Код. Тайный язык информатики. Второе издание

Русский перевод второго издания книги Чарльза Петцольда о том, как из кодов, переключателей и реле возникают логические элементы, память, процессоры, операционные системы и программирование.

Больше крутых материалов в моём телеграм-канале @grogrigs

Глава 1. Лучшие друзья

Вам 10 лет. Ваш лучший друг живёт через дорогу. Окна ваших спален расположены прямо напротив друг друга. Каждый вечер, после того как родители объявляют, что пора ложиться спать, причём в обычный неприлично ранний час, вам всё ещё нужно обменяться мыслями, наблюдениями, секретами, сплетнями, шутками и мечтами. Вас нельзя за это винить. В конце концов, стремление к общению - одна из самых человеческих черт.

Пока в спальнях ещё горит свет, вы с другом можете махать друг другу из окон и с помощью широких жестов и простейшего языка тела передать одну-две мысли. Но более содержательный обмен кажется затруднительным, а после того, как родители провозгласят: «Свет выключить!», приходится искать более незаметные решения.

Как же общаться? Если вам повезло уже в 10 лет иметь мобильный телефон, возможно, подойдёт тайный звонок или беззвучная переписка. Но что, если родители имеют обыкновение отбирать телефоны перед сном и даже отключать Wi-Fi? Спальня без электронной связи и впрямь становится очень одиноким местом.

Однако у вас с лучшим другом есть фонарики. Всем известно, что фонарики изобрели для того, чтобы дети могли читать книги под одеялом; кроме того, фонарики кажутся прекрасным средством общения после наступления темноты. Они, безусловно, работают достаточно тихо, а их свет направлен в одну сторону и, вероятно, не просочится под дверь спальни, чтобы насторожить ваших подозрительных родителей.

Можно ли научить фонарики говорить? Попробовать определённо стоит. Ещё в первом классе вы научились писать на бумаге буквы и слова, поэтому кажется вполне разумным перенести это умение на фонарик. Нужно лишь встать у окна и рисовать буквы светом. Чтобы изобразить O, вы включаете фонарик, описываете им в воздухе круг и выключаете. Чтобы изобразить I, проводите вертикальную черту. Но, как вы быстро обнаруживаете, этот способ совершенно не годится. Наблюдая, как фонарик вашего друга выписывает в воздухе дуги и линии, вы понимаете, что мысленно соединять отдельные штрихи слишком трудно. Этим завиткам и росчеркам света попросту недостаёт точности.

Возможно, вы когда-то видели фильм, в котором двое моряков подавали друг другу сигналы через море мигающими огнями. В другом фильме шпион покачивал зеркалом, отражая солнечный свет в комнату, где держали в плену другого шпиона. Быть может, вот оно, решение. И тогда вы сначала придумываете простой способ: каждой букве алфавита соответствует определённое число вспышек фонарика. A - одна вспышка, B - две, C - три и так далее, вплоть до 26 вспышек для Z. Слово BAD передаётся двумя вспышками, одной вспышкой и четырьмя вспышками, а между буквами делаются короткие паузы, чтобы не принять все семь вспышек за G. Между словами пауза будет немного длиннее.

Всё выглядит многообещающе. Хорошая новость состоит в том, что теперь фонариком не нужно размахивать в воздухе: достаточно направить его в нужную сторону и щёлкать выключателем. Плохая новость выясняется, когда для одного из первых сообщений («How are you?» - «Как дела?») требуется в общей сложности целых 131 вспышка! К тому же вы забыли о знаках препинания, поэтому не знаете, сколько вспышек должно соответствовать вопросительному знаку.

Но вы близки к цели. Кто-нибудь наверняка уже сталкивался с этой задачей, думаете вы, и оказываетесь совершенно правы. Сходив в библиотеку или выполнив поиск в интернете, вы узнаёте о замечательном изобретении под названием азбука Морзе. Это именно то, что вы искали, хотя теперь вам придётся заново учиться «писать» все буквы алфавита.

Разница вот в чём. В придуманной вами системе каждой букве алфавита соответствует определённое число вспышек: от одной для A до 26 для Z. В азбуке Морзе используются вспышки двух видов - короткие и длинные. Разумеется, из-за этого азбука Морзе сложнее, но на деле она оказывается гораздо эффективнее. Для предложения «How are you?» теперь нужно всего 32 вспышки - несколько коротких и несколько длинных - вместо 131, причём в это число уже входит код вопросительного знака.

Объясняя устройство азбуки Морзе, люди говорят не о «коротких вспышках» и «длинных вспышках», а о «точках» и «тире», потому что так коды удобно изображать на печатной странице. В азбуке Морзе каждой букве алфавита соответствует короткая последовательность точек и тире, как показано в следующей таблице.

Буква	**Код**	**Буква**	**Код**	**Буква**	**Код**
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	`.-.`		

Хотя азбука Морзе не имеет совершенно никакого отношения к компьютерам, знакомство с природой кодов служит обязательной подготовкой к глубокому пониманию скрытых языков и внутреннего устройства компьютерного оборудования и программного обеспечения.

В этой книге слово *код* обычно означает систему передачи информации между людьми, между людьми и компьютерами либо внутри самих компьютеров.

Код позволяет общаться. Иногда коды бывают секретными, но большинство кодов вовсе не таковы. Более того, большинство кодов должно быть хорошо понятно, поскольку на них основано человеческое общение.

Звуки, которые мы произносим ртом, складывая их в слова, образуют код, понятный каждому, кто слышит наши голоса и понимает язык, на котором мы говорим. Мы называем этот код «устным словом» или «речью».

В сообществах глухих различные жестовые языки используют кисти и руки для движений и жестов, передающих отдельные буквы, целые слова и понятия. В Северной Америке наиболее распространены две системы: американский жестовый язык (American Sign Language, ASL), разработанный в начале XIX века в Американской школе для глухих, и квебекский жестовый язык (Langue des signes Québécoise, LSQ), представляющий собой разновидность французского жестового языка.

Для слов на бумаге и других носителях мы используем иной код, называемый «письменным словом» или «текстом». Текст можно написать или набрать вручную, а затем напечатать в газетах, журналах и книгах либо вывести в цифровом виде на самые разные устройства. Во многих языках между речью и текстом существует тесное соответствие. Например, в английском языке буквы и группы букв более или менее соответствуют произносимым звукам.

Для людей с нарушением зрения письменное слово можно заменить шрифтом Брайля, в котором система выпуклых точек соответствует буквам, группам букв и целым словам. (Более подробно шрифт Брайля рассматривается в главе 3.)

Когда устную речь необходимо очень быстро перевести в текст, полезны стенография и скоропись. В судах, а также при создании субтитров в реальном времени для телевизионных новостей или спортивных программ стенографисты пользуются стенотипной машиной с упрощённой клавиатурой и собственными кодами, соответствующими тексту.

Для общения друг с другом мы используем множество разных кодов, потому что одни из них в определённых обстоятельствах удобнее других. Код устной речи нельзя сохранить на бумаге, поэтому вместо него применяют код письменного слова. Беззвучно обмениваться информацией на расстоянии в темноте невозможно ни посредством речи, ни посредством бумаги. Следовательно, азбука Морзе становится удобной альтернативой. Код полезен, если он служит цели, которой не может послужить никакой другой код.

Как мы увидим далее, в компьютерах тоже применяются различные типы кодов: для хранения и передачи текста, чисел, звуков, музыки, изображений и фильмов, а также инструкций внутри самого компьютера. Компьютерам нелегко иметь дело с человеческими кодами, поскольку они не способны в точности воспроизводить то, как люди пользуются глазами, ушами, ртом и пальцами. Научить компьютеры говорить трудно, а убедить их понимать речь ещё труднее.

Однако достигнут значительный прогресс. Теперь компьютеры способны захватывать, хранить, обрабатывать и воспроизводить многие виды информации, используемой людьми для общения: зрительную - текст и изображения, слуховую - произносимые слова, звуки и музыку, а также сочетающую оба вида - анимацию и фильмы. Каждому виду такой информации требуются собственные коды.

Даже только что приведённая таблица азбуки Морзе сама по себе является своего рода кодом. Она показывает, что каждая буква представлена последовательностью точек и тире. Но передавать настоящие точки и тире мы не можем. При передаче азбуки Морзе фонариком точкам и тире соответствуют вспышки.

Чтобы передавать азбуку Морзе фонариком, для точки нужно быстро включить и выключить фонарик, а для тире оставить его включённым немного дольше. Например, чтобы передать А, вы быстро включаете и выключаете фонарик, а затем снова включаете и выключаете его, но уже не так быстро, после чего делаете паузу перед следующим знаком. По соглашению длительность тире должна примерно втрое превышать длительность точки. Человек на принимающей стороне видит короткую и длинную вспышки и понимает, что это А.

Паузы между точками и тире азбуки Морзе имеют решающее значение. Например, при передаче А между точкой и тире фонарик должен оставаться выключенным в течение времени, примерно равного длительности одной точки. Буквы одного слова разделяются более долгими паузами, приблизительно равными длительности одного тире. Вот, например, запись слова «hello» азбукой Морзе, показывающая паузы между буквами:

```
.... . .-.. .-.. ---
```

Слова разделяются периодом без света продолжительностью примерно в два тире. Вот код фразы «hi there»:

```
.... ..   -  ....  .  .-.  .
```

Промежутки времени, в течение которых фонарик остаётся включённым и выключенным, не имеют фиксированной продолжительности. Все они задаются относительно длительности точки, а та зависит от того, насколько быстро удаётся нажимать выключатель фонарика и насколько быстро отправитель способен вспомнить код конкретной буквы. Тире быстрого отправителя может иметь ту же продолжительность, что и точка медленного. Из-за этой небольшой трудности чтение сообщения азбукой Морзе могло бы оказаться непростым, но после одной-двух букв человек на принимающей стороне обычно понимает, что считать точкой, а что тире.

На первый взгляд определение азбуки Морзе - под *определением* я имею в виду соответствие различных последовательностей точек и тире буквам алфавита - кажется столь же случайным, как раскладка компьютерной клавиатуры. Однако при более внимательном рассмотрении выясняется, что это не совсем так. Более простые и короткие коды назначены буквам, чаще встречающимся в алфавите, например E и T. Игроки в Scrabble и поклонники телепередачи *Wheel of Fortune* могут заметить это сразу. У менее распространённых букв, таких как Q и Z, за которые в Scrabble дают 10 очков и которые редко встречаются в головоломках *Wheel of Fortune*, коды длиннее.

Почти каждый знает хотя бы немного азбуку Морзе. Три точки, три тире и три точки обозначают SOS - международный сигнал бедствия. SOS ничего не сокращает: это просто легко запоминающаяся последовательность азбуки Морзе. Во время Второй мировой войны Британская радиовещательная корпорация начинала некоторые радиопередачи первыми звуками Пятой симфонии Бетховена - БА-БА-БА-БА-А-АМ, - а ведь, сочиняя эту музыку, Бетховен не знал, что однажды она станет кодом Морзе для буквы V, первой буквы слова Victory - «победа».

Один из недостатков азбуки Морзе состоит в том, что она не различает прописные и строчные буквы. Однако помимо букв в ней представлены и цифры - для этого используются последовательности из пяти точек и тире:

Цифра	Код	Цифра	Код
1	`.----`	6	`-....`
2	`..---`	7	`--...`
3	`...--`	8	`---..`
4	`....-`	9	`----.`
5	`.....`	0	`-----`

По крайней мере, коды цифр устроены несколько упорядоченнее кодов букв. Для большинства знаков препинания применяются последовательности из пяти, шести или семи точек и тире:

Знак	Код	Знак	Код
`.`	`.-.-.-`	`'`	`.----.`
`,`	`--..--`	`(`	`-.--.`
`?`	`..--..`	`)`	`-.--.-`
`:`	`---...`	`=`	`-...-`
`;`	`-.-.-.`	`+`	`.-.-.`
`-`	`-....-`	`$`	`...-..-`
`/`	`-..-.`	`¶`	`.-.-..`
`"`	`.-..-.`	`_`	`..--.-`

Существуют также дополнительные коды для букв с диакритическими знаками в некоторых европейских языках и сокращённые последовательности специального назначения. Код SOS - одна из таких сокращённых последовательностей: его следует передавать непрерывно, выдерживая между тремя буквами паузу длиной всего в одну точку.

Вы обнаружите, что передавать азбуку Морзе вам с другом гораздо легче, если фонарик специально предназначен для этой цели. Помимо обычного ползункового выключателя у таких фонариков имеется кнопка, которую достаточно нажать и отпустить, чтобы включить и выключить свет. Немного потренировавшись, вы, вероятно, сумеете передавать и принимать по 5 или 10 слов в минуту. Это всё ещё намного медленнее речи, скорость которой составляет около 100 слов в минуту, но, несомненно, вполне достаточно.

Когда вы с лучшим другом наконец выучите азбуку Морзе - а только так можно научиться уверенно передавать и принимать сообщения, - её можно будет использовать и вслух вместо обычной речи. Для наибольшей скорости точку произносят как *ди* - или *дит*, если это последняя точка буквы, - а тире как *да*. Например, V произносится как *ди-ди-ди-да*. Подобно тому как азбука Морзе сводит письменную речь к точкам и тире, её устная форма сводит речь всего к двум гласным звукам.

Ключевое слово здесь - *два*. Два вида вспышек, два гласных звука, два разных чего угодно - при подходящих сочетаниях всё это способно передать информацию любого вида.

Глава 2. Коды и комбинации

Азбуку Морзе изобрёл примерно в 1837 году Сэмюэл Финли Бриз Морзе (1791-1872), с которым мы подробнее познакомимся далее в этой книге. Другие люди продолжили её совершенствовать, и прежде всего Альфред Вейл (1807-1859); в результате возникло несколько различных вариантов. Система, описанная в этой книге, носит более официальное название «международная азбука Морзе».

Изобретение азбуки Морзе неразрывно связано с изобретением телеграфа, который позднее в этой книге мы также рассмотрим подробнее. Подобно тому как азбука Морзе хорошо знакомит нас с природой кодов, в состав телеграфа входит оборудование, способное имитировать работу компьютера.

Большинство людей считает, что передавать азбуку Морзе легче, чем принимать. Даже если вы не помните её наизусть, можно просто воспользоваться уже знакомой по предыдущей главе таблицей, удобно расположенной в алфавитном порядке:

Буква	Код	Буква	Код	Буква	Код
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	.-.		

Принимать азбуку Морзе и переводить её обратно в слова значительно труднее и дольше, чем передавать, поскольку приходится двигаться в обратном направлении и выяснять, какая буква соответствует конкретной кодовой последовательности точек и тире. Если вы не выучили коды и принимаете последовательность тире-точка-тире-тире, вам придётся просмотреть таблицу буква за буквой, прежде чем вы наконец обнаружите, что это Y.

Проблема состоит в том, что у нас есть таблица, выполняющая такое преобразование:

```
Буква алфавита -> точки и тире азбуки Морзе
```

Но нет таблицы, позволяющей двигаться в обратном направлении:

```
Точки и тире азбуки Морзе -> буква алфавита
```

На первых этапах изучения азбуки Морзе такая таблица, безусловно, была бы удобна. Однако совершенно непонятно, как её построить. В точках и тире нет ничего, что можно было бы расположить в алфавитном порядке.

Поэтому забудем об алфавитном порядке. Возможно, коды лучше организовать, сгруппировав их по числу входящих в них точек и тире. Например, последовательность азбуки Морзе, содержащая всего одну точку или одно тире, может обозначать только две буквы - E и T:

Код	Буква
.	E
-	T

Комбинация ровно из двух точек или тире даёт ещё четыре буквы - I, A, N и M:

Код	Буква	Код	Буква
..	I	-.	N
.-	A	--	M

Узор из трёх точек или тире даёт ещё восемь букв:

Код	Буква	Код	Буква
...	S	-..	D
..-	U	-.-	K
.-.	R	--.	G
.--	W	---	O

Наконец, если мы хотим завершить это упражнение, не переходя к цифрам и знакам препинания, последовательности из четырёх точек и тире позволяют получить ещё 16 знаков:

Код	Буква	Код	Буква
....	H	-...	B
...-	V	-..-	X
..-.	F	-.-.	C
..--	Ü	-.--	Y
.-..	L	--..	Z
.-.-	Ä	--.-	Q
.--.	P	---.	Ö
.---	J	----	Ş

Вместе эти четыре таблицы содержат 2 плюс 4 плюс 8 плюс 16 кодов, то есть в общей сложности 30 букв - на 4 больше, чем требуется для 26 букв латинского алфавита. Поэтому можно заметить, что 4 кода в последней таблице обозначают буквы с диакритическими знаками: три с умлаутами и одну с седилью.

Эти четыре таблицы, несомненно, помогут, когда кто-нибудь будет передавать вам сообщение азбукой Морзе. Приняв код конкретной буквы, вы знаете, сколько в нём точек и тире, а значит, по крайней мере можете открыть нужную таблицу и отыскать его. Каждая таблица организована методично: она начинается в левом верхнем углу с кода из одних точек и заканчивается в правом нижнем углу кодом из одних тире.

Видите ли вы закономерность в *размерах* четырёх таблиц? В каждой следующей таблице вдвое больше кодов, чем в предыдущей. Это логично: в неё входят все коды предыдущей таблицы, после которых добавлена точка, и все коды предыдущей таблицы, после которых добавлено тире.

Эту любопытную закономерность можно обобщить так:

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

В каждой из четырёх таблиц вдвое больше кодов, чем в предыдущей. Поэтому, если в первой таблице 2 кода, во второй их 2 × 2, а в третьей - 2 × 2 × 2. Это можно показать и по-другому:

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

Как только мы начинаем иметь дело с числом, умножаемым на само себя, для обозначения степеней можно применять показатели степени. Например, 2 × 2 × 2 × 2 можно записать как 2^4 - 2 в четвёртой степени. Числа 2, 4, 8 и 16 являются степенями числа 2, поскольку их можно вычислить, умножая 2 само на себя. Сводную таблицу можно представить и так:

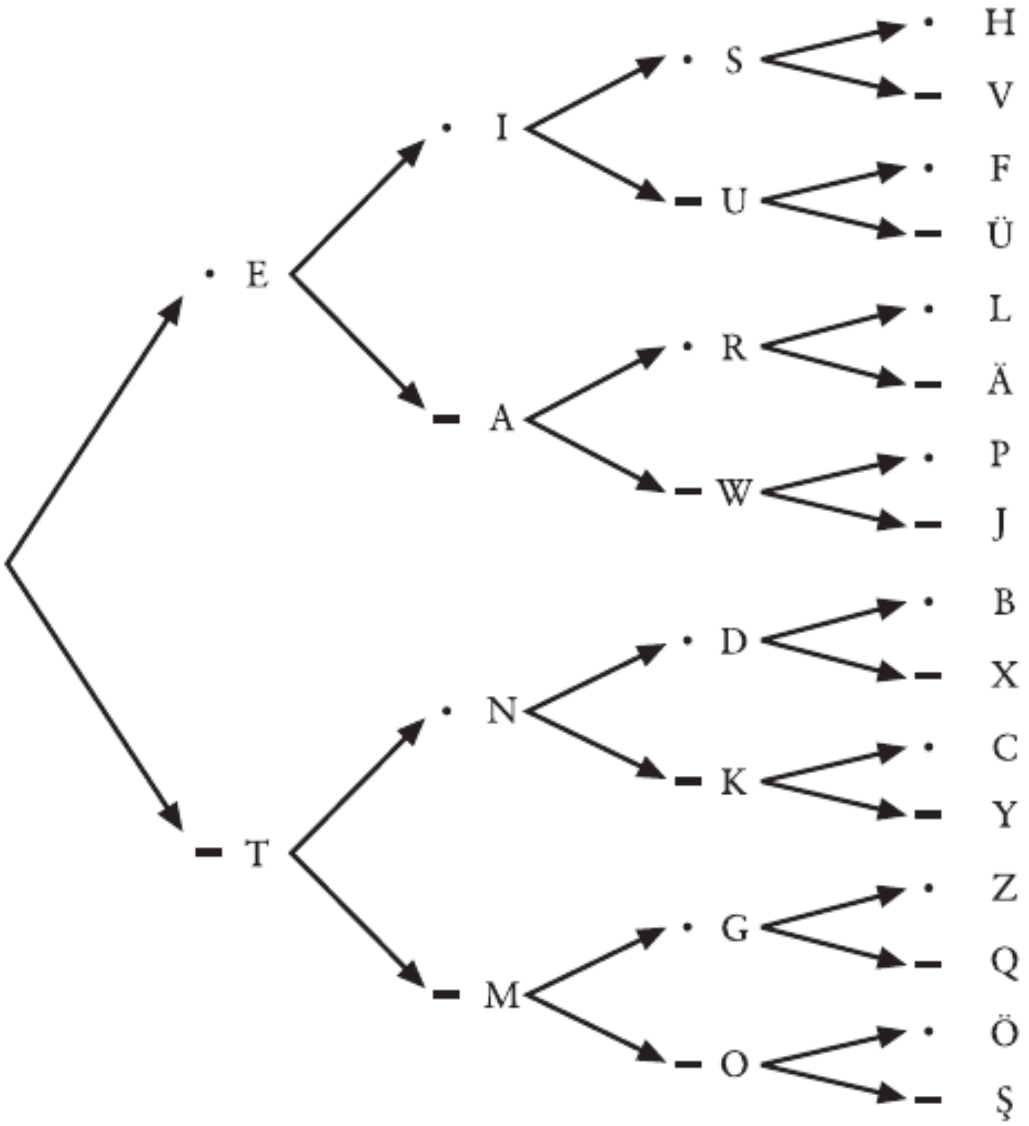
Число точек и тире	Число кодов
1	2^1
2	2^2
3	2^3
4	2^4

Эта таблица стала очень простой. Число кодов - это просто число 2, возведённое в степень, равную числу точек и тире:

```
число кодов = 2^(число точек и тире)
```

Степени двойки очень часто встречаются в кодах, и особенно в этой книге. Ещё один пример вы увидите в следующей главе.

Чтобы ещё больше облегчить декодирование азбуки Морзе, можно нарисовать нечто подобное приведённой ниже большой древовидной схеме.



Эта схема показывает буквы, получаемые из каждой конкретной последовательности точек и тире. Чтобы декодировать определённую последовательность, следуйте по стрелкам слева направо. Например, предположим, что вы хотите узнать, какая буква соответствует коду точка-тире-точка. Начните слева и выберите точку, затем продолжайте двигаться вправо по стрелкам и выберите тире, а потом ещё одну точку. Получится буква R, указанная рядом с третьей точкой.

Если задуматься, такая таблица, вероятно, была необходима уже при первоначальном определении азбуки Морзе. Во-первых, она позволяет избежать нелепой ошибки, при которой один код назначается двум разным буквам. Во-вторых, она гарантирует использование всех возможных кодов без ненужного удлинения последовательностей точек и тире.

Рискуя вывести таблицу за пределы печатной страницы, мы могли бы продолжить её кодами из пяти точек и тире. Последовательность ровно из пяти точек и тире даёт ещё 32 кода: 2 × 2 × 2 × 2 × 2, или 2^5. Обычно этого хватило бы для десяти цифр и 16 знаков препинания, определённых в азбуке Морзе, и цифры действительно кодируются пятью точками и тире. Но многие другие коды, состоящие из пяти точек и тире, обозначают не знаки препинания, а буквы с диакритическими знаками.

Чтобы включить все знаки препинания, систему необходимо расширить до шести точек и тире, что даёт ещё 64 кода: 2 × 2 × 2 × 2 × 2 × 2, или 2^6, - а всего 2 + 4 + 8 + 16 + 32 + 64, то есть 126 знаков. Для азбуки Морзе это явно избыточно, поэтому многие длинные коды остаются *неопределёнными*. В данном контексте это означает, что такой код ничего не обозначает. Если бы при приёме азбуки Морзе вы получили неопределённый код, то могли бы почти не сомневаться: кто-то допустил ошибку.

Поскольку нам хватило сообразительности вывести небольшую формулу

```
число кодов = 2^(число точек и тире),
```

можно и дальше вычислять число кодов, получаемых при использовании более длинных последовательностей:

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	$2^{10} = 1024$

К счастью, для определения числа возможных кодов нам не нужно выписывать их все. Достаточно снова и снова умножать 2 само на себя.

Азбуку Морзе называют *двоичной* - буквально это слово означает «по два», - потому что её коды состоят всего из двух компонентов: точки и тире. Это напоминает монету, которая может упасть только одной из двух сторон - орлом или решкой. При десяти подбрасываниях монеты возможны 1024 разные последовательности орлов и решек.

Комбинации двоичных объектов, таких как монеты, и двоичные коды, такие как азбука Морзе, всегда описываются степенями двойки. Два - очень важное число в этой книге.

Глава 3. Шрифт Брайля и двоичные коды

Сэмюэл Морзе был не первым человеком, которому удалось преобразовать буквы письменного языка в понятный код. И не его первого стали помнить больше по названию кода, чем по собственным заслугам. Эта честь принадлежит слепому французскому подростку, который родился примерно на 18 лет позже Морзе, но проявил себя в гораздо более юном возрасте. О его жизни известно немного, однако известные нам события складываются в захватывающую историю.

Луи Брайль родился в 1809 году в Кувре, Франция, всего в 25 милях к востоку от Парижа. Его отец был шорником. В три года - в возрасте, когда маленьким мальчикам не следует играть в мастерских своих отцов, - он случайно повредил глаз острым инструментом. В рану попала инфекция, которая затем распространилась на второй глаз, и мальчик полностью ослеп. В те времена большинство людей, которых постигла подобная судьба, были бы обречены на жизнь в невежестве и нищете, но ум и стремление юного Луи к знаниям вскоре заметили. Благодаря вмешательству деревенского священника и школьного учителя он сначала посещал местную школу вместе с другими детьми, а в 10 лет был отправлен в Королевский институт для слепой молодёжи в Париже.

Фото: ullstein bild Dtl/Getty Images.

Главным препятствием для образования слепых детей было отсутствие доступных материалов для чтения. Валентин Гаюи (1745-1822), основатель парижской школы, изобрёл систему тиснения на бумаге крупных букв округлого шрифта, которые можно было читать на ощупь. Но пользоваться этой системой было очень трудно, и таким способом изготовили лишь несколько книг.

