s4, $s5
```

Упражнение 7.29 Повторите **упражнение 7.28** для приведенной ниже программы. Не забудьте, что у конвейерного процессора MIPS есть блок обнаружения и разрешения конфликтов.

```
add $s0, $t0, $t1
sub $s1, $t2, $t3
and $s2, $s0, $s1
or  $s3, $t4, $t5
slt $s4, $s2, $s3
```

Упражнение 7.30 Используя такую же диаграмму, как на **Рис. 7.52**, покажите пересылки через байпас и приостановки конвейера, возникающие при выполнении следующих инструкций в конвейерном процессоре MIPS.

```
add $t0, $s0, $s1
sub $t0, $t0, $s2
Iw  $t1, 60($t0)
and $t2, $t1, $t0
```

Упражнение 7.31 Повторите **упражнение 7.30** для следующих команд.

```
add $t0, $s0, $s1
Iw  $t1, 60($s2)
sub $t2, $t0, $s3
and $t3, $t1, $t0
```

Упражнение 7.32 Сколько тактов потребуется конвейерному процессору MIPS, чтобы запустить на выполнение все команды программы из **упражнения 7.23**? Чему равно CPI процессора для этой программы?

Упражнение 7.33 Повторите **упражнение 7.32** для программы из **упражнения 7.24**.

Упражнение 7.34 Объясните, как добавить в конвейерный процессор MIPS поддержку команды addi.

Упражнение 7.35 Объясните, как добавить в конвейерный процессор MIPS поддержку команды j. Особое внимание уделите очистке конвейера после выполнения безусловного перехода.

Упражнение 7.36 В **примерах 7.9** и **7.10** показано, что производительность конвейерного процессора MIPS могла бы быть выше, если бы вычисление условия перехода выполнялось бы в стадии Execute, а не в стадии Decode. Объясните, как модифицировать конвейерный процессор, показанный

на **Рис. 7.58**, чтобы вычисление условия перехода выполнялось в стадии Execute. Как изменятся сигналы приостановки и очистки конвейера? Вычислите новое значение CPI, длительность такта процессора и общее время выполнения программы в **примерах 7.9** и **7.10**.

Упражнение 7.37 Ваша подруга – гуру схемотехники – предложила переделать один из блоков конвейерного процессора MIPS так, чтобы задержка этого блока существенно уменьшилась. Используя значения задержек из **Табл. 7.6** и **примера 7.10**, определите, какой блок ей стоит улучшить, чтобы эффект, оказанный на производительность процессора, оказался наибольшим. Как быстро должен работать новый блок? Учтите, что попытки сделать его быстрее, чем необходимо – напрасная трата времени вашей подруги. Какова будет длительность такта процессора?

Упражнение 7.38 Взгляните на задержки в **Табл. 7.6** и **примере 7.10**. Изменилась бы длительность такта конвейерного процессора MIPS, если бы АЛУ работало на 20% быстрее? А на 20% медленнее?

Упражнение 7.39 Предположим, что конвейерный процессор MIPS поделен на 10 стадий длительностью 400 пс каждая, включая накладные все накладные расходы на конвейеризацию. Будем считать, что процентное соотношение разных типов команд такое же, как в **примере 7.7**, при этом в половине случаев результат команд загрузки требуется немедленно, что приводит к приостановке конвейера на шесть тактов. Также будем считать, что 30% условных переходов предсказываются неверно, а адрес перехода для команд условного и

безусловного перехода вычисляется в конце второго такта. Вычислите для этого десятистадийного процессора среднее значение CPI и время выполнения 100 миллиардов команд из тестового набора SPECINT2000.

Упражнение 7.40 Напишите HDL-код конвейерного процессора MIPS. Процессор должен быть совместим с модулем верхнего уровня из **примера HDL 7.13**. Он должен поддерживать все команды, рассмотренные в этой главе, включая `addi` и `j` (см. **упражнения 7.34** и **7.35**). Протестируйте ваш процессор, используя тестовое окружение из **примера HDL-кода 7.12**.

Упражнение 7.41 Разработайте для конвейерного процессора MIPS блок разрешения конфликтов, показанный на **Рис. 7.58**. Используйте язык описания аппаратуры. Изобразите в общих чертах схему, которую мог бы сгенерировать из вашего кода HDL-синтезатор.

Упражнение 7.42 Предположим, что сигнал немаскируемого прерывания (nonmaskable interrupt, NMI) подается на один из входов процессора. Когда сигнал на этом входе равен единице, текущая команда должна завершится, после чего процессор должен установить в регистр причины исключения значение 0 и перейти к обработчику исключения. Объясните, какие изменения нужно внести в многотактовый процессор, показанный на **Рис. 7.63** и **Рис. 7.64**, чтобы он мог обрабатывать немаскируемые прерывания.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

Ниже приведены некоторые из вопросов, которые задают на собеседованиях на вакансии разработчиков цифровых устройств.

Вопрос 7.1 Объясните преимущества конвейерных микропроцессоров.

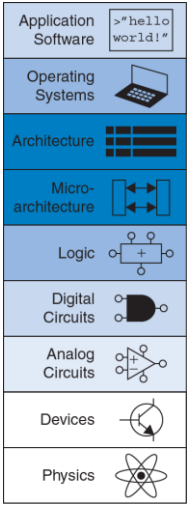
Вопрос 7.2 Если большее количество стадий конвейера позволяет процессору работать быстрее, почему нет процессоров с сотней стадий?

Вопрос 7.3 Объясните, что такое конфликты в микропроцессоре и пути их разрешения. Какие преимущества и недостатки у каждого из способов?

Вопрос 7.4 Расскажите, что такое суперскалярный процессор, каковы его достоинства и недостатки.

Иерархия памяти и подсистема ввода-вывода

8



8.1 ВВЕДЕНИЕ

Способность компьютера решать задачи сильно зависит от его подсистемы памяти и устройств ввода-вывода, таких как мониторы, клавиатуры и принтеры – того, что позволяет нам управлять компьютером и видеть результаты этих вычислений. В этой главе речь пойдет об оперативной памяти и подсистемах ввода-вывода.

Производительность компьютерной системы зависит от ее подсистемы памяти так же сильно, как и от микроархитектуры процессора. В **главе 7** мы считали, что память идеальна и к ней можно обратиться всего за один такт. Однако это было бы правдой только для очень маленькой памяти (или для очень медленного процессора)! Ранние процессоры были сравнительно медленные, так что память была в состоянии справляться с нагрузкой. Но скорость процессоров росла быстрее, чем скорость памяти. В настоящее время оперативаня память типа DRAM (Dynamic Random Access Memory, динамическая память с произвольным доступом) медленнее процессора от 10 до 100 раз. Увеличивающийся разрыв в скорости между процессором и оперативной памятью требует все более и более изощренных подсистем памяти, чтобы попытаться приблизить скорость работы памяти к скорости процессора. Первая половина этой главы рассказывает о подсистемах памяти и анализирует различные компромиссы между их скоростью, емкостью и ценой.

Процессор работает с памятью через интерфейс памяти (memory interface). **Рис. 8.1** показывает простой интерфейс к памяти, использованный в нашем многотактном MIPS процессоре. Процессор выставляет адреса на шину адреса (address bus), идущую к подсистеме памяти. Для чтения управляющий сигнал записи (MemWrite) устанавливается в 0, а память возвращает данные по шине чтения данных (ReadData). Для записи MemWrite устанавливается в 1 и процессор посылает данные в память по шине записи данных (WriteData).

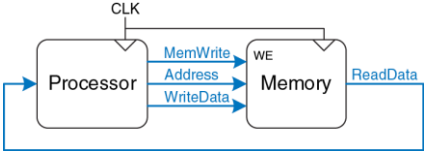


Рис. 8.1 **Интерфейс к памяти**

Основные проблемы при разработке подсистемы памяти можно описать, рассматривая в качестве метафоры книги в библиотеке. В библиотеке на полках есть много книг. Если вы собрались написать научную работу по толкованию снов, вы можете пойти в библиотеку[14],

[14] Мы понимаем, что в век Интернета использование библиотек среди учащихся стремительно падает. Однако мы верим, что в библиотеках хранятся огромные богатства

взять с полки книгу Фрейда «Толкование сновидений» и принести ее к себе в кабинет. После того, как вы ее просмотрите, вы можете вернуть ее обратно и взять работу Юнга «Психология подсознания». Затем вы могли бы пойти обратно за «Толкованием сновидений», чтобы использовать еще одну цитату из нее. А потом опять пойти в библиотеку за книгой Фрейда «Я и Оно». Очень скоро вы устанете бегать в библиотеку и, если вы умный, то вы будете просто держать нужные книги в своем кабинете вместо того, чтобы бегать за ними взад-вперед. Более того, когда вы возьмете книгу Фрейда, то можете взять еще несколько его книг с той же полки (на всякий случай).

Эта метафора подчеркивает принцип, изложенный в **разделе 6.2.1** – «типичный сценарий должен быть быстрым». Оставляя в своем кабинете книги, которые вы только что использовали или которые вы, может быть, будете скоро использовать, вы уменьшаете количество отнимающих много времени походов в библиотеку. В частности, вы используете принципы временно́й (англ.: temporal) и пространственной (англ.: spatial) локальности. Временная локальность означает, что если вы только что использовали книгу, то, вероятно, она вам снова скоро понадобится. Пространственная локальность означает, что когда вам

знаний, которые достались человечеству тяжелым трудом, и не все они доступны в электронном виде. Мы надеемся, что искусство поиска знаний в книгах не будет полностью вытеснено запросами во всемирной паутине.

понадобилась определенная книга, то, вероятно, вас заинтересуют и другие книги с той же полки.

Библиотека сама старается ускорить типичный сценарий, используя принцип локальности. У нее нет ни места на полках, ни денег, чтобы собрать все книги в мире. Вместо этого, она хранит редко используемые книги в подвале. Также она использует систему межбиблиотечного обмена с соседними библиотеками, так что она может предложить вам больше книг, чем физически имеет в наличии.

В итоге вы получаете выгоду как от большой коллекции книг, так и от быстрого доступа к наиболее популярным книгам, используя иерархию хранения книг. Наиболее часто используемые книги находятся у вас на столе. Бóльшая коллекция – на полках вашей библиотеки. Еще большая коллекция доступна из хранилища и из других библиотек. Точно так же подсистемы памяти используют иерархию хранилищ для быстрого доступа к наиболее часто используемым

данным, одновременно обеспечивая возможность хранения больших объемов данных.

Подсистемы памяти, используемые для создания такой иерархии, были описаны в **разделе 5.5**. Компьютерная память в основном построена на базе динамической (DRAM) и статической (SRAM) памяти. В идеале память должна быть быстрой, большой и дешёвой. Однако на практике любой тип памяти имеет только два из этих свойств; память либо медленная, либо дорогая, либо маленького объема. Несмотря на это, компьютерные системы могут приближаться к этому идеалу, сочетая дешёвую, быструю и маленькую память с дешевой, медленной и большой. Быстрая память используется для хранения часто используемых данных и команд, так что создается впечатление, что подсистема памяти всегда работает довольно быстро. Остальные данные и команды хранятся в большой памяти, которая работает медленнее, но позволяет иметь большой общий объем памяти. Комбинация двух дешёвых типов памяти – это намного менее дорогой вариант, чем одна большая и быстрая память. Этот принцип распространяется на всю иерархию памяти, так как с увеличением объема памяти уменьшается ее скорость работы.

Оперативная память компьютера обычно строится на микросхемах динамической памяти (DRAM). В 2012 году типичный персональный компьютер имел оперативную память (main memory) размером от 4 до

8 Гбайт, и эта DRAM-память стоила около десяти долларов за гигабайт. Цены на DRAM падали в среднем на 25% в год на протяжении последних трех десятилетий, при этом емкость памяти росла примерно с такой же скоростью, так что общая цена памяти в персональном компьютере оставалась приблизительно одинаковой. К сожалению, скорость работы самих микросхем DRAM возрастала только на 7% в год, в то время как производительность процессоров возрастала на 25–50% в год. На **Рис. 8.2** показан график увеличения скорости работы оперативной памяти и процессоров с 1980 года по настоящее время. В начале 1980-х годов скорость процессоров и памяти была примерно одинаковой, но затем разрыв в производительности сильно увеличился и память серьезно отстала.[15]

Память DRAM могла успешно идти в ногу с процессорами в 1970-х и в начале 1980-х годов, но сейчас она чудовищно медленная. Время доступа к данным в DRAM на порядок или два медленнее, чем длительность такта процессора (десятки наносекунд против долей наносекунды).

[15] Хотя производительность отдельных ядер в процессорах остается примерно одинаковой где-то с 2005 года, как показано на **Рис. 8.2**, но переход на многоядерные системы (на рисунке не показан) только усугубляет разрыв в производительности процессоров и памяти.

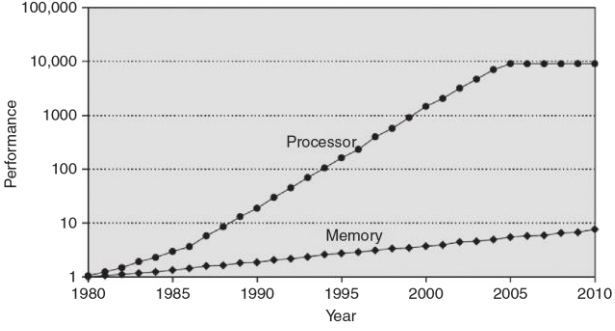


Рис. 8.2 **Разрыв в производительности процессоров и памяти.**
График из книги Hennessy and Patterson, Computer Architecture:
A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012, с разрешения авторов

Чтобы справиться с этой проблемой, компьютеры хранят наиболее часто используемые команды и данные в быстрой, но небольшой по объему памяти, называемой кэш-памятью или просто кэшем (англ.: cache). Кэш-память обычно сделана с использованием статической памяти (SRAM) и находится в той же микросхеме, что и процессор. Скорость кэша сравнима со скоростью процессора, так как, во-первых, память SRAM работает быстрее, чем DRAM, а во-вторых, она находится на кристалле процессора и позволяет избавиться от

задержек распространения сигналов по пути к внешним микросхемам памяти. В 2012 году стоимость SRAM, расположенной кристалле процессора, составляла порядка 10 тысяч долларов за Гбайт, но так как размер кэша сравнительно мал (от нескольких Кбайт до нескольких Мбайт), то общая стоимость кэша не такая большая. Кэш-память может хранить как данные, так и команды, но для краткости мы будем говорить, что она содержит просто «данные».

Если процессор запрашивает данные, которые уже находятся в кэше, то он получает их очень быстро. Это называется попаданием в кэш (англ.: cache hit). В противном случае процессор вынужден читать данные из оперативной памяти (DRAM). Это называется промахом кэша, кэш-промахом или промахом доступа в кэш (англ.: cache miss). Если процессор попадает в кэш бóльшую часть времени, то он редко простаивает в ожидании доступа к медленной оперативной памяти, и среднее время доступа мало.

Третий уровень в иерархии памяти – жесткий диск (примечание переводчика: «жестким» диск называли для отличия его от «гибких» дискет, популярных с середины 1970-х до 2000-х годов). Тем же самым способом, как библиотека использует подвал для хранения книг, не умещающихся на полках, компьютерные системы используют жесткий диск для хранения данных, которые не помещаются в оперативной памяти. В 2012 году жесткий диск, сделанный на основе технологии

магнитной записи (Hard Disk Drive, HDD), стоил менее десяти центов за Гбайт и имел время доступа около 10 миллисекунд. Цены на жесткие диски падают на 60% в год, но время доступа почти не улучшается. Твердотельные диски (Solid State Drive, SSD), которые сделаны на основе флэш-памяти, становятся все более популярной альтернативой HDD. SSD использовались для специальных применений на протяжении двух десятилетий, но вышли на потребительский рынок только в 2007 году. Они не подвержены механическим отказам, но и стоят в десять раз больше, чем HDD – порядка одного доллара за Гбайт.

Жесткий диск обеспечивает иллюзию наличия бóльшего объема памяти, чем реально доступно в оперативной памяти. Это называется виртуальной памятью. Как и доступ к книгам в хранилище, доступ к данным в виртуальной памяти занимает длительное время. Оперативная память, также называемая физической памятью, содержит только часть данных, находящихся в виртуальной памяти, остальные данные находятся на жестком диске. Следовательно, оперативная память может рассматриваться как кэш-память для наиболее часто используемых данных с жесткого диска.

В этой главе мы рассмотрим иерархию памяти компьютерной системы, показанную на **Рис. 8.3**. Процессор сначала ищет данные в маленькой, но быстрой кэш-памяти, обычно расположенной на той же самой

микросхеме. Если данные в кэше отсутствуют, процессор обращается к оперативной памяти (Main Memory). Если данных нет и там, то процессор читает данные из большого, хотя и медленного, жесткого диска, используя механизм виртуальной памяти. **Рис. 8.4** иллюстрирует соотношение емкости и скорости в многоуровневой иерархии памяти компьютерной системы и показывает типичную стоимость, время доступа и пропускную способность для технологий памяти по состоянию на 2012 год. Как видите, с уменьшением времени доступа скорость возрастает.

В **разделе 8.2** мы покажем, как анализировать производительность систем памяти. В **разделе 8.3** мы рассмотрим несколько методов организации кэш-памяти, а в **разделе 8.4** расскажем о виртуальной памяти. В заключение этой главы мы узнаем, как процессор получает доступ к устройствам ввода-вывода, таким как клавиатура и монитор, тем же способом, как получает доступ к памяти. В **разделе 8.5** изучаются такие отображаемые в адресное пространство устройства ввода-вывода. В **разделе 8.6** будут описаны устройства ввода-вывода для встроенных систем, а в **разделе 8.7** – основные стандарты ввода-вывода для персональных компьютеров.

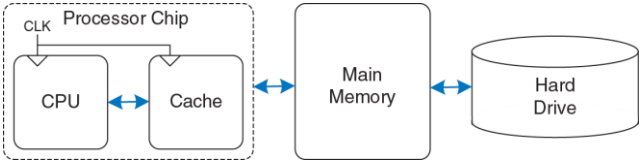
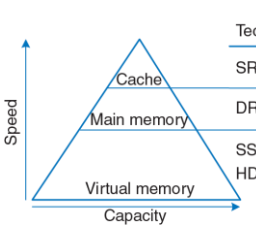


Рис. 8.3 **Типичная иерархия памяти**



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10–50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Рис. 8.4 **Компоненты иерархии памяти и их характеристики на 2012 год**

8.2 АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ СИСТЕМ ПАМЯТИ

Чтобы оценить соотношение цены и производительности у разных вариантов систем памяти, разработчикам (и покупателям) компьютеров нужны количественные способы измерения производительности. Мерами измерения производительности систем памяти являются процент попаданий (hit rate) или промахов (miss rate), а также среднее время доступа. Процент попаданий и промахов вычисляется так:

$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Number of total memory accesses}} = 1 - \text{Hit Rate}$$
$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Number of total memory accesses}} = 1 - \text{Miss Rate} \quad (8.1)$$

Пример 8.1 ВЫЧИСЛЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ КЭШ-ПАМЯТИ

Предположим, что программа имеет 2000 команд обращения к данным (загрузок и сохранений), но только 1250 из этих команд нашли запрошенные ими данные в кэш-памяти. Остальным 750 командам пришлось получать данные из оперативной памяти или с диска. Чему равен процент промахов и попаданий в кэш-память в этом случае?

Решение: Процент промахов находится как 750/2000 = 0.375 = 37.5%. Процент попаданий равен 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

Среднее время доступа (англ.: Average Memory Access Time, AMAT) – это среднее время, которое процессор тратит, ожидая доступа к памяти при выполнении команд загрузки или сохранения данных. В типичной компьютерной системе, показанной на **Рис. 8.3**, процессор сначала ищет данные в кэше. Если данных в кэше нет, то процессор обращается к оперативной памяти. Если же данных нет и там, то процессор выполняет обращение к виртуальной памяти на диске. Следовательно, AMAT вычисляется так:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM}+MR_{MM}t_{VM}) \quad (8.2)$$

где $t$cache, t_{MM}, и t_{VM} – это времена доступа к кэшу, оперативной памяти и диску соответственно, а MRcache и MR_{MM} – это процент промахов кэша и оперативной памяти.

Пример 8.2 ВЫЧИСЛЕНИЕ СРЕДНЕГО ВРЕМЕНИ ДОСТУПА К ПАМЯТИ

Предположим, что компьютерная система имеет память всего с двумя уровнями иерархии: кэшем и оперативной памятью. Чему равно среднее время доступа, если время доступа и процент промахов заданы в **Табл. 8.1**?

Решение: Среднее время доступа будет равно 1 + 0.1(100) = 11 тактов

Табл. 8.1 **Время доступа и процент промахов**

Уровень памяти	Время доступа в тактах	Процент промахов
Кэш-память	1	10%
Оперативная память	100	0%

Пример 8.3 УЛУЧШЕНИЕ ВРЕМЕНИ ДОСТУПА

Среднее время доступа к памяти в 11 тактов означает, что процессор тратит 10 тактов на ожидание данных на каждый такт реального использования этих данных. Какой процент промахов в кэш необходим для уменьшения среднего времени доступа к памяти до 1,5 тактов при заданном в **Табл. 8.1** времени доступа к памяти?

Решение: Обозначим процент промахов в кэш как m, тогда среднее время доступа будет равно 1 + 100m. Вычислим m, приравняв это выражение к 1,5. Ответ: требуемый процент промахов должен быть равен 0.5%.

В качестве предупреждения: улучшение производительности в реальности может быть не таким красивым, как оно выглядит на бумаге. Например, увеличение скорости памяти в десять раз не обязательно сделает компьютерную программу в десять раз быстрее. Если 50% команд в программе – это команды загрузки и сохранения данных, то десятикратное увеличение скорости памяти приведет к ускорению программы всего лишь в 1,82 раза. Этот общий принцип называется

законом Амдала и гласит, что усилия, потраченные на улучшение производительности подсистемы, оправдываются только тогда, когда она оказывает значительное влияние на общую производительность системы.

Джин Амдал, 1922–

Джин Амдал (Gene Amdahl) наиболее известен как автор «закона Амдала» – наблюдения, которое он сделал в 1965 году. Будучи аспирантом, Амдал начал в свободное время разрабатывать компьютеры. Эта работа принесла ему степень доктора философии по теоретической физике (Ph.D., западный аналог степени кандидата наук) в 1952 году. Сразу после окончания аспирантуры Амдал устроился на работу в IBM, а позже основал три компании, одну из которых в 1970 году назвал Amdahl Corporation.

8.3 КЭШ-ПАМЯТЬ

Кэш[16] содержит часто используемые данные из памяти. Число слов данных, которое он может хранить, называется ёмкостью кэша (англ.: capacity). Поскольку ёмкость кэша меньше, чем ёмкость оперативной памяти, то разработчик компьютерной системы должен решить, какое подмножество оперативной памяти хранить в кэше.

Когда процессор пытается получить доступ к данным, он сначала ищет их в кэше. Если данные там есть, то есть произошло попадание в кэш, то процессор получает их немедленно. Если же их там нет, то есть произошел промах кэша, то процессор вычитывает данные из оперативной памяти и помещает их в кэш для последующего использования. Для этого кэш должен заменить, или *заместить*, какие-то старые данные на новые. В этом разделе мы рассмотрим разработку кэшей, попытавшись ответить на следующие вопросы: (1) Какие данные хранятся в кэш-памяти? (2) Как найти данные в кэш-памяти? и (3) Какие данные заместить в кэш-памяти, когда нужно разместить новые данные, а кэш заполнен?

При чтении следующих разделов помните, что ответы на эти вопросы связаны с присущей большинству программ пространственной и

[16] Кэш – потайное место для хранения оружия и продовольствия (словарь Merriam Webster Online Dictionary, 2012, www.merriam-webster.com)

временно́й локальностью при обращении к данным. Эту локальность кэш использует для предсказания того, какие данные понадобятся следующими. Если программа обращается к памяти в случайном порядке, она не получит никакой выгоды от использования кэша (прим. перев.: скорее всего, такая программа будет работать даже медленнее, так как кэш-памяти присущи определенные накладные расходы).

Как мы увидим в следующих разделах, кэш-память характеризуется емкостью *C* (от англ. capacity), числом наборов *S* (от англ. set), длиной строки, иногда называемой размером блока *b* (от англ. block), количеством строк или блоков *B* и степенью ассоциативности *N*.

Хотя мы сфокусируем внимание на чтении из кэша данных, но те же самые принципы применяются и для чтения из кэша команд. Запись в кэш данных похожа на чтение и будет рассмотрена в **разделе 8.3.4** (прим. перев.: во многих архитектурах запись в кэш команд не имеет смысла и потому не реализована).

8.3.1 Какие данные хранятся в кэш-памяти?

Идеальный кэш должен предугадывать, какие данные понадобятся процессору, и выбирать их из оперативной памяти заранее таким образом, чтобы кэш имел нулевой процент промахов. Но поскольку точно предсказать будущее невозможно, то кэш должен угадывать,

какие данные понадобятся, основываясь на предыдущих обращениях в память. В частности, кэш использует временну́ю и пространственную локальность, чтобы уменьшить процент промахов в кэш.

Напомним, что временна́я локальность означает, что процессор, вероятно, еще раз обратится к тем данным, которые он недавно использовал. Поэтому, когда процессор читает или записывает данные, которых нет в кэше, то эти данные копируются из оперативной памяти в кэш, так что последующие обращения к ним уже не вызовут промаха кэша.

Напомним также, что пространственная локальность означает, что когда процессор обращается к каким-либо данным, то, вероятно, ему понадобятся и расположенные рядом данные. Поэтому, когда кэш читает одно слово данных из памяти, он заодно читает и несколько соседних слов. Эта группа слов называется *строкой* кэша (англ.: cache line), также иногда используют термин «блок кэша» (англ.: cache block). Число слов в строке b называется длиной строки (line size или block size). Кэш емкостью *C* содержит $B = C/b$ строк.

Принципы пространственной и временно́й локальности данных были экспериментально подтверждены на реальных программах. Если переменная используется в программе, то та же самая переменная будет, скорее всего, использована снова, тем самым создавая

временную локальность. Если используется какой-либо элемент массива, то, скорее всего, и другие элементы этого массива тоже будут использованы, тем самым создавая пространственную локальность.

8.3.2 Как найти данные в кэш-памяти?

Кэш состоит из *S* наборов, каждый из которых содержит одну или несколько строк (блоков данных). Взаимосвязь между адресом данных в оперативной памяти и расположением этих данных в кэше называется отображением. Каждый адрес памяти всегда отображается в один и тот же набор кэша. Несколько бит адреса используются, чтобы определить, какой именно набор кэша содержит искомые данные. Если в наборе больше одной строки, то данные могут находиться в любой из них.

Кэш-память классифицируется по числу строк в наборе. В *кэше прямого отображения* (англ.: direct mapped cache) каждый набор содержит только одну строку (один блок), так что кэш содержит $S = B$ наборов. Таким образом, каждый из адресов в оперативной памяти отображается в одну-единственную строку кэша. В случае же наборно-ассоциативного кэша с *N* секциями (англ.: *N*-way set associative cache) каждый набор состоит из *N* строк. Каждый адрес памяти по-прежнему отображается в один-единственный набор, но число наборов в этом случае равно $S = B/N$, а данные могут оказаться в любой из *N* строк

этого набора. В отличие от кэша прямого отображения и наборно-ассоциативного кэша, *полностью ассоциативный кэш* (англ.: fully associative cache) имеет только один набор (S=1), и данные могут оказаться в любой из B строк этого набора. Таким образом, полностью ассоциативный кэш – это то же самое, что и наборно-ассоциативный кэш с B секциями (количество секций совпадает с количеством строк во всем кэше).

Для иллюстрации этих вариантов организации кэша мы рассмотрим подсистему памяти процессора MIPS с 32-битными адресами и 32-битными словами. В наших примерах память адресуется побайтово, а каждое слово состоит из четырех байт, так что память содержит 230 слов, выровненных по 4-байтной границе (т.е. находящихся по адресам 0, 4, 8, ...). Для простоты мы будем рассматривать кэши с емкостью C = 8 слов. Мы начнем с длины строки (b), равной одному слову, после чего перейдем к большим по размеру строкам.

Кэш-память прямого отображения

В кэш-памяти прямого отображения каждый набор содержит только одну строку (блок данных), так что у него $S = B$ наборов и строк. Чтобы понять способ отображения адресов памяти в определенные строки такого кэша, представьте, что оперативная память поделена на блоки по b слов так же, как кэш поделен на строки по b слов. Адрес одного из слов, находящихся в блоке 0 оперативной памяти, отображается в

набор 0 кэша. Адрес слова из блока 1 оперативной памяти отображается в набор 1 кэша, и так далее, пока адрес слова из блока $B - 1$ оперативной памяти не отобразится в строку $B - 1$ кэша. Больше строк в кэше нет, так что следующий блок оперативной памяти (блок B) снова отображается строку 0 кэша, и так далее.

Отображение для такого кэша емкостью 8 слов и размером строки в одно слово показано на **Рис. 8.5**. В кэше 8 наборов, каждый из которых содержит по одной строке, длина которых равна одному слову. Младшие два бита адреса всегда равны нулю, потому что все адреса выровнены на границу слова. Следующие log28 = 3 бита адресуют один из восьми наборов, в который будет отображен этот адрес памяти. Таким образом, данные из адресов 0x00000004, 0x00000024, …, 0xFFFFFFE4 будут отображены в один и тот же набор 1, как показано синим цветом. Аналогично, данные из адресов 0x00000010, …, 0xFFFFFFF0 отображаются в набор 4, и так далее. Каждый адрес оперативной памяти отображается строго в один набор кэша.

Пример 8.4 ЧАСТИ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЭШ

В какой набор кэша на **Рис. 8.5** будет отображено слово с адресом 0x00000014? Назовите другой адрес, который отображается в этот же самый набор.

Решение: Два младших бита адреса всегда равны нулю, потому что адрес выровнен по границе слова. Следующие 3 бита равны 101, так что слово будет

отображено в набор 5. Слова с адресами 0x34, 0x54, 0x74, …, 0xFFFFFFF4 тоже будут отображены в этот набор.

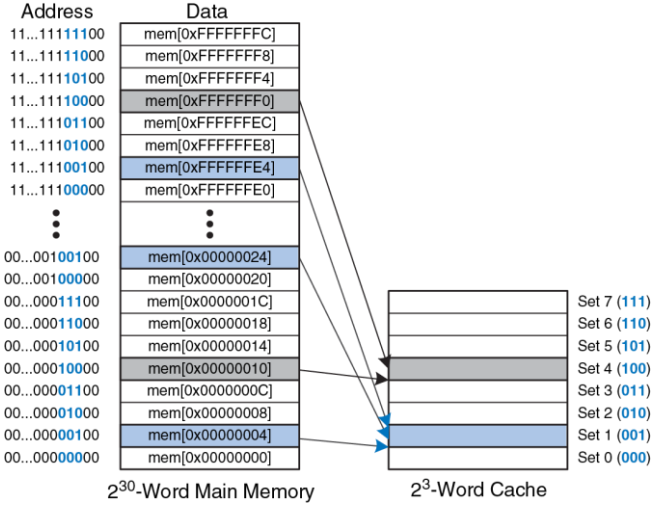


Рис. 8.5 **Отображение оперативной памяти на кэш прямого отображения**

Так как в один набор кэша отображается множество адресов, то кэш должен отслеживать адреса данных, находящихся в каждом из наборов в текущий момент времени. Младшие биты адреса определяют набор, в котором хранятся данные. Оставшиеся биты адреса называются тегом (англ.: tag) и указывают, какой именно из всех возможных адресов сейчас находится в этом наборе.

В наших предыдущих примерах два младших бита адреса называются байтовым смещением (byte offset), поскольку они указывают на номер байта внутри слова. Следующие три бита называются *индексом* (cache index) или *номером набора* (set bits), так как они указывают на номер набора, в который отображается этот адрес (в общем случае номер набора состоит из log2*S* битов, где *S* – число наборов). Оставшиеся 27 бит тега указывают на адрес слова, которое в текущий момент находится в этом наборе кэша. На **Рис. 8.6** показано, на какие части делится адрес 0xFFFFFFE4. Он отображается в набор 1, а его тег содержит одни единицы.

Рис. 8.6 **Части адреса при отображении в кэш**

Пример 8.5 ЧАСТИ АДРЕСА ПРИ ОТОБРАЖЕНИИ В КЭШ

Найти число битов тега и номера набора (индекса) для кэш-памяти прямого отображения с 1024 (2^{10}) наборами и длиной строки, равной одному слову. Размер адреса равен 32 битам.

Решение: для кэш-памяти, у которой 2^{10} наборов, требуются $\log_2(2^{10}) = 10$ битов для хранения номера набора (индекса). Два младших бита адреса хранят байтовое смещение, а оставшиеся 32 − 10 – 2 = 20 бит используются для тега.

Иногда, особенно когда компьютер только включили, наборы кэша еще не содержат никаких данных. Для каждого набора в кэше есть бит достоверности (*valid bit*), который равен единице, если в нем находятся корректные данные, и нулю, если находящееся в нем значение бессмысленно.

На **Рис. 8.7** изображена блок-схема аппаратной реализации кэша прямого отображения, показанного на **Рис. 8.5**. Кэш использует блок статической памяти SRAM с восемью ячейками. Каждая ячейка, или набор (Set), содержит 32 бита данных (Data), 27 битов тега (Tag) и 1 бит достоверности (V). К кэшу обращаются, используя 32-битные адреса. Два младших бита адреса – байтовое смещение – игнорируются при обращении к словам. Следующие 3 бита указывают на ячейку, или набор, кэш-памяти. Команда загрузки читает эту ячейку из кэш-памяти и проверяет тег и бит достоверности. Если тег совпадает со старшими

27 битами адреса и бит достоверности равен 1, то происходит попадание в кэш (Hit) и данные передаются процессору. В противном случае происходит промах кэша и подсистема памяти должна прочитать запрошенные данные из оперативной памяти.

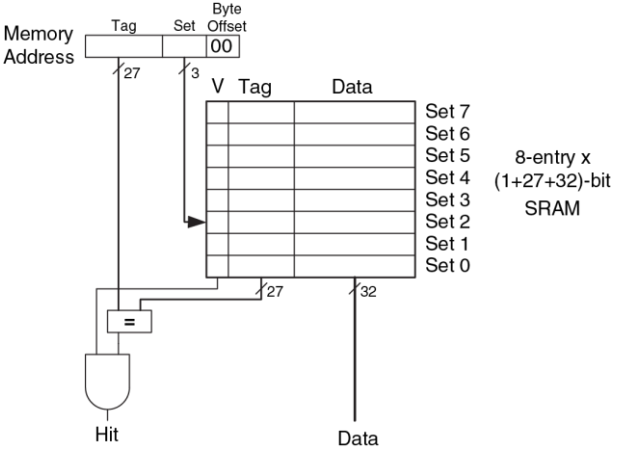


Рис. 8.7 **Кэш прямого отображения с восемью наборами**

Пример 8.6 ВРЕМЕННАЯ ЛОКАЛЬНОСТЬ С КЭШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Циклы – это типичный источник временной и пространственной локальности данных в приложениях. Используя кэш с восемью ячейками, изображенный на **Рис. 8.7**, покажите, чему будет равно содержимое кэша после выполнения следующего небольшого цикла на языке ассемблера MIPS. Считайте, что изначально кэш пуст. Каким будет процент промахов?

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Решение: Эта программа содержит цикл, повторяющийся пять раз. Каждая итерация содержит три обращения в память (три инструкции загрузки lw), всего 15 обращений. Когда цикл выполняется в первый раз, кэш пуст и данные, расположенные в оперативной памяти по адресам 0x4, 0xC и 0x8, должны быть загружены в наборы кэша 1, 3 и 2 соответственно. Однако в следующих четырех итерациях цикла данные будут получены уже из кэша. На **Рис. 8.8** показано содержимое кэша во время последнего обращения по адресу 0x4. Все теги равны нулю, потому что старшие 27 бит всех адресов равны нулю. Количество промахов кэша составляет 3/15 = 20%.

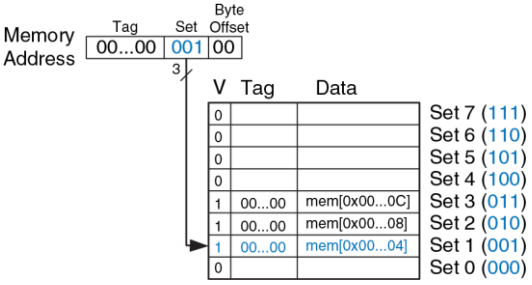


Рис. 8.8 **Содержимое кэша прямого отображения**

Когда два обращения к памяти по разным адресам отображаются в одну и ту же строку кэша, то возникает конфликт и данные, загруженные во время последнего обращения, *вытесняют* (англ.: evict) из кэша данные, загруженные во время предыдущего обращения. В кэше прямого отображения в каждом наборе есть только одна строка, так что два адреса, отображаемые в одну строку, всегда вызывают конфликт. Один из таких конфликтов рассмотрен в **примере 8.7**.

Пример 8.7 КОНФЛИКТЫ ПРИ ОБРАЩЕНИИ В КЭШ-ПАМЯТЬ

Чему будет равен процент промахов при выполнении следующего цикла при наличии кэша прямого отображения емкостью 8 слов, показанного на **Рис. 8.8**? Считайте, что изначально кэш пуст.

```
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Решение: Оба адреса памяти (0x4 и 0x24) отображаются в набор 1. Во время первой итерации цикла данные по адресу 0x4 будут загружены в набор 1. Затем в тот же набор загружаются данные по адресу 0x24, вытесняя данные по адресу 0x4. Во время второй итерации все повторяется: кэш должен повторно прочитать данные по адресу 0x4, вытеснив данные по адресу 0x24. Эти два адреса конфликтуют, так что процент промахов кэша будет 100%.

Многосекционный Наборно-Ассоциативный Кэш

N-секционный наборно-ассоциативный кэш (англ.: *N*-way set associative cache) уменьшает число конфликтов путем расширения набора до *N* строк. Каждый адрес памяти по-прежнему отображается в строго определенный набор, но теперь он может быть отображен в любую из *N* строк этого набора. Можно сказать, что кэш прямого отображения –

это односекционный наборно-ассоциативный кэш. Число *N* называют степенью ассоциативности кэша.

На **Рис. 8.9** показана блок-схема аппаратной реализации наборно-ассоциативного кэша емкостью *C* = 8 слов, с *N* = 2 секциями. Теперь в кэше только *S* = 4 набора вместо 8. Таким образом, только log24 = 2 бита, а не 3, как раньше, используются для выбора нужного набора. Соответственно, размер тега увеличивается с 27 до 28 бит. Каждый набор теперь содержит две секции (2-*way*). Каждая секция состоит из строки (блока данных), тега и бита достоверности. Кэш читает теги и биты достоверности одновременно из обеих секций выбранного набора, после чего сравнивает их с адресом для определения попадания или промаха. Если происходит попадание в одну из секций кэша, то мультиплексор выбирает данные из этой секции и передает их процессору.

Наборно-ассоциативные кэши, как правило, имеют меньший процент промахов, чем кэши прямого отображения той же ёмкости, так как в них происходит меньше конфликтов. Однако, они обычно медленнее и дороже в реализации, так как необходимо использовать дополнительные компараторы и выходной мультиплексор. Кроме того, в таких кэшах возникает вопрос о том, какую именно секцию замещать, когда все они заняты; мы рассмотрим эту проблему в **разделе 8.3.3**.

Большинство коммерческих систем сегодня используют наборно-ассоциативные кэши.

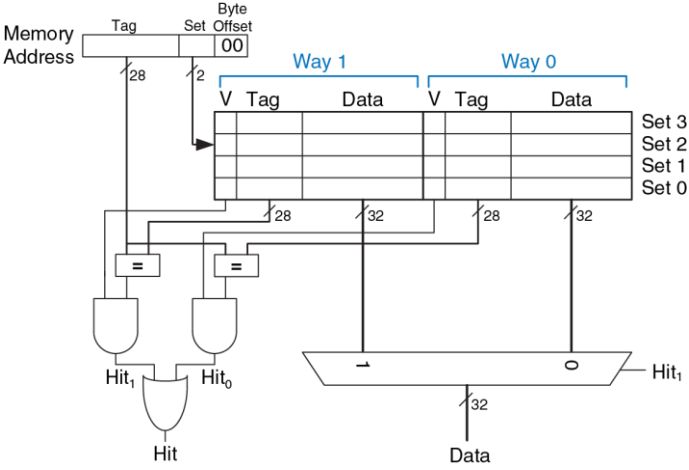


Рис. 8.9 **Двухсекционный наборно-ассоциативный кэш**

Пример 8.8 ПРОЦЕНТ ПРОМАХОВ НАБОРНО-АССОЦИАТИВНОГО КЭША

Повторите **пример 8.7**, используя двухсекционный кэш, показанный на **Рис. 8.9**, емкость которого равна восьми словам.

Решение: Оба обращения в память (по адресу 0x4 и по адресу 0x24) отображаются в набор 1. Однако теперь кэш имеет две секции, так что он может разместить данные для этих обращений в одном наборе. Во время первой итерации цикла пустой кэш приводит к двум промахам, после чего загружает два слова данных в две секции строки 1, как показано на **Рис. 8.10**. Во время следующих четырех итераций данные будут прочитаны из кэша. В результате, процент промахов будет 2/10 = 20%. Напомним, что для кэша прямого отображения того же размера из **примера 8.7** процент промахов был равен 100%.

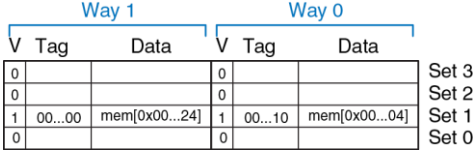


Рис. 8.10 **Содержимое двухсекционного наборно-ассоциативного кэша**

Полностью ассоциативный кэш

Полностью ассоциативный кэш (англ.: fully associative cache) состоит из одного набора с *B* секциями, где *B* – число строк (блоков данных). Адрес памяти может быть отображен в строку любой из этих секций. Можно сказать, что полностью ассоциативный кэш – это *B*-секционный наборно-ассоциативный кэш с одним набором.

На **Рис. 8.11** показан массив памяти SRAM полностью ассоциативного кэша, содержащего 8 строк. При запросе данных должны быть сделаны восемь сравнений адреса с тегами, так как данные могут быть в любой строке. Восьмивходовой мультиплексор (на рисунке не показан) выбирает соответствующую строку и подает ее на выход, если произошло попадание. Полностью ассоциативные кэши обеспечивают при прочих равных условиях минимально возможное количество конфликтов, но требуют еще больше аппаратуры для дополнительных сравнений тегов. Из-за этого они применяются лишь в относительно маленьких кэшах.

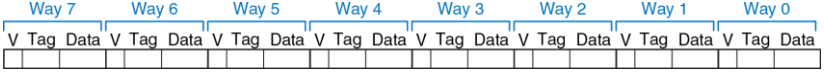


Рис. 8.11 **Полностью ассоциативный кэш с восемью строками**

Длина строки

В предыдущих примерах мы использовали преимущества исключительно временной локальности данных, так как длина строки (т.е. размер блока данных) была равна одному слову. Чтобы воспользоваться пространственной локальностью, в кэш-памяти используют бóльшие по размеру строки, содержащие несколько последовательных слов.

Преимущества строк с длиной, превышающей одно слово, заключается в том, что когда случается промах кэша и требуется прочитать слово данных из памяти, то в эту строку заодно загружаются и соседние слова. Таким образом, последующие обращения с большей вероятностью приведут к попаданию в кэш из-за пространственной локальности данных. Однако, увеличившаяся длина строки означает, что кэш того же размера теперь будет иметь меньшее число самих строк. Это может привести к увеличению числа конфликтов и, соответственно, увеличить вероятность промахов кэша. Более того, потребуется больше времени на чтение данных в строку после промаха, т.к. из памяти необходимо будет прочитать не одно, а несколько слов. Время, требуемое для загрузки данных в строку кэша после промаха, называется *ценой промаха* (англ.: *miss penalty*). Если соседние слова данных в строке не будут использованы в дальнейшем, то усилия на их загрузку будут

потрачены зря. Тем не менее, большинству реальных программ увеличение длины строки приносит пользу.

На **Рис. 8.12** показана блок-схема аппаратной реализации кэша прямого отображения емкостью 8 слов с длиной строки b = 4 слова. В кэше теперь есть только $B = C/b$ = 2 строки. Так как в кэше прямого отображения число строк и наборов совпадает, то в данном случае кэш содержит два набора, соответственно только log22 = 1 бит адреса используется для определения индекса (номера набора). Теперь понадобится новый мультиплексор для выбора одного из слов строки, которое и будет передано процессору. Этот мультиплексор управляется log24 = 2 битами адреса, которые называются *смещением в строке* (англ.: line offset или block offset). Оставшиеся 27 старших бит адреса образуют тег. На всю строку нужен всего один тег, так как слова в ней находятся по последовательным адресам.

На **Рис. 8.13** показано, на какие части делится адрес 0x8000009C при доступе в кэш прямого отображения, показанный на **Рис. 8.12**. Байтовое смещение (Byte Offset) байта всегда равно нулю при доступе к словам. Следующие log2b = 2 бита – номер слова в строке, или смещение в строке (Block Offset). Следующий бит выбирает один из двух наборов (Set), а оставшиеся 27 битов образуют тег (Tag). Следовательно, слово по адресу 0x8000009C отображается в третье слово набора 1 кэша. Принцип использования более длинных строк для

использования свойства пространственной локальности данных применяется и в наборно-ассоциативных кэшах.

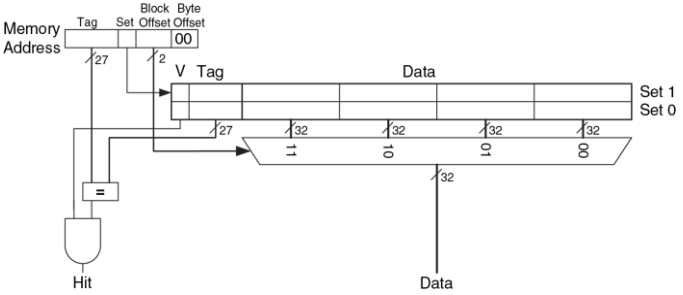


Рис. 8.12 **Кэш прямого отображения с длиной строки в 4 слова**

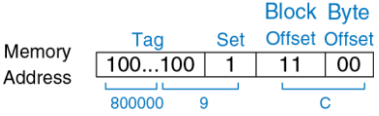


Рис. 8.13 **Части адреса 0x8000009C при доступе в кэш, показанный на Рис. 8.12**

Пример 8.9 ПРОСТРАНСТВЕННАЯ ЛОКАЛЬНОСТЬ С КЭШ-ПАМЯТЬЮ ПРЯМОГО ОТОБРАЖЕНИЯ

Повторите **пример 8.6** для кэша прямого отображения емкостью 8 слов, длина строки которого равна четырем словам.

Решение: На **Рис. 8.14** показано содержимое кэша после первого обращения к памяти. Во время первой итерации цикла происходит промах кэша при обращении в память по адресу 0x4, после чего в строку кэша загружаются данные с адреса 0x0 по адрес 0xC. Все последующие обращения (как показано на рисунке для адреса 0xC) попадают в кэш. Следовательно, процент промахов будет равен 1/15 = 6.67%.

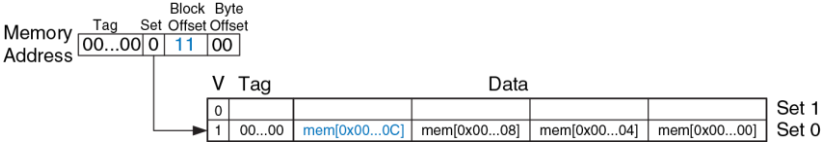


Рис. 8.14 **Содержимое кэша с длиной строки, равной четырем словам**

Подводя итоги

Кэш представляет собой двумерный массив. Строки этого массива называют наборами, а колонки – секциями. Каждый элемент массива содержит строку (т.е. блок данных) и связанные с ней тег и бит достоверности. Кэш характеризуется:

- емкостью C
- длиной строки b и соответствующим числом строк $B = C/b$
- числом строк в наборе (N)

В **Табл. 8.2** перечислены различные способы организации кэш-памяти. Любой адрес в памяти отображается только в один набор, но соответствующие этому адресу данные могут оказаться в любой из секций этого набора.

Табл. 8.2 Способы организации кэш-памяти

Способ организации	Количество секций (N)	Количество наборов (S)
Прямого отображения	1	B
Наборно-ассоциативный	$1 < N < B$	B/N
Полностью ассоциативный	B	1

Ёмкость кэша, степень ассоциативности, число наборов и длина строки обычно кратны степени двойки. Это позволяет однозначно соотносить определенные биты адреса с битами тега, индекса (номера набора) и смещения в строке.

Увеличение степени ассоциативности N обычно уменьшает процент промахов кэша, вызванных конфликтами. При этом бóльшая степень ассоциативности требует большего числа компараторов для сравнения

адреса с тегами. Увеличение длины строки b позволяет использовать пространственную локальность данных для уменьшения процента промахов, однако, при прочих равных условиях, уменьшает число наборов и может привести к увеличению числа конфликтов. Вдобавок большая длина строки увеличивает цену промаха (miss penalty).

8.3.3 Какие данные заместить в кэш-памяти?

В кэш-памяти прямого отображения каждый адрес всегда отображается в одну и ту же строку одного и того же набора, поэтому когда нужно загрузить новые данные в набор, который уже содержит данные, то строка в наборе просто замещается на новые данные. В наборно-ассоциативной и полностью ассоциативной кэш-памяти нужно решить, какую именно из нескольких строк в наборе вытеснить. Учитывая принцип временной локальности, наилучшим вариантом было бы заменить ту строку, которая дольше всего не использовалась, потому что маловероятно, что она будет использована снова. Именно поэтому большинство кэшей используют стратегию замены редко используемых данных (англ.: *least recently used, LRU*).

В двухсекционном наборно-ассоциативном кэше *бит использования U* (от англ. *used*) содержит номер той секции в наборе, которая дольше не использовалась. Каждый раз, когда происходит доступ к одной из секций набора, бит U устанавливается таким образом, чтобы указывать

на другую секцию. Для наборно-ассоциативных кэшей с бóльшим количеством секций отслеживать самые редко используемые строки становится сложно. Для упрощения реализации секции часто делят на две группы, а бит использования указывает на ту группу, которая дольше не использовалась. При необходимости заместить строку вытесняется случайным образом выбранная строка из той группы, которая дольше не использовалась. Такая стратегия называется «*псевдо-LRU*» (*pseudo-LRU*) и на практике достаточно хорошо работает.

Пример 8.10 СТРАТЕГИЯ ЗАМЕЩЕНИЯ LRU

Покажите содержимое двухсекционного наборно-ассоциативного кэша емкостью 8 слов после выполнения следующего кода. Используйте стратегию замещения LRU, длину строки, равную одному слову. Считайте, что изначально кэш пуст.

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

Решение: Первые две инструкции загружают данные из памяти по адресам 0x4 и 0x24 в набор 1 кэша, как показано на **Рис. 8.15 (а)**. Бит использования $U = 0$ показывает, что данные в секции 0 (Way 0) использовались раньше, чем в секции 1. Следующее обращение в память по адресу 0x54 также отображается в строку 1 и вытесняет дольше не использовавшиеся данные из секции 0, как показано на **Рис. 8.15 (и)**. Бит использования при этом устанавливается в 1, указывая, что теперь именно данные в секции 1 не использовались дольше.

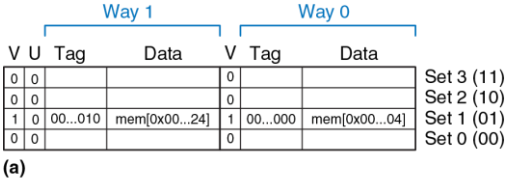


Рис. 8.15 Двухсекционный кэш со стратегией замещения LRU

8.3.4 Улучшенная кэш-память*

В современных системах для сокращения времени доступа к памяти используются несколько уровней кэша. В этом разделе мы рассмотрим производительность двухуровневой системы кэширования и выясним, как длина строки, ассоциативность и емкость кэша влияют на частоту промахов. Также мы рассмотрим, как кэш-память ведет себя при записи

данных в память с использованием стратегий сквозной (write-though) или отложенной (write-back) записи.

Многоуровневые кэши

Чем больше размер кэша, тем больше вероятность, что интересующие нас данные в нем найдутся, и, следовательно, тем меньше у него будет частота промахов. Но большой кэш обычно медленнее, чем маленький, поэтому в современных системах используются как минимум два уровня кэша, как показано на **Рис. 8.16**. Кэш первого уровня (L1) достаточно мал, чтобы обеспечить время доступа в один или два такта. Кэш второго уровня (L2) тоже сделан на основе SRAM, но больше по размеру и поэтому медленнее, чем кэш L1. Сначала процессор ищет данные в кэше L1, а если происходит промах – то в кэше L2. Если и там происходит промах, то процессор обращается за данными к оперативной памяти. Многие современные системы используют еще больше уровней кэша в иерархии памяти, так как доступ к оперативной памяти чрезвычайно медленный.

Пример 8.11 СИСТЕМА С КЭШЕМ L2

Предположим, что в системе, показанной на **Рис. 8.16**, времена доступа к кэшу L1, кэшу L2 и оперативной памяти равны 1, 10 и 100 тактам соответственно. Чему равно среднее время доступа к памяти при условии, что доля промахов у кэша L1 равна 5%, у кэша L2 – 20%, то есть 20% из тех обращений к памяти, которые привели к промаху кэша L1, также приводят и к промаху кэша L2.

Решение: при каждом обращении к памяти процессор сначала ищет запрошенные данные в кэше L1. Когда происходит промах (5% случаев), процессор ищет их в кэше L2. Если снова возникает промах кэша (20% случаев), то процессор обращается за данными в оперативную память. Используя формулу 8.2, мы можем вычислить среднее время доступа к памяти как 1 такт + 0,05 (10 тактов + 0,2 (100 тактов)) = 2,5 такта.

Процент промахов в кэше L2 выше, поскольку до него доходят лишь «трудные» обращения в память – те, которые уже привели к промаху кэша L1. Если бы все обращения шли непосредственно в кэш L2, доля его промахов была бы около 1%.

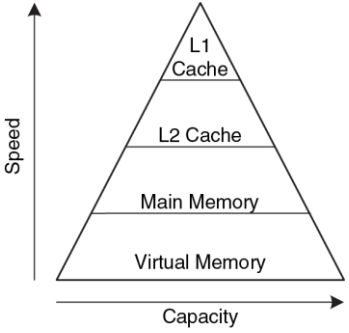


Рис. 8.16 **Иерархия памяти с двумя уровнями кэша**

Сокращение частоты промахов

Процент промахов кэша можно сократить, изменяя его емкость, длину строки и/или ассоциативность. Для этого сначала необходимо разобраться с причинами промахов. Промахи кэша делятся на *неизбежные* промахи (compulsory misses), промахи из-за недостаточной емкости (capacity misses) и промахи из-за конфликтов (conflict misses). Первое обращение к строке кэша всегда приводит к неизбежному промаху, так как эту строку нужно прочесть из оперативной памяти хотя бы один раз независимо от архитектуры кэша. Промахи из-за недостаточной емкости происходят, когда кэш слишком мал для хранения всех одновременно используемых данных. Промахи из-за конфликтов случаются, если несколько адресов памяти отображаются на один и тот же набор кэша и выталкивают из него данные, которые все еще нужны.

Изменение параметров кэша может повлиять на частоту одного или нескольких типов промахов. Например, увеличение размера кэша может сократить промахи из-за конфликтов и промахи из-за недостатка емкости, но никак не повлияет на число неизбежных промахов. С другой стороны, увеличение длины строки может сократить число неизбежных промахов (благодаря локальности данных), но одновременно может увеличить частоту промахов из-за конфликтов, поскольку большее

число адресов будет отображаться на один и тот же набор, увеличивая вероятность конфликтов.

Системы памяти настолько сложны, что лучший способ оценивать их производительность – это запускать тестовые программы, варьируя параметры кэша. На **Рис. 8.17** изображен график зависимости частоты промахов от размера кэша и степени ассоциативности для набора тестовых программ SPEC2000. Небольшое число неизбежных промахов показано темным цветом вдоль оси X и не зависит от емкости кэша. С другой стороны, как и ожидалось, с увеличением емкости кэша частота промахов из-за недостатка емкости сокращается. Увеличение ассоциативности, особенно для кэшей небольшого размера, сокращает количество промахов из-за конфликтов, показанных вдоль верхней части кривой. При этом ассоциативность свыше четырех или восьми секций приводит лишь к незначительному сокращению частоты промахов.

Как уже говорилось, частоту промахов можно уменьшить, используя пространственную локальность данных с помощью более длинных строк кэша. Однако, при прочих равных условиях, с увеличением длины строки в кэше уменьшается количество наборов, что увеличивает вероятность конфликтов. На **Рис. 8.18** показана зависимость частоты промахов от длины строки в байтах (Block Size) для кэшей разной емкости. Для небольших кэшей, таких как кэш емкостью 4 Кбайта, длина строки свыше 64 байтов увеличивает частоту промахов из-за

конфликтов. Для кэшей большей емкости длина строки свыше 64 байтов не влияет на частоту промахов. При этом большая длина строки все же может вызвать увеличение времени выполнения из-за более высокой *цены промаха* (miss penalty) – времени, требуемого для выборки отсутствующей строки кэша из оперативной памяти.

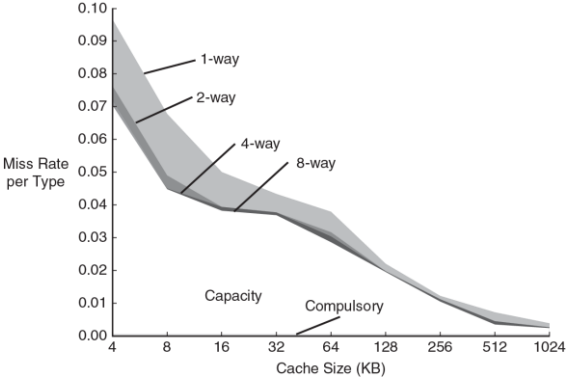


Рис. 8.17 Зависимость частоты промахов от размера и ассоциативности кэша на тестах SPEC2000.
График из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012, с разрешения авторов

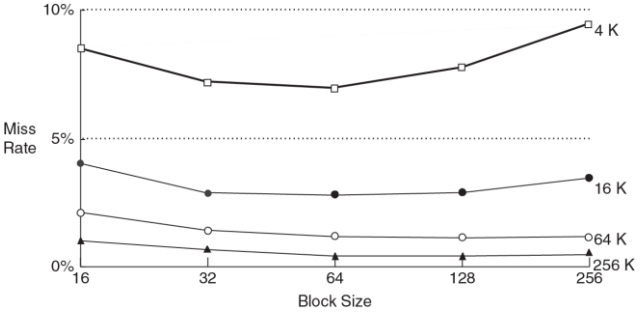


Рис. 8.18 Зависимость частоты промахов от длины строки и размера кэша на тестах SPEC92.
График из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012, с разрешения авторов

Стратегии записи

В предыдущих разделах мы рассматривали чтение из памяти. Запись в память выполняется примерно так же, как и чтение. При выполнении команды сохранения данных процессор сначала проверяет кэш. В случае промаха кэша соответствующая строка выбирается из оперативной памяти в кэш, а затем в нее записывается нужное слово. В случае попадания в кэш слово просто записывается в строку.

Кэши делятся на два типа – со *сквозной записью* (write-through) и с *отложенной записью* (write-back). В кэше со сквозной записью данные, записываемые в кэш, одновременно записываются и в оперативную память. В кэше с отложенной записью у каждой строки есть бит изменения (*D*, от англ. dirty). Если в строку производилась запись, то этот бит равен 1, в противном случае он равен 0. Измененные строки записываются обратно в оперативную память только тогда, когда они вытесняются из кэша. В кэше со сквозной записью биты изменения не нужны, но такой кэш обычно приводит к большему количеству операций записи в память, чем кэш с отложенной записью. Из-за того, что время обращения к оперативной памяти очень велико, в современных системах обычно используют кэши с отложенной записью.

Пример 8.12 Сквозная и отложенная запись

Допустим, что длина строки кэша – четыре слова. Сколько обращений к оперативной памяти потребуется при выполнении кода, приведенного ниже, если используется стратегия сквозной записи, и сколько, если используется отложенная запись?

```
sw $t0, 0x0($0)
sw $t0, OxC($0)
sw $t0, 0x8($0)
sw $t0, 0x4($0)
```

Решение: Все четыре команды сохранения пишут в одну и ту же строку кэша. При сквозной записи каждая команда сохраняет слово в оперативную память, соответственно, потребуется четыре обращения к памяти. При отложенной записи потребуется только одно обращение – тогда, когда эта строка будет вытеснена из кэша.

