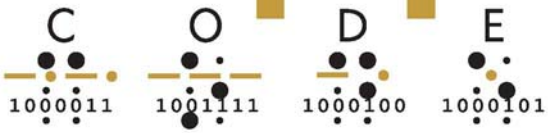
тайный язык информатики

КОД



Чарльз Петцольд

РУССКАЯ РЕДАКЦИЯ
Microsoft Press

The Hidden Language of
Computer Hardware and Software

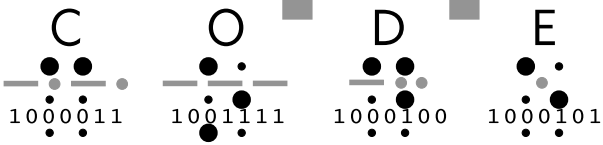
C O D E

Charles Petzold

Microsoft® Press

тайный язык информатики

КОД



Чарльз Петцольд

Москва 2001 г.

РУССКАЯ РЕДАКЦИЯ

УДК 004
ББК 32.973.26–018
П33

Петцольд Ч.

П33 Код. — М.: Издательско-торговый дом «Русская Редакция», 2001. — 512 с.: ил.

ISBN 978-5–7502–0159–4

Эта книга — азбука компьютерных технологий. Шаг за шагом автор знакомит читателя с сущностью кодирования информации, рассказывает об истории возникновения компьютеров, на практических примерах помогает освоить основные концепции информационных технологий, подробно излагает принципы работы процессора и других устройств компьютера.

Написанная живо, доступно, иногда иронично, книга богато иллюстрирована, состоит из 25 глав и предметного указателя.

Издание адресовано в первую очередь студентам вузов (как гуманитарных, так и технических), а также всем, кто интересуется принципами создания и работы компьютеров.

УДК 004
ББК 32.973.26–018

Оглавление

Предисловие

Идея книги *Код* стучалась мне в голову в течение десятилетия, прежде чем я начал ее писать. И все то время, что я обдумывал книгу, писал и даже после того, как она была опубликована, люди спрашивали меня: «О чем эта книга?»

На этот вопрос я всегда отвечал с неохотой. Мямлил что-то об «уникальном путешествии по истории цифровых технологий, определивших облик нашего времени», надеясь, что этого будет достаточно.

Но в конце концов мне пришлось признать: «*Код* — книга о том, как работают компьютеры».

Как я и опасался, реакция оказалась неблагоприятной. «А-а, такая книга у меня уже есть», — говорили некоторые, а я сразу возражал: «Нет, нет, нет, вот как раз такой книги у вас и нет». И до сих пор считаю, что это верно. *Код*— это не просто описание действия компьютера. В ней нет больших цветных изображений жестких дисков со стрелками, указывающими направление перемещения данных в компьютере. Нет рисунков, на которых локомотивы тащили бы вагоны, груженные нулями и единицами. Метафоры и сравнения, с одной стороны, украшают текст, а с другой — мешают постичь красоту технологии.

Приходилось мне слышать и такое: «Народу нет дела до того, как работают компьютеры». И это, по-моему, тоже верно. Мне, например, доставляет удовольствие узнавать, как работает тот или иной прибор. Но при этом я предпочитаю выбирать, в чем я хочу разобраться, а в чем — нет. Мне, скажем, трудно без насилия над собой объяснить принцип действия холодильника.

И все же люди часто задают вопросы, которые подтверждают их интерес к внутреннему устройству компьютера. Один из них: «В чем разница между оперативной и постоянной памятью?»

Это, без сомнения, очень важный вопрос. На подобных понятиях основан весь рынок персональных компьютеров. Предполагается, что даже новичок понимает, сколько *мегов* одной и *гигов* другой потребуется его программам. Считается также, что в самом начале знакомства с компьютером пользователь постигает концепцию файла и представляет себе, как файлы загружаются из постоянной в оперативную память, а затем из оперативной записываются обратно в постоянную.

На вопрос о постоянстве и оперативности часто отвечают, используя такую аналогию: «Считайте, что оперативная память — это ваш рабочий стол, а постоянная — шкаф с папками». И думают, что это прекрасное сравнение! Но я его таковым не считаю: создается впечатление, что архитектура компьютера срисована с обычной конторы. Правда же заключается в том, что различие между постоянной и оперативной памятью искусственно и существует лишь потому, что нам до сих пор не удалось создать накопитель данных, который был бы одновременно быстрым, объемным и способным хранить информацию в течение долгого времени. То, что сегодня называется архитектурой Неймана — доминирующая компьютерная архитектура в течение последних 50 лет, — прямое следствие этого технического несовершенства.

Еще меня как-то спросили: «Почему программы для Macintosh нельзя запускать под Windows?» Я уже было открыл рот, чтобы ответить, но вдруг осознал, что ответ будет содержать гораздо больше технических подробностей, чем рассчитывает услышать спросивший.

Я хочу, чтобы книга *Код* помогла вам разобраться в подобных вещах, причем не абстрактно, а глубоко, практически на уровне инженера или программиста. Надеюсь также, что в ней компьютер предстанет перед вами как одно из наиболее выдающихся изобретений XX века и вы поймете, что он прекрасен сам по себе, даже без изысканных метафор и сравнений.

Компьютер представляет собой иерархию, в основании которой лежат транзисторы, а на вершине — информация, отображаемая на мониторе. Переходы с одного уровня этой иерархии на следующий — а книга *Код* построена именно так — далеко не так тяжелы, как думает большинство. Конечно, действие современного компьютера состоит из бесчисленных

операций, но среди них множество простых или сходных между собой.

Компьютеры наших дней сложнее тех, что появились 25 или 50 лет назад, но в основе своей они остались теми же. Вот почему изучать историю технологии так удобно: чем дальше вы уходите в прошлое, тем проще становится технология. Рано или поздно вы достигаете этапа, разобраться в котором уже не представляет особого труда.

В книге *Код* я ушел настолько далеко в прошлое, насколько это было возможно. К своему удивлению, я обнаружил, что точка отсчета лежит в XIX столетии и я могу использовать для описания работы компьютера принципы устройства телеграфного оборудования. Все, что описано в первых 17-ти главах книги, можно построить, хотя бы теоретически, из простых электрических устройств, известных уже более 100 лет.

Мне кажется, что обращение к античным технологиям придает этой книге этакий ностальгический аромат. Она не могла бы называться *Быстрее и еще быстрее* или, скажем, *Бизнес со скоростью нервного импульса*. Понятие «бит» определено в ней лишь на 76-й странице, слово «байт» не встречается до 215-й. Транзисторы не упоминаются до 167-й страницы, да и там — лишь бегло.

Вот почему читать эту книгу довольно легко, хотя работа компьютера разбирается в ней с глубочайшими подробностями (много ли вы знаете книг, в которых действительно описывалась бы суть работы процессора?). Я старался, чтобы глубина изложения не сделала путешествие менее комфортабельным.

Но без локомотивов, которые тащат вагоны, груженные нулями и единицами.

Чарльз Петцольд
16 августа 2000 г.

code *n*

1. 1) кодекс, свод законов 2) законы, принципы *(чести, морали и т. п.)*

2. 1) код; Morse ~ код /азбука/ Морзе; telegraphic ~ телеграфный код 2) шифр

3. *биол.* генетический код

4. *вчт.* программа (*особ. прикладная*)

5. *ком.* маркировка; шифр. индекс (*продукта*)…

Новый большой англо-русский словарь (под общим руководством академика Ю.Д.Апресяна и доктора филологических наук, профессора Э.М. Медниковой). Издание 2-е, исправленное. М.: «Русский язык», 1997.

Глава 1

Верные друзья

Представьте себе, что вам 10 лет. Ваш лучший друг живет в доме напротив, так что окна ваших комнат смотрят друг на друга. Вечером, когда родители, как всегда, не вовремя отправили вас в кровать, вам хочется посекретничать, обменяться мыслями, наблюдениями, новостями, мечтами или рассказать свежий анекдот. В этом нет ничего зазорного. В конце концов тяга к общению — одна из самых человеческих черт.

Как же вам пообщаться? Может, по телефону? Но часто ли у 10-летнего ребенка в комнате стоит телефон? А если и есть, то разговор услышат в другой комнате. Компьютер, подключенный к телефонной линии, позволяет обмениваться сообщениями бесшумно, но и он тоже стоит не в вашей комнате.

Что у вас обоих есть *наверняка*, так это карманные фонарики. Всем известно: фонарики изобретены для того, чтобы дети могли читать книжки под одеялом. Кроме того, это прекрасное средство для общения в темноте. Фонарик достаточно бесшумен, а его луч имеет определенное направление и не просачивается под дверь спальни, так что о разговоре с помощью фонарика ваше бдительное семейство не догадается.

Но можно ли заставить фонарик говорить? Давайте попробуем. В первом классе вы научились выводить на бумаге буквы и слова, теперь пришла пора поделиться этим знанием с фонариком. Все, что для этого нужно, — подойти к окну и рисовать буквы лучом света. Чтобы «написать» О, включите

фонарик, обведите в воздухе круг, а затем выключите его. Буква Г рисуется так — включите фонарик и проведите им вверх и вбок. Впрочем, легко убедиться, что этот способ для общения не годится. Увидев, какие кренделя выписывает в воздухе луч фонарика вашего друга, вы поймете, что разобрать в мелькании света отдельные штрихи и буквы невозможно. Все эти световые росчерки недостаточно *определенны*.

Вы наверняка видели в каком-нибудь фильме, как моряки на кораблях посылают друг другу сигналы с помощью мигающих огней. А в другой картине шпион, покачивая зеркальцем, пускал солнечные зайчики в окно дома, где лежал связанным его коллега. Может быть, это выход? Начнем с самого простого, что приходит на ум. Обозначим каждую букву алфавита определенным числом вспышек: А будем обозначать одной вспышкой, Б — двумя, В — тремя и так до Я (33 вспышки). Например, чтобы передать слово «бег», нужно мигнуть фонариком 2, 6 и 4 раза с небольшими паузами между буквами, чтобы друг не принял их за одну букву К (12 вспышек). Между словами паузы нужно еще увеличить.

Это уже кое-что. По крайней мере не нужно размахивать фонариком: направляй луч в нужную сторону и щелкай кнопкой. Правда, у этого способа есть и недостаток — чтобы передать обычный для начала разговора вопрос «Как жизнь?», понадобится целых 95 вспышек, даже если забыть о знаке вопроса, для которого тоже нужно задать число вспышек.

Но кажется, решение где-то рядом. Наверняка кто-то уже ломал голову над подобной проблемой, думаете вы, и вы абсолютно правы. Наступает утро, вы идете в библиотеку и находите там книги об удивительном изобретении, которое называют азбукой Морзе. Это *в точности* то, что вам нужно, хотя теперь придется заново учиться писать все буквы.

Вот в чем отличие азбуки Морзе от вашей системы. Вы поставили каждой букве в соответствие определенное число одинаковых вспышек (от 1 для А до 33 для Я). В азбуке Морзе используются вспышки двух типов — короткие и длинные. Это, конечно, усложняет ее, но с другой стороны делает куда более эффективной. С помощью азбуки Морзе фраза «Как жизнь?» кодируется всего 30 вспышками (как короткими, так и длинными), а не 95, причем с учетом знака вопроса.

Говоря об азбуке Морзе, обычно говорят не о коротких и длинных вспышках, а о точках и тире, поскольку с их помощью код удобно изображать на бумаге. Каждой букве алфавита соответствует собственная последовательность точек и тире.

А	A	·–	К	K	–·–	Ф	F	··–·
Б	B	–···	Л	L	·–··	Х	H	····
В	W	·––	М	M	––	Ц	C	–·–·
Г	G	––·	Н	N	–·	Ч		––– ·
Д	D	–··	О	O	–––	Ш		––––
Е	E	·	П	P	·––·	Щ	Q	––·–
Ж	V	···–	Р	R	·–·	Ъ,Ь	X	–··–
З	Z	––··	С	S	···	Ы	Y	–·––
И	I	··	Т	T	–	Ю		··––
Й	J	·–––	У	U	··–	Я		·–·–

Хотя азбука Морзе не имеет к компьютерам абсолютно никакого отношения, тесное знакомство с концепцией кодирования необходимо для постижения тайного языка и внутренней сути аппаратного и программного воплощений компьютера.

В этой книге слово *код* обычно обозначает некую систему для обмена информацией между людьми и машинами. Иначе говоря, код — это средство коммуникации. И хотя коды иногда ассоциируются с какой-то тайной, чаще их используют для других целей. В определенном смысле коды вообще составляют основу любого человеческого общения.

В романе «Сто лет одиночества» Габриэль Гарсия Маркес вспоминает о временах, когда «мир был так юн, что вещи еще не обрели свои имена, и чтобы как-то обозначить их, нужно было указать на них пальцем». Обычно имена, которые мы даем вещам, кажутся произвольными. Непонятно, например, почему кошку не назвали собакой, а собаку — кошкой. Можно сказать, что обычный толковый словарь — это разновидность кода.

Звуки, издаваемые при произнесении слова, — это код, понятный любому, кто способен услышать наш голос и понима-

ет язык, на котором мы говорим. Мы называем этот код речью. Для слов, изображаемых на бумаге (или на камне, или на дереве), существует другой код — в виде написанных или напечатанных символов. Мы называем этот код письмом или текстом. Во многих языках между речью и письмом имеется строгое соответствие. В европейских языках, например, буквам или группам букв отвечают (в той или иной степени) определенные звуки.

Для глухонемых разработан особый код — язык знаков, в котором движения рук обозначают буквы, целые слова и понятия. Для слепых письмо заменяется азбукой Брайля, в которой буквы, группы букв и целые слова закодированы наборами выпуклых точек. Когда устную речь нужно очень быстро перевести в текст, применяют особый код — стенографию (скоропись).

Для общения люди используют множество различных кодов, потому что одни коды иногда удобнее других. Например, устную речь нельзя сохранить на бумаге, поэтому вместо нее мы применяем письмо. Беззвучный обмен информацией на расстоянии и в темноте посредством речи или письма невозможен, поэтому в ряде случаев удобной альтернативой им оказывается азбука Морзе. Код полезен, если выполняет задачу, с которой в данных обстоятельствах не справляются другие коды.

Как мы увидим ниже, в компьютерах также применяются различные типы кодов для хранения и передачи чисел, звуков, музыки, изображений и фильмов. Компьютер не может иметь дело с человеческими кодами напрямую — ведь у него нет глаз, ушей, рта и пальцев, столь необходимых людям при общении. Правда, в последнее время появилась тенденция учить персональные компьютеры воспринимать, хранить, обрабатывать и воспроизводить все типы информации, применяемые в человеческом общении: видимую (текст и изображения), слышимую (устная речь, звуки и музыка) или их сочетание (мультипликация и кино). Каждый из этих видов информации требует собственного кода подобно тому, как человеку нужны одни органы для речи (рот и уши) и другие (руки и глаза) — для письма.

Даже таблица с расшифровкой азбуки Морзе сама по себе — тоже своеобразный код. Из нее видно, что каждая буква представлена набором точек и тире, однако в реальности мы не можем посылать точки и тире, а потому заменяем их вспышками.

Чтобы передать точку азбуки Морзе с помощью фонарика, нужно очень быстро включить и выключить его (короткая вспышка). Для передачи тире фонарик остается включенным немного дольше (долгая вспышка). Например, чтобы передать букву А, надо включить фонарик и тут же выключить, а затем включить опять и выключить уже с меньшей поспешностью. Перед следующим символом нужно сделать паузу. Традиционно тире длится примерно в три раза дольше точки. Если, скажем, для передачи точки фонарик горит секунду, то для передачи тире — три (в действительности, сообщения «морзянкой» передаются гораздо быстрее). Получатель, видя короткую вспышку, а за ней длинную, понимает, что это буква А.

Паузы между точками и тире в азбуке Морзе имеют большое значение. При передаче буквы А между точкой и тире фонарик следует выключить на время, примерно равное длительности точки (если точка длится секунду, то промежуток между точками и тире также составляет одну секунду). Буквы одного слова разделяются более длинными паузами, равными примерно одному тире (например, трем секундам, если такова длительность тире). В следующем примере показано слово «привет» с надлежащими паузами между буквами.

Между словами фонарик остается выключенным на время примерно двух тире (6 секунд, если тире длится 3 секунды). Вот код для фразы «как жизнь».

Продолжительность пауз между буквами и словами жестко не зафиксирована. Все определяется только длительностью точки, а она зависит от того, как быстро вы способны нажимать кнопку фонарика, а еще от того, как долго отправитель вспоминает код очередной буквы в азбуке Морзе. Тире опытного «морзянщика» может длиться как точка медлительного. Это, конечно, может затруднить чтение сообщения на азбуке Морзе, однако после одной-двух букв получатель обычно приноравливается различать точки и тире.

Поначалу определение азбуки Морзе — а *определением* я называю соответствие между последовательностями точек и тире и буквами алфавита — кажется столь же произвольным,

как расположение букв на клавиатуре пишущей машинки. Однако при ближайшем рассмотрении это оказывается совсем не так. Самые простые и короткие коды присвоены часто употребляемым буквам, например, Е и Т, а у менее популярных букв, таких как Ш или Щ, коды подлиннее.

Есть некий минимум азбуки Морзе, известный почти каждому человеку: 3 точки, 3 тире, 3 точки представляют собой международный сигнал бедствия — SOS. Это не сокращение, а просто легко запоминающаяся последовательность кодов. Еще один пример из жизни. Во время второй мировой войны радио BBC начинало некоторые передачи вступлением к Пятой симфонии Бетховена — БА, БА, БА, БА-А-АММ. Композитор и не подозревал, что начинает симфонию кодом Морзе для буквы V (Victory, т. е. победа).

Недостаток азбуки Морзе в том, что в ней не различаются прописные и строчные буквы. Зато в ней нашлось место цифрам, зашифрованным наборами из пяти точек и тире.

1	•▬▬▬▬	6	▬····
2	••▬▬▬	7	▬▬···
3	•••▬▬	8	▬▬▬··
4	••••▬	9	▬▬▬▬·
5	•••••	0	▬▬▬▬▬

В отличие от букв в этих кодах есть порядок. Знаки препинания зашифрованы пятью, шестью или даже семью точками и тире.

.	•▬•▬•▬	'	•▬▬▬▬·
,	▬▬••▬▬	(	▬•▬▬•
?	••▬▬••	)	▬•▬▬•▬
:	▬▬▬···	=	▬···▬
;	▬•▬•▬•	+	•▬•▬•
-	▬···▬	$	···▬••▬
/	▬••▬•	¶	•▬•▬••
"	•▬••▬•	_	••▬▬•▬

Есть еще специальные коды для букв с диакритическими знаками (вроде знака ударения) из некоторых европейских языков и несколько служебных последовательностей (начало передачи, готовность к приему, ошибка и пр.). К последним относится и код SOS — его передают без обычных пауз между буквами.

Со временем вы обнаружите, что гораздо легче посылать «морзянку» с помощью специально приспособленного фонарика. В нем, кроме обычного выключателя, есть кнопка с пружиной. Фонарик горит, пока она нажата. Немного попрактиковавшись, можно добиться скорости передачи 5–10 слов в минуту — это, конечно, медленнее речи (мы произносим примерно 100 слов в минуту), но вполне приемлемо.

Когда вы с другом вызубрите азбуку Морзе (а зубрежка — единственный способ стать специалистом по отправке и приему «морзянки»), то сможете даже говорить азбукой Морзе, произнося вместо точки *то*, а вместо тире — *ти*. Письмо сводится азбукой Морзе к двум символам (точке и тире), а устную речь с ее помощью можно сократить до двух слогов.

Ключевое слово здесь — *два*. Два вида вспышек, два слога, два чего угодно способны в соответствующих комбинациях передать любую информацию.

Глава 2

Коды и комбинации

Азбуку Морзе придумал Сэмюэль Финли Бриз Морзе (Samuel Finley Breese Morse) (1791–1872), с которым мы на страницах этой книги встретимся еще не раз. Изобретение азбуки Морзе неразрывно связано с изобретением телеграфа, с которым мы тоже познакомимся. Подобно тому, как знакомство с азбукой Морзе помогает постичь сущность кодирования, изучение телеграфа позволяет получить начальное представление о компьютерном оборудовании.

Для большинства людей передать сообщение на азбуке Морзе гораздо проще, чем принять его. Даже если вы не выучили азбуку наизусть, всегда можно положить перед собой таблицу, в которой все буквы расставлены по алфавиту.

А	A	·–	К	K	–·–	Ф	F	··–·
Б	B	–···	Л	L	·–··	Х	H	····
В	W	·––	М	M	––	Ц	C	–·–·
Г	G	––·	Н	N	–·	Ч		–––
Д	D	–··	О	O	–––	Ш		–––––
Е	E	·	П	P	·––·	Щ	Q	––·–
Ж	V	···–	Р	R	·–·	Ъ,Ь	X	–··–
З	Z	––··	С	S	···	Ы	Y	–·––
И	I	··	Т	T	–	Ю		··––
Й	J	·–––	У	U	··–	Я		·–·–

На прием «морзянки» и ее перевод в слова уходит гораздо больше усилий и времени, чем на передачу. Это связано с тем, что в этом случае работать приходится «против шерсти» — искать букву, соответствующую последовательности точек и тире. Например, чтобы расшифровать последовательность «тире-точка-тире-тире», вам придется просмотреть буква за буквой почти всю таблицу — ведь это Ы!

Беда в том, что у нас есть таблица для перевода в направлении:

буква алфавита → точки и тире азбуки Морзе

но *нет* таблицы, позволяющей выполнить обратное преобразование:

точки и тире азбуки Морзе → буква алфавита.

А ведь при первом знакомстве с азбукой Морзе такая таблица была бы очень кстати. Увы, принципы ее построения уловить ой как непросто. В наборах точек и тире нет ничего такого, что можно было расставить в алфавитном порядке.

Забудем пока об алфавите. Попробуем сгруппировать коды в зависимости от количества точек и тире. Сначала поставим коды, состоящие из единственного значка, — это буквы Е (точка) и Т (тире):

·	Е
–	Т

Сочетания двух точек и (или) тире дают четыре буквы — И, А, Н и М.

··	И	–·	Н
·–	А	––	М

Из сочетаний трех точек и тире получается уже 8 букв:

···	С	–··	Д
··–	У	–·–	К
·–·	Р	––·	Г
·––	В	–––	О

Наконец, цепочки из четырех точек и тире дадут нам еще 16 символов (забудем пока о цифрах и знаках препинания):

••••	Х	—•••	Б
•••—	Ж	—••—	Ь
••—•	Ф	—•—•	Ц
••——	Ю	—•——	Ы
•—••	Л	——••	З
•—•—	Я	——•—	Щ
•——•	П	———•	Ч
•———	Й	————	Ш

Что получилось? Во всех четырех таблицах содержится 2 + 4 + + 8 + 16 = 30 кодов для букв, т. е. на 4 больше, чем необходимо для латинского алфавита, в котором 26 букв, и на 2 меньше, чем для русского (в азбуке Морзе твердый и мягкий знаки не различаются).

Используя эти четыре таблицы, расшифровать сообщение, написанное «морзянкой», гораздо легче. Подсчитав число точек и тире в коде буквы, вы сразу возьмете нужную таблицу. Таблицы составлены так, что код, состоящий только из точек, находится в левом верхнем углу, а состоящий только из тире — в правом нижнем.

Нетрудно заметить закономерность и в *размерах* таблиц — каждая следующая таблица вдвое больше предыдущей. Это и понятно: в каждую таблицу включены все коды из предыдущей таблицы с дополнительной точкой и те же коды с дополнительным тире.

Подведем итог.

Число точек и тире	**Число кодов**
1	2
2	4
3	8
4	16

От таблицы к таблице число кодов удваивается: в первой 2 кода, во второй 2 × 2, в третьей — 2 × 2 × 2, т. е.:

Число точек и тире	Число кодов
1	2
2	2 × 2
3	2 × 2 × 2
4	2 × 2 × 2 × 2

Раз уж мы начали умножать числа сами на себя, можно записать эти примеры в виде возведения в степень. Например, 2 × 2 × 2 × 2 есть просто *2 в четвертой степени* — 2^4. Числа 2, 4, 8 и 16 — это степени числа 2, поэтому перепишем таблицу так:

Число точек и тире	Число кодов
1	2^1
2	2^2
3	2^3
4	2^4

Таблица здорово упростилась. Число доступных кодов есть 2 в степени, равной числу точек и тире:

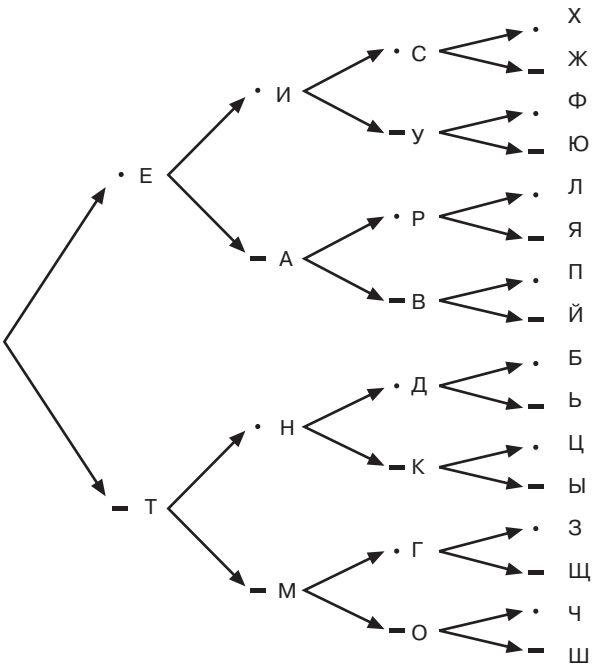
$$\text{Количество кодов} = 2^{\text{Количество точек и тире}}$$

Степени числа 2 часто встречаются в кодировании, и еще одно подтверждение этой закономерности мы встретим в следующей главе.

Для облегчения расшифровки азбуки Морзе нарисуем древовидную диаграмму (см. стр. 13). На ней показано, как найти букву, соответствующую заданной последовательности точек и тире. Для расшифровки кода нужно идти по направлению, указанному стрелками. Пусть нужно определить, какая буква соответствует коду «точка-тире-точка». Начнем с крайней левой точки: перемещаясь по стрелке вправо, переходим к тире, а затем — к точке. Итак, это буква Р, показанная справа от последней точки.

Составитель азбуки Морзе вряд ли обошелся без подобной схемы. Во-первых, она гарантирует, что один и тот же код не будет использован для двух различных символов! Во-вторых,

она наглядно показывает, все ли возможные комбинации точек и тире уже использованы, что позволяет избежать введения излишне длинных последовательностей.



Рискуя выйти за пределы страницы, мы можем дополнить схему кодами, состоящими из большего числа точек и тире. Последовательность из пяти точек и тире дает 32 дополнительных кода ($2 \times 2 \times 2 \times 2 \times 2 = 2^5$). Этого вполне хватит для 10 цифр и 16 основных знаков препинания, и цифры в азбуке Морзе действительно кодируются пятью точками и тире.

Чтобы включить все знаки препинания, схему нужно расширить до шести точек и тире, добавив в общий набор символов еще 64 ($2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^6$) дополнительных кода, увеличив общее число кодов до $2 + 4 + 8 + 16 + 32 + 64 = 126$. Для азбуки Морзе это уже явный перебор, и действительно, многие из длинных последовательностей остались в ней «неопределенными». В данном контексте выражение «*неопределенный код*» означает, что коду не соответствует никакой символ.

Получив в послании на азбуке Морзе неопределенный код, не сомневайтесь — отправитель допустил ошибку.

Раз уж мы блеснули математическими познаниями и вывели формулу:

$$\text{Количество кодов} = 2^{\text{Количество точек и тире}}$$

нам ничего не стоит подсчитать число кодов для еще более длинных последовательностей точек и тире:

Число точек и тире	Число кодов
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

Благодаря этой формуле, нам не нужно выписывать все возможные коды, чтобы подсчитать их количество. Достаточно нужное число раз умножить число 2 само на себя.

Азбука Морзе называется *двоичным* (binary) кодом, поскольку элементов кода всего два: точка и тире. Этим он похож на монету, которая может выпасть либо орлом, либо решкой. При описании двоичных объектов или кодов (подобных азбуке Морзе) часто прибегают к степеням числа 2.

Приведенный выше подсчет числа двоичных кодов — простое упражнение из раздела математики, который называется *комбинаторикой* или *комбинаторным анализом*. Обычно комбинаторный анализ применяют в теории вероятностей, поскольку он позволяет вычислить вероятность выпадения определенной стороны монеты или числа на игральных костях. Но он полезен и при разработке кодов и шифров.

Глава 3

Брайль и двоичные коды

Сэмюэль Морзе не первым изобрел кодирование букв и не стал первым человеком, чье имя воспринимается скорее как название азбуки, а не как собственно имя. Этой чести удостоился слепой французский подросток, появившийся на свет 18 годами позже Морзе. О его жизни известно немногое, но и это немногое заслуживает внимания.

Луи Брайль (Louis Braille) родился в 1809 г. во Франции, в местечке Кувре, в 25 милях восточнее Парижа. Его отец был шорником. Когда Луи было три года — в этом возрасте малышу еще не следовало играть в отцовской мастерской — он случайно уколол глаз шилом. В рану попала инфекция, заражение распространилось на второй глаз, и мальчик полностью ослеп. Казалось, он обречен остаться неграмотным и влачить нищенское существование (обычная участь слепого в те времена). Однако у мальчика обнаружился живой ум и страстное желание учиться. Благодаря поддержке деревенского священника и школьного учителя он поначалу посещал вмес-

те с другими детьми сельскую школу, а в 10 лет его отправили в Королевский институт для слепых в Париже.

Одно из главных препятствий в обучении слепого человека — его неспособность читать. Основатель парижского института Валентен Ойи (Valentin Haüy) (1745–1822) придумал систему рельефных букв, которые можно было читать, касаясь пальцами. Однако на практике с этой системой было очень трудно работать, и потому книги, напечатанные таким способом, успехом не пользовались.

Ойи не был слепым и оказался заложником штампов. Он, например, считал, что буква А — это А и ничто другое и потому должна походить на А даже на ощупь. Это напоминает наши попытки рисовать фонариком буквы в воздухе, но мы быстро поняли бесперспективность этого способа общения. Очевидно, Ойи не пришло в голову, что для слепых более удобным может оказаться код, существенно отличающийся от печатных букв.

Идея такого кода пришла с неожиданной стороны. Капитан французской армии Шарль Барбье (Charles Barbier) разработал в 1819 г. систему кодирования, которую назвал *écriture nocturne*, т. е. «ночное письмо». В этой системе применялись выпуклые точки и тире на плотной бумаге, и предлагалась она солдатам как средство беззвучной ночной связи в полевых условиях. Солдаты должны были продавливать точки и тире с обратной стороны бумаги пером, похожим на шило. Получатель письма читал эти выпуклости, водя по ним кончиками пальцев.

Недостатком системы Барбье была ее сложность. Барбье решил кодировать с помощью сочетаний точек и тире не буквы, а звуки, и потому для шифрования некоторых слов требовалось очень много кодов. Система оправдывала себя при передаче коротких сообщений, но была совершенно непригодна для длинных текстов, не говоря уже о книгах.

Луи Брайль познакомился с системой Барбье в возрасте 12 лет. Система рельефных точек пришлась ему по душе, так как позволяла не только легко *читать* с помощью пальцев, но и *писать*. Ученик, вооружившись картоном и специальным пером, делал в классе заметки, которые мог затем прочитать дома. Совершенствуя в течение трех лет эту систему, Луи Брайль разработал собственный шрифт (ему тогда было 15 лет), основы которого используются и поныне. Многие годы его система применялась только в парижском институте, но посте-

пенно распространилась более широко. Умер Луи Брайль в 1852 г. от туберкулеза в возрасте 43 лет.

В наши дни у людей, лишенных зрения, появилась возможность знакомиться с письменными источниками с помощью магнитофона. Но для слепоглухонемых шрифт Брайля и сегодня остается бесценным, единственным способом чтения. В последние годы о шрифте Брайля узнали многие, так как надписи на нем стали делать в лифтах и банкоматах, чтобы ими могли пользоваться слепые.

В этой главе мы рассмотрим принципы построения азбуки Брайля. *Учить* ее нам не потребуется. Мы просто попытаемся с помощью шрифта Брайля глубже проникнуть в природу кодирования.

В шрифте Брайля символы письменного языка — буквы, цифры и знаки препинания — кодируются комбинациями от одной до шести выпуклых точек, расположенных в ячейке размерами 2 × 3. Точки в ячейке нумеруются с 1 по 6:

1 ○ ○ 4
2 ○ ○ 5
3 ○ ○ 6

Для выдавливания точек используются специальные пишущие машинки и станки.

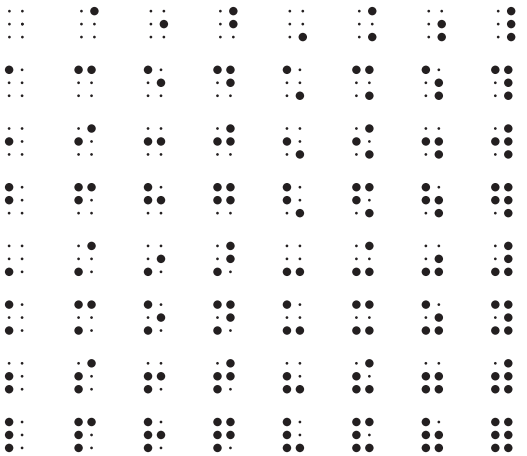
Я, конечно, мог бы включить в эту книгу пару страниц с рельефными буквами, но из-за этого она стала бы неоправданно дорогой. Чтобы не прибегать к тиснению, я воспользуюсь системой, применяемой для отображения символов Брайля на страницах обычных книг. В каждой ячейке я буду показывать все шесть точек: крупные точки будут соответствовать выпуклым точкам, а мелкие — плоским. Например, в этом символе

точки 1, 3 и 5 — выпуклые, а точки 2, 4 и 6 — плоские.

Для нас самое интересное в шрифте Брайля то, что он является *двоичным*. Любая точка может пребывать в одном из двух состояний: плоская или выпуклая. Это значит, что к шрифту Брайля применимы наши познания об азбуке Морзе

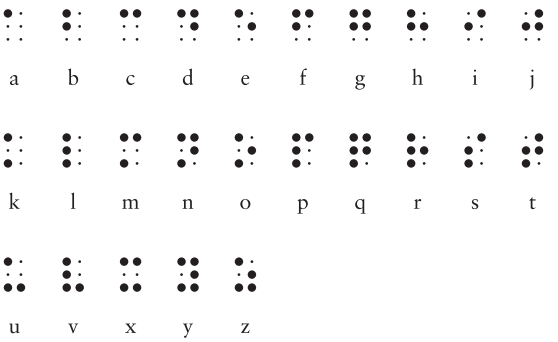
и комбинаторном анализе. Общее число комбинаций шести точек, каждая из которых может быть плоской или выпуклой, равно $2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^6 = 64$.

Следовательно, система Брайля содержит 64 различных кода. Вот как они выглядят:



Если окажется, что в шрифте Брайля меньше 64 кодов, мы зададимся вопросом, почему используются не все возможности. Если кодов окажется *больше* 64, сомнению подвергнутся либо наш рассудок, либо фундаментальные основы математики (а именно верность утверждения, что дважды два — четыре).

Анализ шрифта Брайля начнем со строчных букв латинского алфавита:



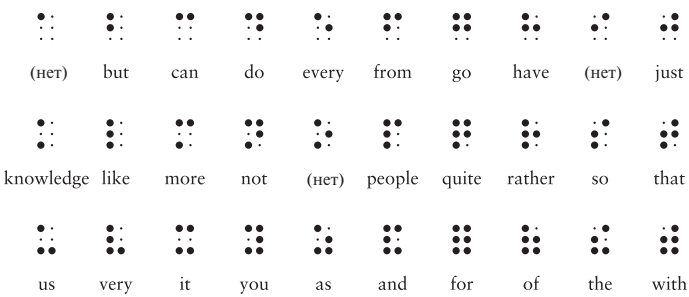
Например, фраза «you and me» кодируется так:

Заметьте: ячейки-буквы в пределах слова разделены небольшими интервалами, а между словами интервал побольше (размером в одну ячейку).

Эта основа разработана самим Луи Брайлем по крайней мере применительно к буквам латинского алфавита. Брайль ввел также коды для букв с диакритическими знаками, которых во французском языке немало. Кстати, обратите внимание на отсутствие буквы *w*, которой в классическом французском языке нет (не беспокойтесь, найдется код и для нее). Мы познакомились с 25 из 64 возможных кодов.

Рассмотрев показанные выше строчки, вы без труда обнаружите в них систему. В первой строке (от *a* до *j*) используются только четыре верхние точки — 1, 2, 4 и 5. Вторая строка является повторением первой, но с добавлением выпуклой точки 3, в третьей строке выпуклой стала и точка 6.

Со времен Брайля его код не раз дополнялся. Современный вариант этой системы, наиболее часто употребляемый в публикациях на английском языке, называется «Брайлем 2-й ступени» (Grade 2 Braille). В нем несколько нововведений, призванных сократить затраты бумаги и облегчить чтение. Например, если код отделен от своих соседей большим интервалом, он означает не букву, а одно из часто встречающихся слов. Ниже показана «словесная» расшифровка шрифта Брайля (заметьте: третья строка «завершена»):



Это значит, что фразу «you and me» можно записать компактнее:

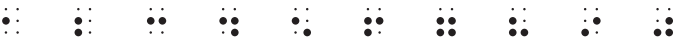
Пока что мы видели 31 код — промежуток между словами соответствует коду без единой выпуклой точки, и в дополнение к нему 3 строки по 10 кодов для букв и некоторых слов. До теоретического предела в 64 кодов все еще далеко. Но не сомневайтесь: в современном шрифте Брайля забытых кодов нет.

Во-первых, дополним коды букв от *a* до *j* выпуклой точкой 6. Эти коды используются в основном для обозначения частых сочетаний букв и забытой буквы «w»:

ch gh sh th wh ed er ou ow w (или "will")

Вот, например, как выглядит слово «about»:

Во-вторых, коды букв от *a* до *j* можно сдвинуть на один ряд точек вниз, задействовав только точки 2, 3, 5 и 6. Эти коды в зависимости от контекста применяются для некоторых знаков препинания или буквосочетаний:



ea bb cc dis en to gg his in was
, ; : . ! () " "

Первые четыре кода — запятая, точка с запятой, двоеточие и точка. Заметьте: открывающаяся и закрывающаяся скобки представлены одним и тем же кодом, а открывающая и закрывающая кавычки — разными.

Мы насчитали уже 51 код. В следующих 6 кодах сочетания точек 3, 4, 5 и 6 используются для представления буквосочетаний и знаков препинания:

st ing ble ar ' com
/ # -

Здесь особенно важен код «ble». Если он не является частью слова, значит, следующие за ним коды следует интерпретировать как цифры. Коды цифр полностью совпадают с кодами букв от *a* до *j*:

1 2 3 4 5 6 7 8 9 0

Следовательно, последовательность кодов

означает число 256.

Для достижения максимального количества нам осталось рассмотреть 7 кодов:

Первый (выпуклая точка 4) — признак знака ударения. Остальные применяются как префиксы в некоторых сокращениях или выполняют иные функции. Код, в котором выпуклы точки 4 и 6 (пятый в этой строке), в зависимости от контекста означает десятичный разделитель в числах или знак ударения. Код с выпуклыми точками 5 и 6 отменяет действие признака числа.

Наконец (вы наверняка мучаетесь вопросом, как в шрифте Брайля обозначаются прописные буквы), у нас остался код с выпуклой точкой 6. Это и есть признак прописной буквы, указывающий, что следующая за ним буква является прописной. Вот как записывается имя основоположника этой системы (Louis Braille):

Код состоит из признака прописной буквы, буквы l, сокращения для ou, букв i и s, пробела, второго признака прописной буквы и букв b, r, a, i, l, l и e (кстати, на практике последнее слово еще немного сокращают, убирая две последние непроизносимые буквы).

Итак, мы увидели, как из шести двоичных элементов (точек) получается 64 кода, ни больше, ни меньше. Правда, многие из кодов несут двойную нагрузку. Особо отметим код-при-

знак числа и отменяющий его код-признак буквы. Эти коды изменяют смысл последующих символов: буквы становятся цифрами и наоборот. Такие коды иногда называют кодами *переключения* (shift). Код переключения меняет смысл всех следующих кодов, пока его действие не будет отменено.

Признак прописной буквы означает, что следующая за ним буква (и только она) должна интерпретироваться как прописная, а не строчная. Код такого типа называется *escape*-кодом. Это название происходит от английского слова «escape» (убегать, вырываться): escape-код как бы позволяет «избежать» обыденного восприятия кода и взглянуть на него по-новому. Как мы увидим в других главах книги, к кодам переключения и escape-кодам часто прибегают при двоичном кодировании печатных символов.

Глава 4

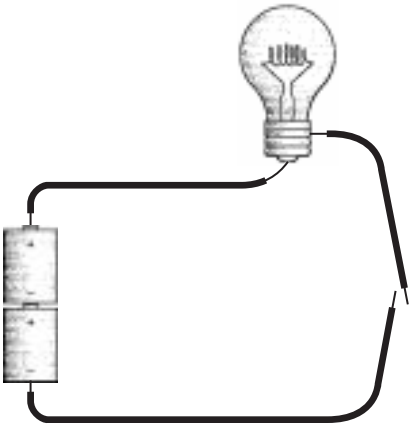
Анатомия фонарика

Фонарику можно найти массу применений, из которых чтение под одеялом или отправка зашифрованных сообщений — лишь наиболее очевидные. Скажем, обычный фонарик, который, наверное, есть в каждом доме, — отличное наглядное пособие для изучения электричества.

Электричество — удивительное явление. Мы не можем без него жить, оно окружает нас повсюду и все же во многом остается загадкой даже для людей, которым по роду деятельности положено разбираться в его сущности. Боюсь, что нам не обойтись без некоторого знакомства с этой тайной. К счастью, мы ограничимся лишь основными понятиями, без которых роль электричества в работе компьютеров будет неясна.

Фонарик — одно из простейших электрических устройств. Разобрав его, вы обнаружите, что он состоит из пары батареек, лампочки, выключателя, нескольких металлических деталей и пластмассового корпуса, в котором все это собрано.

Чтобы изготовить незатейливый фонарик самостоятельно, из этого набора можно оставить только батарейки и лампочку. Кроме того, понадобится несколько отрезков изолированного провода с оголенными концами, а чтобы все это удержать, хватит рук.



Два оголенных конца провода в правой части схемы между собой не соединены. Они играют роль переключателя. Коснитесь одним проводом другого, и лампочка сразу загорится. Конечно, если она не перегорела, а батарейки не сели.

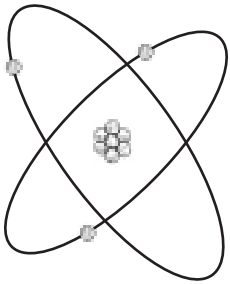
Собранная схема являет собой простейшую электрическую цепь, которая наглядно иллюстрирует важное правило: цепь должна быть *замкнутой*. Лампочка зажжется, только если на пути от батареек через провод, лампочку, переключатель и обратно к батарейкам нет разрывов. Стоит разомкнуть цепь, и лампочка тут же погаснет. Переключатель нужен для управления этим процессом.

Замкнутость электрической цепи предполагает, что по ней что-то движется, подобно тому как вода течет по трубам. В популярных объяснениях работы электричества аналогия с водопроводными трубами применяется довольно часто, хотя иногда она подводит (как, впрочем, все аналогии). Во Вселенной нет ничего, что можно было бы сравнить с электричеством, поэтому лучше обойтись без аналогий и говорить о нем на его языке.

Научные премудрости, объясняющие работу электричества, называются *электронной теорией*. Согласно ей, электричеством называется совокупность явлений, связанных с взаимодействием и движением заряженных частиц, например, электронов.

Как известно, материя, т. е. то, что нам дано видеть и ощущать (как правило), состоит из крохотных частичек, называе-

мых атомами. Атом состоит из элементарных частиц трех видов: нейтронов, протонов и электронов. Атомы часто изображают в виде маленьких планетных систем: в центре расположено ядро, состоящее из нейтронов и протонов, а вокруг него, подобно планетам вокруг Солнца, носятся электроны.



Нужно признать, что это не более чем очередная аналогия, и мощный микроскоп, позволяющий разглядеть атомы, показал бы другую картину. Однако нам годится и такая модель.

У атома на рисунке 3 электрона, 3 протона и 4 нейтрона. Это атом химического элемента лития, одного из ста с лишним известных *элементов*, каждый из которых обозначается собственным *атомным номером*, принимающим значения от 1 до 112. Атомный номер элемента равен числу протонов в его ядре. Атомный номер лития — 3.

Посредством химических реакций атомы соединяются друг с другом в *молекулы*. Обычно свойства молекулы существенно отличаются от свойств составляющих ее атомов. Так, молекула воды состоит из двух атомов водорода и одного атома кислорода. Отличие воды как от кислорода, так и от водорода очевидно. Аналогично молекула поваренной соли состоит из атома натрия и атома хлора, однако ни тот, ни другой по отдельности не способны улучшить вкусовые качества жареной картошки.

Обычно число электронов в атоме совпадает с числом протонов. Но при некоторых обстоятельствах электроны покидают свои атомы и порождают электрические явления.

Слова *электрон* и *электричество* происходят от древнегреческого слова *ηλεκτρον* (электрон), что в переводе значит «янтарь». Объясняется это просто. Полируя кусочки янтаря шер-

стью, древние греки познакомились с явлением, которые мы называем статическим электричеством. Шерсть, трущаяся о янтарь, срывает с поверхности янтаря электроны и скоро накапливает электронов больше, чем имеет протонов. Янтарь, напротив, остается с относительным избытком протонов.

Протоны и электроны характеризуются *зарядом*. У протонов он положительный, у электронов — отрицательный. Нейтроны нейтральны и заряда не имеют. Для обозначения протонов и электронов часто используют знаки «+» и «–», подчеркивая электрическую противоположность этих частиц.

Конфигурация, состоящая из протонов и электронов, наиболее устойчива, когда и те и другие находятся в ней в равных количествах. Нарушьте баланс протонов и электронов, и система постарается восстановить равновесие. Ковер при ходьбе захватывает электроны с подошв ваших туфель, но стоит вам коснуться какого-нибудь предмета, и вы ощутите пробежавшую искру. Так восстанавливается равновесие: искра — это разряд статического электричества, по замысловатому пути перемещающий электроны от ковра через ваше тело обратно к подошвам.

Статическое электричество — это не только искра, проскакивающая между пальцами и дверной ручкой. Во время грозы нижние слои туч накапливают электроны, а верхние их теряют. В конце концов, равновесие восстанавливается разрядом молнии. Молния — это множество электронов, с огромной скоростью перетекающих из одного места в другое.

Электричество в фонарике более управляемо, чем искра или молния. Свет фонарика ровен и непрерывен, так как электроны не просто скачут с места на место. Атом, потеряв электрон (перескочивший к другому атому), захватывает электрон у соседнего атома, тот в свою очередь отнимает электрон у своего соседа и т. д. Электричество в цепи — это перетекание электронов от атома к атому.

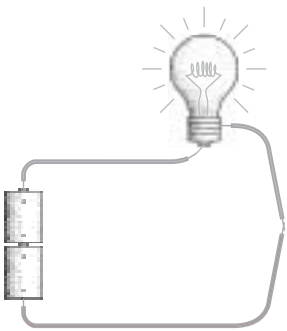
Естественно, происходит это не само собой. Не ждите появления электричества, просто соединив два куска провода. Нужно нечто, что вынудило бы электроны двигаться по цепи. Посмотрев еще раз на схему нашего нехитрого фонарика, мы смело предположим, что источник движения электронов заключен не в проводах и не в лампочке, а скорее всего в батарейках.

Самое важное о батарейках известно, наверное, всем.

- Чаще всего они изготавливаются в виде цилиндров различных размеров.
- Напряжение большинства батареек равно 1,5 вольта.
- Один торец батарейки плоский и помечен знаком «–», на другом есть небольшой выступ, и помечен он знаком «+». Полярность батареек нужно обязательно учитывать при их установке.
- Со временем батарейки разряжаются («садятся»).
- Наконец, батарейки как-то производят электричество.

Во всех батарейках протекают те или иные химические реакции, в ходе которых одни молекулы, разрушаясь или соединяясь друг с другом, превращаются в другие молекулы. Химический состав батареек подобран так, что в результате реакций между различными молекулами на том конце батарейки, что помечен знаком «–» (он называется отрицательным полюсом или *анодом*), образуется избыток электронов, а на противоположном — помеченном знаком «+» (положительном полюсе, или *катоде*) — возникает их недостаток.

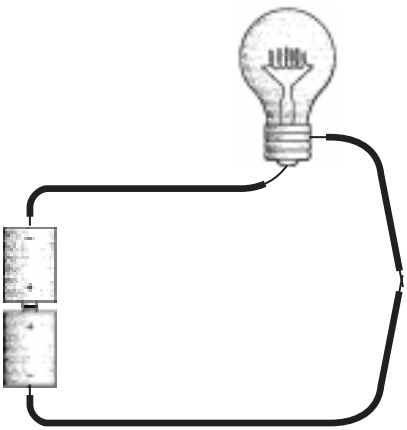
Сформировав избыток электронов на одном полюсе и их недостаток на другом, химические реакции останавливаются (точнее, протекают очень медленно), поэтому с неподключенной батарейкой ничего не происходит. Чтобы возобновить реакцию, избыточным электронам на отрицательном полюсе батарейки нужно обеспечить проход к ее положительному полюсу. Таким образом, реакция происходит только при наличии замкнутой электрической цепи. По нашей цепи электроны движутся против часовой стрелки:



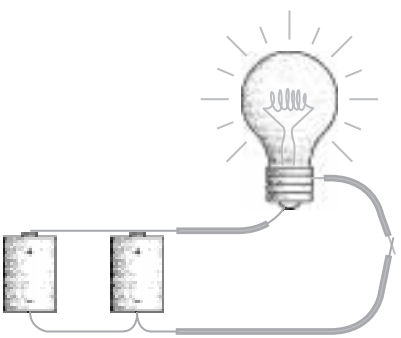
В этой книге провода, по которым течет ток, выделены светлым оттенком.

Заметьте: обе батарейки ориентированы одинаково. Положительный полюс нижней батарейки получает электроны из отрицательного полюса верхней. Две батарейки, соединенные таким образом (*последовательно*), ведут себя, как одна большая батарейка с напряжением 3 вольта вместо прежних полутора.

Если одну из батареек перевернуть, цепь перестанет работать:



Двум положительным полюсам батареек для химических реакций нужны электроны, но полюса обращены друг к другу, и путь, по которому электроны могли бы туда попасть, закрыт. Теперь попробуем соединить и отрицательные полюса:



Теперь цепь работает. Такое соединение батареек называется *параллельным*. Их общее напряжение равно 1,5 В, как и у каждой батарейки в отдельности. Лампочка горит не так ярко, как при последовательно соединенных батарейках, зато прослужат батарейки вдвое дольше прежнего.

Обычно батарейки рассматривают как источник электричества в цепи. Но мы теперь знаем, что для протекания химических реакций батарейке и самой нужны электроны. Цепь забирает электроны с анода батарейки и возвращает их на катод. Реакции в батарейке продолжатся, пока не истощатся движущие их химические вещества, после чего батарейки надо выбросить или перезарядить.

От анода к катоду электроны текут по проводам и через лампочку. А провода зачем? Почему нельзя пропустить электричество по воздуху? Потому, что некоторые вещества проводят электрический ток гораздо лучше других. Способность химического элемента проводить ток связана со строением его атома. Электроны вращаются вокруг ядра на разных уровнях, которые называются оболочками. Атом, у которого во внешней оболочке только один электрон, легко расстается с ним, обеспечивая прохождение тока. Вещества, способные проводить электричество, называют *проводниками*. Лучшие проводники — медь, серебро и золото. Не случайно все три этих элемента находятся в одном столбце (или, по-научному, в одной группе) периодической системы Д. И. Менделеева. Чаще всего электрические провода изготавливают из меди.

Вещества, не проводящие ток, называются *изоляторами*. Хорошие изоляторы — резина и пластмасса, поэтому их часто применяют в качестве покрытия проводов. В сухую погоду хорошими изоляторами являются также ткань и древесина. Но если напряжение очень велико, проводником становится практически любое вещество.

С другой стороны, даже проводники в той или иной степени *сопротивляются* течению электрического тока. Это свойство характеризует величина, называемая *сопротивлением*. Сопротивление меди очень мало, но нулю оно все-таки не равно. Чем длиннее проводник, тем больше его сопротивление. Если подключить к лампочке провода длиной в несколько километров, их сопротивление будет настолько велико, что фонарик

работать не будет. Чтобы снизить сопротивление провода, его нужно сделать толще.

Я уже несколько раз использовал термин «напряжение», но пока не объяснил его значения. Что означает полуторавольтовое напряжение батарейки? Напряжение — одно из наиболее сложных понятий элементарного электричества. Под напряжением понимают потенциальную *способность* батарейки к выполнению работы. Напряжение существует независимо от того, подключено что-то к батарее или нет. Единица измерения напряжения — вольт (В) — названа в честь графа Алессандро Вольта (Alessandro Volta) (1745–1827), в 1800 г. изобретшего первую батарейку.

Более очевидным понятием является *сила тока* — мера количества электронов, реально текущих по цепи. Единица измерения силы тока называется ампером (А) в честь французского физика Андре Мари Ампера (André Marie Ampére) (1775–1836). Для получения тока в 1 А надо пропустить через поперечное сечение проводника 6 240 000 000 000 000 000 электронов в секунду.

Если все-таки прибегнуть к аналогии с водой и трубами, то сила тока подобна *количеству* воды, текущей по трубе, напряжение — *давлению*, а сопротивление по смыслу прямо противоположно толщине трубы — чем у́же труба, тем выше сопротивление. Чем больше давление, тем больше воды протекает по трубе. Чем у́же труба, тем меньше воды по ней пройдет. Количество воды, пропущенной по трубе (ток), прямо пропорционально давлению воды (напряжению) и обратно пропорционально сопротивлению.

Силу тока в цепи можно вычислить, если известно напряжение и сопротивление. Единица измерения сопротивления (т. е. свойства вещества препятствовать прохождению электронов) названа именем немецкого ученого Георга Симона Ома (Georg Simon Ohm) (1789–1854). Сформулированный им знаменитый закон Ома гласит, что:

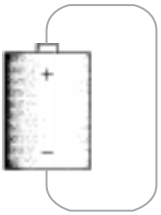
$$I = E / R,$$

где I — сила тока, E — напряжение (или *электродвижущая сила*), а R — сопротивление.

Взгляните на батарейку, которая ни к чему не подключена:

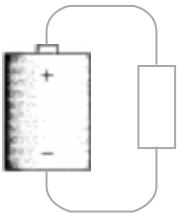
Напряжение батарейки E равно 1,5 В. Но ее положительный и отрицательный полюса соединены друг с другом только через воздух, а значит сопротивление (R) между ними весьма, весьма и весьма велико. Сила тока (I) равна результату деления 1,5 В на очень большое число, т. е. практически нулю.

Теперь соединим полюса небольшим отрезком медной проволоки (изоляцию на схемах я больше рисовать не буду):



Это соединение называется *коротким замыканием.* Напряжение по-прежнему 1,5 В, но сопротивление очень мало. Сила тока получается делением 1,5 В на малую величину, т. е. очень велика. Прямо-таки несметное количество электронов пронесется по проводу. Долгое время батарейка такой большой ток поддерживать не сможет, и потому ее напряжение упадет ниже номинального значения.

Большинство цепей занимают промежуточное положение между двумя этими крайностями. Условно сопротивление цепи можно показать так:



Прямоугольниками на электрических схемах показывают резисторы — электрические приборы, обладающие известным (не слишком большим и не слишком малым) сопротивлением.

Если сопротивление провода невелико, он раскаляется и начинает светиться, поскольку электрическая энергия в нем преобразуется в тепловую. На этом принципе основано действие лампы накаливания. Обычно ее создание приписывают знаменитому американскому изобретателю Томасу Эдисону (Thomas Edison) (1847–1931), но к 1879 г., когда он запатентовал лампу, основные принципы ее работы были уже хорошо известны[1].

Внутри лампы размещен тонкий провод — спираль, которая обычно изготавливается из вольфрама. Один конец спирали соединен с контактом в нижней части лампы; другой — с металлическим цоколем. При пропускании тока спираль нагревается и светится.

В большинстве фонарей устанавливается две последовательно соединенных батарейки, суммарное напряжение которых равно 3 В. Сопротивление лампочек в наиболее распространенных карманных фонариках составляет примерно 4 Ом. Значит, ток равен 3 В ÷ 4 Ом = 0,75 А, т. е. каждую секунду через лампочку проходит 4 680 000 000 000 000 000 электронов.

Кстати, если вы попробуете измерить сопротивление лампы с помощью омметра, то обнаружите, что оно намного меньше 4 Ом. Дело в том, что сопротивление вольфрама зависит от температуры, увеличиваясь по мере нагрева.

Вы, конечно, знаете, что на лампах, кроме рабочего напряжения, указывается мощность в ваттах (Вт). Единица измерения мощности названа в честь английского изобретателя Джеймса Уатта[2] (James Watt) (1736–1819), более известного благодаря своей работе над паровой машиной. Мощность P вычисляется по формуле:

$$P = E \times I.$$

[1] В 1874 г. угольную лампу накаливания запатентовал российский электротехник Александр Николаевич Лодыгин (1847–1923). — *Прим. перев.*

[2] В русском языке между именем изобретателя и названием единицы мощности имеется некоторое противоречие — единица называется ваттом, тогда как имя (Watt) традиционно пишется как Уатт. — *Прим. перев.*

Напряжение фонарика 3 В в сочетании с током 0,75 ампер доказывают, что мы имеем дело с лампочкой мощностью 2,25 Вт.

Мы проанализировали практически все детали фонарика — батарейки, провода и лампочку, — забыв о его важнейшей части!

Да-да, о переключателе! А ведь именно от него зависит, потечет электричество по цепи или нет. Когда переключатель пропускает через себя электрический ток, говорят, что он *включен*, или *замкнут*. *Выключенный*, или *разомкнутый*, переключатель электрический ток не пропускает.

Переключатель либо замкнут, либо разомкнут. Ток либо течет, либо нет. Лампочка либо горит, либо нет. Подобно двоичным кодам, изобретенным Морзе и Брайлем, наш простой фонарик пребывает в одном из двух взаимоисключающих состояний — включен или выключен. Третьего не дано. В следующих главах мы убедимся, что из этого сходства между двоичными кодами и электрическими цепями можно извлечь немалую пользу.

Глава 5

Там, за поворотом

Вам стукнуло двенадцать лет. И вот в один ужасный день семья вашего лучшего друга уезжает в другой город. Время от времени вы болтаете с другом по телефону, но это даже отдаленно не напоминает полуночные сеансы связи с фонариками и азбукой Морзе. Со временем вашим новым лучшим другом становится парень, живущий по соседству. Пришло время сдуть пыль с фонарика и обучить друга азбуке Морзе.

Но вот беда — из окна вашей комнаты не видно окна вашего приятеля! Дома рядом, да вот окна обращены в одну сторону. Пока вы не придумаете способ установить снаружи несколько зеркал, общаться по ночам с помощью фонариков у вас не получится.

Или получится?

Возможно, к этому времени вы уже узнали кое-что об электричестве и потому решили собрать из батареек, лампочек, переключателей и проводов фонарики с дистанционным управлением. Для начала вы устанавливаете в своей комнате батарейки и переключатель. Два провода выходят из вашего окна, пересекают забор и проходят в окно комнаты друга, где соединяются с лампочкой.

Ваш дом Дом вашего друга

Здесь я показываю только одну батарейку, но можно использовать и две. На этой и последующих схемах разомкнутый переключатель будет изображаться так:

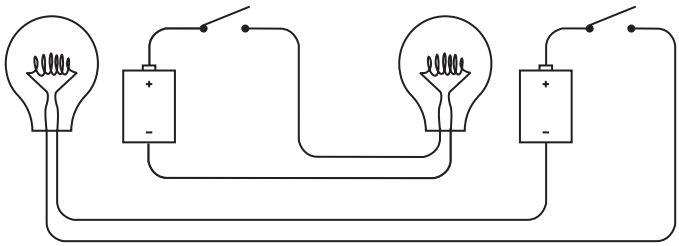
а замкнутый — так:

Фонарик, о котором мы говорим в этой главе, работает так же, как и фонарик из предыдущей, просто провода, соединяющие его компоненты, стали немного длиннее. Когда вы включаете переключатель у себя, лампочка загорается в комнате вашего друга.

Ваш дом Дом вашего друга

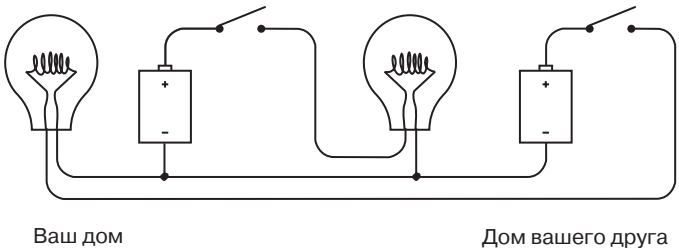
Теперь вы можете посылать другу сообщения с помощью азбуки Морзе.

Заставив один фонарик работать на расстоянии, вы можете собрать вторую такую же схему, чтобы друг мог посылать вам ответные сообщения.



Поздравляю! Вы создали настоящий двухсторонний телеграф. Он состоит из двух одинаковых цепей, полностью независимых и не соединенных друг с другом. Теоретически вы можете посылать сообщение своему другу в то самое время, когда он отправляет свое вам (хотя одновременно принимать и посылать сообщения будет нелегко).

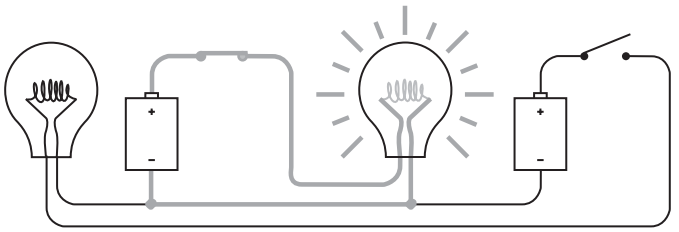
Если вы достаточно сообразительны, то сократите расход провода на 25%, немного изменив схему:



Теперь отрицательные контакты обеих батарей соединены. Две замкнутые цепи (батарея — переключатель — лампа — батарея) все еще независимы, хотя и связаны, как сиамские близнецы.

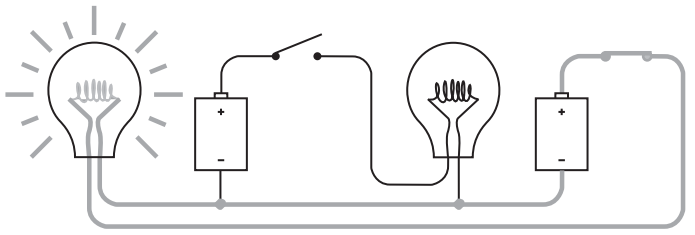
Такое соединение называется соединением с *общим проводом* (common). В нашей цепи общий провод начинается в точке соединения левой лампы и батареи, а заканчивается в точке соединения правой лампы и батареи. Эти соединения отмечены точками.

Рассмотрим работу схемы подробнее и убедимся, что она нам понятна. Начнем с того, что когда вы включаете переключатель у себя, в доме вашего друга загорается лампа. Провода, по которым течет ток, отмечены светлым оттенком.

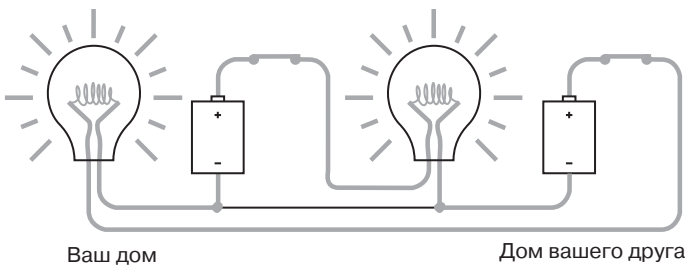


В другой части цепи тока нет, так как электроны не могут течь по разомкнутым проводам.

Если сигналы посылает ваш друг, а не вы, управление лампой в вашем доме осуществляется переключателем в доме вашего друга. И снова провода, по которым течет ток, отмечены светлым оттенком.



Если вы оба пытаетесь послать сообщения одновременно, ваши переключатели либо выключены, либо один переключатель включен, а другой выключен, либо оба включены. В последнем случае поток электронов в цепи течет так:



Через общую часть цепи ток не идет.

Соединив две цепи в одну общим проводом, мы сократили количество проводов между домами с четырех до трех, сэкономив 25% провода.

Если провода нужно протянуть на очень большое расстояние, хорошо было бы сократить затраты, отказавшись еще от одного провода. К сожалению, в цепи с 1,5-вольтовыми батарейками и небольшими лампочками это невыполнимо, но если вы имеете дело со 100-вольтовыми батареями и большими лампами, убрать провод можно.

Тут есть небольшая хитрость. Объединив цепи общим соединением, вы не должны использовать для него провод. Его можно заменить любым проводником. Например, гигантским шаром радиусом около 6 400 км, состоящим из металлов, камней, воды и органических веществ. Этот шар — планета Земля.

Говоря в предыдущей главе о хороших проводниках, я упоминал серебро, медь и золото, но отнюдь не гравий или перегной. Правду сказать, земля не такой уж хороший проводник, хотя некоторые виды почв (например, влажная земля) проводят ток намного лучше других (например, сухой песок). Но у проводников есть одно общее свойство: чем проводника больше, тем лучше. Толстый провод проводит ток лучше, чем тонкий. И здесь Земля не знает себе равных. Она действительно очень, очень велика.

Чтобы использовать землю в качестве проводника, недостаточно просто воткнуть проводок в помидорную грядку. Вам нужно нечто, поддерживающее с землей прочный контакт, я имею в виду проводник с большой площадью поверхности, например, медный штырь длиной 2,5 метра и толщиной 1,5 см. Он контактирует с землей на площади 1 200 кв. см. Забив штырь в землю кувалдой, подсоедините к нему провод от схемы. Иногда из меди делают водопроводные трубы для холодной воды. Если в вашем доме есть такая труба, уходящая под землю за пределами дома, подсоедините провод к ней.

Электрический контакт с землей называется *заземлением*, или просто *землей*. Вокруг термина «земля» возникает небольшая путаница, так как иногда так же называют часть цепи, которую мы назвали общим проводом. В этой и следующих главах под заземлением будет пониматься физическое соединение с землей, если я явно не скажу об обратном.

На электрических схемах заземление изображается так:

Электрики используют этот значок, чтобы не тратить время на рисование 2,5-метрового штыря, зарытого в землю.

Рассмотрим работу цепи с заземлением. Мы начали эту главу с создания односторонней сигнальной схемы.

Ваш дом Дом вашего друга

Если бы вы использовали мощные батареи и лампы, для соединения домов вам понадобился бы только один провод, так как в качестве второго проводника вы использовали бы землю.

Когда вы включите переключатель, ток потечет так:

Электроны вытекают из земли у дома вашего друга, проходят через лампу и провод, минуют переключатель в вашем доме и попадают на положительный контакт батареи. С отрицательного контакта батареи электроны уходят в землю.

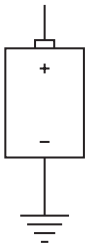
Воображение невольно рисует картину, как электроны выпрыгивают из медного штыря, забитого в землю на заднем дворе вашего дома, и пробираются под землей к другому медному штырю, забитому в землю на заднем дворе дома вашего друга.

Но если вы вспомните, что земля выполняет ту же функцию для многих тысяч электрических цепей по всему миру, то, вероятно, заинтересуетесь, откуда электроны знают, к какому именно штырю им направляться. Разумеется, они не знают. Поэтому лучше придумать для земли другое сравнение.

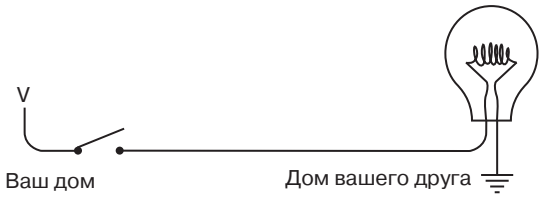
Да, земля — это очень большой проводник, но ее можно рассматривать и как огромный резервуар с электронами. *Земля для электронов — то же, что океан для капли воды.* Земля — это практически безграничный источник электронов и бездонный сток для них.

Однако земля обладает и *некоторым* сопротивлением. Именно поэтому мы и не смогли уменьшить с ее помощью расход провода, работая с 1,5-вольтовыми батарейками и маленькими лампочками. Для маломощных батареек сопротивление земли слишком велико.

Кстати, на двух предыдущих рисунках батарейка соединена с землей отрицательным контактом.



Я больше не буду рисовать на схемах батарейку, соединенную с землей. Вместо этого я буду ставить большую букву V, обозначающую *входное напряжение* (voltage). Теперь односторонняя телеграфная система с лампочкой выглядит так:

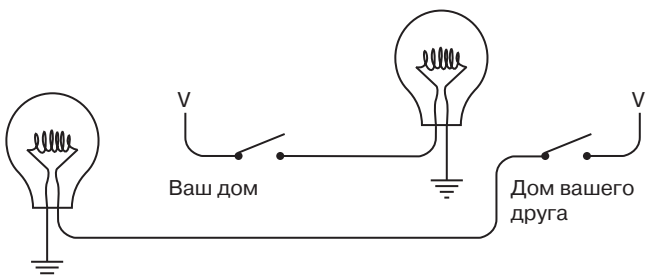


Считайте, что буква V символизирует электронный вакуум, противоположный земле — океану электронов. Электронный вакуум через цепь вытягивает электроны из земли, производя по пути работу (например, раскаляя спираль лампы).

Иногда землю называют точкой с *нулевым потенциалом* (zero potential). Это означает, что напряжения на ней нет. Напряжение, как я уже объяснял, являет собой потенциал для совершения работы, подобно тому, как потенциальным источником энергии является кирпич, висящий в воздухе. Нулевой потенциал — это кирпич, лежащий на земле: падать-то ему уже некуда.

В главе 4 мы в первую очередь обращали внимание на то, что электрические цепи замкнуты. Наша новая цепь совсем не похожа на замкнутый круг, но все же является таковым. Достаточно заменить букву V на батарейку, отрицательный контакт которой соединен с землей, а затем нарисовать провод, соединяющий все точки заземления. В результате у вас получится первый рисунок из этой главы.

Таким образом, используя пару медных штырей (или водопроводных труб), мы можем соорудить двухстороннюю сигнальную систему, перебросив через забор между домами всего два провода.



По своему действию эта цепь не отличается от предыдущей, в которой изгородь между домами пересекали три провода.

В этой главе мы сделали важный шаг вперед в развитии коммуникации. Раньше мы могли общаться с помощью азбуки Морзе только в зоне прямой видимости и только на расстоянии, которое способен преодолеть свет фонарика.

С помощью проводов мы не только собрали систему связи, позволяющую общаться вне зоны прямой видимости, но еще и сняли ограничение в расстоянии. Мы теперь можем передавать информацию на сотни и тысячи километров, протянув достаточно длинные провода. Так?

Так, да не совсем. Медь — очень хороший проводник, но не идеальный. Чем длиннее провода, тем больше у них сопротивление. Чем больше сопротивление, тем меньший течет ток. Чем меньше ток, тем тусклее светит лампа.

Насколько длинными могут быть провода? Посчитаем. Допустим, вы реализуете нашу первую четырехпроводную двунаправленную схему без заземления и общего провода, с маломощными батарейками и лампочками. Чтобы не слишком ударить по карману, вы для начала купили провод №20 по цене 9,99 доллара за 100 футов (около 30 м), который обычно применяется для подключения колонок к стереосистеме. Он двухжильный и потому прекрасно подходит для нашего телеграфа. Если от вашей комнаты до комнаты вашего друга меньше 15 м, вы обойдетесь одним мотком.

В США для измерения толщины провода используется система стандартов *American Wire Gauge* (AWG). Чем меньше номер AWG, тем толще провод и тем ниже его сопротивление. Диаметр провода №20 — примерно 0,8 мм, а его сопротивление — около 10 ом на 300 м или 1 ом на удвоенное расстояние между вашими комнатами.

Это совсем неплохо. Но что если вы захотите протянуть провод на полтора километра? Его сопротивление будет около 50 ом. Как вы помните из предыдущей главы, сопротивление лампочки — всего 4 ома. По закону Ома легко вычислить, что сила тока в удлиненной цепи будет уже не 0,75 ампера (3 вольта разделить на 4 ома), как раньше, а меньше 0,06 ампера (3 вольта разделить на 50 ом). Я почти уверен, что при таком слабом токе свечения спирали лампочки вы не увидите.

Можно, конечно, решить проблему, использовав провод потолще, хотя обойдется он вам недешево. У провода №10 толщина около 2,5 мм и сопротивление всего около 1 ома на 300 м, т. е. 5 ом на 1,5 км. Его 35-футовый (около 10 м) моток стоит 11,99 долларов, при этом учитывайте, что провод одножильный, поэтому покупать его придется вдвое больше, чем «двадцатки».

Другой выход — увеличить напряжение и использовать лампы с более высоким сопротивлением. Скажем, обычная 100-ваттная лампа, которая освещает вашу комнату, предназначена для работы в сети с напряжением в 120 вольт и имеет сопротивление около 144 ом. Понятно, что в цепи с такой лампой сопротивление проводов играет меньшую роль.

Именно такие проблемы встали 150 лет назад перед людьми, протягивавшими первые телеграфные линии по Америке и Европе. Ни толщина проводов, ни повышенное напряжение не помогут тянуть телеграфные провода бесконечно. Максимальный длина системы, работающей по этой схеме, составляет три сотни километров — ничтожная величина по сравнению с тысячами километров между Нью-Йорком и Калифорнией.

Решить эту проблему — конечно, не для фонариков, а для настоящего телеграфа — удалось с помощью простого и невзрачного устройства, на основе которого, как выяснилось позднее, можно создать настоящий компьютер.

Глава 6

Телеграф и реле

Сэмюэль Морзе родился в 1791 г. в городе Чарльстон (штат Массачусетс, США), который ныне вошел в состав Бостона. К этому времени уже четыре года существовала Конституция Соединенных Штатов, шел первый президентский срок Джорджа Вашингтона, а Россией правила Екатерина Великая. Через два года после рождения Морзе сложили головы на плахе король Людовик XVI и королева Мария-Антуанетта. В конце 1791 г. в возрасте 35 лет умер Моцарт, закончив перед этим свою последнюю оперу «Волшебная флейта».

Морзе закончил Йельский университет, обучался изобразительному искусству в Лондоне и стал известным художником-портретистом. Его картина «Генерал Лафайет» (1825) до сих пор висит в нью-йоркском Сити-Холле. В 1836-м он баллотировался на пост мэра Нью-Йорка в качестве независимого кандидата и набрал 5,7% голосов. Также он был одним из первых фотографов, обучался искусству делать дагерротипные фотографии у самого Луи Дагерра (Louis Daguerre) и сделал несколько первых американских снимков. В 1840 г. он передал свое мастер-

ство фотографа 17-летнему Мэтью Брэди (Mathew Brady), который прославился фотографиями на темы Гражданской войны и фотопортретами Авраама Линкольна и самого Сэмюэля Морзе.

Но это лишь небольшие штрихи в эклектичной карьере Морзе. Более всего он известен как изобретатель телеграфа и азбуки, носящей его имя.

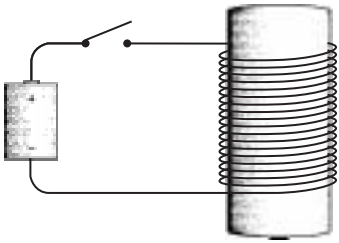
Мгновенная всемирная связь, к которой мы привыкли, появилась относительно недавно. В начале XIX в. передавать информацию можно было либо быстро, либо на большое расстояние, но не то и другое одновременно. Мгновенная связь была возможна или на расстоянии слышимости, или на расстоянии видимости. Для передачи информации на большие расстояния — с помощью писем — требовалось время, а также лошади, поезда или корабли.

Неоднократные попытки ускорить передачу информации на далекие расстояния предпринимались за десятилетия до изобретения Морзе. В простейшем варианте дальнюю связь организовывали в виде цепочки людей-передатчиков, которые обменивались сообщениями на расстоянии прямой видимости, размахивая флажками. В технически более совершенных системах роль человека-передатчика играла массивная механическая конструкция с подвижными рычагами.

Идея телеграфа (от слов «далеко» и «писать») витала в воздухе с самого начала XIX в., и многие изобретатели приложили к ней руку еще до того, как в 1832 г. свои опыты начал Сэмюэль Морзе. В принципе суть электрического телеграфа проста: вы делаете нечто на одном конце провода, в результате чего на другом конце провода что-то происходит. Именно этим мы занимались в предыдущей главе, создавая дистанционную систему управления фонариком. Однако Морзе в своем передающем устройстве использовать лампочку не мог, так как она была изобретена лишь в 1879 г. Вместо лампочки Морзе использовал явление *электромагнетизма*.

Возьмите железный стержень, намотайте на него пару сотен витков тонкого провода, а затем пропустите по проводу ток. Железный стержень станет магнитом и будет притягивать кусочки железа и стали (провод в электромагните должен быть именно тонким, чтобы обладать заметным сопротивлением и

не приводить к короткому замыканию). Выключите ток, и железный стержень потеряет свои магнитные свойства.



Электромагнит — основа телеграфа. Замыкание и размыкание переключателя на одном конце провода приводит в действие электромагнит на другом.

Первый телеграфный аппарат Морзе был, как ни странно, сложнее своих более поздних модификаций. Морзе полагал, что телеграфный аппарат должен писать что-то на бумаге, хотя и не слова, так как это было бы слишком сложно. Но *что-то* (какие-нибудь закорючки или тире с точками) должно быть написано. Интересно, что Морзе, как и Ойи с его идеей выпуклых букв, оказался в плену штампа, считая, что передача информации без записей на бумаге невозможна.

Сэмюэль Морзе известил патентное бюро об изобретении телеграфа уже в 1836 г., но лишь в 1843 г. ему удалось убедить Конгресс профинансировать публичную демонстрацию этого устройства. В исторический день 24 мая 1844 г. по телеграфной линии, соединившей Вашингтон с Балтимором, была успешно передана цитата из Библии.

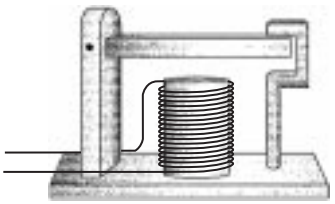
Обычный телеграфный ключ выглядел примерно так:

Несмотря на причудливый вид, это всего лишь переключатель, форма которого позволяет осуществлять передачу с максимальной скоростью. Удобнее всего в течение долгого времени работать с ключом так: держать ручку между большим, указа-

тельным и средним пальцами, поднимая и опуская ее. Когда ключ задерживается в нижнем положении ненадолго, получается точка азбуки Морзе, длинное нажатие соответствует тире.

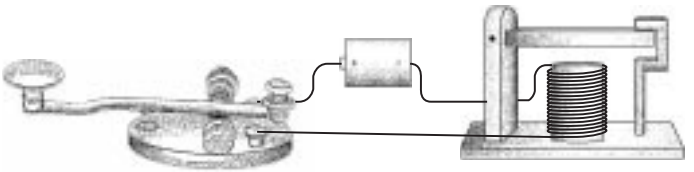
На другом конце линии находился приемник — по существу электромагнит, притягивавший металлический рычаг. Поначалу к рычагу прикреплялся карандаш. Пружинный механизм медленно протягивал через аппарат бумажную ленту, а карандаш, управляемый электромагнитом, прыгал вверх-вниз и рисовал на бумаге точки и тире. Человек, знающий азбуку Морзе, переводил точки и тире в буквы и слова.

Но вскоре телеграфные операторы обнаружили, что могут переводить сообщения, просто слушая щелчки электромагнита. Карандаш в конце концов был упразднен, уступив место устройству такого вида:



Когда телеграфный ключ нажат, электромагнит тянет подвижный металлический молоточек вниз, и раздается звук одного тона («тик»). Когда ключ отпущен, электромагнит выключается, и молоточек возвращается в исходное положение, производя звук другого тона («так»). Быстрое чередование звуков («тик-так») соответствует точке, медленное — тире.

Ключ, электромагнит с молоточком и батарейку можно проводами соединить в схему, подобную устройству из предыдущей главы.

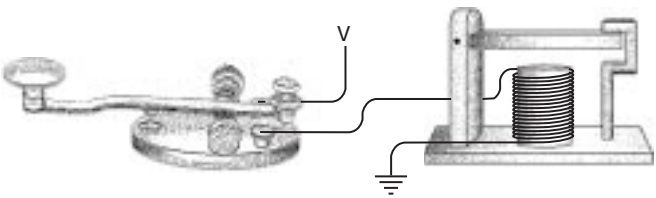


Ваша телеграфная станция

Телеграфная станция вашего друга

Мы уже выяснили, что для соединения двух телеграфных станций два провода не нужны. Если половину цепи заменит земля, хватит и одного провода.

Как и в предыдущей главе, вместо батарейки и земли мы будем рисовать букву V. А посему окончательная схема с одним проводом выглядит так:



Для двухсторонней связи попросту понадобится еще один ключ и электромагнит с молоточком.

Изобретение телеграфа положило начало современной связи. Впервые люди смогли передавать информацию дальше, чем видит глаз или слышит ухо, и быстрее почтовых лошадей. Но самое пикантное в этом изобретении — двоичный код. В более поздних формах электрической и беспроводной связи (телефон, радио, телевидение) двоичные коды были забыты, но лишь с тем, чтобы снова вернуться к жизни в компьютерах, компакт-дисках и цифровом телевидении.

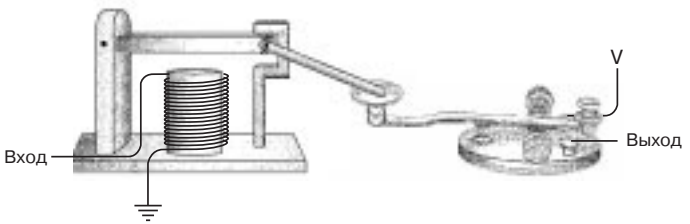
Телеграф Морзе одержал победу над другими проектами не в последнюю очередь благодаря своей нетребовательности. Протяните провод между ключом и электромагнитом, и система будет работать. Другие телеграфные системы не были такими простыми и потому безотказными. И все же у телеграфа есть недостатки — в главе 5 я уже говорил о большом сопротивлении длинных проводов. Хотя некоторые телеграфные линии с напряжением до 300 В работали на расстояниях до 500 км, бесконечно тянуть провода нельзя.

Одно решение кажется очевидным — организовать систему ретрансляции: посадить через каждую пару сотен километров человека с ключом, который будет принимать сообщения и пересылать их дальше по линии.

Теперь представьте, что для работы в качестве ретранслятора телеграфной компанией наняты именно вы. И сидите вы где-то между Нью-Йорком и Калифорнией в маленькой будке со столом и стулом. Провод, входящий в восточное окно, соединен с электромагнитом, а телеграфный ключ подключен к батарейке и проводу, выходящему в западное окно. Ваша работа — принимать сообщения из Нью-Йорка и передавать их в Калифорнию.

Сначала вы предпочитаете полностью принять сообщение и лишь потом отправлять его. Вы записываете буквы по щелчкам молоточка, а когда сообщение заканчивается, передаете их с помощью ключа. Но после некоторой практики вы уже можете передавать сообщение сразу, на слух, вообще не записывая. Это здорово экономит время.

И вот в один прекрасный день, принимая очередное сообщение, вы обращаете внимание на молоточек, который скачет вверх и вниз, а потом на свои пальцы, поднимающие и опускающие ключ. Потом вы снова смотрите на молоточек и на ключ, и вдруг понимаете, что они совершают одно и то же движение. Рядом с будкой вы находите небольшую палочку и с ее помощью физически соединяете ключ с молоточком.

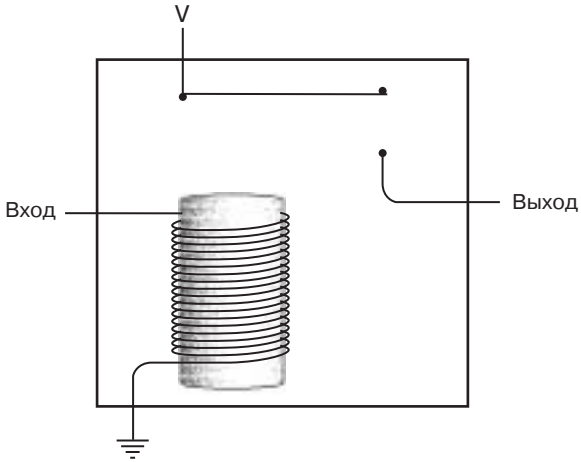


Теперь аппарат работает самостоятельно, а вы можете немного вздремнуть или пойти на рыбалку.

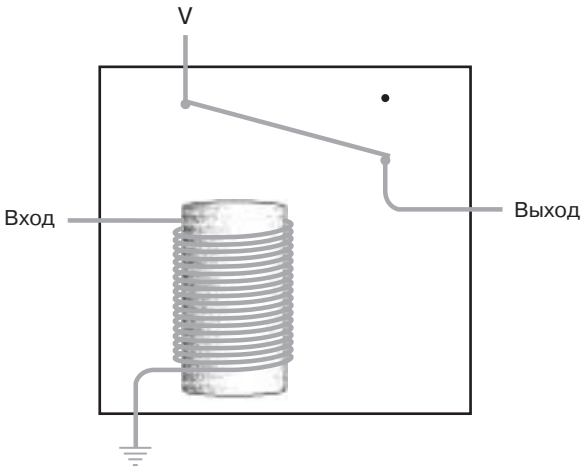
Интересно немного пофантазировать, но в действительности Сэмюэль Морзе с самого начала понимал, что телеграфный аппарат должен работать именно так. «Изобретенное» нами устройство называется *повторителем* (repeater), или *реле* (relay). В реле входящий ток приводит в действие электромагнит, который притягивает металлический рычаг. Рычаг в свою очередь используется как часть переключателя, соединяющего батарейку с выходящим проводом. Таким образом, слабый

входящий ток «усиливается», превращаясь в сильный выходящий.

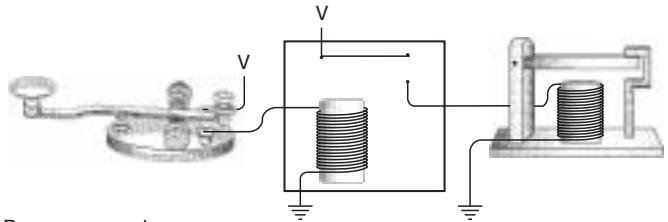
Схематически реле можно изобразить так:



Входящий ток приводит в действие электромагнит, тот притягивает гибкую металлическую полоску, а она замыкает цепь для выходящего тока.



Поэтому ключ, реле и электромагнит с молоточком соединяются примерно так:



Реле — замечательное устройство. Конечно, это просто переключатель, но его включает и выключает не человек, а ток. Поразительно, какие замечательные устройства можно создать на основе такого простого прибора. По правде говоря, практически из одних только реле можно собрать целый компьютер.

Да, оставить реле в музее телеграфа — слишком дорогое удовольствие. Давайте-ка стащим его, спрячем под пиджаком и быстро-быстро пройдем мимо охраны. Это реле нам очень пригодится. Но прежде чем начать им пользоваться, мы должны научиться считать.

Глава 7

Великолепная десятка

Представление о языке как о коде кажется вполне естественным. Большинство из нас в школе изучали (или хотя бы пытались) иностранный язык, поэтому мы вполне допускаем, что животное, которое мы зовем кошкой, может называться также *cat*, *gato*, *chat*, *Katze* или *κατα*.

Цифры далеко не так разнообразны. На каком бы языке мы ни говорили и как бы мы ни произносили *названия* цифр, почти все жители планеты *пишут* их одинаково:

1 2 3 4 5 6 7 8 9 10

Неспроста математику называют «универсальным языком»!

Числа, безусловно, являются самым абстрактным кодом из всех, с которыми нам приходится иметь дело в повседневной жизни. Увидев число:

3

мы не должны немедленно соотнести его с чем-либо. Можно представить себе 3 яблока или 3 других предмета, но мы не почувствуем себя менее комфортно, узнав из контекста, что речь идет о возрасте ребенка, номере телевизионного канала, счете хоккейного матча или количестве стаканов муки в рецепте торта. Мы настолько привыкли к абстрактности чисел, что нам скорее трудно будет понять, что такое количество яблок:

необязательно обозначать символом:

3

Большую часть этой и следующей глав мы посвятим убеждению себя в том, что такое количество яблок:

можно обозначить также числом:

11

Для начала избавимся от представления об исключительности числа десять. В том, что большинство цивилизаций основало свои системы счисления на десяти (или пяти), нет ничего удивительного. Испокон веку люди использовали для счета пальцы. Будь у нас восемь или двенадцать пальцев, наши методы счета были бы немного другими. Неслучайно у слов *пять* и *пятерня* один корень.

В этом смысле выбор системы счисления, основанной на десяти, или *десятичной*, совершенно произволен. Тем не менее мы наделяем числа, основанные на 10, почти магическим значением и даже даем им специальные имена. Вот как выглядит последовательность степеней числа 10:

$10^1 = 10$
$10^2 = 100$ (сотня)
$10^3 = 1\,000$ (тысяча)
$10^4 = 10\,000$
$10^5 = 100\,000$
$10^6 = 1\,000\,000$ (миллион)
$10^7 = 10\,000\,000$
$10^8 = 100\,000\,000$
$10^9 = 1\,000\,000\,000$ (миллиард)

Большинство историков полагает, что числа были изначально придуманы торговцами для счета людей, имущества и пр. Например, факт наличия у кого-то четырех утят можно записать так:

В конце концов человек, работа которого заключалась в рисовании утят, подумал: «Зачем мне рисовать четырех утят? Почему не нарисовать одного утенка и отметить, что их четыре штуки... я не знаю... черточками или чем-нибудь еще?»

Однажды наступил день, когда рисовальщику понадобилось отобразить 27 утят. Картина получилась, прямо скажем, смехотворная.

Глядя на нее, кто-то сказал: «Должен быть более удобный способ». Так появились числа.

Из первых систем счисления сейчас широко используются только римские цифры. Вы встретите их на циферблатах часов, в виде дат на памятниках, в виде номеров страниц в книгах и — что особенно обидно — в отметках об авторских правах на кинофильм. На вопрос: «В каком году была сделана картина?» — может ответить лишь человек, способный быстро расшифровать MCMLIII в конце длинного хвоста титров.

Римскими цифрами 27 утят записываются так

Суть кодирования проста: вместо 10 черточек пишем X, вместо 5 черточек — V.

Римские цифры, дожившие до наших дней, выглядят так

I V X L C D M

Символ I означает «один». Он мог произойти от черточки или от указательного пальца. Символ V, означающий 5, возможно, символизирует руку с отогнутым большим пальцем. Символ X, означающий 10, состоит из двух символов V. L равно пятидесяти. Цифра C происходит от латинского слова *centum* (100), а цифра M — от слова *mille* (1000).

Некогда считалось, что римские цифры легко складывать и вычитать, поэтому в Европе их долго использовали для ведения бухгалтерии. Фактически при сложении двух римских чисел вы просто объединяете все цифры и упрощаете результат, применяя несколько правил: пять I равно V, два V равно X, пять X равно L и т. д.

Но вот умножать и делить римские числа трудно. По той же причине не годятся для сложных вычислений и другие ранние системы счисления (например, древнегреческая). Геометрия, разработанная древними греками, и по сей день практически в неизменном виде преподается в школах, а вот своей алгебры они так и не создали.

Система счисления, которой мы пользуемся, называется арабской. Разработана она в Индии, но в Европу ее принесли арабские математики. Среди них особенно прославился персидский математик Мухаммед ибн-Муса аль-Хорезми (от имени которого происходит слово «алгоритм»). Приблизительно в 825 г. он написал книгу по алгебре, в которой использовал индийскую систему счисления. Латинский перевод этой книги датируется 1120 г. Он сыграл важную роль в ускорении перехода Европы от римских чисел к современной системе.

Арабская система счисления отличается от своих предшественниц тремя моментами.

- Она является *позиционной*, т. е. количество, представляемое цифрой, зависит от ее положения в числе. Позиция цифры в числе не менее (а на самом деле более) важна, чем сама цифра. В числах 100 и 1 000 000 содержится по одной единице, но никого не нужно убеждать, что миллион намного больше ста.

- В арабской системе *нет* того, что имеется практически во всех ранних системах счисления, а именно специального символа для числа 10.
- С другой стороны, в арабской системе есть нечто более важное, чем символ для 10, отсутствующее почти во всех ранних системах счисления, — ноль.

Да, именно 0. Скромный ноль — без сомнения, одно из самых важных изобретений в истории чисел и математики. Он необходим для позиционной записи, так как позволяет отличить 25 от 205 или 250. Ноль также облегчает выполнение многих математических операций, которые в непозиционных системах сопряжены с многочисленными трудностями, особенно умножения и деления.

Вся структура построения арабских чисел раскрывается при их произношении. Например, число 4 825 мы называем «четыре тысячи восемьсот двадцать пять». Это означает:

четыре тысячи
восемь сотен
два десятка и
пять

Мы можем записать компоненты этого числа так:

$$4825 = 4000 + 800 + 20 + 5$$

или с еще большей степенью детализации:

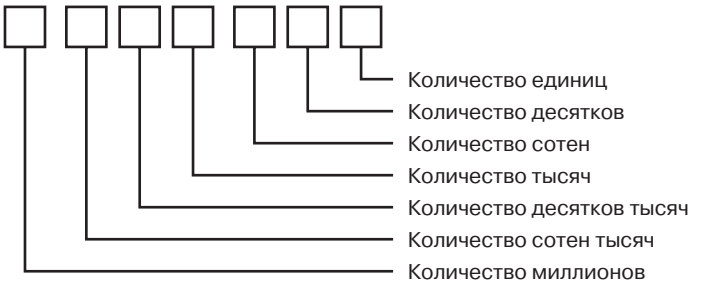
$$\begin{aligned} 4825 = {} & 4 \times 1000 + \\ & 8 \times 100 + \\ & 2 \times 10 + \\ & 5 \end{aligned}$$

С помощью степеней числа десять, число представляется так:

$$\begin{aligned} 4825 = {} & 4 \times 10^3 + \\ & 8 \times 10^2 + \\ & 2 \times 10^1 + \\ & 5 \times 10^0 \end{aligned}$$

Помните, что любое число в степени 0 равно 1.

В числе, состоящем из нескольких цифр, каждая позиция имеет определенное значение. С помощью семи квадратиков на схеме можно представить любое число от 0 до 9 999 999:



Каждая позиция соответствует степени десяти. Специальный символ для десяти нам не нужен, так как мы сдвигаем 1 в следующую позицию, а на освободившемся месте пишем 0.

Что особенно приятно, дробные части чисел, записанные в виде цифр справа от десятичного разделителя, следуют тому же принципу. Число 42 705,684 =

$$4 \times 10\,000 +$$
$$2 \times 1000 +$$
$$7 \times 100 +$$
$$0 \times 10 +$$
$$5 \times 1 +$$
$$6 \div 10 +$$
$$8 \div 100 +$$
$$4 \div 1000$$

Можно записать это число и без знака деления:

$$4 \times 10\,000 +$$
$$2 \times 1000 +$$
$$7 \times 100 +$$
$$0 \times 10 +$$
$$5 \times 1 +$$
$$6 \times 0{,}1 +$$
$$8 \times 0{,}01 +$$
$$4 \times 0{,}001$$

или с помощью степеней 10

$$4 \times 10^4 +$$
$$2 \times 10^3 +$$
$$7 \times 10^2 +$$

$$0 \times 10^{1} +$$
$$5 \times 10^{0} +$$
$$6 \times 10^{-1} +$$
$$8 \times 10^{-2} +$$
$$4 \times 10^{-3}$$

Заметьте: показатели степени доходят до 0, а затем становятся отрицательными.

Мы знаем, что 3 + 4 = 7. Точно так же, 30 + 40 = 70, 300 + 400 = = 700, 3000 + 4000 = 7000. В этом и состоит красота арабской системы счисления. Складывая десятичные числа любой длины, вы разбиваете процедуру на одни и те же простые шаги. На каждом шаге выполняется не более чем сложение пары одноразрядных (т. е. записываемых с помощью одной цифры) чисел. Вот для чего вас заставляли учить таблицу сложения.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Найдите в верхней строке и левом столбце два числа, которые хотите сложить. На пересечении столбца и строки будет их сумма, например, 4 + 6 = 10.

При умножении двух десятичных чисел приходится проделывать более сложные действия, но смысл их опять же со-

стоит в разбиении процедуры на несколько шагов, на каждом из которых складываются или умножаются одноразрядные числа. Возможно, в школе вы учили и таблицу умножения.

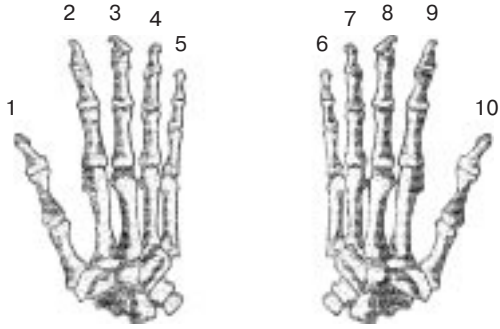
×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Ключевое преимущество позиционной записи не в том, что она хорошо работает в десятичной системе счисления, а в том, что она хорошо работает в системах счисления, основанных *не* на десяти. Десятичная система хороша для нас, но неудобна для мультипликационных персонажей, наделенных не пятью, а всего четырьмя пальцами на каждой руке (или лапе). Они определенно предпочтут систему счисления, основанную на восьми. Интересно, что многое из того, что мы знаем о десятичной системе, применимо и к системе счисления, которую применяют наши друзья из мультфильмов.

Глава 8

Альтернативы десяти

Десять — число для людей исключительно важное. У большинства из нас десять пальцев на руках (и на ногах), и мы определенно предпочитаем иметь по десять и тех, и других. Поскольку с помощью пальцев удобно считать, мы придумали целую систему счисления, основанную на числе десять.



Я уже говорил в предыдущей главе, что система счисления, которой мы пользуемся, называется *десятичной*. Она кажется настолько естественной, что об альтернативах поначалу и помыслить трудно. Действительно, когда мы видим число *10*, мы не можем не думать, что оно относится именно к такому количеству утят:

10 = 

Но причина, по которой 10 обозначает именно столько утят, в том, что их количество совпадает с количеством пальцев на руках. Если бы у людей было другое количество пальцев, у нас был бы другой способ счета, и 10 означало бы что-то другое, например, вот столько утят:

10 = 

или столько:

10 = 

или даже вот столько:

10 = 

Поняв, как обозначить числом 10 всего двух утят, мы обретем способность представлять любые числа с помощью переключателей, проводов, лампочек и реле (что необходимо для создания компьютера).

Если бы у человека было на каждой руке только по четыре пальца, как у мультяшки, нам, вероятно, никогда не пришла бы в голову мысль о системе счисления, основанной на десяти. Вместо нее мы считали бы нормальной, естественной, рациональной, неизбежной, бесспорной и единственно разумной систему счисления, основанную на восьми. И называли бы ее не десятичной, а *восьмеричной*.

В восьмеричной системе счисления не нужен символ:

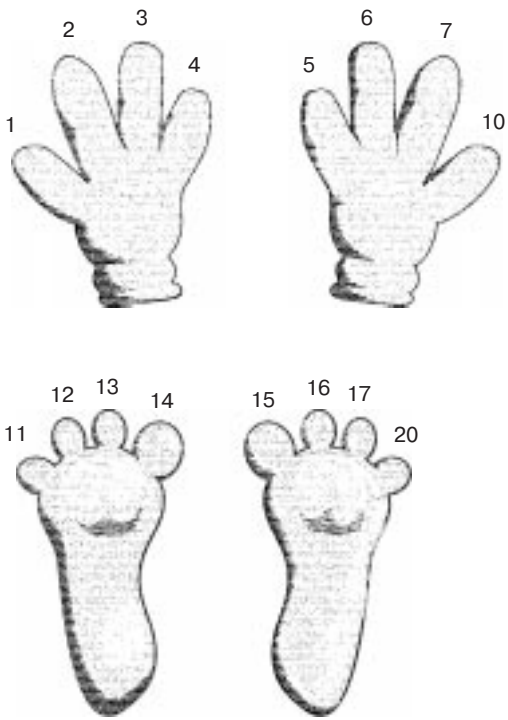
9

Покажите этот символ мультяшке, и услышите от него: «А для чего нужна эта загогулина?» Подумав еще немного, вы поймете, что в восьмеричной системе можно обойтись и без символа:

8

В десятичной системе нет специального символа для десяти, поэтому в восьмеричной системе нет специального символа для восьми.

В десятичной системе мы считаем так: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 и 10. В восьмеричной системе счет идет так: 0, 1, 2, 3, 4, 5, 6, 7 и... что дальше? Символы-то кончились. Выход один — после 7 ставить 10. Но эта десятка уже не равна числу пальцев на руках человека. В восьмеричной системе 10 — это количество пальцев на руках мультяшки.



Даже когда пальцы на ногах и руках будут исчерпаны, счет в восьмеричной системе можно продолжать. В целом он идет, как и счет в десятичной системе, но без чисел, содержащих цифры 8 и 9:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 21, 22, 23, 24, 25, 26, 27, 30, 31, 31, 32, 33, 34, 35, 36, 37, 40, 41, 41, 42, 43, 44, 45, 46, 47, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 64, 65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77, 100…

Последнее число — 100 — есть число пальцев мультяшки, умноженное само на себя.

Чтобы избежать путаницы при написании восьмеричных и десятичных чисел, будем указывать тип системы в нижнем индексе. Индекс ДЕСЯТЬ будет означать десятичную систему, а ВОСЕМЬ — восьмеричную.

Итак, число гномов, которых встретила Белоснежка, равно $7_{\text{десять}}$ или $7_{\text{восемь}}$.

Число пальцев на руках мультяшек равно $8_{\text{десять}}$ или $10_{\text{восемь}}$.

Число симфоний, написанных Бетховеном, равно $9_{\text{десять}}$ или $11_{\text{восемь}}$.

Число пальцев на руках человека равно $10_{\text{десять}}$ или $12_{\text{восемь}}$.

Число месяцев в году равно $12_{\text{десять}}$ или $14_{\text{восемь}}$.

Число дней в двух неделях равно $14_{\text{десять}}$ или $16_{\text{восемь}}$.