Зрячий Гаюи оставался в плену привычной парадигмы. Для него А была А и только А, поэтому буква А непременно должна была выглядеть - или ощущаться - как А. (Если бы для общения ему дали фонарик, он, вероятно, попытался бы рисовать буквы в воздухе, как это делали мы, прежде чем убедились, что способ работает не слишком хорошо.) Гаюи, по всей видимости, не понимал, что для незрячих людей больше подошёл бы код, совершенно не похожий на рельефные обычные буквы.

Истоки альтернативного кода обнаружились в неожиданном месте. К 1815 году капитан французской армии Шарль Барбье разработал систему письма, позднее названную *écriture nocturne*, то есть «ночным письмом». В ней использовались узоры из выпуклых точек на плотной бумаге; система предназначалась для солдат, которым требовалось беззвучно передавать друг другу записки в темноте. Солдаты могли прокалывать точки с обратной стороны бумаги похожим на шило грифелем, а затем читать получившиеся выпуклости пальцами.

Луи Брайль познакомился с системой Барбье в 12 лет. Выпуклые точки понравились ему не только потому, что их было легко читать пальцами, но и потому, что ими было легко *писать*. Ученик, располагавший бумагой и грифелем, действительно мог делать на уроке заметки, а затем самостоятельно их читать. Брайль усердно совершенствовал систему и за три года, к 15 годам, создал собственную, основы которой используются по сей день. Много лет она была известна только внутри школы, но постепенно распространилась по всему миру. В 1835 году Луи Брайль заболел туберкулёзом, от которого в 1852 году умер вскоре после своего 43-го дня рождения.

Сегодня различные варианты системы Брайля соперничают с аудиокнигами, обеспечивая слепым людям доступ к письменному слову. Однако шрифт Брайля остаётся бесценной системой и единственным способом чтения для слепоглухих. В последние десятилетия широкая публика стала чаще встречать шрифт Брайля благодаря тому, что для повышения доступности его начали использовать в лифтах и банкоматах.

В этой главе я разберу код Брайля на составные части и покажу, как он работает. Вам не нужно на самом деле *учить* шрифт Брайля или что-либо запоминать. Единственная цель этого упражнения - ещё немного углубить наше понимание природы кодов.

В шрифте Брайля каждый знак обычной письменной речи - в частности, буква, цифра или знак препинания - кодируется одной или несколькими выпуклыми точками в ячейке размером два на три. Точки ячейки принято нумеровать от 1 до 6:

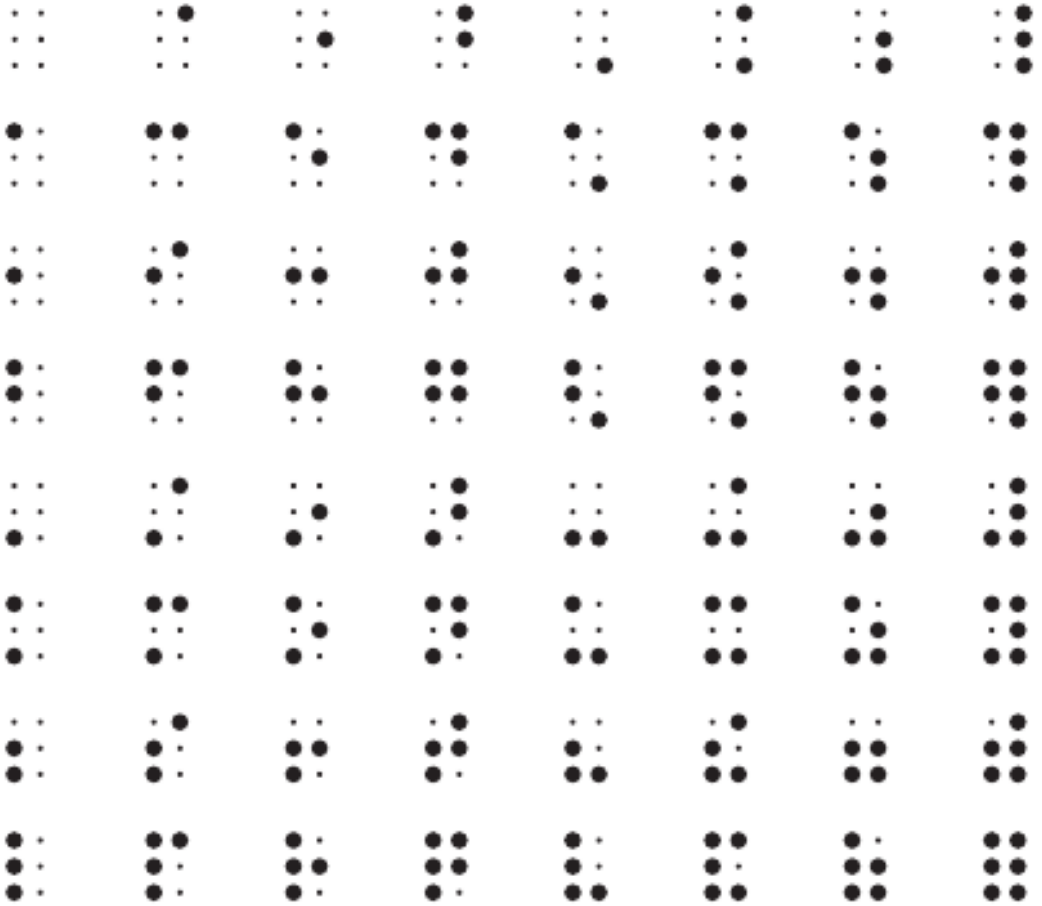
Для выдавливания точек Брайля на бумаге были разработаны специальные пишущие машинки, а в наши дни эту работу выполняют управляемые компьютерами принтеры для рельефной печати.

Поскольку выдавить шрифтом Брайля даже пару страниц этой книги было бы непомерно дорого, я использую распространённое обозначение, позволяющее показывать символы Брайля на печатной странице. В такой записи отображаются все шесть точек ячейки. Крупные точки обозначают места, где бумага приподнята, а мелкие - места, где она остаётся плоской. Например, в следующем символе Брайля

точки 1, 3 и 5 выпуклые, а точки 2, 4 и 6 - нет.

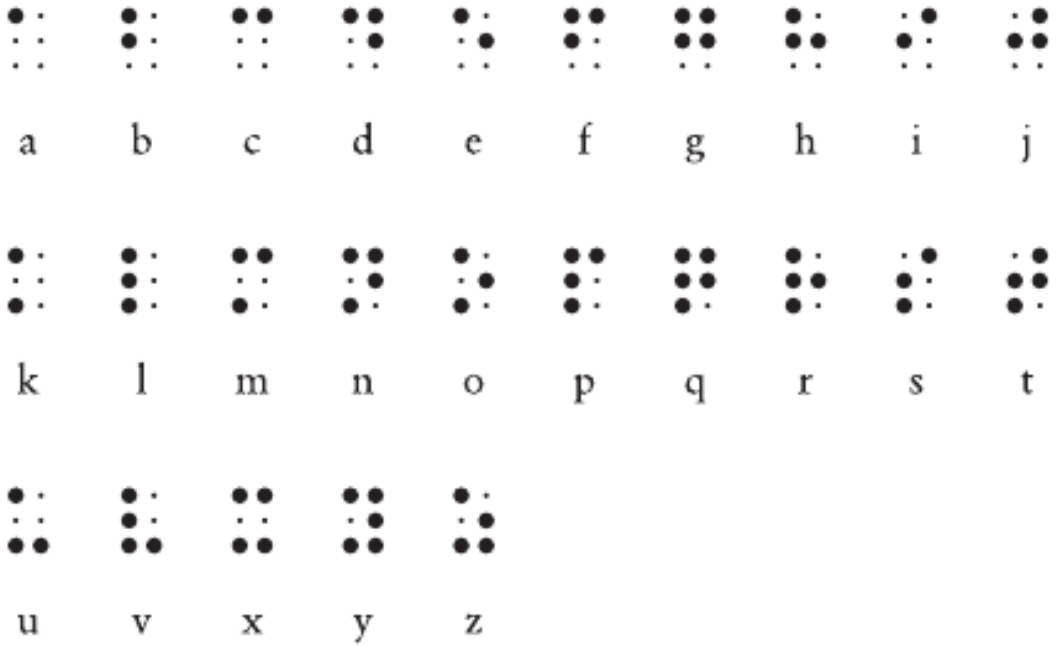
На этом этапе нам должно быть особенно интересно то, что точки *двоичны*. Каждая конкретная точка либо плоская, либо выпуклая. Следовательно, к шрифту Брайля можно применить знания об азбуке Морзе и двоичных комбинациях. Мы знаем, что точек шесть и каждая может быть либо плоской, либо выпуклой, поэтому общее число комбинаций шести плоских и выпуклых точек равно $2 \times 2 \times 2 \times 2 \times 2 \times 2$, то есть 2^6, или 64.

Таким образом, система Брайля способна представить 64 уникальных кода. Вот все 64:



Необязательно использовать в шрифте Брайля все 64 кода, но число 64 совершенно точно является верхним пределом, заданным шеститочечной структурой.

Чтобы начать разбирать код Брайля, рассмотрим основной алфавит строчных букв:



Например, фраза «you and me» в записи Брайля выглядит так:

Обратите внимание: ячейки отдельных букв внутри слова разделены небольшими промежутками, а между словами оставлен более широкий промежуток - по существу, ячейка без выпуклых точек.

Такова основа системы, придуманной Луи Брайлем, по крайней мере в части, относящейся к буквам латинского алфавита. Луи Брайль также разработал коды для распространённых во французском языке букв с диакритическими знаками. Обратите внимание, что кода для w здесь нет, поскольку эта буква не употребляется в классическом французском языке. (Не беспокойтесь, она ещё появится.) На данный момент задействованы лишь 25 из 64 возможных кодов.

При внимательном рассмотрении вы обнаружите закономерность в кодах Брайля для 25 строчных букв. В первом ряду, от *a* до *j*, используются только четыре верхние позиции ячейки - точки 1, 2, 4 и 5. Второй ряд, от *k* до *t*, повторяет первый, но в нём дополнительно поднята точка 3. Третий ряд, от *u* до *z*, устроен так же, но в нём подняты точки 3 и 6.

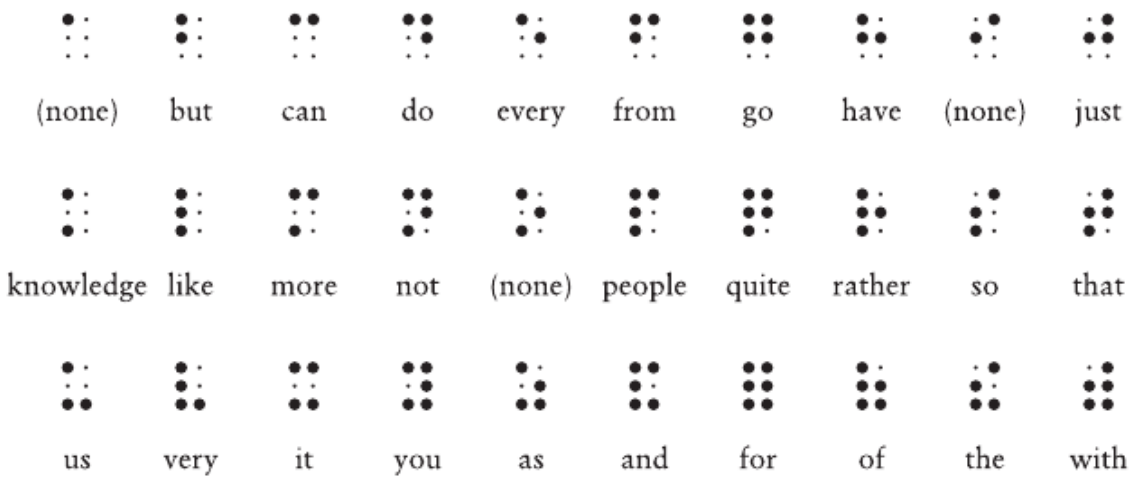
Изначально Луи Брайль рассчитывал, что символы его системы будут прокалывать вручную. Он понимал, что добиться высокой точности, скорее всего, не удастся, поэтому находчиво определил 25 строчных букв так, чтобы уменьшить неоднозначность. Например, среди 64 возможных кодов Брайля есть шесть кодов с одной выпуклой точкой. Но для строчной буквы используется только один из них - код буквы *a*. В четырёх из 64 кодов присутствуют две соседние точки по вертикали, однако и здесь применяется лишь один - для буквы *b*. В трёх кодах есть две соседние точки по горизонтали, но используется только один - для *c*.

В действительности Луи Брайль определил набор уникальных *форм*, которые можно немного сдвинуть на странице без изменения их значения. Буква *a* - это одна выпуклая точка, *b* - две соседние точки по вертикали, *c* - две соседние точки по горизонтали и так далее.

Коды нередко подвержены ошибкам. Ошибка, возникающая при записи кода, например когда ученик прокалывает точки Брайля в бумаге, называется ошибкой *кодирования*. Ошибка, допущенная при чтении кода, называется ошибкой *декодирования*. Кроме того, возможны ошибки *передачи* - например, если страница со шрифтом Брайля каким-либо образом повреждена.

Более сложные коды часто содержат встроенные средства исправления ошибок разных типов. В этом смысле первоначальная система Луи Брайля представляет собой сложную систему кодирования: благодаря избыточности она допускает небольшую неточность при прокалывании и чтении точек.

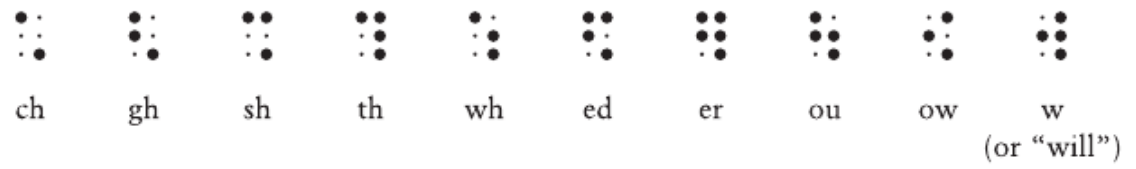
Со времён Луи Брайля его код расширяли в разных направлениях, в том числе создавали системы записи математики и музыки. В настоящее время в англоязычных печатных изданиях чаще всего применяется шрифт Брайля уровня 2 (Grade 2 Braille). В нём используется множество сокращений, позволяющих расходовать меньше бумаги и читать быстрее. Например, если буквенные коды стоят отдельно, они обозначают распространённые английские слова. В следующих трёх рядах, включая «завершённый» третий ряд, показаны эти словесные коды:



Таким образом, фразу «you and me» в шрифте Брайля уровня 2 можно записать так:

До сих пор я описал 31 код: промежуток без выпуклых точек между словами и три ряда по десять кодов для букв и слов. До теоретически доступных 64 кодов нам ещё далеко. Но, как мы увидим, в шрифте Брайля уровня 2 ничто не пропадает зря.

Коды букв от *a* до *j* можно дополнить выпуклой точкой 6. В основном они обозначают сокращения сочетаний букв внутри слов, а кроме того, включают букву *w* и ещё одно сокращение целого слова:



Например, английское слово «about» можно записать шрифтом Брайля уровня 2 следующим образом:

На следующем этапе появляется возможность неоднозначности, которой не было в первоначальной системе Луи Брайля. Коды букв от *a* до *j* можно как бы опустить вниз, чтобы задействовать только точки 2, 3, 5 и 6. В зависимости от контекста эти коды обозначают некоторые знаки препинания и сокращения:

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

Первые четыре кода обозначают запятую, точку с запятой, двоеточие и точку. Обратите внимание: для открывающей и закрывающей круглых скобок применяется один и тот же код, а для открывающей и закрывающей кавычек - два разных. Поскольку эти коды можно принять за буквы от *a* до *j*, понять их значение можно только в более широком контексте среди других букв.

Итак, мы дошли до 51 кода. В следующих шести кодах различные неиспользованные комбинации точек 3, 4, 5 и 6 обозначают сокращения и дополнительные знаки препинания:

st	ing	ble	ar	'	com
/		#			-

Код «ble» очень важен, поскольку вне слова он означает, что следующие коды следует интерпретировать как числа. Числовые коды совпадают с кодами букв от *a* до *j*:

1	2	3	4	5	6	7	8	9	0

Таким образом, следующая последовательность кодов

означает число 256.

Если вы вели подсчёт, до максимума в 64 нам не хватает ещё семи кодов. Вот они:

Первый из них - выпуклая точка 4 - служит индикатором диакритического знака. Остальные применяются как префиксы некоторых сокращений, а также для других целей. Если подняты точки 4 и 6 - это пятый код в данном ряду, - то в зависимости от контекста код означает десятичную точку в числе или выделение. Если подняты точки 5 и 6 - шестой код, - это буквенный индикатор, отменяющий действие числового индикатора.

Наконец, если вы задавались вопросом, как шрифт Брайля кодирует прописные буквы, для этого используется точка 6 - индикатор прописной буквы. Он указывает, что следующую букву следует считать прописной. Например, имя создателя этой системы можно записать так:

Эта последовательность начинается с индикатора прописной буквы, за которым следуют буква *l*, сокращение *ou*, буквы *i* и *s*, пробел, ещё один индикатор прописной буквы и буквы *b*, *r*, *a*, *i*, *l*, *l* и *e*. (На практике имя можно было бы сократить ещё сильнее, отбросив две последние непроизносимые буквы или записав его как «brl».)

Подведём итог. Мы увидели, что шесть двоичных элементов - точек - дают 64 возможных кода и ни одним больше. Так получилось, что многие из этих 64 кодов исполняют две роли в зависимости от контекста. Особый интерес представляет числовой индикатор вместе с буквенным индикатором, отменяющим его действие. Эти коды изменяют смысл следующих за ними кодов: буквенные значения превращаются в числовые, а затем числовые снова в буквенные. Такие коды часто называют кодами *предшествования*, или *сдвига*. Они изменяют смысл всех последующих кодов, пока сдвиг не будет отменён.

Код сдвига напоминает удержание клавиши Shift на компьютерной клавиатуре. Название возникло потому, что соответствующая клавиша старых пишущих машинок механически сдвигала механизм, позволяя печатать прописные буквы.

Индикатор прописной буквы Брайля означает, что следующая - и только следующая - буква должна быть прописной, а не строчной. Такой код называется *escape-кодом*. Escape-коды позволяют «выйти» из обычной интерпретации кода и истолковать его иначе. Коды сдвига и escape-коды часто встречаются при представлении письменных языков двоичными кодами, но они способны вносить сложности: отдельные коды нельзя интерпретировать сами по себе, не зная, какие коды им предшествовали.

Уже в 1855 году некоторые сторонники шрифта Брайля начали расширять систему ещё одним рядом из двух точек. Восьмиточечный шрифт Брайля использовался для специальных задач, в том числе для записи музыки, стенографии и японских иероглифов кандзи. Поскольку число уникальных кодов при этом увеличивается до 2^8, или 256, такая система оказалась удобной и в некоторых компьютерных приложениях: строчные и прописные буквы, цифры и знаки препинания могут получить собственные уникальные коды без неудобств, связанных с кодами сдвига и escape-кодами.

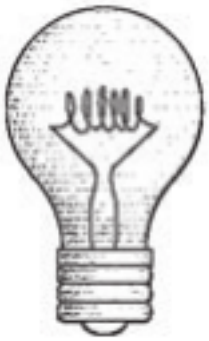
Глава 4. Устройство фонарика

Фонарики полезны для множества задач, среди которых чтение под одеялом и передача кодированных сообщений - лишь две наиболее очевидные. Обычный бытовой фонарик может также стать главным экспонатом наглядного учебного рассказа о вездесущей субстанции, называемой электричеством.

Электричество - удивительное явление: оно повсеместно приносит пользу, но при этом остаётся в значительной степени загадочным даже для тех, кто делает вид, будто понимает принцип его действия. К счастью, чтобы разобраться, как электричество используется внутри компьютеров, нам достаточно усвоить лишь несколько основных понятий.

Фонарик, несомненно, относится к самым простым электроприборам, встречающимся почти в каждом доме. Разобрав обычный фонарик, вы обнаружите одну или несколько батареек, лампочку, выключатель, несколько металлических деталей и корпус, удерживающий всё вместе.

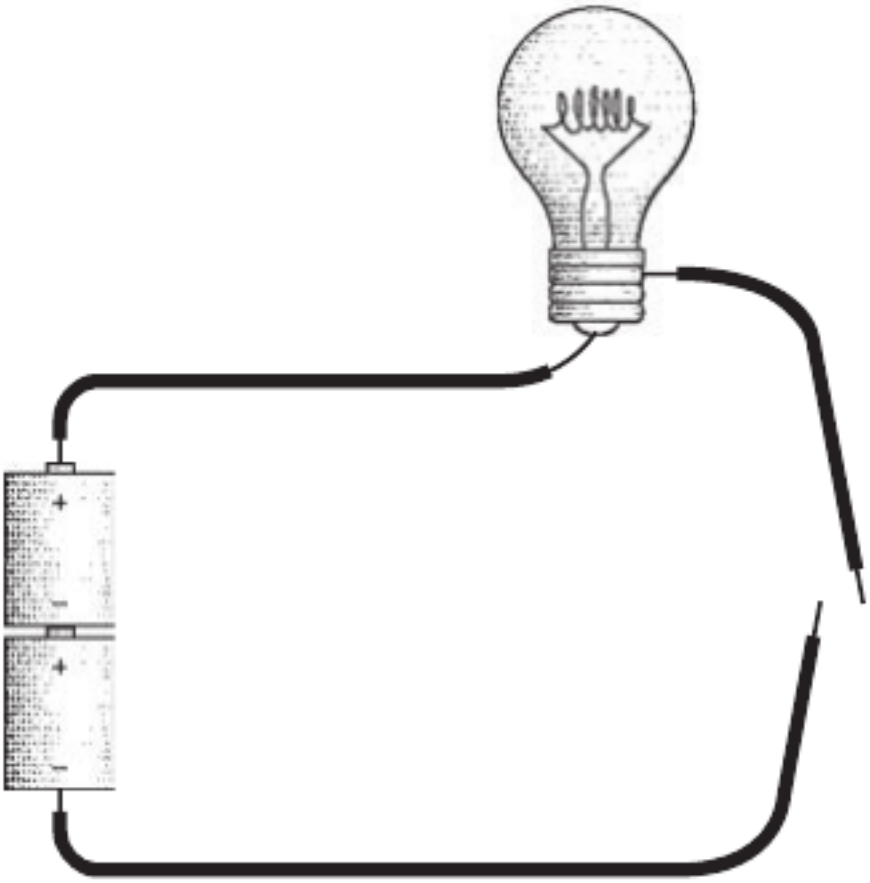
В наши дни в большинстве фонариков применяются светоизлучающие диоды (light-emitting diodes, LED), но одно из преимуществ более старомодных лампочек состоит в том, что сквозь стекло можно увидеть их внутреннее устройство:



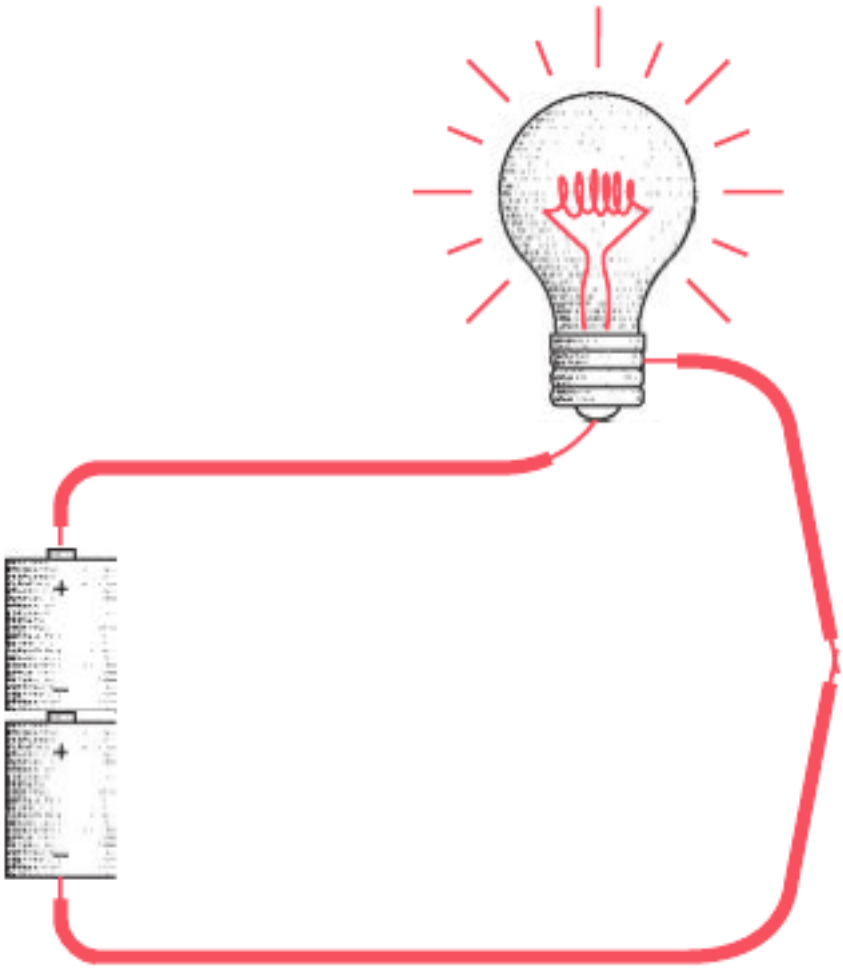
Такая лампочка называется *лампой накаливания*. Большинство американцев полагает, что лампу накаливания изобрёл Томас Эдисон, а британцы совершенно уверены, что это заслуга Джозефа Суона. В действительности многие другие учёные и изобретатели совершили важнейшие шаги ещё до того, как к работе подключились Эдисон и Суон.

Внутри лампочки находится вольфрамовая нить, которая светится при прохождении электричества. Колба заполнена инертным газом, не позволяющим вольфраму сгореть при нагревании. Два конца нити соединены с тонкими проводниками, прикреплёнными к цилиндрическому цоколю лампочки и к контакту на его нижнем конце.