8.3.5 Эволюция кэш-памяти процессоров MIPS

В **Табл. 8.3** прослежена эволюция организации кэш-памяти в процессорах MIPS с 1985 по 2010 год. Основные тенденции – введение нескольких уровней кэша, увеличение емкости и степени ассоциативности. Движущими силами этих изменений явились увеличивающийся разрыв между частотой процессора и скоростью оперативной памяти, а также снижение цены транзисторов в микросхемах. Разрыв между частотой процессора и скоростью памяти делает необходимым сокращение вероятности промахов во избежание появления узкого места в производительности системы при обращениях к памяти, а снижение цен на транзисторы позволяет увеличивать размеры кэшей.

Табл. 8.3 Эволюция кэш-памяти процессоров MIPS*

Год	Процессор	МГц	Кэш L1	Кэш L2
1985	R2000	16.7	нет	нет
1990	R3000	33	32 Кбайт, прямого отображения	нет
1991	R4000	100	8 Кбайт, прямого отображения	1 Мбайт, прямого отображения
1995	R10000	250	32 Кбайт, двухсекционный	4 Мбайт, двухсекционный
2001	R14000	600	32 Кбайт, двухсекционный	16 Мбайт, двухсекционный
2004	R16000A	800	64 Кбайт, двухсекционный	16 Мбайт, двухсекционный
2010	MIPS32 1074K	1500	32 Кбайт	переменного размера

* Адаптировано из книги D. Sweetman, See MIPS Run, Morgan Kaufmann, 1999.

8.4 ВИРТУАЛЬНАЯ ПАМЯТЬ

Большинство современных вычислительных систем в качестве нижнего уровня в иерархии памяти используют жесткие диски, представляющие собой магнитные или твердотельные запоминающие устройства (см. **Рис. 8.4**). По сравнению с идеальной памятью, которая должна быть быстрой, дешевой и большой, жесткий диск имеет большой объем и недорого стоит, однако невероятно медленно работает. Жесткий диск обеспечивает намного больший объем, чем недорогая оперативная память (DRAM). Однако если существенная часть обращений к памяти осуществляется к жесткому диску, скорость работы сильно снижается. Вы могли сталкиваться с этой проблемой на персональном компьютере, если одновременно запускали слишком много программ.

На **Рис. 8.19** показан магнитный жесткий диск со снятой крышкой. Как следует из названия, жесткий диск состоит из одной или нескольких пластин, каждой из которых касается головка считывания-записи, расположенная на конце длинного треугольного кронштейна. Головка перемещается в правильное положение на диске и считывает или записывает информацию при помощи магнитного поля в то время, пока диск вращается под ней. Головка ищет правильное положение на диске в течение нескольких миллисекунд – это быстро с точки зрения человека, но в миллионы раз медленнее, чем скорость работы процессора.

Рис. 8.19 **Жесткий диск**

Цель включения жесткого диска в иерархию памяти – занедорого создать видимость памяти большого объема, одновременно

обеспечивая для большинства обращений к памяти скорость доступа, равную скорости более быстрых типов памяти. Например, компьютер с оперативной памятью на 128 Мбайт может обеспечить видимость наличия 2 Гбайт оперативной памяти, используя для этого жесткий диск.

В этом случае большая память, объёмом 2 Гбайта, называется *виртуальной памятью*, а меньшая память, объёмом 128 Мбайт, называется *физической памятью*. В этом разделе мы будем использовать термин физическая память, подразумевая оперативную память.

> Компьютер с 32-битной адресацией имеет доступ к 2^{32} байтам = 4 Гбайтам памяти. Эта одна из причин перехода к 64-битным компьютерам, которые могут получать доступ к памяти большего объема.

Программы могут обращаться к данным в любом месте виртуальной памяти, поэтому они должны использовать *виртуальные адреса*, которые определяют расположение данных в виртуальной памяти. Физическая память хранит последние запрошенные из виртуальной памяти блоки данных. Таким образом, физическая память выступает в роли кэша для виртуальной памяти, то есть большинство обращений

происходит к быстрой физической памяти (DRAM), и в то же время программа имеет доступ к большей по объему виртуальной памяти.

Подсистемы виртуальной памяти используют другие термины для тех же самых принципов кэширования, которые были рассмотрены в **разделе 8.3**. В **Табл. 8.4** приведены сходные термины.

Табл. 8.4 Соответствие терминов кэша и виртуальной памяти

Кэш	Виртуальная память
Строка	Страница
Длина строки	Размер страницы
Смещение относительно начала строки	Смещение относительно начала страницы
Промах	Страничная ошибка
Тег	Номер виртуальной страницы

Виртуальная память разделена на виртуальные страницы, обычно размером 4 Кбайт. Физическая память аналогичным образом разделена на физические страницы. Размер виртуальных и физических страниц одинаков. Виртуальная страница может располагаться в физической памяти (DRAM) или на жестком диске.

Например, на **Рис. 8.20** показана виртуальная память, которая больше физической памяти. Прямоугольники обозначают страницы. Некоторые виртуальные страницы расположены в физической памяти, а

некоторые – на жестком диске. Процесс преобразования виртуального адреса в физический называется *трансляцией адреса*.

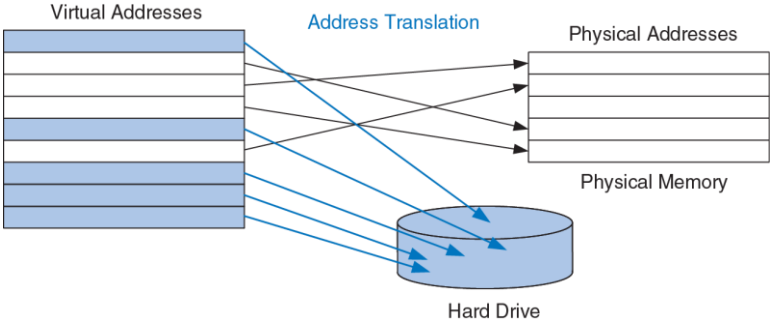


Рис. 8.20 **Виртуальные и физические страницы**

Если процессор обращается к виртуальному адресу, которого нет в физической памяти, происходит страничная ошибка (англ.: page fault) и операционная система загружает соответствующую страницу с жесткого диска в физическую память.

Чтобы избежать страничных ошибок, вызванных конфликтами, любая виртуальная страница может отображаться на любую физическую

страницу. Другими словами, физическая память работает как полностью ассоциативный кэш для виртуальной памяти. В традиционном полностью ассоциативном кэше каждая секция содержит компаратор, который проверяет старшие биты адреса на соответствие тегу, чтобы определить, находятся ли там нужные данные. В аналогичной системе виртуальной памяти каждой физической странице нужен был бы компаратор, чтобы сверять старшие биты виртуальных адресов с тегом и определять, отображается ли виртуальная страница на эту физическую страницу.

На практике виртуальная память имеет настолько много физических страниц, что обеспечить компаратором каждую страницу было бы чересчур дорого. Вместо этого в подсистемах виртуальной памяти используется трансляция адреса при помощи *таблицы страниц* (англ.: page table). Таблица страниц содержит запись для каждой виртуальной страницы, указывающую ее расположение в физической памяти или на жестком диске. Каждая команда загрузки или сохранения требует доступа к таблице страниц с последующим доступом к физической памяти. Обращение к таблице страниц позволяет транслировать виртуальный адрес, используемый программой, в физический адрес. Затем физический адрес используется для фактического чтения или записи данных.

Таблица страниц обычно настолько велика, что сама находится в физической памяти. Таким образом, каждая команда загрузки или сохранения данных включает два обращения к физической памяти: обращение к таблице страниц и собственно обращение к данным. Чтобы ускорить трансляцию адреса, используется *буфер ассоциативной трансляции* (англ.: translation lookaside buffer, TLB), который содержит наиболее часто используемые записи таблицы страниц.

В оставшейся части этого раздела мы более подробно рассмотрим трансляцию адресов, таблицы страниц и TLB.

8.4.1 Трансляция адресов

В системах с виртуальной памятью программы используют виртуальные адреса и поэтому имеют доступ к памяти большого объема. Компьютер должен транслировать эти виртуальные адреса, чтобы либо найти соответствующий адрес в физической памяти, либо получить страничную ошибку и загрузить данные с жесткого диска.

Вспомните, что виртуальная память и физическая память разделены на страницы. Старшие биты виртуального и физического адресов определяют номер виртуальной и физической страницы

соответственно. Младшие биты определяют положение слова внутри страницы и называются смещением относительно начала страницы.

На **Рис. 8.21** показана страничная организация подсистемы виртуальной памяти объёмом 2 Гбайта и физической памятью объёмом 128 Мбайт, разделенными на страницы по 4 Кбайт. MIPS использует 32-битную адресацию. Так как виртуальная память имеет объём 2 Гбайт = 2^{31} байт, то используются только младшие 31 бит виртуального адреса, а старший бит всегда равен нулю. Аналогично, так как физическая память имеет объём 128 Мбайт = 2^{27} байт, то используются только младшие 27 бит физического адреса, а старшие 5 бит всегда равны нулю.

Поскольку размер страницы составляет 4 Кбайт = 2^{12} байт, существует $2^{31}/2^{12} = 2^{19}$ виртуальных страниц и $2^{27}/2^{12} = 2^{15}$ физических страниц. Таким образом, номера страниц виртуальной и физической памяти состоят из 19 и 15 бит соответственно. В любой момент времени физическая память может хранить максимум 1/16 от числа страниц виртуальной памяти. Остальные виртуальные страницы хранятся на жестком диске.

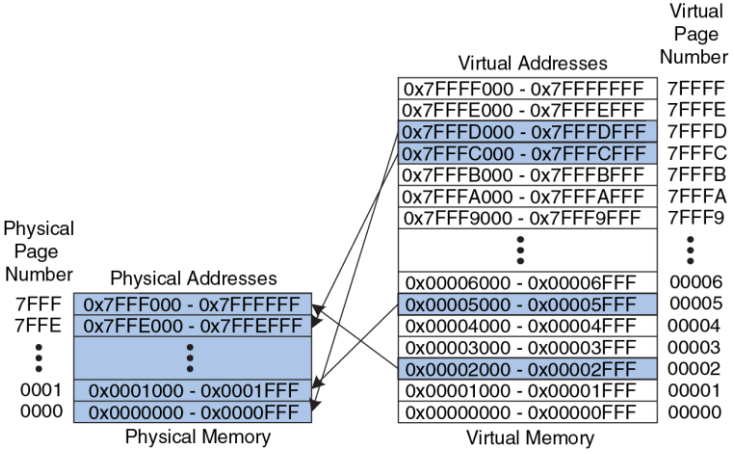


Рис. 8.21 **Физические и виртуальные страницы**

На **Рис. 8.21** показано, как виртуальная страница 5 отображается на физическую страницу 1, виртуальная страница 0x7FFFC отображается на физическую страницу 0x7FFE и т. д. Например, виртуальный адрес 0x53F8 (смещение 0x3F8 от начала виртуальной страницы номер 5) отображается на физический адрес 0x13F8 (смещение 0x3F8 от начала

физической страницы номер 1). Младшие 12 бит виртуального и физического адресов одинаковы (0x3F8) и определяют смещение от начала виртуальной и физической страницы. Таким образом, чтобы получить физический адрес из виртуального, необходимо транслировать только номер страницы.

На **Рис. 8.22** показан процесс трансляции виртуального адреса в физический. Младшие 12 бит определяют смещение от начала страницы и не нуждаются в транслировании.

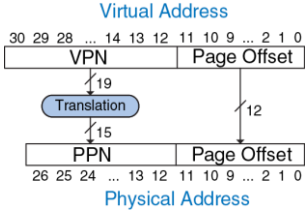


Рис. 8.22 **Трансляция виртуального адреса в физический**

Старшие 19 бит виртуального адреса определяют номер виртуальной страницы (англ.: *virtual page number*, *VPN*) и транслируются в 15-битный номер физической страницы (англ.: *physical page number*, *PPN*).

В следующих двух разделах рассказывается, как для трансляции адресов используются таблицы страниц и TLB.

Пример 8.13 ТРАНСЛЯЦИЯ ВИРТУАЛЬНОГО АДРЕСА В ФИЗИЧЕСКИЙ

Найдите физический адрес, соответствующий виртуальному адресу 0x247C, используя подсистему виртуальной памяти, показанную на **Рис. 8.21**.

Решение: 12 бит, обозначающие смещение от начала страницы (0x47C), не нуждаются в транслировании. Оставшиеся 19 бит виртуального адреса определяют номер виртуальной страницы. Это означает, что виртуальный адрес 0x247C находится внутри виртуальной страницы 0x2. Согласно **Рис. 8.21**, виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C.

8.4.2 Таблица страниц

Процессор использует таблицу страниц для трансляции виртуальных адресов в физические. Таблица страниц содержит отдельную запись для каждой виртуальной страницы. Эта запись содержит номер физической страницы и бит достоверности (англ.: valid bit). Если бит достоверности равен 1, виртуальная страница отображается на физическую страницу, номер которой указан в записи. В обратном случае виртуальная страница находится на жестком диске.

Поскольку таблица страниц велика, она хранится в физической памяти. Предположим, что она хранится в виде непрерывного массива, как показано на **Рис. 8.23**.

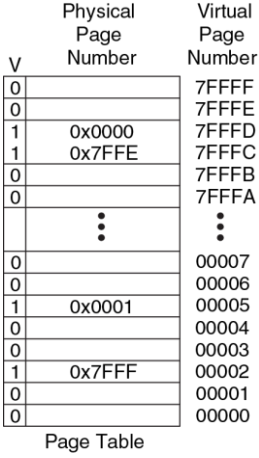


Рис. 8.23 **Таблица страниц для случая, показанного на Рис. 8.21**

Эта таблица страниц содержит информацию об отображении страниц, показанных на **Рис. 8.21**. Номера строк в таблице страниц соответствуют номерам виртуальных страниц (VPN). Например, строка номер 5 определяет, что виртуальная страница 5 отображается на физическую страницу 1. Запись 6 недействительна (бит достоверности $V = 0$), то есть виртуальная страница 6 расположена на жестком диске, а не в физической памяти.

Пример 8.14 ИСПОЛЬЗОВАНИЕ ТАБЛИЦЫ СТРАНИЦ ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Найти физический адрес, соответствующий виртуальному адресу 0x247C, используя таблицу страниц, показанную на **Рис. 8.23**.

Решение: На **Рис. 8.24** показана трансляция виртуального адреса в физический для виртуального адреса 0x247C. 12 бит, обозначающие смещение от начала страницы, не нуждаются в транслировании. Оставшиеся 19 бит виртуального адреса – это номер виртуальной страницы, 0x2, они определяют номер в таблице страниц. Таблица страниц содержит информацию о том, что виртуальная страница 0x2 отображается на физическую страницу 0x7FFF. Таким образом, виртуальный адрес 0x247C отображается на физический адрес 0x7FFF47C. Младшие 12 бит одинаковы для физического и виртуального адресов.

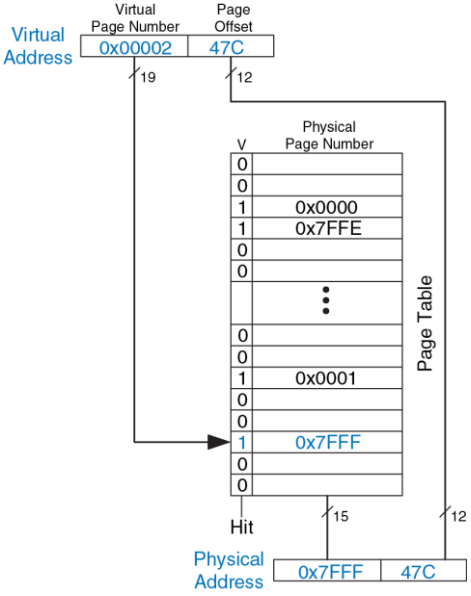


Рис. 8.24 **Трансляция адреса при помощи таблицы страниц**

Таблица страниц может храниться в любом месте физической памяти, ее расположение определяется операционной системой. Процессор обычно использует выделенный регистр, называемый регистром таблицы страниц, для хранения ее базового адреса.

Чтобы выполнить операцию загрузки или сохранения данных, процессор должен сначала транслировать виртуальный адрес в физический, а затем обратиться к физической памяти, используя полученный физический адрес. Процессор извлекает номер виртуальной страницы из виртуального адреса и прибавляет его к содержимому регистра таблицы страниц, чтобы найти физический адрес соответствующей записи в таблице страниц, расположенной в физической памяти. Затем процессор считывает эту запись и получает номер физической страницы. Если запись действительна, т.е. бит достоверности равен 1, то процессор объединяет номер физической страницы и смещение и получает физический адрес. Наконец, он читает или записывает данные, используя этот физический адрес. Поскольку таблица страниц тоже хранится в физической памяти, каждая команда загрузки или сохранения требует два обращения к физической памяти.

8.4.3 Буфер ассоциативной трансляции

Виртуальная память оказывала бы огромное негативное влияние на производительность, если бы требовалось обращение к таблице страниц при выполнении каждой команды загрузки или сохранения – это удваивало бы время выполнения этих команд. К счастью, обращения к таблице страниц имеют высокую временную локальность. Временная и пространственная локальность обращений к данным и большой размер страницы означают, что с большой вероятностью многие следующие друг за другом команды загрузки или сохранения обращаются к одной и той же странице. Поэтому, если процессор запомнит последнюю запись таблицы страниц, которую он прочитал, то он, скорее всего, сможет повторно использовать результат трансляции, не выполняя повторное чтение таблицы страниц. В целом, процессор может хранить несколько последних записей, прочитанных из таблицы страниц в небольшой кэш-памяти, называемой буфером ассоциативной трансляции *(TLB).* Процессор «заглядывает» в TLB в поисках информации о трансляции, прежде чем обратиться к таблице страниц в физической памяти. В реальных программах большинство обращений находят в TLB нужную информацию (в этом случае говорят, что произошло попадание в TLB), что избавляет от затрат на повторное чтение таблицы страниц из физической памяти.

TLB организован как полностью ассоциативный кэш и обычно хранит от 16 до 512 записей. Каждая запись в TLB хранит номер виртуальной страницы и соответствующий ей номер физической страницы. Обращение к TLB происходит по номеру виртуальной страницы. Если происходит попадание в TLB, то возвращается соответствующий номер физической страницы. В противном случае процессор должен прочитать нужную запись из таблицы страниц в физической памяти. Буфер ассоциативной трансляции разрабатывают таким образом, чтобы он был маленьким и чтобы доступ к нему занимал менее одного такта. Даже при этом доля попаданий в него обычно превышает 99%. TLB уменьшает число обращений к памяти, требуемое для большинства команд загрузки и сохранения, с двух до одного.

Пример 8.15 ИСПОЛЬЗОВАНИЕ TLB ДЛЯ ТРАНСЛЯЦИИ АДРЕСА

Рассмотрим подсистему виртуальной памяти, показанную на **Рис. 8.21**. Используйте TLB с двумя записями или объясните, почему необходимо обращение к таблице страниц, чтобы осуществить трансляцию виртуальных адресов 0x247C и 0x5FB0 в физические адреса. Предположим, что TLB хранит корректные записи для виртуальных страниц 0x2 и 0x7FFFD.

Решение: На **Рис. 8.25** показан TLB с двумя записями и входящим запросом на трансляцию виртуального адреса 0x247C. Из поступившего на вход TLB адреса извлекается номер виртуальной страницы, равный 0x2, после чего этот номер сравнивается с номерами виртуальных страниц, находящимися в записях.

Совпадение номеров обнаружено для записи 0 (Entry 0), при этом запись действительна (V = 1), поэтому происходит попадание в TLB. Транслированный физический адрес представляет собой номер физической страницы из записи с совпавшим номером, равный 0x7FFF, объединенный со смещением, скопированным из виртуального адреса. Как всегда, смещение не требует трансляции.

Запрос на трансляцию виртуального адреса 0x5FB0 приводит к промаху TLB, поэтому для трансляции необходимо использовать таблицу страниц.

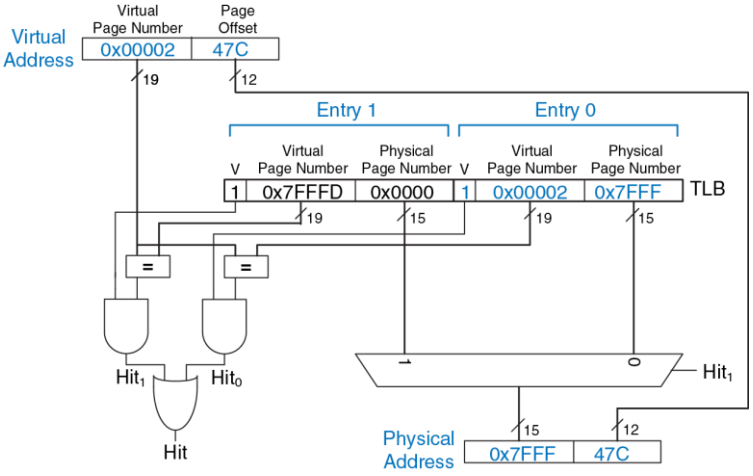


Рис. 8.25 **Трансляция адреса с использованием TLB**

8.4.4 Защита памяти

До сих пор в этом разделе мы рассматривали применение виртуальной памяти для создания видимости наличия быстрой, недорогой и большой по объему памяти. Не менее важной причиной использования виртуальной памяти является обеспечение защиты нескольких одновременно выполняемых программ друг от друга.

Как вы, возможно, знаете, современные компьютеры обычно выполняют несколько программ или процессов одновременно. Все эти программы одновременно присутствуют в физической памяти. В хорошо спроектированной компьютерной системе программы должны быть защищены друг от друга, чтобы работа одной программы не могла нарушить работу другой программы. Точнее, ни одна программа не должна иметь доступ к памяти другой программы без разрешения. Это называется *защитой памяти*.

Системы виртуальной памяти обеспечивают защиту памяти, предоставляя каждой программе персональное виртуальное адресное пространство. Каждая программа может использовать столько памяти в пределах виртуального пространства, сколько необходимо, но только часть виртуального адресного пространства находится в физической памяти в каждый момент времени. Каждая программа может полностью использовать свое виртуальное адресное пространство, не беспокоясь

о том, где расположены остальные программы. Однако программа имеет доступ только к тем физическим страницам, информация о которых находится в ее таблице страниц. Таким образом, программа не может случайно или преднамеренно получить доступ к физическим страницам другой программы, поскольку они не отображены в ее таблице страниц. Иногда несколько программ должны иметь доступ к общим командам или данным. Для этого операционная система добавляет к каждой записи в таблице страниц несколько служебных битов, позволяющих определить, какие именно программы могут писать в общие (разделяемые, англ.: shared) физические страницы.

8.4.5 Стратегии замещения страниц*

Подсистемы виртуальной памяти используют стратегию обратной записи (англ.: write-back) и стратегию вытеснения редко используемых страниц (англ.: least recently used, LRU) для замещения страниц в физической памяти. Стратегия сквозной записи (англ.: write-though), при которой каждая запись в физическую память приводит заодно и к записи на жесткий диск, была бы непрактична, потому что команды сохранения выполнялись бы со скоростью жесткого диска, а не со скоростью процессора (миллисекунды вместо наносекунд). При стратегии обратной записи физическая страница записывается обратно на жесткий диск только тогда, когда она вытесняется из

физической памяти. Процесс записи физической страницы обратно на жесткий диск и размещение на ее месте другой виртуальной страницы называется *замещением страниц* (англ.: paging), а жесткий диск в подсистемах виртуальной памяти иногда называется *пространством подкачки* (англ.: swap). Когда происходит страничная ошибка, процессор замещает одну из редко используемых физических страниц на отсутствующую виртуальную страницу, вызвавшую страничную ошибку. Такая реализация требует наличия в каждой записи таблицы страниц двух дополнительных служебных битов: бита изменения *D (dirty bit)* и бита доступа *U (use bit)*.

Бит изменения равен 1, если любая из команд сохранения изменила содержимое физической страницы с момента ее считывания с жесткого диска. Когда физическая страница с битом изменения, равным 1, замещается на новую, то ее необходимо записать обратно на жесткий диск. Если же бит изменения замещаемой страницы равен 0, то точная копия этой страницы и так уже находится на жестком диске, поэтому записывать ее туда не имеет смысла.

Бит доступа равен 1, если к физической странице недавно обращались. Как и в случае кэш-памяти, точный алгоритм LRU было бы слишком сложно реализовать. Вместо этого операционная система использует приближенный алгоритм вытеснения редко используемых страниц, периодически сбрасывая биты доступа в таблице страниц. Если к

странице обращаются, то бит доступа устанавливается в 1. При возникновении страничной ошибки операционная система находит страницу с $U = 0$ и вытесняет ее из физической памяти. Таким образом, замещается не обязательно самая редко используемая страница, но одна из нескольких относительно редко используемых страниц.

8.4.6 Многоуровневые таблицы страниц*

Таблицы страниц могут занимать большой объем физической памяти. Например, для показанной ранее таблицы страниц при размере виртуальной памяти, равном 2 Гбайт и размере страниц, равном 4 Кбайт, потребуется 2^{19} записей. Если размер каждой записи равен 4 байтам, то размер таблицы страниц равен $2^{19} \times 2^2$ байт = 2^{21} байт = 2 Мбайта.

Чтобы сэкономить физическую память, таблицы страниц можно поделить на несколько уровней (обычно два). Таблица страниц первого уровня всегда хранится в физической памяти. Она указывает, в каком месте виртуальной памяти хранятся маленькие таблицы страниц второго уровня. Каждая таблица страниц второго уровня хранит информацию о некотором диапазоне виртуальных страниц. Если какой-то диапазон виртуальных адресов не используется, то соответствующая таблица страниц второго уровня может быть

сохранена на жесткий диск, чтобы не занимать место в физической памяти.

В двухуровневой таблице страниц номер виртуальной страницы разделен на две части: номер таблицы страниц (англ.: page table number) и смещение в таблице страниц (англ.: page table offset), как показано на **Рис. 8.26**. Номер таблицы страниц указывает на номер строки в таблице первого уровня, которая должна всегда находиться в физической памяти. Запись в таблице страниц первого уровня содержит базовый адрес таблицы страниц второго уровня или, если V = 0, указывает, что ее необходимо загрузить с жесткого диска. Смещение в таблице страниц указывает на номер строки в таблице второго уровня. Оставшиеся 12 бит виртуального адреса – это, как и ранее, смещение от начала страницы размером 2^{12} = 4 Кбайт.

На **Рис. 8.26** 19-битный номер виртуальной страницы разделен на две части по 9 и 10?бит, определяющие номер таблицы страниц и смещение в таблице страниц соответственно. Таким образом, таблица страниц первого уровня содержит 2^9 = 512 записей. Каждая из 512 таблиц страниц второго уровня содержит 2^{10} = 1024 записей.

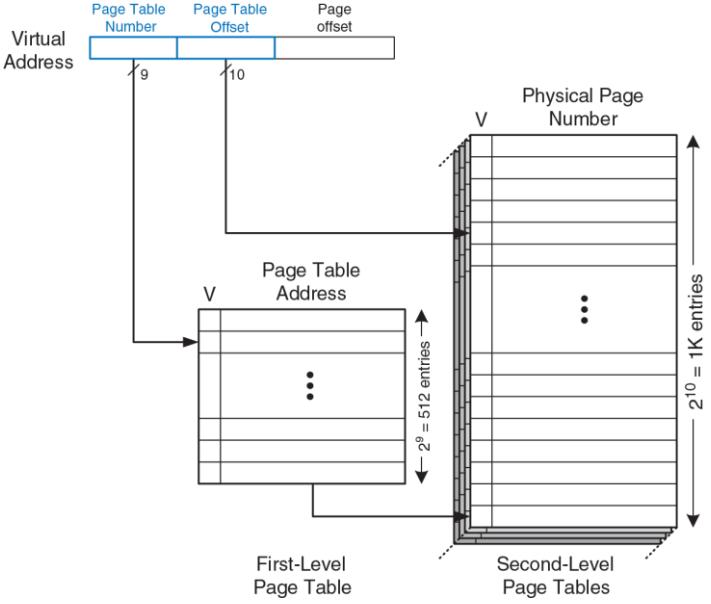


Рис. 8.26 Иерархические таблицы страниц

Если размер каждой записи в таблицах страниц первого и второго уровня равен 32 битам (4 байтам) и только две таблицы страниц второго уровня присутствуют в физической памяти одновременно, то иерархическая таблица страниц занимает всего (512 × 4 байт) + 2 × (1024 × 4 байт) = 10 Кбайт физической памяти. Очевидно, что двухуровневая таблица страниц требует лишь малую часть физической памяти, необходимой для хранения всей одноуровневой таблицы страниц (2 Мбайта). Недостаток двухуровневой таблицы состоит в том, что при промахе TLB будет необходимо выполнить на одно обращение к памяти больше.

Пример 8.16 ИСПОЛЬЗОВАНИЕ МНОГОУРОВНЕВОЙ ТАБЛИЦЫ СТРАНИЦ

На **Рис. 8.27** показан возможный вариант содержимого двухуровневой таблицы страниц, изображенной на **Рис. 8.26**. Показано содержимое только одной таблицы страниц второго уровня. Используя эту двухуровневую таблицу, опишите, что происходит при обращении к виртуальному адресу 0x003FEFB0.

Решение: Как и всегда, требуется транслировать только номер виртуальной страницы. Старшие девять бит виртуального адреса – это номер таблицы страниц, равный 0x0. Он определяет номер строки в таблице первого уровня. Соответствующая запись в таблице первого уровня указывает, что нужная таблица страниц второго уровня уже располагается в памяти (V = 1), а ее физический адрес равен 0x2375000.

Следующие десять бит виртуального адреса, равные 0x3FE – это смещение в таблице страниц, которое равно номеру строки в таблице второго уровня. В таблице второго уровня 1024 записи, а строки пронумерованы снизу вверх. Таким образом, запись 0x3FE в таблице страниц второго уровня – вторая сверху. Она указывает, что виртуальная страница тоже находится в физической памяти (V = 1), а номер физической страницы равен 0x23F1. Физический адрес получается путем объединения номера физической страницы и смещения от начала страницы и равен 0x23F1FB0.

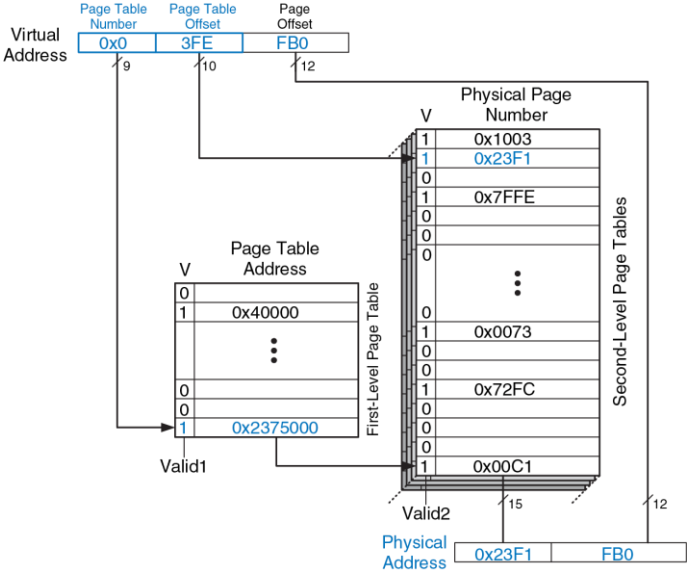


Рис. 8.27 **Трансляция адреса с использованием двухуровневой таблицы страниц**

8.5 СИСТЕМЫ ВВОДА-ВЫВОДА

Системы ввода-вывода (Input/Output, I/O) используются для подключения компьютера к внешним устройствам, которые называются периферийными. В персональном компьютере периферийные устройства включают в себя клавиатуры, мониторы, принтеры и беспроводные сети. Во встраиваемых системах – нагревательный элемент тостера, синтезатор речи куклы, топливный инжектор двигателя, двигатели позиционирования солнечных панелей спутника и так далее. Процессор получает доступ к устройствам ввода-вывода, используя шины адреса и данных так же, как при получении доступа к памяти.

> *Встроенные системы* – это компьютеры специального назначения, управляющие некоторым физическим устройством. Обычно они состоят из микроконтроллера (МК) или цифрового сигнального процессора, подключенного к одному или нескольким физическим устройствам ввода-вывода. Например, в микроволновой печи может быть простой микроконтроллер для чтения кнопок, работы часов и таймера, а также для включения и выключения магнетрона в нужное время.

Часть адресного пространства отводится под устройства ввода-вывода вместо памяти. Например, адреса от 0xFFFF0000 до 0xFFFFFFFF. Как было уже с казано в разделе 6.6.1, эти адреса расположены в зарезервированной части карты памяти. Каждому устройству ввода-

вывода присваивается один или несколько адресов в этом диапазоне. При сохранении по заданному адресу посылаются данные в устройства. При загрузке – данные получаются от устройства. Этот метод связи с устройствами ввода-вывода называется вводом-выводом, отображенным в память (MMIO, memory-mapped I/O).

> Некоторые архитектуры, в частности, x86, используют для связи с устройствами ввода-вывода специализированные команды вместо ввода-вывода, отображаемого в память. Эти команды представлены в следующем виде, где `device1` и `device2` являются уникальными идентификаторами периферийного устройства):
>
> ```
> lwio $t0, device1
> swio $t0, device2
> ```
>
> Этот тип связи с устройствами ввода-вывода называется программируемым вводом-выводом.

В системе с MMIO процедуры загрузки или сохранения могут получать доступ или к памяти, или устройству ввода-вывода. На **Рис. 8.28** изображены аппаратные средства, необходимые для поддержки двух устройств ввода-вывода, отображенных в память. *Дешифратор адреса* определяет, какое устройство связывается с процессором. Он использует сигналы *Address* и *MemWrite*, чтобы генерировать управляющие сигналы для остальной части аппаратных средств. Мультиплексор *ReadData* производит выбор между памятью и

различными устройствами ввода-вывода. Регистры с разрешением записи содержат в себе значения, записываемые в устройства ввода-вывода.

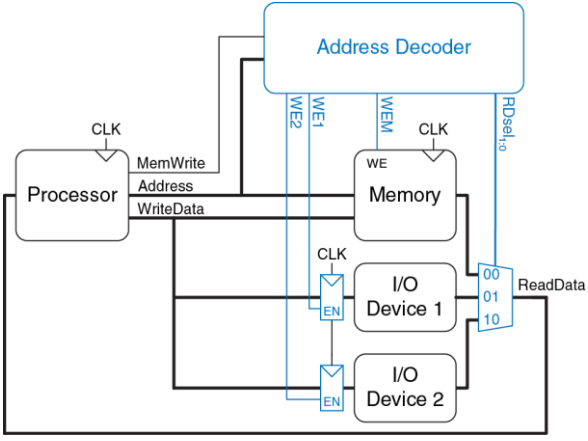


Рис. 8.28 **Отображение устройств ввода-вывода в память**

Пример 8.17 ОБМЕН ДАННЫМИ С УСТРОЙСТВАМИ ВВОДА-ВЫВОДА

Допустим, что устройству ввода-вывода “I/O Device 1” на **Рис. 8.28** назначен адрес памяти 0xFFFFFFF4. Покажите код на языке ассемблера для MIPS, предназначенный для записи значения 7 в “I/O Device 1” и для чтения данных из “I/O Device 1”.

Решение: Приведенный ниже ассемблерный код для MIPS записывает значение 7 в “I/O Device 1”.

```
addi $t0, $0, 7
sw $t0, 0xFFF4($0) # FFF4 is sign-extended to 0xFFFFFFF4
```

Дешифратор адреса устанавливает WE1, потому что адрес 0xFFFFFFF4 и значением MemWrite является TRUE. Данные на шине WriteData, значение 7, записывается в регистр, который подключен к входным контактам I/O Device 1.

Чтобы считать данные из “I/O Device 1”, процессор должен выполнить команду:

```
lw $t1, 0xFFF4($0)
```

Дешифратор адреса устанавливает RDsel1:0 в 01, поскольку он определяет, что адрес – 0xFFFFFFF4 и значение MemWrite – FALSE. Выходные данные из “I/O Device 1” проходят через мультиплексор на шину ReadData и загружаются в регистр процессора $t1.

Программное обеспечение, которое взаимодействует с устройством ввода-вывода, называется *драйвером устройства*. Вы, вероятно, загружали или устанавливали драйверы для вашего принтера или другого устройства ввода-вывода. Написание драйвера требует детального знания об аппаратной части этого устройства. Другие программы вызывают функции в драйвере для получения доступа к устройству без необходимости понимания низкоуровневого аппаратного обеспечения.

Адреса, связанные с устройствами ввода-вывода, часто называются *регистрами ввода-вывода*, потому что могут соответствовать регистрам устройства, как показано на **Рис. 8.28**. В следующих частях этой главы приведены конкретные примеры устройств. **раздел 8.6** рассматривает ввод-вывод в контексте встраиваемых систем, демонстрируя, как использовать микроконтроллер на основе MIPS для управления множеством физических устройств. В **разделе 8.7** дается обзор основных систем ввода-вывода, используемых в ПК.

8.6 ВВОД-ВЫВОД ВО ВСТРОЕННЫХ СИСТЕМАХ

В 2011 году было продано микроконтроллеров на сумму, составляющую около 16 млрд долларов, и этот рынок продолжает расти примерно на 10% в год. Микроконтроллеры стали вездесущими и практически невидимыми: предположительно, по 150 штук в каждом доме и по 50 – в каждом автомобиле в 2010 году. 8051 – классический 8-битный микроконтроллер, изначально разработанный Intel в 1980 году и в настоящее время продающийся целым рядом производителей. Серия микроконтроллеров Atmel AVR стала популярной среди энтузиастов-любителей в качестве «мозга» платформы Arduino. Среди 32-разрядных микроконтроллеров ведущей компанией рынка является Renesas, в то время как ARM является крупным игроком в мобильных системах, включая iPhone. Другие крупные производители микроконтроллеров – компании Freescale, Samsung, Texas.

Встроенные системы используют процессор для управления взаимодействиями с физической средой. Обычно они выстроены вокруг *микроконтроллеров* (МК), которые сочетают в себе микропроцессор с набором простых в использовании периферийных устройств, таких как цифровые и аналоговые пины ввода-вывода общего назначения, последовательные порты, таймеры и т.д. В большинстве своем МК недорогие и спроектированы так, чтобы минимизировать стоимость и размеры систем путем интеграции большинства необходимых компонентов в один чип. Большинство из них меньше и легче, чем

монета в десять центов, потребляет милливатты мощности, и цена их варьируется в пределах от нескольких десятков центов до нескольких долларов. Микроконтроллеры классифицируются по размеру данных, с которыми они оперируют. 8-битные микроконтроллеры – самые маленькие и самые дешевые, в то время как 32-битные микроконтроллеры предоставляют больше памяти и имеют большую производительность.

Для большей конкретики в этом разделе будет показан ввод-вывод во встроенной системе на примере коммерческого микроконтроллера. В частности, речь пойдет о PIC32MX675F512H, представителе серии Microchip PIC32 на базе 32-битного MIPS-микропроцессора. Семейство PIC32 также имеет широкий ассортимент чипов со встроенными периферийными устройствами и памятью, так что система может быть построена с малым количеством внешних компонентов. Мы выбрали это семейство, потому что оно имеет недорогую и легкую в использовании среду разработки, основано на архитектуре MIPS, изучаемой в этой книге, а также потому что Microchip является ведущим поставщиком микроконтроллеров, который продает более миллиарда чипов в год. Системы ввода-вывода микроконтроллера довольно похожи у разных производителей, поэтому принципы, проиллюстрированные на примере PIC32, легко могут быть применены к другим микроконтроллерам.

Оставшаяся часть этого раздела покажет читателям, как реализовать цифровой, аналоговый и последовательный ввод-вывод общего назначения с помощью микроконтроллера. Таймеры также широко используются для генерации или измерения точных временных интервалов. Раздел завершается примерами других интересных периферийных устройств, таких как дисплеи, моторы и беспроводные каналы.

8.6.1 Микроконтроллер PIC32MX675F512H

На **Рис. 8.29** показана блок-схема серии микроконтроллеров (МК) PIC32. В центре системы находится 32-разрядный процессор MIPS. Процессор подключается через 32-разрядные шины к флэш-памяти, содержащей программу, и к СОЗУ, содержащей данные. PIC32MX675F512H имеет 512 КБ флэш-памяти и 64 КБ оперативной памяти; другие разновидности с флэш-памятью от 16 до 512 КБ и оперативной памятью от 4 до 128 КБ доступны по различным ценам. Высокопроизводительные интерфейсы, такие как USB и Ethernet, также напрямую общаются с ОЗУ через матрицу шин данных. Периферия более низкой производительности, включающая последовательные порты, таймеры и аналого-цифровые преобразователи, разделяют между собой отдельную периферийную шину. Чип содержит цепь

генерации тактовой частоты и детектор напряжения для определения, когда чип включается или когда он близок к потере питания.

На **Рис. 8.30** показана виртуальная карта памяти микроконтроллера. Все адреса, используемые программистом, виртуальны. Архитектура MIPS предлагает 32-разрядное адресное пространство для доступа до 232 байт = 4 ГБ памяти, но лишь небольшая часть из этой памяти фактически реализована на чипе. Соответствующие разделы из адреса 0xA0000000–0xBFC02FFF включают в себя оперативную память, флэш и регистры специального назначения (Special Function Registers, SFR), используемые для связи с периферийными устройствами. Обратите внимание, что существует дополнительные 12 КБ загрузочной флэш-памяти, которая, как правило, выполняет некоторую инициализацию, а затем переходит к основной программе во флэш-памяти. При поступлении команды сброса счетчик команд инициализируется на начало загрузочной флэш-памяти с адреса 0xBFC00000.

На **Рис. 8.31** представлена схема расположения пинов микроконтроллера. Они включают в себя питание и землю, тактовый сигнал, сброс и множество пинов, которые можно использовать как пины ввода-вывода общего назначения и/или как периферийные устройства специального назначения.

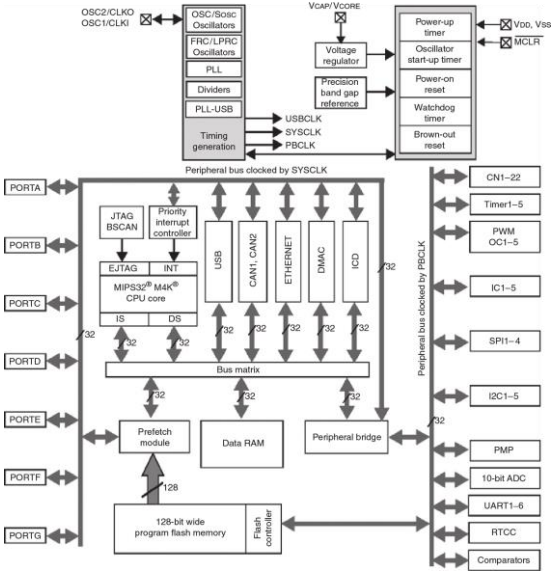


Рис. 8.29 **Блок-схема микроконтроллера PIC32MX675F512H**
(печатается с разрешения Microchip Technology ©, 2012)

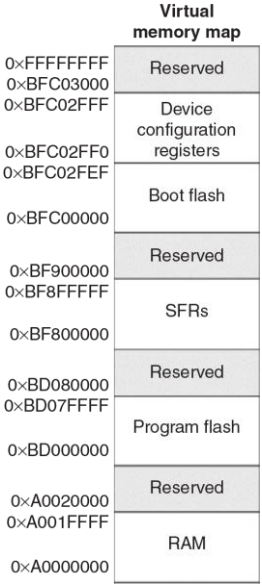


Рис. 8.30 **Карта памяти МК серии PIC32**
(печатается с разрешения Microchip Technology ©, 2012)

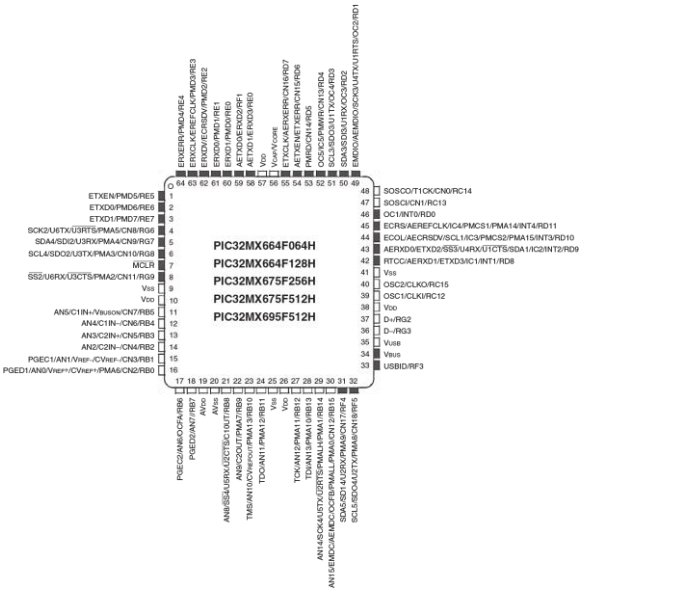


Рис. 8.31 **Схема расположения пинов микроконтроллеров PIC32MX6xxFxxH. К темным пинам может быть приложено напряжение до 5В.**

На **Рис. 8.32** – фотография микроконтроллера в 64-контактном корпусе TQFP (Thin Quad Flat Pack) с выводами вдоль четырех сторон, расположенными с интервалом 0,5 мм = 0,02 дюйма (20mil).

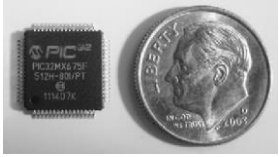


Рис. 8.32 **PIC 32 в корпусе TQFP с 64 контактами**

Микроконтроллер также доступен в 100-контактном корпусе с большим количеством пинов ввода-вывода; данная версия имеет обозначение с L на конце вместо H.

На **Рис. 8.33** показан микроконтроллер, подключенный в минимальной рабочей конфигурации с блоком питания, внешним тактирующим сигналом, переключателем сброса и разъемом для кабеля программатора. PIC32 и внешние цепи смонтированы на печатной плате; эта плата может быть фактическим продуктом (например, контроллером тостера) или отладочной платой, облегчающей доступ к чипу при тестировании.

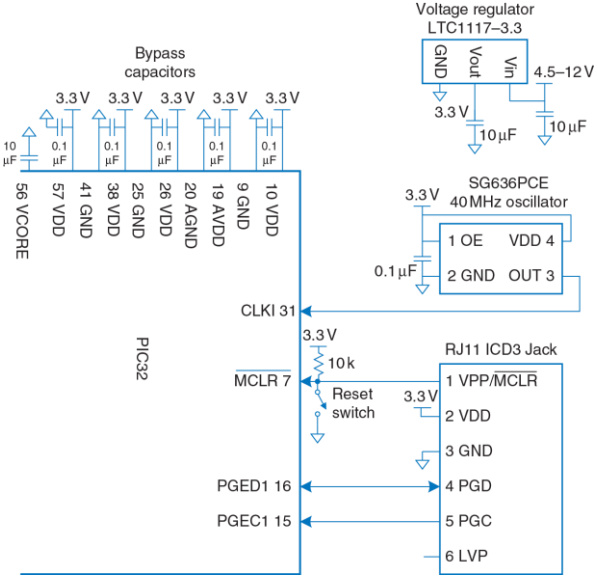


Рис. 8.33 **PIC32 в составе минимальной рабочей конфигурации**

Регулятор LTC1117-3.3 принимает на вход 4,5–12 В (например, от настенного трансформатора, батареи или внешнего источника питания) и понижает его до напряжения 3,3 В, необходимого для питания МК. PIC32 имеет несколько VDD- и GND-пинов, что уменьшает помехи по питанию и обеспечивает путь с низким импедансом. Набор проходных конденсаторов обеспечивает запас заряда, чтобы питание оставалось стабильным в случае внезапного роста потребления. Проходной конденсатор также соединяется с пином VCORE, который обслуживает внутренний 1,8-вольтовый регулятор напряжения. PIC32 обычно потребляет ток 1–2 мА / МГц от источника питания. Например, при 80 МГц максимальная рассеиваемая мощность $P = VI$ = (3,3В)(120 мА) = 0,396 Вт. 64-контактный TQFP имеет тепловое сопротивление 47 °C/Вт, так что чип может нагреваться примерно на 19 °C, если работает без теплоотвода или охлаждающего вентилятора.

Внешний генератор, выдающий частоту до 50 МГц, может быть соединен с пином тактовой частоты. В этой схеме, которую мы рассматриваем в качестве примера, генератор работает на частоте 40,000 МГц. Другой вариант: микроконтроллер может быть запрограммирован на использование внутреннего генератора, работающего на 8,00 МГц ± 2%. Это гораздо менее точная частота, но этого может быть достаточно. Тактовая частота периферийной шины PBCLK для устройств ввода-вывода (таких как последовательные

порты, АЦП, таймеры), как правило, работает на меньшей (например, половинной) скорости относительно основного системного тактового сигнала SYSCLK. Эта схема синхронизации может быть установлена с помощью конфигурационных битов в среде разработки MPLAB или путем размещения следующих строк кода в начале вашей программы, написанной на языке C.

```
#pragma config FPBDIV = DIV_2 // peripherals operate at half
                              // sysckfreq (20 MHz)
#pragma config POSCMOD = EC   // configure primary oscillator in
                              // external clock mode
#pragma config FNOSC = PRI  // select the primary oscillator
```

Периферийная тактовая шина PBCLK может быть настроена для работы на той же частоте, что и основной тактовый сигнал, или на половину, четверть или одну восьмую данной частоты. Ранние PIC32-микроконтроллеры были ограничены максимум половинной скоростью, потому что некоторые периферийные устройства были слишком медленными, но большинство современных продуктов поддерживают полную скорость. Если вы меняете скорость PBCLK, вам нужно изменить фрагменты кода-примера, приведенного в этом разделе, отвечающие за такие параметры как количество тактов PBCLK для измерения определенного промежутка времени.

Всегда удобно, когда у устройства есть кнопка сброса для перевода чипа в известное начальное состояние. Схема сброса состоит из кнопочного выключателя и резистора, подключенного к выводу сброса $\overline{MCLR}$. Активный уровень пина сброса – низкий, что означает срабатывание сброса процессора при подаче на пин логического 0. Если кнопка не нажата, переключатель разомкнут и резистор подтягивает пин сброса до 1, разрешая нормальную работу. При нажатии на кнопку, переключатель замыкается и тянет пин сброса до 0, заставляя процессор перезапуститься. PIC32 также сбрасывается автоматически при включении питания.

Проще всего программировать микроконтроллер с помощью Microchip *In Circuit Debugger* (ICD) 3, которую также ласково называют шайбой (puck). ICD3, показанная на **Рис. 8.34**, позволяет программисту связываться с PIC32 для загрузки кода и отладки программы напрямую с ПК.

ICD3 подключается к порту USB на ПК и в 6-контактный модульный разъем RJ-11 на отладочной плате PIC32. Разъем RJ-11 – разъем, знакомый по использованию в телефонных гнездах в США. ICD3 общается с PIC32 по 2-проводному последовательному внутрисхемному интерфейсу программирования (In-Circuit Serial Programming, ICSP) с пином тактового сигнала и двунаправленным

пином данных. Вы можете использовать свободно распространяемую компанией Microchip интегрированную в среду разработки MPLAB (IDE) для программирования на ассемблере или C, отладки программ при моделировании, установки и тестирования этих программ на отладочной плате посредством ICD.

Рис. 8.34 **Microchip ICD3**

Большинство контактов микроконтроллера PIC32 по умолчанию функционируют как цифровые пины ввода-вывода общего назначения. Поскольку чип имеет ограниченное количество пинов, эти же пины используются для специальных функций ввода-вывода, таких как последовательные порты, входы АЦП и т.д.; эти пины становятся активными, когда включено соответствующее периферийное устройство. Использовать каждый пин только для одной цели в любой

момент времени – обязанность программиста. В остальной части **раздела 8.6** детально исследуются функции ввода-вывода микроконтроллера.

Возможности МК выходят далеко за пределы того, что можно осветить в ограниченном пространстве этой главы. Ознакомьтесь с деталями в технической документации производителя МК. В частности, документация по семейству PIC32MX5XX / 6XX / 7XX и руководство пользователя семейства PIC32 от Microchip авторитетны и достаточно легко читаемы.

8.6.2 Цифровой ввод-вывод общего назначения

Пины ввода-вывода общего назначения (GPIO) используются для чтения или записи цифровых сигналов. Например, на **Рис. 8.35** показаны восемь светодиодов (LED) и четыре переключателя, подключенных к 12-битному GPIO-порту.

На схеме показаны названия и номер пина каждого из 12 выводов порта; отсюда программист узнает функции каждого пина, а разработчик аппаратной части – данные о том, какие соединения необходимо реализовать физически. Светодиоды подключены таким образом, чтобы светиться, когда на них приходит логическая «1», и затухают, когда приходит «0». Переключатели соединены

определенным образом для получения «1», когда они закрыты, и «0» – при открытом состоянии. Микроконтроллер может использовать порт как для управления светодиодами, так и для чтения состояний переключателей.

PIC32 организует группы GPIO в порты, которые считываются и записываются вместе. Наш PIC32 называет эти порты RA, RB, RC, RD, RE, RF, и RG; к ним обращаются просто как порт A, порт B и т.д. Каждый порт может иметь до 16 GPIO-пинов, хотя PIC32 не хватает пинов, чтобы обеспечить такое множество сигналов для всех своих портов.

Каждый порт управляется двумя регистрами: TRISx и PORTx, где x-метка (A–G) с указанием порта, представляющего интерес в данный момент. Регистры TRISx определяют, является ли пин порта вводом или выводом, в то время как регистры PORTx определяют значение, считываемое из порта ввода или подаваемые на выход. 16 младших битов каждого регистра соответствуют шестнадцати пинам порта GPIO. Когда данный бит регистра TRISx равен 0, пин становится выходом, а когда он равен 1 – входом. Разумно оставлять неиспользуемые пины GPIO в качестве входов (их состояние по умолчанию), чтобы они случайно не принимали нежелательных значений.

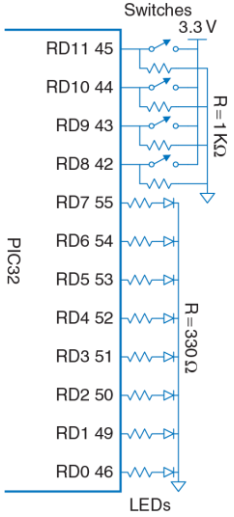


Рис. 8.35 Светодиоды и переключатели, подключенные к 12-ти разрядному порту общего назначения D

Каждый регистр отображается в память в виде слова в части виртуальной памяти, отведенной под Специальные Функциональные Регистры (Special Function Registers, SFR) (0xBF80000-BF8FFFF). Например, TRISD расположен по адресу 0xBF8860C0, а PORTD – по адресу 0xBF8860D0. Файл заголовка p32xxxx.h объявляет эти регистры 32-разрядными целыми числами без знака. Следовательно, программист может получить доступ к ним по имени, вместо того, чтобы искать адрес.

Пример 8.18 ПОРТЫ ВВОДА-ВЫВОДА GPIO ДЛЯ ПЕРЕКЛЮЧАТЕЛЕЙ И СВЕТОДИОДОВ

Напишите программу на C, которая бы считывала состояние четырех переключателей и включала четыре нижние светодиода в соответствии с положением переключателей. Используйте схему, показанную на **Рис. 8.35**.

Решение: Сконфигурируйте TRISD так, чтобы контакты RD[7:0] работали на вывод, а RD[11:8] – на ввод. Затем считайте состояние переключателей, проверяя данные на контактах RD[11:8], и выведите

это состояние на контакты RD[3:0], чтобы включить соответствующие светодиоды.

```
#include <p32xxxx.h>
int main(void) {
  int switches;
  TRISD = 0xFF00;                  // set RD[7:0] to output,
                                   // RD[11:8] to input
  while (1) {
    switches = (PORTD >> 8) & 0xF; // Read and mask switches from
                                   // RD[11:8]
   PORTD = switches;               // display on the LEDs
 }
}
```

Пример 8.18 записывает весь регистр сразу. Кроме того, можно получить доступ к отдельным битам. Например, следующий код копирует значение первого переключателя на первый светодиод.

```
PORTDbits.RD0 = PORTDbits.RD8;
```

Каждый порт также имеет соответствующие регистры SET и CLR, которые могут быть записаны с маской, указывающей, какие биты следует установить или очистить.

> В контексте манипуляции битами «установку» означает запись «1», а «очистку» означает запись «0».

Например,

```
PORTDSET = 0b0101;
PORTDCLR = 0b1000;
```

устанавливает первый и третий биты PORTD и очищает четвертый бит. Если нижние четыре бита PORTD были 1110, они станут 0111.

Число доступных выводов GPIO зависит от размера корпуса. **Табл. 8.5** суммирует данные о том, какие контакты доступны в различных корпусах. Например, 100-выводной TQFP предоставляет пины PA [15:14], PA [10:9], и PA [7:0] порта A. Помните, что RG [3: 2] являются только входами. Кроме того, RB [15:0] являются общими аналоговыми входами, и другие выводы тоже имеют несколько функций.

Табл. 8.5 Пины общего назначения МК PIC32MX5xx/6xx/7xx

Порт	64-pin QFN/TQFP	100-pin TQFP, 121-pin XBGA
RA	None	15:14, 10:9, 7:0
RB	15:0	15:0
RC	15:12	15:12, 4:0
RD	11:0	15:0
RE	7:0	9:0
RF	5:3, 1:0	13:12, 8, 5:0
RG	9:6, 3:2	15:12, 9:6, 3:0

Логические уровни являются LVCMOS-совместимыми. Входные контакты ожидают логические уровни V_{IL} = 0.15V_{DD} и VIH = 0.8V_{DD}, или 0,5 В и 2,6 В при условии, что V_{DD} составляет 3,3В. Выходные контакты производят V_{OL}, составляющий 0,4 В, и V_{OH} = 2,4 V, в то время как выходной ток Iвых не превышает ничтожные 7 мА.

8.6.3 Последовательный ввод-вывод

Если микроконтроллеру необходимо послать больше битов, чем количество свободных GPIO-пинов, ему понадобится разбить сообщение на несколько меньших посылок. На каждом шаге он может посылать либо один бит, либо несколько битов. Первый метод называется последовательным вводом-выводом, а второй – параллельным вводом-выводом. Последовательный ввод-вывод популярен, потому что он использует небольшое число проводов и достаточно быстр для множества задач. Действительно, он настолько популярен, что для последовательного ввода-вывода было установлено множество стандартов и в PIC32 выделено аппаратное обеспечение для простой пересылки данных по этим стандартам. Этот раздел описывает протоколы Последовательного Периферийного Интерфейса (Serial Peripheral Interface, SPI) и Универсального Асинхроннго Приемопередатчика (Universal Asynchronous Receiver/Transmitter, UART).

Другими распространенными последовательными стандартами являются Двупроводная Двунаправленная Шина (Inter-Integrated Circuit, I2C), Универсальная Последовательная Шина (Universal Serial Bus, USB) и Ethernet. I2C – двупроводной интерфейс с тактирующим сигналом и двунаправленным портом данных; он используется примерно тем же образом, что и SPI. USB и Ethernet – более сложные высокопроизводительные стандарты; они описаны в **разделах 8.7.1** и **8.7.4**, соответственно.

Последовательный Периферийный Интерфейс SPI

Последовательный Периферийный Интерфейс SPI – простой синхронный последовательный протокол, легкий в использовании и относительно быстрый. Физический интерфейс состоит из трех пинов: SerialClock (SCK), SerialDataOut (SDO) и Serial Data In (SDI).

SPI подключает ведущее устройство (master) к ведомому устройству (slave), как это показано на **Рис. 8.36 (а)**. Ведущее устройство производит сигнал тактовой частоты. Оно инициирует установку связи путем посылки серии импульсов тактовой частоты на SCK. Если оно хочет послать данные на ведомое устройство, оно посылает их на SDO, начиная со старшего бита. Ведомое устройство может одновременно ответить, посылая данные на SDI ведущего устройства. На **Рис. 8.36 (б)** показана диаграмма сигналов SPI для передачи 8 битов данных.

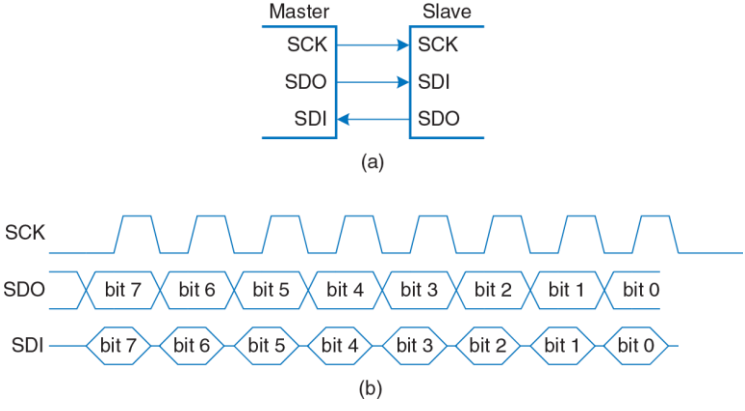


Рис. 8.36 Схема подключения по SPI интерфейсу и временная диаграмма ведущего устройства

У PIC32 есть до 4-х портов SPI, названных, что неудивительно, SPI1–SPI4. Каждый порт может выступать в роли ведущего или ведомого устройства. В данном разделе описан режим ведущего устройства, но режим ведомого похож на него. Чтобы использовать SPI-порт, программа PICR сначала должна сконфигурировать порт. Затем она может записывать данные в регистр; данные последовательно

передаются на ведомое устройство. Данные, полученные от ведомого устройства, собираются в другом регистре. Когда передача завершена, PIC32 может прочитать полученные данные.