Число килограммов в пуде равно $16_{\text{десять}}$ или $20_{\text{восемь}}$.

Число часов в сутках равно $24_{\text{десять}}$ или $30_{\text{восемь}}$.

Число букв в латинском алфавите равно $26_{\text{десять}}$ или $32_{\text{восемь}}$.

Число зубов у взрослого человека равно $32_{\text{десять}}$ или $40_{\text{восемь}}$.

Число карт в колоде равно $52_{\text{десять}}$ или $64_{\text{восемь}}$.

Число клеток на шахматной доске равно $64_{\text{десять}}$ или $100_{\text{восемь}}$.

Число созвездий на небе равно $88_{\text{десять}}$ или $130_{\text{восемь}}$.

Число сантиметров в метре равно $100_{\text{десять}}$ или $144_{\text{восемь}}$.

Число женщин-одиночек в начале Уимблдонского турнира равно $128_{\text{десять}}$ или $200_{\text{восемь}}$.

Площадь Мемфиса в квадратных милях равна $256_{\text{десять}}$ или $400_{\text{восемь}}$.

В этом списке обратите внимание на красивые круглые восьмеричные числа, например, $100_{\text{восемь}}$, $200_{\text{восемь}}$ и $400_{\text{восемь}}$. *Красивым круглым числом* мы обычно называем число с нулями на конце. Два нуля на конце десятичного числа означают, что оно

кратно $100_{\text{десять}} = 10_{\text{десять}} \times 10_{\text{десять}}$. В восьмеричной системе два нуля на конце означают, что число кратно $100_{\text{восемь}} = 10_{\text{восемь}} \times \times 10_{\text{восемь}}$ (т. е. $8_{\text{десять}} \times 8_{\text{десять}} = 64_{\text{десять}}$).

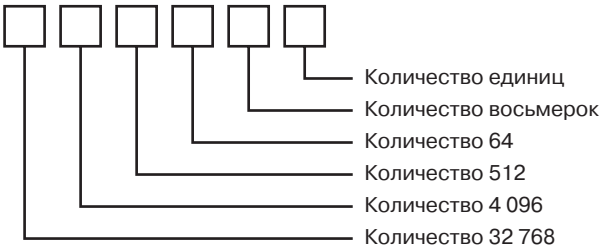
Заметьте также, что все десятичные эквиваленты круглых восьмеричных чисел, а именно $64_{\text{десять}}$, $128_{\text{десять}}$ и $256_{\text{десять}}$, являются степенями двойки. В этом есть смысл. Число $400_{\text{восемь}}$, например, равно произведению чисел $4_{\text{восемь}}$, $10_{\text{восемь}}$ и $10_{\text{восемь}}$, каждое из которых является степенью двойки. Естественно, при умножении степени двойки на степень двойки, мы опять-таки получаем степень двойки.

В таблице некоторые степени двойки показаны в десятичном и восьмеричном представлении.

Степень двойки	**Десятичное**	**Восьмеричное**
2^0	1	1
2^1	2	2
2^2	4	4
2^3	8	10
2^4	16	20
2^5	32	40
2^6	64	100
2^7	128	200
2^8	256	400
2^9	512	1000
2^{10}	1024	2000
2^{11}	2048	4000
2^{12}	4096	10000

Красивые круглые числа в крайнем правом столбце указывают на то, что при работе с двоичными кодами удобны недесятичные системы счисления.

По способу построения чисел восьмеричная система не отличается от десятичной за исключением некоторых тонкостей. Например, каждой позиции в восьмеричном числе соответствует цифра, умноженная на степень восьми.



Например, восьмеричное число $3\,725_{\text{восемь}}$ раскладывается так:

$$3\,725_{\text{восемь}} = 3\,000_{\text{восемь}} + 700_{\text{восемь}} + 20_{\text{восемь}} + 5_{\text{восемь}}$$

Затем выделяем круглые числа:

$$\begin{aligned} 3\,725_{\text{восемь}} = {} & 3 \times 1\,000_{\text{восемь}} + \\ & 7 \times 100_{\text{восемь}} + \\ & 2 \times 10_{\text{восемь}} + \\ & 5 \times 1_{\text{восемь}} \end{aligned}$$

или в десятичном представлении:

$$\begin{aligned} 3\,725_{\text{восемь}} = {} & 3 \times 8^3 + \\ & 7 \times 8^2 + \\ & 2 \times 8^1 + \\ & 5 \times 8^0 \end{aligned}$$

Заменяем степени восьмерки их десятичными значениями:

$$\begin{aligned} 3\,725_{\text{восемь}} = {} & 3 \times 512_{\text{десять}} + \\ & 7 \times 64_{\text{десять}} + \\ & 2 \times 8_{\text{десять}} + \\ & 5 \times 1_{\text{десять}} \end{aligned}$$

Выполнив десятичное сложение, получаем $2\,005_{\text{десять}}$. Именно так восьмеричные числа преобразуются в десятичные.

Восьмеричные числа складываются и умножаются точно так же, как и десятичные. Единственное отличие — в использовании других таблиц сложения и умножения. Таблица сложения восьмеричных чисел приведена на стр. 67.

Например, $5_{\text{восемь}} + 7_{\text{восемь}} = 14_{\text{восемь}}$. Восьмеричные числа можно складывать и столбиком.

$$\begin{array}{r} 135 \\ +\,643 \\ \hline 1000 \end{array}$$

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

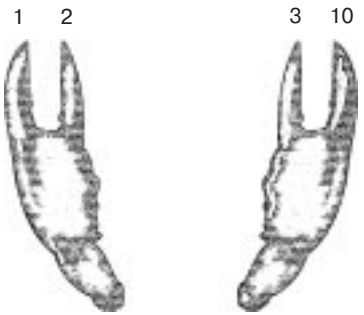
Начнем с правого столбца: 5 плюс 3 равно 10. Пишем 0, а 1 переносим. $1 + 3 + 4 = 10$. Пишем 0, 1 переносим. $1 + 1 + 6 = 10$.

Великое равенство $2 \times 2 = 4$ выполняется и в восьмеричной системе. А вот произведение двух троек равно не 9. А чему же? Оказывается, $3 \times 3 = 11_{\text{восемь}} = 9_{\text{десять}}$. Вот как выглядит таблица умножения в восьмеричной системе.

×	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

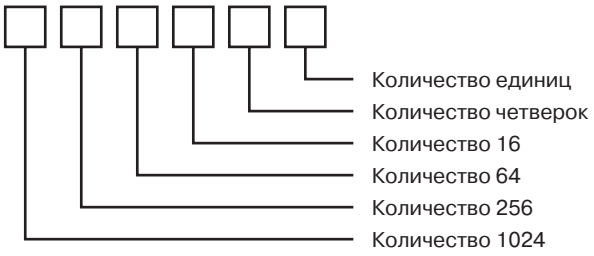
В данном случае $4 \times 6 = 30_{\text{восемь}}$, что эквивалентно числу $24_{\text{десять}}$, равному 4×6 в десятичной системе.

Итак, восьмеричная система не менее обоснована, чем десятичная. Но мы на этом не остановимся. Теперь, когда мы построили систему счисления для мультяшек, поищем что-нибудь подходящее для раков. У раков нет пальцев, но есть клешни. Система счисления для раков основана на 4 и называется *четверичной*.



Счет в этой системе происходит так: 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110 и т. д.

Я не хочу уделять много внимания четверичной системе, так как нам пора двигаться к еще более важным вещам. Посмотрим лишь, как позиции в четверичном числе соответствуют степеням *четырех*.



Четверичное число 31 232 можно записать так:

$$31\,232_{\text{четыре}} = 3 \times 10\,000_{\text{четыре}} + 1 \times 1\,000_{\text{четыре}} + 2 \times 100_{\text{четыре}} + 3 \times 10_{\text{четыре}} + 2 \times 1_{\text{четыре}}$$

так:

$$31\,232_{\text{четыре}} = 3 \times 256_{\text{десять}} + 1 \times 64_{\text{десять}} + 2 \times 16_{\text{десять}} + 3 \times 4_{\text{десять}} + 2 \times 1_{\text{десять}}$$

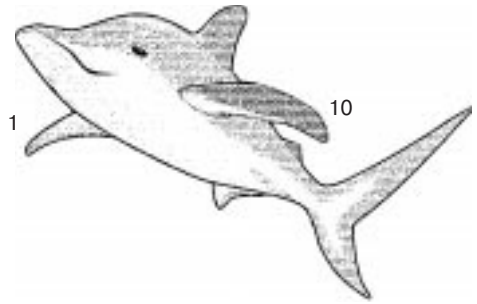
или так:

$$31\,232_{\text{четыре}} = 3 \times 4^4 + 1 \times 4^3 + 2 \times 4^2 + 3 \times 4^1 + 2 \times 4^0$$

Проделав вычисления в десятичной системе, получаем, что $31\,232_{\text{четыре}} = 878_{\text{десять}}$.

Наконец, мы готовы взять последнее, самое серьезное препятствие. Представим себе, что мы дельфины и должны научиться считать с помощью двух плавников. Теперь в нашем распоряжении всего две цифры: 0 и 1. Система счисления, основанная на двух цифрах, называется *двоичной*.

Чтобы привыкнуть к двоичным числам, требуется некоторая практика. Проблема в том, что цифры очень быстро заканчиваются. Смотрите, как считает дельфин.



Да, в двоичной системе сразу после 1 идет 10. Это удивительно, но не так уж неожиданно. В любой системе счисления, когда заканчиваются одноразрядные числа, первым двухразрядным числом является 10. В двоичной системе мы считаем так:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, 10001…

Эти числа кажутся большими, но на самом деле таковыми не являются. Говоря точнее, двоичные числа не большие, а *длинные*.

Число голов у человека равно $1_{\text{десять}}$ или $1_{\text{два}}$.

Число плавников у дельфина равно $2_{\text{десять}}$ или $10_{\text{два}}$.

Число чайных ложек в столовой равно $3_{\text{десять}}$ или $11_{\text{два}}$.

Число сторон у квадрата равно $4_{\text{десять}}$ или $100_{\text{два}}$.

Число пальцев на руке у человека равно $5_{\text{десять}}$ или $101_{\text{два}}$.

Число ног у насекомого равно $6_{\text{десять}}$ или $110_{\text{два}}$.

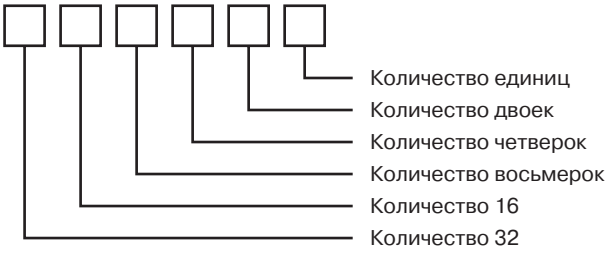
Число дней в неделе равно $7_{\text{десять}}$ или $111_{\text{два}}$.

Число музыкантов в октете равно $8_{\text{десять}}$ или $1000_{\text{два}}$.

Число планет в Солнечной системе равно $9_{\text{десять}}$ или $1001_{\text{два}}$.

Число миллиметров в сантиметре равно $10_{\text{десять}}$ или $1010_{\text{два}}$ и так далее.

В многозначном двоичном числе позиции цифр соответствуют степеням двойки.



Поэтому любое двоичное число, состоящее из 1 с несколькими нулями, является степенью двойки. Показатель степени равен числу нулей. Таблица степеней двойки демонстрирует это правило наглядно.

Степень двойки	**Десятичное**	**Восьмеричное**	**Четверичное**	**Двоичное**
2^0	1	1	1	1
2^1	2	2	2	10
2^2	4	4	10	100
2^3	8	10	20	1000
2^4	16	20	100	10000

(*продолжение*)

2^5	32	40	200	100000
2^6	64	100	1000	1000000
2^7	128	200	2000	10000000
2^8	256	400	10000	100000000
2^9	512	1000	20000	1000000000
2^{10}	1024	2000	100000	10000000000
2^{11}	2048	4000	200000	100000000000
2^{12}	4096	10000	1000000	1000000000000

Рассмотрим в качестве примера число 101101011010. Его можно записать так:

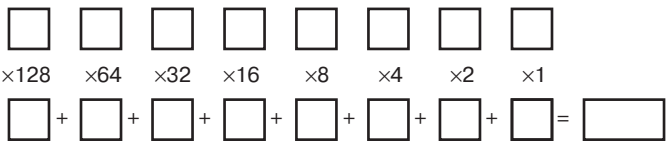
$$\begin{aligned} 101101011010_{\text{два}} = {} & 1 \times 2\,048_{\text{десять}} + \\ & 0 \times 1\,024_{\text{десять}} + \\ & 1 \times 512_{\text{десять}} + \\ & 1 \times 256_{\text{десять}} + \\ & 0 \times 128_{\text{десять}} + \\ & 1 \times 64_{\text{десять}} + \\ & 0 \times 32_{\text{десять}} + \\ & 1 \times 16_{\text{десять}} + \\ & 1 \times 8_{\text{десять}} + \\ & 0 \times 4_{\text{десять}} + \\ & 1 \times 2_{\text{десять}} + \\ & 0 \times 1_{\text{десять}} \end{aligned}$$

или так:

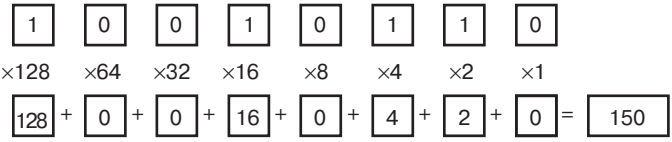
$$\begin{aligned} 101101011010_{\text{два}} = {} & 1 \times 2^{11} + \\ & 0 \times 2^{10} + \\ & 1 \times 2^{9} + \\ & 1 \times 2^{8} + \\ & 0 \times 2^{7} + \\ & 1 \times 2^{6} + \\ & 0 \times 2^{5} + \\ & 1 \times 2^{4} + \\ & 1 \times 2^{3} + \\ & 0 \times 2^{2} + \\ & 1 \times 2^{1} + \\ & 0 \times 2^{0} \end{aligned}$$

Сложив эти числа в десятичной форме, получаем 2 048 + 512 + + 256 + 64 + 16 + 8 + 2 = 2 906$_{\text{десять}}$.

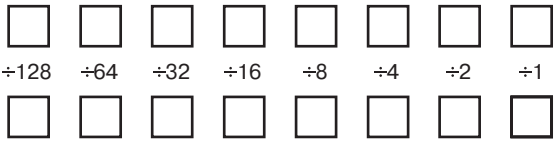
Чтобы облегчить преобразование двоичных чисел в десятичные, используйте созданный мной шаблон.



Этот шаблон позволяет преобразовать двоичные числа длиной в восемь цифр, но его легко продлить. Поместите восьмизначное двоичное число в 8 верхних квадратиков, по одной цифре в каждый квадратик. Проделайте восемь умножений и запишите результаты в восьми нижних квадратиках. Сложите эти результаты и получите искомое число. Допустим, вам нужно найти десятичный эквивалент двоичного числа 10010110.

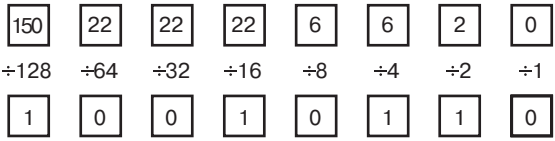


Преобразовать число из десятичной системы в двоичную уже не так просто, но и эту операцию можно проводить по шаблону.



Работать с этим шаблоном сложнее, чем кажется, поэтому внимательно следуйте моим указаниям. Поместите десятичное число (меньшее или равное 255) целиком в верхний левый квадратик. Разделите это число на первый делитель (128) с остатком. Поместите частное в квадратик под делителем, а остаток — в следующий верхний квадратик. Этот остаток станет делимым для следующего вычисления, в котором используется делитель 64. Продолжайте вычисления до конца.

Помните: частное может быть равно либо 0, либо 1. Если делимое меньше делителя, частное равно 0, а остаток — делимому. Если делимое больше или равно делителю, частное равно 1, а остаток равен разности между делимым и делителем. Рассмотрим пример с числом 150.



Складывать и умножать двоичные числа даже легче, чем десятичные. Думаю, вам это понравится. Представьте себе, насколько быстрее вы бы складывали, если бы для этого достаточно было выучить такую маленькую табличку.

+	**0**	**1**
0	0	1
1	1	10

Попробуем с ее помощью сложить два числа:

$$\begin{array}{r} 1100101 \\ +\,0110110 \\ \hline 10011011 \end{array}$$

Начнем с правого столбца: 1 + 0 = 1. Второй столбец справа: 0 + 1 = 1. Третий столбец: 1 + 1 = 0, а 1 переносим. Четвертый столбец: 1 (перенос) + 0 + 0 = 1. Пятый столбец: 0 + 1 = 1. Шестой столбец: 1 + 1 = 0, а 1 переносим. Седьмой столбец: 1 (перенос) + 1 + 0 = 10.

Таблица умножения даже проще таблицы сложения, так как ее можно целиком получить из двух основных правил умножения: умножение любого числа на 0 дает 0; умножение любого числа на 1 дает то же самое число.

×	**0**	**1**
0	0	0
1	0	1

Умножим $13_{\text{десять}}$ на $11_{\text{десять}}$ в двоичном виде:

```
   1101
 x 1011
 ------
   1101
  1101
 0000
1101
--------
10001111
```

Получаем $143_{\text{десять}}$.

Двоичные числа из эстетических соображений часто дополняют нулями спереди (т. е. слева от первой 1), например, 0011 вместо 11. На значение числа это не влияет. В таблице приводятся первые 16 двоичных чисел с их десятичными эквивалентами.

Двоичное	**Десятичное**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Посмотрите на этот список двоичных чисел внимательно, уделив внимание каждому из четырех вертикальных столбцов нулей и единиц. Смотрите, как по мере продвижения вниз по столбцу в нем чередуются цифры:

- в крайнем столбце справа чередуются 0 и 1;
- в следующем столбце чередуются пара 0 и пара 1;
- в следующем столбце чередуются четверка 0 и четверка 1;
- в следующем столбце чередуются восемь 0 и восемь 1.

Чередование *весьма* систематическое, правда? Вы легко напишете и следующие шестнадцать двоичных чисел, добавив к предыдущим числам одну единицу слева.

Двоичное	**Десятичное**
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

Посмотрим на эту же последовательность немного под другим углом. Когда вы считаете в двоичной системе, крайняя цифра справа (ее иногда называют младшим разрядом) поочередно равна 0 и 1. Всякий раз, когда она меняется с 1 на 0, следующая цифра справа также изменяется — с 0 на 1 или с 1 на 0. Это правило распространяется и на следующие разряды: когда двоичная цифра меняется с 1 на 0, следующая за ней также изменяется — либо с 0 на 1, либо с 1 на 0.

Работая с большими десятичными числами, мы часто разделяем пробелами составляющие их цифры на группы по три,

чтобы легче было оценить величину числа. Например, чтобы распознать число 12000000, вам придется считать, сколько в нем цифр. Но разделите его пробелами (12 000 000), и вы сразу поймете, что это 12 миллионов.

Двоичные числа удлиняются очень быстро. Например, 12 миллионов в двоичном представлении пишутся так — 101101110001101100000000. Чтобы сделать это число *немного* более читаемым, отделяйте группы из четырех цифр дефисами (1011-0111-0001-1011-0000-0000) или теми же пробелами (1011 0111 0001 1011 0000 0000). Позже мы рассмотрим более удобный способ записи двоичных чисел.

Сократив число цифр до двух, мы дошли до предельно упрощенной системы счисления, которая является своеобразным мостиком между арифметикой и электричеством. В предыдущих главах мы работали с переключателями, проводами, лампами и реле. Любое из этих устройств может символизировать двоичные цифры 0 и 1.

Провод может быть двоичной цифрой. Если по проводу течет ток, двоичная цифра равна 1. Если тока нет, двоичная цифра принимает значение 0.

Переключатель может быть двоичной цифрой. Если переключатель включен (замкнут), двоичная цифра равна 1. Если выключен (разомкнут), двоичная цифра равна 0.

Лампочка может быть двоичной цифрой. Если лампочка горит, двоичная цифра равна 1. Если не горит, значение двоичной цифры равно 0.

Реле может быть двоичной цифрой. Если реле сработало, двоичная цифра равна 1. Если не сработало, двоичная цифра равна 0.

Для компьютеров двоичные числа — *неисчерпаемый* ресурс.

Примерно в 1948 г. американский математик Джон Уайлдер Таки (John Wilder Tukey) (род. 1915) осознал, что словосочетание «binary digit» (двоичная цифра) по мере распространения компьютеров будет приобретать все большее значение. Он решил заменить его новым, более коротким словом. Рассмотрев такие варианты как *bigit* и *binit*, он остановился на простом, элегантном и симпатичном слове *бит* (bit).

Глава 9

За битом бит

Когда Тони Орландо (Tony Orlando) в 1973 г. просил в песне, чтобы его возлюбленная повязала вокруг дуба желтую ленточку, он не нуждался ни в подробных объяснениях, ни в длительной дискуссии. Ему не нужны были «если», «кроме того» и «тем не менее». В ситуации, о которой рассказывала песня, непременно были бы замешаны сложные чувства, но на самом деле ее герой хотел услышать лишь «да» или «нет». Он хотел, чтобы желтая ленточка, обвязанная вокруг дерева, означала: «Да, ты долго валял дурака и три года просидел в тюрьме, но я все равно хочу, чтобы ты вернулся и жил со мной под одной крышей». А отсутствие ленточки значило бы : «Даже носа своего не показывай».

Как видите, есть лишь две четкие взаимоисключающие возможности. Тони не пел «Обвяжи дерево половиной ленточки, если должна немного подумать» или «Обвяжи дерево синей ленточкой, если больше не любишь меня, но хочешь, чтобы мы остались друзьями». Нет, он выразился очень и очень просто.

Не менее эффективными, чем отсутствие или наличие желтой ленточки, были бы дорожные знаки в воротах — «Въезд разрешен» или «Въезд запрещен» (хотя спеть о них было бы труднее).

Или плакатик на двери — «Закрыто» или «Открыто».

Или фонарик в окне — включенный или выключенный.

Сказать «да» или «нет» можно множеством способов, если это все, что вам нужно сказать. Для этого не нужны предложения или слова, не нужны даже буквы. Все, что требуется, — это *бит* (bit), а под битом я понимаю 0 или 1.

Из прошлых глав мы узнали, что в десятичной системе, которую мы обычно используем для счета, нет ничего особенного. Наша система счисления основана на десяти, потому что именно столько у нас пальцев на руках, — это понятно. Мы с тем же успехом могли положить в основу системы счисления восемь (если бы были персонажами мультфильма), четыре (если бы были раками) и даже два (если бы были дельфинами).

А вот у двоичной системы *есть* особенность: это *простейшая* возможная система счисления. Есть только две двоичные цифры — 0 и 1. Если мы захотим упростить двоичную систему, нам придется избавиться от единицы, и мы останемся с нулем. А с одним нулем ничего толкового не сделаешь.

Слово *бит* (сокращение от *binary digit* — двоичная цифра), — безусловно, одно из самых симпатичных слов, появившихся в связи с компьютерами. Конечно, в английском языке у него есть и обычное значение — «небольшая часть, степень или количество» — и это значение вполне отвечает и компьютерному смыслу бита, поскольку бит — одна двоичная цифра — это и вправду очень мало.

Иногда новое слово появляется за счет получения нового значения. Так и со словом «бит». Бит — это нечто большее, чем просто двоичная цифра, используемая дельфинами для расчетов. В компьютерный век бит обрел значение *основного структурного блока, из которого строится информация*.

Это, конечно, очень самоуверенное заявление. Разумеется, передавать информацию можно не только в форме битов, но буквами, словами, азбукой Морзе, азбукой Брайля и десятичными цифрами. Главное в бите то, что он содержит очень *мало* информации. Один бит информации — это минимально возможное ее количество. Все, что меньше бита, информации вообще не содержит. С другой стороны, поскольку бит представляет собой минимальный объем, более сложную информацию можно передать в нескольких битах. Кстати, говоря о малости содержащейся в бите информации, я вовсе не хочу сказать, что она не важна. Вспомните: желтая ленточка была очень важна для двух людей, которые о ней знали.

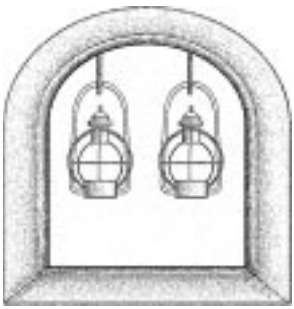
Информацию можно также рассматривать как выбор между двумя или более возможностями. В устной речи, например, вы выбираете слова из всего вашего словарного запаса. Одиночный бит соответствует выбору из двух возможностей (например, «уйди прочь» или «вернись домой»), и две возможности есть, безусловно, минимальное осмысленное их количество. Несколько битов означают выбор из нескольких (больше двух) возможностей. «Послушайте, дети, вместе со мной, О Пола Ревира скачке ночной», — писал Генри Уодсворт Лонгфелло, оставив нам, возможно, не совсем исторически точное описание того, как Пол Ревир (Paul Revere) предупредил американских колонистов о вторжении британцев. Эта история — прекрасный пример того, как с помощью битов можно передать важную информацию:

Коль из города выйдут британцы сегодня
По суше иль морем, — он другу сказал, —
Повесишь фонарь на верху колокольни
Северной церкви, как особый сигнал,—
Один, если сушей, и два, если морем…

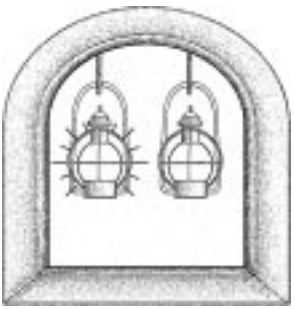
Говоря коротко, у друга Пола Ревира было два фонаря. Если бы британцы двинулись в наступление по суше, он вывесил бы на колокольне один фонарь, если бы наступление началось морем, он вывесил бы оба.

Всех вариантов Лонгфелло явно не указывает. Он ничего не сказал о третьей возможности — британцы вовсе не будут наступать. Лонгфелло предполагает, что этому варианту будет соответствовать *отсутствие* фонарей на колокольне.

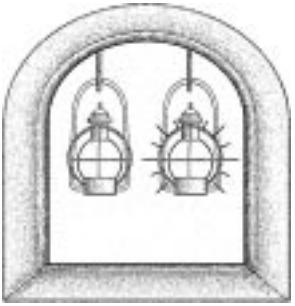
Допустим для простоты, что два фонаря висят на арке колокольни всегда. Обычно они потушены.



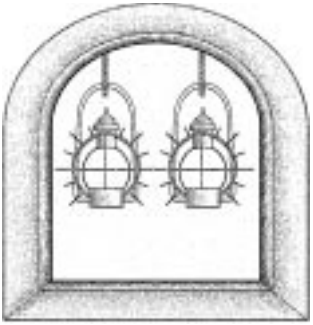
Это означает, что наступление британцев еще не началось. Если горит один фонарь,



или



британцы наступают сушей. Если горят оба фонаря,



британцы наступают морем.

Каждый фонарь — это бит. Зажженный фонарь соответствует 1, потушенный — 0. Тони Орландо доказал нам, что для выбора одной из двух возможностей нужен всего один бит. Если бы Полу Ревиру достаточно было знать лишь о самом факте наступления британцев (но не о его направлении), его другу хватило бы и одного фонаря. Фонарь горит — наступление началось. Фонарь потушен — судьба дарит вам еще один спокойный вечер.

Выбор одной из трех возможностей требует дополнительного светильника. Однако после его добавления становится возможным передать уже один из четырех вариантов:

00 = Британцы не наступают
01 = Наступление началось на суше
10 = Наступление началось на суше
11 = Наступление началось морем

Ограничившись всего тремя вариантами сообщения, Пол Ревир вообще поступил весьма предусмотрительно. На языке теории коммуникаций это называется *избыточной надежностью* (redundancy), необходимой для противодействия *шуму* (noise). Шумом в теории коммуникаций называется все, что мешает связи. Характерный пример — шуршание в телефонной трубке. И все же, несмотря на шум, общение по телефону, как правило, оказывается успешным, поскольку устная речь в высокой степени избыточна. Нам не нужно слышать каждый слог в каждом слове, чтобы понять, о чем идет разговор.

Говоря о фонарях на колокольне, шумом можно называть ночную тьму или расстояние от Пола Ревира до церкви, т. е. факторы, которые могли помешать ему отличить один фонарь от другого. В стихотворении Лонгфелло есть важный фрагмент:

И вот он увидел далекий сигнал —
Мерцанье, а после огонь замигал!
Мгновенье проходит, и он в седле,
Но медлит и видит, как в темной мгле
Второй фонарь вдали замерцал!

Судя по всему, Пол Ревир вряд ли мог узнать, какой именно из двух фонарей был зажжен первым.

Здесь важно понять, что *информация построена на выборе из двух или более возможностей*. Как уже говорилось, в устной речи мы используем слова из словаря. Можно было бы пронумеровать все слова в словаре от 1 до, скажем, 351 482, а затем продолжать разговор, используя номера вместо слов (конечно, обоим участникам разговора понадобятся словари с одинаковой нумерацией и безграничное терпение).

Верно также и то, что *любую информацию, которую можно свести к выбору из двух или более возможностей, можно представить набором битов*. Нет нужды напоминать, что способов общения, не ограниченных выбором из дискретного набора возможностей и жизненно важных для существования рода человеческого, масса. Вот почему у людей не бывает ро-

мантических отношений с компьютерами (надеюсь!). Если вы не можете облечь нечто в слова, изображения или звуки, вы не сможете закодировать эту информацию в виде битов. Да, вероятно, и не захотите.

Большой палец, развернутый вверх или вниз в конце боя гладиаторов, представляет собой один бит информации, два больших пальца, повернутых вверх или вниз, — два бита. Такую систему использовали для оценки новых фильмов кинокритики Роджер Эберт (Roger Ebert) и покойный Джин Сискел (Gene Siskel). Всего в их системе было четыре варианта, представленных следующими парами битов (подробно *мнение* Эберта и Сискела нас сейчас не интересует; только их оценки):

00 = Обоим не понравилось
01 = Сискелу не понравилось; Эберту понравилось
10 = Сискелу понравилось; Эберту не понравилось
11 = Обоим понравилось

Первый бит относится к Сискелу, причем 0 означает, что фильм ему не понравился, а 1 — что понравился. Точно так же второй бит относится к Эберту.

Теперь, если друг спросит вас: «Что думали Сискел и Эберт о фильме *Impolite Encounter*?» — вы, вместо того чтобы ответить «С точки зрения Сискела, большой палец вверх, с точки зрения Эберта большой палец вниз» или даже «Сискелу понравилось; Эберту нет», можете просто сказать: «Один ноль». Если ваш друг знает, какой бит относится к Сискелу, а какой — к Эберту, и понимает, что бит 1 означает «за», а бит 0 — «против», ваш ответ будет ему совершенно понятен. Но сначала вам и вашему другу нужно будет изучить этот шифр.

С тем же успехом мы могли бы приписать биту 1 значение «палец вниз», а биту 0 — «палец вверх». Кому-то это покажется противоестественным. Конечно, мы ожидаем, что бит 1 означает что-то утвердительное, а бит 0 — наоборот, но в реальности присвоение значений происходит вполне произвольно. Единственное требование — все люди, использующие шифр, должны знать, что означают биты 0 и 1.

Значение конкретного бита или группы битов всегда постигается в контексте. Значение желтой ленточки на стволе дуба, вероятно, ясно лишь человеку, который ее там повязал, и тому, кому она предназначается. Измените цвет, дерево или

день, и ленточка превратится в бессмысленный обрывок ткани. Точно так же, чтобы извлечь полезную информацию из жестов Сискела и Эберта, нужно по меньшей мере знать, о каком фильме идет речь.

Если вы ведете список фильмов, отрецензированных Сискелом и Эбертом, с их оценками, вы можете добавить в него еще один бит для учета собственного мнения. После его добавления число возможных вариантов вырастет до восьми.

000 = Сискелу не понравилось; Эберту не понравилось; мне не понравилось
001 = Сискелу не понравилось; Эберту не понравилось; мне понравилось
010 = Сискелу не понравилось; Эберту понравилось; мне не понравилось
011 = Сискелу не понравилось; Эберту понравилось; мне понравилось
100 = Сискелу понравилось; Эберту не понравилось; мне не понравилось
101 = Сискелу понравилось; Эберту не понравилось; мне понравилось
110 = Сискелу понравилось; Эберту понравилось; мне не понравилось
111 = Сискелу понравилось; Эберту понравилось; мне понравилось

Благодаря тому, что для представления информации используются биты, мы можем быть уверены, что рассмотрели все возможности. Мы знаем, что вариантов может быть только 8, не больше и не меньше. Три бита позволяют посчитать от 0 до 7. Других трехзначных двоичных чисел нет.

Теперь, познакомившись с описанием битов Сискела и Эберта, вы вправе задать очень серьезный и непростой вопрос: а как быть со справочником по кино- и видеофильмам Леонарда Мэлтина (Leonard Maltin)? Ведь Леонард Мэлтин не оперирует большими пальцами. Он расставляет фильмам оценки, используя более традиционную систему звездочек.

Чтобы определить, сколько битов нам понадобится для кодирования системы Мэлтина, рассмотрим его систему подробнее. Мэлтин присваивает фильму от 1 до 4 звезд с точностью до половинки звезды (точнее, одну звезду он не присваивает;

настолько плохому фильму присваивается титул «бомба»). Всего в системе 7 оценок, а это означает, что для их представления мы можем ограничиться тремя битами.

000 = Бомба
001 = ★ 1/2
010 = ★★
011 = ★★ 1/2
100 = ★★★
101 = ★★★ 1/2
110 = ★★★★

«А что насчет кода 111?» — спросите вы. Что ж, этот код не значит ничего. Он не определен. Если для представления оценки Мэлтина использован код 111, вы сразу знаете, что произошла ошибка (виновен в ней, вероятно, компьютер, так как люди не ошибаются никогда).

Вспомните: в паре битов, использовавшихся для представления оценок Сискела и Эберта, левый бит был битом Сискела, а правый — Эберта. Значат ли что-нибудь отдельные биты в новой системе? Как сказать… Если вы прибавите 2 к численному значению битового кода, а потом разделите полученное число на 2, то получите число звездочек. Но это возможно лишь потому, что мы использовали логичную и согласованную систему кодирования оценок. Но мы могли определить коды и иначе:

000 = ★★★
001 = ★ 1/2
010 = ★★ 1/2
011 = ★★★★
101 = ★★★ 1/2
110 = ★★
111 = Бомба

Этот шифр имеет столько же прав на существование, сколько и предыдущий, лишь бы все знали, что он означает.

Если бы Мэлтину однажды попался фильм, недостойный даже единственной звезды, он мог бы получить вполовину меньше. И для этой оценки у критика нашелся бы свободный код. Таблица кодов могла бы выглядеть, например, так:

000 = Большая бомба
001 = Бомба
010 = ★ 1/2
011 = ★★
100 = ★★1/2
101 = ★★★
110 = ★★★ 1/2
111 = ★★★★

А вот для включения в рейтинг фильма, который не стоит и половины звезды (0 звезд или «атомная бомба»?), пришлось бы добавлять в систему дополнительный бит. Свободных трехбитовых кодов более не осталось.

Журнал *Entertainment Weekly* расставляет оценки не только фильмам, но и телепрограммам, музыкальным и компьютерным компакт-дискам, книгам, Web-узлам и пр. Оценки варьируются от A+ до F. Пересчитав их, вы обнаружите 13 возможных вариантов. Для их представления понадобится 4 бита.

0000 = F
0001 = D–
0010 = D
0011 = D+
0100 = C–
0101 = C
0110 = C+
0111 = B–
1000 = B
1001 = B+
1010 = A–
1011 = A
1100 = A+

Три кода (1101, 1110 и 1111) остались неиспользованными, полное же их число равно 16.

Говоря о битах, мы часто имеем в виду определенное их *количество*. Чем больше битов мы используем, тем больше число доступных нам вариантов.

Конечно, с десятичными числами дела обстоят точно так же. Например, каково число доступных региональных телефонных кодов? Код региона строится из трех десятичных цифр, поэтому всего может быть 10^3, или 1000 кодов, от 000 до 999.

Сколько семизначных телефонных номеров доступно в регионе с кодом 212? 10^7, или 10000000. Сколько наберется телефонов в регионе с кодом 212, начинающихся с цифр 260? Их будет 10^4, или 10000.

Так и в двоичной системе число возможных кодов определяется как 2 в степени, равной числу битов.

Число битов	**Число кодов**
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

Каждый дополнительный бит удваивает число доступных кодов.

Как посчитать нужное число битов для заданного количества кодов? Иначе говоря, как совершить обратный переход в приведенной таблице?

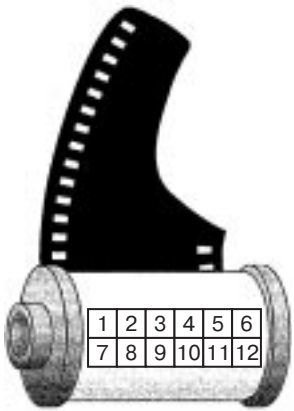
Метод, который можно для этого использовать, заключается в использовании *логарифма по основанию 2*. Логарифмирование обратно возведению в степень. Допустим, мы знаем, что $2^7 = 128$. Значит, логарифм числа 128 по основанию 2 равен 7. На языке математики это записывается так:

$$\log_2 128 = 7.$$

Итак, логарифм по основанию 2 числа 128 равен 7, а логарифм по основанию 2 числа 256 — 8. Чему тогда равен логарифм по основанию 2 числа 200? Если быть точными, то 7,64, но на самом деле такая точность нас не интересует. Для представления 200 различных вариантов с помощью битов нам их понадобится 8.

Как правило, биты скрыты от поверхностного взгляда в глубинах электронных устройств. Вы не увидите их на компакт-диске, в электронных часах или внутри компьютера. Но иногда биты различимы вполне отчетливо.

Приведу один пример. Если у вас есть обычный фотоаппарат с 35-миллиметровой пленкой, возьмите кассету и поверните ее, как показано на рисунке.



Перед вами предстанет набор серебристых и черных квадратов, отдаленно напоминающий шахматную доску. На рисунке я пронумеровал их от 1 до 12. Этот набор называется *DX-кодировкой*. Ее 12 квадратов в действительности представляют 12 битов. Серебристый квадрат соответствует 1, а черный квадрат — 0. Квадраты 1 и 7 всегда серебристые (1).

Что значат эти биты? Вы, возможно, знаете, что пленки различаются по светочувствительности. Ее иногда называют *скоростью* (speed) пленки. Пленка высокой чувствительности считается *быстрой* (fast), поскольку на ней можно производить съемку с очень короткими экспозициями. Скорость пленки измеряется в единицах ASA (American Standards Association, Американская ассоциация по стандартам), причем наиболее популярны пленки с чувствительностью 100, 200 и 400. Чувствительность в единицах ASA не только напечатана на упаковке, но и закодирована на кассете.

Существует 24 стандартных чувствительности для фотопленок. Вот они:

25	32	40
50	64	80
100	125	160
200	250	320
400	500	640
800	1000	1250
1600	2000	2500
3200	4000	5000

Сколько битов нужно, чтобы закодировать чувствительность пленки? Ответ прост — 5. Мы знаем, что $2^4 = 16$ — этого слишком мало. А вот $2^5 = 32$ — больше, чем достаточно.

Соответствие между квадратами на кассете и чувствительностью пленки показано в таблице.

Квадрат 2	**Квадрат 3**	**Квадрат 4**	**Квадрат 5**	**Квадрат 6**	**Чувствительность**
0	0	0	1	0	25
0	0	0	0	1	32
0	0	0	1	1	40
1	0	0	1	0	50
1	0	0	0	1	64
1	0	0	1	1	80
0	1	0	1	0	100
0	1	0	0	1	125
0	1	0	1	1	160
1	1	0	1	0	200
1	1	0	0	1	250
1	1	0	1	1	320
0	0	1	1	0	400
0	0	1	0	1	500
0	0	1	1	1	640
1	0	1	1	0	800
1	0	1	0	1	1000
1	0	1	1	1	1250

(*продолжение*)

0	1	1	1	0	1600
0	1	1	0	1	2000
0	1	1	1	1	2500
1	1	1	1	0	3200
1	1	1	0	1	4000
1	1	1	1	1	5000

Эти коды используются в большинстве современных 35-миллиметровых фотоаппаратов. Если на вашем фотоаппарате выдержка или тип пленки устанавливаются вручную, эти коды в нем не применяются. Если же коды используются в вашем аппарате, присмотритесь к нему в следующий раз, когда будете вставлять пленку. Вы увидите шесть металлических контактов, соответствующих квадратам с 1 по 6 на кассете. Серебристые квадраты — это просто открытая металлическая поверхность кассеты, которая является проводником. Черные квадраты покрыты краской — она электрического тока не проводит.

Электрическая схема фотоаппарата построена так, что ток подводится к первому квадрату на кассете (он всегда серебристый). Этот ток будет (или не будет) проведен пятью контактами на квадратах со 2 по 6, в зависимости от того, окрашены они изолирующей краской или нет. Так, если ток присутствует на контактах 4 и 5, но отсутствует на контактах 2, 3 и 6, в фотоаппарат вставлена пленка 400 ASA. При съемке выдержка будет установлена автоматически.

В недорогих фотоаппаратах считываются только квадраты 2 и 3, а чувствительность пленки считается равной 50, 100, 200 или 400 единицам ASA.

Квадраты с 8 по 12 в большинстве аппаратов также не используются. В квадратах 8, 9 и 10 зашифровано число кадров на пленке, а квадраты 11 и 12 содержат сведения о том, черно-белая пленка или цветная и позитивная или негативная.

Вероятно, чаще всего вы сталкиваетесь с двоичными числами в коде UPC (Universal Product Code, универсальный код продукта), или просто штрих-коде, — наборе черных штрихов, который в наши дни присутствует практически на любой упаковке. Штрих-код — это лучший символ того, насколько компьютеры внедрились в нашу жизнь.

Хотя у некоторых людей штрих-код вызывает приступы паранойи, это совершенно безобидная вещь, изобретенная для автоматизации розничной торговли и учета товаров. Со своей задачей он справляется вполне успешно. Благодаря ему, например, современные кассовые аппараты выдают покупателю чек, в котором подробно расписаны все его покупки, чего без штрих-кода сделать было бы нельзя.

Нас же в первую очередь интересует то, что код UPC является двоичным, хотя на первый взгляд этого и не скажешь. Давайте разберемся, как строится штрих-код и как он работает.

Чаще всего встречается штрих-код, состоящий из нескольких цифр и 30 вертикальных полосок различной толщины, разделенных пустыми интервалами также переменной толщины. Рассмотрим в качестве примера штрих-код, нанесенный на банку с «Супом куриным с вермишелью» фирмы Campbell Soup.

Возникает искушение разделить код UPC на тонкие и толстые полоски, узкие и широкие промежутки, и это действительно один из способов разобраться в его структуре. Черные полоски и пустые промежутки штрих-кода бывают четырех различных ширин.

Но нам удобнее рассматривать UPC как набор битов. Имейте в виду, что сканирующему устройству у кассира нет нужды просматривать штрих-код целиком, и уж тем более неспособно оно интерпретировать цифры в его основании, поскольку это потребовало бы применения сложной компьютерной технологии *распознавания символов* (optical character recognition, OCR). Сканеру достаточно увидеть тонкий срез штрих-кода. Код UPC делают таким большим просто для того, чтобы кассиру легче было «нацелить» на него сканер. Срез, попадающий в сканер, выглядит так.

Похоже на азбуку Морзе, правда?

Сканируя эту информацию слева направо, компьютер присваивает бит 1 первой встреченной черной полоске, и бит 0 первому промежутку. Следующие промежутки и штрихи считываются как последовательности одного, двух, трех или четырех битов в зависимости от ширины штриха или промежутка. В битовом представлении данный штрих-код выглядит так:

1010001101011000100110010001101000110100011010101011100101100110110110010011101100101000100101

Итак, штрих-код представляет собой набор 95 битов. В данном примере эти биты можно разбить на следующие группы.

Биты	**Значение**
101	Левый контрольный узор
0001101 0110001 0011001 0001101 0001101 0001101	Цифры с левой стороны
01010	Центральный контрольный узор
1110010 1100110 1101100 1001110 1100110 1000100	Цифры с правой стороны
101	Правый контрольный узор

Первые три бита — всегда 101. Они называются *левым контрольным узором* (left-hand guard pattern) и нужны для того, чтобы настроить сканирующее устройство. По контрольному узору сканер определяет ширину штриха и промежутка, соответствующую одному биту. Иначе на всех упаковках код UPC пришлось бы делать одного и того же размера.

За левым контрольным узором следует шесть групп по 7 битов в каждой. В них закодированы десятичные цифры от 0 до 9, в чем мы убедимся чуть позже. Затем идет 5-битовый центральный контрольный узор — фиксированная группа битов (всегда 01010), используемая как встроенная защита от оши-

бок. Не найдя центрального контрольного узора там, где он должен быть, сканер считает штрих-код неверным. Это один из нескольких способов выявить плохо напечатанный или подделанный штрих-код.

За центральным контрольным узором идут еще шесть 7-битовых групп и правый контрольный узор (всегда 101). Позже я объясню, как контрольные узоры позволяют сканировать штрих-код как слева направо, так и справа налево.

Всего в коде UPC зашифровано 12 десятичных цифр. Шесть из них закодированы с его левой стороны, по 7 битов в каждой. Для их расшифровки можете использовать такую таблицу.

Левосторонние коды

0001101 = 0	0110001 = 5
0011001 = 1	0101111 = 6
0010011 = 2	0111011 = 7
0111101 = 3	0110111 = 8
0100011 = 4	0001011 = 9

Заметьте: каждый 7-битовый код начинается с 0 и кончается 1. Натолкнувшись на 7-битовый код, который начинается с 1, а кончается 0, сканер понимает, что код UPC либо неверно прочитан, либо подделан. Кроме того, в каждом коде группы единиц встречаются лишь дважды. Это значит, что каждая десятичная цифра в коде UPC зашифрована двумя вертикальными штрихами.

Еще одна особенность кодов в этой таблице — нечетное число 1 в каждом из них. Она также позволяет проверить правильность штрих-кода — это так называемый контроль *четности* (parity).

Для интерпретации шести 7-битовых кодов в правой части UPC используйте следующую таблицу.

Правосторонние коды

1110010 = 0	1001110 = 5
1100110 = 1	1010000 = 6
1101100 = 2	1000100 = 7
1000010 = 3	1001000 = 8
1011100 = 4	1110100 = 9

Эти коды являются дополнительными по отношению к предыдущим: там, где в левосторонних кодах был 0, теперь стоит 1, и наоборот. Правосторонние коды всегда начинаются с 1 и заканчиваются 0. Кроме того, число битов 1 в них всегда четно, что можно применять для контроля четности.

Теперь мы окончательно готовы к расшифровке UPC. С помощью двух приведенных выше таблиц мы можем определить 12 цифр, зашифрованных на банке куриного супа с вермишелью фирмы Campbell Soup емкостью 10 3/4 унции. Вот они:

0 51000 01251 7

Какое разочарование! Да ведь это те самые цифры, которые и без того напечатаны под штрих-кодом! Да, и это очень удобно: если сканер по каким-то причинам не смог прочитать код, кассир может ввести его вручную. Вы наверняка неоднократно видели, как это происходит. Конечно, получается, что весь наш труд по расшифровке штрих-кода был напрасным, к тому же никакой секретной информации мы так и не получили: просто 30 вертикальных штрихов превратились в 12 цифр.

Первая цифра (в данном случае 0) символизирует тип кода. 0 означает, что код является обычным кодом UPC. Если код нанесен на упаковку с товаром переменного веса, например, мясом или овощами, он начинается с 2. Купоны на скидки обозначаются цифрой 5.

Следующие 5 цифр — это код производителя. В нашем примере код 51000 соответствует компании Campbell Soup. Его несут на себе все продукты с маркой Campbell. За ними следует пятизначный (01251) код конкретного продукта данной компании, в данном случае код банки с куриным супом емкостью 10 3/4 унции. Код продукта имеет смысл лишь в сочетании с кодом производителя. У куриного супа с вермишелью, произведенного другой компанией, будет другой код продукта, в свою очередь код 01251 может значить нечто совершенно иное у другого производителя.

Вопреки распространенному мнению в код UPC не включается цена продукта. Информация о ней извлекается из компьютерной базы данных, которую магазин использует в сочетании со сканерами у кассовых аппаратов.

Последняя цифра (в нашем случае 7) называется *символом проверки остатка* (modulo check character) и тоже использует-

ся для исключения ошибок. Чтобы проверить его в деле, присвоим каждой из первых 11 цифр (0 51000 01251 в нашем примере) букву:

A BCDEF GHIJK

Теперь вычислим следующее выражение:

$$3 \times (A + C + E + G + I + K) + (B + D + F + H + J)$$

и вычтем результат из ближайшего большего числа, кратного десяти. Полученное число и будет символом проверки остатка. Для куриного супа с вермишелью Campbell:

$$3 \times (0 + 1 + 0 + 0 + 2 + 1) + (5 + 0 + 0 + 1 + 5) =$$
$$= 3 \times 4 + 11 = 23$$

Ближайшее большее число, кратное десяти, — 30. Далее:

$$30 - 23 = 7$$

Это число напечатано под штрих-кодом и зашифровано в нем. Используется проверка остатка для вящей надежности. Если остаток, вычисленный по штрих-коду, не совпадет с остатком, явно указанным в нем, штрих-код не будет считаться прочитанным.

Вообще для представления десятичной цифры от 0 до 9 достаточно 4 битов. С другой стороны, в UPC их используется 7. Всего в штрих-коде 11 осмысленных десятичных цифр закодировано 95 битами. А если учесть, что UPC с обеих сторон выделен пустым пространством, эквивалентным 9 нулевым битам, получается, что во всем штрих-коде 11 цифр закодировано 113 битами, по 10 бит на цифру!

Как мы уже видели, избыток битов частично применяется для надежности считывания. От кода продукта было бы мало толку, если бы его можно было в два счета исправить обычным фломастером.

Кроме того, благодаря им UPC можно считывать в обоих направлениях. Если в первых считанных цифрах количество единиц четно, сканер распознает, что код читается справа налево. Для расшифровки правосторонних цифр компьютер использует следующую таблицу.

Правосторонние коды в обратном порядке

0100111 = 0	0111001 = 5
0110011 = 1	0000101 = 6
0011011 = 2	0010001 = 7
0100001 = 3	0001001 = 8
0011101 = 4	0010111 = 9

Для расшифровки левосторонних цифр используется следующая таблица.

Левосторонние коды в обратном порядке

1011000 = 0	1000110 = 5
1001100 = 1	1111010 = 6
1100100 = 2	1101110 = 7
1011110 = 3	1110110 = 8
1100010 = 4	1101000 = 9

Эти 7-битовые коды отличаются от кодов, считываемых слева направо. Никакой путаницы не возникает.

Знакомство с кодами в этой книге началось с азбуки Морзе, составленной из точек, тире и промежутков между ними. Азбука Морзе, на первый взгляд, имеет мало общего с нулями и единицами, тем не менее, на деле это почти одно и то же.

Вспомните правила азбуки Морзе. Тире втрое длиннее точки. Точки и тире в пределах одной буквы разделены паузами продолжительностью в одну точку. Промежутки между буквами по длительности равны одному тире. Слова разделяются паузами в два тире.

Чтобы немного упростить анализ, допустим, что длина тире превышает длину точки не в 3, а в 2 раза. Это означает, что точка соответствует одному единичному биту, а тире — двум единичным битам. Паузы составляются из нулевых битов.

Вот как выглядит таблица с расшифровкой азбуки Морзе для латинских букв:

A	·–	J	·–––	S	···
B	–···	K	–·–	T	–
C	–·–·	L	·–··	U	··–
D	–··	M	––	V	···–
E	·	N	–·	W	·––
F	··–·	O	–––	X	–··–
G	––·	P	·––·	Y	–·––
H	····	Q	––·–	Z	––··
I	··	R	·–·		

А вот та же таблица, преобразованная в биты.

A	101100	J	101101101100	S	1010100
B	1101010100	K	110101100	T	1100
C	11010110100	L	1011010100	U	10101100
D	11010100	M	1101100	V	1010101100
E	100	N	110100	W	101101100
F	1010110100	O	1101101100	X	11010101100
G	110110100	P	10110110100	Y	110101101100
H	101010100	Q	110110101100	Z	11011010100
I	10100	R	10110100		

Заметьте: все коды начинаются с 1 и кончаются парой 0, представляющей паузу между буквами в пределах одного слова. Кодом пробела между словами является дополнительная пара 0. Таким образом, на азбуке Морзе фраза «hi there» выглядит так:

но, представив ее в битах, мы получим нечто, очень похожее на срез штрих-кода:

101010100101000011001010101001001011010010000

С точки зрения битов азбука Брайля гораздо проще азбуки Морзе. Шрифт Брайля является 6-битовым кодом. Каждый символ представляется набором из шести точек, каждая из которых может быть выпуклой или плоской. Как я объяснял в главе 3, точки обычно нумеруются с 1 до 6.

Слово «code», например, представляется такими символами азбуки Брайля:

Если заменить выпуклую точку на 1, а плоскую — на 0, любой символ азбуки Брайля можно представить 6-битовым двоичным числом. Четыре символа Брайля для букв из слова «code» будут выглядеть так:

100100 101010 100110 100010

где самый левый бит соответствует первой позиции в наборе, а самый правый — шестой позиции.

Позже мы узнаем, что с помощью битов можно зашифровать не только коды товаров, чувствительность пленки, художественную ценность фильма, способ наступления британской армии или послание любимой женщины, но и любые слова, изображения, звуки, музыку и кино. Но в основе своей биты — это числа. Для представления информации в форме битов достаточно пересчитать количество доступных возможностей. Это количество определяет, сколько битов понадобится для того, чтобы присвоить каждой возможности уникальный номер.

Биты также играют важную роль в *логике* — странном сплаве философии и математики, главная цель которого заключается в определении истинности или ложности некоего утверждения. Истину и ложь также можно обозначить через 1 и 0.

Глава 10

Логика и переключатели

Что есть истина? Аристотель считал, что ответить на этот вопрос можно с помощью логики. Собрание его поучений, известное как *Органон* (датируемое IV в. до н. э.), — самое раннее подробное произведение о логике. Для древних греков логика была средством для анализа языка в поисках истины и потому считалась отраслью философии. В основе аристотелевой логики лежит *силлогизм*. Самый известный силлогизм (кстати, в трудах Аристотеля его нет) звучит так:

Все люди смертны;
Сократ — человек;
Следовательно, Сократ смертен.

В силлогизме из двух истинных посылок выводится третья.

Смертность Сократа и без силлогизма достаточно очевидна, но приведенным выше примером круг силлогизмов не ограничен. Задумайтесь, например, над следующими двумя посылками, предложенными математиком XIX века Чарльзом Доджсоном, носившем также псевдоним Льюис Кэрролл:

Все философы логичны;
Нелогичный человек всегда упрям.

Из этой пары вывести верное заключение уже не так просто. Кстати, звучит оно так: «Некоторые упрямые люди не являются философами» (заметьте, что в заключении появилась неопределенность, выраженная словом «некоторые»).

В течение двух тысяч лет математики сражались с логикой Аристотеля, пытаясь обуздать ее математическими символами и операторами. До XIX в. подойти близко к решению этой проблемы удалось лишь Готфриду Вильгельму Лейбницу (1648–1716), который в начале своей научной деятельности увлекся было логикой, но затем заинтересовался другими проблемами (например, независимо от Ньютона и одновременно с ним изобрел дифференциальное исчисление).

А затем наступило время Джорджа Буля.

Джордж Буль (George Boole) появился на свет в Англии в 1815 г. Шансы на успех в жизни у него были весьма невелики. Он родился в семье башмачника и бывшей горничной, и жесткая классовая структура Британии предполагала, что сам он также не достигнет больших высот. Но благодаря своему пытливому уму и помощи отца (серьезно увлекавшегося математикой и литературой), юный Джордж преуспел в науках, которые считались привилегией мальчиков из высших классов, в том числе в латыни, греческом и математике. Написав несколько научных статей по математике, в 1849 г. Буль стал первым профессором математики Колледжа Королевы в ирландском городе Корк.

Над математической формулировкой законов логики работали в середине XIX в. многие математики (наибольших успехов добился Огастес Морган), но только Булю удалось совершить концептуальный прорыв в этой области, сначала в короткой книжке «Математический анализ логики, или очерк исчисления дедуктивного рассуждения» (1847), а затем в более объемном и амбициозном труде «Исследование законов мышления, на которых основаны математические теории логики и вероятностей» (1854), который часто сокращенно называют просто «Исследование законов мышления». Умер Буль в 1864 г. от воспаления легких, которое он подхватил, попав под дождь по дороге на занятия. Ему было всего 49 лет.

Название основного труда Буля предполагает решение грандиозной задачи. Рациональное мышление происходит по законам логики, а значит, если мы найдем способ описать эти законы средствами математики, тем самым мы получим математическое описание работы мозга. Конечно, в наши дни это представление кажется довольно-таки наивным (возможно, мы до него все еще не доросли!).

По внешнему виду и действию изобретенная Булем алгебра подобна обычной алгебре. В последней для обозначения чисел используются *операнды* (как правило, буквы латинского алфавита), а для указания способа объединения чисел — операторы (например, знаки + или ×). Традиционная алгебра используется, например, для решения таких задач. У Ани было 3 яблока, у Бетти в 2 раза больше, а у Кармен на 5 яблок больше, чем у Бетти. Наконец, у Дейдры было яблок в 3 раза больше, чем у Кармен. Сколько яблок было у Дейдры?

Чтобы решить эту задачу, мы должны заменить слова арифметическими выражениями, подставив вместо имен буквы латинского алфавита:

$$A = 3$$
$$B = 2 \times A$$
$$C = B + 5$$
$$D = 3 \times C$$

Подстановками эти выражения можно объединить в одно, а затем вычислить его:

$$D = 3 \times C$$
$$D = 3 \times (B + 5)$$
$$D = 3 \times ((2 \times A) + 5)$$
$$D = 3 \times ((2 \times 3) + 5)$$
$$D = 33$$

Вычисляя алгебраические выражения, мы следуем определенным правилам. Эти правила настолько глубоко проникли в наши повседневные расчеты, что мы даже не принимаем их за правила и не помним их названий. И все же они положены в основу любых вычислений.

Первое правило заключается в том, что операции сложения и умножения *коммутативны*. Это значит, что их операнды можно менять местами и результат не изменится:

$$A + B = B + A$$
$$A \times B = B \times A$$

А вот операции вычитания и деления коммутативными не являются.

Сложение и умножение также являются *ассоциативными*, т. е.:

$$A + (B + C) = (A + B) + C$$
$$A \times (B \times C) = (A \times B) \times C$$

Наконец, умножение *дистрибутивно* относительно сложения:

$$A \times (B + C) = (A \times B) + (A \times C)$$

Другая характерная особенность обычной алгебры в том, что она всегда работает с числами: количествами, весами, расстояниями, возрастами и пр. Понадобился гений Буля, чтобы сделать алгебру более абстрактной, отделив ее от концепции числа. В булевой алгебре (так стали называть алгебру, разработанную Булем) операнды обозначают не числа, а *множества*, т. е. наборы чего угодно.

Поговорим о кошках. Кошки бывают мужского и женского пола. Для удобства будем обозначать котов буквой М, а кошек — Ж. Не забывайте, что эти буквы *не* обозначают количество кошек. Число котов и кошек непрерывно меняется — рождаются новые котята, а старые кошки (увы) отходят в мир иной. Буквы М и Ж символизируют множества кошек с определенными свойствами. Чтобы обозначить множество всех котов, достаточно сказать просто «М».

Буквами можно обозначить и окрас кошки. Например, буква Р может соответствовать рыжим кошкам, буква Ч — черным, Б — белым. В множество Д можно определить кошек всех других цветов, т. е. не входящих в множества Р, Ч и Б.

Наконец, кошка может быть стерилизованной или нестерилизованной. Пусть первых обозначает буква С, а вторых — буква Н.

В обычной (численной) алгебре операторы + и × символизируют сложение и умножение. В булевой алгебре эти знаки также используются, что может стать причиной некоторой путаницы. Как складывать и умножать обычные числа, знают практически все, а вот можно ли складывать и умножать *множества*?

Если говорить точно, то не совсем. В действительности знаки + и × в булевой алгебре имеют несколько иное значение.

Символом + в булевой алгебре обозначается *объединение* (union) двух множеств, т. е. множество, состоящее из всех элементов первого множества и всех элементов второго множества. Например, в множество Ч + Б входят все черные и белые кошки.

Символ × в булевой алгебре соответствует *пересечению* (intersection) двух множеств, т. е. множеству, которое содержит только элементы, входящие *как в первое, так и во второе* множество. Например, множество Ж × Р состоит из рыжих кошек женского пола. Как и умножение в обычной алгебре, пересечение двух множеств можно записать в виде Ж · Р или просто ЖР (такую запись предпочитал сам Буль).

Чтобы избежать путаницы между обычными и булевыми алгебраическими операциями, вместо + и × объединение и пересечение иногда обозначают знаками ∪ и ∩. Однако значение нововведений Буля для математики отчасти связано с тем, что он придал знакомым операторам более общий смысл, поэтому я решил присоединиться к его мнению и не вводить в булеву алгебру новые символы.

Коммутативный, ассоциативный и дистрибутивный законы действуют и в булевой алгебре. Более того, в ней оператор + дистрибутивен относительно оператора ×. Вот такое выражение в обычной алгебре несправедливо:

$$Б + (Ч \times Ж) = (Б + Ч) \times (Б + Ж)$$

Объединение белых кошек с черными кошками женского пола содержит те же элементы, что и пересечение двух объединений: белых кошек с черными и белых — с кошками женского пола. Звучит, конечно, не очень внятно, но так оно и есть на самом деле.

Чтобы завершить описание булевой алгебры, нужно ввести еще два символа. Выглядят они обычными цифрами, но таковыми на самом деле не являются, поскольку иногда ведут себя не так, как цифры. Символ 1 в булевой алгебре означает все, о чем идет речь, например, в нашем случае он означает множество всех кошек без исключения. Таким образом:

$$М + Ж = 1$$

Иначе говоря, в объединение кошек мужского и женского пола попадают вообще все кошки. Все кошки содержатся также в объединении множеств рыжих кошек, черных кошек, белых кошек и кошек других окрасов:

$$Р + Ч + Б + Д = 1$$

Построить множество, содержащее всех кошек, можно и так:

$$С + Н = 1$$

Если символ 1 используется в сочетании со знаком «минус», он означает все, *кроме* чего-то. Так, множество:

$$1 – М$$

содержит всех кошек, кроме кошек мужского пола. Понятно, что в это множество входят кошки женского пола:

$$1 – М = Ж$$

Последний символ, который нам понадобится, — 0. В булевой алгебре он означает пустое множество, т. е. множество, не содержащее ни одного элемента. Пустое множество получается в результате пересечения неперекрывающихся множеств, например, множеств кошек женского и мужского пола:

$$Ж \times М = 0$$

Заметьте: иногда символы 1 и 0 в булевой алгебре ведут себя подобно своим аналогам в обычной алгебре. Например, пересечение всех кошек с кошками женского пола есть множество кошек женского пола:

$$1 \times Ж = Ж$$

Пересечение пустого множества с множеством кошек женского пола есть пустое множество:

$$0 \times Ж = 0$$

Объединение пустого множества с множеством кошек женского пола есть множество кошек женского пола:

$$0 + Ж = Ж$$

Но иногда результат получается не такой, как в обычной алгебре. Например, объединение множеств всех кошек и кошек женского пола является множеством всех кошек:

$$1 + Ж = 1$$

В обычной алгебре такое выражение особого смысла не имеет.

Поскольку множество Ж содержит всех кошек женского пола, а множество (1 – Ж) — всех кошек противоположного пола, объединение двух этих множеств равно 1:

$$Ж + (1 - Ж) = 1$$

а их пересечение — 0:

$$Ж \times (1 - Ж) = 0$$

Это выражение сыграло важную роль в истории алгебры логики, поэтому у него есть собственное название — закон противоречия. Он гласит, что ничто не может быть одновременно собой и своей противоположностью.

А вот в следующем выражении уже проявляется отличие булевой алгебры от обычной:

$$Ж \times Ж = Ж$$

В булевой алгебре это выражение исполнено смысла: пересечение множества кошек женского пола с самим собой равно тому же самому множеству. Если мы подставим вместо множества обычное число, смысл выражения потеряется. Буль считал, что его алгебра отличается от обычной только следующим выражением:

$$X^2 = X$$

Еще одно выражение, которое с точки зрения обычной алгебры выглядит странно:

$$Ж + Ж = Ж$$

объединение множества кошек женского пола с самим собой равно тому же самому множеству.

В булевой алгебре предлагается математический способ разрешения аристотелева силлогизма. Вспомним две его первые строки:

Все люди смертны;
Сократ — человек.

Обозначим буквой Л множество всех людей, буквой Х — множество смертных существ, а буквой К — множество Сократов. Что означает высказывание «Все люди смертны»? Оно означает, что пересечение множества людей со множеством смертных существ равно множеству людей:

$$Л \times Х = Л$$

А вот выражение Л × Х = Х было бы неверным, поскольку в множество смертных существ помимо людей входят кошки, собаки и даже пальмы.

Высказывание «Сократ — человек» можно передать следующим образом: пересечение множества Сократов (в нем элементов не так много) со множеством людей (в нем элементов гораздо больше) равно множеству Сократов

$$К \times Л = К$$

Из первого выражения мы знаем, что множество Л равно Л × Х. Подставим это во второе выражение:

$$К \times (Л \times Х) = К$$

Используя ассоциативный закон, переписываем это в виде:

$$(К \times Л) \times Х = К$$

Но, как мы знаем, пересечение К × Л равно К, а значит:

$$К \times Х = К$$

Готово! Эта формула говорит о том, что пересечение множества Сократов со множеством смертных существ равно множеству Сократов, а это как раз и означает, что Сократ смертен. Равенство этого же выражения нулю означало бы, что Сократ не является смертным. Если бы мы нашли, что пересечение К × Х равно Х, то пришли бы к выводу, что Сократ — единственное смертное существо, а все остальные существа бессмертны!

Пока что у вас могло сложиться впечатление, что мы стреляем из пушки по воробьям, доказывая математическими методами смертность Сократа (особенно учитывая, что сам Сократ убедительно доказал свою смертность 2 400 лет назад). Но теми же методами можно проверять выполнение сложного набора условий. Представьте себе, заходите вы однажды в зоомагазин и говорите продавцу: «Мне нужна кошка мужского пола, стерилизованная, белая или рыжая; или кошка женского пола, стерилизованная, любого цвета, кроме белого; или любая кошка черного окраса». А продавец вам отвечает: «А-а, так вам нужна кошка из множества, описываемого выражением:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

Верно?» И вы отвечаете: «Да! Именно так!»

Чтобы проверить, правильно ли продавец написал выражение, забудем на время о понятиях объединения и пересечения и будем использовать вместо них операторы ИЛИ (OR) и И (AND). Я специально написал их прописными буквами, чтобы не путать их с обычными словами. Формируя объединение двух множеств, вы в действительности выбираете элементы из одного множества ИЛИ из другого множества. В пересечение вы включаете только элементы, входящие в первое множество И во второе множество. Кроме того, вместо единицы со знаком «минус» можно использовать оператор НЕ (NOT). Итак:

- вместо оператора объединения + пишем ИЛИ;
- вместо оператора пересечения × пишем И;
- вместо 1 – (все за исключением чего-то) пишем НЕ.

В современных текстах вместо И и ИЛИ иногда используют знаки $\wedge$ и $\vee$.

С учетом этих замен выражение приобретает вид:

(М И С И (Б ИЛИ Р)) ИЛИ (Ж И С И (НЕ Б)) ИЛИ Ч

Это условие почти совпадает с тем, что вы с самого начала выразили словами. Обратите внимание на скобки, которые существенно прояснили ваше пожелание. Итак, вам нужна кошка, входящая в одно из трех множеств:

(М И С И (Б ИЛИ Р))

ИЛИ

(Ж И С И (НЕ Б))

ИЛИ

Ч

Записав эту формулу, продавец может осуществить проверку выполнения ваших условий. А вы и не заметили, как я потихонечку перешел к другой форме булевой алгебры! В ней буквы уже не обозначают множеств, и им теперь можно присваивать численные значения. Фокус в том, что выбор допустимых значений очень ограничен: 0 или 1. 1 означает «истина»: да, эта кошка удовлетворяет данному условию. 0 означает «ложь»: нет, эта кошка не удовлетворяет данному условию.

Для начала продавец приносит вам нестерилизованного рыжего кота. Условие покупки кошки выглядит так:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

а если мы подставим вместо отдельных условий 0 и 1, то так:

$$(1 \times 0 \times (0 + 1)) + (0 \times 0 \times (1 - 0)) + 0$$

Обратите внимание, что единице равны только условия М и Р, поскольку продавец принес рыжую кошку мужского пола.

Теперь нужно вычислить значение этого выражения. Если оно равно 1, кошка удовлетворяет вашим условиям, 0 — нет. Проводя вычисления, помните, что наши действия лишь внешне напоминают сложение и умножение, но по сути ими не являются. Впрочем, к ним применимо большинство правил обычной алгебры.

При умножении чисел по правилам булевой алгебры возможны следующие результаты:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Иными словами, результат равен 1, только если левый И правый операнд оба равны 1. Эта операция работает точно так же, как обычное умножение. Составим для ее обобщения небольшую таблицу, как мы делали в главе 8.

И	**0**	**1**
0	0	0
1	0	1

В логическом сложении возможные результаты таковы:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Результат равен 1, если левый ИЛИ правый операнд равны 1. И эта операция мало отличается от обычного сложения за исключением того, что сумма двух единиц также равна 1. Построим таблицу и для операции ИЛИ.

ИЛИ	**0**	**1**
0	0	1
1	1	1

С помощью этих таблиц легко вычислить значение выражения:

$$(1 \times 0 \times 1) + (0 \times 0 \times 1) + 0 = 0 + 0 + 0 = 0$$

Нулевой результат означает, что эта кошка вам не подойдет.

Тогда продавец предлагает вам белую стерилизованную кошечку. Исходное выражение выглядит так:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

Подставляем в него 0 и 1:

$$(0 \times 1 \times (1 + 0)) + (1 \times 1 \times (1 - 1)) + 0$$

Это равно:

$$(0 \times 1 \times 1) + (1 \times 1 \times 0) + 0 = 0 + 0 + 0 = 0$$

Эту бедную киску вы тоже отвергнете…

Наконец, продавец приносит дымчатую стерилизованную самку (ее цвет попадает в категорию Д — другой, т. е. не белый, не черный, не рыжий). Выражение:

$$(0 \times 1 \times (0 + 0)) + (1 \times 1 \times (1 - 0)) + 0$$

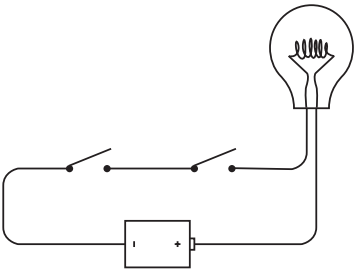
сводится к виду:

$$(0 \times 1 \times 0) + (1 \times 1 \times 1) + 0 = 0 + 1 + 0 = 1$$

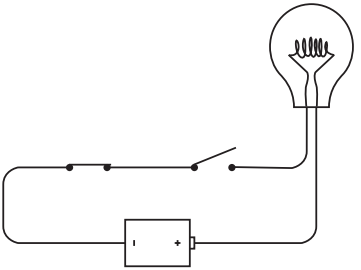
Окончательный результат равен 1 — кошка нашла свой новый дом!

Вечером того же дня кошка дремлет, свернувшись в клубок у вас на коленях, а вы размышляете, можно ли соорудить из нескольких тумблеров и лампочки устройство, которое определяло бы, подходит вам данная кошка или нет (ничего не скажешь, ребенок вы странноватый). При этом вы не полностью осознаете, что готовы совершить судьбоносный шаг: объединить алгебру Джорджа Буля с электричеством, т. е. создать устройство, принципиально сходное с компьютером, работающим с двоичными числами. Надеюсь, вы не слишком разволновались.

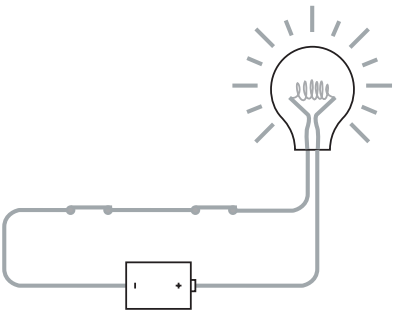
Чтобы начать эксперимент, соедините лампочку с батарейкой, как обычно, но включите в цепь два переключателя, а не один.



Переключатели, соединенные таким способом (один за другим), называются подключенными *последовательно* (in series). Если вы замкнете левый переключатель, не случится ничего.



Ничего не случится, если вы замкнете правый переключатель при разомкнутом правом. Лампочка загорится лишь при одном условии: если вы включите левый и правый переключатели одновременно.



Ключевое слово тут — *и*. Чтобы по цепи шел ток, должны быть включены левый *и* правый переключатели.

Эта цепь решает простую логическую задачу. Говоря попросту, лампочка отвечает на вопрос «Оба ли переключателя замкнуты?». Работу цепи можно суммировать в небольшой таблице.

Левый переключатель	**Правый переключатель**	**Лампочка**
Разомкнут	Разомкнут	Не горит
Разомкнут	Замкнут	Не горит
Замкнут	Разомкнут	Не горит
Замкнут	Замкнут	Горит

В предыдущей главе я рассказал, как представлять информацию с помощью двоичных цифр, или битов, — от чисел до направления пальца Роджера Эберта. Мы говорили, что 0 означает палец вниз, а 1 означает палец вверх. У переключателя две позиции, поэтому для его описания достаточно одного бита. Можно сказать, например, что 0 соответствует разомкнутой цепи, а 1 — замкнутой. У лампочки также два состояния, поэтому и ее можно описать одним битом. Значение 0 этого бита означает, что лампа не горит, а 1 соответствует горящей лампе. Теперь перепишем таблицу.

Левый переключатель	**Правый переключатель**	**Лампочка**
0	0	0
0	1	0
1	0	0
1	1	1

Заметьте: если левый и правый переключатели поменять местами, результат не изменится. Поэтому их можно не различать и переписать таблицу в том же виде, что ранее мы использовали для операторов И и ИЛИ.

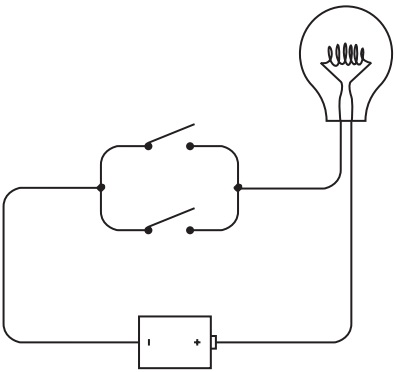
Переключатели при последовательном соединении	**0**	**1**
0	0	0
1	0	1

Эта таблица выглядит *точно так же*, как таблица для оператора И. Проверьте:

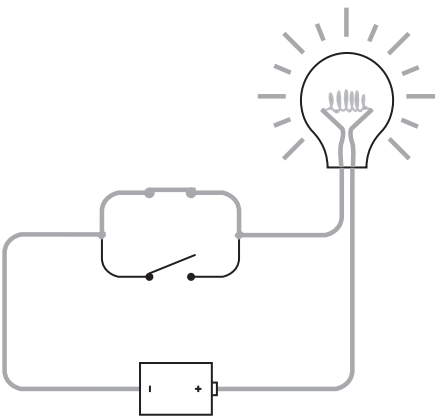
И	**0**	**1**
0	0	0
1	0	1

Эта простая схема выполняет операцию И булевой алгебры.

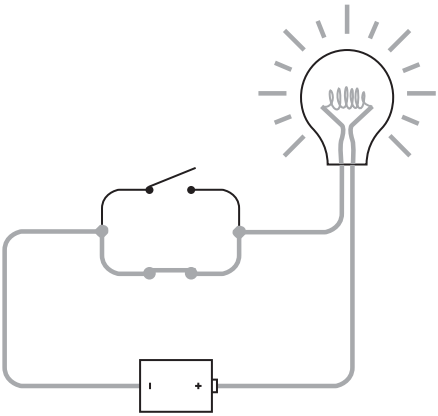
Теперь подключим переключатели иначе.



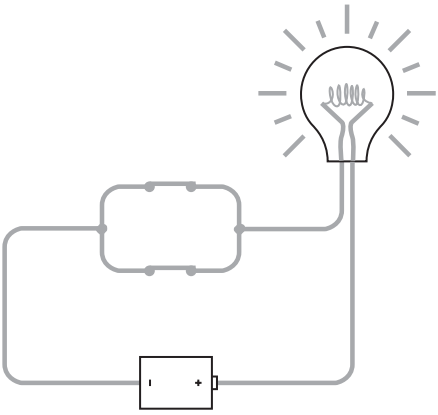
Это *параллельный* (in parallel) способ подключения. Отличие от предыдущего в том, что лампочка будет гореть независимо от того, включили вы верхний переключатель:



нижний:



или оба:



Лампа горит, если включен верхний *или* нижний переключатель. Ключевое слово здесь *или*.

И снова схема решает логическую задачу. На этот раз вопрос звучит так: «Включен ли хоть один переключатель?» Работа схемы проиллюстрирована в следующей таблице.

Левый переключатель	Правый переключатель	Лампочка
Разомкнут	Разомкнут	Не горит
Разомкнут	Замкнут	Горит
Замкнут	Разомкнут	Горит
Замкнут	Замкнут	Горит

Изменим таблицу, заменяя разомкнутое положение переключателя и потушенную лампу 0, а замкнутое положение и горящую лампу — 1.

Левый переключатель	Правый переключатель	Лампочка
0	0	0
0	1	1
1	0	1
1	1	1

Учитывая, что положение переключателя роли не играет, перепишем таблицу в упрощенном виде:

Переключатели при параллельном соединении	0	1
0	0	1
1	1	1

Вы, вероятно, *уже* догадались, что это таблица для логического ИЛИ:

ИЛИ	0	1
0	0	1
1	1	1

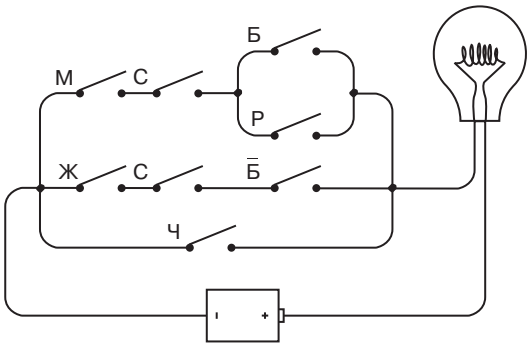
А это значит, что два параллельных переключателя выполняют логическую операцию ИЛИ.

Зайдя в зоомагазин, вы сказали следующее: «Мне нужен кот, стерилизованный, белый или рыжий; или кошка, стерилизованная, любого цвета, кроме белого; или любая кошка черного окраса», а продавец перевел это высказывание в выражение:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

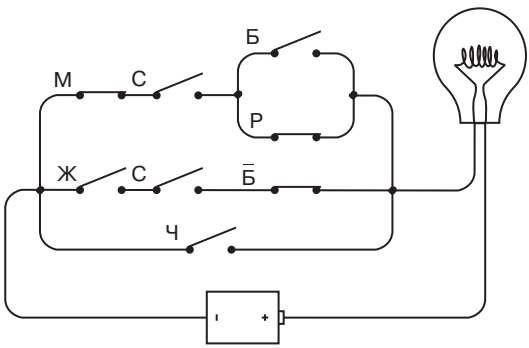
Теперь вы знаете, что последовательно соединенные переключатели выполняют логическую операцию И (выражаемую зна-

ком ×), а параллельно соединенные переключатели выполняют логическую операцию ИЛИ (выражаемую знаком +), и можете соединить 8 переключателей в такую схему:

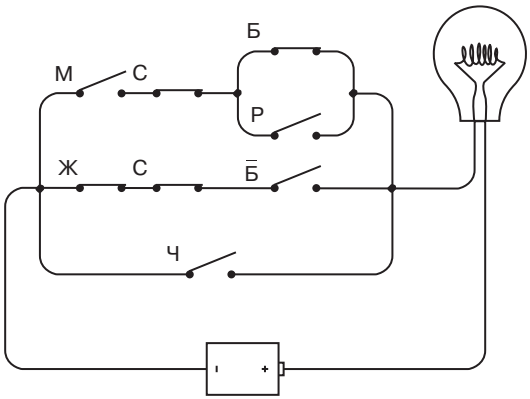


Каждый переключатель в этой схеме помечен буквой — той же, что и в булевом выражении ($\overline{Б}$ эквивалентно выражению НЕ Б и является другой формой записи выражения 1 – Б). Просмотрев схему слева направо и сверху вниз, вы увидите, что буквы стоят на ней в том же порядке, что и в выражении. Каждому знаку × в выражении соответствует точка схемы, в которой два переключателя (или две группы переключателей) соединены последовательно. Каждому знаку + в выражении соответствует место на схеме, в котором два переключателя (или две группы переключателей) соединены параллельно.

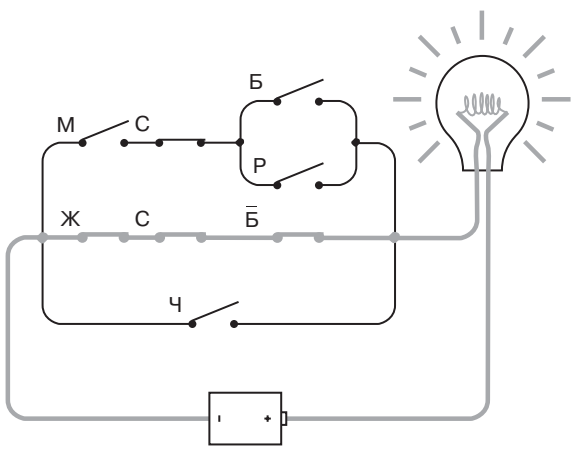
Как вы помните, сначала продавец принес нестерилизованного рыжего кота. Включите соответствующие переключатели.



Хотя переключатели М, Р и НЕ Б замкнуты, цепь в целом разомкнута, и лампочка не горит. Затем продавец принес стерилизованную белую кошечку:



И снова цепь осталась разомкнутой. И наконец на сцене появляется стерилизованная серая кошечка:



Вот теперь цепь замкнута, а лампочка горит, сигнализируя, что все ваши условия выполнены.

Джордж Буль такой схемы не разработал. Ему никогда не хотелось увидеть, как выражения его алгебры оживают в виде переключателей, проводов и лампочек. Отчасти этому препят-

ствовало отсутствие лампы накаливания, которая была изобретена лишь через 15 лет после смерти Буля. Но работу телеграфа Сэмюэль Морзе продемонстрировал в 1844 г. — через десять лет после публикации булевского «Исследования законов мышления», и в приведенных выше схемах лампочку вполне можно было заменить зуммером телеграфа.

Тем не менее, никто в XIX в. не увидел связи между логическими операторами И и ИЛИ и последовательным и параллельным соединением переключателей. Ни математики, ни электрики, ни операторы телеграфа — никто. Не сделал этого даже провозвестник компьютерной революции Чарльз Бэббидж (Charles Babbage) (1792–1871), который переписывался с Булем, знал о его работах и посвятил большую часть своей жизни созданию Аналитической Машины, которую век спустя стали называть прототипом современных компьютеров. Теперь мы знаем: помочь Бэббиджу могло осознание того, что для автоматизации вычислений вместо колес и рычажков нужно было использовать телеграфные реле.

Да, телеграфные реле.

Глава 11

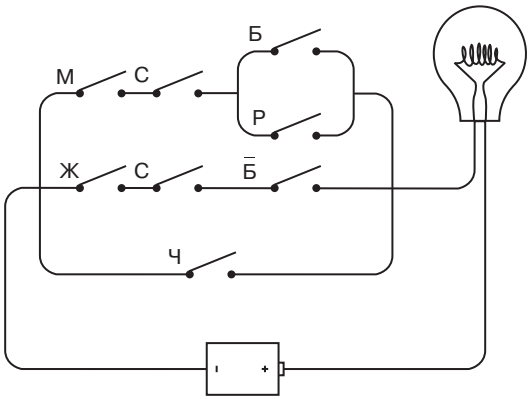
Вентили, которые не протекают

В отдаленном будущем, когда история примитивных вычислений XX в. начнет постепенно забываться, кто-нибудь, вероятно, предположит, что *логический вентиль* (gate) ведет свою родословную от одноименного сантехнического приспособления. Однако это не совсем так. Хотя, как мы вскоре узнаем, логические вентили и впрямь похожи на обычные водопроводные краны: они выполняют простые логические операции, останавливая или пропуская через себя электрический ток.

Помните, в предыдущей главе мы вообразили, что вы пришли в зоомагазин и заявили: «Хочу белого или рыжего стерилизованного кота, или стерилизованную кошку любого цвета, кроме белого, или любую кошку или кота черного цвета». Все эти пожелания описываются логическим выражением:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

или такой электрической схемой с переключателями и электрической лампочкой:



Все до единого компоненты этой схемы изобретены еще в XIX в., но никто в то время не осознал, что они позволяют реализовать булевы выражения. О связи электричества и логики не было известно до 1930-х годов, когда в своей магистерской диссертации «Символьный анализ цепей с реле и переключателями» на нее указал Клод Элвуд Шеннон (Claude Elwood Shannon) (род. 1916). Десять лет спустя в статье Шеннона «Математическая теория коммуникации» слово *бит* (bit) впервые было использовано в значении «двоичная цифра» (binary digit).

Конечно, и до 1938 г. было известно, что для протекания тока по цепи с двумя переключателями, соединенными последовательно, оба они должны быть замкнуты, а для протекания тока по цепи с параллельным соединением переключателей замкнут должен быть только один из них. Но никто не показал с ясностью и убедительностью Шеннона, что для проектирования цепей с переключателями можно применять средства булевой алгебры. В частности, если булево выражение, описывающее работу схемы, удается упростить, упростить можно и саму схему.

Итак, выражение, которое описывает свойства нужной вам кошки или кота, выглядит так:

$$(М \times С \times (Б + Р)) + (Ж \times С \times (1 - Б)) + Ч$$

Коммутативный закон умножения позволяет изменить порядок переменных, связанных знаком × , и переписать выражение в виде:

$$(С \times М \times (Б + Р)) + (С \times Ж \times (1 - Б)) + Ч$$

Чтобы мои действия стали понятнее, введем две вспомогательных переменные X и Y:

$$X = М \times (Б + Р)$$

$$Y = Ж \times (1 - Б)$$

С их помощью выражение для кошачьих свойств запишется так:

$$(С \times X) + (С \times Y) + Ч$$

Закончив упрощение, мы опять подставим вместо X и Y их значения.

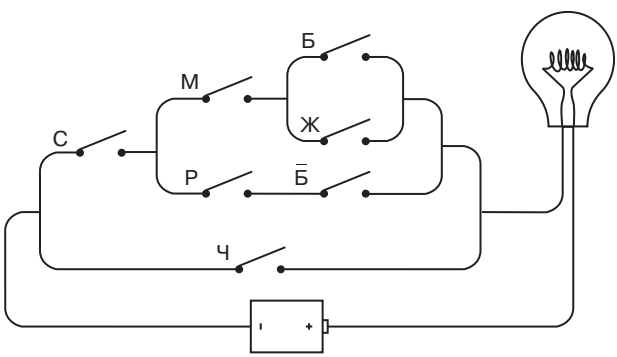
Как видите, переменная С появляется в выражении дважды. Применив дистрибутивный закон умножения относительно суммы, перепишем выражение так, чтобы С встречалась в нем только один раз:

$$(С \times (X + Y)) + Ч$$

Теперь подставим исходные значения X и Y:

$$(С \times ((М \times (Б + Р)) + (Ж \times (1 - Б)))) + Ч.$$

В этом выражении так много скобок, что на первый взгляд трудно назвать его упрощенным. И все же в нем переменных на одну меньше, чем в первоначальном варианте, а значит, одним переключателем меньше в схеме. Вот как выглядит измененная версия схемы:



Честно говоря, проще разглядеть эквивалентность двух схем, чем проверить равенство булевых выражений.

А ведь из этой схемы можно убрать еще три переключателя. Теоретически для задания параметров кошки хватит четырех переключателей. Почему четырех? По сути каждый переключатель — это 1 бит. Один переключатель позволяет задавать пол (выключен — мужской, включен — женский), другой служил бы признаком стерилизации (выключен — стерилизована, включен — нет), с помощью оставшихся двух можно указывать цвет. Выбирая кошку, вы задаете четыре цветовых признака: белый, черный, рыжий и другой. Мы знаем, что четыре свойства можно задать двумя битами, поэтому для выбора цвета нужно всего два переключателя. Скажем, два переключателя разомкнуты — цвет белый, первый переключатель замкнут — черный, второй переключатель замкнут — рыжий, оба переключателя замкнуты — все остальные цвета.

Теперь создадим для выбора кошки специальный прибор. На его пульте размещены четыре переключателя и лампочка.

Если тумблер вверху, переключатель замкнут, внизу — разомкнут. Боюсь, что обозначения переключателей для выбора цвета несколько туманны, но это плата за стремление сделать пульт максимально экономичным. Левый переключатель в этой паре обозначен Ч. Это значит, что когда он замкнут (как показано на рисунке), выбран черный цвет. Правый переключатель в паре обозначен Р, и его включение значит, что выбран рыжий окрас. Если оба тумблера вверху, выбран цвет «другой», что символизирует буква Д. Если оба тумблера внизу, выбран белый цвет (буква Б).

На компьютерном языке этот набор переключателей называется *устройством ввода* (input device) информации, управляющей поведением цепи. У нас переключатели позволяют

ввести 4 бита информации, которые описывают кошку или кота. Электрическая лампочка — *устройство вывода* (output device) — загорается, если параметры кошки, заданные переключателями, соответствуют вашим требованиям. Положение переключателей на рисунке соответствуют выбору черной нестерилизованной кошки. Она удовлетворяет вашим критериям, и потому лампочка горит.

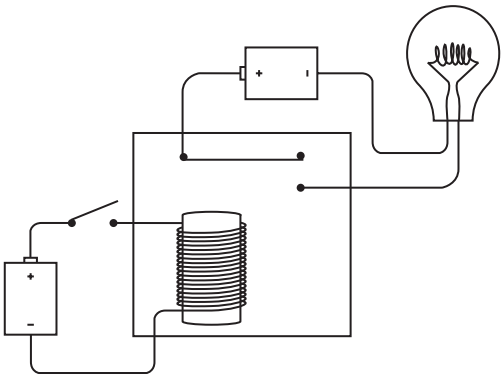
Теперь осталось разработать электрическую схему, которая обеспечивала бы работу этого пульта.

Как вы помните, диссертация Клода Шеннона называлась «Символьный анализ цепей с реле и переключателями». Реле, которые в ней упоминаются, походят на телеграфные, о которых мы говорили в главе 6. Однако во времена Шеннона реле уже применяли и для других целей, в частности, в телефонной сети.

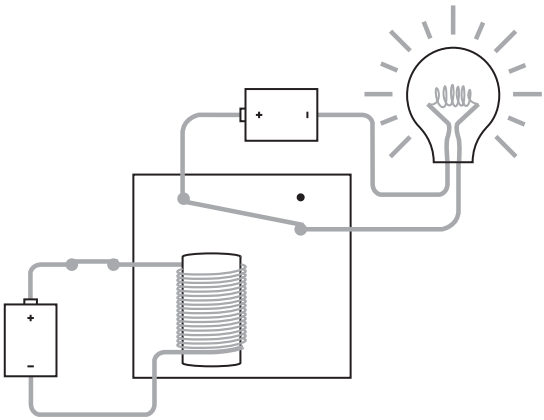
Как и переключатели, для решения простых логических задач реле можно включать в схемы параллельно и последовательно. Такие комбинации реле называются *логическими вентилями*. Говоря о том, что вентили решают простые логические задачи, я имею в виду *простейшие* задачи. Преимущество реле по сравнению с переключателями в том, что их можно выключать и включать, используя другие реле, а не вручную. Это значит, что простые вентили можно комбинировать для решения более сложных задач, например, выполнения основных действий арифметики. В следующей главе мы на практике убедимся в том, что из переключателей, лампочек, источника питания и реле можно собрать суммирующую машину (хотя и работающую только с двоичными числами).

Вы, конечно, помните, что в работе телеграфа реле играли важнейшую роль. Провода, соединяющие телеграфные станции, обладают высоким сопротивлением, поэтому требуется устройство, которое принимало бы слабый сигнал и передавало идентичный мощный. В реле это происходит благодаря электромагниту, который управляет переключателем. Фактически для получения мощного сигнала реле *усиливает* слабый сигнал.

Но нам нет нужды использовать реле в качестве усилителя. Реле интересует нас как переключатель, управляемый не вручную, а с помощью электричества. Соединим реле с переключателем, парой батареек и лампочкой:



Переключатель слева разомкнут, и лампочка не горит. Если вы замкнете этот переключатель, ток из батареи пойдет по виткам катушки, намотанной на железный сердечник. Сердечник станет магнитом, притянет гибкую металлическую полоску, она замкнет цепь и включит лампочку:

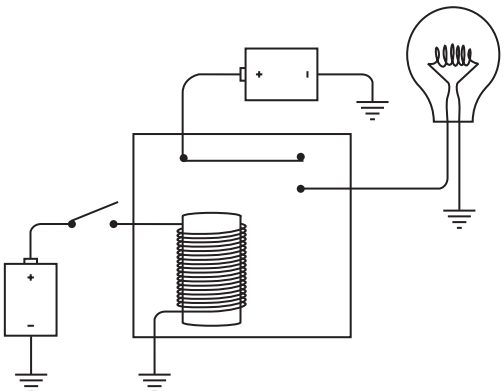


Если полоска притянута электромагнитом, говорят, что реле *сработало* (защелкнулось). Когда переключатель размыкается, железный сердечник перестает быть магнитом, и полоска возвращается в первоначальное (незамкнутое) состояние.

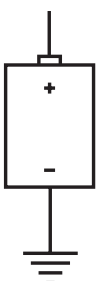
Не слишком ли замысловатый способ мы избрали, чтобы включить свет? Конечно, если бы мы были заинтересованы

только в этом, то обошлись бы и без реле, но наши притязания включением света не ограничиваются.

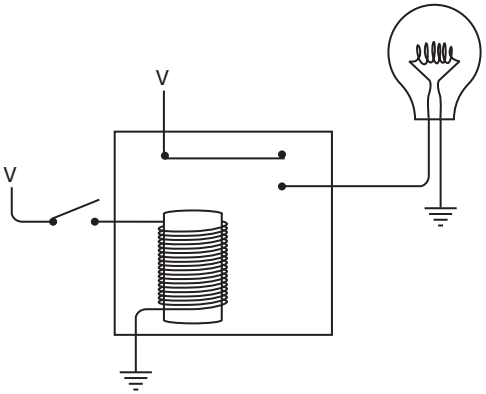
В этой главе мы будем использовать реле очень часто (а потом, построив все логические вентили, практически навсегда забудем о них), поэтому я хочу упростить изображение схемы. Исключим некоторые провода с помощью земли. В данном случае, «земля» значит просто общий провод и с реальной землей соединяться не должна.



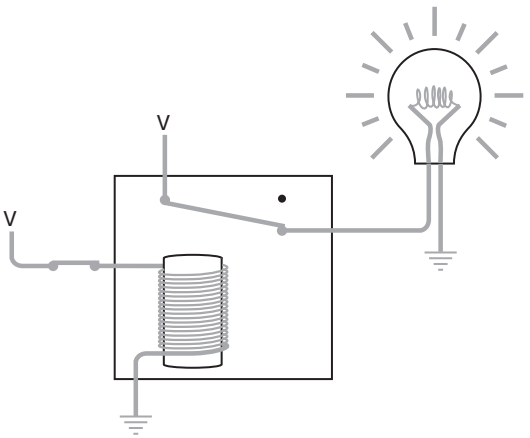
Я понимаю, что схема не слишком упростилась, но мы еще не закончили. Обратите внимание: отрицательные контакты обеих батарей соединены с землей. Поэтому везде, где мы увидим нечто подобное:



заменим это изображение заглавной латинской V, как поступали в главах 5 и 6. Теперь схема выглядит так:

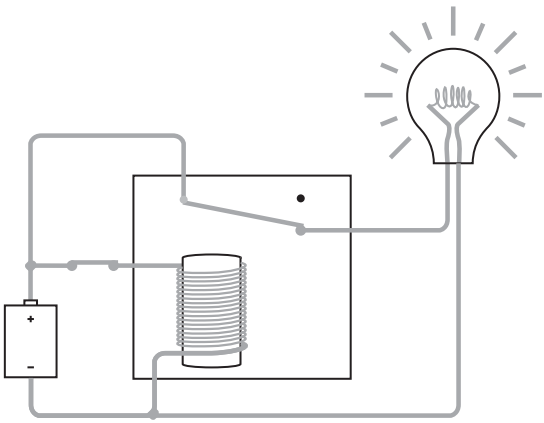


Когда переключатель замкнут, ток между источником питания и землей протекает через катушку электромагнита, электромагнит притягивает металлическую полоску, и та замыкает цепь между питанием, лампочкой и землей. Лампочка загорается.

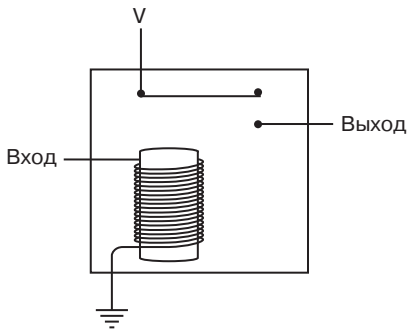


На этой схеме два источника питания и две земли, но на схемах в этой главе все земли можно соединить и все источники питания тоже можно соединить друг с другом. Во всех релейных схемах и логических вентилях в этой и следующей главах достаточно одного (достаточно мощного) источника пи-

тания. Например, предыдущую схему можно перерисовать с использованием только одной батареи:

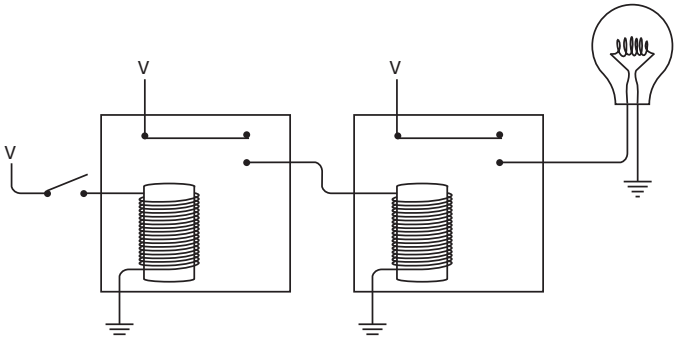


Но для наших целей эта схема не слишком годится. Лучше избегать замкнутых цепей и описывать работу реле с помощью понятий *входного* (input) и *выходного* (output) сигналов.

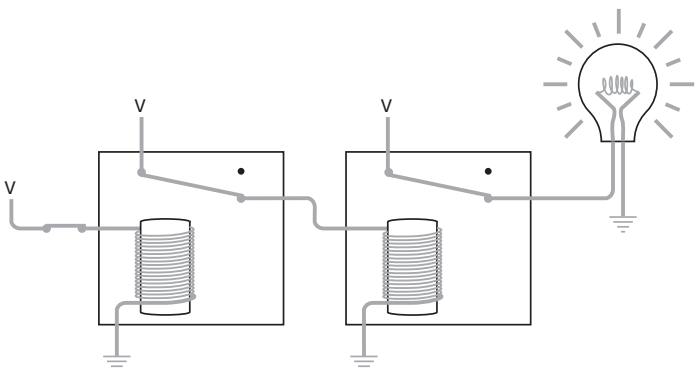


Если на вход поступает ток (например, если он с помощью переключателя подключен к источнику питания), электромагнит срабатывает, и на выходе появляется напряжение.

На входе реле не обязательно должен стоять переключатель, а на выходе — лампочка. Выход одного реле может подключаться ко входу другого реле, например, так:

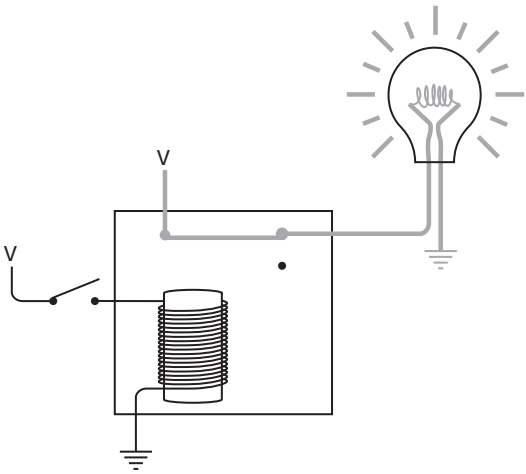


Когда вы замыкаете переключатель, первое реле срабатывает и подает питание на второе реле. Оно тоже сработает, и включится свет:

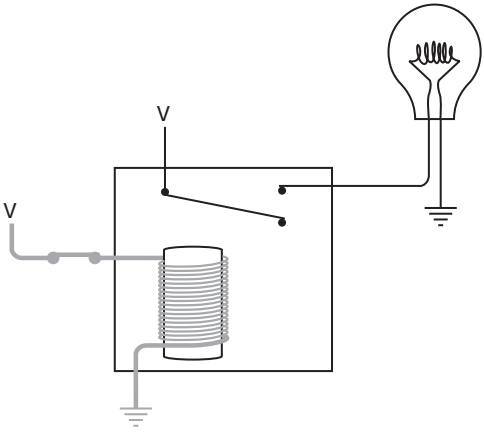


Соединение нескольких реле лежит в основе построения логических вентилей.

В действительности лампочку можно соединить с реле двумя способами. Видите гибкую металлическую полоску, которая притягивается электромагнитом? В состоянии покоя она касается верхнего контакта, а когда электромагнит ее притягивает — нижнего. До сих пор мы использовали в качестве выхода реле нижний контакт, но в этом качестве можно использовать и верхний. В последнем случае лампочка горит, когда переключатель разомкнут.