Простейший фонарик можно собрать самостоятельно, отказавшись от всех деталей, кроме батареек и лампочки. Понадобятся также несколько коротких отрезков изолированного провода - с зачищенными концами - и достаточное количество рук, чтобы удерживать всё вместе:



Обратите внимание на два свободных конца проводов справа на схеме. Это и есть наш выключатель. Если батарейки исправны, а лампочка не перегорела, при соприкосновении свободных концов свет включится:



В этой книге красный цвет обозначает, что по проводам течёт электричество и зажигает лампочку.

Мы построили простую электрическую цепь, и прежде всего следует заметить, что *цепь* представляет собой *кольцо*. Лампочка загорается лишь тогда, когда путь от батареек к проводу, затем к лампочке, выключателю и обратно к батарейкам остаётся непрерывным. Любой разрыв в цепи погасит лампочку. Выключатель как раз и предназначен для управления этим процессом.

Кольцевая форма электрической цепи наводит на мысль, что по ней что-то движется, возможно подобно воде, текущей по трубам. Аналогия с «водой и трубами» очень часто используется при объяснении электричества, но в конце концов перестаёт работать, как неизбежно происходит со всеми аналогиями. Электричество не похоже ни на что другое во Вселенной, поэтому нам придётся рассматривать его таким, каково оно есть.

Один из подходов к пониманию работы электричества называется *электронной теорией*. Она объясняет электричество как движение электронов.

Как нам известно, вся материя - то есть вещество, которое мы обычно можем видеть и осязать, - состоит из чрезвычайно малых объектов, называемых атомами. Каждый атом образован частицами трёх типов: нейтронами, протонами и электронами. Иногда атом изображают как крохотную Солнечную систему, где нейтроны и протоны объединены в ядро, а электроны вращаются вокруг него, подобно планетам вокруг Солнца, однако эта модель устарела.

Число электронов в атоме обычно совпадает с числом протонов. Но при определённых обстоятельствах электроны могут отрываться от атомов. Именно так возникает электричество.

Слова *электрон* и *электричество* происходят от древнегреческого слова ηλεκτρον (*elektron*), которое, как ни странно, означает «янтарь» - похожую на стекло затвердевшую древесную смолу. Причина такого неожиданного происхождения в опытах древних греков: они натирали янтарь шерстью и получали то, что теперь называется *статическим* электричеством. При трении шерсти о янтарь шерсть забирает у янтаря электроны. В результате в шерсти оказывается больше электронов, чем протонов, а в янтаре - меньше электронов, чем протонов. В современных вариантах этого опыта ковровое покрытие забирает электроны с подошв нашей обуви.

Протоны и электроны обладают характеристикой, называемой *зарядом*. Считается, что протоны имеют положительный заряд (+), а электроны - отрицательный (-), но эти обозначения не означают плюс и минус в арифметическом смысле и не указывают, будто у протонов есть нечто, отсутствующее у электронов. Знаки + и - лишь показывают, что протоны и электроны в некотором отношении противоположны. Эта противоположность проявляется в их взаимодействии.

Протоны и электроны наиболее «довольны» и устойчивы, когда присутствуют вместе в равных количествах. Система с неравным числом протонов и электронов стремится восстановить равновесие. Когда ковровое покрытие забирает электроны с вашей обуви, в конце концов равновесие восстанавливается: вы касаетесь какого-либо предмета и чувствуете искру. Эта искра статического электричества представляет собой движение электронов по довольно извилистому пути: от ковра через ваше тело и обратно к обуви.

Статическое электричество не ограничивается маленькими искрами, возникающими, когда пальцы касаются дверной ручки. Во время грозы в нижней части облаков накапливаются электроны, а верхняя часть облаков их теряет; в конце концов разность выравнивается ударом молнии. Молния - это огромное число электронов, очень быстро перемещающихся из одного места в другое.

Электричество в цепи фонарика явно ведёт себя гораздо спокойнее искры или молнии. Лампочка горит ровно и непрерывно, потому что электроны не просто перескакивают из одного места в другое. Когда один атом в цепи отдаёт электрон соседнему атому, он забирает другой электрон у следующего атома, который забирает электрон у ещё одного соседнего атома, и так далее. Электричество в цепи - это переход электронов от атома к атому.

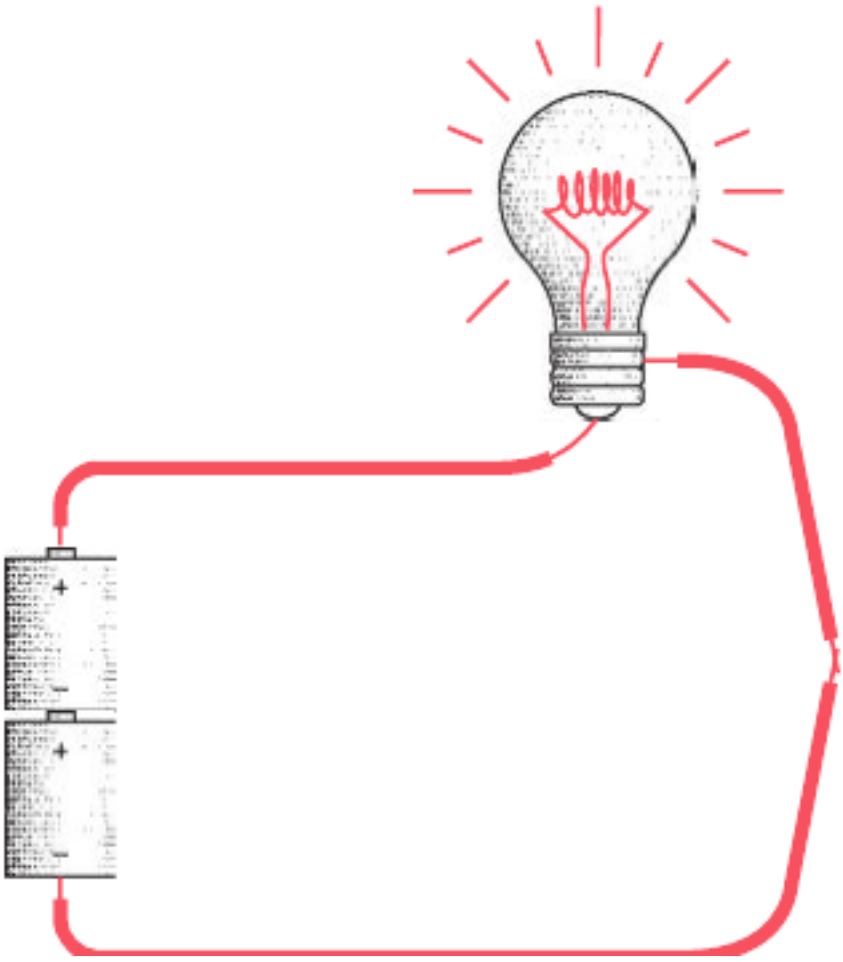
Само по себе это не происходит. Нельзя просто соединить проводами произвольный набор предметов и ожидать появления электричества. Нужно нечто, что вызовет движение электронов по цепи. Вернувшись к схеме простейшего фонарика, можно с уверенностью предположить, что движение электричества начинают не провода и не лампочка, а, вероятнее всего, батарейки.

Батарейки для фонариков обычно имеют цилиндрическую форму и в зависимости от размера обозначаются D, C, A, AA или AAA. Плоский конец батарейки помечен знаком минус (-), а на другом конце находится небольшой выступ со знаком плюс (+).

Батарейки вырабатывают электричество посредством химической реакции. Химические вещества выбирают так, чтобы в результате реакции на стороне со знаком минус возникали лишние электроны - эта сторона называется отрицательным выводом, или *анодом*, - а на другой стороне батарейки требовались дополнительные электроны - это положительный вывод, или *катод*. Так химическая энергия преобразуется в электрическую.

Батарейки, применяемые в фонариках, создают напряжение около 1,5 вольта. Что именно это означает, я вскоре объясню.

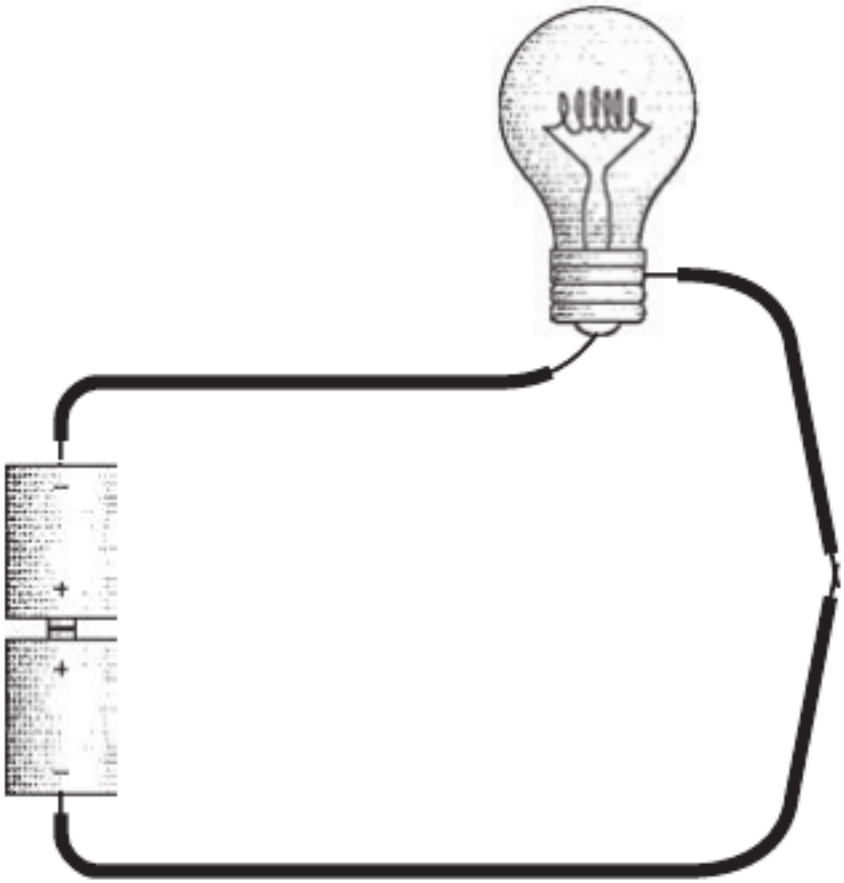
Химическая реакция не может продолжаться, если лишние электроны не будут каким-либо способом уходить с отрицательного вывода батарейки и возвращаться на положительный. Для этого служит электрическая цепь, соединяющая два вывода. По такой цепи электроны движутся против часовой стрелки:



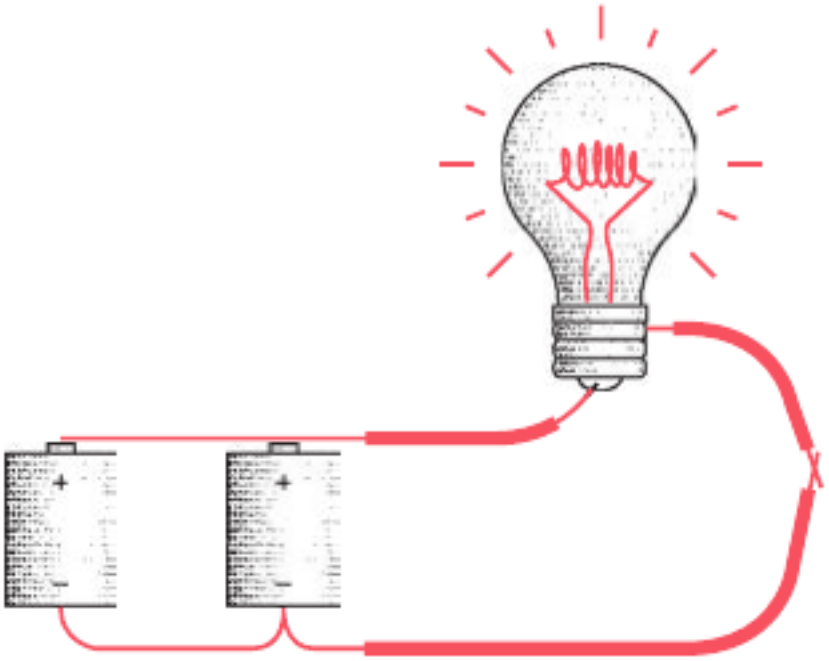
Электроны из химических веществ батареек, возможно, не смешивались бы так свободно с электронами в медных проводах, если бы не один простой факт: все электроны, где бы они ни находились, одинаковы. Электрон меди ничем не отличается от любого другого электрона.

Обратите внимание, что обе батарейки ориентированы одинаково. Положительный конец нижней батарейки принимает электроны с отрицательного конца верхней. Получается, будто две батарейки объединены в одну более крупную с положительным выводом на одном конце и отрицательным на другом. Напряжение объединённой батареи равно 3 вольтам, а не 1,5.

Если перевернуть одну из батареек, цепь работать не будет:



Двум положительным концам батареек для химических реакций необходимы электроны, но попасть туда они не могут, поскольку эти концы соединены друг с другом. Если соединены два положительных конца батареек, следует соединить и два отрицательных:



Такая схема работает. Говорят, что батарейки соединены *параллельно*, а не *последовательно*, как было показано ранее. Общее напряжение равно 1,5 вольта - столько же, сколько у каждой отдельной батарейки. Лампочка, вероятно, по-прежнему будет гореть, хотя и не так ярко, как при двух последовательно соединённых батарейках. Зато батарейки прослужат вдвое дольше.

Обычно мы представляем батарейку как источник электричества для цепи. Но, как мы уже увидели, цепь можно также считать средством, позволяющим протекать химическим реакциям в батарейке. Цепь забирает электроны с отрицательного конца батарейки и доставляет их к положительному. Реакции продолжаются, пока все химические вещества не будут израсходованы, после чего батарейку следует надлежащим образом утилизировать или перезарядить.

От отрицательного конца батарейки к положительному электроны проходят по проводам и через лампочку. Но зачем нужны провода? Разве электричество не может просто пройти по воздуху? И да и нет. Электричество способно проходить через воздух, особенно влажный, иначе мы не видели бы молний. Однако через воздух оно проходит далеко не так легко.

Одни вещества проводят электричество заметно лучше других. Способность элемента проводить электричество связана с его субатомным строением. Электроны окружают ядро на нескольких уровнях, называемых оболочками. Атом, у которого на внешней оболочке находится всего один электрон, легко его отдаёт, а именно это необходимо для переноса электричества. Такие вещества способствуют прохождению электричества, поэтому их называют *проводниками*. Лучшими проводниками являются медь, серебро и золото. Не случайно эти три элемента находятся в одном столбце периодической таблицы. Чаще всего провода изготавливают из меди.

Противоположность проводимости - *сопротивление*. Одни вещества сильнее других сопротивляются прохождению электричества; их называют *резистивными материалами*, или *резисторами*. Если вещество обладает очень высоким сопротивлением, то есть почти совсем не проводит электричество, оно называется *изолятором*. Хорошими изоляторами служат резина и пластмасса, поэтому ими часто покрывают провода. Хорошими изоляторами являются также ткань,

дерево и сухой воздух. Впрочем, если напряжение достаточно велико, электричество проводит практически всё.

Сопротивление меди очень мало, но всё же не равно нулю. Чем длиннее провод, тем выше его сопротивление. Если попытаться собрать фонарик с проводами длиной в несколько миль, их сопротивление окажется настолько высоким, что фонарик не заработает.

Чем толще провод, тем ниже его сопротивление. Это может показаться несколько нелогичным. Можно вообразить, будто для «наполнения» толстого провода требуется гораздо больше электричества. В действительности же благодаря толщине провода в нём имеется гораздо больше электронов, способных перемещаться.

Я уже упоминал напряжение, но пока не дал ему определения. Что означает утверждение, что батарейка имеет напряжение 1,5 вольта? На самом деле напряжение, названное в честь графа Алессандро Вольты (1745-1827), который в 1800 году изобрёл первую батарею, относится к самым трудным понятиям начального курса электричества. Напряжение означает *потенциальную возможность* совершить работу. Оно существует независимо от того, подключено ли что-либо к батарейке.

Представьте кирпич. Лежащий на полу кирпич имеет очень небольшой потенциал. Если держать его в руке на высоте четырёх футов над полом, потенциал будет больше. Чтобы реализовать этот потенциал, достаточно отпустить кирпич. Если держать его в руке на крыше высокого здания, потенциал станет гораздо больше. Во всех трёх случаях вы держите кирпич и он ничего не делает, однако *потенциал* различается.

Гораздо более простое электрическое понятие - *ток*. Ток связан с числом электронов, действительно проносящихся по цепи. Силу тока измеряют в *амперах*, названных в честь Андре-Мари Ампера (1775-1836), а в разговорной речи часто говорят просто «амперы», например «предохранитель на 10 ампер». Чтобы получить ток силой в один ампер, через определённую точку каждую секунду должно проходить свыше 6 квинтиллионов электронов. Это 6 с 18 нулями, то есть 6 миллиардов миллиардов.

Здесь снова помогает аналогия с водой и трубами. Ток подобен *количеству* воды, проходящей по трубе. Напряжение подобно *давлению* воды. Сопротивление подобно ширине трубы: чем уже труба, тем выше сопротивление. Следовательно, чем выше давление, тем больше воды проходит по трубе; чем уже труба, тем меньше воды через неё проходит. Количество проходящей по трубе воды - ток - прямо пропорционально давлению воды - напряжению - и обратно пропорционально узости трубы - сопротивлению.

В электрической цепи силу тока можно рассчитать, если известны напряжение и сопротивление. Сопротивление - склонность вещества препятствовать потоку электронов - измеряется в *омах*, названных в честь Георга Симона Ома (1789-1854), который также сформулировал знаменитый закон Ома. Закон гласит:

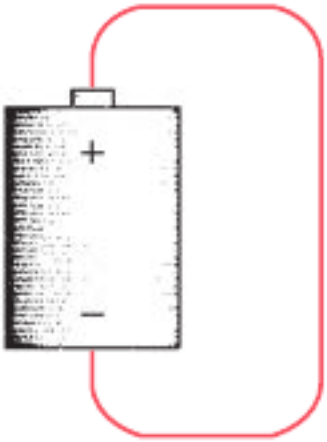
```
I = E / R
```

Здесь буквой I традиционно обозначают силу тока в амперах, буквой E - напряжение (она происходит от выражения *electromotive force*, «электродвижущая сила»), а буквой R - сопротивление.

Для примера рассмотрим батарейку, которая просто лежит и ни к чему не подключена:

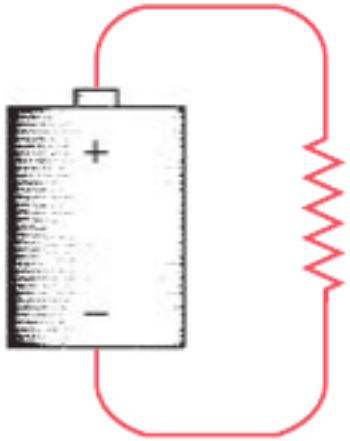
Напряжение E равно 1,5 вольта. Это потенциальная возможность совершить работу. Но поскольку положительный и отрицательный выводы соединяет только воздух, сопротивление, обозначаемое R, очень, очень, очень велико. Следовательно, ток I равен 1,5 вольта, делённым на большое число. Иными словами, сила тока почти равна нулю.

Теперь соединим положительный и отрицательный выводы коротким отрезком медного провода. Начиная с этой схемы, изоляция проводов больше не изображается:



Это называется *коротким замыканием*. Напряжение по-прежнему равно 1,5 вольта, но сопротивление теперь очень, очень мало. Сила тока равна 1,5 вольта, делённым на очень маленькое число, то есть окажется очень, очень большой. По проводу будет двигаться огромное число электронов. В действительности ток ограничивается физическим размером батарейки. Вероятно, она не сможет отдать столь большой ток, и напряжение упадёт ниже 1,5 вольта. Если батарейка достаточно велика, провод нагреется, поскольку электрическая энергия будет преобразовываться в тепло. При очень сильном нагреве провод начнёт светиться и даже может расплавиться.

Большинство цепей находится где-то между двумя этими крайностями. Условно их можно изобразить так:



Зигзагообразная линия знакома инженерам-электрикам как условное обозначение резистора. Здесь она означает, что сопротивление цепи не слишком мало и не слишком велико.

Провод с низким сопротивлением может нагреться и начать светиться. Именно так работает лампа накаливания.

Сопротивление нити, обычно применяемой в лампах накаливания для фонариков, составляет около 4 ом. Если фонарику нужны две батарейки, соединённые одна за другой, сила тока равна 3 вольтам, делённым на 4 ома, то есть 0,75 ампера. Это значение можно также записать как 750 миллиампер. Следовательно, через лампочку каждую секунду проходит свыше 4,5 квинтиллиона электронов. Из-за сопротивления нити электрическая энергия преобразуется в свет и тепло.

Ещё одна распространённая единица измерения электрических величин - ватт, названный в честь Джеймса Уатта (1736-1819), наиболее известного своими работами над паровой машиной. В ваттах измеряется мощность P, которую можно вычислить по формуле

```
P = E × I
```

Напряжение 3 вольта и ток 0,75 ампера показывают, что в нашем фонарике используется лампочка мощностью 2,25 ватта. Светодиоды в целом вытесняют лампы накаливания, поскольку дают столько же света при меньшем выделении тепла и меньшей мощности. Счета за электроэнергию основаны на ваттах, поэтому снижение мощности лампочек экономит деньги и одновременно помогает окружающей среде.

Казалось бы, мы уже разобрали все части фонарика: батарейки, провода и лампочку. Но забыли о самом важном!

Да, о выключателе. Он определяет, течёт ли по цепи электричество. Когда выключатель позволяет электричеству проходить, говорят, что он *включён*, или *замкнут*. *Выключенный*, или *разомкнутый*, выключатель ток не пропускает. (Слова «замкнутый» и «разомкнутый» применительно к выключателям употребляются противоположно словам «закрытая» и «открытая» применительно к двери. Закрытая дверь не даёт ничему пройти, а замкнутый выключатель пропускает электричество.)

Выключатель либо замкнут, либо разомкнут. Ток либо течёт, либо нет. Лампочка либо горит, либо не горит.

Подобно двоичным кодам, изобретённым Морзе и Брайлем, простой фонарик либо включён, либо выключен. Промежуточного состояния нет. Это сходство между двоичными кодами и простыми электрическими цепями окажется очень полезным в следующих главах.

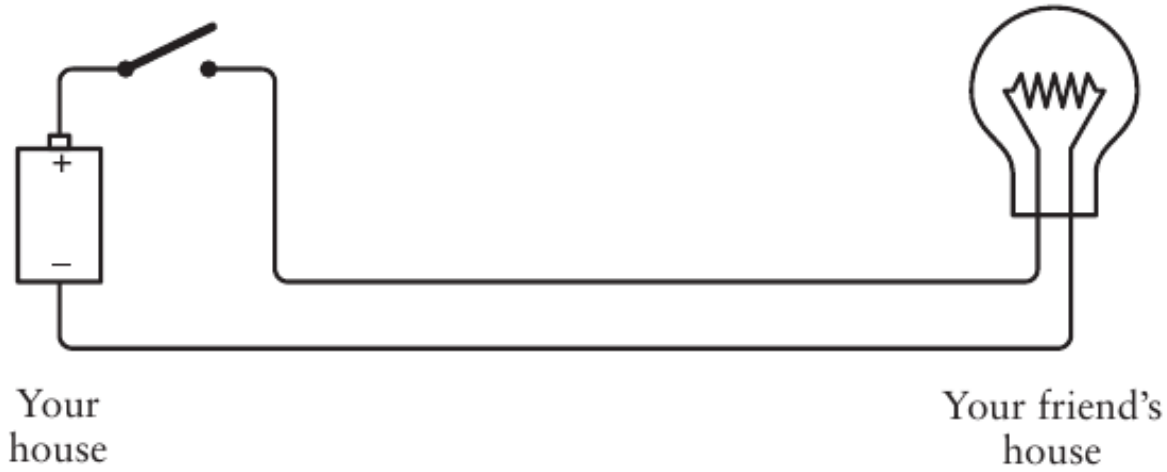
Глава 5. Связь за углом

Вам 12 лет. В один ужасный день семья вашего лучшего друга переезжает в другой город. Время от времени вы обмениваетесь электронными письмами и сообщениями, но это уже не так увлекательно, как ночные разговоры с помощью фонариков, мигающих азбукой Морзе. В конце концов ваш второй по близости друг, живущий в соседнем доме, становится новым лучшим другом. Пора научить его азбуке Морзе и снова возобновить ночное мигание фонариками.

Проблема в том, что окно спальни нового друга не обращено к окну вашей спальни. Дома стоят рядом, но окна спален смотрят в одну сторону. Если только не придумать, как установить снаружи несколько зеркал, фонарики больше не подходят для общения после наступления темноты.

Или всё-таки подходят?