Каждый SPI-порт ассоциируется с четырьмя 32-битными регистрами: SPIxCON, SPIxSTAT, SPIxBRG и SPIxBUF. Например, SPI1CON является управляющим регистром для SPI-порта 1. Он используется для включения SPI и установки атрибутов, таких как число битов, которые необходимо передать, и полярность тактового генератора. В **Табл. 8.6** перечислены имена и функции всех битов регистров CON. Все они имеют значение 0 по умолчанию после перезагрузки.

В этом разделе нет информации о большинстве функций, таких как кадрирование, расширенная буферизация, выбор сигналов ведомого устройства и прерывания, но подробно почитать о них вы можете в технической документации. STAT – регистр состояния, указывающий, например, заполнен ли принимающий регистр. Опять же, детальную информацию об этом регистре вы можете найти в документации PIC32.

Табл. 8.6 Поля регистра SPIxCON

Биты	Название	Назначение
31	FRMEN	1: Enable framing
30	FRMSYNC	Frame sync pulse direction control
29	FRMPOL	Frame sync polarity (1 = active high)
28	MSSEN	1: Enable slave select generation in master mode
27	FRMSYPW	Frame sync pulse width bit (1 = 1 word wide, 0 = 1 clock wide)
26:24	FRMCNT[2:0]	Frame sync pulse counter (frequency of sync pulses)
23	MCLKSEL	Master clock select (1 = master clock, 0 = peripheral clock)
22:18	Не используется	
17	SPIFE	Frame sync pulse edge select
16	ENHBUF	1: Enable enhanced buffering
15	ON	1: SPI ON
14	unused	
13	SIDL	1: Stop SPI when CPU is in idle mode
12	DISSDO	1: disable SDO pin
11	MODE32	1: 32-bit transfers
10	MODE16	1: 16-bit transfers

Биты	Название	Назначение
9	SMP	Sample phase (см. **Рис. 8.39**)
8	CKE	Clock edge (см. **Рис. 8.39**)
7	SSEN	1: Enable slave select
6	CKP	Clock polarity (см. **Рис. 8.39**)
5	MSTEN	1: Enable master mode
4	DISSDI	1: disable SDI pin
3:2	STXISEL[1:0]	Transmit buffer interrupt mode
1:0	SRXISEL[1:0]	Receive buffer interrupt mode

Последовательный тактовый сигнал может быть сконфигурирован для переключения с частотой, составляющей половину от частоты периферийного тактового сигнала или меньше. У SPI нет теоретического предела по скорости передачи данных, и он может работать на тактовой частоте в десятки МГц, хотя и может испытывать проблемы с шумом при использовании тактовой частоты больше 1МГц. BRG – регистр скорости передачи данных, который устанавливает скорость SCK по отношению к периферийной тактовой частоте согласно следующей формуле:

$$f_{SP1} = \frac{f_{peripheral\_clock}}{2 \times (\text{BRG}+1)} \qquad (8.3)$$

BUF – буфер данных. Данные, записываемые в BUF, передаются через SPI-порт на SDO-пин, а полученные с SDI-пина данные могут быть получены путем считывания буфера после завершения передачи данных. Чтобы приготовить SPI к работе в режиме ведущего устройства, сначала выключите его обнулением пятнадцатого бита CON-регистра (бита включения). Вычистите все, что может оказаться в принимающем буфере путем считывания BUF-регистра. Выставьте желаемую скорость передачи данных путем записи BRG-регистра. Например, если периферийная тактовая частота составляет 20МГц и требуемая частота передачи данных составляет 1,25 МГц, установите BRG на (20 / (2 × 1,25)) – 1 = 7. Переведите SPI в режим ведущего устройства установкой пятого бита CON-регистра (MSTEN) на 1. Установите восьмой бит CON-регистра (CKE) так, чтобы SDO был центрирован по отношению к фронту тактовой частоты. Наконец, включите SPI снова путем установки ON-бита CON-регистра в 1.

Чтобы отправить данные на ведомое устройство, запишите данные в BUF-регистр. Данные будут переданы последовательно, и ведомое устройство одновременно пошлет данные ведущему. Дождитесь, пока одиннадцатый бит STAT-регистра (бит SPIBUSY) станет равен нулю, показывая, что SPI завершил свою операцию. Тогда данные, полученные от ведомого устройства, смогут быть прочитаны из BUF-регистра.

Скорость передачи данных (baudrate) показывает скорость отправки сигналов, которая измеряется в символах в секунду, в то время как битрейт (bitrate) показывает скорость передачи данных, которая измеряется в битах в секунду. Отправка сигналов, которую мы обсуждали в данном тексте, является двухуровневой, где каждый символ представляет бит. Тем не менее, многоуровневая связь может отправить несколько бит за символ; например, 4-уровневая система передачи сигналов посылает два бита на символ. В этом случае битрейт в два раза превышает скорость передачи данных. В простой системе, где каждый символ является битом и каждый символ представляет данные, скорость передачи данных равна скорости передачи битов. Некоторые соглашения по передаче сигналов требуют дополнительных битов к данным (см **раздел 8.6.3** об UART). Например, двухуровневая система передачи сигналов, который использует два дополнительных бита на каждые 8 бит данных со скоростью передачи, равной 9600, имеет скорость передачи данных (9600 символов/с) × (8 бит/10 символов) = 7680 бит/с.

SPI-порт на PIC32 является конфигурируемым, так что он может связываться с разнообразными последовательными устройствами. К сожалению, это ведет к возможности установки неверной конфигурации порта и передаче испорченных данных. Относительная синхронизация тактовой частоты и сигналы данных конфигурируются с помощью трех CON-регистров, называемых CKP, CKE, и SMP. По умолчанию, эти биты равны нулю, но временная диаграмма на

Рис. 8.36 (б) использует CKE=1. Ведущее устройство меняет SDO на заднем фронте SCK, чтобы ведомое проводило выборку значений на переднем фронте, используя триггер, синхронизируемый по положительному фронту. Ведущее устройство ожидает, что SDI установится вблизи переднего фронта SCK, чтобы ведомое могло изменить его по спаду, как показано на временной диаграмме. В битах MODE32 и MODE16 регистра CON указывают, какое слово должно быть отправлено – 32-битное или 16-битное; по умолчанию оба этих бита устанавливаются в 0, указывая на 8-битную передачу.

> SPI всегда посылает данные в оба направления при каждой передаче. Если системе необходима лишь односторонняя связь, она может проигнорировать ненужные данные. Например, если ведущему устройству требуется лишь послать данные на ведомое, то байт, полученный от ведомого, может быть проигнорирован. Если ведущее устройство должно только получить данные от ведомого, он все еще должен запустить SPI-связь путем посылки произвольного байта, который ведомое проигнорирует. Затем он может считать данные, полученные от ведомого устройства.

Пример 8.19 ПЕРЕДАЧА И ПРИЕМ БАЙТОВ ЧЕРЕЗ ИНТЕРФЕЙС SPI

Разработайте систему для обмена данными между ведущим контроллером PIC® и ведомым FPGA через интерфейс SPI. Набросайте в общем виде схему интерфейса. Напишите на C программу для микроконтроллера, которая

посылает символ ‘A’ в FPGA и принимает его обратно. Напишите код на HDL для той части интерфейса SPI, которая находится в FPGA. Как можно упростить эту часть интерфейса, если требуется только получать данные?

Решение: На **Рис. 8.37** показано, как соединить устройства, используя порт 2 SPI. Номера контактов определяются по схемам расположения ножек на микросхемах (например, см. **Рис. 8.31**). Обратите внимание, что на схеме приведены номера контактов и названия сигналов, чтобы показать, как физические, так и логические соединения. Эти же контакты также используются для порта RG[8:6] линейки GPIO. Когда интерфейс SPI включен, эти разряды порта G не могут быть использованы для GPIO.

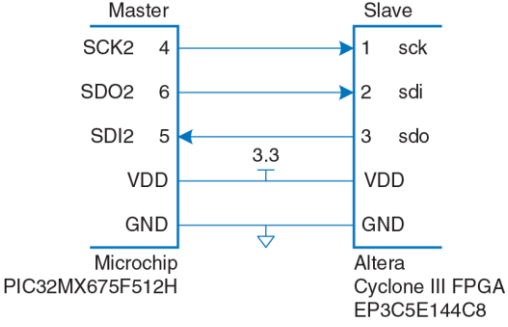


Рис. 8.37 **Связь между PIC32 и FPGA при помощи SPI интерфейса**

Приведенная ниже программа на C инициализирует интерфейс SPI, а затем отправляет и принимает однобайтовый символ.

```
#include <p32xxxx.h>
void initspi(void) {
  char junk;
  SPI2CONbits.ON = 0;     // disable SPI to reset any previous state
  junk = SPI2BUF;         // read SPI buffer to clear the receive buffer
  SPI2BRG = 7;
  SPI2CONbits.MSTEN = 1; // enable master mode
  SPI2CONbits.CKE = 1;    // set clock-to-data timing
  SPI2CONbits.ON = 1;     // turn SPI on
}
char spi_send_receive(char send) {
  SPI2BUF = send;               // send data to slave
  while (SPI2STATbits.SPIBUSY); // wait until SPI transmission complete
  return SPI2BUF;               // return received data
}
int main(void) {
  char received;
  initspi();                          // initialize the SPI port
  received = spi_send_receive('A'); // send letter A and receive byte
                                      // back from slave
}
```

HDL-код для FPGA приведен ниже, а внутренняя схема и временная диаграмма показаны на **Рис. 8.38**. FPGA использует регистр сдвига для временного хранения битов, поступающих от ведущего контроллера, и тех битов, которые будут отправлены обратно. После сигнала reset, начиная с первого переднего фронта

сигнала SCK и в течение 8-ми его циклов, в сдвиговый регистр загружается новый байт из D. На каждом последующем цикле сигнала, один бит принимается из SDI на FPGA и один бит записывается в SDO. SDO работает с задержкой и выводимый бит находится в SDO до завершения заднего фронта SCK, так что он может быть считан ведущим устройством на следующем фронте синхронизирующего сигнала. После 8 импульсов сигнала SCK, принятый байт будет находиться в Q.

```
module spi_slave(input  logic       sck,   // from master
                 input  logic       sdi,   // from master
                 output logic       sdo,   // to master
                 input  logic       reset, // system reset
                 input  logic [7:0] d,     // data to send
                 output logic [7:0] q);    // data received
  logic [2:0] cnt;
  logic       qdelayed;
  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else   cnt = cnt + 3'b1;
  // loadable shift register
  // loads d at the start, shifts sdi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], sdi} : {q[6:0], sdi};
  // align sdo to falling edge of sck
  // load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign sdo = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

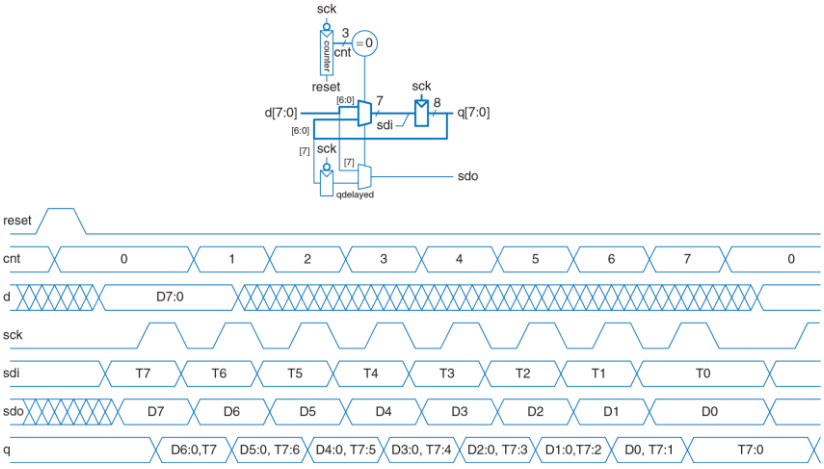


Рис. 8.38 Блок-схема ведомого устройства и его временная диаграмма

Если ведомое устройство должно только принимать данные от ведущего, то HDL-код можно упростить, обойдясь лишь сдвиговым регистром.

```
module spi_slave_receive_only(input logic        sck,  // from master
                              input logic        sdi,  // from master
                              output logic [7:0] q);   // data received
  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

Иногда необходимо изменить конфигурационные биты для связи с устройством, ожидающим другую синхронизацию. Когда CKP = 1, SCK инвертируется. При CKE = 0, такты переключаются на полцикла раньше по отношению к данным. Когда SAMPLE = 1, ведущее устройство осуществляет выборку SDI на полцикла позже (и ведомое устройство должно убедиться, что он стабилен в тот момент). Эти режимы показаны на **Рис. 8.39**.

Необходимо учитывать, что различные версии SPI могут использовать разные имена и полярности для этих вариантов; тщательно проверьте временную диаграмму для вашего устройства. Также может быть полезно проверить SCK, SDO и SDI осциллографом, если у вас возникают проблемы с соединением.

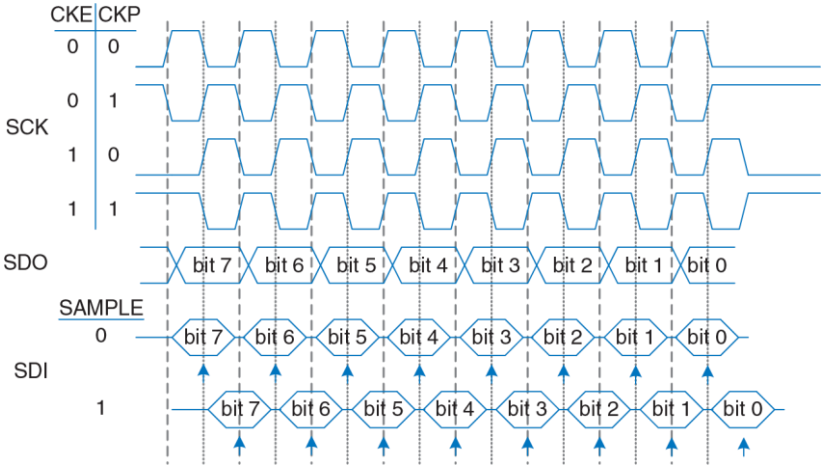


Рис. 8.39 **Временные диаграммы синхросигнала и данных, контролируемые сигналами CKE, CKP и SAMPLE**

Универсальный Асинхронный Приемопередатчик (UART)

UART (произносится как "ю-арт") является периферийным устройством последовательного ввода-вывода без пересылки тактового сигнала. Вместо этого системы должны заранее договориться о скорости передачи данных, и каждая из них должна локально генерировать свои собственный тактовый сигнал. Хотя эти системные тактовые сигналы могут иметь небольшую погрешность частоты и неизвестное соотношение фаз, UART обеспечивает надежную асинхронную связь. UART используется в таких протоколах, как RS-232 и RS-485. Например, компьютерные последовательные порты используют стандарт RS-232C, введенный в 1969 году Ассоциацией электронной промышленности (Electronic Industries Association). Стандарт первоначально предполагалось использовать для подключения Терминального Оборудования Данных (Data Terminal Equipment, DTE), таких как мейнфреймы, к устройствам обмена данными (Data Communication Equipment, DCE), например, модемам. Хотя UART относительно медленный по сравнению с SPI, стандарты применялись так долго, что они остаются важными и сегодня.

На **Рис. 8.40 (a)** показана асинхронная последовательная связь. DTE посылает данные в DCE по линии TX и получает данные обратно по линии RX. На **Рис. 8.40 (b)** показана одна из этих линий, отправляющая символ со скоростью передачи данных 9600 бод.

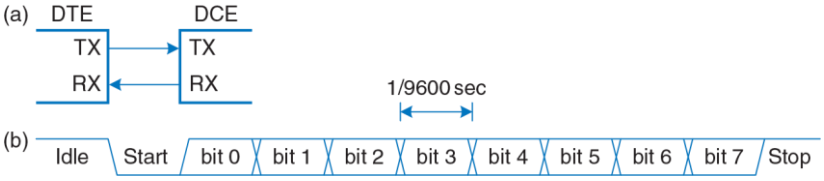


Рис. 8.40 **Асинхронная последовательная связь**

Линия простаивает при логической "1", когда не используется. Каждый знак состоит из стартового бита (0), 7–8 битов данных, опционального бита четности и одного или более стоп-битов (1). UART детектирует перепад сигнала от состояния простоя до старта для своевременного начала осуществления передачи. Хотя семи битов данных достаточно, чтобы отправить ASCII-символ, как правило, используется восемь битов, потому что они могут передать произвольный байт данных.

Дополнительный бит, *бит четности*, также может быть отправлен, позволяя системе проверить, не был ли какой-нибудь бит поврежден во время передачи. Он может быть сконфигурирован на *чётность* или *нечётность*; контроль чётности означает, что бит четности выбирается так, чтобы общий набор данных и бита чётности имел чётное число единиц. Другими словами, бит четности является исключающим ИЛИ (XOR) для битов данных. Приемник может затем проверить, было ли

получено чётное число единиц или нет и сигнализировать об ошибке. Нечетность является обратной величиной, и, следовательно, является функцией XNOR для битов данных.

Обычно останавливаются на 8 битах данных без контроля четности и 1 стоп-бите, что в общей сложности составляет 10 символов для передачи 8-битного знака информации. Следовательно, скорость подачи сигналов измеряется скорее в единицах бод, а не бит/сек. Например, 9600 бит указывает, 9600 символов/сек или 960 знаков/сек, что дает скорость передачи данных 960×8 = 7680 бит данных/сек. Обе системы должны быть настроены на соответствующую скорость передачи и количество данных, четности и стоп-бита, в противном случае данные будут искажены. Это является препятствием, особенно для пользователей без хорошей технической подготовки, что явилось одной из причин, по которой универсальная последовательная шина Universal Serial Bus (USB) заменила UART в персональных компьютерах.

Типичные скорости передачи данных включают 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600 и 115200 бод. Малые значения скорости использовались в 1970-х и 1980-х годах для модемов, которые посылали данные по телефонным линиям как последовательности тонов. В современных системах 9600 и 115 200 являются двумя наиболее распространенными скоростями передачи; 9600 встречается там, где скорость не имеет значения, а 115 200 является самой быстрой

стандартной скоростью, хотя по-прежнему медленной, по сравнению с другими современными стандартами последовательного ввода-вывода.

В 1950–1970-х годах первые хакеры, называвшие себя телефонными фриками, научились контролировать переключатели телефонных компаний свистом соответствующего тона. Тон в 2600 Гц, производимый игрушечным свистком из коробки с кукурузными хлопьями Капитан Кранч (Рисунок 8.41), мог быть использован для осуществления бесплатной междугородной и международной связи.

Рис. 8.41 Боцманский свисток капитана Кранча.
Напечатано с разрешения Evrim Sen

Стандарт RS-232 определяет несколько дополнительных сигналов. Сигналы запроса на передачу (Request to Send, RTS) и готовности к передаче (Clear to Send, CTS) могут использоваться для аппаратного подтверждения связи (handshaking). Они могут работать в одном из двух режимов. В *режиме управления потоком*, DTE сбрасывает RTS в 0, когда он готов к приему данных от DCE. Точно так же, DCE устанавливает CTS в 0, когда он готов к приему данных от DTE. Некоторые технические описания используют надчеркивание, чтобы

указать, что разрешающий сигнал – низкого уровня. В старом режиме односторонней передачи (симплексном) DTE обнуляет RTS, когда он готов к передаче. DCE отвечает обнулением CTS, когда он готов принимать передачу.

Некоторые системы, особенно связанные по телефонной линии, также используют Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR) и Ring Indicator (RI), чтобы указать, когда оборудование подключено к линии.

Оригинальный стандарт рекомендовал массивный 25-контактный DB-25 разъем, но ПК рационализировал разъём до 9-контактного DE-9 штыревого типа с распиновкой, показанной на **Рис. 8.42 (а)**. Кабельные провода обычно подключаются напрямую, как показано на **Рис. 8.42 (б)**. Однако когда непосредственно соединяются два DTE, может понадобиться *нуль-модемный* кабель, показанный на **Рис. 8.42 (с)**, чтобы поменять RX и TX и завершить подтверждение связи.

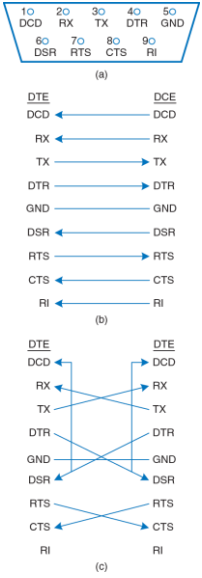


Рис. 8.42 Кабель DE-9:a) распиновка, b) стандартноесоединение, c) нуль-модемное соединение

Вдобавок, некоторые разъемы бывают штыревыми, а некоторые гнездовыми. В итоге может потребоваться большая коробка кабелей и определенное количество «угадайки» для соединения двух систем по интерфейсу RS232, что снова объясняет переход к USB. К счастью, встраиваемые системы, как правило, используют упрощенную 3- или 5-проводную установку, состоящую из GND, TX, RX, и, возможно, RTS и CTS.

> Подтверждение связи (от англ. handshaking – «рукопожатие») относится к «переговорам» между двумя системами; обычно одна система сообщает о том, что готова послать или получить данные, а другая система отвечает на запрос подтверждением.

RS-232 для логического «0» имеет электрический эквивалент с 3 до 15 В, а для «1» – с –3 до –15 В; это называется биполярной системой подачи сигналов. Приемопередатчик преобразует цифровые логические уровни UART с положительным и отрицательным уровнями, ожидаемыми RS-232, а также обеспечивает защиту от электростатического разряда для предотвращения повреждения последовательного порта, когда пользователь подключает кабель. MAX3232E является приёмопередатчиком, совместимым с 3,3 и 5 В цифровой логикой. Он содержит схему накачки заряда (charge pump), что, в сочетании с внешними конденсаторами, генерирует ±5 В выходы из одного источника питания низкого напряжения.

PIC32 имеет шесть UART-модулей, названных U1-U6. Как и с SPI, программе PIC® необходимо сначала сконфигурировать порт. В отличие от SPI, чтение и запись могут происходить независимо, потому что либо система может осуществлять передачу без получения, либо наоборот. Каждый UART связан с пятью 32-разрядными регистрами: UxMODE, UxSTA (статус), UxBRG, UxTXREG и UxRXREG. Например, U1MODE является регистром режима для UART1. Регистр режимов используется для настройки UART, а регистр STA используется для проверки, когда данные доступны. Реестр BRG используется для установки скорости передачи. Данные передаются или принимаются записью TXREG или чтением RXREG.

Регистр MODE по умолчанию устанавливает 8 бит данных, 1 стоп-бит, без проверки четности и без управления потоком RTS / CTS, поэтому для большинства приложений программист заинтересован только в 15 бите, бите ON, который включает UART.

Регистр STA содержит биты, активирующие пины приёма и передачи и проверяет, заполнены ли буферы приёма и передачи. После установки бита ON UART в 1, программист должен также установить UTXEN- и URXEN-биты (биты 10 и 12) из регистра STA, чтобы включить эти два пина. UTXBF (бит 9) указывает, что буфер передачи заполнен. URXDA (бит 0) указывает, что буфер приема имеет доступные данные. Регистр STA также содержит биты для индикации ошибок чётности и

кадрирования; ошибки кадрирования возникают, если старт- или стоп-биты не найдены в ожидаемое время. 16-битный регистр BRG используется для установки скорости передачи данных до долей частоты шины периферийного такового сигнала.

$$f_{UART} = \frac{f_{peripheral\_clock}}{16 \times (BRG+1)} \quad (8.4)$$

В **Табл. 8.7** перечислены настройки BRG для общецелевых скоростей передачи данных в предположении, что периферийная тактовая частота равна 20МГц. Иногда невозможно достичь с высокой точностью целевой скорости передачи данных.

Тем не менее, так как погрешность частоты значительно меньше, чем 5%, погрешность фаз между передатчиком и приемником будет оставаться небольшой по длительности 10-битового кадра, так что данные получаются правильным образом. Система будет пересинхронизироваться по следующему стартовому биту.

Чтобы передать данные, подождите, пока STA.UTXBF не укажет ясно, что буфер передачи имеет свободное пространство, а затем запишите байт в TXREG. Для получения данных проверьте STA.URXDA, чтобы увидеть, поступают ли данные, а затем прочитайте байт из RXREG.

Табл. 8.7 Настройки BRG для тактовой периферийной частоты 20 МГц

Целевая скорость передачи данных, бод	BRG	Реальная скорость передачи данных, бод	Ошибка
300	4166	300	0.0%
1200	1041	1200	0.0%
2400	520	2399	0.0%
9600	129	9615	0.2%
19200	64	19231	0.2%
38400	32	37879	−1.4%
57600	21	56818	−1.4%
115200	10	113636	−1.4%

Связь с последовательным портом от программы, написанной на C, на ПК немного хлопотна, потому что библиотеки драйверов последовательных портов не стандартизированы в различных операционных системах. Другие среды программирования, такие как Python, Matlab, или LabVIEW делают последовательную связь безболезненной.

Пример 8.20 ПОСЛЕДОВАТЕЛЬНЫЙ ОБМЕН ДАННЫМИ С КОМПЬЮТЕРОМ

Разработайте схему и напишите программу на C для PIC32, которая будет осуществлять обмен данными с персональным компьютером через последовательный порт со скоростью передачи 115200 бод в формате: 8 бит

данных, 1 стоп-бит и без бита паритета. На компьютере должна быть запущена консоль, например PuTTY[17], для передачи данных через последовательный порт. Программа должна запрашивать пользователя ввести любую строку, а затем напечатать в консоли эту строку.

Решение: На **Рис. 8.43** показана схема последовательного соединения. Поскольку компьютеры с последовательными портами уже не выпускаются, то мы используем внешний адаптер Plugable USB to RS-232 DB9 Serial Adapter фирмы plugable.com, показанный на **Рис. 8.44**. Этот адаптер подключается кабелем с разъемом-"мамой" DE-9 к трансиверу в PIC32, который преобразует биполярные напряжения RS-232 к уровню 3.3 В в микроконтроллере.

Для этого примера мы выбрали UART2 в PIC32. Микроконтроллер и персональный компьютер являются терминальными устройствами (Data Terminal Equipment), поэтому контакты TX и RX должны быть соединены накрест. Аппаратный контроль передачи данных RTS/CTS в PIC32 не используется, так что контакты RTS и CTS на разъеме DE9 соединены друг с другом и персональный компьютер всегда будет получать подтверждение передачи.

При настройке PuTTY для работы с последовательным соединением, установите Connection type как Serial и Speed – 115200. Для Serial line укажите номер COM-порта, назначенного операционной системой для USB адаптера. В операционной системе Windows, номер порта можно найти в Device Manager. Например, это может быть COM3. В табе Connection → Serial укажите контроль передачи данных как NONE или RTS/CTS. В табе Terminal установите Local Echo

[17] PuTTY доступен для скачивания и свободного использования на сайте www.putty.org.

в Force On, чтобы символы отображались в консоли по мере того, как вы их набираете на клавиатуре.

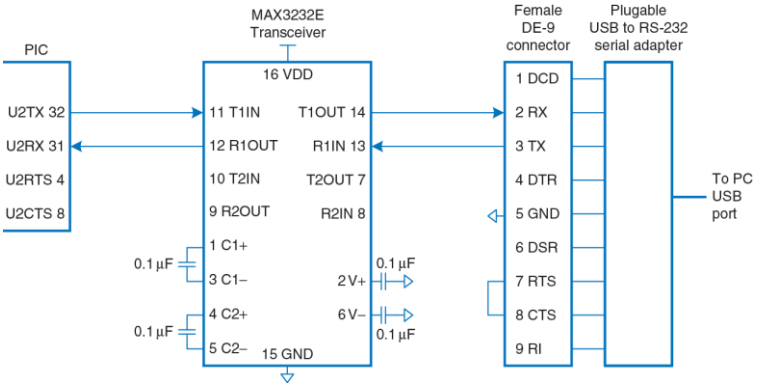


Рис. 8.43 **Подключение PIC32 к PC при помощи последовательного интерфейса**

Текст программы приведен ниже. Клавиша Enter соответствует в языке C символу возврата каретки ‘`\r`’ с ASCII-кодом 0x0D. Чтобы перевести позицию печати в

начало следующей строки, отправьте на вывод два символа ‘`\n`‘ and ‘`\r`‘ ("новая строка" и "возврат каретки").[18]

Рис. 8.44 **Съемный последовательный адаптер USB – RS-232 DB9**
(печатается с разрешения Plugable Technologies ©, 2012)

Функции для инициализации последовательного порта и операций чтения/записи в/из него являются простым драйвером устройства. Драйвер устройства реализует дополнительный уровень абстракции и модульности между программистом и оборудованием, поэтому программисту нет нужды погружаться в изучение набора регистров UART. Это также упрощает портирование кода для другого микроконтроллера. Драйвер устройства нужно будет переписать, а весь программный код, использующий внешний интерфейс драйвера останется без изменений.

Функция `main` выполняет печать в консоль и чтение из консоли, используя функции `putstrserial` и `getstrserial`. В ней также демонстрируется использование функции `printf` из `stdio.h`, которая автоматически печатает через UART2. К сожалению, библиотеки для PIC32 недостаточно хорошо

[18] PuTTY выполняет переход на новую строку корректно даже без символа `\r`.

поддерживают работу функции `scanf` через UART, но функция `getstrserial` работает вполне удовлетворительно.

```
#include <P32xxxx.h>
#include <stdio.h>
void inituart(void) {
  U2STAbits.UTXEN = 1;        // enable transmit pin
  U2STAbits.URXEN = 1;        // enable receive pin
  U2BRG = 10;                 // set baud rate to 115.2k
  U2MODEbits.ON = 1;          // enable UART
}
char getcharserial(void) {
  while (!U2STAbits.URXDA);   // wait until data available
  return U2RXREG;             // return character received from
                              // serial port
}
void getstrserial(char *str) {
  int i = 0;
  do {                         // read an entire string until detecting
    str[i] = getcharserial(); // carriage return
  } while (str[i++] != '\r');   // look for carriage return
  str[i-1] = 0;               // null-terminate the string
}
void putcharserial(char c) {
  while (U2STAbits.UTXBF);  // wait until transmit buffer empty
  U2TXREG = c;              // transmit character over serial port
}
void putstrserial(char *str) {
  int i = 0;
  while (str[i] != 0) {     // iterate over string
```

```
    putcharserial(str[i++]); // send each character
  }
}
int main(void) {
  char str[80];
  inituart();
  while(1) {
    putstrserial("Please type something: ");
    getstrserial(str);
    printf("\n\rYou typed: %s\n\r", str);
  }
}
```

8.6.4 Таймеры

Встроенные системы обычно нуждаются в измерении времени. Например, микроволновой печи необходим один таймер для отслеживания времени суток и другой, чтобы определить, как долго готовить. Она может использовать еще один таймер, чтобы генерировать импульсы двигателя, вращающего блюдо, а четвертый – чтобы контролировать уровень мощности, активируя микроволны лишь на долю каждой секунды.

PIC32 имеет пять 16-разрядных таймеров на плате. Таймер 1 называется таймером типа A, который может принять асинхронный внешний источник сигнала синхронизации, например, кварцевый

генератор, дающий частоту 32 кГц. Таймеры 2/3 и 4/5 имеют тип B. Они работают синхронно с периферическим сигналом синхронизации и могут работать в паре (например, 2 с 3), чтобы образовать 32-разрядные таймеры для измерения длительных периодов времени.

Каждый таймер связан с тремя 16-разрядными регистрами: TxCON, TMRx и PRX. Например, T1CON является регистром управления для Таймера 1. CON является регистром управления. TMR содержит текущее значение счетчика времени. PR является регистром периода. Когда таймер достигнет указанного периода, он откатывается обратно в 0 и устанавливает бит TXIF в регистре флагов прерывания IFS0. Программа может опрашивать этот бит для обнаружения переполнения. Кроме того, он может генерировать прерывание.

По умолчанию, каждый таймер действует как 16-разрядный счетчик, накапливая такты внутренних периферийных часов (20 МГц в нашем примере).

Пятнадцатый бит CON-регистра, называемый ON-битом, начинает отсчёт таймера. TCKPS-биты регистра CON указывают меру деления частоты (предварительное деление), как это представлено в **Табл. 8.8** и **Табл. 8.9** для таймеров типа A и типа B.

Предварительное деление на k:1 заставляет таймер делать отсчет лишь раз в k тактов; это может быть полезно для генерации более

длительных промежутков времени, особенно тогда, когда периферийная частота синхронизации высока.

Остальные биты CON-регистров немного отличаются для счетчиков типа A и типа B; обратитесь к технической документации для деталей.

Табл. 8.8 Предварительное деление частоты для таймеров типа A

TCKPS[1:0]	Prescale
00	1:1
01	8:1
10	64:1
11	256:1

Табл. 8.9 Предварительное деление частоты для таймеров типа B

TCKPS[2:0]	Prescale
000	1:1
001	2:1
010	4:1
011	8:1
100	16:1
101	32:1
110	64:1
111	256:1

Другой полезной особенностью таймера является режим закрытого накопления времени, в котором таймер ведет отсчёт, лишь пока на внешнем пине поддерживается высокий уровень. Это позволяет таймеру измерить длительность внешнего импульса. Разрешение на использование этого режима получается через CON-регистр.

Пример 8.21 ГЕНЕРАЦИЯ ЗАДЕРЖКИ

Напишите две функции для создания задержки на указанное количество микросекунд и миллисекунд, используя Timer1. Внешний таймер работает на частоте 20 MHz.

Решение: В одной микросекунде – 20 тактовых циклов внешнего таймера. Мы установили эмпирически, используя осциллограф, что накладной расход функции `delaymicros` на инициализацию таймера и вызов самой функции составляет примерно 6 микросекунд. Таким образом, мы устанавливаем значение PR в 20 × (`micros` – 6). То есть, функция будет работать некорректно для задержек менее 6 микросекунд. Функции выполняет проверку значения задержки, чтобы предотвратить переполнение PR с размером 16 бит.

Функция `delaymillis` выполняет в цикле вызов `delaymicros(1000)` для создания серии задержек по 1 миллисекунд каждая.

```
#include <P32xxxx.h>
void delaymicros(int micros) {
  if (micros >1000) {              // avoid timer overflow
    delaymicros(1000);
    delaymicros(micros-1000);
  }
  else if (micros >6){
    TMR1 = 0;                      // reset timer to 0
    T1CONbits.ON = 1;              // turn timer on
    PR1 = (micros-6)*20;           // 20 clocks per microsecond.
                                   // Function has overhead of ~6 us
    IFS0bits.T1IF = 0;             // clear overflow flag
```

```
    while (!IFS0bits.T1IF);        // wait until overflow flag is set
  }
}
void delaymillis(int millis) {
  while (millis--) delaymicros(1000);// repeatedly delay 1 ms until done
}
```

8.6.5 Прерывания

Таймеры часто используются в сочетании с прерываниями таким образом, что программа может работать как обычно и затем периодически обрабатывать задачу, когда таймер генерирует прерывание. В **разделе 6.7.2** описаны MIPS-прерывания с архитектурной точки зрения. В этом разделе рассматривается, как использовать их в PIC32.

Запросы на прерывание возникают, когда происходит событие в аппаратной части, такое как переполнение таймера, получение символа по UART или переключение определенных выводов GPIO. Каждый тип запроса прерывания устанавливает конкретный бит в регистры Статуса флага прерывания (Interrupt Flag Status, IFS). Затем процессор проверяет соответствующий бит в регистрах разрешения прерываний управления (Interrupt Enable Control, IEC). Если бит установлен, микроконтроллер должен ответить на запрос прерывания посредством вызова программы обработки прерывания (Interrupt Service Routine,

ISR). ISR представляет собой функцию с аргументами пустого типа данных (void), которая обрабатывает прерывание и очищает бит из IFS до возврата из обработки. Система прерываний PIC32 поддерживает одно- и многовекторные режимы. В одновекторном режиме все прерывания вызывают один и тот же ISR, который должен проверить регистр CAUSE, чтобы определить причину прерывания (если может возникнуть несколько типов прерываний) и обрабатывать его соответствующим образом. В многовекторном режиме каждый тип прерывания вызывает свой ISR. MVEC-бит в регистре INTCON определяет режим. В любом случае, система прерываний MIPS должна быть включена с командой EI, прежде чем она будет принимать какие-либо прерывания.

PIC32 также позволяет каждому источнику прерываний иметь настраиваемый приоритет и субприоритет. Приоритет выставляется в диапазоне 0–7, где 7 – наивысший приоритет. Прерывание с более высоким приоритетом приостановит обработку прерывания, которое имело место в текущий момент. Например, предположим, для прерывания UART установлен приоритет 5, а для прерывания таймера – приоритет 7. Если программа выполняется в нормальном режиме и на UART появляется символ, то произойдет прерывание, и микроконтроллер сможет считать данные из UART и обработать их. Если таймер переполнится в то время когда UART ISR активен,

ISR будет прерван, так что микроконтроллер сможет сразу обработать переполнение таймера. Когда это будет сделано, он вернется к завершению прерывания UART, прежде чем вернуться к основной программе. С другой стороны, если прерывание таймера имеет приоритет 2, UART ISR завершится первым, а затем будет вызван ISR таймера, и, наконец, микроконтроллер вернется к основной программе. Субприоритет находится в диапазоне 0–3. Если два события с одинаковым приоритетом рассматриваются одновременно, в первую очередь будет обработано событие с наивысшим субприоритетом. Тем не менее, субприоритет не вызовет новое прерывание для упреждения прерывания того же приоритета, которое обрабатывается в данный момент. Приоритет и субприоритет каждого события конфигурируется с помощью IPC-регистров.

Каждый источник прерываний имеет номер вектора в диапазоне 0–63. Например, прерывание по переполнению Таймера 1 является вектором 4, прерывание UART2 RX является вектором 32, а внешнее прерывание INT0, которое инициируется при изменении на выводе RD0, является вектором 3. Поля регистров IFS, IEC и IPC, соответствующие номеру этого вектора, указаны в документации PIC32. Объявление функции ISR активируется двумя специальными директивами `(__attribute__)`, указывающими уровень приоритета и номер вектора. Компилятор использует эти атрибуты, чтобы связать ISR с

соответствующим запросом прерывания. Руководство пользователя Microchip MPLAB® C Compiler для PIC32 микроконтроллеров содержит больше информации о написании процедуры обработки прерываний.

Пример 8.22 ПЕРИОДИЧЕСКИЕ ПРЕРЫВАНИЯ

Напишите программу для мигания светодиода с частотой 1 Гц, используя прерывания.

Решение: Мы реализуем алгоритм, который будет приводить к переполнению Timer1 каждые 0.5 секунд и переключать светодиод между состояниями ON и OFF в обработчике прерываний. Пример программы ниже демонстрирует многовекторный режим прерываний, несмотря на то, что разрешено только прерывание от Timer1. Функции `blinkISR` заданы атрибуты, указывающие, что она имеет уровень приоритета 7 (IPL7) и обслуживает вектор 4 (вектор переполнения Timer 1). ISR переключает светодиод. Перед возвратом из функции флаг переполнения таймера очищается – это бит Timer1 Interrupt Flag (T1IF) в регистре флагов прерывания IFS0.

Функция `initTimer1Interrupt` устанавливает для таймера период 0.5 секунд, используя делитель частоты 256:1 и счетчик на 39063 тактов. Функция также устанавливает многовекторный режим прерываний. Приоритет и субприоритет указываются в битах [4:2] и [1:0] регистра IPC1. Флаг прерывания Timer 1 (T1IF, bit 4 of IFS0) очищается, а флаг разрешения прерываний от Timer1 (T1IE, bit 4 of IEC0) активируется. В конце функции, директива `asm` генерирует инструкцию `ei`, чтобы включить прерывания.

После инициализации прерываний от таймера, функция `main` исполняет бесконечный цикл `while`. Хотя, она может делать и более интересные вещи одновременно. Например, играть в игру с пользователем в то же самое время, когда прерывания будут переключать светодиод с заданной периодичностью.

```
#include <p32xxxx.h>
// The Timer 1 interrupt is Vector 4, using enable bit IEC0<4>
// and flag bit IFS0<4>, priority IPC1<4:2>, subpriority IPC1<1:0>
void _ _attribute_ _((interrupt(IPL7))) _ _attribute_ _((vector(4)))
blinkISR(void) {
  PORTDbits.RD0 = !PORTDbits.RD0;  // toggle the LED
  IFS0bits.T1IF = 0;               // clear the interrupt flag
  return;
}
void initTimer1Interrupt(void) {
  T1CONbits.ON = 0;       // turn timer off
  TMR1 = 0;               // reset timer to 0
  T1CONbits.TCKPS = 3;    // 1:256 prescale: 20 MHz / 256 = 78.125 KHz
  PR1 = 39063;            // toggle every half-second (one second period)
INTCONbits.MVEC = 1;      // enable multi-vector mode - we' re using
                            vector 4
  IPC1 = 0x7 << 2 | 0x3; // priority 7, subpriority 3
  IFS0bits.T1IF = 0;      // clear the Timer 1 interrupt flag
  IEC0bits.T1IE = 1;      // enable the Timer 1 interrupt
  asm volatile("ei");     // enable interrupts on the micro-controller
  T1CONbits.ON = 1;       // turn timer on
}
int main(void) {
  TRISD = 0;              // set up PORTD to drive LEDs
```

```
  PORTD = 0;
  initTimer1Interrupt();
  while(1);              // just wait, or do something useful here
}
```

8.6.6 Аналоговый ввод-вывод

Реальный мир является аналоговым. Многие встроенные системы нуждаются в аналоговых входах и выходах для взаимодействия с миром. Они используют аналого-цифровые преобразователи (АЦП) для квантования аналоговых сигналов в цифровые значения и цифроаналоговые преобразователи (ЦАП), чтобы сделать обратное. На **Рис. 8.45** показаны условные графические обозначения для этих компонентов. Такие преобразователи характеризуются разрешением, динамическим диапазоном, частотой дискретизации и точностью.

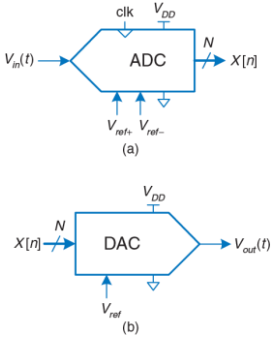


Рис. 8.45 **Условные обозначения АЦП и ЦАП**

Например, АЦП могут иметь N = 12-битное разрешение в диапазоне V_{ref-} к V_{ref+} от 0 до 5 В с частотой дискретизации f_s = 44 кГц и точностью ±3 младших значащих бита. Частота дискретизации также измеряется в выборках в секунду (samples per second, sps), где 1 sps = 1 Гц. Отношение между напряжением аналогового входа $V_{in}(t)$ и цифровой выборкой $X[n]$ можно выразить следующим образом:

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}}$$
$$n = \frac{t}{f_s} \tag{8.5}$$

Например, входное напряжение 2,5 В (половина от полной шкалы) будет соответствовать выходу 1000000000002 = 80016, с погрешностью до 3 младших значащих разрядов. Аналогично, ЦАП могут иметь N = 16-битное разрешение в полной шкале относительно V_{ref} = 2,56 В. Выходное напряжение в результате составляет:

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref} \tag{8.6}$$

Многие микроконтроллеры имеют встроенные АЦП умеренной производительности. Для более высокой производительности (например, 16-битным разрешением или частотой дискретизации свыше 1 МГц), часто необходимо использовать отдельный АЦП, соединенный с микроконтроллером. Меньшее число микроконтроллеров имеет встроенные ЦАП, поэтому также могут быть использованы отдельные чипы. Тем не менее, микроконтроллеры часто

имитируют аналоговые выходы, используя метод, называемый широтно-импульсной модуляцией (ШИМ). В этом разделе описываются такие аналоговые вводы-выводы в контексте PIC32-микроконтроллера.

Аналого-цифровое преобразование

PIC32 имеет 10-битный АЦП с максимальной скоростью 1 миллион выборок/с (Million samples per second, Msps). АЦП может подключиться к любому из 16-аналоговых входных контактов через аналоговый мультиплексор. Аналоговые входы называются AN0–15 и делят свои пины с цифровым портом ввода-вывода RB. По умолчанию, V_{ref+} – это потенциал пина аналогового питания V_{DD}, а V_{ref-} – потенциал пина аналоговой земли; в нашей системе это 3,3 и 0 В, соответственно. Программист должен инициализировать АЦП, указать, с каких пинов производить выборку, выждать достаточно долго, чтобы произвести выборку напряжения, начать преобразование, подождать его завершения и прочитать результат.

АЦП легко конфигурируются, имея возможность автоматического сканирования через несколько аналоговых входов с программируемыми интервалами и генерацией прерывания по завершении. Этот раздел кратко описывает, как производить считывание с одного пина аналогового входа; обратитесь к PIC32 Family Reference Manual для получения дополнительной информации о других особенностях.

> Обратите внимание, что у АЦП есть путаница в использовании понятий частоты дискретизации и времени выборки. Время выборки, также называемое временем захвата, является промежутком времени, необходимым для установки входа до того, как начнется преобразование. Частота дискретизации – это количество выборок в секунду. Она составляет не более 1 / (время выборки + 12 TAD).

АЦП управляется набором регистров: AD1CON1–3, AD1CHS, AD1PCFG, AD1CSSL и ADC1BUF0-F. AD1CON1 является основным регистром управления. У него есть ON-бит для включения АЦП, SAMP-бит для контроля при выборке и преобразовании, и DONE-бит для определения завершения преобразования. AD1CON3 имеет биты ADCS[7:0] для управления скоростью аналого-цифрового преобразования. AD1CHS является регистром выбора канала аналогового входа, для которого необходимо проводить выборку. AD1PCFG является регистром конфигурации пинов. Когда бит равен 0, соответствующий пин действует как аналоговый вход. Когда он равен 1, пин действует как цифровой вход. ADC1BUF0 содержит 10-битный результат преобразования. Другие регистры не требуются в нашем простом примере.

АЦП использует регистр последовательного приближения, который производит один бит результата на каждом такте АЦП. Требуются два дополнительных цикла – итого 12 тактов АЦП на преобразование.

Период тактового сигнала TAD должен быть не менее 65 нс для корректной работы. Он устанавливается кратным периоду периферийной тактовой частоты TPB с помощью битов ADCS в соответствии с соотношением:

$$T_{AD} = 2T_{PB}(\text{ADCS} + 1) \quad (8.7)$$

Следовательно, для периферийных тактовых сигналов до 30 МГц биты ADCS можно оставить значениями по умолчанию – нулями. Время выборки является промежутком времени, необходимым для стабилизации нового входного сигнала до начала его преобразования. Т.к. сопротивление источника, от которого производится выборка, меньше 5 кОм, время выборки может составлять всего 132 нс – небольшое число тактов.

Пример 8. 23 АНАЛОГОВЫЙ ВВОД

Напишите программу для чтения аналогового значения на контакте AN11.

Решение: Функция `initadc` инициализирует аналого-цифровой преобразователь (АЦП) и выбирает указанный аналоговый вход. После этого АЦП находится в режиме выборки напряжений. Функция `readadc` считывает напряжение и запускает преобразование. Она ожидает окончания преобразования, затем снова переводит АЦП в режим выборки и возвращает результат аналого-цифрового преобразования.

```
#include <P32xxxx.h>
void initadc(int channel) {
  AD1CHSbits.CH0SA = channel;  // select which channel to sample
  AD1PCFGCLR = 1 << channel;   // configure pin for this channel to
                               // analog input
  AD1CON1bits.ON = 1;          // turn ADC on
  AD1CON1bits.SAMP = 1;        // begin sampling
  AD1CON1bits.DONE = 0;        // clear DONE flag
}
int readadc(void) {
  AD1CON1bits.SAMP = 0;        // end sampling, start conversion
  while (!AD1CON1bits.DONE);   // wait until done converting
  AD1CON1bits.SAMP = 1;        // resume sampling to prepare for next
                               // conversion
  AD1CON1bits.DONE = 0;        // clear DONE flag
  return ADC1BUF0;             // return conversion result
}
int main(void) {
```

```
  int sample;
  initadc(11);
  sample = readadc();
}
```

Цифроаналоговое преобразование

PIC32 не имеет встроенного ЦАП, так что этот раздел описывает цифроаналоговое преобразование с помощью внешних ЦАП. Он также иллюстрирует сопряжение PIC32 с другими чипами через параллельные и последовательные порты. Такой же подход можно было бы использовать для сопряжения PIC32 с АЦП более высокого разрешения или более быстрыми внешними АЦП. Некоторые ЦАП принимают *N*-битный цифровой вход на параллельном интерфейсе с *N* проводами, в то время как другие принимают его через последовательный интерфейс, такой как SPI. Некоторые ЦАП требуют как положительного, так и отрицательного напряжения питания, в то время как другие работают от одного источника. Некоторые поддерживают гибкий диапазон напряжений питания, в то время как другие требуют определенное напряжение. Входные логические уровни должны быть совместимы с цифровым источником. Некоторые ЦАП дают выходное напряжение, пропорциональное цифровому входу, в то время как другие производят токовый выход; может понадобиться операционный усилитель для преобразования этого тока в напряжение в требуемом диапазоне.

В этом разделе мы используем 8-битный параллельный ЦАП AD558 фирмы Analog Devices и 12-битный последовательный ЦАП LTC1257 от Linear Technology. Оба просты в применении, производят напряжение на выходах, работают от одного источника питания 5–15В используют V_{IH} = 2,4 В, так что они совместимы с выходами 3,3В из PIC32, поставляются в DIP-корпусах, которые легко монтировать на печатную плату. AD558 производит выходной сигнал на уровне от 0 В до 2.56 В, потребляет мощность 75 мВт, поставляется в 16-выводном корпусе, имеет время установки 1 мкс и скорость преобразования 1 Мвыборок/с. С технической документацией можно ознакомиться на сайте tanalog.com. LTC1257 производит выходной сигнал на уровне от 0 В до 2.048 В, потребляет менее 2 мВт мощности, выпускается в 8-выводном корпусе, и его время установки составляет 6 мкс. Его SPI работает на максимальной частоте в 1,4 МГц. Техническое описание представлено на сайте linear.com. Texas Instruments является еще одним ведущим производителем АЦП и ЦАП.

Пример 8. 24 АНАЛОГОВЫЙ ВЫВОД ЧЕРЕЗ ВНЕШНИЙ ЦАП

Набросайте схему и напишите программу простого генератора сигналов синусоидальной и треугольной формы, используя PIC32, AD558 и LTC1257.

Решение: Схема показана на **Рис. 8.46**. AD558 подключен к параллельному 8-битному порту RD в контроллере PIC32. Линии Vout Sense и Vout Select подключены к Vout, которая имеет полный диапазон выходных напряжений до

2.56 В. LTC1257 подключается к PIC32 через порт SPI2. Оба ЦАП запитываются напряжением 5В и шунтируются развязывающим конденсатором 0.1 мкФ для снижения помех от блока питания. Низкие активные уровни сигнала готовности ЦАП и сигнала загрузки на ЦАП запускают преобразование входного цифрового значения. Во время загрузки нового значения в ЦАП, оба сигнала должны иметь высокий уровень.

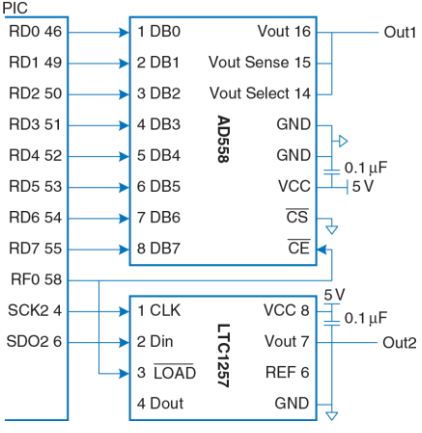


Рис. 8.46 Подключение ЦАП к МК PIC32 при помощи параллельного и последовательного интерфейса

Ниже приводится пример программы. Функция initio инициализирует параллельный и последовательный порты, а также устанавливает таймер для генерации напряжений через определенные интервалы времени. Порт SPI установлен в 16-битный режим на частоте 1 МГц, хотя LTC1257 обрабатывает только последние принятые 12 бит. Функция initwavetables генерирует массив заранее вычисленных констант для представления синусоидальных и треугольных волн. Синусоидальные волны имеют 12-битную точность, а треугольные волны определены 8-битными значениями. Каждая волна аппроксимируется по 64 точкам на длину волны. Увеличение этого значения повышает точность формы волны, но снижает максимально возможную частоту генерируемых сигналов. Функция genwaves выполняет цикл по массиву со значениям уровней напряжений. Для каждого значения, она снимает сигналы CE и LOAD для обоих ЦАП, отправляет новое значение по параллельному и последовательному портам, активирует оба ЦАП и затем ожидает, когда таймер выдаст сигнал, что настало время отправлять очередное значение напряжения. Минимальная частота синусоидальных и треугольных волн – 5 Гц. Она определяется тем, что регистр периода в Timer1 имеет размер 16 бит, а максимальная частота 605 Гц (38.7 Ксемплов/сек) установлена в функции genwaves. Она ограничивается, главным образом, скоростью передачи данных через интерфейс SPI.

```
#include <P32xxxx.h>
#include <math.h>            // required to use the sine function
#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];

void initio(int freq) {      // freq can be 5-605 Hz
```

```
  TRISD = 0xFF00;             // make the bottom 8 bits of PORT D outputs
  SPI2CONbits.ON = 0;          // disable SPI to reset any previous state
  SPI2BRG = 9;                // 1 MHz SPI clock
  SPI2CONbits.MSTEN = 1;      // enable master mode
  SPI2CONbits.CKE = 1;        // set clock-to-data timing
  SPI2CONbits.MODE16 = 1;     // activate 16-bit mode
  SPI2CONbits.ON = 1;         // turn SPI on
TRISF = 0xFFFE;               // make RF0 an output to control load and ce
  PORTFbits.RF0 = 1;             // set RF0 = 1
  PR1 = (20e6/NUMPTS)/freq - 1; // set period register for desired wave
                                 // frequency
  T1CONbits.ON = 1;              // turn Timer1 on
}

void initwavetables(void) {
  int i;
  for (i=0; i<NUMPTS; i++) {
    sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1);    // 12-bit scale
    if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;     // 8-bit scale
    else     triangle[i] = 510-i*511/NUMPTS;
  }
}

void genwaves(void) {
  int i;
  while (1) {
    for (i=0; i<NUMPTS; i++) {
      IFS0bits.T1IF = 0;        // clear timer overflow flag
      PORTFbits.RF0 = 1;        // disable load while inputs are changing
      SPI2BUF = sine[i];        // send current points to the DACs
```

```
      PORTD = triangle[i];
      while (SPI2STATbits.SPIBUSY);// wait until transfer completes
      PORTFbits.RF0 = 0;           // load new points into DACs
      while (!IFS0bits.T1IF);      // wait until time to send next point
    }
  }
}
int main(void) {
  initio(500);
  initwavetables();
  genwaves();
}
```

Широтно-импульсная модуляция

Другим способом генерировать аналоговый выходной сигнал в цифровой системе является широтно-импульсная модуляция (ШИМ), в которой периодический выходной сигнал принимает высокое значение в течение части периода передачи и низкое для оставшейся части. Доля периода, для которой сигнал имеет высокое значение, называется коэффициентом заполнения (duty cycle), см. **Рис. 8.47**. Среднее значение на выходе пропорционально коэффициенту заполнения. Например, если выходные перепады разбросаны между 0 и 3,3 В и коэффициент заполнения составляет 25%, среднее значение будет 0,25 × 3,3 = 0,825 В. Низкочастотная фильтрация сигнала ШИМ исключает колебания, и выходной сигнал принимает требуемое среднее значение.

PIC32 содержит пять модулей сравнения выходных сигналов OC1–OC5, и каждый, в сочетании с Таймером 2 или 3, может производить ШИМ-сигналы на выходе.[19] Каждый модуль сравнения связан с тремя 32-разрядными регистрами: OCxCON, OCxR и OCxRS. CON является регистром управления. OCM-биты CON-регистра должны быть установлены в 1102, чтобы активировать режим ШИМ, и ON-бит должен быть установлен в 1. По умолчанию, модуль сравнения выходного сигнала использует Таймер 2 в 16-битном режиме, но биты OCTSEL и OC32 могут быть использованы для выбора Таймера 3 и/или 32-битном режиме. В режиме ШИМ RS устанавливает коэффициент заполнения, регистр периода таймера PR устанавливает период, а OCxR может быть проигнорирован.

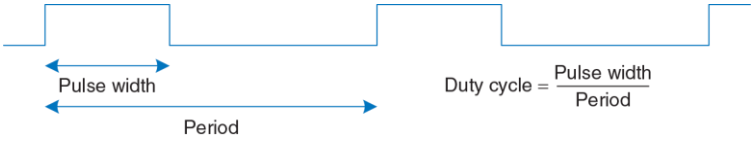


Рис. 8.47 ШИМ-сигнал

[19] Модули сравнения выходных сигналов можно сконфигурировать для формирования одиночных импульсов с использованием таймера

Пример 8.25 АНАЛОГОВЫЙ ВЫВОД С ИСПОЛЬЗОВАНИЕМ ШИМ

Напишите функцию для генерации аналогового вывода в виде создания некоторого уровня напряжения с использованием широтно-импульсной модуляции (ШИМ) и внешнего RC-фильтра. Аргументом функции является целое число в диапазоне от 0 и 256. Эти значения должны соответствовать уровням напряжения на выходе в диапазоне от 0В до 3.3В.

Решение: Используйте модуль OC1 для генерации сигнала с частотой 78.125 КГц на выходном контакте OC1. Фильтр нижних частот на **Рис. 8.48** имеет угловую частоту

$$f_c = \frac{1}{2\pi RC} = 1.6\text{KHz}$$

Фильтр подавляет высокие частоты и сглаживает напряжение на выходе.

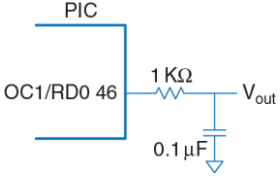


Рис. 8.48 **Аналоговый выход, использующий ШИМ и фильтр нижних частот**

Таймер должен работать на частоте 20 МГц с периодом 256 тактов, поскольку 20 МГц/256 дает нужную частоту дискретизации ШИМ 78.125 КГц. Для задания рабочего цикла ШИМ указывается количество тактов, в течение которых сигнал на выходе должен иметь высокий уровень. Если это значение равно 256 или больше, то уровень сигнала на выходе будет высоким постоянно.

Программа для ШИМ использует OC1 и Timer2. В регистр периода записывается 255 для периода в 256 тактов. В OC1RS определяется коэффициент заполнения ШИМ. Таким образом, OC1 сконфигурирован для режима ШИМ, а таймер и выходной компаратор включены. Во время работы ШИМ, программа одновременно может выполнять другие действия. На выходе OC1 будет удерживаться постоянное напряжение до тех пор, пока ШИМ не будет выключен программно.

```
#include <P32xxxx.h>
void genpwm(int dutycycle) {
  PR2 = 255;                  // set period to 255+1 ticks = 78.125 KHz
  OC1RS = dutycycle;          // set duty cycle
  OC1CONbits.OCM = 0b110;     // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;           // turn on timer 2 in default mode (20 MHz,
                              // 16-bit)
  OC1CONits.ON = 1;           // turn on output compare 1 module
}
```

8.6.7 Другие внешние устройства микроконтроллера

Микроконтроллеры часто взаимодействуют с другими внешними периферийными устройствами. В этом разделе приведены различные распространенные примеры таких устройств, в том числе символьные жидкокристаллические дисплеи (LCD), VGA мониторы, беспроводные линии Bluetooth и управление двигателем. Стандартные интерфейсы связи, включая USB и Ethernet, описаны **в разделах 8.7.1** и **8.7.4**.

Символьный ЖК-дисплей

Символьный ЖК-дисплей – небольшой жидкокристаллический дисплей, способный показывать одну или несколько строк текста. Они широко используются в передних панелях приборов, таких как кассовые аппараты, лазерные принтеры и факсы, которые должны отображать лишь ограниченное количество информации. Они легко взаимодействуют с микроконтроллером через параллельный интерфейс, RS-232 или SPI. Crystalfontz America продает широкий спектр символьных ЖК-дисплеев, начиная от 8 столбцов × 1 строку до 40 столбцов × 4 строк с выбором цвета, подсветки, питанием 3,3 / 5 В и видимостью при дневном свете. Эти ЖК-дисплеи могут стоить $20 или более в розницу и менее 5$ при больших объемах покупки.