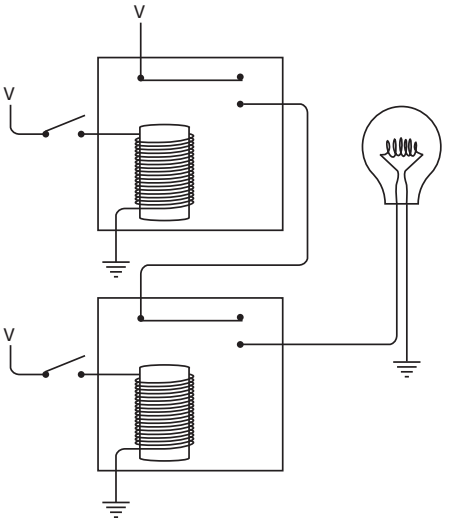


Когда входной переключатель замкнут, лампочка гаснет.

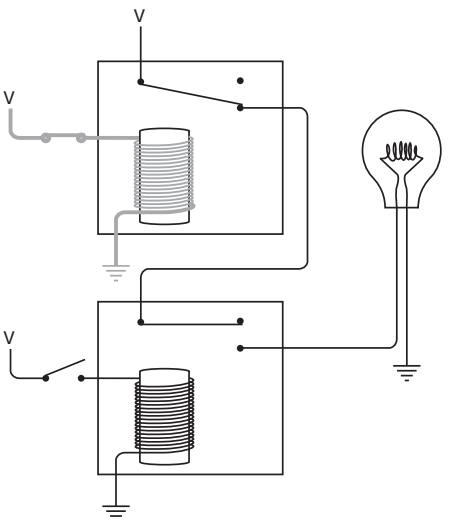


Такое реле называется *двухпозиционным* (double-throw). У него два электрически противоположных выхода: когда на одном есть напряжение, на другом его нет.

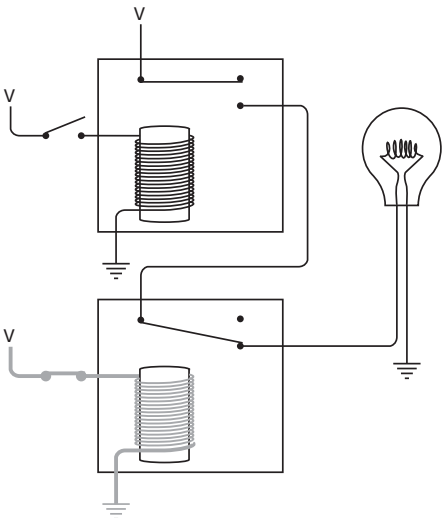
Подобно переключателям, реле можно соединить последовательно:



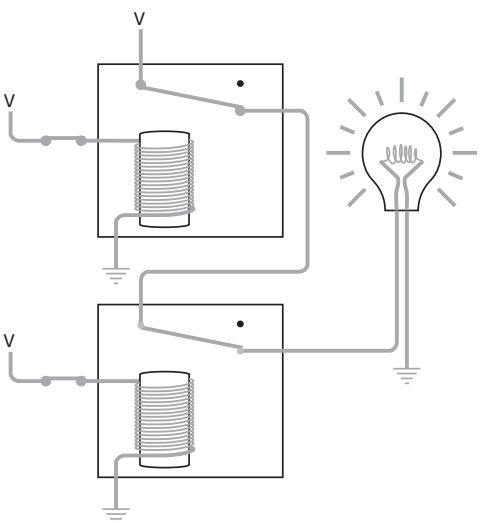
Выход верхнего реле подает питание на нижнее. Как видите, если оба переключателя разомкнуты, лампочка не горит. Попробуем замкнуть верхний переключатель:



Лампочка не включилась, так как нижний переключатель разомкнут, и нижнее реле не сработало. Теперь разомкнем верхний переключатель и замкнем нижний.

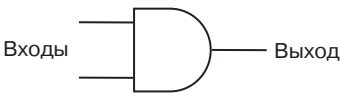


Все равно не горит. Ток не доходит до лампочки, так как не сработало первое реле. Единственный способ заставить ее светить — замкнуть оба переключателя.



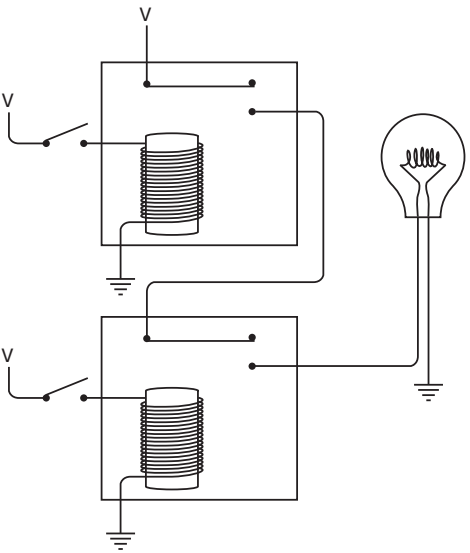
Теперь сработали оба реле, и ток течет между источником питания, лампочкой и землей.

Подобно двум переключателям, соединенным последовательно, эти два реле выполняют простую логическую функцию. Лампочка горит, если сработали оба реле. Эта схема называется *вентилем И (AND)*. Символически он изображается так:

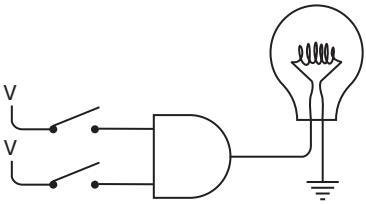


Это первый из четырех основных логических вентилей. У вентиля И два входа (на схеме слева) и один выход (на схеме справа). Вентиль И часто рисуют именно так: со входами слева и выходом справа. Да оно и понятно: людям, привыкшим читать слева направо, и электрические схемы удобно рассматривать слева направо. Но вентиль И с тем же успехом можно рисовать и со входами вверху, справа или внизу.

Первоначально схема с двумя последовательно соединенными реле, двумя переключателями и лампочкой выглядела так:

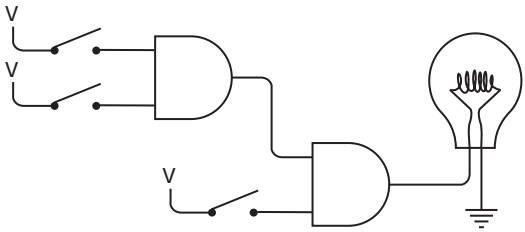


Использовав символ вентиля И, ту же схему можно изобразить так:



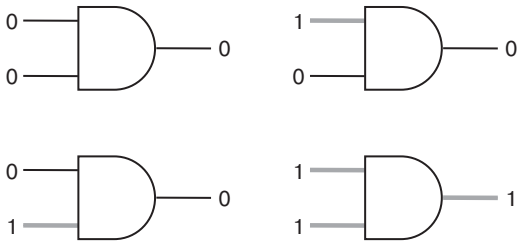
Символ вентиля И не только заменяет два последовательно соединенных реле, но и подразумевает, что верхнее реле соединено с источником питания и оба соединены с землей. Еще раз: лампочка загорается, только если замкнуты верхний *и* нижний переключатели. Поэтому схема и называется вентилем *И*.

Входы вентиля И не обязательно должны соединяться с переключателями, а выход не обязательно должен быть подключен к лампочке. По сути, мы имеем дело только с напряжениями на входе и на выходе. Например, выход одного вентиля И может быть входом второго вентиля И:



Эта лампочка горит, только когда замкнуты все три переключателя. Если замкнуты *оба* верхних переключателя, выход первого вентиля И заставит сработать первое реле второго вентиля И. Нижний же переключатель заставит сработать второе реле второго вентиля И.

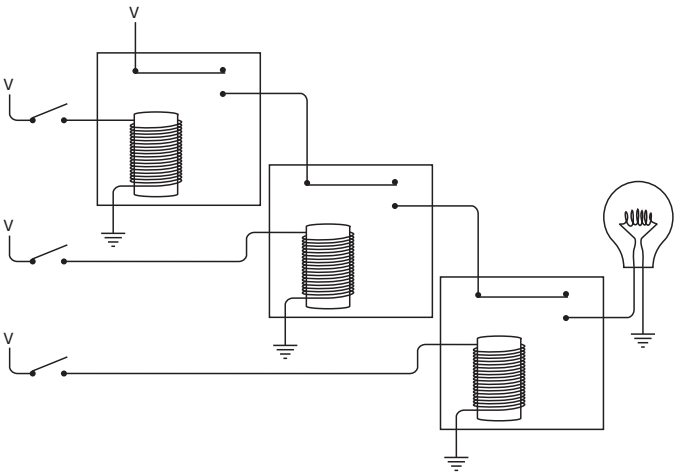
Если мы обозначим отсутствие напряжения 0, а его присутствие — 1, выходной сигнал вентиля И зависит от входного сигнала следующим образом:



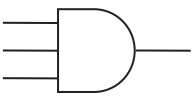
Как и работу двух последовательных переключателей, работу вентиля И можно проиллюстрировать таблицей.

И	**0**	**1**
0	0	0
1	0	1

У вентиля И может быть и больше двух входов. Допустим, что мы соединили последовательно три реле.

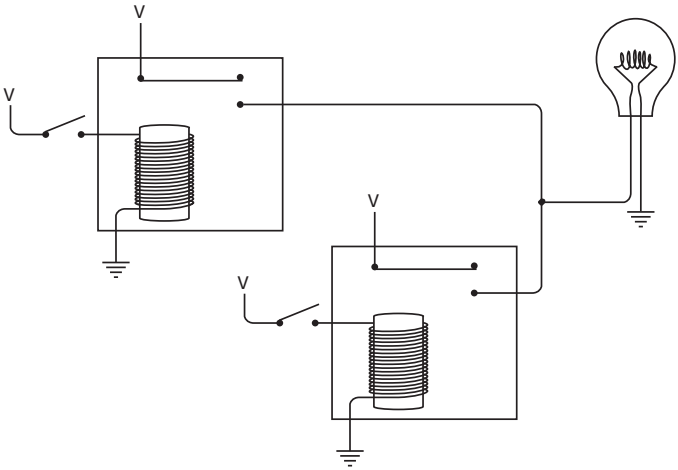


Лампочка загорится, только если будут замкнуты все три переключателя. Эту конфигурацию можно обозначить символом

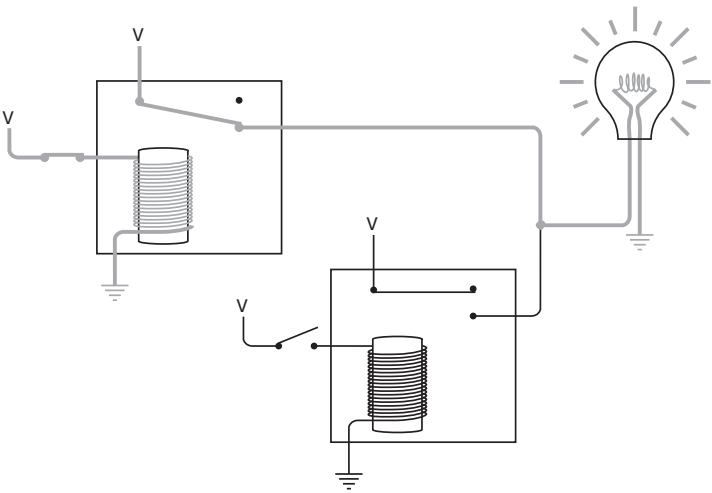


Называется она трехвходовым вентилем И.

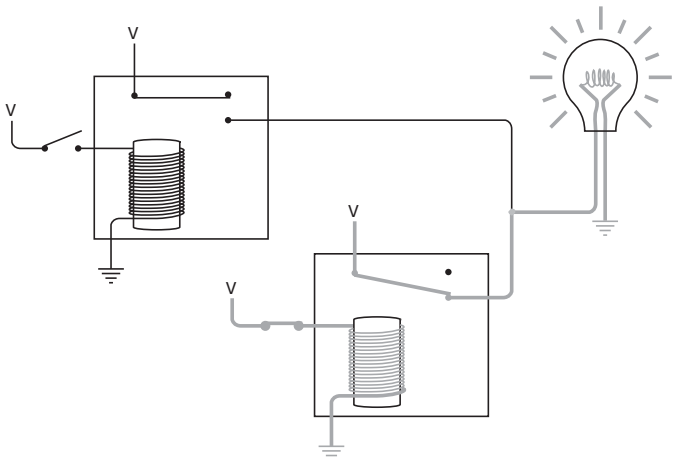
Следующий логический вентиль состоит из двух реле, соединенных параллельно:



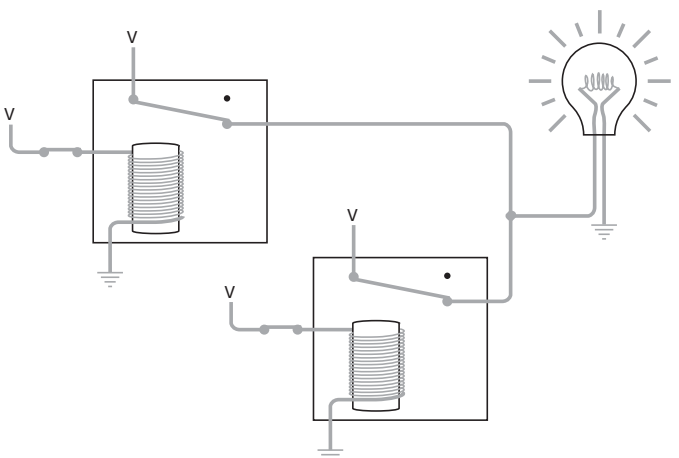
Выходы двух реле соединены друг с другом, и с этого объединенного выхода питание поступает в лампочку. К включению лампочки приведет срабатывание любого из двух реле. Например, если мы замкнем верхний переключатель, лампочка загорится, так как получит питание от левого реле.



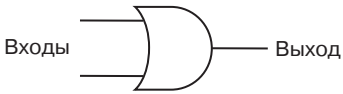
Если мы разомкнем верхний переключатель и замкнем нижний, лампочка опять загорится.



Лампочка будет гореть, если будут замкнуты оба переключателя.

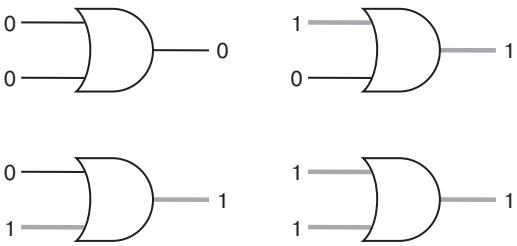


В данном случае лампочка горит, когда замкнут верхний *или* нижний переключатель. Ключевое слово здесь «или», и потому представленная схема называется *вентилем ИЛИ (OR)*. На схемах для вентиля ИЛИ используют такое обозначение:



Этот символ похож на символ вентиля И за исключением того, что сторона входов закруглена.

На выходе вентиля ИЛИ есть напряжение, если напряжение подается на любой из двух его входов. Обозначим опять отсутствие напряжения как 0, а его присутствие — 1. Вентиль ИЛИ может находиться в следующих состояниях:

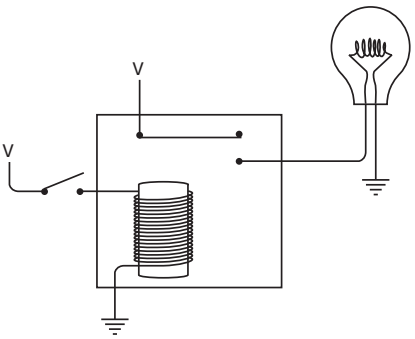


Эти состояния можно свести в таблицу.

ИЛИ	**0**	**1**
0	0	1
1	1	1

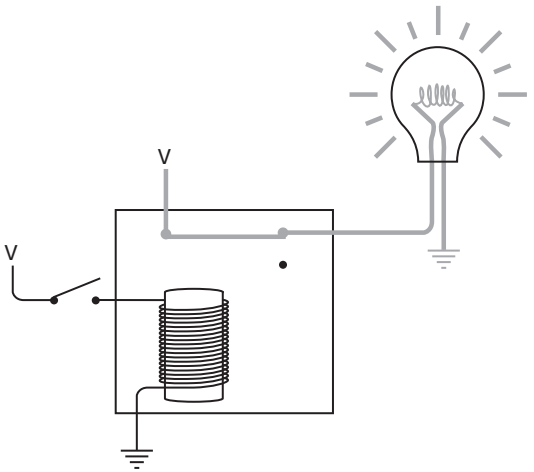
У вентиля ИЛИ может быть более двух входов. Выход его равен 1, если сигнал хотя бы на одном из входов равен 1. Выход, равный 0, соответствует 0 на всех входах.

Ранее я объяснял, что наши реле являются двухпозиционными. Обычно, когда переключатель разомкнут, лампочка не горит.

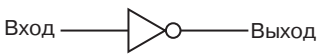


Когда переключатель замкнут, лампочка загорается.

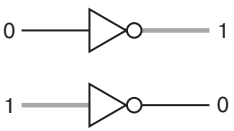
Но если мы подключим к выходу другой контакт двухпозиционного реле, лампочка будет гореть, если переключатель *разомкнут*.



В этой схеме лампочка гаснет, когда мы замыкаем переключатель. Одиночное реле, подключенное таким способом, называется *инвертором* (inverter). Инвертор не является логическим вентилем (у вентиля всегда два или более входов), но это не уменьшает его полезности. На схемах он обозначается так:

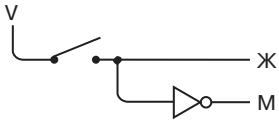


Как следует из названия, инвертор инвертирует 0 (нет напряжения) в 1 (напряжение есть) и наоборот:

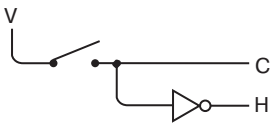


Теперь, когда в нашем распоряжении имеются инвертор, вентиль И и вентиль ИЛИ, можно начинать сборку прибора для автоматического выбора идеальной кошки. Начнем с пе-

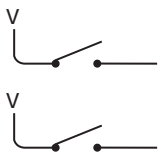
реключателей. Первый переключатель замкнут, если вы выбрали кошку, и разомкнут, если выбран кот. Для генерации сигналов Ж и М годится такая схема.



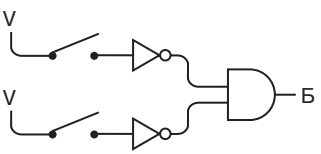
Когда сигнал на выходе Ж равен 0, сигнал на выходе М равен 1, и наоборот. Подобным же образом можно организовать работу второго переключателя, который замкнут для стерилизованной кошки и разомкнут для нестерилизованной.



Работу следующих двух переключателей организовать сложнее. В разных комбинациях они должны воспроизводить четыре разных цвета. Итак, у нас два переключателя, каждый из которых подключен к источнику питания.

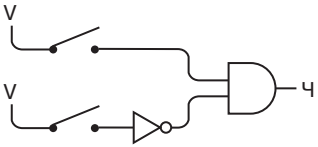


Когда оба переключателя разомкнуты (как на рисунке), они символизируют выбор белого цвета. Вот как можно, используя два инвертора и один вентиль И, сгенерировать сигнал Б, который равен 1 (напряжение есть), если вы выбрали белую кошку, и 0 (нет напряжения), если вы этого не сделали.



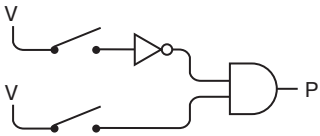
Когда переключатели разомкнуты, входные сигналы обоих инверторов равны 0, а выходные — 1. А поскольку выходы обоих инверторов являются входами вентиля И, на его выходе мы также получаем сигнал 1. Если любой из двух переключателей замкнут, выход вентиля И равен 0.

Чтобы выбрать черную кошку, мы замыкаем первый переключатель. Это можно реализовать с помощью инвертора и вентиля И.

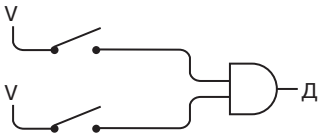


Выход вентиля И равен 1, только если первый переключатель замкнут, а второй разомкнут.

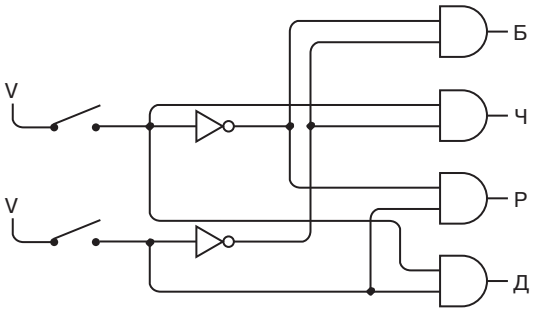
Подобным же образом мы выбираем рыжую кошку, замыкая второй переключатель.



Наконец, если оба переключателя разомкнуты, мы хотим кошку «другого» цвета.



Теперь объединим эти четыре небольшие схемы в одну большую (как обычно, черные точки на схеме обозначают соединение проводов; провода, на пересечениях которых нет черных точек, *не соединены*).



Понимаю, эта путаница проводов кажется очень сложной. Но если вы приглядитесь повнимательнее, то убедитесь, что схема действительно работает именно так, как нужно. Проследите, откуда приходят сигналы на входы вентилей И, стараясь не отвлекаться на то, куда *еще* они идут. Если оба переключателя разомкнуты, выход Б равен 1, в любом другом случае он равен 0. Если первый переключатель замкнут, сигнал Ч равен 1, во всех остальных случаях он равен 0 и т. д.

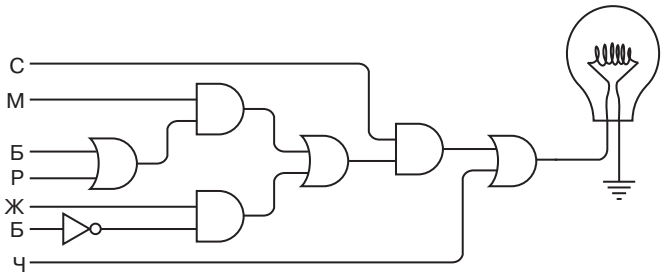
Соединение вентилей и инверторов подчиняется нескольким простым правилам. Выход одного вентиля (или инвертора) может быть входом одного или нескольких вентилей (или инверторов). Выходы двух или более вентилей (или инверторов) никогда не соединяются между собой.

Эта схема, состоящая из четырех вентилей И и двух инверторов, называется *дешифратором 2 линии на 4* (2-Line-to-4-Line Decoder). На вход этого дешифратора подаются 2 бита, которые в различных комбинациях представляют 4 различные величины. На выходе дешифратора имеем 4 сигнала, из которых в любой момент времени значение 1 имеет только один. Какой именно, зависит от входных величин. На тех же принципах строятся дешифраторы 3 линии на 8, 4 линии на 16 и т. д.

В простейшем виде булево выражение для выбора кошки выглядит так:

$$(С \times ((М \times (Б + Р)) + (Ж \times (1 - Б)))) + Ч$$

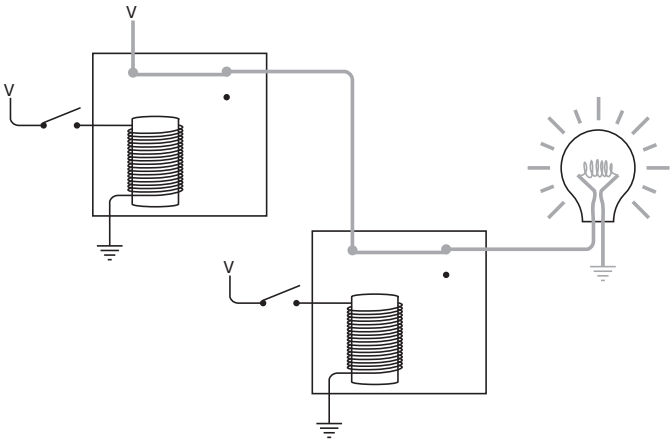
Каждому знаку «+» в этом выражении на схеме должен соответствовать вентиль ИЛИ, а каждому знаку «×» — вентиль И.



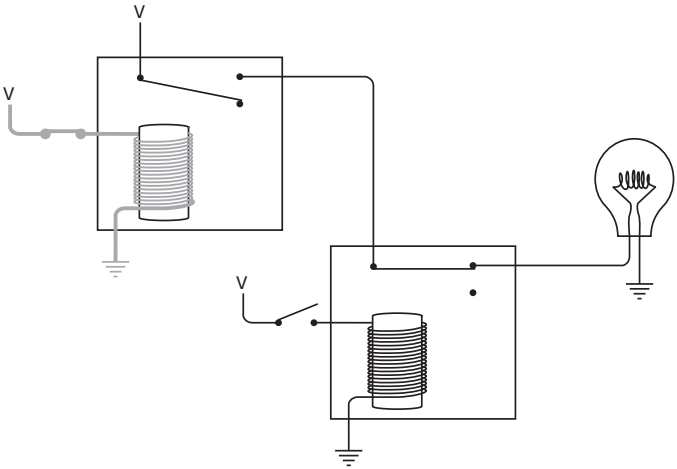
Символы в левой части схемы идут в том же порядке, что и в выражении. Соответствующие сигналы приходят от переключателей, соединенных проводами с инверторами и дешифратором «2 на 4». Обратите внимание на использование инвертора для реализации выражения (1 – Б).

Теперь вы вправе воскликнуть: «Какая прорва реле!» — и будете правы. По два реле приходится на каждый из вентилей И и ИЛИ и по одному — на каждый инвертор. Ничего более умного, чем «Привыкайте!» я ответить не могу. В следующих главах нам понадобится гораздо больше реле. Радуйтесь, что вам не приходится по-настоящему покупать их и соединять проводами.

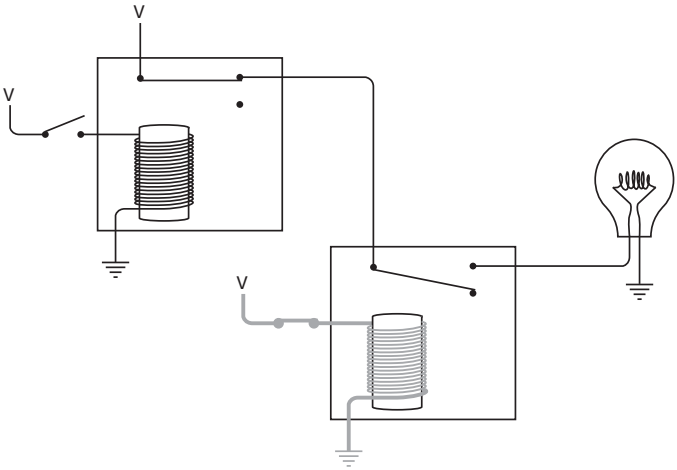
В этой главе мы познакомимся еще с двумя логическими вентилями. В обоих применяется выход реле, на котором есть напряжение, когда реле не сработало (т. е. тот выход, что используется в инверторе). Например, в следующей конфигурации выход одного реле подает питание на вход второго реле. Лампочка горит, когда оба входа отключены от питания.



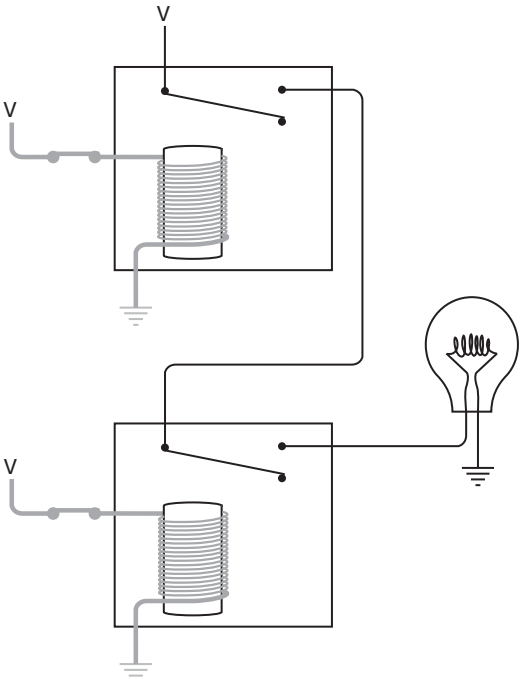
Если верхний переключатель замкнут, лампочка гаснет.



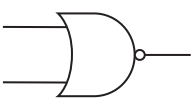
Это происходит потому, что питание перестает поступать на второе реле. Свет также не горит, если замкнут нижний переключатель.



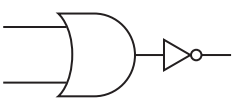
Если оба переключателя замкнуты, лампочка тоже не горит.



Такое поведение в точности противоположно поведению вентиля ИЛИ. Такая схема называется *вентилем ИЛИ-НЕ (NOT OR или NOR)*. Его обозначение:



т. е. такое же, как у вентиля ИЛИ, но с небольшим кружком на выходе. Вентиль ИЛИ-НЕ эквивалентен схеме:

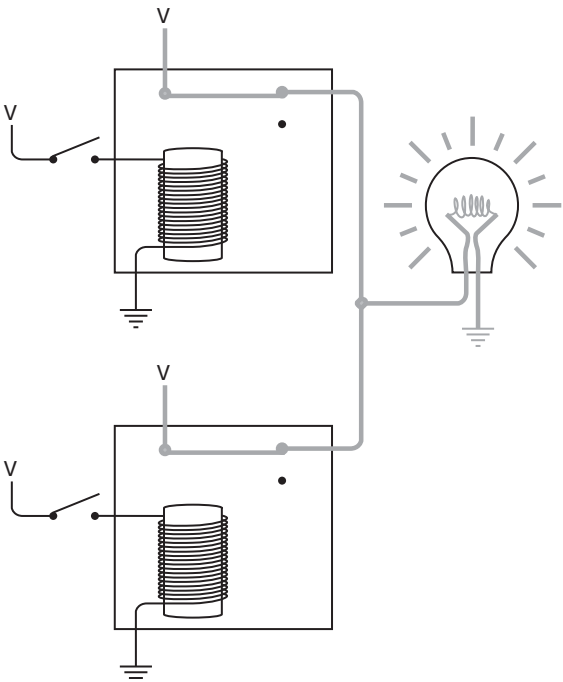


Работа вентиля ИЛИ-НЕ иллюстрируется следующей таблицей.

ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

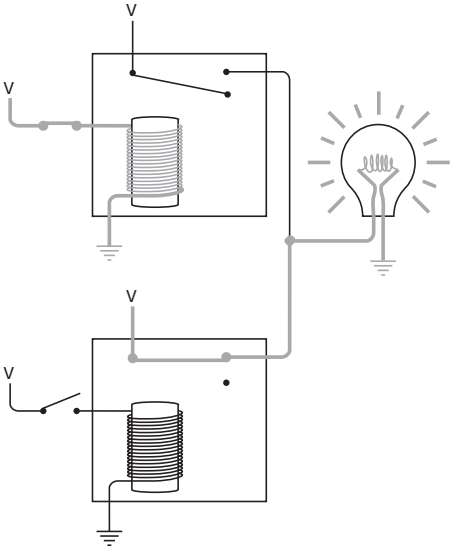
Эта таблица противоположна таблице для вентиля ИЛИ, выходной сигнал которого равен 1, если на любом из двух входов есть сигнал 1, и равен 0, если на оба входа подается 0.

Вот еще один способ соединения двух реле:

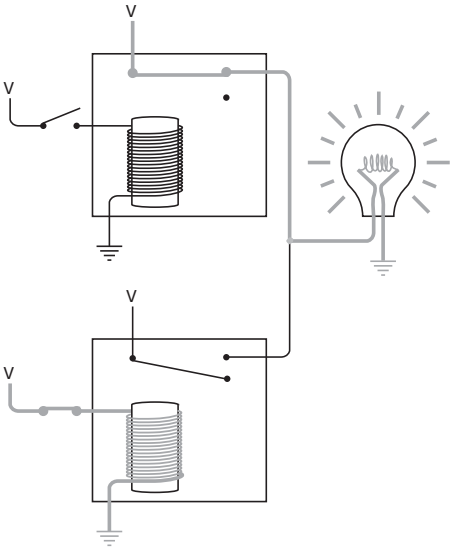


Здесь оба выхода соединены друг с другом, как в вентиле ИЛИ, но с использованием других контактов. Лампочка включена, если оба переключателя разомкнуты.

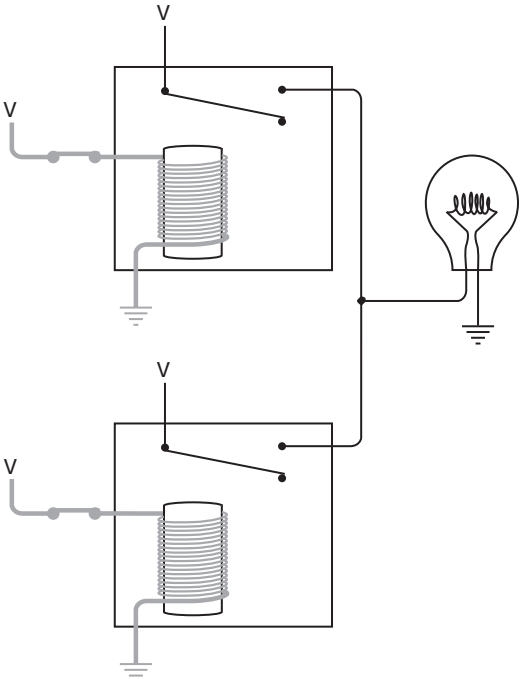
Свет не погаснет, если замкнуть верхний переключатель.



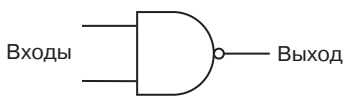
Не погаснет он, если будет замкнут и только нижний переключатель.



Только когда оба переключателя замкнуты, лампочка гаснет.



Такое поведение в точности противоположно работе вентиля И. Эта схема называется *вентилем И-НЕ (NOT AND или NAND)*. Вентиль И-НЕ изображается, как вентиль И, но с кружком на выходе, означающим инверсию.



Поведение вентиля И-НЕ описано в следующей таблице.

И-НЕ	**0**	**1**
0	1	1
1	1	0

Заметьте: выход вентиля И-НЕ противоположен выходу вентиля И. У последнего выход равен 1, только если оба входа равны 1, в противном случае его выход имеет значение 0.

На данный момент мы рассмотрели четыре различных способа соединения реле с двумя входами и одним выходом. Каждая конфигурация ведет себя по-своему. Чтобы каждый раз не рисовать реле, мы назвали эти конфигурации логическими вентилями и решили использовать для них стандартные обозначения. Зависимость выходного сигнала каждого из четырех вентилей от сигналов на входе иллюстрируется следующими таблицами.

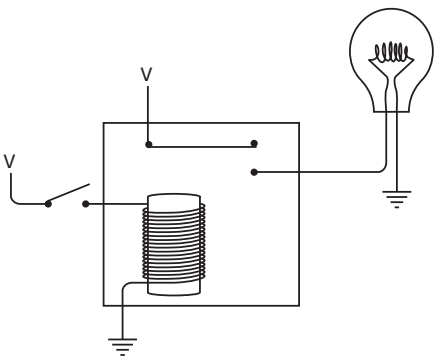
И	**0**	**1**
0	0	0
1	0	1

ИЛИ	**0**	**1**
0	0	1
1	1	1

ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

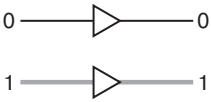
И-НЕ	**0**	**1**
0	1	1
1	1	0

Теперь в нашем арсенале четыре логических вентиля и инвертор. Дополним его обычным старым добрым реле:



Это устройство называется *буфером* (buffer) и обозначается символом:

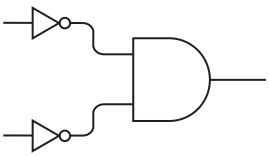
Это тот же символ, что и у инвертора, но без кружка на выходе. Ничего особенного буфер не делает. Сигнал на его выходе повторяет сигнал на входе.



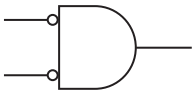
Буфер можно использовать, если входной сигнал слаб. Как вы, конечно, помните, именно это много лет назад стало причиной применения реле в телеграфе. Буфер применяется и для небольшой задержки сигнала. Она происходит из-за того, что для срабатывания реле требуется немного времени — доли секунды.

Начиная с этого момента, изображения реле в книге вам встречаться практически не будут. Вместо них вы увидите схемы с буферами, инверторами, четырьмя основными логическими вентилями и более изощренными приборами (вроде дешифратора «2 на 4»), построенными из логических вентилей. Конечно, все эти компоненты собраны из реле, но это еще не повод вырисовать их на схемах.

Ранее, собирая дешифратор «2 на 4», мы встречались с небольшой схемой типа:

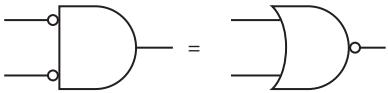


Два входных сигнала инвертируются и подаются на входы вентиля И. Иногда такую конфигурацию изображают без инверторов.



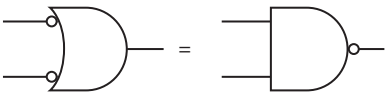
Обратите внимание на кружки на входе вентиля И. Они означают, что сигнал в этой точке инвертируется — 0 (нет напряжения) становится 1 (есть напряжение) и наоборот.

Работа вентиля И с двумя инверторами на входах аналогична работе вентиля ИЛИ-НЕ:



Выход равен 1, только если оба входа равны 0.

Так же и вентиль ИЛИ с двумя инвертированными входами эквивалентен вентилю И-НЕ:



Выход равен 0, только если оба входа равны 1.

Две этих пары эквивалентных схем являются электрическим воплощением *законов Моргана*. Огастес Морган (Augustus De Morgan) на 9 лет старше Буля. Его книга «Формальная логика» опубликована в 1847 г., в тот же самый день (как гласит предание), что и «Математический анализ логики» Буля. Больше того, на занятия логикой Буля подвигла открытая вражда между Морганом и другим британским математиком, связанная с обвинениями в плагиате (история Моргана оправдала). Морган с самого начала понял важность исследований Буля. Он бескорыстно вдохновлял Буля и помогал ему, но сегодня, увы, практически забыт за исключением своих знаменитых законов.

Проще всего записать законы Моргана так:

$$\overline{A} \times \overline{B} = \overline{A + B}$$
$$\overline{A} + \overline{B} = \overline{A \times B}$$

Здесь A и B — булевы операнды. В первом выражении они инвертируются, а затем объединяются булевым оператором И. Оказывается, что это эквивалентно логическому сложению этих же операндов с последующим инвертированием суммы (что соответствует оператору ИЛИ-НЕ). Во втором выражении операнды инвертируются, а затем объединяются булевым

оператором ИЛИ. Это выражение эквивалентно булеву умножению операндов с последующим инвертированием произведения (что соответствует оператору И-НЕ).

Законы Моргана — важный инструмент для упрощения булевых выражений и соответствующих им электрических цепей. Исторически именно в этом соответствии и состоит смысл статьи Клода Шеннона. В этой книге мы не ставим перед собой цель до умопомрачения упрощать электрические цепи. Всегда предпочтительнее получить работающее устройство, а не устройство, работающее с максимальной простотой. Именно этим мы и займемся в следующей главе — создадим действующее устройство для суммирования чисел.

Глава 12

Двоичный сумматор

Сложение — основное арифметическое действие, и потому, если мы хотим построить компьютер (а в этом и заключается тайная цель моей книги), надо прежде всего разобраться, как создать устройство, которое складывало бы два числа. Когда вы сделаете это, то поймете, что сложение — это практически *единственное* действие, которое выполняют компьютеры. Разработав суммирующее устройство, мы окажемся на пути к созданию прибора, в котором сложение используется для вычитания, умножения, деления, расчета выплат по закладной, вычисления траектории полета на Марс, игры в шахматы и обогащения провайдеров Интернета.

Сумматор, который мы создадим в этой главе, будет громоздким, неуклюжим, медленным и шумным, по крайней мере в сравнении с современными калькуляторами и компьютерами. Самое интересное, что мы соберем его из простейших электронных устройств, изученных нами в предыдущих главах: переключателей, лампочек, проводов, источника питания и реле, объединенных в различные логические вентили. В этом сумматоре не будет ни единой детали, не изобретенной минимум 120 лет назад. Что особенно приятно, мы не будем захламлять всей этой электроникой вашу квартиру. Мы построим сумматор на бумаге и в воображении.

Наш сумматор работает только с двоичными числами и лишен некоторых современных прелестей. В нем, например,

нельзя применять для ввода складываемых чисел клавиатуру. Вместо нее вы будете использовать вереницу переключателей. Роль дисплея для отображения результата будет играть ряд лампочек.

И все же эта машина будет находить сумму двух чисел, причем почти так же, как это происходит в современном компьютере.

Сложение двоичных чисел очень похоже на сложение десятичных. Чтобы сложить два десятичных числа, например, 245 и 673, вы разбиваете задачу на несколько простых шагов. На каждом шаге требуется сложить две десятичных цифры. В данном примере суммирование начинается с цифр 5 и 3. Сделать это гораздо проще, если вы на каком-то жизненном этапе выучили таблицу сложения.

Важное различие между десятичными и двоичными числами в том, что таблица сложения для двоичных чисел значительно проще аналогичной таблицы для десятичных чисел.

+	**0**	**1**
0	0	1
1	1	10

Если бы вы учились в дельфиньей школе, вам пришлось бы учить именно эту таблицу. Вы, наверное, читали бы ее вслух, хором.

0 плюс 0 равно 0.
0 плюс 1 равно 1.
1 плюс 0 равно 1.
1 плюс 1 равно 0 и 1 в уме.

Можно переписать эту таблицу с дополнительным нулями, чтобы каждый результат был двухбитовой величиной.

+	**0**	**1**
0	00	01
1	01	10

Результат сложения пары двоичных чисел, представленный подобным образом, содержит 2 бита: *разряд суммы* (sum bit) и *разряд переноса* (carry bit). Теперь можно разделить таблицу

сложения двоичных чисел на две, первая из которых предназначена для разряда суммы.

+ **сумма**	**0**	**1**
0	0	1
1	1	0

А вторая — для разряда переноса.

+ **перенос**	**0**	**1**
0	0	0
1	0	1

Двоичное сложение удобно рассматривать именно с такой точки зрения, так как в нашем сумматоре суммы и переносы будут вычисляться раздельно. Построение сумматора состоит в проектировании схемы, которая выполняла бы эти операции. Работа в двоичной системе здорово упрощает нашу задачу, так как все части схемы — переключатели, лампочки и провода — могут символизировать двоичные разряды.

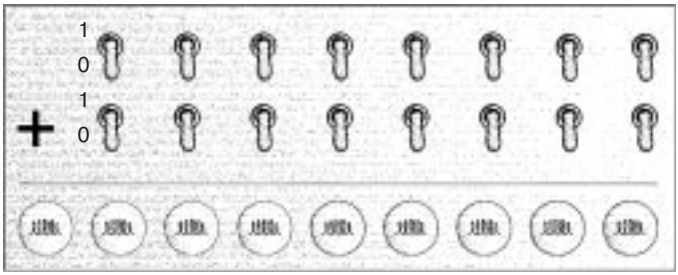
Как и в десятичном сложении, мы будем складывать цифры столбец за столбцом, начиная с крайнего правого:

```
 01100101
+10110110
---------
100011011
```

Заметьте, что при сложении цифр в третьем столбце справа 1 переносится в следующую колонку. То же самое происходит в шестом, седьмом и восьмом столбцах справа.

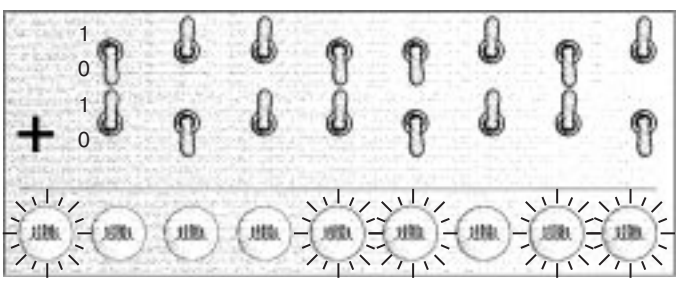
Какого размера двоичные числа мы собираемся складывать? Конечно, воображаемый сумматор в принципе способен складывать числа любой разрядности. Но давайте ограничимся разумной длиной и будем складывать числа не длиннее 8 битов, т. е. в диапазоне от 0000-0000 до 1111-1111, или в десятичном выражении от 0 до 255. Сумма двух 8-битовых чисел может достигать значения 1-1111-1110, или 510.

Пульт управления нашим сумматором может выглядеть примерно так.



На пульте размещены два ряда переключателей по 8 в каждом. Это устройство ввода, которое мы будем использовать для ввода 8-разрядных чисел. В нем нижнее положение переключателя (выключен) соответствует вводу 0, а верхнее (включен) — вводу 1. Устройство вывода в виде ряда из 9 лампочек находится в нижней части пульта. Эти лампочки будут показывать ответ. Горящая лампочка соответствует 1, выключенная — 0. Нам нужны 9 лампочек, так как сумма двух 8-разрядных чисел может быть 9-разрядным числом.

В остальном сумматор состоит из логических вентилей, соединенных разными способами. Переключатели будут вызывать срабатывание реле в логических вентилях, которые подадут питание на нужные лампочки. Например, чтобы сложить 0110-0101 и 1011-0110, мы включим переключатели, как показано на рисунке.



Горящие лампочки показывают ответ — 1-0001-1011 (пока примем его на веру — ведь мы еще ничего не собирали!).

Я говорил в предыдущей главе, что намерен в этой книге активно использовать реле. Создаваемый нами 8-разрядный сумматор будет состоять из 144 реле, по 18 для каждой из 8 пар

битов, которые мы будем складывать. Если бы я показал вам схему соединений всех этих реле, вы определенно посинели бы. Нет абсолютно никакой надежды, что кто-то сумеет разобраться в хитросплетениях схемы, состоящей из 144 реле. Чтобы не сойти с ума до срока, мы предпримем модульный подход к решению этой проблемы, а помогут нам логические вентили.

Вполне вероятно, что вы уже увидели связь между логическими вентилями и двоичным сложением. Для этого достаточно взглянуть на таблицу переноса разряда при сложении двух однобитовых чисел.

+ **перенос**	**0**	**1**
0	0	0
1	0	1

Она ведь совершенно идентична таблице с результатами работы вентиля И, описанного в предыдущей главе.

И	**0**	**1**
0	0	0
1	0	1

Итак, вентиль И считает перенос разряда при сложении двух двоичных цифр.

Ага! Мы подбираемся к сути. Очередной шаг состоит, видимо, в проверке, ведут ли себя какие-нибудь реле следующим образом:

+ **сумма**	**0**	**1**
0	0	1
1	1	0

Это вторая часть задачи сложения пары двоичных цифр. Разряд суммы получается не столь прямолинейно, как разряд переноса, но в дальнейшем мы решим и эту проблему.

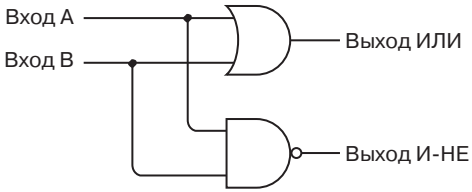
Для начала сообразим, что вентиль ИЛИ дает почти то, что нужно, не считая ячейки в правом нижнем углу таблицы.

ИЛИ	**0**	**1**
0	0	1
1	1	1

Результат действия вентиля И-НЕ тоже близок к нашим потребностям, кроме верхней левой ячейки таблицы.

И-НЕ	**0**	**1**
0	1	1
1	1	0

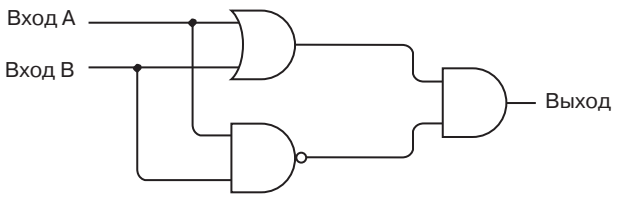
Подключим к одним и тем же входам вентиль ИЛИ и вентиль И-НЕ.



В следующей таблице возможные сигналы на выходах соединенных вентилей ИЛИ и И-НЕ сравниваются с тем, что нужно для сумматора.

Вход A	**Вход B**	**Выход ИЛИ**	**Выход И-НЕ**	**Что нужно**
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Обратите внимание: единица на выходе нам нужна, только если единице равны выход вентиля ИЛИ *и* выход вентиля И-НЕ. Это наводит на мысль, что два этих выхода могут быть входами вентиля И.

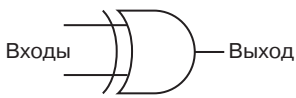


То, что нужно.

Отметим, что во всей схеме по-прежнему всего два входа и один выход. Два входа принадлежат как вентилю ИЛИ, так и вентилю И-НЕ. Два выхода от вентиля ИЛИ и от вентиля И-НЕ идут на вход вентиля И, который и выдает необходимый нам результат.

Вход А	Вход В	Выход ИЛИ	Выход И-НЕ	Выход И
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

У созданной нами схемы есть собственное имя — *исключающее ИЛИ* (Exclusive OR, Искл-ИЛИ). Исключающей схема названа потому, что ее выход равен 1, если единичный сигнал есть на входе А *или* на входе В, но не на обоих. Чтобы не рисовать каждый раз вентили ИЛИ, И-НЕ и И, мы будем использовать для схемы *Искл-ИЛИ* обозначение:



Оно очень похоже на символ вентиля ИЛИ за исключением того, что на нем со стороны входа нарисована еще одна кривая линия. Поведение вентиля Искл-ИЛИ проиллюстрировано в таблице.

Искл-ИЛИ	0	1
0	0	1
1	1	0

Вентиль Искл-ИЛИ — последний, подробно описанный в этой книге. На практике иногда применяют еще один вентиль — *вентиль совпадения* (coincidence gate) или *эквивалентности* (equivalence), выход которого равен 1, только если сигналы на обоих входах одинаковы. На выходе вентиль совпадения ведет себя противоположно вентилю Искл-ИЛИ, поэтому обозначается почти так же, только с кружком со стороны выхода.

Подведем итог. Сложение двух двоичных чисел приводит к появлению бита суммы и бита переноса.

+ сумма	0	1
0	0	1
1	1	0

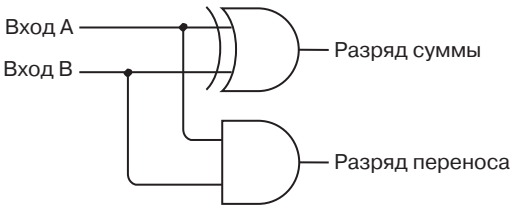
+ перенос	0	1
0	0	0
1	0	1

Разряд суммы двух двоичных чисел задается выходом вентиля Искл-ИЛИ, а разряд переноса — выходом вентиля И.

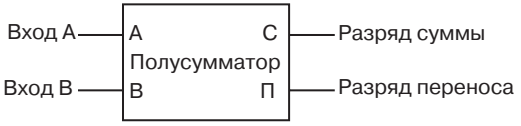
Искл-ИЛИ	0	1
0	0	1
1	1	0

И	0	1
0	0	0
1	0	1

Для сложения двух двоичных цифр A и B мы можем объединить два этих вентиля.



Чтобы не рисовать многократно вентили И и Искл-ИЛИ, заменим эту схему обозначением

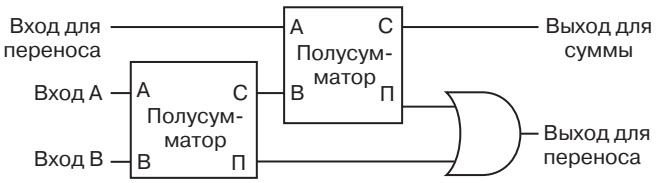


Полусумматором (half-adder) эта схема названа неспроста. Конечно, он складывает две двоичные цифры и выдает разряд суммы и разряд переноса. Но подавляющее большинство двоичных чисел записывается несколькими битами. Наш полусумматор не делает одной важной вещи: не прибавляет к сумме возможный разряд переноса от предыдущего суммирования. Допустим, мы складываем два двоичных числа, как показано ниже:

```
  1111
+1111
------
 11110
```

Полусумматор можно использовать только для сложения крайнего правого столбца: 1 плюс 1 равно 0, и 1 идет в перенос. Для второго столбца справа из-за переноса нам на самом деле нужно сложить *три* двоичных цифры. То же относится и к остальным столбцам. Каждое последующее сложение двух двоичных цифр может включать перенос из предыдущего столбца.

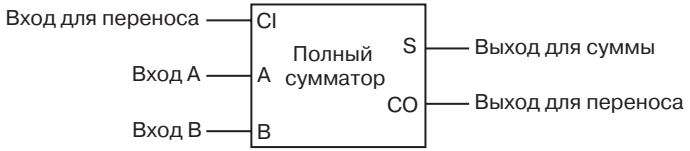
Чтобы сложить три двоичных цифры, нам нужны два полусумматора и вентиль ИЛИ:



Чтобы разобраться в работе этой схемы, начнем со входов А и В в первом полусумматоре (том, что слева). Результатом его работы являются сумма и перенос. Эту сумму нужно прибавить к переносу из предыдущего столбца, поэтому оба числа подаются на вход второго полусумматора. Сумма из второго полусумматора будет окончательной суммой. Два переноса из полусумматоров попадают на входы вентиля ИЛИ. В этом месте, конечно, можно использовать еще один полусумматор, который выполнит необходимые действия. Но, проанализировав возможные варианты, вы убедитесь, что выходы для переноса из двух полусумматоров никогда не будут одновременно равны 1. Для их сложения вполне достаточно вентиля

ИЛИ, который действует аналогично вентилю Искл-ИЛИ, если его входы не равны 1 одновременно.

Теперь назовем эту схему *полным сумматором* (full adder) и введем для нее обозначение (расшифровка английских обозначений будет дана чуть позже):

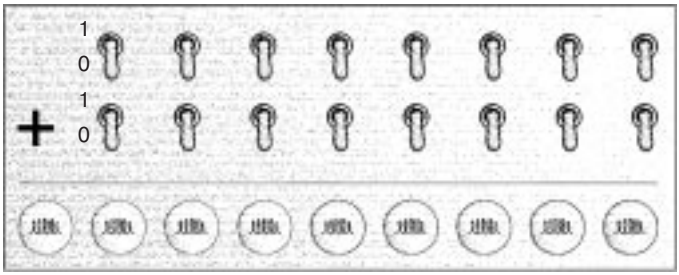


В следующей таблице приводятся все возможные комбинации входных и выходных сигналов полного сумматора.

Вход A	Вход B	Вход для переноса	Выход для суммы	Выход для переноса
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

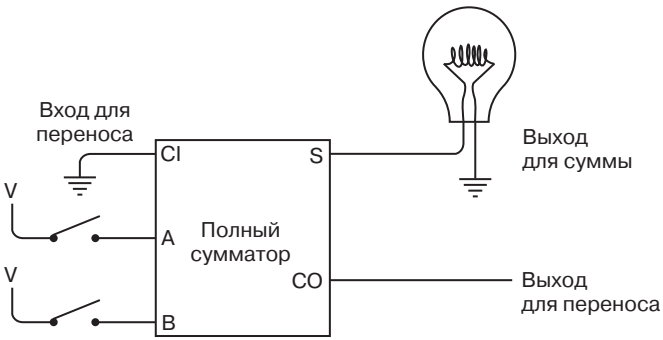
Я говорил чуть раньше, что для всего сумматора нам понадобится 144 реле. Теперь я могу пояснить это число. В каждом вентиле И, ИЛИ и И-НЕ по 2 реле. Таким образом, вентиль Искл-ИЛИ построен из 6 реле. Полусумматор состоит из вентиля Искл-ИЛИ и вентиля И, т. е. из 8 реле. Каждый полный сумматор состоит из двух полусумматоров и одного вентиля ИЛИ, итого 18 реле. Для нашего сумматора нам нужны 8 полных сумматоров. Всего получается 144 реле.

Вспомним пульт управления с переключателями и лампочками.



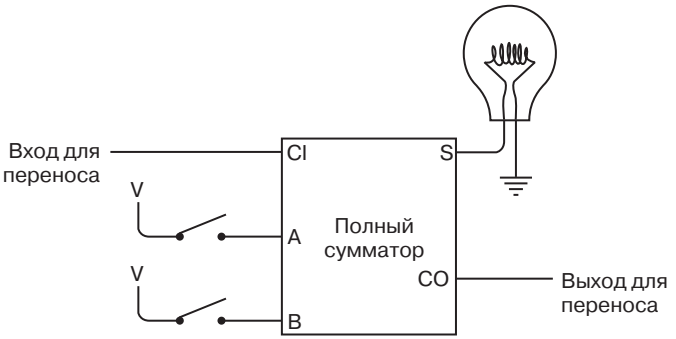
Можем приступать к соединению полных сумматоров с переключателями и лампочками.

Начнем с того, что соединим два крайних правых переключателя, крайнюю правую лампочку и полный сумматор.



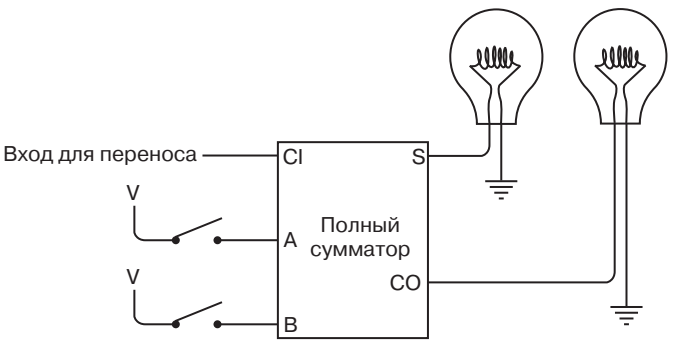
При сложении двух двоичных чисел первый столбец цифр, которые вы складываете, не похож на другие тем, что в отличие от последующих столбцов в нем не может быть разряда переноса из предыдущего столбца. Поэтому вход для переноса у первого полного сумматора соединяется с землей. Это значит, что его значение всегда 0. Конечно, разряд переноса может появиться в *результате* сложения первой пары двоичных чисел.

Со следующей парой переключателей и следующей лампочкой полный сумматор соединяется так:



Выход для переноса из первого полного сумматора становится входом для переноса во второй полный сумматор. Схемы для сложения всех последующих столбцов аналогичны. Разряд переноса из одного столбца подается на вход для переноса следующего столбца.

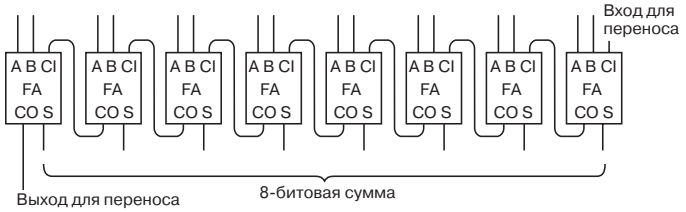
И, наконец, восьмая и последняя пара переключателей соединяется с последним полным сумматором так:



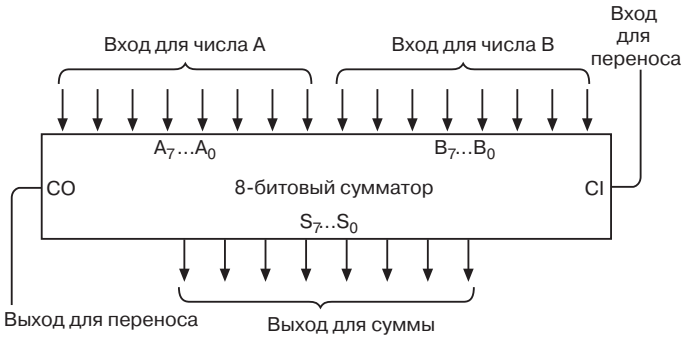
Последний выход для переноса соединяется с девятой лампочкой.

Готово!

Теперь изобразим все восемь полных сумматоров (Full Adder, FA) сразу, подключив все выходы для переноса (Carry Out, CO) к последующим входам для переноса (Carry In, CI) и обозначив сумму буквой S (Sum).



Наконец, введем единое обозначение для 8-битового сумматора. Его входы обозначены буквами от A_0 до A_7 и от B_0 до B_7, а выходы — буквами от S_0 до S_7.



Отдельные биты многобитового числа часто обозначают буквами с нижними индексами. Биты A_0, B_0 и S_0 называются *младшими* (least-significant), а A_7, B_7 и S_7 — *старшими* (most-significant). Ниже в качестве примера показано соответствие буквенных обозначений разрядам двоичного числа 0110-1001.

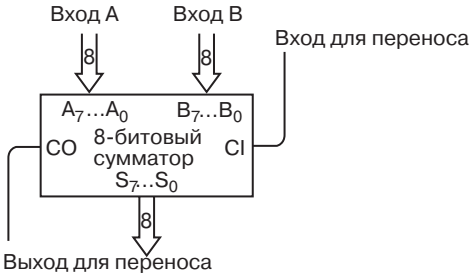
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	0	0	1

Индексы начинаются с 0 и увеличиваются с продвижением ко все более значимым цифрам, так как они соответствуют показателю степени двойки.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	1	0	0	1

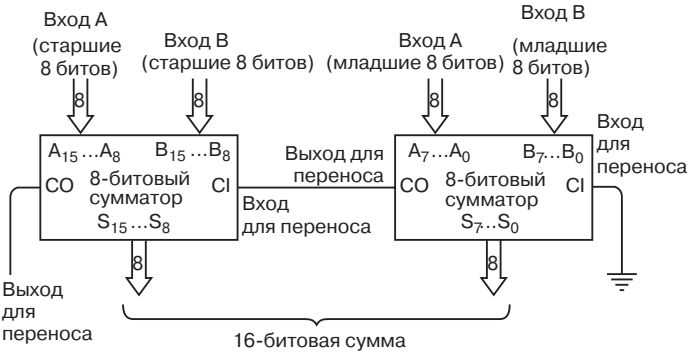
Если вы умножите каждую степень двойки на расположенную под ней цифру, а затем сложите результаты, то получите десятичный эквивалент суммы — 64 + 32 + 8 + 1 = 105.

8-битовый сумматор можно изобразить и так:



Восьмерки внутри стрелок означают, что каждая из них соответствует группе из 8 отдельных сигналов. Восьмиразрядность числа подчеркивается также нумерацией символов A_7… A_0, B_7… B_0 и S_7… S_0.

Создав один 8-битовый сумматор, вы легко построите и второй. Их можно расположить *каскадом* для сложения 16-битовых чисел.



Выход для переноса сумматора справа соединяется с входом для переноса сумматора слева. Сумматор слева получает на входе 8 старших цифр двух складываемых чисел и выдает на выходе 8 старших разрядов результата.

Теперь вы можете спросить: «Неужели в компьютерах сложение *действительно* осуществляется именно так?»

В общем, да, но не совсем.

Во-первых, сумматор может работать быстрее, чем тот, с которым мы разбирались. Взгляните, как действует цепь: выход переноса от младшей пары цифр требуется для сложения со следующей парой цифр, выход переноса второй пары цифр требуется для третьей пары цифр и т. д. Скорость суммирующей схемы равна произведению числа битов на скорость действия одного полного сумматора. Такой перенос называется *сквозным* (ripple). В быстродействующих сумматорах с помощью дополнительной цепи реализован *ускоренный* (look-ahead) *перенос*.

Во-вторых (и это более важно), в компьютерах больше не используют реле! Лишь первые цифровые компьютеры, созданные в начале 1930-х, были основаны на реле и чуть позже на вакуумных лампах. Сегодняшние компьютеры собирают из транзисторов. В компьютере транзисторы выполняют те же функции, что и реле. Разница в том, что они гораздо быстрее, компактнее, тише, экономичнее и дешевле. Для создания 8-битового сумматора вам по-прежнему потребуется 144 транзистора (или больше, если вы решите заменить сквозной перенос ускоренным), но схема будет иметь микроскопические размеры.

Глава 13

А как же вычитание?

Итак, реле действительно можно соединить проводами и складывать с помощью полученной схемы двоичные числа, но возникает вопрос: «А как же вычитание?» Задавая его, не опасайтесь прослыть занудой. На самом деле вы просто осмотрительны. Сложение и вычитание определенным образом дополняют друг друга, но механика двух этих операций различна. Сложение выполняется последовательно от крайнего правого столбца цифр к крайнему левому. Перенос из каждого столбца прибавляется к следующему столбцу. При вычитании мы не *переносим*, а *заимствуем*, и это приводит к внутренне отличной последовательности действий, усложненной своего рода движением вперед и назад.

Рассмотрим типичную задачу на вычитание, усложненную заимствованием:

$$\begin{array}{r} 253 \\ -176 \\ \hline ??? \end{array}$$

Начнем с крайнего правого столбца. Очевидно, что 6 больше 3, поэтому приходится занять 1 у 5 и вычесть 6 из 13. Получается 7. Поскольку мы заняли у пятерки единицу, она превратилась в 4. Эта цифра меньше 7, и мы занимаем 1 у 2 и вычитаем 7 из 14. Получаем 7. Вспоминаем, что заняли 1 у 2, так что вместо 2 имеем 1 и вычитаем 1 из 1. Получаем 0. Окончательный ответ 77.

```
 253
−176
 ---
  77
```

Как нам заставить набор вентилей разобраться в столь запутанной логике?

На самом деле, не стоит даже пробовать. Вместо этого мы прибегнем к небольшой хитрости, которая позволит выполнять вычитание *без* заимствования. Это порадовало бы Полония («Не занимай и не ссужай», — напутствовал он Лаэрта) да и всех остальных. Кроме того, подробное исследование этого способа вычитания полезно и с теоретической точки зрения, так как он напрямую связан с методикой хранения двоичных отрицательных чисел в компьютерах.

Для начала вспомним, что числа, участвующие в вычитании, называются *уменьшаемым* и *вычитаемым*. Вычитаемое вычитается из уменьшаемого, а результат называется *разностью*.

```
Уменьшаемое
−Вычитаемое
-----------
Разность
```

Чтобы обойтись в вычитании без заимствования, во-первых, вычтем вычитаемое *не из уменьшаемого*, а из 999:

```
 999
−176
 ---
 823
```

Здесь мы используем 999, поскольку вычитаемое является трехзначным числом. Если бы оно было четырехзначным, мы вычли бы его из 9999. Результат вычитания числа из строки девяток называется *дополнением до девяти* (nines' complement). Дополнение до девяти числа 176 есть 823. Верно и обратное: дополнение до девяти числа 823 есть 176. И что приятно: каким бы ни было вычитаемое, вычисление его дополнения до девяти *никогда не требует заимствования*.

Вычислив дополнение числа до девяти, прибавьте к нему уменьшаемое:

```
 253
+823
 ---
1076
```

Наконец, сложите результат с 1 и вычтите 1000:

$$\begin{array}{r} 1076 \\ +1 \\ \hline -1000 \\ \hline 77 \end{array}$$

Готово. Мы получили тот же результат, что и раньше, без единого заимствования.

Как это получилось? Исходная задача на вычитание выглядит так:

$$253 - 176$$

К этому выражению можно прибавить и вычесть любое число — результат от этого не изменится. Прибавим и вычтем 1000.

$$253 - 176 + 1000 - 1000$$

Это выражение эквивалентно выражению

$$253 - 176 + 999 + 1 - 1000$$

Эти числа можно перегруппировать:

$$253 + (999 - 176) + 1 - 1000.$$

Это как раз и есть тот расчет, что я провел, используя дополнение вычитаемого до девяти. Мы заменили одно вычитание двумя вычитаниями и двумя сложениями, попутно освободившись от всех заимствований.

Что делать, если вычитаемое больше уменьшаемого? Ведь задача на вычитание может выглядеть и так:

$$\begin{array}{r} 176 \\ -253 \\ \hline ??? \end{array}$$

Опытный математик, глядя на это, говорит: «Хммм. Я вижу, что вычитаемое больше уменьшаемого. Я меняю числа местами и произвожу вычитание, помня при этом, что результат будет отрицательным». Ответ записывается следующим образом:

$$\begin{array}{r} 176 \\ -253 \\ \hline -77 \end{array}$$

Чтобы выполнить такой расчет без заимствования, придется поступить несколько иначе, чем в предыдущем примере. Начинаем опять с вычисления дополнения вычитаемого (253) до девяти:

```
 999
–253
 746
```

Теперь складываем дополнение до девяти с уменьшаемым:

```
 176
+746
 922
```

В предыдущей задаче на этом этапе нужно было прибавить 1 и вычесть 1000, чтобы получить окончательный ответ. В данном случае эта стратегия не сработает. Пришлось бы вычесть 1000 из 923, а это в реальности оборачивается вычитанием 923 из 1000 и требует заимствований.

На самом деле, вычисляя дополнение вычитаемого до девяти, мы фактически прибавили к нему 999, поэтому теперь будем вычитать 999:

```
 922
–999
 ???
```

Глядя на этот пример, мы понимаем, что ответ будет отрицательным. Поэтому числа нужно поменять местами и вычесть 922 из 999. Это действие снова не требует заимствования. Ответ, как мы и ожидали, равен –77:

```
 922
–999
  77
```

При вычитании двоичных чисел используется точно такой же метод. Более того, он проще, чем в десятичном случае. Посмотрим, как он работает.

Вот как выглядит наша задача на вычитание:

```
 253
–176
 ???
```

Записав эти числа в двоичном представлении, получаем задачу такого вида:

```
  11111101
–10110000
----------
  ?????????
```

Шаг 1. Вычтем вычитаемое из 11111111 (т. е. 255).

```
  11111111
–10110000
----------
  01001111
```

Когда мы работали с десятичными числами, вычитаемое вычиталось из строки девяток, а результат назывался дополнением до девяти. При расчетах с двоичными числами вычитаемое вычитается из строки единиц, и результат называется *дополнением до единицы* (ones' complement). Но заметьте, чтобы получить дополнение до единицы, вычитать на самом деле не нужно. И вот почему: каждый 0 в исходном числе становится 1 в дополнении, а каждая 1 превращается в 0. Именно поэтому дополнение до единицы иногда также называют *отрицанием* (negation) или *инверсией* (inversion). Здесь уместно вспомнить, что в главе 11 мы встречались с инвертором, который как раз и меняет 0 на 1, а 1 на 0.

Шаг 2. Складываем дополнение вычитаемого до 1 с уменьшаемым.

```
  11111101
+01001111
----------
  101001100
```

Шаг 3. Прибавляем к результату 1.

```
  101001100
          +1
----------
  101001101
```

Шаг 4. Вычитаем 100000000 (т. е. 256).

```
  101001101
–100000000
----------
    1001101
```

Этот результат соответствует десятичному числу 77.

Теперь повторим последовательность действий, выполненных для тех же чисел, но с заменой вычитаемого на уменьшаемое. В десятичном виде эта задача записывалась так:

```
  176
 −253
 ----
  ???
```

В двоичном она выглядит так:

```
  10110000
 −11111101
 ---------
  ?????????
```

Шаг 1. Вычитаем вычитаемое из 11111111, чтобы получить его дополнение до единицы.

```
  11111111
 −11111101
 ---------
  00000010
```

Шаг 2. Прибавляем дополнение вычитаемого до единицы к уменьшаемому.

```
  10110000
 −00000010
 ---------
  10110010
```

Теперь из результата нужно вычесть 11111111. Когда вычитаемое было меньше уменьшаемого, эта задача решалась добавлением 1 и вычитанием 100000000. Теперь сделать это без заимствования не удастся, поэтому мы вычитаем полученный результат из 11111111:

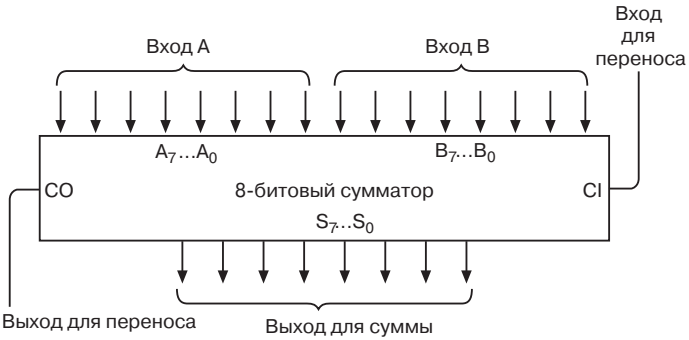
```
  11111111
 −10110010
 ---------
  01001101
```

Как и ранее, это действие эквивалентно инвертированию. В ответе получилось 77, но мы помним, что в действительности это −77.

Теперь у нас есть все необходимые познания для модернизации сумматора из предыдущей главы, чтобы он выполнял не только сложение, но и вычитание. Чтобы не слишком ус-

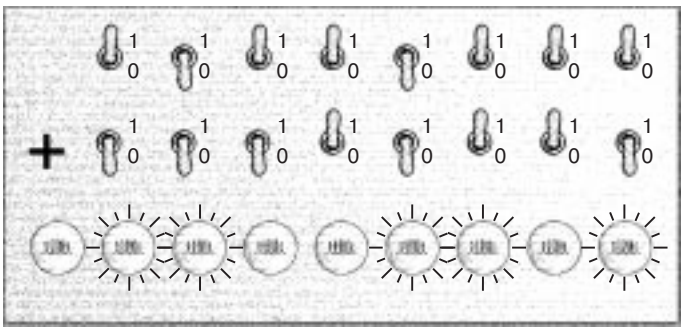
ложнять задачу, наша новая суммирующая и вычитающая машина будет выполнять вычитание, только когда вычитаемое меньше уменьшаемого, т. е. когда результат положителен.

Основой суммирующей машины был 8-битовый сумматор, собранный из логических вентилей:



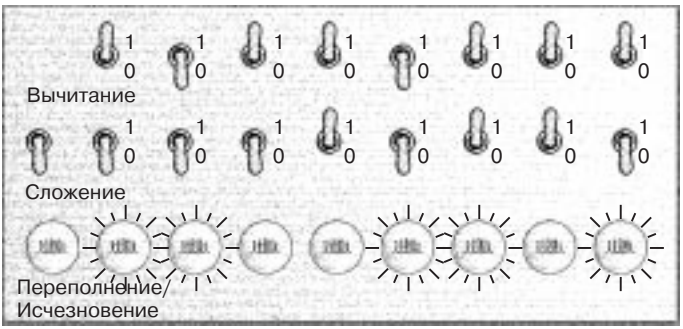
Как вы помните, входы с A_7 по A_0 и с B_7 по B_0 соединялись с переключателями, задававшими два 8-битовых слагаемых. Вход для переноса соединялся с землей. Выходы с S_7 по S_0 соединялись с 8 лампочками, отображавшими результат сложения. Поскольку результатом сложения могла быть 9-битовая величина, выход для переноса соединялся с девятой лампочкой.

Пульт управления сумматором выглядел примерно так:



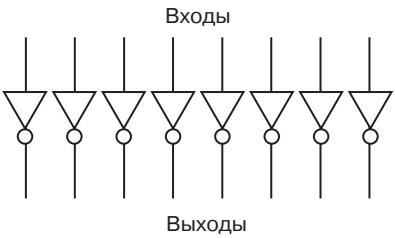
На этом рисунке переключатели установлены в положения, соответствующие сложению 183 (10110111) и 22 (00010110). Результат сложения, о чем свидетельствуют лампочки, равен 205 (11001101).

Вид пульта управления для сложения или вычитания двух 8-битовых чисел слегка иной. На нем появился дополнительный переключатель, с помощью которого можно задать действие — вычитание или сложение.



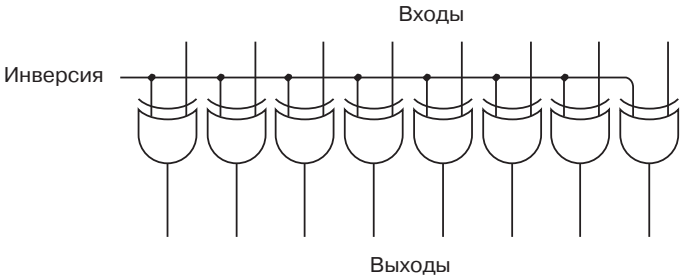
Если этот переключатель разомкнут, производится сложение, если замкнут — вычитание. Кроме того, теперь для отображения результата используются только правые 8 лампочек. Девятая лампочка подписана «Переполнение/Исчезновение». Она сигнализирует, что вычисленное число не может быть представлено только 8 лампочками. Это происходит, если сумма больше 255 (переполнение разрядов) или если разность оказалась отрицательной (исчезновение разрядов).

Основным новшеством в сумматоре станет схема, которая вычисляет дополнение 8-разрядного числа до единицы. Как вы помните, это эквивалентно инвертированию битов, так что для вычисления дополнения 8-разрядного числа до единицы можно использовать просто 8 инверторов.



Но эта схема *всегда* инвертирует биты, поступающие на ее входы, а в машине для сложения и вычитания нам нужно устрой-

ство, которое инвертировало бы биты, только если производится вычитание. Более подходящая схема выглядит так:

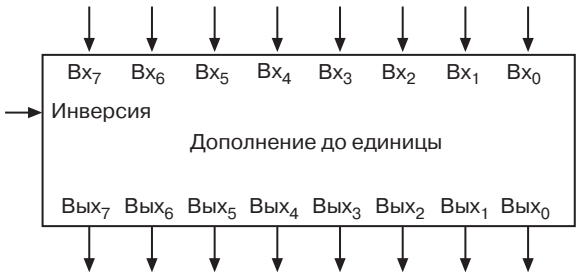


Сигнал Инверсия подается на входы всех девяти вентилей Искл-ИЛИ. Напомню, что вентиль Искл-ИЛИ работает так:

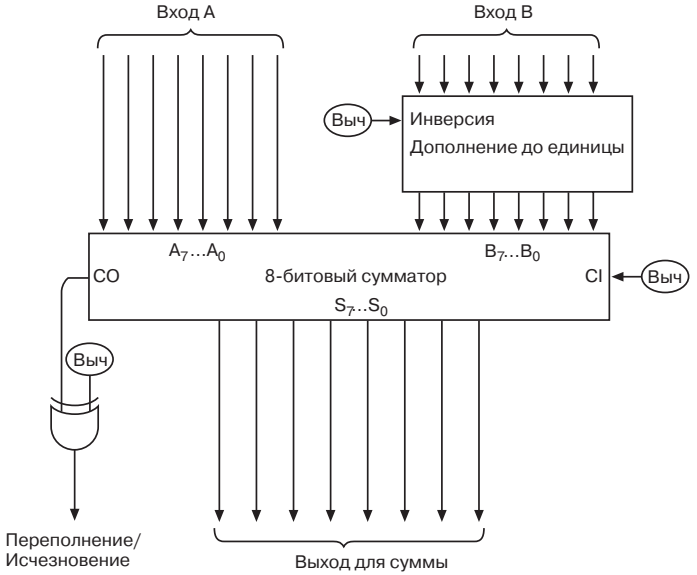
Искл-ИЛИ	**0**	**1**
0	0	1
1	1	0

Если сигнал Инверсия равен 0, восемь выходов вентилей Искл-ИЛИ повторяют сигнал на восьми их входах. Например, если на вход подается число 01100001, на выходе также будет 01100001. Если же сигнал Инверсия равен 1, восемь входных сигналов будут инвертироваться. Если на входе 01100001, на выходе имеем 10011110.

Давайте поместим восемь вентилей Искл-ИЛИ в общий прямоугольник под названием *дополнение до единицы* (ones' complement):



Теперь вентиль «дополнение до единицы», 8-битовый сумматор и выходной вентиль Искл-ИЛИ можно собрать в единую схему:



Обратите внимание на сигнал Выч, идущий с переключателя «Сложение/Вычитание». Этот сигнал равен 0, если нужно выполнить сложение, и 1, если нужно выполнить вычитание. При вычитании входы B (второй ряд переключателей) перед сумматором инвертируются схемой «дополнение до единицы», а к результату сумматора прибавляется 1, для чего его вход CI установлен в 1. При сложении схема «дополнение до единицы» не делает ничего; сигнал на входе CI равен 0.

Сигнал Выч и выход CO (выход для переноса) сумматора также поступают в вентиль Искл-ИЛИ, который управляет включением лампочки «Переполнение/Исчезновение». При равенстве 0 сигнала Выч (выполняется сложение) лампочка горит, если сигнал CO сумматора равен 1, т. е. результат сложения оказался больше 255.

Пусть вас не смущает, что выход CO сумматора равен 1, если выполняется вычитание и вычитаемое (переключатели B) меньше уменьшаемого (переключатели A). Так проявляется число 100000000, которое в этом случае должно вычитаться на заключительном шаге. Лампочка «Переполнение/Исчезновение» горит, только если выход CO сумматора равен 0. Это

означает, что вычитаемое больше уменьшаемого и результат отрицателен. Отображение отрицательных чисел в нашем устройстве не предусмотрено.

Теперь вы, верно, радуетесь, что спросили: «А как же вычитание?»

Я в этой главе уже неоднократно говорил об отрицательных числах, но все еще не показал, на что они похожи в двоичном представлении. Можно, конечно, обозначать отрицательные двоичные числа знаком «минус», как и в десятичной системе, скажем, записать число –77 как –1001101. Но одна из целей введения двоичных чисел в том, чтобы представлять в виде нулей и единиц *все что угодно*, в том числе и знак «минус» перед отрицательным числом.

Признаком знака может служить и дополнительный бит, который, например, равнялся бы 1 у отрицательных чисел и 0 у положительных. Однако есть другой способ представления отрицательных чисел, который позволяет без особых хлопот складывать отрицательные и положительные числа. Недостаток его лишь в том, что вы должны заранее решить, сколько цифр понадобится для представления чисел, с которыми вам предстоит иметь дело.

Давайте малость подумаем. Преимущество стандартного способа записи положительных и отрицательных чисел в том, что он не накладывает никаких ограничений на их величины. Как отрицательные, так и положительные числа расходятся от 0 до бесконечности.

... –1000000 –999999 ... –3 –2 –1 0 1 2 3 ... 999999 1000000 ...

Но допустим, что бесконечный числовой ряд нам не нужен, так как мы заранее знаем, что все числа, с которыми мы столкнемся, заключены в определенных пределах.

Рассмотрим в качестве примера чековый банковский счет, где нам иногда приходится на практике сталкиваться с отрицательными числами. Предположим, что мы никогда не держали на счете больше 500 долларов и что банк разрешает кратковременный перерасход также на 500 долларов. Это значит, что баланс нашего чекового счета всегда заключен между 499 долларами и –500 долларами. Допустим также, что мы никогда не вносим на счет больше 500 долларов, никогда не выпи-

сываем чеки больше, чем на 500 долларов, и, наконец, имеем дело только с долларами, не обращая внимания на центы.

Этот набор условий означает, что в работе со счетом нам никогда не приходится иметь дело с числами, выходящими за пределы от –500 до 499. Всего в этом диапазоне 1000 целых чисел. Это значит, что для представления всех нужных чисел мы можем использовать три десятичные цифры и не прибегать к знаку «минус». Хитрость в том, что нам не нужны положительные числа из диапазона от 500 до 999, так как по нашим условиям максимальное положительное число, в котором мы нуждаемся, — 499. Значит, с помощью трехзначных чисел из диапазона от 500 до 999 можно представлять отрицательные числа. Вот как это сделать:

Вместо –500 используем 500
Вместо –499 используем 501
Вместо –498 используем 502
…
Вместо –2 используем 998
Вместо –1 используем 999
Вместо 0 используем 000
Вместо 1 используем 001
Вместо 2 используем 002
…
Вместо 497 используем 497
Вместо 498 используем 498
Вместо 499 используем 499

Иначе говоря, все трехзначные числа, начинающиеся с 5, 6, 7, 8 и 9, представляют отрицательные значения. Вместо записи чисел с минусами

–500 –499 –498 … –4 –3 –2 –1 0 1 2 3 4 … 497 498 499

мы записываем их так:

500 501 502 … 996 997 998 999 000 001 002 003 004 … 497 498 499

Заметьте: числа идут по кругу. Наименьшее отрицательное число (500) выглядит как продолжение наибольшего положительного (499). Число 999 (представляющее –1) на единицу меньше 0. Если прибавить к 999 единицу, мы получим 1000,

но поскольку мы имеем дело только с тремя цифрами, в действительности получаем 000.

Такой способ обозначения отрицательных чисел называется *дополнением до десяти* (ten's complement). Чтобы преобразовать трехзначное отрицательное число в дополнение до 10, вычитаем его из 999 и добавляем 1. Иными словами, дополнение до десяти — это дополнение до девяти плюс 1. Например, чтобы переделать –255 в дополнение до десяти, вычтем его из 999, получив 744, и прибавим 1, что дает 745.

Вам, вероятно, приходилось слышать, что «вычитание — это просто сложение с отрицательным числом». На это вы скорее всего отвечали: «Да, но при этом оно остается *вычитанием*». Дополнение до десяти позволяет избавиться от вычитания, полностью заменив его сложением.

Пусть на вашем счете 143 доллара. Вы выписали чек на 78 долларов. Это означает, что к 143 нужно прибавить –78. Дополнение –78 до десяти равно 999 – 078 + 1, или 922. Таким образом, ваш баланс составляет 143 + 922, что равно 65 долларам (без учета переполнения). Если вы после этого выпишете чек на 150 долларов, то должны будете прибавить к балансу число –150, дополнение которого до десяти равно 850. Сумма предыдущего баланса 065 и 850 равна 915. В обычном представлении число 915 эквивалентно –85 долларам — вашему новому балансу.

Аналогичная система в двоичном счислении называется *дополнением до двух* (two's complement). Предположим, что мы работаем только с 8-битовыми числами, заключенными в пределах от 00000000 до 11111111, или от 0 до 255 в десятичной системе. Чтобы отображать с их помощью также и отрицательные числа, придется отдать в отрицательную область все 8-битовые числа, начинающиеся с 1, как показано в таблице.

Двоичное	**Десятичное**
10000000	–128
10000001	–127
10000010	–126
10000011	–125
…	
11111101	–3

(*продолжение*)

Двоичное	Десятичное
11111110	–2
11111111	–1
00000000	0
00000001	1
00000010	2
…	
01111100	124
01111101	125
01111110	126
01111111	127

Диапазон чисел, которые вы теперь можете представить, ограничен пределами от –128 до +127. Старший значащий бит (крайний слева) называется *знаковым разрядом* (sign bit). Знаковый разряд равен 1 для отрицательных чисел и 0 для положительных.

Чтобы вычислить дополнение до двух, нужно посчитать дополнение до единицы и прибавить 1 или, что эквивалентно, инвертировать все цифры и прибавить 1. Например, десятичное число 125 в двоичной системе выглядит как 01111101. Чтобы представить –125 в виде дополнения до двух, инвертируем цифры в числе 01111101, получая в результате 10000010, а затем прибавляем единицу, что дает 10000011. Проверьте результат по таблице. Чтобы проделать обратную операцию, нужно сделать то же самое — инвертировать все биты и прибавить 1.

Эта система позволяет выражать положительные и отрицательные числа без знака «минус», а также складывать положительные и отрицательные числа, используя только правила сложения. Давайте в качестве примера сложим двоичные эквиваленты –127 и 124. Используя предыдущую таблицу как шпаргалку, запишем

```
 10000001
+01111100
---------
 11111101
```

Этот результат эквивалентен десятичному числу –3.

При этом необходимо следить за переполнением или исчезновением разрядов. Такое случится, если результат сложения окажется больше 127 или меньше –128. Допустим, мы хотим сложить число 125 с ним самим.

```
 01111101
+01111101
--------
 11111010
```

Старший бит равен 1, т. е. результат интерпретируется как отрицательное число и становится равным –6. То же самое происходит, если с самим собой вы сложите число –125.

```
 10000011
+10000011
--------
100000110
```

Мы условились в начале, что ограничимся 8-битовыми числами, поэтому крайнюю левую цифру результата приходится игнорировать. Правые же 8 битов эквивалентны +6.

Вообще результат сложения положительных и отрицательных чисел неверен, если знаковые разряды двух операндов совпадают, а знаковый разряд результата от них отличается.

Итак, есть два способа использования двоичных чисел. Двоичное число может быть *со знаком* (signed) или *без знака* (unsigned). 8-битовые числа без знака попадают в диапазон от 0 до 255. 8-битовые числа со знаком попадают в диапазон от –128 до 127. Только по виду числа вы не сможете определить, является оно числом со знаком или без знака. Например, кто-то скажет вам: «У меня есть 8-битовое двоичное число, равное 10110110. Каков его десятичный эквивалент?» Вам придется осведомиться: «Со знаком или без знака? Это может быть либо –74, либо 182».

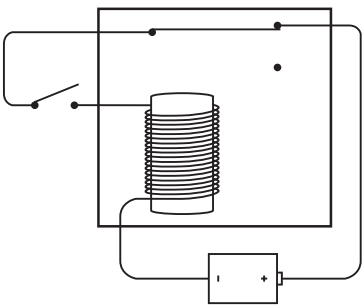
Вот такие они, биты — всего лишь нули и единицы, которые ничего не говорят вам *о себе*.

Глава 14

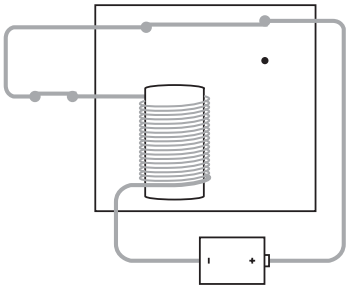
Обратная связь и триггеры

Все знают, что электричество приводит предметы в движение. Даже при поверхностном осмотре типичного жилища вы обнаружите электродвигатели в таких разных устройствах, как часы, вентиляторы, кухонные комбайны и проигрыватели компакт-дисков. Электричество заставляет вибрировать конус громкоговорителя, и вы слышите звуки, речь и музыку, воспроизводимые магнитофоном или телевизором. Но, вероятно, самый простой и элегантный способ превратить электричество в движение используется в электрических зуммерах и звонках, которые уже практически превратились в антиквариат.

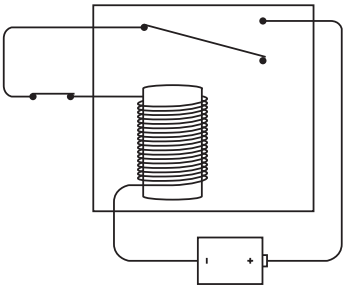
Взгляните на реле, соединенное с переключателем и электрической батарейкой.



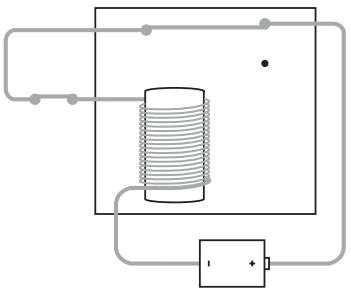
Странноватый способ соединения, правда? Реле, подключенное таким образом, нам еще не встречалось. Обычно вход реле отделен от выхода, здесь же они образуют кольцо. Когда вы замыкаете переключатель, цепь становится непрерывной, и по ней течет ток.



Ток заставляет электромагнит притянуть гибкую металлическую полоску.

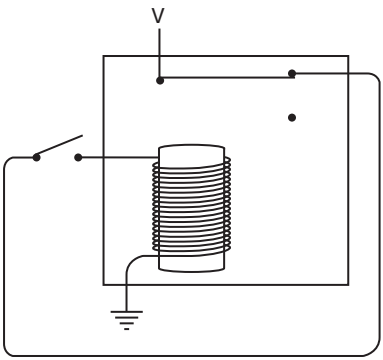


Но как только она смещается, цепь размыкается, электромагнит теряет свои магнитные свойства, полоска возвращается в первоначальное положение

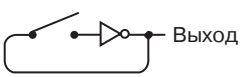


и, конечно, снова замыкает цепь. В целом работу этой схемы можно описать так: пока переключатель замкнут, металлическая полоска будет метаться между двумя контактами, то замыкая, то размыкая цепь, причем ее движения, вероятно, будут сопровождаться шумом. Такое шумящее реле называется *зуммером* (buzzer). Прикрепите к полоске металла молоточек, поставьте рядом металлическую чашку — вот вам и электрический звонок.

Есть и другой способ собрать зуммер с помощью реле. На схеме он показан с использованием общепринятых обозначений для источника питания и земли.



Вы, вероятно, узнали в этой схеме инвертор из главы 11. А значит, ее можно перерисовать так:



Как вы помните, выход инвертора равен 1, если вход равен 0, и наоборот. Замыкание переключателя в этой цепи вызовет попеременное включение и выключение реле. Чтобы схема работала вечно, можно обойтись и без переключателя:

В этом рисунке, кажется, есть логическое противоречие: выход инвертора, как мы знаем, электрически противоположен его входу, но здесь вход и выход — одно и то же! Но не забывайте, что инвертор — это только реле, и ему требуется

время, чтобы перейти из одного состояния в другое. Поэтому, даже если в данный момент вход равен выходу, через мгновение реле сработает, и выход станет противоположен входу (при этом, конечно, изменится и входной сигнал, в свою очередь изменив выход и т. д.).

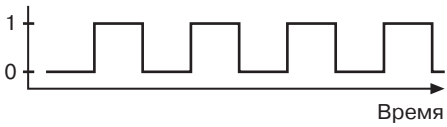
И чему же равен выход такой цепи? А его значение будет периодически меняться: есть напряжение, нет напряжения. Или, иначе говоря, *выход попеременно равен то 0, то 1.*

Такая цепь называется *вибратором* (oscillator). Это устройство существенно отличается от всего, с чем мы имели дело до сих пор. Все предыдущие цепи изменяли состояние после вмешательства человека, который замыкал или размыкал переключатель. Вибратор же, по существу, работает сам по себе.

Конечно, в вибраторе, который работает сам по себе, мало проку. Однако и в этой главе, и в некоторых следующих мы убедимся, что это устройство, включенное в электрическую схему, — важная часть автоматизации. Так, в любом компьютере имеется вибратор определенного типа, обеспечивающий синхронную работу всех компонентов компьютера.

Выход вибратора осциллирует между 0 и 1. Обычно такое изменение сигнала изображают так:

Это можно считать графиком, у которого по горизонтальной оси отложено время, а по вертикальной — выходное напряжение (0 или 1).

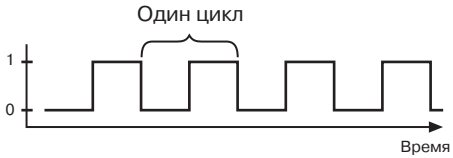


Смысл этого графика в том, что с течением времени сигнал на выходе вибратора периодически переключается с 0 на 1 и обратно. Поэтому вибратор иногда сравнивают с часами: подсчитав число колебаний, вы (после некоторых вычислений) сможете сказать, сколько прошло времени.

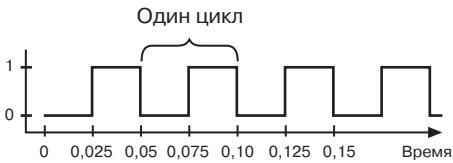
Как быстро переключается вибратор? Точнее, сколько времени уходит на одно колебание металлического контакта?

Сколько раз в секунду это происходит? Это, очевидно, зависит от устройства реле. Легко представить себе большое неповоротливое реле, которое с глухим стуком медленно замыкает и размыкает контакт, и небольшую легкую релюшку, которая тоненько жужжит.

Циклом колебаний называется интервал, в течение которого выход вибратора изменяется, а затем возвращается в исходное состояние.



Время, которое занимает один цикл, называется *периодом* колебаний. Предположим, что период нашего вибратора равен 0,05 сек, и проградуируем горизонтальную ось в секундах, начиная с некоторого момента времени, который будем считать нулевым.



Частота колебаний есть величина, обратная периоду. Например, если период колебаний равен 0,05 сек, частота колебаний 1/0,05 = 20 *колебаний в секунду*, т. е. выход вибратора меняет свое состояние и возвращается обратно 20 раз в секунду.

Число *колебаний в секунду* — единица столь же понятная, как число *километров в час*, *килограммов на квадратный сантиметр*, *килокалорий на 100 г продукта*. Но в отличие от приведенных примеров у нее есть собственное название. В признание заслуг Генриха Рудольфа Герца (1857–1894), который первым отправил и принял радиоволны, частоту измеряют не в *колебаниях в секунду*, а в *герцах (Гц)*. В 20-х годах XX века эту единицу измерения начали использовать в Германии, и за несколько лет она распространилась на другие страны.

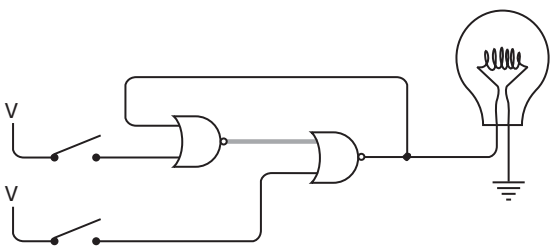
Таким образом, мы можем сказать, что наш вибратор имеет частоту 20 герц, или сокращенно 20 Гц.

Конечно, эту частоту мы просто придумали. Однако к концу этой главы мы создадим устройство, с помощью которого частоту колебаний можно будет измерять.

Чтобы привести это начинание в жизнь, рассмотрим пару вентилей ИЛИ-НЕ, соединенных особым образом. Мы помним, что напряжение на выходе вентиля ИЛИ-НЕ имеется, только если напряжения нет ни на одном из входов.

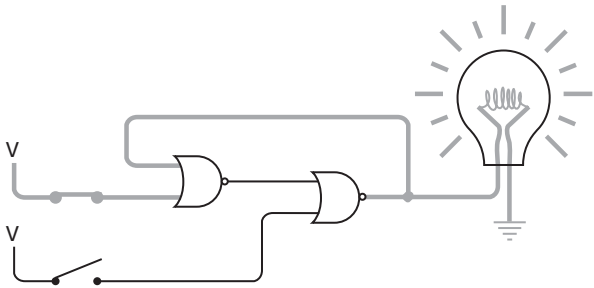
ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

Вот как выглядит нужная нам цепь с двумя вентилями ИЛИ-НЕ, двумя переключателями и лампочкой.

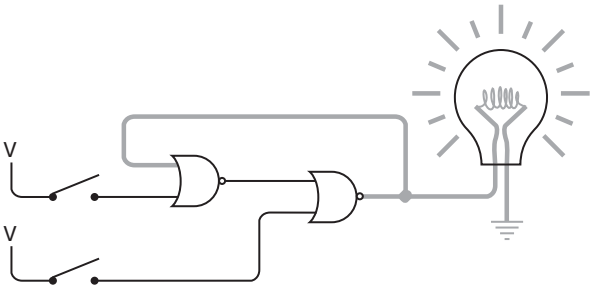


Провода идут весьма странно: выход левого вентиля ИЛИ-НЕ подключен ко входу правого вентиля ИЛИ-НЕ, а выход правого вентиля ИЛИ-НЕ является входом левого вентиля ИЛИ-НЕ. Такое соединение называется *обратной связью* (feedback). Как и в вибраторе, выход цепи является ее же входом. Это особенность подавляющего большинства схем, рассматриваемых в этой главе.

Поначалу ток протекает только по цепи от выхода левого вентиля ИЛИ-НЕ. Это происходит из-за того, что оба входа этого вентиля равны 0. Замкнем верхний переключатель. Выход левого вентиля ИЛИ-НЕ становится 0, что означает появление 1 на выходе правого вентиля ИЛИ-НЕ — лампочка загорается:

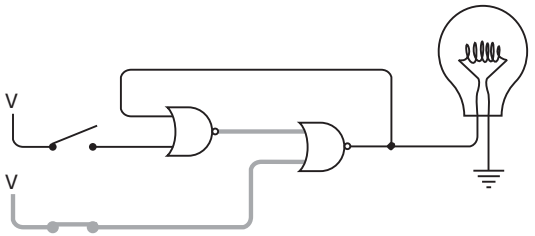


Чудеса начнутся, когда вы снова разомкнете верхний переключатель. Выход вентиля ИЛИ-НЕ равен 0 независимо от того, на каком входе он получает 1, поэтому выход левого вентиля ИЛИ-НЕ остается неизменным, и свет продолжает гореть.

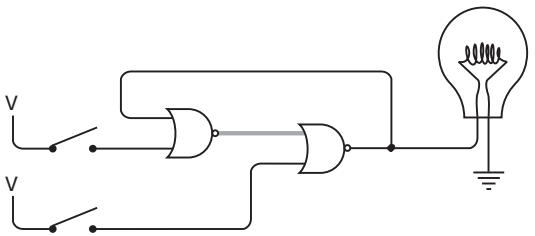


Не странно ли? Оба переключателя разомкнуты, как и на первом рисунке, но теперь свет горит. Ни с чем подобным мы еще не сталкивались. Обычно выход цепи зависел исключительно от ее входа, но в данном случае это не так. Более того, теперь вы вольны сколько угодно замыкать и размыкать верхний переключатель — свет все равно будет гореть. Этот переключатель больше на цепь не влияет, поскольку выход левого вентиля ИЛИ-НЕ независимо от него остается 0.

Замкнем теперь нижний переключатель. На один из входов правого вентиля ИЛИ-НЕ теперь подается 1, его выход становится 0, и лампочка гаснет. Выход левого вентиля ИЛИ-НЕ становится равным 1.



Теперь нижний переключатель можно разомкнуть — лампочка останется выключенной.



Мы вернулись к тому, с чего начали. На этот раз без каких бы то ни было последствий можно замыкать и размыкать нижний переключатель. Подведем итог наших экспериментов.

- Замыкание верхнего переключателя приводит к включению лампочки. Она продолжает гореть, даже если верхний переключатель после этого разомкнуть.
- Замыкание нижнего переключателя приводит к выключению лампочки. Она не загорается, даже если нижний переключатель после этого разомкнуть.

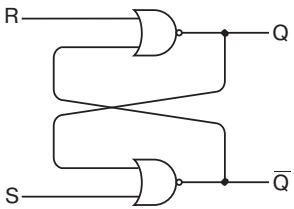
Странность этой схемы в том, что когда оба переключателя разомкнуты, в одном случае лампочка горит, а в другом — нет. Можно сказать, что при разомкнутых переключателях у этой схемы есть два *устойчивых состояния*. Такая схема называется *триггером* (flip-flop). История триггера началась в 1918 г. с работ английского физика Вильяма Генри Эклса (William Henry Eccles) (1875–1966) и Ф. В. Джордана (F. W. Jordan), о котором, кажется, больше ничего не известно.

Триггер *сохраняет информацию*. Он «запоминает». В частности, в последней схеме триггер помнит, какой переключатель был замкнут последним. Если в своих изысканиях вы натолкнетесь на такой триггер и увидите, что лампочка горит,

вы сделаете вывод, что последним был замкнут верхний переключатель, а если не горит — нижний.

Триггер — весьма полезное устройство. Он снабжает электрическую схему памятью, в которой хранится информация о том, что в этой схеме происходило раньше. Представьте себе, что пытаетесь считать, не обладая способностью к запоминанию. Ничего бы у вас не получилось: ведь чтобы назвать очередное число, нужно помнить, какое число было названо перед этим! Поэтому схема, предназначенная для подсчета чего-либо, обязательно содержит триггеры.