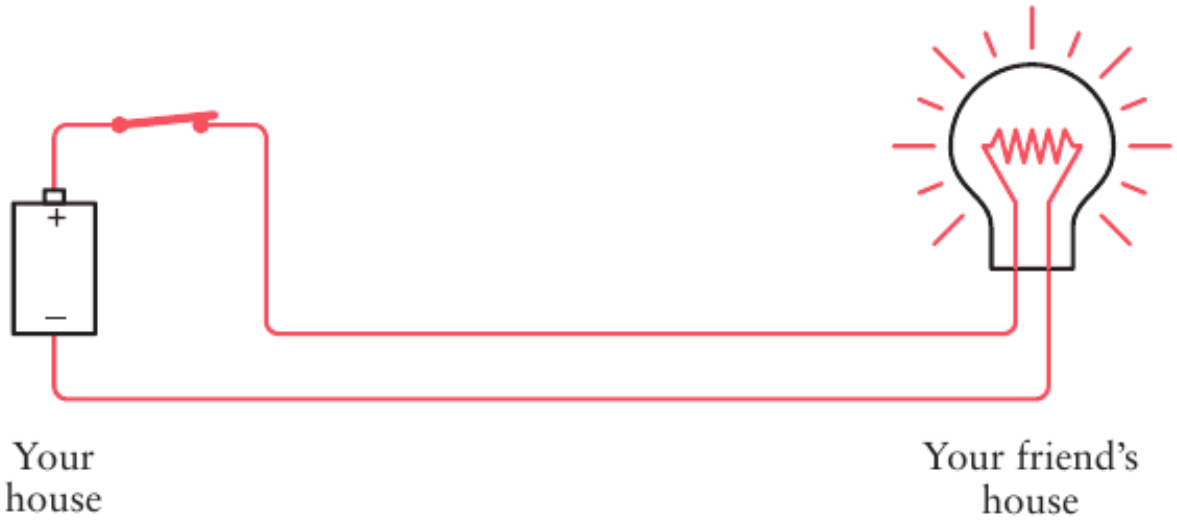
Возможно, к этому времени вы уже кое-что узнали об электричестве, поэтому решаете собрать собственные фонарики из батареек, лампочек, выключателей и проводов. В первом опыте батарейки и выключатель устанавливаются в вашей спальне. Два провода выходят из окна, пересекают забор и входят в спальню друга, где подключаются к лампочке:



Начиная с этого момента электрические цепи будут изображаться скорее условно, чем реалистично. Хотя на схеме показана только одна батарейка, на практике вы можете использовать две. На этой и последующих схемах выключенный, или разомкнутый, выключатель будет выглядеть так:

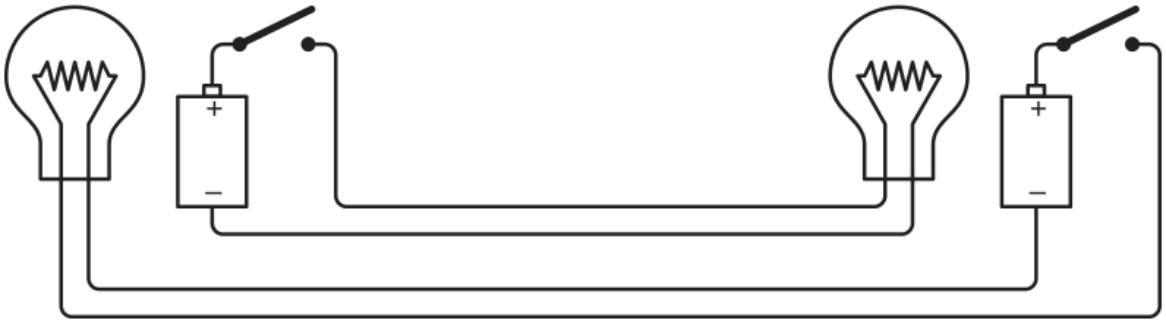
А так будет выглядеть включённый, или замкнутый, выключатель:

Фонарик в этой главе работает так же, как показанный в предыдущей, только соединяющие детали провода теперь немного длиннее. Когда вы замыкаете выключатель у себя, в доме друга загорается лампочка:



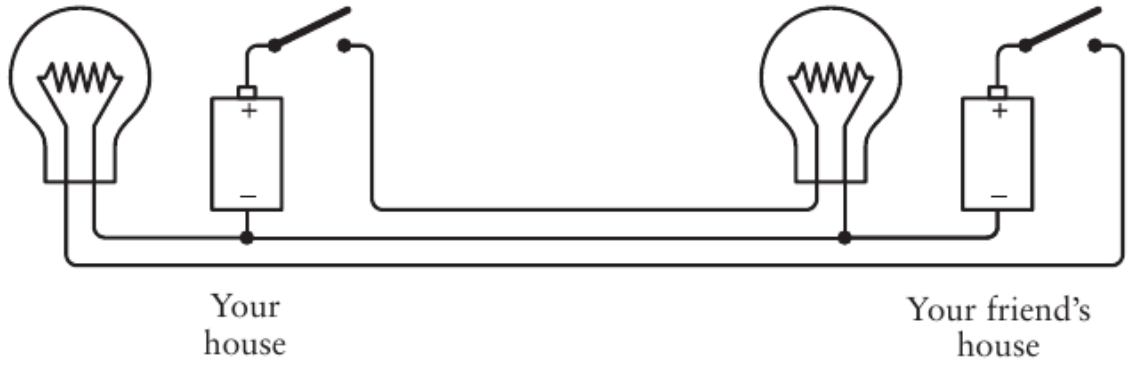
Теперь можно передавать сообщения азбукой Морзе.

После того как первый фонарик заработал, можно протянуть ещё одну линию дальней связи, чтобы друг тоже мог передавать вам сообщения:



Поздравляем! Вы только что соорудили двунаправленную телеграфную систему. Обратите внимание: перед нами две одинаковые, совершенно независимые друг от друга цепи. Теоретически вы можете отправлять сообщение другу одновременно с тем, как он отправляет сообщение вам, хотя вашему мозгу, вероятно, будет трудно читать и передавать сообщения в одно и то же время.

Возможно, вы проявите достаточно сообразительности и заметите, что между домами необязательно протягивать столько проводов. Один из четырёх проводов можно убрать, соединив цепи следующим образом:

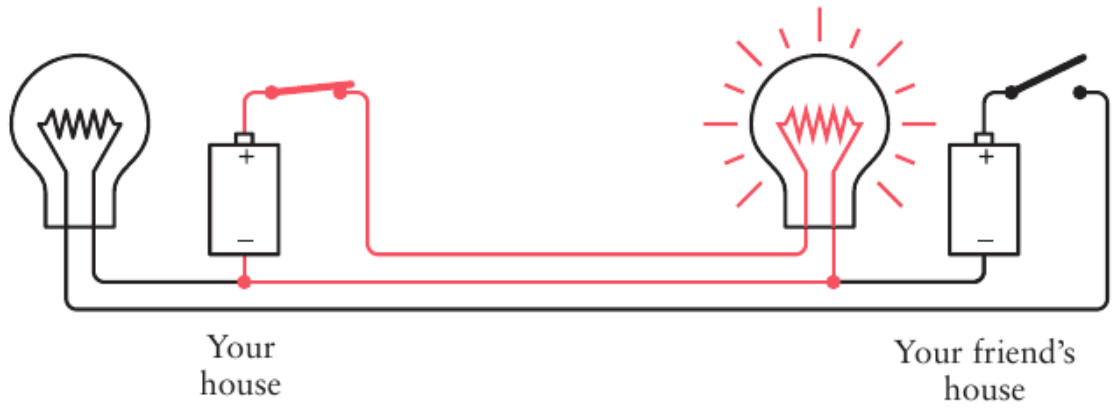


В этой книге соединённые друг с другом провода обозначаются маленькой точкой в месте соединения. На данной схеме таких соединений два: одно под батарейкой в вашем доме, второе под лампочкой в доме друга.

Обратите внимание, что отрицательные выводы двух батареек теперь соединены. Две кольцевые цепи - от батарейки к выключателю, лампочке и обратно к батарейке - по-прежнему работают независимо, хотя теперь объединены.

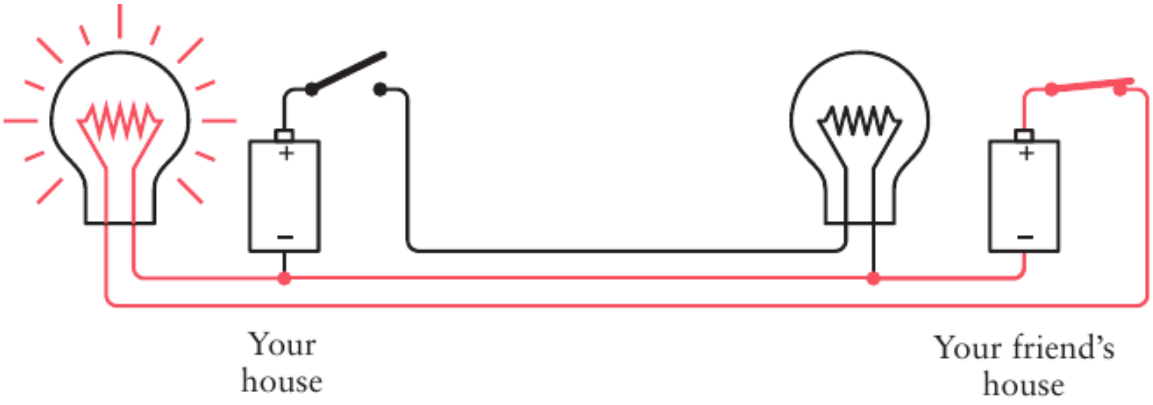
Соединение между двумя цепями называется *общим проводом* (common). На этой схеме общий провод проходит между двумя точками соединения: от места, где соединены крайняя левая лампочка и батарейка, до места соединения крайней правой лампочки и батарейки.

Рассмотрим схему подробнее, чтобы убедиться, что в ней нет никакой хитрости. Сначала вы замыкаете выключатель на своей стороне, и в доме друга загорается лампочка. Красные провода показывают путь электрического тока:

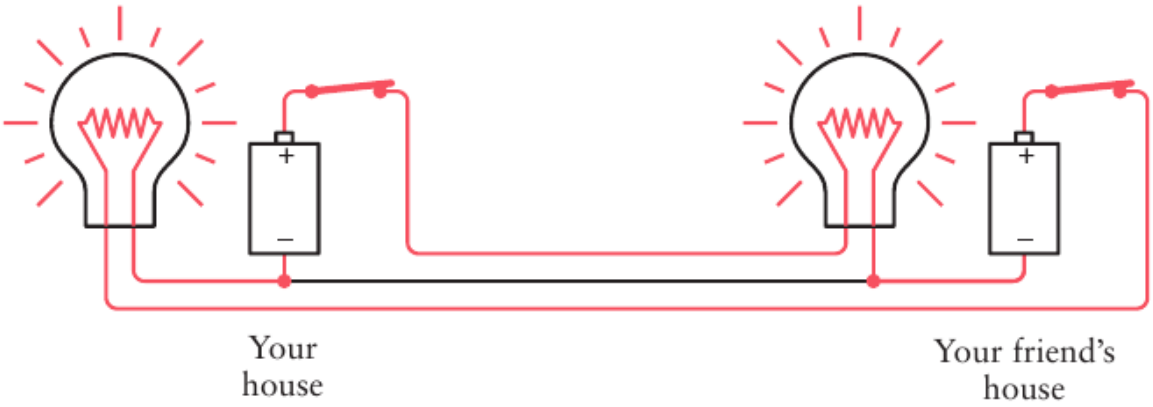


В другой части схемы ток не течёт, потому что у электронов нет пути, по которому они могли бы завершить цепь.

Когда вы не передаёте сообщение, а друг передаёт, выключатель в доме друга управляет лампочкой в вашем доме. Красные провода снова показывают путь электрического тока:



Когда вы с другом пытаетесь передавать одновременно, иногда оба выключателя разомкнуты, иногда один замкнут, а другой разомкнут, иногда же замкнуты оба. При двух замкнутых выключателях электричество течёт по цепи так:



Любопытно, что, когда горят обе лампочки, через общую часть цепи ток не течёт.

Объединив общим проводом две отдельные цепи в одну, мы сократили число проводов между домами с четырёх до трёх, а расходы на провод - на 25 процентов.

Если бы провода требовалось протянуть на очень большое расстояние, возникло бы искушение ещё сильнее сократить затраты, убрав ещё один провод. К сожалению, с батарейками D напряжением 1,5 вольта и маленькими лампочками это невозможно. Но если бы мы располагали батареями на 100 вольт и гораздо более крупными лампочками, сделать это вполне удалось бы.

Вот в чём хитрость: после создания общей части цепи для неё необязательно использовать провод. Провод можно заменить кое-чем другим. А заменить его можно гигантским шаром диаметром около 7900 миль, состоящим из металла, камня, воды и органического вещества, по большей части мёртвого. Этот гигантский шар известен нам как Земля.

В предыдущей главе, перечисляя хорошие проводники, я упомянул серебро, медь и золото, но не гравий и перегной. В действительности земля - не слишком хороший проводник, хотя некоторые её виды, например влажная почва, лучше других, например сухого песка. Но о проводниках мы узнали одно: чем больше, тем лучше. Очень толстый провод проводит ток намного лучше очень тонкого. Именно здесь Земля не знает себе равных. Она действительно очень, очень, очень велика.

Чтобы использовать Землю как проводник, недостаточно воткнуть короткий провод в почву рядом с кустами помидоров. Нужен предмет, обеспечивающий значительный контакт с землёй, то есть

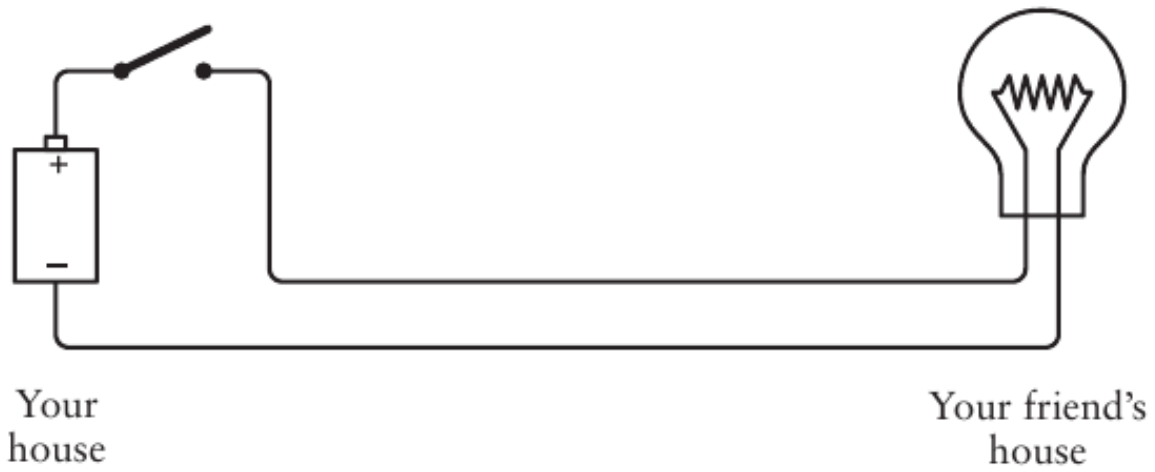
проводник с большой площадью поверхности. Подойдёт медный стержень длиной не менее 8 футов и диаметром 1/2 дюйма. Он обеспечит 150 квадратных дюймов контакта с землёй. Стержень можно забить в землю кувалдой, а затем присоединить к нему провод. Или, если трубы холодного водоснабжения в доме сделаны из меди и выходят из земли снаружи, провод можно подключить к трубе.

Электрический контакт с Землёй в Англии называют *earth*, а в Америке - *ground*. Слово *ground* способно вызвать некоторую путаницу, поскольку им также часто называют часть цепи, которую мы именовали *общим проводом*. В этой главе и далее, пока не будет сказано иное, под заземлением понимается физическое соединение с Землёй.

На электрических схемах заземление обозначают следующим символом:

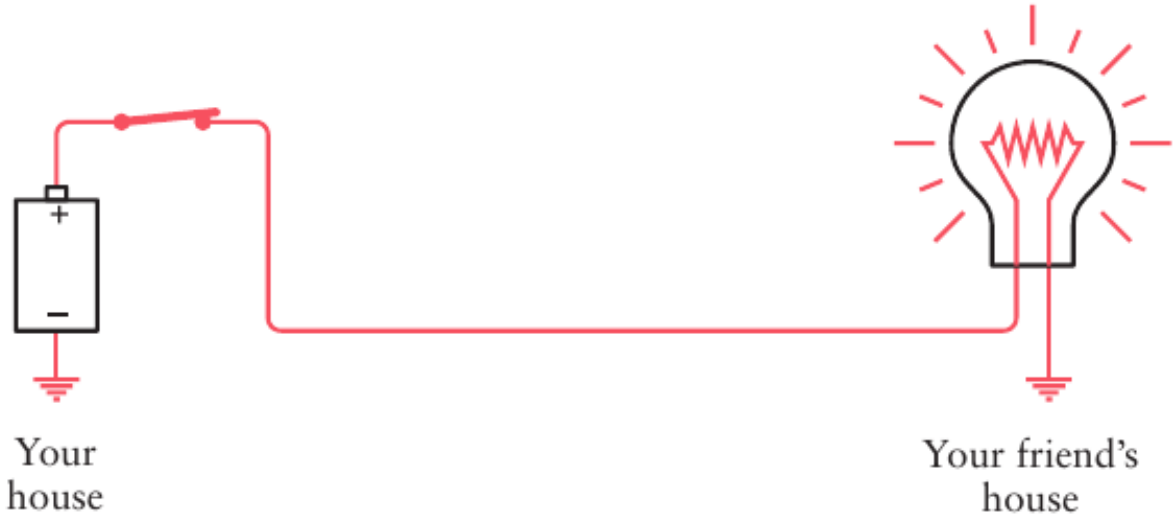
Электрики используют этот знак, поскольку не хотят каждый раз рисовать восьмифутовый медный стержень, закопанный в землю. О цепи с таким соединением говорят *connected to ground* («соединена с землёй») или *grounded* («заземлена»), а не используют более многословное *connected to the ground*.

Посмотрим, как это работает. В начале главы мы рассматривали следующую однонаправленную схему:



При использовании высоковольтных батарей и лампочек между вашим домом и домом друга потребовался бы всего один провод, поскольку вторым соединителем могла бы служить Земля:

После включения выключателя ток течёт следующим образом:



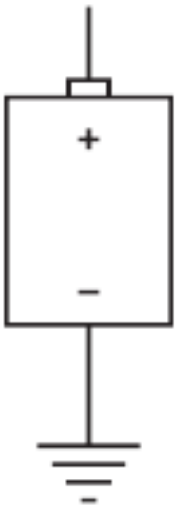
Электроны выходят из земли у дома вашего друга, проходят через лампочку и провод, затем через выключатель в вашем доме и входят в положительный вывод батарейки. Электроны из отрицательного вывода батарейки уходят в землю.

Можно также представить, как электроны спрыгивают с восьмифутового медного стержня, закопанного во дворе вашего дома, в землю, а затем торопливо пробираются сквозь неё к восьмифутовому медному стержню во дворе друга. Но если учесть, что Земля выполняет ту же работу для многих тысяч электрических цепей по всему миру, возникает вопрос: откуда электроны знают, куда идти? Очевидно, что никак. Гораздо уместнее представить Землю иначе.

Да, Земля - огромный проводник электричества, но её можно считать и источником, и хранилищем электронов. *Земля относится к электронам так же, как океан к каплям воды.* Земля служит практически неисчерпаемым источником электронов и одновременно их гигантским морем.

Однако Земля всё же обладает некоторым сопротивлением. Именно поэтому при работе с батарейками D напряжением 1,5 вольта и лампочками для фонариков нельзя использовать заземление, чтобы сократить число проводов. Для низковольтных батареек сопротивление Земли попросту слишком велико.

Обратите внимание: на двух предыдущих схемах отрицательный вывод батарейки соединён с землёй:



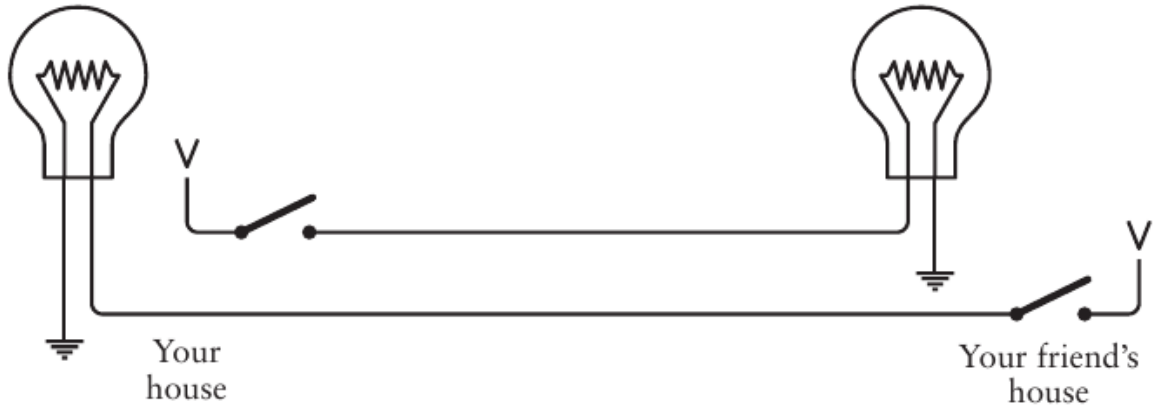
Далее я больше не буду рисовать такую соединённую с землёй батарейку. Вместо неё будет использоваться фигура, похожая на прописную букву V, которая обозначает *напряжение* - voltage. Провод, отходящий от прописной V, равнозначен проводу, подключённому к положительному выводу батарейки, отрицательный вывод которой заземлён. Теперь схема однонаправленного лампочного телеграфа выглядит так:

Буква V означает *voltage*, напряжение, но в некотором смысле могла бы означать и *vacuum*, вакуум. V можно представить как электронный пылесос, а землю - как океан электронов. Электронный пылесос вытягивает электроны из земли через цепь, по пути совершая работу, например зажигая лампочку.

Землю иногда также называют точкой *нулевого потенциала*. Это означает отсутствие напряжения. Как я уже объяснял, напряжение представляет собой потенциальную возможность совершить работу, подобно тому как подвешенный в воздухе кирпич служит потенциальным источником энергии. Нулевой потенциал подобен кирпичу, лежащему на земле: падать ему больше некуда.

В главе 4 одним из первых наших наблюдений было то, что цепи представляют собой кольца. Новая схема совсем не похожа на кольцо, но по-прежнему им является. Можно заменить V батарейкой с заземлённым отрицательным выводом, а затем нарисовать провод, соединяющий все места, где показан символ заземления. В результате получится та же схема, с которой мы начали эту главу.

Итак, с помощью пары медных стержней или труб холодного водоснабжения можно построить двунаправленную систему азбуки Морзе, протянув через забор между домами всего два провода:



Функционально эта цепь совпадает со схемой на страницах 33-34, где через забор между домами проходили три провода, однако работать она будет только с высоковольтными батареями и лампочками.

В этой главе мы сделали важный шаг в развитии связи. Ранее азбука Морзе позволяла нам общаться только в пределах прямой видимости и лишь на расстоянии, которого достигал луч фонарика.

Благодаря проводам мы не только построили систему для связи за углом, вне прямой видимости, но и освободились от ограничения расстояния. Протягивая всё более длинные провода, можно общаться на расстоянии сотен и тысяч миль.

Впрочем, не совсем. Медь очень хорошо проводит электричество, но всё же не идеально. Чем длиннее провода, тем выше их сопротивление. Чем выше сопротивление, тем меньше ток. Чем меньше ток, тем тусклее лампочки.

Так какой же длины могут быть провода? Это зависит от обстоятельств. Предположим, вы используете первоначальную четырёхпроводную двунаправленную схему без земли и общих проводов, а также батарейки и лампочки для фонариков. Ради экономии вы могли сначала купить бухту из 100 футов акустического кабеля, обычно применяемого для подключения громкоговорителей к высококачественным усилителям. Акустический кабель состоит из двух удобно соединённых вместе изолированных проводов, поэтому хорошо подходит для нашей телеграфной системы. Если расстояние между вашей спальней и спальней друга меньше 50 футов, одной бухты будет достаточно.

Толщина провода измеряется по американскому калибру проводов (American Wire Gauge, AWG). Чем меньше номер AWG, тем толще провод и тем ниже его сопротивление. Если вы купили акустический кабель калибра 20, диаметр проводника составит около 0,032 дюйма, а сопротивление - около 10 ом на 1000 футов, или 1 ом на полный путь в 100 футов от одной спальни до другой и обратно.

Это совсем неплохо, но что, если растянуть провод на милю? Его общее сопротивление превысит 100 ом. Вспомним из предыдущей главы, что сопротивление нашей лампочки составляло всего 4 ома. По закону Ома легко рассчитать, что сила тока в цепи теперь будет не 0,75 ампера - 3 вольта, делённые на 4 ома, - а меньше 0,03 ампера - 3 вольта, делённые более чем на 100 ом. Почти наверняка такого тока не хватит, чтобы зажечь лампочку.

Хорошим решением был бы более толстый провод, но он может оказаться дорогим. Провод калибра 10 имеет толщину около 0,1 дюйма и сопротивление всего 1 ом на 1000 футов, или 5 ом на милю.

Другое решение - увеличить напряжение и использовать лампочки с гораздо более высоким сопротивлением, например те, которыми освещают дома. Тогда сопротивление проводов будет гораздо слабее влиять на цепь в целом.

Именно с такими проблемами в середине XIX века столкнулись люди, прокладывавшие первые телеграфные линии через Америку и Европу. Независимо от толщины проводов и высокого напряжения телеграфные линии нельзя было продолжать бесконечно. В лучшем случае предел работоспособности системы по этой схеме составлял пару сотен миль. Этого совершенно недостаточно, чтобы преодолеть тысячи миль между Нью-Йорком и Калифорнией.

Решением этой проблемы - не для фонариков, а для щёлкающих и стучащих телеграфов прошлого - оказалось простое и скромное устройство, из которого, тем не менее, можно построить целый компьютер.

Глава 6. Логика переключателей

Что есть истина? Аристотель полагал, что логика имеет к ней некоторое отношение. Собрание его трудов, известное как *«Органон»* и относящееся к IV веку до н. э., является самым ранним обширным сочинением о логике. Для древних греков логика была средством анализа языка в поисках истины и потому считалась разделом философии. Основу аристотелевской логики составлял *силлогизм*. Самый известный силлогизм, которого на самом деле нет в трудах Аристотеля, гласит:

```
Все люди смертны;
Сократ - человек;
следовательно, Сократ смертен.
```

В силлогизме два исходных суждения, или посылки, принимаются за истинные, а затем из них выводится заключение.

Смертность Сократа может показаться вполне очевидной, однако существует множество разновидностей силлогизмов. Рассмотрим, например, две посылки, предложенные математиком XIX века Чарльзом Доджсоном, известным также как Льюис Кэрролл:

```
Все философы логичны;
нелогичный человек всегда упрям.
```

Заключение - *некоторые упрямые люди не являются философами* - совсем не очевидно. Обратите внимание на неожиданное и тревожное появление слова *некоторые*.

Более двух тысяч лет математики пытались справиться с логикой Аристотеля, стремясь подчинить её при помощи математических символов и операторов.

До XIX века ближе всех к цели подошёл Готфрид Вильгельм фон Лейбниц (1648-1716), который в молодости занимался логикой, а затем переключился на другие интересы, например независимо от Исаака Ньютона и одновременно с ним изобрёл математический анализ.

А затем появился Джордж Буль.