В этом разделе приведён пример взаимодействия PIC32-микроконтроллера с символьным ЖК-дисплеем по 8-битному

параллельному интерфейсу. Интерфейс совместим с LCD контроллером HD44780, являющимся промышленным стандартом, изначально разработанным Hitachi. На **Рис. 8.49** показан CFAH2002A-TMI-JT 20 × 2 параллельный ЖК-дисплей фирмы Crystalfontz.

Рис. 8.49 **ЖК-дисплей CFAH2002A-TMI 20 x 2 фирмы Crystalfontz**
(напечатано с разрешения Crystalfontz America ©, 2012)

На **Рис. 8.50** показан ЖК-дисплей, соединенный с PIC32 через параллельный 8-битный интерфейс. Логика у него работает на 5 В, но совместима с 3,3В входными сигналами от PIC32. Контрастность ЖК-дисплея устанавливается вторым напряжением, производимым с помощью потенциометра; как правило, экран наилучшим образом читается в диапазоне напряжений 4,2–4,8 В. ЖК-экран получает три управляющих сигнала: RS (1 – для символов, 0 – для команды), $R/\overline{W}$ (1 читать с дисплея, 0 записать) и E (поднимаемый до высокого

уровня, по крайней мере, на 250 нс для того, чтобы включить ЖК-дисплей, когда следующий байт будет готов). Когда читается команда, бит 7 возвращает флаг занятости, указывая 1, если ЖК-экран занят, и 0, когда он готов принять еще одну команду.

Тем не менее, определенные шаги инициализации и команды очистки экрана требуют определенной задержки вместо проверки флага занятости.

Для инициализации ЖК-экрана, PIC32 должен записать последовательность команд, как показано ниже:

- Подождать > 15000 мкс после включения VDD
- Записать 0x30 для установки 8-битного режима
- Подождите > 4100 мкс
- Записать 0x30, снова установить 8-битный режим
- Подождать > 100 мкс
- Записать 0x30 для установки 8-битного режима еще раз
- Подождите, пока флаг занятости не очистится
- Записать 0x3c для установки 2-х линий и 5×8 точечного шрифта

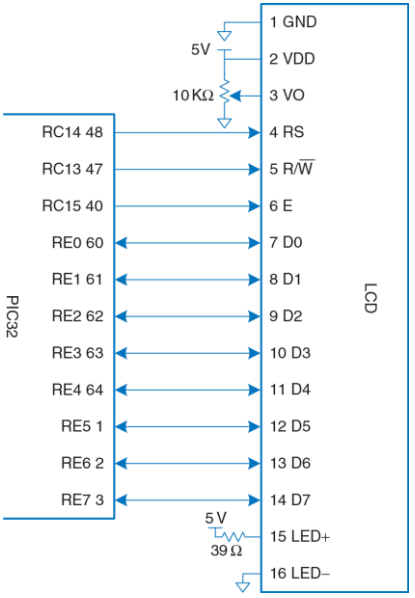


Рис. 8.50 **Подключение ЖК-дисплея при помощи параллельного интерфейса**

- Подождать, пока флаг занятости не очистится
- Записать 0x08, чтобы выключить дисплей
- Подождать, пока флаг занятости не очистится
- Записать 0x01 для очистки дисплея
- Подождать > 1530 мкс
- Записать 0x06, чтобы установить режим ввода для сдвига курсора после каждого символа
- Подождите, пока флаг занятости очистится
- Напишите 0x0C, чтобы включить дисплей без курсора

Затем, чтобы записать текст на ЖК-экран, микроконтроллер может отправить последовательность ASCII-символов. Также он может послать команды 0x01 для очистки дисплея или 0x02 для возвращения в «домашнюю» позицию в верхнем левом углу.

Пример 8.26 УПРАВЛЕНИЕ ЖК-ДИСПЛЕЕМ

Напишите программу для вывода строки “I love LCDs” на символьном ЖК-дисплее.

Решение: Программа, приведенная ниже, выводит строку “I love LCDs” ЖК-дисплее. Эта программа использует функцию delaymicros из **примера 8.21**.

```
#include <P32xxxx.h>
typedef enum {INSTR, DATA} mode;
char lcdread(mode md) {
  char c;
  TRISE = 0xFFFF;                 // make PORTE[7:0] input
  PORTCbits.RC14 = (md == DATA); // set instruction or data mode
  PORTCbits.RC13 = 1;             // read mode
  PORTCbits.RC15 = 1;             // pulse enable
  delaymicros(10);                // wait for LCD to respond
  c = PORTE & 0x00FF;             // read a byte from port E
  PORTCbits.RC15 = 0;             // turn off enable
  delaymicros(10);                // wait for LCD to respond
}

void lcdbusywait(void)
{
  char state;
  do {
    state = lcdread(INSTR);  // read instruction
  } while (state & 0x80);    // repeat until busy flag is clear
}
```

```
char lcdwrite(char val, mode md) {
  TRISE = 0xFF00;                  // make PORTE[7:0] output
  PORTCbits.RC14 = (md == DATA);   // set instruction or data mode
  PORTCbits.RC13 = 0;              // write mode
  PORTE = val;                     // value to write
  PORTCbits.RC15 = 1;              // pulse enable
  delaymicros(10);                 // wait for LCD to respond
  PORTCbits.RC15 = 0;              // turn off enable
  delaymicros(10);                 // wait for LCD to respond
}

char lcdprintstring(char *str)
{
  while(*str != 0) {       // loop until null terminator
   lcdwrite(*str, DATA);   // print this character
   lcdbusywait();
   str++;                  // advance pointer to next character in string
  }
}
void lcdclear(void)
{
  lcdwrite(0x01, INSTR); // clear display
  delaymicros(1530);     // wait for execution
}
void initlcd(void) {
  // set LCD control pins
  TRISC = 0x1FFF;         // PORTC[15:13] are outputs, others are inputs
  PORTC = 0x0000;         // turn off all controls
  // send instructions to initialize the display
  delaymicros(15000);
```

```
  lcdwrite(0x30, INSTR); // 8-bit mode
  delaymicros(4100);
  lcdwrite(0x30, INSTR); // 8-bit mode
  delaymicros(100);
  lcdwrite(0x30, INSTR); // 8-bit mode yet again!
  lcdbusywait();
  lcdwrite(0x3C, INSTR); // set 2 lines, 5x8 font
  lcdbusywait();
  lcdwrite(0x08, INSTR); // turn display off
  lcdbusywait();
  lcdclear();
  lcdwrite(0x06, INSTR); // set entry mode to increment cursor
  lcdbusywait();
  lcdwrite(0x0C, INSTR); // turn display on with no cursor
  lcdbusywait();
}
int main(void) {
  initlcd();
  lcdprintstring("I love LCDs");
}
```

VGA Монитор

Более гибкий вариант – управлять монитором компьютера. Стандарт монитора *Video Graphics Array* (VGA) был введен в 1987 году для компьютеров IBM PS/2, с разрешением 640 × 480 пикселей на электронно-лучевой трубке (Cathode Ray Tube, CRT) и 15-контактным разъемом, передающим цветовую информацию с помощью аналоговых

напряжений. Современные ЖК-мониторы имеют более высокое разрешение, но остаются обратно совместимы со стандартом VGA.

В электронно-лучевой трубке электронная пушка сканирует экран слева направо, активируя флуоресцентный материал для отображения изображения. Цветные ЭЛТ-экраны используют три разных типа люминофоров для красного, зеленого и синего цветов и три электронных луча. Мощность каждого луча определяет интенсивность каждого цвета в пикселе. В конце каждой строки развертки пушка должна выключиться на интервал горизонтального гашения, чтобы вернуться к началу следующей строки. После того как все растровые строки пройдены, пушку необходимо отключить снова для интервала вертикального гашения (обратного хода луча), чтобы вернуться в верхний левый угол. Процесс повторяется около 60–75 раз в секунду, чтобы дать визуальную иллюзию постоянного изображения.

В VGA-мониторе с разрешением 640 × 480 пикселей с частотой обновления кадра 59,94 Гц тактовая частота пикселя работает на 25,175 МГц, так что каждый пиксель имеет ширину в 39,72 нс. Полный экран можно рассматривать как 525 горизонтальных линий развертки в 800 пикселей каждая, но только 480 из линий развертки и 640 пикселей в строке развертки фактически передают изображение, в то время как остальные остаются черными. Сигнал сканирования начинается со строчного гасящего импульса СГИ (так называемого «заднего крыльца»,

back porch) – пустой секции в левой части экрана. Затем он включает 640 пикселей, после чего идёт *кадровый гасящий импульс* КГИ (так называемое *«переднее крыльцо»*, *front porch*) в правой части экрана и импульс горизонтальной синхронизации (Hsync), который быстро перемещает луч к левому краю. **Рис. 8.51 (a)** показывает синхронизацию каждого из этих участков строки развертки, начиная с активных пикселей. Вся линия сканирования занимает 31,778 мкс. В вертикальном направлении экран начинается с пустой области в верхней части, с последующим 480 активными строками развертки, с последующей пустой секцией и импульсом вертикальной синхронизацией (VSync) для возвращения в начало, чтобы стартовать следующий кадр. Новый кадр прорисовывается 60 раз в секунду. На **Рис. 8.51 (b)** показана вертикальная синхронизация; заметим, что единицы времени теперь являются строками развертки, а не пиксельными тактами. Более высокое разрешение использует и более высокочастотный тактовый генератор пикселей, до 388 МГц при 2048 × 1536 при 85 Гц. Например, 1024 × 768 при 60 Гц может быть достигнуто пиксельным генератором тактовой частоты 65 МГц. Горизонтальная синхронизация включает в себя КГИ из 24 тактов, Hsync-импульс из 136 тактов и СГИ из 160 тактов. Вертикальная временная диаграмма включает в КГИ из 3 линий сканирования, VSync-импульс из 6 линий и СГИ из 29 линий.

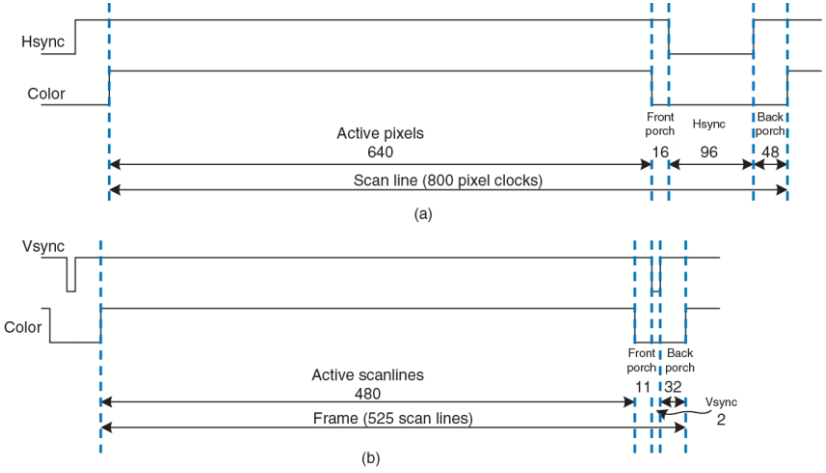


Рис. 8.51 **Временная диаграмма VGA-монитора a) горизонтальная, b) вертикальная**

На **Рис. 8.52** представлена схема расположения выводов для гнездового разъёма, поступающего из источника видео. Информация о пикселе передается тремя аналоговыми напряжениями для красного, зеленого и синего цветов. Каждое напряжение колеблется от 0,7 В,

увеличиваясь для повышения яркости пикселя. Напряжения должны быть равны 0 во время КГИ и СГИ.

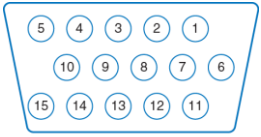


Рис. 8.52 **Распиновка VGA разъема**

Кабель может также обеспечить последовательную связь I2C для настройки монитора. Видеосигнал должен быть сгенерирован на высокой скорости в реальном времени, что трудно достичь на микроконтроллере, но легко на ПЛИС. Простой чёрно-белый дисплей может быть реализован подачей на все три цветовые пина напряжений с 0 или 0,7 В с помощью делителя напряжения, подключенного к цифровому выходному контакту. Цветной монитор, с другой стороны, использует видео-ЦАП с тремя отдельными ЦАП для независимой подачи напряжения на цветовые пины.

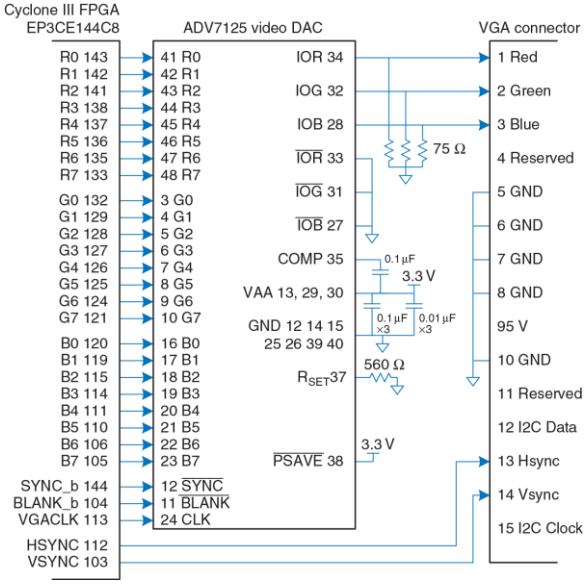


Рис. 8.53 **VGA кабель, управляемый ПЛИС через ЦАП**

Рис. 8.53 показывает, как ПЛИС управляет VGA-монитором через тройной 8-битный видео-ЦАП. ЦАП ADV7125 получает 8 бит R, G и B от ПЛИС. Он также получает сигнал SYNC_b, который устанавливается в низкий активный уровень, независимо от того, когда объявляются сигналы HSYNC или VSYNC. Видео-ЦАП производит три выходных тока для управления красной, зеленой и синей аналоговыми линиями, которые обычно являются 75-омными параллельными линиями передачи, терминируемые как на видео-ЦАП, так и мониторе. Резистор RSET устанавливает величину выходного тока для достижения полного спектра цвета. Тактовая частота зависит от разрешения и частоты обновления; которая может достигать значения 330 МГц с моделью быстродействующего ЦАП ADV7125JSTZ330.

Пример 8.27 ВЫВОД НА VGA МОНИТОР

Подготовьте HDL-код для вывода текста и зеленого прямоугольника на VGA монитор, используя схему, показанную на **Рис. 8.53**.

Решение: Мы предполагаем, что тактовая частота синхронизации равна 40 МГц и использует петлю фазовой синхронизации в FPGA для генерации синхросигнала VGA частотой 25.175 МГц. Реализация петли фазовой синхронизации может быть разной в различных FPGA; в Cyclone III, набор частот специфицирован в специальной настроечной функции от Altera. Как альтернативный вариант, синхросигнал VGA может поступать от внешнего генератора сигналов.

Контроллер VGA выполняет расчеты для строк и колонок на экране, генерируя сигналы HSYNC and VSYNC в нужные моменты времени. Он также генерирует сигнал BLANK_B, который гасит луч, чтобы пространство экрана за пределами рабочей области 640 × 480 было черным.

Видеогенератор вычисляет значения красного, зеленого и синего цветов для каждого пиксела на экране, адресуемого координатами (x, y). Точка начала координат (0, 0) расположена в левом верхнем углу экрана. Генератор выводит на экран набор символов и зеленый прямоугольник. Изображения символов имеют размер 8 × 8 пикселов. Таким образом, рабочая область экрана вмещает 80 × 60 символов. Генератор символов хранит изображения символов в своей постоянной памяти (ROM). Каждый символ представляется в виде таблицы размером 8 строк и 6 колонок, где находятся однобитовые значения 0 и 1. Ещё две колонки – пустые, то есть заполнены нулями. В коде SystemVerilog порядок битов обратный, так как самая левая колонка в ROM-файле предназначена для старшего бита, который должен отображаться в наименьшей горизонтальной X-координате символа.

На **Рис. 8.54** приводится снимок экрана VGA монитора во время выполнения этой программы. Цвет строк с буквами чередуется: одна строка красная, а другая – синяя. Зеленый прямоугольник накладывается на изображение на экране.

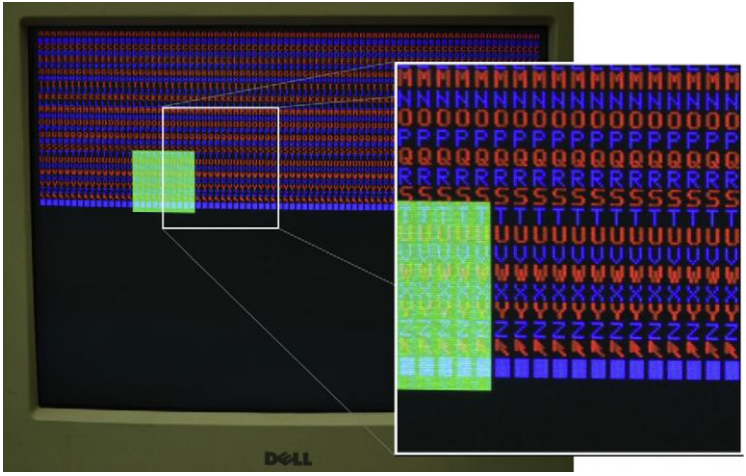


Рис. 8.54 Вывод VGA-монитора

```
module vga(input  logic      clk,
           output logic      vgaclk,          // 25.175 MHz VGA clock
           output logic      hsync, vsync,
           output logic      sync_b, blank_b, // to monitor & DAC
           output logic [7:0] r, g, b);       // to video DAC

  logic [9:0] x, y;

  // Use a PLL to create the 25.175 MHz VGA pixel clock
  // 25.175 MHz clk period = 39.772 ns
  // Screen is 800 clocks wide by 525 tall,
  // but only 640 x 480 used for display
  // HSync = 1/(39.772 ns *800) = 31.470 KHz
  // Vsync = 31.474 KHz / 525 = 59.94 Hz (~60 Hz refresh rate)
  pll vgapll(.inclk0(clk), .c0(vgaclk));

  // generate monitor timing signals
  vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

  // user-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
```

```
                                VFP     = 10'd11,
                                VSYN    = 10'd2,
                                VMAX    = VACTIVE + VFP + VSYN + VBP)
                        (input  logic      vgaclk,
                         output logic      hsync, vsync, sync_b, blank_b,
                         output logic [9:0] x, y);

  // counters for horizontal and vertical positions
  always @(posedge vgaclk) begin
   x++;
   if (x == HMAX) begin
     x = 0;
     y++;
     if (y == VMAX) y = 0;
   end
  end

  // compute sync signals (active low)
  assign hsync = ~(hcnt >= HACTIVE + HFP & hcnt < HACTIVE + HFP + HSYN);
  assign vsync = ~(vcnt >= VACTIVE + VFP & vcnt < VACTIVE + VFP + VSYN);
  assign sync_b = hsync & vsync;

  // force outputs to black when outside the legal display area
  assign blank_b = (hcnt <HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(inputlogic [9:0] x, y, output logic [7:0] r, g, b);

  logic       pixel, inrect;
```

```
  // given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue.Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen  rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
  assign g =   inrect   ? 8'hFF        : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic   pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;       // a line read from the ROM

  // initialize ROM with characters from text file
  initial
    $readmemb("charrom.txt", charrom);
  // index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}]; // subtract 65 because A
                                               // is entry 0
  // reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic       inrect);
  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

Беспроводная связь Bluetooth

В настоящее время есть много стандартов, доступных для беспроводной связи, в том числе Wi-Fi, ZigBee и Bluetooth. Эти стандарты детально проработаны и требуют сложных интегральных схем, но растущий ассортимент модулей позволяет абстрагироваться от сложности и предоставить пользователю простой интерфейс для беспроводной связи. Одним из этих модулей является BlueSMiRF, простой беспроводной интерфейс Bluetooth, который можно использовать вместо последовательного кабеля. Bluetooth является беспроводным интерфейсом, разработанным компанией Ericsson в 1994 году для маломощной связи на умеренной скорости на расстояниях 5–100 метров, в зависимости от уровня мощности передатчика. Он широко используется для подключения гарнитуры к телефону или клавиатуры к компьютеру. В отличие от инфракрасных каналов связи, он не требует прямой видимости между устройствами.

> Bluetooth назван в честь короля Дании Харальда Синий Зуб, монарха X века, который объединил враждующие датские племена. Этот беспроводной стандарт был лишь отчасти успешен в деле унификации множества конкурирующих протоколов беспроводной связи!

Bluetooth работает в диапазоне 2,4 ГГц нелицензионной промышленно-научно-медицинской (ISM) полосы. Он определяет 79 радиоканалов с интервалом в 1 МГц, начиная с 2402 МГц. Он переключается между

этими каналами в псевдослучайной последовательности, чтобы избежать постоянной помехи для других устройств, таких как мобильные телефоны, работающие в той же полосе. Как указано в **Табл. 8.10**, передатчики Bluetooth классифицируются по одному из трех уровней мощности, которые диктуют диапазон и энергопотребление.

В режиме основной скорости он работает на 1 Мбит/с, используя Гауссову частотную манипуляцию (FSK, Frequency Shift Keying). В обычном FSK каждый бит передается посредством передачи частоты fc±fd, где fc – центральная частота канала, а fd является частотой, смещенной по крайней мере на 115 кГц. Резкий переход в частотах между битами занимает дополнительную пропускную способность. В Гауссовой FSK изменение частоты сглаживается, чтобы более эффективно использовать спектр частот. **Рис. 8.55** показывает, как частоты передаются для последовательности нулей и единиц на канале в 2402 МГц, используя FSK и GFSK.

Табл. 8.10 Классы Bluetooth

Класс	Мощность передатчика, мВт	Дальность, м
1	100	100
2	2.5	10
3	1	5

Модуль BlueSMiRF Silver, как показано на **Рис. 8.56**, содержит Bluetooth-приемопередатчик класса 2, модем, и схеме интерфейса на маленькой карточке с последовательным интерфейсом. Он связывается с другим устройством Bluetooth, например, USB Bluetooth адаптером, подключенным к компьютеру.

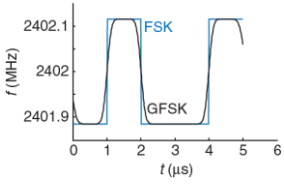


Рис. 8.55 **Временные диаграммы FSK и GFSK**

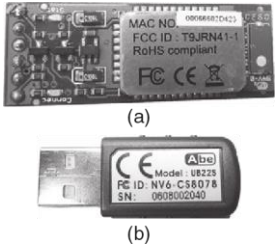


Рис. 8.56 **Модуль BlueSMiRF и карта с последовательным интерфейсом**

Таким образом, можно обеспечить беспроводную последовательную связь между PIC32 и ПК, аналогичную линии связи на **Рис. 8.43**, но без кабеля. На **Рис. 8.57** показана схема для такой связи. TX-вывод

BlueSMiRF подключается к RX-выводу PIC32, и наоборот. Выводы RTS и CTS соединены так, что BlueSMiRF «пожимает собственную руку».

В BlueSMiRF по умолчанию работает при 115.2k бод с 8 бит данных, 1 стоп-битом, без проверки четности или управления потоком. Он работает от 3,3 В цифровых логических уровней, так что не нужно никаких RS-232 приёмопередатчиков для соединения с другим устройством, работающим от 3,3 В.

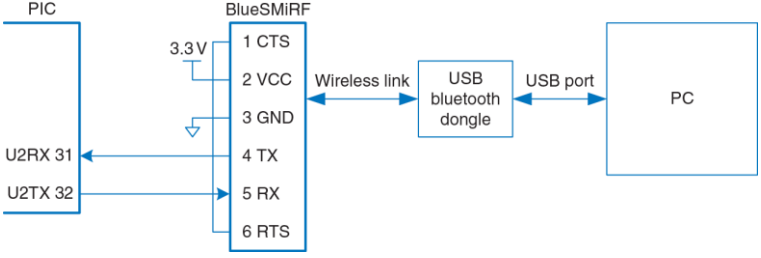


Рис. 8.57 **Схема соединения между PIC32 и ПК**

Чтобы использовать этот интерфейс, подключите USB-адаптер Bluetooth к ПК. Включите PIC32 и BlueSMiRF. Красный STAT-огонёк будет мигать на BlueSMiRF, указывая, что он ждёт, чтобы образовать связь. Откройте значок Bluetooth в системном трее ПК и используйте

функцию ADD Bluetooth Device Wizard, чтобы выполнить сопряжение адаптера с BlueSMiRF. По умолчанию ключ доступа для BlueSMiRF – 1234. Обратите внимание, какой COM-порт назначен защитной заглушкой. Тогда связь будет работать так, как это было бы в случае с последовательным кабелем. Обратите внимание, что адаптер обычно работает со скоростью 9600 бод и что PuTTY должен быть настроен соответствующим образом.

Управление двигателями

Еще одним важным применением микроконтроллеров является управление приводами, такими как двигатели. В этом разделе описываются три типа двигателей: двигатели постоянного тока, серводвигатели и шаговые двигатели. *Двигатели постоянного тока* требуют высокого тока, поэтому между микроконтроллером и двигателем следует подключить мощный драйвер, такой как *H-мост*. Они также требуют *отдельного датчика угла поворота*, если пользователь хочет знать текущее положение двигателя. *Серводвигатели* принимают ШИМ-сигнал, чтобы обозначить своё положение в ограниченном диапазоне углов. С ними очень легко взаимодействовать, но они не такие мощные и не подходят для продолжительного непрерывного вращения. *Шаговые двигатели* принимают последовательность импульсов, каждый из которых вращает двигатель на фиксированный угол, называемый шагом. Они

стоят дороже, и им также нужен Н-мост для управления большим током, но положение подвижной части двигателя можно точно контролировать.

Двигатели могут потреблять значительный ток, что может привести к сбоям на источнике питания, которые нарушают цифровую логику. Один из способов сглаживания этой проблемы заключается в использовании одного источника питания или батареи для питания двигателя и другого – для цифровой логики.

Электродвигатели постоянного тока

На **Рис. 8.58** показана работа щёточного двигателя постоянного тока. Этот двигатель – двухтерминальное устройство. Он содержит постоянные неподвижные магниты, называемые статорами, и вращающийся электромагнит, называемый ротором или якорем, подключенный к валу. Передний конец ротора соединяется с металлическим кольцом, называемым коллектором (commutator). Металлические щетки, присоединённые к входам питания (входные терминалы), трутся о коллектор, обеспечивая поступление тока к электромагниту ротора. Это индуцирует магнитное поле в роторе, которое заставляет ротор вращаться, чтобы выровняться с полем статора. После того, как ротор проходит часть пути при вращении и подходит к выравниванию со статором, щетки касаются противоположных сторон коллектора, изменив направление тока и

магнитного поля на противоположное и заставляя его продолжать вращаться неограниченно долго.

Двигатели постоянного тока, как правило, вращаются, совершая тысячи оборотов в минуту (RPM) при очень низком крутящем моменте. В большинство систем добавляют зубчатую передачу, чтобы уменьшить скорость до более приемлемого уровня и для увеличения крутящего момента.

Поищите зубчатую передачу, предназначенную для сопряжения с вашим двигателем. Pittman производит широкий спектр высококачественных двигателей постоянного тока и аксессуаров к ним, в то время как недорогие игрушечные двигатели популярны среди любителей.

Двигатель постоянного тока требует значительного тока и напряжения, чтобы доставить значительную мощность на нагрузку. Ток должен быть обратимым, если двигатель может вращаться в обоих направлениях.

Большинство микроконтроллеров не могут производить достаточный ток для управления двигателем постоянного тока непосредственно.

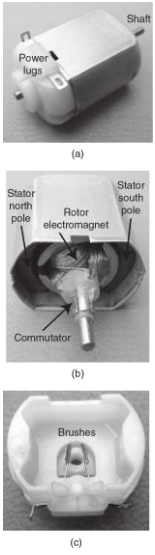


Рис. 8.58 Двигатель постоянного тока

Вместо этого они используют H-мост, который концептуально содержит четыре электрически управляемых переключателя, как показано на **Рис. 8.59 (а)**. Если переключатели A и D замкнуты, ток течет слева направо, через двигатель, и он вращается в одном направлении. Если B и C замкнуты, ток течет справа налево через двигатель, и он вращается в другом направлении. Если A и C или B и D замкнуты, напряжение на двигателе устанавливается в 0, в результате чего двигатель активно тормозится. Если ни один из переключателей не будет замкнут, двигатель будет работать по инерции до полной остановки.

Переключатели в H-мосте являются силовыми транзисторами. H-мост также содержит некоторую цифровую логику для того, чтобы удобно контролировать переключатели. Когда ток двигателя резко меняется, индуктивность электромагнитного двигателя будет вызывать большое напряжение, которое может превышать напряжение питания и повредит силовые транзисторы.

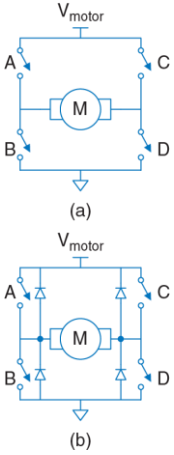


Рис. 8.59 Н-мост

Таким образом, многие Н-мосты также имеют защитные диоды, включенные параллельно с переключателями, как показано на **Рис. 8.59 (b)**. Если индуктивный удар приведёт терминал двигателя к значению напряжения выше Vmotor или ниже уровня GND, диоды включатся и удержат напряжение на безопасном уровне. Н-мосты могут рассеивать большое количество энергии, и может потребоваться теплоотвод, чтобы держать их рабочем состоянии.

Пример 8.28 АВТОНОМНАЯ МАШИНА

Разработайте систему, где PIC32 управлял бы двумя электромоторами машины-робота. Напишите библиотеку, состоящую из функций для инициализации драйвера для двигателя и управления машиной, чтобы она двигалась вперед и назад, поворачивала вправо и влево, а также останавливалась. Используйте ШИМ для управления скоростью двигателей.

Решение: На **Рис. 8.60** приводится схема, где два электромотора постоянного тока управляются микроконтроллером PIC32 с использованием микросхемы Texas Instruments SN754410 (двойной Н-мост). Этой микросхеме требуется питание напряжением 5В, поступающее на V_{CC1}, а также питание с напряжением 4.5–36В для двигателей, поступающее на V_{CC2}. Напряжение на V_{IH} составляет 2 В,

то есть этот порт совместим с портами ввода-вывода PIC32, которые работают с логическими уровнями 3.3 В. Микросхема SN754410 обеспечивает постоянный ток силой до 1 А для каждого из двигателей. **Табл. 8.11** показывает, как каждый из двух H-мостов управляет своим двигателем. Микроконтроллер управляет скоростью двигателей, используя ШИМ. Для управления направлением вращения роторов двигателей используются четыре других контакта.

ШИМ настроен на работу на частоте примерно 781 Гц. Коэффициент заполнения ШИМ может изменяться в диапазоне от 0 до 100%.

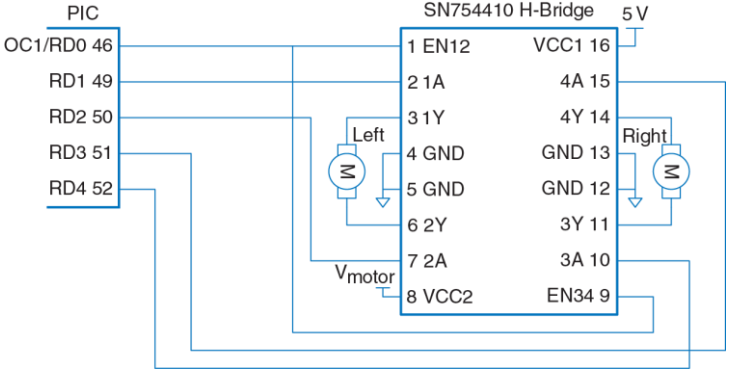


Рис. 8.60 **Управление двигателем при помощи двойного H-моста**

```
#include <P32xxxx.h>

void setspeed(int dutycycle) {
  OC1RS = dutycycle;                  // set duty cycle between 0 and 100
}

void setmotorleft(int dir) {          // dir of 1 = forward, 0 = backward
  PORTDbits.RD1 = dir; PORTDbits.RD2 = !dir;
}

void setmotorright(int dir) {         // dir of 1 = forward, 0 = backward
  PORTDbits.RD3 = dir; PORTDbits.RD4 = !dir;
}

void forward(void) {
  setmotorleft(1); setmotorright(1); // both motors drive forward
}
void backward(void) {
  setmotorleft(0); setmotorright(0); // both motors drive backward
}

void left(void) {
  setmotorleft(0); setmotorright(1);// left back, right forward
}

void right(void) {
  setmotorleft(1); setmotorright(0);// right back, left forward
}
```

```
void halt(void) {
  PORTDCLR = 0x001E;                // turn both motors off by
                                    // clearing RD[4:1] to 0
}

void initmotors(void) {
  TRISD = 0xFFE0;                   // RD[4:0] are outputs
  halt();                           // ensure motors aren' t spinning
                                    // configure PWM on OC1 (RD0)
  T2CONbits.TCKPS = 0b111;          // prescale by 256 to 78.125 KHz
  PR2 = 99;                         // set period to 99+1 ticks = 781.25 Hz
  OC1RS = 0;                        // start with low H-bridge enable signal
  OC1CONbits.OCM = 0b110;           // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;                 // turn on timer 2
  OC1CONbits.ON = 1;                // turn on PWM
}
```

Табл. 8.11 Управление Н-мостом

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

В предыдущем примере нет способа измерить положение каждого двигателя. Два двигателя вряд ли будут в точности повторять состояние друг друга, так что, скорее всего, один будет вращаться немного быстрее другого, в результате чего робот отклонится от курса. Чтобы решить эту проблему, в некоторых системах добавляют датчики угла поворота (ДУП).

На **Рис. 8.61 (a)** показан простой ДУП, состоящий из диска с прорезями, прикрепленного к валу двигателя. На одной стороне размещен Светодиод, а на другой – датчик. ДУП генерирует импульс каждый раз при возникновении прерывания луча света при вращении вала. Микроконтроллер может считать эти импульсы для измерения суммарного угла, на которые вал провернулся. При использовании двух пар светодиод/датчик, расположенных на расстоянии половины ширины внутреннего пространства корпуса мотора друг от друга, улучшенный датчик может подавать на выход квадратуры, показанные на **Рис. 8.61 (b)**, указывающие направление поворота вала, а также угол, на который он провернулся. Иногда к ДУП добавляют еще один зазор, чтобы указать, когда вал находится в заданной позиции.

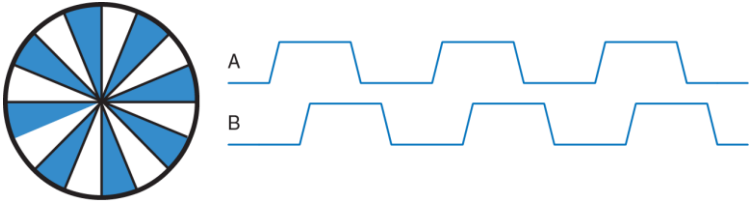


Рис. 8.61 **Датчик угла поворота (ДУП): a) диск, b) квадратурные выходы**

Серводвигатель

Серводвигатель – двигатель постоянного тока, интегрированный с зубчатой передачей, ДУП и некоторой логикой управления, поэтому он проще в использовании. Такие двигатели имеют ограниченный угол поворота: обычно 180°.

На **Рис. 8.62** изображен серводвигатель со снятой крышкой, чтобы показать внутренний механизм. Серводвигатель имеет 3-контактный интерфейс с пинами питания (как правило, 5 В), земли, и управляющим входом. Управляющий вход, как правило, является ШИМ-сигналом с рабочей частотой 50 Гц. Логика управления серводвигателя приводит вал в положение, определяемое коэффициентом заполнения входного сигнала управления.

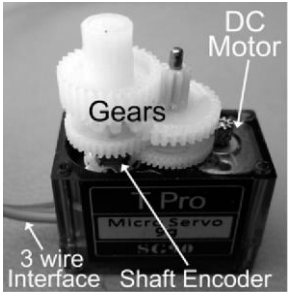


Рис. 8.62 **Серводвигатель SG90**

В качестве ДУП серводвигателя, как правило, выступает поворотный потенциометр, который создает напряжение, величина которого зависит от положения вала.

В типичном серводвигателе с вращением до 180° импульса длительностью 0,5 мс приводит вал к 0°, 1,5 мс доводит до 90°, и от 2,5 мс – до 180°. Например, на **Рис. 8.63** показан управляющий сигнал с длительностью импульса 1,5 мс.

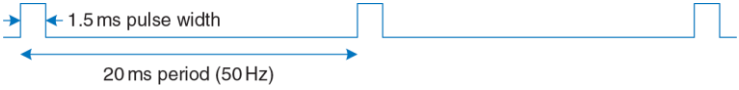


Рис. 8.63 **Временная диаграмма управляющего сигнала серводвигателя**

Выведение серводвигателя за его диапазон может привести к его упору в механические ограничители и его повреждению. Питание серводвигателя приходит от вывода питания, а не вывода управления, поэтому управление можно подключить непосредственно к микроконтроллеру без Н-моста. Серводвигатели обычно используются

в игрушечных самолетах с дистанционным управлением и маленьких роботах из-за их размеров, лёгкости и удобства. Поиск двигателя с адекватной технической документацией может оказаться трудным. Центральный контакт с красным проводом, как правило, является выводом питания, а черный или коричневый провод – это обычно земля.

Пример 8.29 СЕРВОДВИГАТЕЛЬ

Разработайте систему, в которой микроконтроллер PIC32 управляет серводвигателем и поворачивает вал двигателя на заданный угол.

Решение: На **Рис. 8.64** показана схема подключения серводвигателя SG90 к PIC32. Двигатель запитывается постоянным током с напряжением в диапазоне 4.0В – 7.2В. Для передачи сигнала от ШИМ достаточно одного провода, по которому передается сигнал с напряжением 5В или 3.3В.

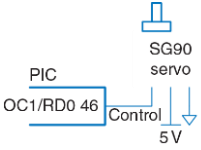


Рис. 8.64 **Блок-диаграмма управление серводвигателем**

Программа реализует ШИМ-генератор с использованием модуля Output Compare 1 module. В программе устанавливается коэффициент заполнения ШИМ, который необходим для поворота вала двигателя на заданный угол.

```
#include <P32xxxx.h>
void initservo(void) {       // configure PWM on OC1 (RD0)
  T2CONbits.TCKPS = 0b111;  // prescale by 256 to 78.125 KHz
  PR2 = 1561;              // set period to 1562 ticks = 50.016 Hz (20 ms)
  OC1RS = 117;              // set pulse width to 1.5 ms to center servo
  OC1CONbits.OCM = 0b110;   // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;         // turn on timer 2
  OC1CONbits.ON = 1;        // turn on PWM
}
void setservo(int angle) {
  if (angle <0)    angle = 0;          // angle must be in the range of
                                       // 0-180 degrees
  else if (angle >180) angle = 180;
  OC1RS = 39+angle*156.1/180;          // set pulsewidth of 39-195 ticks
                                       // (0.5-2.5 ms) based on angle
}
```

Кроме того, можно преобразовать обычный серводвигатель в непрерывно вращающийся сервопривод, разобрав его и удалив механический упор, а вместо потенциометра установить фиксированный делитель напряжения. Многие вебсайты дают подробные указания для конкретных серводвигателей. В таких случаях ШИМ будет контролировать скорость, а не положение, с импульсом указания остановки в 1,5 мс, импульсом 2,5 мс, указывающим на полную скорость и импульсом 0,5 мс, указывающим включение обратного хода привода на полной скорости. Непрерывно вращающийся серводвигатель может быть более удобным и менее

дорогим, чем простой двигатель постоянного тока в сочетании с Н-мостом и зубчатой передачей.

Шаговый двигатель

Шаговый двигатель продвигается на дискретные шаги по мере того, как к его входам поочередно прикладываются импульсы. Шаг обычно составляет несколько градусов, что позволяет выполнить точное позиционирование и продолжительное вращение. Малые шаговые двигатели, как правило, имеют два набора катушек, называемых *фазами*, исполненных в биполярном или однополярном виде. Биполярные двигатели мощнее и дешевле при заданном размере, но требуют наличия Н-моста в качестве драйвера, в то время как униполярные двигатели могут управляться транзисторами, работающими как переключатели. В этом разделе рассматривается более эффективный биполярный шаговый двигатель.

На **Рис. 8.65 (а)** показан упрощенный двухфазный биполярный двигатель с шагом 90°. Ротор является постоянным магнитом с одним северным и одним южным полюсом. Статор – электромагнит с двумя парами катушек, содержащий две фазы. Двухфазные биполярные двигатели, таким образом, имеют четыре терминала. На **Рис. 8.65 (б)** показан символ для шагового двигателя, моделирующий две намотки как катушки индуктивности.

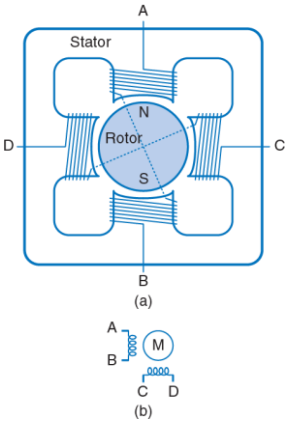


Рис. 8.65 **a) Упрощенный биполярный шаговый двигатель, b) символ шагового двигателя**

На **Рис. 8.66** показаны три общие управляющие последовательности для биполярного двухфазного двигателя. На **Рис. 8.66 (a)** проиллюстрирован *привод вала*, в котором на катушки подается напряжение в последовательности AB – CD – BA – DC. Следует отметить, что BA означает, что обмотка AB находится под напряжением

с током, бегущим в обратную сторону; отсюда и возникло название *«биполярный»*. Ротор поворачивается на 90 градусов на каждом шаге.

Рис. 8.66 (b) иллюстрирует работу *привода с двумя одновременно включаемыми фазами*, работающего по последовательности (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) указывает на то, что обе катушки AB и CD находятся под напряжением одновременно. В этом случае ротор тоже поворачивается на 90 градусов на каждом шаге, но производит автоматическое выравнивание на полпути между двумя полюсами. Это дает самое быстрое вращение, так как обе катушки подают мощность одновременно.

Рис. 8.66 (c) иллюстрирует привод *с полушагом*, работающий по схеме (AB, CD) – CD (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. Ротор вращается на 45 градусов в каждом полушаге. Скорость выполнения этой последовательности определяет скорость вращения двигателя. Чтобы изменить направление вращения двигателя, те же управляющие последовательности подаются в обратном порядке.

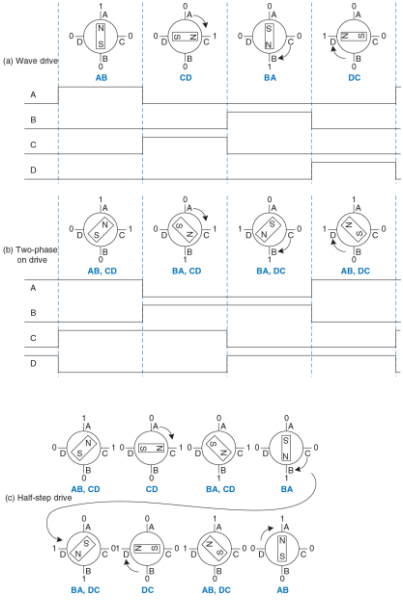


Рис. 8.66 Управление двигателем

В реальном двигателе, ротор имеет множество полюсов, чтобы угол между шагами был намного меньше. Например, на **Рис. 8.67**показан биполярный шаговый двигатель AIRPAXLB82773-M1 с размером шага 7,5 градуса. Двигатель работает от 5 В, и пропускает 0,8 А через каждую катушку.

Рис. 8.67 Биполярный шаговый двигатель AIRPAX LB82773-M1

Крутящий момент в моторе пропорционален току катушки. Этот ток определяется напряжением, приложенным и к индуктивности *L*, и к сопротивлению *R* катушки. Самый простой режим работы называют приводом прямого напряжения или *L/R*-приводом, в котором напряжение *V* прикладывается непосредственно к катушке. Ток достигает значения *I* = *V*/*R* с постоянной времени, определяемой соотношением *L*/*R*, как показано на **Рис. 8.68 (а)**. Это хорошо работает при малых скоростях. Однако, при более высокой скорости ток не

успевает достичь максимального уровня, как показано на **Рис. 8.68 (b)**, и вращающий момент падает.

Более эффективный способ управления шаговым двигателем – метод широтно-импульсной модуляции более высокого напряжения. Высокое напряжение заставляет ток нарастать до максимального значения быстрее, после чего он выключается ШИМ, чтобы избежать перегрузки двигателя. Затем напряжение модулируется или ограничивается, чтобы поддерживать ток около нужного уровня. Это называется *приводом с ограничителем по постоянному току,* что показано на **Рис. 8.68 (c)**. Контроллер использует небольшой резистор, последовательно подключенный к двигателю, чтобы определить протекающий ток путем измерения падения напряжения, и посылает разрешающий сигнал к H-мосту, чтобы выключить привод, когда ток достигает желаемого уровня. В принципе, микроконтроллер может генерировать сигналы правильной формы, но это проще сделать с помощью контроллера шагового двигателя. Контроллер L297 от ST Microelectronics – подходящий выбор, особенно в сочетании с двойным H-мостом L298 с пинами измерения тока и максимально допустимым пиком тока в 2 А. К сожалению, L298 не доступен в DIP-корпусе, поэтому его сложнее монтировать на печатную плату. Документация компании ST на AN460 и AN470 для разработчиков шаговых двигателей является очень ценной.

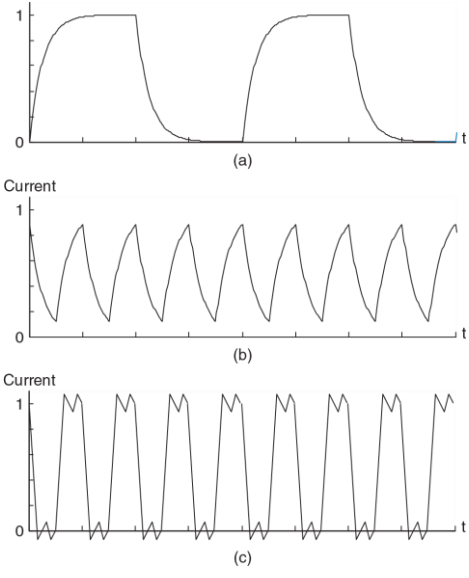


Рис. 8.68 Прямой ток в биполярном шаговом двигателе при a) медленном вращении, b) быстром вращении, c) быстром вращении при ограничении постоянного тока

Пример 8.30 ДВУХПОЛЮСНЫЙ ШАГОВЫЙ ДВИГАТЕЛЬ С ПРИВОДОМ ПРЯМОГО НАПРЯЖЕНИЯ

Разработайте систему, в которой PIC32 микроконтроллер управляет двухполюсным шаговым двигателем с приводом прямого напряжения фирмы AIRPAX с заданными скоростью и направлением вращения.

Решение: На **Рис. 8.69** показано подключение двухполюсного шагового двигателя, используя микросхему SN754410 (H-мост), управляемую микроконтроллером PIC32.

Функция spinstepper инициализирует массив последовательностью значений, которые будут поступать на контакты RD[4:0] для управления приводом прямого напряжения. После вывода очередного значения, программа делает паузу на некоторое время, чтобы ротор делал заданное число оборотов в минуту. Используя тактовую частоту 20 МГц, шаг угла поворота 7.5 градусов, 16-битный таймер и делитель частоты 256:1, диапазон возможных скоростей вращения будет 2–230 об/мин, где нижняя граница обусловлена разрешающей способностью таймера, а верхняя граница ограничена конструкцией двигателя LB82773-M1.

```
#include <P32xxxx.h>
#define STEPSIZE 7.5 // size of step, in degrees

int curstepstate;
// keep track of current state of stepper motor in sequence
void initstepper(void) {
   TRISD = 0xFFE0;        // RD[4:0] are outputs
   curstepstate = 0;
```