Существует несколько разновидностей триггеров. Простейший триггер — я его только что показал — называется RS-триггером (Reset-Set, Сброс-Установка). Вентили ИЛИ-НЕ на схеме RS-триггера обычно изображаются симметрично:



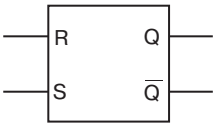
Выход, к которому мы подключали лампочку, традиционно обозначается Q. Его дополняет выход $\overline{Q}$, электрически противоположный Q. Если Q равен 0, то $\overline{Q}$ равен 1 и наоборот. Входы S и R используются для *установки* (set) и *сброса* (reset). Когда на вход S подается сигнал 1 (это соответствует замыканию верхнего переключателя на предыдущей схеме), выход Q равен 1, а $\overline{Q}$ — 0. Когда на входе R — 1 (что соответствует замыканию нижнего переключателя), Q становится 0, а $\overline{Q}$ — 1. Когда на оба входа подается 0, значение на выходе Q зависит от предыдущего действия. Эти правила обобщены в таблице.

Входы		**Выходы**	
S	**R**	**Q**	$\overline{\mathbf{Q}}$
1	0	1	0
0	1	0	1
0	0	Q	$\overline{Q}$
1	1	Запрещено	

Эта таблица называется *функциональной таблицей,* или *таблицей логики,* или *таблицей истинности,* или *таблицей состояний.* В ней показана зависимость результата на выходах от комбинаций сигналов на входах. У RS-триггера всего два входа, поэтому число возможных комбинаций на входах равно 4. Они соответствуют четырем строкам таблицы.

Обратите внимание на вторую строку снизу: в ней значения входов S и R равны 0. Значения выходов в ней обозначены Q и $\overline{Q}$, т. е. выходы Q и $\overline{Q}$ остаются такими же, какими они были до того, как входы S и R обратились в 0. Последняя строка указывает, что ситуация, в которой входы S и R имеют значение 1, *запрещена.* Это не значит, что вас оштрафуют, если вы подадите 1 на оба входа. Просто если в этой схеме на оба входа попадает 1, оба выхода будут иметь значения 0, а это противоречит нашему желанию, чтобы выход $\overline{Q}$ был противоположен выходу Q. Проектируя схему с RS-триггером, избегайте ситуаций, при которых входы R и S одновременно равны 1.

RS-триггер часто изображают как небольшой прямоугольник с двумя входами и двумя выходами:



RS-триггер, безусловно, интересен как первый пример схемы, которая как бы «запоминает», который из двух входов был в последний раз под напряжением. Однако полезнее схема, способная запоминать, был ли данный сигнал нулем или единицей *в определенный момент времени.*

Прежде чем начинать ее сборку, подумаем о том, как такая схема должна себя вести. У нее будет два входа. Назовем один из них *Данные.* Как обычно, сигнал на входе Данные может быть 0 или 1. Второй вход назовем «*Запомнить этот бит*». Обычно на этот вход подается 0, и сигнал Данные на состояние схемы не влияет. Когда сигнал «Запомнить этот бит» равен 1, сигнал Данные копируется на выходе схемы. После этого сигнал «Запомнить этот бит» может вернуться в состояние 0. В этот момент схема запоминает последнее значение сигнала Данные; последующие изменения сигнала Данные на состояние схемы не влияют.

Словом, нам нужна схема с такой таблицей логики.

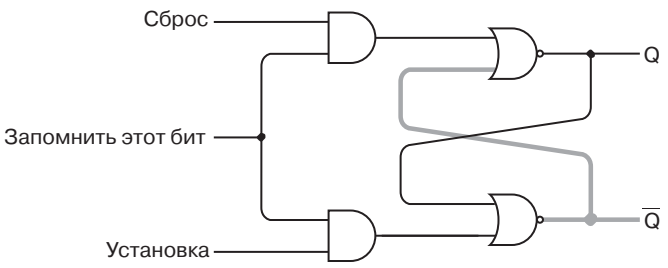
Входы		**Выходы**
Данные	**«Запомнить этот бит»**	**Q**
0	1	0
1	1	1
0	0	Q
1	0	Q

В первых двух случаях, когда сигнал «Запомнить этот бит» равен 1, выход Q имеет то же значение, что и вход Данные. Когда сигнал «Запомнить этот бит» обращается в 0, выход Q сохраняет значение, которое было на нем до этого. Заметьте: при нулевом сигнале «Запомнить этот бит», значение выхода Q не зависит от изменений в сигнале Данные. Таблицу логики можно упростить так:

Входы		**Выходы**
Данные	**«Запомнить этот бит»**	**Q**
0	1	0
1	1	1
X	0	Q

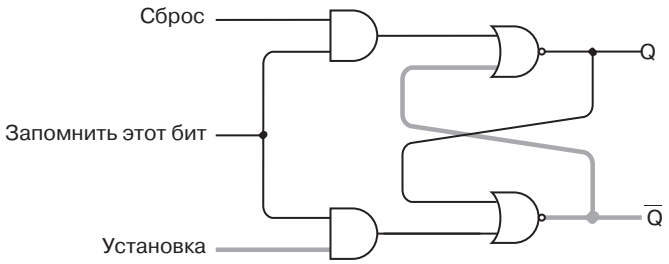
Символ «X» означает «не важно», т. е. при нулевом сигнале на входе «Запомнить этот бит» значение входа Данные не имеет значения — выход Q останется неизменным.

Чтобы реализовать сигнал «Запомнить этот бит» в нашем RS-триггере, придется добавить на входе два вентиля И, как показано на схеме:

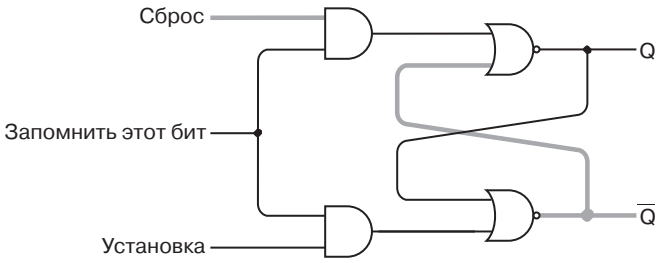


Как вы помните, выход вентиля И равен 1, если сигнал на обоих входах равен 1. В этой схеме выход Q равен 0, а выход $\overline{Q}$ — 1.

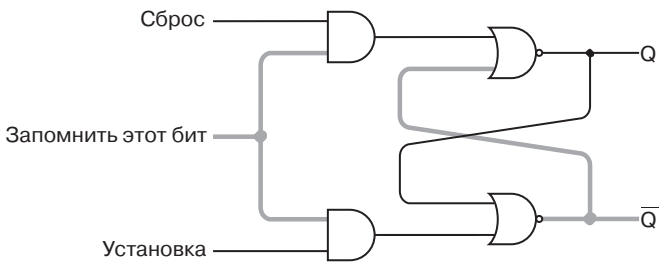
Если сигнал «Запомнить этот бит» равен 0, значение сигнала Установка не влияет на выходные сигналы:



Не влияет на них и сигнал Сброс:



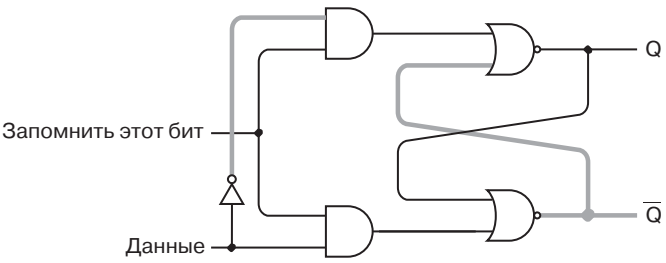
Чтобы схема работала подобно обычному RS-триггеру, сигнал на входе «Запомнить этот бит» должен равняться 1:



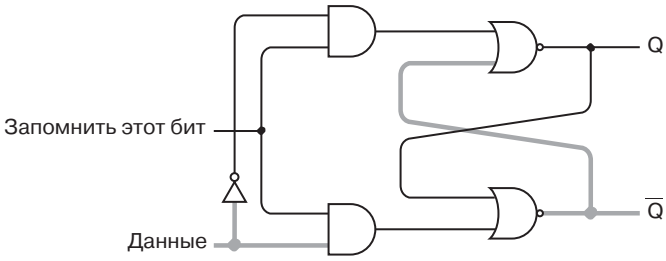
В этом случае выход верхнего вентиля И совпадает с сигналом Сброс, а выход нижнего вентиля И — с сигналом Установка.

Но мы пока не достигли поставленной цели. Нам нужно два выхода, а не три. Как сократить их число? Вспомним исходную таблицу логики RS-триггера: одновременное равенство 1 сигналов Сброс и Установка не допускается, и подобных ситуаций следует избегать. Кроме того, не имеет смысла и их одновременное равенство 0: эта ситуация означает неизменность выходного сигнала, что в новой схеме соответствует установке в 0 сигнала «Запомнить этот бит».

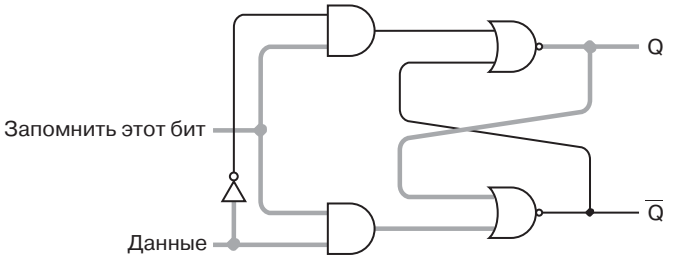
Схему следует организовать так, чтобы при единичном сигнале Установка сигнал Сброс равнялся 0, а при сигнале Установка равном 0 сигнал Сброс устанавливался бы в 1. Сигнал Данные может быть эквивалентен сигналу Установка, а инвертированный сигнал Данные — сигналу Сброс.



В такой ситуации оба входа равны 0, и выход Q тоже равен 0 (т. е. выход $\overline{Q}$ равен 1). Пока сигнал «Запомнить этот бит» равен 0, сигнал Данные на состояние схемы не влияет.



Когда сигнал «Запомнить этот бит» обращается в 1, сигнал на выходе схемы равен сигналу на входе Данные.



Выход Q теперь имеет то же значение, что и вход Данные, а выход $\overline{Q}$ противоположен ему. Теперь входу «Запомнить этот бит» можно вернуть значение 0.

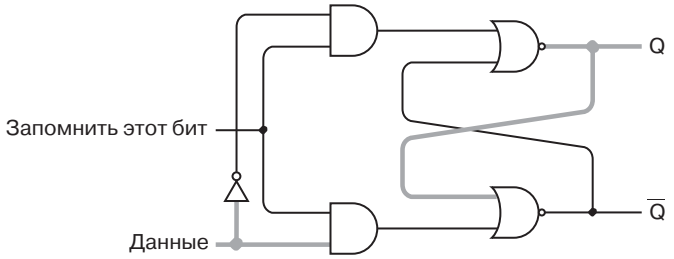
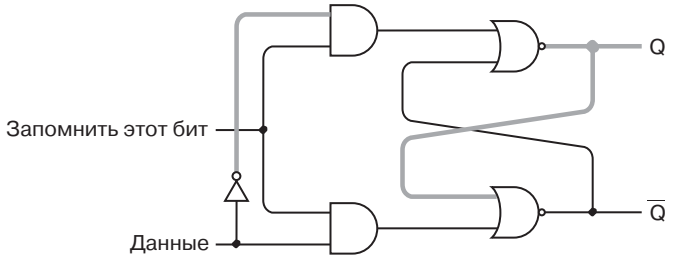


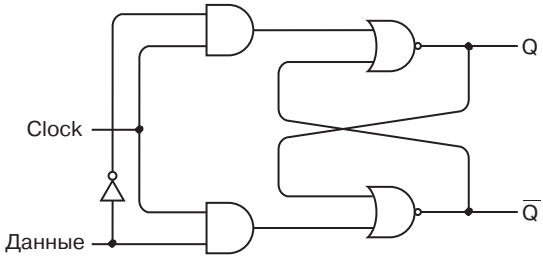
Схема запомнила величину сигнала Данные, имевшую место в тот момент, когда сигнал «Запомнить этот бит» равнялся 1. Последующие изменения сигнала на нее не влияют.



Эта схема называется *D-триггером со срабатыванием по уровню* (level-triggered D-type flip-flop). Буква D происходит от английского слова «data» (данные). Срабатывание по уровню означает, что триггер сохраняет значение сигнала на входе Данные, когда сигнал «Запомнить этот бит» достигает определен-

ного *уровня*, в нашем случае 1 (вскоре мы рассмотрим другой способ срабатывания триггера).

Обычно в книгах по схемотехнике вместо «Запомнить этот бит» применяется обозначение *Синхронизация* (Clock) — оно говорит о том, что иногда этот сигнал используется в качестве метронома, который периодически колеблется между 0 и 1. Но пока сигнал на входе Clock просто говорит, что значение на входе Данные нужно сохранить.



В таблицах логики и на схемах часто название входа Данные (data) сокращается до D, а вход Clock записывают как Clk.

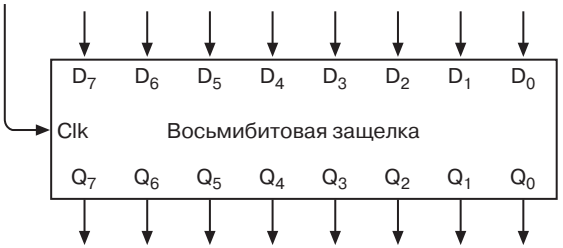
Входы		**Выходы**	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

Иногда разобранную схему называют также *защелкой* (latch) D-типа со срабатыванием по уровню. Термин «защелка» означает, что схема как бы «запирает» один бит информации и хранит его для дальнейшего использования. При желании такую схему можно считать *памятью емкостью в 1 бит*. В главе 16 я покажу, как из множества подобных триггеров можно собрать память гораздо большего объема.

Необходимость сохранения многобитовых величин возникает довольно часто. Допустим, вы хотите использовать сумматор из главы 12 для сложения трех 8-битовых чисел. Первое число вы, как обычно, набираете с помощью первого ряда тумблеров, второе число — с помощью второго ряда, но при этом вам

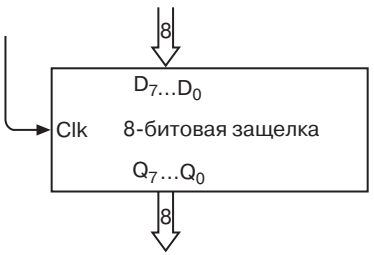
придется записать результат их сложения на бумажке. Затем вы набираете его с помощью первого ряда тумблеров, а второй ряд используете для ввода третьего слагаемого. Можно ли избавиться от необходимости записывать промежуточный результат, как-то «запомнив» его внутри сумматора?

Мы решим эту проблему с помощью 8 защелок, в каждой из которых используются 2 вентиля ИЛИ-НЕ, 2 вентиля И и 1 инвертор. Объединим их в одно устройство, соединив между собой все входы Clk:

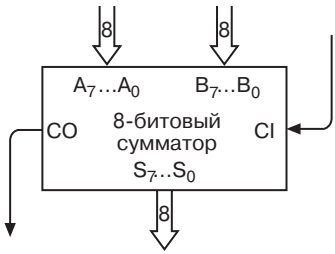


Эта защелка может одновременно хранить 8 битов. Восемь входов обозначены D_0–D_7, восемь выходов — Q_0–Q_7. С левой стороны расположен вход Clk. Обычно сигнал на нем равен 0. Когда же он обращается в 1, 8-битовое значение на входах D передается на выходы Q. Это 8-битовое число остается на выходе и после того, как сигнал Clk снова становится нулевым и сохраняется там до следующего перехода сигнала Clk в 1.

В обозначении 8-битовой защелки 8 входов D и 8 выходов Q можно сгруппировать:

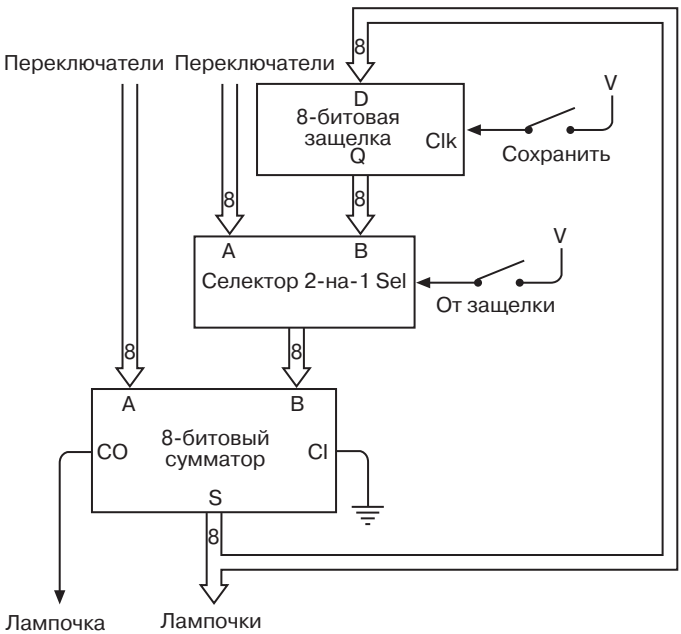


А вот и наш старый знакомый — восьмибитовый сумматор.



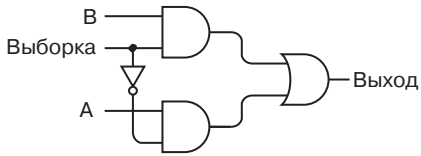
Обычно (забудем пока про вычитание) входы A и B соединяются с переключателями, вход для переноса CI — с землей, а выходы S и CO — с лампочками.

В модифицированном варианте сумматора выходы S можно подключить не только к лампочкам, но и ко входам D 8-битовой защелки. Для сохранения суммы ко входу Clk защелки можно подключить тумблер «Сохранить».



Селектор 2 линии на 1 (2-Line-to-1-Line Selector) позволяет с помощью переключателя выбирать, откуда следует подавать

информацию на вход B — со второго ряда переключателей или с выходов Q защелки. Чтобы выбрать выходы 8-битовой защелки, переключатель нужно замкнуть. В селекторе «2 на 1» используется 8 одинаковых схем:



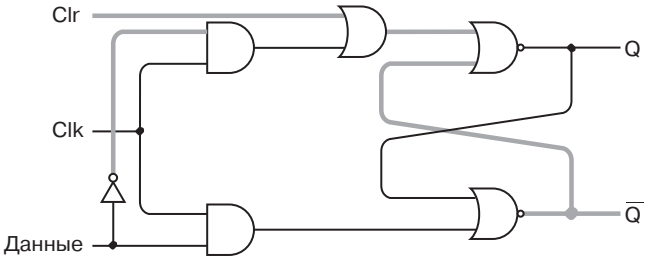
Если вход Выборка (Sel) равен 1, выход вентиля ИЛИ совпадает со входом B. Дело в том, что выход верхнего вентиля И равен входу B, а выход нижнего вентиля И — 0. Если вход Выборка равен 0, выход совпадает со входом A. Правила работы этой схемы обобщены в такой таблице логики.

Входы			**Выходы**
Выборка	**A**	**B**	**Q**
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

В селекторе, включенном в модифицированный сумматор, содержится 8 таких 1-битовых селекторов. Их входы Выборка соединены между собой.

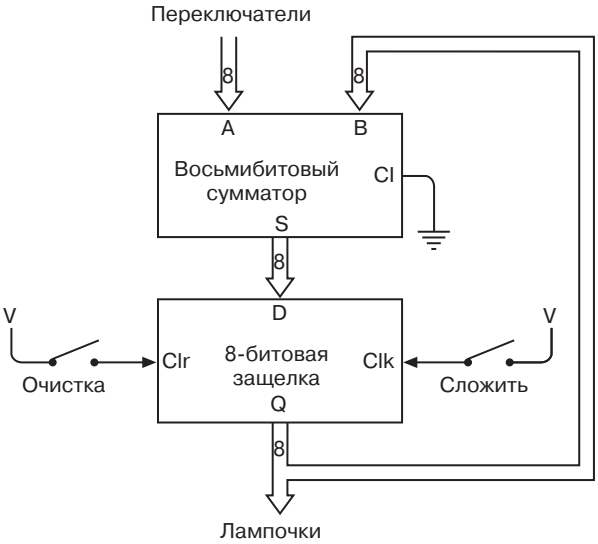
Новый сумматор не совсем корректно обрабатывает сигнал CO (выход для переноса). Даже если сложение двух чисел приводит к появлению переноса (сигнал CO равен 1), при сложении суммы с третьим числом этот сигнал игнорируется. Решить эту проблему можно, создав сумматор, защелку и селектор, разрядность которых равна 16 или хотя бы превышает разрядность максимального числа, с которым вам предстоит иметь дело. Но мы отложим решение этой проблемы до главы 17.

Пока же займемся делом поинтереснее — избавимся от одного ряда переключателей. Но для начала немного модифицируем D-триггер, добавив к нему вентиль ИЛИ и входной сигнал Очистка (Clear или Clr), обычно равный 0. Но когда он обращается в 1, сигнал на выходе Q становится равным 0:



Обращение сигнала Q в 0 происходит независимо от сигналов на других входах: информация, записанная в триггере, как бы стирается.

Зачем? Почему нельзя очистить триггер, подав 0 на вход Данные и 1 на вход Clk? Дело в том, что состояние входа Данные не всегда поддается управлению. Допустим, у нас имеются 8 защелок, соединенных с выходами 8-битового сумматора:



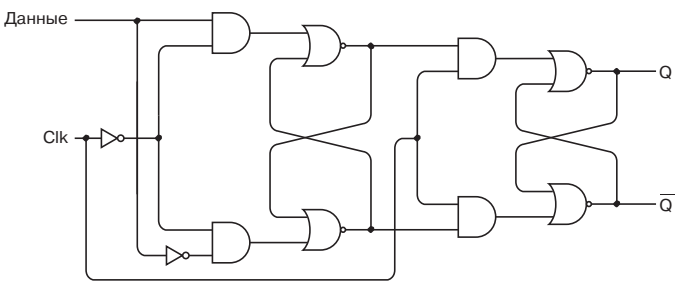
Заметьте: сигнал на вход Clk защелки теперь подается с помощью переключателя Сложить (вместо него и переключателя Очистка лучше использовать кнопки).

Не исключено, что этот сумматор покажется вам удобнее предыдущего, особенно если вам нужно сложить несколько чисел. В начале работы вы нажимаете кнопку Очистка. В результате этого действия все выходы защелки обращаются в 0, все лампочки выключаются, на второй набор входов 8-битового сумматора подаются нули. Затем вы вводите первое число и нажимаете кнопку Сложить. Лампочки отображают введенное число. Вы вводите второе число и снова нажимаете кнопку Сложить. Число, набранное с помощью переключателей, складывается с предыдущим числом, и сумма отображается с помощью лампочек. Работу можно продолжить, набирая новые числа и нажимая кнопку Сложить.

Я уже говорил, что спроектированный нами D-триггер *срабатывает по уровню*. Это значит, что для сохранения в защелке сигнала на входе Данные *уровень* сигнала на входе Clk должен измениться с 0 до 1. Если в течение того времени, пока сигнал Clk равен 1, сигнал Данные изменится, все его изменения будут отражаться на величине выходов Q и $\overline{Q}$.

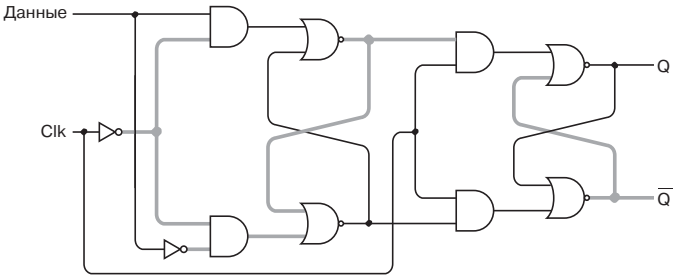
Часто это именно то, что нужно. Но иногда предпочтительнее вход Clk *со срабатыванием по фронту* (edge-triggered), при котором выход может меняться, *только когда сигнал Clk переходит из 0 в 1*. Как и в триггере со срабатыванием по уровню, при нулевом сигнале на входе Clk триггера со срабатыванием по фронту изменения на входе Данные не отражаются на выходах. Отличие в том, что в этом триггере изменения на входе Данные не отражаются на выходах и при сигнале Clk, равном 1. Вход Данные влияет на выходы только в момент перехода сигнала Clk из 0 в 1.

D-триггер со срабатыванием по фронту, собирается из двух блоков RS-триггера, соединенных так:

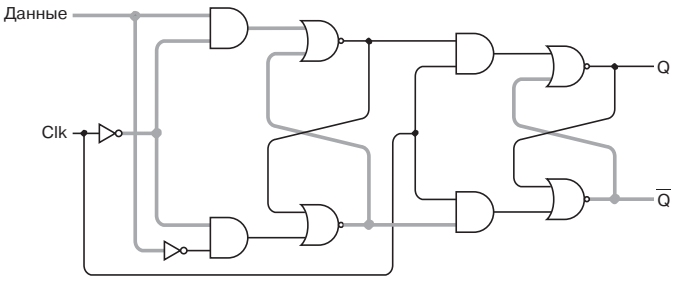


Суть этой схемы в том, что вход Clk управляет как первым блоком, так и вторым, но в первом блоке сигнал Clk инвертируется. Это значит, что первый блок работает как D-триггер за исключением того, что содержимое входа Данные сохраняется, когда сигнал Clk равен 0. Выходы первого блока являются входами для второго блока, и их содержимое сохраняется при обращении сигнала Clk в 1. Общий результат таков: сигнал на входе Данные сохраняется только при переходе сигнала Clk из 0 в 1.

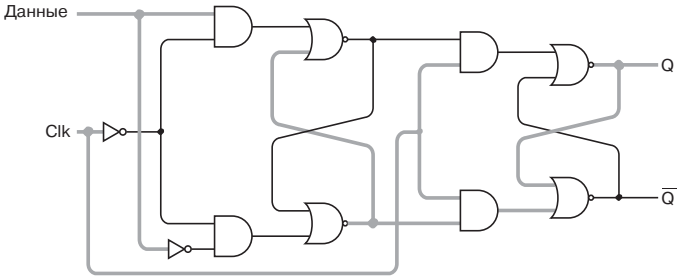
Рассмотрим работу триггера подробнее. На схеме он показан в состоянии покоя с нулями на входах Данные и Clk и нулем же на выходе Q.



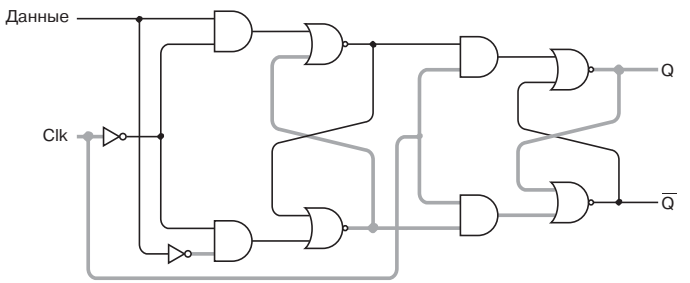
Подадим 1 на вход Данные.



При этом изменяется состояние первого блока триггера, так как инвертированный сигнал Clk равен 1. Второй блок на изменение входа Данные не реагирует, так как неинвертированный сигнал Clk равен 0. Подаем 1 на вход Clk:



Срабатывает второй блок, и выход Q становится равным 1. Теперь сигнал на входе Данные может меняться (например, снова стать нулевым), но на выход Q это не повлияет.

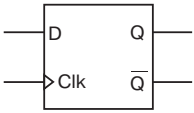


Выходы Q и $\overline{Q}$ меняются только при изменении сигнала Clk с 0 на 1.

В таблице логики D-триггера, срабатывающего по фронту, нам потребуется новый символ — стрелка, направленная вверх. Она означает изменение сигнала с 0 на 1.

Входы		**Выходы**	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
0	↑	0	1
1	↑	1	0
X	0	Q	$\overline{Q}$

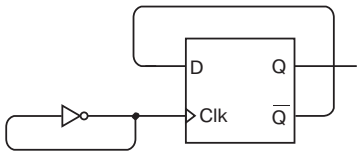
Стрелка указывает, что выход Q копирует вход Данные, когда сигнал Clk переходит с 0 на 1, т. е. совершает *положительный переход* (отрицательным называется переход с 1 на 0). На схемах этот триггер обозначается так:



Треугольная скобка указывает, что триггер срабатывает по фронту.

А теперь я покажу вам схему с D-триггером, срабатывающим по фронту, которую нельзя продублировать с помощью триггера, срабатывающего по уровню. Вспомните вибратор, о котором мы говорили в начале главы. Выход вибратора осциллирует между 0 и 1:

Соединим выход вибратора со входом Clk D-триггера, срабатывающего по фронту, а выход $\overline{Q}$ соединим с входом D:



Здесь выход триггера является его же входом — обратная связь с обратной связью! (Практическое воплощение этой схемы не лишено трудностей. Основу конструкции вибратора составляет реле, которое осциллирует с некоторой скоростью. Выход вибратора соединяется с другими реле, из которых собран триггер. Может статься, что эти реле обладают другой конструкцией и неспособны срабатывать с той же скоростью, что и вибратор. Чтобы не столкнуться с этой проблемой, будем считать, что реле в вибраторе работает гораздо медленнее остальных реле в схеме.)

Чтобы разобраться, что происходит в этой схеме, рассмотрим таблицу ее логики. Для начала примем, что вход Clk и выход Q равны 0, т. е. выход $\overline{Q}$, соединенный со входом D, равен 1.

Входы		**Выходы**	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1

Когда вход Clk меняется с 0 на 1, на выходе Q копируется значение со входа D.

Входы		Выходы	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1
1	↑	1	0

Значение на выходе $\overline{Q}$ обратилось в 0, такой же сигнал пошел и на вход D. Сигнал Clk обратился в 1.

Входы		Выходы	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1
1	↑	1	0
0	1	1	0

Значение сигнала Clk, поступающего из вибратора, снова обнулилось, но на выходе это не сказалось.

Входы		Выходы	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0

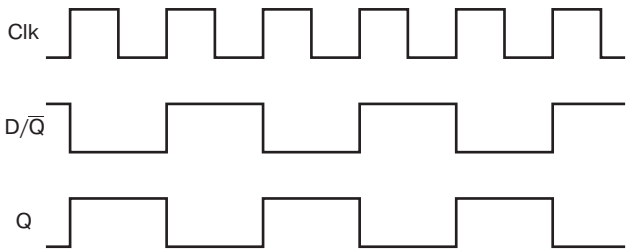
Затем сигнал Clk снова обратился в 1. Выход D равен 0, и потому выход Q обращается в 0, а выход $\overline{Q}$ — соответственно в 1.

Входы		Выходы	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

Сигнал на входе D тоже обращается в 1.

Входы		**Выходы**	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1

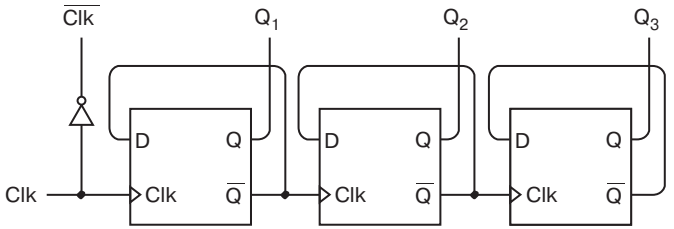
Изменения, которые мы проследили, можно подытожить так: каждый раз, когда вход Clk меняется с 0 на 1, выход Q меняется или с 0 на 1, или с 1 на 0. Ситуацию проясняет график:



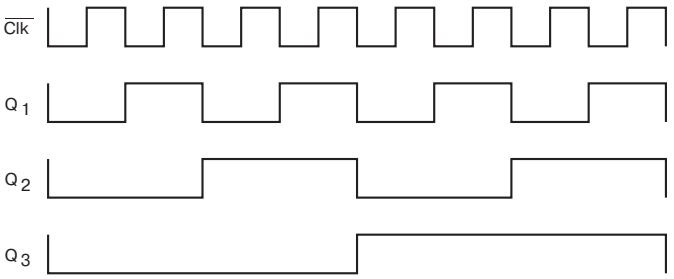
Когда вход Clk переходит из 0 в 1, значение на входе D (совпадающее со значением на выходе $\overline{Q}$) передается на выход Q, одновременно изменяя $\overline{Q}$, а значит, и D до следующего перехода входа Clk из 0 в 1.

Если частота вибратора — 20 Гц (т. е. 20 колебаний в секунду), частота изменений выхода Q вдвое меньше, т. е. 10 Гц. По этой причине схема, в которой выход $\overline{Q}$ соединен со входом Данные триггера, называется *делителем частоты* (frequency divider).

Конечно, выход делителя частоты может быть входом Clk другого делителя для следующего деления частоты. Вот схема из трех делителей частоты:

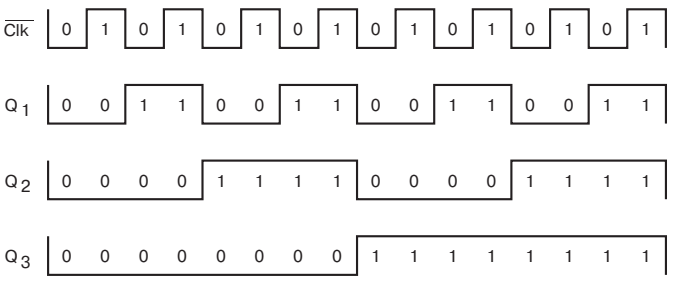


Посмотрим как изменяются четыре сигнала, которые я отметил в верхней части схемы.



Ничего не напоминает?

Я вам помогу. Пометим сигналы нулями и единицами.



Все еще не догадались? Тогда разверните эту диаграмму по часовой стрелке на 90° и прочитайте 4-битовые числа по горизонтали. Они соответствуют десятичным числам от 0 до 15.

Двоичное число	**Десятичное число**
0000	1
0001	2

(*продолжение*)

0010	3
0011	4
0100	5
0101	6
0110	7
0111	8
1000	9
1001	10
1010	11
1011	12
1100	13
1101	14
1111	15

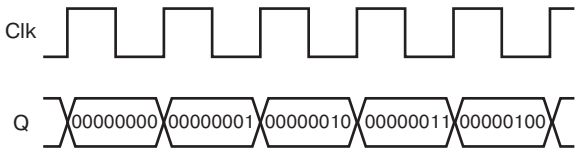
Таким образом, схема *считает в двоичном формате* от 0 до максимального числа, зависящего от количества триггеров в схеме. В главе 8 я уже говорил, что в последовательности увеличивающихся двоичных чисел каждая колонка двоичных цифр меняется между 0 и 1 с частотой, вдвое меньшей, чем колонка справа от нее. Данный счетчик имитирует эту последовательность. При каждом положительном переходе сигнала Clk происходит *приращение* (increment) выходов счетчика на 1.

Соединим 9 триггеров и поместим их в общий корпус.

Эта схема называется *сквозным счетчиком* (ripple counter), так как в ней выход каждого триггера становится входом Clk следующего триггера. Изменения сигнала последовательно проходят через все триггеры, и триггеры в конце схемы могут срабатывать с небольшой задержкой. Более надежный счетчик — *синхронный* (synchronous), в котором все выходы меняются одновременно.

Я обозначил выходы от Q_0 до Q_7. Они собраны так, что выход первого триггера в цепочке (Q_0) расположен справа. Подключив к этим выводам лампочки, вы сможете прочитать 8-битовое число.

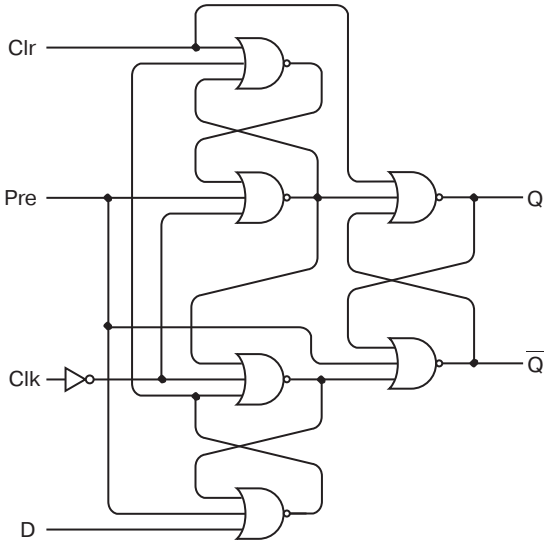
На временной диаграмме выходы такого счетчика можно отображать как раздельно, так и все вместе:



При каждом положительном переходе сигнала Clk некоторые выходы меняются, некоторые — нет, но все вместе они отображают увеличивающиеся двоичные числа.

Я говорил в начале главы, что мы определим частоту колебаний вибратора. Пришла пора это сделать. Если соединить вибратор с входом Clk 8-битового счетчика, счетчик покажет, сколько колебаний совершил вибратор. Когда результат достигнет 11111111 (255 в десятичной системе), счетчик сбросится в 00000000. Вероятно, легче всего определить частоту вибратора так. Соедините с выходами 8-битового счетчика лампочки. Дождитесь, чтобы сигнал на всех выходах стал 0 (когда это произойдет, все лампочки погаснут), и запустите секундомер. Остановите его, когда все лампочки снова погаснут. Вы получите время, которое потребовалось для 256 колебаний вибратора. Допустим, это 10 секунд. Частота колебаний, таким образом, равна 256 ÷ 10, или 25,6 Гц.

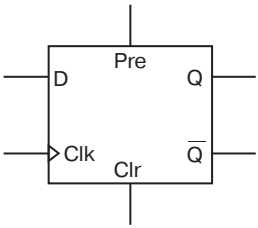
По мере того, как у триггеров появляются новые возможности, возрастает и их сложность. Взгляните на *D-триггер с предустановкой и очисткой, срабатывающий по фронту* (edge-triggered D-type flip-flop with preset and clear):



Входы Pre и Clr обладают более высоким приоритетом, чем входы Clk и Данные. Обычно оба они равны 0. Когда вход Pre обращается в 1, выход Q также обращается в 1, а $\overline{Q}$ — в 0. Когда вход Clr равен 1, выход Q равен 0, а $\overline{Q}$ — 1. Подобно входам Установка и Сброс RS-триггера, входы Pre и Clr не должны одновременно обращаться в 1. В остальном этот триггер ведет себя как обычный D-триггер со срабатыванием по фронту.

Входы				**Выходы**	
Pre	**Clr**	**D**	**Clk**	**Q**	$\overline{\mathbf{Q}}$
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	$\overline{Q}$

D-триггер, срабатывающий по фронту, с предустановкой и сбросом обозначается так:



Итак, мы научили реле складывать, вычитать и считать в двоичной системе. Это немалое достижение, особенно учитывая, что все устройства, которые мы использовали, изобретены более ста лет назад. Но на этом наши открытия не заканчиваются. Пока же мы немного оторвемся от сборки электрических схем и еще раз посмотрим на системы счисления.

Глава 15

Байты и шестнадцатеричные числа

Две усовершенствованных версии суммирующей машины из предыдущей главы наглядно иллюстрируют концепцию *потоков данных* (data paths). В пределах схемы от одного компонента к другому перемещаются 8-битовые значения. Они подаются на входы сумматоров, защелок и селекторов, они же выводятся из этих устройств. 8-битовые значения задаются переключателями и отображаются лампочками. Говорят, что поток данных в этой схеме *имеет ширину 8 битов*. Почему именно *8*? Почему не 6, или 7, или 9, или 10?

Можно было бы сказать: потому, что в основу этих усовершенствованных версий положена исходная суммирующая машина из главы 12, а она работает с 8-битовыми значениями. Но никаких особых причин делать ее 8-битовой у нас не было. Просто это число в тот момент показалось особенно удобным. Впрочем, признаюсь, что все это время я немного хитрил, так как прекрасно знал (как, вероятно, и вы), что именно 8 битов составляют один *байт* (byte).

Слово «байт» родилось в фирме IBM году этак в 1956. Произошло оно от английского слова «bite» (кусок), но писалось

через «у», чтобы никто не путал его со словом «bit» (т. е. бит). В течение некоторого времени термином «байт» обозначалось просто число битов в потоке данных. Но в середине 60-х, в связи с созданием в IBM компьютеров System/360, это слово стало обозначать группу из 8 битов.

Будучи 8-разрядным числом, байт может принимать значения от 00000000 до 11111111. Этими кодами можно обозначить положительные числа от 0 до 255 или положительные и отрицательные числа от –127 до 128 (если они записаны с помощью дополнения до 2). 1 байт позволяет зашифровать 2^8, или 256, различных вариантов.

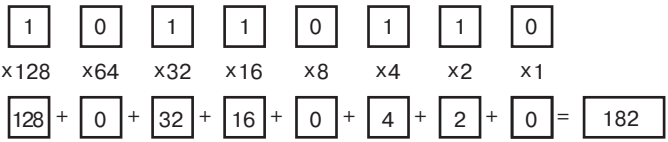
Потом выяснилось, что 8-разрядность байта очень удобна по многим причинам. Для IBM она оказалась привлекательной, так как облегчала хранение данных в формате BCD, к которому я вернусь в главе 23. Но, как мы убедимся в следующих главах, по случайному совпадению именно 8-битовый байт удобен для хранения текста, поскольку практически любой язык мира (кроме идеографического письма, используемого в Китае, Японии и Корее) можно представить в виде набора менее чем 256 символов. Байт также идеален для кодирования оттенков серого на черно-белых фотографиях, поскольку человеческий глаз этих оттенков также различает приблизительно 256. А там, где не хватает 1 байта (как, скажем, в упомянутых идеографических языках), обычно любую проблему можно решить с помощью 2 байтов, которыми кодируются уже 2^{16}, или 65 536 вариантов.

Половину байта, т. е. 4 бита, иногда называют *тетрадой* (nibble или nybble), но используют этот термин гораздо реже термина «байт».

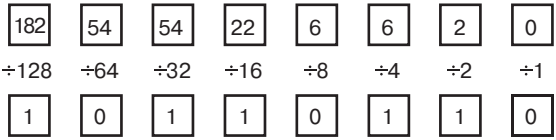
Поскольку в описании внутренней жизни компьютеров байты используются очень часто, их значения желательно записывать максимально кратко. 8-разрядное двоичное число 10110110, например, весьма наглядно, но коротким его не назовешь.

Конечно, к байтам всегда можно обращаться по их десятичным значениям, но для этого их придется постоянно переводить из двоичной в десятичную систему счисления. Вычисление хоть и не особо сложное, но утомительное. В главе 8 я продемонстрировал способ решения этой задачи «в лоб». По-

скольку каждая двоичная цифра соответствует степени двойки, мы можем просто выписать все цифры двоичного числа, указав под каждым из них двойку в соответствующей степени. Затем перемножаем числа в столбцах и складываем результаты. Вот как преобразуется число 10110110:



Для обратного преобразования потребуется более неуклюжая процедура, состоящая в последовательном делении десятичного числа на убывающие степени двойки. При этом частное есть соответствующая двоичная цифра, а остаток делится на следующую по убыванию степень двойки. Вот как преобразовать число 182 обратно в двоичный формат:



В главе 8 этот метод описан очень подробно. Но даже из одного примера ясно, что для преобразования десятичных чисел в двоичные и обратно нужно запастись листом бумаги, карандашом и терпением.

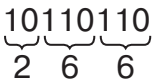
В главе 8 мы познакомились также с восьмеричной системой счисления, в которой используются только цифры 0, 1, 2, 3, 4, 5, 6 и 7. Превращать восьмеричные числа в двоичные и обратно очень просто. Нужно лишь запомнить 3-битовые эквиваленты всех восьмеричных цифр, указанных в следующей таблице.

Двоичное	**Восьмеричное**
000	0
001	1
010	2
011	3

(*продолжение*)

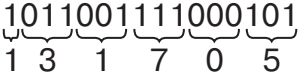
Двоичное	Восьмеричное
100	4
101	5
110	6
111	7

Возьмите нужное двоичное число (например, 10110110) и разбейте на группы по три бита, начиная справа. Каждая группа из трех битов соответствует восьмеричной цифре.



Таким образом, байт 10110110 выражается восьмеричным числом 266. Что ж, число существенно сократилось, а значит восьмеричные числа действительно удобны для записи байтов. К сожалению, с восьмеричной системой связана небольшая проблема.

В двоичном выражении значение байта варьируется от 00000000 до 11111111. В восьмеричной системе этому диапазону соответствуют числа от 000 до 377. Очевидно, что в предыдущем примере, правой и средней восьмеричным цифрам соответствуют полные тройки битов, а на левую цифру осталось только два бита. Это значит, что восьмеричное представление 16-битового числа:



не совпадает с восьмеричным представлением двух составляющих его байтов:

Чтобы представление многобайтового значения согласовывалось с представлением входящих в него байтов, нужна систе-

ма, в которой байт делился бы на группы с равным числом битов. Это значит, что мы должны делить байт либо на 4 группы по 2 бита (четверичная система), либо на 2 группы по 4 бита (шестнадцатеричная система).

С шестнадцатеричной системой мы до сих пор не сталкивались, и хорошо. У нее и название-то — язык сломаешь! — *шестнадцатеричная* (hexadecimal). Но это еще полбеды, есть и другие проблемы. В десятичной системе мы считаем так:

0 1 2 3 4 5 6 7 8 9 10 11 12...

В восьмеричной системе, как вы помните, нам удалось избавиться от цифр 8 и 9:

0 1 2 3 4 5 6 7 10 11 12...

В четверичной системе отпадает нужда в цифрах 4, 5, 6 и 7:

0 1 2 3 10 11 12...

И, наконец, в двоичной можно обойтись только 0 и 1:

0 1 10 11 100...

С шестнадцатеричной системой все не так просто — в ней требуется *больше* цифр, чем в десятичной. Счет в этой шестнадцатеричной системе выглядит примерно так:

0 1 2 3 4 5 6 7 8 9 ? ? ? ? ? ? 10 11 12...

где 10 в действительности означает $16_{\text{десять}}$. Знаки вопроса наглядно демонстрируют, что для представления шестнадцатеричных чисел нам необходимо еще 6 символов. Что это за символы? Откуда их взять? Что ж, ни из какой многовековой традиции они не проистекают, поэтому мы вольны выбрать их по своему усмотрению, например:

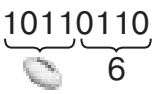
В отличие от символов, которыми обозначаются другие цифры, у этих есть большое преимущество — их легко запомнить и отождествить с тем количеством, что они выражают: ковбойская шляпа (емкость — 10 галлонов), футбольный мяч (в команде 11 игроков), дюжина (т. е. 12) пончиков, черная кошка

(неразрывно связанная с числом 13), полная луна (красуется на небе через 14 дней после новолуния) и, наконец, кинжал, который напомнит нам об убийстве Юлия Цезаря в 15-й день марта.

Для выражения байта нужны две шестнадцатеричных цифры. Иначе говоря, шестнадцатеричная цифра эквивалентна четырем битам, или тетраде. В таблице показано, как преобразовывать двоичные, шестнадцатеричные и десятичные числа.

Двоичное	Шестнадцатеричное	Десятичное	Двоичное	Шестнадцатеричное	Десятичное
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010		10
0011	3	3	1011		11
0100	4	4	1100		12
0101	5	5	1101		13
0110	6	6	1110		14
0111	7	7	1111		15

Вот как выглядит в шестнадцатеричном формате число 10110110:



При работе с многобайтовыми числами все происходит точно так же.

Один байт всегда представляется одной и той же парой шестнадцатеричных цифр.

К сожалению (а может, и к счастью), использовать футбольные мячи и пончики в качестве цифр мы не будем, помня, однако, что они с этой ролью вполне справлялись. Вместо них в шестнадцатеричной системе приняты куда менее понятные обозначения цифр, которые многих навсегда сбивают с толку. Шесть недостающих цифр в действительности представляются первыми шестью буквами латинского алфавита:

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12...

В следующей таблице показано *истинное* соответствие между двоичными, шестнадцатеричными и десятичными числами.

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

Таким образом, байт 10110110 представляется парой шестнадцатеричных чисел B6, а футбольный мяч вам рисовать не придется. В предыдущих главах я указывал вид системы счисления в нижнем индексе, например:

$10110110_{\text{ДВА}}$

— для двоичной системы,

$2312_{\text{ЧЕТЫРЕ}}$

— для четверичной,

$$266_{\text{восемь}}$$

— для восьмеричной,

$$182_{\text{десять}}$$

— для десятичной. Продолжая ту же традицию, записываем

$$B6_{\text{шестнадцать}}$$

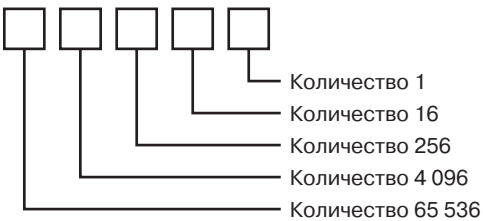
— для шестнадцатеричной системы. Получилось очень громоздко, но, к счастью, на практике используются другие, более экономные способы обозначения шестнадцатеричных чисел. Можно, например, идентифицировать шестнадцатеричное число так:

$$B6_{\text{hex}}$$

В дальнейшем я буду использовать еще более популярный способ обозначения шестнадцатеричных чисел, который состоит в добавлении к числу строчной латинской буквы *h*:

B6h

В шестнадцатеричной системе положение цифры соответствует степени числа 16.



Шестнадцатеричное число 9A48Ch представляется так:

$$\begin{aligned} 9A48Ch = {} & 9 \times 10000h + \\ & A \times 1000h + \\ & 4 \times 100h + \\ & 8 \times 10h + \\ & C \times 1h \end{aligned}$$

Переписываем это выражение с помощью степеней числа 16:

$$\begin{aligned} 9A48Ch = {} & 9 \times 16^4 + \\ & A \times 16^3 + \\ & 4 \times 16^2 + \\ & 8 \times 16^1 + \\ & C \times 16^0 \end{aligned}$$

или их десятичных значений:

$$\begin{aligned} 9A48Ch = {} & 9 \times 65\,536 + \\ & A \times 4\,096 + \\ & 4 \times 256 + \\ & 8 \times 16 + \\ & C \times 1 \end{aligned}$$

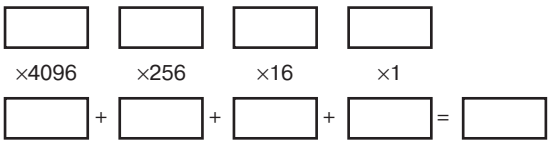
Заметьте, что при записи одиночных цифр (9, A, 4, 8, C) нет нужды указывать в индексе систему счисления. 9 всегда 9 независимо от того, десятичная или шестнадцатеричная система счисления использована. С другой стороны, A очевидно является шестнадцатеричной цифрой, эквивалентной числу 10 в десятичной системе.

Чтобы завершить вычисление, все шестнадцатеричные цифры нужно заменить их десятичными значениями:

$$\begin{aligned} 9A48Ch = {} & 9 \times 65\,536 + \\ & 10 \times 4\,096 + \\ & 4 \times 256 + \\ & 8 \times 16 + \\ & 12 \times 1 \end{aligned}$$

Ответ — 631 948. Так шестнадцатеричные числа преобразуются в десятичные.

Нарисуем шаблон для преобразования произвольного 4-значного шестнадцатеричного числа в десятичный формат.



Подставим в него число 79ACh. Учитывая, что шестнадцатеричные числа A и C равны десятичным 10 и 12, получаем:

7		9		A		C	
×4 096		×256		×16		×1	
28 672	+	2304	+	160	+	12	= 31 148

При преобразовании десятичного числа в шестнадцатеричный формат, не обойтись без деления. Если число меньше либо равно 255, его можно представить одним байтом, т. е. двумя шестнадцатеричными цифрами. Чтобы вычислить их, разделите число на 16, чтобы получить частное и остаток. Рассмотрим уже знакомый пример — десятичное число 182. Разделив его на 16, получаем 11 (шестнадцатеричная цифра B) и 6 в остатке. Итак, шестнадцатеричный эквивалент — B6h.

Если преобразуемое число меньше 65 536, для его представления понадобится не более четырех цифр. Вот как выглядит шаблон для преобразования такого числа в шестнадцатеричный формат:

÷4 096	÷256	÷16	÷1

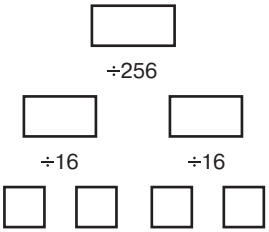
Для начала поместите исходное десятичное число в левый верхний прямоугольник. Это ваше первое делимое. Разделите его на 4 096, первый делитель. Частное записываете в прямоугольник под делимым, а остаток — в прямоугольник справа от делимого. Остаток становится следующим делимым, которое нужно разделить на 256. Вот как преобразовать в шестнадцатеричный формат число 31 148:

31 148	2476	172	12
÷4 096	÷256	÷16	÷1
7	9	10	12

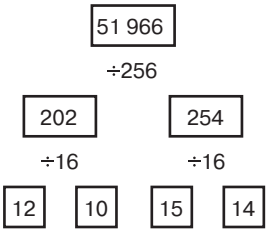
Разумеется, вместо 10 и 12 нужно написать соответствующие шестнадцатеричные цифры и получить в результате 79ACh.

Если вы решите выполнять все вычисления на калькуляторе, то столкнетесь с проблемой: калькулятор не показывает остаток от деления. Разделив 31 148 на 4 096, вы получите 7,6044921875. Чтобы узнать остаток, вам придется умножить 4 096 на 7 (получив 28 672) и вычесть результат из 31 148 или умножить 4 096 на 0,6044921875, т. е. дробную часть результата деления (или просто применить функцию преобразования чисел между десятичным и шестнадцатеричным форматами, которая имеется во многих калькуляторах!).

Еще один способ преобразования числа от 0 до 65 535 в шестнадцатеричный формат состоит в его делении на 256, чтобы сразу разбить число на два байта. Затем каждый байт нужно разделить на 16. Вот как выглядит шаблон для этой процедуры:



Начинайте с самого верха. При каждом делении записывайте частное в левый прямоугольник под делимым, а остаток — в правый прямоугольник под делимым. Вот как выглядит преобразование числа 51 966:



Заменив десятичные числа 12, 10, 15 и 14 шестнадцатеричными цифрами, получаем CAFE, что больше напоминает слово, чем число!

Ниже приводится таблица сложения для шестнадцатеричного формата.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

С помощью этой таблицы и обычных правил сложения в столбик, вы будете легко складывать шестнадцатеричные числа.

```
  4A3378E2
+ 877AB982
----------
  D1AE3264
```

Как вы помните из главы 13, для представления отрицательных чисел можно использовать дополнения до 2. В этом случае в двоичном формате все 8-разрядные отрицательные числа начинаются с 1. В шестнадцатеричном формате двузначные числа со знаком являются отрицательными, если они начинаются с 8, 9, A, B, C, D или F, поскольку в двоичном представлении все эти цифры начинаются с 1. Например, число 99h

в десятичном формате может означать 153 (если вы знаете, что имеете дело с однобайтовым беззнаковым числом) или –103 (если это число со знаком).

Вообще-то байт 99h может соответствовать и десятичному числу 99! В этом что-то есть, а? Но как это согласуется со всем тем, что вы до сих пор узнали? На этот вопрос я отвечу в главе 23, а пока поговорим о памяти.

Глава 16

Сборка памяти

Каждое утро мы пробуждаемся ото сна, и память заполняет пробелы в нашем сознании. Человек вспоминает, кто он такой, что делал вчера и чем собирается заняться сегодня. Память возвращается мгновенно или по каплям. Иногда даже несколько минут спустя в ней все еще остаются белые пятна («Странно, совершенно не помню, что ложился спать в носках»), но в конце концов мы собираем из этой мозаики целостную картину и готовы вступить в новый день.

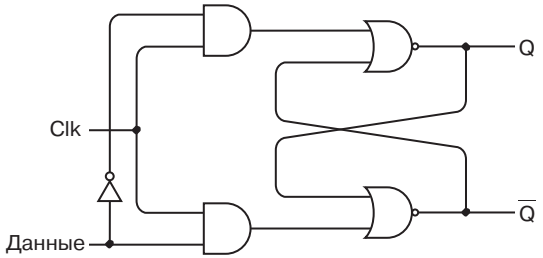
Конечно, человеческая память не очень-то упорядочена. Попробуйте вспомнить что-нибудь из школьного курса геометрии. Скорее всего начнете вы с воспоминаний о пареньке, сидевшем прямо перед вами, или об учебной пожарной тревоге, которая случилась как раз в тот момент, когда учитель собирался рассказать вам о треугольнике ABC.

И работает наша память небезупречно. Ведь и письмо, вероятно, было изобретено именно для противодействия «провалам памяти». Представьте: в три часа ночи вы просыпаетесь от того, что в голову пришел замечательный сюжет. Вы хватаете ручку и листок бумаги, которые держите у кровати как раз для таких оказий, и записываете все-все, чтобы до утра не забыть. С первыми лучами солнца вы прочитываете свою блестящую задумку («Парень встречает дев. с маш., погоня, все взрыв.» Я угадал?) и начинаете работать над сценарием блокбастера.

Мы *пишем*, чтобы позже *прочитать*. Мы *переносим слова и числа на бумагу*, чтобы позже при необходимости иметь возможность *мгновенно обратиться* к ним. Мы *сохраняем* информацию, чтобы позже *использовать* ее. Назначение запоминающих устройств как раз и состоит в том, чтобы хранить информацию между двумя этими событиями. Сохранение информации требует некоторого носителя. Бумага — замечательный носитель для текста, а магнитная лента идеально подходит для записи музыки и кино.

Реле — в составе вентилей и триггеров — также хранят информацию. Мы уже убедились, что триггер способен хранить 1 бит. Это, конечно, немного, но для начала сойдет. Ведь научившись сохранять 1 бит, мы легко сохраним и 2, и 3.

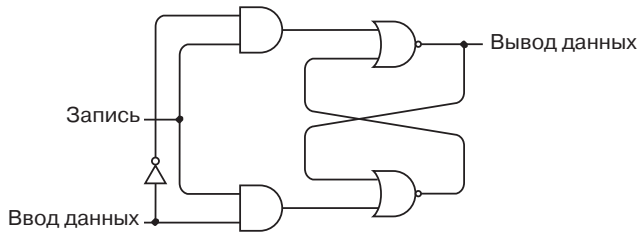
В главе 14 мы познакомились с D-триггером со срабатыванием по уровню, состоящим из инвертора, двух вентилей И и двух вентилей ИЛИ-НЕ:



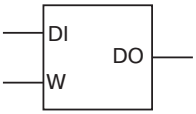
Если сигнал на входе Clk равен 1, выход Q совпадает со входом Данные. Когда сигнал Clk обращается в 0, выход Q сохраняет последнее значение на входе Данные. Дальнейшие изменения сигнала Данные не влияют на выходной сигнал, пока сигнал Clk снова не станет равным 1. Таблица логики триггера выглядит так:

Входы		**Выходы**	
D	**Clk**	**Q**	$\overline{\mathbf{Q}}$
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

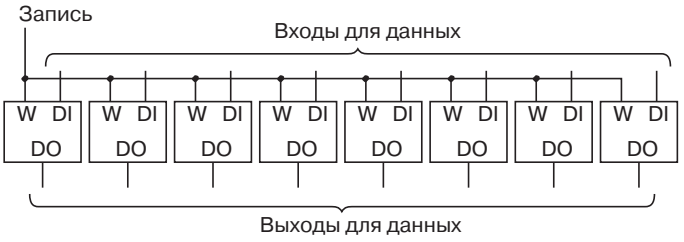
В главе 14 мы использовали этот триггер по-разному, но отныне у него останется только одна забота — хранить 1 бит информации. Поэтому я намерен дать его входам и выходам новые имена, более отвечающие их предназначению:



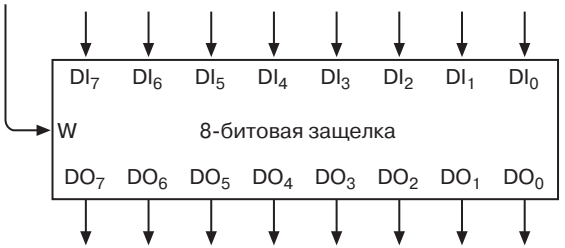
Это тот же триггер, только его выход Q назван Выводом данных (Data Out, DO), а вход Clk (который мы называли «Запомнить этот бит») получил имя Запись (Write, W). Сигнал на этом входе означает, что значение сигнала на входе Ввод данных (Data In, DI) нужно *записать*, или *сохранить*. В обычном состоянии сигнал Запись равен 0, и сигнал Ввод данных на выходной сигнал не влияет. Если значение сигнала Ввод данных нужно сохранить, мы подаем на вход Запись 1, а затем снова 0. Я уже говорил в главе 14, что триггеры такого типа называются *защелками*, так как данные в них как бы заперты. Чтобы не рисовать все компоненты, 1-битовую защелку можно изобразить так:



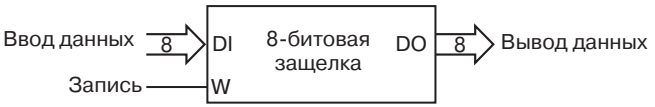
Из нескольких 1-битовых защелок легко собрать многобитовую. Достаточно соединить их входы Запись:



У этой 8-битовой защелки 8 входов и 8 выходов. Кроме того, у нее имеется единственный вход Запись, в обычном состоянии равный 0. Чтобы сохранить 8 битов информации, на вход Запись подается 1, а затем снова 0. Для 8-битовой защелки тоже можно ввести единое обозначение:



Или такое, не слишком отличающееся от обозначения 1-битовой защелки:



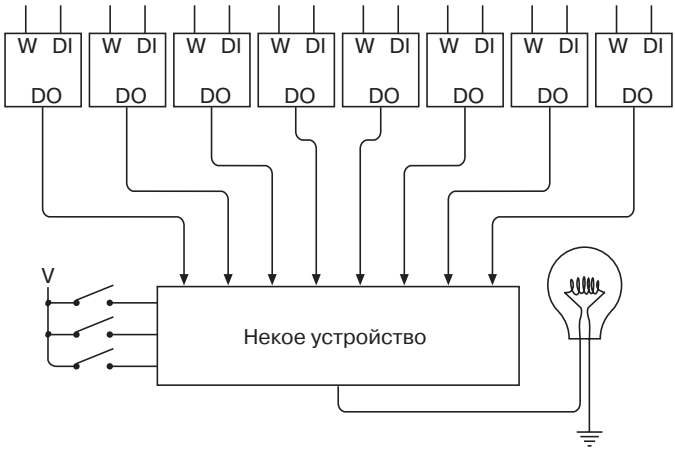
Другой способ соединения восьми 1-битовых защелок не столь нагляден. Допустим, у нас всего один вход и один выход для данных. При этом нам нужно записать 8 однобитовых значений, а затем и просмотреть их.

Иными словами, мы хотим сохранить не одно 8-битовое число (как в 8-битовой защелке), а 8 независимых 1-битовых величин, причем используя только один вход и один выход. Представьте, что у нас в наличии только одна лампочка.

Понятно, что нам понадобятся восемь 1-битовых защелок. Пока не будем беспокоиться о том, как именно в них попадут данные. Сосредоточимся на поиске способа проверять значение сигнала Вывод данных для каждой защелки, используя единственную лампочку. Конечно, мы всегда можем переносить лампочку от защелки к защелке *вручную*, но хочется чего-то более автоматизированного. Хорошо бы, скажем, номер защелки для просмотра задавать переключателями.

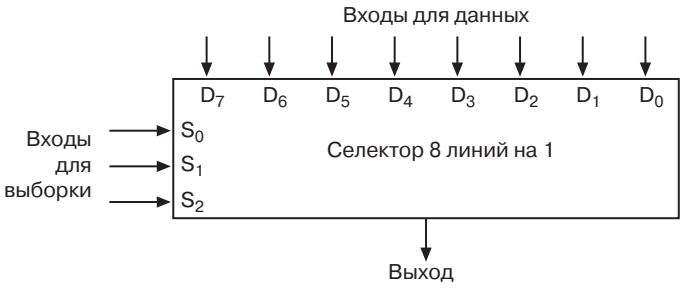
Сколько их нужно? Чтобы выбрать один из 8 вариантов нам понадобится три переключателя — ими можно представить 8 чисел: 000, 001, 010, 011, 100, 101, 110 и 111.

Итак: 1-битовых защелок — 8, переключателей — 3, лампочка — 1 и еще некое устройство, которое должно располагаться между лампочкой и переключателями.



Это устройство представляет собой корпус с 8 входами сверху и 3 входами слева. Замыкая и размыкая три переключателя, вы задаете, какой из входов должен перенаправляться на выход, подключенный к лампочке.

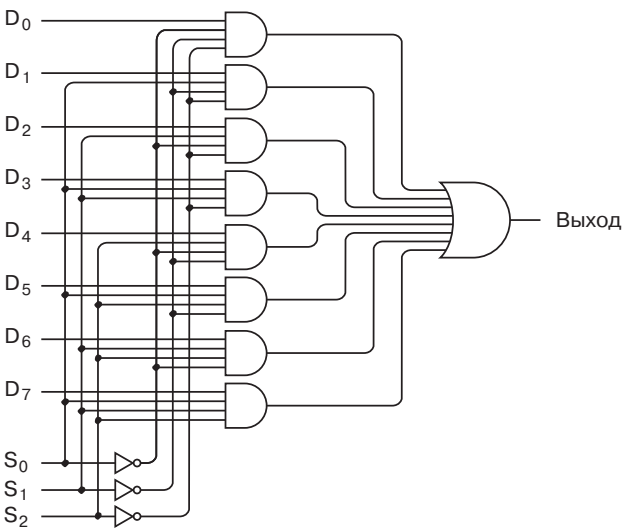
Что это за устройство? Что-то похожее мы уже встречали, хотя и не с таким большим количеством входов. Подобную схему мы применяли в модернизированном сумматоре (глава 14). С ее помощью мы задавали, откуда должен поступать сигнал на вход сумматора: с набора переключателей или с защелки. Тогда мы называли этой устройство селектором 2 линии на 1. Сейчас нам нужен *селектор 8 линий на 1*:



Наверху показаны 8 входов для данных, а слева — 3 входа для выборки (Select). Используя их, вы задаете вход, сигнал с которого попадает на выход. Если сигналы Select равны 000, на выход попадает сигнал со входа D_0. Сигналы 111 задают вывод данных со входа D_7, а число 101 на входах для выборки означает вывод сигнала D_5. Вот как выглядит таблица логики селектора.

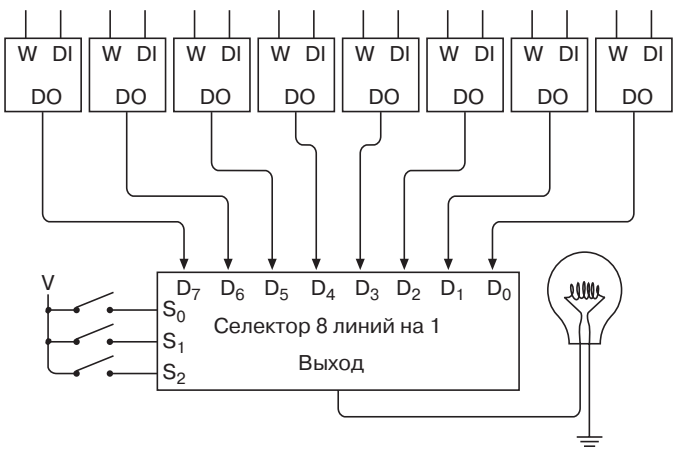
Входы			**Выходы**
S_2	**S_1**	**S_0**	**Q**
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

Селектор «8 на 1» состоит из трех инверторов, восьми 4-входовых вентилей И и одного 8-входового вентиля ИЛИ:



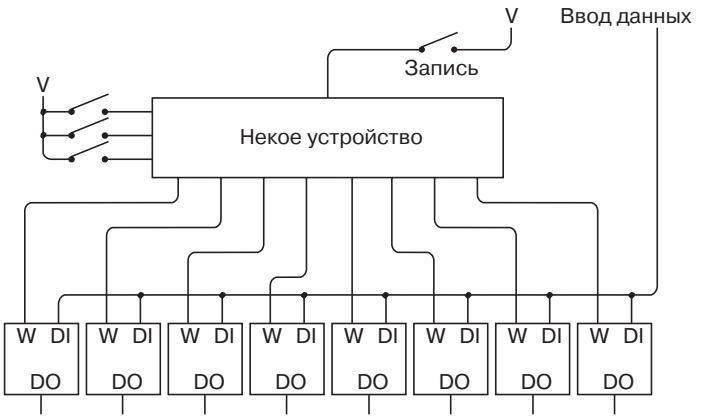
Я попытаюсь на одном примере убедить вас, что эта паутина проводов все-таки работает. Предположим, что сигналы S_2 и S_0 равны 1, а сигнал S_1 — 0. На шестой сверху вентиль И подаются сигналы S_0, $\overline{S}_1$ и S_2, все равные 1. Ни на один другой вентиль И эти сигналы не попадают, поэтому выход всех остальных вентилей И равен 0. Понятно, что выход шестого вентиля И будет равен 0, если сигнал $D_5 = 0$, или 1 при $D_5 = 1$. То же относится и к вентилю ИЛИ. Таким образом, если сигналы для выборки равны 101, выход совпадает с сигналом D_5.

Вы еще не забыли, чем мы занимаемся? Мы пытаемся связать восемь 1-битовых защелок так, чтобы данные в них можно было записывать и считывать индивидуально, используя единственный вход DI и единственный выход DO. Мы уже убедились, что для вывода содержимого одной из защелок можно применить селектор 8 линий на 1:



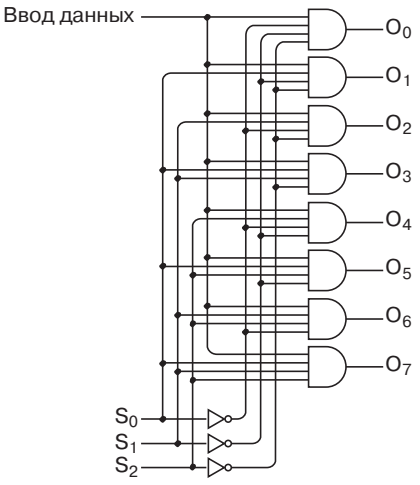
Мы на полпути к успеху. Теперь, зная, какое устройство нужно на выходе, займемся входом.

На входе мы имеем сигналы Данные и Запись. И если входы D всех 1-битовых защелок можно соединить между собой, с сигналами Запись это уже не пройдет, поскольку мы хотим осуществлять запись в каждую защелку индивидуально. Нам нужен сигнал Запись, подаваемый на одну (и только одну) защелку:



Для решения этой задачи нам понадобится схема, которая напоминает селектор, но в действительности выполняет прямо противоположное действие, а именно *дешифратор 3 линии на 8* (3-to-8 Decoder). С простым дешифратором мы уже встречались — в главе 11 он помогал нам задать цвет идеальной кошки.

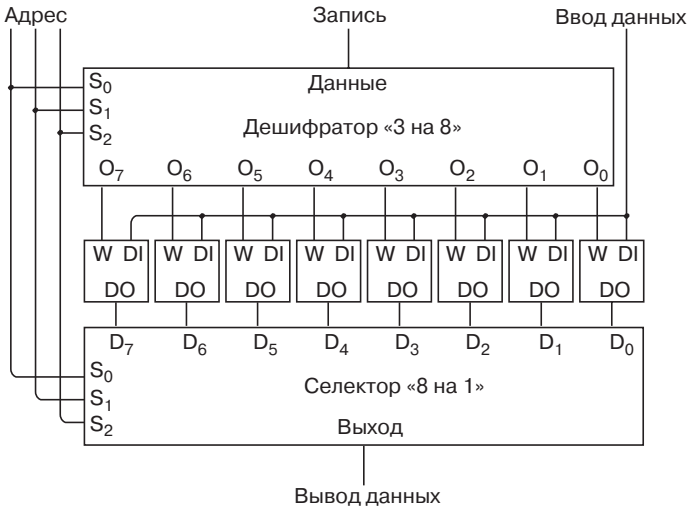
У дешифратора «3 на 8» восемь выходов. В любой момент времени все они равны 0, кроме выхода, указанного сигналами S_0, S_1 и S_2. Его значение совпадает со значением входа Ввод данных.



И снова обратим внимание на шестой сверху вентиль И. Три из его сигналов равны S_0, $\overline{S}_1$ и S_2, которые ни на один другой вентиль не подаются. Это значит, что если на входы для выборки подается 101, то выход всех вентилей, кроме шестого, будет равен 0. Его же выход равен 1, если сигнал Данные равен 1, и 0, если сигнал Данные равен 0. Заполним таблицу логики для этого дешифратора.

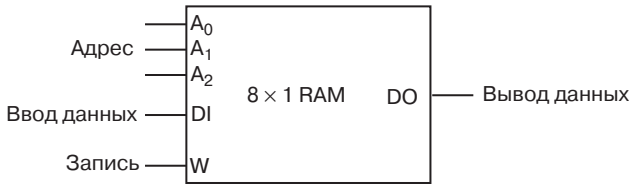
Входы			**Выходы**							
S_2	**S_1**	**S_0**	**O_7**	**O_6**	**O_5**	**O_4**	**O_3**	**O_2**	**O_1**	**O_0**
0	0	0	0	0	0	0	0	0	0	Данные
0	0	1	0	0	0	0	0	0	Данные	0
0	1	0	0	0	0	0	0	Данные	0	0
0	1	1	0	0	0	0	Данные	0	0	0
1	0	0	0	0	0	Данные	0	0	0	0
1	0	1	0	0	Данные-	0	0	0	0	0
1	1	0	0	Данные	0	0	0	0	0	0
1	1	1	Данные	0	0	0	0	0	0	0

А теперь посмотрим, как полностью выглядит схема с защелками.



Заметьте: сигналы выборки для дешифратора и селектора совпадают. Обратите внимание и на слово, которым я назвал эти сигналы — *адрес* (address). 3-битовый адрес действует подобно почтовому индексу, указывая, к которой из 8 защелок происходит обращение. На входе сигнал Адрес определяет, в какую защелку будет записан сигнал Ввод данных. На выходе (в нижней части рисунка) адрес используется, чтобы с помощью селектора «8 на 1» считать содержимое одной из восьми защелок.

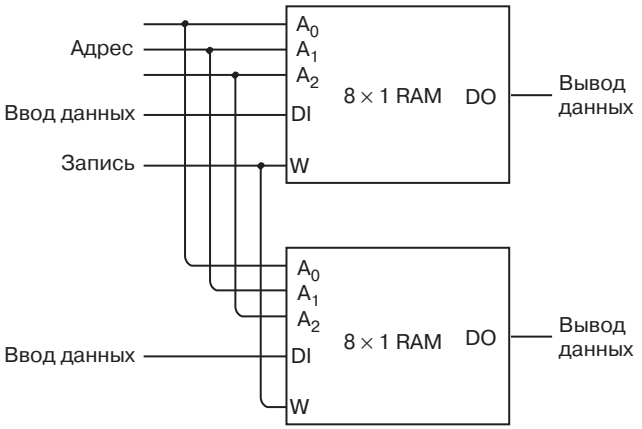
Защелки в такой конфигурации иногда называют *памятью с записью/чтением* (read/write memory), а чаще — *памятью с произвольным доступом, или произвольной выборкой* (random access memory, RAM). В данном варианте память RAM используется для хранения 8 независимых 1-битовых величин. На схеме ее можно обозначить так:



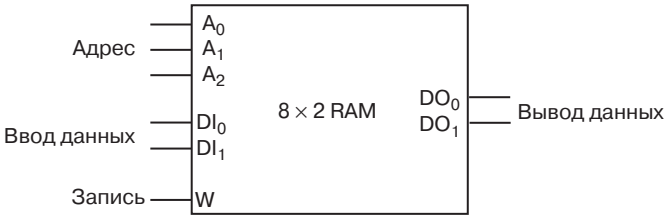
Памятью это устройство названо из-за его способности сохранять информацию. Возможность *записи/чтения* означает, что вы вольны сохранить (*записать*) нужное значение в любую из защелок или определить (*прочитать*), какое значение в ней уже находится. Термин «произвольный доступ» означает, что данные могут быть записаны или прочитаны из любой защелки — достаточно подать соответствующие сигналы на вход Адрес. Этим память с произвольным доступом отличается от устройств с последовательным доступом: в них нельзя прочитать данные по адресу 101, не прочитав перед этим данные по адресу 100.

Разобранную нами конфигурацию RAM часто называют *массивом RAM* (RAM array). Наш массив собран по схеме, которую иногда сокращенно обозначают «8 × 1», т. е. восемь 1-битовых значений. Перемножив два этих числа, вы получите полное число битов, которое можно сохранить в массиве.

Массивы RAM можно объединять по-разному. Вот, например, как создать память для хранения восьми 2-битовых значений.

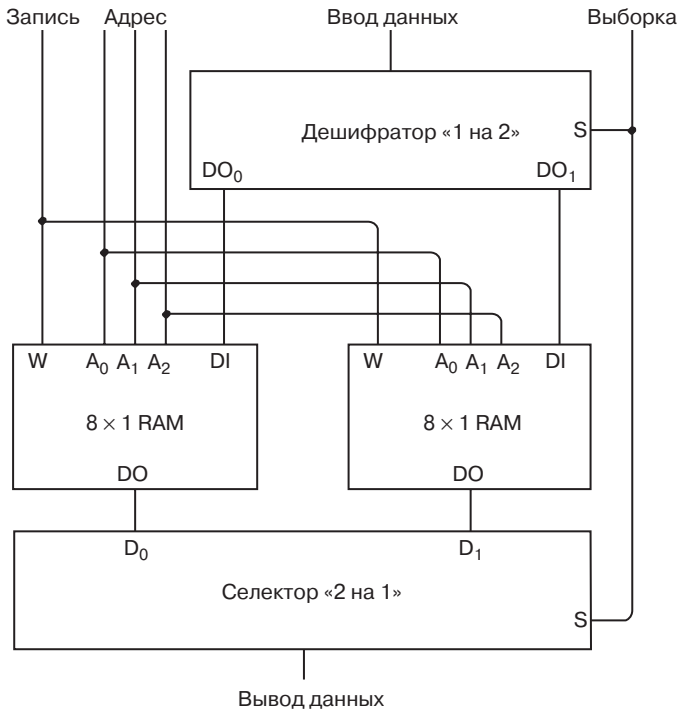


Как видите, сигналы Адрес и Запись двух массивов соединены между собой, поэтому в результате получаем массив 8 × 2.



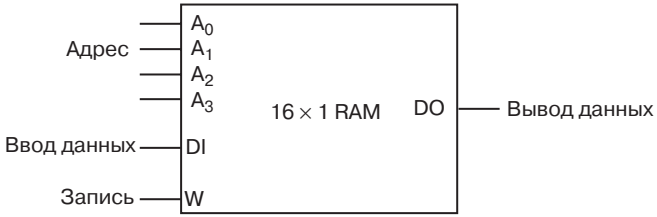
Массив по-прежнему способен хранить 8 значений, но каждое из них имеет размер 2 бита.

В другой схеме два массива RAM 8 × 1 соединены подобно отдельным защелкам — с помощью селектора «2 на 1» и дешифратора «1 на 2».



Сигнал Выборка, подаваемый на селектор и дешифратор, по сути выбирает один из двух массивов RAM 8 × 1 и является

четвертой адресной линией. Иначе говоря, мы собрали массив RAM 16 × 1

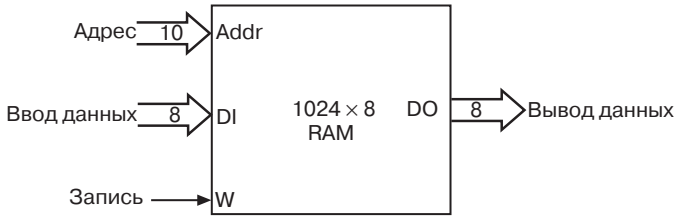


В нем можно хранить шестнадцать 1-битовых значений.

Число значений, хранение которых допускается в массиве, непосредственно связано с количеством адресных линий. Если бы входов Адрес вовсе не было (как в 1-битовой или 8-битовой защелке), хранить можно было бы всего одно значение. С одним входом Адрес допускается хранение уже двух значений, с тремя входами — 8 значений, с четырьмя — 16 и т. д. в соответствии с формулой:

$$\text{Число значений в массиве RAM} = 2^{\text{Количество входов Адрес}}$$

Собрав небольшой массив RAM, мы легко представим себе сборку большого массива, например, такого:



Это массив предназначен для хранения 8 196 битов, разбитых на 1 024 восьмибитовых числа. Для выбора из них нужного используются 10 адресных линий, поскольку $1024 = 2^{10}$. Кроме того, в массиве 8 входов и 8 выходов для данных.

Другими словами, в этом массиве, как в почтовом отделении с 1 024 абонентскими ящиками, хранятся 1 024 байта, или 1 килобайт. Термин «килобайт» часто становится причиной некоторой путаницы. Приставка «кило», происходящая от греческого слова, означающего тысячу, часто используется в обыден-

ной жизни. Например, в килограмме 1 000 граммов, а в километре 1 000 метров. Но в килобайте 1 024 байта, а вовсе не 1 000.

Проблема в том, что метрическая система мер основана на десятичной системе счисления, а компьютерные единицы — на двоичной системе. Степени 10 равны 10, 100, 1 000, 10 000, 100 000 и т.д. Степени 2 равны 2, 4, 8, 16, 32, 64 и т.д. Ни одна степень 10 не равна степени 2.

Но иногда они подходят близко друг к другу. Да, 1 000 действительно мало отличается от 1 024, что выражается математическим знаком «приблизительно равно»:

$$2^{10} \approx 10^3.$$

Ничего волшебного в этом соотношении нет. Оно просто говорит о том, что данная степень 2 приблизительно равна данной (другой) степени 10. Это замечательное совпадение и позволяет называть 1 024 байта *килобайтом*. Сокращенно килобайт обозначается кб или просто К. Говорят, что объем памяти равен 1 024 байтам, или 1 килобайту, или 1 кб, или 1 К. Но, конечно, нельзя говорить, что в 1 килобайте 1 000 байтов.

У памяти объемом 1 кб по 8 входов для ввода и вывода данных и 10 адресных линий. Этих десяти линий необходимо и достаточно для адресации 2^{10} байтов. Добавив еще один адресный вход, мы удваиваем объем адресуемой памяти. В приведенном ниже примере каждая следующая строчка есть удвоение предыдущей строчки:

1 килобайт = 1 024 байта = 2^{10} байтов $\approx 10^3$ байтов
2 килобайта = 2 048 байтов = 2^{11} байтов
4 килобайта = 4 096 байтов = 2^{12} байтов
8 килобайтов = 8 192 байта = 2^{13} байтов
16 килобайтов = 16 384 байта = 2^{14} байтов
32 килобайта = 32 768 байтов = 2^{15} байтов
64 килобайта = 65 536 байтов = 2^{16} байтов
128 килобайтов = 131 072 байта = 2^{17} байтов
256 килобайтов = 262 144 байта = 2^{18} байтов
512 килобайтов = 524 288 байтов = 2^{19} байтов
1024 килобайта = 1 048 576 байтов = 2^{20} байтов $\approx 10^6$ байтов

Та же логика, что позволила нам называть 1 024 байта килобайтом, приводит и к следующему обозначению — 1 024

килобайта будем называть *мегабайтом* (приставка «мега» в обычных единицах означает миллион и происходит от греческого слова «великий»), или сокращенно Мб или MB. Продолжаем удваивать память.

1 мегабайт = 1 048 576 байтов = 2^{20} байтов ≈ 10^6 байтов
2 мегабайта = 2 097 152 байта = 2^{21} байтов
4 мегабайта = 4 194 304 байта = 2^{22} байтов
8 мегабайтов = 8 388 608 байтов = 2^{23} байтов
16 мегабайтов = 16 777 216 байтов = 2^{24} байтов
32 мегабайта = 33 554 432 байта = 2^{25} байтов
64 мегабайта = 67 108 864 байта = 2^{26} байтов
128 мегабайтов = 134 217 728 байтов = 2^{27} байтов
256 мегабайтов = 268 435 456 байтов = 2^{28} байтов
512 мегабайтов = 536 870 912 байта = 2^{29} байтов
1024 мегабайта = 1 073 741 824 байта = 2^{30} байтов

Следующей по очереди идет приставка «гига» (от греческого «гигантский»), и мы уже уверенно называем 1024 мегабайта *гигабайтом* (Гб или GB).

Далее, один *терабайт* (от греческого «тера» — «чудовищный») равен 2^{40} байтов (приблизительно 10^{12}) = 1 099 511 627 776 байтов. Сокращается он как Тб или TB.

Килобайт примерно равен тысяче байтов, мегабайт — миллиону байтов, гигабайт — миллиарду байтов, терабайт — триллиону байтов.

Теперь мы вступаем туда, где до нас отваживались побывать немногие, и знакомимся с *петабайтом* (2^{50} = 1 125 899 906 842 624 байтов, т. е. приблизительно 10^{15} или квадриллион) и *экзабайтом* (2^{60} = 1 152 921 504 606 846 976 байтов, т. е. приблизительно 10^{18}, или квинтиллион).

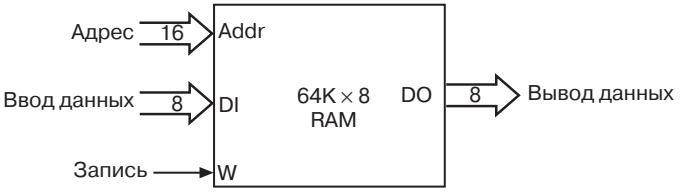
В качестве практического примера сообщу, что во время написания этой книги (1999 г.) домашние компьютеры, как правило, обладали 32 или 64 (реже 128) Мб памяти с произвольным доступом (не запутайтесь — я говорю именно о памяти RAM, пока не упоминая жесткие диски). Эти объемы соответствуют 33 554 432, 67 108 864 и 134 217 728 байтам.

Конечно, на практике никто такие страшные числа и слова не использует. Говорит человек просто: «У моего компьютера 64 К памяти», — и все понимают, что это гость из далекого прошлого. Обладатели памяти объемом 33 554 432 байта на-

зывают ее «32 мега». Счастливый владелец 1 073 741 824 байтов говорит о них, как об «одном гиге».

Иногда приходится слышать и о *килобитах* или *мегабитах* (именно битах, а не байтах), но в подобных случаях речь почти наверняка идет не о памяти. Как правило, в этих единицах измеряют объем информации, передаваемой по различным кабелям, или о скорости передачи. В последнем случае, конечно, говорят о килобитах в секунду или о мегабитах в секунду. Например, 56K в обозначении модема означает именно 56 килобит в секунду.

Теперь, когда мы научились собирать память любого объема, главное не увлечься и не утратить контроль над происходящим. Поэтому пока ограничимся сборкой массива емкостью 65 536 байтов:



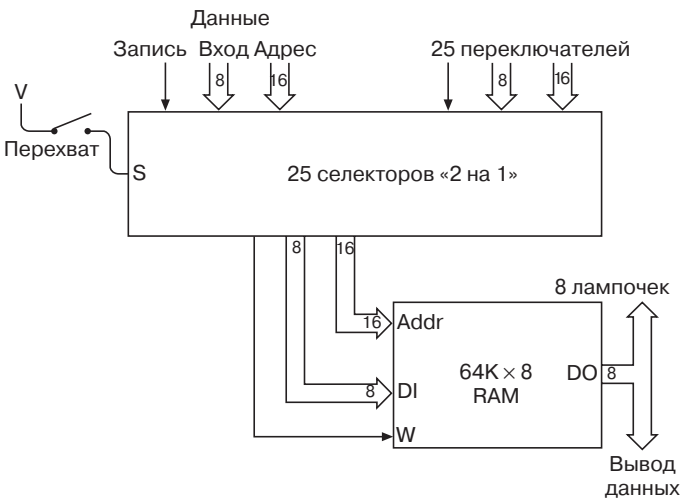
Почему именно 64 K, а не 32 или 128? Потому что *65 536 — очень красивое и круглое число*, равное 2^{16}. Для адресации ячеек памяти понадобится 16-битовый адрес, т. е. адрес, занимающий ровно 2 байта. В шестнадцатеричной системе адрес может принимать значение от 0000h до FFFFh.

Я намекал чуть раньше, что в далеком прошлом (в начале 1980-х) компьютеры часто выпускались с памятью объемом 64 K, хотя делали ее, конечно, не из реле. Да и можно ли собрать память такой емкости из реле? Уверен, у вас хватит ума никогда этим не заниматься. В нашей схеме для хранения одного бита используется 9 реле, стало быть массив RAM 64K × 8 будет содержать их почти 5 миллионов!

Было бы неплохо обзавестись пультом управления памятью, который позволял бы записывать данные в ячейки памяти и проверять их содержимое. На этом пульте будет 16 переключателей для указания адреса, 8 — для ввода числа, которое нужно записать, еще один — для подачи сигнала на запись и 8 лампочек для отображения 8-разрядного числа из памяти.

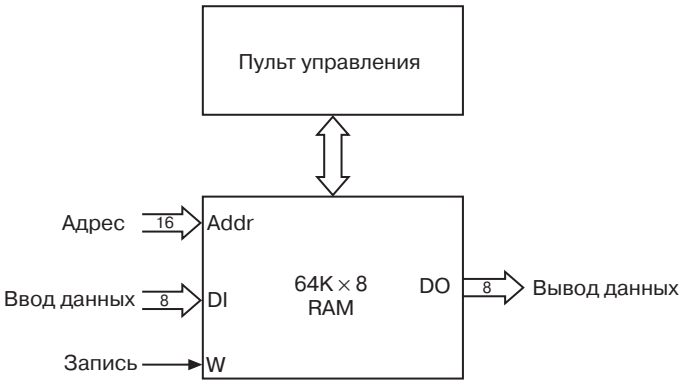
На рисунке все переключатели стоят в положении «выключено» (0). Обратите внимание на переключатель Перехват. Его назначение — разрешить другим устройствам использование той же памяти, к которой подключен пульт. Когда этот переключатель выключен (как на рисунке), остальные переключатели на пульте с памятью не связаны. Когда же включен, управление памятью осуществляется исключительно с помощью пульта.

Для реализации переключателя Перехват нам потребуется 25 селекторов «2 на 1» — 16 для адресных переключателей, 8 для переключателей, с помощью которых вводится число, и еще один для переключателя Запись. Вот как выглядит нужная схема:



Когда переключатель Перехват разомкнут, как показано на рисунке, информация на входы массива памяти Адрес, Данные и Запись поступает извне (эти сигналы схематически изображены в верхней левой части рисунка). Если переключатель замкнут, информация на эти входы подается с переключателей. Независимо от того, откуда в память поступает информация, из нее она идет на 8 лампочек и, возможно, куда-нибудь еще.

Массив памяти RAM 64K × 8 с пультом управления я буду изображать так:



Если переключатель Перехват замкнут, с помощью 16 адресных переключателей вы можете задать любой из 65 536 адресов. На 8 лампочках будет отображено число, хранящееся в соответствующей ячейке в данный момент. Чтобы записать в ячейку другое число, введите его с помощью переключателей D и замкните переключатель Запись.

Пульт управления, подключенный к массиву RAM 64K × 8, безусловно поможет вам разобраться во всех 65 536 восьмибитовых значениях, записанных в памяти. Но заметьте: пульт не захватывает память в единоличное владение. Другим схемам также позволено записывать в нее данные и считывать их.

В заключение — *важнейший* комментарий. Определив в главе 11 понятие логического вентиля, я перестал рисовать на схемах отдельные реле, из которых эти вентили состоят. А это, в частности, значит, что на схемах теперь не указывается подключение каждого реле к тому или иному источнику питания.

А ведь именно электричество заставляет железный брусок становиться магнитом и притягивать металлический контакт.

Что же случится, если вы соберете память объемом 64K × 8, заполните ее до отказа самыми любимыми байтами, а потом выключите питание? Электромагниты утратят магнитные свойства, и все металлические контакты вернутся в исходные положения с громким «блямс»! А содержимое памяти превратится в ничто, и навсегда.

Вот почему память с произвольным доступом называют иногда *энергозависимой* (volatile). Для хранения информации ей требуется непрерывное энергоснабжение.

Глава 17

Автоматизация

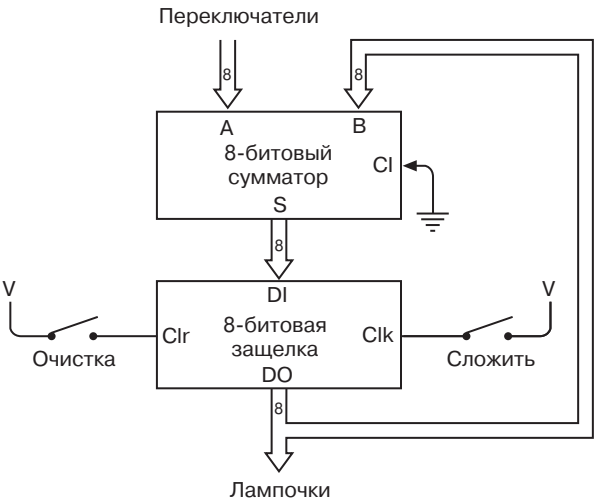
Род человеческий наделен удивительной смекалкой и усердием, но в то же время чудовищно ленив. Не открою страшной тайны, сказав, что работать люди не любят. Наше отвращение к труду столь сильно, а ум столь изобретателен, что мы готовы посвящать бесчисленные часы проектированию и сборке устройств, которые на пару минут сократят рабочий день. И ошеломляюще воздействует на мозговой центр удовольствий волшебная греза: я блаженно раскачиваюсь в гамаке, а агрегат, только что собранный мною по последнему слову техники, подстригает лужайку перед домом.

Не надейтесь, что на ближайших страницах вам попадутся чертежи автоматической газонокосилки. Вам предстоит познакомиться с набором устройств куда более сложных, посредством которых я намерен автоматизировать сложение и вычитание. Знаю: звучит это не слишком внушительно. И все же к концу главы в нашем распоряжении появится устройство, способное решить практически любую задачу, в которой используется сложение и вычитание, а круг таких задач гораздо шире, чем вы представляете.

Конечно, разнообразие внешних функций подразумевает внутреннюю сложность, и потому путь, по которому нам предстоит идти, не всегда будет гладким. Никто не обвинит вас, если несколько особо запутанных страниц вы пролистаете, не читая. Временами вам будет хотеться стукнуть кулаком по сто-

лу и поклясться страшной клятвой никогда не прибегать к помощи электрических и механических приспособлений для решения математических задач. Но не сдавайтесь — устройство, которое мы соберем, может с полным правом именоваться *компьютером*.

Последнюю модель сумматора мы разработали в главе 14. В эту модель входила 8-битовая защелка, в которой накапливалась сумма чисел, вводимых с помощью набора из 8 переключателей:



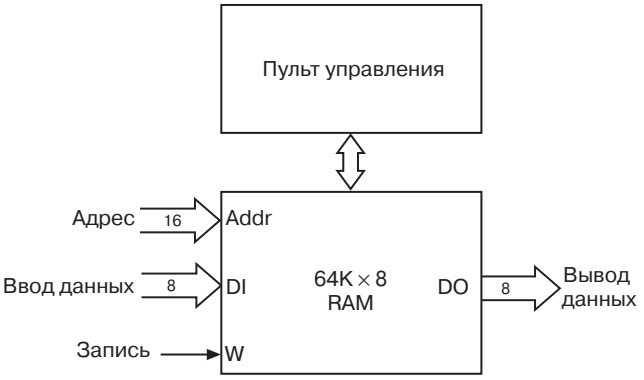
Как вы помните, в защелке с помощью триггеров сохраняется 8-битовое значение. Чтобы задействовать сумматор, на мгновение нажмите кнопку Очистка, чтобы записать в защелку 8 нулей, а затем введите с помощью переключателей первое число и нажмите кнопку Сложить. Сумматор прибавит его к нулевому содержимому защелки, получив в результате все то же введенное число. Оно будет отображено лампочками. Введите с помощью переключателей второе число и снова нажмите кнопку Сложить. Сумматор прибавит введенное значение к содержимому защелки, запишет в защелку текущий результат суммирования и отобразит его с помощью лампочек. Таким образом вы можете складывать длинные ряды чисел, помня, естественно, об ограничении — с помощью 8 лампочек нельзя отобразить число, превышающее 255.

К тому времени, когда я в главе 14 показал вам эту схему, мы успели познакомиться только с защелками со *срабатыванием по уровню*. В такой защелке запись входного сигнала осуществляется, когда сигнал Clk обращается сначала в 1, а потом в 0. Пока сигнал Clk равен 1 все изменения входного сигнала отражаются на содержимом защелки. Потом в той же главе мы узнали еще об одном типе защелок — со *срабатыванием по фронту*. В таких защелках входной сигнал сохраняется в краткий промежуток времени, когда значение сигнала Clk меняется с 0 на 1. Защелками со срабатыванием по фронту легче пользоваться, поэтому далее я буду предполагать, что нам встречаются защелки только этого типа.

Защелка, применяемая для хранения текущего значения суммы, называется *аккумулятором* (accumulator). Чуть позже мы убедимся, что накоплением суммы роль аккумулятора не ограничивается. В нем часто сначала хранится исходное число, а потом результат произведенного над этим числом действия (сложения или вычитания).

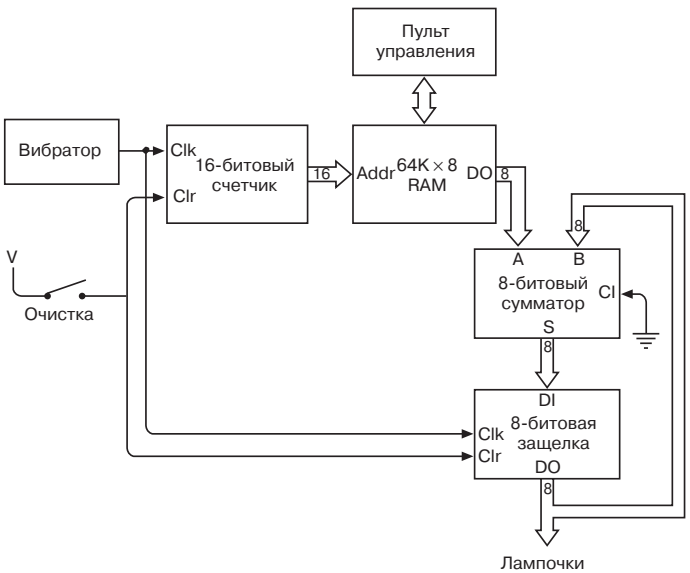
Недостаток показанного сумматора очевиден. Допустим, вам нужно найти сумму 100 двоичных чисел. Вы подсаживаетесь к сумматору и усердно вводите всю сотню число за числом. И вот, когда работа закончена, вдруг оказывается, что пару чисел из списка вы ввели неправильно. Придется начинать все сызнова.

Или не придется? В главе 16 мы узнали, как из 5 миллионов реле собрать память RAM объемом 64 кб, и разработали пульт для работы с этой памятью. На этом пульте есть переключатель Перехват, позволяющий перехватывать управление памятью и вручную изменять ее содержимое с помощью тумблеров.



Если бы вы ввели 100 ваших слагаемых в память, а не прямо в сумматор, исправить ошибки было бы куда легче.

Итак, перед нами нелегкая задача: подключить память RAM к сумматору с накоплением суммы. Совершенно ясно, что вместо переключателей к сумматору нужно подключить выходы DO массива памяти. А вот возможность управлять адресными входами памяти с помощью двоичного счетчика (мы познакомились с ним в главе 14) уже не так очевидна. Сигналы массива DI и Запись в этой схеме не нужны:



Смело утверждаю, что людям случалось изобретать вычислительные устройства и поудобнее. Чтобы воспользоваться этой штукой, вы должны прежде всего замкнуть переключатель Очистка, тем самым обнулив содержимое защелки и установив выходной сигнал 16-битового счетчика в 0000h. Затем замкнем переключатель Перехват на пульте управления памятью и введем первое слагаемое в память по адресу 0000h. Если вы собираетесь сложить 100 чисел, они займут в памяти адреса с 0000h по 0063h. По всем неиспользованным адресам должны быть записаны нули. Разомкните переключатель Перехват на пульте управления памятью (чтобы вернуть сумма-

тору доступ к памяти) и переключатель Очистка. А теперь расслабьтесь и получайте удовольствие от созерцания мигающих лампочек!

А чтобы вы при этом не скучали, я расскажу, что именно происходит в сумматоре. В момент размыкания переключателя Очистка текущий адрес памяти равен 0000h. На один из входов сумматора подается сохраненная по этому адресу величина. На второй вход подается число 00h, поскольку содержимое защелки также очищено.

Вибратор подает в схему синхронизирующий сигнал, т. е. сигнал, осциллирующий между 0 и 1. Если переключатель Очистка разомкнут, при переходе синхронизирующего сигнала из 0 в 1 одновременно происходят две вещи. Во-первых, в защелку записывается результат суммирования, во-вторых, выход счетчика увеличивается на 1, указывая теперь на следующую ячейку памяти. Когда сигнал вибратора переходит из 0 в 1 в первый раз после размыкания переключателя Очистка, в защелку записывается первое слагаемое, а выход счетчика обращается в 0001h. Во второй раз в защелку записывается сумма первого и второго слагаемых, а сигнал на выходе счетчика становится 0002h. И т. д.

Конечно, для работы этой схемы требуется соблюдение некоторых условий. Например, колебания сигнала вибратора должны происходить достаточно медленно, чтобы схема успевала на них реагировать. Ведь каждое изменение синхронизирующего сигнала приводит в действие множество реле, которые переключают другие реле и т. д.

У этой схемы есть и крупный недостаток: мы не в состоянии остановить ее работу! В какой-то момент лампочки перестанут мигать, так как, начиная с некоторой ячейки, в памяти хранятся одни нули. Но когда счетчик достигнет значения FFFFh, он обнулится, совсем как счетчик километров на спидометре автомобиля, и сумматор начнет прибавлять те же слагаемые к уже посчитанной сумме.

Этот не единственный недостаток. Наш сумматор способен только складывать и работает только с 8-битовыми значениями. Величиной 255 ограничены как числа, записанные в память, так и их сумма. Если вы решите реализовать не только сложение, но и вычитание (с помощью дополнений до 2), доступный вам диапазон будет ограничен значениями от –128 до

127. Очевидный способ научить сумматор работать с большими числами (например, 16-битовыми) — удвоить ширину массива RAM, сумматора, защелки, а также добавить на пульт управления 8 лампочек. Но давайте считать, что к такому вложению капитала вы пока не готовы.