Джордж Буль родился в Англии в 1815 году, и обстоятельства определённо складывались не в его пользу. Он был сыном сапожника и бывшей служанки, поэтому жёсткая классовая структура Британии в обычной ситуации не позволила бы ему добиться чего-либо, сильно отличавшегося от достижений его предков. Но благодаря пытливому уму и помощи отца, глубоко интересовавшегося естественными науками, математикой и литературой, юный Джордж самостоятельно получил образование, которое обычно было привилегией мальчиков из высших классов; среди прочего он изучал латинский, греческий язык и математику. Благодаря ранним математическим работам в 1849 году Буль был назначен первым профессором математики Королевского колледжа в Корке, Ирландия.

Фото: Science & Society Picture Library/Getty Images.

В середине XIX века несколько математиков работали над математическим определением логики, прежде всего Огастес де Морган, но настоящий концептуальный прорыв совершил Буль. Сначала он изложил идеи в небольшой книге *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning* («Математический анализ логики, представляющий собой опыт исчисления дедуктивного рассуждения», 1847), а затем в гораздо более длинном и амбициозном труде *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities* («Исследование законов мышления, на которых основаны математические теории логики и вероятности», 1854), который для удобства называют *The Laws of Thought* («Законы мышления»). Буль умер в 1864 году в возрасте 49 лет: он спешил на занятие под дождём, после чего заболел воспалением лёгких.

Название книги Буля 1854 года указывает на грандиозность его замысла. Буль считал, что человеческий мозг мыслит с помощью логики, а значит, найдя способ представить логику математически, мы получили бы математическое описание работы мозга. Однако математику Буля можно изучать, не обязательно соглашаясь с его нейропсихологическими взглядами.

Буль изобрёл совершенно иной вид алгебры, который впоследствии назвали булевой алгеброй, чтобы отличать её от обычной.

В обычной алгебре буквами часто обозначают числа. Эти буквы называются *операндами* и различным образом объединяются *операторами*, чаще всего + и ×. Например:

```
A = 3 × (B + 5)
```

Занимаясь обычной алгеброй, мы следуем определённым правилам. Вероятно, эти правила настолько укоренились в нашей практике, что мы уже не воспринимаем их как правила и, возможно, даже забыли их названия. Тем не менее в основе всех действий в любом разделе математики действительно лежат правила.

Первое правило состоит в том, что сложение и умножение *коммутативны*. Иными словами, символы по обе стороны операторов можно менять местами:

```
A + B = B + A
A × B = B × A
```

Вычитание и деление, напротив, *не* коммутативны.

Сложение и умножение также *ассоциативны*, то есть:

```
A + (B + C) = (A + B) + C
A × (B × C) = (A × B) × C
```

Наконец, говорят, что умножение *дистрибутивно* относительно сложения:

```
A × (B + C) = (A × B) + (A × C)
```

Ещё одна особенность обычной алгебры заключается в том, что она всегда имеет дело с числами: фунтами тофу, количеством уток, расстоянием, пройденным поездом, или секундами в сутках.

Гениальность Буля состояла в том, что он сделал алгебру более абстрактной, отделив её от понятия числа. В булевой алгебре операнды обозначают не числа, а *классы*. Класс - это просто группа объектов, подобная тому, что позднее стали называть *множеством*.

Поговорим о кошках. Кошки могут быть мужского или женского пола. Для удобства буквой M обозначим класс котов, а F - класс кошек. Помните, что эти два символа *не* обозначают число котов и кошек. Их количество может меняться каждую минуту: рождаются новые животные, а старые, к сожалению, умирают. Буквы обозначают классы кошек, то есть животных с определёнными признаками. Вместо слов «коты» можно просто говорить «M».

Другими буквами можно обозначить окрас. Например, T будет представлять класс светло-коричневых кошек, B - чёрных, W - белых, а O - кошек всех «остальных» окрасов, то есть не входящих в классы T, B и W.

Наконец, по крайней мере в рамках этого примера, кошки могут быть стерилизованными или нестерилизованными. Буквой N обозначим класс стерилизованных кошек, а U - нестерилизованных.

В обычной числовой алгебре операторы + и × обозначают сложение и умножение. В булевой алгебре используются те же символы + и ×, и здесь может возникнуть путаница. Все знают, как складывать и умножать числа в обычной алгебре, но как складывать и умножать классы?

В действительности в булевой алгебре мы ничего не складываем и не умножаем. Символы + и × означают совершенно иное.

Символ + в булевой алгебре означает *объединение* двух классов. Объединение включает всё из первого класса вместе со всем из второго. Например, B + W представляет класс всех чёрных или белых кошек.

Символ × в булевой алгебре означает *пересечение* двух классов. Пересечение включает всё, что входит одновременно *и* в первый, *и* во второй класс. Например, F × T представляет класс всех кошек, которые одновременно являются самками и имеют светло-коричневый окрас. Как и в обычной алгебре, F × T можно записать как F·T или просто FT, что и предпочитал Буль. Эти две буквы можно представить как поставленные рядом два прилагательных: «самки светло-коричневого окраса».

Чтобы избежать путаницы между обычной и булевой алгеброй, иногда вместо + и × для объединения и пересечения применяют символы ∪ и ∩. Но одна из сторон освободительного влияния Буля на математику состояла в более абстрактном использовании привычных операторов, поэтому я решил последовать его выбору и не вводить в эту алгебру новые символы.

Коммутативное, ассоциативное и дистрибутивное правила выполняются и в булевой алгебре. Более того, в булевой алгебре оператор + дистрибутивен относительно оператора ×. Для обычной алгебры это неверно:

```
W + (B × F) = (W + B) × (W + F)
```

Объединение белых кошек и чёрных кошек женского пола совпадает с пересечением двух объединений: объединения белых и чёрных кошек и объединения белых кошек с кошками женского пола. Понять это несколько трудно, но правило работает.

Для завершения булевой алгебры нужны ещё три символа. Два из них могут показаться числами, но в действительности числами не являются, поскольку используются несколько иначе. Символ 1 в булевой алгебре означает «универсум», то есть всё множество объектов, о которых идёт речь. В нашем примере 1 означает «класс всех кошек». Поэтому:

```
M + F = 1
```

Это означает, что объединение котов и кошек представляет собой класс всех кошек. Аналогично, объединение светло-коричневых, чёрных, белых кошек и кошек остальных окрасов также образует класс всех кошек:

```
T + B + W + O = 1
```

Класс всех кошек можно получить и так:

```
N + U = 1
```

Символ 1 вместе со знаком минус позволяет обозначить универсум, *исключая* что-либо. Например:

```
1 - M
```

Это класс всех кошек, кроме котов. Универсум без всех котов совпадает с классом кошек женского пола:

```
1 - M = F
```

Третий необходимый символ - 0, ноль. В булевой алгебре 0 означает пустой класс, то есть класс, не содержащий ничего. Пустой класс получается при пересечении двух взаимоисключающих классов, например класса животных одновременно мужского и женского пола:

```
F × M = 0
```

Обратите внимание: иногда символы 1 и 0 работают в булевой алгебре так же, как в обычной. Например, пересечение всех кошек с кошками женского пола даёт класс кошек женского пола:

```
1 × F = F
```

Пересечение отсутствующих кошек с кошками женского пола даёт класс, не содержащий кошек:

```
0 × F = 0
```

Объединение отсутствующих кошек со всеми кошками женского пола даёт класс кошек женского пола:

```
0 + F = F
```

Но иногда результат выглядит не так, как в обычной алгебре. Например, объединение всех кошек с кошками женского пола даёт класс всех кошек:

```
1 + F = 1
```

В обычной алгебре это не имеет особого смысла.

Поскольку F - класс всех кошек женского пола, а (1 - F) - класс всех кошек, которые не являются самками, объединение двух классов равно 1:

```
F + (1 - F) = 1
```

А пересечение двух классов равно 0:

```
F × (1 - F) = 0
```

Исторически эта формулировка представляет важное понятие логики, называемое *законом противоречия*: нечто не может одновременно быть собой и своей противоположностью.

Особенно заметно отличие булевой алгебры от обычной в таком выражении:

```
F × F = F
```

Для булевой алгебры оно совершенно разумно: пересечением кошек женского пола с кошками женского пола остаётся класс кошек женского пола. Но если бы F обозначало число, выражение выглядело бы весьма странно. Буль считал равенство

```
X² = X
```

единственным утверждением, отличающим его алгебру от обычной. Ещё одно булево равенство, которое выглядит странно с точки зрения обычной алгебры:

```
F + F = F
```

Объединение кошек женского пола с кошками женского пола по-прежнему представляет класс кошек женского пола.

Булева алгебра предоставляет математический способ решения силлогизмов Аристотеля. Снова рассмотрим первые две трети знаменитого силлогизма, но теперь сформулируем их без указания пола:

```
Все люди смертны;
Сократ - человек.
```

Буквой P обозначим класс всех людей, M - класс всех смертных существ, S - класс, содержащий Сократа. Что значит фраза «все люди смертны»? Она означает, что пересечение класса всех людей с классом всех смертных существ равно классу всех людей:

```
P × M = P
```

Было бы неверно написать P × M = M, поскольку класс всех смертных существ включает кошек, собак и вязы.

Фраза «Сократ - человек» означает, что пересечение класса, содержащего Сократа, то есть очень маленького класса, с гораздо более широким классом всех людей даёт класс, содержащий Сократа:

```
S × P = S
```

Из первого равенства мы знаем, что P равно (P × M), поэтому можем подставить это выражение во второе равенство:

```
S × (P × M) = S
```

По закону ассоциативности это равнозначно выражению

```
(S × P) × M = S
```

Но нам уже известно, что (S × P) равно S, поэтому с помощью подстановки выражение можно упростить:

```
S × M = S
```

На этом доказательство завершено. Формула сообщает, что пересечением Сократа с классом всех смертных существ является S, а значит, Сократ смертен. Если бы оказалось, что (S × M) равно 0, мы заключили бы, что Сократ не смертен. Если бы (S × M) оказалось равно M, пришлось бы заключить, что все смертные существа являются Сократом!

Использование булевой алгебры для доказательства столь очевидного факта может показаться избыточным, особенно если учесть, что Сократ продемонстрировал свою смертность 2400 лет назад. Однако булева алгебра позволяет также определить, удовлетворяет ли нечто заданному набору критериев.

Предположим, однажды вы заходите в зоомагазин и говорите продавцу: «Мне нужен стерилизованный кот белого или светло-коричневого окраса; либо стерилизованная кошка любого окраса, кроме белого; либо я возьму любую имеющуюся у вас кошку, если она чёрная». Продавец отвечает: «Значит, вам нужна кошка из класса, представленного следующим выражением:

```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

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

Проверяя правоту продавца, понятия объединения и пересечения удобно представить словами OR и AND. Я записываю их прописными буквами, поскольку обычно эти слова выражают понятия английского языка, но могут обозначать и операции булевой алгебры. Объединяя два класса, вы принимаете объекты из первого класса OR из второго. Пересекая классы, вы принимаете только объекты, входящие и в первый класс AND во второй. Кроме того, слово NOT можно использовать везде, где встречается 1 со следующим за ней знаком минус. Подведём итог:

• +, то есть объединение, может также означать OR.
• ×, то есть пересечение, может также означать AND.
• 1 -, то есть универсум без чего-либо, означает NOT.

Таким образом, выражение можно переписать:

```
(M AND N AND (W OR T)) OR (F AND N AND (NOT W)) OR B
```

Это почти дословно совпадает с тем, что вы сказали. Обратите внимание, как скобки проясняют ваши намерения. Вам нужна кошка из одного из трёх классов:

```
(M AND N AND (W OR T))
OR
(F AND N AND (NOT W))
OR
B
```

Имея эту формулу, продавец может выполнить так называемую *булеву проверку*. Здесь используется другой вариант булевой алгебры, в котором буквы обозначают свойства, характеристики или признаки кошек и могут принимать значения 0 или 1. Число 1 означает «да», «истина»: конкретная кошка удовлетворяет критерию. Число 0 означает «нет», «ложь»: кошка критерию не удовлетворяет.

Сначала продавец приносит нестерилизованного светло-коричневого кота. Вот выражение, описывающее подходящих кошек:

```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

А вот как оно выглядит после подстановки нулей и единиц:

```
(1 × 0 × (0 + 1)) + (0 × 0 × (1 - 0)) + 0
```

Обратите внимание: значение 1 получили только M и T, поскольку перед нами кот светло-коричневого окраса.

Теперь выражение нужно упростить. Если результат равен 1, кошка удовлетворяет критериям; если 0 - не удовлетворяет. Упрощая выражение, помните, что в действительности мы не складываем и не умножаем, хотя в целом можем делать вид, будто выполняем именно эти действия. Большинство тех же правил действует, когда + означает OR, а × означает AND. (В современных текстах для AND и OR иногда применяют символы ∧ и ∨ вместо × и +. Но, возможно, именно здесь знаки + и × упрощают работу, поскольку правила напоминают обычную алгебру.)

Когда знак × означает AND, возможны следующие результаты:

```
0 × 0 = 0
0 × 1 = 0
1 × 0 = 0
1 × 1 = 1
```

Иными словами, результат равен 1 только тогда, когда и левый операнд AND, и правый операнд равны 1. Эта операция выполняется точно так же, как обычное умножение, и её можно свести в небольшую таблицу. Операция указана в левом верхнем углу, а возможные комбинации операторов - в верхней строке и левом столбце:

AND	**0**	**1**
0	0	0
1	0	1

Когда знак + означает OR, возможны следующие результаты:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 1
```

Результат равен 1, если левый OR правый операнд равен 1. Эта операция даёт результаты, очень похожие на обычное сложение, за исключением того, что в данном случае 1 + 1 равно 1. (Если кошка светло-коричневая или кошка светло-коричневая, значит, она светло-коричневая.) Операцию OR можно свести в ещё одну небольшую таблицу:

OR	**0**	**1**
0	0	1
1	1	1

Теперь мы готовы воспользоваться таблицами и вычислить результат выражения:

```
(1 × 0 × 1) + (0 × 0 × 1) + 0 = 0 + 0 + 0 = 0
```

Результат 0 означает «нет», «ложь»: этот котёнок нам не подходит.

Затем продавец приносит стерилизованную белую кошку. Исходное выражение имело вид:

```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

Снова подставим нули и единицы:

```
(0 × 1 × (1 + 0)) + (1 × 1 × (1 - 1)) + 0
```

И упростим:

```
(0 × 1 × 1) + (1 × 1 × 0) + 0 = 0 + 0 + 0 = 0
```

Ещё одного бедного котёнка приходится отвергнуть.

Следующей продавец приносит стерилизованную серую кошку. (Серый считается «другим» окрасом: не белым, не чёрным и не светло-коричневым.) Получаем выражение:

```
(0 × 1 × (0 + 0)) + (1 × 1 × (1 - 0)) + 0
```

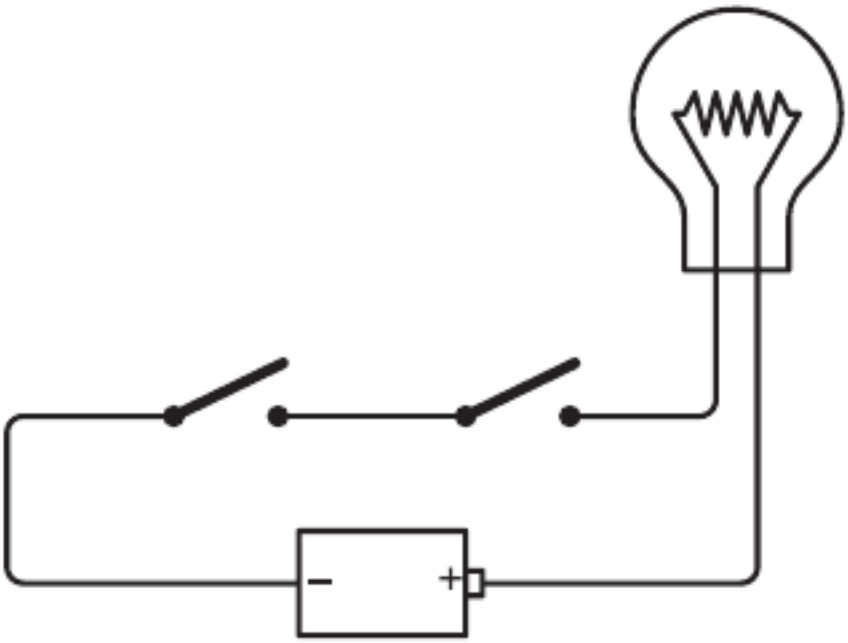
Теперь упростим его:

```
(0 × 1 × 0) + (1 × 1 × 1) + 0 = 0 + 1 + 0 = 1
```

Окончательный результат 1 означает «да», «истина»: котёнок нашёл дом. (И к тому же он оказался самым милым!)

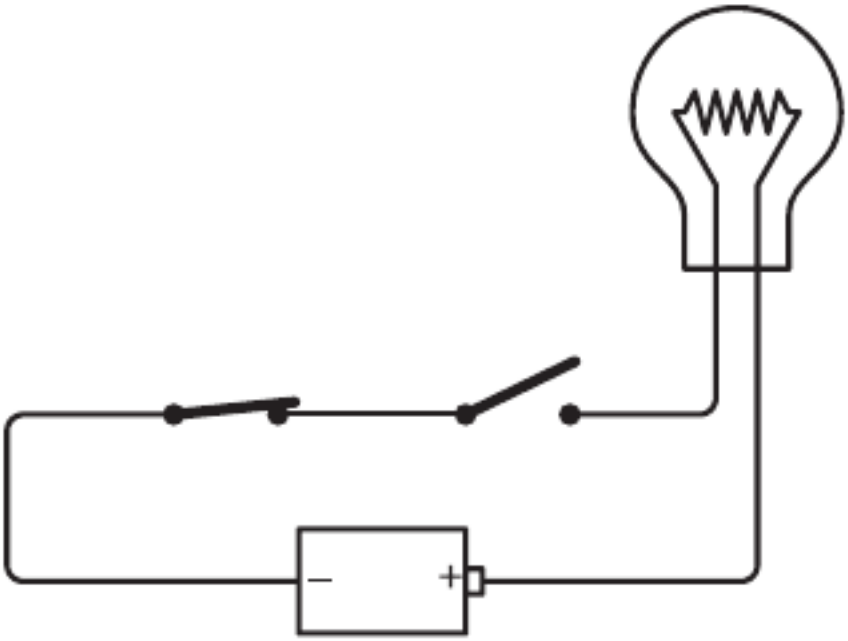
Позднее тем же вечером, когда котёнок свернулся клубком и спит у вас на коленях, вы задумываетесь, нельзя ли соединить несколько выключателей и лампочку так, чтобы определять, удовлетворяют ли конкретные котята вашим критериям. (Да, вы странный ребёнок.) Вы ещё не осознаёте, что стоите на пороге решающего концептуального прорыва. Вам предстоят опыты, которые объединят алгебру Джорджа Буля с электрическими цепями и тем самым сделают возможными проектирование и создание цифровых компьютеров. Но пусть это вас не пугает.

В начале опыта вы подключаете лампочку и батарейку обычным способом, но вместо одного выключателя используете два:

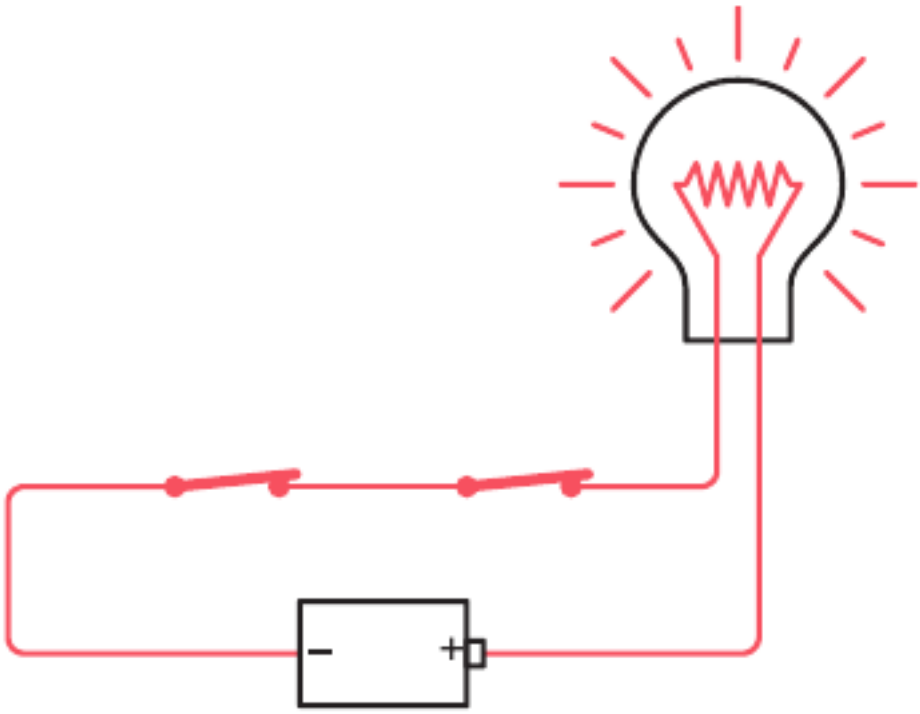


Значок земного шара на внешнем поле страницы сообщает, что интерактивная версия схемы доступна на сайте CodeHiddenLanguage.com.

О соединённых таким образом выключателях - одном непосредственно за другим - говорят, что они включены *последовательно*. Если замкнуть левый выключатель, ничего не произойдёт:



Аналогично, если оставить левый выключатель разомкнутым и замкнуть правый, ничего не произойдёт. Лампочка загорается лишь тогда, когда замкнуты и левый, и правый выключатели:



Ключевое слово здесь - *и*. Чтобы по цепи пошёл ток, должны быть замкнуты и левый, *и* правый выключатели.

Эта цепь выполняет небольшое логическое упражнение. По существу, лампочка отвечает на вопрос: «Замкнуты ли оба выключателя?» Работу цепи можно свести в таблицу:

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

Если рассматривать выключатели и лампочку как булевы операторы, этим состояниям можно назначить числа 0 и 1. Ноль может означать «выключатель разомкнут», а единица - «выключатель замкнут». У лампочки два состояния: 0 может означать «лампочка не горит», а 1 - «лампочка горит». Теперь просто перепишем таблицу:

Левый выключатель	**Правый выключатель**	**Лампочка**
0	0	0
0	1	0
1	0	0
1	1	1

Обратите внимание: если поменять левый и правый выключатели местами, результаты не изменятся. В действительности нам необязательно различать выключатели. Поэтому таблицу можно переписать в форме, напоминающей показанные ранее таблицы AND и OR:

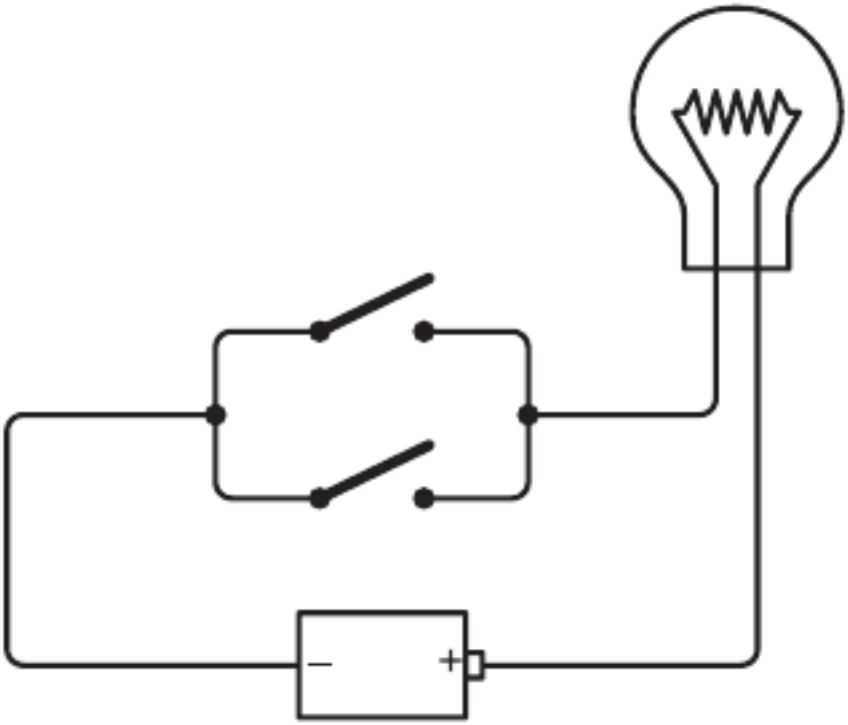
Последовательные выключатели	**0**	**1**
0	0	0
1	0	1

И действительно, это та же таблица, что и для AND. Сравните:

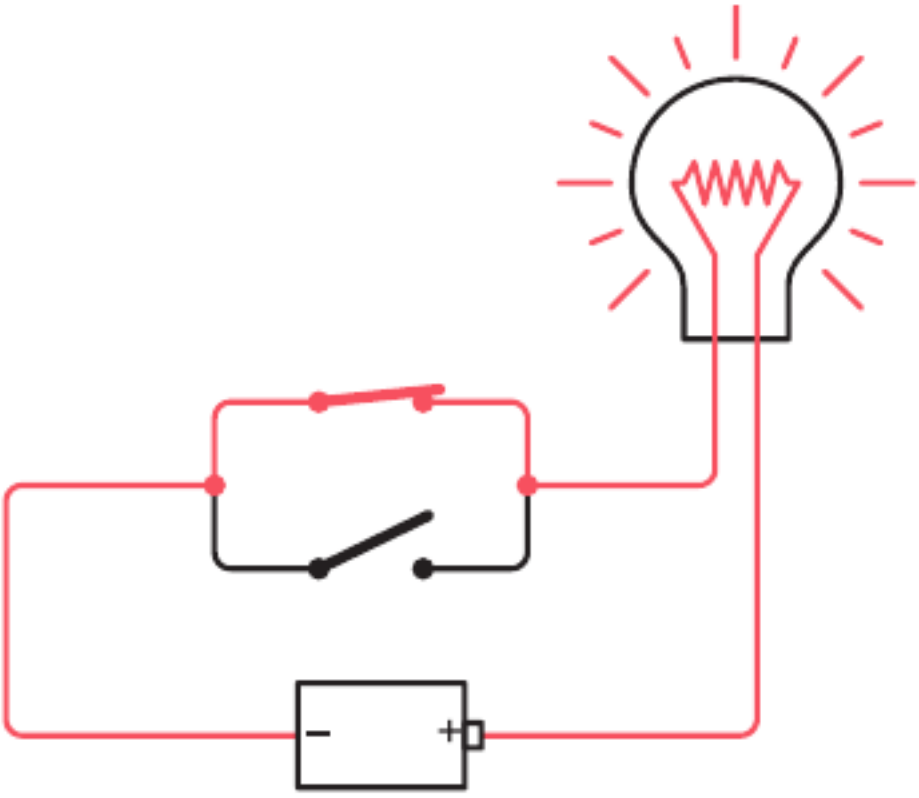
AND	**0**	**1**
0	0	0
1	0	1

Эта простая цепь фактически выполняет операцию AND булевой алгебры.