```
   T1CONbits.ON = 0;      // turn Timer1 off
   T1CONbits.TCKPS = 3;   // prescale by 256 to run slower
}
// dir = 0 for forward, 1 = reverse
void spinstepper(int dir, int steps, float rpm) {
{
  // wave drive sequence
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001};
  int step;
  // time/step w/ 20 MHz peripheral clock
  PR1 = (int)(20.0e6/(256*(360.0/STEPSIZE)*(rpm/60.0)));
  TMR1 = 0;
  T1CONbits.ON = 1;       // turn Timer1 on
  // take specified number of steps
  for (step = 0; step <steps; step++) {
    // apply current step control
    PORTD = sequence[curstepstate];
    // determine next state forward
    if (dir == 0) curstepstate = (curstepstate + 1) % 4;
    // determine next state backward
    else          curstepstate = (curstepstate + 3) % 4;
    // wait for timer to overflow
    while (!IFS0bits.T1IF);
    IFS0bits.T1IF = 0;          // clear overflow flag
  }
  T1CONbits.ON = 0;             // Timer1 off to save power when done
}
```

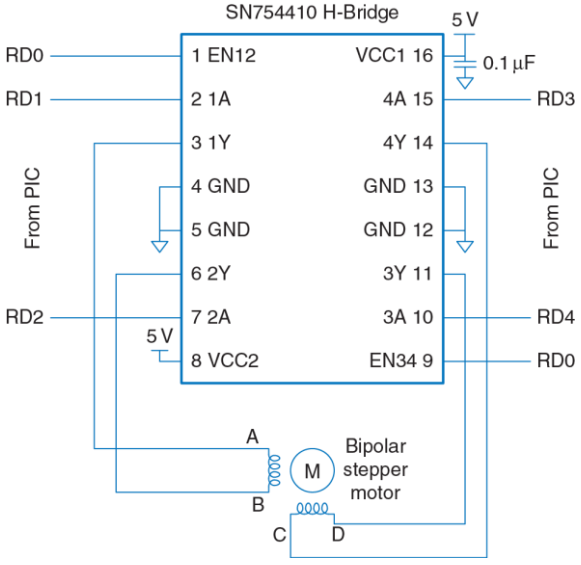


Рис. 8.69 **Биполярный шаговый двигатель управляемый с помощью Н-моста**

8.7 ИНТЕРФЕЙСЫ ВВОДА-ВЫВОДА ПЕРСОНАЛЬНЫХ КОМПЬЮТЕРОВ

Персональные компьютеры (ПК) используют огромное количество протоколов ввода-вывода для различных целей: передачи и получения данных из памяти, работы с дисками, с сетью, с картами расширений и с внешними устройствами. Эти протоколы развивались с целью обеспечения высокой производительности передачи данных и возможности для пользователей без труда подключать внешние устройства. Для реализации этих двух особенностей протоколы ввода-вывода пришлось сильно усложнить. В данном разделе рассматриваются основные стандарты интерфейсов ввода-вывода современных ПК, а также возможности подключения к ПК пользовательской цифровой логики и другого внешнего оборудования.

На **Рис. 8.70** показана материнская плата с разъёмом для процессора Core i5 или i7. Процессор помещен в корпус с 1156 золочеными контактными площадками – LGA-корпус (англ. LGA – land grid array). Эти площадки подводят к процессору питание, землю, подсоединяют к нему память и устройства ввода-вывода. На материнской плате расположено два слота для оперативной памяти (DRAM), различные разъёмы для устройств ввода-вывода и разъем для подключения питания, регуляторы напряжений и конденсаторы. Оперативная память

подключается через интерфейс DDR3. Периферийные устройства, такие как клавиатуры или веб-камеры, подключаются через интерфейс USB. Высокопроизводительные платы расширения, например, видеокарты, подключаются к слоту PCI Express с 16-ю линиями передачи (x16), в то время как менее требовательные по производительности карты расширения могут использовать слот с одной линией передачи (x1) или более старый интерфейс PCI. ПК подключается к сети через Ethernet-кабель. Жесткий диск подключается к разъему SATA. Оставшаяся часть раздела дает общее представление о работе каждого из перечисленных стандартов ввода-вывода.

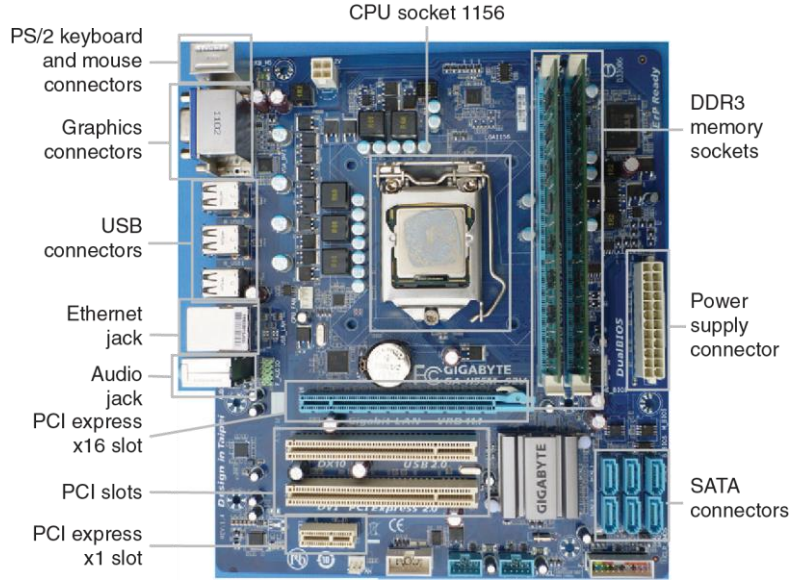


Рис. 8.70 **Материнская плата Gigabyte GA-H55M-S2V**

Одним из главных достижений в стандартах ввода-вывода ПК является разработка высокоскоростных последовательных интерфейсов. До сегодняшнего дня большинство интерфейсов были параллельными. Они включали в себя широкую шину данных и отдельный тактирующий сигнал. Разница задержек сигнала между проводниками в шине накладывает ограничения на максимально возможную скорость передачи данных. Более того, у шин, подключаемых к нескольким устройствам, возникают проблемы, присущие длинным линиям (когда длина проводника сравнима с длиной волны передаваемого сигнала – прим. пер.): отражения и разное время прохождения к различным нагрузкам. Шум также может повредить данные. Последовательные интерфейсы «точка-точка» решают множество таких проблем. Данные обычно передаются по дифференциальной паре. Внешний шум воздействует на оба проводника в паре, и, таким образом, перестает влиять на передаваемый сигнал. Линии передачи проще соединять, так как отражения малы (смотри линии передачи в **разделе A.8**). Тактирующий сигнал не подается в явном виде; вместо этого, приемник восстанавливает его по временам переключения данных (из 0 в 1 и из 1 в 0). Проектирование высокоскоростных последовательных интерфейсов – это отдельный предмет. Хорошие интерфейсы могут работать на скоростях более 10 Гбит/с по медным проводникам, а по оптоволокну – даже быстрее.

8.7.1 USB

До середины 1990-х годов подключение периферийных устройств к ПК требовало некоторую «техническую подкованность». Для добавления плат расширения необходимо было открывать корпус, устанавливать перемычки в правильное положение и вручную устанавливать драйвер устройства. Добавление устройства через интерфейс RS-232 требовало выбора подходящего кабеля и правильной настройки скорости передачи данных, битов данных, битов четности и стоповых битов. Универсальная последовательная шина (USB), разработанная компаниями Intel, IBM, Microsoft и другими, значительно упростила добавление периферийных устройств благодаря стандартизации кабелей и процессу конфигурации программного обеспечения. В настоящее время каждый год продаются миллиарды устройств, подключаемых к компьютеру через интерфейс USB.

Стандарт USB 1.0 был выпущен в 1996 году. Согласно стандарту протокол использует простой кабель с 4-мя проводниками: питание (+5В), земля и дифференциальная пара для передачи данных. Кабель невозможно подключить не той стороной или «вверх ногами». Данные по интерфейсу передаются со скоростью до 12 Мбит/с. Подключаемое устройство может получать до 500мА тока от порта USB, таким образом, клавиатурам, мышам и другим периферийным устройствам нет необходимости использовать батарейки или отдельные кабели питания.

Стандарт USB 2.0, выпущенный в 2000 году, поднял скорость передачи данных до 480 Мбит/с путем увеличения скорости работы дифференциальной пары. После повышения производительности стало возможным подключение веб-камер и внешних жестких дисков. Flash-накопители с интерфейсом USB также заменили гибкие магнитные диски (дискеты) как средство передачи файлов между компьютерами.

Стандарт USB 3.0, выпущенный в 2008 году, принёс ещё большее увеличение скорости передачи данных до 5 Гбит/с. Он использует ту же форму разъема, но кабель имеет больше проводников, сигнал по которым передается на очень высокой скорости. Он лучше подходит для подключения высокопроизводительных жестких дисков. Примерно в то же время стандарт специфицировали для возможности заряда аккумуляторных батарей, тем самым увеличив мощность, подаваемую на порт.

Простота в использовании достигается за счет более сложной аппаратной и программной реализации. Построение USB-интерфейса с нуля является сложным мероприятием. Даже написание простого драйвера устройства – дело нелегкое. PIC32 поставляется со встроенным USB-контроллером. Однако, драйвер фирмы Microchip для подключения мыши к PIC32 – более чем 500 строк кода, и это выходит за рамки этой главы.

8.7.2 PCI и PCI Express

Шина связи периферийных устройств (PCI) является стандартом шин расширения, разработанным компанией Intel и ставшим широко распространенным с 1994 года. Данный интерфейс использовался для добавления плат расширения, таких как дополнительные порты последовательного ввода-вывода или USB-порты, сетевые интерфейсы, звуковые карты, модемы, дисковые контроллеры или видеокарты. Интерфейс представляет собой 32-разрядную параллельную шину, работающую на частоте 33 МГц и обеспечивающую полосу пропускания в 133 Мбайт/с.

Спрос на платы расширения PCI сильно упал. В настоящее время в материнскую плату интегрируются больше стандартных портов, таких как Ethernet и SATA. Многие устройства, которые когда-то можно было подключить только как плату расширения, сейчас могут быть подсоединены через быстрые порты USB 2.0 или USB 3.0. А видеокарты требуют гораздо более широкую полосу пропускания, чем может обеспечить PCI.

На современных материнских платах до сих пор часто можно встретить небольшое количество слотов PCI, но высокопроизводительные устройства, такие как видеокарты, подключаются через PCI Express (PCIe). Слоты PCI Express содержат одну или несколько линий

высокоскоростного последовательного интерфейса. Согласно стандарту PCIe 3.0 каждая линия может обеспечить пропускную способность до 8 Гбит/с. На большинстве материнских плат располагается слот с 16-ю линиями, в сумме обеспечивающих полосу пропускания в 16 Гбайт/с для ресурсоемких устройств, таких как видеокарты.

8.7.3 Память DDR3

На **Рис. 8.71** изображен 4 ГБ DDR3 модуль памяти с двухрядным расположением выводов (DIMM). Этот модуль имеет по 120 контактов с двух сторон, что в сумме дает 240 контактов, включая 64-разрядную шину данных, 16-разрядную адресную шину с временным мультиплексированием, управляющие сигналы и множество контактов земли и питания. В 2012 году модули DIMM обычно содержали от 1 до 16 ГБ оперативной памяти. Емкость памяти увеличивается вдвое примерно каждые 2–3 года.

Рис. 8.71 **Модуль памяти DDR3**

Динамическая оперативная память в настоящее время работает на тактовой частоте 100-266 МГц. DDR3 работает с шиной памяти на тактовой частоте, в 4 раза большей, чем частота тактирующего сигнала. Более того, данные передаются по переднему и заднему фронтам тактовой частоты. Следовательно, передаётся 8 машинных слов данных по каждому тактирующему импульсу. Для 64 бит/слово это соответствует полосе пропускания в 6.4-17 Гбайт/с. Например, DDR3-1600 использует тактовую частоту памяти в 200 МГц, а частоту ввода-вывода – в 800 МГц, и посылает 1600 миллионов слов/сек или 12800 Мбайт/с. Из-за этого модули также называют PC3-12800. К сожалению, задержка оперативной памяти остается высокой с примерным отставанием в 50 нс от запроса на чтение до прибытия первого слова данных.

8.7.4 Сеть

Компьютеры соединяются с интернетом через сетевой интерфейс по протоколу TCP/IP. Физическое соединение может осуществляться посредством Ethernet-кабеля или беспроводной сети Wi-Fi.

Ethernet определяется стандартом IEEE 802.3. Он был разработан в исследовательском центре Xerox PARC в 1974 году. Изначально он работал на скорости 10 Мбит/с, но теперь – обычно на 100 Мбит/с и на 1 Гбит/с. Сигнал передается по кабелям категории 5, содержащим

4 витые пары. 10-Гигабитный Ethernet, работающий на оптоволоконных кабелях, становится все популярнее для серверных и других высокопроизводительных вычислений, Ethernet 100 Гбит только появляется.

Wi-Fi – популярное название беспроводного сетевого стандарта IEEE 802.11. Он работает в диапазонах нелицензируемой беспроводной связи на частотах 2,4 и 5 ГГц. Это означает, что пользователю не нужно получать лицензию радиооператора для передачи в этих диапазонах на низкой мощности. В **Табл. 8.12** представлены возможности трех поколений Wi-Fi.

Табл. 8.12Таблица 8.12 Протоколы 802.11 (Wi-Fi)

Протокол	Год	Частота, ГГц	Скорость передачи данных, Мбит/с	Дальность, м
802.11b	1999	2,4	5,5–11	35
802.11g	2003	2,4	6–54	38
802.11n	2009	2,4 / 5	7,2–150	70

Появление стандарта 802.11ac обещает увеличение скорости беспроводной передачи данных вплоть до 1 Гбит/сек. Увеличение производительности происходит путем улучшения модуляции и обработки сигнала, увеличения числа антенн и использования более широкой полосы пропускания.

8.7.5 SATA

Внутренние жесткие диски требуют быстрого соединения с ПК. В 1986 году компания WD представила интерфейс IDE (Integrated Drive Electronics), который был включен в стандарт соединения ATA. Стандарт использует широкий шлейф с 40 или 80 проводниками с максимальной длиной 18'' (45 см) для передачи данных со скоростями от 16 до 133 Мбайт/с.

Стандарт ATA был вытеснен стандартом Serial ATA, являющимся высокоскоростным последовательным интерфейсом. Интерфейс работает на скоростях 1,5, 3 или 6 Гбит/с, а подключение происходит через более удобный 7-жильный кабель, показанный на **Рис. 8.72**. Самые быстрые твердотельные диски в 2012 году достигли пропускной способности в 500 Мбайт/с, тем самым они полностью раскрыли возможности SATA.

Связанный с ним (с SATA) стандарт – стандарт SAS (Serial Attached SCSI) – результат эволюции параллельного интерфейса SCSI (Small Computer System Interface). SAS предлагает сравнимую с SATA производительность и поддерживает более длинные кабели. Этот интерфейс обычно применяется в серверных компьютерах.

Рис. 8.72 SATA-кабель

8.7.6 Подключения к ПК

Все стандарты ввода-вывода ПК, описанные выше, оптимизированы для высокой производительности и простоты подключения, однако их трудно аппаратно реализовать. Инженерам и ученым часто нужно найти способ подключения к ПК внешних цепей, таких как датчики, приводы, микроконтроллеры или ПЛИС (FPGA). Последовательное соединение, описанное в **разделе 8.6.3**, достаточно для низкоскоростного подключения микроконтроллера по UART (Универсальный асинхронный приёмопередатчик). Этот раздел вводит еще два понятия: система сбора данных и USB-подключение.

Системы сбора данных

Системы сбора данных (ССД) подключают компьютер к реальному миру, используя несколько аналоговых и/или цифровых каналов ввода-вывода. ССД сейчас широко доступны в качестве USB-устройств. Они позволяют легко установить эти устройства. Фирма National Instruments (NI) – лидер по производству ССД. Высокопроизводительные ССД как правило стоят тысячи долларов, в основном потому, что рынок небольшой и имеет ограниченную конкуренцию. К счастью, с 2012 года NI теперь продает студентам удобную систему myDAQ по цене со скидкой в 200$, включая программное обеспечение LabVIEW.

На **Рис. 8.73** изображена myDAQ. У нее есть два аналоговых канала, способных производить до 200 тысяч выборок в секунду с 16-разрядным разрешением в диапазоне ±10В. Эти каналы могут быть сконфигурированы для работы в качестве осциллографа и в качестве генератора сигналов. Она также имеет 8 цифровых входных и выходных линий, совместимых с 3.3 и 5-ти вольтовыми системами. Более того, она оснащена силовыми выходами с напряжениями +5, +15 и -15 вольт, а также включает в себя цифровой мультиметр, способный измерять напряжение, ток и сопротивление. Таким образом, myDAQ может заменить целый стенд измерительного оборудования, в дополнение к этому позволяя производить автоматизированный сбор

данных. Большинство ССД управляются с помощью LabVIEW – графического языка для проектирования измерительных и следящих систем. Некоторыми ССД также можно управлять из программ, написанных на языке Си с использованием среды LabWindows, а также из Microsoft.NET-приложений с использованием среды Measurement Studio, или с помощью Matlab, используя Data Acquisition Toolbox.

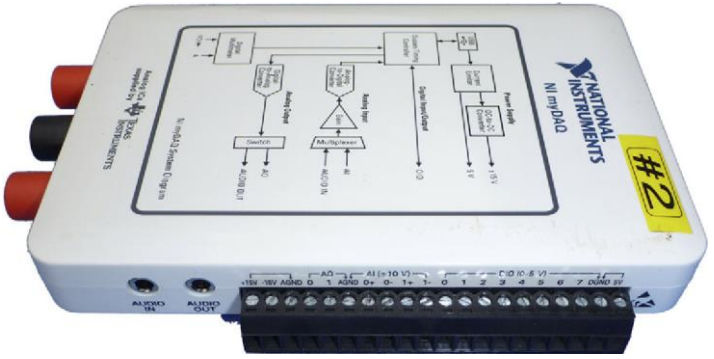


Рис. 8.73 **Система сбора данных myDAQ фирмы National Instruments**

USB-подключение

Всё больше продуктов оснащаются простым и дешевым цифровым USB-интерфейсом. Вместе с этими продуктами поставляются драйверы и библиотеки, позволяющие пользователю написать простую программу на своем ПК, которая обменивается данными с ПЛИС или микроконтроллером.

FTDI – лидирующая фирма по производству таких систем. Например, у кабеля FTDI C232HM-DDHSL USB to MPSSE (Multi-Protocol Synchronous Serial Engine), изображенного на **Рис. 8.74**, на одном конце имеется USB-разъем, а на другом – интерфейс SPI, работающий на скоростях до 30 Мбит/с, линия питания 3.3В и 4 линии ввода-вывода общего назначения (GPIO). На **Рис. 8.75** изображен пример соединения ПК с ПЛИС с помощью этого кабеля. Кабель может опционально питать ПЛИС от линии 3.3В. Три линии SPI подключены к ведомой (slave) ПЛИС таким же образом, как в **примере 8.19**. На **Рис. 8.75** также показано, что одна из линий GPIO используется для управления светодиодом.

Для работы с модулем D2XX необходимо установить динамически подключаемую библиотеку. После этого вы можете приступить к написанию программы на языке Си, используя установленную библиотеку для передачи данных по кабелю.

Рис. 8.74 Кабель USB – MPSSE (напечатано с разрешения FTDI)

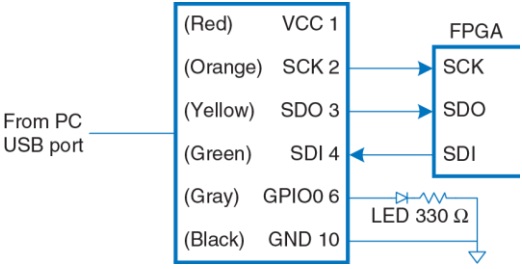


Рис. 8.75 Подключение ПК к FPGA USB-кабелем C232HM-DDHSL

Для случаев, когда требуется более высокоскоростное соединение, FTDI предлагает модуль UM232H, изображенный на **Рис. 8.76**. Модуль является мостом из USB в 8-разрядный синхронный параллельный интерфейс, работающий со скоростями до 40 Мбайт/с.

Рис. 8.76 **Модуль FTDI UM232H**

8.8 ЖИВОЙ ПРИМЕР: СИСТЕМЫ ПАМЯТИ И ВВОДА-ВЫВОДА СЕМЕЙСТВА X86

Поскольку скорость работы процессоров постоянно увеличивается, им необходимы все более совершенные иерархии памяти для сохранения стабильного поступления данных и команд. Этот раздел описывает системы памяти процессоров семейства x86, чтобы показать, как развивались системы памяти. В **разделе 7.9** представлены фотографии процессоров, где отмечены области расположения кэш-памяти на кристаллах. Процессоры x86 также имеют необычную программируемую систему ввода-вывода, которая отличается от большинства систем ввода-вывода, отображаемых в пространство памяти.

8.8.1 Системы кэш-памяти процессоров семейства x86

Процессоры 80386, работавшие на частоте 16 МГц, начали производить в 1985 году. В них не было кэш-памяти, поэтому для доступа ко всем командам и данным процессор обращался непосредственно к оперативной памяти. В зависимости от скорости памяти процессор мог получить либо немедленный ответ, либо ему приходилось ждать ответа один или несколько тактов. Эти такты называются *тактами ожидания (wait states)*, и они увеличивают количество тактов на команду (clocks

per cycle, CPI). С тех пор тактовые частоты микропроцессоров увеличивались минимум на 25% в год, в то время как латентность памяти (memory latency) практически не уменьшалась. На сегодняшний день задержка от момента посылки процессором адреса в оперативную память до получения данных может превышать 100 тактов процессора. Поэтому кэш-память с небольшим количеством промахов просто необходима для хорошей производительности. В **Табл. 8.13** собраны сведения о развитии систем кэш-памяти процессоров Intel x86.

В процессорах 80486 появился общий кэш со сквозной записью для хранения как команд, так и данных. Большинство высокопроизводительных компьютерных систем также размещали на материнской плате кэш второго уровня большего объёма с использованием доступных на рынке микросхем SRAM, которые были значительно быстрее, чем оперативная память.

В процессоре Pentium были применены раздельные кэши для команд и данных, чтобы избежать конфликтов при одновременных обращениях к данным и командам. Кэши использовали стратегию отложенной записи, уменьшая количество обращений к оперативной памяти. Опять же, больший объём кэшей второго уровня (обычно 256-512 Кбайт) обычно размещался на материнской плате.

Табл. 8.13 Эволюция систем памяти процессоров Intel x86

Процессор	Год	Частота, МГц	Кэш данных L1	Кэш команд L1	Кэш L2
80386	1985	16–25	нет	нет	нет
80486	1989	25–100	8 Кбайт, общий		нет на кристалле
Pentium	1993	60–300	8 Кбайт	8 Кбайт	нет на кристалле
Pentium Pro	1995	150–200	8 Кбайт	8 Кбайт	256 Кбайт–1 Мбайт на МКМ
Pentium II	1997	233–450	16 Кбайт	16 Кбайт	256–512 Кбайт на картридже
Pentium III	1999	450–1400	16 Кбайт	16 Кбайт	256–512 Кбайт на кристалле
Pentium 4	2001	1400–3730	8–16 Кбайт	12 K микроопераций в кэше трасс	256 Кбайт–2 Мбайт на кристалле
Pentium M	2003	900–2130	32 Кбайт	32 Кбайт	1–2 Мбайт на кристалле
Core Duo	2005	1500–2160	32 Кбайт на ядро	32 Кбайт на ядро	2 Мбайт на кристалле, общий для всех ядер
Core i7	2009	1600–3600	32 Кбайт на ядро	32 Кбайт на ядро	256 Кбайт на ядро + 4–15 Мбайт L3

Процессоры серии P6 (Pentium Pro, Pentium II, и Pentium III) были разработаны для достижения значительно более высоких тактовых частот. Кэши второго уровня на материнской плате уже не справлялись, поэтому их передвинули ближе к процессору для улучшения латентности и пропускной способности. Pentium Pro был помещен в многокристальный модуль (MKM), содержащий как микросхему процессора, так и кристалл кэш-памяти второго уровня, как показано на **Рис. 8.77**. Как и Pentium, этот процессор имел отдельные восьмикилобайтные кэши первого уровня для команд и данных. Однако эти кэши были неблокируемыми, так что Pentium Pro с внеочередным выполнением команд мог продолжать обращаться к кэшу в то время, пока из оперативной памяти загружались данные для вызвавшего промах предыдущего обращения. Кэш второго уровня имел размер 256 Кбайт, 512 Кбайт или 1 Мбайт и мог работать на той же скорости, что и процессор. К сожалению, многокристальные модули оказались слишком дорогими для крупносерийного производства. Поэтому Pentium II продавался в недорогих картриджах, содержащих процессор и кэш второго уровня. Размер кэшей первого уровня был увеличен вдвое, чтобы компенсировать тот факт, что кэши второго уровня работали в два раза медленнее, чем процессор. В Pentium III кэш второго уровня был размещен на кристалле процессора и работал с той же скоростью, что и процессор. Кэш, расположенный на том же кристалле, что и процессор, имеет меньшую латентность и лучшую

пропускную способность, то есть значительно эффективнее, чем внешний кэш того же размера.

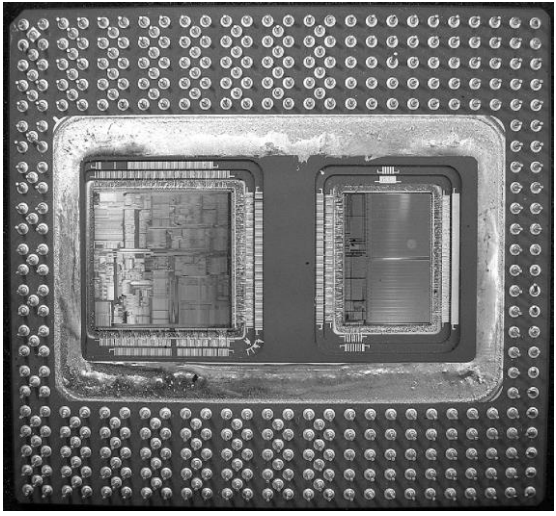


Рис. 8.77 Многокристальный модуль в корпусе PGA с процессором Pentium Pro (слева) и кэшем L2 (справа). Приведено с разрешения Intel

В Pentium 4 был неблокирующий кэш данных первого уровня, а вместо обычного кэша команд был использован *кэш трасс микроопераций* (trace cache) для хранения дешифрованных команд, то есть микроопераций, что позволило избежать повторной дешифрации команд при выборке команд из кэша.

Pentium M был разработан на основе Pentium III. Размер кэшей первого уровня был увеличен до 32 Кбайт, а размер кэшей второго уровня был равен 1 или 2 Мбайт. Core Duo состоял из двух усовершенствованных процессоров Pentium M и имел на кристалле общую для обоих ядер двухмегабайтную кэш-память второго уровня. Общая кэш-память используется для связи между процессорами: один из них может писать данные в кэш, а другой читать их оттуда.

В архитектуре Nehalem (Core i3–i7) появился общий для всех ядер на кристалле кэш третьего уровня, что позволило облегчить обмен информацией между ними. У каждого ядра при этом есть свой собственный кэш первого уровня объёмом 64 Кбайт и кэш второго уровня объемом 256 Кбайт. Размер общего кэша третьего уровня равен 4–8 Мбайт и больше.

8.8.2 Виртуальная память x86

Процессоры x86 работают либо в реальном, либо в защищенном режиме. *Реальный режим* обладает обратной совместимостью с изначальными процессорами 8086, при этом используются только 20 бит адреса, что позволяет адресовать память объемом до 1 Мбайт. В этом режиме нельзя использовать виртуальную память.

Защищенный режим впервые появился в процессорах 80286, затем в процессорах 80386 стали применяться 32-битные адреса. В этом режиме поддерживается виртуальная память со страницами объемом 4 Кбайта. Также обеспечивается защита памяти, так что программа не может получить доступ к страницам, принадлежащим другим программам. Таким образом, вредоносные или некорректно работающие программы не могут нарушить работу других программ. Все современные операционные системы используют защищенный режим процессоров x86.

32-битный адрес позволяет использовать до 4 Гбайт памяти. Начиная с Pentium Pro, процессоры Intel x86 поддерживали до 64 Гбайт памяти, используя способ, называемый *расширением физического адреса* (physical address extension). Каждый процесс продолжал использовать 32-битные адреса, но подсистема виртуальной памяти отображала эти адреса на 36-битное виртуальное адресное пространство. Для каждого

процесса использовались отдельные таблицы страниц, так что каждый процесс мог иметь свое собственное адресное пространство объемом до 4 Гбайт.

> Хотя аппаратная поддержка защиты памяти стала доступна еще в начале 1980-х годов, компании Microsoft потребовалось почти 15 лет на то, чтобы использовать преимущества такой защиты в операционной системе Windows и не позволять одной плохо написанной программе крушить весь компьютер. До выпуска Windows 2000 эта операционная система была печально известна своей нестабильностью. Как видите, промежуток времени между появлением аппаратной реализации некоторой функциональности и её программной поддержкой может быть очень большим.

Чтобы более элегантно обойти проблему ограниченного адресного пространства, архитектура x86 была модернизирована и превратилась в x86-64, которая поддерживает 64-битные виртуальные адреса и регистры общего назначения. На сегодняшний день используются только 48 бит виртуального адреса, позволяющие иметь 256 терабайт виртуального адресного пространства. В будущем это ограничение может быть снято, и максимальные 64 бита позволят адресовать 16 эксабайт памяти.

8.8.3 Программируемый ввод-вывод x86

Большинство архитектур используют ввод-вывод, отображаемый в память, который описан в **разделе 8.5**, когда программы получают доступ к устройствам ввода-вывода, считывая и записывая адреса ячеек памяти. Процессоры x86 используют программируемый ввод-вывод, при котором для записи и считывания информации используются специальные инструкции IN и OUT. Процессоры x86 позволяют использовать 2^{16} портов ввода-вывода. Инструкция IN считывает один, два или четыре байта информации из порта, определяемого регистром DX, в AL, AX, или EAX. Инструкция OUT работает аналогично, но записывает информацию в порт.

Подсоединение периферийного устройства к системе с программируемым вводом-выводом подобно подключению к системе, использующей ввод-вывод, отображаемый в память. При необходимости доступа к порту ввода-вывода, процессор отправляет на 16 младших бит шины адреса не адрес ячейки памяти, а номер порта. Устройство считывает или записывает информацию с шины данных. Основное отличие состоит в том, что процессор генерирует еще и сигнал $M/\overline{IO}$. Когда $M/\overline{IO}$ =1, процессор получает доступ к памяти. Когда 0, процессор получает доступ к одному из устройств ввода-вывода. Дешифратор адреса тоже должен

анализировать состояние сигнала $M/\overline{IO}$, чтобы генерировать необходимые разрешающие сигналы для оперативной памяти и устройств ввода-вывода. Устройства ввода-вывода также могут посылать в процессор запросы прерывания, показывающие, что они готовы к обмену данными.

8.9 РЕЗЮМЕ

Организация системы памяти – важный фактор, влияющий на производительность компьютера. Различные технологии производства памяти, такие как SRAM, DRAM и жесткие диски, позволяют найти компромисс между емкостью, скоростью работы и ценой памяти. В этой главе мы рассмотрели организацию иерархии памяти, включающую кэш-память и виртуальную память, которая позволяет разработчикам приблизиться к идеалу – большой, быстрой и дешевой памяти. Оперативная память обычно использует динамическую память (DRAM) и работает существенно медленнее, чем процессор. Кэш, который хранит часто используемые данные в гораздо более быстрой статической памяти SRAM, используется для уменьшения времени доступа к оперативной памяти. Виртуальная память позволяет увеличить доступный объём памяти, используя жесткий диск, на котором располагаются данные, не помещающиеся в оперативную

память. Кэш и виртуальная память требуют дополнительной аппаратуры и усложняют компьютерную систему, но чаще всего их преимущества перевешивают недостатки. Во всех современных персональных компьютерах используются кэш и виртуальная память. Большинство процессоров также используют интерфейс памяти для обращения к устройствам ввода-вывода. Такой подход называется отображаемым в память вводом-выводом (memory-mapped I/O, MMIO). При таком подходе для работы с внешними устройствами программы пользуются командами загрузки и сохранения данных.

ЭПИЛОГ

Эта глава подводит нас к завершению путешествия по миру цифровых систем. Мы надеемся, что в этой книге мы смогли донести не только красоту и увлекательность искусства их проектирования, но и инженерные знания. Вы изучили, как разрабатывать комбинационную и последовательностную логику на уровне схем и языков описания аппаратуры. Вы познакомились с более крупными строительными блоками, такими как мультиплексоры, АЛУ и память.

Компьютеры – одно из наиболее занимательных приложений цифровых систем. Вы изучили, как программировать процессор MIPS на его родном языке ассемблера, и как построить процессор и систему памяти из цифровых строительных блоков.

В процессе чтения вы наблюдали применение принципов абстракции, дисциплины, иерархии, модульности и регулярности. С их помощью мы сложили пазл внутреннего устройства микропроцессора. От мобильных телефонов до цифрового телевидения, от марсоходов до медицинских систем визуализации – наш мир становится всё более и более цифровым.

Представьте, какую цену был бы готов, как Фауст, заплатить Чарльз Бэббидж, чтобы узнать всё это полтора столетия назад. Он мечтал

всего лишь вычислять математические таблицы с механической точностью.

Сегодняшние цифровые системы вчера были фантастикой. Мог бы Дик Трейси (детектив-персонаж комикса 1930-х годов, у которого был телефон в наручных часах – прим. перев.) слушать iTunes на своем телефоне? Запустил бы Жюль Верн в космос навигационные спутники? Мог бы Гиппократ лечить с помощью МРТ? В то же время, оруэлловский кошмар повсеместной государственной слежки становится ближе к реальности день ото дня.

Хакеры и правительства ведут необъявленные кибервойны, атакуя промышленную инфраструктуру и финансовые системы. Страны-изгои разрабатывают ядерное оружие с помощью ноутбуков более мощных, чем суперкомпьютеры, занимавшие целые машинные залы и использовавшиеся при расчете бомб времен холодной войны. Микропроцессорная революция продолжает ускоряться. Темп грядущих изменений превзойдет их темп в прошедшие десятилетия. Теперь у вас есть инструменты для разработки и построения систем, которые сформируют наше будущее. С этими знаниями приходит и большая ответственность. Мы надеемся, что вы используете их не только для развлечения и обогащения, но и на пользу человечеству.

УПРАЖНЕНИЯ

Упражнение 8.1 В пределах одной страницы опишите четыре повседневных занятия, обладающих временной или пространственной локальностью. Приведите два конкретных примера для каждого типа локальности.

Упражнение 8.2 Одним абзацем опишите два коротких компьютерных приложения, обладающих временной или пространственной локальностью. Опишите, как именно это происходит.

Упражнение 8.3 Придумайте последовательность адресов, для которых кэш с прямым отображением ёмкостью 16 слов и длиной страницы, равной четырем словам, будет более производительным, чем полностью ассоциативный кэш с такой же емкостью и длиной строки, использующий стратегию вытеснения редко используемых данных (LRU).

Упражнение 8.4 Повторите **упражнение 8.3** для случая, когда полностью ассоциативный кэш более производителен, чем кэш с прямым отображением.

Упражнение 8.5 Опишите компромиссы при увеличении каждого из следующих параметров кэша при сохранении остальных параметров неизменными:

a) длина строки

b) степень ассоциативности

c) ёмкость кэша

Упражнение 8.6 Процент промахов у двухсекционного ассоциативного кэша всегда меньше, обычно меньше, иногда меньше или никогда не меньше, чем у кэша с прямым отображением с такой же ёмкостью и длиной строки? Поясните ответ.

Упражнение 8.7 Утверждения ниже относятся к проценту промахов кэша. Укажите, истинно ли каждое из утверждений. Кратко поясните ход рассуждений; приведите контрпример, если утверждение ложно.

a) Процент промахов у двухсекционного ассоциативного кэша всегда ниже, чем у кэша прямого отображения с такой же ёмкостью и длиной строки.

b) Процент промахов у 16-килобайтного кэша прямого отображения всегда ниже, чем у 8-килобайтного кэша прямого отображения с той же длиной строки.

c) Процент промахов у кэша команд с 32-байтной строкой обычно ниже, чем у кэша команд с 8-байтной строкой при той же ёмкости и степени ассоциативности.

Упражнение 8.8 Дан кэш со следующими параметрами: *b*, длина строки в словах; *S*, количество наборов; *N*, количество секций; *A*, число битов адреса.

a) Выразите через перечисленные параметры ёмкость кэша *C*.

b) Выразите через перечисленные параметры число бит, необходимое для хранения тегов.

c) Чему равны *S* и *N* для полностью ассоциативного кэша ёмкостью *C* слов, длина строки которого равна *b*?

d) Чему равно S для кэша прямого отображения кэша ёмкостью *C* слов, длина строки которого равна *b*?

Упражнение 8.9 Дан содержащий 16 слов кэш, параметры которого приведены в **упражнении 8.8**. Рассмотрите следующую повторяющуюся последовательность шестнадцатеричных адресов для команд загрузки (`lw`):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Предполагая использование стратегии вытеснения редко используемых данных (LRU) для ассоциативных кэшей, определите процент промахов при выполнении

этой последовательности команд при использовании одного из приведенных ниже кэшей. Неизбежными промахами (compulsory misses) пренебречь.

a) кэш прямого отображения, b = 1 слово

b) полностью ассоциативный кэш, b = 1 слово

c) двухсекционный ассоциативный кэш, b = 1 слово

d) кэш прямого отображения, b = 2 слова

Упражнение 8.10 Повторите **упражнение 8.9** для следующей повторяющейся последовательности шестнадцатеричных адресов для команд загрузки (`lw`) приведенных нижу конфигураций кэша. Ёмкость кэша – 16 слов.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

a) кэш прямого отображения, b = 1 слово

b) полностью ассоциативный кэш, b = 2 слова

c) двухсекционный ассоциативный кэш, b = 2 слова

d) кэш прямого отображения, b = 4 слова

Упражнение 8.11 Предположим, что выполняется программа со следующей последовательностью адресов обращений к памяти, выполняющейся один раз:

0x0 0x8 0x10 0x18 0x20 0x28

a) Если используется кэш прямого отображения ёмкостью 1 Кбайт и длиной строки 8 байт (2 слова), то сколько в кэше наборов?

b) Каков процент промахов кэша прямого отображения для данной последовательности обращений? Емкость и длина строки такие же, как в пункте a).

c) Что сильнее всего сократит процент промахов при данной последовательности обращений к памяти? Ёмкость кэша постоянна. Выберите один из вариантов.

   i) Увеличение степени ассоциативности до двух.

   ii) Увеличение длины строки до 16 байт.

   iii) Любое из i) или ii)

   iv) Ни i), ни ii)

Упражнение 8.12 Вы разрабатываете кэш команд для процессора MIPS. Его ёмкость равна 4C = 2^{c+2} байт. Его степень ассоциативности $N = 2^n$ ($N \geq 8$), длина строки (размер блока) $b = 2^{b'}$ байт ($b \geq 8$). Используя эти сведения, ответьте на следующие вопросы.

a) Какие биты адреса используются для выбора слова в строке?

b) Какие биты адреса используются для выбора набора в кэше?

c) Сколько битов в каждом теге?

d) Сколько битов тега во всем кэше целиком?

Упражнение 8.13 Дан кэш со следующими параметрами: *N* (ассоциативность) = 2, *b* (дина строки) = 2 слова, *W* (размер слова) = 32 бит, *C* (емкость кэша) = 2^{15} слов, *A* (размер адреса) = 32 бита. Нас интересуют только адреса слов.

a) Какие биты адреса занимают тег, индекс (номер набора), смещение в строке и байтовое смещение? Укажите, сколько бит требуется для каждого из этих полей.

b) Каков общий размер *всех* тегов кэш-памяти в битах?

c) Предположим, что каждая строка кэша содержит бит достоверности (*V*) и бит изменения (*D*). Чему равен размер набора кэша, включая данные, тег и служебные биты?

d) Разработайте кэш, используя строительные блоки, показанные на **Рис. 8.78**, и небольшое число двухвходовых логических элементов. Кэш должен включать память тегов, память данных, сравнение адресов, выбор данных, поступающих на выход, а также любые другие части, которые вы сочтете необходимыми. Учтите, что мультиплексоры и компараторы могут быть любой ширины (*n* или *p* битов соответственно), но блоки памяти SRAM должны обязательно быть 16K × 4 бит. Не забудьте приложить блок-схему с пояснениями. Вам достаточно реализовать только операции чтения.

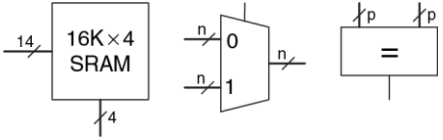


Рис. 8.78 **Строительные блоки**

Упражнение 8.14 Вы устроились в новый крутой интернет-стартап, разрабатывающий наручные часы со встроенным пейджером и веб-браузером. В разработке используется встроенный процессор с многоуровневой схемой кэширования, изображенной на **Рис. 8.79**. Процессор включает небольшой кэш на кристалле и большой внешний кэш второго уровня (да, часы весят полтора килограмма, но зато по интернету просто летают!)

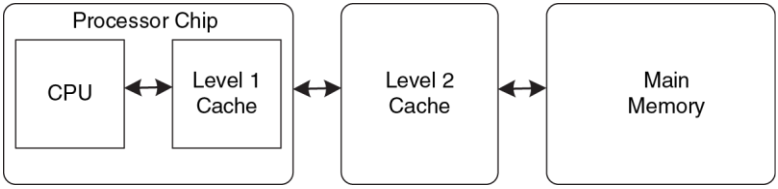


Рис. 8.79 **Компьютерная система**

Предположим, что процессор использует 32-битные физические адреса, но обращается к данным только по адресам, выровненным по границе слова. Характеристики кэшей приведены в **Табл. 8.14**. Время доступа к оперативной памяти (Main Memory) равно t_m, а ее размер – 512 Мбайт.

Табл. 8.14 Характеристики памяти

Характеристика	Кэш на кристалле	Внешний кэш
Способ организации	4-хсекционный наборно-ассоциативный	Прямого отображения
Процент попаданий	*A*	*B*
Время доступа	t_a	t_b
Длина строки	16 байт	16 байт
Число строк	512	256K

a) Во сколько разных мест в кэше на кристалле и в кэше второго уровня может быть загружено слово, находящееся в памяти?

b) Чему равен размер тега в битах для кэша на кристалле и кэша второго уровня?

c) Напишите выражение для вычисления среднего времени доступа к данным при чтении. Обращение к кэшам происходит последовательно (прим. переводчика: то есть сначала к кэшу на кристалле, затем к кэшу второго уровня, затем к оперативной памяти).

d) Измерения показывают, что для некоторой задачи процент попаданий в кэш на кристалле – 85%, а в кэш второго уровня – 90%. Но если кэш на кристалле отключен, то процент попаданий в кэш второго уровня подскакивает до 98,5%. Дайте краткое объяснение такому поведению.

Упражнение 8.15 В этой главе был описан механизм замещения редко используемых строк (LRU) для многосекционных наборно-ассоциативных кэшей. Существуют и другие, хотя и менее распространенные, механизмы замещения, такие как «первым пришел, первым ушел» (first-in-first-out, FIFO) и замещение в случайном порядке. Замещение по принципу FIFO вытесняет ту строку, которая находится в кэше дольше всего, независимо от того, как давно к ней обращались. Случайное замещение выбирает строку для вытеснения случайным образом.

a) Каковы достоинства и недостатки каждого из этих механизмов?

b) Опишите последовательность обращений к памяти, при которой FIFO будет работать лучше, чем LRU.

Упражнение 8.16 Вы строите компьютер с иерархической системой памяти, состоящей из раздельного кэша команд и данных и оперативной памяти, и используете многотактный процессор MIPS, показанный на **Рис. 8.41**, работающий на частоте 1 ГГц.

a) Предположим, что кэш команд работает идеально (т.е. промахов кэша нет), а доля промахов кэша данных – 5%. При промахе кэша процессор приостанавливает выполнение текущей команды на 60 нс, в течение которых происходит доступ к оперативной памяти, после чего возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кэша?

b) Сколько тактов на команду (clocks per instruction, CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?

c) Рассмотрим тестовое приложение из **примера 7.7**, содержащее 25% команд загрузки, 10% команд сохранения, 11% команд условных переходов (ветвлений), 2% команд безусловных переходов и 52% команд типа R.[20] Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?

d) Теперь предположим, что кэш команд тоже неидеален и его промахи происходят в 7% случаев. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи и кэша команд, и кэша данных.

Упражнение 8.17 Повторите **упражнение 8.16** со следующими параметрами.

a) Кэш команд идеален (т.е. промахов кэша нет), а доля промахов кэша данных – 15%. При промахе процессор приостанавливает выполнение текущей команды на 200 нс, в течение которых обращается к оперативной памяти, а затем возобновляет работу. Чему равно среднее время доступа к памяти с учетом промахов кэша?

b) Сколько тактов на команду (clocks per instruction, CPI) в среднем требуется командам чтения и записи слов, учитывая неидеальность системы памяти?

c) Рассмотрим тестовое приложение из **примера 7.7**, содержащее 25% команд загрузки, 10% команд сохранения, 11% команд условных переходов (ветвлений), 2% команд безусловных переходов и 52% команд типа R.

[20] Данные из книги Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011. Использовано с разрешения авторов.

Каково среднее значение CPI для этого теста, учитывая неидеальность системы памяти?

d) Теперь предположим, что кэш команд тоже неидеален и его промахи происходят в 10% случаев. Каково тогда среднее значение CPI для теста из пункта (c)? Учитывайте промахи и кэша команд, и кэша данных.

Упражнение 8.18 Если в компьютере используются 64-битные виртуальные адреса, сколько виртуальной памяти он может адресовать? Помните, что 2^{40} байт = 1 *терабайт*, 2^{50} байт = 1 *петабайт*, а 2^{60} байт = 1 *эксабайт* (прим. переводчика: строго говоря, 1 терабайт равен 10^{12}, а не 2^{40} байт. Корректным названием величины, равной 2^{40} байт, является *тебибайт*. То же самое относится и к остальным величинам, упомянутым в этом упражнении).

Упражнение 8.19 Разработчик суперкомпьютера решил потратить 1 миллион долларов на оперативную память, изготовленную по технологии DRAM, и столько же на жесткие диски для виртуальной памяти. Сколько физической и виртуальной памяти будет у этого компьютера при ценах, приведенных на **Рис. 8.4**? Какого размера должны быть физические и виртуальные адреса для обращения к этой памяти?

Упражнение 8.20 Рассмотрим подсистему виртуальной памяти, способную адресовать в общей сложности 2^{32} байт. У вас есть неограниченное дисковое пространство, но всего 8 Мбайт полупроводниковой (физической) памяти. Предположим, что размер физических и виртуальных страниц равен 4 Кбайт.

a) Чему будет равна длина физического адреса в битах?

b) Каково максимальное число виртуальных страниц в подсистеме?

c) Сколько физических страниц в подсистеме?

d) Каков размер номеров виртуальных и физических страниц в битах?

e) Допустим, вы решили остановиться на схеме прямого отображения виртуальных страниц на физические. При таком отображении для определения номера физической страницы используются младшие биты номера виртуальной страницы. Сколько виртуальных страниц отображается на каждую физическую страницу в этом случае? Почему такое отображение – плохая идея?

f) Очевидно, что необходима более гибкая и динамичная схема трансляции виртуальных адресов в физические, нежели та, что была описана в пункте (e). Предположим, что вы используете таблицу страниц для хранения отображений (то есть информации, позволяющей однозначно соотнести виртуальную страницу с физической). Сколько элементов будет в такой таблице?

g) Предположим, что каждая запись в таблице страниц содержит помимо номера физической страницы еще и некоторую служебную информацию, состоящую из битов достоверности (*V*) и изменения (*D*). Сколько байт понадобится для хранения каждой записи? Округлите результат вверх до ближайшего целого числа байт.

h) Нарисуйте эскиз таблицы страниц. Каков общий размер таблицы в байтах?

Упражнение 8.21 Рассмотрим подсистему виртуальной памяти, способную адресовать 2^{50} байт. У вас есть неограниченное дисковое пространство, но всего 2 Гбайт полупроводниковой (физической) памяти. Предположим, что размер физической и виртуальной страницы равен 4 Кб.

a) Чему будет равна длина физического адреса в битах?

b) Каково максимальное число виртуальных страниц в подсистеме?

c) Сколько физических страниц в подсистеме?

d) Каков размер номеров виртуальных и физических страниц в битах?

e) Сколько записей будет в таблице страниц?

f) Предположим, что каждая запись в таблице страниц содержит, помимо номера физической страницы, еще и некоторую служебную информацию, состоящую из битов достоверности (*V*) и изменения (*D*). Сколько байт понадобится для хранения каждой записи? Округлите результат вверх до ближайшего целого числа байт.

g) Нарисуйте эскиз таблицы страниц. Каков общий размер таблицы в байтах?

Упражнение 8.22 Вы решили ускорить работу подсистемы виртуальной памяти, описанной в **упражнении 8.20**, при помощи буфера ассоциативной трансляции (TLB). Предположим, что система памяти обладает характеристиками, приведенными в **Табл. 8.15**. Процент промахов TLB и кэша показывает, как часто требуемый элемент будет не найден в соответствующей памяти. Процент промахов оперативной памяти показывает, как часто случаются страничные ошибки.

Табл. 8.15 Характеристики памяти

Тип памяти	Время доступа в тактах	Процент промахов
TLB	1	0.05%
Кэш	1	2%
Оперативная память	100	0.0003%
Жесткий диск	1,000,000	0%

a) Каково среднее время доступа в подсистеме виртуальной памяти до и после добавления TLB? Считайте, что таблица страниц всегда расположена в физической памяти и никогда не загружается в кэш данных.

b) Чему равен суммарный размер TLB, состоящего из 64 элементов, в битах? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к своим результатам.

c) Сделайте эскиз TLB. Обозначьте все поля и размеры.

d) SRAM какого размера потребуется для организации TLB, описанного в пункте (c)? Дайте ответ в форме «глубина × ширина».

Упражнение 8.23 Вы решили ускорить систему виртуальной памяти из **упражнения 8.21** с помощью буфера трансляции адресов (TLB) из 128 элементов.

a) Каково суммарное число бит в TLB? Укажите значения для данных (номеров физических страниц), тегов (номеров виртуальных страниц) и битов достоверности для каждого элемента. Покажите, как вы пришли к этим результатам.

b) Нарисуйте эскиз TLB. Обозначьте все поля и их размеры.

c) Блок памяти SRAM какого размера потребуется для организации описанного в пункте (b) TLB? Ответ должен выглядеть как «глубина × ширина».

Упражнение 8.24 Предположим, что многотактный процессор MIPS, описанный в **разделе 7.4**, использует подсистему виртуальной памяти.

a) Покажите расположение TLB в блок-схеме многотактного процессора.

b) Опишите, как добавление TLB повлияет на производительность процессора.

Упражнение 8.25 Подсистема виртуальной памяти, которую вы разрабатываете, использует аппаратно реализованную одноуровневую таблицу страниц, использующую блоки памяти SRAM и сопутствующую логику. Она поддерживает 25-битные виртуальные адреса, 22-битные физические адреса и страницы размером 2^{16} байт (64 Кбайт). В каждой записи таблицы страниц имеется бит достоверности *V* и бит изменения *D*.

a) Чему равен размер всей таблицы в битах?

b) Группа разработчиков ОС предлагает сократить размер страницы с 64 до 16 Кбайт, но разработчики аппаратуры возражают, мотивируя это стоимостью дополнительной аппаратуры. Объясните, почему они против.

c) Таблица страниц будет расположена бок о бок с кэш-памятью на том же кристалле, что и процессор. Эта кэш-память работает только с физическими (а не виртуальными) адресами. Возможно ли при доступе в память одновременно обращаться к нужному набору в кэше и к таблице страниц? Кратко поясните условия, необходимые для одновременного обращения к набору кэша и к записи таблицы страниц.

d) Возможно ли при доступе в память одновременно выполнять сравнение тегов и обращаться к таблице страниц? Кратко поясните.

Упражнение 8.26 Опишите ситуацию, при которой наличие виртуальной памяти могла бы повлиять на разработку приложения. Порассуждайте о том, как размер страницы и физической памяти влияют на производительность приложения.

Упражнение 8.27 Предположим, что у вас есть персональный компьютер (ПК), использующий 32-битные виртуальные адреса.

a) Чему равен максимальный объем виртуальной памяти, доступный каждой программе?

b) Как влияет на производительность размер жесткого диска ПК?

c) Как влияет на производительность размер физической памяти ПК?

Упражнение 8.28 Используйте систему ввода-вывода MIPS, отображаемую в память, для взаимодействия с пользователем. Каждый раз, когда пользователь нажимает на кнопку, высвечивайте произвольный узор на пяти светодиодах (light-emitting diodes, LED). Считайте, что вход с кнопки отображен на адрес 0xFFFFFF10, а LED отображены на адрес 0xFFFFFF14. Когда кнопка нажата, вводится 1, иначе 0.

a) Напишите код для MIPS для реализации этой функциональности.

b) Нарисуйте схему описанной системы ввода-вывода, отображаемого в память.

c) Напишите код на HDL для реализации декодера адреса описанной системы ввода-вывода.

Упражнение 8.29 Конечные автоматы (КА), наподобие построенных в **главе 3**, могут также быть реализованы программно.

a) Реализуйте КА светофора на **Рис. 3.25** на языке ассемблера MIPS. Входы (T_A and T_B) отображены на бит 1 и бит 0, соответственно, слова по адресу 0xFFFFF000. Два 3-битных выхода (L_A and L_B) отображены на биты 0–2 и биты 3–5, соответственно, слова по адресу 0xFFFFF004. Предположите унарное кодирование каждого светового сигнала, L_A and L_B; красный – 100, желтый – 010, зеленый – 001.

b) Нарисуйте схему описанной системы ввода-вывода, отображаемого в память.

c) Напишите код на HDL для реализации декодера адреса описанной системы ввода-вывода.

Упражнение 8.30 Повторите **упражнение 8.29** для КА на **Рис. 3.30 (a)**. Вход A и выход Y отображены на биты 0 и 1, соответственно, слова по адресу 0xFFFFF040.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ

Следующие упражнения задавались в качестве вопросов на собеседованиях при приеме на работу.

Вопрос 8.1 Объясните разницу между кэшем прямого отображения, наборно-ассоциативным и полностью ассоциативным кэшами. Для каждого типа кэша опишите приложение, для которого кэш данного типа будет более эффективен, чем остальные.

Вопрос 8.2 Объясните, как работают подсистемы виртуальной памяти.

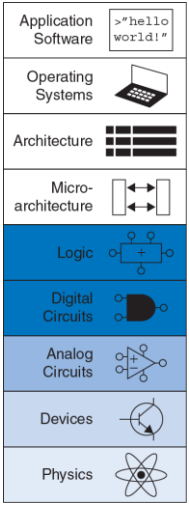
Вопрос 8.3 Объясните достоинства и недостатки использования подсистем виртуальной памяти.

Вопрос 8.4 Объясните, как производительность кэша может зависеть от размера виртуальной страницы в подсистеме виртуальной памяти.

Вопрос 8.5 Могут ли кэшироваться адреса, используемые для отображения устройств ввода-вывода в память? Объясните, почему.

Реализация цифровых систем

A



A.1 ВВЕДЕНИЕ

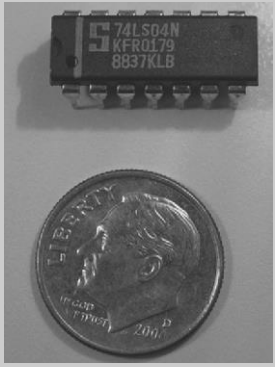
В приложении описываются практические вопросы разработки цифровых систем. Этот материал не является необходимым для понимания содержания остальной части книги, однако он проливает свет на процесс практической разработки цифровых систем. Более того, можно быть уверенным, что наилучший путь к пониманию цифровых систем – это их самостоятельная сборка и отладка в лаборатории.

Обычно цифровые системы состоят из одной или нескольких микросхем. Одна из технологий создания схем – соединение друг с другом микросхем, состоящих из отдельных логических вентилей или более крупных элементов – арифметико-логических устройств (АЛУ) или памяти. Другая технология – использование программируемой логики, которая состоит из набора стандартных элементов и может быть запрограммирована для выполнения требуемых логических операций. Третья технология – разработка непосредственно для заказчика специализированной микросхемы, содержащей определенные логические элементы, необходимые для системы. Эти три технологии предлагают компромиссы по стоимости, быстродействию, энергопотреблению, времени разработки, которые рассмотрены ниже. В приложении также описаны корпуса и монтаж микросхем, линии передачи, соединяющие микросхемы и экономические аспекты разработки и изготовления цифровых систем.

А.2 ЛОГИЧЕСКИЕ МИКРОСХЕМЫ СЕРИИ 74XX

В 1970-80х годах многие цифровые системы строились из простых микросхем, содержащих лишь несколько логических элементов на кристалле . Например, микросхема 7404 содержит шесть элементов НЕ, микросхема 7408 – четыре элемента И, микросхема 7474 – два триггера. Эти микросхемы относятся к серии логических элементов 74xx. Их продавали многие производители, обычно по цене от 10 до 25 центов за микросхему. По большей части эти микросхемы вышли из употребления, однако они до сих пор полезны для простых цифровых систем и лабораторных работ, поскольку они дешевы и просты в использовании. Микросхемы серии 74xx обычно продаются в 14-ти выводных корпусах DIP (корпуса с двухрядным расположением выводов).

А.2.1 Логические элементы



Микросхема инвертора 74LS04 представлена в 14-ти выводном корпусе DIP. Условное обозначение микросхемы находится в первой строке. Буквы LS обозначают семейство логических схем (см. **раздел А.6**). Суффикс N указывает на тип корпус – DIP. S – это товарный знак производителя (Signetics). Комбинаций цифр и букв в двух последних строчках – информация о заводской партии, в которой была изготовлена микросхема.

На **Рис. А.1** показано расположение выводов для нескольких известных микросхем серии 74xx, содержащих основные логические элементы. Их иногда называют малыми интегральными схемами (МИС), поскольку они состоят всего лишь из нескольких транзисторов. 14-ти выводные корпуса обычно имеют вырез сверху или точку в левом верхнем углу для определения ориентации. Выводы нумеруются, начиная с 1 в верхнем левом углу и далее вокруг корпуса против часовой стрелки. Выводы 14 и 7 микросхемы следует подключить к питанию (V_{DD}=5 В) и общему проводу (GND=0 В) соответственно. Количество логических элементов в микросхеме определяется количеством выводов. Обратите внимание, что выводы 3 и 11 микросхемы 7421 ни к чему не подсоединены (NC). Триггер 7474 содержит стандартные выводы *D*, *CLK* и *Q*. Также у него есть инверсный выход $\overline{Q}$. Более того, у этого триггера есть входы асинхронной установки (также называемая предустановкой PRE) и сброса (CLR). Эти выводы активируются низким уровнем, другими словами, триггер устанавливается, когда $\overline{PRE} = 0$, сбрасывается, когда $\overline{CLR} = 0$, и работает в нормальном режиме, когда $\overline{PRE} = \overline{CLR} = 1$

A.2.2 Другие логические функции

Другие микросхемы серии 74xx также могут включать в себя более сложные логические функции, некоторые их них представлены на **Рис. A.2** и **Рис. A.3**. Такие микросхемы называются средними интегральными схемами (СИС). Большинство из них использует более крупные корпуса для размещения большего количества входов и выходов. Выводы питания и земли по-прежнему находятся в правом верхнем и левом нижнем углах каждой микросхемы соответственно. Общее функциональное описание прилагается к каждой микросхеме. Более полная информация о микросхеме приведена в технической документации производителя.

A.3 ПРОГРАММИРУЕМАЯ ЛОГИКА

Программируемая логика состоит из матриц элементов, которые можно сконфигурировать для выполнения определенных логических функций. Мы уже описали три вида программируемой логики: программируемое постоянное запоминающее устройство (PROM), программируемые логические матрицы (PLA) и программируемая пользователем матрица логических элементов (FPGA). Данный раздел описывает реализацию микросхем каждого из этих трех видов.

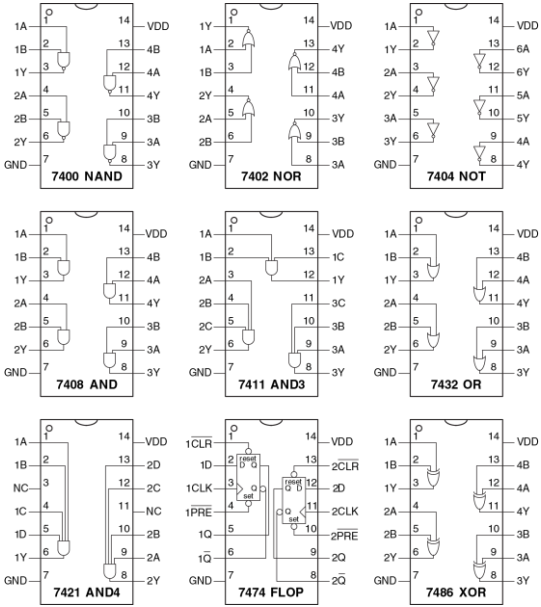
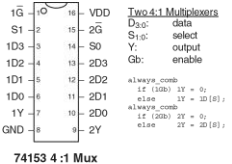


Рис. А.1 Стандартные логические вентили серии 74xx

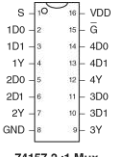


74153 4 :1 Mux

Two 4:1 Multiplexers

$D_{3:0}$:	data
$S_{1:0}$:	select
Y:	output
Gb:	enable

```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[$];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[$];
```

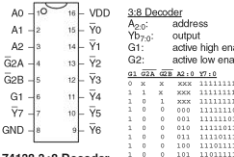


74157 2 :1 Mux

Four 2:1 Multiplexers

$D_{1:0}$:	data
S:	select
Y:	output
Gb:	enable

```
always_comb
  if (Gb) 1Y = 0;
  else    1Y = S ? 1D[1] : 1D[0];
  if (Gb) 2Y = 0;
  else    2Y = S ? 2D[1] : 2D[0];
  if (Gb) 3Y = 0;
  else    3Y = S ? 3D[1] : 3D[0];
  if (Gb) 4Y = 0;
  else    4Y = S ? 4D[1] : 4D[0];
```

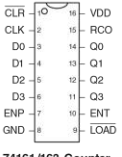


74138 3 :8 Decoder

3:8 Decoder

$A_{2:0}$:	address
$Yb_{7:0}$:	output
G1:	active high enable
G2:	active low enables

```
G1 G2A G2B A2:0 Y7:0
0  X   X   XXX  11111111
1  1   X   XXX  11111111
1  0   1   XXX  11111111
1  0   0   000  11111110
1  0   0   001  11111101
1  0   0   010  11111011
1  0   0   011  11110111
1  0   0   100  11101111
1  0   0   101  11011111
1  0   0   110  10111111
1  0   0   111  01111111
```

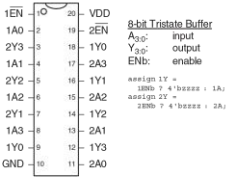


74161/163 Counter

4-bit Counter

CLK:	clock
$Q_{3:0}$:	counter output
$D_{3:0}$:	parallel input
CLRb:	async reset (161) sync reset (163)
LOADb:	load Q from D
ENP, ENT:	enables
RCO:	ripple carry out

```
always @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```

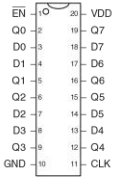


74244 Tristate Buffer

8-bit Tristate Buffer

$A_{3:0}$:	input
$Y_{3:0}$:	output
ENb:	enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```



74377 Register

8-bit Enableable Register

CLK:	clock
$D_{7:0}$:	data
$Q_{7:0}$:	output
ENb:	enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Note: SystemVerilog variable names cannot start with numbers, but the names in the example code in Figure A.2 are chosen to match the manufacturer's data sheet.

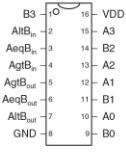
Рис. А.2 Средние интегральные схемы

7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out

RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	1	0	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0



7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always_comb
   if (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin)) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```