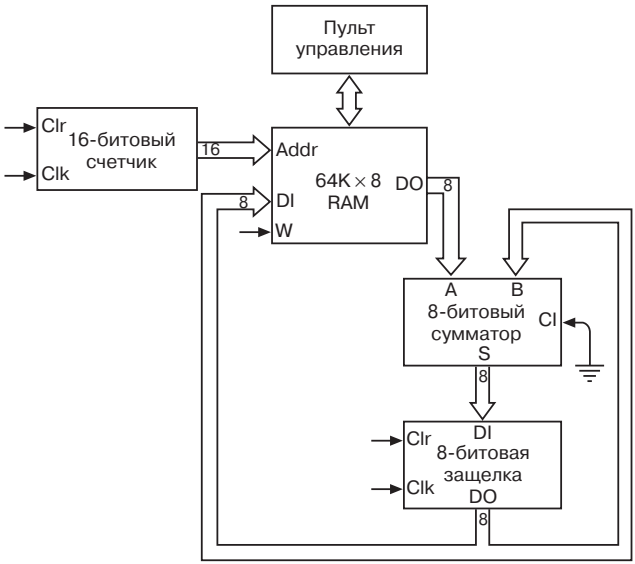
Конечно, я не заговорил бы об этих проблемах, если бы не собирался в конце концов решить их. Но давайте пока сформулируем еще одну проблему. Что если сумма сотни слагаемых вас не интересует, а хочется вам сложить 100 чисел попарно, получив 50 различных сумм? Что если вам хочется машину, которой можно было бы указывать, как именно нужно разбить сотню чисел на суммы? Причем все результаты суммирования должны сохраняться для последующего использования.

Автоматический сумматор, с которым мы работали до сих пор, с помощью лампочек, подключенных к защелке, отображает только текущее значение суммы. От этих лампочек мало проку, если вам нужно просмотреть 50 сумм, полученных в результате попарного суммирования 100 слагаемых. Удобнее записывать эти суммы обратно в память. С помощью пульта управления памятью вы потом просмотрите эти числа в любое удобное для вас время. Специально для таких случаев на пульте управления памятью предусмотрен собственный набор лампочек.

Но это значит, что от лампочек, подключенных к защелке, мы и вовсе можем отказаться. Подключим выходы защелки вместо лампочек ко входам массива RAM, чтобы результаты суммирования записывались прямо в память.

Из схемы, показанной на следующей странице, я также удалил некоторые другие компоненты сумматора, а именно вибратор и переключатель Очистка. Я сделал это потому, что теперь уже не так очевидно, откуда на счетчик и защелку поступают сигналы Clk и Clr. Кроме того, мы задействовали входы массива RAM, а значит нам понадобится вход W (Запись).

Но давайте не будем пока тревожиться о деталях схемы, сосредоточившись вместо этого на проблеме, которую пытаемся решить. Нам нужно изменить конфигурацию сумматора так, чтобы его функции не ограничивались простым накоплением суммы набора чисел. Мы хотим полной свободы в выборе количества слагаемых и количества сумм, которые мы собираемся записывать в память для дальнейшего использования.



Допустим, нам нужно найти три суммы: трех слагаемых, двух слагаемых и еще трех слагаемых. Представим себе, что мы ввели все слагаемые в память, начиная с адреса 0000h. Содержимое памяти после этого выглядит так:

Именно так я буду показывать содержимое памяти. Прямоугольники представляют собой отдельные ячейки памяти. Каждый байт находится в своей ячейке. Адрес ячейки указан

слева. Все адреса указывать нужды нет, так как они идут последовательно друг за другом, и вы всегда легко подсчитаете, чему равен отсутствующий адрес. Справа указаны пояснения к содержимому памяти. В данном случае я отметил, что сумматор должен записывать суммы в пустые ячейки. (Кстати, на самом деле память не бывает пустой. Она всегда *что-то* содержит, даже если это что-то — случайное сочетание битов. Важно, что в данный момент эти ячейки не содержат нужной информации.)

Я знаю, вы сейчас боретесь с искушением попрактиковаться в шестнадцатеричной арифметике и заполнить пустые ячейки самостоятельно. Но не забывайте, что наша задача не в этом. Мы хотим, чтобы сложением вместо нас занимался автоматический сумматор.

До сих пор он выполнял единственное действие: складывал содержимое ячейки памяти с содержимым 8-битовой защелки, которую я назвал аккумулятором. Но этого мало — теперь он должен выполнять *четыре различных операции*. Чтобы начать сложение, сумматор должен переписать содержимое ячейки памяти в аккумулятор. Эту операцию я буду называть *загрузкой*. Второе действие — *сложить* байт из памяти с содержимым аккумулятора. В-третьих, мы должны *сохранить* в памяти сумму из аккумулятора. Наконец, надо как-то *остановить* работу сумматора.

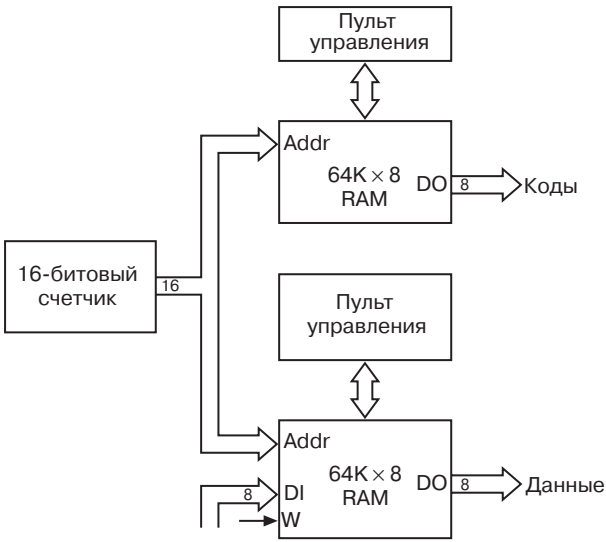
Если разобрать эту последовательность по косточкам, то в данном конкретном примере сумматор должен сделать следующее:

- *загрузить* значение из ячейки 0000h в аккумулятор;
- *сложить* значение из ячейки 0001h с аккумулятором;
- *сложить* значение из ячейки 0002h с аккумулятором;
- *сохранить* значение из аккумулятора в ячейке 0003h;
- *загрузить* значение из ячейки 0004h в аккумулятор;
- *сложить* значение из ячейки 0005h с аккумулятором;
- *сохранить* значение из аккумулятора в ячейке 0006h;
- *загрузить* значение из ячейки 0000h в аккумулятор;
- *сложить* значение из ячейки 0001h с аккумулятором;
- *сложить* значение из ячейки 0002h с аккумулятором;
- *сохранить* значение из аккумулятора в ячейке 0003h;
- *остановить* работу.

Заметьте: как и в исходном сумматоре, байты в памяти адресуются последовательно, начиная с адреса 0000h. Раньше использование памяти сводилось к тому, что содержимое конкретной ячейки складывалось с содержимым аккумулятора. Потребность в этой функции сохранилась и теперь. Но в дополнение к ней мы хотим иногда прямо *загружать* в аккумулятор значение из памяти или *сохранять* в памяти значение из аккумулятора. Кроме того, сумматор, закончив работу, должен остановиться и дать нам возможность проверить содержимое памяти.

Как этого достичь? Не ждем же мы, что сумматор сам сообразит, что делать с введенными в память числами. Каждое число в памяти нужно сопроводить кодом, который указывал бы сумматору нужное действие: *загрузить*, *сложить*, *сохранить* или *остановить*.

Вероятно, проще всего (хотя и не дешевле всего) завести для решения этой задачи отдельный массив RAM, доступ к которому будет осуществляться одновременно с первым. Только содержать он будет не складываемые числа, а коды, символизирующие действия, которые сумматор должен выполнять с заданными ячейками памяти. Будем называть старый массив Данные, а новый — Коды.



Мы уже разобрались, что в новом автоматическом сумматоре нам понадобится возможность записи чисел в массив Данные. Доступ к массиву Коды, напротив, будет осуществляться только с пульта управления.

Для четырех действий, выполнения которых мы ожидаем от сумматора, нам понадобятся четыре кода. Коды эти в принципе произвольны. Например, они могут быть такими:

Действие	Код
Загрузить	10h
Сохранить	11h
Сложить	20h
Остановить	FFh

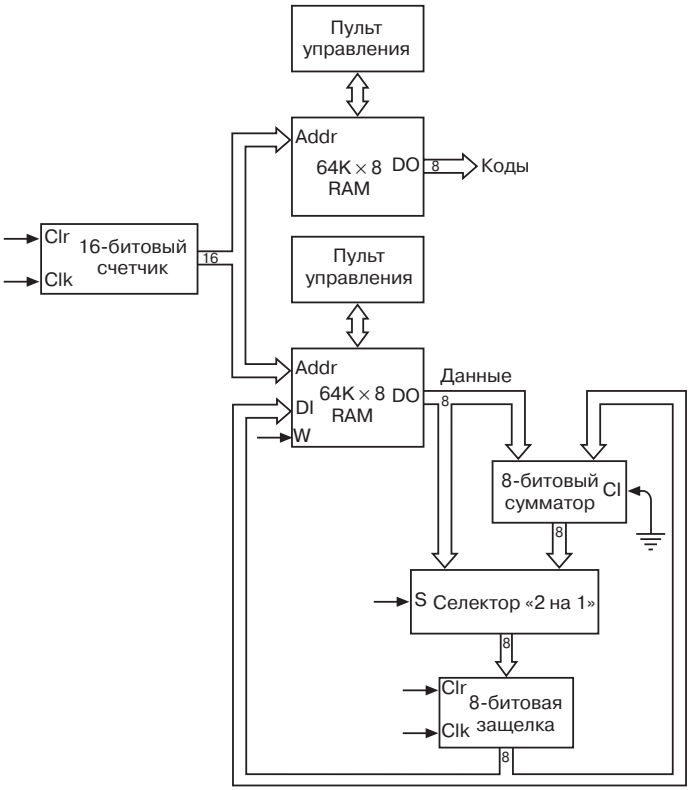
Для выполнения троекратного сложения в нашем примере вы должны с помощью пульта управления записать в массив Коды такие значения:

0000h:	10h	Загрузить
	20h	Сложить
	20h	Сложить
	11h	Сохранить
0004h:	10h	Загрузить
	20h	Сложить
	11h	Сохранить
0007h:	10h	Загрузить
	20h	Сложить
	20h	Сложить
	11h	Сохранить
000Bh:	FFh	Остановить

Сравните содержимое этого массива с массивом Данные, который содержит числа для сложения. Каждому коду в массиве Коды соответствует число из массива Данные, которое нужно загрузить в аккумулятор или сложить с его содержимым, или пустая ячейка, в которую нужно поместить значение из акку-

мулятора. Используемые таким образом численные коды называются *кодами команд* (instruction code), или *кодами операций*. Они «командуют» схеме выполнить определенную «операцию».

Раньше я уже упоминал, что выход 8-битовой ячейки должен быть входом массива Данные. Это необходимо для выполнения команды Сохранить. Необходимо и другое изменение. Поначалу входом 8-битовой защелки был выход 8-битового сумматора. Однако для выполнения команды Загрузить вход 8-битовой защелки иногда должен соединяться с выходом массива Данные. Нам, очевидно, нужен селектор 2 линии на 1. Исправленная схема сумматора выглядит так:



В этой схеме все еще много не хватает, но все 8-битовые каналы между различными компонентами показаны. Адреса для двух массивов RAM выдает 16-битовый счетчик. Выход массива Данные, как обычно, соединен со входом сумматора для выполнения команды Сложить. На вход 8-битовой защелки сигнал подается либо с выхода массива Данные (для выполнения команды Загрузить), либо с выхода сумматора (для выполнения команды Сложить). Вот здесь-то и нужен селектор «2 на 1». Выход защелки, как обычно, возвращается на вход сумматора, но он же является и входом массива Данные (для выполнения команды Сохранить).

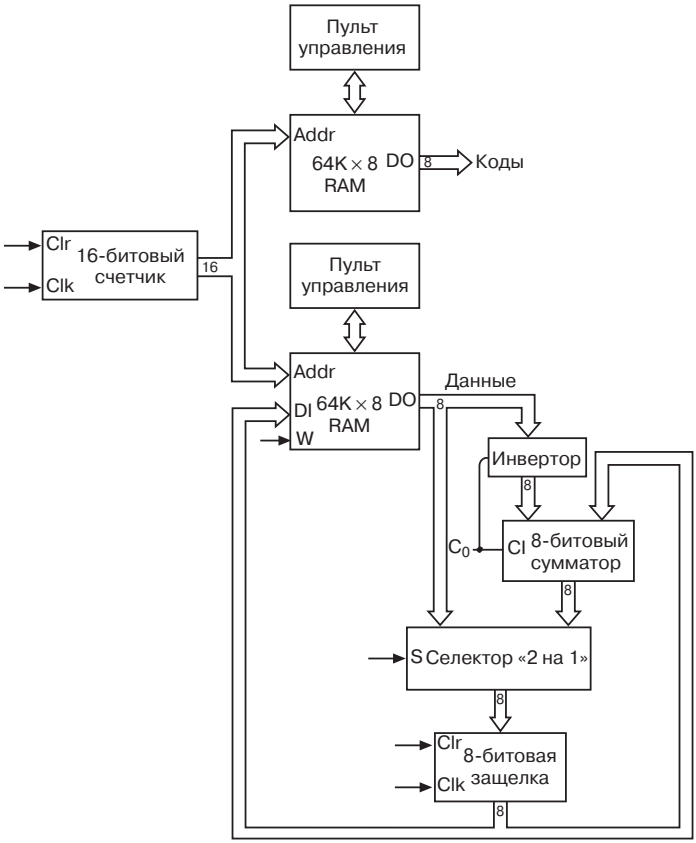
Чего на этой схеме нет, так это сущего пустяка — сигналов, которые руководили бы работой всех компонентов, т. е. *управляющих сигналов* (control signals). К ним относятся входы Clk и Clr 16-битового счетчика, входы Clk и Clr защелки, вход W массива данные и вход S селектора «2 на 1». Некоторые из этих сигналов, очевидно, будут определяться содержимым массива Коды. Например, на вход S селектора «2 на 1» подается 0 (выбрано поступление данных из памяти), если в текущей ячейке массива Коды записана команда Загрузить. Вход W массива Данные обращается в 1, только если в текущей ячейке массива Коды содержится команда Сохранить. Эти управляющие сигналы можно генерировать с помощью различных комбинаций логических вентилей.

Еще немного оборудования и новая команда позволят нам, используя ту же схему, вычитать число из содержимого аккумулятора. Начнем с расширения таблицы кодов команд.

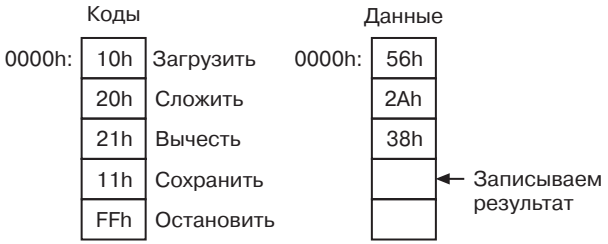
Команда	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Остановить	FFh

Коды команд Сложить и Вычесть отличаются только младшим битом — обозначим его C_0. Если текущий код команды равен 21h, схема осуществляет те же действия, что и при выполнении команды Сложить, только на этот раз данные из памяти

перед поступлением в сумматор инвертируются, и вход сумматора для переноса (CI) устанавливается в 1. Обе эти задачи можно решить с помощью сигнала C_0 при условии, что в конструкцию сумматора добавлен инвертор:



Допустим теперь, что нам нужно сложить числа 56h и 2Ah, а затем вычесть из суммы число 38h. Для решения этой задачи в массивы Данные и Коды записываются такие числа:



После выполнения команды Загрузить в аккумуляторе записано число 56h. После команды Сложить в нем содержится сумма 56h и 2Ah, т. е. 80h. При выполнении команды Вычесть биты следующего по порядку числа в массиве Данные (38h) инвертируются. Инвертированное значение C7h складывается с числом 80h, при этом на вход сумматора для переноса (CI) подается 1.

```
  C7h
 +80h
  +1h
 ----
  48h
```

Окончательный результат равен 48h (в десятичной системе 86 + 42 – 56 = 72).

Настало время обратиться к проблеме недостаточной разрядности сумматора и подключенных к нему устройств. Раньше я говорил лишь о наиболее прямолинейном решении этой проблемы: включить в схему еще один 8-битовый сумматор (и еще одну защелку, и еще один набор лампочек и т. д.), чтобы в итоге получить 16-битовое устройство.

Но есть и более экономичное решение. Рассмотрим в качестве примера сумму 16-разрядных чисел:

```
 76ABh
+232Ch
```

Сложить два этих 16-битовых числа есть то же самое, что по отдельности сложить их младшие байты:

```
  ABh
 +2Ch
 ----
  D7h
```

и старшие байты:

```
  76h
+23h
 ---
  99h
```

Получаем 99D7h. Если мы расположим 16-разрядные числа в памяти таким образом:

число D7h будет сохранено по адресу 0002h, а 99h — по адресу 0005h.

Конечно, такая последовательность действий далеко не всегда будет приводить к желаемому результату. В данном конкретном примере она сработала, но что если придется складывать числа 76ABh и 236Ch? В этом случае сложение двух младших байтов приведет к появлению переноса:

```
  ABh
+6Ch
 ---
 117h
```

Это перенос нужно добавить к сумме старших байтов:

```
  1h
+76h
+23h
 ---
 9Ah
```

чтобы получить окончательный результат — 9A17h.

Можно ли усовершенствовать наш автоматический сумматор, чтобы он корректно складывал 16-разрядные числа? Можно. Для этого нужно лишь *сохранить бит переноса* после

первого сложения, а затем подать его на вход сумматора для переноса при втором сложении. Как сохранить этот бит? Конечно, в 1-битовой защелке, которую мы назовем *защелкой для переноса*.

Для работы с защелкой для переноса нам понадобится еще одна команда. Назовем ее *Сложить с переносом*. При сложении 8-битовых чисел вы используете обычную команду Сложить. Вход CI сумматора равен 0, а выход CO сохраняется в защелке для переноса (хотя в данном случае запоминать его и не нужно).

Суммируя 16-битовые числа, вы сначала складываете их младшие байты с помощью обычной команды Сложить. Вход CI сумматора и в этом случае равен 0, а выход CO сохраняется в защелке для переноса. Для сложения старших байтов вы применяете команду Сложить с переносом. При ее выполнении на вход сумматора для переноса подается содержимое защелки для переноса. Если первое суммирование привело к появлению переноса, он используется во втором суммировании. Если переноса не было, выход защелки для переноса равен 0.

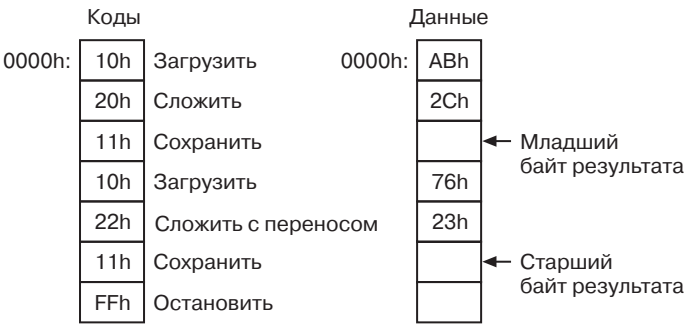
Для вычитания 16-битовых чисел нам понадобится еще одна новая команда — *Вычесть с заимствованием*. Для выполнения обычной команды Вычесть надо инвертировать вычитаемое и установить в 1 вход сумматора для переноса. При этом обычно 1 равен и выход сумматора для переноса, но на это при обычном вычитании можно закрыть глаза. Другое дело — вычитание 16-битовых чисел. Тут значение выхода для переноса надо сохранять в защелке для переноса. При втором вычитании содержимое этой защелки подается на вход CI сумматора.

С учетом двух новых команд мы набрали уже 7 кодов:

Команда	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Остановить	FFh

При выполнении команд Вычесть и Вычесть с заимствованием число, направляемое в сумматор, инвертируется. Выход CO сумматора является входом защелки для переноса. Его значение сохраняется в защелке при выполнении команд Сложить, Вычесть, Сложить с переносом и Вычесть с заимствованием. Вход сумматора для переноса устанавливается в 1 при выполнении команды Вычесть или при выполнении команд Сложить с переносом и Вычесть с заимствованием при условии, что 1 равен выход защелки для переноса.

Не забывайте, что при выполнении команды Сложить с переносом вход сумматора для переноса устанавливается в 1, только если к появлению переноса привело выполнение предыдущей команды Сложить или Сложить с переносом. При сложении многобайтовых чисел всегда следует применять команду Сложить с переносом независимо от того, появится при суммировании перенос или нет. В разобранном выше примере правильная последовательность команд такова:



Она корректно сложит любые 16-разрядные числа.

Всего две новые команды, но как они расширили возможности машины! Мы более не связаны необходимостью работать только с 8-битовыми числами. Многократное применение команды Сложить с переносом позволит нам суммировать 16-битовые, 24-битовые, 32-битовые, 40-битовые значения и т. д. Например, для сложения 32-битовых чисел 7A892BCDh и 65A872FFh нам понадобится одна команда Сложить и три команды Сложить с переносом:

Конечно, заносить эти числа в память не ахти как интересно. Мало того, что придется изрядно пощелкать переключателями, нужно еще и следить за тем, чтобы числа вводились в нужные ячейки. Число 7A892BCDh, например, занимает адреса 0000h, 0003h, 0006h и 0009h, начиная с младшего байта. А результат сложения придется извлекать из ячеек 0002h, 0005h, 0008h и 000Bh.

Кроме того, сумматор в его теперешнем виде не допускает повторного использования результатов вычислений. Скажем, сложили вы три 8-битовых числа, а затем вычли из суммы еще одно 8-битовое число. Это легко сделать командами Загрузить, Сложить, Сложить, Вычесть и Сохранить. Но если вы теперь решите вычесть из суммы другое 8-битовое число, вам придется считать ее заново, так как нигде в памяти она не сохранилась.

Проблема нашего сумматора в том, что в нем доступ к ячейкам массивов Данные и Коды осуществляется одновременно и последовательно, начиная с адреса 0000h. Каждой команде в массиве Коды соответствует ячейка массива Данные, расположенная по такому же адресу. Величину, записанную в массив Данные командой Сохранить, вернуть обратно в аккумулятор нельзя.

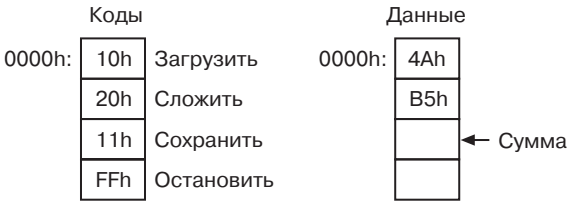
Для решения этой проблемы я намерен внести в сумматор фундаментальное изменение, которое на первый взгляд кажется довольно сложным. Но со временем вы поймете (надеюсь!), какие просторы оно перед нами открывает.

Поехали! Итак, у нас сейчас 7 команд.

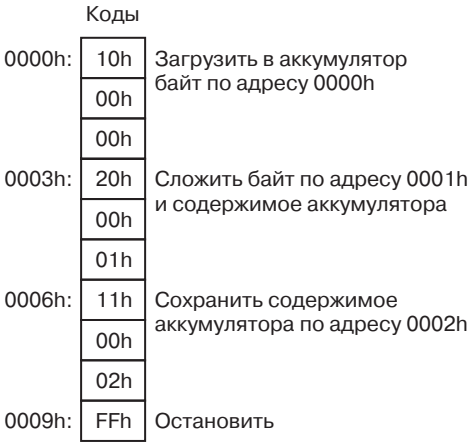
Команда	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Остановить	FFh

Пока каждый из кодов занимает в памяти 1 байт. Но я своей властью выделяю каждому из них, кроме кода Остановить, по 3 байта памяти. В первом будет находиться сам код, а в двух оставшихся — 16-битовый адрес ячейки памяти. В команде Загрузить, например, это будет адрес ячейки, число из которой нужно загрузить в аккумулятор. В командах Сложить, Вычесть, Сложить с переносом и Вычесть с заимствованием по этому адресу будет храниться число, которое нужно прибавить к содержимому аккумулятора или вычесть из него. В команде Сохранить в этих двух байтах будет указываться адрес ячейки, в которую нужно записать содержимое аккумулятора.

Рассмотрим в качестве примера простейшую задачу, которую в состоянии решить наш сумматор, — сложение пары чисел. Для ее решения вы заполняете массивы Данные и Коды такими значениями:



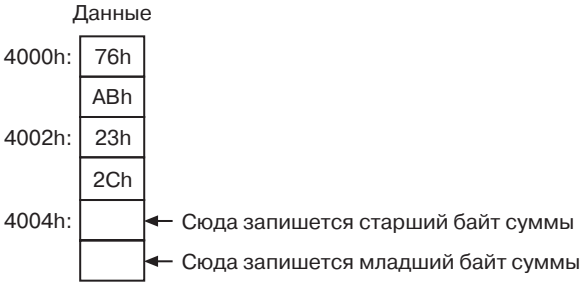
В модернизированном сумматоре все команды, кроме Остановить, занимают по 3 байта:



За каждой командой, кроме Остановить, следуют два байта, указывающие 16-битовый адрес ячейки в массиве Данные. В данном случае это адреса 0000h, 0001h и 0002h.

Чуть раньше я показал вам, как командами Сложить и Сложить с переносом найти сумму двух 16-битовых чисел — 76ABh и 232Ch. При этом младшие байты слагаемых мы должны были записать в ячейки 0000h и 0001h, а старшие — в ячейки 0003h и 0004h. Результат сложения оказывался в ячейках 0002h и 0005h.

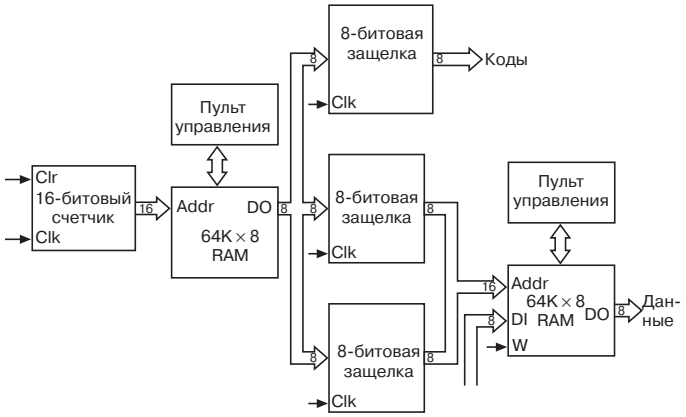
С учетом нашего нововведения слагаемые и результат можно разместить в памяти более понятным способом, причем в той ее области, куда мы еще не забирались.



В таком порядке я разместил все числа исключительно для удобства. На самом деле они могут быть разбросаны по всем 64 К памяти в произвольном порядке. Для сложения чисел, сохраненных в указанных выше ячейках, в массив Коды мы должны ввести такие команды:

Сначала суммируются два младших байта, расположенные по адресам 4001h и 4003h, а их сумма записывается в ячейку 4005h. Старшие (из ячеек 4000h и 4002h) — суммируются командой Сложить с переносом, а сумма сохраняется по адресу 4004h. Теперь мы можем убрать команду Остановить и записать в массив Коды дополнительные команды. Во всех последующих вычислениях можно будет использовать как сами слагаемые, так и результат их суммирования, просто обращаясь к соответствующим ячейкам памяти.

Чтобы воплотить эту систему в жизнь, мы должны подключить к выходам массива Коды три 8-битовых защелки. В каждой будет храниться один из байтов 3-байтовой команды. В первую защелку попадет код команды, во вторую — старший байт адреса, в третью — младший. Выход второй и третьей защелок станет 16-битовым адресом ячейки в массиве Данные:



Процесс извлечения команды из памяти называется *выборкой команды* (instruction fetch). В нашем сумматоре каждая команда занимает по 3 байта и извлекается из памяти побайтово. Выборка одной команды занимает три цикла синхронизирующего сигнала, а полный *командный цикл* — четыре цикла синхронизирующего сигнала. Из-за этого система управляющих сигналов, конечно, усложнится.

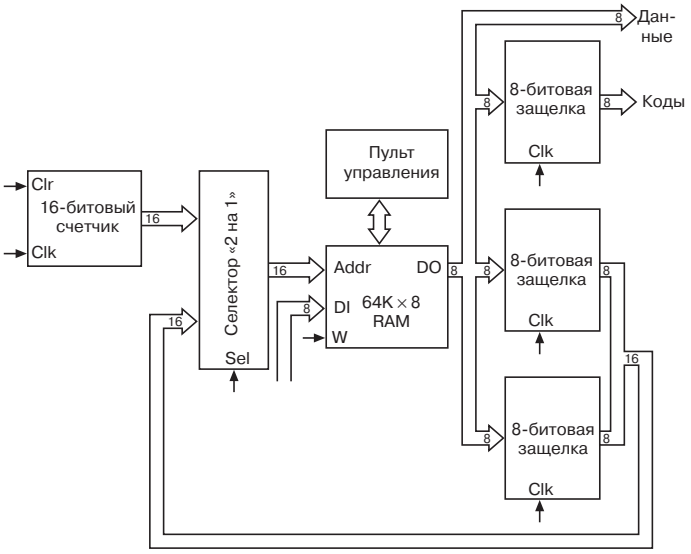
Слова «машина выполнила команду» подразумевают, что в машине реализованы некие действия, зашифрованные кодом команды. Это, разумеется, не означает, что машина *живая* и способна анализировать код, чтобы решить, что делать дальше. Назначение кода команды в том, чтобы генерировать уникальную последовательность управляющих сигналов, которые приводят к определенным последствиям.

Обратите внимание, что расширение возможностей машины отрицательно сказалось на ее быстродействии. При той же частоте вибратора она занимается сложением вчетверо дольше, чем исходная версия сумматора. Это следствие инженерного принципа «Бесплатных обедов не бывает», который означает, что усовершенствование машины в одном отношении приводит к ее ухудшению в чем-то другом.

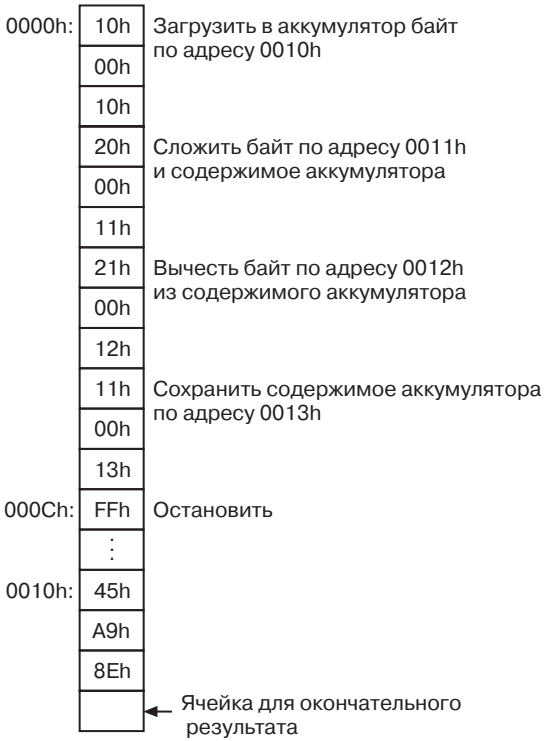
Если бы вы в самом деле собирали эту машину из реле, большую часть схемы составляли бы, очевидно, два 64-килобайтных массива памяти. Конечно, вы можете решить, что для начала вам вполне хватит двух массивов объемом по 1 кб, и

скорее всего так оно и есть. Но нельзя ли как-нибудь обойтись *одним* массивом? Вообще-то можно. Я разбил память на два массива — для данных и для кодов — с единственной целью: сделать архитектуру автоматического сумматора максимально простой и понятной. Но теперь, когда мы увеличили длину команд до 3 байтов, используя второй и третий байты для указания адреса в памяти, надобность в двух отдельных массивах отпала. И коды, и данные можно хранить *в одном и том же* массиве.

Для этого нам понадобится селектор 2 линии на 1, который будет определять способ адресации массива RAM. Один адрес, как и раньше, поступает на селектор с 16-битового счетчика. Выход массива данных по-прежнему подключен к трем защелкам, в которых сохраняется код команды и сопровождающие его два байта адреса. Этот 16-битовый адрес также подается на вход селектора. Когда в защелки записывается адрес, селектор пропускает его на адресный вход массива RAM:



Как мы продвинулись! Теперь можно вводить команды и данные в один и тот же массив RAM. Вот как, например, можно сложить два 8-битовых числа и вычесть из суммы третье:

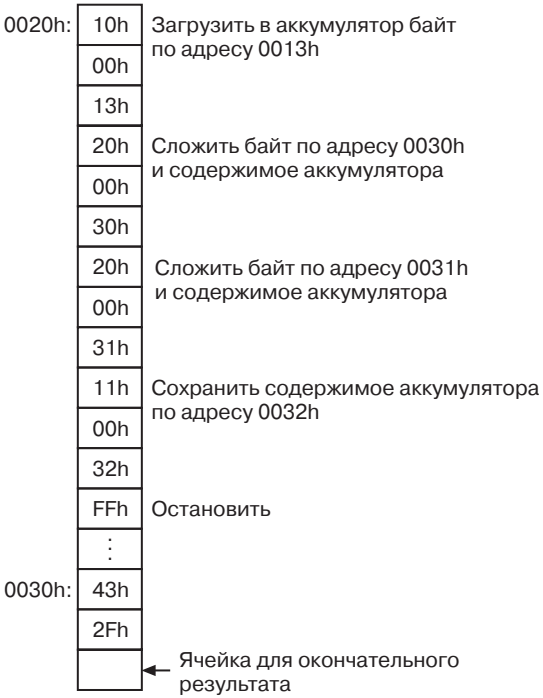


Как обычно, размещение команд начинается с адреса 0000h, так как именно с этого значения начинает считать 16-битовый счетчик. Заключительная команда Остановить находится в ячейке 000Ch. Три обрабатываемых числа и результат можно было разместить в памяти где угодно, кроме, конечно, первых 13 ячеек, занятых командами. Я решил занять для них ячейки, начиная с адреса 0010h.

Теперь допустим, что вам понадобилось прибавить к результату еще два числа. Что ж, можно заменить только что введенные команды новыми. Но не исключено, что вы предпочтете просто продлить уже введенную последовательность, заменив команду Остановить в ячейке по адресу 000Ch новой командой Загрузить. За ней последуют две команды Сложить, команда Сохранить и команда Остановить. Но вот беда — начиная с адреса 0010h, память занята данными. Придется пере-

двинуть их дальше, соответствующим образом отредактировав команды, в которых эти данные используются.

Да-а-а, думаете вы. Наверное, с объединением данных и кодов в одном массиве мы поторопились. Но уверяю вас, что проблема такого рода возникла бы рано или поздно в любом случае, поэтому мы должны ее решить. Попробуем, не сдвигая старые данные, разместить новые команды, начиная с адреса 0020h, а новые данные — начиная с адреса 0030h:



Заметьте: первая команда Загрузить обращается по адресу 0013h, где сохранен результат предыдущего вычисления.

Теперь память заполнена так: по адресу 0000h — команды, по адресу 0010h — данные, по адресу 0020h — опять команды, а по адресу 0030h — снова данные. Мы хотим, чтобы сумматор автоматически выполнил все команды, начав работу с ячейки 0000h.

Прежде всего нужно убрать из ячейки 000Ch команду Остановить. Проблема в том, что записанная по этому адресу

величина будет интерпретироваться как код команды. Та же судьба постигнет и величину в ячейке, отстоящей от 000Ch на 3 байта, т. е. 000Fh, а также и в следующих ячейках с 3-байтовым интервалом — 0012h, 0015h, 0018h, 001Bh и 001Eh. Что если один из этих байтов случайно окажется равен 11h — коду команды Сохранить, а два байта за ним — 00h и 23h? Машина послушно запишет содержимое аккумулятора в ячейку с адресом 0023h. Но эта ячейка содержит важную для нас информацию! Даже если ничего подобного и не случится, нас все равно ждут неприятности. С точки зрения машины, следующая за 001Eh команда расположена в ячейке 0021h, а не 0020h, где она находится на самом деле.

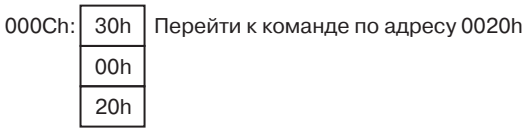
Все согласны с тем, что нельзя просто убрать из ячейки 000Ch команду Остановить и уповать на лучшее?

Мы должны заменить ее на новую команду, а именно команду Перейти, которую уже давно пора включить в наш репертуар.

Команда	Код
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Перейти	30h
Остановить	FFh

Обычно адресация ячеек в автоматическом сумматоре происходит последовательно. Команда Перейти позволяет нарушить это правило. После ее выполнения адресация массива RAM начинается с нового адреса. Команды такого рода называют еще командами *ветвления* (branch).

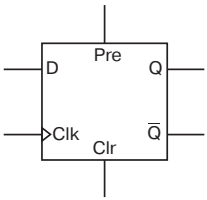
В нашем примере командой Перейти нужно заменить команду Остановить в ячейке 000Ch:



Величина 30h — это код команды Перейти. За ней располагается 16-битовый адрес ячейки, где содержится следующая команда, которую должен выполнять сумматор.

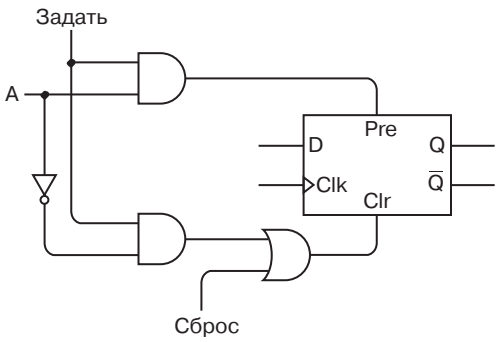
В нашем примере в начале работы сумматор выполняет команду в ячейке 0000h — Загрузить. Затем исполняются команды Сложить, Вычесть и Сохранить. Следующая — Перейти — заставляет сумматор прервать эту последовательность и перейти к команде в ячейке 0020h. Это команда Загрузить, за которой следуют Сложить, Сложить, Сохранить и наконец Остановить.

Команда Перейти действует на 16-битовый счетчик. Всякий раз, когда она встречается в последовательности команд, на выходе счетчика должен появляться стоящий в ней адрес. Это осуществляется с помощью входов Pre и Clr триггера D-типа со срабатыванием по фронту,



который, как вы помните, является основным компонентом счетчика. Обычно входы Pre и Clr равны 0. Если сигнал Pre равен 1, в 1 обращается и сигнал Q. Если 1 равен сигнал Clr, сигнал Q обращается в 0.

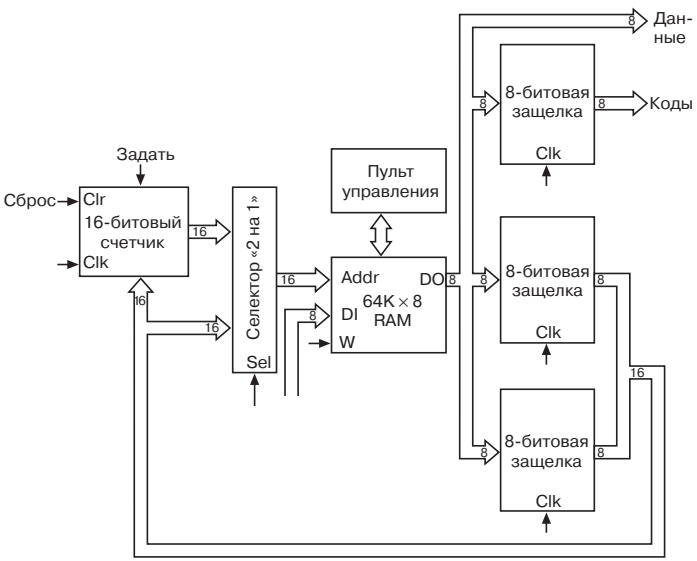
Чтобы записать в триггер новое значение (будем называть его A — адрес), его нужно включить в такую схему:



Как правило, сигнал Задать равен 0. В этом случае вход Pre триггера также равен 0. Если сигнал Сброс не равен 1, нулю равен и сигнал Clr. Вход Сброс нужен, чтобы триггер можно было очистить независимо от значения сигнала Задать. Если сигнал Задать равен 1 и сигнал A тоже равен 1, сигнал Pre обратится в 1, а Clr — в 0. Если сигнал A равен 0, сигнал Pre обратится в 0, а Clr — в 1. Иными словами, значение выхода Q совпадает со значением на входе A.

В 16-битовом счетчике нам понадобится 16 таких схем. Загрузите в счетчик новое значение, и он будет считать, начиная с него.

Других серьезных изменений в схеме пока нет. 16-битовый адрес, переписанный из памяти в защелки, подается как на вход селектора «2 на 1» (откуда он попадает на адресный вход массива RAM), так и на вход 16-битового счетчика.

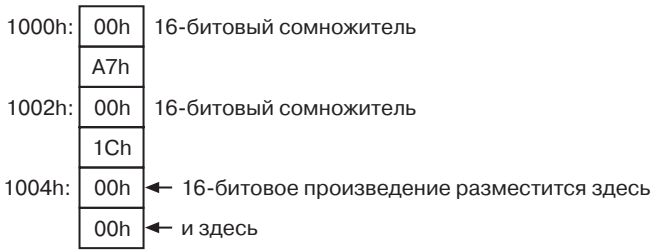


Разумеется, сигнал должен обращаться в 1 лишь при выполнении двух условий: код команды равен 30h и адрес перехода сохранен в защелках.

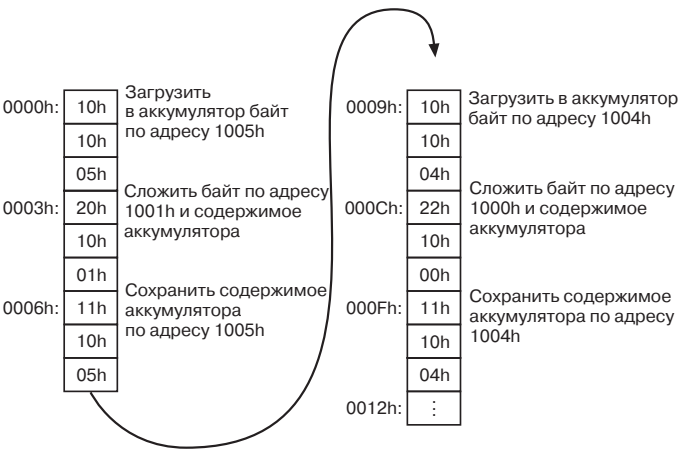
Команда Перейти очень полезна. Но несравненно полезнее команда, осуществляющая переход лишь при выполнении определенного условия. Такой переход называется *условным*

(conditional jump). Чтобы проиллюстрировать его необходимость, зададимся простым вопросом: способен ли наш сумматор *перемножить* два 8-битовых числа, например, A7h и 1Ch?

Произведение двух 8-битовых чисел является 16-битовым числом. Для простоты будем считать, что 16-битовыми являются и сомножители. Для начала решим, куда поместить исходные числа и их произведение.



В десятичном счислении 1Ch равно 28. Нет нужды доказывать вам, что умножение A7h на 1Ch равносильно сложению 28 чисел A7h. Поэтому в ячейках 1004h и 1005h на самом деле будет размещен 16-битовый результат этих многократных сложений. Вот как выглядит набор кодов для размещения по этому адресу первой суммы:



В начале работы ячейки 1004h и 1005h содержат нули. После выполнения шести этих команд размещенное в них 16-бито-

вое значение равно A7h умножить на 1. Чтобы в ячейках 1004h и 1005h оказалось нужное значение, ту же последовательность кодов следует исполнить еще 27 раз. Достичь этого можно, скопировав последовательность 27 раз, начиная с адреса 0012h, или же разместить по этому адресу команду Остановить и 28 раз нажать кнопку Сброс.

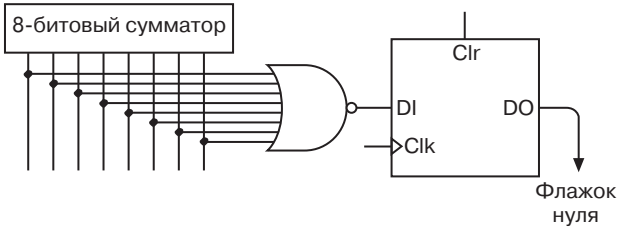
Конечно, и тот, и другой вариант на практике использовать неудобно. В обоих случаях от вас требуется некое действие: ввести набор кодов или нажать кнопку Сброс, причем ровно столько раз, на сколько нужно умножить число. Один-два раза — куда ни шло, но перемножать числа таким способом всю жизнь охотников найдется немного.

А что если разместить по адресу 0012h команду Перейти? После ее выполнения счетчик снова начнет счет с 0000h:

0012h:	30h	Перейти к команде по адресу 0000h
	00h	
	00h	

Вроде бы то, что нужно. После первого выполнения кодов ячейки 1004h и 1005h содержат значение, равное произведению A7h и 1. Затем команда Перейти осуществляет переход к началу фрагмента, и после следующего его выполнения в ячейках 1004h и 1005h содержится число A7h, умноженное на 2. В конце концов машина доберется и до произведения A7h и 1Ch, но, увы, в нужный момент не остановится!

Мы хотим, чтобы команда Перейти срабатывала только нужное число раз, т. е. совершала условный переход. Добиться этого на самом деле не так сложно. Для начала добавим в схему 1-битовую защелку, похожую на защелку для переноса, и назовем ее *нулевой защелкой*, так как ее содержимое будет равняться 1, только если все выходы 8-битового сумматора равны 0.



Выход 8-входового вентиля ИЛИ-НЕ равен 1, только если все входы равны 0. Как и в защелке для переноса, синхронизирующий сигнал Clk в нулевой защелке вызывает запоминание входного сигнала только при выполнении команд Сложить, Вычесть, Сложить с переносом и Вычесть с заимствованием. Величина, хранимая в нулевой защелке, называется *флажком нуля* (Zero flag). Не запутайтесь: флажок нуля равен 1, если все выходы сумматора равны 0; флажок нуля равен 0, если на некоторых выходах сумматора имеются 1.

Защелка для переноса и нулевая защелка позволяют расширить набор доступных действий на 4 команды.

Команда	**Код**
Загрузить	10h
Сохранить	11h
Сложить	20h
Вычесть	21h
Сложить с переносом	22h
Вычесть с заимствованием	23h
Перейти	30h
Перейти если 0	31h
Перейти если перенос	32h
Перейти если не 0	33h
Перейти если не перенос	34h
Остановить	FFh

Например, команда *Перейти если не 0* вызывает переход на заданный адрес, только если выход нулевой защелки равен 0. Иначе говоря, перехода *не* будет, если результат последней команды Сложить, Вычесть, Сложить с переносом или Вычесть с заимствованием равен 0. Для реализации этой команды достаточно расширить набор управляющих сигналов, применяемых для обычной команды Перейти. При выполнении команды *Перейти если 0* сигнал Задать 16-битового счетчика устанавливается в 1, только если флажок нуля равен 0.

Имеющегося в нашем распоряжении набора команд хватит для написания кодов, умножающих одно число на другое:

0012h:	10h	Загрузить в аккумулятор байт по адресу 1003h
	10h	
	03h	
0015h:	20h	Сложить байт по адресу 001Eh и содержимое аккумулятора
	00h	
	1Eh	
0018h:	11h	Сохранить содержимое аккумулятора по адресу 1003h
	10h	
	03h	
001Bh:	33h	Перейти к команде по адресу 0000h, если флажок нуля не равен 1
	00h	
	00h	
001Eh:	FFh	Остановить

Как мы уже установили, к началу этого фрагмента в ячейках 0004h и 0005h содержится 16-битовое число A7h, умноженное на 1. Первая команда фрагмента загружает в аккумулятор число из ячейки 1003h, т. е. 1Ch. Оно складывается с содержимым ячейки 001Eh. Да, в ячейке 001Eh записан код команды Остановить, но это не мешает ему быть и просто обычным числом. Сложить FFh и 1Ch — это все равно что вычесть из 1Ch единицу. В результате получается 1Bh. Это не 0, поэтому флажок нуля равен 0. Число 1Bh записывается обратно в ячейку 1003h. Далее следует команда Перейти если не 0. Поскольку результат действительно не равен 0, условие перехода соблюдается, и следующей будет выполнена команда по адресу 0000h.

Не забывайте: команда Сохранить на значение флажка нуля не влияет. Он задается лишь при выполнении команд Сложить, Вычесть, Сложить с переносом и Вычесть с заимствованием и сохраняет значение до следующего выполнения одной из этих команд.

При втором выполнении фрагмента ячейки 0004h и 0005h содержат произведение числа A7h на 2. Число 1Bh прибавляется к FFh, и получается 1Ah. Это не 0, поэтому вновь происходит возвращение к началу.

На 28-й раз ячейки 0004h и 0005h содержат произведение чисел A7h и 1Ch. По адресу 1003h хранится число 1. Его сумма с числом FFh равна 0, и флажок нуля наконец-то устанавливается в 1! Условие команды *Перейти если не 0* не выполняется, перехода нет, и следующей командой становится Остановить. Есть!

Наступает великий миг: я утверждаю, что собранное нами устройство может с полным правом именоваться *компьютером*! Очень примитивным, но компьютером. Его ключевое свойство — наличие команды условного перехода. Именно возможность управляемых *циклических* процедур отличает компьютер от калькулятора. Я только что продемонстрировал, как наша машина с помощью условного перехода умножила одно число на другое. Подобным образом можно осуществить и деление. Далее, действие нашей машины не ограничено 8-битовыми числами. Она способна складывать, вычитать, умножать и делить 16-битовые, 24-битовые, 32-битовые числа и числа большей разрядности. А значит, ей подвластны квадратные корни, логарифмы и тригонометрические функции.

Раз уж мы собрали компьютер, пора начать говорить о нем, как о компьютере.

Разработанное нами устройство относится к *цифровым* (digital) компьютерам, поскольку работает с дискретными цифрами. Это отличает его от *аналоговых* (analog) компьютеров, которые существовали некоторое время назад, а сейчас практически исчезли. Напомню: дискретными называются данные, способные принимать лишь набор фиксированных значений. Аналоговые данные непрерывны и могут принимать любое значение из заданного интервала.

Цифровой компьютер состоит из 4 основных компонентов: *процессора*, *памяти*, минимум одного *устройства ввода* и минимум одного *устройства вывода*. В нашей машине память состоит из 64-килобайтового массива RAM. Роль устройств ввода и вывода играют ряды переключателей и лампочек на пульте управления памятью. Они позволяют нам (исполняющим в этом шоу роли людей) вводить в память числа и просматривать результаты вычислений.

Все остальное — процессор (processor) или, как его еще называют, *центральное процессорное устройство* (Central Processor Unit) — ЦПУ (CPU). Процессор часто называют *мозгом* компьютера, но я стараюсь не прибегать к подобным срав-

нениям, поскольку лично мне собранная схема мозг не напоминает. В наши дни часто приходится слышать слово *микропроцессор*. Это тот же процессор, только маленький-маленький, благодаря изобретениям, о которых я расскажу в главе 18. К нашему релейному детищу приставка «микро» вряд ли подходит!

Процессор, который мы собрали, является *8-разрядным*. 8 битам равна ширина аккумулятора и большинства каналов данных. Из 16 битов состоит только адрес в массиве RAM. Если бы он был 8-битовым, мы могли бы адресовать не 65 536, а лишь 256 байтов — тут не разгуляешься.

Процессор тоже состоит из нескольких компонентов. Я уже неоднократно упоминал *аккумулятор* — защелку для хранения чисел внутри процессора. 8-битовый инвертор и 8-битовый сумматор вместе составляют *арифметико-логическое устройство* (Arithmetic Logic Unit) — АЛУ (ALU). В нашем случае оно способно выполнять только арифметические действия, а именно сложение и вычитание. В несколько более сложных компьютерах (с которыми мы еще познакомимся) арифметико-логическое устройство выполняет также логические операции — И, ИЛИ и исключающее ИЛИ. 16-битовый счетчик называется *программным счетчиком* (program counter).

Наш компьютер состоит из реле, проводов, переключателей и лампочек. Все вместе они называются *аппаратным обеспечением* (hardware) компьютера. Команды и другие числа, которые мы вводили в память, составляют его *программное обеспечение* (software).

В разговорах о компьютерах программное обеспечение обычно называют просто *программами*, а процесс их создания — *программированием*. Придумывая набор команд, которые позволили бы нашему компьютеру умножать одно число на другое, именно программированием я и занимался.

Обычно в компьютерных программах различают *код* (code), т. е. собственно команды, и *данные* (data), т. е. числа, с которыми работает код. Иногда это различие не столь очевидно. Например, в разобранном нами примере код команды Остановить одновременно играет роль числа –1. Коды команд (например, 10h для команды Загрузить), на основании которых процессор выполняет то или иное действие, называются машинными кодами или машинным языком. Слово «язык» вполн-

не оправданно, так как именно с помощью кодов человек «объясняет» машине, что она должна сделать.

До сих пор я называл команды, выполняемые машиной, довольно длинными словами и целыми фразами, например, Сложить с переносом. Но на практике для обозначения команд используют двух- трехбуквенные английские сокращения — мнемонические коды, — записываемые прописными буквами. Вот как может выглядеть набор мнемокодов для нашего компьютера.

Команда	**Английское название**	**Код**	**Мнемокод**
Загрузить	Load	10h	LOD
Сохранить	Store	11h	STO
Сложить	Add	20h	ADD
Вычесть	Subtract	21h	SUB
Сложить с переносом	Add with Carry	22h	ADC
Вычесть с заимствованием	Subtract with Borrow	23h	SBB
Перейти	Jump	30h	JMP
Перейти если 0	Jump If Zero	31h	JZ
Перейти если перенос	Jump If Carry	32h	JC
Перейти если не 0	Jump If Not Zero	33h	JNZ
Перейти если не перенос	Jump If Not Carry	34h	JNC
Остановить	Halt	FFh	HLT

Удобство мнемокодов станет более очевидным, если мы присовокупим к ним еще пару обозначений. Например, условимся писать вместо многословной инструкцию «Загрузить в аккумулятор байт по адресу 1003h» короткое выражение:

```
LOD A,[1003h]
```

Обозначения A и [1003h], стоящие справа от мнемокода, называются *аргументами*. Аргументы конкретизируют выполнение команды. Например, в команде Загрузить они указывают, *куда* (аргумент слева) следует загрузить данные (A означает аккумулятор) и *откуда* (аргумент справа) их следует извлечь. Квадратные скобки означают, что в аккумулятор надлежит загрузить не число 1003h, а содержимое ячейки памяти по адресу 1003h.

Подобным же образом инструкцию «Сложить байт по адресу 001Eh и содержимое аккумулятора» можно сократить до:

```
ADD A,[001Eh]
```

а инструкцию «Сохранить содержимое аккумулятора по адресу 1003h» — до:

```
STO [1003h],A
```

Заметьте: в команде STO мы также слева указываем, куда сохраняются данные, и справа — откуда они извлекаются: содержимое аккумулятора записывается в ячейку 1003h. Наконец, команда «Перейти к команде по адресу 0000h, если флажок нуля не равен 1» превращается в краткое указание:

```
JNZ 0000h
```

Здесь квадратные скобки не применяются, поскольку переход всегда осуществляется только по адресу. Числом аргумент этой команды быть не может.

С учетом всех обозначений команды удобно записывать в столбик, причем обводить их рамками уже не нужно; они и без этого прекрасно читаются. Адрес, по которому хранится команда (точнее, ее первый байт), мы будем указывать шестнадцатеричными числом с двоеточием:

```
0000h: LOD A,[1005h]
```

А вот так будут обозначаться записанные в память данные:

```
1000h: 00h, A7h
1002h: 00h, 1Ch
1004h: 00h, 00h
```

Перечисление байтов через запятую означает, что первый байт записан в ячейке, адрес которой указан слева, а второй байт —

в следующей за ней ячейке. Эти три строчки можно было бы заменить одной:

```
1000h: 00h, A7h, 00h, 1Ch, 00h, 00h
```

Целиком программа для умножения выглядит так:

```
0000h:    LOD A,[1005h]
          ADD A,[1001h]
          STO [1005h],A

          LOD A,[1004h]
          ADC A,[1000h]
          STO [1004h],A

          LOD A,[1003h]
          ADD A,[001Eh]
          STO [1003h],A

          JNZ 0000h

001Eh:    HLT

1000h:    00h, A7h
1002h:    00h, 1Ch
1004h:    00h, 00h
```

Пробелы и пустые строки расставлены здесь с единственной целью — сделать программу более понятной.

При написании кодов численные значения адресов лучше не использовать, так как они могут измениться. Если вы, например, решите хранить данные, начиная с адреса 2000h, а не 1000h, вам придется переписывать многие команды. Для обозначения ячеек памяти предпочтительнее пользоваться *метками* (labels):

```
BEGIN:    LOD A,[RESULT + 1]
          ADD A,[NUM1 + 1]
          STO [RESULT + 1],A

          LOD A,[RESULT]
          ADC A,[NUM1]
          STO [RESULT],A
```

```
          LOD A,[NUM2 + 1]
          ADD A,[NEG1]
          STO [NUM2 + 1],A

          JNZ BEGIN

NEG1:     HLT

NUM1:     00h, A7h
NUM2:     00h, 1Ch
RESULT:   00h, 00h
```

Метки NUM1, NUM2 и RESULT ссылаются на ячейки, хранящие по 2 байта. В приведенной выше программе выражения NUM1 + 1, NUM2 + 1 и RESULT + 1 ссылаются на второй байт соответствующей метки. Обратите внимание на метку NEG1 у команды HLT: она означает «negative one», т. е. «минус один».

Наконец, на случай, если вы забудете, зачем нужна та или иная команда, в программе предусмотрены *комментарии*. Они пишутся на простом человеческом языке и отделяются от кода точкой с запятой:

```
BEGIN:    LOD A,[RESULT + 1]
          ADD A,[NUM1 + 1]       ; Сложить младшие байты
          STO [RESULT + 1],A

          LOD A,[RESULT]
          ADC A,[NUM1]           ; Сложить старшие байты
          STO [RESULT],A

          LOD A,[NUM2 + 1]
          ADD A,[NEG1]           ; Уменьшить число на 1
          STO [NUM2 + 1],A

          JNZ BEGIN

NEG1:     HLT

NUM1:     00h, A7h
NUM2:     00h, 1Ch
RESULT:   00h, 00h
```

Язык программирования такого типа называется *языком ассемблера* (assembly language). Это своеобразный компромисс между бессловесными машинными кодами и многословными командами на «человеческом» языке, дополненный текстовыми ссылками на ячейки памяти. Язык ассемблера и машинные коды часто путают, поскольку это просто различные способы представления одной и той же программы. Каждому оператору языка ассемблера соответствует определенный числовой код.

Если бы вам пришлось писать программу для компьютера, разработанного в этой главе, вы, наверное, предпочли бы сначала написать ее на языке ассемблера (и на бумаге). Затем, когда вы решили бы, что программа в основном написана и готова к тестированию, вам пришлось бы *вручную ассемблировать* ее, т. е. переводить операторы ассемблера в соответствующие числовые машинные коды, опять же на бумаге. Далее машинные коды вводились бы в память компьютера с помощью переключателей, и программу можно было бы *запускать*, т. е. указать компьютеру выполнить все составляющие ее команды.

После знакомства с концепцией программирования самое время поговорить об ошибках. Совершать их при написании программ, особенно, в машинных кодах, очень легко. Нехорошо, конечно, когда в программе используются неверные числа. Но что если с ошибкой вы введете код команды, например, наберете с помощью переключателей 11h (код команды Сохранить) вместо 10h (код команды Загрузить)? Компьютер не только не выполнит нужное действие, т. е. не загрузит число в аккумулятор, но и выполнит ненужное — запишет поверх этого числа теперешнее содержимое аккумулятора.

Последствия некоторых ошибок бывают непредсказуемы, например, когда по адресу, указанному в команде Перейти, отсутствует корректный код команды или когда командой Сохранить в ячейку с кодом команды по ошибке записывается число. Произойти в таких случаях может (и происходит) все что угодно.

Ошибка имеется даже в моей программе для умножения. Если вы запустите ее во второй раз, она умножит A7h на 256 и запишет результат поверх уже посчитанного произведения. Причина понятна: после первого выполнения программы в ячейке 1003h записан 0. Запустите программу еще раз, и к нему будет прибавлено число FFh. Результат не равен 0, поэтому выполнение программы продолжится.

Мы убедились, что наша машина умеет умножать, а значит, и делить. Я заявил также, что эти простейшие операции позволяют находить корни, логарифмы и тригонометрические функции. На аппаратную часть компьютера возлагаются три обязанности: сложение, вычитание и условный переход. Как говорят программисты: «Для остального напишем программу».

Конечно, программа эта может оказаться очень сложной. Об *алгоритмах* решения конкретных вычислительных задач написано множество книг. Но нам пока рано задумываться об этом. Мы ведь до сих пор ограничивались целыми числами, не взяв на себя смелость подумать о дробях. Но мы еще вернемся к ним — в главе 23.

Я несколько раз говорил о том, что все приборы, нужные для сборки компьютера, изобретены более 100 лет назад. Но это, конечно, не означает, что уже тогда создание подобного устройства было возможно. Многие принципы, рассмотренные нами в этой главе, даже в эпоху появления первых релейных компьютеров (в 1930-е годы) все еще не были известны. Додуматься до них люди смогли лишь в середине XX в. До того времени многие изобретатели, например, упорно пытались создать устройства, работающие не с двоичными, а с десятичными числами. Не сразу оформилась и идея совместного хранения кода и данных. Часто предпочитали вводить программу в компьютер по мере ее выполнения с помощью перфоленты. С памятью на заре компьютерной эры вообще были большие трудности. Сборка 64 К памяти из 5 млн. реле сто лет назад была такой же абсурдной задачей, как и сейчас.

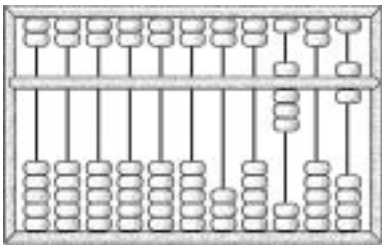
Настало время немного оторваться от практических упражнений и коротко познакомиться с историей появления счетных устройств и машин. Не исключено, что в конце концов окажется, что мы напрасно возились с нашим релейным компьютером. В главе 12 я уже говорил, что на смену реле пришли вакуумные лампы и транзисторы. Мы узнаем также, что за это время люди научились делать разработанные нами процессор и память совсем-совсем миниатюрными.

Глава 18

От счетов к микросхемам

На протяжении писаной истории человечество изобрело немало хитроумных приспособлений и машин в непрерывном стремлении хоть немного облегчить математические вычисления. Хотя человеческий род с рождения наделен способностью к счету, в вычислениях нам требуется помощь. Зачастую мы способны формулировать такие сложные задачи, что решить их самостоятельно нам не по силам.

Развитие систем счисления можно рассматривать как первое средство для облегчения управления недвижимостью и собственностью. По-видимому, для счета многие культуры, в том числе древние греки и американские индейцы, использовали камешки. В Европе это привело к появлению счетных досок, а на востоке — к изобретению счетов.



Хотя появление счетов обычно связывают с азиатскими культурами, торговцы в Китае пользовались ими уже в 1200 г. до н. э.

Умножение и деление никому и никогда не доставляли истинного удовольствия, но внести свой вклад в решение этой проблемы сумели немногие. В числе этих немногих — шотландский математик Джон Непер (1550–1617), который как раз для упрощения этих операций изобрел логарифмы. Произведение двух чисел попросту равно сумме их логарифмов. Поэтому, чтобы перемножить два числа, вы находите в таблице их логарифмы, складываете их, а затем ищете в таблице число, логарифм которого равен полученной сумме. Это и есть искомое произведение.

На протяжении последующих 400 лет одни лучшие умы человечества строили таблицы логарифмов, другие — изобретали приспособления, которые позволяли бы обходиться без этих таблиц. Долгая история логарифмической линейки началась со счетной линейки, изобретенной Эдмундом Гюнтером (1581–1626), усовершенствованной Вильямом Отредом (1574–1660), и практически закончилась в 1976 г., когда компания Keuffel & Esser презентовала последнюю изготовленную ею логарифмическую линейку Смитсоновскому институту в Вашингтоне (США).

Непер изобрел и другое приспособление для умножения, состоявшее из рядов чисел, выгравированных на кости, роге или слоновой кости. Потому это приспособление иногда называли *костями Непера*. Первая механическая счетная машина, созданная приблизительно в 1620 г. Вильгельмом Шикардом (1592–1635), была в некоторой степени автоматизированным вариантом костей Непера. Примерно в то же время появились и механические калькуляторы, состоящие из зацепляющихся колес и рычагов. Из наиболее известных создателей механических счетных устройств упомянем философов и математиков Блеза Паскаля (1623–1662) и Готфрида Вильгельма Лейбница (1646–1716).

Вы, без сомнения, помните, сколько хлопот доставила нам реализация переноса бита в следующий разряд как в исходном 8-разрядном сумматоре, так и в компьютере, который (среди прочего) автоматически складывал числа шириной более 8 битов. На первый взгляд, перенос кажется лишь незначительным нюансом процесса сложения, но в суммирующих маши-

нах он в действительности является главной проблемой. Создав суммирующую машину, которая делает все, кроме переноса, вы не создали почти ничего!

Способность переносить числа из разряда в разряд — главный критерий в оценке счетных устройств. Так, в машине Паскаля перенос осуществлялся так, что машина не могла вычитать. Чтобы вычесть одно число из другого приходилось прибавлять дополнение до девяти примерно так, как я описал в главе 13. «Нормальные» механические калькуляторы, которые можно было эффективно использовать в повседневной работе, появились лишь в конце XIX в.

Приблизительно в 1801 г. появилось занятное изобретение, которое позже сказалось на истории вычислительной техники и оказало громадное влияние на текстильную промышленность, — автоматический ткацкий станок Жозефа Мари Жаккарда (1752–1834). В «машине Жаккарда» узор, вышиваемый на ткани, задавался с помощью металлических пластин с пробитыми отверстиями. Высшим достижением Жаккарда стал его черно-белый автопортрет, вытканный на шелке и закодированный на 10 000 пластин.

В XVIII в. (и вплоть до 40-х годов XX в.) *компьютером* (computer) назывался человек, нанятый специально для проведения вычислений. Спрос на таблицы логарифмов был высок, а для проведения навигационных и астрономических расчетов требовались еще и таблицы тригонометрических функций. Человек, желавший опубликовать новый набор таблиц, нанимал компьютеров, организовывал их работу, а потом собирал воедино все результаты. Естественно, на всех этапах работы — от вычислений до набора таблиц в типографии — в них вкрадывались ошибки.

Стремление избавиться от ошибок в математических таблицах побудило к работе Чарльза Бэббиджа (1791–1871), британского математика, жившего почти в то же самое время, что и Сэмюэль Морзе.

В то время при создании таблиц логарифмов на самом деле лога-

рифмы всех чисел в таблице не вычислялись (на это ушло бы слишком много времени). Их рассчитывали лишь для некоторых чисел, а логарифмы остальных чисел вычислялись с помощью интерполяции. Расчеты при этом оказывались довольно простыми и выполнялись с помощью *разностей* (differences).

Около 1820 г. Бэббидж решил, что способен спроектировать и построить машину, которая автоматизирует создание математических таблиц вплоть до их верстки, полностью устранив все ошибки. Задуманная им Машина Разностей (Difference Engine) была в действительности большой механической суммирующей машиной. Многозначные десятичные числа представлялись зубчатыми колесами, каждое из которых могло находиться в одном из десяти положений. Отрицательные числа обрабатывались с помощью дополнений до 10. Хотя первые модели показали перспективность проекта Бэббиджа и он даже получил какие-то деньги от британского правительства (впрочем, недостаточные), Машина Разностей так и не была построена. В 1833 г. Бэббидж прекратил работу над ней.

К этому времени у него появилась лучшая идея. Он назвал этот проект Аналитической Машиной (Analytical Engine) и проработал над ней до самой смерти, постоянно перерабатывая конструкцию (в «железе» были созданы лишь небольшие модели и отдельные компоненты). В конструкцию Бэббиджа входило *хранилище* (концептуально подобное компьютерной памяти) и «*мельница*» (арифметическое устройство). Умножение выполнялось с помощью повторных сложений, а деление — с помощью повторных вычитаний. Самый занимательный аспект проекта Аналитической Машины в том, что ее можно было программировать с помощью карточек, идею которых Бэббидж позаимствовал у Жаккарда. Августа Ада Байрон графиня Лавлейс (1815–1852) высказалась об этом так (в заметках к ее переводу статьи, написанной об Аналитической Машине Бэббиджа итальянским математиком): «Можно сказать, что Аналитическая Машина вышивает алгебраические узоры, подобно тому, как станок Жаккарда вышивает цветы и листья».

По-видимому, Бэббидж был первым человеком, осознавшим важность условных переходов в программировании. Снова процитируем Аду Байрон: «*Цикл* операций нужно понимать как любой *набор операций*, который выполняется *более одного*

раза. Цикл является таковым независимо от того, повторяется он *дважды* или неопределенное число раз; поскольку циклом его делает *сам факт повторения*. Во многих задачах встречаются *вложенные группы* одного или нескольких циклов, т. е. *циклы циклов*».

Хотя в 1853 г. машина разностей была наконец построена Георгом и Эдуардом Шуцами (отцом и сыном), об изобретениях Бэббиджа на многие годы позабыли и вспомнили опять лишь в 1930-е годы, когда люди начали интересоваться корнями современной вычислительной техники. К тому времени все, сделанное Бэббиджем, уже устарело. Ему нечего было предложить компьютерному инженеру XX в., кроме замечательного предвидения автоматизации.

Вторым верстовым столбом истории вычислительной техники мы обязаны Конституции Соединенных Штатов Америки. Среди прочего в ней говорится, что каждые 10 лет в США проводится перепись населения. В перепись 1880 г. включалась информация о возрасте, поле и национальности. На обработку и обобщение данных ушло 7 лет.

Опасаясь, что обработка переписи 1890 г. займет больше 10 лет, Бюро переписей исследовало возможность ее автоматизации и остановилось на разработке Германа Холлерита (Herman Hollerith) (1860–1929), который участвовал в обработке переписи 1880 г. в качестве статистика.

Суть проекта Холлерита заключалась в использовании картонных карточек 6 5/8 × 3 1/4 дюйма (Холлерит скорее всего не слышал о программировании с помощью карточек Аналитической Машины Бэббиджа, но почти несомненно знал об использовании пластин в ткацком станке Жаккарда). Для отверстий на каждой карточке было предусмотрено 288 позиций — 24 столбца по 12 позиций в каждом. Эти позиции обозначали определенные характеристики человека, опрашиваемого в ходе переписи. Получив ответ на вопрос, переписчик пробивал в соответствующей позиции карточки отверстие диаметром в четверть дюйма.

Эта книга, вероятно, уже настолько приучила вас мыслить в терминах двоичных кодов, что вы сразу сообразили: карточка с 288 позициями может хранить 288 битов информации. Однако до этого не доходило.

Например, если бы в переписи применялась чисто двоичная система кодирования, на карточке для обозначения пола нужна была бы всего одна позиция (пробита — мужской, не пробита — женский или наоборот). Но на карточках Холлерита для пола было предусмотрено две позиции. В одной отверстие пробивали для мужчин, в другой — для женщин. Два отверстия пробивалось для записи возраста опрашиваемого. Первое означало пятилетний интервал: от 0 до 4, от 5 до 9, от 10 до 14 и т. д. Второе отверстие пробивалось в одной из пяти возможных позиций и указывало точный возраст в пределах 5-летнего интервала. Всего на карточке для кодирования возраста использовалось 28 позиций. В чисто двоичной системе для кодирования любого возраста от 0 до 127 лет требуется всего 7 позиций.

Вряд ли стоит обвинять Холлерита в том, что в кодировании информации он не «дошел» до двоичной системы. Нельзя требовать от переписчиков 1890 г. умения преобразовывать возраст в двоичный формат. Карточки с отверстиями не могли быть двоичными и по практическим причинам. При двоичном кодировании возможны ситуации, когда пробиты должны быть *все* (или почти все) отверстия. Весьма вероятно, что карточка при этом превратится в лохмотья.

Данные переписи должны обобщаться в виде, удобном для вычислений, т. е. в виде *таблиц*. Важно, конечно, знать, сколько человек живет в том или ином районе, но иногда нужны сведения и о том, как население распределяется по возрастам. Для проведения таких расчетов Холлерит создал табулирующую машину, в которой сочетались автоматизация и ручное управление. Оператор прижимал к каждой карточке пресс с 288-ю штырьками на пружинах. В тех позициях, где на карточке было отверстие, штырек погружался в кювету с ртутью, замыкая электрическую цепь, которая приводила в действие электромагнит. Он в свою очередь увеличивал на единицу значение десятичного счетчика.

Электромагниты Холлерит применил и в машине, сортировавшей карточки. Допустим, вы хотите статистически проана-

лизировать возрасты представителей различных профессий. Для этого нужно сначала отсортировать карточки по профессиям, а потом в каждой группе провести анализ возрастов. В сортирующей машине использовался такой же пресс, как и в табуляторе, а электромагниты применялись для того, чтобы открывать задвижки на одном из 26 отделений. Оператор опускал карточку в отделение и вручную закрывал задвижку.

Эксперимент по автоматизации переписи 1890 г. закончился полным успехом. Всего было обработано 62 000 000 карточек. Данных в них содержалось вдвое больше, чем в переписи 1880 г., а времени на обработку ушло втрое меньше. О Холлерите и его изобретениях заговорили во всем мире. В 1895 г. он даже ездил в Россию и продал свое оборудование для использования в первой российской переписи, проведенной в 1897 г.

Герман Холлерит положил начало длинной цепи событий. В 1896 г. он основал компанию Tabulated Machine, которая сдавала напрокат и продавала оборудование для работы с карточками. В 1911 г. после пары слияний она превратилась в компанию Computing-Tabulating-Recording (C-T-R). С 1915 г. президентом компании C-T-R был Томас Ватсон (Thomas Watson) (1874–1956). В 1924 г. он переименовал ее в корпорацию International Business Machines, или просто IBM.

К 1928 г. карточки переписи 1890 г. окончательно превратились в знаменитые перфокарты IBM с 80 столбцами и 12 строками. Они активно применялись в течение 50 лет и даже на закате своей эпохи иногда назывались *картами Холлерита*. К ним я еще вернусь в главах 20, 21 и 24.

Нам пора перебираться в XX век, но мне не хотелось бы покидать век XIX, не убедившись, что у вас осталось правильное впечатление об этой эпохе. По очевидным причинам в этой книге я в основном уделил внимание изобретениям, цифровым по своей природе: телеграфу, азбуке Брайля, машинам Бэббиджа и картам Холлерита. Имея дело только с цифровыми понятиями и устройствами, легко впасть в иллюзию, что весь мир должен быть исключительно цифровым. Но XIX век более прославлен открытиями и изобретениями, решительно *не* цифровыми. Честно говоря, окружающий мир, данный нам в ощущениях, очень редко предстает в цифровом обличье. Он скорее представляет собой континуум, который не так-то легко выразить в числах.

Хотя Холлерит уже использовал реле в своих карточных табуляторах и сортировщиках, применять реле в компьютерах — которые со временем получили название *электромеханических* — начали лишь в середине 1930-х. В них использовали, как правило, не телеграфные, а телефонные реле, разработанные для управления телефонными вызовами.

Первые релейные компьютеры совсем не походили на устройство, которое мы разработали в предыдущей главе (позже мы увидим, что архитектуру нашего компьютера я построил на основе микропроцессоров 1970-х). В частности, нам сегодня кажется очевидным, что действие компьютеров должно основываться на двоичных числах, но в действительности это имело место далеко не всегда.

Другое отличие нашего релейного компьютера от настоящих устройств заключается в том, что никто в 30-х годах не терял рассудок настолько, чтобы пытаться собрать из реле 524 288 битов памяти! По цене, занимаемому пространству и потребляемой мощности память такого объема была совершенно нереализуема. То мизерное ее количество, что все-таки было доступно, применялось только для хранения промежуточных результатов. Сами программы записывали на физических носителях, например ленте с перфорацией. Вообще использованная нами концепция совместного хранения кода и данных в памяти возникла несколько позже.

По-видимому, хронологически первый релейный компьютер был построен Конрадом Цузе (Conrad Zuse) (1910–1995), который начал собирать его в 1935 г. в квартире своих родителей в Берлине, будучи студентом инженерного факультета. В нем уже применялось двоичное кодирование, но в первых версиях в качестве памяти использовались не реле, а механические приспособления. Программы для своих компьютеров Цузе записывал на 35-мм кинопленке, пробивая в ней отверстия.

В 1937 г. Джордж Стибиц (George Stibitz) (1904–1995) из Bell Telephone Laboratories принес домой пару телефонных реле и собрал из них на кухонном столе 1-разрядный сумматор, который его жена позже окрестила «К-машиной» (т. е. кухонной машиной). С этих экспериментов началось создание устройства под названием «Complex Number Computer» (1939).

Тем временем студенту выпускного курса Гарвардского университета Говарду Эйкену (Howard Aiken) (1900–1973) по-

надобилось устройство, способное выполнить большой объем однообразных вычислений. На этой основе возникло сотрудничество между Гарвардом и IBM, выразившееся в создании компьютера «Automated Sequence Controlled Calculator» (ASCC), позже получившего имя «Марк I». Его строительство завершилось в 1943 г. Это был первый цифровой компьютер, способный печатать таблицы, т. е. наконец-то реализовавший мечту Чарльза Бэббиджа. А компьютер «Марк II» стал самым большим релейным устройством. В нем использовалось 13 000 реле. В Гарвардской вычислительной лаборатории, возглавляемой Эйкеном, был прочитан и первый в истории курс информатики.

Реле оказались далеко не идеальным прибором для создания компьютеров. Будучи механическими устройствами, действие которых основывалось на изгибании металлической пластины, после продолжительной работы они в самом прямом смысле слова ломались. К неисправности реле приводили и кусочки грязи или бумаги, застрявшие между контактами. Особенно прославилась в этом отношении мошка, в 1947 г. извлеченная из реле компьютера «Марк II». Грейс Мюррей Хоппер (Grace Murray Hopper) (1906–1992), в 1944 г. ставшая сотрудником Эйкена, а позже прославившаяся как специалист по языкам программирования, приклеила ее в журнал с примечанием «первый случай выловленного бага»[1].

Реле можно заменить вакуумной лампой, изобретенной Джоном Эмброузом Флемингом (John Ambrose Fleming) (1849–1945) и Ли Форестом (Lee de Forest) (1873–1961) для использования в радио. К 1940-м радиолампы повсеместно применяли для усиления телефонных сигналов, и практически в каждом доме имелся радиоприемник, заполненный этими тускло светящимися стеклянными колбами. Из радиоламп, как и из реле, можно собрать вентили И, ИЛИ, И-НЕ и ИЛИ-НЕ.

Совершенно не важно, из чего собрана вентильная схема — из реле или из радиоламп. В любом случае из вентилей можно собрать сумматоры, селекторы, дешифраторы, триггеры и счетчики. Все, что я говорил в предыдущих главах о компонентах, собранных из реле, остается истинным и после замены реле на радиолампы.

---

[1] От англ. «bug» — жучок. На компьютерном жаргоне *багом* называется ошибка в программе. — *Прим. перев.*

Но и у радиоламп свои недостатки. Они дороги, энергоемки, сильно греются и, главное, рано или поздно сгорают. С этим можно было только смириться. Никто из владельцев ламповых приемников не избежал необходимости время от времени менять в них лампы. Телефонные системы разрабатывались с избыточной надежностью, поэтому периодический выход ламп из строя не наносил большого ущерба (да никто и не ждет, что телефоны будут работать безупречно). Другое дело — компьютер. Во-первых, в нем выход лампы из строя сразу можно и не заметить. Во-вторых, ламп в компьютере должно быть так *много*, что они будут сгорать в среднем каждые несколько минут.

У ламп по сравнению с реле есть одно большое преимущество — переключение лампы из состояния в состояние занимает всего одну миллионную долю секунды — одну *микросекунду*. Это в тысячу раз быстрее, чем реле, которое переключается в лучшем случае за 1 миллисекунду, т. е. тысячную долю секунды. Забавно, что создатели первых компьютеров их быстродействию большого внимания не уделяли, поскольку скорость вычислений определялась скоростью считывания программы с бумажной или пластиковой ленты. В подобных устройствах более высокая по сравнению с реле скорость радиоламп была не важна.

И все же в начале 1940-х вакуумные лампы применялись в компьютерах все чаще, а к 1945 г. реле были вытеснены окончательно. В отличие от релейных электромеханических компьютеров новые устройства, основанные на вакуумных лампах, стали называться *электронными* компьютерами.

Британский компьютер «Колосс» (начал работать в 1943 г.) создавался в противовес немецкой шифровальной машине «Энигма». В его разработке (а также в разработке других британских компьютерных проектов) принимал участие Алан Тьюринг (Alan Turing) (1912–1954), прославившийся главным образом благодаря двум своим статьям. В первой, опубликованной в 1937 г., он ввел концепцию «вычислимости», проанализировав, что могут и чего не могут делать компьютеры. Разработанная им абстрактная модель компьютера известна теперь как «машина Тьюринга». Вторая знаменитая статья Тьюринга посвящена проблеме искусственного интеллекта. В ней он описал способ проверки машины на разумность (тест Тьюринга).

Компьютер ENIAC (Electronic Numerical Integrator and Computer, Электронный числовой интегратор и компьютер) на 18 000 лампах был создан в Школе электроинженерии им. Мура при Пенсильванском университете Преспером Экертом (J. Presper Eckert) (1919–1995) и Джоном Моучли (John Mauchly) (1907–1980). Его создание завершилось в конце 1945 г. По тоннажу (30 т) ENIAC был самым большим компьютером в истории и, вероятно, сохранит этот титул навечно. Домашние компьютеры обогнали его по быстродействию в 1977 г. Попытка Экерта и Моучли запатентовать компьютер не удалась из-за конфликтной заявки конкурента — Джона Атанасоффа (John Atanasoff) (1903–1995), который несколько ранее собрал электронный компьютер, который толком так и не заработал.

ENIAC заинтересовал математика Джона фон Неймана (John von Neumann) (1903–1957), венгра по происхождению, который с 1930 г. жил в США. Нейман был профессором математики в Принстонском Институте сложных исследований, славился способностью производить в уме сложнейшие вычисления и занимался всем, начиная с квантовой механики и кончая применением теории игр в экономике.

Джон Нейман помогал проектировать ENIAC следующего поколения — EDVAC (Electronic Discrete Variable Automatic Computer, Электронный автоматический компьютер с дискретными переменными). В статье «Предварительное обсуждение логической конструкции электронной вычислительной машины», опубликованной в 1946 г. в соавторстве с Артуром Берксом (Arthur Burks) и Германом Голдстайном (Herman Goldstine), он описал некоторые особенности компьютеров, благодаря которым EDVAC оказался значительным шагом вперед в сравнении с ENIAC. Разработчики EDVAC понимали, что в компьютерах нужно использовать двоичное исчисление (в ENIAC применялась десятичная система). У компьютера также должно быть как можно больше памяти, и во время выполнения программы эту память надо использовать для хранения как команд, так и данных (програм-

мирование ENIAC осуществлялось с помощью тумблеров и переключения кабелей). Команды должны храниться в памяти последовательно и адресоваться с помощью счетчика команд, вместе с тем допуская условные переходы. Эта концепция теперь известна как *концепция запоминаемой программы.*

Эти нововведения оказались столь важным эволюционным этапом, что мы теперь говорим о них, как об *архитектуре Неймана.* Компьютер, разработанный нами в предыдущей главе, — типичная машина Неймана. Но в архитектуре Неймана есть и «узкое место». Машина Неймана обычно тратит значительное время не на выполнение команд, а на их загрузку из памяти. Как вы помните, компьютер из главы 17 в его окончательной редакции три четверти времени работы с командой тратил на ее перемещения.

Во времена EDVAC создание объемной памяти из радиоламп представлялось неэкономичным, поэтому на замену лампам предлагались порой весьма странные устройства. В одном из них, успешно работавшем и называвшемся *памятью с ртутной линией задержки* (mercury delay line memory), применялись 5-футовые трубки с ртутью. С одного конца трубки в ртуть с интервалом около микросекунды посылались слабые пульсации, которые приблизительно через 1 миллисекунду достигали противоположного конца трубки (там они детектировались как звуковые волны и направлялись обратно). Каждая трубка с ртутью могла хранить около 1 024 битов информации. В середине 50-х появились запоминающие устройства, состоящие из большого количества маленьких намагниченных металлических колец, подвешенных на проводах. Каждое кольцо могло хранить 1 бит информации.

О природе компьютеров в 40-е годы задумывался не только Нейман. В этом же направлении размышлял и Клод Шеннон (род. 1916). В главе 11 я обсуждал его магистерскую диссертацию (1938), в которой он установил взаимосвязь между тумблерами, реле и булевой алгеброй. В 1948 г., работая в Bell Telephone Laboratories, он опубликовал в журнале *Bell System Technical Journal* статью «Математическая теория коммуникации», в которой не только впервые в печати использовал слово «бит», но и заложил основы того, что сегодня называется *теорией информации.* Предметом теории информации является передача цифровой информации при наличии шума (ко-

торый обычно не дает передать информацию полностью) и способы его компенсации. В 1949 г. он написал первую статью о программировании на компьютере игры в шахматы, а в 1952 г. разработал механическую мышь, которая с помощью реле находила выход из лабиринта. В Bell Telephone Laboratories Шеннон также был хорошо известен как человек, умеющий жонглировать, катаясь на одноколесном велосипеде.

Самым известным трудом Норберта Винера (Norbert Wiener) (1894–1964), который 18 лет от роду защитил диссертацию по математике в Гарвардском университете, является книга «Кибернетика, или Управление и связь в животном и машине» (1948). Именно он назвал *кибернетикой* (от греческого слова, означающего «кормчий») теорию о взаимосвязи биологических процессов в человеке и животном с механикой компьютеров и роботов. В наши дни очень распространена приставка *кибер-*, указывающая на отношение чего-либо к компьютерам. Например, миллионы компьютеров, подключенных к Интернету, называют *киберпространством.* Слово это ввел в романе «Невромант» (1984) писатель Вильям Гибсон (William Gibson), работающий в жанре «*киберпанк*».

В 1948 г. корпорация Eckert-Mauchly Computer (позже ставшая частью Remington Rand) начала работу над первым массовым компьютером — UNIVAC (Universal Automatic Computer, Универсальный автоматический компьютер). Его разработка завершилась в 1952 г., и первый экземпляр был направлен в Бюро переписей. В 1952 г. UNIVAC впервые вышел в эфир на канале CBS, где его использовали для предсказания результата президентских выборов, при этом ведущий Уолтер Кронкайт (Walter Croncite) величал его «электронным мозгом». В том же 1952 г. свою первую коммерческую компьютерную систему — 701 — выпустила IBM.

С этого времени начинается история массового применения компьютеров в бизнесе и государственном секторе. Она очень интересна, но мы все-таки отвлечемся и перейдем теперь к другой истории — истории, которая сделала компьютеры компактными и дешевыми приборами из разряда бытовой техники. Началась она в 1947 г. с прорыва в области электроники, который прошел почти незамеченным.

Фирма Bell Telephone Laboratories в течение многих лет была местом, где умные люди могли заниматься почти всем, что их

интересовало. Некоторых из них, к счастью, интересовали компьютеры. Я уже упоминал Джорджа Стибица и Клода Шеннона, которые внесли значительный вклад в развитие вычислительной техники, работая в Bell Telephone Laboratories. Позже, в 70-е годы в этой компании родилась компьютерная операционная система UNIX и язык программирования C, к которым я вернусь в следующих главах.

Bell Telephone Laboratories возникла 1 января 1925 г., когда компания American Telephone & Telegraph официально заявила о выделении научных и технических исследовательских подразделений в отдельную фирму. Компания Bell Telephone Laboratories создавалась для разработки технологических решений по улучшению телефонной связи. К счастью, это целеуказание довольно расплывчато и охватывает самые разнообразные исследования, включая очевидную вечную тему — усиление звукового сигнала, передаваемого по проводам, без внесения в него искажений.

С 1912 г. в Bell System работали над ламповыми усилителями, посвятив их усовершенствованию многочисленные исследования и проекты. И все же вакуумные лампы оставляли желать много лучшего. Они были громоздки, потребляли много энергии и часто перегорали. Альтернативы им, увы, не было.

Все изменилось 16 декабря 1947 г., когда два физика из Bell Telephone Laboratories — Джон Бардин (John Bardeen) (1908–1991) и Уолтер Браттейн (Walter Brattain) (1902–1987) собрали усилитель иного типа. Он состоял из германиевой пластины и полоски золотой фольги. Через неделю они продемонстрировали усилитель своему начальнику Вильяму Шокли (William Shockley) (1910–1989). Это был первый *транзистор*, прибор, который порой называют самым важным изобретением XX века.

Транзистор возник не на пустом месте. Восемью годами раньше, 29 декабря 1939 г. Шокли записал в своем блокноте: «Сегодня мне пришло в голову, что в принципе усилитель можно собрать с помощью полупроводников, а не вакуумных ламп». После демонстрации первого транзистора несколько лет ушло на его усовершенствование. Лишь в 1956 г. Шокли, Бардин и Браттейн были удостоены Нобелевской премии по фи-

зике «за исследования полупроводников и открытие транзисторного эффекта».

Раньше я уже говорил о проводниках и изоляторах. Проводники получили свое название благодаря тому, что они хорошо проводят электричество. Лучшими проводниками являются медь, серебро и золото, и отнюдь не случайно все они находятся в одном и том же столбце Периодической таблицы.

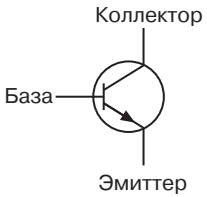
Как вы, конечно, знаете, электроны в атоме распределены по оболочкам, окружающим ядро. Атомы всех трех лучших проводников характеризуются единственным электроном на самой внешней оболочке. Этот электрон легко покидает атом и может свободно двигаться, создавая электрический ток. Противоположностью проводникам являются изоляторы — например, резина или пластмасса, который ток практически не проводят.

Элементы германий и кремний (и некоторые их соединения) называются *полупроводниками*, но не потому, что они проводят ток в два раза хуже проводников, а потому, что их проводимостью можно управлять. У атома полупроводника на внешней оболочке 4 электрона — половина того, что может вместить эта оболочка. В чистом полупроводнике между атомами формируются очень прочные связи, и возникает кристаллическая решетка, подобная кристаллической решетке алмаза. Такие полупроводники проводят ток довольно плохо.

Но полупроводники можно *легировать* (dope), т. е. добавлять в них различные примеси. Примеси одного рода добавляют в кристалл полупроводника избыточные электроны. Полученное вещество называется *полупроводником n-типа* (n означает негативный, т. е. *отрицательный*). Добавление примеси другого типа приводит к созданию *полупроводника p-типа* (p — позитивный, т. е. положительный).

Для создания усилителя достаточно между двумя слоями полупроводника n-типа разместить прослойку из полупроводника p-типа. Полученное устройство называется npn-транзистором, а входящие в его состав полупроводниковые фрагменты известны как *коллектор* (collector), *база* (base) и *эмиттер* (emitter).

Вот как схематически изображают npn-транзистор:



Небольшое напряжение на базе управляет гораздо большим током, проходящим из коллектора в эмиттер. Если на базе напряжения нет, транзистор практически закрывается.

Транзисторы обычно делают в виде небольших металлических цилиндров с тремя проводами-выводами.

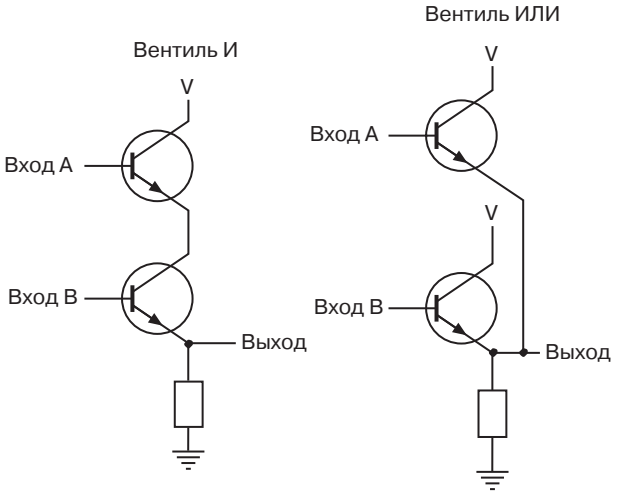
Появление транзистора ознаменовало зарождение *твердотельной* электроники: транзисторы изготавливаются не из вакуума, а из «твердого» вещества — полупроводника, создаваемого в наши дни почти исключительно на основе кремния. Транзисторы не только компактнее вакуумных ламп, но и потребляют меньше энергии, меньше греются и гораздо долговечнее. Невозможно представить себе карманный ламповый приемник. А вот транзисторному приемнику для питания достаточно небольшой батарейки и не нужен мощный теплоотвод. Первые счастливчики получили карманные транзисторные приемники в подарок на рождество 1954 г. Транзисторы в них были изготовлены фирмой Texas Instruments, которая сыграла в полупроводниковой революции важную роль.

Но *самым* первым коммерческим приложением транзисторов стали не приемники, а слуховые аппараты. Александр Грейам Белл (Alexander Graham Bell) много работал с глухими людьми, и в память об этом компания AT&T разрешила производителям слуховых аппаратов применять транзисторные технологии без платы за пользование патентом. В 1960 г. дебютировал транзисторный телевизор, и к настоящему времени ламповые устройства практически исчезли (хотя и не совсем — отдельные аудиофилы и электрогитаристы по-прежнему предпочитают звук ламповых усилителей).

В 1956 г. Шокли ушел из Bell Telephone Laboratories, переехал в город своего детства Пало-Альто (штат Калифорния) и основал там компанию Shockley Semiconductor Laboratories. Это была первая компания такого рода в той местности. Со временем там открылись отделения и других полупроводниковых и компьютерных компаний, и теперь область к югу от Сан-Франциско неофициально известна как Силиконовая долина[1].

Изначально предполагалось использовать вакуумные лампы в усилителях, но потом оказалось, что они также пригодны для сборки логических схем. То же самое случилось и с транзисторами. На рисунке показан транзисторный вентиль И, структура которого почти не отличается от релейной версии. Выход равен 1, только если оба транзистора проводят ток, а это в свою очередь возможно лишь при условии, что сигналы на входах A и B оба равны 1. Резистор в цепи нужен, чтобы в такой ситуации избежать короткого замыкания.

Если соединить транзисторы, как показано на схеме справа, получится вентиль ИЛИ. В вентиле И эмиттер верхнего транзистора соединен с коллектором нижнего. В вентиле ИЛИ к питанию подключены оба транзистора. Эмиттеры соединены друг с другом.



[1] Правильнее и понятнее было бы перевести ее название как Кремниевая долина. — *Прим. перев.*

Итак, все, что мы знаем о сборке логических схем и других компонентов из реле, применимо и к транзисторам. И реле, и радиолампы, и транзисторы изначально создавались в основном для усиления сигнала, но из всех этих устройств можно также создавать и логические схемы, из которых собираются компьютеры. Первые транзисторные компьютеры появились в 1956 г., и уже через несколько лет они полностью вытеснили проекты ламповых компьютеров.

Словом, благодаря транзисторам компьютеры, конечно, стали надежнее, компактнее и экономичнее. Но облегчили ли транзисторы *сборку* компьютеров?

По правде говоря, нет. Конечно, с помощью транзисторов на небольшом пространстве можно разместить больше логических схем, но это не снимает с вас заботы о *соединении* всех компонентов. Собирать вентили из транзисторов ничуть не легче, чем из реле или вакуумных ламп. В какой-то степени это даже сложнее, так как за небольшой транзистор не так-то просто ухватиться. Если бы вы решили собрать компьютер из главы 17 с оперативной памятью 64 кб из транзисторов, вам пришлось бы приложить немало усилий, чтобы разработать некую структуру, на которой крепились бы все компоненты. А собственно процесс сборки в значительной степени состоял бы из кропотливой спайки миллионов соединений между миллионами транзисторов.

Однако, как мы уже видели, имеются определенные комбинации транзисторов, которые встречаются в схемах неоднократно. Из пар транзисторов составляются вентили, из вентилей — триггеры, сумматоры, селекторы или дешифраторы. Из триггеров собирают многобитовые защелки или массивы RAM. Собрать компьютер было бы много легче, если бы наиболее популярные комбинации транзисторов изначально существовали в собранном виде.

По-видимому, первым эту идею в мае 1952 г. сформулировал британский физик Джеффри Даммер (Geoffrey Dummer) (род. 1909):

Я хотел бы заглянуть в будущее, — сказал он. — Благодаря распространению транзисторов и успешным исследованиям полупроводников теперь, по-видимому, возможно представить себе электронное устройство в виде сплошного твердого тела, лишен-

ного соединительных проводов. Это тело может состоять из слоев изолирующих, проводящих, фильтрующих и усиливающих веществ, в которых электрические функции выполняются просто за счет придания этим слоям определенной формы.

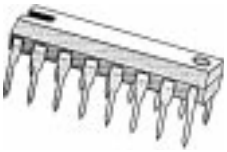
Однако первый работающий образец появился лишь через несколько лет.

В июле 1958 г. мысль о возможности изготовления нескольких транзисторов, резисторов и других деталей из единого куска кремния пришла в голову Джека Килби (Jack Kilby) (род. 1923) из Texas Instruments, который ничего не знал о предсказаниях Даммера. Еще через полгода, в январе 1959 г. примерно к такому же заключению пришел и Роберт Нойс (Robert Noyce) (1927–1990). Поначалу Нойс работал в Shockley Semiconductor Laboratories, но в 1957 г. он с семью другими учеными уволился из нее и основал корпорацию Fairchild Semiconductor.

В истории техники одновременные изобретения происходят чаще, чем можно предположить. Хотя Килби изобрел устройство на полгода раньше Нойса, а фирма Texas Instruments подала заявку на патент раньше, чем Fairchild Semiconductor, Нойс получил патент первым. Последовавшие за этим судебные тяжбы завершились ко всеобщему удовлетворению лишь десять лет спустя. Теперь Килби и Нойса считают соизобретателями *интегральной микросхемы* (ИС), или попросту *чипа*, хотя они никогда не работали вместе.

Для изготовления интегральной микросхемы используется сложный процесс, заключающийся в сборке тончайших пленок легированного кремния определенной формы. Разработка новой интегральной схемы стоит очень дорого, но благодаря массовому производству цена падает: чем больше делают микросхем, тем они дешевле.

Сама по себе кремниевая микросхема очень тонка и хрупка, поэтому она нуждается в прочном корпусе, который с одной стороны защитил бы ее, а с другой — обеспечивал бы возможность ее соединения с другими микросхемами. Чаще других используется прямоугольный пластмассовый корпус *DIP* (dual inline package, корпус с двухрядным расположением выводов) с 14, 16 или даже 40 выводами.



На рисунке показана микросхема с 16 выводами. Если взять ее так, чтобы небольшая прорезь в корпусе оказалась слева (как на рисунке), то нумерация выводов от 1 до 16 будет идти против часовой стрелки, начинаясь с левого нижнего вывода и заканчиваясь на левом верхнем. Расстояние между выводами равно 1/10 дюйма (около 2,5 мм).

В 60-е годы развитию рынка интегральных микросхем способствовали гонка вооружений и освоение космоса. Первым гражданским коммерческим продуктом, в котором использовалась микросхема, опять же стал слуховой аппарат, выпущенный в 1964 г. фирмой Zenith. В 1971 г. фирма Texas Instruments продала первый электронный калькулятор, а фирма Pulsar — первые электронные часы (разумеется, микросхемы для часов размещаются в особых корпусах). Затем последовала целая лавина различных устройств на микросхемах.

В 1965 г. Гордон Мур (Gordon Moore), в то время работавший в компании Fairchild Semiconductor, а позже ставший одним из основателей корпорации Intel, обратил внимание на то, что с 1959 г. число транзисторов в одной микросхеме ежегодно удваивалось. Он предсказал, что эта тенденция сохранится и в будущем. В действительности технологическое развитие шло несколько медленнее, поэтому в окончательной редакции «закон Мура» предсказывает удвоение числа транзисторов в микросхеме каждые полтора года. Столь стремительный прогресс объясняет, почему в наше время компьютеры безнадежно устаревают всего за несколько лет. Некоторые специалисты полагают, что действие закона Мура продлится до 2015 г.

Для описания микросхем используются следующие понятия: *малый уровень интеграции* (small-scale integration, SSI) — микросхемы менее чем с 10 логическими схемами; *средний уровень интеграции* (medium-scale integration, MSI), или СИС (средняя интегральная схема) — микросхемы с 10–100 логическими схемами; *высокий уровень интеграции* (large-scale

integration, LSI), или БИС (большая интегральная схема) — микросхемы с 100–5000 логических схем; *сверхвысокий уровень интеграции* (very-large-scale integration, VLSI), или СБИС (сверхбольшая интегральная схема) — микросхемы с 5000–50000 логических схем; *суперсверхвысокий уровень интеграции* (super-large-scale integration, SLSI) — микросхемы с 50000–100000 логических схем; *ультравысокий уровень интеграции* (ultra-large-scale integration, ULSI), или УБИС (ультрабольшая интегральная схема) — микросхемы более чем с 100000 логических схем.

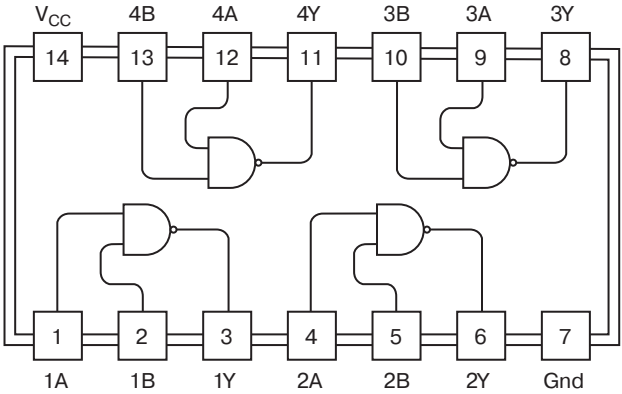
До конца этой главы и на всю следующую мы остановим нашу машину времени на середине 70-х годов прошлого века — темной эпохе, когда люди еще не слыхивали о фильме «Звездные войны», а СБИС лишь едва замаячили на горизонте. В те времена при сборке интегральных схем использовалось несколько различных технологий, каждая из которых определяет собственное *семейство* (family) микросхем. В середине 70-х главенствующее положение занимали два семейства: ТТЛ и КМОП.

Сокращение ТТЛ означает *транзисторно-транзисторная логика* (transistor-transistor logic). В те времена неизменным атрибутом стола инженера цифрового проектирования (т. е. человека, разрабатывавшего из микросхем большие схемы) была книга «The TTL Data Book for Design Engineers» — справочник по ТТЛ-микросхемам фирмы Texas Instruments, опубликованный в 1973 г. В ней содержались подробные сведения об интегральных микросхемах ТТЛ серии 7400 (номера всех микросхем в этой серии начинались с 74), которые продавались компанией Texas Instruments и некоторыми другими.