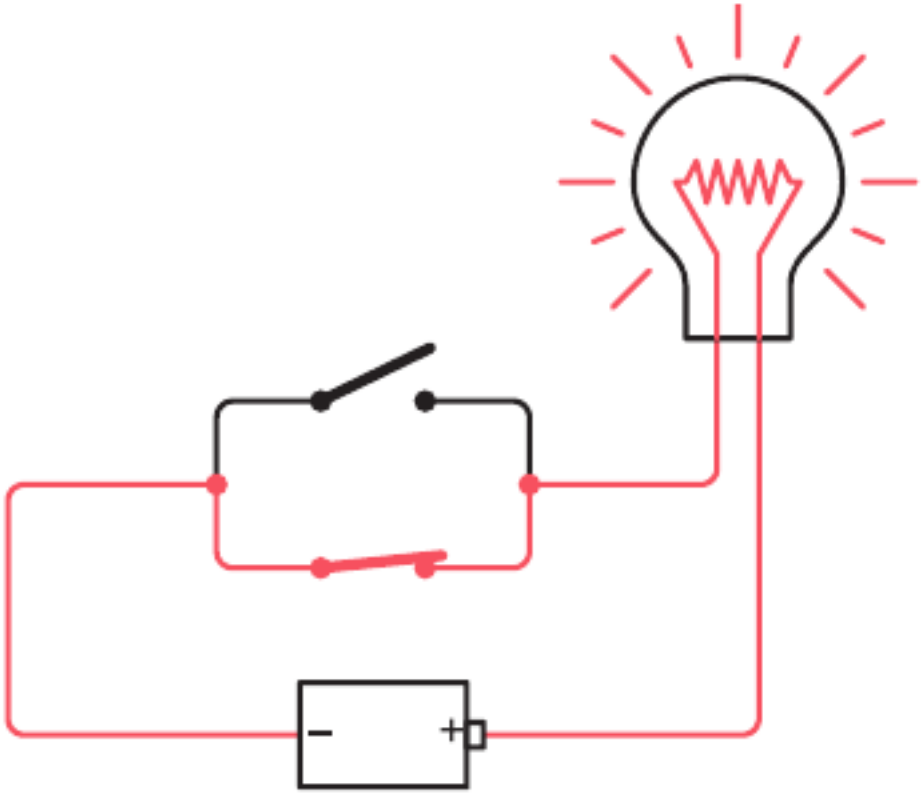
Теперь попробуем соединить два выключателя несколько иначе:



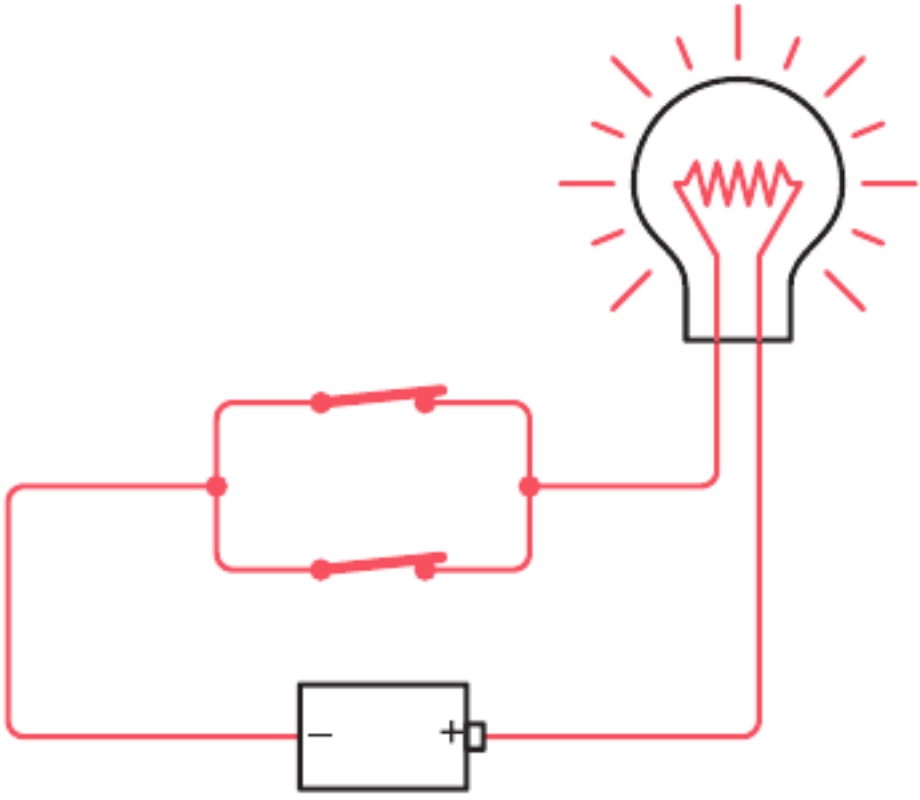
Такие выключатели называют соединёнными *параллельно*. Отличие от предыдущего соединения состоит в том, что теперь лампочка загорится, если замкнуть верхний выключатель:



или нижний:



или оба:



Лампочка загорается, если замкнут верхний *или* нижний выключатель. Ключевое слово здесь - *или*.

И снова цепь выполняет логическое упражнение. Лампочка отвечает на вопрос: «Замкнут ли какой-либо из выключателей?» Работа цепи сведена в следующую таблицу:

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

Если снова использовать 0 для разомкнутого выключателя или негорящей лампочки, а 1 для замкнутого выключателя или горящей лампочки, таблицу можно переписать так:

Левый выключатель	**Правый выключатель**	**Лампочка**
0	0	0
0	1	1
1	0	1
1	1	1

Перестановка двух выключателей снова ни на что не влияет, поэтому таблицу можно представить и в следующем виде:

Параллельные выключатели	**0**	**1**
0	0	1
1	1	1

И вы уже догадались, что она совпадает с таблицей булевой операции OR:

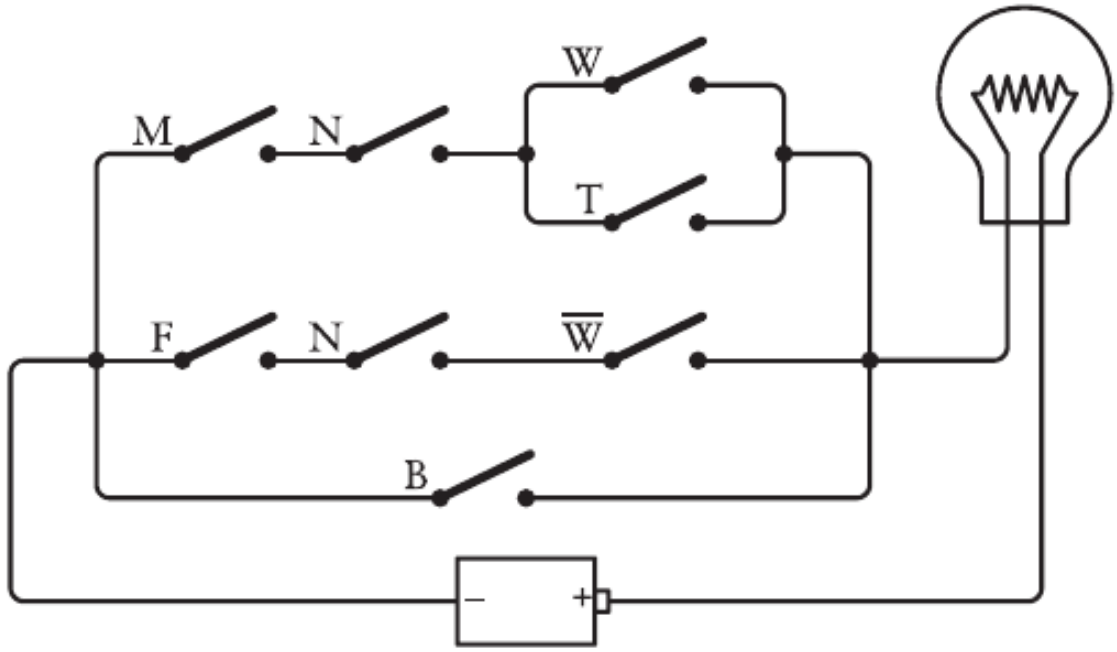
OR	**0**	**1**
0	0	1
1	1	1

Следовательно, два параллельно соединённых выключателя выполняют эквивалент булевой операции OR.

Войдя в зоомагазин, вы сказали продавцу: «Мне нужен стерилизованный кот белого или светло-коричневого окраса; либо стерилизованная кошка любого окраса, кроме белого; либо я возьму любую имеющуюся у вас кошку, если она чёрная», после чего продавец составил выражение:

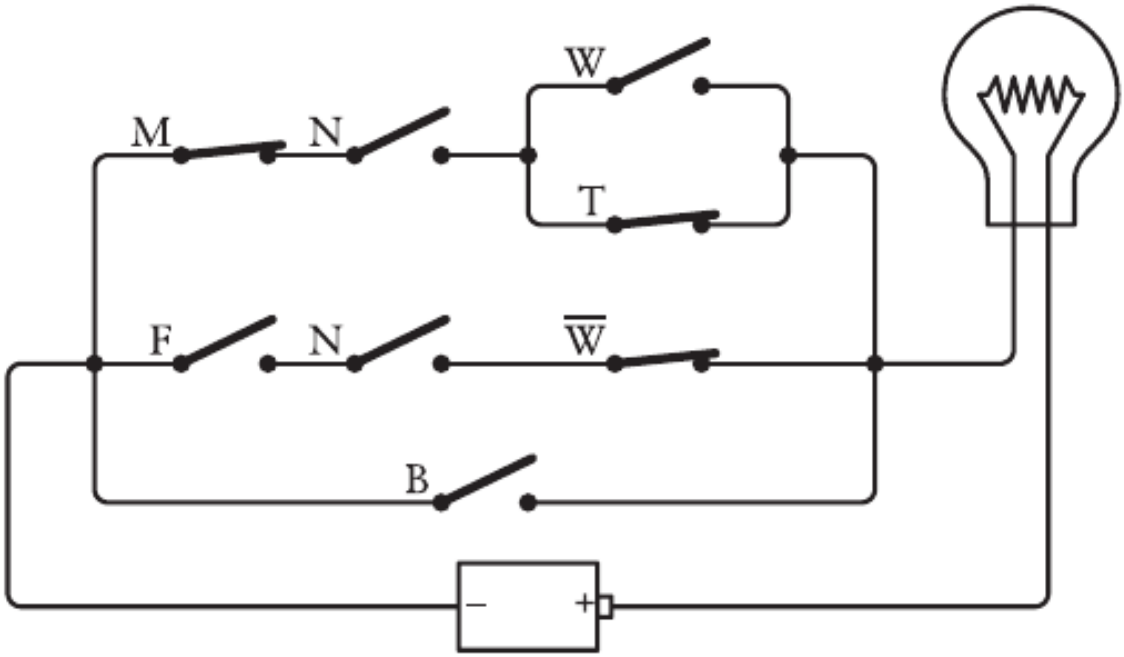
```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

Теперь вы знаете, что два последовательно соединённых выключателя выполняют логическую операцию AND, обозначаемую знаком ×, а два параллельно соединённых выключателя - логическую операцию OR, обозначаемую знаком +. Поэтому восемь выключателей можно соединить так:

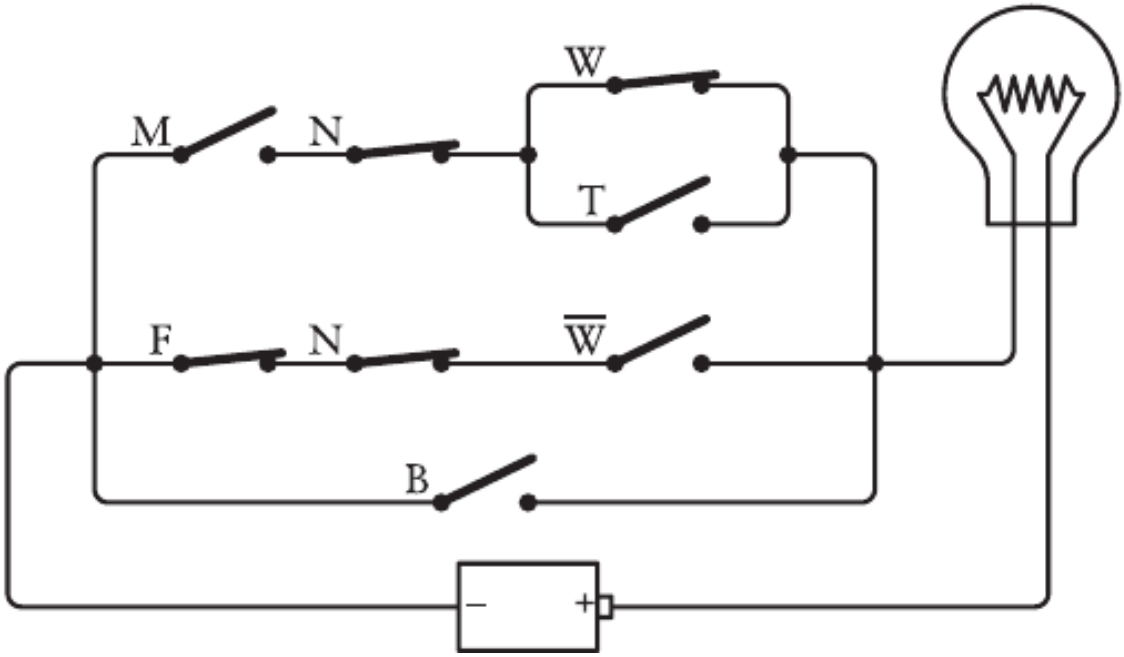


Каждый выключатель в этой цепи помечен буквой - той же, которая используется в булевом выражении. $\overline{W}$ означает NOT W и является другим способом записать 1 - W. Действительно, если пройти по схеме слева направо, начиная сверху и двигаясь вниз, буквы встретятся в том же порядке, в каком они стоят в выражении. Каждому знаку × в выражении соответствует место цепи, где два выключателя или две группы выключателей соединены последовательно. Каждому знаку + соответствует место, где два выключателя или две группы выключателей соединены параллельно.

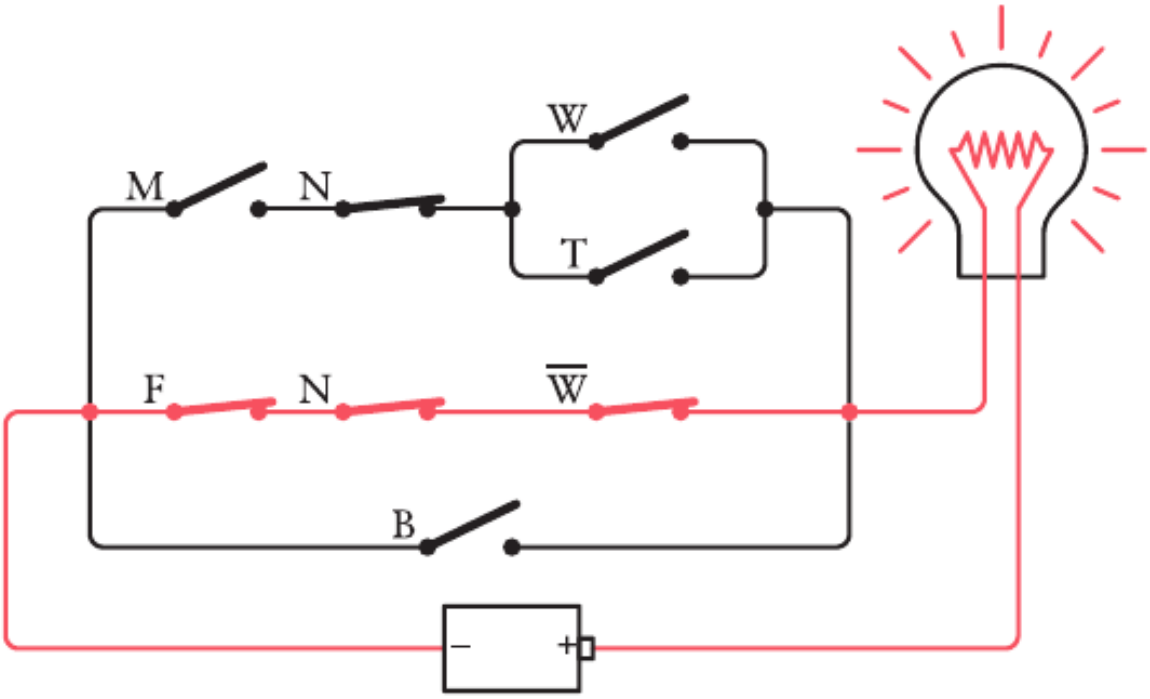
Как вы помните, сначала продавец принёс нестерилизованного светло-коричневого кота. Замкнём соответствующие выключатели:



Хотя выключатели M, T и NOT W замкнуты, полной цепи, способной зажечь лампочку, не получается. Затем продавец принёс стерилизованную белую кошку:



И снова замкнуты не те выключатели, которые нужны для завершения цепи. Но наконец продавец принёс стерилизованную серую кошку:



Этого достаточно, чтобы замкнуть цепь, зажечь лампочку и показать, что котёнок удовлетворяет всем вашим критериям.

Джордж Буль никогда не соединял такую цепь. Ему не довелось испытать восторг от вида булева выражения, воплощённого в выключателях, проводах и лампочках. Одним из препятствий,

разумеется, было то, что лампу накаливания изобрели лишь через 15 лет после смерти Буля. Однако телеграф появился за десять лет до публикации булевых *«Законов мышления»*, а важной частью телеграфной системы являлось простое устройство, способное выполнять логические операции гораздо проворнее обычных выключателей.

Глава 7. Телеграфы и реле

Сэмюэл Финли Бриз Морзе родился в 1791 году в Чарлстауне, штат Массачусетс, городе, возле которого произошла битва при Банкер-Хилле и который теперь является северо-восточной частью Бостона. В год рождения Морзе Конституция Соединённых Штатов действовала всего два года, а Джордж Вашингтон отбывал первый президентский срок. Россией правила Екатерина Великая. Во Франции всё ещё продолжалась революция, и через два года Людовика XVI и Марию-Антуанетту отправят на гильотину. В 1791 году Моцарт завершил *«Волшебную флейту»*, свою последнюю оперу, и умер в том же году в возрасте 35 лет, но двадцатилетний Бетховен уже начинал привлекать внимание.

Морзе получил образование в Йельском университете и изучал искусство в Лондоне. Он стал успешным художником-портретистом. Его картина *«Генерал Лафайет»* (1825) висит в мэрии Нью-Йорка. Гораздо более личной стала последняя картина Морзе - выставленный в Метрополитен-музее портрет его дочери Сьюзен под названием *«Муза»*.

Морзе также был одним из первых любителей фотографии. Он научился делать дагеротипные снимки у самого Луи Дагера и создал несколько первых дагеротипов в Америке. В 1840 году он обучил этому процессу семнадцатилетнего Мэтью Брэди, который вместе с коллегами впоследствии создал наиболее запоминающиеся фотографии Гражданской войны, Авраама Линкольна и самого Сэмюэла Морзе.

Фото: ullstein bild Dtl/Getty Images.

Но всё это лишь примечания к разнообразной карьере. Сегодня Сэмюэла Морзе помнят прежде всего как изобретателя телеграфа и носящего его имя кода.

Привычная нам мгновенная всемирная связь появилась относительно недавно. В начале XIX века можно было общаться мгновенно и можно было общаться на больших расстояниях, но нельзя было делать и то и другое одновременно. Мгновенное общение ограничивалось дальностью человеческого голоса, поскольку усилителей ещё не существовало, или пределами видимости,

возможно расширенными телескопом. Пересылка писем на большие расстояния занимала время и требовала лошадей, поездов или кораблей.

За несколько десятилетий до изобретения Морзе предпринималось множество попыток ускорить дальнюю связь. В технически простых способах использовалась цепочка людей, стоявших на холмах и размахивавших флагами по кодовым схемам, известным как *семафор*. В более сложных решениях применялись большие сооружения с подвижными рычагами, которые, по существу, делали то же самое, что люди с флагами.

Идея телеграфа, буквально «письма на расстоянии», определённо витала в воздухе в начале XIX века, и другие изобретатели пробовали осуществить её ещё до того, как Сэмюэл Морзе начал эксперименты в 1832 году. В принципе идея электрического телеграфа была проста: вы делаете что-либо на одном конце провода, и это вызывает некоторое событие на другом. Именно так мы поступили в главе 5, создавая фонарик дальней связи. Однако Морзе не мог использовать лампочку как сигнальное устройство, поскольку пригодную для практики лампу изобрели лишь в 1879 году. Вместо неё Морзе воспользовался явлением *электромагнетизма*.

Первое систематическое исследование связи электричества и магнетизма приписывают датскому физику Хансу Кристиану Эрстеду. В опубликованной им в 1820 году статье было показано, как электрический ток отклоняет намагниченную стрелку компаса. После этого явление заняло лучшие научные умы XIX века, в том числе Майкла Фарадея и Джеймса Клерка Максвелла, чей труд *«Трактат об электричестве и магнетизме»* 1873 года остаётся классикой математической физики. Но к тому времени изобретательные новаторы, подобные Сэмюэлу Морзе, уже давно использовали электромагнетизм в своих хитроумных устройствах. Если взять железный стержень, обмотать его парой сотен витков тонкого изолированного провода и пропустить по проводу ток, стержень станет магнитом и начнёт притягивать другие предметы из железа и стали. После отключения тока стержень теряет магнитные свойства:

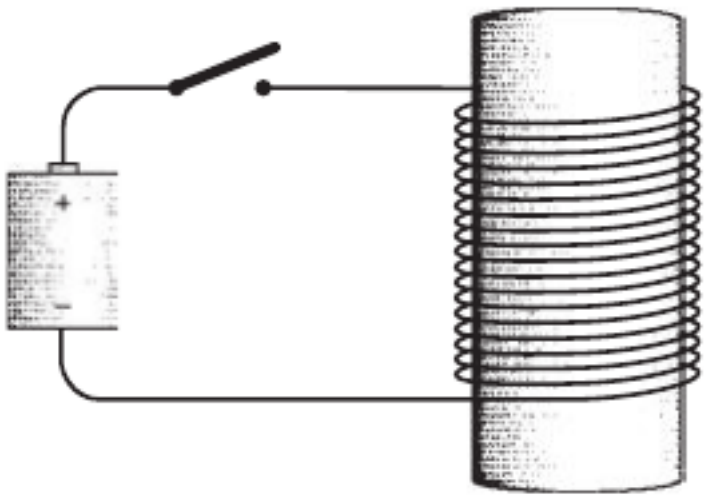


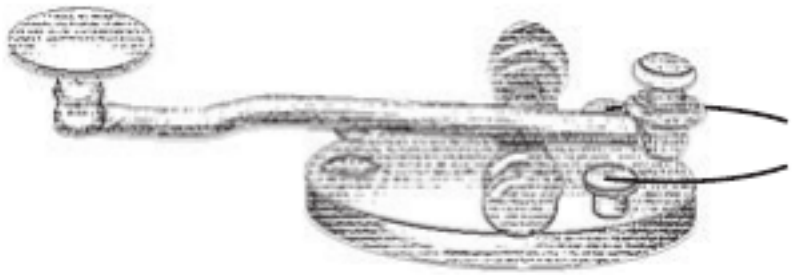
Схема может показаться коротким замыканием, но провод вокруг железного стержня обычно очень тонок, а его длины достаточно для создания необходимого электрического сопротивления.

Электромагнит лежит в основе телеграфа. Включение и выключение переключателя на одном конце заставляет электромагнит что-либо делать на другом.

Первые телеграфы Морзе на самом деле были сложнее тех, которые получили распространение позднее. Морзе считал, что телеграфная система должна действительно что-то писать на бумаге или, как впоследствии сказали бы пользователи компьютеров, «выдавать твёрдую копию». Разумеется, это необязательно должны были быть слова, поскольку такая система оказалась бы слишком сложной. Но *что-нибудь* следовало записывать на бумаге: росчерки, точки и тире. Обратите внимание: Морзе оставался в плену парадигмы, требовавшей бумаги и чтения, подобно тому как Валентин Гаюи считал, что в книгах для слепых должны использоваться выпуклые буквы алфавита.

Хотя в 1836 году Сэмюэл Морзе уведомил патентное ведомство об изобретении работающего телеграфа, лишь в 1843 году ему удалось убедить Конгресс выделить средства на публичную демонстрацию устройства. Историческим днём стало 24 мая 1844 года, когда телеграфная линия между Вашингтоном, округ Колумбия, и Балтимором, штат Мэриленд, успешно передала библейскую фразу из Книги Чисел 23:23: «Вот что сотворил Бог!» Это не вопрос, а восклицание со смыслом: «Посмотрите, что сделал Бог!»

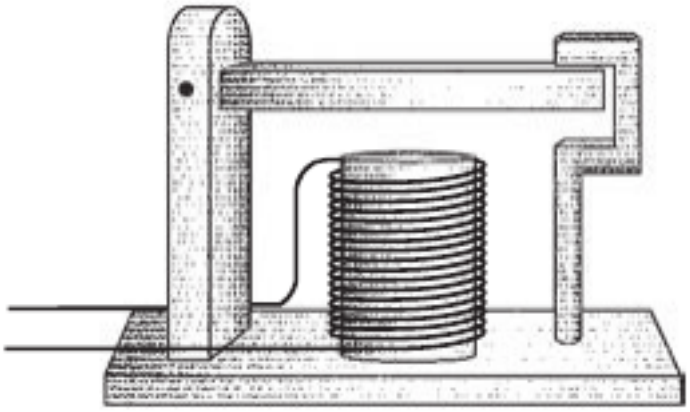
Традиционный телеграфный «ключ» для отправки сообщений выглядел так:



Несмотря на затейливый вид, это всего лишь переключатель, рассчитанный на максимальную скорость. При долгой работе удобнее всего держать рукоятку между большим, указательным и средним пальцами и постукивать ею вверх-вниз. Короткое нажатие создаёт точку азбуки Морзе, более долгое - тире.

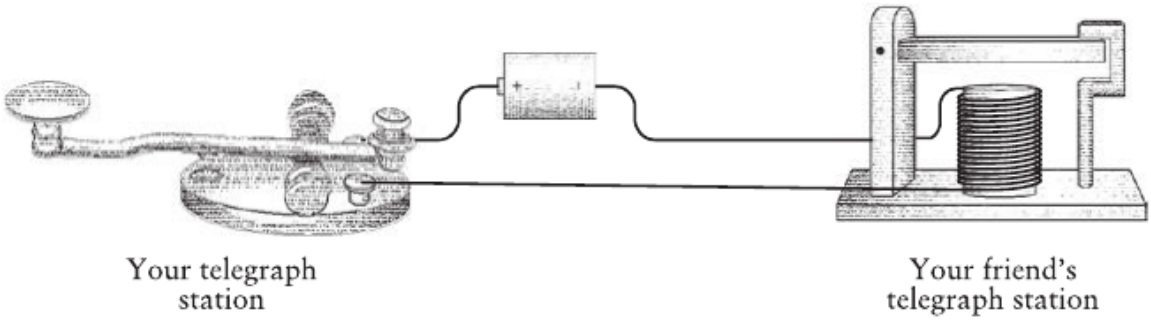
На другом конце провода находился приёмник, представлявший собой электромагнит, тянувший металлический рычаг. Изначально электромагнит управлял пером. Подпрыгивая вверх-вниз, перо рисовало точки и тире на бумажной ленте, которую медленно протягивала заведённая пружина. Человек, умевший читать азбуку Морзе, затем переписывал точки и тире буквами и словами.

Разумеется, люди - вид не только изобретательный, но и ленивый, поэтому телеграфисты вскоре обнаружили, что могут расшифровывать код, просто слушая, как перо подпрыгивает вверх-вниз. В конце концов механизм пера заменили традиционным звуковым телеграфным приёмником, или *sounder*, который выглядел примерно так:



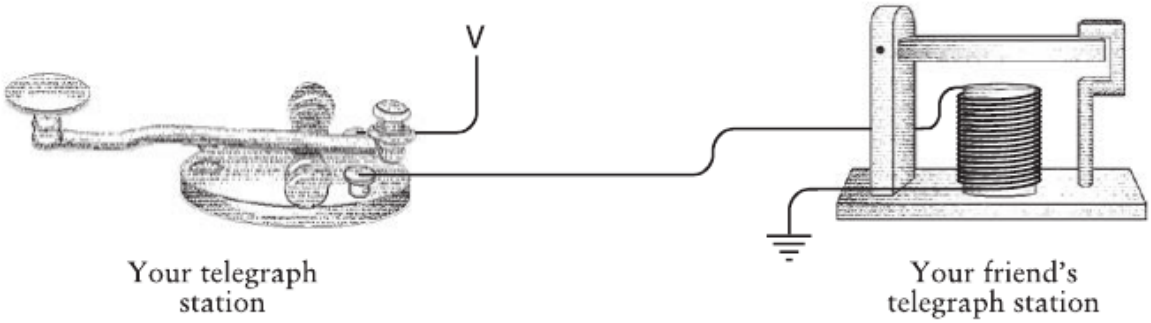
Верхнюю планку обычно удерживали горизонтально грузом или пружиной внутри левой вертикальной части, однако она могла поворачиваться. При нажатии телеграфного ключа электромагнит тянул поворотную планку вниз, и раздавался «щёлк». После отпускания ключа планка возвращалась пружиной в обычное положение со звуком «клац». Быстрое «щёлк-клац» означало точку, а более медленное «щёлк... клац» - тире.

Ключ, звуковой приёмник, батарейку и провода можно соединить так же, как ламповый телеграф в предыдущей главе:



Как мы выяснили, соединять две телеграфные станции двумя проводами необязательно. При достаточно высоком напряжении хватит одного провода, а вторую половину цепи обеспечит Земля.

Как и в главе 5, соединённую с землёй батарейку можно заменить прописной буквой V. Полная однонаправленная система будет выглядеть примерно так:



Для двунаправленной связи достаточно добавить ещё один ключ и звуковой приёмник. Это подобно тому, что мы делали ранее.

Изобретение телеграфа действительно отмечает начало современной связи. Впервые люди смогли передавать сообщения дальше, чем видел глаз или слышало ухо, и быстрее, чем могла скакать лошадь.

Тем более примечательно, что изобретение использовало двоичный код. В последующих видах электрической и беспроводной связи, включая телефон, радио и телевидение, от двоичных кодов отказались, но позднее они вновь появились в компьютерах, после чего множество других двоичных кодов распространилось практически во всех видах электронных носителей.

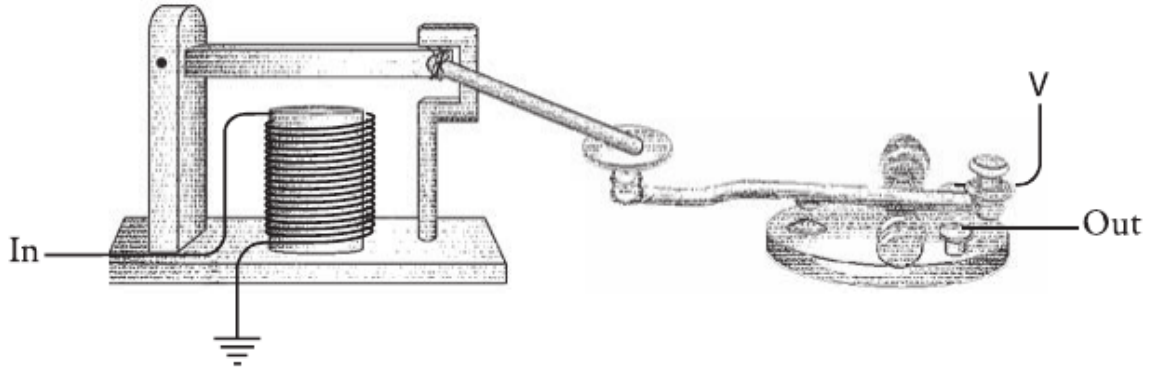
Телеграф Морзе одержал верх над другими конструкциями отчасти потому, что был устойчив к плохому состоянию линии. Если протянуть провод между ключом и звуковым приёмником, система обычно работала. Другие телеграфные системы были не столь неприхотливы. Однако, как говорилось в главе 5, чем длиннее провод, тем больше его сопротивление электрическому току. Это серьёзно препятствовало дальней телеграфной связи. Хотя некоторые телеграфные линии использовали напряжение до 300 вольт и работали на расстоянии 300 миль, бесконечно удлинять провода было нельзя.

Очевидное решение - система ретрансляции. Через каждые пару сотен миль человек со звуковым приёмником и ключом мог принимать сообщение и передавать его дальше.

Представьте, что телеграфная компания наняла вас для работы в такой системе. Вас посадили за стол в маленькой хижине где-то между Нью-Йорком и Калифорнией. Входящий через восточное окно провод подключён к звуковому приёмнику. Ваш телеграфный ключ соединён с батареей и проводом, выходящим через западное окно. Ваша задача - принимать сообщения из Нью-Йорка и передавать их дальше, чтобы в конце концов они дошли до Калифорнии. Такая же система ретранслирует сообщения из Калифорнии в Нью-Йорк.

Сначала вы предпочитаете полностью принять сообщение и лишь затем передать его. Вы записываете буквы, соответствующие щелчкам приёмника, а по окончании сообщения начинаете передавать его ключом. Со временем вы учитесь отправлять сообщение прямо во время приёма, не записывая его целиком. Так удаётся сэкономить время.

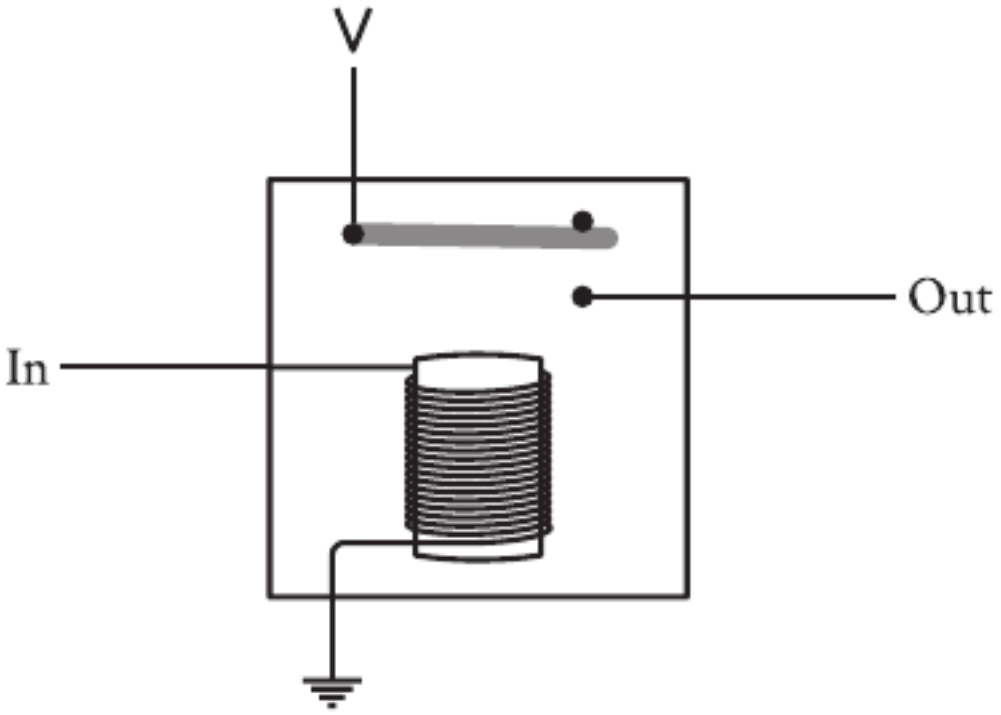
Однажды, передавая очередное сообщение, вы смотрите на подпрыгивающую планку приёмника, а затем на свои пальцы, нажимающие ключ вверх-вниз. Снова смотрите на приёмник, снова на ключ и понимаете, что планка приёмника движется вверх-вниз точно так же, как ключ. Тогда вы выходите наружу, подбираете небольшую деревянную планку и при помощи неё и бечёвки физически соединяете приёмник с ключом:



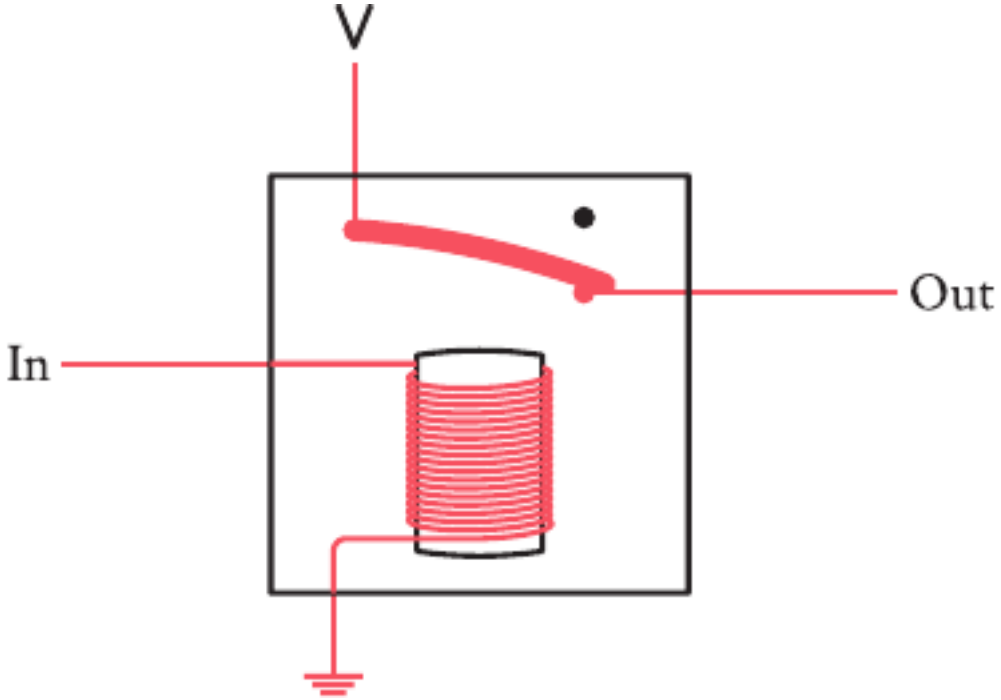
Теперь система работает самостоятельно, а вы можете взять выходной до конца дня и отправиться на рыбалку.

Это занятная фантазия, но в действительности Сэмюэл Морзе понял принцип такого устройства уже на раннем этапе. Изобретённое нами устройство называется *повторителем*, или *реле*. Реле похоже на звуковой приёмник: входящий ток питает электромагнит, который притягивает металлический рычаг. Однако рычаг используется как часть переключателя, соединяющего батарею с выходящим проводом. Так слабый входящий ток «усиливается», создавая более сильный выходящий.

На условной схеме реле выглядит так:



Когда входящий ток приводит электромагнит в действие, тот притягивает поворотную или гибкую металлическую планку, которая работает как переключатель и включает выходящий ток:



Слова In и Out описывают, как телеграфная линия входит через одно окно вашей хижины и выходит через противоположное, но одновременно служат удобными сокращёнными обозначениями *входа* (input) и *выхода* (output). Это электрические *сигналы*. Сигнал с меткой In вызывает изменение сигнала с меткой Out. Перед нами причина и следствие.

Реле - замечательное устройство. Несомненно, это переключатель, но включается и выключается он не рукой человека, а электрическим током. С такими устройствами можно совершать удивительные вещи. Из них действительно можно собрать значительную часть компьютера.

Да, реле - слишком прекрасное изобретение, чтобы оставлять его в музее телеграфии. Схватим одно, спрячем под курткой и быстро пройдём мимо охраны. В наших головах уже зреет идея.

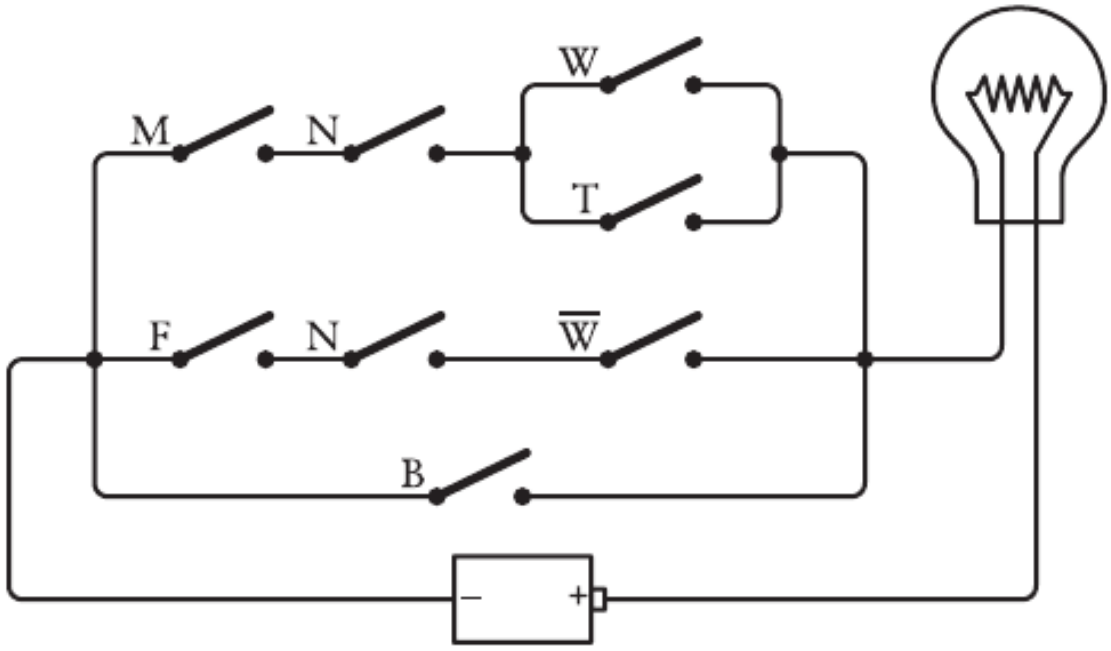
Глава 8. Реле и вентили

Если свести компьютер к самому существенному, он представляет собой синтез булевой алгебры и электричества. Важнейшие компоненты, воплощающие это соединение математики и аппаратуры, называются *логическими вентилями*. Эти вентили не так уж отличаются от привычных ворот, через которые проходят вода или люди. Логические вентили выполняют простые операции булевой логики, перекрывая или пропуская электрический ток.

Вспомните, как в главе 6 вы вошли в зоомагазин и смело объявили: «Мне нужен кастрированный кот, белый или рыжевато-коричневый; либо стерилизованная кошка любого окраса, кроме белого; либо я возьму любую кошку или кота, если животное чёрное». Эти критерии выражаются следующей булевой формулой:

```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

Эту формулу можно интерактивно реализовать в цепи из переключателей, батарейки и лампочки:



Такую цепь иногда называют *сетью*, хотя теперь это слово гораздо чаще относится к соединённым компьютерам, а не к совокупности простых переключателей.

Цепь содержит переключатели, часть которых соединена последовательно, а часть - параллельно. Последовательно соединённые переключатели выполняют логическую операцию AND, обозначенную в булевом выражении знаком x. Параллельно соединённые переключатели выполняют логическую операцию OR, которой соответствует знак +. Поскольку эта цепь эквивалентна булеву выражению, упрощение выражения позволяет упростить и цепь.

Вот выражение, задающее желательные признаки кошки:

```
(M × N × (W + T)) + (F × N × (1 - W)) + B
```

Попробуем его упростить. Согласно коммутативному закону переменные, соединённые знаками AND (x), можно переставить и переписать выражение так:

```
(N × M × (W + T)) + (N × F × (1 - W)) + B
```

Чтобы было понятнее, что я собираюсь сделать, определю два новых символа, X и Y:

```
X = M × (W + T)
Y = F × (1 - W)
```

Теперь выражение для желательной кошки можно записать так:

```
(N × X) + (N × Y) + B
```

Когда закончим, мы вернём на место выражения X и Y.

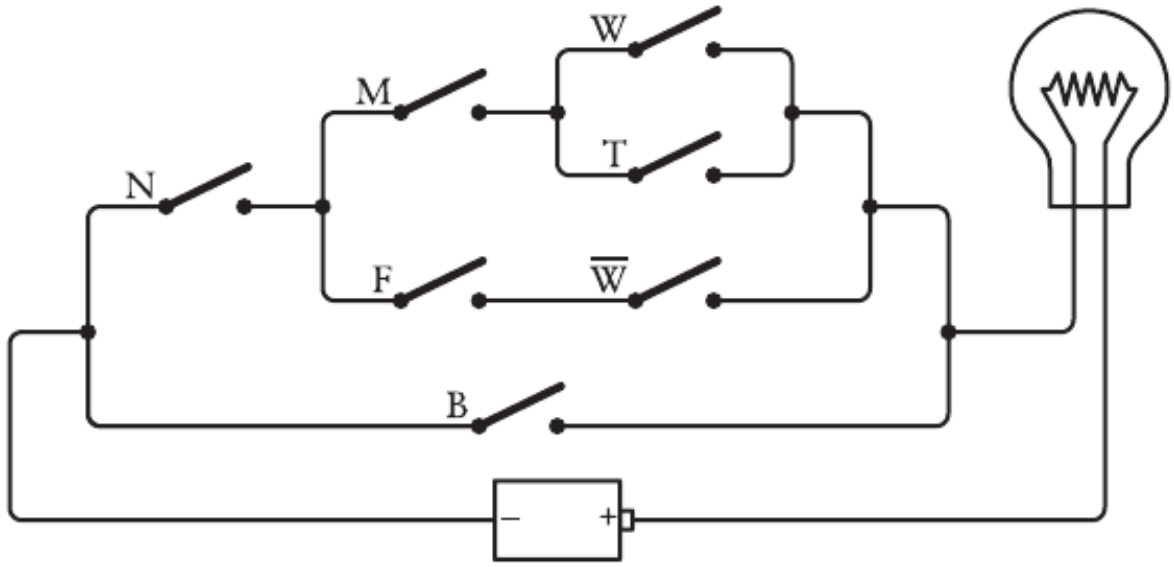
Обратите внимание, что переменная N встречается в выражении дважды. По дистрибутивному закону выражение можно переписать так, чтобы N осталась только одна:

```
(N × (X + Y)) + B
```

Теперь вернём выражения X и Y:

```
(N × ((M × (W + T)) + (F × (1 - W)))) + B
```

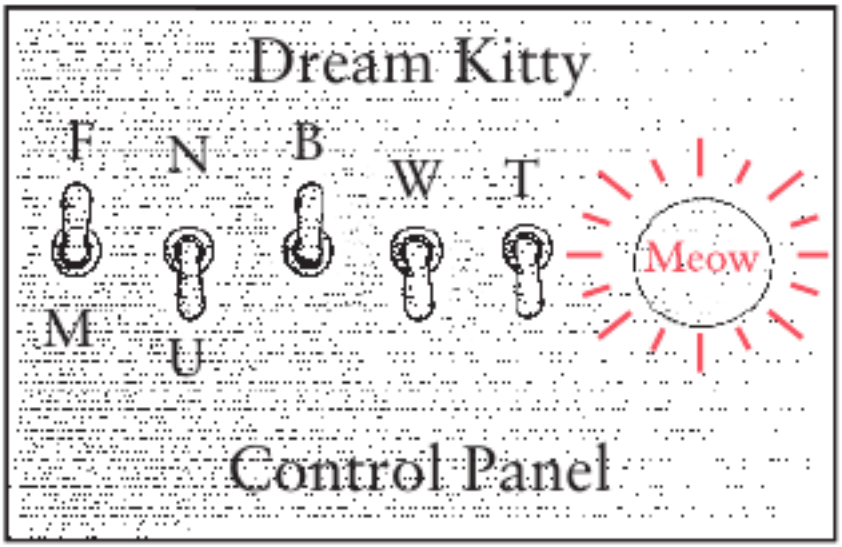
Из-за множества скобок это выражение едва ли выглядит упрощённым. Но в нём на одну переменную меньше, а значит, один переключатель можно убрать. Вот изменённая схема:



Действительно, увидеть эквивалентность этой сети предыдущей, вероятно, проще, чем проверить совпадение выражений!

Но переключателей в сети всё ещё слишком много. Для мужского и женского пола предусмотрены отдельные переключатели, хотя хватило бы одного: например, включённого, или замкнутого, для кошки и выключенного, или разомкнутого, для кота. Точно так же имеются отдельные переключатели для белого и небелого окраса.

Давайте прямо сейчас сделаем пульт для выбора кошки. Это всего лишь пять переключателей, очень похожих на настенные выключатели света, и лампочка, установленные на панели:



Переключатели включены, то есть замкнуты, когда подняты вверх, и выключены, то есть разомкнуты, когда опущены вниз. Первый переключатель выбирает кошку или кота, второй - кастрированное или некастрированное животное. Ещё три переключателя задают окрас: чёрный, белый и рыжевато-коричневый. В каждый момент должен быть включён только один из них либо ни одного, если выбирается кошка другого окраса.

В компьютерной терминологии панель переключателей представляет собой *устройство ввода*. Ввод - это информация, управляющая поведением цепи; в данном случае она описывает признаки идеальной кошки. *Устройство вывода* - лампочка. Она загорается, если положения переключателей описывают подходящую кошку. На показанном пульте выбрана нестерилизованная чёрная кошка. Она удовлетворяет нашим критериям, поэтому лампочка горит.

Теперь остаётся лишь разработать цепь, которая заставит этот пульт работать.

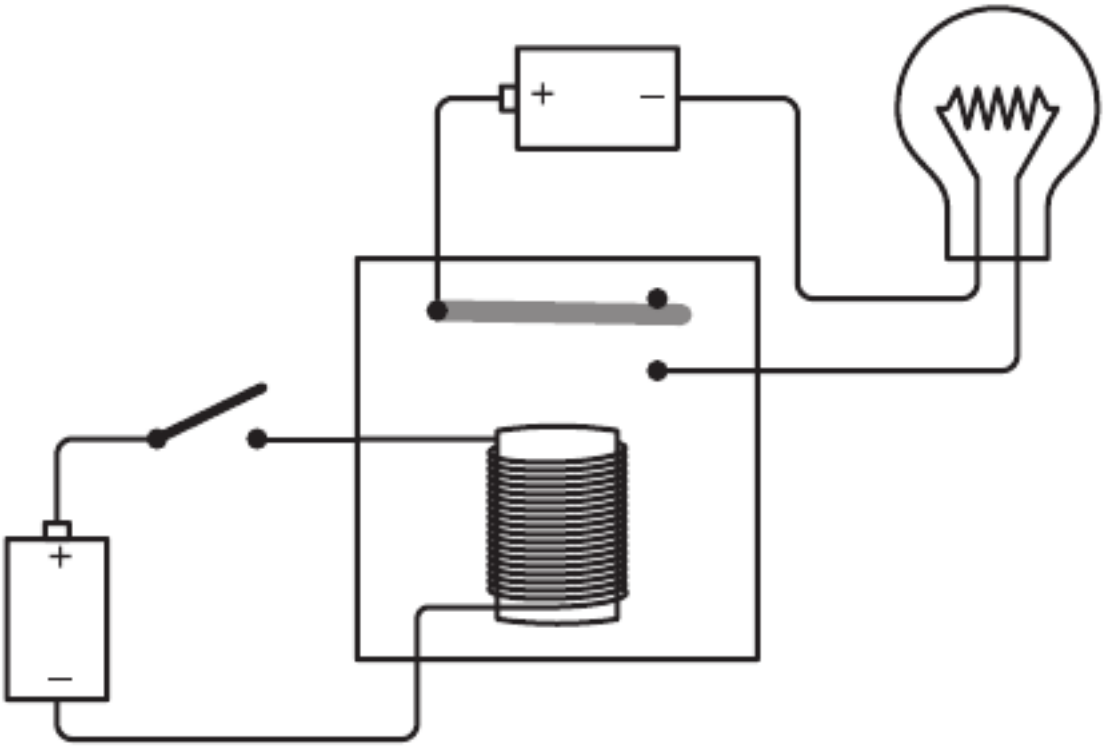
В предыдущей главе вы увидели, насколько важны были устройства, называемые реле, для работы телеграфной системы. На больших расстояниях провода между телеграфными станциями обладали очень высоким сопротивлением. Требовался способ принять слабый сигнал и заново передать такой же, но сильный. Реле делало это при помощи электромагнита, управляющего переключателем, фактически *усиливая* слабый сигнал и создавая сильный.