74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{n+4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A      :  A              + ~Cbn;
    0001: Y = M ? ~(A | B) :  A         + B  + ~Cbn;
    0010: Y = M ? (~A) & B :  A         + ~B + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111       + ~Cbn;
    0100: Y = M ? ~(A & B) :  A         + (A & ~B) + ~Cbn;
    0101: Y = M ? ~B       :  (A | B)   + (A & ~B) + ~Cbn;
    0110: Y = M ? A ^ B    :  A         - B        -  Cbn;
    0111: Y = M ? A & ~B   :  (A & ~B)              -  Cbn;
    1000: Y = M ? ~A + B   :  A         + (A & B)  + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A         + B        + ~Cbn;
    1010: Y = M ? B        :  (A | ~B)  + (A & B)  + ~Cbn;
    1011: Y = M ? A & B    :  (A & B)               + ~Cbn;
    1100: Y = M ? 1        :  A         + A        + ~Cbn;
    1101: Y = M ? A | ~B   :  (A | B)   + A        + ~Cbn;
    1110: Y = M ? A | B    :  (A | ~B)  + A        + ~Cbn;
    1111: Y = M ? A        :  A                    -  Cbn;
  endcase
```

Рис. А.3 Другие средние интегральные схемы (СИС)

Конфигурирование в таких микросхемах может быть осуществлено пережиганием находящихся в микросхемах перемычек для соединения/разъединения элементов схемы. Это так называемая однократно программируемая логика (OTP), поскольку после того, как перемычка разрушена, её невозможно восстановить. Альтернативным решением является сохранение конфигурации в памяти микросхемы, которую можно перепрограммировать как угодно. Перепрограммируемая логика удобна в лаборатории, поскольку одну и ту же схему можно повторно использовать в процессе разработки.

A.3.1 PROM

Как обсуждалось в **разделе 5.5.7**, блоки PROM могут быть использованы в качестве таблицы преобразования. PROM емкостью 2^N слов×M-бит можно запрограммировать для выполнения любой комбинационной функции с N входами и M выходами. Изменения во время разработки представляют собой простую замену содержимого блока PROM, а не переконфигурирование соединений между микросхемами. Таблицы преобразования полезны для реализации простых функций, однако с ростом количества входов становятся недопустимо дорогими.

Например, классическое стираемое ППЗУ (EPROM) 2764 на 8 кбайт (64 кбит) показано на **Рис. A.4**. EPROM имеет 13 адресных входов для

выбора одного из 8К слов и 8 выводов данных для считывания байта информации из выбранного слова. Выводы «выбор кристалла» и «разрешение выхода» должны активироваться совместно для чтения данных.

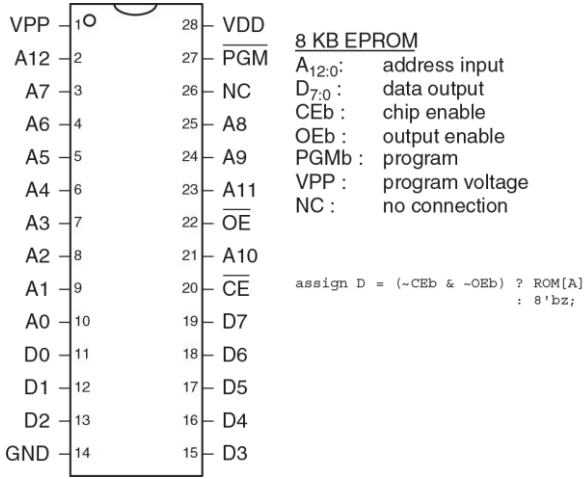


Рис. А.4 ЭСППЗУ 2764 на 8 кбайт

Максимальная задержка распространения сигнала составляет 200 пикосекунд. При нормальном функционировании выводы $\overline{PGM} = 1$ и вывод VPP не используется. EPROM обычно программируется через специальный программатор, который устанавливает $\overline{PGM} = 0$ подает напряжение 13 В на вывод VPP и использует специальную последовательность входных сигналов для конфигурирования содержимого памяти.

Современные PROM основаны на тех же принципах, однако имеют значительно большую емкость и больше выводов. Флэш-память – самый дешевый вариант PROM, продававшийся по цене примерно $1 за гигабайт в 2012 году. Цены традиционно снижаются на 30-40% в год.

A.3.2 Блоки PLA

Как уже обсуждалось в **разделе 5.6.1**, PLA (программируемые логические матрицы) содержат массивы элементов И и ИЛИ для вычисления любых комбинационных функций, написанных в совершенной дизъюнктивной нормальной форме. Массивы элементов И и ИЛИ могут быть запрограммированы по той же технологии, что и PROM. Матрицы PLA содержат два столбца для каждого входа и один столбец для каждого выхода, а также одну строку для каждого минтерма. Такая структура более эффективна, чем PROM для многих

функций, однако матрицы опять-таки становятся слишком большими для сложных функций с большим количеством входов/выходов и минтермов.

Многие производители расширили базовую концепцию PLA до уровня программируемых логических устройств (PLD), которые включают в себя регистры. Микросхема 22V10 – одна из наиболее распространенных PLD. Она содержит 12 выделенных входов и 10 выходов. Выходы могут подключаться напрямую к PLA или к тактируемым регистрам микросхемы. Также выходы можно подключать обратно ко входам PLA. Таким образом, с помощью микросхемы 22V10 можно непосредственно создавать конечный автомат с 12 входами, 10 выходами и 10 битами состояния. Микросхема 22V10 стоит примерно $2 в партиях 100 штук. PLD стали устаревшей технологией из-за быстрого увеличения емкости и снижения цен на FPGA.

A.3.3 FPGA

Как уже говорилось в **разделе 5.6.2**, FPGA состоят из массивов конфигурируемых логических элементов, также называемых конфигурируемыми логическими блоками (КЛБ), соединенных друг с другом программируемыми межсоединениями. Логические элементы состоят из небольших таблиц преобразования и триггеров. FPGA

изящно масштабируются до больших размерностей с тысячами таблиц преобразования. Xilinx и Altera – два ведущих производителя FPGA.

Таблицы преобразования и программируемые межсоединения позволяют создать практически любую логическую функцию. Однако, они на порядок менее эффективны по показателю скорости и себестоимости (на единицу площади микросхемы), чем аппаратно реализованные версии тех же функций. Поэтому FPGA часто включают в себя специальные блоки – память, умножители и даже целые микропроцессоры.

На **Рис. А.5** показан процесс разработки цифровой системы на FPGA. Разработка обычно ведется на языке описания аппаратуры (HDL), хотя некоторые среды разработки FPGA поддерживают графический ввод схем. После разработки проводится моделирование. На входы схемы подаются тестовые воздействия и выходные сигналы сравниваются с ожидаемыми, чтобы *проверить* работу устройства. Обычно требуется некоторая отладка. Далее в процессе логического *синтеза* HDL описание преобразуется в булевы функции. Современные средства синтеза строят принципиальные схемы реализаций этих функций и скрупулезный разработчик анализирует эти схемы, а также все предупреждения, появившиеся в процессе синтеза, чтобы убедиться, что реализована именно желаемая логика.

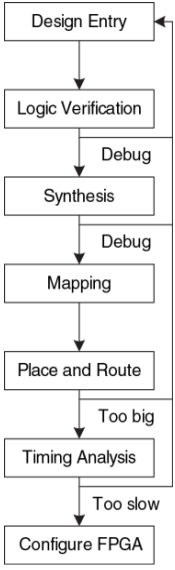


Рис. А.5 Процесс разработки цифровой системы на FPGA

Иногда небрежное описание приводит к генерации схем, больших по размерам, чем требуется, и схем с асинхронной логикой работы. Когда результаты синтеза удовлетворительны, среда разработки *отображает* функции на логические элементы конкретной микросхемы. Инструмент *размещения и трассировки* определяет, к какой таблице преобразования относится каждая функция, и как эти таблицы соединены между собой. Задержка распространения сигнала возрастает с увеличением длины проводника, поэтому наиболее ответственные цепи следует размещать как можно ближе друг к другу. Если проект слишком велик, чтобы уместиться на одной микросхеме FPGA, его придется пересматривать. *Временной анализ* сравнивает ограничения во временной области (например, заданная тактовая частота 100 МГц) с реальными задержками, получаемыми в схеме, и выдает отчет об ошибках. Если схема работает слишком медленно, её, возможно, следует модернизировать или изменить конвейерную структуру. После того, как проект исправлен, генерируется файл,

определяющий содержание каждого логического элемента и конфигурацию всех трассировочных ресурсов или межсоединений в FPGA. Многие FPGA сохраняют *конфигурационную* информацию в статическом ОЗУ, которое должно перезагружаться каждый раз при включении FPGA. FPGA может загружать эту информацию с лабораторного компьютера или считывать ее из энергонезависимой памяти при подаче напряжения питания.

Пример А1. АНАЛИЗ ВРЕМЕННЫХ ДИАГРАММ FPGA

Алиса П. Хэкер использует FPGA для реализации сортировщика драже M&M с датчиком цвета и моторами, чтобы складывать красные конфеты в одну банку, а зеленые – в другую. Ее разработка представляет собой конечный автомат, и она использует FPGA Cyclone IV GX. Согласно технической документации, данная FPGA имеет временные характеристики, представленные в **Табл. А.1**

Алиса хотела бы, чтобы ее конечный автомат работал на скорости 100 МГц. Каково максимальное количество логических элементов в критическом пути? Какова максимальная скорость, на которой сможет работать конечный автомат?

Решение: На частоте 100 МГц время цикла T_c составляет 10 нс. Алиса использует **уравнение (3.13)** для получения минимальной комбинационной задержки распространения t_{pd} за время цикла:

$$t_{pd} \leq 10\text{ns} - (0{,}199\text{ ns} + 0{,}076\text{ ns}) = 9{,}725\text{ ns} \quad \text{(A.1)}$$

С учетом задержек в логических элементах и линиях передач, для автомата Алиса может использовать самое большее 15 последовательно включенных логических элементов (9.725/0.627) для реализации логики переходов.

Самая высокая скорость работы автомата на FPGA Cyclone IV GX достигается при использовании одного логического элемента для перехода к следующему логическому состоянию. Минимальное время цикла

$$T_c \geq 381 \text{ ps} + 19 \text{ ps} + 76 \text{ ps} = 656 \text{ ps} \quad \text{(A.2)}$$

Таким образом, максимальная частота составляет 1,5 ГГц.

Табл. А.1 Временные характеристики Cyclone IV GX

величина	значение (пс)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (на один логический элемент)	381
t_{wire} (между логическими элементами)	246
t_{skew}	0

Фирма Altera сообщает о выпуске Cyclone IV FPGA, содержащей 14 400 логических элементов по цене $25 в 2012 году. В большом количестве FPGA среднего размера обычно стоят несколько долларов.

Самые большие FPGA стоят сотни и даже тысячи долларов. Цена снижаются примерно на 30% в год, и FPGA становятся очень популярными.

А.4 ПРОГРАММИРУЕМАЯ ЛОГИКА

Заказные интегральные схемы (application specific integrated circuit, ASIC) – это микросхемы, разработанные для специальных применений. Графические ускорители, микросхемы сетевых интерфейсов, и микросхемы мобильных телефонов – примеры заказных ИМС. Разработчик таких схем располагает транзисторы таким образом, чтобы сформировать логические элементы и соединить их друг с другом. Поскольку архитектура заказных ИМС аппаратно фиксирована, они обычно в несколько раз быстрее FPGA и занимают на порядок меньше места на кристалле микросхемы (и, соответственно, стоят меньше), чем FPGA с такими же функциями. Однако, стоимость производства *фотошаблонов* для литографии, определяющих, где должны располагаться транзисторы и проводники на микросхеме, составляет сотни тысяч долларов. Технологический процесс на полупроводниковом производстве обычно занимает от 6 до 12 недель и включает в себя производство, корпусирование и испытания ИМС. Если ошибки обнаружены после производства заказной ИМС, разработчик должен ее исправить, изготовить новые фотошаблоны и дождаться выпуска новой партии микросхем. Таким образом, заказные ИМС применимы только для изделий, производимых массово, и функции которых строго определены заранее.

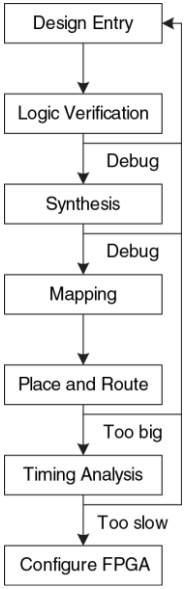


Рис. А.6 Процесс разработки заказной ИМС

На **Рис. А.6** показан процесс разработки заказной ИМС, который схож с процессом разработки FPGA, показанным на **Рис. А.5**. Логическая верификация проекта особенно важна, поскольку исправление ошибок после производства фотошаблонов очень дорогостоящее мероприятие. В результате выполнения синтеза получается *список соединений или нетлист*, в котором перечислены логические элементы и соединения между ними. Логические элементы, указанные в этом списке, размещаются на кристалле, между ними прокладываются соединяющие их проводники. После того, как проект признается удовлетворительным, создаются фотошаблоны, используемые далее для производства ИМС. Одна единственная пылинка может испортить всю схему, поэтому микросхемы тестируются после производства.

Доля рабочих микросхем называется *выходом годных изделий* и обычно составляет от 50 до 90%, в зависимости от размеров микросхем и степени отработанности производственного процесса. Наконец, рабочие кристаллы микросхемы корпусируются, этот вопрос рассматривается в **разделе А.7**.

А.5 ЗАКАЗНЫЕ СПЕЦИАЛИЗИРОВАННЫЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ

Производители микросхем публикуют техническую документацию (ТД), в которой описывается назначение и технические характеристики микросхем. Необходимо уметь читать и понимать техническую документацию (ТД). Одной из основных причин ошибок при разработке цифровых систем является непонимание того, как работает микросхема.

ТД обычно можно найти на сайте производителя. Если вы не можете найти техническую документацию на компонент или у вас нет полной документации из других источников, не используйте этот компонент. Некоторые разделы в технической документации могут быть непонятны. Часто производители издают справочники, включающие в себя ТД сразу на многие однотипные компоненты. В начале справочника приводится дополнительная пояснительная информация. Эту информацию обычно можно найти в Интернете путем тщательных поисков.

В данном разделе рассматривается ТД компании Texas Instruments (TI) на микросхему инвертора 74HC04. Эта ТД довольно проста, однако описывает многие основные элементы. Компания TI производит широкий ассортимент микросхем серии 74xx. В прошлом микросхемы этой серии производились многими компаниями, однако из-за падения продаж рынок консолидируется.

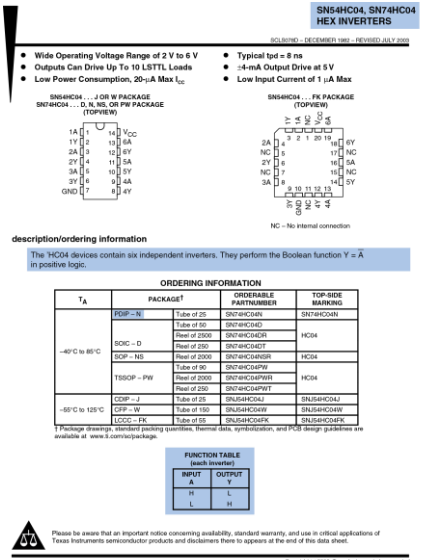
На **Рис. А.7** показана первая страница ТД. Некоторые основные разделы документа выделены синим. Название ТД – SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS означает, что микросхема содержит шесть инверторов. SN обозначает компанию производителя (TI). Обозначения других производителей: MC для Motorola и DM для National Semiconductor. В целом, можно не обращать внимания на эти обозначения, поскольку все производители выпускают совместимые микросхемы серии 74xx. HC обозначает семейство логических схем (высокоскоростные КМОП). Семейство логических схем определяет быстродействие и энергопотребление микросхемы, но не назначение. Например, микросхемы 7404, 74HC04 и 74LS04 состоят из шести инверторов, однако у них разные производительность и стоимость. Остальные семейства логических схем рассмотрены в разделе А.6. Микросхемы серии 74xx работают в коммерческом или промышленном температурных диапазонах (от 0 до 70°C или от -40 до 85°C соответственно), тогда как микросхемы серии 54xx работают в

военном диапазоне (от -55 до 125°C) и стоят дороже. По другим же параметрам эти микросхемы взаимозаменяемы.

Микросхемы 7404 доступны в различных корпусах, при покупке необходимо заказывать ту, которая вам нужна. Тип корпуса определяется суффиксом в обозначении компонента. N обозначает пластмассовый корпус с двухрядным расположением выводов (*PDIP*), который устанавливается на макетную плату или может быть впаян в сквозные отверстия на печатной плате. Другие типы корпусов рассматриваются в **разделе А.7**.

Таблица истинности показывает, что каждый вентиль инвертирует свое входное значение. Если на выводе *A* высокий уровень сигнала, то на выводе *Y* – низкий, и наоборот. В данном случае таблица проста, однако, для сложных микросхем она будет менее тривиальна.

На **Рис. А.8** показана вторая страница ТД. Условное обозначение показывает, что микросхема состоит из инверторов. Раздел *absolute maximum* определяет условия, при нарушении которых микросхема может быть разрушена. В частности, напряжение питания (V_{CC}, в этой книге обозначается также как V_{DD}) не должно превышать 7 В. Постоянный выходной ток не должен превышать 25 мА. *Тепловое сопротивление*, или импеданс, θ_{JA}, используется при расчете повышения температуры из-за рассеяния мощности микросхемой.



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- Wide Operating Voltage Range of 2 V to 6 V
- Outputs Can Drive Up To 10 LSTTL Loads
- Low Power Consumption, 20-μA Max I_{CC}
- Typical tpd = 8 ns
- ±4-mA Output Drive at 5 V
- Low Input Current of 1 μA Max

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOP VIEW)

1A	1	14	V_{CC}
1Y	2	13	6A
2A	3	12	6Y
2Y	4	11	5A
3A	5	10	5Y
3Y	6	9	4A
GND	7	8	4Y

SN54HC04 . . . FK PACKAGE
(TOP VIEW)

2A	4	18	6Y
NC	5	17	NC
2Y	6	16	5A
NC	7	15	NC
3A	8	14	5Y

NC – No internal connection

description/ordering information

The 'HC04 devices contain six independent inverters. They perform the Boolean function Y = $\overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Рис. А.7 Техническая документация на микросхему серии 7404, страница 1

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC} . . . . . −0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1) . . . . . ±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1) . . . . . ±20 mA
Continuous output current, I_O (V_O = 0 to V_{CC}) . . . . . ±25 mA
Continuous current through V_{CC} or GND . . . . . ±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package . . . . . 86°C/W
N package . . . . . 80°C/W
NS package . . . . . 76°C/W
PW package . . . . . 131°C/W
Storage temperature range, T_{stg} . . . . . −65°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

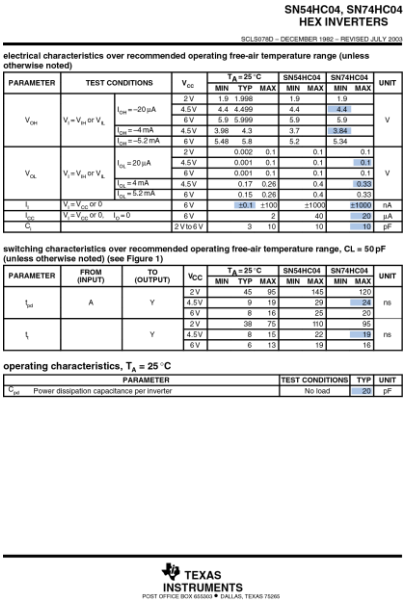
recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		−55		125	−40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Рис. А.8 Техническая документация на микросхему серии 7404, страница 2



SN54HC04, SN74HC04
HEX INVERTERS

SCLS070D – DECEMBER 1982 – REVISED JULY 2003

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	V_I = V_{IH} or V_{IL}	I_{OH} = –20 μA	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		I_{OH} = –4 mA	4.5 V	3.98	4.3		3.7		3.84		
		I_{OH} = –5.2 mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	V_I = V_{IH} or V_{IL}	I_{OL} = 20 μA	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		I_{OL} = 4 mA	4.5 V		0.17	0.26		0.4		0.33	
		I_{OL} = 5.2 mA	6 V		0.15	0.26		0.4		0.33	
I_I	V_I = V_{CC} or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	V_I = V_{CC} or 0,	I_O = 0	6 V			2		40		20	μA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, T_A = 25°C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Рис. А.9 Техническая документация на микросхему серии 7404, страница 3

Если *температура окружающей среды* в вокруг микросхемы T_A и микросхема рассеивает мощность P_{chip}, то температура кристалла микросхемы в месте *соединения* с корпусом равна

$$T_j = T_A + P_{\text{chip}}\,\theta_{JA} \tag{A.3}$$

Например, если микросхема 7404 в пластиковом DIP корпусе работает в нагретом устройстве при температуре 50 °C и потребляет 20 мВт, температура *соединения* кристалла с корпусом поднимется до 50 °C + 0,02 Вт × 80 °C/Вт = 51,6 °C. Внутреннее рассеяние мощности редко имеет значение для микросхем серии 74xx, но оно становится важным для современных микросхем, которые рассеивают мощность в десятки ватт и больше.

Раздел *recommended operating conditions* (рекомендуемые условия работы) определяет, в каких условиях должна работать микросхема. При работе в таких условиях параметры микросхемы должны соответствовать техническим условиям. Эти условия более строгие, чем предельные значения. Например, напряжение питания должно быть между 2 и 6 В. Уровень входного сигнала логической единицы семейства микросхем HC зависит от напряжения V_{DD}. При V_{DD}=5 В для логической 1 следует подавать на вход напряжение 4,5 В, чтобы предусмотреть 10%-ное падение напряжения питания, вызванное шумами в системе.

На **Рис. А.9** показана третья страница ТД. Раздел *electrical characteristics* (электрические характеристики) описывает поведение микросхемы при работе в рекомендуемых режимах работы при постоянном напряжении на входах.

Например, если V_{CC} = 5 В (и может снизится до 4,5 В) и выходной ток I_{OH}/I_{OL} не превышает 20 мкА, то V_{OH} = 4,4 В и V_{OL} = 0,1 В в наихудшем случае. Если выходной ток возрастет, то разница между выходными напряжениями V_{OH} и V_{OL} уменьшается, поскольку транзисторы в микросхеме с трудом могут обеспечивать такой ток. Семейство HC использует КМОП транзисторы, потребляющие малые токи. Ток на каждом входе гарантированно меньше 1000 нА и обычно порядка 0,1 нА при комнатной температуре. *Ток покоя* (I_{DD}), потребляемый, когда микросхема находится в статическом режиме, составляет менее 20 мкА. Каждый вход имеет емкость менее 10 пФ.

Раздел *switching characteristics* (характеристики переключения) определяет, как микросхема ведет себя при рекомендованных режимах работы при изменении состояния входов. *Задержка распространения сигнала*, t_{pd}, измеряется от момента перехода входного сигнала через 0,5 V_{CC} до момента перехода выходного сигнала через 0,5 V_{CC}. Если VCC обычно 5 В и емкость нагрузки микросхемы составляет менее 50 пФ, задержка распространения не будет превышать 24 нс (обычно она даже меньше). Следует помнить, что каждый вход может иметь

емкость 10 пФ, поэтому микросхема не может управлять более чем пятью одинаковыми микросхемами на максимальной скорости. Разумеется, паразитная емкость проводников, соединяющих микросхемы, уменьшает возможную полезную нагрузку. *Время переключения* (время нарастания/спада) измеряется как время между значениями 0,1 V_{CC} и 0,9 V_{CC}.

В **разделе 1.8** говорилось, что микросхемы потребляют *статическую и динамическую* мощность. Статическая мощность у микросхем семейства HC невысока. При температуре 85°C максимальный ток покоя составляет 20 мкА. При напряжении 5 В статическая мощность составляет 0,1 мВт. Динамическая мощность зависит от емкости нагрузки и частоты переключения. Микросхемы серии 7404 имеют внутреннюю емкость (учитываемую при расчете рассеиваемой мощности) 20 пФ на инвертор. Если все шесть инверторов микросхемы серии 7404 переключаются с частотой 10 МГц и управляют внешними нагрузками по 25 пФ, то динамическая мощность согласно **уравнению (1.4)** равна ½(6)(20 пФ + 25 пФ)(5^2)(10 МГц)=33,75 мВт и максимальная полная мощность составляет 33,85 мВт.

А.6 РАБОТА С ДОКУМЕНТАЦИЕЙ

Микросхемы серии 74xx производятся при помощи различных технологий, называемых *семействами логических микросхем*, которые имеют различные быстродействие, мощность и логические уровни. Другие микросхемы обычно совместимы с некоторыми семействами этих микросхем. Оригинальные микросхемы, например, 7404, производились на основе биполярных транзисторов при помощи технологии транзисторно-транзисторной логики (ТТЛ). Современные технологии добавляют одну или несколько букв в маркировке после цифр 74 для отображения семейства, например, 74LS04, 74HC04 или 74AHCT04. В **Табл. А.2** приведены наиболее распространенные семейства микросхем с напряжением питания 5 В.

Достижения в биполярных схемах и развитии технологии привели к созданию семейств на основе транзисторно-транзисторной логики с диодами Шоттки (ТТЛШ) (S) и маломощной транзисторно-транзисторной логики с диодами Шоттки (LS). Оба семейства работают быстрее ТТЛ. ТТЛШ потребляют большую мощность, а маломощный ТТЛШ – меньшую. Усовершенствованные ТТЛШ (AS) и усовершенствованные маломощный ТТЛШ (ALS) семейства микросхем имеют улучшенные характеристики быстродействия и мощности по сравнению с семействами S и LS. Микросхемы семейства быстрой логики (F) работает быстрее и потребляет меньше мощности, чем AS.

Табл. А.2 Характеристики семейств логики с напряжением 5 В

Характеристики	Биполярные/ТТЛ						КМДП		КМДП/ТТЛ совместимые	
	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (нс)	22	9	12	7,5	10	6	21	7,5	30	7,7
V_{IH} (В)	2	2	2	2	2	2	3,15	3,15	2	2
V_{IL} (В)	0,8	0,8	0,8	0,8	0,8	0,8	1,35	1,35	0,8	0,8
V_{OH} (В)	2,4	2,7	2,7	2,5	2,5	2,5	3,84	3,8	3,84	3,8
V_{OL} (В)	0,4	0,5	0,5	0,5	0,5	0,5	0,33	0,44	0,33	0,44
I_{OH} (мА)	0,4	1	0,4	2	0,4	1	4	8	4	8
I_{OL} (мА)	16	20	8	20	8	20	4	8	4	8
I_{IL} (мА)	1,6	2	0,4	0,5	0,1	0,6	0,001	0,001	0,001	0,001
I_{IH} (мА)	0,04	0,05	0,02	0,02	0,02	0,02	0,001	0,001	0,001	0,001
I_{DD} (мА)	33	54	6,6	26	4,2	15	0,02	0,02	0,02	0,02
C_{Pd} (пФ)	Неизвестно						20	12	20	14
Стоимость* ($США)	Устаревшие	0,63	0,25	0,53	0,32	0,22	0,12	0,12	0,12	0,12

Все микросхемы этих семейств обеспечивают больший ток для выходов с низким уровнем сигнала, чем с высоким, и поэтому имеют несимметричные логические уровни. Они согласовываются с

логическими уровнями «ТТЛ»: V_{IH} = 2 В, V_{IL} = 0,8 В, V_{OH} > 2,4 В и V_{OL} < 0,5 В.

После того, как в 1980-х и 1990-х годах были разработаны КМОП микросхемы, они стали широко распространены благодаря малому энергопотреблению и малому входному току. Семейства микросхем на базе высокоскоростной КМОП (HC) и усовершенствованной высокоскоростной КМОП (AHC) технологии практически не потребляют статической мощности. Также они обеспечивают одинаковые токи на выходах с высоким и низким уровнем сигнала. Они согласовываются с логическими уровнями «КМОП»: V_{IH} = 3,15 В, V_{IL} = 1,35 В, V_{OH} > 3,8 В и V_{OL} < 0,44 В. К сожалению, эти логические уровни несовместимы с уровнями ТТЛ схем, поскольку высокий уровень выходного сигнала ТТЛ 2,4 В может быть не распознан как высокий уровень входного сигнала КМОП. Это служит причиной использования высокоскоростной ТТЛ-совместимой КМОП (*HCT*) технологии и усовершенствованной высокоскоростной ТТЛ-совместимой КМОП (*AHCT*) технологии, Элементы этих технологий принимают входные логические уровни ТТЛ и формируют выходные логические уровни КМОП. Микросхемы этих семейств работают немного медленнее, чем их КМОП аналоги. Все КМОП микросхемы чувствительны к электростатическим разрядам (ESD), вызываемым статическим электричеством. Перед работой с КМОП микросхемами обязательно следует заземляться (прим.

переводчика: используя, например, специальный браслет), иначе можно их испортить

Стоимость микросхем серии 74xx невысока. Микросхемы новых серий часто бывают дешевле, чем устаревшие. Микросхемы семейства LS широко распространены и надежны и подходят для лабораторных и домашних проектов, где не предъявляются специальные требования по производительности.

Микросхемы со стандартом напряжения питания 5 В стали исчезать в середине 1990-х, когда транзисторы стали настолько маленькими по размерам, что не могли выдерживать такое напряжение. Более того, низкое напряжение влечет за собой более низкое энергопотребление. Сегодня используются напряжения 3,3; 2,5; 1,8; 1,2 В и даже ниже. Разнообразие питающих напряжение микросхем вызывает трудности согласования микросхем с различными источниками питания. В **Табл. А.3** перечислены некоторые семейства низковольтных микросхем. Не все компоненты серии 74xx доступны в каждом семействе.

Все низковольтные семейства микросхем построены на КМОП транзисторах, основе современных интегральных схем. Они работают в широком диапазоне напряжений V_{DD}, но скорость работы падает при низком напряжении. Микросхемы семейств низковольтных КМОП (*LVC*)

и усовершенствованных низковольтных КМОП (*ALVC*) обычно используются при напряжениях 3,3; 2,5 или 1,8 В. *LVC* микросхемы выдерживают входное напряжение до 5,5 В, поэтому могут использоваться совместно с микросхемами КМОП и ТТЛ стандарта 5 В. Микросхемы семейства усовершенствованных ультра низковольтных КМОП (*AUC*) обычно используют напряжения 2,5; 1,8 или 1,2 В и работают очень быстро. Микросхемы ALVC и AUC выдерживают входное напряжение до 3,6 В, поэтому их можно использовать с микросхемами стандарта 3,3 В.

В микросхемах FPGA обычно присутствуют отдельные напряжения питания для внутренней логики (называемой ядром) и для входных/выходных выводов. По мере развития FPGA напряжение питания ядра снизилось с 5 до 3,3; 2,5; 1,8 и 1,2 В для экономии энергии и во избежание выхода из строя очень маленьких по размерам транзисторов. FPGA имеют конфигурируемые входы/выходы, которые могут работать при различных напряжениях для обеспечения совместимости с остальными элементами системы.

Табл. А.3 Характеристики семейств низковольтной логики

	LVC			ALVC			AUC		
V_{dd} (В)	3,3	2,5	1,8	3,3	2,5	1,8	2,5	1,8	1,2
t_{pd} (нс)	4,1	6,9	9,8	2,8	3	?[1]	1,8	2,3	3,4
V_{IH} (В)	2	1,7	1,17	2	1,7	1,17	1,7	1,17	0,78
V_{IL} (В)	0,8	0,7	0,63	0,8	0,7	0,63	0,7	0,63	0,42
V_{OH} (В)	2,2	1,7	1,2	2	1,7	1,2	1,8	1,2	0,8
V_{OL} (В)	0,55	0,7	0,45	0,55	0,7	0,45	0,6	0,45	0,3
I_O (мА)	24	8	4	24	12	12	9	8	3
I_I (мА)	0,02			0,005			0,005		
I_{DD} (мА)	0,01			0,01			0,01		
C_{pd} (пФ)	10	9,8	7	27,5	23	?*	17	14	14
стоимость ($США)	0,17			0,20			не доступно		

* Задержка распространения и емкость не доступны на момент написания книги

А.7 СЕМЕЙСТВА ЛОГИЧЕСКИХ ЭЛЕМЕНТОВ

Интегральные схемы чаще всего помещаются в пластиковые или керамические корпуса. Корпуса выполняют несколько функций: соединение крошечных металлических выводов кристалла с большими выводами корпуса для простоты подключения, защита микросхемы от физических повреждений и рассеяние выделяемого микросхемой тепла на большей поверхности для охлаждения. Корпуса размещаются на макетной плате или печатной плате и соединяются проводниками в систему.

Корпуса

На **Рис. А.10** показаны разнообразные корпуса интегральных схем. Корпуса могут быть разделены на два главных класса: корпуса для монтажа в отверстия и корпуса поверхностного монтажа (*SMT*). Корпуса для монтажа в отверстия, как следует из их названия, имеют выводы, которые могут быть вставлены в отверстия в печатной плате или в гнездо. Корпуса с двухрядным расположением выводов (DIP) имеют два ряда выводов с расстоянием между выводов 0,1 дюйм. Корпус с матрицей стержневых выводов (PGA) содержит больше выводов в меньшем корпусе благодаря расположению выводов под корпусом. SMT корпуса припаиваются непосредственно к поверхности печатной платы, а не в отверстия.

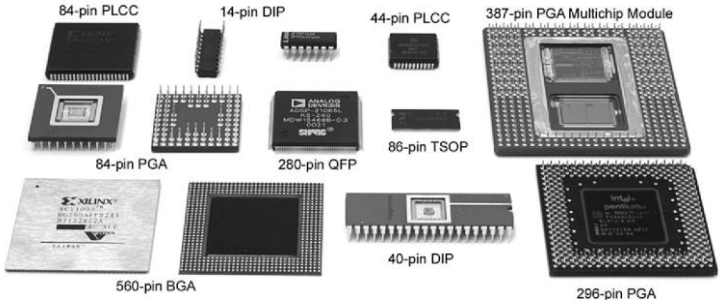


Рис. А.10 Корпуса интегральных схем

Контакты SMT микросхем называются выводами. Тонкий малогабаритный корпус (*TSOP*) состоит из двух рядов близко расположенных выводов (обычно расстояние между выводами составляет 0,02 дюйма). Пластмассовые кристаллоносители (*PLCC*) имеют J-образные выводы со всех четырех сторон, расстояние между выводами 0,05 дюймов. Их можно припаивать непосредственно на плату или размещать в специальных гнездах. Плоский корпус с четырёхсторонним расположением выводов (*QFP*) содержат большее количество контактов благодаря близкому расположению выводов с четырех сторон. Корпуса с матрицей шариковых контактов (*BGA*)

полностью исключают использование обычных выводов. Вместо этого, у них сотни маленьких шариков припоя на нижней стороне корпуса. Они тщательно размещаются на контактные площадки на печатной плате, затем нагреваются – припой плавится и соединяет корпус микросхемы с платой.

Макетные платы

Корпуса DIP удобно использовать для макетирования, поскольку их легко устанавливать на макетные платы. Макетная плата – пластмассовая плата, содержащая ряды гнезд, как показано на **Рис. А.11**.

Все пять отверстий в одном ряду соединены друг с другом. Каждый контакт корпуса вставляется в отверстие в отдельном ряду. В соседние отверстия в том же ряду можно подсоединить проводники для соединения контактов. В макетных платах часто есть отдельные столбцы объединенных отверстий вверху платы для распределения питания и общего провода.

На **Рис. А.11** показана макетная плата с мажоритарным вентилем, построенным на базе элементов И микросхемы 74LS08 и элементов ИЛИ микросхемы 74LS32.

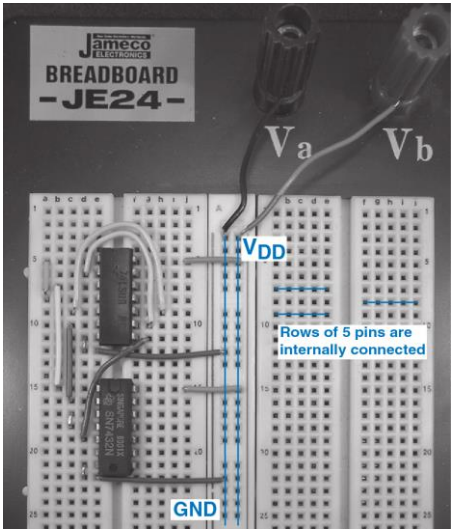


Рис. А.11 **Схема мажоритарного вентиля на макетной плате**

Схема показана на **Рис. А.12**. Все элементы на схеме помечены номером микросхемы (08 или 32) и номером контактов входов и

выходов (см. **Рис. А.1**). Обратите внимание, что те же самые соединения выполнены на макетной плате.

Входы соединены с контактами 1, 2 и 5 микросхемы 08, выходные сигналя снимаются с контакта 6 микросхемы 32. Питание и общий провод подаются на выводы 14 и 7 каждой микросхемы соответственно с вертикальных столбцов отверстий с линий питания и общего провода, которые подсоединены к разъемам Vb и Va. Разметка схемы ярлычками и проверка соединений – хороший способ избежать ошибок при макетировании.

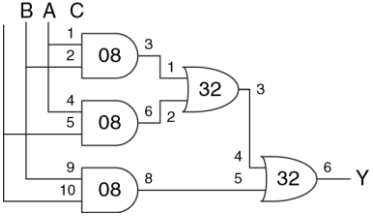


Рис. А.12 Схема мажоритарного вентиля с номерами микросхем и выводов

К сожалению, в данном случае легко подключить проводник не к тому отверстию или проводник может выпасть, поэтому макетирование требует концентрированного внимания (и обычно отладки в

лаборатории). Макетные платы пригодны только для прототипирования, а не для производства.

Печатные платы

Вместо макетных плат микросхемы можно припаивать на печатные платы (ПП). Печатные платы состоят из чередующихся слоев проводящей меди и изолирующей эпоксидной смолы. Медь вытравливается для формирования проводящих дорожек. Отверстия, называемые переходными, просверливают сквозь плату и металлизируют, чтобы соединить слои между собой. Печатные платы обычно разрабатываются в системах автоматизированного проектирования (САПР/CAD-программы). Можно протравить и просверлить свою плату в лаборатории или можно отправить проект платы на специализированное предприятие для недорогого массового производства. Предприятия имеют оборотное время длительностью несколько дней (или недель для дешевых партий товаров массового производства) и обычно выставляют счет в несколько сотен долларов на запуск производства и в несколько долларов за каждую плату средней сложности, при массовом выпуске.

Трассировка печатных плат обычно выполняется медью из-за ее низкой стоимости. Проводящие дорожки выполняются на изолирующем материале, обычно зеленом огнеупорном пластике FR4. Также в печатных платах есть медные слои питания и общего провода (земля),

расположенные между сигнальными слоями. На **Рис. А.13** показана печатная плата в разрезе. Сигнальные слои расположены сверху и снизу, слои питания и общего провода расположены посередине платы. Слои питания и земли имеют низкое сопротивление, и поэтому равномерно распределяют мощность по компонентам платы. Также они делают емкость и индуктивность проводящих дорожек одинаковой и предсказуемой.

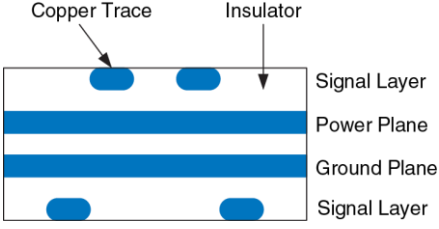


Рис. А.13 Печатная плата в разрезе

На **Рис. А.14** показана печатная плата компьютера Apple II+ 1970х годов. На ней стоит микропроцессор 6502. Ниже расположены 6 микросхем ROM на 16 кбит, образующие 12 кбайт памяти ROM, хранящей операционную систему. Три ряда из восьми микросхем DRAM на 16 кбит обеспечивают 48 кбайт памяти RAM. Справа расположены несколько рядов микросхем серии 74xx для дешифрации адресов

памяти и других целей. Линии между микросхемами – проводящие дорожки, соединяющие их. Точки на концах некоторых дорожек – сквозные отверстия, заполненные металлом.

Общий обзор

Большинство современных микросхем с большим количеством выводов корпусируются в SMT корпуса, в особенности в QFP и BGA. Для этих корпусов необходимо использовать печатную плату, а не макетную плату. Работа с корпусами BGA особенно сложна, поскольку необходимо специальное оборудование для сборки. Более того, контактные шарики невозможно проверить вольтметром или осциллографом при отладке в лаборатории, поскольку они спрятаны под корпусом.

Таким образом, разработчик должен учитывать проблему корпусов заранее, чтобы определить, будет ли использована макетная плата для отладки и потребуются ли BGA компоненты. Профессионалы редко пользуются макетными платами, когда они уверены в правильности соединения микросхем без экспериментирования.

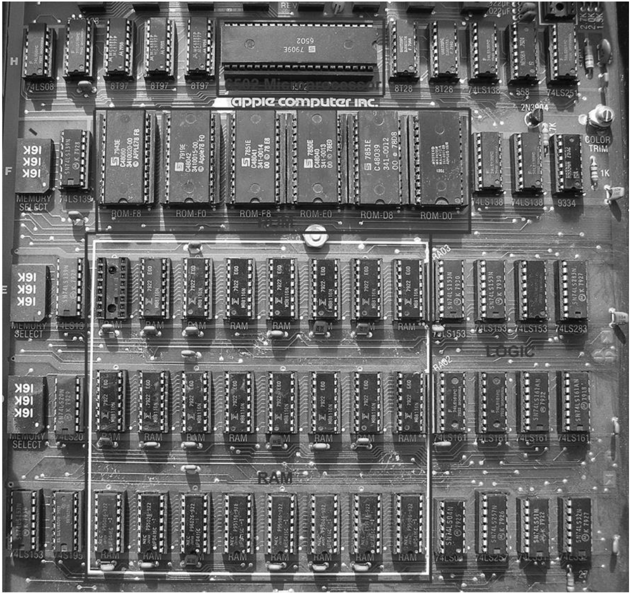


Рис. А.14 Печатная плата компьютера Apple II+

А.8 КОРПУСА И МОНТАЖ ИНТЕГРАЛЬНЫХ СХЕМ

Ранее мы полагали, что проводники – это эквипотенциальные соединения, которые имеют одинаковое напряжение по всей длине. На самом деле сигналы распространяются по проводникам со скоростью света в виде электромагнитных волн. Если проводники достаточно короткие, или сигналы изменяются медленно, то допущение эквипотенциальности приемлемо. Если проводник очень длинный или сигнал изменяется быстро, время прохождения сигнала по проводнику становится важным для точного определения задержки цепи. Мы должны моделировать такие проводники как линии передачи, в которых волна напряжения и тока распространяется со скоростью света. Когда волна достигает конца линии, она может отразиться в обратную сторону. Отражение может вызвать шумы и сбои в работе системы, если не приняты меры по его ограничению. Поэтому разработчик цифровых систем должен учитывать поведение линии передачи и точно определять задержку и шумовые эффекты в длинных проводниках.

Электромагнитные волны распространяются со скоростью света в конкретном веществе, что довольно быстро, но не мгновенно.

Скорость света ν зависит от диэлектрической проницаемости ε и магнитной проницаемости μ среды[21]: $\nu = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$

Скорость света в свободном пространстве $\nu = c = 3\times10^8$ м/с. Сигналы в печатных платах распространяются со скоростью, примерно вдвое меньшей вышеуказанной, поскольку изоляционный материал FR4 имеет диэлектрическую проницаемость в четыре раза больше, чем воздух. Таким образом, сигналы в печатной плате распространяются со скоростью $1{,}5\times10^8$ м/с или 15 см/нс. Задержка распространения сигнала по линии передачи длиной *l*

$$t_d = \frac{l}{\nu} \tag{A.4}$$

Волновое сопротивление линии передачи Z_0 (произносится “Z-нулевое”) – это отношение напряжения к току в волне, распространяющейся по линии: $Z_0 = V/I$. Это *не* сопротивление проводника (хорошая линия передачи в цифровой системе обычно имеет пренебрежимо малое сопротивление). Z_0 зависит от

[21] Емкость *C* и индуктивность *L* проводника связаны с диэлектрической и магнитной проницаемостью физической среды, в которой расположен проводник.

индуктивности и емкости линии (см. выкладки в **разделе А.8.7**) и обычно имеет значение от 50 до 75 Ом.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.5}$$

На**Рис. А.15** показано схематическое изображение линии передачи. Изображение напоминает коаксиальный кабель с внутренним проводником сигналов и внешним заземленным проводником, как в телевизионном кабеле.

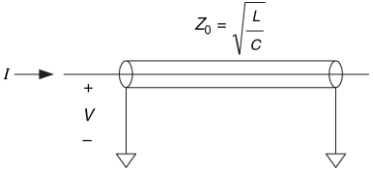


Рис. А.15 Обозначение линии передачи

Ключом к пониманию поведения линии передачи является представление волны напряжения, распространяющейся вдоль линии со скоростью света. Когда волна достигает конца линии, она может быть поглощена или отражена, в зависимости от оконечного устройства

или нагрузки на конце. Отраженные волны распространяются в обратную сторону по линии, накладываясь на уже существующее в линии напряжение. Нагрузка бывает согласованной, холостого хода, короткого замыкания и рассогласованной. В следующих разделах рассматривается распространение волны в линии и что происходит, когда она достигает нагрузки.

A.8.1 Согласованная нагрузка

На **Рис. A.16** показана линия передачи длиной *l* с согласованной нагрузкой – это означает, что полное сопротивление нагрузки *ZL* равно волновому сопротивлению Z_0.

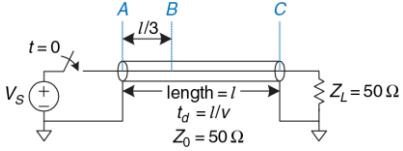


Рис. A.16 Линия передачи с согласованной нагрузкой

Линия передачи имеет волновое сопротивление 50 Ом. Один конец линии подсоединен к источнику напряжения через ключ, который замыкается в момент времени $t = 0$. Другой конец подключен к согласованной нагрузке в 50 Ом. В этом разделе анализируются

напряжения и токи в точках *A*, *B* и *C* – в начале линии, на расстоянии в одну треть длины линии и в конце линии, соответственно.

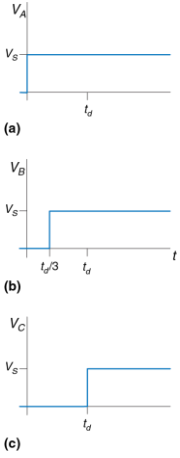


Рис. А.17РФормы напряжения в схеме на Рис. А.16 в точках *A*, *B* и *C*

На **Рис. А.17** показаны зависимости напряжения в точках *A*, *B* и *C* от времени. Первоначально в линии нет напряжения, и не течет ток, поскольку переключатель разомкнут. В момент времени $t = 0$ ключ замыкается, и источник напряжения генерирует в линии волну с напряжением $V = V_S$. Такая волна называется падающей волной. Поскольку волновое сопротивление составляет Z_0, волна имеет ток $I = V_S/Z_0$. Напряжение достигает начала линии (точка *A*) сразу, как показано на **Рис. А.17 (а)**. Волна распространяется по линии со скоростью света. В момент времени $t_d/3$ волна достигает точки *B*. Напряжение в этой точке резко возрастает от 0 до V_S, как показано на **Рис. А.17 (b)**. В момент времени t_d падающая волна достигает точки *C* на конце линии и там напряжение тоже повышается. Весь ток *I* течет через сопротивление Z_L, порождая напряжение на сопротивлении $Z_L I = Z_L (V_S/Z_0) = V_S$, поскольку $Z_L = Z_0$.

Напряжение соответствует волне, распространяющейся вдоль линии передачи. Таким образом, волна поглощается сопротивлением нагрузки, и линия передачи достигает установившегося режима. В установившемся режиме линия передачи ведет себя как идеальный эквипотенциальный проводник, поскольку, в итоге, это только провод. Напряжение во всех точках линии должно быть одинаковым.

На **Рис. А.18** показана эквивалентная схема установившегося режима схемы, представленной на **Рис. А.16**. Напряжение равно V_S в каждой точке проводника.

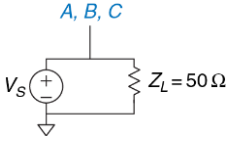


Рис. А.18 Эквивалентная схема линии на Рис. А.16 в установившемся режиме

Пример А.2 ЛИНИЯ ПЕРЕДАЧИ С СОГЛАСОВАННЫМ СОПРОТИВЛЕНИЕМ ИСТОЧНИКА И НАГРУЗКИ

На **Рис. А.19** показана линия передачи с согласованными сопротивлением источника и нагрузки Z_S и Z_L. Нарисуйте изменения напряжения в точках *A*, *B* и *C*. Когда система достигает установившегося режима и какова эквивалентная схема установившегося режима?

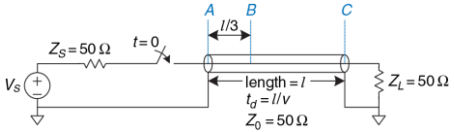


Рис. А.19 Линия передачи с согласованными сопротивлениями источника и нагрузки

Решение: Когда источник напряжения имеет сопротивление Z_S, подключенное последовательно с линией передачи, часть напряжения падает на Z_S и оставшееся напряжение распространяется в линии передачи. Сначала линия передачи ведет себя как сопротивление Z_0, поскольку нагрузка в конце линии не может влиять на поведение линии из-за задержки распространения волны со скоростью света. Таким образом, применяя уравнение делителя напряжения, напряжение падающей волны в линии равно

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \qquad \text{(A.6)}$$

Так, в момент времени $t = 0$ волна напряжения $V = V_S/2$ начинает распространяться в линии из точки *A*. Как и ранее, сигнал достигает точки *B* в момент времени $t_d/3$ и точки *C* – в момент времени t_d, как показано на **Рис. А.20**. Весь ток поглощается сопротивлением нагрузки Z_L, поэтому схема достигает установившегося режима в момент времени $t = t_d$. В установившемся режиме вся линия находится под напряжением $V_S/2$ – именно так, как можно предсказать по эквивалентной схеме установившегося режима, представленной на **Рис. А.21**.

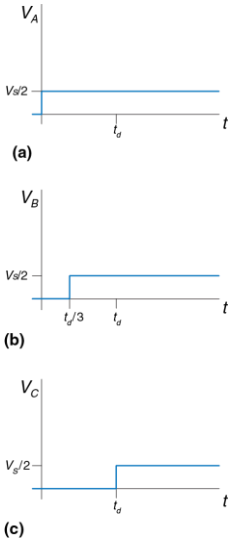


Рис. А.20 Формы напряжений в схеме на Рис. А.19 в точках *A*, *B* и *C*

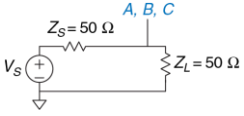


Рис. А.21 Эквивалентная схема линии на Рис. А.19 в установившемся режиме

А.8.2 Нагрузка холостого хода

Если сопротивление нагрузки не равно Z_0, нагрузка не может поглощать весь ток и некоторая часть волны должна отражаться. На **Рис. А.22** показана линия передачи с нагрузкой холостого хода. Через такую нагрузку не может течь ток, поэтому ток в точке С всегда должен быть равен 0.

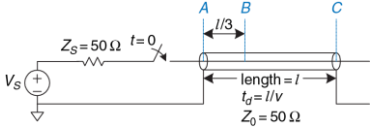


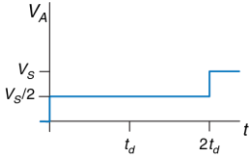
Рис. А.22 Линия передачи с разомкнутым концом

Напряжение в линии изначально равно 0. В момент времени t = 0, ключ замыкается и волна напряжения $V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2}$ начинает распространяться в линии.

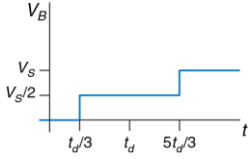
Следует обратить внимание, что исходная волна такая же, как в **примере А.2**, и она не зависит от оконечного устройства, поскольку нагрузка на конце линии не может влиять на ее поведение в начале линии, по крайней мере, пока не пройдет время $2t_d$ Эта волна достигает

точки *B* в момент времени $t_d/3$ и точки *C* в момент времени t_d, как показано на **Рис. А.23**. Когда падающая волна достигает точки *C*, она не может распространяться вперед, поскольку провод разомкнут. Вместо этого, она должна отразиться обратно к источнику. Отраженная волна также имеет напряжение $\frac{V_S}{2}$, поскольку нагрузка холостого хода полностью отражает волну.

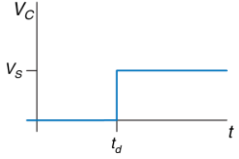
Напряжение в любой точке равно сумме падающей и отраженной волн. В момент времени $t = t_d$ напряжение в точке *C* равно $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. Отраженная волна достигает точки B в момент времени $5t_d/3$ и точки *A* в момент времени $2t_d$. Когда волна достигает точки *A*, она поглощается сопротивлением источника, которое согласовано с волновым сопротивлением линии. Таким образом, система достигает установившегося состояния в момент времени $t = 2t_d$ и линия передачи становится эквивалентной эквипотенциальному проводнику с напряжением V_S и током $I = 0$.



(a)



(b)



(c)

Рис. А.23 Формы напряжений в схеме на Рис. А.22 в точках *A*, *B* и *C*

А.8.3 Нагрузка короткого замыкания

На **Рис. А.24** показана линия передачи, конец которой короткозамкнут на общий провод (заземлен). Таким образом, напряжение в точке *C* должно быть всегда равно 0.

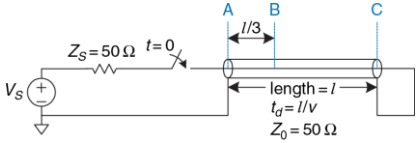


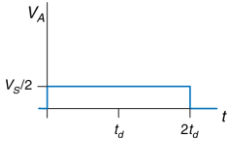
Рис. А.24 **Короткозамкнутая линия передачи**

Как и в предыдущих примерах, изначально напряжения в линии равны 0.

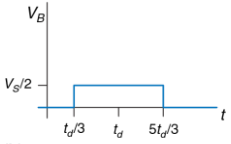
Когда ключ замыкается, волна напряжения $V = \frac{V_S}{2}$, начинает распространяться по линии (**Рис. А.25**). Когда волна достигает конца линии, она должна отразиться, поменяв полярность.

Отраженная волна с напряжением $V = \frac{-V_S}{2}$ накладывается на падающую волну, благодаря чему напряжение в точка *C* остается равным 0. Отраженная волна достигает источника в момент времени $t = 2t_d$ и поглощается сопротивлением источника. В этот момент система

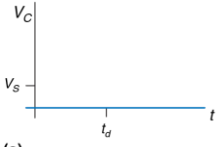
достигает установившегося состояния и линия передачи становится эквивалентна потенциальному проводнику с напряжением $V = 0$.



(a)



(b)



(c)

Рис. А.25 Формы напряжений в схеме на рисунке А.24 в точках *A*, *B* и *C*

А.8.4 Рассогласованная нагрузка

Сопротивление нагрузки считается рассогласованным, если оно не равно волновому сопротивлению линии. В общем случае, когда падающая волна достигает рассогласованной нагрузки, она частично поглощается и частично отражается. Коэффициент отражения k_r показывает, какая часть падающей волны V_i отразилась: $V_r = k_r V_i$.

В **разделе А.8.8** приводится вывод коэффициента отражения с использованием соображений сохранения величины тока. Также показано, что когда падающая волна, распространяющаяся по линии передачи с волновым сопротивлением Z_0, достигает нагрузки с сопротивлением на конце линии Z_T, коэффициент отражения равен:

$$k = \frac{Z_T - Z_0}{Z_T + Z_0} \qquad \text{(A.7)}$$

Следует обратить внимание на несколько частных случаев. Если цепь на конце разомкнута ($Z_T = \infty$), то $k_r = 1$, потому что падающая волна полностью отражается (так что ток за пределами линии остается равным нулю). Если на конце цепи короткое замыкание ($Z_T = 0$), то $k_r = -1$, потому что падающая волна отражается с отрицательной полярностью (так что напряжение на конце линии остается равным нулю). Если

нагрузка согласована ($Z_T = Z_0$), то $k_r = 0$, потому что падающая волна полностью поглощается.

На **Рис. А.26** показаны отражения в линии передачи с рассогласованной нагрузкой сопротивлением 75 Ом. $Z_T = Z_L = 75$ Ом и $Z_0 = 50$ Ом, поэтому $k_r = 1/5$. Как и в предыдущих примерах первоначально напряжение в линии равно 0. Когда ключ замыкается, волна напряжения $V = \frac{V_S}{2}$ распространяется по линии, достигая конца в момент времени $t = t_d$. Когда падающая волна достигает нагрузки на конце линии, одна пятая волны отражается, а оставшиеся четыре пятых протекают через сопротивление нагрузки. Таким образом, отраженная волна имеет напряжение $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. Суммарное напряжение в точке *C* складывается из падающего и отраженного напряжения $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. В момент времени $t = 2t_d$ отраженная волна достигает точки *A*, где она поглощается согласованной нагрузкой в 50 Ом, Z_S. На **Рис. А.27** показаны графики токов и напряжений в линии. Следует снова обратить внимание, что в установившемся режиме (в данном случае в момент времени $t > 2t_d$) линия передачи эквивалентна эквипотенциальному проводнику, как показано на **Рис. А.28**. В установившемся режиме система работает как делитель напряжения, поэтому

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + ZS}\right) = \left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

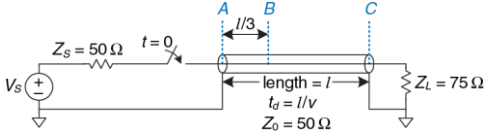


Рис. А.26 **Линия передачи с несогласованной нагрузкой**

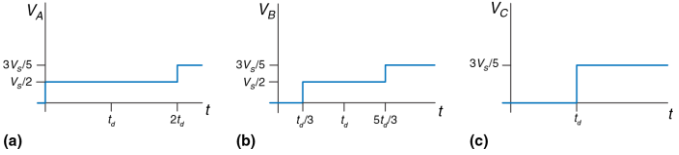


Рис. А.27 **Формы напряжений в схеме на Рис. А.26 в точках *A*, *B* и *C***

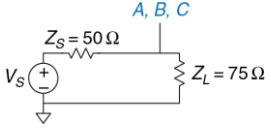


Рис. А.28 **Эквивалентная схема линии на Рис. А.26 в установившемся режиме**

Отражения могут происходить с обеих сторон линии передачи. На **Рис. А.29** показана линия передачи с сопротивлением источника Z_S в 450 Ом и разомкнутой цепью на конце. Коэффициенты отражения от нагрузки и источника, k_{rL} и k_{rS}, равны 1 и 4/5, соответственно. В данном случае волны отражаются от обоих концов линии передачи до достижения установившегося режима.

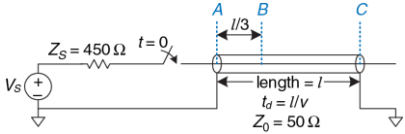


Рис. А.29 Линия передачи с несогласованным сопротивлением источника и нагрузки

Диаграмма отражений, представленная на **Рис. А.30**, помогает представить отражения от обоих концов линии передачи. Горизонтальная ось представляет расстояние вдоль линии передачи, вертикальная ось представляет время, возрастающее сверху вниз. Две стороны диаграммы отражений представляют концы с источником и нагрузкой в линии передачи, точки *A* и *C*. Волны падающего и отраженного сигналов изображены диагональными линиями между точками *A* и *C*. В момент времени $t = 0$ сопротивление источника и линия передачи ведут себя как делитель напряжения, начиная

распространять волну напряжением $\frac{V_S}{10}$ от точки *A* к точке *C*. В момент времени $t = t_d$ сигнал достигает точки C и полностью отражается ($k_{rL} = 1$). В момент времени $t = 2t_d$ отраженная волна напряжением $\frac{V_S}{10}$ достигает точки *A* и отражается с коэффициентом отражения $k_{rS} = 4/5$, порождая волну напряжением $\frac{2V_S}{25}$, распространяющуюся до точки *C*, и так далее.

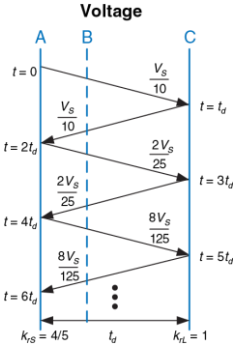


Рис. А.30 Диаграмма отражений для линии на Рис. А.29

Напряжение в определенный момент времени в любой точке линии передачи равно сумме напряжений всех падающих и отраженных волн. Так, в момент времени $t = 1.1t_d$ напряжение в точке C равно $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. В момент времени $t = 3.1t_d$ напряжение в точке C равно $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, и так далее. На **Рис. А.31** показан график зависимости напряжения от времени. Когда t стремится к бесконечности, достигается установившийся режим с напряжениями $V_A = V_B = V_C = V_S$.

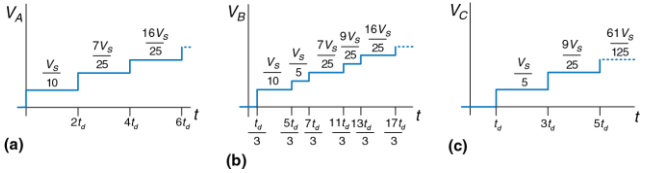


Рис. А.31 Формы напряжения и тока для линии на Рис. А.29

А.8.5 Когда нужно применять модели линии передачи

Модели линии передачи для проводников необходимы всегда, когда задержка распространения сигнала в проводе t_d дольше, чем часть (например, 20%) фронта сигнала (время нарастания и спада). Если задержка распространения сигнала в проводе меньше, то ее влияние на задержку распространения сигнала незначительно и отражения рассеиваются в процессе прохождения сигнала. Если задержка распространения сигнала в проводе больше, ее следует учитывать, чтобы точно предсказать задержку распространения и форму сигнала. В частности отражения могут исказить цифровые характеристики формы сигнала, что приводит к неверным логическим операциям.

Следует помнить, что сигналы распространяются в печатной плате со скоростью примерно 15см/нс. Для ТТЛ логики, где фронты составляют 10 нс, проводники следует моделировать линиями передачи, только если они длиннее 30 см (10 нс × 15 см/нс × 20%). Печатные проводники плат обычно короче 30 см, поэтому большинство печатных проводников можно моделировать как идеальные эквипотенциальные проводники. Напротив, многие современные микросхемы имеют фронты длительностью 2 нс или меньше, поэтому печатные проводники длиннее 6 см (около 2,5 дюймов) следует моделировать как линии

передачи. Очевидно, что использование фронтов сигналов, которые короче, чем необходимо, ведет только к трудностям для разработчика.

У макетных плат нет плоскости заземления, поэтому электромагнитные поля сигналов неоднородны и трудно поддаются моделированию. Более того, эти поля взаимодействуют с другими сигналами. Это может привести к странным отражениям и взаимным помехам между сигналами. Таким образом, макетные платы ненадежны на частотах выше нескольких мегагерц.

Напротив, печатные платы имеют хорошие линии передачи с постоянными волновым сопротивлением и скоростью распространения по всей линии. Пока они нагружены сопротивлением источника или нагрузки, согласованным с сопротивлением линии, проводники печатных плат не испытывают отражений.

А.8.6 Правильное подключение нагрузки к линии передачи

Существует два вида правильного подключения нагрузки к линии передачи, показанные на **Рис. А.32**. При *параллельном подключении нагрузки* задающее устройство имеет малое сопротивление ($Z_S \approx 0$). Нагрузочный резистор Z_L с сопротивлением Z_0 располагается параллельно нагрузке (между входом принимающего устройства и общим проводом).

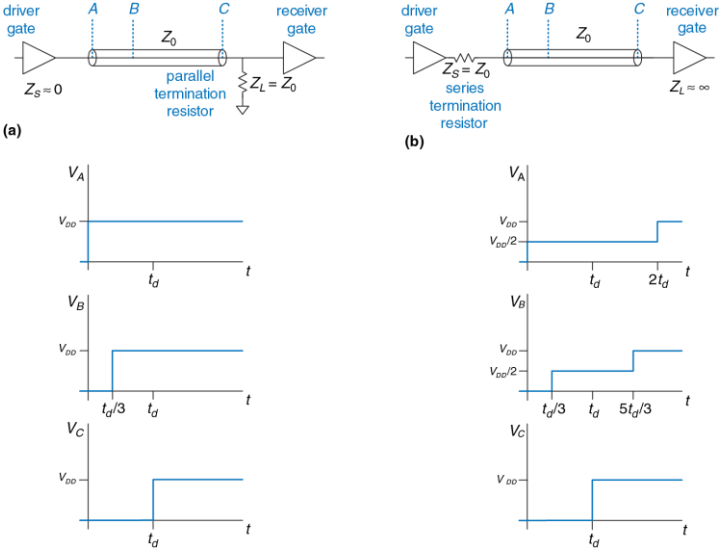


Рис. А.32 **Варианты подключения нагрузки к линии передачи: (а) параллельная, (б) последовательная**

Когда задающее устройство переключает напряжение от 0 до V_{DD}, оно начинает распространять волну с напряжением V_{DD} по линии. Волна поглощается согласованной нагрузкой, и отражений не происходит. При *последовательном подключении нагрузки* сопротивление Z_S располагается последовательно с задающим устройством, чтобы повысить сопротивление источника до Z_0. Нагрузка имеет высокое сопротивление ($Z_L \approx \infty$). Когда задающее устройство переключается, оно начинает распространять волну с напряжением $V_{DD}/2$ по линии. Волна отражается от открытого конца линии и возвращается, поднимая напряжение в линии до V_{DD}. Волна поглощается сопротивлением источника. Обе схемы похожи тем, что напряжение на принимающем устройстве изменяется от 0 до V_{DD} в момент времени $t = t_d$, именно так, как и требуется. Они различаются потребляемой мощностью и формой волны в каждой точке линии. Параллельная нагрузка постоянно рассеивает мощность на нагрузочном резисторе, когда линия находится под высоким напряжением. Последовательная нагрузка не рассеивает мощность постоянного тока, поскольку нагрузкой является разомкнутая цепь. Однако, в линиях с последовательной нагрузкой точки в середине линии передачи изначально находятся под напряжением $V_{DD}/2$ до тех пор, пока их не достигнет отраженная волна. Если другие элементы будут подключены к середине линии, на них будет кратковременно действовать неправильный логический уровень. Поэтому последовательная нагрузка лучше работает в соединениях типа

"точка-точка" с одним задающим и одним принимающим устройством. Параллельная нагрузка лучше для шины со множеством принимающих устройств, поскольку принимающие устройства в середине линии никогда не подвергаются воздействию неправильных логических уровней.

A.8.7 Вывод формулы для Z0*

Z_0 равно отношению напряжения к току в волне, распространяющейся вдоль линии передачи. В данном разделе выводится формула для Z_0; вывод предполагает наличие некоторых знаний, полученных из анализа схем типа резистор-индуктивность-емкость (RLC).

Предположим, что ко входу полубесконечной линии передачи приложено ступенчатое напряжение (то есть отражений не происходит). На **Рис. A.33** показана полубесконечная линия передачи и модель части этой линии длиной *dx*. *R*, *L*, и *C* – *величины сопротивления, индуктивности и емкости на единицу длины.* На **Рис. A.33 (b)** показана модель линии передачи с резистивным компонентом *R*. Такая модель называется моделью линии передачи с потерями, поскольку энергия рассеивается или теряется в сопротивлении проводника.

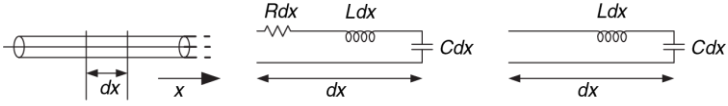


Рис. А.33 **Модели линии передачи: (а) полубесконечный провод, (b) линия с потерями, (c) идеальная линия**

Однако, часто эти потери пренебрежимо малы, и можно упростить анализ, не принимая во внимание сопротивление и считая линию передачи идеальной, как показано на **Рис. А.33 (c)**.

Напряжение и ток являются функциями времени и расстояния в линии передачи, как показано в **уравнениях (А.8)** и **(А.9)**.

$$\frac{\partial}{\partial x}V(x,t) = L\frac{\partial}{\partial t}I(x,t) \tag{A.8}$$

$$\frac{\partial}{\partial x}I(x,t) = C\frac{\partial}{\partial t}V(x,t) \tag{A.9}$$

Если взять производную по расстоянию от **уравнения (А.8)** и производную по времени от **уравнения (А.9)** и затем подставить одно в другое, то получим **уравнение (А.10)** – волновое уравнение.

$$\frac{\partial^2}{\partial x^2}V(x,t) = LC\frac{\partial^2}{\partial t^2}V(x,t) \qquad \text{(A.10)}$$

Z_0 равно отношению напряжения к току в линии передачи, как показано на **Рис. А.34 (а)**. Z_0 должно быть независимым от длины линии, поскольку поведение волны не может зависеть от удаленных объектов. Поскольку оно не зависит от длины, сопротивление должно быть равно Z_0 после удлинения линии на величину dx, как показано на **Рис. А.34 (b)**.

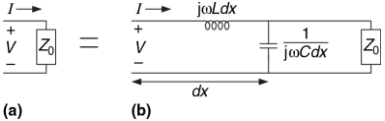


Рис. А.34 Модели линии передачи: (а) для всей линии, (b) с дополнительным отрезком длиной dx

Используя сопротивления индуктивности и емкости можно переписать соотношение на **Рис. А.34** в виде уравнения:

$$Z_0 = j\omega L dx + [Z_0 || (1/(j\omega C dx))] \qquad \text{(A.11)}$$

После преобразований получим:

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 LCdx = 0 \qquad \text{(A.12)}$$

В пределе при *dx* стремящемся к 0, последнее слагаемое исчезает, и в итоге получается:

$$Z_0 = \sqrt{\frac{L}{C}} \qquad \text{(A.13)}$$

A.8.8 Вывод формулы для коэффициента отражения*

Формула для коэффициента отражения выводится, используя принцип сохранения тока. На **Рис. A.35** показана линия передачи с волновым сопротивлением Z_0 и сопротивлением нагрузки Z_L. Предположим, что падающая волна имеет напряжение V_i и ток I_i. Когда волна достигает нагрузки, некоторый ток I_L протекает через сопротивление нагрузки, вызывая падение напряжения V_L. Оставшийся ток отражается обратно в линию в виде волны с напряжением V_r и током I_r. Z_0 равно отношению напряжения к току в волнах, распространяющихся вдоль линии, поэтому $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

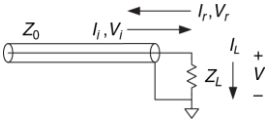


Рис. А.35 Линия передачи с падающими, отраженными и нагрузочными напряжениями и токами

Напряжение в линии равно сумме напряжений падающей и отраженной волн. Ток, протекающий в линии в положительном направлении, равен разности между токами падающей и отраженной волн.

$$V_L = V_i + V_r \tag{A.14}$$

$$I_L = I_i - I_r \tag{A.15}$$

Используя закон Ома, и подставляя выражения для I_L, I_i и I_r в **уравнение (А.15)**, получим

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A.16}$$

После преобразований найдем решение для коэффициента отражения k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \qquad \text{(A.17)}$$

А.8.9 Подводя итог

С помощью линий передачи можно смоделировать тот факт, что сигналам необходимо время для распространения по длинным проводникам, поскольку скорость света конечна. Идеальная линия передачи имеет одинаковую величину индуктивности L и емкости на единицу длины и нулевое сопротивление. Линия передачи характеризуется волновым сопротивлением Z_0 и задержкой *распространения сигнала* t_d, которые можно вывести, зная индуктивность, емкость и длину проводника. Линии передачи имеют значительную временную задержку и шумовые эффекты при распространении сигналов, у которых длительность нарастания/спада меньше, чем примерно $5t_d$. Это означает. что для систем со временем нарастания/спада 2 нс, печатные проводники плат длиннее 6 см должны рассматриваться как линии передачи для того, чтобы верно понимать их поведение.

Цифровая система, состоящая из элемента, управляющего длинным проводником, подключенным ко входу второго элемента, может быть смоделирована линией передачи, показанной на **Рис. А.36**.

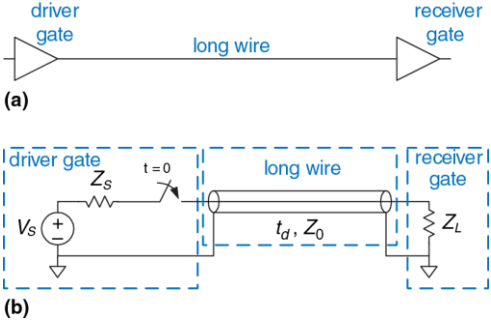


Рис. А.36 Цифровая система, смоделированная с помощью линии передачи

Источник напряжения, сопротивление источника Z_S и ключ имитируют первый элемент, переключающийся из 0 в 1 в момент времени 0. Элемент-источник не может поддерживать бесконечно большой ток, это моделируется при помощи Z_S. Z_S обычно мало для логических схем, но разработчик может добавить сопротивление, чтобы повысить Z_S и согласовать сопротивление линии. Вход второй схемы моделируется

при помощи Z_L. У КМДП схем обычно малый входной ток, поэтому величина Z_L может быть близка к бесконечности. Разработчик также может поставить параллельный резистор рядом со вторым элементом между его входом и общим проводом, чтобы Z_L согласовывало сопротивление линии.

Когда первый элемент переключается, волна напряжения начинает распространяться по линии передачи. Сопротивление источника и линия передачи формируют делитель напряжения, поэтому напряжение в падающей волне равно

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A.18}$$

В момент времени t_d волна достигает конца линии. Часть волны поглощается сопротивлением нагрузки, а часть отражается. Коэффициент отражения k_r показывает, какая часть волны отразилась. $k_r = V_r/V_i$, где V_r – напряжение отраженной волны, V_i – напряжение падающей волны.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A.19}$$

Отраженная волна складывается с уже существующим в линии напряжением. Она достигает источника в момент времени $2t_d$, где часть

поглощается, а часть снова отражается. Отражения продолжаются в обе стороны, и напряжение в линии в итоге достигает значения, которое было бы, если бы линия была простым эквипотенциальным проводником.

А.9 ЭКОНОМИКА

Хотя разработка цифровых схем – забавное занятие и многие готовы заниматься этим бесплатно, большинство разработчиков и компаний стремятся заработать деньги. Поэтому экономические соображения – главный фактор при принятии проектных решений.

Стоимость цифровой системы может быть разделена на единовременные инженерные издержки (nonrecurring engineering costs – NRE) и периодические издержки (recurring costs). Единовременные издержки включают в себя стоимость разработки системы. Она включает в себя зарплату команды разработчиков, затраты на затраты на вычислительные мощности и программное обеспечение и стоимость производства первого рабочего образца. Полная стоимость разработчика в США в 2012 году (включая зарплату, страховку, пенсионное обеспечение и компьютер с необходимыми средствами разработки) составляла примерно $200 000 в год, так что затраты на разработку могут быть очень значительны. Периодические издержки –

стоимость каждого дополнительного образца, которая включает в себя компоненты, производство, маркетинг, техническую поддержку и транспортировку.

Отпускная цена должна покрывать не только стоимость системы, а также и другие издержки вроде стоимости аренды помещений, налоги и зарплату персонала, которые непосредственно не задействованы в разработке (например, сторож и исполнительный директор). После всех этих издержек компании необходимо еще получить прибыль.

Пример А.3 БЕН ПЫТАЕТСЯ ЗАРАБОТАТЬ

Бен Битдиддл разработал хитроумную схему для подсчета дождевых капель. Он решает продать прибор и попытаться заработать немного денег, но ему нужна помощь, чтобы решить, как реализовать схему. Он решает использовать либо FPGA, либо ASIC. Набор для разработки и тестирования на FPGA стоит $1500. Каждая FPGA стоит $17. ASIC стоит $600 000 за набор шаблонов и $4 за микросхему.

Независимо от того, какой метод исполнения он выберет, Бену необходимо установить корпусированную микросхему на печатную плату, которая будет стоить $1,50 за штуку. Он считает, что сможет продавать 1000 приборов в месяц. Бен заставил команду талантливых студентов разрабатывать схему в качестве выпускного проекта, поэтому разработка обошлась ему бесплатно.

Если продажная цена будет в два раза выше себестоимости (100% лимит прибыли) и срок эксплуатации изделия 2 года, какое исполнение лучше выбрать?

Решение: Бен выясняет полную стоимость каждой реализации за 2 года, расчеты приведены в **Табл. А.4**. За 2 года Бен рассчитывает продать 24 000 устройств, общая стоимость дана в **Табл. А.4** в каждой графе. Если срок эксплуатации составляет всего 2 года, то реализация на FPGA очевидно побеждает. Стоимость одного изделия составляет $445 500/24 000 = $18,56, и продажная цена составляет $37,13 за единицу товара, чтобы получить 100% прибыль. Вариант на ASIC стоил бы $732000/24000 = $30,50 и продавался бы за $61 за изделие.

Табл. А.4 Сравнение стоимости ASIC и FPGA

Стоимость	ASIC	FPGA
Единовременные расходы	$600 000	$1500
микросхема	$4	$17
печатная плата	$1,50	$1,50
ВСЕГО	$600000 + (24000 × $5,50) = $732000	$1500 + (24000 × $18,50) = $445 500
за единицу	$30,50	$18,56

Пример А.4 БЕН СТАНОВИТСЯ ЖАДНЫМ

После того, как Бен увидел рекламу своего товара, он решает, сможет продавать больше микросхем в месяц, чем первоначально планировалось. Если бы он выбрал реализацию на ASIC, сколько приборов в месяц ему бы надо было продать, чтобы эта реализация стала более прибыльной, чем FPGA?

Решение: Бен определяет минимальное количество изделий N, которое ему необходимо продать за 2 года:

$$\$600{,}000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

Решение уравнения дает N = 46039 устройств или 1919 устройств в месяц. Ему придется почти удвоить ежемесячные продажи, чтобы получить прибыть от варианта с ASIC.

Пример А.5 БЕН СТАНОВИТСЯ МЕНЕЕ ЖАДНЫМ

Бен понимает, что он слишком многого хочет, и не думает, что сможет продать больше 1000 устройств в месяц. Но он считает, что срок эксплуатации может быть больше 2 лет. При объеме продаж 1000 устройств в месяц, каков должен быть срок эксплуатации, чтобы сделать ASIC реализацию прибыльной?

Решение: Если Бен продаст больше 46039 устройств, то ASIC реализация будет лучшим выбором. Так что Бену придется продавать по 1000 изделий в течение как минимум 47 месяцев (примерно), что составляет почти 4 года. К тому времени устройство, вероятно, станет устаревшим.

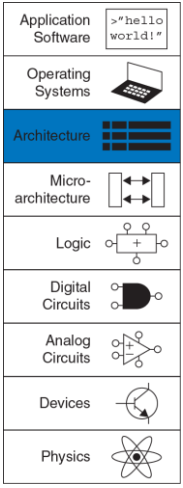
Микросхемы обычно приобретаются у дистрибьютора, а не напрямую у производителя (если не заказываются десятки тысяч микросхем). Digikey (www.digikey.com) – основной дистрибьютор, продающий большую номенклатуру изделий электроники. Jameco (www.jameco.com) и All Electronics (www.allelectronics.com) имеют разнообразные каталоги с конкурентными ценами, и хорошо подходят для любителей.

Инструкции архитектуры MIPS

B

В этом приложении приведены инструкции архитектуры MIPS, встречающиеся в этой книге. В **Табл. B.1** – **Табл. B.3** для каждой из инструкций приведены поля opcode и funct вместе с кратким описанием назначения инструкции. Используются следующие обозначения:

- `[reg]:` содержимое регистра
- `imm:` 16-битное поле непосредственного операнда для инструкции типа I
- `addr:` 26-битное поле адреса для инструкции типа J



- `SignImm:` 32-битное значение, полученное знаковым расширением из 16-битного непосредственного операнда
  `= {{16{imm[15]}}, imm}`
- `ZeroImm:` 32-битное значение, полученное расширением влево нулями из 16-битного непосредственного операнда
  `= {16'b0, imm}`
- `Address:` `[rs] + SignImm`
- `[Address]:` содержимое памяти по адресу `Address`
- `BTA:` целевой адрес ветвления[22]
  `= PC + 4 + (SignImm << 2)`
- `JTA:` адрес перехода
  `= {(PC + 4)[31:28], addr, 2'b0}`
- `label:` текстовая метка для адреса инструкции

[22] В симуляторе SPIM целевой адрес ветвления (BTA) вычисляется как PC + (SignImm << 2) потому, что в нем не реализовано отложенное ветвление. Поэтому, если вы используете ассемблер SPIM для генерации машинного кода для настоящего процессора MIPS, в каждой инструкции ветвления для компенсации необходимо вычитать единицу из поля с непосредственным операндом immediate.

Табл. B.1 Инструкции, отсортированные по полю opcode

Opcode	Имя	Описание	Операция
000000 (0)	`тип R`	все инструкции типа R	смотри Табл. B.2
000001 (1) (rt = 0/1)	`bltz rs, label / bgez rs, label`	ветвление, если меньше нуля (branch less than zero) / ветвление, если больше или равно нулю (branch greater than or equal to zero)	`if ([rs] < 0) PC = BTA/ if ([rs] ≥ 0) PC = BTA`
000010 (2)	`j label`	безусловный переход (jump)	`PC = JTA`
000011 (3)	`jal label`	безусловный переход с возвратом (jump and link)	`$ra = PC + 4, PC = JTA`
000100 (4)	`beq rs, rt, label`	ветвление, если равно (branch if equal)	`if ([rs] == [rt]) PC = BTA`
000101 (5)	`bne rs, rt, label`	ветвление, если не равно (branch if not equal)	`if ([rs] != [rt]) PC = BTA`
000110 (6)	`blez rs, label`	ветвление, если меньше или равно нулю (branch if less than or equal to zero)	`if ([rs] ≤ 0) PC = BTA`
000111 (7)	`bgtz rs, label`	ветвление, если больше нуля (branch if greater than zero)	`if ([rs] > 0) PC = BTA`

Opcode	Имя	Описание	Операция
001000 (8)	`addi rt, rs, imm`	сложение с непосредственным операндом со знаком, арифметическое переполнение вызвает исключение (add immediate)	`[rt] = [rs] + SignImm`
001001 (9)	`addiu rt, rs, imm`	сложение с непосредственным операндом со знаком, арифметическое переполнение не вызвает исключение (add immediate unsigned)	`[rt] = [rs] + SignImm`
001010 (10)	`slti rt, rs, imm`	установить, если меньше непосредственного операнда, сравнение выполняется со знаком (set less than immediate)	`[rs] < SignImm ? [rt] = 1 : [rt] = 0`
001011 (11)	`sltiu rt, rs, imm`	установить, если меньше непосредственного операнда, сравнение выполняется без знака (set less than immediate unsigned)	`[rs] < SignImm ? [rt] = 1 : [rt] = 0`

Opcode	Имя	Описание	Операция
001100 (12)	`andi rt, rs, imm`	побитовое логическое «И» с непосредственным операндом, расширенным влево нулями (and immediate)	`[rt] = [rs] & ZeroImm`
001101 (13)	`ori rt, rs, imm`	побитовое логическое «ИЛИ» с непосредственным операндом, расширенным влево нулями (or immediate)	`[rt] = [rs] |ZeroImm`
001110 (14)	`xori rt, rs, imm`	побитовое логическое «Исключающее ИЛИ» с непосредственным операндом, расширенным влево нулями (xor immediate)	`[rt] = [rs] ^ ZeroImm`
001111 (15)	`lui rt, imm`	загрузить непосредственный операнд в старшие 16 битов регистра, загрузить нули в младшие 16 битов (load upper immediate)	`[rt] = {imm, 16'b0}`
010000 (16) (rs = 0/4)	`mfc0 rt, rd` / `mtc0 rt, rd`	загрузка из регистра сопроцессора 0 / загрузка в регистр сопроцессора 0 (move from/to coprocessor 0)	`[rt] = [rd]` / `[rd] = [rt]` (rd принадлежит сопроцессору 0)

Opcode	Имя	Описание	Операция
010001 (17)	`тип F`	fop = 16/17: инструкции типа F	смотри **Табл. B.3**
010001 (17) (rt = 0/1)	`bc1f label/bc1t label`	fop = 8: ветвиться, если fpcond ЛОЖЬ/ПРАВДА (branch if fpcond is FALSE/TRUE)	`if (fpcond == 0) PC = BTA/ if (fpcond == 1) PC = BTA`
011100 (28) (func = 2)	`mul rd, rs, rt`	умножение слов с записью младших 32 бит результата в регистр общего назначения (multiply)	`[rd] = [rs] x [rt]`
100000 (32)	`lb rt, imm(rs)`	загрузка байта с расширением знака (load byte)	`[rt] = SignExt` $([Address]_{7:0})$
100001 (33)	`lh rt, imm(rs)`	загрузка полуслова с расширением знака (load halfword)	`[rt] = SignExt ([Address]15:0)`
100011 (35)	`lw rt, imm(rs)`	загрузка слова (load word)	`[rt] = [Address]`
100100 (36)	`lbu rt, imm(rs)`	загрузка байта без знака, с расширением слева нулями (load byte unsigned)	`[rt] = ZeroExt` $([Address]_{7:0})$
100101 (37)	`lhu rt, imm(rs)`	загрузка полуслова без знака, с расширением слева нулями (load halfword unsigned)	`[rt] = ZeroExt` $([Address]_{15:0})$

Opcode	Имя	Описание	Операция
101000 (40)	`sb rt, imm(rs)`	сохранение байта (store byte)	`[Address]7:0 = [rt]`$_{7:0}$
101001 (41)	`sh rt, imm(rs)`	сохранение полуслова (store halfword)	`[Address]15:0 = [rt]`$_{15:0}$
101011 (43)	`sw rt, imm(rs)`	сохранение слова (store word)	`[Address] = [rt]`
110001 (49)	`lwc1 ft, imm(rs)`	загрузка слова из математического сопроцессора 1 (load word to FP coprocessor 1)	`[ft] = [Address]`
111001 (56)	`swc1 ft, imm(rs)`	сохранение слова в математический сопроцессор 1 (store word to FP coprocessor 1)	`[Address] = [ft]`

Табл. B.2 Инструкции типа R, отсортированные по полю funct

Funct	Имя	Описание	Операция
000000 (0)	`sll rd, rt, shamt`	логический сдвиг влево (shift left logical)	`[rd] = [rt] << shamt`
000010 (2)	`srl rd, rt, shamt`	логический сдвиг вправо (shift right logical)	`[rd] = [rt] >> shamt`
000011 (3)	`sra rd, rt, shamt`	арифметический сдвиг вправо (shift right arithmetic)	`[rd] = [rt] >>> shamt`
000100 (4)	`sllv rd, rt, rs`	логический переменный сдвиг влево (shift left logical variable)	`[rd] = [rt] << [rs]`$_{4:0}$
000110 (6)	`srlv rd, rt, rs`	логический переменный сдвиг вправо (shift right logical variable)	`[rd] = [rt] >> [rs]`$_{4:0}$
000111 (7)	`srav rd, rt, rs`	арифметический переменный сдвиг вправо (shift right arithmetic variable)	`[rd] = [rt] >>> [rs]`$_{4:0}$
001000 (8)	`jr rs`	безусловный переход по регистру (jump register)	`PC = [rs]`
001001 (9)	`jalr rs`	безусловный переход по регистру с возвратом (jump and link register)	`$ra = PC + 4, PC = [rs]`

Funct	Имя	Описание	Операция
001100 (12)	`syscall`	системный вызов (system call)	`исключение типа «системный вызов»`
001101 (13)	`break`	точка останова (break)	`исключение типа «останов в контрольной точке»`
010000 (16)	`mfhi rd`	пересылка из регистра hi (move from hi)	`[rd] = [hi]`
010001 (17)	`mthi rs`	пересылка в регистр hi (move to hi)	`[hi] = [rs]`
010010 (18)	`mflo rd`	пересылка из регистра lo (move from lo)	`[rd] = [lo]`
010011 (19)	`mtlo rs`	пересылка в регистр lo (move to lo)	`[lo] = [rs]`
011000 (24)	`mult rs, rt`	умножение 32-битных операндов со знаком и 64-битным результатом (multiply)	`{[hi], [lo]} = [rs] ×[rt]`
011001 (25)	`multu rs, rt`	умножение 32-битных операндов без знака и 64-битным результатом (multiply unsigned)	`{[hi], [lo]} = [rs] ×[rt]`
011010 (26)	`div rs, rt`	деление со знаком (divide)	`[lo] = [rs]/[rt], [hi] = [rs]%[rt]`

Funct	Имя	Описание	Операция
011011 (27)	`divu rs, rt`	деление без знака (Divide unsigned)	`[lo] = [rs]/[rt], [hi] = [rs]%[rt]`
100000 (32)	`add rd, rs, rt`	сложение со знаком, арифметическое переполнение вызвает исключение (add)	`[rd] = [rs] + [rt]`
100001 (33)	`addu rd, rs, rt`	сложение без знака, арифметическое переполнение не вызвает исключение (add unsigned)	`[rd] = [rs] + [rt]`
100010 (34)	`sub rd, rs, rt`	Вычитание со знаком, арифметическое переполнение вызвает исключение (subtract)	`[rd] = [rs] - [rt]`
100011 (35)	`subu rd, rs, rt`	вычитание без знака, арифметическое переполнение не вызвает исключение (subtract unsigned)	`[rd] = [rs] - [rt]`
100100 (36)	`and rd, rs, rt`	побитовое логическое «И» (and)	`[rd] = [rs] & [rt]`
100101 (37)	`or rd, rs, rt`	побитовое логическое «ИЛИ» (or)	`[rd] = [rs] | [rt]`

Funct	Имя	Описание	Операция
100110 (38)	`xor rd, rs, rt`	побитовое логическое «исключающее ИЛИ» (xor)	`[rd] = [rs] ^ [rt]`
100111 (39)	`nor rd, rs, rt`	побитовое отрицание логического «ИЛИ» (nor)	`[rd] = ~([rs] | [rt])`
101010 (42)	`slt rd, rs, rt`	установить, если меньше, сравнение выполняется со знаком (set less than)	`[rs] <[rt] ?` `[rd] = 1 : [rd] = 0`
101011 (43)	`sltu rd, rs, rt`	установить, если меньше, сравнение выполняется без знака (set less than unsigned)	`[rs] <[rt] ?` `[rd] = 1 : [rd] = 0`

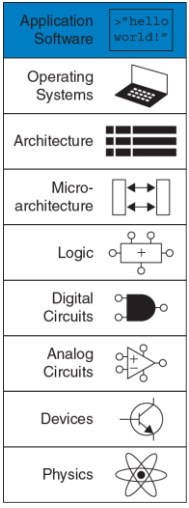
Табл. B.3 Инструкции типа F (fop = 16/17)

Funct	Имя	Описание	Операция
000000 (0)	`add.s fd, fs, ft /` `add.d fd, fs, ft`	сложение с плавающей точкой (FP add)	`[fd] = [fs] + [ft]`
000001 (1)	`sub.s fd, fs, ft /` `sub.d fd, fs, ft`	вычитание с плавающей точкой (FP subtract)	`[fd] = [fs] – [ft]`
000010 (2)	`mul.s fd, fs, ft /` `mul.d fd, fs, ft`	умножение с плавающей точкой (FP multiply)	`[fd] = [fs] × [ft]`
000011 (3)	`div.s fd, fs, ft /` `div.d fd, fs, ft`	деление с плавающей точкой (FP divide)	`[fd] = [fs]/[ft]`

Funct	Имя	Описание	Операция
000101 (5)	`abs.s fd, fs` / `abs.d fd, fs`	модуль с плавающей точкой (FP absolute value)	`[fd] = ([fs] < 0) ? [-fs] : [fs]`
000111 (7)	`neg.s fd, fs` / `neg.d fd, fs`	отрицание с плавающей точкой (FP negation)	`[fd] = [-fs]`
111010 (58)	`c.seq.s fs, ft` / `c.seq.d fs, ft`	сравнение с соотношением «равно» с плавающей точкой (FP equality comparison)	`fpcond = ([fs] == [ft])`
111100 (60)	`c.lt.s fs, ft` / `c.lt.d fs, ft`	сравнение с соотношением «меньше» с плавающей точкой (FP less than comparison)	`fpcond = ([fs] < [ft])`
111110 (62)	`c.le.s fs, ft` / `c.le.d fs, ft`	сравнение с соотношением «меньше или равно» с плавающей точкой (FP less than or equal comparison)	`fpcond = ([fs] ≤ [ft])`

Программирование на языке Си

C



С.1 ВВЕДЕНИЕ

Цель этой книги – показать работу компьютера на различных уровнях, начиная с транзисторных схем и заканчивая выполнением программ. Первые пять глав книги описывают нижние уровни абстракции от транзисторов к логическим вентилям и далее к логическим схемам. В шестой, седьмой и восьмой главах сначала рассматривается архитектура, чтобы потом опуститься на уровень микроархитектуры и перекинуть мостик от аппаратного к программному обеспечению. Приложение С логически располагается между **главами 5** и **6**. Оно представляет собой краткое введение в язык программирования Си, являющийся наивысшим уровнем абстракции в этой книге. Такой порядок изложения связывает описания аппаратной архитектуры с практикой программирования, где у читателя возможно уже имеется личный опыт. Данный материал вынесен в отдельное приложение, чтобы читатель имел возможность либо изучить материал, либо пропустить его, в зависимости от степени знакомства с предметом.

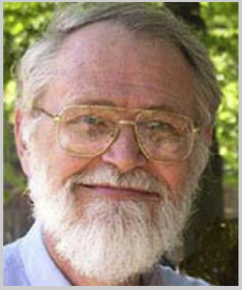
Чтобы описать действия, которые должен выполнить компьютер, используются различные языки программирования. По существу, компьютер выполняет команды на машинном языке, которые состоят из единиц и нулей, как было описано в **главе 6**. Но создание программ напрямую в машинных кодах – это утомительное и медленное занятие, что побуждает использовать языки более высокого уровня для

повышения продуктивности программистов. В **Табл. С.1** перечислены некоторые языки программирования различных уровней абстракции.

Один из самых популярных языков программирования – это **язык Си**, который был создан в период между 1969 и 1973 годами группой сотрудников Bell Laboratories, включавшей Денниса Ритчи и Брайана Кернигана. Целью группы было создания языка высокого уровня, позволяющего переписать код операционной системы UNIX, первоначально написанной на языке ассемблера. По многочисленным оценкам, язык Си (ставший базовым для семейства таких языков, как C++, C# и Objective-C) является самым популярным языком программирования из ныне существующих. Его широкое признание вызвано удачным сочетанием ряда свойств языка. Вот некоторые из них:

- Доступность на большом количестве платформ от микроконтроллеров до суперкомпьютеров
- Относительная простота и большое количество пользователей

Деннис Ритчи, 1941–2011



Брайан Керниган, 1942–

Первым официальным описанием языка Си стала классическая книга Брайана Кернигана и Денниса Ричи “Язык программирования Си” (The C Programming Language), которая была опубликована в 1978 году. В 1989 году Американский национальный институт стандартов (ANSI) расширил и стандартизировал спецификации языка Си. Эта версия языка стала известна под названиями ANSI C, Standard C или C89. Вскоре после этого, в 1990 году, стандарт ANSI C был принят Международной организацией по стан-дартизации (ISO) и Международной электротех-нической комиссией (IEC). В 1999 году организа-ции ISO/IEC пересмотрели стандарт и опубли-ковали обновлённую версию спецификаций языка, получившую название C99, которую мы и будем рассматривать в этой книге.

Табл. С.1 Языки программирования в порядке убывания уровня абстракции

Язык программирования	Описание
Matlab	Богатые возможности для записи математических операций
Perl	Небольшие программы для обработки текстов
Python	Внимание к повышению читаемости текста программы
Java	Безопасное выполнение на различных платформах
C	Гибкий доступ к широкому спектру системных ресурсов, включая драйверы устройств
Ассемблер	Представление машинного кода в удобной для чтения форме
Машинные коды	Двоичное представление программы

- Средний уровень абстракции языка, что позволяет писать программы продуктивнее по сравнению с использованием языка ассемблера, а с другой стороны даёт программисту представление, как будет выполняться код программы
- Пригодность для создания высокопроизводительных программ
- Возможность напрямую работать аппаратным обеспечением

Эта глава посвящена языку Си по многим причинам, самой важной из которых является возможность напрямую обращаться к оперативной памяти. Это свойство языка хорошо иллюстрирует связь между аппаратным и программным обеспечением, на которую мы обращаем внимание в этой книге. Все инженеры и учёные, работающие в области разработки аппаратного и программного обеспечения, должны знать язык программирования Си. Язык Си используется в многих областях компьютерных технологий, таких как разработка программного обеспечения, программирование встроенных систем, моделирование устройств. Умение программировать на Си – это важный и востребованный навык.

Следующие разделы содержат полное описание синтаксиса языка Си, а также составляющих элементов программы, включая заголовочные файлы, описания функций и переменных, типы данных и часто используемые библиотечные функции. **Раздел 8.6** описывает практическое применение языка Си для программирования микроконтроллера PIC32.

КРАТКИЙ ИТОГ

- **Высокоуровневое программирование**: Использование высокоуровневых языковых конструкций удобно при создании широкого круга приложений от программного обеспечения для анализа и моделирования до программ для микроконтроллеров.
- **Низкоуровневый доступ**: Преимущество языка Си состоит в том, что помимо высокоуровневых конструкций, он предоставляет непосредственный доступ к аппаратуре и памяти.

С.2 ДОБРО ПОЖАЛОВАТЬ В ЯЗЫК СИ

Программа на языке Си состоит из одного или нескольких текстовых файлов, в которых описываются действия, которые должен выполнить компьютер. Перед выполнением, текст программы необходимо перевести в машинное представление, понятное для компьютера. Процесс перевода называется компиляцией. Простая программа, которая выводит фразу "Hello world!" на дисплей, приведена в **примере С.1**. Имена файлов, содержащих текст на языке Си, принято заканчивать суффиксом ".c". Хороший стиль программирования подразумевает, что имена файлов будут отражать их содержание. Скажем, файл из **примера С.1** можно назвать `hello.c`.

Пример С.1 ПРОСТАЯ ПРОГРАММА НА С

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
  printf("Hello world!\n");}
```