Любая интегральная микросхема серии 7400 состоит из нескольких логических вентилей, собранных в некоторой заданной конфигурации. В одних микросхемах содержатся простые логические схемы, из которых можно создавать более крупные компоненты, другие представляют собой готовые триггеры, сумматоры, селекторы и дешифраторы.

Первая микросхема в серии 7400 имела (вполне ожидаемо) номер 7400 и описана в справочнике как «счетверенная двухвходовая положительная схема И-НЕ». Это сложное название означает, что в данной микросхеме присутствуют четыре двухвходовых вентиля И-НЕ. *Положительными* (positive) они на-

зываются потому, что в них наличие напряжения означает 1, а его отсутствие — 0. У микросхемы 14 выводов, назначение которых описано на следующей диаграмме.



На диаграмме показан вид микросхемы сверху (выводы уходят вниз); прорезь расположена слева.

Вывод 14, обозначенный символом V_{CC}, эквивалентен контакту со значком V, которым я обозначал питающее напряжение. Нижний индекс C означает, что в микросхеме питание подается на *коллектор* транзистора; удвоен индекс потому, что так по традиции обозначают входное напряжение. Символы Gnd в обозначении 7-го контакта означают «ground» — земля. Любая микросхема должны быть подключена к питанию и к земле.

Питающее напряжение для ТТЛ-микросхем серии 7400 заключено в пределах от 4,75 до 5,25 В, иначе говоря 5 вольт ± 5%. Если напряжение упадет ниже 4,75 В, микросхема не будет работать. Если оно превысит 5,25 В, микросхема может испортиться. Батарейки для питания микросхем обычно не применяются. Даже если вы найдете 5-вольтовую батарейку, неточность напряжения в ней превысит допустимые для микросхем пределы. Для устройств на микросхемах ТТЛ, как правило, используется питание от розетки.

У каждого из четырех вентилей И-НЕ микросхемы 7400 предусмотрено два входа и один выход. Работают они независимо друг от друга. В предыдущих главах мы говорили о том,

что входной сигнал равен либо 1 (есть напряжение), либо 0 (нет напряжения). В действительности, входной сигнал любого из вентилей может варьироваться от 0 (земля) до 5 В (V_{CC}). Напряжения в диапазоне 0–0,8 В считаются логическим нулем, а напряжения от 2 до 5 В — логической единицей. Напряжений между 0,8 и 2 В в схеме следует избегать.

Напряжение около 0,2 В на выходе вентиля ТТЛ соответствует нулю, напряжение 3,4 В — единице. Поскольку эти напряжения могут несколько варьироваться, иногда говорят не о 0 и 1, а о *низком* (low) и *высоком* (high) уровнях выходного сигнала. Более того, иногда высокое напряжение соответствует 0, а низкое — 1. Тогда говорят о конфигурации с *отрицательной* (negative) логикой.

Выходные напряжения 0,2 В (логический 0) и 3,4 В (логическая 1) на выходе вентиля ТТЛ заведомо попадают в допустимые пределы — от 0 до 0,8 В для нуля и от 2 до 5 В для единицы. Так микросхемы ТТЛ защищаются от *шума*. Единичный сигнал может упасть на 1,4 В и все же остаться единицей. Нулевой сигнал может вырасти на 0,6 В и остаться нулем.

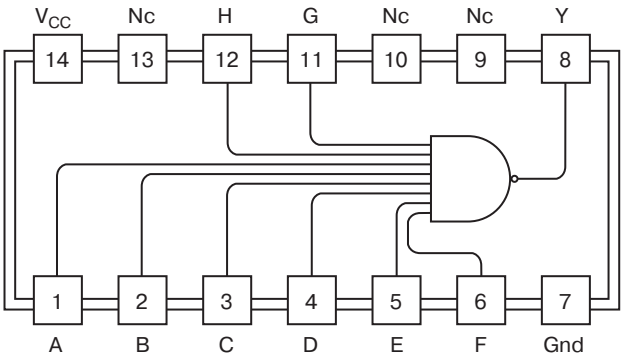
Вероятно, самой важной характеристикой конкретной микросхемы является *время установки* (propagation time), т. е. временной интервал между изменением сигнала на входе и соответствующим изменением сигнала на выходе.

Время установки микросхем обычно измеряется в *наносекундах* (нс). Наносекунда — это *очень* короткий промежуток времени. Одна тысячная секунды называется миллисекундой, одна миллионная секунды — микросекундой. Наносекунда — это одна миллиардная доля секунды. Время установки для вентилей И-НЕ в микросхеме 7400 гарантированно не превышает 22 нс — 0,000000022 секунды или 22 миллиардных секунды.

Прочувствовали наносекунду? Нет? Что ж, вы не одиноки. Никто на этой планете не способен представить себе такое короткое время. Наносекунда намного короче самых быстрых изменений, воспринимаемых человеком, поэтому она всегда будет непостижима. Любая попытка объяснения лишь затрудняет восприятие. Например, я могу сказать так: если вы держите эту книгу на расстоянии 30 см, наносекунда — это время, за которое свет проходит расстояние от страницы до глаза. Как, стало понятнее?

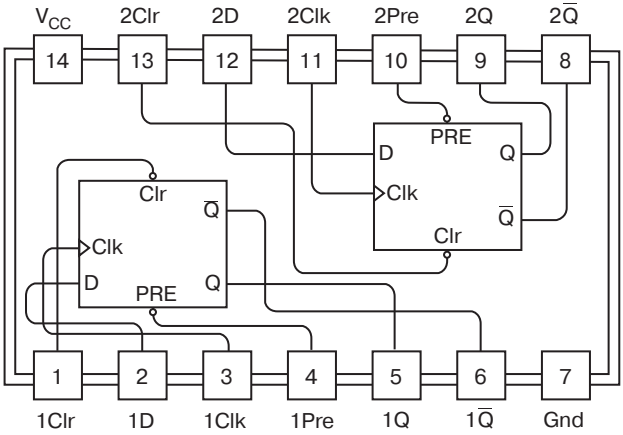
Но именно благодаря наносекунде возможно само существование компьютеров. Как мы убедились в главе 17, действие компьютера ограничено простыми до занудства операциями: переместить байт из памяти в регистр, сложить с другим байтом, вернуть результат в память. Сделать с помощью компьютера (настоящего, а не из главы 17) хоть что-то стоящее удается лишь потому, что эти операции выполняются очень быстро. Процитируем Роберта Нойса: «Когда вы свыкаетесь с мыслью о наносекунде, концептуально действие компьютера оказывается очень простым».

Но вернемся к справочнику по микросхемам ТТЛ. Вы, конечно, увидите в нем много старых знакомых. В микросхеме 7402 содержатся 4 двухвходовых вентиля ИЛИ-НЕ, в микросхеме 7404 — 6 инверторов, в микросхеме 7408 — 4 двухвходовых вентиля И, в микросхеме 7432 — 4 двухвходовых вентиля ИЛИ, в микросхеме 7430 — 8-входовый вентиль И-НЕ:

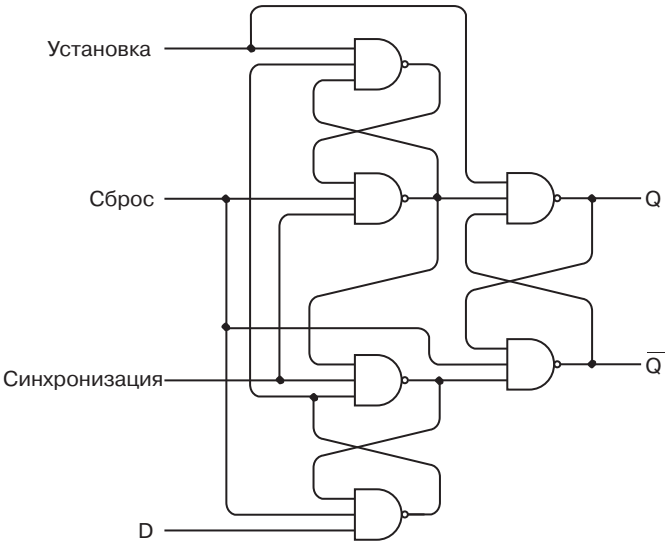


Сокращение «Nc» означает «*no connection*» — не подключено.

Название микросхемы 7474 также звучит знакомо — «сдвоенный D-триггер со сбросом и предустановкой, срабатывающий по фронту». На диаграмме схематически показано его устройство.



В справочник по микросхемам ТТЛ включены даже логические схемы триггеров в этой микросхеме:



Этот рисунок очень похож на диаграмму в конце главы 14 за исключением того, что там я применил вентили ИЛИ-НЕ. Таблица логики триггера из справочника тоже выглядит немного иначе.

Входы				Выходы	
Pre	**Clr**	**Clk**	**D**	**Q**	$\overline{\mathbf{Q}}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	L	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

В таблице буква B означает высокий уровень сигнала, H — низкий. Если хотите, можете считать их 1 и 0. В моем триггере нормальное значение сигналов «Сброс» и «Установка» 0; здесь же оно равно 1.

Листая страницы справочника по микросхемам ТТЛ, вы обнаружите, что в микросхеме 7483 скрывается 4-разрядный двоичный полный сумматор, в микросхеме 74151 — селектор с 8 входами и 1 выходом, в микросхеме 74154 — дешифратор с 4 входами и 16 выходами, в микросхеме 74161 — синхронный 4-разрядный двоичный счетчик, в микросхеме 74175 — счетверенный D-триггер со сбросом.

Ну что ж, вот вы и догадались, что описания всех компонентов, что мы использовали, начиная с главы 11, я позаимствовал из книги «The TTL Data Book for Design Engineers».

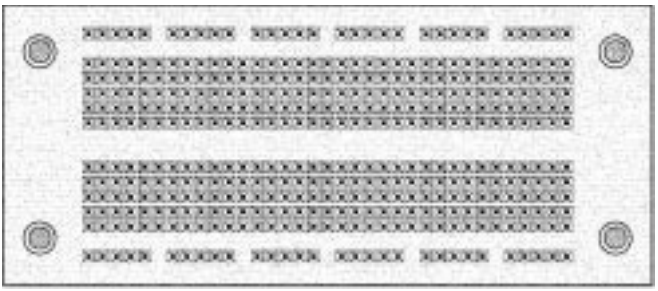
Инженеры проводили над ней долгие часы, знакомясь со всеми доступными типами микросхем ТТЛ. В принципе из микросхем ТТЛ уже можно собирать компьютер, описанный в главе 17. Соединять микросхемы гораздо легче, чем отдельные транзисторы. Но вот оперативную память собирать из микросхем ТТЛ я бы вам не советовал. Самый емкий чип RAM, описанный в издании книги «The TTL Data Book for Design Engineers» 1973 г., имеет объем 256 × 1 бит. Для создания памяти RAM объемом 64 кб таких схем понадобилось бы 2 048! Короче, для памяти нужно что-то получше, чем ТТЛ. Но об этом подробнее в главе 21.

Вам, вероятно, также понадобится лучший вибратор. Можно, конечно, просто подключить вход ТТЛ-инвертора к его же входу, но вибратор с более предсказуемой частотой предпочтительней. Такой прибор легко создать, используя кварцевый кристалл, обычно размещаемый в цилиндрическом корпусе с двумя выводами. Такие кристаллы вибрируют с определенной частотой, обычно не меньше миллиона колебаний в секунду. Миллиону колебаний в секунду соответствует частота 1 *мегагерц* (сокращенно МГц). Если бы мы собрали компьютер из главы 17 с помощью микросхем ТТЛ, для его работы лучше всего подошла бы тактовая частота 10 МГц. На выполнение каждой команды уходило бы 400 нс. Это, конечно, невообразимо быстрее, чем даже в самом быстром релейном компьютере.

Другое популярное семейство микросхем называется *КМОП* (комплементарные МОП-структуры), или *CMOS* (complementary metal-oxide semiconductor). Если бы вы в середине 70-х годов в порядке хобби собирали какие-нибудь устройства из микросхем КМОП, вашей настольной книгой был бы справочник «CMOS Databook», выпущенный фирмой National Semiconductor. В нем содержится информация о КМОП-микросхемах серии 4000.

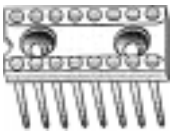
Для микросхем ТТЛ требуется питание от 4,75 до 5,25 В. Диапазон допустимых напряжений для микросхем КМОП — от 3 до 18 В. Есть где размахнуться! Более того, микросхемы КМОП потребляют гораздо меньше энергии, чем ТТЛ, благодаря чему становится возможным создание небольших устройств с микросхемами, работающих от батареек. Недостаток микросхем КМОП — низкое быстродействие. Например, гарантированное время установки 4-разрядного полного сумматора КМОП 4008, работающего от напряжения 5 В, составляет 750 нс. Если напряжение повысить, микросхема начнет работать быстрее: 250 нс при напряжении 10 В и 190 нс при напряжении 15 В. Но даже при повышенном напряжении микросхема КМОП далека от аналогичного ТТЛ-сумматора, время установки которого равно 24 нс. Впрочем, такое ясное различие между быстрыми ТТЛ-микросхемами и экономичными КМОП-микросхемами существовало лет 25 назад. В наши дни уже появились ТТЛ-микросхемы с пониженными требованиями к питанию и быстродействующие КМОП-микросхемы.

Процесс практической сборки устройства на микросхемах начинается на *макетной плате* (breadboard).



Под пластмассовым основанием каждые пять отверстий электрически соединены. Микросхема вставляется в макетную плату над центральной бороздой так, что ее выводы попадают в отверстия по обе стороны от борозды. При этом каждый вывод микросхемы электрически соединен с четырьмя соседними отверстиями. В эти отверстия вставляются провода для соединения с другими микросхемами.

Для более долговечного соединения микросхем используйте *монтаж накруткой* (wire-wrapping). Каждая микросхема устанавливается в гнездо с длинными квадратными штырьками:



Каждый штырек соответствует выводу микросхемы. Сами гнезда вставляются в плату с перфорацией. С ее противоположной стороны с помощью специального устройства на каждый штырек наматывается тонкий провод с изоляцией. На углах квадратного сечения штырька изоляция рвется, и между проводом и штырьком возникает контакт.

При серийном производстве приборов на микросхемах лучше использовать *печатные платы* (printed circuit boards). Впрочем, изготовление печатной платы по силам и любителю. Печатная плата — это пластина, покрытая тонкой медной фольгой. Те участки платы, на которых фольгу нужно оста-

вить, покрывают химически стойким веществом, а потом протравливают ее кислотой. Затем выводы гнезд для микросхем или самих микросхем припаиваются прямо к медному покрытию платы. Поскольку соединений между микросхемами требуется очень много, одного слоя меди часто не хватает. Поэтому промышленные печатные платы обычно бывают многослойными.

В начале 70-х годов появилась возможность собрать компьютерный процессор целиком на одной плате. После этого размещение всего процессора в единой микросхеме было уже вопросом времени. Хотя патент на «однокристальный» компьютер в 1971 г. получила фирма Texas Instruments, честь изготовления микросхемы-процессора принадлежит компании Intel, основанной в 1968 г. бывшими сотрудниками Fairchild Semiconductors Робертом Нойсом и Гордоном Муром. Первым значительным продуктом Intel была микросхема памяти емкостью 1 024 бита — рекорд для того времени, — выпущенная в 1970 г.

Проектируя микросхемы для программируемого калькулятора, который намеревалась выпускать японская компания Busicom, инженеры Intel приняли решение испробовать на них новый подход. Суть его инженер Intel Тед Хофф (Ted Hoff) выразил так: «Я решил проектировать не устройство, которое было бы калькулятором с возможностью программирования, а компьютер общего назначения, запрограммированный выполнять функции калькулятора». Так появилась на свет микросхема 4004 — первый «компьютер в чипе», или *микропроцессор*. Микросхема 4004 появилась на рынке в ноябре 1971 г. и содержала 2 300 транзисторов (согласно закону Мура, микропроцессоры, выпущенные через 18 лет после этого события, должны содержать в 4 000 раз больше транзисторов, т. е. около 10 миллионов — весьма точное предсказание).

Помимо числа транзисторов, нужно упомянуть еще о трех важных характеристиках ИС 4004. Их часто используют и для описания более поздних процессоров.

Во-первых, 4004 был *4-разрядным* процессором, т. е. шина данных в нем имела ширину 4 бита. При сложении или вычитании чисел процессор мог обрабатывать только 4-битовые фрагменты. Этим он отличается от 8-разрядного компьютера из главы 17, шина данных которого имеет ширину 8 битов.

Естественно, на 4 битах технология задержалась недолго. Как мы убедимся позже, 4-разрядные процессоры были очень быстро вытеснены 8-разрядными, а в конце 70-х появились уже 16-разрядные процессоры. Вспомните, что в 8-разрядном компьютере из главы 17 для сложения двух 16-разрядных чисел требовалось несколько команд, и вы оцените преимущество 16-разрядного процессора. В середине 80-х на первый план выдвинулись 32-разрядные процессоры, которые с тех пор стали стандартными для домашних компьютеров.

Во-вторых, 4004 работал на *тактовой частоте* (clock speed) 108 000 циклов в секунду, т. е. 108 *килогерц* (кГц). Тактовая частота это максимально допустимая частота генератора тактовых импульсов, при которой процессор может работать. Чуть больше, и в работе возможно появление сбоев. В 1999 г. процессоры, предназначенные для домашних компьютеров, пересекли отметку 500 МГц — приблизительно в 5 000 раз быстрее, чем у 4004.

В-третьих, объем адресуемой памяти у 4004 составлял 640 байт. По сегодняшним временам этот объем кажется смехотворно маленьким, но на большее при тогдашних микросхемах памяти рассчитывать не приходилось. Как вы узнаете из следующей главы, через пару лет процессоры могли уже обращаться к 64 кб памяти, как компьютер из главы 17. В 1999 г. адресное пространство микропроцессоров Intel составляло уже 64 терабайта, что значительно превосходит потребности домашних компьютеров, оперативная память которых редко превышает 256 Мб.

Три этих характеристики никак не отражают *способности* компьютера. На 4-разрядном процессоре можно складывать 32-разрядные числа, разбивая их на 4-битовые фрагменты. В каком-то смысле все компьютеры одинаковы. Если в аппаратной части одного процессора отсутствуют возможности, имеющиеся у другого, эту недостачу легко возместить средствами программного обеспечения. В конце концов результат будет один и тот же. Это, кстати, одно из заключений работы Алана Тьюринга о «вычислимости».

А вот по *скорости* выполнения одной и той же задачи процессоры *действительно* различаются. А ведь быстродействие — одна из главных привлекательных черт компьютеров.

На быстродействие процессора, очевидно, влияет максимальная тактовая частота, поскольку она определяет время, затрачиваемое на выполнение каждой команды. На быстродействии сказывается и ширина шины данных. Хотя 4-разрядный процессор и способен складывать 32-разрядные числа, делает он это гораздо медленнее 32-разрядного процессора. А вот связь между быстродействием и объемом адресуемой памяти уже не так очевидна. На первый взгляд, размер адресного пространства не имеет отношения к быстродействию и лишь накладывает ограничение на способность процессора решать определенные задачи, требующие значительной памяти. Но ограничение на объем памяти процессор всегда может обойти, используя определенные ее адреса для обмена данными с внешними накопителями. Можно, например, представить себе, что каждый байт, записываемый в определенную ячейку памяти, в действительности записывается на перфоленту, а байт, считываемый из этой ячейки, в действительности считывается с перфоленты. Однако этот обмен замедляет работу компьютера. Так что опять все упирается в скорость.

Конечно, быстродействие процессора этими тремя числами характеризуется только приблизительно. Они ничего не скажут вам об архитектуре процессора, о том, насколько эффективны и функциональны его команды. По мере совершенствования процессоров в них встраиваются многие общие функции, ранее выполнявшиеся программным обеспечением. Примеры этого встретятся нам в следующих главах.

Хотя все цифровые компьютеры наделены равными способностями, хотя они неспособны делать ничего, что выходило бы за пределы возможностей примитивной вычислительной машины, придуманной Аланом Тьюрингом, общая полезность компьютерной системы, *конечно*, определяется быстродействием процессора. Например, нет смысла в создании компьютера, который выполняет вычисления медленнее человеческого мозга. И нам вряд ли удавалось бы смотреть фильмы на экране компьютера, если бы процессор затрачивал по минуте на рисование каждого кадра.

Но вернемся в середину 70-х. Несмотря на ограничения процессора 4004, он был началом новой эры. В апреле 1972 г. компания Intel выпустила микросхему 8008 — 8-разрядный

микропроцессор с тактовой частотой 200 кГц и адресным пространством 16 кб (видите, как можно легко охарактеризовать процессор всего тремя числами). Затем в 1974 г. компании Intel и Motorola выпустили два микропроцессора, которые должны были заменить 8008. Эти микропроцессоры изменили мир.

Глава 19

Два классических микропроцессора

Микропроцессор, т. е. *центральное процессорное устройство*, все компоненты которого объединены в одной микросхеме, появился в 1971 г. Начало было весьма скромным. В первом микропроцессоре — микросхеме 4004 фирмы Intel — содержалось всего 2 300 транзисторов. Три десятилетия спустя даже микропроцессоры для домашних компьютеров приблизились к отметке 10 000 000 транзисторов.

Однако суть действия микропроцессора в основе своей осталась неизменной. Разумеется, миллионы дополнительных транзисторов решают очень важные задачи, но в постижении смысла работы микропроцессора они не только не помогают, но и мешают. Понять, как работает микропроцессор, проще всего на его первых серьезных моделях.

А родились они в 1974 г.: в апреле фирма Intel объявила о выпуске микросхемы 8080, а в августе появилась микросхема 6800 фирмы Motorola, которая с 1950 г. занималась изготовлением полупроводниковых и транзисторных приборов. Кстати, этими процессорами в 1974 г. дело не ограничилось: тогда же появились 4-разрядный процессор TMS 1000 фирмы Texas Instruments, который часто применялся в калькуляторах, игрушках и бытовых приборах, и первый 16-разрядный процессор PACE фирмы National Semiconductor. Но с точки зрения

истории вычислительной техники самыми важными оказались микропроцессоры 8080 и 6800.

Поначалу процессор 8080 стоил 360 долларов — ничтожная сумма по сравнению с многомиллионной стоимостью мэйнфреймов System/360 фирмы IBM, которыми пользовались крупные корпорации. И хотя процессор 8080 ни в коей мере нельзя сравнивать с System/360, уже через несколько лет IBM обратила на эти крохотные микросхемы самое пристальное внимание.

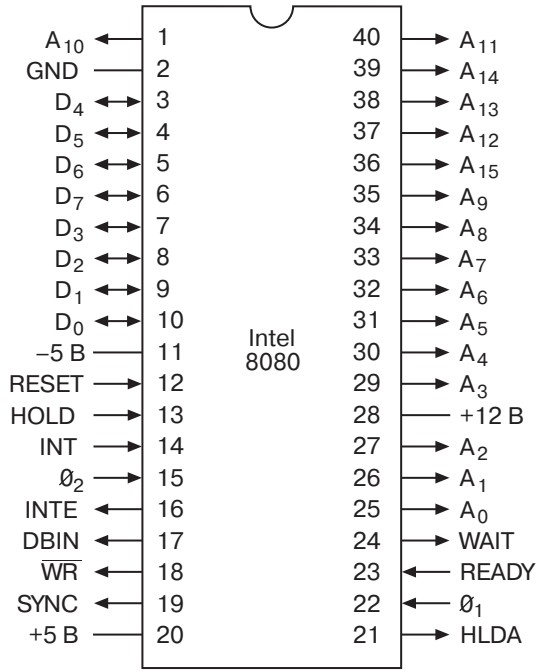
8-разрядный процессор 8080 содержит 6 000 транзисторов, работает с тактовой частотой 2 МГц и способен адресовать 64 кб памяти. Процессор 6800 содержит около 4 000 транзисторов и также способен адресовать 64 кб памяти. Первые микросхемы 6800 работали с тактовой частотой 1 Мгц, но в 1977 г. фирма Motorola выпустила обновленные модели, работавшие на частотах 1,5 и 2 МГц.

Микросхемы 8080 и 6800 называют *однокристальными* (single-chip) процессорами или, менее точно, однокристальными компьютерами. На самом деле, конечно же, микропроцессор — это лишь часть компьютера. Кроме него, компьютеру по меньшей мере нужно некоторое количество оперативной памяти, некий способ ввода информации в компьютер (устройство ввода), некий способ извлечь информацию из компьютера (устройство вывода) и еще несколько микросхем, которые связывали бы все это в единое целое. Подробнее я опишу эти компоненты в главе 21.

Для начала посмотрим на сам микропроцессор. Часто его описание сопровождают блок-схемой, на которой показаны компоненты процессора и способ их соединения. Но я думаю, что вам этого хватило и в главе 17. Сейчас мы попытаемся понять, что делается внутри процессора, рассмотрев, как он сообщается с внешним миром. Иначе говоря, будем считать процессор «черным ящиком», для понимания работы которого нам нет нужды детально вникать в его внутреннее устройство. Достаточно познакомиться с входными и выходными сигналами процессора, а также с его набором команд.

Как 8080, так и 6800 размещались в интегральных микросхемах с 40 выводами. Размеры их чаще всего были таковы: около 5 см в длину, около 1,5 см в ширину и около 3 мм в высоту.

Это, конечно, размеры корпуса. Кремниевая пластина внутри него гораздо меньше — в первых 8-разрядных процессорах она представляла собой квадрат со стороной около 6 мм. Корпус нужен для защиты микросхемы и для обеспечения доступа к его каналам ввода и вывода. Назначение каждого из 40 выводов микропроцессора 8080 таково:



Любому электрическому или электронному устройству необходимо электропитание. Одна из странностей процессора 8080 в том, что ему нужны *три* различных напряжения. Контакт 20 должен подключаться к напряжению 5 В, контакт 11 — к напря-

жению −5 В, а контакт 28 — к 12 В. Контакт 2 подключается к земле. В 1976 г. Intel выпустила процессор 8085, питание которого осуществлялось несколько проще.

Все остальные контакты представлены в виде стрелок. Стрелка, направленная *от* микросхемы, символизирует *выходной* сигнал. Он генерируется процессором и передается для обработки в другие устройства компьютера. Стрелка, направленная *к* микросхеме, символизирует *входной* сигнал. Он генерируется другими микросхемами компьютера и обрабатывается процессором. Некоторые контакты работают *как на вход, так и на выход*.

Процессору из главы 17 для работы требовалось синхронизирующее устройство. Процессору 8080 нужно два синхронизирующих сигнала, отмеченных на схеме $\emptyset_1$ и $\emptyset_2$ и подводимых к контактам 22 и 15. Для генерации этих сигналов удобнее всего использовать другую микросхему Intel — генератор тактовых импульсов 8224. К нему достаточно подключить кристалл кварца с частотой 18 МГц, и генератор сделает все остальное.

Для адресации памяти у любого процессора есть несколько выходных сигналов. Их количество и определяет объем памяти, к которой способен обращаться процессор. У 8080 контактов для обращения к памяти 16 (они обозначены символами от A_0 до A_{15}), а значит, он может адресовать $2^{16} = 65\,536$ байт памяти.

Процессор 8080 является 8-разрядным, т. е. за раз он способен прочитать из памяти или записать в память 8 битов данных. Обмен данными с памятью происходит через контакты с D_0 по D_7. На вход и на выход работают только эти 8 контактов. При чтении байта из памяти они работают на вход; при записи байта в память — на выход.

Еще 10 контактов процессора отведены под управляющие *сигналы*. Входной сигнал RESET, например, используется для сброса процессора. Наличие выходного сигнала $\overline{WR}$ означает, что процессор должен записать байт данных в память (этот сигнал соответствует входному сигналу W памяти). Кроме того, иногда управляющие сигналы подаются на контакты с D_0 по D_7, когда процессор считывает из памяти команду. В компьютерах, построенных на основе 8080, для работы с этими сигналами обычно используется системный контроллер 8228.

Некоторые управляющие сигналы 8080 я опишу позже, в целом же имейте в виду, что они прославились своей запутанностью, поэтому не занимайтесь самоистязанием и приступайте к их изучению, только если твердо решили собрать компьютер на основе этого процессора.

Допустим, что процессор 8080 соединен с 64 кб оперативной памяти, причем у нас есть возможность обмениваться с этой памятью данными независимо от процессора.

После сброса процессор 8080 считывает из памяти байт, расположенный по адресу 0000h, подавая 16 нулей на адресные контакты с A_0 по A_{15}. Считываемый байт должен содержать команду процессора 8080, а сам процесс считывания этого байта называется *выборкой команды* (instruction fetch).

В компьютере из главы 17 все команды (кроме Остановить) занимали по 3 байта — код операции и двухбайтовый адрес. В процессоре 8080 команды бывают длиной в 1, 2 и 3 байта. Некоторые команды указывают процессору прочитать байт из определенной ячейки памяти, другие — записать байт в определенную ячейку памяти, третьи — произвести какие-то внутренние операции, не обращаясь к оперативной памяти. Выполнив первую команду, процессор считывает из памяти следующую и т. д. Взятые вместе, эти команды представляют собой компьютерную программу, цель которой — заставить процессор что-то сделать.

Когда процессор 8080 работает с максимальной частотой 2 МГц, тактовый цикл длится 500 нс (разделив 1 на 2 000 000 секунд, получаем 0,000000500 с). Все команды компьютера из главы 17 занимали по 4 тактовых цикла. Команды из набора 8080 длятся от 4 до 18 тактовых циклов. Это значит, что на выполнение одной команды затрачивается от 2 до 9 микросекунд (миллионных долей секунды).

Вероятно, лучший способ понять, на что способен данный процессор, состоит в систематическом и полном изучении его набора команд.

Набор компьютера из главы 17 в его окончательном варианте содержал 12 команд. Набор 8-разрядного процессора может состоять из 256 команд, если каждому 8-битовому значению соответствует код операции (число команд можно увеличить, приписав некоторым из них 2-байтовые коды). Процессор 8080 так далеко не заходит — у него 244 команды. Число

немаленькое, но, по правде говоря, по способностям 8080 не так уж сильно превосходит компьютер из главы 17. Например, чтобы умножить или разделить два числа, вам и в 8080 придется писать собственную небольшую программу.

Как вы помните из главы 17, код команды из набора процессора обычно связывается с мнемонической комбинацией символов, причем некоторые из этих кодов обладают аргументами. Единственное назначение мнемокодов — облегчить человеку работу с командами. Процессор понимает только код команды; о связанном с ним тексте ему ничего не известно. Кстати, для простоты я излагаю мнемокоды из документации к процессору 8080 с некоторой вольностью.

У компьютера из главы 17 есть две важные команды, которые мы поначалу назвали *Загрузить* и *Сохранить*. Каждая из них занимает 3 байта памяти. Первый байт команды Загрузить отведен под ее код, а оставшиеся два указывают 8-битовый адрес. Содержимое ячейки с этим адресом процессор загружает в аккумулятор. Аналогично команда Сохранить записывает содержимое аккумулятора в ячейку, адрес которой указан в команде.

Позже мы обнаружили, что для сокращения записи эти команды удобно представить в виде мнемокодов:

```
LOD A,[aaaa]
STO [aaaa],A
```

где A означает аккумулятор (место размещения информации для команды Загрузить и ее источник для команды Сохранить), а *aaaa* — 16-битовый адрес ячейки памяти, обычно записываемый в виде 4 шестнадцатеричных цифр.

8-разрядный аккумулятор в 8080 обозначается A, как и в компьютере из главы 17. Есть в наборе 8080 и команды, аналогичные Загрузить и Сохранить. Коды двух этих команд в 8080 равны 3Ah и 32h (причем, за каждым кодом идет 16-битовый адрес), а мнемокоды — STA (Store Accumulator, сохранить из аккумулятора) и LDA (Load Accumulator, загрузить в аккумулятор).

Код	Команда
32	`STA [aaaa],A`
3A	`LDA A,[aaaa]`

Кроме аккумулятора в процессоре 8080 имеется 6 *регистров* (registers), в которых также могут храниться 8-битовые значения. Регистры очень похожи на аккумулятор. Фактически аккумулятор — это регистр особого типа. Подобно аккумулятору, регистры являются защелками; процессор может перемещать данные из памяти в регистры и из регистров в память. Однако регистры не столь гибки, как аккумулятор. При сложении двух чисел, например, результат всегда отправляется в аккумулятор, а не в один из регистров.

6 регистров 8080 называются B, C, D, E, H и L. Первый вопрос о них обычно звучит так: «А что случилось с F и G?», — а второй: «Куда девались I, J и K?» Ответ в том, что регистры H и L получили свои имена не в честь букв алфавита. H означает «high», а L — «low». Очень часто 8-битовые значения в регистрах H и L рассматриваются совместно, как 16-битовая *пара регистров* HL, при этом в H хранится старший (high) байт, а в L — младший (low) байт. Полное 16-разрядное значение часто используется как адрес в памяти, в чем мы убедимся чуть позже.

Насколько нужны регистры и как мы обошлись без них в компьютере из главы 17? Теоретически без них можно обойтись, но работать, имея под рукой регистры, удобнее. Многие компьютерные программы оперируют одновременно несколькими числами. Производить манипуляции с ними легче, если все числа хранятся не в памяти, а в регистрах микропроцессора. Да и работает программа быстрее: чем реже она обращается к памяти, тем меньше времени затрачивается на ее выполнение.

Целых 63 кода отведено в 8080 командам *MOV* (Move, переместить). Занимают команды из этой группы единственный байт и обычно перемещают содержимое одного регистра в другой (или тот же самый). Большое количество команд MOV — закономерное следствие наличия в микропроцессоре 7 регистров (считая аккумулятор).

Вот как выглядят первые 32 кода операции команды MOV. Помните, что целевой регистр указан в левом аргументе, а регистр-источник — в правом аргументе.

Код	Команда	Код	Команда
40	MOV B,B	50	MOV D,B
41	MOV B,C	51	MOV D,C

(*продолжение*)

Код	Команда	Код	Команда
42	MOV B,D	52	MOV D,D
43	MOV B,E	53	MOV D,E
44	MOV B,H	54	MOV D,H
45	MOV B,L	55	MOV D,L
46	MOV B,[HL]	56	MOV D,[HL]
47	MOV B,A	57	MOV D,A
48	MOV C,B	58	MOV E,B
49	MOV C,C	59	MOV E,C
4A	MOV C,D	5A	MOV E,D
4B	MOV C,E	5B	MOV E,E
4C	MOV C,H	5C	MOV E,H
4D	MOV C,L	5D	MOV E,L
4E	MOV C,[HL]	5E	MOV E,[HL]
4F	MOV C,A	5F	MOV E,A

Очень удобные инструкции! Как только значение попало в один из регистров, вы можете легко переместить его в другой регистр. Обратите внимание на четыре инструкции, в которых используется пара регистров HL, например:

```
MOV B,[HL]
```

Эта инструкция переносит в регистр B байт из ячейки памяти, адрес которой записан в паре регистров HL. Этим она отличается от инструкции LDA, загружающей в аккумулятор содержимое ячейки, адрес которой следует сразу за кодом команды. Откуда берется адрес в регистрах HL? Он может попасть туда даже несколькими способами. Например, его можно тем или иным образом вычислить.

Словом, команды:

```
LDA A,[aaaa]
MOV B,[HL]
```

используются для передачи данных из памяти в процессор, но при этом адрес ячейки памяти указывается в них по-разному.

Первый способ называется *прямой адресацией* (direct addressing), второй — *индексной адресацией* (indexed addressing).

Следующая группа из 32 команд доказывает, что по адресу, хранящемуся в регистрах HL, данные не только считываются, но и записываются.

Код	Команда	Код	Команда
40	MOV B,B	50	MOV D,B
60	MOV H,B	70	MOV [HL],B
61	MOV H,C	71	MOV [HL],C
62	MOV H,D	72	MOV [HL],D
63	MOV H,E	73	MOV [HL],E
64	MOV H,H	74	MOV [HL],H
65	MOV H,L	75	MOV [HL],L
66	MOV H,[HL]	76	HLT
67	MOV H,A	77	MOV [HL],A
68	MOV L,B	78	MOV A,B
69	MOV L,C	79	MOV A,C
6A	MOV L,D	7A	MOV A,D
6B	MOV L,E	7B	MOV A,E
6C	MOV L,H	7C	MOV A,H
6D	MOV L,L	7D	MOV A,L
6E	MOV L,[HL]	7E	MOV A,[HL]
6F	MOV L,A	7F	MOV A,A

Некоторые из этих команд, например:

```
MOV A,A
```

не делают ничего полезного, а команда:

```
MOV [HL],[HL]
```

и вовсе не существует. Код, который должен был бы принадлежать ей, отдан команде HLT (Halt, остановить).

Коды команды MOV становятся более наглядными, если посмотреть на их двоичное представление:

```
01ццциии
```

где буквами *ццц* обозначен целевой регистр, а буквами *иии* — регистр-источник. Расшифровка этих трехбитовых кодов такова.

000 = Регистр B
001 = Регистр C
010 = Регистр D
011 = Регистр E
100 = Регистр L
101 = Регистр L
110 = Ячейка памяти по адресу HL
111 = Аккумулятор

Например, команде:

```
MOV L,E
```

соответствует код операции:

```
01101011
```

или 6Bh. Если хотите, проверьте по таблице.

Вероятно, где-то внутри процессора 8080 три бита *иии* используются в селекторе «8 на 1», а три бита *ццц* управляют дешифратором «3 на 8», который определяет в какой регистр направить значение.

В качестве 16-битовых пар можно использовать регистры B и C (BC) или D и E (DE). Если в одной из этих пар записан адрес ячейки памяти, в которую нужно записать или из которой нужно считать байт, используйте следующие команды.

Код	Команда	Код	Команда
02	STAX [BC],A	0A	LDAX A,[BC]
12	STAX [DE],A	1A	LDAX A,[DE]

Еще одна разновидность команды MOV обозначается мнемокодом MVI (Move Immediate, переместить непосредственно). Она состоит из 2 байтов: кода и байта данных. Этот байт переносится в один из регистров или в ячейку памяти, адрес которой записан в паре регистров HL.

Код	Команда
06	MVI B,xx
0E	MVI C,xx
16	MVI D,xx
1E	MVI E,xx
26	MVI H,xx
2E	MVI L,xx
36	MVI [HL],xx
3E	MVI A,xx

Например, в результате выполнения команды

```
MOV E,37h
```

в регистр E записывается число 37h. Этот способ считается третьим способом адресования к памяти и называется *непосредственной адресацией* (immediate addressing).

Для выполнения четырех основных арифметических действий, знакомых нам по процессору, который мы разработали в главе 17, предназначена группа из 32 команд. Эти действия — сложение (ADD), сложение с переносом (ADC), вычитание (SUB) и вычитание с заимствованием (SBB). Во всех случаях один из операндов извлекается из аккумулятора; туда же помещается результат.

Код	Команда	Код	Команда
80	ADD A,B	90	SUB A,B
81	ADD A,C	91	SUB A,C
82	ADD A,D	92	SUB A,D
83	ADD A,E	93	SUB A,E
84	ADD A,H	94	SUB A,H
85	ADD A,L	95	SUB A,L
86	ADD A,[HL]	96	SUB A,[HL]
87	ADD A,A	97	SUB A,A
88	ADC A,B	98	SBB A,B
89	ADC A,C	99	SBB A,C
8A	ADC A,D	9A	SBB A,D

(*продолжение*)

Код	Команда	Код	Команда
8B	ADC A,E	9B	SBB A,E
8C	ADC A,H	9C	SBB A,H
8D	ADC A,L	9D	SBB A,L
8E	ADC A,[HL]	9E	SBB A,[HL]
8F	ADC A,A	9F	SBB A,A

Допустим, в аккумуляторе записан байт 35h, а в регистре B — байт 22h. После выполнения команды:

```
SUB A,B
```

аккумулятор содержит байт 13h.

Если в регистре A записан байт 35h, в регистре H — 10h, в регистре L — 7Ch, а в ячейке памяти по адресу 107Ch — байт 4Ah, то выполнение команды:

```
ADD A,[HL]
```

приведет к сложению содержимого аккумулятора (35h) и содержимого ячейки памяти, адресуемой парой регистров H и L, т. е. 4Ah. Результат (7Fh) будет помещен в аккумулятор.

С помощью команд ADC и SBB микросхема 8080 способна складывать и вычитать числа, разрядность которых равна 16, 24, 32 и т. д. Например, если два 16-битовых числа записаны в пары регистров BC и DE, то для их сложения и помещения результата в пару регистров BC вам понадобится такая последовательность команд:

```
MOV A,C   ; Младший байт
ADD A,E
MOV C,A
MOV A,B   ; Старший байт
ADC A,D
MOV B,A
```

Для сложения используются две команды: ADD для младшего байта и ADC для старшего. Бит переноса, который может появиться в результате первого сложения, учитывается при втором сложении. Одно из слагаемых всегда обязательно располагается в аккумуляторе, поэтому даже в такой короткой про-

грамме команда MOV используется четырежды. Вообще в программах для 8080 команды MOV всегда представлены в изобилии.

Настало время поговорить о флажках процессора 8080. В главе 17 мы использовали два флажка — переноса и нуля. В микросхеме 8080 их на 3 больше — добавляются флажки знака (Sign), четности (Parity) и вспомогательного переноса (Auxiliary Carry). Для хранения всех флажков предназначен особый 8-битовый регистр — *слово состояния программы* (Program Status Word, PSW). Команды LDA, STA и MOV на значения флажков не влияют. Команды ADD, SUB, ADC и SBB изменяют значения флажков следующим образом.

- Флажок знака устанавливается в 1, если старший бит результата равен 1 (результат отрицателен).
- Флажок нуля устанавливается в 1, если результат равен 0.
- Флажок четности устанавливается в 1, если результат *четен*, т. е. четно число 1 в двоичном представлении результата. В 0 флажок четности устанавливается, если результат *нечетен*. Проверку четности иногда используют для контроля корректности результата, хотя в программировании 8080 этот флажок практически не применялся.
- Флажок переноса устанавливается в 1, если выполнение команды ADD или ADC привело к переносу, а также если выполнение команды SUB или SBB *не привело* к его появлению (в компьютере из главы 17 поведение флажка переноса подчинялось другим правилам).
- Флажок дополнительного переноса устанавливается в 1, если выполнение команды DAA (о ней чуть позже) привело к переносу из младшей тетрады в старшую.

С помощью двух команд значение флажка переноса можно изменить непосредственно:

Код	Команда	Действие
37	STC	Установить флажок переноса в 1
3F	CMC	Заменить значение флажка его дополнением

В дополнение к арифметическим командам ADD, ADC, SUB и SBB, которые были доступны и компьютеру из главы 17 (хотя и не с той степенью гибкости), процессор 8080 способен вы-

полнять и логические операции И, ИЛИ и «Исключающее ИЛИ». Как арифметические, так и логические операции выполняются в арифметико-логическом устройстве процессора.

Код	Команда	Код	Команда
A0	AND A,B	B0	OR A,B
A1	AND A,C	B1	OR A,C
A2	AND A,D	B2	OR A,D
A3	AND A,E	B3	OR A,E
A4	AND A,H	B4	OR A,H
A5	AND A,L	B5	OR A,L
A6	AND A,[HL]	B6	OR A,[HL]
A7	AND A,A	B7	OR A,A
A8	XOR A,B	B8	CMP A,B
A9	XOR A,C	B9	CMP A,C
AA	XOR A,D	BA	CMP A,D
AB	XOR A,E	BB	CMP A,E
AC	XOR A,H	BC	CMP A,H
AD	XOR A,L	BD	CMP A,L
AE	XOR A,[HL]	BE	CMP A,[HL]
AF	XOR A,A	BF	CMP A,A

Команды AND, OR и XOR выполняются *побитово*, т. е. действуют независимо на каждую пару битов. Например, после выполнения команд:

```
MVI A,0Fh
MVI B,55h
AND A,B
```

содержимое аккумулятора будет равняться 05h. Если бы последняя команда была OR, в аккумулятор было бы записано число 5Fh. Наконец, результатом команды XOR стало бы число 5Ah.

Действие команды CMP (Compare, сравнить) аналогично действию команды SUB с единственным исключением — результат не сохраняется в аккумуляторе. Иначе говоря, команда CMP вычитает одно число из другого и тут же забывает ре-

зультат. В чем ее смысл? Во флажках! Они расскажут вам об отношениях между числами, которые вы сравниваете. Рассмотрим в качестве примера команды:

```
MVI B,25h
CMP A,B
```

После их выполнения содержимое аккумулятора не изменится. Но если оно равно 25h, будет установлен флажок нуля, а если оно меньше 25h — флажок переноса.

У восьми арифметических и логических команд имеются также версии для работы непосредственно с байтами.

Код	Команда	Код	Команда
C6	ADI A,xx	E6	ANI A,xx
CE	ACI A,xx	EE	XRI A,xx
D6	SUI A,xx	F6	ORI A,xx
DE	SBI A,xx	FE	CPI A,xx

Так показанную выше пару команд можно заменить одной:

```
CPI A,25h
```

Еще две команды для работы с аккумулятором:

Код	Команда
27	DAA
2F	CMA

Команда CMA (Complement Accumulator, дополнить аккумулятор) вычисляет дополнение содержимого аккумулятора до 1 — все нули превращаются в единицы, а все единицы — в нули. Если до выполнения команды CMA в аккумуляторе записано число 01100101, то после в нем будет число 10011010. Дополнение аккумулятора до 1 вычисляет также команда:

```
XRI A,FFh
```

Команда DAA (Decimal Adjust Accumulator, десятичная коррекция аккумулятора) — вероятно в наборе команд 8080 самая сложная. Ее выполнением в микропроцессоре занимается специально предназначенное для этого устройство.

Команда DAA помогает программисту осуществлять арифметические операции с десятичными числами, представленными в кодировке BCD (binary-coded decimal, десятичное в двоичной кодировке). В данных, закодированных с помощью BCD, каждая тетрада может принимать значения только от 0000 до 1001, символизирующие десятичные цифры от 0 до 9. В формате BCD 8 битов байта используются для хранения двух десятичных цифр.

Допустим, в аккумуляторе хранится BCD-значение 27h, соответствующее десятичному числу 27 (в обычной ситуации шестнадцатеричное число 27h равно десятичному 39), а в регистре B — BCD-значение 94h. После выполнения команд:

```
MVI A,27h
MVI B,94h
ADD A,B
```

аккумулятор будет содержать число BBh, которое, конечно, не может быть числом в формате BCD, так как обе его тетрады превышают 1001. Вот тут-то на помощь и приходит команда DAA. После ее выполнения в аккумулятор записывается число 21h и устанавливается флажок переноса, поскольку в десятичном исчислении 27 + 94 = 121. При работе с BCD-арифметикой команда DAA очень полезна.

При программировании довольно часто возникает необходимость прибавить к числу 1 или вычесть из него 1. Например, в программе умножения из главы 17 нам нужно было вычитать 1 из числа, для чего мы складывали его с FFh, т. е. с дополнением до 2 числа –1. В наборе команд 8080 для уменьшения или увеличения на 1 числа в регистре или ячейке памяти предусмотрены специальные команды.

Код	Команда	Код	Команда
04	INR B	05	DCR B
0C	INR C	0D	DCR C
14	INR D	15	DCR D
1C	INR E	1D	DCR E
24	INR H	25	DCR H
2C	INR L	2D	DCR L

(*продолжение*)

34	INR [HL]	35	DCR [HL]
3C	INR A	3D	DCR A

Они действуют на все флажки, кроме флажка переноса.

В набор команд 8080 входят четыре команды *циклического сдвига* (rotate), сдвигающие содержимое аккумулятора на 1 бит влево или вправо.

Код	Команда	Действие
07	RLC	Сдвинуть аккумулятор влево
0F	RRC	Сдвинуть аккумулятор вправо
17	RAL	Сдвинуть аккумулятор влево через разряд переноса
1F	RAR	Сдвинуть аккумулятор вправо через разряд переноса

Из всех флажков они действуют только на флажок переноса.

Допустим, аккумулятор содержит число A7h, или 10100111 в двоичном представлении. RLC сдвигает биты влево. Самый старший бит («выталкиваемый» из числа «сверху») становится самым младшим и определяет состояние флажка переноса. В данном случае в результате сдвига получится число 01001111, а флажок переноса будет установлен в 1. RRC таким же образом сдвигает число вправо. Результат ее действия на число 10100111 будет равен 11010011, флажок переноса также будет установлен в 1.

Команды RAL и RAR действуют немного иначе. RAL сдвигает содержимое аккумулятора влево, записывает во флажок переноса содержимое старшего бита аккумулятора, а предыдущее значение флажка переноса записывает в младший бит аккумулятора. Например, если аккумулятор содержит число 10100111 и флажок переноса равен 0, то после выполнения RAL в аккумуляторе будет записано число 01001110, а флажок переноса равен 1. При тех же начальных условиях команда RAR приведет к установке флажка переноса в 1 и записи в аккумулятор числа 01010011.

Команды сдвига очень удобны при умножении или делении числа на 2, что соответствует сдвигу влево или вправо.

Память, к которой обращается процессор, называется памятью с *произвольным доступом* не случайно. Для обращения к любой ячейке памяти процессору достаточно указать ее адрес. Память RAM подобна книге, которую можно открыть на любой странице, в отличие от недельной подшивки газет на микропленке. Чтобы добраться до субботнего выпуска, надо промотать пленку за большую часть недели. Такие устройства (микропленки, магнитофонные ленты и т. п.) называются устройствами с *последовательным доступом* (sequential access).

Однако временами удобнее оказывается запоминающее устройство, доступ к которому нельзя назвать ни произвольным, ни последовательным. Рассмотрим такую жизненную ситуацию. Вы — самый младший служащий в фирме, которому все вышестоящие работники дают поручения, складывая на стол папки с документами, с которыми что-то нужно сделать. Порой вы обнаруживаете, что не можете продолжить работу с одной папкой, пока не обработаете другую, связанную с ней папку. Вы кладете первую папку на стол, а вторую раскрываете поверх нее. Но вот к вашему столу приближается начальник и передает вам третью папку, работу над которой нужно закончить срочно. Вы, естественно, располагаете ее поверх предыдущих двух. Но вот беда — для выполнения задания вам нужны документы из еще одной папки, и вот стопка на столе состоит уже из четырех папок…

Если задуматься, то по этой стопке легко можно отследить все выполняемые вами задания. Сверху лежит самая важная папка. Закончив работу с ней, вы обращаетесь к следующей папке и т. д. Покончив с самой нижней папкой (с нее вы начали рабочий день), вы вправе отправляться домой.

С технической точки зрения такой способ хранения информации называется *стеком* (stack). «Стопка» информации растет снизу вверх и убывает сверху вниз. Данные в стеке организованы по принципу «последним вошел — первым вышел» (Last In First Out, LIFO). То, что было помещено в стек в последнюю очередь, первым извлекается из него. То, что было помещено в стек в первую очередь, извлекается из него последним.

В компьютерах стек используется, конечно, не для хранения бумаг, а для хранения чисел, что, впрочем, не менее удобно. Помещение информации в стек называется «*вталкиванием*» (push), а ее извлечение — «*выталкиванием*» (pop).

Допустим, вы пишете на языке ассемблера программу, данные для которой хранятся в регистрах A, B и C. На каком-то этапе вы замечаете, что для выполнения очередного расчета вам нужно использовать эти регистры, а затем восстановить в них исходные значения.

Конечно, ничто не мешает вам сохранить числа из регистров в ячейках памяти, а затем вернуть их из памяти обратно в регистры. Но при этом придется следить за тем, в каких ячейках оказались эти числа, чтобы случайно не записать поверх них другую информацию. Гораздо безопаснее на время поместить («втолкнуть») числа в стек:

```
PUSH A
PUSH B
PUSH C
```

Суть действия этих команд я объясню чуть позже. Пока нам достаточно знать, что они каким-то образом сохраняют содержимое регистров в памяти LIFO. После их выполнения вы вольны использовать регистры A, B и C по своему усмотрению. Когда надобность в них отпадает, вы извлекаете числа из стека в обратном порядке:

```
POP C
POP B
POP A
```

Помните: последним вошел — первым вышел. Изменив порядок команд POP, вы скорее всего получите ошибку.

Стек особенно удобен тем, что его можно использовать в различных фрагментах программы, не заботясь при этом о потенциальных проблемах. Если вы, поместив в стек содержимое регистров A, B и C, решили проделать ту же операцию с регистрами D и E, используйте команды:

```
PUSH D
PUSH E
```

Для восстановления содержимого регистров необходимо ввести в программу команды:

```
POP E
POP D
```

до того как предыдущий фрагмент начнет извлекать из стека содержимое регистров A, B и C.

Как работает стек? Для него выделяется область памяти, не занятая другими данными. Для адресации этой области памяти в микропроцессор 8080 включен специальный 16-битовый регистр — *указатель стека* (Stack Pointer, SP).

Приведенные выше примеры записи и извлечения из стека содержимого регистров в случае 8080 не совсем корректны. Команда PUSH в этом процессоре записывает в стек *16-битовое* значение, и такое же значение извлекается из стека командой POP. Поэтому вместо простых команд PUSH C и POP C в этом процессоре применяются 8 команд:

Код	Команда	Код	Команда
C5	PUSH BC	C1	POP BC
D5	PUSH DE	D1	POP DE
E5	PUSH HL	E1	POP HL
F5	PUSH PSW	F1	POP PSW

Команда PUSH BC записывает в стек регистры B и C, а POP BC соответственно извлекает их. В 8-битовом регистре PSW, как вы помните, хранится слово состояния программы, т. е. флажки. Две команды в последней строке таблицы в действительности сохраняют в стеке содержимое не только регистра PSW, но и аккумулятора. Чтобы сохранить в стеке *все* регистры и флажки, используйте команды:

```
PUSH PSW
PUSH BC
PUSH DE
PUSH HL
```

а для восстановления этой информации:

```
POP HL
POP DE
POP BC
POP PSW
```

Чтобы разобраться в работе стека, допустим, что в какой-то момент времени содержимое указателя стека равно 8000h.

Выполнение команды PUSH BC в действительности состоит из четырех шагов.

- Указатель стека уменьшается на 1, становясь равным 7FFFh.
- Содержимое регистра B сохраняется в ячейку с адресом из указателя стека, т. е. 7FFFh.
- Указатель стека уменьшается еще на 1, становясь равным 7FFEh.
- Содержимое регистра C сохраняется в ячейку с адресом из указателя стека, т. е. 7FFEh.

Команда POP BC прокручивает все эти действия в обратном порядке (при условии, что в указателе стека все еще записан адрес 7FFEh).

- В регистр C загружается число из ячейки, адрес которой (7FFEh) записан в указателе стека.
- Указатель стека увеличивается на 1, становясь равным 7FFFh.
- В регистр B загружается число из ячейки, адрес которой (7FFFh) записан в указателе стека.
- Указатель стека увеличивается на 1, становясь равным 8000h.

Каждая команда PUSH увеличивает размер стека на 2 байта. В принципе не исключена ситуация, при которой стек (вероятно, из-за ошибки в программе) так вырастет, что начнет записываться поверх кодов команд и данных, нужных для выполнения программы. Это называется *переполнением стека* (stack overflow). Подобным же образом ошибочное использование лишних команд POP приводит к *исчезновению стека* (stack underflow).

Если вы подключили к процессору 8080 память емкостью 64 кб, начальное значение указателя стека удобно сделать равным 0000h. Первая команда PUSH приведет к его уменьшению на 1, т. е. запись стека начнется с адреса FFFFh, и он займет область памяти, максимально далекую от ваших программ, которые, вероятно, будут располагаться, начиная с адреса 0000h.

Задать значение указателя стека позволяет команда LXI (Load Extended Immediate, расширенная непосредственная заг-

рузка), предназначенная для записи в 16-битовую пару регистров двух байтов, следующих за кодом команды.

Код	Команда
01	LXI BC,xxxx
11	LXI DE,xxxx
21	LXI HL,xxxx
31	LXI SP,xxxx

Команда:

```
LXI BC,527Ah
```

эквивалентна командам:

```
MVI B,52h
MVI C,7Ah
```

но занимает в памяти на целый байт меньше. Последняя команда в таблице служит для записи определенного числа в регистр SP, т. е. в указатель стека. Как правило, она оказывается одной из первых команд, выполняемых процессором после перезапуска:

```
0000h: LXI SP,0000h
```

Для увеличения или уменьшения на 1 содержимого пар регистров и указателя стека служат специальные команды:

Код	Команда	Код	Команда
03	INX BC	0B	DCX BC
13	INX DE	1B	DCX DE
23	INX HL	2B	DCX HL
33	INX SP	3B	DCX SP

Раз уж я заговорил о 16-битовых командах, упомяну еще несколько. Следующие команды складывают содержимое 16-битовой пары регистров с парой HL.

Код	Команда
09	DAD HL,BC
19	DAD HL,DE

(*продолжение*)

29	DAD HL,HL
39	DAD HL,SP

Их использование также сокращает размер программы. Например, чтобы заменить DAD HL,BC, вам понадобилось бы шесть обычных команд:

```
MOV A,L
ADD A,C
MOV L,A
MOV A,H
ADC A,B
MOV H,A
```

Обычно команду DAD применяют для вычисления адресов в памяти. Из всех флажков она влияет лишь на флажок переноса.

Нам осталось рассмотреть еще несколько команд. SHLD и LHLD служат для сохранения в памяти содержимого пары регистров HL и для записи информации из памяти в эту пару регистров.

Код	Команда	Действие
22	SHLD [aaaa],HL	Сохранить содержимое HL
2A	LHLD HL,[aaaa]	Загрузить данные в HL

Число из регистра L записывается по адресу [aaaa], а число из регистра H — по адресу [aaaa + 1].

В программный счетчик (Program Counter, PC) и указатель стека можно записать число из пары регистров HL:

Код	Команда	Действие
E9	PCHL PC,HL	Записать число из HL в PC
F9	SPHL SP,HL	Записать число из HL в SP

PCHL в действительно является разновидностью команды перехода, так как следом за ней будет выполнена команда, адрес которой записан в паре регистров HL. Команда SPHL предоставляет альтернативный способ задать значение указателя стека.

Две следующие команды позволяют поменять местами содержимое пары регистров HL либо с «верхним» элементом стека, либо с парой регистров DE.

Код	Команда	Действие
E3	XTHL HL,[SP]	Поменять местами HL и «верхний» элемент стека
EB	XCNG HL,DE	Поменять местами HL и DE

Теперь о командах ветвления. Как вы помните из главы 17, в процессоре имеется специальный регистр — программный счетчик, — куда записан адрес следующей команды, которую должен выполнить процессор. Обычно команды выполняются последовательно, согласно их расположению в памяти. Но некоторые — а именно команды перехода — позволяют процессору отклониться от этого генерального курса. Их выполнение приводит к записи в программный счетчик значения, выпадающего из общей последовательности.

Хотя отказываться от старого доброго безусловного перехода никто вас не призывает, куда удобнее оказываются команды *условного* перехода, действие которых зависит от состояния определенных флажков, например, флажка переноса или флажка нуля. Именно способность осуществлять условный переход превратила автоматический сумматор из главы 17 в настоящий многоцелевой компьютер.

У процессора 8080 пять флажков, четыре из которых используются в командах условного перехода. Всего в наборе 8080 предусмотрено 9 команд перехода: одна для безусловного перехода и еще 8 для перехода в зависимости от равенства 0 или 1 флажков нуля, переноса, четности и знака.

Прежде чем познакомить вас с ними, я хотел бы ввести еще два типа переходов. Первый из них осуществляет команда CALL (вызов). Она приводит в общем-то к обычному условному переходу с одним исключением: прежде чем записать в программный счетчик новое значение, процессор сохраняет адрес, который был в нем до этого. Где? Конечно, в стеке!

Это означает, что в процессоре сохраняется информация, *откуда был совершен переход*. Сохраненный адрес позволяет процессору вернуться к выполнению прерванной последовательности команд. Для этого обратного перехода служит команда RET (Return, вернуться). Она извлекает 2 байта из стека и помещает их в программный счетчик.

Наличие команд, подобных CALL и RET, в арсенале любого процессора очень важно, так как они позволяют создавать

подпрограммы, т. е. выделять часто используемые фрагменты кода («часто» в данном случае означает «больше одного раза»). Подпрограммы являются главным организующим элементом программы на языке ассемблера.

Рассмотрим пример. Разрабатывая программу на языке ассемблера, вы столкнулись с необходимостью перемножить два байта. Естественно, вы вставляете в код фрагмент, который решает эту задачу, и продолжаете работать. Вскоре потребность в умножении возникает опять. Конечно, можно использовать те же команды, что и раньше. Но нужно ли второй раз вставлять их в программу? Думаю, нет. Это же напрасная трата памяти! Разумно было бы использовать для умножения уже введенные команды, но как к ним перейти? Обычная команда безусловного перехода здесь не подойдет, так как она не позволяет после умножения вернуться обратно. Вот здесь-то в действие и вступают команды CALL и RET.

Набор команд, осуществляющих умножение двух байтов, — прекрасный пример подпрограммы. Рассмотрим ее подробнее. В главе 17 перемножаемые байты и произведение хранились в ячейках памяти. В подпрограмме для процессора 8080 мы разместим множители в регистрах B и C, а произведение запишем в пару регистров HL.

```
Multiply: PUSH PSW       ; Сохраняем регистры, которые
          PUSH BC        ; будут меняться

          SUB H,H        ; Записываем 0000h в HL (результат)
          SUB L,L

          MOV A,B        ; Записываем множитель в A
          CPI A,00h      ; Если он 0, завершаем
          JZ AllDone

          MVI B,00h      ; Записываем 0 в старший байт BC

MultLoop: DAD HL,BC      ; Складываем HL и BC
          DEC A          ; Уменьшаем множитель на 1
          JNZ MultLoop   ; Возврат на начало цикла, если
                         ; не 0
```

```
AllDone:  POP BC ; Восстанавливаем регистры
          POP PSW
          RET    ; Возврат
```

Первая строчка подпрограммы начинается с метки Multiply, соответствующей адресу в памяти, по которому расположена подпрограмма. Начинается выполнение подпрограммы с двух команд PUSH. Обычно регистры, используемые в подпрограмме, желательно сохранять, а после окончания ее работы — восстанавливать.

Затем подпрограмма обнуляет содержимое регистров H и L. То же действие можно выполнить и с помощью команд MVI, но их понадобилось бы 4, а не 2, как команд SUB. По завершении работы подпрограммы пара регистров HL будет содержать искомое произведение.

Следующий шаг: перемещение содержимого регистра B в аккумулятор и проверка его равенства 0. Если множитель равен 0, произведение также равно 0, и работу подпрограммы можно считать законченной. В регистры H и L ноль уже записан, так что можно с помощью команды JZ (переход, если 0) переходить прямо к двум завершающим командам POP.

Если множитель 0 не равен, программа обнуляет регистр B. Теперь один из множителей содержится в аккумуляторе, второй — в паре регистров BC. Команда DAD складывает множитель (BC) с результатом (HL). Значение множителя в аккумуляторе уменьшается на 1. Если результат не равен 0, команда JNZ (переход, если не 0) приводит к повторному сложению BC и HL. Выполнение этого короткого цикла продолжается, пока содержимое аккумулятора не обратится в 0 (отмечу, что более эффективную программу умножения для процессора 8080 можно написать с помощью команд сдвига).

Если в основной программе возникла необходимость умножить 25h на 12h, это делают команды:

```
MVI B,25h
MVI C,12h
CALL Multiply
```

Команда CALL записывает в стек содержимое программного счетчика, т. е. адрес команды, стоящей *следом* за командой CALL. Затем происходит переход к команде, на которую ука-

зывает метка Multiply, т. е. к началу подпрограммы. Когда произведение в подпрограмме вычислено, выполняется команда RET, в результате чего в программный счетчик возвращается значение из стека. Далее выполняется команда, идущая за CALL.

В набор команд процессора 8080 входят также условные команды вызова подпрограммы и возвращения из нее, но используются они гораздо реже обычных команд условного перехода:

Условие	**Код**	**Команда**	**Код**	**Команда**	**Код**	**Команда**
Нет	C9	RET	C3	JMP aaaa	CD	CALL aaaa
Не ноль	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
Ноль	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
Нет переноса	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
Есть перенос	D8	RC	DA	JC aaaa	DC	CC aaaa
Результат нечетный	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
Результат четный	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
Результат положителен	F0	RP	F2	JP aaaa	F4	CP aaaa
Результат отрицателен	F8	RM	FA	JM aaaa	FC	CM aaaa

Вы, вероятно, и без меня знаете, что к процессору подключают не только память. Компьютер — ничто без устройств ввода и вывода, позволяющих ему обмениваться информацией с окружающим миром. Самые популярные устройства ввода и вывода — клавиатура и дисплей.

Как процессор обменивается информацией с *периферийными* (peripheral) устройствами, т. е. со всеми подключенными к нему устройствами, кроме памяти? Обычно внешние устройства конструируют так, что с точки зрения подключения к процессору они ведут себя подобно памяти. Микропроцессор считывает информацию из устройства и записывает ее, задавая адрес, который соответствует этому устройству. В не-

которых микропроцессорах адреса внешним устройствам выделяются из обычного адресного пространства. Такая организация называется *вводом-выводом с распределением памяти* (memory-mapped input/output). В процессоре 8080 для устройств ввода-вывода 256 адресов зарезервированы в дополнение к основной памяти с 65 536 адресами. Эти дополнительные адреса называются *портами ввода-вывода* (input/output ports). Адреса портов подаются на адресные линии с A_0 по A_7. От адресов памяти их с помощью системного контроллера 8228 отличают специальные управляющие сигналы.

Запись содержимого аккумулятора в порт осуществляет команда OUT. Адрес порта указывается вслед за ее кодом. Для считывания байта в аккумулятор предназначена команда IN.

Код	Команда
D3	OUT pp
DB	IN pp

В некоторых случаях периферийным устройствам надо «привлечь внимание» процессора. Например, когда вы нажимаете клавишу на клавиатуре, процессору, как правило, желательно знать об этом сразу. Осуществляет это механизм *прерываний* (interrupts) — сигналов, поступающих в процессор 8080 через вход INT.

После перезапуска процессор 8080 на прерывания не реагирует. Чтобы разрешить прерывания, в программу нужно вставить команду EI (Enable Interrupts, разрешить прерывания), а чтобы отказаться от реагирования на них — DI (Disable Interrupts, запретить прерывания).

Код	Команда
F3	DI
FB	EI

Когда прерывания разрешены, подается сигнал на выход процессора INTE. Когда устройству нужно прервать работу процессора, оно подает сигнал на вход INT. Процессор извлекает из памяти очередную команду программы, но выполняет не ее, а одну из следующих:

Код	Команда	Код	Команда
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
D7	RST 2	F7	RST 6
DF	RST 3	FF	RST 7

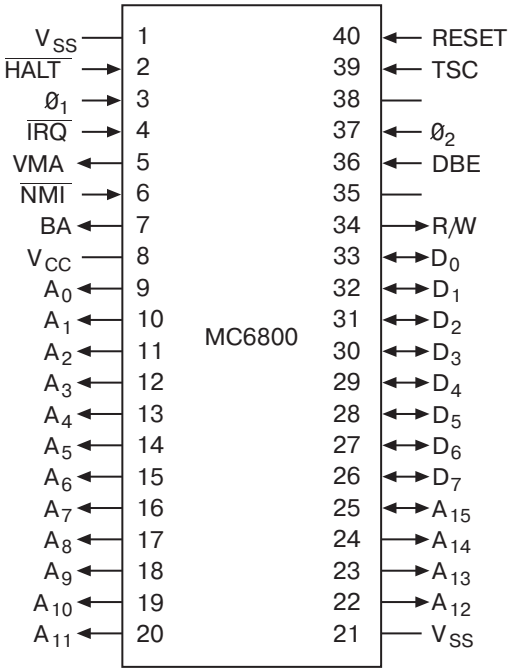
Действуют они подобно команде CALL в том смысле, что при их выполнении в стек записывается содержимое программного счетчика. Но переход при этом осуществляется по строго определенным адресам: команда RST 0 передает управление команде по адресу 0000h, RST 1 — команде по адресу 0008h и т. д. до RST 7, которая передает управление команде по адресу 0038h. По этим адресам должны располагаться фрагменты кода, реагирующие на прерывания. Допустим, что прерывание от клавиатуры привело к срабатыванию RST 4. Значит, по адресу 0020h должны располагаться команды для считывания с клавиатуры введенного байта (подробнее об этом в главе 21).

Я описал 243 команды. Неиспользованными остались коды 08h, 10h, 18h, 20h, 28h, 30h, 38h, CBh, D9h, DDh, EDh и FDh. В заключение упомяну еще один код и соответствующую команду.

Код	Команда
00	NOP

NOP (No Operation, нет операции) не выполняет никаких действий. Зачем она нужна? Например, для заполнения пустых ячеек.

Обсуждать с теми же подробностями процессор 6800 фирмы Motorola я не собираюсь, поскольку он как по конструкции, так и по действию довольно похож на процессор 8080. Вот какие у него есть выводы.



Контакт V_{SS} соединяется с землей, контакт V_{CC} — с питанием 5 В. Подобно микросхеме 8080, у процессора 6800 имеется 16 адресных входов и 8 сигналов для данных, применяемых как для ввода, так и для вывода информации. На вход $\overline{IRQ}$ подается *запрос на прерывание* (interrupt request). Порты ввода-вывода в 6800 не используются. Адреса устройств берутся из обычного адресного пространства.

Есть в процессоре 6800 16-битовый программный счетчик, 16-битовый указатель стека, 8-битовый регистр состояния (для флажков) и два 8-битовых аккумулятора A и B. B считается именно аккумулятором, а не регистром, поскольку с ним можно выполнять все те же операции, что и с A. Других 8-битовых регистров в процессоре нет.

16-битовые значения в 6800 хранятся в *индексном регистре* (index register), подобном паре регистров HL в 8080. Адрес для многих команд вычисляется суммированием содержимого индексного регистра и байта, следующего за кодом команды.

Хотя набор команд процессора 6800 в целом идентичен набору процессора 8080, очевидно, что численные и мнемонические коды этих команд совершенно различны. Вот, например, как выглядят в наборе 6800 команды перехода.

Код	Команда	Действие
20	BRA	Переход
22	BHI	Переход, если больше
23	BLS	Переход, если меньше или равно
24	BCC	Переход, если нет переноса
25	BCS	Переход, если есть перенос
26	BNE	Переход, если не равно
27	BEQ	Переход, если равно
28	BVC	Переход, если нет переполнение
29	BVS	Переход, если есть переполнение
2A	BPL	Переход, если плюс
2B	BMI	Переход, если минус
2C	BGE	Переход, если больше или равно 0
2D	BLT	Переход, если меньше 0
2E	BGT	Переход, если больше 0
2F	BLE	Переход, если меньше или равно 0

Флажка четности в 6800 нет, зато в отличие от 8080 есть флажок переполнения. Действие некоторых команд перехода зависит от *сочетания* флажков.

Различие наборов команд 8080 и 6800 удивления, *разумеется*, не вызывает. Две этих микросхемы разрабатывались практически одновременно двумя группами инженеров двух различных компаний. Поэтому они и *несовместимы*: машинные коды одного процессора не работают на другом процессоре. Даже программу на языке ассемблера, написанную для одного процессора, нельзя перевести в машинные коды другого процессора. О программах, которые работают на разных процессорах, мы поговорим в главе 24.

Между процессорами 8080 и 6800 есть и другое интересное отличие. В обоих наборах есть команда LDA, загружающая в аккумулятор содержимое ячейки памяти. В 8080, например, следующая последовательность байтов:

3Ah	Команда LDA процессора 8080
7Bh	
34h	

приведет к записи в аккумулятор байта из ячейки с адресом 347Bh. Теперь сравните это с командой LDA из набора процессора 6800 в так называемом расширенном режиме адресации:

B6h	Команда LDA процессора 6800
7Bh	
34h	

Эта последовательность байтов приведет к записи в аккумулятор байта из ячейки с адресом 7B34h.

В различии кодов команд (3Ah в 8080 и B6h в 6800) нет, как мы уже говорили, ничего удивительного. Но процессоры также по-разному обрабатывают байты, которые следуют за кодом команды. В программе для процессора 8080 считается, что сначала за кодом идет младший байт адреса, а потом — старший. В программе же для процессора 6800 сначала идет старший байт!

Это фундаментальное различие в способе хранения многобайтовых величин между процессорами фирм Intel и Motorola *так и не устранено*. И в наши дни в процессорах фирмы Intel многобайтовые величины хранятся, начиная с младшего байта, а в процессорах фирмы Motorola — начиная со старшего.

Помните из-за чего Лилипутия воевала с государством Блефуску? Из-за разногласий по поводу того, с какого конца надлежит разбивать вареные яйца: с острого или тупого. Споры между сторонниками двух способов записи многобайтовых

величин сродни спорам между остроконечниками и тупоконечниками (так Джонатан Свифт назвал приверженцев враждующих партий) и настолько же лишены смысла (хотя должен признаться, что мне самому способ, использованный в компьютере из главы 17, не очень нравится!). Сами по себе оба метода «правильны», но различие между ними создает большие трудности при необходимости переносить информацию с компьютеров остроконечников (Intel) на компьютеры тупоконечников (Motorola).

Какая судьба ожидала процессоры 8080 и 6800? Первый стал основой устройства, которое иногда называют первым персональным компьютером, хотя правильнее было бы называть его первым *домашним* компьютером. Это устройство — компьютер «Альтаир 8800», в январе 1975 г. украсивший обложку журнала «Popular Electronics».

Взгляните на него: лампочки, переключатели — все это нам знакомо, правда? Именно так выглядел пульт управления памятью, который я описал вам в главе 16.

За процессором 8080 последовали чипы 8085 и Z-80 фирмы Zilog — конкурента корпорации Intel, основанного быв-

шим работником последней Федерико Фаггином (Federico Faggin), который принимал активное участие в работе над микросхемой 4004. Процессор Z-80 был полностью совместим с 8080, но отличался от него наличием множества очень полезных дополнительных команд.

В 1977 г. появился компьютер «Apple II» компании Apple Computer, основанной Стивеном Джобсом (Steven Jobs) и Стефаном Возняком (Stephen Wozniak). В нем был применен усовершенствованный вариант 6800 — дешевый микрочип 6502 фирмы MOS Technology.

В июне 1978 г. фирма Intel выпустила 16-битовый процессор 8086 с адресным пространством 1 Мб. Его машинные коды были несовместимы с кодами 8080, зато включали специальные команды для умножения и деления. Годом позже появился процессор 8088, внутренне идентичный 8086, но адресовавший память побайтово. Это позволяло использовать в сочетании с ним широко распространенные вспомогательные микросхемы, разработанные для процессора 8080. На процессоре 8088 работал персональный компьютер 5150 фирмы IBM (более известный как IBM PC), появившийся осенью 1981 г.

IBM оказала на рынок персональных компьютеров громадное влияние. Многие компании занялись выпуском компьютеров, совместимых с IBM PC (смысл понятия «совместимость» я разъясню в следующих главах). В 1982 г. семейство процессоров x86 пополнилось чипами 186 и 286, в 1985 г. появился 32-разрядный процессор 386, в 1986 г. — 486. С 1993 г. в IBM-совместимых компьютерах используются процессоры Pentium фирмы Intel. Хотя наборы команд этих процессоров постоянно расширяются, в них неизменно поддерживаются и команды всех предыдущих версий, начиная с 8086.

Прямым наследником процессора 6800 стал 16-разрядный чип 68000 фирмы Motorola, ставший в 1984 г. основой компьютера «Apple Macintosh». Этот процессор и последовавшие за ним процессоры той же серии (иногда ее называют 68K) и по сей день имеют многочисленных поклонников.

С 1996 г. в компьютерах «Macintosh» используется микропроцессор PowerPC, разработанный совместно компаниями Motorola, IBM и Apple. Он построен с использованием архитектуры RISC (Reduced Instruction Set Computing, вычисления

с сокращенным набором команд), в которой предпринята попытка повысить быстродействие процессора за счет его упрощения в некоторых отношениях. В RISC-компьютере все команды, как правило, имеют одну и ту же длину (в PowerPC она равна 32 битам), доступ к памяти органичен командами загрузки и сохранения, и вообще команды выполняют максимально простые действия. Обычно RISC-компьютеры снабжены большим числом регистров, чтобы избежать частых обращений к памяти.

Набор команд PowerPC кардинально отличается от набора серии 68K, поэтому запустить на PowerPC программу для процессора 68K нельзя. Однако в процессорах PowerPC, устанавливаемых на компьютерах «Apple Macintosh», предусмотрена возможность *эмуляции* процессора 68K. Программа-эмулятор по одному просматривает все машинные коды программы для 68K и выполняет соответствующее действие. Это, конечно, не так быстро, как программа, изначально написанная для PowerPC, но работает.

Согласно закону Мура количество транзисторов в микропроцессорах удваивается каждые полтора года. Для чего используются все эти дополнительные транзисторы?

Некоторые из них позволили повысить разрядность процессора, другие — расширить набор его команд. В большинстве современных микропроцессоров имеются специальные команды для арифметических операций с вещественными числами (подробнее в главе 23). Специальные команды разработаны для выполнения многократных вычислений, потребность в которых возникает при отображении на экране компьютера графических изображений (в том числе движущихся).

В современных процессорах используется несколько методов повышения быстродействия, в частности, *конвейеризация* (pipelining). Выполняя одну инструкцию, процессор заранее считывает следующую, пытаясь предсказать возможное ветвление программы. Еще одно нововведение — *кэш* (cache), сверхбыстрая память внутри процессора, в которой хранятся недавно выполнявшиеся команды. Когда в программе встречается короткий цикл, кэш позволяет ускорить его выполнение за счет того, что команды не нужно многократно считывать из памяти. Для всех этих усовершенствований необходимы новые логические схемы, а значит, новые транзисторы.

Я уже говорил, что процессор — лишь часть завершенной компьютерной системы (хотя и самая важная). Такую систему мы спроектируем в главе 21, а пока обратимся к тому, как хранить в памяти нечто, отличное от кодов команд и чисел. Итак, возвращаемся в первый класс и учимся читать и писать.

Глава 20

ASCII — символы нашего времени

В памяти компьютера могут содержаться только биты, поэтому любую информацию, которую мы рассчитываем сохранить, сначала нужно преобразовать в цифровую форму. В том, что битами можно представлять числа и команды, мы уже убедились. Пора заняться текстом. Ведь именно в виде текста — книг, журналов, газет — накоплена большая часть информации в нашем мире. Конечно, со временем мы захотим хранить на компьютере и звуки, и рисунки, и видеозаписи, но начинать нужно с самого легкого — с букв.

Для представления текста в цифровом формате мы должны придумать систему кодирования, в которой каждой букве соответствовал бы уникальный код. Свои коды понадобятся и для цифр, и для знаков препинания, поскольку без них в тексте не обойтись. Короче, нам нужны коды для всех *буквенно-цифровых* (alphanumeric) символов. Такие системы иногда называют *наборами кодированных символов* (coded character set), а отдельно взятый код — *кодом символа* (character code).

Прежде всего мы должны задаться вопросом: а сколько битов нам понадобится для этих кодов? Вопрос не из легких.

Размышляя о цифровом представлении текста, не будем забегать вперед. Текст, который мы привыкли видеть на страницах книг, журналов и газет, как правило, отформатирован

— по меньшей мере разбит на абзацы со строками заданной длины. Но для понимания смысла текста это форматирование несущественно. Прочитав рассказ в журнале, а затем много лет спустя встретив его в книге, мы не считаем, что рассказ изменился только потому, что в книге он разбит на строки большей длины.

Иными словами, откажемся пока от представления о тексте как об отформатированных абзацах на двумерной печатной странице. Будем считать его одномерным потоком букв, цифр, знаков препинания и, возможно, дополнительных символов, указывающих на окончание абзаца.

Столь же малое значение имеет и то, что в журнале и книге рассказ набран разными шрифтами. Журнальная версия начинается так:

Зовите меня Исмаил.

а книжная — так:

Зовите меня Исмаил.

Ну и стоит ли нам из-за этого переживать? Наверное, нет. Да, конечно, шрифт влияет на восприятие текста, но смысл рассказа не изменится от изменения шрифта. Кроме того, исходный шрифт всегда можно вернуть.

Продолжаем упрощать задачу. Мы откажемся не только от шрифта, но и от различных его начертаний: ни полужирного, ни курсива, ни подчеркивания, ни цвета, ни верхних и нижних индексов, ни диакритических знаков. Никаких Å, é, ñ и ö· Только простой текст, набранный 26 буквами латинского алфавита.

На самом деле с двоичным представлением букв мы уже сталкивались, когда говорили об азбуках Морзе и Брайля. Обе эти системы кодирования прекрасно работают там, где должны работать, но для компьютеров они малопригодны. В азбуке Морзе, например, коды имеют разную длину: короткими кодами обозначаются часто используемые буквы, длинными — менее популярные. Телеграфисты, конечно, этому рады, но вот в компьютере такое кодирование было бы неудобным. Кроме того, в азбуке Морзе не различаются прописные и строчные буквы.

В шрифте Брайля все символы имеют одну и ту же длину — в компьютерах это предпочтительнее. Каждая буква представлена 6 битами. Прописные и строчные буквы различаются, хотя

для этого применяется специальный *escape*-код, указывающий, что следующая за ним буква — прописная. Фактически это значит, что для кодирования прописных букв нужен не один код, а два. Цифры представляются с помощью кода переключения. Стоящие после него символы интерпретируются как цифры, пока следующий код переключения не укажет на необходимость возврата к буквенной интерпретации.

Наша цель — разработать систему кодирования, в которой предложение

У меня 27 сестер.

шифровалось бы набором кодов, состоящих из заданного числа битов. Одни коды будут обозначать буквы, другие — знаки препинания, третьи — цифры. Отдельный код понадобится даже для представления пробела между словами. В приведенном предложении 17 символов (включая пробелы). Последовательность кодов для представления таких предложений часто называют текстовой *строкой* (string).

Необходимость введения специальных кодов для цифр может показаться странной. На протяжении многих глав мы записывали числа в двоичном представлении, с какой же стати теперь не кодировать цифры 2 и 7 в нашем предложении как 10 и 111? Но на самом деле оказывается, что цифры удобно кодировать наравне с буквами, поэтому коды цифр в тексте не связаны с их реальным численным значением.

Вероятно, самым экономичным текстовым кодом можно считать 5-битовый код, разработанный в 1874 г. для печатающего телеграфного аппарата служащим французской телеграфной службы Эмилем Бодо (Emile Baudot). «На вооружение» этот код был принят в 1877 г. Позже этот код модернизировал Дональд Мюррей (Donald Murray), а в 1931 г. комитет CCITT (Comité Consultatif International Télégraphique et Téléphonique), известный теперь как Международный телекоммуникационный союз (International Telecommunication Union, ITU), принял его в качестве стандарта. Официальное название этого кода — Международный телеграфный алфавит №2 (International Telegraph Alphabet No. 2, ITA-2), но в просторечии его по-прежнему называют кодом Бодо, хотя правильнее было бы называть его кодом Мюррея.

В XX в. кодировку Бодо часто применяли в телетайпных аппаратах. Клавиатура аппарата Бодо напоминает клавиатуру

обычной пишущей машинки за исключением того, что состоит она всего из 30 клавиш и пробела. Клавиши телетайпа являются по сути электрическими переключателями. Их нажатие приводит к генерации двоичного кода и его посылке по выходному кабелю, один бит за другим. Кроме того, в телетайпный аппарат входит печатающее устройство. Коды, поступающие в телетайп по входному кабелю, управляют электромагнитами, которые отпечатывают на бумаге соответствующие символы.

Поскольку система Бодо является 5-битовой, в ней всего 32 кода, принимающих шестнадцатеричные значения от 00h до 1Fh. В таблице показано их соответствие буквам алфавита.

Шестнадцатеричный код	Буква Бодо	Шестнадцатеричный код	Буква Бодо
00		10	E
01	T	11	Z
02	Возврат каретки	12	D
03	O	13	B
04	Пробел	14	S
05	H	15	Y
06	N	16	F
07	M	17	X
08	Перевод строки	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	Переключение на цифры
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K
0F	V	1F	Переключение на буквы

Код 00h в системе не используется. Из оставшихся кодов 26 обозначают буквы латинского алфавита, и еще 5 имеют специальные значения.

Код 04h соответствует пробелу, т. е. интервалу между словами. Коды 02h и 08h называются возвратом каретки (Carriage Return) и переводом строки (Line Feed). Эта терминология происходит от пишущих машинок. Когда, печатая на машинке, вы доходите до конца строки, вам нужно сделать две вещи (обычно с помощью специального рычага или кнопки) — во-первых, сдвинуть каретку до упора вправо, чтобы начать следующую строку у левого края бумаги (это и есть возврат каретки), во-вторых, прокрутить валик, чтобы новая строка началась ниже только что напечатанной (это перевод строки). В аппаратах Бодо для генерации двух этих кодов использовались разные клавиши.

Но где же в кодировке Бодо цифры и знаки препинания? А вот для них-то и нужен код 1Bh — переключение на цифры. Все коды, следующие за ним, интерпретируются как цифры и знаки препинания, пока в последовательности кодов не попадется 1Fh — переключение на буквы. Вот какие коды используются для цифр и знаков препинания.

Шестнадцатеричный код	**Буква Бодо**	**Шестнадцатеричный код**	**Буква Бодо**
00		10	3
01	5	11	+
02	Возврат каретки	12	Кто это?
03	9	13	?
04	Пробел	14	‘
05	#	15	6
06	,	16	$
07	.	17	/
08	Перевод строки	18	-
09	)	19	2
0A	4	1A	Сигнал
0B	&	1B	Переключение на цифры
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	Переключение на буквы

В стандарте ITU коды 05h, 0Bh и 16h не заданы. Предполагается, что в каждом языке у них собственные значения. В таблице дана их расшифровка, принятая в США. В Европе эти же коды иногда используются для обозначения букв с диакритическими знаками. Получив код Сигнал, телетайп издает звуковой сигнал. В ответ на код «Кто это?» телетайп передает данные о себе.

Как и в азбуке Морзе, в кодировке БОДО прописные и строчные буквы не различаются. Предложение:

I SPENT $25 TODAY.

(Сегодня я потратил 25 долларов) кодируется последовательностью шестнадцатеричных кодов:

0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08

где используется три кода-переключателя: 1Bh перед числом, 1Fh после числа и еще один код 1Bh перед точкой. В конце строки стоят коды возврата каретки и перевода строки.

К сожалению, если вы направите эту последовательность на печатающее устройство два раза подряд, то получите в результате нечто подобное:

I SPENT $25 TODAY.

8 ’03,5 $25 TODAY.

Что произошло? В конце первой строки на печатающее устройство был подан код переключения на цифры, поэтому символы в начале второй строки интерпретируются как цифры и знаки препинания.

Подобные неприятности — прямое следствие использования кодов-переключателей. Хотя код Бодо, безусловно, очень экономичен, для цифр, знаков препинания, а также строчных и прописных букв предпочтительнее использовать отдельные коды.

Посчитаем, сколько битов нам понадобится для системы кодирования, которая была бы *лучше* системы Бодо. Итак, нам нужно 52 кода для прописных и строчных латинских букв и еще 10 кодов для цифр от 0 до 9. Это уже 62. Добавьте несколько знаков препинания, и рубеж в 64 кода перейден, т. е. 6-ю битами мы не обойдемся. С другой стороны, до следующего рубежа в 128 кодов нам пока далеко, а значит, 8 битов нам не понадобится.

Получаем ответ: для представления текстов, написанных латинскими буквами с различающимися строчными и прописными буквами, нам нужны 7-битовые коды.

Как должны выглядеть эти коды? Да как угодно. Если бы мы самостоятельно собрали компьютер и все периферийное оборудование к нему, а также самостоятельно разработали *все* программы к нему да еще никогда не пытались бы соединить его с другими компьютерами, мы могли бы придумать систему кодов самостоятельно. Для этого достаточно приписать каждому символу уникальное сочетание семи нулей и единиц.

Но в реальности компьютеры редко существуют сами по себе. Для обмена информацией всем пользователям лучше договориться и применять одни и те же коды. После этого текстовую информацию легко можно будет переносить с одного компьютера на другой.

При разработке кодов, вероятно, стоит учесть и другие соображения. Например, коды буквам лучше назначать упорядоченно: последовательные коды последовательным буквам алфавита. В будущем это весьма облегчит сортировку текста.

Стандартная система кодирования текста, конечно, уже создана. Она называется *Американским стандартным кодом для обмена информацией* (American Standard Code for Information Interchange, ASCII). Кодировка ASCII была принята в 1967 г. и по сей день остается одним из важнейших стандартов компьютерной индустрии. За одним большим исключением (о нем позже) кодировка ASCII применяется в компьютерах практически при любых работах с текстом.

ASCII — это 7-битовая кодировка. Ее коды принимают значения от 0000000 до 1111111 или в шестнадцатеричном выражении от 00h до 7Fh. Постепенно мы рассмотрим их все, только начнем не с самого начала. В каком-то смысле понять назначение первых 32 символов сложнее, чем остальных, поэтому мы сначала познакомимся со второй 32-кодовой группой, в которую включены цифры и знаки препинания.

Шестнадцатеричный код	**Символ ASCII**	**Шестнадцатеричный код**	**Символ ASCII**
20	Пробел	30	0
21	!	31	1
22	"	32	2

(*продолжение*)

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
23	#	33	3
24	$	34	4
25	%	35	5
26	&	36	6
27	‘	37	7
28	(	38	8
29	)	39	9
2A	*	3A	:
2B	+	3B	;
2C	,	3C	<
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

Обратите внимание на символ 20h — это пробел для разделения слов и предложений.

В следующую группу из 32 кодов входят прописные буквы и дополнительные знаки препинания, которых на клавиатуре пишущих машинок, как правило, нет.

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
40	@	50	P
41	A	51	Q
42	B	52	R
43	C	53	S
44	D	54	T
45	E	55	U
46	F	56	V
47	G	57	W
48	H	58	X
49	I	59	Y
4A	J	5A	Z

(*продолжение*)

4B	K	5B	[
4C	L	5C	\
4D	M	5D	]
4E	N	5E	^
4F	O	5F	_

Следующая группа из 32 кодов — строчные буквы и еще несколько знаков препинания.

Шестнадцатеричный код	**Символ ASCII**	**Шестнадцатеричный код**	**Символ ASCII**
60	‘	70	p
61	a	71	q
62	b	72	r
63	c	73	s
64	d	74	t
65	e	75	u
66	f	76	v
67	g	77	w
68	h	78	x
69	i	79	y
6A	j	7A	z
6B	k	7B	{
6C	l	7C	|
6D	m	7D	}
6E	n	7E	~
6F	o		

Заметьте: символа, соответствующего коду 7Fh, в таблице нет. Если вы еще не сбились со счета, то знаете, что всего в трех таблицах я описал 95 кодов. Поскольку кодировка ASCII является 7-битовой, в ней доступно всего 128 кодов, значит, осталось описать еще 33. Вы узнаете о них буквально через пару минут.

Текстовая строка:

```
Hello, you!
```

в кодировке ASCII представляется кодами:

48 65 6C 6C 6F 2C 20 79 6F 75 21

Помимо кодов букв, в ней встречаются коды запятой (2Ch), пробела (20h) и восклицательного знака (1h). Взгляните на еще одно короткое предложение:

```
I am 12 years old.
```

и его представление в формате ASCII:

49 20 61 6D 20 31 32 20 79 65 61 72 73 20 6F 6C 64 2E

Заметьте: число 12 в этой строке представлено кодами 31h и 32h, т. е. ASCII-кодами цифр 1 и 2. Если число 12 — часть текста, для его представления нельзя применять шестнадцатеричные коды 01h и 02h, или BCD-код 12h, или его шестнадцатеричное представление 0Ch. Все эти числа в кодировке ASCII означают что-то другое.

Коды прописных букв в ASCII отличаются от кодов соответствующих строчных букв на 20h. Благодаря этому легко написать программу, которая заменяла бы все строчные буквы в текстовой строке прописными. Допустим, некоторую область в памяти занимает текстовая строка, по 1 байту на символ. Ниже в подпрограмме для процессора 8080 считается, что адрес первого символа строки записан в паре регистров HL; в регистре C хранится длина строки в символах:

```
Capitalize:  MOV A,C     ; C = число оставшихся символов
             CPI A,00h   ; Сравнить с 0
             JZ AllDone  ; Если C = 0, закончить

             MOV A,[HL]  ; Извлечь следующий символ
             CPI A,61h   ; Меньше, чем "a"?
             JC SkipIt   ; Если да, игнорировать

             CPI A,7Bh   ; Больше, чем "z"?
             JNC SkipIt  ; Если да, игнорировать

             SBI A,20h   ; Буква строчная, значит,
                         ; вычитаем 20h
             MOV [HL],A  ; Сохранить символ
```

```
SkipIt:     INX HL      ; Перейти к следующему символу
            DCR C       ; Уменьшить счетчик на 1
            JMP Capitalize      ; Вернуться к началу

AllDone:    RET
```

Оператор, в котором из кода строчной буквы вычитается 20h для преобразования ее в прописную, можно заменить на:

```
ANI A,DFh
```

Команда ANI (AND Immediate) выполняет побитовую операцию И между содержимым аккумулятора и числом DFh или в двоичном представлении 11011111. *Побитовой* я называю операцию, выполняемую отдельно для каждой пары битов, составляющих ее операнды. Эта операция И сохраняет неизменными все биты аккумулятора, кроме третьего слева, который устанавливается в 0. Его обнуление эквивалентно вычитанию 20h, т. е. преобразованию строчной буквы в прописную.

Описанные до сих пор 95 кодов относятся к отображаемым символам, т. е. к символам, у которых есть *внешний вид*. В наборе ASCII есть также 33 управляющих символа, которые при печати или на экране не отображаются, а используются для выполнения тех или иных действий. Для полноты я привожу их все, но не переживайте, если какие-то покажутся вам непонятными. Формат ASCII изначально разрабатывался для телетайпных аппаратов, поэтому многие его коды сейчас уже утратили смысл.

Шестнадцатеричный код	**Сокращение**	**Название**	**Назначение**
00	NUL	Null	Нет
01	SOH	Start of Heading	Начало заголовка
02	STX	Start of Text	Начало текста
03	ETX	End of Text	Конец текста
04	EOT	End of Transmission	Конец передачи
05	ENQ	Enquire (Inquire)	Запрос
06	ACK	Acknowledge	Подтверждение

(*продолжение*)

Шестнадцатеричный код	Сокращение	Название	Назначение
07	BEL	Bell	Сигнал
08	BS	Backspace	Возврат на символ назад
09	HT	Horizontal Tabulation	Горизонтальная табуляция
0A	LF	Line Feed	Перевод строки
0B	VT	Vertical Tabulation	Вертикальная табуляция
0C	FF	Form Feed	Извлечение страницы
0D	CR	Carriage Return	Возврат каретки
0E	SO	Shift Out	Выход
0F	SI	Shift In	Вход
10	DLE	Data Link Escape	
11	DC1	Device Control 1	Управление устройством 1
12	DC2	Device Control 2	Управление устройством 2
13	DC3	Device Control 3	Управление устройством 3
14	DC4	Device Control 4	Управление устройством 4
15	NAK	Negative Acknowledge	Нет подтверждения
16	SYN	Synchronous Idle	Синхронизация
17	ETB	End of Transmission Block	Конец блока передачи
18	CAN	Cancel	Отмена
19	EM	End of Medium	Конец носителя

(*продолжение*)

1A	SUB	Substitute Character	Замена
1B	ESC	Escape	
1C	FS	File Separator or Information Separator 4	Разделитель файлов или информации 4
1D	GS	Group Separator or Information Separator 3	Разделитель групп или информации 3
1E	RS	Record Separator or Information Separator 2	Разделитель записей или информации 2
1F	US	Unit Separator or Information Separator 1	Разделитель элементов или информации 1
7F	DEL	Delete	Удаление

Смысл некоторых из этих символов в том, что их можно размещать среди отображаемых символов, внося в текст некое примитивное форматирование. Представьте себе устройство для печати (телетайп или принтер), отображающее на бумаге символы, руководствуясь поступающими на него ASCII-кодами. Обычно устройство работает так: печатающая головка печатает символ, соответствующий коду, и сдвигается на одну позицию вправо. Важнейшие управляющие символы нужны, чтобы изменить эту последовательность.

Рассмотрим в качестве примера шестнадцатеричную строку

41 09 42 09 43 09

Код 09h соответствует символу горизонтальной табуляции, или просто табулятору. Символ табуляции сдвигает печатающую головку к следующей позиции, номер которой кратен 8. Результат выглядит примерно так:

A B C

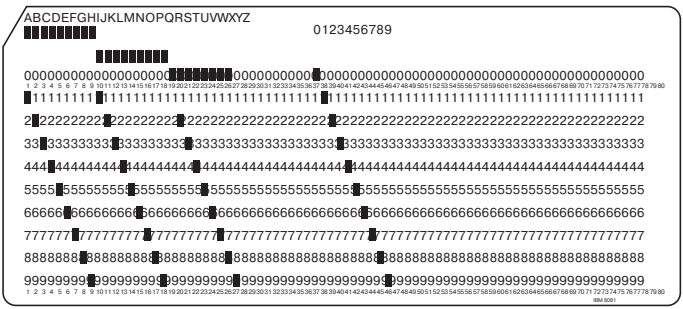
Табуляцию удобно применять, когда при печати текст должен выравниваться по столбцам.

Даже многие современные принтеры реагируют на код 12h, извлекая страницу и начиная печать на новой.

Код 08h (BS) позволяет печатать составные символы на старых принтерах. Так, для печати буквы *е* с диакритическим знаком — обратным апострофом (é) на принтер нужно передать последовательность кодов 65h 08h 60h.

Важнейшими управляющими символами и в наши дни являются символы возврата каретки и перевода строки, действие которых аналогично соответствующим кодам Бодо. Код CR смещает печатающую головку принтера к левому краю страницы, а LF приводит к прокрутке валика на одну строку вверх. Для начала новой строки обычно надо указывать оба кода. Но допускается их применение и по отдельности, скажем, для печати новой строки поверх старой или для начала новой строки не у левого поля.

В мире персональных компьютеров кодировка ASCII — безусловный лидер, но на крупных компьютерных системах фирмы IBM она не используется. Для вычислительной машины System/360 фирма IBM разработала собственный 8-битовый код EBCDIC (Extended BCD Interchange Code, расширенный код обмена BCD). Это расширенный вариант 6-битового кода BCDIC, применявшегося в перфокартах IBM в течение 50 лет.



Разбираясь во взаимосвязях между перфокартами и соответствующими кодами EBCDIC, помните, что на развитие этой кодировки влияло несколько различных технологий. Поэтому не стоит ожидать от нее чрезмерной логичности или внутренней согласованности.

Для кодирования символа в столбце цифр на перфокарте пробивается несколько прямоугольных отверстий. Для ясности поверх столбца часто печатали закодированный символ. Нижние 10 строк пронумерованы от 0 до 9. Ненумерованная строка над 0-й строкой считается 11-й, а самая верхняя строка — 12-й. Десятой строки на карте нет. Всего на одной карте 80 столбцов, а значит закодировать с ее помощью можно 80 символов.

Строки с 0-й по 9-ю иногда называют цифровыми строками или цифровой пробивкой (digital punches), строки 11 и 12 — зонными строками или зонной пробивкой (zone punches). Чтобы жизнь не казалась медом, иногда зонной пробивкой считаются и отверстия в строках 0 и 9.

8-битовый код EBCDIC состоит из старшей и младшей тетрад. Младшая является кодом BCD, соответствующим цифровой пробивке символа. Старшая содержит код, соответствующий (довольно произвольным образом) зонной пробивке символа. Как вы помните из главы 19, в кодировке BCD 4-битовые коды используются для представления цифр от 0 до 9.

У кодов цифр от 0 до 9 зонной пробивки нет, т. е. старшая тетрада равна 1111, младшая — BCD-коду цифры.

Шестнадцатеричный код	**Символ EBCDIC**
F0	0
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9

При кодировании прописных букв тетрада 1100 соответствует зонной пробивке только в строке 12, тетрада 1101 — зонной пробивке только в строке 11, тетрада 1110 — зонной пробивке в строке 0. Вот как выглядят коды EBCDIC для прописных букв латинского алфавита:

Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC
C1	A	D1	J		
C2	B	D2	K	E2	S
C3	C	D3	L	E3	T
C4	D	D4	M	E4	U
C5	E	D5	N	E5	V
C6	F	D6	O	E6	W
C7	G	D7	P	E7	X
C8	H	D8	Q	E8	Y
C9	I	D9	R	E9	Z

Заметьте: коды не образуют сплошной последовательности. При программной обработке текста EBCDIC эти пробелы в нумерации крайне неудобны.

Коды строчных букв обозначаются той же цифровой пробивкой, что и коды прописных букв, но иной зонной пробивкой. В кодах строчных букв от *a* до *i* пробиты строки 12 и 0, чему соответствует код 1000. В кодах строчных букв от *j* до *r* пробиты строки 12 и 11 (код 1001). Наконец, в кодах строчных букв от *s* до *z* пробиты строки 11 и 0 (код 1010).

Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC	Шестнадцатеричный код	Символ EBCDIC
81	a	91	j		
82	b	92	k	A2	s
83	c	93	l	A3	t
84	d	94	m	A4	u
85	e	95	n	A5	v
86	f	96	o	A6	w
87	g	97	p	A7	x
88	h	98	q	A8	y
89	i	99	r	A9	z

Конечно, в EBCDIC есть и другие коды — для знаков препинания и для управляющих символов, но нам вряд ли имеет смысл тратить время на столь глубокое знакомство с этой системой.

Теоретически каждого столбца на перфокарте достаточно для кодирования 12 битов информации (1 отверстие — 1 бит). С другой стороны, для записи 7-битового кода ASCII можно в каждом столбце использовать 7 из 12 строк. Но на практике приходится учитывать, что карта, в которой пробито слишком много отверстий, довольно скоро ветшает.

Многие из 8-битовых кодов EBCDIC остались неопределенными. Это значит, что смысла в 7-битовой кодировке ASCII больше. Во времена разработки ASCII память для компьютеров стоила очень дорого, поэтому некоторые даже настаивали, что для экономии памяти кодировка должна быть 6-битовой, а для набора строчных и прописных букв нужен код-переключатель. Однако эта идея скоро была отвергнута, уступив место представлению о том, что кодировка ASCII должна быть 8-битовой. Это связано с тем, что в основе компьютерной архитектуры лежит не 7-, а 8-битовая единица — байт. И конечно, хотя технически основная кодировка ASCII остается 7-битовой, для хранения отдельных символов в компьютерах почти всегда отводится 8 битов.

Эквивалентность байтов и символов довольно удобна: мы всегда может приблизительно сказать, сколько места в памяти займет тот или иной документ, попросту подсчитав количество символов в нем. Некоторым людям легче представить себе объем памяти компьютера, если они могут сказать, сколько текста можно в ней разместить.

Например, обычная машинописная страница займет в памяти около 1 700 байт. На странице журнала «The New Yorker» три колонки текста по 60 строк в каждой. При средней длине строки в 40 знаков получаем объем страницы 7 200 символов (или байт). На странице «The New York Times» текст разделен на 6 колонок. Если бы в нем не было ни заголовков, ни фотографий (что в высшей степени необычно), всего в каждой колонке умещалось бы 155 строк длиной по 35 символов. Значит, объем страницы равен 32 550 символов, или 32 килобайта.

На книжной странице обычного формата помещается более 400 слов. При средней длине слова в 7 символов (точнее, в 8, так как за каждым словом идет пробел) объем книжной стра-

ницы равен примерно 3 000 символов. Примем, что объем типичной книги — 333 страницы. У этого, прямо скажем, некруглого числа, есть одно преимущество — оно позволяет нам уверенно заявить, что типичная книга занимает в памяти объем около 1 000 000 байт, или 1 Мб.

Конечно, вокруг этого среднего есть значительный разброс.

«Великий Гетсби» Френсиса Скотта Фицджеральда — около 300 кб.

«Над пропастью во ржи» Джерома Сэлинджера — около 400 кб.

«Приключения Гекльберри Финна» Марка Твена — около 540 кб.

«Гроздья гнева» Джона Стейнбека — около 1 Мб.

«Моби Дик» Германа Мелвилла — 1,3 Мб.

«История Тома Джонса, найденыша» Генри Филдинга — 2,25 Мб.

«Унесенные ветром» Маргарет Митчел — 2,5 Мб.

«Война и мир» Льва Толстого — 3,9 Мб.

«В поисках утраченного времени» Марселя Пруста — 7,7 Мб.

В библиотеке Конгресса Соединенных Штатов Америки 20 млн. книг, 20 триллионов символов, 20 терабайт текстовых данных (кроме них, еще масса фотографий и аудиозаписей).

Несмотря на важность, стандарт ASCII далеко не идеален. Беда американского кода для обмена информацией в том, что он слишком американский! Он едва ли удовлетворит даже требования народов, говорящих по-английски. Значок доллара в ASCII есть, но где же значок английского фунта? Где в нем буквы с диакритическими знаками, применяемые в большинстве западноевропейских языков? Я уж молчу о нелатинских алфавитах — греческом, арабском, иврите, кириллице. А слоговое письмо брахми в Индии и Юго-Восточной Азии, к которому восходят языки деванагари, бенгали, тайский и тибетский? Наконец, как прикажете представлять с помощью 7-битового кода *десятки тысяч* иероглифов китайского, японского и корейского языков?

При разработке ASCII потребности других алфавитов учитывались мало, хотя, конечно, о нелатинских алфавитах речь не шла. В стандарте ASCII считается, что 10 его кодов (40h, 5Bh, 5Ch, 5Dh, 5Eh, 60h, 7Bh, 7Ch, 7Dh и 7Eh) другие страны могут переопределять согласно своим потребностям. Кроме того,

предусматривалась возможность замены символа # значком английского фунта £, а знака доллара $ — обобщенным символом денежной единицы ¤. Очевидно, замена имеет смысл, лишь когда о ней знают все работающие с документом, который содержит переопределенные коды.

Поскольку в большинстве компьютерных систем символы все равно хранятся как 8-битовые значения, возникает возможность расширения набора символов ASCII до 256 символов. В расширенной кодировке значения кодов с 00h до 7Fh остались неизменными, а коды с 80h по FFh соответствуют буквам с диакритическими знаками или буквам нелатинских алфавитов. Вот, например, как шифруются в расширенной кодировке ASCII кириллические буквы. В таблице старшая тетрада шестнадцатеричного кода символа указана в первой строке, а младшая — в левом столбце.

	8-	**9-**	**A-**	**E-**
-0	А	Р	а	р
-1	Б	С	б	с
-2	В	Т	в	т
-3	Г	У	г	у
-4	Д	Ф	д	ф
-5	Е	Х	е	х
-6	Ж	Ц	ж	ц
-7	З	Ч	з	ч
-8	И	Ш	и	ш
-9	Й	Щ	й	щ
-A	К	Ъ	к	ъ
-B	Л	Ы	л	ы
-C	М	Ь	м	ь
-D	Н	Э	н	э
-E	О	Ю	о	ю
-F	П	Я	п	я

К сожалению, на протяжении последних десятилетий появилось множество *различных* вариантов расширения таблицы ASCII даже для одного языка, что, конечно же, приводит к многочисленным сложностям. Наиболее радикальным изменени-

ям кодировка ASCII подверглась ради включения в нее китайских, японских и корейских иероглифов. В популярной кодировке Shift-JIS (Japan Industrial Standard, японский промышленный стандарт) коды с 81h по 9Fh в действительности представляют собой первый байт 2-байтового кода символа. Благодаря этому в кодировке Shift-JIS удается дополнительно закодировать около 6 000 дополнительных символов. К сожалению, эта кодировка — не единственная. Кроме нее в Азии применяются еще три варианта двухбайтовых наборов символов.

Наличие нескольких несовместимых двухбайтовых наборов — не единственная проблема. Есть еще одна: некоторые символы, а именно обычные символы ASCII, представляются одним байтом, тогда как другие — тысячи иероглифов — двумя. Конечно, работать с такими наборами очень трудно.

Осознав необходимость единой и всеобщей системы кодирования символов, которая подходила бы для всех языков мира, в 1988 г. несколько крупных компьютерных компаний начали разработку кодировки *Unicode*, которая должна прийти на смену ASCII. В отличие от ASCII кодировка Unicode является не 7-, а 16-битовой. По 2 байта занимают все символы Unicode до единого. Это значит, что в Unicode коды принимают значения от 0000h до FFFFh, а всего их доступно 65 536. Этого достаточно для любых языков мира, по крайней мере для тех, что будут использоваться в компьютерах, да еще и остается место для расширения.

Создание Unicode начиналось не на пустом месте. Первые 128 символов — с кодами от 0000h до 007Fh — совпадают с символами ASCII. Далее нашлось место и для греческих, и для кириллических, и для арабских букв, и для множества других символов.

Преимущества Unicode несомненны, но это, увы, не облегчает ее внедрения. ASCII и ее многочисленные расширения настолько распространены в компьютерном мире, что сместить их с пьедестала будет очень нелегко.

Правда, в Unicode удобное равенство «1 символ = 1 байт» уже не соблюдается. В кодировке ASCII роман «Гроздья гнева» занимает 1 мегабайт, в кодировке Unicode — уже 2 Мб. Но право же, это небольшая плата за универсальную всеобщую систему кодирования символов.

Глава 21

Под шорох шин

Процессор, конечно, — основной компонент компьютера, но далеко не единственный. Компьютеру нужна память для хранения кодов команд, которые процессор будет исполнять. Компьютеру нужно устройство ввода, чтобы эти коды попадали в память, а еще устройство вывода, позволяющее просматривать результаты работы программы. Вы, конечно, помните, что память RAM энергозависима — при отключении питания ее содержимое стирается. Поэтому компьютер нужно снабдить долговременным запоминающим устройством, в котором можно хранить данные и программы, когда компьютер выключен.

Интегральные схемы, из которых состоит компьютер, монтируются на платах. Если компьютер невелик, все схемы можно разместить на одной плате. Но чаще его компоненты распределяют на две или несколько плат. Эти платы обмениваются информацией через *шину* (bus), т. е. набор цифровых сигналов, подаваемых на все платы компьютера. Эти сигналы разделяются на 4 категории.

- *Адресные* — генерируются процессором и обычно используются для адресации оперативной памяти. Могут также применяться для обращения к другим устройствам компьютера.
- *Вывода данных* — также генерируются микропроцессором. Используются для передачи данных в память и на другие

устройства. Не запутайтесь в понятиях «ввод» и «вывод»: сигнал вывода микропроцессора становится сигналом ввода для памяти или внешнего устройства.

- *Ввода данных* — генерируются различными устройствами компьютера и поступают в микропроцессор. Чаще всего в роли источника данных для процессора выступает память.
- *Управляющие* — генерируются как микропроцессором, так и другими устройствами, которым нужно что-то сообщить процессору. Пример сигнала такого рода — сигнал, которым процессор указывает на необходимость записи данных в определенную ячейку памяти.

Кроме того, шина отвечает за подачу питания на различные компоненты компьютера.

Одной из первых широко распространенных шин для домашних компьютеров была S-100, вышедшая в 1975 г. в составе первого же домашнего компьютера «Альтаир». Первоначально она была ориентирована на микропроцессор 8080, но позже ее приспособили и под другие процессоры, например, 6800. Плата S-100 представляет собой прямоугольную пластину размером 5,3 × 10 дюймов (13,5 × 25,4 см), одна из сторон которой представляет собой разъем со 100 контактами (отсюда и обозначение — S-100).

Основой компьютера с шиной S-100 является *материнская плата* (motherboard). На ней размещено несколько (до 12) гнезд для вставки разъемов плат S-100. Иногда эти гнезда называют *слотами расширения* (expansion slots), а вставляемые в них платы — *платами расширения* (expansion cards). Одну плату S-100 занимает сам процессор с вспомогательными микросхемами (некоторые из них я упоминал в главе 19), еще один или несколько разъемов отданы под оперативную память.

Поскольку шина S-100 разрабатывалась под процессор 8080, в ее состав входят 16 адресных линий, 8 линий для ввода и 8 — для вывода данных. Как вы помните, в процессоре 8080 линии ввода и вывода данных объединены. Перед попаданием в шину они разделяются в специальной микросхеме, установленной на той же плате, что и процессор. Кроме того, в шине есть 8 линий прерываний, генерируемых устройствами компьютера, которым нужно привлечь внимание процессора. Так, клавиатура генерирует сигнал прерывания при нажатии клавиши (об этом

ниже). Процессор в ответ запускает короткую программу, которая определяет, какая клавиша нажата, и выполняет ответное действие. Для работы с прерываниями на плату с процессором 8080 обычно устанавливается микросхема Intel 8214 — устройство управления приоритетными прерываниями. Когда одно из устройств запрашивает прерывание, микросхема направляет в процессор соответствующий сигнал. В процессоре выполняется команда RST (перезапуск), в результате которой процессор сохраняет текущее содержимое программного счетчика и переходит к команде по адресу 0000h, 0008h, 0010h, 0018h, 0020h, 0028h, 0030h и 0038h в зависимости от номера прерывания.

Разработав шину нового типа, вы должны решить один важный вопрос: опубликовать параметры шины или сохранить их в тайне.

Если технические характеристики шины стали достоянием общественности, производить платы расширения для нее могут не только изготовители шины, но и другие компании. Доступность множества разнообразных плат расширения делает компьютер гибче, а значит, и привлекательнее. С ростом числа продаваемых компьютеров расширяется и рынок для плат расширения. Эта закономерность побуждает разработчиков большинства небольших компьютерных систем придерживаться принципов *открытой архитектуры* (open architecture), т. е. разрешать сторонним производителям создавать оборудование для своих компьютеров. Шина, отданная во всеобщее пользование, со временем может стать промышленным *стандартом*. Такие стандарты сыграли в развитии персональных компьютеров важную роль.

Самый знаменитый персональный компьютер (ПК) с открытой архитектурой — первый IBM PC — появился на рынке осенью 1981 г. IBM опубликовала технический справочник с полными электрическими схемами самого компьютера и всех плат расширения для него, производимых IBM. Наличие этого справочника позволило многим компаниям производить не только собственные платы расширения, но и целые *клоны* ПК, которые практически не отличались от компьютеров IBM и работали с тем же программным обеспечением (ПО).

В наши дни на долю разнообразных наследников IBM PC приходится 90% рынка ПК. Сама IBM в этих процентах занимает довольно скромное место, но ее доля вполне могла ока-

заться еще скромнее, если бы архитектура первого IBM PC осталась *закрытой*. Доказательство тому — «Apple Macintosh». Архитектура этого компьютера никогда не становилась достоянием общественности, и это, вероятно, объясняет, почему на долю «Macintosh» сейчас приходится менее 10% продаж ПК. Не забывайте, что закрытость архитектуры компьютера не мешает другим компаниям создавать для него *ПО*. Лишь некоторые производители видеоигр запрещают сторонним компаниям писать программы для своих систем.

В первом IBM PC использовался процессор Intel 8088 с адресным пространством в 1Мб. Сам по себе это 16-разрядный процессор, но обмен данным с памятью происходит 8-битовыми фрагментами, т. е. байтами. Шина с 62 контактами, разработанная в IBM для этого компьютера, теперь называется шиной ISA (Industry Standard Architecture, архитектура промышленного стандарта). В ней 20 адресных линий, 8 линий для ввода и вывода данных, 6 — для запросов на прерывания и 3 — для запросов на *прямой доступ к памяти* (Direct Memory Access, DMA). Линии DMA позволяют внешним устройствам перехватывать управление шиной и производить обмен данными с памятью в обход микропроцессора, хотя обычно чтение и запись осуществляет только он.

В компьютере S-100 на платах расширения размещены все компоненты без исключения. В IBM PC микропроцессор, некоторые вспомогательные микросхемы и часть оперативной памяти помещались на *системной плате* (в терминологии IBM), которую часто также называют материнской.

В 1984 г. IBM выпустила компьютер PC AT с 16-разрядным процессором Intel 80286, адресное пространство которого составляло 16 Мб. Шина ISA в новом компьютере осталась, но к ней было добавлено еще 36 выводов, в том числе 7 дополнительных адресных линий (хотя нужно их было всего 4), 8 дополнительных линий для ввода и вывода данных, 5 линий запросов на прерывания и 4 канала DMA.

Шины приходится модернизировать или заменять, когда микропроцессоры перерастают их либо по разрядности данных, либо по объему адресуемой памяти, либо по быстродействию. Первые шины предназначались для микропроцессоров с тактовыми частотами порядка нескольких, но никак не со-

тен мегагерц. Шина, работающая на частоте, которая превышает допустимое значение, становится источником электромагнитных помех, способных сказаться на работе близких радиоприемников и телевизоров.

В 1987 г. в IBM была создана шина MCA (Micro Channel Architecture, микроканальная архитектура). Она была частично запатентована, что давало IBM возможность получать деньги с других компаний, применявших эту шину. Вероятно, поэтому шина MCA *не стала* промышленным стандартом. Более того, в 1988 г. консорциум из 9 компаний (IBM в их число не входила) создал альтернативную 32-разрядную шину EISA (Extended Industry Standard Architecture, усовершенствованная архитектура промышленного стандарта). В последние годы в IBM-совместимых компьютерах повсеместно используется шина PCI (Peripheral Component Interconnect, соединение периферийных компонентов).

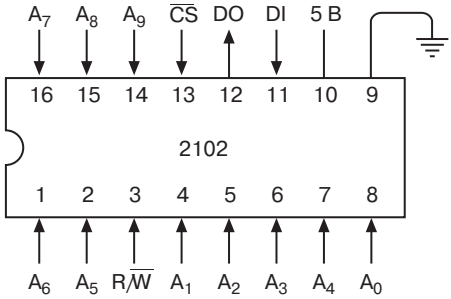
Понять, как работают компоненты компьютера, нам опять поможет обращение к давней и незамысловатой эпохе — середине 1970-х. Представим, что мы занялись разработкой плат расширения для «Альтаира» или для компьютера собственной конструкции с процессором 8080 или 6800. Нам понадобится оперативная память, клавиатура, монитор и, вероятно, некое устройство для хранения информации, когда компьютер выключен.

Как вы помните из главы 16, массиву RAM нужны адресные входы, входы для ввода и вывода данных и сигнал, который управляет записью данных в память. Количеством адресных входов определяется объем информации, которую можно записать в массив RAM:

$$\text{Объем массива RAM} = 2^{\text{Количество адресных входов}}$$

Количеством входов для чтения и записи данных задана на разрядность сохраняемых значений.

В середине 1970-х в оперативной памяти домашних компьютеров часто использовалась микросхема 2102:

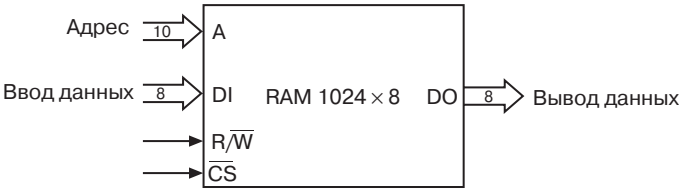


Микросхема 2102 относится к семейству МОП (металл-окисел-полупроводник), или MOS (metal-oxide semiconductor), т. е. к тому же семейству, что и сами микропроцессоры 8080 и 6800. Микросхемы МОП легко соединяются со схемами из семейства ТТЛ; плотность транзисторов в них обычно выше, чем в ТТЛ-схемах, но работают они медленнее.

Подсчитав адресные входы (с A_0 по A_9) и заметив, что входов для ввода (DI) и вывода (DO) данных всего по одному, вы сообразите, что в эту микросхему можно записать 1 024 бита. У различных модификаций микросхемы 2102 время доступа, т. е. время между поступлением адреса на адресные входы и подачей нужного бита на выход DO, составляет от 350 до 1 000 нс. При чтении данных из памяти сигнал $R/\overline{W}$ равен 1. Чтобы записать входной сигнал в память, сигнал $R/\overline{W}$ должен обратиться в 0 на время не меньше 170–550 нс, опять же в зависимости от вида микросхемы 2102.

Особенно интересен сигнал $\overline{CS}$ (Chip Select, выбор чипа). Когда он равен 1, микросхема не выбрана, т. е. не реагирует на сигнал $R/\overline{W}$. Этот сигнал играет в работе микросхемы еще одну немаловажную роль, но о ней чуть позже.

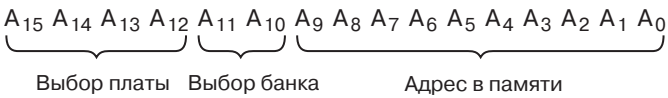
Конечно, если вы собираете память для 8-разрядного процессора, вам хочется организовать ее так, чтобы в нее можно было записывать 8-битовые, а не 1-битовые значения. Для сохранения целых байтов вам понадобится не менее 8 таких микросхем с соединенными адресными входами, сигналами $R/\overline{W}$ и $\overline{CS}$. Результат схематически можно изобразить так:



Это массив RAM 1024 × 8. Его объем равен 1 кб.

С практической точки зрения микросхемы памяти нужно разместить на плате. Если постараться, то на одной плате S-100 их уместится 64, т. е. 8 кб. Но мы удовлетворимся скромными 4 кб, т. е. 32 микросхемами. Набор микросхем, соединенных друг с другом для хранения целого байта, называется *банком* (bank). На плате с памятью объемом 4 кб размещаются 4 банка по 8 микросхем в каждом.

Адресное пространство процессоров 8080 и 6800 равно 64 кб, так как у них по 16 адресных входов. 16 адресных сигналов платы объемом 4 кб, содержащей 4 банка микросхем, выполняют такие функции:



Адресные сигналы с A_0 по A_9 подключены к микросхемам. Сигналы A_{10} и A_{11} задают банк, к которому происходит обращение. Наконец, адресные сигналы с A_{12} по A_{15} определяют диапазон адресов, относящихся к данной плате. Наша плата емкостью 4 кб во всем адресном пространстве процессора (64 кб) может занимать один из шестнадцати 4-килобайтовых интервалов:

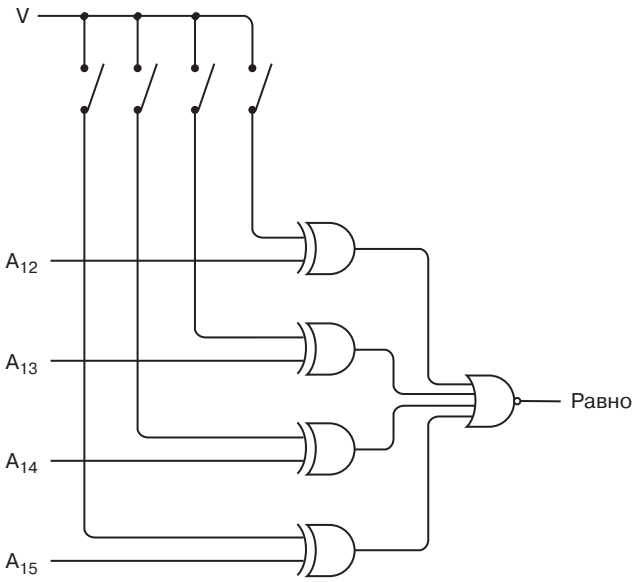
От 0000h до 0FFFh
От 1000h до 1FFFh
От 2000h до 2FFFh
…
От F000h до FFFFh

Допустим, мы решили отвести нашей плате диапазон адресов от A000h до AFFFh. Это значит, что первый банк занимает ад-

реса с A000h по A3FFh, второй — с A400h по A7FFh, третий — с A800h по ABFFh, четвертый — с AC00h по AFFFh.

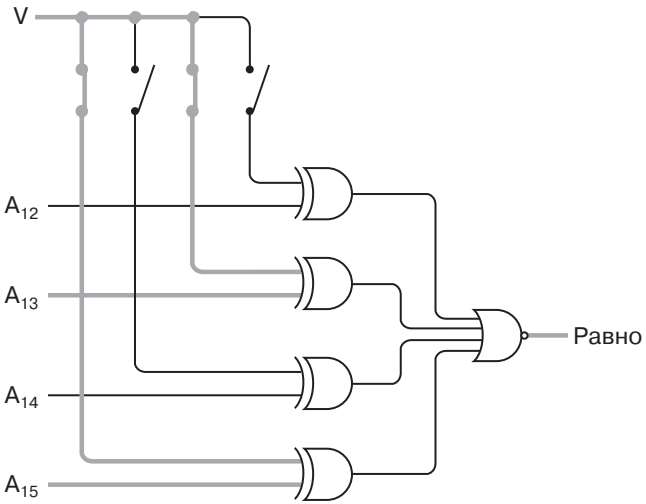
В платах памяти емкостью 4 кб часто предусматривают возможность оперативного изменения соответствующего диапазона адресов. Делается это с помощью *DIP-переключателя* — набора из нескольких (от 2 до 12) крохотных переключателей, собранных в одном корпусе с двухрядным расположением выводов (Dual Inline Package, DIP).

Его можно подключить к 4 старшим адресным линиям шины с помощью специальной схемы — *компаратора* (comparator):

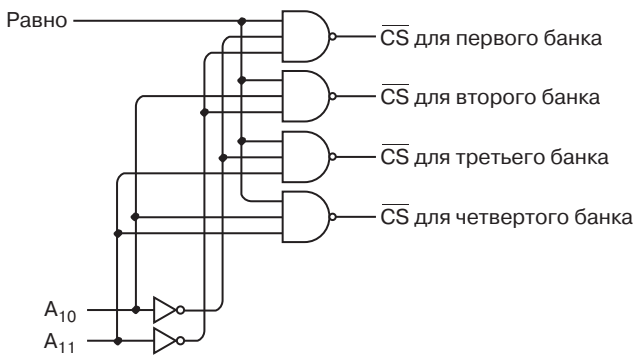


Как вы помните, выход вентиля «Исключающее ИЛИ» равен 1, только если сигналы на двух его входах не совпадают.

Чтобы плата соответствовала адресам от A000h до AFFFh, нужно замкнуть переключатели, соответствующие линиям A_{13} и A_{15}. Если значения адресных сигналов шины A_{12}, A_{13}, A_{14} и A_{15} совпадают со значениями, заданными с помощью переключателей, выходы всех четырех вентилей «Исключающее ИЛИ» равны 0. Это значит, что выход вентиля ИЛИ-НЕ равен 1:



Объединив сигнал Равно с дешифратором 2 линии на 4, вы сможете генерировать сигналы $\overline{CS}$ для всех четырех банков памяти:



Например, если A_{10} равен 0, а A_{11} — 1, выбран третий банк.

Если вы еще не забыли нашу возню со сборкой памяти в главе 16, то можете решить, что нам также понадобится 8 селекторов 4 линии на 1, чтобы выбирать правильные выходные сигналы от четырех банков памяти. Но на самом деле это не так, и вот почему.

Обычно выходной сигнал микросхемы, совместимой с семейством ТТЛ, либо выше 2,2 В (логическая 1), либо ниже 0,4 В (логический 0). А что будет, если соединить выходные сигналы, один из которых равен 1, а второй — 0? Сказать определенно нельзя, и потому выходы интегральных микросхем обычно между собой не соединяют.

Выходной сигнал микросхемы 2102 называется сигналом *с тремя состояниями* (3-state или tristate). Его третье состояние отличается как от логического 0, так и от логической 1 и соответствует, как ни странно, полному отсутствию сигнала, как будто к данному выводу микросхемы вообще ничего не подключено. Выходной сигнал микросхемы 2102 переходит в третье состояние, если сигнал $\overline{CS}$ обращается в 1. Это значит, что выходы для вывода данных всех четырех банков можно соединить и использовать эти 8 объединенных выводов как 8 линий ввода данных в шине.

Я решил рассказать о сигналах с тремя состояниями, поскольку для работы шины они очень важны. Практически все устройства, подключенные к шине, используют линии ввода данных. В данный момент времени передавать данные по шине может только одно из них. Выходы всех остальных устройств должны находиться в третьем состоянии.

Микросхема 2102 является *статической* (static) памятью RAM, или SRAM. Другой тип памяти называется *динамическим* (dynamic) и обозначается сокращением DRAM. В памяти SRAM для хранения 1 бита обычно нужны 4 транзистора (это меньше, чем их требовалось в триггерах, из которых мы собирали память в главе 16). В DRAM на 1 бит приходится всего 1 транзистор, зато память этого типа требует более сложной вспомогательной электроники.

SRAM (например, микросхема 2102) сохраняет данные только при наличии питания. Если питание отключается, содержимое памяти пропадает. DRAM обладает таким же свойством. Кроме того, содержимое памяти DRAM нужно перио-

дически считывать, даже если оно не нужно. Такая *регенерация* (refresh) должна производиться несколько сотен раз в секунду.

Хотя применение DRAM связано с дополнительными хлопотами, благодаря непрерывно возрастающей емкости микросхемы этого типа стали фактически стандартом. В 1975 г. Intel выпустила микросхему DRAM емкостью 16 384 бита. Согласно закону Мура емкость таких чипов каждые 3 года возрастает вчетверо. В современных компьютерах гнезда для памяти размещаются прямо на материнской плате. В эти гнезда вставляются небольшие платы — модули памяти SIMM (Single Inline Memory Module) или DIMM (Dual Inline Memory Module) с несколькими микросхемами DRAM.

Теперь, когда вы узнали, как делать платы памяти, не увлекайтесь — не будем заполнять памятью все адресное пространство процессора, оставив часть его для устройства вывода.

Самым распространенным устройством вывода в современных компьютерах стала *катодно-лучевая трубка* (Cathode-Ray Tube, CRT) — в облике телевизора непременный атрибут человеческого жилища второй половины XX в. Кинескоп, подключенный к компьютеру, называют обычно дисплеем или монитором. Аппаратный компонент, управляющий работой монитора, называется *видеоадаптером* (video display adapter) или *видеоплатой*, поскольку в компьютере он чаще всего занимает отдельную плату.

Хотя изображение на экране монитора или телевизора кажется двумерным, в действительности оно состоит из одной линии, которую очень быстро прочерчивает по экрану электронный луч. Луч начинает свое путешествие в верхнем левом углу. Пройдя вправо до конца экрана, он возвращается назад и начинает рисовать следующую *строку развертки* (scan line). Возвратное движение к началу следующей строки называется *обратным ходом по горизонтали* (horizontal retrace). Закончив последнюю строку, луч возвращается из нижнего правого в верхний левый угол экрана (совершая обратный ход по вертикали), и процесс начинается снова. По стандартам телевизионных сигналов в США это происходит 60 раз в секунду. Именно такая *частота развертки* (field rate) обеспечивает нормальное восприятие изображения.

В телевизорах на самом деле все происходит немного сложнее из-за *чересстрочной развертки* (interlacing): один кадр изображения рисуется в два приема — сначала четные строки, потом нечетные. *Частота строчной развертки* (horizontal scan rate), т. е. скорость прорисовки одной строки, равна 15 750 Гц, Разделив этой число на 60 Гц, получим, что одно поле кадра состоит из 262,5 строк. Целый кадр состоит из 2 полей, т. е. из 525 строк.

Как при чересстрочной, так и при обычной развертке электронный луч, формирующий изображение, управляется единым непрерывным сигналом. В эфире изображение и звук странствуют совместно, но в телевизоре им суждено разделиться. Далее я буду говорить о видеосигнале — том, что подается на видеовходы и видеовыходы видеомагнитофонов, камер и некоторых телевизоров.

Видеосигнал черно-белого телевизора прост и понятен (наличие цвета, конечно, все несколько запутывает). 60 раз в секунду в этот сигнал вставляется *вертикальный синхроимпульс* (vertical sync pulse), указывающий на начало очередного поля. Этот импульс состоит в установке на 400 микросекунд нулевого напряжения (земли). Начало строки обозначается горизонтальным синхроимпульсом — нулевым напряжением, устанавливаемым на 5 микросекунд 15 750 раз в секунду. В промежутках между горизонтальными синхроимпульсами сигнал варьируется от 0,5 В (черный) до 2 В (белый). Промежуточные напряжения соответствуют оттенкам серого.

Таким образом, телевизионное изображение является частично цифровым и частично аналоговым. По вертикали оно разделено на 525 отдельных строк, но внутри каждой напряжение непрерывно меняется, хотя и не с произвольной скоростью. Существует предельная скорость реакции телеэкрана на изменившееся напряжение, определяемая *полосой пропускания* (bandwidth).

Понятие полосы пропускания чрезвычайно важно. Именно она определяет объем информации, который можно передать по данному каналу. В случае телевизора полоса пропускания есть предельная скорость, с которой черный цвет может сменяться белым, а потом вновь становиться черным. В американском телевещании она равна 4,2 Мгц.

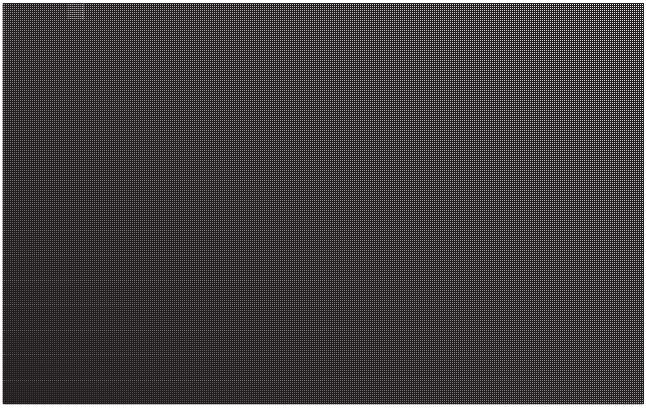
Мы хотим подключить дисплей к компьютеру, поэтому нам неудобно представлять его в виде гибрида цифрового и

аналогового устройства. Легче считать дисплей исключительно цифровым. С компьютерной точки зрения, удобнее всего представлять изображение на мониторе как прямоугольную сетку, отдельные элементы которой называются *пикселами* (pixel). Термин этот — сокращение словосочетания «picture element» (элемент изображения).

Полоса пропускания видеосистемы накладывает ограничение на число пикселов в одной строке развертки. Я определил полосу пропускания как предельную скорость, с которой черный цвет может сменяться белым, а потом вновь становиться черным. Полоса в 4,2 МГц означает, что пары пикселов могут создаваться 4,2 млн. раз в секунду. Разделив удвоенную (за счет двух пикселов) полосу пропускания на частоту строчной развертки, получим, что одна строка состоит из 533 пикселов. Реально же их всего около 320, так как часть времени тратится на обратный ход, кроме того, строка видна на экране не целиком.

Да и по вертикали 525 пикселов тоже не набирается. Они теряются в верхних и нижних невидимых участках экрана, а также за счет обратного хода по вертикали. А если учесть, что при работе с компьютером чересстрочной разверткой лучше себе жизнь не осложнять, разумно предположить, что число пикселов по вертикали примерно равно 200.

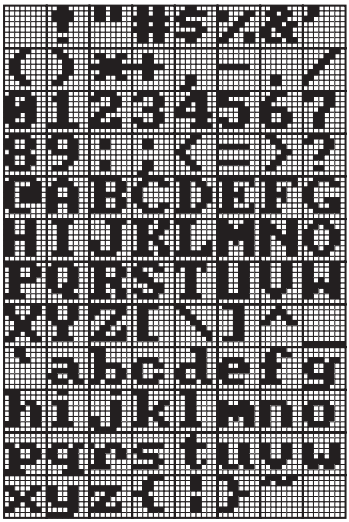
Итак, *разрешение* (resolution) простейшего видеоадаптера, подсоединенного к обычному телевизору, — 320 пикселов по горизонтали и 200 по вертикали, или сокращенно 320 × 200:



Полное число пикселов на экране 320 × 200 = 64 000. В зависимости от того, как вы сконфигурировали свой адаптер, пиксел может быть черно-белым или обладать определенным цветом.

Допустим, монитор нужен нам для отображения текста. Сколько его там уместится?

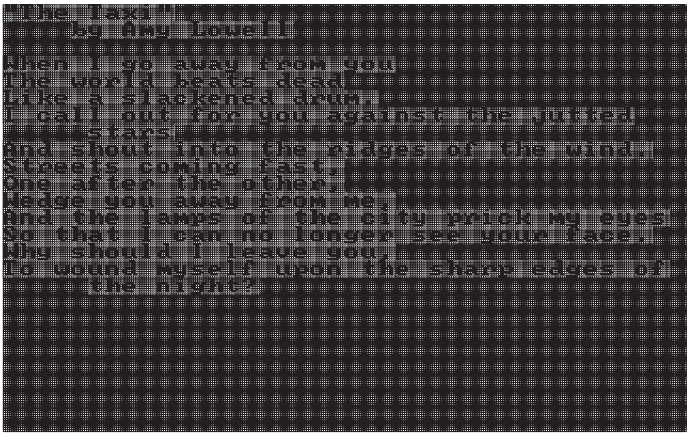
Это, конечно, зависит от того, сколько пикселов занимают отдельные буквы. Можно, не мудрствуя лукаво, выделить под все символы одинаковые квадратики размером 8 на 8 пикселов:



Этим символами соответствуют ASCII-коды от 20h до 7Fh (напомню, что символы с кодами от 00h до 1Fh являются неотображаемыми).

Теперь каждому символу соответствует не только 7-битовый код ASCII, но и 64 бита на экране, которые определяют его внешний вид. Эти 64 бита тоже можно считать своеобразным кодом.

На экране дисплея с разрешением 320 × 200 можно разместить 25 строк по 40 символов размером 8 × 8. Этого хватит, например, для небольшого стихотворения.



Для хранения изображения видеоадаптеру требуется некоторое количество памяти RAM, а микропроцессору — возможность записи в эту память, чтобы выводить информацию на экран. Удобнее всего выделять место для этой памяти в общем адресном пространстве процессора. Сколько же памяти выделить простому адаптеру, о котором я рассказываю?

Ответ не так уж очевиден! От 1 до 192 кб!

Начнем с минимальных требований. Самое малое, что нужно от нашего адаптера, — это отображение текста. Мы уже подсчитали, что на экране умещается 25 строк по 40 символов в каждой, т. е. всего 1 000 символов. В памяти RAM на видеоплате достаточно хранить 7-битовые ASCII-коды этих символов. Тысяча 7-битовых значений как раз и составляют нижний предел памяти — приблизительно 1 килобайт.

Такому видеоадаптеру нужен также *генератор символов* (character gerenator), в котором хранились бы изображения символов, показанные ранее. Генератор символов обычно является *постоянным запоминающим устройством* (ПЗУ; Read-Only Memory; ROM) — интегральной микросхемой, сконструированной так, что в ней по определенному адресу всегда находятся одни и те же данные. В отличие от RAM в ПЗУ сигналы для ввода данных не предусмотрены.

Можно считать ПЗУ устройством для преобразования одного кода в другой. Генератор символов как бы переводит 7-битовый код ASCII в 64-битовый код, определяющий внешний вид символа. Микросхеме ПЗУ, в которой записаны изображения 128 символов ASCII размером 8 × 8, нужно 7 адресных входов (для кодов ASCII) и 64 выхода для данных. Но из-за этих многочисленных выходов физический размер микросхемы может оказаться чересчур большим! На деле можно обойтись 8 выходами для данных, увеличив на 3 количество адресных сигналов. 7 адресных сигналов по-прежнему используются для передачи в микросхему кода символа (они поступают в генератор с выходов памяти на видеоплате). 3 дополнительных адресных сигнала обозначают номер строки в квадрате 8 × 8: биты адреса 000 соответствуют самой верхней строке, биты 111 — самой нижней. На выходы подаются биты, составляющие строку, номер которой задан дополнительными адресными входами.

Рассмотрим в качестве примера символ ASCII с кодом 41h — прописную A. Его внешний вид, как и вид остальных символов, зашифрован 8 строками по 8 битов в каждой. В таблице показаны 10-битовые адреса (код символа отделен от кода строки пробелом) и соответствующие выходные сигналы для прописной A.

Адрес	**Сигнал на выходе**
1000001 000	00110000
1000001 001	01111000
1000001 010	11001100
1000001 011	11001100
1000001 100	11111100
1000001 101	11001100
1000001 110	11001100
1000001 111	00000000

Присмотритесь: во втором столбце единичками на фоне нулей нарисована буква A.

Кроме символов, видеоадаптер должен уметь отображать на экране курсор, или указатель ввода — черточку, указываю-

щую, где на экране окажется следующий символ, введенный вами с клавиатуры. Координаты курсора в виде номера строки и номера столбца, в которых он находится, обычно хранятся в двух 8-битовых регистрах на видеоплате. В эти регистры микропроцессор также может записывать данные.

Видеоадаптер, способный отображать не только текст, называется *графическим* (graphics). Записывая данные в память графического адаптера, процессор рисует на экране произвольные изображения, в том числе текст различного размера и различных шрифтов. У графического адаптера требования к памяти выше, чем у текстового. Если разрешение монитора — 320 × 200, в видеопамяти адаптера нужно хранить сведения о 64 000 пикселов. Если каждому пикселу отведен 1 бит, объем видеопамяти равен 64 000 битам, т. е. 8 000 байт, но это, конечно, абсолютный минимум. Пиксел, которому соответствует единственный бит, может передавать лишь два цвета, например, черный (бит равен 0) и белый (бит равен 1).

Даже на черно-белом экране цветов, конечно, отображается гораздо больше, точнее, не цветов, а оттенков серого. Чтобы отображение оттенков серого поддерживал и видеоадаптер, каждому пикселу в соответствие приходится ставить не бит, а байт. Его значение 00h соответствует черному цвету, а FFh — белому. Промежуточные значения соответствуют полутонам. Видеоплате, способной отображать 256 оттенков серого на мониторе с разрешением 320 × 200, нужна память в 64 000 *байт*. Адресное пространство тех простых 8-битовых процессоров, о которых я рассказывал, она займет почти полностью!

Чтобы увидеть на экране буйство красок, накиньте на каждый пиксел еще по 2 байта. Посмотрев на экран телевизора или монитор компьютера через увеличительное стекло, вы обнаружите, что вся гамма цветов представлена на них сочетаниями красного, зеленого и синего. Поэтому-то на каждый пиксел и должно приходиться по 3 байта — для указания интенсивности трех основных цветов (подробнее о них в главе 25). Это увеличивает объем видеопамяти до 192 000 байт.

Число цветов, которые способен воспроизвести данный адаптер, зависит от количества битов на пиксел. Здесь, как и во многих других соотношениях в этой книге, не обошлось без степени двойки

$$\text{Количество цветов} = 2^{\text{Количество битов на пиксел}}$$

Разрешение 320 × 200 — лучшее, чего можно ожидать от обычного телевизора. Полоса пропускания компьютерных мониторов гораздо шире. На первых мониторах, продававшихся с IBM PC, отображалось 25 строк по 80 символов в каждой. Именно таким разрешением обладали текстовые мониторы на мэйнфреймах IBM. Вообще число 80 для IBM обладает особым смыслом. И почему? Да потому, что именно столько символов умещается на одной перфокарте! Мониторы мэйнфреймов часто использовались именно для просмотра содержимого перфокарт. Даже в наши дни отдельные консерваторы продолжают называть строки текстового экрана *карточками*.

С годами разрешение адаптеров и число воспроизводимых ими цветов неуклонно росли. Важной вехой стало появление в 1987 г. адаптеров с разрешением 640 × 480 для компьютеров PS/2 фирмы IBM и «Macintosh II» фирмы Apple. С тех пор это разрешение считается необходимым минимумом для компьютерных видеосистем.

Вы не поверите, но важность разрешения 640 × 480 восходит к Томасу Эдисону. Году в 1889 Эдисон и его инженер Вильям Диксон (William Dickson) работали над аппаратом «Kinetograph» для создания «движущейся фотографии» и над проектором «Kinetoscope». Они решили, что ширина «движущейся фотографии» должна на треть превосходить ее высоту. Отношение ширины кадра к его высоте называется *характеристическим отношением* (aspect ratio). В аппаратах Эдисона и Диксона оно равнялось 1,33:1, или 4:3. В течение 60 лет эта величина использовалась в большинстве фильмов и в конструкции телевизоров. Лишь в начале 1950-х годов некоторые голливудские студии, преодолев барьер 4:3, начали снимать широкоэкранные фильмы, составившие конкуренцию телевизионным программам.

Характеристическое отношение большинства мониторов, как и у телевизоров, равно 4:3, в чем легко убедиться с помощью линейки. В таком же отношении находятся и числа пикселов по горизонтали (640) и вертикали (480). Это значит, что физическая длина горизонтальной полоски, скажем, из 100

пикселов равна физической длине вертикальной полоски из 100 пикселов. Иначе говоря, компьютерные пикселы — квадратные, что довольно удобно.

Практически все современные адаптеры и мониторы работают с разрешением 640 × 480, кроме того, поддерживая видеорежимы с более высоким разрешением — 800 × 600, 1024 × 768, 1280 × 960 и 1600 × 1200.

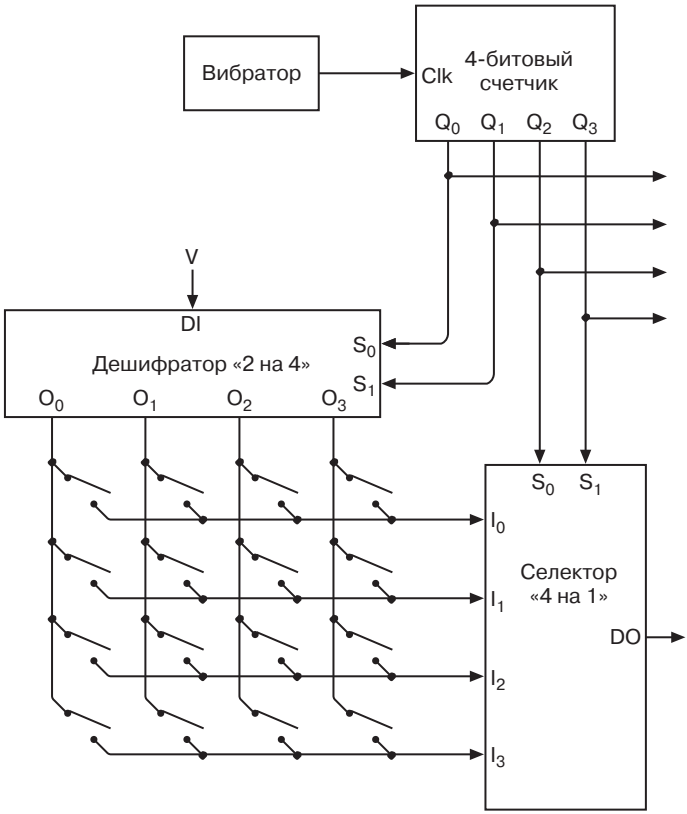
Часто возникает впечатление, что монитор и клавиатура как-то связаны между собой: то, что вы набираете на клавиатуре, сразу отображается на экране. Однако в действительности это совершенно не зависящие друг от друга устройства.

Клавиши на клавиатуре — простые переключатели, которые замыкаются при нажатии. На первых клавиатурах было по 48 клавиш, на современных — больше сотни.

В конструкции клавиатуры, подключенной к компьютеру, должно иметься некое устройство, которое сопоставляло бы каждой нажатой клавише уникальный код. Кажется, что эти коды удобно увязать с кодировкой ASCII, но это только кажется: так делать не стоит. Например, клавиша A должна была бы вырабатывать код 41h при нажатой клавише Shift или код 61h, если клавиша Shift не нажата. Кроме того, на современных клавиатурах есть клавиши, которым ни один код ASCII не соответствует. Код, вырабатываемый клавиатурой, называется *скан-кодом* (scan code). Проверить, связан ли скан-код нажатой клавиши с каким-либо кодом ASCII, поможет небольшая программа.

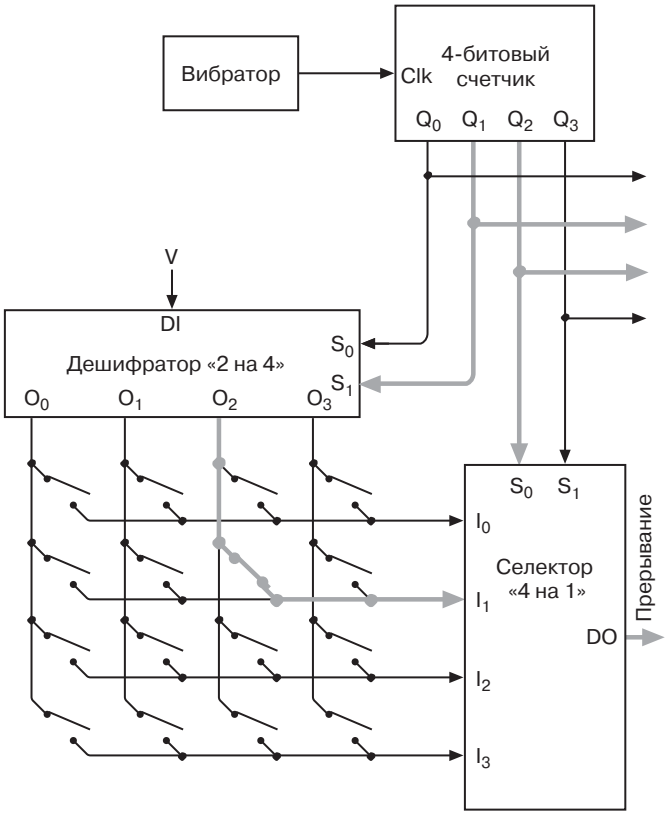
Мне бы не хотелось, чтобы схема клавиатуры была совершенно нечитаемой, поэтому я буду считать, что на ней всего 16 клавиш. Когда одна из них нажата, клавиатура генерирует соответствующий 4-битовый код, принимающий двоичные значения от 0000 до 1111.

Все компоненты на схеме клавиатуры нам знакомы:



Клавиши на этой схеме показаны как переключатели в нижнем левом углу. 4-битовый счетчик циклически с большой скоростью перебирает 16 допустимых кодов клавиш. Происходить это должно очень быстро, быстрее, чем человек успевает нажимать на клавиши.

Выходные сигналы 4-битового счетчика подаются как на дешифратор «2 на 4», так и на селектор «4 на 1». Если ни одна клавиша не нажата, ни один из входов селектора не равен 1. Поэтому и выход самого селектора не равен 1. Но если клавиша нажата, по достижению счетчиком ее скан-кода выход селектора будет равен 1. Скажем, если нажата вторая сверху и справа клавиша, то выход селектора будет установлен в 1 при значении счетчика 0110.



Никакой другой сигнал счетчика не приведет к тому, что выход селектора станет равен 1. Каждой клавише соответствует собственный код, единственный и неповторимый.

Для 64 клавиш вам потребуется 6-битовый скан-код и 6-битовый счетчик. Используя дешифратор «3 на 8» и селектор «1 из 8», клавиши можно расположить в виде таблицы 8 на 8. Для клавиатуры, на которой от 65 до 128 клавиш, скан-код должен быть 7-битовым. Эти клавиши можно расположить в виде таблицы 8 на 16, применив дешифратор «4 на 16» и селектор «1 из 8» (или дешифратор «3 на 8» и селектор «1 из 16»).

Происходящее в этой схеме далее зависит от уровня сложности интерфейса клавиатуры. Например, в клавиатуре мо-

жет присутствовать память RAM, по 1 биту на клавишу. Адресацию к ней будет осуществлять счетчик, а значения кодироваться так: 1, если клавиша нажата, и 0, если нет. Для определения состояния клавиши к этой памяти может обращаться процессор.

Полезнейшая часть интерфейса клавиатуры — сигнал прерывания. Как вы помните, у процессора 8080 есть входной сигнал, позволяющий внешнему устройству прервать работу процессора. Процессор реагирует на прерывание, считывая из памяти команду — обычно команду RST, которая вызывает переход в определенную область памяти, где находится программа обработки прерывания.

Последнее периферийное устройство, которое я здесь опишу, — внешний накопитель. Вы, конечно, помните, что оперативная память теряет содержимое при выключении питания, из чего бы она ни была собрана: из реле, вакуумных ламп или транзисторов. Но компьютеру необходимо и долговременное запоминающее устройство. В старые добрые времена роль такого устройства играли перфокарты и перфоленты. При каждом компьютере хранились рулоны лент или стопки карт с сохраненными на них программами и данными.

Однако и у перфолент, и у перфокарт есть общий недостаток — их нельзя использовать повторно. Заклеить пробитую в бумаге дырку не так-то просто. Кроме того, *физический* объем, занимаемый данными, слишком велик. В наши дни, если вы *видите* сохраненный бит, смело можете считать, что он занимает чересчур много места!

Поэтому сейчас гораздо шире распространены магнитные накопители, родословная которых идет с 1878 г. Принципы их работы описаны американским инженером Оберлином Смитом (Oberlin Smith) (1840–1926), а первое *работающее* устройство собрал в 1898 г. датский изобретатель Вальдемар Паульсен (Valdemar Poulsen) (1869–1942). «Телеграфон» Паульсена должен был записывать сообщения, оставленные по телефону, если некому было ответить на звонок. Для записи звука на стальной проволоке в этом первом автоответчике использовался электромагнит — вездесущее устройство, с которым мы уже встречались. Он намагничивал проволоку пропорционально звуковым колебаниям. Затем проволока протягивалась вдоль обмоток электромагнита и индуцировала в них ток, ко-

торый воспроизводил исходный звук. Кстати, независимо от типа магнитного накопителя электромагнит, используемый в нем для записи и считывания информации, называется *головкой* (head).

В 1928 г. австриец Фриц Пфлоймер (Fritz Pfleumer) запатентовал магнитное записывающее устройство, в котором в качестве носителя использовалась бумажная лента, покрытая железными частичками (технология покрытия изначально была разработана для нанесения металлизированных полосок на сигареты). Бумагу вскоре заменила более прочная целлулоидная основа. Так родился популярнейший способ записи информации. Магнитная лента — теперь удобно упакованная в пластмассовые кассеты — и в наши дни повсеместно используется для записи и воспроизведения музыки и кино.

Первая коммерческая система для записи на магнитную ленту компьютерных данных была выпущена в 1950 г. фирмой Remington Rand. На одной катушке ленты шириной в полдюйма (1,27 см) умещалось несколько мегабайт информации. А в первых домашних компьютерах для записи информации приспосабливали обычные магнитофонные кассеты. Специальные программы позволяли записать на кассету участок памяти, а затем считать его обратно. У первого IBM PC был специальный разъем для кассетного накопителя. Теперь ленты используются в основном для архивирования данных. Для повседневного хранения информации они не столь удобны: чтобы добраться до нужного фрагмента записи, ленту приходится перематывать, а на это иногда уходит немало времени.

С точки зрения ускорения доступа записывать данные удобнее на диск. В современных накопителях диск крутится под неподвижной штангой, по которой перемещаются одна или несколько головок. Доступ к любому участку диска осуществляется практически мгновенно.

Для записи звуков магнитные диски в действительности использовались еще до магнитной ленты. А вот первый дисковый накопитель для компьютерных данных был изобретен в 1956 г. в IBM. Устройство RAMAC (Random Access Method for Accounting and Control) состояло из 50 металлических дисков диаметром 60 см и могло хранить 5 Мб данных.

С тех пор физические размеры дисков существенно уменьшились, а их объем неимоверно вырос. Дисковые накопители

обычно разделяют на *гибкие* (floppy), или попросту дискеты, и *жесткие* (hard). Гибкий диск состоит из пластикового диска с магнитным покрытием и пластмассового корпуса. В современных дискетах пластмассовый корпус защищает диск от изгиба, поэтому называть его гибким можно лишь условно. Для записи и чтения информации с гибкого диска его нужно вставить в специальное компьютерное устройство — дисковод. Самые первые гибкие диски были по 8 дюймов в диаметре. В IBM PC использовались гибкие диски диаметром 5 1/4 дюйма. Сейчас самый популярный их размер — 3,5 дюйма. Особенность гибких дисков в том, что с их помощью информацию легко можно переносить с компьютера на компьютер. В частности, их до сих пор часто используют как дистрибутивный носитель коммерческого ПО.

Жесткий диск обычно состоит из нескольких металлических дисков, «намертво» вмонтированных в дисковод. Работают жесткие диски быстрее гибких, и информации на них умещается больше. С другой стороны, их уже не так легко перенести с компьютера на компьютер.

Поверхность диска разделена на концентрические кольца — *дорожки* (tracks), причем каждая разделена на несколько дуг — *секторов* (sectors). Каждый сектор хранит определенный объем информации, как правило, 512 байт. В первом IBM PC использовалась лишь одна сторона 5 1/4-дюймовой дискеты, разделенная на 40 дорожек по 8 секторов каждая. Поскольку объем сектора был равен 512 байт, полная емкость дискеты составляла 163 840 байт, или 160 кб. В современных 3,5-дюймовых дискетах для IBM-совместимых компьютеров используются обе стороны, разделенные на 80 дорожек каждая (по 18 секторов на дорожке). Полный объем дискеты равен 1 474 560 байт, или 1 440 кб.

Из ПК первым жестким диском был укомплектован компьютер IBM PC XT (1983 г.). Емкость его составляла 10 Мб. В наши дни никого уже не удивишь 20-гигабайтником.

Для работы как с гибкими, так и жесткими дисками нужен специальный электронный интерфейс, обеспечивающий обмен данными с процессором. В наши дни применяются жесткие диски с несколькими типами интерфейсов, в том числе SCSI (Small Computer System Interface), ESDI (Enhanced Small Device Interface) и IDE (Integrated Device Electronics). В всех этих

интерфейсах для обмена данными между накопителем и оперативной памятью в обход микропроцессора используется прямой доступ к памяти.

Новички в компьютерных делах обычно легко путаются в гигабайтах и мегабайтах, в оперативной памяти и жестких дисках. Чтобы всегда отличать одно от другого, помните, что *памятью* называют только оперативную память, оставив термины «накопитель» или «запоминающее устройство» для всего остального — дискет, жестких дисков, магнитных лент… Я старался всегда следовать этому правилу.

Наиболее очевидное отличие оперативной памяти от накопителя в том, что память энергозависима: ее содержимое уничтожается при выключении питания. Накопители обладают большей стойкостью: содержимое дискеты или жесткого диска остается неизменным, пока вы не сотрете его или не запишете поверх новые данные. Другое важное отличие поймет лишь человек, знакомый с работой процессора: адресные выходы процессора используются только для обращения к памяти и никогда к накопителю.

Чтобы с данными *мог* работать микропроцессор, их нужно перенести из накопителя в оперативную память. Для доступа к диску микропроцессор должен запустить специальную небольшую программу.

Если угодно, используйте для понимания различий между оперативной памятью и диском такую аналогию. Оперативная память — это ваш рабочий стол. Со всем, что на нем лежит, вы можете работать сразу. Дисковый накопитель — это шкаф с папками. Чтобы воспользоваться информацией из шкафа, нужно встать, подойти к нему, вытащить папку и вернуться с ней к столу. Если на столе нет места, можно поставить некоторые папки обратно в шкаф.

Конечно, делается это все не руками. Сохранение информации и ее считывание относятся к компетенции очень важной программы — операционной системы (ОС).

Глава 22

Операционная система

Наконец-то мы завершили сборку компьютера (правда, воображаемого), снабдив его процессором, оперативной памятью, клавиатурой, монитором и жестким диском. Оборудование готово к работе, все провода подсоединены, и мы нетерпеливо посматриваем на тумблер с надписью «Вкл», который вдохнет жизнь в наше детище. Не исключено, что точно такие же эмоции обуревали Виктора Франкенштейна и Папу Карло.

Но нам все еще кое-чего недостает, причем отнюдь не удара молнии или волшебных слов «крекс-фекс-пекс». Не мешкайте же — включите компьютер и скажите мне, что произошло.

Когда монитор нагрелся, вы увидели на нем аккуратно расставленный по строкам и столбцам, но абсолютно бессмысленный набор ASCII-символов. В этом, конечно, нет ничего неожиданного. Питание было выключено, и все предыдущее содержимое видеопамяти стерто. При включении компьютера она заполнилась случайным набором битов, какими заполнена и оперативная память. Но процессор ничего не знает об этом и считает их кодами программы, которую должен выполнить. Ничего по-настоящему *плохого*, конечно, не случится, и компьютер не взорвется, но и назвать его работу исключительно продуктивной тоже нельзя.

Нам определенно не достает программного обеспечения (ПО). После включения или перезапуска процессор выполняет команды, записанные начиная с определенного адреса в памя-

ти. Для процессора 8080 этот адрес — 0000h. В умело собранном компьютере при его включении по этому адресу всегда находится исполняемая команда (вероятно, первая из многих).

Как она туда попадает? Запись ПО в новорожденный компьютер — вероятно, один из наиболее запутанных этапов нашего проекта. Самый прямолинейный способ — применить для этого пульт управления памятью из главы 16:

От предыдущего варианта его отличает переключатель Сброс, подключенный к одноименному входу микропроцессора. Пока он замкнут, процессор не работает. Разомкните переключатель, и процессор начнет выполнять программу.

Чтобы запустить компьютер с помощью такого пульта, для начала замкните переключатель Сброс. Работа процессора будет остановлена. Затем замкните переключатель Перехват, чтобы перехватить управление адресными линиями и линиями данных шины. Задайте переключателями A_0–A_{15} 16-битовый адрес в памяти. Содержимое соответствующей ячейки будет показано лампочками D_0–D_7. Чтобы записать в нее новое число, задайте его с помощью переключателей D_0–D_7, а затем включите и выключите переключатель Запись. Введя в память всю программу, разомкните переключатели Перехват и Сброс. Микропроцессор начнет выполнять команды.

Вот так вводятся программы в наш компьютер! Нет нужды говорить, что это очень сложно. При вводе команд вы будете иногда ошибаться — это тоже ясно. Так что считайте мозоли на пальцах и кашу в голове своими профессиональными заболеваниями.

Но все ваши муки будут вознаграждены, когда вы увидите на мониторе результаты работы программы! У текстового дисплея, о котором мы говорили в главе 21, есть своя видеопа-

мять емкостью 1 кб, используемая для хранения в ASCII-кодах 25 строк текста по 40 символов в каждой. Данные в нее записываются так же, как и в обычную оперативную память.

Писать на мониторе не так просто, как кажется. Допустим, в вашей программе проводятся некие вычисления, результатом которых является число 4Bh. Как отобразить его на экране? Просто записать его в видеопамять нельзя — оно будет проинтерпретировано как ASCII-код, и на экран будет выведена буква K, которой этот код соответствует. Чтобы увидеть на экране число 4Bh, вы должны ввести в видеопамять *два* числа: 34h, т. е. код ASCII символа 4, и 42h — код ASCII символа B. Каждая тетрада в 8-битовом результате записывается одной шестнадцатеричной цифрой, которая и отображается на экране.

Конечно, для перевода цифр в соответствующие ASCII-коды нужно написать короткую подпрограмму. Вот как выглядит на языке ассемблера 8080 программа для преобразования тетрады, записанной в аккумуляторе (предполагается, что число заключено между 00h и 0Fh включительно), в ASCII-код соответствующей шестнадцатеричной цифры:

```
NibbleToAscii:  CPI A,0Ah       ; Проверяем, буква или
                                ; цифра
                JC Number
                ADD A,37h       ; Цифры A-F преобразуем в
                                ; 41h-46h
                RET
Number:         ADD A,30h       ; Цифры 0-9 преобразуем в
                                ; 30h-39h
                RET
```

Чтобы преобразовать записанный в аккумуляторе байт в пару цифр ASCII и записать их в регистры B и C, подпрограмму NibbleToAscii придется вызывать дважды:

```
ByteToAscii: PUSH PSW    ; Сохраняем аккумулятор
   RRC                   ; Четырежды сдвигаем содержимое A,
   RRC                   ; чтобы добраться до старшей
   RRC                   ; тетрады
   RRC

   CALL NibbleToAscii    ; Преобразуем в ASCII-код
```

```
MOV B,A              ; Перемещаем результат в регистр B
POP PSW              ; Возвращаем содержимое
                     ; аккумулятора
AND A,0Fh            ; Получаем младшую тетраду
CALL NibbleToAscii   ; Преобразуем в ASCII-код
MOV A,C              ; Перемещаем результат в регистр C
RET
```

Эти подпрограммы позволяют отображать на экране в виде цифр шестнадцатеричные числа. Если вы захотите увидеть на мониторе десятичные числа, повозиться придется больше.

Не забывайте, что в действительности программы на ассемблере в память не вводятся. Вы их пишете на бумаге, затем вручную переводите в машинные коды, а вот уже их вносите в память.

Как прибор наш пульт управления очень прост, но работать с ним почти невозможно. Смело утверждаю, что это наихудшее устройство ввода-вывода в истории компьютерной техники. Очень досадно, что мы, такие умные, собрали на пустом месте компьютер, а информацию в него вводим в двоичном формате. Избавление от пульта управления — вот наша приоритетная задача.

Заменить его нужно, конечно, клавиатурой. Клавиатура, показанная в прошлой главе, прерывает работу процессора при каждом нажатии клавиши. Контроллер прерываний заставляет процессор в ответ исполнять команду RST с номером прерывания в качестве аргумента. Допустим, это команда RST 1. В результате ее выполнения содержимое программного счетчика сохраняется в стеке, а затем происходит переход по адресу 0008h. С помощью пульта управления в эту и следующие ячейки мы введем специальную программу — *обработчик клавиатуры*.

Сначала нам придется выполнять подготовительные операции — мы назовем их *инициализацией*. Инициирующая программа должна задавать значение счетчика стека, чтобы стек корректно размещался в памяти. Во все ячейки видеопамяти нужно записать ASCII-код пробела — 20h, чтобы избавиться от «мусора» на экране. По команде OUT (Output, вывод) программа-инициализатор задает начальное положение курсора (черточки, указывающей, где будет введен следующий символ) — первый столбец первой строки. Следующая коман-

да — EI — разрешает использовать прерывания (чтобы можно было работать с клавиатурой), а команда HLT останавливает работу процессора.

На этом инициирующая программа заканчивается, и процессор переводится в состояние останова, в котором будет пребывать большую часть времени. Вывести его из этого состояния может замыкание переключателя Сброс или прерывание от клавиатуры.

Программа-обработчик клавиатуры гораздо длиннее инициализатора — ведь именно она выполняет полезную работу.

Когда на клавиатуре нажата клавиша, сигнал прерывания заставляет процессор перейти от последней выполненной команды (HLT) к обработчику клавиатуры. С помощью команды IN (Input, ввод) обработчик определяет, какая клавиша была нажата. В зависимости от ее скан-кода он что-то делает (*обрабатывает* нажатие клавиши) и выполняет команду RET, которая возвращает процессор к команде HLT, где он может спокойно ждать нажатия следующей клавиши.

Если нажата буква, цифра или знак препинания, обработчик по скан-коду определяет соответствующий код ASCII, учитывая, была ли нажата клавиша Shift. Затем обработчик записывает найденный ASCII-код в видеопамять, чтобы вывести на экран в текущем положении курсора нужный символ, а курсор передвигается на следующую позицию. При вводе нескольких символов они выстроятся друг за другом в одну строку.

Если нажата клавиша Backspace (ASCII-код 08h), обработчик стирает символ, введенный последним, и возвращает курсор на одну позицию назад. Реального стирания данных из памяти, конечно, не происходит — просто поверх кода этого символа записывается ASCII-код пробела (20h).

Обычно ввод информации происходит так: пользователь набирает строку символов, при необходимости исправляя ошибки с помощью клавиши Backspace, а когда доходит до конца строки, нажимает клавишу Enter. Можно настроить обработчик так, что после нажатия Enter (ее ASCII-код — 0Dh) он будет интерпретировать символы в видеопамяти как *команду* для компьютера, т. е. указание совершить какое-то действие. Для работы с командами включим в обработчик клавиатуры особый фрагмент кода — *командный процессор*, понимающий, например, три команды: W, D и R.

Если введенная текстовая строка начинается с буквы W, она представляет собой указание *записать* (write) байты в память. Например, команда, которая на экране выглядит так:

```
W 1020 35 4F 78 23 9B AC 67
```

указывает командному процессору записать в оперативную память байты 35h, 4Fh и т. д., начиная с адреса 1020h. Чтобы выполнить эту задачу, обработчику клавиатуры придется преобразовать ASCII-коды в шестнадцатеричные числа. Эта операция обратна той, что я недавно демонстрировал.

Команда, начинающаяся с символа D, приказывает *отобразить* (display) содержимое некоторых ячеек памяти. В ответ на команду:

```
D 1030
```

командный процессор выводит на экран 11 байт, записанных в памяти, начиная с адреса 1030h. Именно столько байт можно показать в 40-символьной строке, помимо команды и адреса.

Наконец, командой R вы *запускаете* (run) программу:

```
R 1000
```

Эта команда приводит к запуску программы, начинающейся по адресу 1000h. Командный процессор записывает адрес из аргумента в пару регистров HL и выполняет команду PCHL, загружающую содержимое регистров HL в программный счетчик, т. е. практически осуществляющую переход по заданному адресу.

Написав и отладив обработчик клавиатуры и командный процессор, вы совершаете важный шаг вперед. Отныне вам не придется страдать от примитивности пульта управления. Вводить данные с клавиатуры быстрее, легче и *красивее*.

Увы, введенная информация по-прежнему исчезает при выключении питания компьютера. Вероятно, коды обработчика клавиатуры и командного процессора стоит записать в ПЗУ. В главе 21 я уже говорил о микросхеме ПЗУ, позволяющей хранить данные о внешнем виде символов ASCII, подразумевая, что все нужные биты «зашиты» в микросхему в процессе ее производства. Такие микросхемы называют *программируемыми постоянными запоминающими устройствами* (ППЗУ). Их можно запрограммировать только однажды. Но

есть и *стираемые постоянные запоминающие устройства* (СППЗУ). Их можно перепрограммировать, стерев прежнее содержимое ультрафиолетовым излучением.

Как вы помните, для указания диапазона адресов плат памяти мы применяли DIP-переключатели. Если вы работаете с процессором 8080, вероятно, диапазон адресов одной из плат компьютера начинается с 0000h. После ввода данных в микросхему ПЗУ этот адрес будет принадлежать ей, так что плату памяти придется переключить на другой диапазон.

Создание обработчика клавиатуры можно считать важным этапом работы не только потому, что он облегчает ввод данных в память, но и потому, что обработчик сделал компьютер *интерактивным* (interactive). Вы нажимаете клавиши, и вводимые символы сразу отображаются на экране.

Записав командный процессор в ПЗУ, можно начинать эксперименты с записью данных на жесткий диск (вероятно, порциями, размер которых совпадает с размером сектора на диске) и считыванием их оттуда обратно в память. Хранить данные и программы на жестком диске намного безопаснее, чем в оперативной памяти. Кроме того, диск в сравнении с ПЗУ более многофункционален.

Для работы с диском в командный процессор придется добавить новые команды, например, команду S (store, сохранить):

```
S 2080 2 15 3
```

чтобы записать фрагмент памяти, начинающийся в ячейке 2080h, на диск по адресу — сторона 2, дорожка 15, сектор 3 (размер фрагмента равен размеру сектора). Пару команде S составит L (load, загрузить), выполняющая обратное действие:

```
L 2080 2 15 3
```

Конечно, вам придется помнить, что и куда вы сохранили, и не убирать карандаш с блокнотом далеко от компьютера. Будьте внимательны: записать на диск программу из одной области памяти, а затем загрузить ее в другую нельзя. В командах перехода будут записаны старые адреса, поэтому по новому адресу программа работать не будет. Если размер программы окажется больше размера сектора, вам придется сохранять ее в нескольких секторах. Поскольку часть места на диске, вероятно, занята другими данными, сектора с одной и той же программой не обязательно будут следовать друг за другом.

Очень скоро вы решите, что сидеть перед компьютером и продолжать записывать на бумажке адреса секторов и отмечать, что в них сохранено, — слишком трудоемко. Значит, вы созрели для разработки *файловой системы* (file system).

Файловая система — это способ организации информации на диске, при котором она разделяется на *файлы*, т. е. наборы данных, объединенные общим смыслом, записанные в одном или нескольких секторах. Самое важное: каждому файлу можно присвоить имя, которое поможет вам запомнить, что именно в нем содержится. Если вам нравится сравнивать диск с конторским шкафом, считайте, что имя файла — это бирка, наклеенная на папку с бумагами.

Файловая система почти всегда является частью большого собрания программ — *операционной системы* (operating system), или ОС. Обработчик клавиатуры и командный процессор, о которых мы говорили, вполне могут стать основой для ОС, но мы, пожалуй, не станем тратить времени на ее разработку. С действием ОС и с ее основными функциями мы познакомимся на конкретных примерах.

Исторически самой важной ОС для 8-битовых процессоров стала CP/M (Control Program for Micros, управляющая программа для микрокомпьютеров), написанная Гэри Килдаллом (Gary Kildall) (род. 1942) в середине 1970-х годов для процессора 8080. Позже ее автор стал основателем корпорации Digital Research.

Система CP/M хранилась на диске. Поначалу самым популярным носителем для нее была односторонняя 8-дюймовая дискета с 77 дорожками, 26 секторами на дорожке, 128 байтами в секторе (всего 256 256 байтов). Сама система содержалась на первых двух дорожках. Как она попадала с диска в оперативную память, я расскажу чуть позже.

Остальные 75 дорожек использовались для хранения файлов. Файловая система CP/M довольно проста, но удовлетворяет двум основным требованиям. Во-первых, каждый файл на диске имеет имя, которое также записано на диске. Вообще вся информация, нужная CP/M для работы с файлами, хранится на диске вместе с ними. Во-вторых, файлы на диске могут располагаться в несмежных секторах. Работая на компьютере, вы постоянно создаете и удаляете файлы разных размеров, и потому свободное пространство на диске быстро фрагментируется. В таких обстоятельствах умение системы разбрасывать файл по несмежным свободным секторам весьма полезно.

Секторы 75 дорожек для размещения файлов группируются в блоки по 8 секторов (или по 1 024 байта). Всего на диске 243 блока, пронумерованных от 0 до 242.

В первых двух блоках (2 048 байт) записан *каталог* (directory) — область диска, где хранятся имена и другая важная информация обо всех файлах на диске. Каждому файлу соответствует *элемент каталога* (directory entry) длиной 32 байта. Так как полный размер каталога 2 048 байт, максимальное число файлов, которое можно сохранить на дискете, — 64.

Элемент каталога содержит следующие сведения:

Байты	**Значение**
0	Обычно равен 0
1–8	Имя файла
9–11	Тип файла
12	Продолжение
13–14	Зарезервированы (равны 0)
15	Секторов в последнем блоке
16–31	Карта диска

Первый байт элемента задействуется, только если файловая система применяется в многопользовательском режиме. В CP/M этот байт, а также байты 13 и 14 обычно равны 0.

Имя файла в CP/M состоит из двух частей. Первая часть — собственно *имя файла* (filename) длиной до 8 символов — записывается в байты 1–8. Длина второй — *типа файла* (file type) — ограничена тремя символами, для хранения которых предназначены байты 9–11. Некоторые типы файлов являются стандартными. Например, тип TXT указывает на текстовый файл, т. е. файл, содержащий только ASCII-коды, а тип COM — на файл с командами для процессора 8080, т. е. программу. При указании полного имени файла имя и тип разделяются точкой:

```
MYLETTER.TXT
CALC.COM
```

Такой способ именования файлов известен как 8.3 (восемь-точка-три) — 8 символов имени, точка и 3 символа типа.

В байтах с картой диска указаны блоки, в которых хранится файл. Если, например, первые 4 байта карты равны 14h, 15h,

07h и 23h, а остальные — 0, это значит, что файл занимает 4 блока, т. е. 4 килобайта. В реальности файл может оказаться короче. В байте 15 записано число 128-байтовых секторов, реально занятых файлом в его последнем блоке.

Длина карты диска — 16 байт. Этого хватит, чтобы указать положение файла длиной до 16 384 байтов. Информация о файле длиной более 16 кб хранится в нескольких элементах каталога, называемых *продолжениями* (extents). Байт 12 первого элемента каталога устанавливается в 0, а в элементах-продолжениях он равен 1, 2, 3 и т. д.

Выше я упомянул *текстовые файлы* (text files), известные также как ASCII-файлы. Такой файл содержит только коды ASCII, включая коды возврата каретки и перевода строки, и характеризуется тем, что его содержимое можно читать. Файл, лишенный этих качеств, называется *двоичным* (binary). Двоичными, например, являются файлы типа COM, поскольку в них содержатся не ASCII-коды, а коды команд процессора 8080.

Допустим, нам нужно сохранить в файле (очень коротком) три 16-битовых числа: 5A48h, 78BFh и F510h. Двоичный файл с этими числами будет иметь размер всего 6 байт:

```
48 5A BF 78 10 F5
```

Это, конечно, формат хранения многобайтовых чисел, принятый в Intel: младший байт записывается первым. Программа для процессоров Motorola записала бы их так:

```
5A 48 78 BF F5 10
```

В ASCII-файле те же 16-битовые значения будут сохранены в таком виде:

```
35 41 34 38 68 0D 0A 37 38 42 46 68 0D 0A 46 35 31 30 68
0D 0A
```

Эти числа — ASCII-коды цифр и букв, которыми записываются 16-битовые числа. За каждым числом следуют символы возврата каретки (0Dh) и перевода строки (0Ah). ASCII-файл удобнее отображать в виде не строки кодов, а самих символов:

```
5A48h
78BFh
F510h
```

Текстовый файл с тремя этими числами может выглядеть и так:

```
32 33 31 31 32 0D 0A 33 30 39 31 31 0D 0A 36 32 37 33 36
0D 0A
```

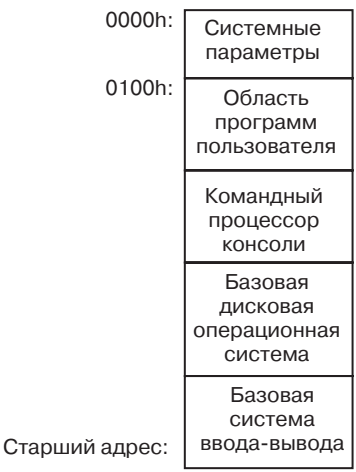
Эти байты являются ASCII-кодами тех же чисел, но в десятичном представлении:

```
23112
30911
62736
```

Поскольку текстовые файлы обычно предназначены для чтения и должны быть максимально понятны человеку, в них, конечно, предпочтительнее использовать десятичные числа.

Как я уже говорил, система CP/M хранится на первых двух дорожках диска. Чтобы запустить систему, ее нужно переписать с диска в память. Программа, выполняющая это действие, хранится в микросхеме ПЗУ и называется *загрузчиком программы раскрутки* (bootstrap loader). Загрузчик считывает с дискеты первый сектор (128 байт), загружает его в память и запускает. Коды, содержащиеся в этом секторе, загружают в память оставшуюся часть системы CP/M. Весь процесс называется *загрузкой* (booting).

В конце загрузки система полностью размещается в оперативной памяти, занимая ее старшие адреса. Структура памяти после этого такова:



Масштаб здесь не соблюден. Три основных компонента системы — базовая система ввода-вывода (Basic Input/Output System, BIOS), базовая дисковая ОС (Basic Disk Operating System, BDOS) и командный процессор консоли (Console Command Processor, CCP) — занимают в памяти всего около 6 кб. Область программ пользователя (Transient Program Area, TPA) — около 58 кб на компьютере с оперативной памятью 64 кб — поначалу не содержит ничего.

Командный процессор консоли выполняет приблизительно те же функции, что и простой командный процессор, о котором мы говорили в начале главы. *Консолью* обычно называют совокупность клавиатуры и дисплея. Процессор консоли отображает на дисплее *приглашение* (prompt) системы, которое выглядит так:

```
A>
```

Наличие приглашения на экране означает, что вы должны что-то ввести. На компьютерах с несколькими дисками буквой A обозначается первый из них, тот, с которого была загружена система CP/M. Набрав команду, вы нажимаете клавишу Enter. Процессор CCP обрабатывает команду, обычно выводя на экран какую-то информацию. Когда выполнение команды закончено, на экране вновь появляется приглашение.

Процессор CCP распознает не так много команд. Самая важная из них, вероятно, DIR — она отображает на экране содержимое каталога диска, т. е. имена всех сохраненных на нем файлов. Символы * и ? позволяют ограничить диапазон выводимых имен файлов. Так, команда:

```
DIR *.TXT
```

вы выведете на экран список всех текстовых файлов, а команда:

```
DIR A???B.*
```

список всех файлов, имена которых состоят из 5 символов, начинаются на A и заканчиваются на B.

Команда ERA (erase, удалить) применяется для удаления файла с диска. Например, в результате выполнения:

```
ERA MYLETTER.TXT
```

с диска будет удален файл MYLETTER.TXT, а командой:

```
ERA *.TXT
```

вы удалите с диска все текстовые файлы. Удаление файла означает освобождения элемента каталога и пространства на диске, которое этот файл занимал.

Для переименования файла служит команда REN (rename, переименовать), а TYPE (напечатать) выводит на экран содержимое текстового файла. SAVE (сохранить) сохраняет на диске с заданными именем один или несколько 256-байтовых блоков оперативной памяти.

Если введенная команда процессору CCP неизвестна, он считает, что введено имя файла с программой на диске. Тип таких файлов — всегда COM. Процессор ищет на диске файл с таким именем, а найдя, загружает в область программ пользователя, которая начинается с адреса 0100h. Например, если вы введете после приглашения команду:

```
CALC
```

процессор найдет на диске файл CALC.COM, загрузит его в память, начиная с адреса 0100h, а затем начнет выполнение команд, расположенных в ячейке 0100h и следующих за ней.

Раньше я говорил, что в принципе программы могут располагаться в памяти где угодно, но у программ для системы CP/M этой свободы нет. Они могут начинаться только в ячейке 0100h.

В комплект CP/M входит несколько готовых программных файлов, например, программа для копирования файлов PIP (Peripheral Interchange Program, программа обмена с периферией) и текстовый редактор ED для создания и изменения текстовых файлов. Небольшие программы наподобие PIP и ED, предназначенные для решения простых задач, часто называют служебными программами или *утилитами* (utilities). Для решения более сложных задач в CP/M служат коммерческие *прикладные программы* (applications), например, текстовые процессоры и электронные таблицы. Все они, как и программы, созданные вами, хранятся на диске в файлах с типом COM.

До сих пор мы говорили лишь о том, как с помощью CP/M (или другой ОС) выполняется повседневная работа с файлами, а также как происходит загрузка и исполнение программ. Однако у ОС есть и третье предназначение.

Программе, выполняемой под CP/M, часто бывает нужно вывести что-то на экран, или считать символы, введенные с клавиатуры, или сохранить данные диск, или прочитать их с диска. Однако обычно программа непосредственно с оборудованием не общается.

Для выполнения этих задач в CP/M включено несколько вспомогательных подпрограмм, к которым обращаются прикладные программы. Назначение этих подпрограмм — обеспечить программисту легкий доступ к оборудованию компьютера, включая монитор, клавиатуру и диск, не заботясь при этом о том, как физически соединены эти компоненты. Самое важное, пожалуй, то, что программе для CP/M *не нужно* беспокоиться о дорожках и секторах на диске. Это работа для ОС. Программе же можно записывать и считывать файлы целиком, не обращая внимания на то, как именно они размещены на диске.

Это и есть третья основная функция ОС — дать программам легкий доступ к оборудованию компьютера, т. е. обеспечить их *интерфейсом прикладного программирования* (Application Programming Interface, API).

Чтобы задействовать API в системе CP/M, программа записывает в регистр C величину, называемую номером функции, и выполняет команду:

```
CALL 5
```

Например, чтобы получить ASCII-код клавиши, нажатой на клавиатуре, в программу нужно вставить команды:

```
MVI C,01h
CALL 5
```

После выполнения этих команд ASCII-код нажатой клавиши будет содержаться в аккумуляторе. Команды:

```
MVI C,02h
CALL 5
```

выводят на экран символ, ASCII-код которого находится в аккумуляторе, и сдвигают курсор в следующую позицию.

Чтобы создать файл, вы должны записать в пару регистров DE адрес области памяти, в которой содержится имя файла, а затем выполнить команды:

```
MVI C,16h
CALL 5
```

В данном случае на диске будет создан пустой файл с заданным именем. Затем с помощью других функций программа может записывать в этот файл данные, а также в конце работы *закрыть* его. Позже та же или другая программа могут *открыть* этот файл и прочитать записанные в нем данные.

Что же делает команда CALL 5? В системе CP/M ячейка с адресом 0005h содержит команду JMP, которая осуществляет переход в область памяти, отведенную базовой дисковой ОС. В ней размещаются подпрограммы для выполнения различных функций CP/M. Система BDOS, как следует из ее имени, отвечает в основном за работу файловой системы. При этом она часто обращается к подпрограммам базовой системы ввода-вывода. Только этим последним и приходится реально иметь дело с оборудованием компьютера. Процессор CCP да и все служебные программы CP/M все свои задачи решают с помощью функций BDOS.

Интерфейс API является *аппаратно независимым*. Это значит, что при написании программы для CP/M вы не должны ничего знать о том, как работают на данном компьютере клавиатура, монитор или диск. Для работы с этими устройствами вы применяете функции CP/M. Преимущество такой организации работы очевидно: программа будет работать на любом компьютере с самыми различными моделями внешних устройств, конечно, при условии, что на компьютере установлен процессор 8080 или другой, понимающий систему команд 8080, например, 8085 фирмы Intel или Z-80 фирмы Zilog. Доступ к оборудованию осуществляется не напрямую, а с помощью функций CP/M. Без стандартного API-интерфейса пришлось бы для каждого компьютера писать собственную программу.

На компьютерах с процессором 8080 CP/M была очень популярна, и важность ее трудно переоценить. Она оказала большое влияние на операционную систему QDOS, написанную Тимом Патерсоном (Tim Paterson) из Seattle Computer Products для 16-разрядных процессоров 8086 и 8088 фирмы Intel. Позже QDOS была переименована в 86-DOS и стала собственностью фирмы Microsoft. Под именем MS-DOS и PC-DOS она устанавливалась на первых компьютерах IBM PC. Для них же

была создана и 16-разрядная версия CP/M — CP/M-86, но популярности MS-DOS она не достигла никогда. Лицензии на установку MS-DOS продавались и другим производителям IBM-совместимых компьютеров.

Файловая система CP/M в MS-DOS не используется. Ее место заняла схема, основанная на применении *таблицы размещения файлов* (File Allocation Table, FAT), изобретенной в Microsoft в 1977 г. Дисковое пространство в файловой системе MS-DOS разделено на *кластеры* (clusters), размер которых в зависимости от размера диска варьируется от 512 до 16 384 байтов. Каждый файл записывается в несколько кластеров. В элементе каталога для файла указывается только *начальный* кластер, а в таблице FAT для каждого кластера хранятся сведения о том, в каком кластере находится продолжение файла.

Длина элемента каталога в MS-DOS также 32 байта, а имена файлов подчиняются той же структуре 8.3, что и в CP/M. Правда, то, что в CP/M называлось типом файла, в MS-DOS называется его *расширением* (extension). Списка блоков в элементе каталога нет. Вместо него в элемент включены такие полезные сведения, как дата и время последнего изменения файла, а также его размер.

Структура первых версий MS-DOS мало отличалась от структуры CP/M. Правда, системы BIOS в ОС уже не было, так как в IBM PC микросхема ПЗУ BIOS встроена в сам компьютер. Командный процессор MS-DOS хранится в файле COMMAND.COM. Программы MS-DOS бывают двух типов. Программные файлы с расширением COM не могут быть больше 64 кб. Более объемные программы хранятся в файлах с расширением EXE.

Хотя поначалу интерфейс CALL 5 для функций API в MS-DOS поддерживался, для новых программ рекомендовалось применять интерфейс *программных прерываний* (software interrupt). Программное прерывание подобно вызову подпрограммы за исключением того, что в данном случае программа не должна знать, к какому адресу она в действительности обращается. Функцию API-интерфейса MS-DOS программа вызывает командой INT 21h.

Теоретически предполагается, что прикладная программа общается с оборудованием компьютера только посредством функций ОС. Однако многие программисты, писавшие при-

кладные программы для небольших компьютеров в 1970-х и начале 1980-х, зачастую обходили ОС, особенно при работе с дисплеем. Программы, обращавшиеся прямо к видеопамяти, работали гораздо быстрее, чем те, в которых применялись функции API. Если же при работе программы возникала необходимость вывода на экран графических изображений, использование функций ОС становилось совершенно неприемлемым. Многим программистам в MS-DOS больше всего нравилось именно то, что система без необходимости «не путалась под ногами» и не мешала программам самим обращаться к оборудованию.

По этой причине во многих программах для IBM PC использовались специфические особенности оборудования, изготавливаемого этой компанией. Производителям компьютеров, которые должны были составить конкуренцию IBM PC, зачастую приходилось копировать эти особенности, иначе возникал риск, что какая-нибудь популярная программа на их компьютерах будет работать неэффективно или вообще не будет работать. В документации к программам часто указывалось, что они предназначены только для работы на IBM PC или на «100%-совместимом компьютере».

В версию MS-DOS 2.0 (март 1983 г.) была добавлена поддержка жестких дисков, которые в те времена были невелики, но быстро увеличивались в емкости. Чем вместительнее диск, тем больше файлов на нем можно хранить. Чем больше файлов на диске, тем сложнее в них разобраться.

В качестве решения этой проблемы в MS-DOS 2.0 была предложена *иерархическая файловая система*, которую удалось добавить к обычной файловой системе MS-DOS с минимумом изменений. Как вы помните, на диске имеется область, называемая каталогом. В ней хранится список файлов и информация об их положении на диске. В иерархической файловой системе некоторые из этих файлов *сами* могут быть каталогами, т. е. содержать список файлов, некоторые из которых также могут быть каталогами и т. д. Обычный дисковый каталог называется *корневым* (root), а каталоги, вложенные друг в друга, — *подкаталогами* (subdirectories). Файлы, относящиеся к различным проектам, удобно хранить в различных подкаталогах.

Иерархическая файловая система и некоторые другие особенности MS-DOS 2.0 были позаимствованы из ОС UNIX, раз-

работанной в начале 1970-х в Bell Telephone Laboratories. Главными ее создателями были Кен Томпсон (Ken Thompson) (род. 1943) и Деннис Ритчи (Dennis Ritchie) (род. 1941). UNIX разрабатывалась как облегченный вариант ОС Multics, которую в Bell Telephone Laboratories создавали в сотрудничестве с Массачусетским технологическим институтом и компанией General Electric.

UNIX пользуется неизменной популярностью у «крутых» программистов. Обычно ОС пишут под конкретную модель компьютера. UNIX же должна была быть *переносимой* (portable), т. е. легко адаптироваться к любым компьютерным системам.

Во времена разработки UNIX компания Bell Telephone Laboratories входила в состав American Telephone & Telegraph, а потому на эту ОС распространялись различные судебные решения, призванные ограничить монополию American Telephone & Telegraph в телефонной индустрии. Поначалу компании запрещалось продавать UNIX, кроме того, она обязана была выдавать другим организациям лицензии на использование ОС. С 1973 г. университетам, коммерческим и правительственным организациям было выдано множество таких лицензий. Только в 1983 г. корпорации American Telephone & Telegraph разрешено было вернуться в компьютерный бизнес, что она ознаменовала выпуском собственной версии UNIX.

Итогом этой запутанной истории стало отсутствие единой версии UNIX. Эту систему продают разные фирмы под разными названиями и с разными особенностями. В работе над UNIX принимало участие множество людей, и все они оставили в системе свой след. И все же существует и общая «философия UNIX», которой следуют практически все разработчики программ для этой ОС. Одна из основ этой философии — широкое применение текстовых файлов. Логика действия многих программ для UNIX заключается в считывании текстового файла, его обработке и записи на диск нового текстового файла. Служебные программы UNIX можно объединять в цепочки для выполнения последовательных действий с текстовым файлом.

Первоначально UNIX создавалась для компьютеров, которые для одного пользователя были слишком велики и недешевы. На таких машинах благодаря *разделению времени* (time

sharing) одновременно могут работать несколько человек. К компьютеру подключено множество *терминалов*, т. е. дисплеев с клавиатурами. Поочередно выделяя время различным терминалам, ОС создает у пользователя впечатление одновременного выполнения нескольких задач.

ОС, способная одновременно выполнять несколько задач, называется *многозадачной* (multitasking). Она, очевидно, более сложна, чем однозадачные системы MS-DOS и CP/M. Многозадачность усложняет также и организацию файловой системы, так как приходится учитывать возможность одновременного обращения к одному файлу нескольких пользователей. Она также влияет на порядок выделения памяти различным программам. Поскольку нескольким одновременно работающим программам памяти нужно больше, чем одной программе, память компьютера вполне может оказаться исчерпанной. Для решения этой проблемы в многозадачных системах применяется *виртуальная память* (virtual memory): блоки памяти, которые в данный момент не нужны, сохраняются во временных файлах на диске, а по мере надобности возвращаются обратно в память.

В последние годы наиболее интересным результатом развития UNIX стали Фонд бесплатных программ (Free Software Foundation, FSF) и проект GNU, основанные Ричардом Столлменом (Richard Stallman). Название проекта GNU пишется так же, как и название знаменитой антилопы гну, а расшифровывается так: GNU — не UNIX (GNU's not UNIX), что, конечно же, чистая правда. Программы GNU совместимы с UNIX, однако распространяются так, что обратить их в чью-либо собственность нельзя. В результате работы проекта GNU появилось не только множество программ для UNIX, но и ядро ОС Linux. В значительной степени написанная Линусом Торвальдсом (Linus Torvalds) из Финляндии, она стала очень популярна в последние годы.

Начиная с 1980-х, главной тенденцией в развитии ОС стало появление больших сложных систем, подобных Mac OS и Microsoft Windows, в которых для облегчения работы с приложениями интенсивно используются графические средства видеосистем. Я подробно опишу эту тенденцию в главе 25.

Глава 23

Фиксированная точка, плавающая точка

В обыденной жизни мы свободно переходим от целых чисел к дробям и процентам. Мы покупаем пол-упаковки яиц, кассир прибавляет к их цене налог 8 1/4 %, и мы расплачиваемся деньгами, полученными за 2 3/4 часа сверхурочной работы по тарифу в 1,5 раза больше обычного. Подобными числами большинство людей оперирует вполне уверенно, хотя и не всегда умело. Мы даже спокойно воспринимаем слова статистиков «средняя американская семья состоит из 2,6 человека» и не содрогаемся при мысли о том, какое ужасное и повсеместное членовредительство к этому привело.

К сожалению, в компьютерной памяти целые числа и дроби не соседствуют с той же легкостью. Да, в компьютерах все закодировано битами, т. е. в виде двоичных чисел. Но некоторые числа выразить битами легче, чем другие.

Свое знакомство с битами мы начали с того, что представляли с их помощью числа, которые на языке математики называются *положительными натуральными* или *положительными целыми*. Мы также убедились, что для представления целых отрицательных чисел удобно применять дополнения до двух, поскольку это облегчает сложение положительных и отрицательных чисел. В таблице показано, какие целые числа без знака (положительные) и со знаком (отрицательные в виде

дополнения до двух) можно хранить в 8-, 16- и 32-разрядных ячейках памяти.

Число битов	Диапазон целых положительных чисел	Диапазон целых отрицательных чисел
8	От 0 до 255	От –128 до 127
16	От 0 до 65 535	От –32 768 до 32 767
32	От 0 до 4 294 967 295	От –2 147 483 648 до 2 147 483 647

На этом мы и остановились. Но, помимо целых, в математике есть также *рациональные* числа, которые можно представить в виде *дроби*, т. е. *отношения* двух целых чисел. Например, 3/4 — рациональное число, так как оно является отношением 3 и 4. В форме *десятичной дроби* это же число записывается так: 0,75. Несмотря на иную форму записи, это тоже рациональное число, равное отношению 75 и 100.

В десятичной системе счисления слева от десятичного разделителя стоят цифры-множители целых положительных степеней десяти. Цифры справа от десятичного разделителя являются множителями отрицательных степеней десяти. Скажем, число 42 705,684 можно записать так:

$$
\begin{aligned}
&4 \times 10\,000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 \div 10 + \\
&8 \div 100 + \\
&4 \div 1000
\end{aligned}
$$

Здесь я использовал знаки деления, но можно обойтись и без них — с помощью десятичных дробей:

$$
\begin{aligned}
&4 \times 10\,000 + \\
&2 \times 1000 + \\
&7 \times 100 + \\
&0 \times 10 + \\
&5 \times 1 + \\
&6 \times 0{,}1 + \\
&8 \times 0{,}01 + \\
&4 \times 0{,}001
\end{aligned}
$$

или степеней десяти:

$$4 \times 10^4 +$$
$$2 \times 10^3 +$$
$$7 \times 10^2 +$$
$$0 \times 10^1 +$$
$$5 \times 10^0 +$$
$$6 \times 10^{-1} +$$
$$8 \times 10^{-2} +$$
$$4 \times 10^{-3}$$

Некоторые рациональные числа не так-то легко представить в виде десятичной дроби. Самый простой пример — 1/3. Разделив 1 на 3, получим:

$$0{,}33333333333333333333333333333333333333\ldots$$

и так без конца. Нужно признать, что в форме десятичной дроби число 1/3 выглядит несколько неуклюже, и все же это полноценное рациональное число, являющееся отношением двух целых чисел. Существуют и более сложные периодические десятичные дроби, например, 1/7:

$$0{,}142857142857142857142857142857142857142857\ldots$$

Числа, которые нельзя представить в виде отношения целых чисел, называются *иррациональными*. Они записываются в виде бесконечных десятичных дробей без каких бы то ни было повторений. Примером может служить квадратный корень из 2 — решение алгебраического уравнения $x^2 - 2 = 0$:

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856967 18\ldots$$

Иррациональное число, которое *не* является решением алгебраического уравнения с целочисленными коэффициентами, называется *трансцендентным*. Все трансцендентные числа иррациональны, но не все иррациональные числа трансцендентны. К трансцендентным числам относится знаменитое число π, равное отношению длины окружности к ее диаметру:

$$3{,}14159265358979328462643383279502884\ldots$$

Другое трансцендентное число — *e*, к которому стремится значение выражения:

$$\left(1+\frac{1}{n}\right)^n$$

при *n*, стремящемся к бесконечности. Оно приблизительно равно:

2,71828182845904523536028747135266249775724 7...

Вместе рациональные и иррациональные числа называются *действительными* или *вещественными* числами. Противоположность им — *мнимые* числа: квадратные корни из отрицательных величин. *Комплексные* числа являются комбинацией реальных и мнимых чисел. Несмотря на подозрительное название, мнимые числа *существуют* и даже активно используются в решении многих научных и практических задач.

Мы привыкли думать о *непрерывном* ряде чисел. Дайте мне два рациональных числа, и я всегда найду между ними третье, например, их среднее арифметическое. Но цифровые компьютеры не имеют дела с непрерывностью. Бит равен либо 0, либо 1 и никаких промежуточных значений не допускает. По самой своей сути компьютеры обречены работать с *дискретными* величинами. Количество таких величин, представимых с помощью битов, напрямую зависит от количества последних. Так, в 32-битовой ячейке памяти можно хранить положительные целые числа от 0 до 4 294 967 295. Чтобы записать в память компьютера значение 4,5, нужно придумать что-то другое.

Можно ли представить в двоичном виде дробные величины? Можно. Самый простой способ — прибегнуть к двоично-десятичному коду (BCD), о котором мы говорили в главе 19. Код BCD предназначен для записи в двоичной системе десятичных чисел. Каждой десятичной цифре (0, 1, 2, 3, 4, 5, 6, 7, 8 и 9) сопоставлено 4-битовое двоичное число:

Десятичная цифра	**Двоичная число**
0	0000
1	0001
2	0010
3	0011
4	0100

(*продолжение*)

5	0101
6	0110
7	0111
8	1000
9	1001

Код BCD, в частности, удобно применять в компьютерных программах, работающих с денежными суммами, когда в дробных числах в основном достаточно двух знаков после запятой.

Две десятичные цифры, представленные кодом BCD, часто хранят в 1 байте. Такая система записи называется *уплотненным кодом BCD* (packed BCD). Дополнение до двух в уплотненном коде BCD не применяется, поэтому для хранения знака обычно нужен дополнительный *бит знака* (sign bit). Поскольку под число в коде BCD удобно отводить целое число байтов, бит знака фактически занимает в памяти 4 или 8 битов.

Рассмотрим пример. Допустим, вашей программе никогда не приходится встречаться с суммами, превышающими 10 млн. долларов, ни в положительном, ни в отрицательном балансе. Иначе говоря, вам нужно представлять суммы в диапазоне от –9 999 999,99 до 9 999 999,99. Для хранения каждой суммы в памяти нужно отвести 5 байт. Скажем, число –4 325 120,25 пятью байтами представляется так:

```
00010100 00110010 01010001 00100000 00100101
```

или в шестнадцатеричном формате:

```
14h 32h 51h 20h 25h
```

Заметьте: первая слева тетрада равна 1, т. е. число отрицательное. Это и есть бит знака. Если бы число было положительным, первая тетрада равнялась бы 0. Для представления каждой цифры требуется 4 бита; десятичные цифры совпадают с шестнадцатеричными.

Чтобы хранить в памяти значения от –99 999 999,99 до 99 999 999,99, вам потребуется 6 байт: 5 для 10 цифр и еще байт для бита знака.

Такой тип записи дробных чисел называется записью с *фиксированной точкой* (fixed-point)[1], так как десятичный разделитель всегда находится в определенном месте числа. В нашем примере после него всегда стоят две цифры. Вы вольны отводить под дробную часть числа любое количество знаков и использовать в одной и той же программе числа с разным количеством знаков после запятой. Но заметьте: информация о положении разделителя вместе с числом не хранится, поэтому программа должна как-то узнавать, где он находится.

Формат с фиксированной точкой хорош, если вы знаете, что числа не «перерастут» ту область памяти, которую вы для них отвели, и что вам не понадобится увеличить количество знаков после запятой. Но он пасует, если числа становятся слишком большими или маленькими. Представьте, что вам нужно предусмотреть в памяти место для записи разнообразных расстояний. Но что это будут за числа? Расстояние от Земли до Солнца составляет 150 000 000 000 метров, а боровский радиус атома водорода — 0,00000000005 м. Чтобы заранее предусмотреть место для хранения любых промежуточных значений в формате с фиксированной точкой, вам придется отвести для них 12 байт памяти.

Мы, вероятно, выработаем лучший способ хранения подобных чисел, если вспомним о *научной нотации*, которую широко применяют ученые и инженеры. В ней длинные ряды нулей заменяются на степень десяти, благодаря чему удобно представлять очень большие и очень маленькие числа. В научной нотации число:

$$150\,000\,000\,000$$

выглядит как:

$$1{,}5 \times 10^{11}$$

а число:

$$0{,}00000000005$$

— как:

$$5 \times 10^{-11}$$

[1] Вообще-то в России в качестве десятичного разделителя традиционно используется запятая. С другой стороны, описываемые форматы записи десятичных чисел применяются при составлении программ, а в языках программирования в качестве разделителя практически всегда используется точка. Поэтому в книге используются термины «фиксированная точка» и «плавающая точка». — *Прим. перев.*

Число перед степенью 10 (в наших примерах 1,5 и 8) иногда называют *мантиссой* (хотя этот термин уместней применять по отношению к логарифмам). Я буду называть его *значащей частью числа* (significand) в соответствии с компьютерной терминологией.

Порядком (exponent) называется степень, в которую возводится 10. В первом примере порядок равен 11; во втором –11. Порядок позволяет узнать, на сколько знаков смещен десятичный разделитель по сравнению с записью числа в формате с фиксированной точкой.

Традиционно в научной нотации значащая часть заключена между 1 (включительно) и 10. Числа, приведенные ниже, равны:

$$1{,}5 \times 10^{11} = 15 \times 10^{10} = 150 \times 10^{9} = 0{,}15 \times 10^{12} = 0{,}015 \times 10^{13}$$

но первый вариант наиболее предпочтителен. Преобразование числа в научной нотации к стандартному виду называется *нормализацией*.

Заметьте: знак показателя степени указывает на порядок числа, но не на его положительность или отрицательность. Вот как выглядят в научной нотации отрицательные числа:

$$-5{,}8125 \times 10^{7} = -58\,125\,000$$

$$-5{,}8125 \times 10^{-7} = -0{,}00000058125$$

В компьютерах научная нотация стала основой для записи чисел в формате с *плавающей точкой* (floating point). В отличие от формата с фиксированной точкой он идеален для хранения маленьких и больших чисел. Важное отличие от традиционной научной нотации в том, что в компьютерах формат с плавающей точкой используется для *двоичных* чисел. Для начала разберемся, как в двоичном формате выглядят дроби.

Вообще-то все проще, чем кажется. В десятичной записи цифры справа от десятичного разделителя соответствуют отрицательным степеням 10. В двоичной записи цифры справа от двоичного разделителя соответствуют отрицательным степеням 2. Например, двоичное число 101,1101 преобразуется в десятичное так:

$$1 \times 4 +$$
$$0 \times 2 +$$
$$1 \times 1 +$$
$$1 \div 2 +$$
$$1 \div 4 +$$
$$0 \div 8 +$$
$$1 \div 16$$

Заменяем знаки деления отрицательными степенями 2:

$$1 \times 2^2 +$$
$$0 \times 2^1 +$$
$$1 \times 2^0 +$$
$$1 \times 2^{-1} +$$
$$1 \times 2^{-2} +$$
$$0 \times 2^{-3} +$$
$$1 \times 2^{-4}$$

Отрицательные степени двух записываем в виде десятичных дробей:

$$1 \times 4 +$$
$$0 \times 2 +$$
$$1 \times 1 +$$
$$1 \times 0{,}5 +$$
$$1 \times 0{,}25 +$$
$$0 \times 0{,}125 +$$
$$1 \times 0{,}0625$$

В результате получаем, что двоичное число 101,1101 эквивалентно десятичному 5,8125.