Вывод

```
Hello world!
```

Язык Си был использован для создания широко распространённых операционных систем – Linux, Windows, OS X, iOS и Android, поскольку язык предоставляет свободный доступ к аппаратным ресурсам компьютера. Сравнивая его с другими языками программирования, как Python или Matlab, необходимо отметить, что в языке Си отсутствует встроенная реализация таких развитых языковых средств, как высокоуровневые функции для работы с файлами, сравнение по шаблону, матричные операции и взаимодействие с графическим интерфейсом пользователя. Также в нём отсутствуют средства, позволяющие выявить распространённые ошибки при обращении к памяти, подобные выходу за границы массива. Мощь языка Си в сочетании с отсутствием контроля за ошибками помогает хакерам проникать в компьютерные системы, где используется программное обеспечение, разработанное без должного внимания к безопасности.

С.2.1 Структура программы на языке СИ

Обычно, программа на языке Си содержит одну или несколько функций. Каждая программа должна иметь функцию с именем main, которая служит стартовой точкой для выполнения программы. Кроме этой главной функции, большинство программ на языке Си содержат набор функций, которые находятся в тексте программы и/или в библиотеках. Наш пример `hello.c` состоит из директив включения в программу заголовочных файлов, объявления функции `main` и её реализации.

> Данное приложение даёт базовое представление о языке Си. Кроме того, существует огромное количество литературы, описывающей язык Си более глубоко и детально. Одним из лучших учебников по языку является классическая книга “Язык программирования Си” (The C Programming Language), написанная создателем языка Деннисом Ритчи в соавторстве с Брайаном Керниганом, где в концентрированной форме изложены все основные возможности языка Си. Ещё одни хороший учебник – это книга Al Kelley и Ira Pohl “A Book on C”.

Заголовочный файл: `#include <stdio.h>`

Заголовочный файл состоит из объявлений функций, требующихся программе. В нашем случае, программа использует функцию printf, которая находится в стандартной библиотеке ввода/вывода и объявлена в заголовочном файле stdio.h. **Раздел С.9** содержит более подробное описание стандартной библиотеки языка Си.

Главная функция: `int main(void)`

Работа программы начинается с выполнения операторов, содержащихся в функции с именем main. Набор этих операторов называется телом функции main. Любая программа на Си должна иметь одну и только одну функцию main. Синтаксис функций описан в **разделе С.6** Вызовы функций. Тело функции содержит последовательность операторов, каждая из которых должна заканчиваться точкой с запятой. Ключевое слово int указывает, что результат функции (или возвращаемое значение) имеет целочисленный тип. Возвращаемое значение показывает, была ли программа выполнена успешно. По завершению программы, результат main передаётся в то окружение, из которого она была запущена.

Тело функции: `printf("Hello world!\n");`

Тело функции `main` из нашего примера содержит единственный вызов функции `printf`, печатающую строку “Hello world!”, которая заканчивается символом перевода строки “`\n`”. Более детальное описание функций ввода/вывода содержится в **разделе С.9.1**.

В общих чертах, все программы устроены подобно нашему простому примеру `hello.c`, с той разницей, что сложные программы могут состоять из миллионов строк текста, разбитых на множество функций и файлов.

С.2.2 Запуск Си-программы

Программы на языке Си могут выполняться процессорами с различными системами команд. Переносимость программ – это ещё одно преимущество языка программирования Си. Перед запуском программа должна быть скомпилирована Си-компилятором. Существует несколько отличающихся реализаций Си-компиляторов, включая cc (C компилятор) и gcc (GNU C компилятор). В данной книге мы используем компилятор gcc для демонстрации компиляции и запуска программ. Этот компилятор доступен для свободного использования. Он поставляется в составе всех дистрибутивов операционной системы Linux и не требует дополнительных действий по инсталляции. В операционной системе Windows может понадобиться установка пакета программ Cygwin. Компилятор gcc имеется также для большого количества встраиваемых систем, например, для микроконтроллеров Microchip PIC32. Процесс подготовки файла с программой, его компиляция и выполнение кода программы описаны ниже. Эти действия одинаковы для всех программ на языке Си.

1. Создайте текстовый файл, например `hello.c`.
2. В командной оболочке перейдите в директорию, содержащую файл hello.c, затем наберите команду gcc hello.c и запустите её на выполнение

3. Компилятор создаст файл с исполняемым кодом. По умолчанию, этот файл получит имя `a.out` (или `a.exe` в системе Windows).
4. Наберите в командной оболочке `./a.out` (или `a.exe` в системе Windows) и нажмите Enter.
5. На экране должна появиться строка “Hello world!”.

Краткий итог

- `filename.c`: Файл с программой на языке Си обычно имеет суффикс “.c”.
- `main`: Каждая программа должна содержать только одну функцию `main`.
- `#include`: Большинство программ на Си используют библиотечные функции. Для работы с ними необходимо включить директиву `#include <библиотека.h>` в начало файла с программой.
- `gcc filename.c`: Файлы с текстом программ преобразуются в исполняемый код при помощи компиляторов, таких как GNU C (gcc) или C (cc) компиляторы.

- **Выполнение программы:** После компиляции, программа запускается вводом команды ./a.out (или a.exe) в командной оболочке.

С.3 КОМПИЛЯЦИЯ

Компилятор – это программа, которая переводит текст, написанный на языке высокого уровня в его низкоуровневое представление: язык ассемблера или машинный код. Иначе говоря, компилятор читает файл с программой на языке Си и преобразует его в файл с исполняемым кодом. Существует огромное количество литературы по компиляторам, поэтому здесь мы ограничимся только кратким введением. Работу компилятора можно разбить на несколько шагов: (1) препроцессирование, в ходе которого в программу включаются объявления из библиотек и выполняется подстановка макросов; (2) удаление всей неиспользуемой для генерации кода информации, как например, комментарии; (3) перевод операторов языка высокого уровня в соответствующие им наборы машинных команд; (4) сборка всех файлов с машинными командами, а также библиотечными функциями, в единый исполняемый файл. Каждый процессор использует свой набор команд, поэтому программу необходимо

скомпилировать именно для того процессора, на котором она будет выполняться. Набор команд для процессора MIPS описан в **главе 6**.

С.3.1 Комментарии

Программисты используют комментарии для пояснения действий в тексте программы и назначения функций. Всякий, кто видел программы без комментариев, может подтвердить их необходимость. В языке С существуют два вида комментариев: однострочные комментарии начинаются с двух символов `//` и продолжаются до конца строки; многострочные комментарии могут занимать произвольное количество строк. Они начинаются с комбинации символов `/*` и заканчиваются `*/`. Комментарии крайне полезны для структурирования и пояснения текста программы, но во время компиляции они пропускаются и не попадают в исполняемый код.

```
// Это пример однострочного комментария.
/* Это Пример
   многострочного комментария. */
```

В начало файла с программой будет полезно вставить комментарий с указанием автора, даты создания, даты последней модификации и назначения программы. Комментарий, приведенный ниже, можно поместить в заголовок файла `hello.c`.

```
// hello.c
// 1 июня 2012 Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// Эта программа печатает "Hello world!" на экране
```

С.3.2 `#define`

Директива `#define` используется для объявления именованных констант. После объявления именованной константы, мы можем использовать в программе её имя как синоним значения константы. Такие глобально определенные константы также называются *макросами*. Предположим, вы пишете программу, позволяющую пользователю сделать не более 5 попыток дать правильный ответ. В таком случае вы можете использовать директиву `#define` для определения количества попыток в виде макроса.

```
#define MAXGUESSES 5
```

Символ `#` обозначает, что эта строка программы должна быть обработана *препроцессором*. Перед компиляцией, препроцессор заменяет все вхождения имени MAXGUESSES на значение 5, указанное в директиве `#define`. Обычно в качестве имён макросов принято использовать прописные буквы и размещать директивы `#define` в начале файла. Объявление именованной константы облегчает дальнейшее сопровождение программы, поскольку можно

быть уверенным, что в тексте программы везде используется одно и то же значение. Кроме того, такой подход позволяет быстро изменить все вхождения значения константы в программе. Для этого вам нужно всего лишь исправить одну строчку с директивой `#define`, вместо поиска и замены значения константы по всей программе.

Числовые константы в языке Си по умолчанию считаются десятичными числами. Вы также можете использовать шестнадцатеричные и восьмеричные числа, добавляя перед ними префиксы "0x" и “0”. Стандарт языка C99 не определяет запись чисел в двоичной системе исчисления, но они поддерживаются в некоторых компиляторах с использованием префикса "0b". Например, в этих трёх строках выполняется присваивание переменной x одного и того же значения:

```
char x = 37;
char x = 0x25;
char x = 045.
```

В **примере С.2** показано, как использовать директиву #define для пересчёта дюймов в сантиметры. Переменные inch и cm, предназначенные для хранения чисел с плавающей точкой, объявлены имеющими тип float. Использование именованной константы, объявленной директивой #define INCH2CM, может помочь избежать ошибок, вызванных опечатками (например, использование 2,53 вместо 2,54), а также упрощает поиск и замену (например, если нам

требуется увеличить точность множителя, используемого для преобразования), особенно если речь идёт о большой программе.

Пример С.2 ИСПОЛЬЗОВАНИЕ #DEFINE ДЛЯ ОБЪЯВЛЕНИЯ КОНСТАНТ

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54
int main(void) {
  float inch = 5.5;    // 5.5 inches
  float cm;

  cm = inch *INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Вывод

```
5.500000 inches = 13.970000 cm
```

Неименованную константу, используемую в программе, называют “*магическим числом*”. Наличие подобной "магии" приводит к трудноуловимым ошибкам. Например, вы можете изменить константу в одном месте программы, но позабыть внести исправление в другом месте. Глобально определенные именованные константы устраняют “магические числа” из текста программ.

С.3.3 `#include`

Модульность подразумевает разделение программы на отдельные файлы и функции. Общеупотребительные функции могут быть сгруппированы для упрощения их повторного использования в других программах. Объявления констант, переменных и функций, которые собраны в одном *заголовочном файле*, можно включить внутрь другого файла с помощью директивы препроцессора `#include`. Доступ к функциям *стандартной библиотеки* осуществляется именно таким образом. Например, для использования функций ввода/вывода таких, как `printf`, в программу необходимо включить строку:

```
#include <stdio.h>
```

Для имени заголовочного файла обычно используется стандартное расширение “`.h`”. Директивы `#include` принято размещать в самом начале файла. Хотя, в общем случае, они могут находиться в любой точке программы, которая предшествует использованию функций, переменных и констант, объявленных в заголовочном файле.

Заголовочный файл, который является частью программы, может быть включен в текст директивой препроцессора #include, где имя файла указано в двойных кавычках (`" "`) вместо угловых скобок (`< >`).

Например, заголовочный файл myfunctions.h, созданный разработчиком программы, включается в текст директивой:

```
#include "myfunctions.h"
```

Файлы, указанные в угловых скобках, во время компиляции ищутся в каталогах системных заголовочных файлов. Файлы, указанные в двойных кавычках, ищутся в том же каталоге, где расположен файл, содержащий директиву `#include`. Если заголовочный файл расположен в другом каталоге, необходимо использовать относительный путь до него.

Краткий итог

- **Комментарии:** В языке Си используются однострочные (`//`) и многострочные (`/* */`) комментарии.
- `#define NAME val`: Директива `#define` позволяет использовать идентификатор (NAME) в коде программы. Перед компиляцией идентификатор макроса заменяется на его значение (`val`), указанное в объявлении макроса
- `#include`: Директива `#include` позволяет включать в программу объявления библиотечных или общеупотребительных функций, переменных, констант. Для использования объявлений из системных библиотек, поместите следующую строку в начало

программы: `#include <library.h>`. Для использования заголовочного файла, созданного программистом, используйте двойные кавычки и путь относительно текущего каталога: `#include "other/myFuncs.h"`.

С.4 ПЕРЕМЕННЫЕ

Переменные в языке Си обладают именем, типом, значением и адресом в памяти. Объявление переменной определяет её имя и тип. Например, следующее объявление определяет переменную, имеющую тип `char` (размером один байт), и даёт этой переменной имя `x`. Адрес, по которому эта однобайтовая переменная будет располагаться в памяти, назначается автоматически при компиляции программы.

```
char x;
```

Память рассматривается в языке Си, как группа последовательных байтов, в которой каждому байту присвоен уникальный номер, называемый *адресом* (смотри **Рис. С.1**). Переменная занимает один или несколько байт памяти. Адресом переменной считается адрес первого занимаемого ею байта. Интерпретация набора байт, отведённых под переменную, зависит от типа переменной. Это может быть целое число, число с плавающей точкой и другие типы данных.

Далее мы рассмотрим базовые типы языка С, объявления глобальных и локальных переменных и их инициализацию.

Имена переменных чувствительны к регистру символов и могут выбираться по усмотрению автора программы. Тем не менее, существуют некоторые ограничения. Имя переменной не может совпадать с зарезервированными словами, перечисленными в стандарте языка Си (`int`, `while` и т.д.). Имя переменной не может начинаться с цифры (например, объявление “`int 1x;`” некорректно), или включать в себя ряд специальных символов, таких как `\`, `*`, `?` или `-`, хотя символ подчеркивания (_) использовать можно.

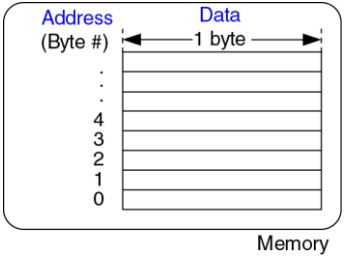


Рис. С.1 Как выглядит память для программы на Си

С.4.1 Базовые типы данных

В языке Си существует набор базовых (или встроенных) типов данных. В первом приближении, их можно разделить на целые числа, числа с плавающей точкой и символы. Для представления целых чисел со знаком используется дополнительный код. Целочисленные типы со знаком и без знака ограничены диапазоном значений. Числа с плавающей точкой также имеют ограниченный диапазон значений и точность. Для их представления используется формат, определенный стандартом IEEE 754. Символы могут рассматриваться как коды ASCII или как 8-битные целые числа. В **Табл. С.2** перечислены размеры и диапазоны значений всех базовых типов данных. Целые числа могут занимать 16, 32 или 64 бита в памяти. Для их представления используется дополнительный код, если только переменная не была объявлена с использованием ключевого слова `unsigned`. Размер типа `int` зависит от конкретной платформы и обычно соответствует размеру слова в используемой машине. Например, для 32-битного процессора MIPS типы `int` или `unsigned int` имеют размер 32 бита. Числа с плавающей точкой могут занимать 32 или 64 бита, что соответствует представлению с одинарной или двойной точностью. Символы имеют размер 8 бит.

Пример С.3 демонстрирует объявления переменных различных типов. Размер памяти, требующийся для хранения переменных, показан на **Рис. С.2**. Переменная `x` занимает один байт, переменная `y` занимает два байта, а для переменной `z` требуется четыре байта. Переменные одного типа всегда имеют одинаковый размер, но располагаются в разных участках памяти. В нашем примере переменные `x`, `y` и `z` расположены по адресам 1, 2 и 4 соответственно. Имена переменных чувствительны к регистру символов. Например, имена `x` и `X` обозначают две разные переменные (хотя использование настолько похожих имён переменных в одной программе может привести к путанице).

Пример С.3 ПРИМЕРЫ ТИПОВ ДАННЫХ

```
// Examples of several data types and their binary representations
unsigned char x = 42;       // x = 00101010
short y = -10;              // y = 11111111 11110110
unsigned long z = 0;        // z = 00000000 00000000 00000000 00000000
```

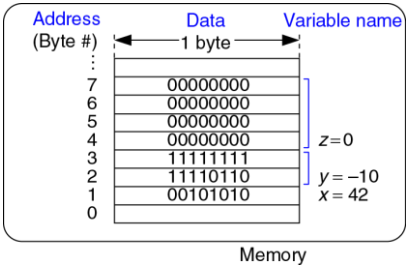


Рис. С.2 Расположение переменных в памяти для примера С.3

В том, что размер типа int не стандартизирован, есть свои плюсы и минусы. С одной стороны, соответствие размеру слова позволяет более эффективно обрабатывать такие данные, но другой стороны, программы, использующие тип int, могут работать по-разному на разных компьютерах. Для примера, рассмотрим банковскую программу, сохраняющую в переменной типа int количество центов. Если скомпилировать эту программу для 64-битного процессора, диапазон допустимых значений удовлетворит даже самого богатого бизнесмена. Перенеся эту программу на 16-битную платформу, мы получим переполнение, если сумма на счете превысит 327,67 долларов. В результате – несчастные и сильно бедствующие клиенты.

Табл. С.2 Базовые типы данных и их размеры

Тип	Размер (бит)	Минимум	Максимум
`char`	8	$-2^{-7} = -128$	$2^{7} - 1 = 127$
`unsigned char`	8	0	$2^{8} - 1 = 255$
`short`	16	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
`unsigned short`	16	0	$2^{16} - 1 = 65,535$
`long`	32	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,647$
`unsigned long`	32	0	$2^{32} - 1 = 4,294,967,295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	аппаратно-зависимый		
`unsigned int`	аппаратно-зависимый		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

С.4.2 Глобальные и локальные переменные

Глобальные и локальные переменные отличаются местом объявления в программе и областью видимости. Глобальные переменные объявляются вне функций и, как правило, в начале программы. К таким переменным могут обращаться любые функции. Глобальные переменные не рекомендуется использовать слишком часто, так как они нарушают принципы модульности и усложняют понимание программы. Однако, если переменная необходима сразу несколькими функциями, она может быть сделана глобальной.

> *Область видимости* переменной, это область программы, в пределах которой можно использовать данную переменную. Например, для локальной переменной её областью видимости является функция, внутри которой объявлена переменная. Локальная переменная будет недоступна вне этой функции.

Локальная переменная, объявленная в функции, может быть использована только внутри тела этой функции. По этой причине, две разных функции могут содержать объявления переменных с одинаковыми именами. Эти переменные не оказывают никакого влияния друг на друга. Они создаются при каждом новом вызове функции и уничтожаются при её завершении. Данные, хранящиеся в таких переменных, не сохраняются между вызовами функции.

Пример С.4 и **пример С.5** позволяют сравнить эти два вида переменных. В первом примере используются глобальные переменные, а во втором – локальные. Глобальная переменная max из **примера С.4** доступна в любой функции. Использование локальных переменных показано в **примере С.5**. Такой подход более предпочтителен, так как он позволяет разделять программу на несколько независимых модулей, взаимодействующих друг с другом через их внешние интерфейсы.

Пример С.4 ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
// Use a global variable to find and print the maximum of 3 numbers
int max;               // global variable holding the maximum value
void findMax(int a, int b, int c) {
  max = a;
    if (b >max) {
      if (c > b) max = c;
      else       max = b;
    } else if (c > max) max = c;
}

void printMax(void) {
  printf("The maximum number is: %d\n", max);
}

int main(void) {
  findMax(4, 3, 7);ы
  printMax();
}
```

Пример С.5 ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
  int result = a; // local variable holding the maximum value

  if (b > result) {
    if (c > b) result = c;
    else       result = b;
  } else if (c > result) result = c;

  return result;
}

void printMax(int m) {
  printf("The maximum number is: %d\n", m);
}

int main(void) {
  int max;

  max = getMax(4, 3, 7);
  printMax(max);
}
```

С.4.3 Инициализация переменных

Перед чтением значения переменной, она должна быть *проинициализирована*, то есть ей должно быть присвоено некоторое начальное значение. В момент объявления переменной ей выделяется блок памяти, но содержимое этого блока пока не определено. Например, там может находиться значение предыдущей переменной, занимавшей ранее тот же блок памяти, или некое произвольное значение. Глобальные и локальные переменные могут быть проинициализированы либо при их объявлении в программе, либо среди операторов одной из функции. В **примере С.3** продемонстрирована инициализация переменных непосредственно при их объявлении. В **примере С.4** показано, как переменные инициализируются после их объявления, но до первого обращения к ним. Глобальная переменная max инициализируется в функции `getMax` до того, как к ней произойдёт первый доступ в функции `printMax`. Чтение значения неинициализированных переменных – это распространенная ошибка, которую бывает довольно трудно обнаружить.

КРАТКИЙ ИТОГ

- **Типы данных:** Тип переменной определяет размер (количество байт, занимаемых переменной) и представление переменной (интерпретацию содержимого памяти). Типы данных, используемые в языке Си, перечислены в **Табл. С.2**.
- **Память:** В языке Си память рассматривается, как упорядоченный набор байт. Значения переменных хранятся в памяти и каждая переменная имеет свой собственный адрес (номер первого байта).
- **Локальные переменные:** Локальные переменные объявляются внутри тела функции и доступны только внутри этой функции.
- **Инициализация переменных:** Переменной должно быть присвоено начальное значение до первого обращения к ней. Инициализация переменной может быть выполнена непосредственно при объявлении переменной, либо операторами программы.

С.5 ОПЕРАЦИИ

Наиболее распространенный вид конструкций языка С – это *выражения*. Например:

```
y = a + 3;
```

Выражение состоит из *операций* (таких как `+` или `*`), выполняющих действия над одним или несколькими *операндами* (переменные или константы). Операции, которые имеются в языке Си, перечислены в **Табл. С.3**. Они сгруппированы по категориям и перечислены в порядке убывания приоритета операций. К примеру, мультипликативные операции более приоритетны и выполняются до аддитивных. Операции одного уровня приоритета выполняются слева направо в том порядке, как они расположены в выражении.

Унарные операции имеют только один операнд. Тернарные операции имеют три операнда. А все остальные операции выполняются над двумя операндами и называются бинарными. Тернарная операция (от латинского слова *ternarius*, означающего «состоящий из трех») возвращает второй или третий операнд. Если первый операнд принимает значения TRUE (ненулевое), то результатом выражения будет второй операнд, а иначе, если первый операнд – FALSE (нулевое), то результатом станет третий операнд выражения. **Пример С.6** демонстрирует два способа вычисления максимума из двух значений.

Табл. С.3 Операции в порядке убывания приоритета

Категория	Операция	Описание	Пример
Унарные	`++`	пост-инкремент	`a++; // a = a+1`
	`--`	пост-декремент	`x--; // x = x-1`
	`&`	получение адреса	`x = &y; // x = the memory // address of y`
	`~`	побитовое НЕТ	`z = ~a;`
	`!`	логическое НЕТ	`!x`
	`-`	унарный минус	`y = -a;`
	`++`	инкремент	`++a; // a = a+1`
	`--`	декремент	`--x; // x = x-1`
	`(type)`	приведение типа	`x = (int)c; // cast c to an // int and assign it to x`
	`sizeof()`	размер переменной или типа	`long int y; x = sizeof(y); // x = 4`
Мульти-пликативные	`*`	умножение	`y = x *12;`
	`/`	деление	`z = 9 / 3; // z = 3`
	`%`	модуль (остаток)	`z = 5 % 2; // z = 1`
Аддитивные	`+`	сложение	`y = a + 2;`
	`-`	вычитание	`y = a - 2;`

Категория	Операция	Описание	Пример
Побитовый сдвиг	<<	побитовый сдвиг влево	`z = 5 << 2; // z = 0b00010100`
	>>	побитовый сдвиг вправо	`x = 9 >> 3; // x = 0b00000001`
Relational	==	равенство	`y == 2`
	!=	неравенство	`x != 7`
	<	меньше	`y <12`
	>	больше	`val >max`
	<=	меньше или равно	`z <= 2`
	>=	больше или равно	`y >= 10`
Побитовые	&	побитовое И	`y = a & 15;`
	^	побитовое исключающее ИЛИ	`y = 2 ^ 3;`
	|	побитовое ИЛИ	`y = a |b;`
Логические	&&	логическое И	`x && y`
	||	логическое ИЛИ	`x || y`

Тернарные	? :	условный оператор	`y = x ? a : b;`
			`// if x is TRUE,`
			`// y=a, else y=b`

Присваивание	=	присваивание	`x = 22;`
	+=	присваиванием с суммированием	`y += 3;  // y = y + 3`

Категория	Операция	Описание	Пример
	`-=`	присваиванием с вычитанием	`z -= 10;  // z = z - 10`
	`*=`	присваиванием с умножением	`x *= 4;  // x = x * 4`
	`/=`	присваиванием с делением	`y /= 10;  // y = y / 10`
	`%=`	присваиванием по модулю	`x %= 4;  // x = x % 4`
	`>>=`	присваиванием с побитовым сдвигом вправо	`x >>= 5;  // x = x>>5`
	`<<=`	присваиванием с побитовым сдвигом влево	`x <<= 2;  // x = x<<2`
	`&=`	присваиванием с побитовым И	`y &= 15;  // y = y & 15`
	`|=`	присваиванием с побитовым ИЛИ	`x |= y;  // x = x | y`
	`^=`	присваиванием с побитовым исключающим ИЛИ	`x ^= y;  // x = x ^ y`

Пример С.6 ТЕРНАРНАЯ ОПЕРАЦИЯ И ЭКВИВАЛЕНТНЫЕ ОПЕРАТОРЫ `IF/ELSE`

```
(a) y = (a >b) ? a : b; // parentheses not necessary,
                        // but makes it clearer

(b) if (a >b)y = a;
      else     y = b;
```

В первом примере используется тернарная операция. Второй способ немного длиннее и в нём использованы условные операторы `if/else`.

В составном операторе присваивании сначала выполняется арифметическая операция над операндами, которыми являются левая и правая части оператора, а затем результат вычисления записывается в переменную из левой части составного оператора. **Пример С.7** показывает различные операции языка Си. Двоичные значения обозначены префиксом «`0b`» в комментариях.

Пример С.7 ПРИМЕРЫ ОПЕРАЦИЙ ЯЗЫКА С

Выражение	Результат	Примечание
`44 / 14`	`3`	Результат округлен
`44 % 14`	`2`	Деление по модулю 14
`0x2C && 0xE //0b101100 && 0b1110`	`1`	Логическое И
`0x2C || 0xE //0b101100 || 0b1110`	`1`	Логическое ИЛИ
`0x2C & 0xE //0b101100 & 0b1110`	`0xC (0b001100)`	Побитовое И
`0x2C |0xE //0b101100 | 0b1110`	`0x2E (0b101110)`	Побитовое ИЛИ
`0x2C ^ 0xE //0b101100 ^ 0b1110`	`0x22 (0b100010)`	Побитовое исключающее ИЛИ
`0xE << 2 //0b1110 << 2`	`0x38 (0b111000)`	Сдвиг влево на 2
`0x2C >> 3 //0b101100 >> 3`	`0x5 (0b101)`	Сдвиг вправо на 3
`x = 14;` `x += 2;`	`x=16`	

`y = 0x2C; // y = 0b101100`
`y &= 0xF; // y &= 0b1111`
	`y=0xC (0b001100)`	

`x = 14; y = 44; y = y + x++;`	`x=15, y=58`	Увеличение переменной после использования
`x = 14; y = 44; y = y + ++x;`	`x=15, y=59`	Увеличение переменной до использования

Правда, только правда и ничего, кроме правды

В языке Си считается, что переменная принимает значение TRUE, если она не равна нулю, и FALSE в противном случае. Результат логических и тернарных операций, а также операторов управления последовательностью действий (`if`, `while` и другие), зависит от истинности или ложности значений переменных. Результат операций сравнения или логических операций представляется в виде числа 1 для логического значения “истина” (TRUE) или числа 0 для значения “ложь” (FALSE).

С.6 ВЫЗОВЫ ФУНКЦИЙ

Модульность программы является признаком хорошего стиля. Большую программу обычно разделяют на небольшие фрагменты, называемые функциями, которые, по аналогии с аппаратными модулями, имеют хорошо спроектированные входы, выходы и назначение. В **примере С.8**демонстрируется функция `sum3`. Объявление функции начинается с указания типа возвращаемого ею значения (int). Затем следует имя функции sum3. В конце идет список параметров функции, заключенный в круглые скобки (`int a`, `int b`, `int c`). Тело функции заключается в фигурные скобки `{}`. Оно может состоять из последовательности операторов или не иметь вообще ни одного

оператора внутри скобок. Оператор `return` определяет результат, который должна вернуть функция. Этот результат можно рассматривать как значение функции на выходе. Функция может вернуть только одно значение.

Пример С.8 ФУНКЦИЯ SUM3

```
// Return the sum of the three input variables

int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

После вызова `sum3` в следующем примере, значение переменной y будет равно 42.

```
int y = sum3(10, 15, 17);
```

Наличие входных данных и результата функции не является обязательным. **Пример С.9** показывает функцию, у которой нет ни параметров, ни результата на выходе. Ключевое слово `void` перед именем функции означает, что она не возвращает результат. То же ключевое слово `void` между круглыми скобками после имени функции говорит о том, что функция не имеет параметров.

Пример С.9 ФУНКЦИЯ `PRINTPROMPT` БЕЗ АРГУМЕНТОВ И РЕЗУЛЬТАТА

```
// Print a prompt to the console
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

Если между круглыми скобками ничего нет, то это также означает отсутствие у функции входных данных. Таким образом, функция без параметров и результата может быть объявлена следующим образом:

```
void printPrompt()
```

Функция должна быть объявлена текстуально выше того места в программе, где она вызывается. Соблюдения этого правила можно добиться двумя способами. Во-первых, мы можем поместить полное определение функции до того места, где она будет вызвана. Такой способ часто используется, когда функция main размещается в конце файла, после объявления всех других функций. Во-вторых, мы можем объявить только *прототип* функции до точки её вызова и последующего полного определения функции. Прототипом называется первая строчка объявления функции, содержащая тип возвращаемого значения, имя функции и её параметры. Ниже приведены образцы прототипов функций из **примера С.8** и **примера С.9**:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

> Использование прототипов может не потребоваться при аккуратном выборе порядка расположения функций в тексте программы. Тем не менее, при наличии циклических зависимостей между функциями, прототипов избежать невозможно. Например, в случае, когда функция `f1` вызывает функцию `f2`, которая в свою очередь вызывает функцию `f1`. Размещение прототипов всех функций в начале файла или в отдельном заголовочном файле считается хорошим стилем программирования.

Пример С.10 демонстрирует использование прототипов функций. Несмотря на то, что определения функций размещаются после функции `main`, прототипы функций в начале программы позволяют вызывать функции из главной функции `main`.

Пример С.10 ПРОТОТИПЫ ФУНКЦИЙ

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
```

```
{
  int y = sum3(10, 15, 20);
  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c)
{  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Вывод

```
sum3 result: 45

Please enter a number from 1-3:
```

Функция `main` всегда возвращает результат типа `int`. Применительно к главной функции, возвращаемое значение служит для сообщения операционной системе о результате работы всей программы. Ноль означает нормальное завершение программы и отсутствие ошибок. Индикатором ошибки служит ненулевое значение результата. Если функция `main` не имеет оператора `return`, она автоматически возвращает ноль в качестве результата. Большинство операционных

систем не выводят на экран это возвращаемое значение по завершению программы.

Имена функций точно также, как имена переменных, чувствительны к регистру символов, и также не могут быть служебными словами, зарезервированными в языке Си. Они не должны начинаться с цифры и включать специальные символы, за исключением символа подчеркивания (_). Обычно, в имя функции принято помещать глагол, который говорит о том, что эта функция делает.

Старайтесь придерживаться единого стиля в использовании прописных и строчных букв в именах, чтобы не приходилось постоянно вспоминать, как должно выглядеть имя функции или переменной. Существуют два наиболее распространенных стиля. При использовании первого стиля, называемого camelCase, все слова, составляющие имя функции, пишутся слитно, при этом каждое слово пишется с прописной буквы. Такое имя, например `printPrompt`, напоминает горбы верблюда. Второй стиль характерен использованием символа подчеркивания для разделения слов. К примеру, `print prompt`. К сожалению, мы обнаружили, что постоянные попытки дотянуться до символа подчеркивания на клавиатуре обостряют приступы туннельного синдрома (мой мизинец начинает болеть, когда я только думаю о подчеркивании). Поэтому мы предпочитаем camelCase. Но самое важное – это придерживаться единого стиля в пределах вашей организации.

С.7 УПРАВЛЕНИЕ ПОСЛЕДОВАТЕЛЬНОСТЬЮ ВЫПОЛНЕНИЯ ДЕЙСТВИЙ

Для управления последовательностью выполнения действий используются конструкции ветвления и циклы. Конструкции ветвления исполняют те или иные наборы операторов, в зависимости от некоторого условия. Циклы повторяют набор операторов до тех пор, пока выполняется заданное условие.

С.7.1 Условные операторы

В языке Си существует стандартный набор условных операторов `if`, `if/else` и оператор выбора `switch/case`.

Оператор `if`

Оператор `if` выполняет следующий за ним оператор, когда логическое выражение, заданное в скобках, принимает значение «истина» (ненулевое). Синтаксис оператора if:

```
if (expression)
statement
```

В **примере С.11** демонстрируется оператор `if`. Когда переменная `aintBroke` равна 1, переменной `dontFix` также присваивается значение 1. При необходимости выполнения нескольких операторов,

их группируют в один составной оператор, используя фигурные скобки, как показано в **примере С.12**.

> Фигурные скобки `{}` используются для объединения одного или нескольких операторов в *составной оператор* (или *блок кода*).

Пример С.11 ОПЕРАТОР IF

```
int dontFix = 0;

if (aintBroke == 1)
  dontFix = 1;
```

Пример С.12 ОПЕРАТОР `IF` С БЛОКОМ КОДА

```
// If amt >= $2, prompt user and dispense candy

if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

Оператор `if/else`

Конструкция `if/else` выполняет один из двух операторов, в зависимости от значения условия, как это показано ниже. Когда выражение `expression` в условии `if` принимает значение «истина»,

выполняется оператор `statement1`. В противном случае, выполняется оператор `statement2`.