В десятичной форме нормализованная значащая часть числа заключена между 1 и 10. Это верно и в двоичной системе: нормализованная значащая часть всегда больше либо равна 1 и меньше 10 (2 в десятичной системе). Поэтому в двоичной нотации число 101,1101 записывается как:

$$1{,}011101 \times 2^2$$

Отсюда интересный вывод: в нормализованном двоичном числе с плавающей точкой слева от разделителя всегда стоит 1 и только 1.

В большинстве современных компьютеров и программ для чисел с плавающей точкой применяется стандарт, введенный

в 1985 г. Институтом инженеров по электротехнике и радиоэлектронике (Institute of Electrical and Electronics Engineers, IEEE) и признанный Американским национальным институтом по стандартизации (American National Standards Institute, ANSI), — *стандарт IEEE для двоичной арифметики с плавающей точкой* (IEEE Standard for Binary Floating-Point Arithmetic). Его описание занимает всего 18 страниц, но основы кодирования двоичных чисел с плавающей точкой изложены в нем очень ясно.

В стандарте IEEE определены два основных формата: формат с *простой точностью* (single precision), в котором для записи числа отводится 4 байта, и формат с *двойной точностью* (double precision), занимающий 8 байт.

Сначала рассмотрим формат с простой точностью. В нем число разделяется на 3 части: 1 бит для знака (0 для положительных чисел и 1 для отрицательных), 8 бит для порядка и 23 бита для дробной значащей части числа, в которой самый младший бит стоит справа:

1 бит знака (s) 8 битов порядка (e) 23 бита дробной значащей части (f)

Всего получается 32 бита, или 4 байта. Поскольку значащая часть нормализованного двоичного числа с плавающей точкой всегда начинается с 1, соответствующий бит в запись числа в формате IEEE *не включается*. Хранится только 23-битовая *дробная часть*, хотя это не мешает говорить, что число хранится с *точностью* 24 бита.

8-битовый порядок может принимать значения от 0 до 255. Он является *смещенным* (biased), т. е. для нахождения истинного значения порядка — с учетом знака — вы должны вычесть из *e* число, называемое *смещением* (bias). Для чисел простой точности с плавающей точкой оно равно 127.

Порядки 0 и 255 имеют особый смысл, о котором чуть позже. Если значение порядка заключено в пределах от 1 до 254, число, представленное конкретными значениями *s* (бита знака), *e* (порядка) и *f* (дробная часть), равно:

$$(-1)^s \times 1.f \times 2^{e-127}$$

Знак числа вычисляется по формуле $(-1)^S$. Если *s* равно 0, число положительно (так как любое число в степени 0 равно +1),

если *s* равно 1, число отрицательно (так как –1 в степени 1 равно –1).

Следующая часть выражения — 1.f — символизирует 23-битовую дробную часть числа, стоящую после 1 и двоичного разделителя (в стандарте IEEE — точка). Все это умножается на 2 в степени *e* – 127.

Кроме того, в стандарте IEEE предусмотрено несколько специальных случаев.

- Если *e* равно 0, а *f* — 0, число равно 0. Обычно 0 представляется нулевыми значениями всех 32 битов. Если бит знака равен 1, число называется *отрицательным 0.* Он символизирует очень маленькое число, для записи которого цифр и степени в простой точности недостаточно, но которое, однако, меньше 0 (не равно ему).
- При *e* = 255 и *f* = 0 число, в зависимости от значения *s* символизирует положительную или отрицательную бесконечности.
- Если *e* = 255, а *f* не равно 0, значение считается *недопустимым числом* и называется *NaN* (Not a Number, не число). Величина NaN часто возникает в результате некорректного математического вычисления.

Самое маленькое нормализованное положительное или отрицательное двоичное число простой точности, которое можно представить в формате с плавающей точкой, равно:

$$1{,}00000000000000000000000_{\text{два}} \times 2^{-126}$$

После запятой стоят 23 двоичных нуля. Самое большое нормализованное положительное или отрицательное двоичное число простой точности, которое можно представить в формате с плавающей точкой, равно:

$$1{,}11111111111111111111111_{\text{два}} \times 2^{127}$$

В десятичном выражении эти числа приблизительно равны $1{,}175494351 \times 10^{-38}$ и $3{,}402823466 \times 10^{38}$. Это и есть диапазон чисел простой точности, которые можно представить в формате с плавающей точкой.

Вы, возможно, еще не забыли, что 10 двоичных цифр приблизительно эквивалентны 3 десятичным. Я хочу сказать, что двоичное число, состоящее из 10 единиц (3FFh в шестнадцате-

ричной системе и 1023 в десятичной), примерно равно десятичному числу, состоящему из трех девяток. Иначе говоря:

$$2^{10} \approx 10^3$$

То есть 24-битовое двоичное число простой точности в формате с плавающей точкой приблизительно эквивалентно 7-значному десятичному числу. Поэтому говорят, что число простой точности с плавающей точкой имеет точность до 24 двоичных или до 7 десятичных знаков. Что это значит?

Точность числа с фиксированной точкой легко определить по его внешнему виду. В денежной сумме, например, число с фиксированной точкой и двумя знаками после разделителя определяется с точностью до цента. В отношении чисел с плавающей точкой ничего подобного сказать нельзя. В зависимости от порядка число с плавающей точкой может быть точным до долей пенса или до нескольких тысяч долларов.

Правильнее сказать, что число простой точности с плавающей точкой точно до 1 части из 2^{24} или приблизительно до 6 частей из ста миллионов. В первую очередь это значит, что если вы запишете в виде чисел простой точности с плавающей точкой величины 16 777 216 и 16 777 217, их представления будут абсолютно одинаковы! Более того, тем же значением будет представлено любое число, заключенное между ними, например, 16 777 216,5. Все три десятичных числа будут храниться в памяти как 32-битовое значение 4B800000h. Если его разделить на биты знака, порядка и значащей части, оно будет выглядеть так:

0 10010111 00000000000000000000000

или:

$$1{,}00000000000000000000000_{\text{два}} \times 2^{24}$$

За ним в сторону увеличения следует число:

$$1{,}00000000000000000000001_{\text{два}} \times 2^{24}$$

соответствующее десятичному 16 777 218. Чаще всего хранение двух близких десятичных чисел в виде одного и того же двоичного особых трудностей не вызывает, но и забывать о такой возможности не следует.

Допустим, вы написали для банка программу, в которой для хранения денежных сумм в долларах и центах использу-

ются числа простой точности с плавающей точкой. Узнав, что с точки зрения программы 262 144,00 долларов не отличаются от 262 144,01 долларов, вы возможно, испытаете глубокое потрясение. Действительно, оба числа представляются в компьютере так:

$$1.00000000000000000000000_{\text{два}} \times 2^{18}$$

Вот почему при работе с долларами и центами формат с фиксированной точкой более предпочтителен. Разрешив точке плавать, вы рискуете нарваться и на другие мелкие, но весьма досадные неприятности. Например, иногда в программе вместо 3,5 будет получаться 3,4999999. При работе с плавающей точкой такие фокусы случаются довольно часто, и поделать с этим ничего нельзя.

Если вас не устраивает простая точность, но применение плавающей точки неизбежно, воспользуйтесь двойной точностью. Числа в этом формате занимают по 8 байт памяти, распределенных так:

1 бит знака (s)	11 бит порядка (e)	52 бита дробной значащей части (f)

Смещение порядка равно 1023 или 3FFh, поэтому число двойной точности записывается так:

$$(-1)^s \times 1.f \times 2^{e-1023}$$

Для значений 0, бесконечности и NaN применяются те же правила, что и в простой точности.

Самое маленькое положительное или отрицательное число двойной точности с плавающей точкой равно:

$$1.00000000000000000000000000000000000000000000000000_{\text{два}} \times 2^{-1022}$$

После двоичной точки стоит 52 нуля. Самое большое число равно:

$$1.11111111111111111111111111111111111111111111111111_{\text{два}} \times 2^{1023}$$

Соответствующий диапазон десятичных чисел — приблизительно от 2,2250738585072014 xxx 10^{-308} до 1,797693138623158 $\times 10^{308}$.

53 бита значащей части числа (с учетом неуказываемого первого бита, который всегда равен 1) примерно эквивалентны 16

десятичным цифрам. Это, конечно, намного больше, чем в простой точности, но все равно от одинакового представления двух близких десятичных чисел нам никуда не деться. Например, в двоичном представлении число 140 737 488 355 328,00 не отличается от 140 737 488 355 328,01. Оба хранятся как 64-битовая величина 42E0000000000000h, равная:

$$1.0000000000000000000000000000000000000000000000000000_{\text{два}} \times 2^{47}$$

Разумеется, разработка формата для хранения чисел с плавающей точкой в памяти компьютера — лишь небольшая часть работы по практическому использованию этих чисел в ассемблерных программах. Если бы вы создавали компьютер на необитаемом острове, то столкнулись бы с необходимостью написания программ для сложения, вычитания, умножения и деления чисел с плавающей точкой. К счастью, их можно разбить на функции для сложения, вычитания, умножения и деления *целых* чисел, работать с которыми вы уже умеете.

Например, сложение двух чисел с плавающей точкой заключается в сложении их значащих частей; труднее всего здесь разобраться с порядками. Рассмотрим пример:

$$(1{,}1101 \times 2^5) + (1{,}0010 \times 2^2)$$

Здесь нужно сложить 11101 и 10010, но не просто, а с приращением разрядности первого числа на величину, определяемую по разности порядков. Первый порядок на 3 больше второго, поэтому добавляем к первому числу 3 нуля и складываем 11101000 и 10010. Итоговая сумма равна $1{,}1111010 \times 2^5$. Иногда порядки различаются столь существенно, что сумма совершенно не отличается от большего из слагаемых. Так произойдет, например, при сложении расстояния от Земли до Солнца с радиусом атома водорода.

Умножение чисел с плавающей точкой равносильно умножению их значащих частей и сложению порядков. Обе операции не выходят за рамки целочисленной арифметики.

Следующий уровень сложности в вычислениях с плавающей точкой — вычисление корней, логарифмов, показательных и тригонометрических функций. Но все эти функции также можно разложить на четыре основные операции: сложение, вычитание, умножение и деление. Так, синус вычисляется с помощью ряда:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Здесь аргумент *x* измеряется не в градусах, а в *радианах* (360° равно 2π радиан), а восклицательным знаком обозначен *факториал* числа, равный произведению всех целых чисел от единицы и до этого числа. Например, 5! = 1 × 2 × 3 × 4 × 5. Таким образом, при вычислении числителя и знаменателя в каждом члене ряда мы используем умножение, делим числитель на знаменатель, а затем складываем и вычитаем результаты. Просто? Да не совсем. Самая неприятная часть этого выражения — многоточие в конце. Оно означает, что теоретически вычисление продолжается *бесконечно*. Но на практике вычислять это выражение для любого *x* не нужно. Синус — периодическая функция, поэтому его достаточно рассчитать для *x* в пределах от 0 до π/2, а затем по простым формулам получить значения синуса для произвольного *x*. Чтобы рассчитать sin *x* при *x* в пределах от 0 до π/2 с точностью до 53 бит, достаточно учесть в этом разложении с десяток слагаемых.

«Что же получается? — скажете вы. — Компьютеры призваны облегчать людям жизнь, а мы вопреки этому постоянно сталкиваемся с необходимостью написания все новых и новых программ!» Но в этом-то и прелесть программирования. Как только кто-то написал программы для вычислений с плавающей точкой, их могут использовать и другие. А с написанием таких программ заминки не возникает. Расчеты с плавающей точкой жизненно важны в научных и инженерных программах, поэтому их реализации всегда придается самое большое значение. В начале компьютерной эпохи написание программ для вычислений с плавающей точкой всегда было одной из первых задач, стоявших перед создателями новой модели компьютера.

На самом деле имеет смысл создать для работы с плавающей точкой специальные машинные команды! Если выполнять операции с плавающей точкой на аппаратном уровне, как умножение и деление в 16-битовых микропроцессорах, сложные математические расчеты будут проводиться гораздо быстрее. Конечно, сказать легче, чем сделать. Но вычисления с плавающей точкой настолько важны, что специальное оборудование для них создано уже давно.

В 1954 г. выпущен IBM 704 — первый коммерческий компьютер, имевший аппаратное обеспечение для работы с плавающей точкой. Все числа в нем хранились как 36-битовые величины: 27 битов под значащую часть, 8 битов для порядка и 1 бит для знака. В процессоре имелись схемы для сложения, вычитания, умножения и деления чисел с плавающей точкой. Другие функции реализовывались программно.

В настольных компьютерах аппаратное обеспечение для арифметики с плавающей точкой появилось в 1980 г., когда компания Intel выпустила микросхему 8087. В наши дни микросхемы этого типа называют *математическими сопроцессорами*. Название *сопроцессор* (coprocessor) означает, что применять эту микросхему отдельно от процессора нельзя. Сопроцессор 8087 работал с первыми 16-разрядными микропроцессорами Intel 8086 и 8088.

Сопроцессор 8087 — микросхема с 40 выводами, в которой используются многие из сигналов микросхем 8086 и 8088. Посредством этих сигналов процессор и сопроцессор взаимодействуют друг с другом. Когда процессор считывает специальную команду (ESC), управление берет на себя сопроцессор, выполняющий следующий за этой командой машинный код — одну из 68 команд для вычисления логарифма, тригонометрических функций, возведения в степень и пр. Типы данных основаны на стандарте IEEE. В свое время сопроцессор 8087 считался самой сложной интегральной схемой.

Можете считать сопроцессор маленьким самостоятельным компьютером. Получив команду на выполнение определенного расчета с плавающей точкой (например, команду FSQRT для вычисления квадратного корня), сопроцессор запускает набор собственных команд, записанных в ПЗУ. Внутренние команды сопроцессора называются *микрокодом*. Обычно выполнение микрокода сопряжено с вычислениями в цикле, поэтому результат готов не сразу. И все же сопроцессор позволяет ускорить вычисления по крайней мере в 10 раз по сравнению с программной реализацией операций с плавающей точкой.

На материнской плате первого IBM PC рядом с микросхемой 8088 имелось 40-контактное гнездо для сопроцессора 8087. К сожалению, при продаже компьютера оно пустовало. Пользователи, нуждавшиеся в высокой скорости вычислений, покупали и устанавливали сопроцессор 8087 самостоятельно.

Но даже после его установки не все приложения начинали работать быстрее. Текстовому редактору, например, арифметика с плавающей точкой почти ни к чему. Но даже программы с обильными математическими расчетами, например, электронные таблицы, зачастую благотворного влияния сопроцессора на себе не испытывали.

Мы уже убедились, что для сопроцессора программист должен вставлять в программу специальные машинные коды. Но сопроцессор не был стандартным оборудованием, и потому многие программисты себя этим не утруждали. Им ведь все равно приходилось писать подпрограммы для работы с плавающей точкой для тех пользователей, у которых не было сопроцессора, поэтому микросхема 8087 не избавляла их от работы, а, напротив, усложняла ее. Позже программисты научились писать приложения так, что, если сопроцессор был на компьютере, он использовался, а если нет — его работа моделировалась программно.

Позже Intel выпустила сопроцессор 287 для процессора 286 и сопроцессор 387 для процессора 386. В 1989 г. сопроцессор наконец перестал был отдельным элементом оборудования, устанавливаемым по желанию, — в процессор Intel 486DX он был встроен! Правда, в 1991 г. Intel снова вернулась к старой схеме, выпустив дешевый процессор 486SX без встроенного сопроцессора и сопроцессор 487SX к нему. Однако в 1993 г. с появлением процессоров серии Pentium сопроцессор снова стал частью процессора, теперь, вероятно, уже навсегда. Компания Motorola объединила сопроцессор с микропроцессором 68040 в 1990 г. До этого сопроцессоры 68881 и 68882 для микропроцессоров семейства 68000 продавались отдельно. Встроенное оборудование для работы с плавающей точкой имеется и в процессорах PowerPC.

Аппаратное обеспечение для арифметики с плавающей точкой — хороший подарок для человека, уставшего программировать на языке ассемблера. Но его значение меркнет в сравнении с другим усовершенствованием, начало которому было положено в 1950-е годы. Наша следующая остановка — языки программирования.

Глава 24

Языки высокие и низкие

Программировать в машинных кодах — все равно что есть зубочисткой. Подхватывать ею кусочки еды столь сложно, а сами они столь малы, что обед растягивается в вечность. Байтами машинных кодов обозначены настолько простые вычислительные задачи — загрузка числа из памяти в процессор, сложение его с другим числом, запись результата обратно в память, — что практически невозможно представить себе, как из таких крохотных кирпичиков складывается целое здание.

Еще в начале главы 22 мы вводили двоичные данные в память переключателями, но с тех пор нам удалось здорово продвинуться вперед. В той же главе я рассказал, как с помощью простых программ задействовать клавиатуру для ввода данных (например, машинных кодов) и просматривать их на экране монитора. После этого работать на нашем воображаемом компьютере стало удобнее, но все возможные усовершенствования нами еще не исчерпаны.

Как вы помните, байтам машинных кодов сопоставлены короткие мнемонические коды MOV, ADD, CALL, HLT, благодаря которым программа по крайней мере отдаленно напоминает набор указаний на обычном английском языке. Часто за мнемокодом следует аргумент, уточняющий действие команды. Например, в системе команд процессора 8080 команда с кодом 46h помещает в регистр B байт из ячейки памяти, адрес которой записан в паре регистров HL. С помощью мнемокода это действие выражается нагляднее:

```
MOV B,[HL]
```

Конечно, составлять программу на языке ассемблера гораздо проще, чем писать ее в машинных кодах. Увы, процессор ассемблера не понимает. Поэтому программу в мнемокодах приходится писать на бумаге. Когда вам кажется, что она готова к запуску, вы вручную преобразуете операторы ассемблера в двоичные машинные коды и вводите их в память.

Но почему не автоматизировать эту задачу? Для этого вполне достаточно компьютера с процессором 8080, работающего под управлением ОС CP/M.

Для начала создайте текстовый файл PROGRAM1.ASM и введите в него программу на ассемблере (тип ASM как раз и означает, что в файле содержится ассемблерная программа). Для создания и редактирования файла можно применить текстовый редактор из комплекта CP/M — программу ED.COM. Слишком сложную задачу перед собой ставить не будем:

```
      ORG 0100h
      LXI DE,Text
      MVI C,9
      CALL 5
      RET
Text: DB 'Hello!$'
      END
```

Почти все операторы этой программы нам знакомы. Команда ORG, с которой мы раньше не встречались, *не входит* в систему команд процессора 8080. Она просто указывает, что следующая команда должна располагаться по адресу 0100h. Как вы помните, начиная именно с этого адреса, CP/M загружает программы в память.

Следующая команда — LXI — загружает 16-битовое значение в пару регистров DE. Здесь адрес значения указан меткой Text, расположенной в конце программы перед оператором DB (Data Byte, байт данных). Аргументами оператора DB могут быть несколько байтовых значений, разделенных запятыми, или (как у нас) текстовая строка, заключенная в апострофы.

Команда MVI помещает в регистр C число 9. Это номер функции CP/M для отображения на экране текстовой строки, начинающейся по адресу, записанному в регистрах DE, и за-

канчивающейся значком доллара $ (немного странно, что окончание строки обозначается именно значком $, но так уж устроена CP/M). Функцию вызывает оператор CALL 5. Наконец, команда RET останавливает выполнение программы и передает управление системе (остановить программу можно и иначе). Оператор END отмечает конец файла с программой на ассемблере.

Итак, у нас есть текстовый файл, содержащий 7 строк. Теперь его нужно ассемблировать, т. е. преобразовать операторы ассемблера в машинные коды. До сих пор мы делали это вручную. Но в CP/M входит специальная программа ASM.COM, которая выполнит это преобразование автоматически. Достаточно ввести после приглашения системы команду:

```
ASM PROGRAM1.ASM
```

Программа ASM находит на диске файл PROGRAM1.ASM и создает файл PROGRAM1.COM, в который помещает машинные коды, соответствующие операторам ассемблера (честно говоря, я пропустил один этап процесса ассемблирования, но для понимания сути происходящего он не важен). После этого программу PROGRAM1 можно запускать из командной строки CP/M. Она отобразит на экране строку «Hello!» и закончит работу.

Файл PROGRAM1.COM содержит следующие 16 байтов:

```
11 09 01 0E 09 CD 05 00 C9 48 65 6C 6C 6F 21 24
```

Первые 3 байта — это команда LXI, следующие 2 — команда MVI. 3 байта занимает команда CALL 5, и 1 — команда RET. Последние 7 байт — это ASCII-коды букв Hello, восклицательного знака и знака $.

Работа программы-ассемблера, например, ASM.COM, сводится к чтению файла с программой на языке ассемблера (программисты часто называют его файлом с исходным кодом) и созданию *исполняемого* (executable) файла, содержащего машинные коды. По сути своей, ассемблеры — очень простые программы, поскольку в основе их работы лежит взаимно однозначное соответствие между мнемоническими и машинными кодами. Ассемблер считывает из файла строку, разделяет ее на мнемокод команды и аргументы, а затем находит в списке машинный код, соответствующий такому сочетанию коман-

ды и аргументов (в списке должны быть *все* возможные сочетания). В результате сравнения появляется последовательность машинных кодов, в точности соответствующая последовательности команд.

Заметьте: вместо метки Text в код подставлен ее адрес 0109h. Именно там окажется текстовая строка после того, как программа будет загружена в память, начиная с адреса 0100h. Обычно программисту, пишущему программы на ассемблере, не приходится беспокоиться из-за подстановки конкретных адресов — это делает ассемблер.

Программисту, написавшему первую программу-ассемблер, конечно, пришлось переводить ее в машинные коды вручную. Следующую, улучшенную версию ассемблера для того же компьютера, уже можно ассемблировать автоматически с помощью первого ассемблера. Новый ассемблер приходится разрабатывать каждый раз при появлении нового процессора. Но заниматься этим можно на компьютере со старым процессором и старым ассемблером.

Конечно, программы-ассемблеры избавляют программиста от самой неблагодарной части работы — ручного перевода операторов в машинные коды. И все же язык ассемблера отягощен двумя крупными недостатками. О первом из них вы, вероятно, уже догадались: программирование на языке ассемблера — задача весьма кропотливая. Непосредственно управляя действием микропроцессора, приходится беспокоиться о бесчисленных мелочах.

Второй недостаток в том, что программа на ассемблере не является *переносимой* (portable). Например, программу, написанную на языке ассемблера для процессора Intel 8080, нельзя запустить на компьютере с процессором Motorola 6800. Предварительно ее придется переписать на ассемблере для этого процессора. Это, конечно, будет несложно, поскольку основные организационные и алгоритмические проблемы вы решили, создавая исходную программу, но работы все же хватит.

В главе 23 я писал о том, что в современные микропроцессоры встроены команды для работы с числами с плавающей точкой. Они, конечно, облегчают программисту жизнь, но не особо. Хотелось бы совершенно забыть о командах процессора для выполнения элементарных арифметических операций,

используя для математических выражений проверенную веками алгебраическую форму записи, например:

$$A \times Sin(2 \times PI + B) / C$$

где A, B и C — произвольные числа, а PI равно 3,14159.

Так в чем проблема? Если это выражение записано в текстовом файле, мы вольны создать программу на языке ассемблера, которая будет читать строки из этого файла и преобразовывать их в машинные коды.

Если бы вам нужно было вычислить это выражение лишь однажды, вы обошлись бы калькулятором. На создание программы вас, вероятно, подвигла необходимость вычислять его многократно, с различными значениями коэффициентов A, B и C. Значит, кроме этого выражения вам понадобятся и другие, для вычисления коэффициентов.

Все это задачи для языка программирования *высокого уровня* (high-level). Язык ассемблера считается языком *низкого уровня* (low-level), поскольку он напрямую взаимодействует с оборудованием компьютера. На практике языками высокого уровня называют любые языки, кроме ассемблера, но это не значит, что все они равнозначны. Уровень одних языков выше уровня других языков. А знаете, что такое язык программирования высочайшего уровня? Это когда президент компании набирает на клавиатуре или, еще лучше, положив ноги на стол, диктует секретарше: «Рассчитать прибыли и убытки за этот год, написать годовой отчет, распечатать 1 000 экземпляров и разослать акционерам»! Конечно, настоящие языки программирования, даже высокого уровня, очень далеки от этого идеала.

Человеческие языки формируются в течение сотен и тысяч лет взаимопроникновения, случайных изменений и заимствований. Даже в основу искусственных языков вроде эсперанто положены реальные языки со всей их многовековой историей. Другое дело — языки программирования. Их создание всегда было продуманным и целенаправленным. Создатель языка программирования воплощал в нем свое представление об общении человека и машины. В 1993 г. было подсчитано, что с 1950-х годов разработано более 1 000 языков программирования.

Конечно, *изобретение* языка (т. е. разработка *синтаксиса* — правил построения выражений) — только полдела. Нужно

также написать *компилятор* (compiler), т. е. программу, преобразующую операторы языка в машинные коды. Подобно ассемблеру, компилятор символ за символом считывает файл с исходной программой, разделяя его на слова, операторы и числа. Написать компилятор, разумеется, гораздо сложнее, чем ассемблер. Оператор ассемблера переводится в единственный машинный код, тогда как оператор языка высокого уровня, как правило, превращается во множество машинных команд. Разработке компиляторов посвящена обширная литература.

У языков программирования высокого уровня есть свои плюсы и минусы. Главное преимущество языка высокого уровня в том, что он обычно намного превосходит ассемблер в легкости изучения и использования. Программа на языке высокого уровня короче и понятнее ассемблерной. Языки высокого уровня в основном переносимы, т. е. одинаково хорошо работают на компьютерах с разными процессорами. Это значит, что, создавая программу, не нужно вникать в тонкости архитектуры компьютера, на котором она будет работать. Конечно, при этом для каждого процессора нужен собственный компилятор. Созданный им исполняемый файл также будет годиться только для этого процессора.

С другой стороны, программа на ассемблере практически всегда эффективнее аналогичной программы на языке высокого уровня. Это значит, что исполняемый файл, созданный компилятором языка высокого уровня, занимает больше места и работает медленнее, чем функционально идентичная программа на ассемблере. Нужно отметить, что в последние годы это правило уже не выполняется с прежней неизменностью. Микропроцессоры стали более сложными, а создатели компиляторов достигли больших успехов в оптимизации генерируемых кодов.

Язык высокого уровня облегчает использование процессора, но не делает его мощнее. Любые действия, которые процессор в состоянии выполнить, могут быть запрограммированы на языке ассемблера. Это значит, что программа на языке высокого уровня не наделяет процессор никакими новыми возможностями. Более того, чтобы сделать программу переносимой, от некоторых специфических возможностей данного процессора в ней приходится отказываться.

Приведу пример. Во многих процессорах есть команды для выполнения побитового сдвига содержимого аккумулятора влево или вправо. Аналогичных команд практически ни в одном языке высокого уровня нет. Чтобы осуществить эту операцию, вам придется заменить ее делением или умножением на 2 (впрочем, во многих современных компиляторах для умножения или деления на степень двойки используются именно команды побитового сдвига). Нет во многих языках высокого уровня и команд для выполнения с битами булевых операций.

Большинство прикладных программ для первых домашних компьютеров писалось на ассемблере. В наши дни его практически не используют, разве что для решения каких-либо специфических задач. Чем изощреннее архитектура процессоров, тем сложнее писать для них ассемблерные программы. Одновременно все совершеннее становятся и компиляторы. Немаловажную роль в снижении популярности ассемблера сыграло и увеличение объема доступной оперативной и дисковой памяти. Программистам более не нужно создавать коды, которые обходились бы минимальным количеством памяти и целиком умещались на одной дискете.

Проектировщики многих первых компьютеров пытались формулировать задания для них с помощью алгебраической записи, но создать действительно работающий компилятор удалось только в 1952 г. Его написала для компьютера UNIVAC Грейс Мюррей Хоппер (1906–1992), работавшая тогда в компании Remington Rand. Грейс Хоппер связала свою жизнь с вычислительной техникой еще в 1944 г., но даже разменяв девятый десяток, продолжала работать в компьютерной индустрии, отвечая за связи с общественностью в корпорации Digital Equipment.

Старейшим языком высокого уровня, не утратившим своей актуальности и сегодня, является ФОРТРАН (FORTRAN), хотя от исходной версии в нем мало что сохранилось. Кстати, названия многих языков программирования принято писать прописными буквами, так как они являются сокращениями. Например, название ФОРТРАНа составлено из словосочетания «FORmula TRANslation» (трансляция формул). ФОРТРАН разработан в IBM в середине 1950-х для компьютеров серии 704 и долгое время интенсивно использовался в научном и инженерном программировании. Он особенно удобен для ма-

тематических расчетов благодаря обширнейшей поддержке операций с плавающей точкой, включая возможность работы с комплексными числами.

У каждого языка масса сторонников и противников. Споры их зачастую протекают весьма эмоционально. Стараясь сохранить нейтралитет, я решил использовать в качестве примера язык, который сейчас уже почти никто не применяет, — АЛГОЛ (ALGOL). Его имя — тоже сокращение, составленное из «ALGOrithmic Language» (алгоритмический язык). Исследовать природу языков высокого уровня на примере АЛГОЛа удобно еще и потому, что он во многих отношениях — прямой предок многих распространенных языков, появившихся за последние 40 лет. Даже в наши дни иногда приходится слышать о «языках программирования типа АЛГОЛа».

Первую версию — АЛГОЛ 58 — разработал в 1957–58 гг. международный комитет программистов. Два года спустя был выпущен усовершенствованный вариант — АЛГОЛ 60. В конце концов дело дошло и до АЛГОЛа 68, но в этой главе я расскажу о версии языка, описанной в документе «Переработанное описание алгоритмического языка АЛГОЛ 60», работа над которым была закончена в 1962, а выход в свет состоялся в 1963 г.

Давайте напишем на АЛГОЛе небольшую программу. Будем считать, что у нас имеется компилятор этого языка ALGOL.COM, работающий в CP/M или, скажем, MS-DOS. Запишем эту программу в файл FIRST.ALG (обратите внимание на тип файла).

Программа на АЛГОЛе заключается между ключевыми словами (keywords) *begin* и *end*. Для начала выведем на экран одну строку текста:

```
begin
   print ('This is my fist ALGOL program!');
ende
```

Для запуска компилятора нужно набрать в командной строке:

```
ALGOL FIRST.ALG
```

В ответ компилятор сообщит о нераспознанном ключевом слове в третьей строке программы:

```
Line 3: Unrecognized keyword 'ende'.
```

В вопросах правописания компилятор даст фору самому придирчивому учителю английского языка. Набирая программу, я неправильно ввел ключевое слово *end*. Компилятор не замедлил сообщить мне о *синтаксической ошибке*. Там, где стоит слово *ende*, компилятор ожидал увидеть известное ему ключевое слово АЛГОЛа.

Исправляем ошибку и запускаем компилятор снова. На диске появляется исполняемый файл — FIRST.COM в CP/M или FIRST.EXE в MS-DOS. Вообще-то чаще для преобразования файла с исходной программой в исполняемый файл нужно выполнить не одно, а два действия, но я для простоты об этом умолчу. Так или иначе, для запуска созданного исполняемого файла в командную строку нужно ввести его имя:

```
FIRST
```

На экране появятся слова:

```
This is my fist ALGOL program!
```

Знатоки английского укоризненно качают головами. В этой фразе тоже есть ошибка: в слове «first» (первый) пропущена буква r. Однако на нее компилятор никак не прореагировал, да это и понятно. В задачу программы входит вывод на экран заданной текстовой строки, и компилятор, конечно, не анализирует ее содержимое.

Вы, наверное, уже догадались, что оператор *print* осуществляет вывод на экран некоей информации, в данном случае — текстовой строки. В этом смысле показанная выше программа на языке АЛГОЛ эквивалентна программе на языке ассемблера, приведенной в начале главы. Оператор *print* в официальную спецификацию языка АЛГОЛ не входит, но мы будем считать, что у нашего компилятора есть соответствующая *встроенная функция* (built-in function). Большинство операторов АЛГОЛа (не считая *begin* и *end*) должны заканчиваться точкой с запятой. Вставлять отступ перед оператором *print* не обязательно, но вообще отступы часто используют, чтобы прояснить структуру программы.

Теперь напишем программу для умножения двух чисел. В любом языке программирования имеется понятие *переменной* (variable). Переменная обозначается буквой или сочетанием букв, например, коротким словом. Хотя для хранения чисел

по-прежнему используются ячейки памяти, переменные позволяют обращаться к ним без явного указания адреса, что делает программу понятнее. Нам понадобятся переменные *a*, *b* и *c* — для двух сомножителей и для произведения:

```
begin
   real a, b, c;

   a := 535.43;
   b := 289.771;
   c := a × b;

   print ('Произведение ', a, ' и ', b, ' равно ', c);
end
```

Оператор *real* используется для описания переменных, т. е. для введения их имен и указания типа. В данном случае, переменные *a*, *b* и *c* предназначены для хранения вещественных чисел, т. е. чисел с плавающей точкой (для описания целых переменных в АЛГОЛе служит оператор *integer*). Практически во всех языках имена переменных могут включать цифры, но начинаться должны обязательно с буквы. Применение пробелов и специальных символов, как правило, не допускается. Во многих компиляторах ограничена и длина имени. Я в примерах будут использовать имена переменных, состоящие из одной буквы.

Если наш компилятор АЛГОЛа поддерживает стандарт IEEE для хранения чисел с плавающей точкой, для любой из трех переменных понадобится 4 байта с простой точностью и 8 байт — с удвоенной.

Следующие три выражения являются операторами *присваивания* (assignment). В АЛГОЛе их легко распознать по двоеточию и знаку равенства (в других языках используется только знак равенства). В левой части оператора присваивания стоит имя переменной, в правой — выражение. Переменной присваивается значение, полученное в результате вычисления выражения. В первых двух операторах присваиваются конкретные численные значения. В третьем операторе в переменную *c* записывается произведение переменных *a* и *b*.

В современных языках символ × не используется, так как его нет в кодировках ASCII и EBCDIC. Вместо него применяется символ *. Деление в АЛГОЛе обозначается косой чертой (/), ÷ обозначает деление нацело, а символ ↑, также не входящий в кодировку ASCII, — возведение в степень.

Для отображения переменных и констант служит оператор *print*. Отдельные элементы списка вывода разделяются запятыми. Отображение на экране текста — не великая заслуга, но если вы захотите вывести на экран значение числовой переменной с плавающей точкой, оно будет автоматически преобразовано в ASCII:

```
Произведение 535.43 и 289.771 равно 155152.08653
```

Выведя эту строку, программа завершает работу и передает управление ОС.

Чтобы найти произведение двух других чисел, вам придется отредактировать программу, перекомпилировать ее и запустить вновь. Чтобы не возиться с перекомпиляцией, вызовите встроенную функцию *read*:

```
begin
   real a, b, c;

   print ('Введите первое число: ');
   read (a);

   print ('Введите второе число: ');
   read (b);

   c := a × b;

   print ('Произведение ', a, ' и ', b, ' равно ', c);
end
```

Оператор *read* считывает введенные с клавиатуры ASCII-символы и преобразует их в числа с плавающей точкой.

Очень важной структурным понятием в языках высокого уровня является *цикл* (loop), позволяющий многократно выполнить один и тот же фрагмент кода. Допустим, вы хотите написать программу, вычисляющую кубы чисел 3, 5, 7 и 9. Выглядеть она будет примерно так:

```
begin
   real a, b;

   for a := 3, 5, 7, 9 do
   begin
      b := a × a × a;
      print ('Куб числа ', a, ' равен ', b);
   end
end
```

В операторе *for* переменной *a* присваивается значение 3, а затем выполняется оператор, следующий за ключевым словом *do*. Если таких операторов несколько (как у нас), их нужно разместить между ключевыми словами *begin* и *end*. Операторы между двумя этими словами называются *блоком* (block). Те же операторы выполняются для значений переменной *a* 5, 7 и 9.

У оператора *for* есть и другой вариант. Следующая программа вычисляет кубы всех нечетных чисел от 3 до 99:

```
begin
   real a, b;

   for a := 3 step 2 until 99 do
   begin
      b := a × a × a;
      print ('Куб числа ', a, ' равен ', b);
   end
end
```

В операторе *for* переменной *a* присваивается значение 3, а затем выполняется блок операторов, идущий за ключевым словом *do*. При следующем выполнении цикла значение переменной *a* увеличивается на значение, указанное после ключевого слова *step*, т. е. на 2. Блок операторов выполняется с переменной *a*, равной 5. Затем значение переменной *a* увеличится еще на 2. Выполнение цикла завершится, когда значение *a* станет больше 99.

Синтаксические правила языков программирования обычно не допускают никаких исключений. В АЛГОЛе 60, например, за ключевым словом *for* может идти только имя переменной.

Другой важный элемент языка программирования — условные структуры, позволяющие организовать выполнение опе-

ратора или блока операторов только при соблюдении определенного условия. В приведенном ниже примере используется встроенная функция АЛГОЛа *sqrt*, вычисляющая квадратный корень. С отрицательными числами она не работает, что учитывается в программе.

```
begin
   real a, b;

   print ('Введите число: ');
   read (a);

   if a < 0 then
      print('Извините, введено отрицательное число.');
   else
      begin
         b := sqrt(a);
         print (‘Квадратный корень из ', a, ' равен ', b);
      end
end
```

Символ < соответствует математическому знаку «меньше». Если пользователь ввел отрицательное число, выполняется первый оператор *print* в структуре *if*. Если введенное число больше 0 или равно ему, выполняется блок, содержащий второй оператор *print*.

До сих пор все переменные в этой главе использовались для хранения одиночных значений. Но в одной переменной их можно хранить и несколько. Такая переменная называется *массивом* (array). В программе на АЛГОЛе массив описывается так:

```
real array a[1:100];
```

Здесь переменную *a* предполагается использовать для хранения 100 чисел с плавающей точкой, называемых элементами массива. Для обращения к первому элементу массива используется обозначение *a[1]*, ко второму — *a[2]*, к последнему — *a[100]*. Число в скобках называется *индексом* (index).

В следующей программе вычисляются квадратные корни всех чисел от 1 до 100. Результаты вычислений сохраняются в массиве, а затем выводятся на печать.

```
begin
   real array a[1:100];
   integer i;

   for i := 1 step 1 until 100 do
      a[i] := sqrt(i);

   for i := 1 step 1 until 100 do
      print ('Квадратный корень из ', i, ' равен ', a[i]);
end
```

Помимо массива, в программе описана целая переменная *i*. В первом цикле каждому элементу массива присваивается значение квадратного корня из его индекса. Во втором цикле значения элементов массива выводятся на печать.

В дополнение к типам *real* и *integer* в АЛГОЛе имеется также тип *Boolean*. Переменные этого типа могут принимать лишь два значения: *true* и *false*. Массив булевых переменных поможет нам написать последнюю в этой главе программу, которая составляет список простых чисел с помощью алгоритма, известного как решето Эратосфена. Эратосфен (около 276–196 до н. э.) был библиотекарем в знаменитой Александрийской библиотеке и более всего прославился вычислением длины окружности Земли.

Простыми называют целые числа, которые без остатка делятся только на себя и на 1. Самое маленькое простое число — 2. Оно же — единственное четное простое число. Далее идут 3, 5, 7, 11, 13, 17 и т. д.

В методе Эратосфена рассматривается ряд целых чисел, начинающийся с двойки. Сначала из списка вычеркиваются все числа, кратные наименьшему простому числу (2), т. е. все четные числа, кроме самой двойки. Следующее по порядку простое число — 3. Вычеркиваем из списка все числа, кратные ему. За тройкой идет 4, но это число уже вычеркнуто, так как оно четно. 5 — опять простое число, и мы вычеркиваем из списка все числа, которые без остатка делятся на 5. По мере того как мы будем продвигаться вперед, невычеркнутыми будут оставаться только простые числа.

В программе на АЛГОЛе для определения всех простых чисел, меньших 10 000, используется булев массив, индекс которого принимает значения от 2 до 10 000.

```
begin
   Boolean array a[2:10000];
   integer i, j;

   for i := 2 step 1 until 10000 do
      a[i] := true;

   for i := 2 step 1 until 100 do
      if a[i] then
         for j := 2 step 1 until 10000  ÷  i do
            a[j  ×  j] := false;

   for i := 2 step 1 until 10000 do
      if a[i] then
         print (i);
end
```

В первом цикле всем элементам булева массива присваивается значение *true*, т. е. изначально программа полагает, что простыми являются все числа. Во втором цикле переменная *i* пробегает значения от 2 до 100 (квадратный корень из 10 000). Если значение *i* — простое число (это значит, что *a[i]* равно *true*), во вложенном цикле всем элементам массива с номерами, кратными *i*, присваиваются значения *false* (эти числа простыми не являются). В последнем цикле на печать выводятся номера всех элементов массива, равных *true*, т. е. все простые числа.

Иногда приходится слышать споры о том, является программирование наукой или искусством. С одной стороны, вы вспоминаете о приятеле, получившем степень магистра *компьютерных наук*, с другой стороны, на полке у вас стоит знаменитая книга Дональда Кнута (Donald Knuth) «Искусство программирования». Физик Ричард Фейнман писал так: «Я бы сказал, что программирование сродни машиностроению — всего-то нужно заставить что-то сделать что-то».

Попросите 100 человек написать программу для печати простых чисел, и вы получите 100 разных решений. Даже те,

что воспользуются для решения решетом Эратосфена, напишут программу иначе, чем я. Если бы программирование было наукой, вряд ли у одной задачи было бы так много правильных решений, а неверные решения были бы более очевидны. Зачастую программирование связано с творческими озарениями и интуитивными решениями, и это сближает его с искусством. И все-таки процесс проектирования и создания программы не отличается существенно, скажем, от строительства моста.

Первыми программистами были в основном ученые и инженеры, которые умели формулировать свои задачи на языке математики, положенном в основу ФОРТРАНа и АЛГОЛа. Однако на протяжении всей истории языков программирования неоднократно предпринимались попытки разработать язык, который могли бы использовать и люди, не столь близко знакомые с математикой.

Одним из первых языков, специально предназначенных для бизнеса, был КОБОЛ (COBOL), созданный в конце 1950-х комитетом из представителей промышленности и Министерства обороны США. КОБОЛ широко применяется и по сей день. Его название расшифровывается как «COmmon Business Oriented Language» (язык, ориентированный на общие коммерческие задачи). Одно из основных требований, предъявлявшихся к КОБОЛу, заключалось в том, чтобы менеджеры, сами не занимавшиеся программированием, могли хотя бы *читать* программы и убеждаться, что они делают именно то, что должны делать (чего на практике, конечно, не бывает).

В КОБОЛе имеются обширные возможности по чтению *записей* (records) и созданию *отчетов* (reports). Записью в программировании называется собрание взаимосвязанных сведений. Например, страховая компания может вести базу данных с информацией о проданных полисах. Отдельные элементы этой базы и есть записи, в которых хранятся имя клиента, дата его рождения и другие сведения. Поначалу для хранения информации в программах на КОБОЛе использовали 80-столбцовые перфокарты IBM. Для максимальной экономии места в номере года на картах зачастую указывались только две последние цифры, что позже отчасти обусловило знаменитую «проблему 2000 года».

В середине 1960-х в IBM разработали для компьютеров System/360 язык PL/I (Programming Language I, язык программирования №1). Предполагалось, что в PL/I будут объединены модульная структура программ на АЛГОЛе, обширный математический аппарат ФОРТРАНа и средства КОБОЛа для работы с записями. Но уровня популярности ФОРТРАНа или КОБОЛа этот язык так и не достиг.

Компиляторы ФОРТРАНа, АЛГОЛа, КОБОЛа и PL/I создавались и для домашних компьютеров, но ни один из них не оказал на эти машины такого влияния, как БЕЙСИК.

Язык БЕЙСИК (BASIC, Beginner's All-purpose Symbolic Instruction Code, универсальный символьный программный код для начинающих) разработан в 1964 г. Джоном Кемени (John Kemeny) и Томасом Курцем (Thomas Kurtz) из Дартмутского университета. Большинство студентов в Дартмуте не были ни математиками, ни инженерами, поэтому их не стоило заставлять возиться с перфокартами или сложными языками программирования. Вместо этого студент, сидя перед терминалом, набирал простую программу прямо на экране. Если строка начиналась с номера, она считалась строкой программы на БЕЙСИКе. Строка без номера считалась командой для системы. Например, командой SAVE пользователь сохранял программу на диске, командой LIST — выводил ее на экран, а RUN — компилировал и запускал. Самое первое печатное руководство по БЕЙСИКу начиналось с такой программы:

```
10 LET X = (7 + 8) / 3
20 PRINT X
30 END
```

В отличие от АЛГОЛа в БЕЙСИКе программист не должен был указывать тип переменной. Большинство переменных по умолчанию считались вещественными.

Во многих последующих реализациях БЕЙСИКа использовались не компиляторы, а *интерпретаторы* (interpreters). Я уже говорил, что компилятор считывает файл с исходной программой целиком, а затем создает исполняемый файл. Интерпретатор считывает программу оператор за оператором и сразу выполняет их. При этом исполняемый файл не создается. Разрабатывать интерпретаторы проще, чем компиляторы, но работает интерпретируемая программа, как правило, медлен-

нее скомпилированной. На домашних компьютерах дебют БЕЙСИКа состоялся в 1975 г., когда два приятеля Билл Гейтс (Bill Gates) (род. 1955) и Пол Аллен (Paul Allen) (род. 1953) написали интерпретатор БЕЙСИКа для компьютера «Альтаир 8800». Этот интерпретатор стал первым продуктом основанной ими корпорации Microsoft.

Язык программирования Паскаль (Pascal) унаследовал структуру АЛГОЛа и средства КОБОЛа для работы с записями. Он разработан в конце 1960-х швейцарским профессором информатики Николасом Виртом (Niklaus Wirth). Среди программистов компьютеров IBM PC Паскаль был очень популярен, правда, только в одной специфической реализации — Turbo Pascal фирмы Borland. Эта программа, написанная Андерсом Хейлсбергом (Anders Hejlsberg) из Дании (род. 1960), поступила в продажу в 1983 г. Она представляла собой *интегрированную среду разработки* (Integrated Development Environment, IDE) — текстовый редактор и компилятор были объединены в единую программу, что существенно облегчало разработку кодов. На больших компьютерах интегрированные среды использовались задолго до этого, но с Turbo Pascal началось их пришествие на персональные компьютеры.

На Паскале частично основан язык программирования Ада, разработанный для Минобороны США. Он назван в честь Августы Ады Байрон, которую я упоминал в главе 18, рассказывая об Аналитической Машине Бэббиджа.

И наконец — C (Си). Этот чрезвычайно популярный язык был создан в 1969–1973 гг. в основном усилиями Денниса Ритчи (Dennis Ritchie) из Bell Telephone Laboratories. Часто спрашивают, откуда взялось название C. Ответ прост: его предшественником был язык B, который в свою очередь был упрощенным вариантом BCPL (Basic CPL), основанного на CPL (Combined Programming Language, комбинированный язык программирования).

Помните, я говорил в главе 22 о переносимости ОС UNIX? В те времена ОС, как правило, писались на языке ассемблера для конкретного процессора. В 1973 г. UNIX была написана (точнее, переписана) на C, и с тех пор язык и система идут по жизни рука об руку.

Одно из основных качеств программы на C — краткость. Например, для указания границ блоков используются не клю-

чевые слова, как в АЛГОЛе или Паскале, а фигурные скобки: { и }. Или другой пример. В программах часто возникает необходимость прибавить к переменной некое число и записать результат в ту же переменную:

```
i = i + 5;
```

В C этот оператор можно сократить:

```
i += 5;
```

Если значение переменной нужно увеличить на 1, оператор становится еще короче:

```
i++;
```

На 16-разрядном или 32-разрядном микропроцессоре этот оператор превращается в единственную машинную команду.

Чуть раньше я говорил, что в большинстве языков высокого уровня нет операций побитового сдвига или булевых операций над битами. Язык C — исключение из этого правила. Кроме того, важной особенностью C является поддержка *указателей* (pointers), т. е. фактически работы непосредственно с адресами в памяти. Из-за этой близости к командам процессора C иногда называют *языком ассемблера высокого уровня*.

В*се* языки типа АЛГОЛа, т. е. *большинство* распространенных языков, предназначены для компьютеров с архитектурой Неймана. Вырваться из пут неймановской модели при разработке языка нелегко, но еще сложнее убедить других людей им пользоваться. Один из таких «не-неймановских» языков — LISP (LIst Processing, обработка списков), созданный в конце 1950-х Джоном Маккарти (John MacCarthy) — используется при работах в области искусственного интеллекта. Не менее экзотичен, чем LISP, хотя и не похож на него, APL (A Programming Language), созданный также в конце 1950-х Кеннетом Айверсоном (Kenneth Iverson).

Так или иначе, пока в программировании доминируют языки типа АЛГОЛа, хотя в последние годы в них внесено несколько важных усовершенствований, результатом которых стало появление *объектно-ориентированных* (object-oriented) языков. Эти языки очень удобны при работе в графических операционных системах, о которых речь пойдет в следующей (последней) главе книги.

Глава 25

Графическая революция

10 сентября 1945 г. читателей журнала «Life» ожидала в основном обычная пестрая мозаика статей и фотографий: материалы о конце Второй мировой войны, описание жизни танцора Вацлава Нижинского в Вене, иллюстрированное эссе о профсоюзе United Auto Workers. Кроме того, в номере был и неожиданный материал: статья Ванневара Буша (Vannevar Bush) (1890–1974) о будущем научных исследований. В 1927–1931 гг., работая в Массачусетском технологическом институте, Буш уже внес свою лепту в историю компьютеров, разработав один из самых удачных аналоговых компьютеров — дифференциальный анализатор. Во время написания статьи для «Life» Буш руководил Управлением научных исследований и разработок США, которое в годы войны отвечало за координацию различных научных исследований, включая «Манхэттенский проект».

Статья Буша «Как мы предполагаем» представляла собой сжатый вариант публикации, появившейся за два месяца до этого в «The Atlantic Monthly». В ней описывались некие гипотетические будущие изобретения, призванные облегчить жизнь ученого и исследователя, которому приходится иметь дело со все возрастающим объемом специализированных изданий. Буш видел выход в использовании микропленок. Он предложил использовать для хранения книг, статей, звуков и изображений воображаемое устройство «Memex». Между отдельными элементами хранилища предполагалось создание

тематических связей, основанных на ассоциациях, которые обычно рождаются в человеческом мышлении. По мнению Буша, разработкой этих связей должны заниматься специально подготовленные профессионалы.

В XX в. статьи о технологических чудесах будущего появлялись довольно часто. Но статья Буша выпадает из общего ряда. Речь в ней идет не о диковинных бытовых устройствах или фантастических средствах передвижения. Буша интересовала *информация* и технологические способы ее хранения и обработки.

За шесть с половиной десятилетий, отделяющих нас от первых релейных вычислительных машин, компьютеры стали компактнее, быстрее и дешевле. Благодаря этому изменилась сама природа *работы с информацией*. Из сложного научного и инженерного прибора компьютер перешел в разряд бытовой техники.

Повышение мощности и быстродействия компьютеров необходимо, хотя бы частично, использовать для совершенствования самой важной части компьютерной системы — *пользовательского интерфейса*, т. е. точки, в которой компьютер соприкасается с человеком. Человек и компьютер — разные, и, увы, многие считают, что легче человека приспособить к компьютеру, чем наоборот.

Первые цифровые компьютеры не отличались интерактивностью. Некоторые из них нужно было программировать с помощью переключателей и кабелей, другие — с помощью перфокарт и перфолент. В 1950-е и 1960-е годы (да и в 1970-е) пользователь зачастую был отделен от процесса вычислений: он писал программу на бумаге и отдавал ее в вычислительный центр. Сотрудники ВЦ переносили ее на перфокарты, вводили в компьютер и запускали. Затем пользователь получал отпечатанные на бумаге результаты работы программы (при условии, конечно, что компиляция прошла успешно).

Первые интерактивные устройства для обмена информацией с компьютером напоминали пишущие машинки. К одному компьютеру их можно было подключить несколько. Пользователь набирал на клавиатуре команду, которая тут же отпечатывалась на рулоне бумаги и отправлялась в компьютер. Тот обрабатывал ее и печатал на том же рулоне результаты. Обмен информацией между терминалом и компьютером

осуществлялся исключительно в виде ASCII-кодов (или в другой аналогичной кодировке), т. е. поток данных целиком состоял из букв, цифр, знаков препинания и простых управляющих кодов, например, кодов возврата каретки и перевода строки.

Катодно-лучевые трубки, широко распространившиеся в 1970-е годы, позволяли осуществлять вывод информации с большей гибкостью, но создатели первых программ для небольших компьютеров пользовались этой гибкостью неохотно, вероятно, не желая отступать от общепринятой в то время логики представления информации. Во многих случаях мониторы оставались «стеклянными пишущими машинками»: экран заполнялся информацией построчно, а когда места на нем не оставалось, строки «прокручивались» вверх, по одной исчезая за верхней границей экрана. В таком режиме действуют все программы для CP/M и многие программы для MS-DOS. В манере пишущей машинки до сих пор работает UNIX.

Интересно, что в кодировке ASCII изначально предусмотрено средство для работы с катодно-лучевым экраном: символ 1Bh (Escape). В 1979 г. Американский институт по стандартизации опубликовал стандарт «Дополнительные управляющие символы для использования с ASCII». Вводился он с целью «удовлетворить ожидаемые потребности в управлении вводом-выводом информации на двумерных устройствах… включая интерактивные терминалы с катодно-лучевыми экранами и принтеры».

Вот здесь и нашлось применение коду 1Bh. Символ Escape в потоке информации означает, что несколько следующих за ним символов нужно не отображать, а интерпретировать как код действия. Например, последовательность:

```
1Bh 5Bh 32h 4Ah
```

т. е. Escape-код и символы *[2J* стирает содержимое экрана и перемещает курсор в его верхний левый угол. Понятно, что с пишущей машинкой такое проделать не удастся. Последовательность:

```
1Bh 5Bh 35h 3Bh 32h 39h 48h
```

т. е. Escape-код и символы *[5;29H*, передвигает курсор в 29-й столбец 5-й строки.

Монитор, изображение на котором формируется ASCII-кодами и Escape-последовательностями, поступающими с удаленного компьютера, вкупе с клавиатурой иногда называют *«немым» терминалом* (dumb terminal). Терминалы работают быстрее телетайпных аппаратов и обладают большей гибкостью, но для радикальных перемен в пользовательском интерфейсе их быстродействия все же маловато. Настоящая революция в этой области началась лишь в 1970-е — с появлением небольших компьютеров, у которых видеопамять была частью общего адресного пространства процессора.

Первым указанием на то, что небольшие компьютеры будут радикально отличаться от своих больших дорогостоящих «собратьев», стала, вероятно, программа VisiCalc (1979) для компьютера «Apple II», разработанная и написанная Дэном Бриклином (Dan Bricklin) (род. 1951) и Бобом Фрэнкстоном (Bob Frankston) (род. 1949). До тех пор для вычислений с наборами чисел использовались разлинованные листы бумаги. В VisiCalc двумерное изображение таблицы впервые оказалось не на бумаге, а на экране компьютера. Пользователь мог свободно перемещать курсор по таблице, вводя в ее ячейки числа и формулы, которые по его команде вычислялись.

Пожалуй, одно из наиболее удивительных качеств VisiCalc в том, что создание аналогичной программы на большом компьютере *принципиально невозможно*. У подобных программ часто возникает потребность в быстром обновлении содержимого экрана. Поэтому программа VisiCalc записывала отображаемые данные прямо в видеопамять дисплея «Apple II», входившую в адресное пространство процессора. Скорость обмена данными между большим компьютером и «немым» терминалом для оперативного обновления изображения слишком низка.

Чем быстрее компьютер реагирует на сигналы клавиатуры и обновляет содержимое экрана, тем шире возможности для организации его взаимодействия с пользователем. Большинство программ, написанных в первое десятилетие существования IBM PC (1980-е), записывали данные прямо в видеопамять. Поскольку аппаратным стандартам IBM следовали и другие производители компьютеров, авторы ПО могли спокойно обходить ОС и обмениваться данными прямо с оборудованием, не опасаясь, что на каких-то компьютерах их программы работать откажутся. Если бы на различных клонах PC обмен

данных с дисплеем был организован по-разному, писать для них программы было бы гораздо сложнее.

Первые приложения для IBM PC выводили на экран почти исключительно текст, без графики. Это позволяло создавать максимально быстрые приложения. В компьютере, подобном тому, что мы обсуждали в главе 21, для отображения на экране конкретного символа программа просто записывает в видеопамять его ASCII-код. В графическом режиме для вывода на экран изображения того же символа в видеопамять придется передать не менее 8 байт.

И все же именно переход из текстового в графический режим стал чрезвычайно важным шагом в эволюции компьютеров. Поначалу разработка аппаратного и программного обеспечения для работы не просто с цифрами и буквами, а с изображениями шла очень медленно. Еще в 1945 г. Джон фон Нейман обсуждал возможность создания дисплея, работающего по принципу осциллографа, на котором можно было бы отображать графику. Но в жизнь эти идеи начали воплощаться лишь в начале 1950-х, когда Массачусетский технологический институт при участии IBM организовал Лабораторию им. Линкольна, которая разрабатывала компьютеры для системы ПВО ВВС США. В систему SAGE (Semi-Automatic Ground Environment, полуавтоматическая наземная система) входили графические дисплеи, помогавшие операторам анализировать большие объемы информации.

Действие современных мониторов для персональных компьютеров основано на иных принципах, чем действие первых графических систем типа SAGE. В наши дни в компьютерах применяются в основном растровые дисплеи, напоминающие обычные телевизионные экраны. Изображение состоит из набора горизонтальных строк, прорисовываемых электронным лучом. Наглядно экран можно представить в виде двумерного массива точек — пикселов. В компьютере для хранения изображения отводится специальная область памяти, в которой каждому пикселу на экране соответствует один или несколько битов. Значения этих битов определяют яркость и цвет соответствующего пиксела.

Разрешение большинства современных мониторов составляет минимум 640 пикселов по горизонтали и 480 по вертикали. Полное число пикселов, таким образом, равно 307 200. Если

каждому пикселу в памяти сопоставить единственный бит, можно будет закодировать всего два цвета — черный и белый. При этом для содержимого экрана понадобится 307 200 бит или 38 400 байт.

Чтобы отображать на экране большее количество цветов, придется увеличить число битов, приходящихся на пиксел. Так, чтобы отображать одним пикселом 256 оттенков серого, понадобится уже целый байт. Его значение 00h может соответствовать черному, значение FFh — белому, а промежуточные значения — различным оттенкам серого.

Чтобы получить на экране электронно-лучевой трубки цвет, приходится использовать уже не одну, а три электронные пушки, по одной для каждого из основных цветов — красного, зеленого, синего (посмотрите на экран через увеличительное стекло и убедитесь, что так оно и есть). Сочетание красного и зеленого дает желтый цвет, красного и синего — малиновый, зеленого и синего — голубой, сочетание всех трех основных цветов — белый.

В простейшем адаптере цветного дисплея на каждый пиксел должно приходиться по 3 бита — по одному на каждый основной цвет. Таблица кодирования цветов может выглядеть так:

Биты	Цвет
000	Черный
001	Синий
010	Зеленый
011	Голубой
100	Красный
101	Малиновый
110	Желтый
111	Белый

Конечно, ни для чего, кроме мультиков, такая палитра не годна. Чтобы получить на экране реалистичное изображение, надо уметь отображать на нем *оттенки* основных цветов. Если вы готовы пожертвовать по 2 байта на пиксел, каждому основному цвету будет сопоставлено по 5 бит (1 бит останется не у дел). Это позволит кодировать 32 градации яркости каждого основного цвета или всего 32 768 различных цветов. Такую систему кодирования часто называют палитрой High Color.

Следующий шаг к реалистичности — кодировать каждый пиксел 3 байтами, по байту на основной цвет. В такой схеме кодируются уже не 32, а 256 градаций яркости красного, зеленого и синего, или всего 16 777 216 различных цветов (палитра True Color). При разрешении дисплея 640 × 480 для 3-байтового представления цветов понадобится видеопамять объемом 921 600 байт, т. е. почти мегабайт.

Число битов на пиксел называют иногда *цветовой глубиной* (color depth). Количество доступных цветов связано с цветовой глубиной соотношением:

$$\text{Число цветов} = 2^{\text{Число битов на пиксел}}$$

Объем памяти на видеоплате не бесконечен, что накладывает ограничение на доступные комбинации разрешения и цветовой глубины. Если на плате адаптера установлена видеопамять объемом 1 Мб, при разрешении 640 × 480 можно будет использовать палитру True Color. Но если вы решите установить разрешение 800 × 600, выделить каждому пикселу по 3 байта уже не удастся; придется ограничить свои амбиции палитрой High Color.

Теперь растровые дисплеи кажутся нам вполне естественными, но в начале компьютерной эры они не находили широкого распространения именно по причине чрезмерных требований к памяти. Дисплеи системы SAGE были *векторными* (vector), более напоминая не телевизор, а осциллограф. Сигнал от компьютера произвольно перемещал электронную пушку, и луч электронов рисовал на экране прямые и кривые линии. Нарисованная линия некоторое время сохранялась на экране, благодаря чему на нем можно было создавать простые изображения.

На компьютерах SAGE также использовались *световые карандаши* (light pens), позволявшие оператору рисовать прямо на экране. Принцип работы карандаша нелегко разгадать даже технически подготовленному человеку, а фокус в том, что световой карандаш не *излучает* свет, он его *детектирует*. Электронная схема, управляющая движением пушки, попутно определяет, попал ли свет, излученный под воздействием электронного пучка, на световой карандаш, таким образом вычисляя его экранные координаты.

Одним из первых пришествие новой эры интерактивных вычислений предугадал Айвен Сазерленд (Ivan Sutherland)

(род. 1938). В 1963 г. он продемонстрировал революционную графическую программу Sketchpad для компьютеров SAGE. Она сохраняла описания изображений в памяти компьютера и могла при необходимости выводить их на экран. Световым карандашом можно было рисовать на экране новые изображения и корректировать уже имеющиеся, сохраняя внесенные изменения в памяти.

Другим провидцем эры интерактивных вычислений был Дуглас Энджелбарт (Douglas Engelbart) (род. 1925), с 1950 г. разрабатывавший компьютерные интерфейсы. В середине 1960-х Энджелбарт предложил использовать в качестве устройств ввода небольшую клавиатуру, специально приспособленную для набора команд (она так и не стала популярной), и устройство с колесиками и кнопкой, которой он назвал *мышью* (mouse). Мышь теперь используется для работы с объектами на экране практически повсеместно.

Многие сторонники графического компьютерного интерфейса работали в фирме Xerox. В 1970 г. фирма организовала в Пало-Альто (штат Калифорния) исследовательский центр PARC (Palo-Alto Research Center) с целью создания продуктов, которые позволили бы ей выйти на компьютерный рынок.

Первым большим проектом PARC был компьютер «Alto» (1972–1973). По тем временам это было впечатляющее устройство: 16-битовая обработка чисел, два диска емкостью по 3 Мб, 128 кб памяти (с возможностью расширения до 512 кб) и мышь с тремя кнопками! Об однокристальных микропроцессорах в ту пору еще не слышали, поэтому процессор «Alto» собирался из 200 отдельных микросхем.

В конструкцию «Alto» входило несколько нетривиальных устройств, в том числе, растровый дисплей. По размеру экран почти не отличался от стандартного листа бумаги — 8 дюймов в ширину и 10 дюймов в высоту. Дисплей работал в графическом режиме с разрешением 606 пикселов по горизонтали и 808 пикселов по вертикали (всего 489 648 пикселов). Каждому пикселу соответствовал 1 бит, т. е. доступны были лишь два цвета — черный и белый. Видеопамять емкостью 64 кб входила в адресное пространство процессора.

Записывая информацию в видеопамять, программа формировала на экране изображение, в частности, оформленный различными шрифтами текст. Передвигая мышь по столу,

пользователь перемещал по экрану указатель и работал с экранными объектами. Дисплей «Alto» уже совершенно не напоминал телетайпный аппарат с одномерной последовательностью команд и ответов на них. Он окончательно превратился в емкий двумерный массив информации.

В конце 1970-х у программ для компьютера «Alto» появилось несколько очень интересных особенностей. На экране одновременно могли отображать информацию несколько программ; у каждой из них для этого имелось свое окно. Появились и первые графические элементы пользовательского интерфейса — кнопки, меню и маленькие значки-«иконки» (icons), которые приводились в действие мышью. То были первые признаки того, что компьютеры перестают быть достоянием профессионалов и переходят в мир обычных людей, а область их применения выходит за рамки простого «пережевывания» чисел.

Разработки центра PARC для компьютера «Alto» положили начало *графическому интерфейсу пользователя* (Graphic User Interface, GUI). Правда, на рынок компьютеры «Alto» так и не поступили (стоил бы такой компьютер свыше 30 000 долларов), и прошло целое десятилетие, прежде чем заложенные в них идеи получили дальнейшее развитие.

В 1979 г. центр PARC посетил Стив Джобс и его коллеги из «Apple Computer». Увиденное произвело на них большое впечатление, но собственный компьютер с графическим интерфейсом они создали лишь 3 года спустя, в январе 1983 г. Это была печально известная машина «Apple Lisa». Годом позже появился куда более удачный «Macintosh».

Первые «Macintosh» комплектовались процессором Motorola 68000, ПЗУ емкостью 64 кб, оперативной памятью 128 кб, 3,5-дюмовым дисководом для дискет емкостью 400 кб, клавиатурой, мышью и монитором с разрешением 512 × 342 и диагональю 9 дюймов. Каждому пикселу соответствовал 1 бит; объем видеопамяти составлял 22 кб.

Оборудование первого «Macintosh» было элегантным, но не экстравагантным. От других компьютеров, продававшихся в 1984 г., его существенно отличала операционная система, в то время называвшаяся *системным программным обеспечением*, а позже прославившаяся под именем *Mac OS*.

Текстовые однопользовательские ОС наподобие CP/M или MS-DOS очень компактны и развитым интерфейсом прикладного программирования (API) не обладают. В главе 22 я говорил, что от этих ОС в основном требовалось дать другим программам доступ к файловой системе. Графическая ОС, например Mac OS, занимает гораздо больше места и наделена сотнями функций API. Каждая из них обозначается именем, в котором коротко зашифровано ее назначение.

В текстовой ОС, подобной MS-DOS, достаточно пары функций API, которые позволяли бы приложениям построчно печатать на экране текст. В графической ОС приложениям требуется возможность выводить на экран *изображения*. Теоретически для этого достаточно единственной функции API, которая задавала бы цвет и яркость пиксела с определенными координатами. Однако на практике поточечное формирование изображения оказывается слишком медленным.

Программировать для ОС гораздо легче, если она предоставляет в распоряжение разработчика полную систему графического программирования, т. е. набор функций API для отображения текста и рисования прямых линий, прямоугольников и эллипсов (в том числе окружностей). Удобно, если можно линии делать не только сплошными, но и пунктирными или штриховыми, для замкнутых фигур задавать заливку, для текста выбирать шрифты, размеры и начертания (полужирное, подчеркнутое и пр.). Преобразованием этих объектов в набор разноцветных точек на экране занимается не программист, а система графического программирования.

Программы, работающие под управлением графической ОС, обращаются к одним и тем же функциям API для рисования изображений на экране и их печати на принтере. Благодаря этому документ текстового процессора на экране выглядит практически так же, как и на печати. Такая технология называется WYSIWYG (What You See Is What You Get, что видишь, то и получаешь).

Привлекательность графических ОС отчасти обусловлена тем, что различные приложения в целом организованы одинаково, и потому опыт, приобретенный при работе с одним приложением, применим и во всех остальных. Для практической реализации такого единообразия в набор API включены функции для отображения элементов пользовательского ин-

терфейса, например, кнопок и меню. Обычно считают, что стандартизированный графический интерфейс облегчает жизнь пользователю, но на самом деле он не менее удобен и для программиста. При создании нового приложения автору не приходится изобретать велосипед.

Еще до появления «Macintosh» некоторые компании пытались создать графическую ОС для IBM-совместимых компьютеров. В каком-то смысле перед разработчиками «Apple» стояла более легкая задача, поскольку они проектировали оборудование и программы одновременно. Системной программе «Macintosh» достаточно было поддерживать один тип дисковода для гибких дисков, один дисплей и два принтера. При создании графической ОС для PC приходилось учитывать куда более широкое аппаратное разнообразие.

И это не единственная проблема. За несколько лет существования IBM PC многие люди привыкли к приложениям MS-DOS и не собирались от них отказываться. К графической ОС для PC предъявлялось непременное требование: под ней должны были работать программы для MS-DOS (на «Macintosh» программы для «Apple II» не работали главным образом из-за различий в микропроцессорах).

В 1985 г. появились сразу три графических оболочки для PC: GEM компании Digital Research (в свое время создавшей CP/M), VisiOn компании VisiCorp (она распространяла программу VisiCalc) и Windows 1.0 компании Microsoft. Скоро стало ясно, что победителем в «войне окон» станет последняя. Однако массовое внимание пользователей Windows привлекла лишь в мае 1990 г., после выхода версии 3.0. С тех пор популярность Windows возросла неимоверно, и в наши дни под управлением этой ОС работает 90% персональных компьютеров. Внешне системы Windows и Mac OS довольно похожи, но интерфейсы API у них совершенно разные.

Теоретически графическая ОС по сравнению с текстовой требует установки на компьютере лишь одного дополнительного устройства — дисплея с графическими возможностями. Даже жесткий диск необязателен: у первого «Macintosh» его вовсе не было, для работы Windows 1.0 он тоже был не нужен. Не требовалась в Windows 1.0 и мышь, хотя по общему мнению она существенно облегчала работу с системой.

И все же именно развитие компьютерного оборудования — повышение быстродействия процессоров, увеличение объема оперативной памяти и емкости жестких дисков — обусловило растущую популярность графических ОС. По мере того как ОС становится удобнее, растут и ее потребности. Современной графической ОС нужны пара сотен мегабайт на жестком диске и оперативная память не менее 32 Мб.

Приложения для графических ОС практически никогда не писались на ассемблере. В прежние годы программы для «Macintosh» разрабатывались на Паскале, а для Windows — на C. В 1972 г. сотрудники центра PARC приступили к разработке языка SmallTalk, опирающегося на концепцию *объектно-ориентированного программирования* (object-oriented programming), на которой основан новый подход к созданию графических программ.

В традиционных языках программирования имеется четкое различие между программой (т. е. операторами, начинающимися с какого-либо ключевого слова — Set, For, If и т. д.) и данными (т. е. константами и переменными). Корни этого различия, несомненно, лежат в архитектуре неймановского компьютера, в котором нет ничего, кроме машинного кода и данных, на которые этот код воздействует.

В объектно-ориентированном программировании код и данные сосуществуют в составе единого *объекта* (object). Конкретный способ хранения данных в объекте понятен лишь коду, связанному с этим объектом. Друг с другом объекты взаимодействуют, отправляя и принимая *сообщения* (messages), в которых содержатся команды или запросы на информацию.

Объектно-ориентированные языки особенно удобны при создании приложений для графических ОС. Благодаря им программист, работая с экранными объектами (окнами, кнопками), воспринимает их практически так же, как и пользователь. Возьмите в качестве примера объекта кнопку. Она характеризуется следующими данными: размерами, положением на экране, текстом. А связанный с кнопкой код отслеживает, была ли она «нажата» с помощью клавиатуры или мыши, и отправляет другим объектам сообщение об этом.

Несмотря на новый подход к программированию, все популярные объектно-ориентированные языки для небольших компьютеров являются расширенными вариантами традици-

онных «алголоподобных» языков, например, C или Паскаля. Объектно-ориентированный вариант языка C называется C++. Он разработан в основном Бьерном Страуструпом (Bjarne Stroustrup) (род. 1950) из Bell Telephone Laboratories. Поначалу C++ был реализован как транслятор, который преобразовывал программу на C++ в программу на C (весьма неуклюжую и совершенно нечитабельную). Затем программа на C компилировалась обычным способом.

Разумеется, возможностей у объектно-ориентированных языков ничуть не больше, чем у традиционных. Но решения программистских задач, подсказываемые объектно-ориентированными языками, зачастую технически более совершенны. При известном старании можно (хотя и не обязательно легко) написать даже объектно-ориентированную программу, которая будет компилироваться как под Mac OS, так и под Windows. В такой программе прямого обращения к API-функциям быть не должно. Программа обращается к объектам, а уже объекты вызывают API-функции. При компиляции программы под Mac OS и под Windows используются различные определения объектов.

В наши дни запускать компилятор из командной строки, как правило, уже не приходится. Большинство программистов перешли на *интегрированные среды разработки* — удобные программы, в которых объединены все инструменты написания и отладки программ. Широкое распространение получил также метод *визуального программирования* (visual programming), позволяющий разрабатывать окна приложения в интерактивном режиме, размещая в них кнопки и другие элементы с помощью мыши.

В главе 22 я рассказывал о текстовых файлах, которые содержат только коды ASCII и понятны человеку без использования дополнительных приспособлений. В текстовых ОС такие файлы — идеальное средство для обмена информацией между приложениями. У них есть одно большое преимущество: в текстовом файле легко найти нужную последовательность символов. Но как только у вас возникает желание отображать текст во множестве шрифтов, размеров и начертаний, вы понимаете, что возможностей текстового файла для этого недостаточно. Большинство современных текстовых процессоров хранят свои документы в двоичном формате. И уже совсем не годится текстовый формат для хранения изображений.

Точнее сказать, закодировать в текстовом файле такие атрибуты текста, как параметры шрифта или абзаца, можно. В формате RTF (Rich Text Format), разработанном фирмой Microsoft, для этого применяются фигурные скобки и обратная косая черта, за которой следует команда, описывающая форматирование текста.

В текстовом формате PostScript эта концепция доведена до крайности. Его разработал один из основателей компании Adobe Systems Джон Уорнок (John Warnock) (род. 1940). Он представляет собой настоящий универсальный язык графического программирования. Сейчас PostScript применяется в основном для печати текста и изображений на высококачественных принтерах.

Изображения украсили экраны ПК благодаря совершенствованию компьютерного оборудования и его удешевлению. Микропроцессоры заработали быстрее, память подешевела, экраны и листы бумаги, выползающие из принтеров, запестрели яркими красками, существенно возросло экранное и печатное разрешение. Все эти достижения сразу находили применение в компьютерной графике.

Компьютерные изображения, как и мониторы, бывают растровыми и векторными.

Векторные изображения создаются алгоритмически, в виде сочетания прямых и кривых линий и заполненных фигур. Их активно используют в *системах автоматизированного проектирования* (computer-aided drawing) для создания технических или архитектурных чертежей. Формат для хранения векторного изображения на диске компьютера называют *метафайлом* (metafile). Обычно в метафайл входят просто команды для рисования компонентов векторного изображения, записанные в двоичном виде.

Замкнутых и разомкнутых линий вполне достаточно для создания проекта, скажем, моста, но они абсолютно непригодны, если вам нужно показать, как этот мост выглядит «в жизни». Мост — объект из реального мира. Он слишком сложен, чтобы его можно было представить в виде комбинации простых геометрических фигур.

Вот здесь-то на помощь и приходят *растровые*, или *точечные* (bitmap) изображения. В точечном формате изображение представлено в виде прямоугольного массива пикселов, соот-

ветствующих пикселам устройства вывода. Подобно мониторам, точечные изображения характеризуются размерами по горизонтали и вертикали в пикселах и цветовой глубиной, зависящей от того, сколько битов соответствует 1 пикселу.

Хотя точечное изображение двумерно, файл, в котором оно записано, представляет собой всего лишь длинную последовательность битов. В ней, как правило, сначала закодирована первая строка пикселов, потом вторая и т. п.

Создавать точечные изображения можно как «вручную», с помощью специального графического приложения, так и программно. В точечный формат можно перевести и изображение, созданное более традиционным способом: рисунок или фотографию. Для переноса в компьютер образов из реального мира придумано несколько устройств, основу которых, как правило, составляет *прибор с зарядовой связью* (ПЗС, charge-coupled device, CCD) — полупроводниковый прибор, вырабатывающий электрический ток при облучении светом.

Старейшее устройство подобного рода — *сканер*. В нем, как и в копировальном аппарате, по сканируемому изображению (например, по фотографии) проходит ПЗС-линейка, составленная из отдельных ПЗС-элементов. В зависимости от интенсивности света, отраженного от изображения, каждый ПЗС-элемент вырабатывает определенный электрический заряд. Программа, обслуживающая сканер, переводит сигналы от отдельных элементов в пикселы точечного изображения и записывает его в файл.

В видеокамерах для записи изображения используются двумерные ПЗС-матрицы. Обычно запись осуществляется на магнитную ленту, но сигнал с ПЗС-матрицы можно направить и в *устройство для захвата кадра* (video frame grabber) — компьютерную плату, преобразующую аналоговый видеосигнал в последовательность битов. Источником видеосигнала может быть не только видеокамера, но и видеомагнитофон, проигрыватель лазерных видеодисков и даже обычный телевизор.

В недавнее время в разряд бытовых приборов попали цифровые видеокамеры. Выглядят они, как обычные видеокамеры, но сигнал с ПЗС-матрицы записывается в них не на ленту, а в электронную память, откуда его позже можно переписать на диск компьютера.

Зачастую в графической ОС для хранения точечных изображений используется свой специфический формат. На «Macintosh», например, применяется формат Paint. Это название происходит от имени графического редактора MacPaint, в котором формат был впервые применен (в настоящее время предпочтительнее формат PICT, в котором могут храниться как точечные, так и векторные изображения). Точечный формат для Windows называется BMP (такое расширение в этой ОС имеют файлы с точечными изображениями).

Точечные изображения зачастую весьма объемны, поэтому желательно придумать какой-то способ делать их компактнее. Изобретение подобных способов относится к компетенции раздела информатики, известного как *уплотнение данных*.

Рассмотрим в качестве примера изображение, в котором каждому пикселу соответствуют 3 бита. На нем запечатлено голубое небо и дом с лужайкой, т. е. существенные части изображения окрашены в голубой и зеленый цвета. Допустим, верхняя строка изображения содержит 72 пиксела голубого цвета, идущих друг за другом. Чтобы сделать файл компактнее, нужно записывать в него не все 72 пиксела, а лишь один, но с указанием повторить его 72 раза. Такой способ уплотнения называется кодированием повторяющихся последовательностей (Run-Length Encoding, RLE) и применяется в обычных факсимильных аппаратах для сжатия изображения перед его отправкой по телефонной линии. Поскольку факс-аппарат различает только черный и белый цвета, в отправляемом изображении обычно не бывает недостатка в длинных последовательностях одноцветных пикселов.

Для сжатия графических файлов уже больше десятилетия используется метод LZW, названный так по именам его создателей (Lempel, Ziv, Welch). Он, в частности, применяется в графическом формате GIF (Graphics Interchange Format, формат для обмена изображениями), разработанном в 1987 г. компанией CompuServe. Метод LZW в отличие от RLE способен распознавать не только идущие подряд одинаковые пикселы, но и более сложные закономерности в их расположении.

Методы RLE и LZW обеспечивают уплотнение *без потерь*. Это значит, что при восстановлении уплотненного файла он полностью возвращается в исходное состояние. Иными словами, уплотнение методом RLE или LZW *обратимо*. Легко до-

казать, что сжать без потерь можно не всякий файл. Иногда файл, «уплотненный» по обратимому методу, оказывается больше исходного!

В последние годы выросла популярность «*затратных*» методов сжатия. Они обратимостью уже не обладают, поскольку часть исходных данных в уплотненный файл не записывается. Разумеется, никто не посоветует вам таким способом сжимать документ текстового процессора или файл электронной таблицы, в которых важна каждая цифра или буква. Но к графическим файлам затратное сжатие вполне применимо, при условии, конечно, что выброшенные данные не ухудшают существенно качества изображения. Вот почему разработчикам затратных методов приходится опираться на результаты психологических исследований, в которых определяется, что кажется человеческому взгляду важным, а что — нет.

Самые популярные затратные методы сжатия точечных изображений известны под общим обозначением JPEG (Joint Photography Experts Group, Объединенная группа специалистов по фотографии). Правда, формат JPEG включает в себя несколько методов сжатия, из которых затратными являются не все.

Преобразовать метафайл в точечное изображение довольно просто. Концептуально видеопамять и точечное изображение организованы одинаково. Если программа «умеет» рисовать изображение на экране компьютера, она с тем же успехом может записать его в точечный файл.

Обратное преобразование выполнить гораздо сложнее, если вообще возможно. Частный случай такого преобразования — *оптическое распознавание символов* (Optical Character Recognition, OCR). Оно используется, когда нужно перевести печатные символы, содержащиеся в точечном изображении, в соответствующие коды ASCII. OCR-программа анализирует последовательности пикселов и пытается определить, изображение какого символа ей встретилось. Алгоритмически эта задача очень сложна, поэтому 100%-ую точность программы OCR обеспечивают редко. Еще хуже дело обстоит с переводом в коды ASCII рукописного текста.

Точечные и векторные изображения являют собой способ цифрового представления визуальной информации. В биты и байты можно преобразовать также и звуковую информацию.

Цифровой звук произвел фурор в 1983 г., когда на рынке появились первые *компакт-диски* (compact-disks, CD). Это событие положило начало самому успешному этапу развития бытовой электроники. Компакт-диск разработан фирмами Philips и Sony для записи 74 минут цифрового звука на одной стороне диска диаметром 12 см. Такая продолжительность записи выбрана для того, чтобы на одном диске можно было целиком хранить Девятую симфонию Бетховена.

Звук на компакт-диске кодируется с помощью *кодово-импульсной модуляции* (pulse code modulation, PCM). Несмотря на заковыристое название, концепция метода PCM проста.

Звук — это вибрация. Вибрируют голосовые связки человека, труба в оркестре, падающее дерево. Вибрация заставляет воздух двигаться. Он начинает периодически разрежаться и сжиматься с частотой несколько сотен или тысяч раз в секунду — по воздуху распространяется звуковая волна. Долетев до уха, она колеблет барабанные перепонки, благодаря чему мы слышим звук.

В фонографе Томаса Эдисона, изобретенном в 1877 г., звуковые волны продавливалась на поверхности цилиндра, покрытого оловянной фольгой. Записанный на цилиндре звук позже можно было воспроизвести. До появления компакт-дисков этот принцип записи оставался практически неизменным, хотя вместо цилиндров позже стали использовать диски, а вместо фольги — сначала воск, а потом пластмассу. Первые фонографы были полностью механическими, затем для усиления звука в них начали применять электрические схемы. Переменный резистор в микрофоне превращает звук в электрический сигнал, а электромагнит в динамике преобразует электричество обратно в звук.

Электрический ток, которым закодирован звук, не похож на те дискретные сигналы, о которых мы говорили на протяжении всей книги. Давление в звуковой волне меняется непрерывно, и потому непрерывно меняется также и напряжение тока. Этот электрический сигнал является аналоговым, и для его преобразования в двоичную форму необходимо специальное устройство, обычно изготавливаемое в виде микросхемы, — *аналого-цифровой преобразователь* (АЦП; analog-to-digital converter). Цифровые сигналы на выходе АЦП — обычно их 8, 12 или 16 — символизируют относительный уровень напря-

жения. 12-битовый АЦП, например, преобразует звуковой электрический сигнал в число от 000h до FFFh, различая 4096 уровней напряжения.

В случае кодово-импульсной модуляции аналоговый сигнал преобразуется в цифровые значения с постоянной скоростью. Эти значения записываются на компакт-диск в виде крохотных ямок на его поверхности и считываются лучом лазера, отражающимся от поверхности диска. При воспроизведении двоичные значения преобразуются обратно в аналоговый сигнал с помощью *цифро-аналогового преобразователя* (digital-to-analog converter). Подобное устройство используется и в цветных видеоадаптерах для преобразования битового кода пиксела в аналоговый сигнал, который подается на монитор.

Частота, с которой аналоговый звуковой сигнал преобразуется в цифровой, называется *частотой дискретизации* (sampling rate). В 1928 г. Гарри Найквист (Harry Nyquist) из Bell Telephone Laboratories доказал, что она должна минимум вдвое превосходить максимальную частоту звука, который предполагается записывать и воспроизводить. Обычно считается, что человеческое ухо способно воспринимать звук в диапазоне частот от 20 до 20 000 Гц. При записи компакт-дисков используется частота дискретизации 44 100 Гц, что несколько выше требуемой.

Количество битов для кодирования элемента записи определяет динамический диапазон компакт-диска, т. е. различие между самым тихим и самым громким звуком, который можно на него записать. Для измерения ширины этого диапазона введена единица *бел*, названная в честь Александра Белла. 1 бел соответствует 10-кратному увеличению интенсивности звука. Чаще применяется единица *децибел*, равная 0,1 бела. 1 децибел приблизительно равен минимальному изменению интенсивности звука, которое человек в состоянии различить.

16-разрядное кодирование звука позволяет охватить динамический диапазон 96 децибел, что приблизительно соответствует разнице между порогом слышимости и болевым порогом. При записи компакт-дисков один дискретный звуковой элемент кодируется 16 битами.

Итак, 1 секунда звуковой записи представлена на диске 44 100 элементами по 2 байта каждый. Поскольку мы уже привыкли к стереозвучанию, удваиваем это число, получив в итоге 176

400 байт на секунду или 10 584 000 байт на минуту записи (теперь вы, конечно, понимаете, почему цифровая запись не пользовалась популярностью до 1980-х). Для записи на компакт-диске 74 минут звука требуется 783 216 000 байт.

У цифрового звука по сравнению с аналоговым много хорошо известных преимуществ. В частности, при каждом копировании аналогового звука качество записи ухудшается. Цифровой же звук состоит из чисел, которые можно многократно копировать без потери информации. В прежних телефонных сетях качество сигнала ухудшалось при увеличении расстояния между абонентами. Теперь это не так. В телефонных системах используется в основном цифровой звук, и по качеству звонок на другой континент не отличается от звонка в соседний дом.

На компакт-диске можно хранить не только звук, но и данные. Компакт-диск, используемый исключительно для данных, называют иногда CD-ROM-диском. Емкость таких дисков обычно не превышает 660 Мб. Дисководы для их чтения стали стандартным компьютерным оборудованием, а сами диски — традиционным средством для распространения приложений и игр.

Лет 10 назад возможность записи и воспроизведения звука и видео с помощью компьютера назвали красивым словом *мультимедиа* (multimedia), но теперь эти функции настолько стандартны, что необходимость в специальном названии отпала. В комплект большинства компьютеров входит звуковая плата со всеми необходимыми устройствами для воспроизведения звука через колонки и записи с помощью микрофона. На диске записанный звук сохраняется в WAV-файлах (от «wave» — волна).

При воспроизведении звука, записанного с помощью компьютера, редко нужно качество компакт-диска, поэтому звукозаписывающие программы и на «Macintosh», и на Windows-компьютерах используют 8-разрядное кодирование звука и небольшие частоты дискретизации — 22 050, 11 025 и 8000 Гц. При минимальных запросах (без стерео) объем 1 секунды записи можно сократить до 8 000 байт, т. е. 480 000 байт на минуту.

Любители фантастических фильмов, бесспорно, знают, что компьютеры будущего общаются с людьми исключительно на человеческом языке. Что ж, если компьютер оборудован устройствами для записи и воспроизведения цифрового звука,

достаточно разработать соответствующую программу, чтобы заставить его говорить.

Есть два способа научить компьютер произносить узнаваемые слова и предложения. Можно записать фрагменты предложений, фразы, слова и числа, сказанные человеком, сохранить их в файлах, а затем комбинировать различными способами. Этот подход часто применяется в информационных системах, доступ к которым осуществляется по телефону. Он удобен, когда воспроизводить предполагается ограниченный набор словосочетаний и чисел.

В более общем случае для синтеза человеческой речи нужно как-то преобразовывать в цифровой звук текст в кодировке ASCII. Во многих языках написание слов не всегда согласуется с их произношением, поэтому для преобразования потребуется программа, определяющая произношение по словарю или с помощью сложного алгоритма. Можно, например, строить слова из отдельных звуков (фонем). Кроме того, такая программа должна будет учитывать и другие тонкости. Например, если предложение завершается вопросительным знаком, интонация в его конце должна изменяться.

Распознавание голоса, т. е. преобразование цифрового звука в коды ASCII, — задача куда более сложная. Из-за разнообразия диалектов иногда даже люди, говорящие на одном языке, не понимают друг друга. Программы для персональных компьютеров, воспринимающие человеческую речь, существуют, но им, как правило, нужна некоторая «тренировка», прежде чем они начинают понимать речь определенного человека. Но даже задача превращения звука в коды ASCII меркнет перед окончательной целью — научить компьютер *понимать* человеческую речь. Это уже проблема из области *искусственного интеллекта.*

Звуковые платы в современных компьютерах снабжаются миниатюрными электронными синтезаторами, которые способны имитировать звучание 175 музыкальных инструментов, включая 47 ударных. Они называются синтезаторами MIDI (Musical Instrument Digital Interface, цифровой интерфейс для музыкальных инструментов). Спецификация MIDI разработана в начале 1980-х консорциумом производителей электронных синтезаторов для их подключения друг к другу и к компьютерам.

В разных синтезаторах MIDI для генерации звука музыкальных инструментов используются разные методы. Одни позволяют получить реалистичное звучание, другие — не очень. Общее качество работы данного синтезатора к спецификации MIDI отношения не имеет. Она требует от синтезатора лишь верной реакции на короткие сообщения длиной от 1 до 3 байт. Обычно в этих сообщениях указывается, какую ноту и на каком инструменте нужно играть.

MIDI-файл представляет собой последовательность сообщений MIDI с информацией о времени исполнения. Как правило, в MIDI-файле полностью содержится некое музыкальное произведение, которое можно воспроизвести на MIDI-синтезаторе компьютера. MIDI-файл обычно гораздо короче WAV-файла с той же записью. В отношении размера WAV-файл можно сравнить с точечным файлом, а MIDI-файл — с векторным. Недостаток MIDI-записи в том, что она может прекрасно звучать на одном синтезаторе и отвратительно — на другом.

От цифрового звука логично перейти к цифровому видео. Кажущаяся подвижность кино и телевидения обеспечивается быстрой сменой неподвижных изображений, называемых *кадрами*. В кинопроекторе изображение меняется с частотой 24 кадра в секунду, в телевидении США — 30 кадров в секунду, в телевидении большинства других стран — 25 кадров в секунду.

Компьютерный видеофайл — это просто последовательность точечных изображений, сопровождаемых звуком. Если не прибегать к уплотнению, размеры видеофайлов будут просто огромными. Считайте сами: кадр размером 640 × 480 пикселов с 24-битовым кодированием цвета занимает 921 600 байт. При частоте смены изображений 30 кадров в секунду понадобится 27 648 000 байт для записи всего только одной секунды фильма! Минута записи займет 1 658 880 000 байт, а двухчасовой фильм — 199 065 600 000, т. е. почти 200 Гб. Вот почему видеоролики, предназначенные для воспроизведения на персональном компьютере, как правило, отличаются краткостью, небольшим форматом кадра и скверным разрешением.

Для сжатия видеозаписей применяется формат MPEG (Motion Pictures Expert Group, Группа специалистов по видеозаписям). Объем записи в нем сокращается за счет того факта, что значительная часть текущего кадра является копией предыдущего кадра.

Для формата MPEG разработано несколько стандартов. Например, в высококачественном телевидении HDTV и в дисках DVD (Digital Video Disks) применяется стандарт MPEG-2. По размеру диски DVD сходны с компакт-дисками, но в них информация записывается на обеих сторонах по два слоя на каждой стороне. Объем DVD-диска — около 16 Гб — превышает объем компакт-диска в 25 раз. Поскольку формат MPEG-2 обеспечивает сжатие приблизительно в 50 раз, двухчасовой фильм занимает всего 4 Гб, т. е. один слой на одной стороне. Вероятно, в ближайшем будущем для распространения ПО будут применяться именно DVD-ROM-диски.

Можно ли считать эти диски реализацией идей Ванневара Буша? Он, правда, писал о хранении информации на микропленках, но диски CD-ROM и DVD-ROM удобнее: на электронном носителе найти нужные сведения гораздо проще, чем, скажем, в обычной книге. С другой стороны, Буш предполагал, что человек с помощью системы «Memex» сможет работать с несколькими микропленками одновременно. На большинстве же компьютеров дисковод для работы с CD-ROM- или DVD-ROM-дисками только один. Чтобы эффективнее хранить и обмениваться информацией, приходится *соединять* компьютеры между собой. Поскольку почти все места обитания или работы человека связаны телефонными линиями, можно воспользоваться ими и для установления контакта между компьютерами.

Назначение телефонной системы — в передаче по проводам звуков, но не битов. Для передачи по телефонному проводу двоичную информацию нужно преобразовать в звук. Монотонная звуковая волна с постоянной частотой и постоянной амплитудой никакой информации в себе не несет. Но измените эту волну, точнее, промодулируйте ее так, чтобы один из ее параметров по вашему желанию осциллировал между двумя фиксированными состояниями, и вы сможете представлять с их помощью 0 и 1. Преобразует биты в звук *модем* (*мо*дулятор/*де*модулятор; modem). Модем — последовательное устройство, так как в нем биты одного байта передаются друг за другом, а не все сразу (параллельный интерфейс, позволяющий благодаря наличию 8 проводов передавать байт целиком, часто применяется для подсоединения к компьютеру принтера).

В первых модемах для передачи информации использовался сдвиг по частоте: 0 соответствовал одной частоте сигнала, 1

— другой. Поскольку к каждому байту добавляются еще и биты начала и окончания передачи (старт-бит и стоп-бит), реально для передачи байта требуется отправить по телефонной линии 10 бит. Первые модемы работали на скоростях 300 бит (30 байт) в секунду; в современных модемах с помощью всяческих ухищрений удается достигать скоростей в 100 раз выше.

Первые энтузиасты компьютерных коммуникаций организовывали с помощью персонального компьютера и модема *электронные доски объявлений* (Bulletin Board System, BBS), к которым другие компьютеры подключались по телефонным линиям. Эту концепцию использовали и крупные информационные службы, например, CompuServe. Как правило, обмен информацией в таких системах производился исключительно в форме ASCII-текста.

Ключевое отличие Интернета от первых массовых информационных систем — отсутствие выделенного центра. Действие Интернета основано на наборе протоколов для осуществления связи между компьютерами. Главный из них — протокол TCP/IP (Transmission Control Protocol/Internet Protocol). В сетях TCP/IP передаваемые блоки данных разбиваются на небольшие *пакеты* (packets), которые посылаются по коммуникационной (в случае домашних компьютеров, как правило, телефонной) линии независимо друг от друга, а на ее приемном конце вновь собираются в единое целое.

Важную часть Интернета составляет его графическая подсистема — World Wide Web (WWW), использующая протокол *HTTP* (Hypertext Transfer Protocol, протокол передачи гипертекста). Информация, представляемая на Web-страницах, оформляется с помощью *языка разметки гипертекста* (Hypertext Markup Language, HTML). *Гипертекстом* называется текст, связанный с дополнительной информацией на ту же тему посредством гиперссылок (нечто подобное и предлагал Ванневар Буш). Гиперссылки в HTML-файле указывают на другие Web-страницы, к которым можно из него перейти.

Внешне формат HTML подобен RTF, о котором я уже говорил: он содержит только ASCII-текст и команды, определяющие его оформление. В HTML-файл можно также вставлять изображения в форматах GIF, JPEG и PNG (Portable Network Graphics). Несмотря на название, HTML в действительности

не является языком, подобным тем, о которых мы говорили в главах 19 и 24: это всего лишь способ форматирования текста.

Иногда необходимо, чтобы при просмотре Web-страницы запускалась определенная программа. Она может работать как на компьютере-сервере (на котором хранится Web-страница), так и на компьютере-клиенте (на котором вы ее просматриваете) На сервере программная обработка страницы (например, электронного бланка, который вы заполнили) выполняется обычно с помощью сценариев CGI (Common Gateway Interface, интерфейс общего шлюза). Если программу предполагается запускать на компьютере-клиенте, она пишется на простом языке программирования *JavaScript* и включается в состав HTML-файла. Web-браузер интерпретирует операторы JavaScript и выполняет необходимые действия.

Почему программа передается на компьютер-клиент в виде текста, а не исполняемого файла? Во-первых, это связано с тем, что программист заранее не знает, на каком компьютере она будет выполняться. Если это «Macintosh», в программе должны содержаться машинные коды процессора PowerPC и вызовы функций API для Mac OS. На PC-совместимом компьютере в ней должны использоваться машинные коды процессора Pentium и обращения к функциям API для Windows. А ведь есть и другие компьютеры, и другие графические ОС. Кроме того, вряд ли стоит допускать на свой компьютер все исполняемые файлы без разбора. Некоторые из них могут попасть к вам из ненадежного источника и нанести компьютеру какой-либо вред.

Чтобы как-то справиться с этой проблемой, компания Sun Microsystems разработала язык программирования Java. Не путайте его с JavaScript — Java представляет собой настоящий объектно-ориентированный язык, напоминающий C++. Если помните, в главе 24 я объяснял разницу между компилируемыми и интерпретируемыми языками. Java занимает промежуточное положение. Программу на Java нужно компилировать, но в результате получается не машинный код, а *байтовые коды Java* (Java Byte codes). По структуре они напоминают машинные коды, но не реального, а воображаемого компьютера — *виртуальной машины Java* (Java Virtual Machine, JVM). Действие этого компьютера программным способом моделирует тот компьютер, на котором программу нужно запустить,

используя для этого средства установленной на нем графической ОС. Таким образом, Java-программу можно запускать на любом компьютере.

Большая часть этой книги посвящена передаче информации по металлическим проводам с помощью электричества, но гораздо более надежной и быстрой оказывается передача данных с помощью света по оптоволоконному кабелю, изготовленному из стекла или полимерных материалов. Скорость передачи данных по такому кабелю может достигать миллиарда бит в секунду.

По-видимому, в будущем большую часть информации в наши дома и служебные кабинеты будут доставлять не электроны, а фотоны. Они словно возвращают нас в далекие дни детства, когда вспышки света помогали нам обмениваться полуночной мудростью с лучшим другом, жившим на противоположной стороне улицы.

Благодарности

Книга *Код* была задумана в 1987 г. Она стучалась мне в голову в течение десятилетия и лишь с января 1996 г. по июль 1999 г. наконец оформилась в виде файла Microsoft Word. Я чрезвычайно признателен:

- Шерил Кантер (Sheryl Canter), Иену Истлунду (Jan Eastlund), Питеру Голдеману (Peter Goldeman), Линн Магалска (Lynn Magalska) и Дейдре Синнотт (Deirdre Sinnott) — читателям первых набросков за их комментарии, замечания и предложения;
- моему агенту Клодетт Мур (Claudette Moore) из Moore Literary Agency и всем сотрудникам Microsoft Press, которые помогли воплотить замысел *Кода* в жизнь;
- моей маме;
- Киске, делившей со мной кров с 1982 по май 1999 г., — вдохновительнице всех кошачьих примеров, которые встречаются в книге;
- Web-узлам, подобным Bibliofind (*www.bibliofind.com*) и Advanced Book Exchange (*www.abebooks.com*), предоставляющим удобный доступ к старым книгам, а также сотрудникам отдела научных, технических и деловых книг Нью-Йоркской публичной библиотеки (*www.nypl.org*);
- моим друзьям, без которых книга не была бы написана;
- и еще раз Дейдре — идеальному читателю и не только.

Библиография

Аннотированная библиография к этой книге размещена на Web-узле *www.charlespetzold.com/code*.

Предметный указатель

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Об авторе

Чарльз Петцольд (Charles Petzold) пишет программы более 20 лет. И вот уже 15 лет он пишет книги и статьи о программировании.

Его знакомство с вычислительной техникой состоялось в начале 1970-х, когда он собственноручно собрал компьютер с процессором Z-80 для управления музыкальным синтезатором. С 1985 г. он сотрудник журнала *PC Magazine*, с 2000 г. — журнала *MSDN Magazine*. С 1987 по 2000 г. он был редактором журнала *Microsoft Systems Journal*. Его статья, опубликованная в декабре 1986 г. во втором номере этого журнала, считается первой статьей о программировании для Windows. А в 1988 г. он посвятил этой теме целую книгу — *Programming Windows*, которую порой называют *Войной и миром* для этой операционной системы. К настоящему времени она выдержала уже пять изданий. В 1994 г. Чарльзу Петцольду была вручена премия «Пионер Windows», учрежденная Windows Magazine и Microsoft Corporation, «в знак признания его вклада в успех Microsoft Windows».

Помимо программирования, Чарльз Петцольд увлекается фотографией (но не автоматизированной цветной, а старомодной черно-белой), астрономией и эволюционной психологией. Он живет в Нью-Йорке.

Петцольд Чарльз

Код

Перевод с английского под общей редакцией **Д. З. Вибе**
Компьютерный дизайн и верстка **Д. В. Петухов**
Технический редактор **С. В. Дергачев**
Дизайнер обложки **Е. В. Козлова**

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

Главный редактор **А. И. Козлов**

Подготовлено к печати издательско-торговым домом «Русская Редакция»

РУССКАЯ РЕДАКЦИЯ

Лицензия ЛР № 066422 от 19.03.99 г.
Подписано в печать 21.06.2001 г. Тираж 4 000 экз.
Формат 84x108/32. Физ. п. л. 16

Отпечатано в ОАО «Типография "Новости"»
107005, Москва, ул. Ф. Энгельса, 46

Чарльз Петцольд

КОД

тайный язык информатики

Современный пользователь работает на компьютере механически. Нажал на пару клавиш, подвигал мышью — и из принтера появляется лист бумаги с напечатанными буквами. Еще пара манипуляций, и документ полетит по электронной почте. Но что стоит за этими действиями? На этот вопрос отвечает Чарльз Петцольд — патриарх информационных технологий, стоявший у истоков Windows, одной из самых популярных сегодня операционных систем. Эта книга для тех, кого не устраивает зазубривание последовательности действий непонятных ритуалов. В ней Вы прочитаете и о кодировании информации, и об электронной «начинке» компьютера, и о сути программирования. Книгу можно было бы назвать ***Основы информатики***, если бы простой и ясный, временами ироничный язык не отличал ее от простого учебника. Сам автор говорит, что написал ее, устав отвечать на вопрос: ***Как работают компьютеры?*** Но это не совсем так: книга скорее о том, ***почему они работают***.

ISBN 5-7502-0159-7

Web-узел издательства: **www.rusedit.ru**

РУССКАЯ РЕДАКЦИЯ