```
if (expression)
  statement1
else
statement2
```

Использование конструкции `if/else` можно увидеть в **примере С.6 (b)**. Переменной `max` присваивается значение переменной `a`, если `a` больше, чем `b`. В противном случае, переменной max присваивается значение `b`.

Оператор `switch/case`

Конструкция switch/case выполняет операторы, в зависимости от значения в заголовке switch. Общая форма записи конструкции `switch`:

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
default:              statement4}
```

Например, если значение variable совпадает со значением выражения `expression2`, выполнение программы будет продолжено, начиная с оператора `statement2` и до первого встреченного оператора `break`,

который завершит выполнение конструкции `switch`. Если не удовлетворяется ни одно из условий, то будет выполнен оператор `statement4`, расположенный после ключевого слова `default`.

Если ключевое слово `break` после одного из операторов в теле конструкции `switch` пропущено, то после завершения данного оператора выполнение действий продолжится с переходом на следующую ветвь `case` и так далее. Возможно, это вовсе не то поведение, которое вы ожидаете. Остерегайтесь этой ошибки, распространенной среди начинающих программистов на языке Си.

Оператор `switch` используется в **примере С.13**, чтобы присвоить переменной `amt` сумму платежа, которая зависит от значения переменной `option`. Конструкция `switch` эквивалентна серии вложенных операторов `if/else`, как показано в **примере С.14**.

Пример С.13 ОПЕРАТОР `SWITCH/CASE`

```
// Assign amt depending on the value of option
switch (option) {
  case 1: amt = 100;break;
  case 2: amt = 50; break;
  case 3: amt = 20; break;
  case 4: amt = 10; break;
  default: printf("Error: unknown option.\n");
}
```

Пример С.14 ВЛОЖЕННЫЕ ОПЕРАТОРЫ IF/ELSE

```
// Assign amt depending on the value of option
if      (option == 1) amt = 100;
else if (option == 2) amt = 50;
else if (option == 3) amt = 20;
else if (option == 4) amt = 10;
else printf("Error: unknown option.\n");
```

С.7.2 Циклы

Язык Си имеет все основные виды операторов цикла, которые встречаются в языках программирования высокого уровня. Это операторы `while`, `do/while` и `for`.

Цикл `while`

Цикл `while` повторяет выполнение оператора, пока заданное условие принимает истинное значение.

```
while (condition)
statement
```

Цикл `while` из **примера С.15** вычисляет значение факториала 9 = 9 × 8 × 7 × … × 1. Обратите внимание, что условие цикла проверяется до выполнения оператора statement. В этом примере тело цикла состоит из нескольких операторов, поэтому они заключены в фигурные скобки.

Пример С.15 ЦИКЛ `WHILE`

```
// Compute 9!(the factorial of 9)
int i = 1, fact = 1;

// multiply the numbers from 1 to 9
while (i <10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

Цикл `do/while`

Оператор `do/while` похож на цикл `while`, но только с той разницей, что проверка условия `condition` происходит после выполнения тела цикла `statement`. Общая форма записи оператора `do/while` приведена ниже. Обратите внимание, что после условия condition необходима точка с запятой.

```
do
  statement
while (condition);
```

В **примере С.16** цикл `do/while` запрашивает у пользователя ввод числа до тех пор, пока не будет угадано верное значение. Условие станет истинным, когда введенное пользователем число совпадёт с заданным значением. Программа проверяет условие только после того,

как тело цикла `do/while` будет выполнено. Цикл `do/while` удобен, когда перед проверкой условия выхода из цикла должны быть выполнены некоторые действия (например, запрос данных у пользователя).

Пример С.16 ЦИКЛ `DO/WHILE`

```
// Query user to guess a number and check it against the correct number.

#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);  // read user input
  numGuesses++;
} while ( (numGuesses <MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

Цикл `for`

Цикл `for` подобен циклам while и `do/while` тем, что он повторяет оператор `statement` до тех пор, пока выполняется условие цикла. В цикле `for` принято использовать *переменную цикла*, которая служит

для подсчета количества выполненных итераций цикла. Общая форма записи цикла `for` выглядит так:

```
for (initialization; condition; loop operation)
  statement
```

Инструкция `initialization` выполняется один раз до начала работы цикла. Условие `condition` проверяется каждый раз перед началом нового цикла. Если условие не выполняется, то цикл завершается. Выражение `loop` operation выполняется в конце каждой итерации. **Пример С.17** демонстрирует применение цикла `for` для вычисления факториала числа 9.

Пример С.17 ЦИКЛ FOR

```
// Вычисление 9!

int i;  // переменная цикла
int fact = 1;

for (i=1; i<10; i++)
  fact *= i;
```

В **примере С.15** и **примере С.16** циклы `while` и `do/while` увеличивают значения переменных `i` и `numGuesses` на каждой итерации цикла. Аналогичные действия “встроены” в заголовок

цикла `for`. Цикл `for` может быть переписан с использованием оператора `while`, но очевидно, что такая форма записи цикла менее удобна.

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

КРАТКИЙ ИТОГ

- **Условные операторы:** Условные операторы выполняют один или несколько операторов, если заданное условие принимает значение TRUE. В языке Си имеются следующие условные операторы: `if`, `if/else`, and `switch/case`.
- **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides `while`, `do/while`, and `for` loops.
- **Циклы:** Операторы циклов повторяют выполнение операторов до тех пор, пока условие цикла не примет значение FALSE. Язык Си имеет такие средства для организации циклов, как операторы `while`, `do/while` и `for`

С.8 ДРУГИЕ ТИПЫ ДАННЫХ

Стандарт языка Си определяет целочисленные типы данных различного размера и числа с плавающей запятой. Кроме того, язык включает и другие специальные типы данных, такие как указатели, массивы, строки и структуры. Сейчас мы познакомимся с этими типами данных, а также рассмотрим динамическое выделение памяти.

С.8.1 Указатели

Указатель – это переменная, содержащая адрес (местонахождение в памяти) другой переменной. **Пример С.18** демонстрирует использование указателей. Переменные `salary1` и `salary2` предназначены для хранения целых чисел. Переменная `ptr` может хранить адрес целочисленной переменной. Компилятор назначает каждой переменной некоторый адрес в памяти. Длина адреса зависит от платформы, где программа будет выполняться. Для определенности предположим, что наша программа компилируется на 32-битной системе. Переменная `salary1` занимает участок памяти в диапазоне адресов 0x70-73, переменная `salary2` занимает участок с адресами 0x74-77, а переменная `ptr` находится в диапазоне 0x78-7B. На **Рис. С.3** показан блок памяти, выделенный для переменных, а также его содержимое после выполнения программы.

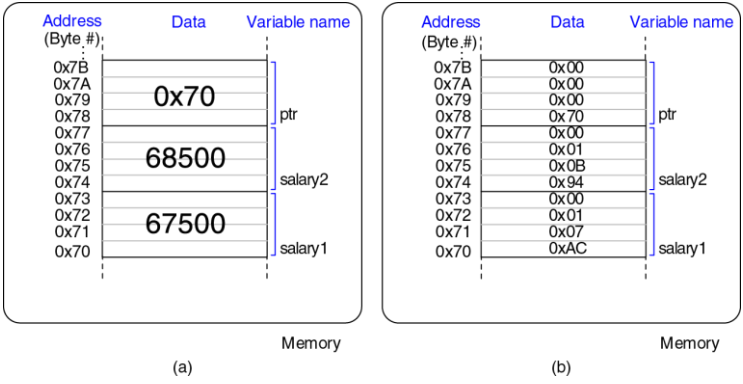


Рис. С.3 Содержимое памяти, после выполнения кода из примера С.18. а) значения переменных. b) содержимое ячеек памяти, при использовании порядка байтов от младшего к старшему.

При объявлении переменной, звездочка (*) перед именем переменной обозначает, что переменная является указателем. Если операция * применяется к переменной-указателю в выражении, то она *разыменовывает* указатель, возвращая значение, хранящееся в памяти по адресу, который хранится в переменной-указателе. Когда к

переменной применяется операция `&`, результатом операции будет *адрес* переменной в памяти.

> Если переменная-указатель содержит некорректный адрес или адрес, выходящий за диапазон адресов, доступных программе, то вероятнее всего разыменовывание такого указателя приведет к аварийному завершению программы. Этот тип ошибки часто называют *ошибкой сегментации*.

Пример С.18 УКАЗАТЕЛИ

```
// Example pointer manipulations

int salary1, salary2;  // 32-bit numbers
int *ptr;                 // a pointer specifying the address of an int
variable

salary1 = 67500;        // salary1 = $67,500 = 0x000107AC
ptr = &salary1;         // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000;  /* dereference ptr to give the contents of
address 70 = $67,500,
                           then add $1,000 and set salary2 to $68,500 */
```

Указатели особенно полезны, когда функции необходимо изменить значения переданных ей аргументов. В языке Си значения аргументов копируются внутрь функции (передача аргументов *по значению*) и поэтому функция не может изменить внешние аргументы напрямую.

Вместо этого, программист может передать ей указатели на переменные, значения которых нужно модифицировать. Такой способ передачи аргументов называется передачей *по ссылке*. В предыдущих примерах использовалась только передача аргументов *по значению*. В следующем **примере С.19** демонстрируется передача переменной `x` *по ссылке* в функцию `quadruple`, что позволяет функции изменить значение переменной x.

Пример С.19 ПЕРЕДАЧА АРГУМЕНТА ФУНКЦИИ ПО ССЫЛКЕ

```
// Quadruple the value pointed to by a#include <stdio.h>

void quadruple(int *a)

{
  *a = *a * 4;
}
int main(void)
{
  int x = 5;
  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Вывод

```
x before: 5
x after: 20
```

Указатель на адрес 0 называется *нулевым указателем* и используется для особого случая, когда переменная не указывает ни на какую другую переменную. Для обозначения такого указателя предназначен системный макрос NULL.

С.8.2 Массивы

Массивом называется группа однотипных переменных, последовательно расположенных в непрерывном участке памяти. Элементы массива нумеруются, начиная с 0. Если массив состоит из *N* элементов, то индексом последнего элемента будет *N*−1. В **примере С.20** объявляется массив `scores`, содержащий результаты экзаменов трех студентов. Для хранения массива выделяется участок памяти, размер которого равен размеру трех переменных типа `long`, что в сумме составляет 3 × 4 = 12 байт.

Пример С.20 ОБЪЯВЛЕНИЕ МАССИВА

```
long scores[3];  // array of three 4-byte numbers
```

Предположим, что массив scores располагается в памяти, начиная с адреса 0x40. Адрес первого элемента массива (обозначаемого `scores[0]`) будет 0x40, адрес второго элемента массива – 0x44, а третий элемент массива расположен по адресу 0x48. Распределение памяти показано на **Рис. С.4**. Переменная, обозначающая массив, в нашем случае `scores`, эквивалентна указателю на первый элемент массива. Язык Си не контролирует доступ к элементам массива, выходящим за его границы. Вся ответственность при работе с элементами массива лежит на разработчике программы.

Программа, модифицирующая данные за границами массива, будет успешно скомпилирована, но её запуск может привести к непредсказуемым изменениям значений других переменных программы.

Элементы массива можно проинициализировать в момент его объявления, используя фигурные скобки `{}`, как это показано в **примере С.21**. Данный способ инициализации доступен только в точке объявления массива. Альтернативный способ – индивидуальная инициализация элементов массива. Такой способ демонстрируется в **примере С.22**. Для доступа к элементам массива используются квадратные скобки `[]`. Содержимое памяти, выделенной для массива, показано на **Рис. С.4**. Цикл `for` является стандартным способом чтения или записи элементов массива, как показано в **примере С.23**.

Пример С.21 ИНИЦИАЛИЗАЦИЯ МАССИВА В МОМЕНТ ОБЪЯВЛЕНИЯ, ИСПОЛЬЗУЯ { }

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81;
scores[2]=97;
```

Пример С.22 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕМ ПРИСВАИВАНИЙ

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

Пример С.23 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕ ЦИКЛА `FOR`

```
// User enters 3 student scores into an array

long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

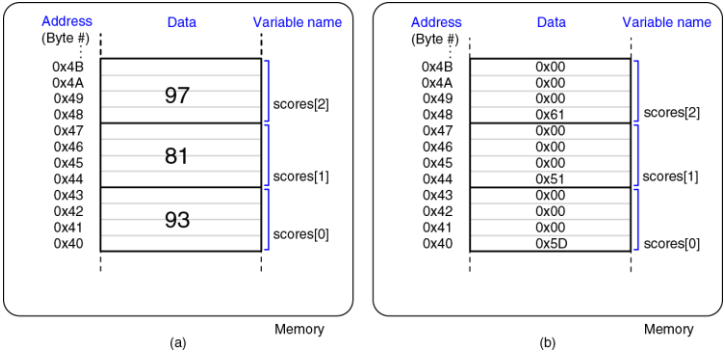


Рис. С.4 **Распределение памяти для массива scored**

Размер массива должен быть известен в момент его объявления. Это необходимо для того, чтобы компилятор мог назначить блок памяти фиксированного размера для хранения массива. В то же время, размер массива не играет роли при передаче массива аргументом функции. Функции необходимо знать только адрес начала массива. **Пример С.24** демонстрирует передачу массива в функцию. Параметр функции `arr` – это просто указатель на первый элемент массива.

Пример С.24 ПЕРЕДАЧА МАССИВОВ В ФУНКЦИЮ

```
// Initialize a 5-element array, compute the mean, and print the result.

#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;
  for (i=0; i <len; i++)
    total += arr[i];
  mean = total / len;
  return mean;
}
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;
  avg = getMean(data, 4);
  printf("The average value is: %f.\n", avg);
}
```

Вывод

```
The average value is: 54.500000.
```

Зачастую, количество элементов массива также передается в функцию дополнительным аргументом. Параметр типа `int[]` в объявлении функции указывает на то, что в функцию передается массив элементов

типа `int`. Язык Си допускает объявления функций, принимающих массивы любого типа.

При объявлении параметров функции, указатель на массив эквивалентен указателю на первый элемент массива. Таким образом, функция getMean может быть объявлена следующим образом:

```
float getMean(int *arr, int len);
```

Если в функцию передаётся массив, то предпочтительнее объявлять её параметр именно как массив. Хотя указатель на первый элемент массива является эквивалентной формой объявления, использование массива подчеркивает ожидаемый тип аргумента.

Функция может возвращать только одно значение. В тоже время, использование массива в качестве параметра функции позволяет возвращать произвольное количество значений, изменяя внутри тела функции элементы переданного массива. В **примере С.25** функция получает на вход массив целых чисел, сортирует его в порядке возрастания и возвращает результат в том же массиве. Все три объявления функций, приведенные ниже, эквивалентны. Длина массива, указанная в объявлении третьей функции, игнорируется.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

Пример С.25 ПЕРЕДАЧА МАССИВА И ЕГО РАЗМЕРА КАК АРГУМЕНТОВ ФУНКЦИИ

```
// Sort the elements of the array vals of length len from lowest to
highest

void sort(int vals[], int len)
{
  int i, j, temp;
  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] >vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Массивы в языке Си могут иметь более одной размерности. Использование двухмерного массива для хранения результатов решения восьми задач десятью студентами, показано в **примере С.26**. Еще раз повторим, что инициализация массива с использованием фигурных скобок { } допустима только в момент его объявления.

Пример С.26 ИНИЦИАЛИЗАЦИЯ ДВУХМЕРНОГО МАССИВА

```
// Initialize 2-D array at declaration
 int grades[10][8] = { {100, 107, 99, 101, 100, 104, 109, 117},
                       {103, 101, 94, 101, 102, 106, 105, 110},
                       {101, 102, 92, 101, 100, 107, 109, 110},
                       {114, 106, 95, 101, 100, 102, 102, 100},
                       {98, 105, 97, 101, 103, 104, 109, 109},
                       {105, 103, 99, 101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99, 107, 98,101, 109, 104, 110, 108} };
```

Пример С.27 демонстрирует функцию для обработки двухмерного массива с оценками из **примера С.26**. При объявлении функции с многомерным массивом в качестве параметра, необходимо указывать размеры всех его измерений, кроме первого. Оба следующих объявления функций являются корректными и эквивалентными:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

Пример С.27 РАБОТА С МНОГОМЕРНЫМИ МАССИВАМИ

```
#include <stdio.h>
// Print the contents of a 10 × 8 array
void print2dArray(int arr[10][8])

{
  int i, j;
  for (i=0; i<10; i++) {      // for each of the 10 students
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}
//  Calculate  the  mean  score  of  a  10  ×  8  arrayfloat  getMean(int
arr[10][8]){  int i, j;
  float mean, total = 0;
  // get the mean value across a 2D array  for (i=0; i<10; i++) {
   for (j=0; j<8; j++) {
    total += arr[i][j];   // sum array values
   }
  }
  mean = total/(10*8);
  printf("Mean is: %f\n", mean);
  return mean;
}
```

Так как массив представляется указателем на его первый элемент, язык Си не поддерживает полное поэлементное копирование или сравнение массивов с использованием операций `=` или `==`. Вместо этого, вы должны использовать цикл для копирования или последовательного сравнения элементов массива один за другим.

С.8.3 Символы

Для хранения символов в языке С существует тип `char`, представляющий переменные размером 8 бит. Такие переменные могут рассматриваться как числа в дополнительном коде в диапазоне от −128 до 127 или как коды ASCII, обозначающие буквы, цифры или символы. Символы ASCII записываются в виде числовых значений (десятичных, шестнадцатеричных или восьмеричных) или непосредственно как символы, заключенные в одиночные кавычки. К примеру, латинской букве A соответствует код 0x41, букве B соответствует код 0x42 и так далее. Таким образом, результат выражения `'A' + 3` равен 0x44 или символу `'D'`. **Табл. 6.2** содержит полный набор кодов ASCII. В **Табл. С.4** перечислены символы, используемые для форматирования выводимого текста, а также другие специальные символы. Символы форматирования включают в себя возврат каретки (`\r`), перевод строки (`\n`), горизонтальную табуляцию (`\t`), символ конца строки (`\0`). Символ \r упомянут для полноты картины, поскольку он используется

очень редко. Этот символ возвращает каретку (позицию, куда будет выведен следующий символ) в начало строки (влево). При последующем выводе текста, старые символы будут затерты. В отличие от `\r`, символ `\n` передвигает позицию вывода в начало новой строки.[23] Нулевой символ *NULL* (`'\0'`) обозначает конец текстовой строки. Мы обсудим его подробнее в **разделе С.8.4** Строки символов.

Табл. С.4 Специальные символы

Символ	Шестнадцатеричный код	Описание
`\r`	0x0D	возврат каретки
`\n`	0x0A	новая строка
`\t`	0x09	табуляция
`\0`	0x00	конец строки
`\\`	0x5C	обратная косая черта (слеш)
`\"`	0x22	двойная кавычка
`\'`	0x27	одинарная кавычка
`\a`	0x07	Звонок

[23] В операционной системе Windows для обозначения перевода строки используется комбинация символов `\r\n`. В системе UNIX для этого служит одиночный символ `\n`. Будьте внимательны, так как это может быть причиной ошибок при переносе текстов между разными операционными системами.

Термин «возврат каретки» происходит из эпохи печатных машинок. Для начала печати с левой стороны листа, нужно было передвинуть каретку – приспособление, удерживающее лист бумаги – до упора вправо. Рычаг перевода каретки, который вы можете видеть на фотографии, нажимался таким образом, чтобы каретка сдвигалась вправо и одновременно перемещала лист бумаги по вертикали (т.е. делала перевод строки).

Печатная машинка Ремингтон, использовавшаяся Уинстоном Черчиллем (*http://cwr.iwm.org.uk/server/show/conMediaFile.71979*)

С.8.4 Строки символов

Строка – это массив символов, содержащий текст, длина которого ограничена размером массива. При этом длина строки не обязательно должна совпадать с размером массива и может быть меньше. Каждый символ занимает один байт. Для кодирования букв, цифр или символов

используется код ASCII. Для обозначения конца строки используется нулевой символ с кодом 0x00. Длину строки можно определить подсчетом символов, начиная с первого и заканчивая предпоследним символом перед нулевым символом конца строки.

> Строки, использующиеся в языке Си, называются строками с завершающим нулем. Их длина определяется поиском нулевого символа в конце строки. Некоторые другие языки программирования, такие как Паскаль, используют другое представление строк. Длина строки в Паскале хранится в её первом байте. В таком представлении длина строки ограничена 255 байтами. Первый байт называется префиксным байтом, а строки языка Паскаль называют *P*-строками. Преимуществом строк с завершающим нулем является отсутствие ограничений на их размер. Преимущество *P*-строк в возможности быстрого определения их длины, без необходимости перебора всех символов строки в поисках завершающего нулевого символа.

В **примере С.28** показано объявление массива `greeting` из десяти символов, содержащего строку “`Hello!`”.

Пример С.28 ОБЪЯВЛЕНИЕ СТРОКИ

```
char greeting[10] = "Hello!";
```

Для определенности предположим, что массив `greeting` расположен в памяти, начиная с адреса 0x50. На **Рис. С.5** показано содержимое ячеек

памяти с 0x50 до 0x59, содержащих строку `"Hello!"`. Обратите внимание, что строка занимает только первые семь элементов массива, хотя в памяти для него выделено десять байт.

Пример С.29 демонстрирует ещё один способ объявления строки `greeting`. Указатель `greeting` хранит адрес первого элемента массива, состоящего из семи символов составляющих строку `"Hello!"`, включая завершающий нулевой символ. Пример также демонстрирует использование функции `printf` и форматирующего спецификатора `%s` для вывода строки на экран.

Пример С.29 АЛЬТЕРНАТИВНЫЙ СПОСОБ ОБЪЯВЛЕНИЯ СТРОКИ

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Вывод

```
greeting: Hello!
```

В отличие от обычных переменных, строка не может быть скопирована в другую строку прямым присваиванием `=`. Как и для массивов других типов, каждый символ исходной строки должен быть скопирован индивидуально. В **пример С.30** строка `src` копируется в строку `dst`. При копировании нам не нужно знать размер исходной строки, так как все символы копируются до тех пор, пока не будет найден

завершающий нулевой символ. Тем не менее, массив `dst` должен быть достаточно большим, чтобы он мог вместить все символы исходной строки. Стандартная библиотека языка С содержат набор функций для работы со строками, и в том числе, функцию `strcpy`, используемую для копирования строк (смотри **раздел С.9.4** string).

Пример С.30 КОПИРОВАНИЕ СТРОК

```
// Copy the source string, src, to the destination string, dst

void strcpy(char *dst, char *src)
{
int i = 0;
  do {
    dst[i] = src[i];       // copy characters one byte at a time
  } while (src[i++]);      // until the null terminator is found
}
```

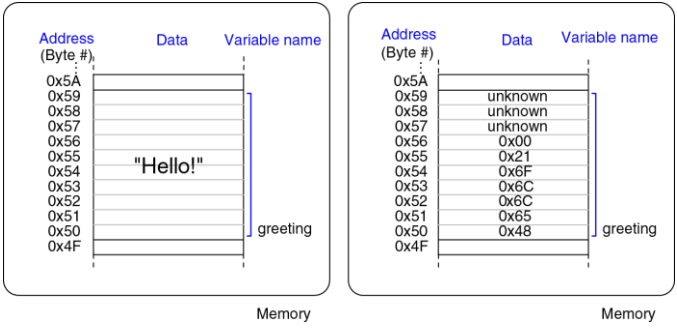


Рис. С.5 **Размещение строки "Hello!" в памяти.**

С.8.5 Структуры

В языке Си структуры используются для хранения наборов данных различных типов. Ниже приведена обобщенная форма объявления структуры:

```
struct name {
  type1 element1;
  type2 element2;
  …
};
```

Объявление структуры начинается с ключевого слова `struct`. За ним следует имя структуры `name`. Далее, в фигурных скобках перечислены компоненты структуры `element1` и `element2`. Структура может содержать произвольное количество компонент. В **примере С.31** объявляется структура contact, используемая для хранения контактной информации. Пример демонстрирует объявление переменной `c1` имеющей тип `struct contact`.

Пример С.31 ОБЪЯВЛЕНИЕ СТРУКТУР

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
strcpy(c1.name,"Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Вы можете создавать массивы структур и указатели на структуры, используя те же конструкции, что и для встроенных типов данных. В **примере С.32** показано объявления массива структур, хранящих контактные данные.

Пример С.32 МАССИВ СТРУКТУР

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

Общеупотребительной практикой является использование указателей на структуры. Для разыменовывания указателя и доступа к компоненте структуры, используется операция `->` (*доступ к компоненте*). В **примере С.33** показано объявление указателя на структуру `contact`, присваивание ему адреса сорок второго элемента массива структур `classlist` из **примера С.32** и использование операции доступа к компоненте для присваивания нового значения элементу структуры.

Пример С.33 ДОСТУП К КОМПОНЕНТЕ СТРУКТУРЫ, ИСПОЛЬЗУЯ УКАЗАТЕЛЬ И `->`

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9; // equivalent to: (*cptr).height = 1.9;
```

Структуры могут быть переданы как аргументы функции или получены из нее по значению или по ссылке. При передаче структуры по значению, компилятор копирует все элементы структуры для дальнейшей работы с этой копией внутри функции. Если размер структуры достаточно большой, такой подход может потребовать существенного расхода памяти и времени. Передача структуры по

ссылке сводится к передаче в функцию указателя на структуру и такой способ гораздо эффективнее. При передаче по ссылке, функция может также изменить полученную структуру, а не возвращать новую копию. В **примере С.34** демонстрируются два варианта функции `stretch`, уменьшающей значение компоненты структуры height на 2 сантиметра. Функция `stretchByReference`, использует передачу по ссылке, чтобы изменить значение компоненты без двойного копирования всех данных структуры.

Пример С.34 ПЕРЕДАЧА СТРУКТУР ПО ССЫЛКЕ И ПО ЗНАЧЕНИЮ

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}
int main(void)
{
  struct contact George;
  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);     // and stretch some more
}
```

С.8.6 * Оператор `typedef`

Язык Си позволяет программисту определять свои собственные имена для типов данных, используя оператор `typedef`. В частности, вместо использования длинного имени типа `struct` contact, мы можем определить новое короткое имя `contact`, как это продемонстрировано в **примере С.35**.

Пример С.35 ОПРЕДЕЛЕНИЕ НОВОГО ИМЕНИ ТИПА ОПЕРАТОРОМ `TYPEDEF`

```
typedef struct contact {
  char name[30];
  int phone;
  float height; // in meters
} contact;        // defines contact as shorthand for "struct contact"

contact c1;       // now we can declare the variable as type contact
```

Оператор `typedef` может быть использован для создания нового типа данных, занимающего тот же объем памяти, что и встроенный тип. В **примере С.36** определяются два новых типа данных `byte` и `bool` размером 8 бит. Использование типа byte в объявлении переменной pos делает более ясным её назначение. Она представляет собой 8-битное число, а не символ в коде ASCII. Тип `bool` указывает, что переменная является 8-битным числом с логическими значениями

TRUE или FALSE. Пользовательские имена типов делают программу более понятной, в отличие от использования встроенного типа char для любых маленьких переменных.

Пример С.36 ОПРЕДЕЛЕНИЕ ТИПОВ ДАННЫХ `BYTE` И `BOOL`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

В **примере 37** показано использование оператора `typedef` для определения типов `vector` и `matrix`, представляющих собой одномерный массив из трех элементов и двухмерный массив 3 × 3, соответственно.

Пример С.37 ОПРЕДЕЛЕНИЕ ИМЁН ТИПОВ `VECTOR` И `MATRIX`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

С.8.7 Динамическое распределение памяти

Во всех предыдущих примерах, память для структур данных выделялась *статически*, так как размер данных был известен в момент компиляции. Такой способ выделения памяти может быть неудобен для массивов и строк переменной длины. Программист вынужден определять размер массива с запасом, чтобы он мог вместить самый большой объем данных, который может встретиться программе. Альтернативный способ – *динамическое* выделение памяти во время выполнения программы. В этом случае мы можем запросить ровно тот объем памяти, который нам нужен.

Функция `malloc` , объявленная в системном заголовочном файле `stdlib.h`, выделяет блок памяти требуемого размера и возвращает указатель на него. При невозможности выделить запрошенное количество памяти, функция возвращает NULL. В следующем примере показано выделение памяти для десяти переменных типа `short` (10×2 = 20 байт). Функция `sizeof` возвращает размер в байтах переменной или типа данных.

```
// динамическое выделение 20 байт памяти
short *data = malloc(10*sizeof(short));
```

Пример С.38 демонстрирует динамическое выделение и освобождение памяти. Программа получает произвольное количество чисел, затем

сохраняет их в динамически созданном массиве и вычисляет их среднее значение. Количество требуемой памяти зависит от числа элементов массива и размера каждого элемента. Например, если размер типа `int` составляет четыре байта и массив состоит из десяти элементов, необходимо выделить 10x4=40 байт. Функция `free` освобождает выделенную память, после чего она может быть повторно использована. Динамически выделяемую память необходимо всегда освобождать, а иначе это приведёт к *утечкам памяти* в программе.

Пример С.38 ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ПАМЯТИ

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Вставьте сюда код функции getMean из примера С.24.

int main(void) {
  int len, i;
  int *nums;

  printf("How many numbers would you like to enter?  ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:  ");
```

```
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

С.8.8 Связные списки

Связный список – это популярная структура данных, применяемая для хранения переменного числа элементов. Каждый элемент списка представляет собой структуру, содержащую от одного до нескольких полей с данными и указатель на следующий элемент списка. Первый элемент списка называется *головой* списка. С помощью связных списков можно проиллюстрировать несколько концепций языка С, таких как структуры, указатели и динамическое выделение памяти.

В **примере С.39** связный список используется для хранения списка учетных записей пользователей, размер которого может изменяться. Каждый пользователь имеет имя, пароль, уникальный идентификатор (UID) и флаг, указывающий на обладание административными привилегиями. Связный список содержит структуры типа `userL`, содержащие всю информацию о пользователе и указатель на следующий элемент в списке. Указатель на *голову* списка хранится в

глобальной переменной `users`. Его начальное значение NULL указывает на то, что список пустой.

Пример С.39 СВЯЗНЫЙ СПИСОК

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];  // user name
  char passwd[80]; // password
  int uid;         // user identification number
  int admin;       // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;
void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;
  newUser = malloc(sizeof(userL)); // create space for new user
  strcpy(newUser->uname, uname);   // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;          // insert at start of linked list
  users = newUser;
}

void deleteUser(int uid) { // delete first user with given uid
```

```
  userL *cur = users;
  userL *prev = NULL;
  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;      // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
```

```
    cur = cur->next;
  }
  return count;
}
```

В программе объявлены функции для вставки, удаления и поиска пользователя, а также функция для подсчета количества пользователей. Функция `insertUser` выделяет память для нового элемента и добавляет этот элемент в голову списка. Функция `deleteUser` перебирает элементы списка до тех пор, пока не будет найден пользователь с указанным идентификатором UID. После этого, функция удаляет элемент из списка, соединяя его соседей напрямую, и освобождает память, выделенную удаляемому элементу. Функция `findUser` перебирает элементы списка до тех пор, пока не будет найден пользователь с указанным идентификатором UID и возвращает указатель на найденный элемент или NULL, когда идентификатор не найден. Функция `numUsers` подсчитывает количество элементов в списке.

КРАТКИЙ ИТОГ

- **Указатели:** Указатель хранит адрес переменной.
- **Массивы:** Массив, это набор однотипных элементов, объявленный с использованием квадратных скобок `[]`.
- **Символы:** Тип char может хранить небольшие целые числа или коды ASCII, обозначающие текстовые или специальные символы.
- **Строки:** Строка – это массив символов. Концом строки является завершающий символ 0x00.
- **Динамическое выделение памяти:** Функция `malloc` из стандартной библиотеки служит для выделения блока памяти во время работы программы. Функция `free` освобождает выделенную память.
- **Связные списки:** Связный список – распространенная структура данных для хранения переменного числа элементов.

С.9 СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА С

Программисты регулярно используют стандартный набор функций, таких как процедуры печати или функции тригонометрических вычислений. Вместо того, чтобы изобретать велосипед и писать каждый раз подобные функции заново, язык С предоставляет набор *библиотек*, содержащих часто используемые функции. Каждая библиотека состоит из заголовочного файла и связанного с ним файла скомпилированного объектного кода. Заголовочный файл содержит объявления переменных, функций и определения типов данных. Объектный файл содержит непосредственно код функций. Для получения исполняемого кода, объектный код программы должен быть скомпонован с библиотеками, которые используются программой. Подключение библиотек уменьшает общее время компиляции программы, поскольку они не компилируются повторно, а используется уже готовый объектный код. В **Табл. С.5** перечислены некоторые наиболее употребительные библиотеки С и дано их краткое описание.

С.9.1 stdio

Стандартная библиотека ввода/вывода stdio.h содержит функции для вывода на консоль, чтения ввода с клавиатуры, чтения и записи файлов. Для использования этих функций, заголовочный файл должен быть включен в начало Си-файла:

```
#include <stdio.h>
```

printf

Функция `printf` используется для *форматного вывода* текста. Функция принимает один обязательный параметр – строку, заключенную в кавычки " ", и необязательный список переменных, значения которых необходимо вывести на консоль. Строка содержит текст и, опционально, *спецификаторы* форматирования выводимых данных. Перечень имеющихся спецификаторов приведен в **Табл. С.6**. **Пример 40** демонстрирует использование функции printf.

Табл. С.5 Часто используемые файлы стандартной библиотеки

Заголовочный файл библиотеки	Описание
`stdio.h`	**Библиотека ввода/вывода**. Содержит функции печати и чтения данных в/из файла или консоли (`printf`, `fprintf` и `scanf`, `fscanf`) и функции открытия и закрытия файлов (`fopen` и `fclose`).
`stdlib.h`	**Стандартная библиотека**. Содержит функции для генерации случайных чисел (`rand` и `srand`), динамического выделения и освобождения памяти (`malloc` и `free`), завершения программы (`exit`) и преобразований строк в числовые типы данных и обратно (`atoi`, `atoll` и `atof`).
`math.h`	**Математическая библиотека**. Содержит стандартные математические функции, такие как `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor` и `ceil`.
`string.h`	**Библиотека функция для работы со строками**. Содержит функции для сравнения, копирования, объединения строк и вычисления длины строки.

Табл. С.6 Коды форматирования, используемые функцией printf

Код	Формат вывода
`%d`	Десятичное число
`%u`	Беззнаковое десятичное число
`%x`	Шестнадцатеричное число
`%o`	Восьмеричное число
`%f`	Число с плавающей запятой (тип float или double)
`%e`	Число с плавающей запятой (тип float или double) в экспоненциальной форме (например, 1.56e7)
`%c`	Символ (тип char)
`%s`	Строка (массив символов, завершающийся нулем)

Пример С.40 ИСПОЛЬЗОВАНИЕ `PRINTF` ДЛЯ ВЫВОДА НА КОНСОЛЬ

```
// Simple print function
#include <stdio.h>
int num = 42;
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Вывод:

```
The answer is 42.
```

По умолчанию, типы данных `float` и `double` выводятся с шестью знаками после запятой. Для изменения формата вывода, `%f` необходимо заменить на `%w.df`, где *спецификатор ширины* `w` указывает минимальное количество знаков, используемых для вывода числа, а *спецификатор точности* `d` задает минимальное количество символов после десятичной запятой (точки). Обратите внимание, что при подсчете ширины вывода учитывается и десятичная запятая. В **примере С.41**, для вывода переменной `pi` используется четыре символа, из них два – для вывода дробной части: 3.14. Для переменной `e` используется восемь символов, три из которых приходятся на дробную часть. Так как целая часть числа содержит только одну цифру, то при выводе она дополняется тремя нулями до требуемой ширины. Минимальная ширина вывода для переменной `c` составляет пять символов и три из них – это дробная часть. Но содержимое переменной не укладывается в этот размер и поэтому минимальная ширина превышается. Выводятся все цифры, составляющие целую часть числа, и три цифры дробной части.

Пример С.41 ФОРМАТИРОВАНИЕ ВЫВОДА ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

```
// Печать чисел с плавающей запятой, используя различные форматы
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne = %8.3f\nc = %5.3f\n", pi, e, c);
```

Вывод:

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Символы `%` и `\` используются как элементы специальных знаков для управления форматом вывода. Поэтому, для вывода символов процента и обратного слеша на экран, их необходимо удвоить, как показано в **примере С.42**.

Пример С.42 ИСПОЛЬЗОВАНИЕ `PRINTF` ДЛЯ ПЕЧАТИ ЗНАКА ПРОЦЕНТА И ОБРАТНОГО СЛЕША

```
// Вывод символов % и \ на консоль
printf("Here are some special characters: %% \\ Пример С.\");
```

Вывод:

```
Here are some special characters: % \
```

scanf

Функция `scanf` считывает данные, вводимые с клавиатуры. Она использует те же спецификаторы формата, что и функция `printf`. **Пример С.43** демонстрирует использование функции `scanf`. При вызове функции `scanf` выполнение программы приостанавливается в ожидании ввода данных пользователем. Параметрами функции `scanf`

является строка с набором спецификаторов формата вводимых данных и указатели на переменные, в которых будут сохранены полученные значения.

Пример С.43 ИСПОЛЬЗОВАНИЕ SCANF ДЛЯ ЧТЕНИЯ ДАННЫХ С КЛАВИАТУРЫ

```
// Чтение переменных из командной строк
и#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;
  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);   // note no & needed: str is a pointer
}
```

Чтение и запись файлов

Многим программам требуется читать данные из файлов или записывать в файлы большой объём информации. Перед началом работы с файлом его необходимо открыть, используя функцию `fopen`.

Для чтения из файла и записи в файл предназначены две функции `fscanf` и `fprintf`. По завершению всех операций с файлом, он должен быть закрыт вызовом функции `fclose`.

Параметрами функции `fopen` являются имя файла и строка символов, определяющая *режим доступа* к файлу. В случае успешного открытия файла, функция возвращает указатель на *дескриптор файла*, имеющий тип `FILE*`. Если файл не может быть открыт, то `fopen` возвращает NULL. Это может случиться, когда для чтения данных открывается несуществующий файл или делается попытка открыть для записи файл, в который уже производит запись другая программа. Имеются следующие режимы доступа:

`"w"`: Write to a file. If the file exists, it is overwritten.

`"r"`: Read from a file.

`"a"`: Append to the end of an existing file. If the file doesn't exist, it is created.

`"w"`: Открытие файла для записи. Если файл уже существует, он будет перезаписан.

`"r"`: Открытие файла для чтения.

`"a"`: Открытие для записи данных в конец файла. Если файла нет, он будет создан.

Вызов функции открытия файла и проверку того, что она вернула ненулевой указатель на дескриптор файла, обычно записывают в одной строке. Такое способ показан в **примере С.44**. Тем не менее, вызов функции fopen и проверку результата её работы можно разбить на две строки:

```
fptr = fopen("result.txt", "w");
if (fptr == NULL)
…
```

Пример С.44 демонстрирует открытие файла, запись в него данных и закрытие файла. Хорошей практикой является проверка результата открытия файла и выдача пользователю сообщения в случае ошибки. Функция `fprintf` похожа на функцию `printf`. Помимо форматной строки и данных для вывода, она получает указатель на файловый дескриптор. Функция `fclose` сохраняет в файл все данные из файлового буфера, затем закрывает файл и освобождает выделенные системные ресурсы. Функция `exit` будет описана в **разделе С.9.2**.

Пример С.44 ИСПОЛЬЗОВАНИЕ `FPRINTF` ДЛЯ ЗАПИСИ ДАННЫХ В ФАЙЛ

```
// Write "Testing file write." to result.txt#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1);  // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

Пример С.45 показывает чтение чисел из файла `data.txt` с использованием функцию `fscanf`. Перед вызовом этой функции файл должен быть открыт на чтение. Затем программа считывает из файла числа и выводит их на экран. Эти действия повторяются до конца файла. Для определения достижения конца файла служит функция `feof`. По завершению работы, программа закрывает файл для освобождения системных ресурсов.

Пример С.45 ИСПОЛЬЗОВАНИЕ FSCANF ДЛЯ ЧТЕНИЯ ДАННЫХ ИЗ ФАЙЛА

```
#include <stdio.h>
int main(void)
{
  FILE *fptr;
  int data;
  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) {   // check that the end of the file hasn't been
reached
   fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }
  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Вывод:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Другие полезные функции из библиотеки STDIO

Функция `sprintf` записывает отформатированные данные в строку символов в памяти. Функция `sscanf` считывает данные из строки. Для чтения одиночного символа из файла используется функция `fgetc`. Функция `fgets` предназначена для чтения из файла целой строки текста.

Возможности функции `fscanf` по чтению данных из файла довольно ограничены. Функцию `fgets` удобно использовать для последовательного чтения файла строка за строкой и применять функцию `sscanf` для обработки прочитанной строки. Еще один способ – чтение и анализ содержимого файла по одному символу, используя вызов `fgetc`.

С.9.2 `stdlib`

Стандартная библиотека `stdlib.h` предоставляет набор функций общего назначения. В ней имеются функции для генерации случайных чисел (`rand` и `srand`), динамического выделения памяти (`malloc` и `free`, обсуждались в **разделе С.8.8** Динамическое распределение памяти), функция завершения работы программы (`exit`) и преобразования форматов данных. Чтобы использовать эти функции, необходимо включить следующую директиву в начало программы:

```
#include <stdlib.h>
```

Функции `rand` и `srand`

Функция `rand` возвращает псевдослучайное целое число. Последовательность псевдослучайных чисел обладает статистическими характеристиками последовательности случайных чисел, но является детерминированной. Конкретный набор чисел зависит от начального значения, называемого ключом или зерном. Для приведения числа к заданному диапазону используется операция деления по модулю (%). **Пример С.46** демонстрирует данный метод для получения случайных чисел в диапазоне от 0 до 9. Значения переменных `x` и `y` – это случайные числа, но они будут одинаковыми при каждом запуске программы.

Пример С.46 ИСПОЛЬЗОВАНИЕ RAND ДЛЯ ГЕНЕРАЦИИ СЛУЧАЙНЫХ ЧИСЕЛ

```
#include <stdlib.h>i
nt x, y;x = rand();         //

x = a random integery = rand() % 10;    //
y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Вывод:

```
x = 1481765933, y = 3
```

По историческим причинам, функция `time` возвращает количество секунд, прошедшее с 00:00 UTC 1 января 1970 года. Аббревиатура UTC обозначает универсальное координированное время, которое может рассматриваться, как эквивалент среднего времени по Гринвичу (GMT). Эта дата установлена в честь операционной системы UNIX, разработанной в 1969 году группой сотрудников Bell Labs, включавшей Денниса Ритчи и Брайана Кернигана. По аналогии с празднованием Нового Года, группы энтузиастов UNIX празднуют наступление моментов времени, соответствующих особым значениям, возвращаемым функцией `time`. Например, 1 февраля 2009 года в 23 часа 31 минуту 30 секунд UTC, функция `time` вернула значение 1 234 567 890. В 2038 году переменные целого 32-битного типа со знаком, использующиеся для хранения времени UNIX, переполнятся и вернутся в 1901 год, так как в типе со знаком интервалы времени могут быть отрицательными.

Для генерации отличающихся последовательностей случайных чисел программист должен изменять начальное значение ключа. Для этого вызывается функция `srand`, получающая в качестве аргумента новое значение ключа, который должен быть всегда разным. Как показано в **примере С.47**, для получения ключей можно использовать вызов функции `time`, возвращающей текущее время в секундах.

Пример С.47 ИСПОЛЬЗОВАНИЕ SRAND ДЛЯ ИНИЦИАЛИЗАЦИИ ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>   // needed to call time()

int main(void)
{
  int x;

  srand(time(NULL));     // seed the random number generator
  x = rand() % 10;       // random number from 0 to 9
  printf("x = %d\n", x);
}
```

Функция `exit`

Функция `exit` завершает работу программы. Она имеет единственный числовой параметр, который программа возвращает операционной системе. Это число должно сообщать результат выполнения программы. Ноль означает нормальное завершение программы, ненулевое значение обозначает ошибку.

Преобразование форматов: `atoi`, `atol`, `atof`

Стандартная библиотека предоставляет функции для преобразования строк ASCII символов в числовые типы данных `integer`, `long integer`

и `double`. Это функции `atoi`, `atoll` и `atof`, использование которых показано в **примере С.48**. Помимо прочего, эти функции удобно использовать для чтения разнородных типов данных из файла и для разбора аргументов командной строки, представляющих смешанный набор чисел и строк. Работа с командной строкой рассмотрена подробнее в **разделе С.10.3** «Аргументы командной строки».

Пример С.48 ПРЕОБРАЗОВАНИЕ ФОРМАТОВ

```
// Преобразование строки ASCII символов в типы данных ints, longs и floats
#include <stdlib.h>int main(void)
{

  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");
  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Вывод:

```
x = 42 y = 833 z = 3.822000
```

С.9.3 `math`

Библиотека `math.h` содержит часто используемые функции для математических вычислений. В их число входят тригонометрические функции, процедуры для вычисления корней и логарифмов. Вызовы математических функций демонстрируются в **примере С.49**. Для использования этих функций необходимо включить соответствующий заголовочный файл в начало программы:

```
#include <math.h>
```

Пример С.49 МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);          // 1, note: the input argument is in radians
  b = 2 * acos(0);     // pi (acos means arc cosine)
  c = sqrt(144);       // 12
  d = exp(2);          // e^2 = 7.389056,
  e = log(7.389056);   // 2 (natural logarithm, base e)
  f = log10(1000);     // 3 (log base 10)
  g = floor(178.567);  // 178, rounds to next lowest whole number
```

```
  h = pow(2, 10);       // computes 2 raised to the 10th power

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f,
          g = %.2f, h = %.2f\n", a, b, c, d, e, f, g, h);
}
```

Вывод:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00,
h = 1024.00
```

С.9.4 string

Библиотека `string.h` предоставляет широкий набор функций для манипуляций со строками. Ниже приведены основные функции из этой библиотеки:

```
// копирует строку src в строку dst и возвращает dst
char *strcpy(char *dst, char *src);

// объединяет (добавляет) src в конец строки dst и возвращает dst
char *strcat(char *dst, char *src);

// сравнивает две строки. Возвращает 0, если строки совпадают
// и ненулевое значение в противном случае
int strcmp(char *s1, char *s2);

// возвращает длину строки, не включая завершающий нулевой символ
int strlen(char *str);
```

С.10 КОМПИЛЯТОР И ОПЦИИ КОМАНДНОЙ СТРОКИ

До этого момента мы рассматривали достаточно простые Си-программы. В реальном мире программы могут состоять из десятков или даже тысяч файлов. Это требуется для удобства работы с исходными текстами, модульной организации программы и обеспечения совместной работы множества программистов. В этом разделе мы обсудим использование компилятора для генерации кода программы из множества исходных файлов, а также рассмотрим опции компилятора и аргументы командной строки.

С.10.1 Компиляция нескольких исходных с-файлов

Чтобы скомпилировать сразу несколько файлов с исходными текстами, вы должны перечислить имена всех файлов в командной строке при запуске компилятора. Помните, что в программе должна быть только одна функция `main`, которая обычно находится в файле `main.c`.

```
gcc main.c file2.c file3.c
```

С.10.2 Опции компилятора

Опции компилятора позволяют программисту задать такие параметры, как имя и формат генерируемого файла, настройки оптимизации и так далее. Имена и назначение опций не стандартизированы, но в

Табл. С.7 перечислены опции, используемые наиболее часто. Каждая опция начинается с символа тире (-). Например, опция «`-o`» позволяет указать своё имя файла с исполняемым кодом вместо имени `a.out`, которое будет дано компилятором по умолчанию. Количество опций очень велико. Полный список опций можно получить, запустив в консоли команду «`gcc -help`».

Табл. С.7 Опции компилятора

Опция	Описание	Пример
`-o` файл	Задает имя выходного файла	`gcc -o hello hello.c`
`-S`	Компиляция программы в файл на языке ассемблера (не исполняемый)	`gcc -S hello.c` на выходе получаем `hello.s`
`-v`	Расширенная диагностика – вывод расширенных данных о ходе компиляции	`gcc -v hello.c`
`-O`уровень	Уровень оптимизации программы (обычно, уровень принимает значения от 0 до 3). Повышение уровня позволяет получить более быстрый и/или маленький код в обмен на скорость компиляции	`gcc -O3 hello.c`
`--version`	Вывод версии компилятора	`gcc -version`
`--help`	Вывод всех опций компилятора	`gcc --help`
`-Wall`	Вывод всех предупреждений при компиляции	`gcc -Wall hello.c`

С.10.3 Аргументы командной строки

Как и другие функции, `main` может принимать входные аргументы. Аргументы для `main` перечисляются в командной строке при запуске программы. **Пример С.50** демонстрирует обработку командной строки. Параметр `argc` называется *счетчиком аргументов* и содержит число аргументов, переданных программе. Параметр `argv` называется *вектором аргументов* и является массивом строк, указанных в командной строке. Предположим, что программа из **примера С.50** компилируется в файл `testargs`. Если запустить её, как показано в примере ниже, значение параметра `argc` будет равно четырем, а массив `argv` будет содержать следующие значения `{"./testargs", "arg1", "25", "lastarg!"}`. Обратите внимание, что первым идет имя самой программы. Результат выполнения программы показан после **примера С.50**.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Если программе требуются числовые аргументы, то она может получить их из строковых значений с использованием функций для преобразования форматов данных, объявленных в `stdlib.h`.

Пример С.50 АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ

```
// Print command line arguments
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i])
;}
```

Вывод:

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

С.11 ТИПИЧНЫЕ ОШИБКИ

Как и с любым другим языком программирования, вы наверняка будете допускать ошибки при написании нетривиальных программ. Далее мы перечислим некоторые из наиболее распространенных ошибок, допускаемых при программировании на языке Си. Часть из них способна доставить особенно серьёзные неприятности, так как программа с этими ошибками будет успешно скомпилирована, но результат её работы окажется неверным.

Навыки отладки приходят с практикой. Тем не менее, мы можем дать несколько советов.

- Начинайте исправлять код, начиная с первого сообщения об ошибке, выданного компилятором. Последующие сообщения могут быть вызваны наведённым эффектом от первой ошибки. После коррекции текста, запустите компиляцию снова и продолжайте работу над текстом программы до тех пор, пока все ошибки, обнаруживаемые компилятором, не будут исправлены.
- Если вы не видите ошибки в строке, на которую указывает компилятор, посмотрите на предыдущие строки. Возможно, там пропущена точка с запятой после оператора.
- Разделяйте длинные и сложные выражения на несколько строк.
- Используйте функцию printf для печати промежуточных результатов программы.
- Если результат программы не соответствует ожидаемому результату, начинайте отладку кода с самого раннего места, где обнаружилось отклонение в данных.
- Обращайте внимание на все предупреждения компилятора. Некоторые из них можно проигнорировать, но другие могут указывать на скрытые проблемы, которые приводят к ошибочной работе программы.

Ошибка С.1 ПРОПУЩЕННЫЙ & В ВЫЗОВЕ SCANF

Ошибочный код

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // перед a пропущен &
```

Правильный код:

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

Ошибка С.2 ИСПОЛЬЗОВАНИЕ = ВМЕСТО == ДЛЯ СРАВНЕНИЯ

Ошибочный код

```
if (x = 1)  // всегда принимает значение TRUE
  printf("Found!\n");
```

Правильный код

```
if (x == 1)
  printf("Found!\n");
```

Ошибка С.3 ИНДЕКС ПОСЛЕДНЕГО ЭЛЕМЕНТА МАССИВА

Ошибочный код

```
int array[10];
array[10] = 42;     // индекс должен быть в диапазоне 0-9
```

Правильный код

```
int array[10];
array[9] = 42;
```

Ошибка С.4 ИСПОЛЬЗОВАНИЕ = В ДИРЕКТИВЕ #DEFINE

Ошибочный код

```
// это приведет к замене NUM на строку "= 4
"#define NUM = 4
```

Правильный код

```
#define NUM 4
```

Ошибка С.5 ИСПОЛЬЗОВАНИЕ НЕИНИЦИАЛИЗИРОВАННОЙ ПЕРЕМЕННОЙ

Ошибочный код

```
int i;
if (i == 10) // переменная i не инициализирована
  ...
```

Правильный код

```
int i = 10;
if (i == 10)
  ...
```

Ошибка С.6 НЕПОЛНЫЙ ПУТЬ К ПОЛЬЗОВАТЕЛЬСКОМУ ЗАГОЛОВОЧНОМУ ФАЙЛА

Неполный путь к файлу

```
#include  "myfile.h"
```

Правильный путь к файлу

```
#include  "othercode\myfile.h"
```

Ошибка С.7 ИСПОЛЬЗОВАНИЕ ЛОГИЧЕСКИХ ОПЕРАЦИЙ (`!`, `||`, `&&`) ВМЕСТО БИТОВЫХ ОПЕРАЦИЙ (`~`, `|`, `&`)

Ошибочный код

```
char x=!5;    // логическое НЕТ: x = 0
char y=5||2;  // логическое ИЛИ: y = 1
char z=5&&2;  // логическое И: z = 1
```

Правильный код

```
char x=~5;   // bitwise NOT: x = 0b11111010
char y=5|2;  // bitwise OR:  y = 0b00000111
char z=5&2;  // logical AND: z = 0b00000000
```

Ошибка С.8 ЗАБЫТЫЙ `BREAK` В ОПЕРАТОРЕ `SWITCH/CASE`

Ошибочный код

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
 case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default:direction = 0;}
// direction = 0
```

Правильный код

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default:direction = 0;
}
// direction = 2
```

Ошибка С.9 ПРОПУЩЕННЫЕ ФИГУРНЫЕ СКОБКИ { }

Ошибочный код

```
if (ptr == NULL) // пропущены фигурные скобки
   printf("Unable to open file.\n");
   exit(1);      // выполняется независимо от условия
```

Правильный код

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

Ошибка С.10 ИСПОЛЬЗОВАНИЕ ФУНКЦИИ ДО ЕЁ ОБЪЯВЛЕНИЯ

Ошибочный код

```
int main(void)
{
  test();}

void test(void)
{...
}
```

Правильный код

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

Ошибка С.11 ОБЪЯВЛЕНИЕ ГЛОБАЛЬНЫХ И ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ С ОДИНАКОВЫМ ИМЕНЕМ

Ошибочный код

```
int x = 5;  // объявление глобальной переменной x
int test(void)
{
  int x = 3; // объявление локальной переменной x
  ...
}
```

Правильный код

```
int x = 5;  // объявление глобальной переменной x
int test(void)
{
  int y = 3; // объявление локальной переменной y
 ...
}
```

Ошибка С.12 ИНИЦИАЛИЗАЦИЯ МАССИВА С ИСПОЛЬЗОВАНИЕМ { } ПОСЛЕ ЕГО ОБЪЯВЛЕНИЯ

Ошибочный код

```
int scores[3];
scores = {93, 81, 97}; // ошибка компиляции
```

Правильный код

```
int scores[3] = {93, 81, 97};
```

Ошибка С.13 ПРИСВАИВАНИЕ МАССИВОВ, ИСПОЛЬЗУЯ =

Ошибочный код

```
int scores[3]= {88, 79, 93};
int scores2[3];

scores2 = scores;
```

Правильный код

```
int scores[3]= {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
  scores2[i] = scores[i];
```

Ошибка С.14 ПРОПУЩЕННАЯ ТОЧКА С ЗАПЯТОЙ ПОСЛЕ ЦИКЛА DO/WHILE

Ошибочный код

```
int num;
do {
  num = getNum();
} while (num <100)  // пропущена ;
```

Правильный код

```
int num;
do {
  num = getNum();
} while (num <100);
```

Ошибка С.15 ИСПОЛЬЗОВАНИЕ ЗАПЯТОЙ ВМЕСТО ТОЧКИ С ЗАПЯТОЙ В ЦИКЛЕ FOR

Ошибочный код

```
for (i=0, i <200, i++)
 ...
```

Правильный код

```
for (i=0; i <200; i++)
 ...
```

Ошибка С.16 ДЕЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ ВМЕСТО ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Ошибочный код

```
// целочисленное деление (с отбрасыванием остатка)
// происходит, когда оба операнда операции деления являются
// целыми числами
float x = 9 / 4;     // x = 2.0
```

Правильный код

```
// для получения числа с плавающей точкой,
// хотя бы один из операндов операции деления
// должен иметь тип float или double
float x = 9.0 / 4;  // x = 2.25
```

Ошибка С.17 ЗАПИСЬ ПО НЕИНИЦИАЛИЗИРОВАННОМУ УКАЗАТЕЛЮ

Ошибочный код

```
int *y = 77;
```

Правильный код

```
int x, *y = &x;
*y = 77;
```

Ошибка С.18 ЗАВЫШЕННЫЕ ОЖИДАНИЯ ИЛИ НАДЕЖДА НА УДАЧУ

Начинающий программист часто пишет программу в виде длинной последовательности операторов и при этом надеется, что его программа сразу заработает и будет выполняться правильно. Но реальная практика показывает, что любую нетривиальную программу лучше разбить на короткие фрагменты, оформленные в виде функций, и протестировать каждую из них. С ростом размера программы, сложность и время её отладки возрастают экспоненциально.

Другая распространенная ошибка – надежда на удачу. Это происходит, когда программист, закончив подготовку текста, запускает программу и проверяет, что она работает, но при этом не проверяет корректность результатов, выдаваемых программой. Очень важно выполнить тестирование программы, когда на вход программе подаются разные заранее подготовленные наборы входных данных и затем фактические результаты сравниваются с ожидаемыми результатами.

В этом приложении рассматривается использование языка Си для написания программ, выполняемых на системах, подобных компьютерам с операционной системой Linux. В **главе 8.6** описывается применение Си для программирования микроконтроллеров PIC32, используемых во встраиваемых системах. Язык Си используется для программирования микроконтроллеров благодаря тому, что он обеспечивает практически такой же низкоуровневый доступ к оборудованию, как и язык ассемблера. Но в отличие от языка ассемблера, программы на Си более лаконичны и пишутся быстрее.

Литература для дальнейшего чтения

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

Захватывающая биография Роберта Нойса, одного из изобретателей интегральной микросхемы и основателя Fairchild Semiconductor и Intel. Для каждого, кто думает о работе в Силиконовой долине, книга даст понимание культуры региона, культуры, на которое Нойс оказал большее влияние, чем любой другой человек.

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005

История разработки нескольких поколений процессоров Pentium, рассказанная одним из руководителей проекта. Для тех, кто рассматривает карьеру в этой области, книга покажет, как выглядит управление огромными проектами и вид из-за кулис на разработку одной из наиболее значимых линеек микропроцессоров.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.
Наиболее полный текст по машинной арифметике. Великолепный ресурс по построению высококачественных арифметических устройств для компьютеров.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach, 5th ed.*, Morgan Kaufmann, 2011.
Авторитетная книга по компьютерной архитектуре высокой производительности. Если вы интересуетесь работой передовых микропроцессоров, эта книга для вас.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981
Классическая история о создании компьютерной системы. Три десятилетия спустя история по-прежнему захватывает, а изложенные секреты управления проектами по-прежнему актуальны.

Pedroni V., *Circuit Design and Simulation with VHDL, 2nd ed.*, MIT Press, 2010.
Справочное пособие показывает, как создавать схемы на языке VHDL.

Ciletti M., *Advanced Digital Design with the Verilog HDL, 2nd ed.*, Prentice Hall, 2010.
Хорошее справочное пособие по Verilog 2005 (но не по SystemVerilog)

SystemVerilog IEEE Standard (IEEE STD 1800).
IEEE стандарт по SysemVerilog HDL; последнее обновление в 2009. Доступен по ссылке: *ieeexplore.ieee.org*.

VHDL IEEE Standard (IEEE STD 1076).
IEEE стандарт по VHDL; последнее обновление в 2008. Доступен по ссылке: *ieeexplore.ieee.org.*

Wakerly J., *Digital Design: Principles and Practices, 4th ed.,* Prentice Hall, 2006.
Всеобъемлющее и хорошо написанное руководство про цифровой схемотехнике и отличный справочник.

Weste N., and Harris D., *CMOS VLSI Design, 4th ed.*, Addison-Wesley, 2011.
Разработка СБИС (VLSI) это наука и искусство по построению микросхем, содержащих кучу транзисторов. Эта книга, написанная в соавторстве с одним из наших любимых авторов, охватывает область разработки СБИС от начала и до самых передовых приемов, используемых в коммерческих продуктах.

Дальнейшая информация

Вы можете скачать руководство для преподавателей и дополнительные материалы к этому учебному пособию по ссылке:

http://textbooks.elsevier.com/web/product_details.aspx?isbn=9780123944245&Search=9780123944245&SearchCriteria=Keyword

Чтобы узнать больше об Университетской программе Imagination Technologies, посетите сайт:

www.imgtec.com/university

www.imgtec.com