Сейчас нас не интересует применение реле для усиления слабого сигнала. Нас занимает лишь сама идея реле как переключателя, которым вместо пальцев управляет электричество. Хотя реле первоначально создавались для телеграфа, со временем они вошли в коммутационные цепи огромной телефонной сети, и именно там изобретательные инженеры-электрики яснее увидели их универсальность.

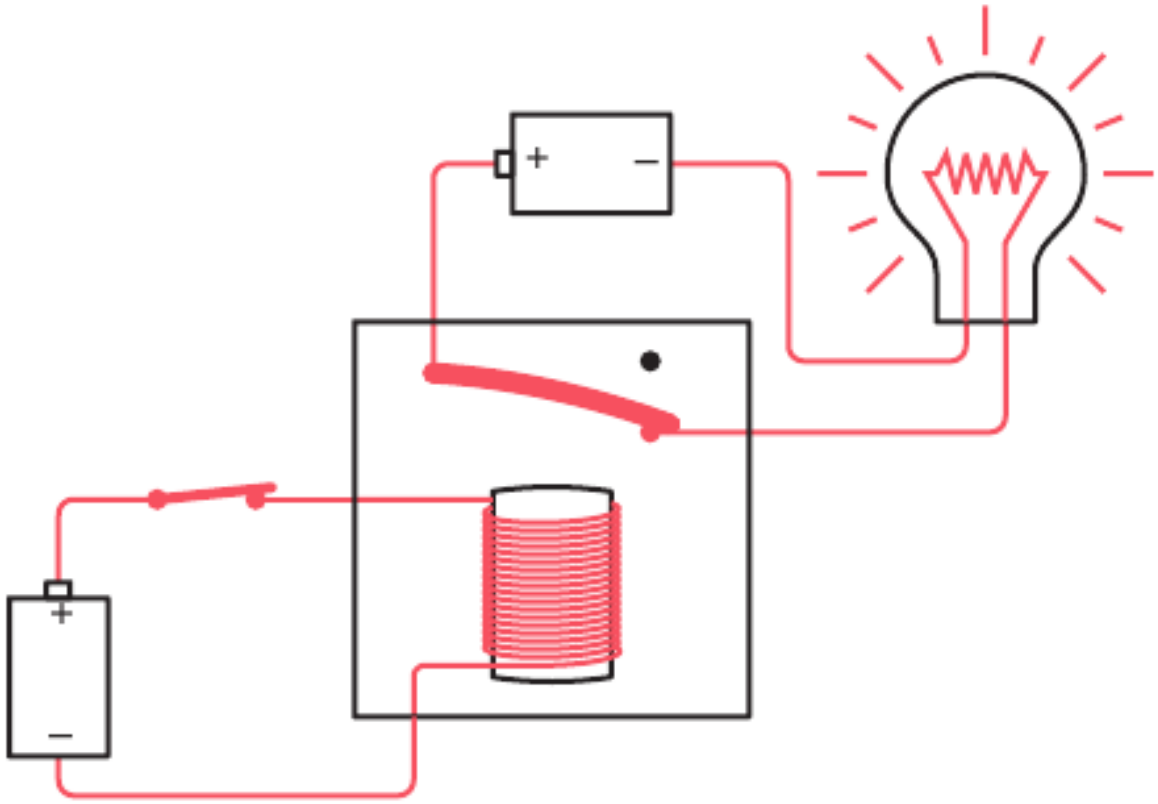
Подобно переключателям, реле можно последовательно и параллельно соединять в логические вентили, выполняющие простые логические задачи. Говоря, что эти логические вентили выполняют *простые* логические задачи, я имею в виду самые простые из возможных. Преимущество реле перед переключателями состоит в том, что включать и выключать реле могут другие реле, а не человеческие пальцы. Поэтому логические вентили можно объединять для решения более сложных задач: простых арифметических операций, а в конце концов - для обеспечения работы целых компьютеров.

Открытие возможности выполнять булевы операции при помощи реле обычно приписывают пионеру вычислительной техники Клоду Элвуду Шеннону (1916-2001), чья знаменитая магистерская диссертация в Массачусетском технологическом институте 1938 года называлась *«Символический анализ релейных и переключательных схем»*. Однако сходную эквивалентность на пару лет раньше описал японский инженер-электрик Акира Накасима.

Реле можно соединить с переключателем, лампочкой и двумя батарейками следующим образом:



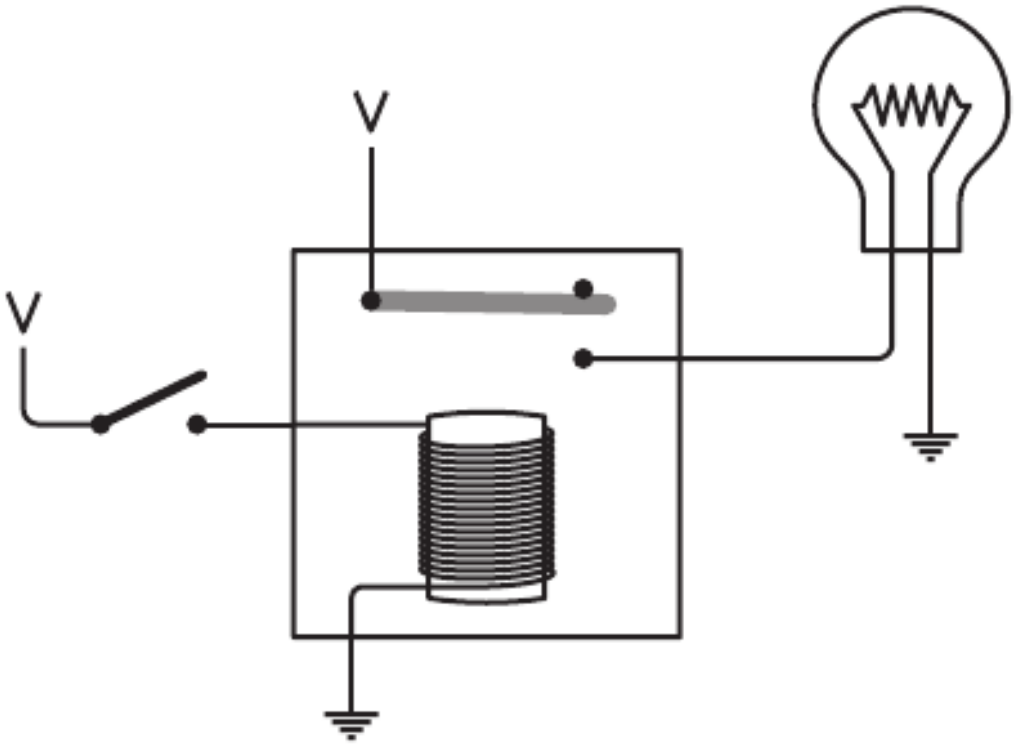
Переключатель слева разомкнут, и лампочка не горит. Когда вы замыкаете переключатель, левая батарейка заставляет ток пройти через множество витков провода вокруг железного стержня. Стержень становится магнитом и притягивает вниз гибкий или поворотный металлический контакт, который замыкает цепь и включает лампочку:



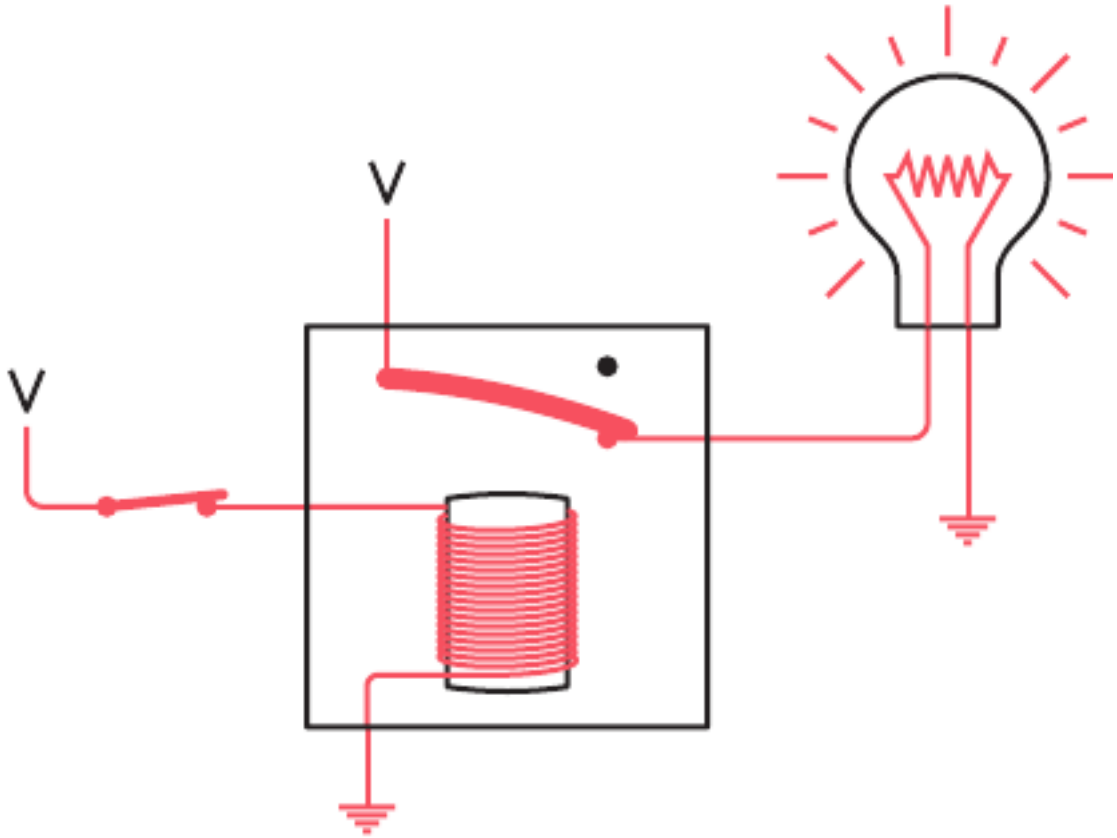
Когда электромагнит притягивает металлический контакт, говорят, что реле *сработало*. После размыкания переключателя железный стержень перестаёт быть магнитом, и металлический контакт возвращается в нормальное положение.

Чтобы зажечь лампочку, это кажется довольно окольным путём, и так оно и есть. Если бы нас интересовало только освещение лампочки, от реле можно было бы полностью отказаться. Но нас интересуют не лампочки. У нас гораздо более честолюбивая цель.

В этой главе я буду постоянно использовать реле, а после построения логических вентилей почти перестану, поэтому схему желательно упростить. Часть проводов можно убрать, обозначив батарейку заземлением и прописной буквой V, от слова *voltage*, то есть «напряжение», как мы делали в главах 5 и 7. В данном случае заземления обозначают лишь общее соединение; с физической землёй их соединять не требуется. Теперь реле выглядит так:

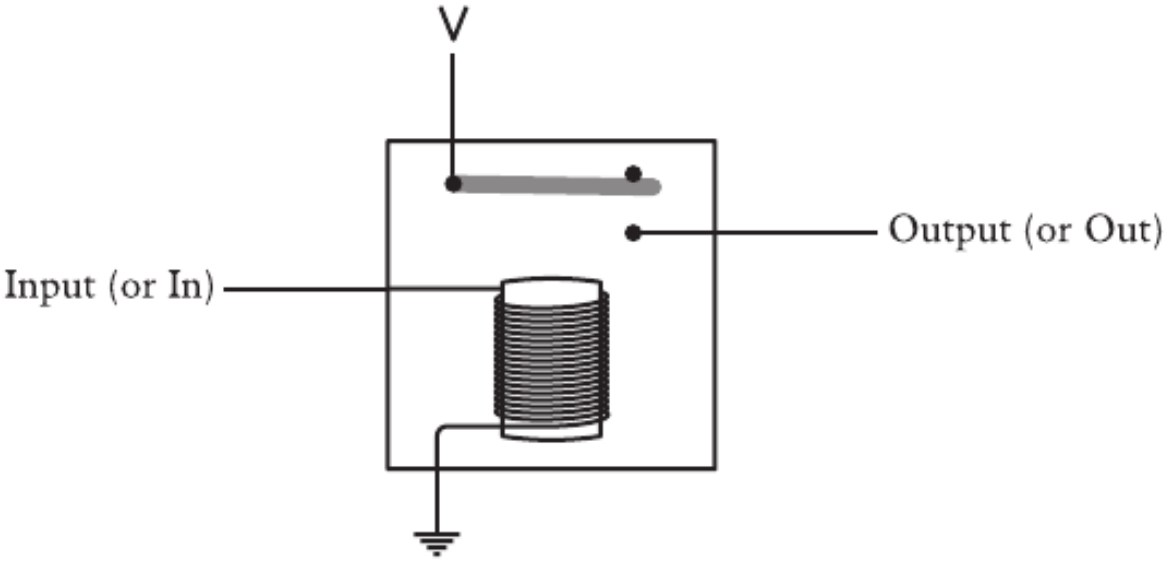


Когда переключатель замкнут, ток течёт от V к земле через обмотку электромагнита. Из-за этого электромагнит притягивает гибкий металлический контакт. Тот замыкает цепь между V, лампочкой и землёй. Лампочка загорается:



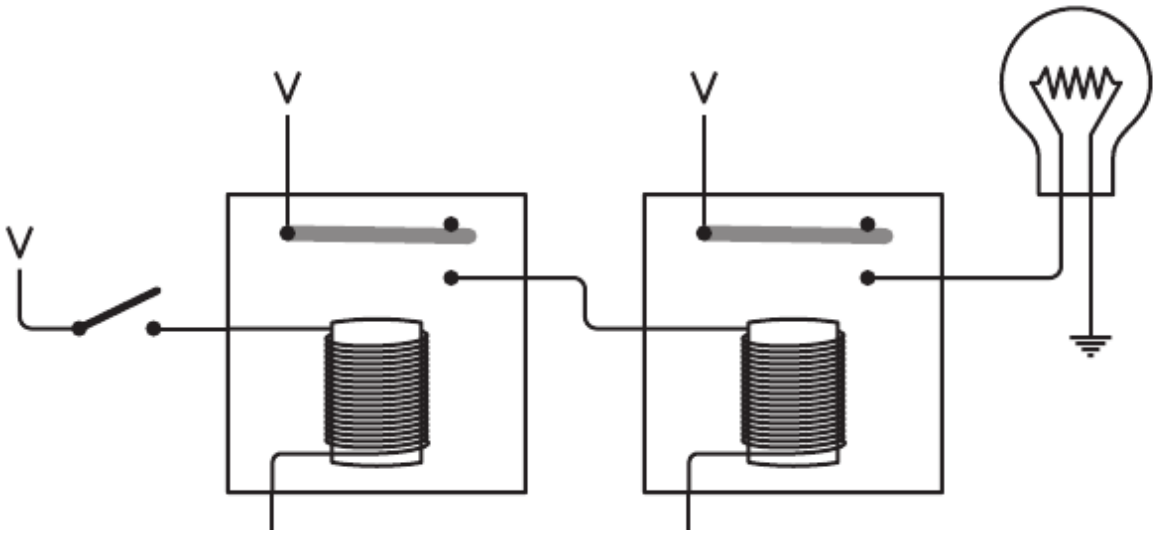
На этих схемах реле показаны два источника напряжения и два заземления, но во всех схемах этой главы все обозначения V можно соединить друг с другом, как и все обозначения земли.

В более абстрактном виде реле можно показать без переключателя и лампочки, но с обозначенными *входом* и *выходом*:

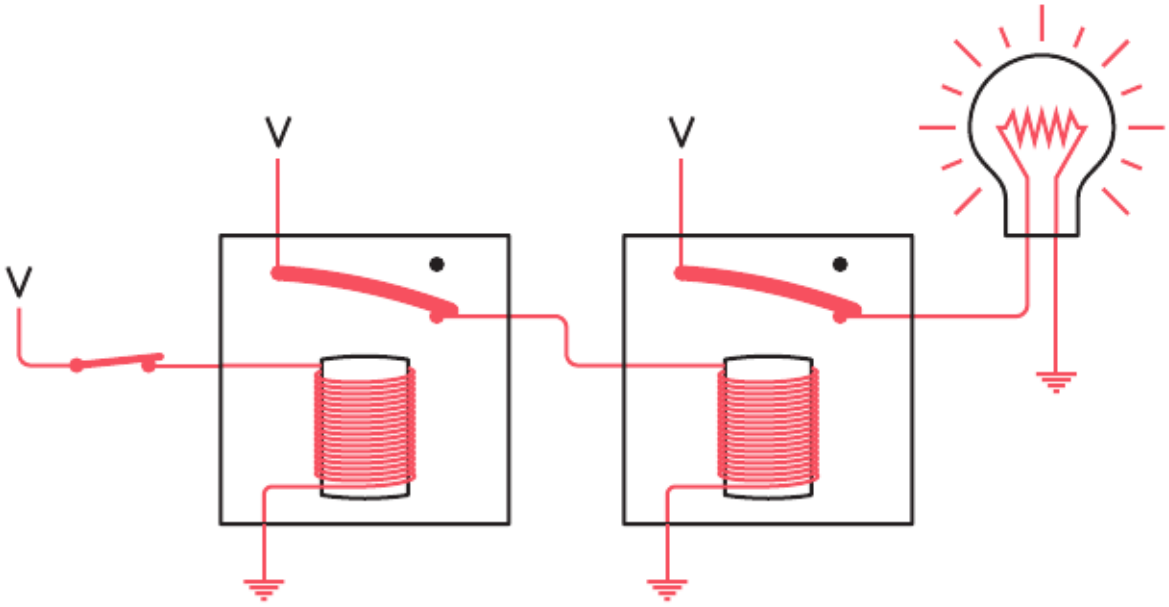


Если через вход течёт ток, например если переключатель соединяет вход с V, электромагнит срабатывает, и на выходе появляется напряжение.

Входом реле необязательно должен быть переключатель, а выходом - лампочка. Например, выход одного реле можно соединить со входом другого:

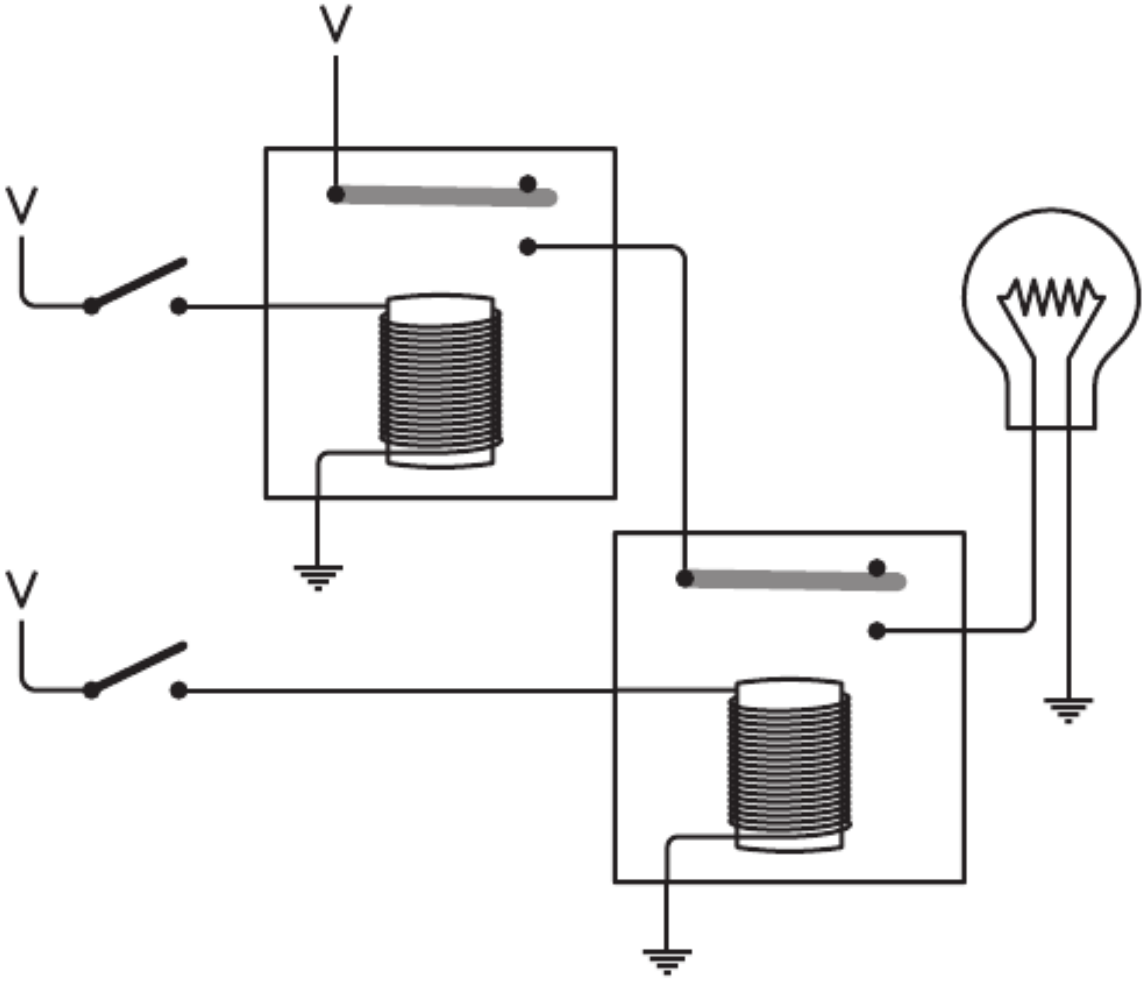


Говорят, что эти реле соединены *каскадно*. Когда вы замыкаете переключатель, срабатывает первое реле и подаёт напряжение на второе. Второе реле срабатывает, и лампочка загорается:

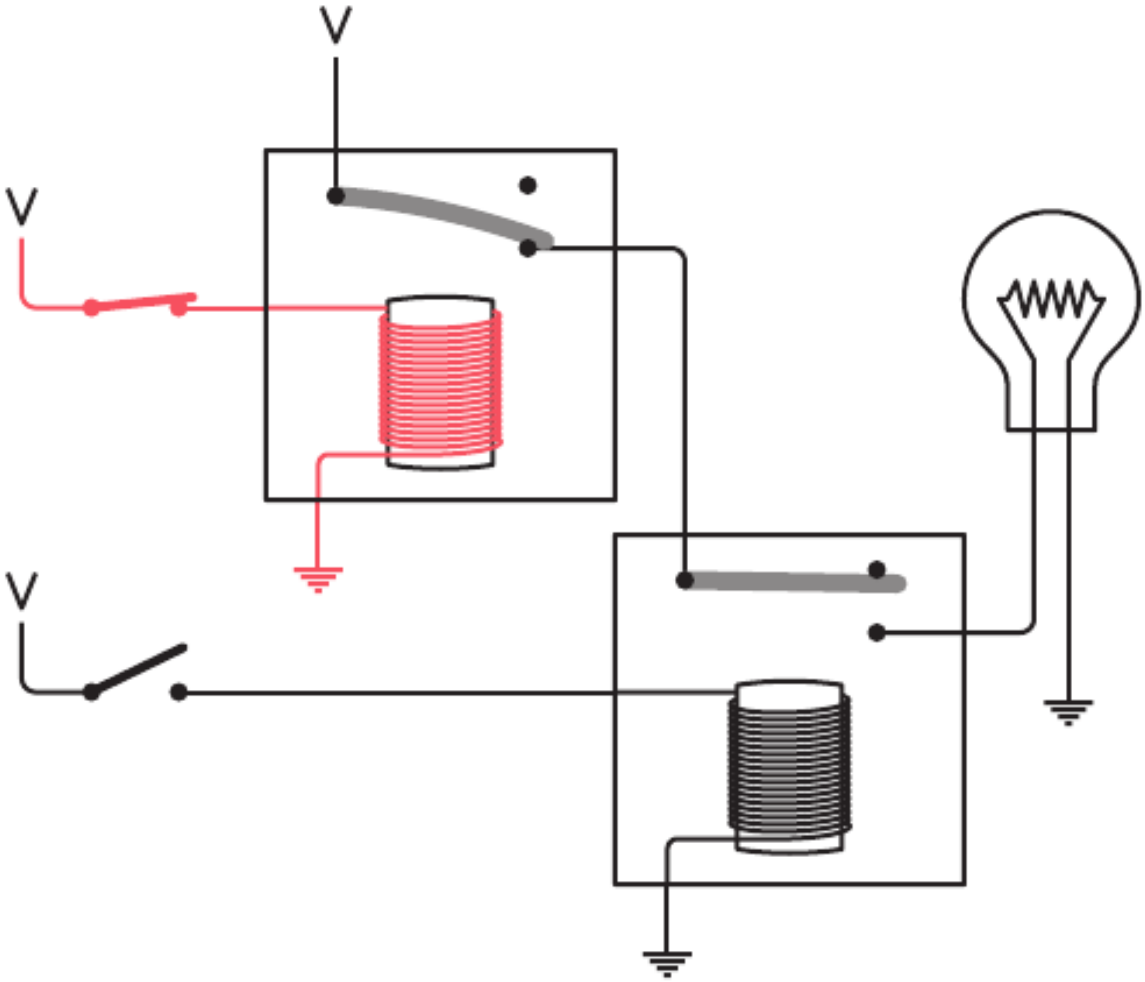


Соединение реле - ключ к построению логических вентилей.

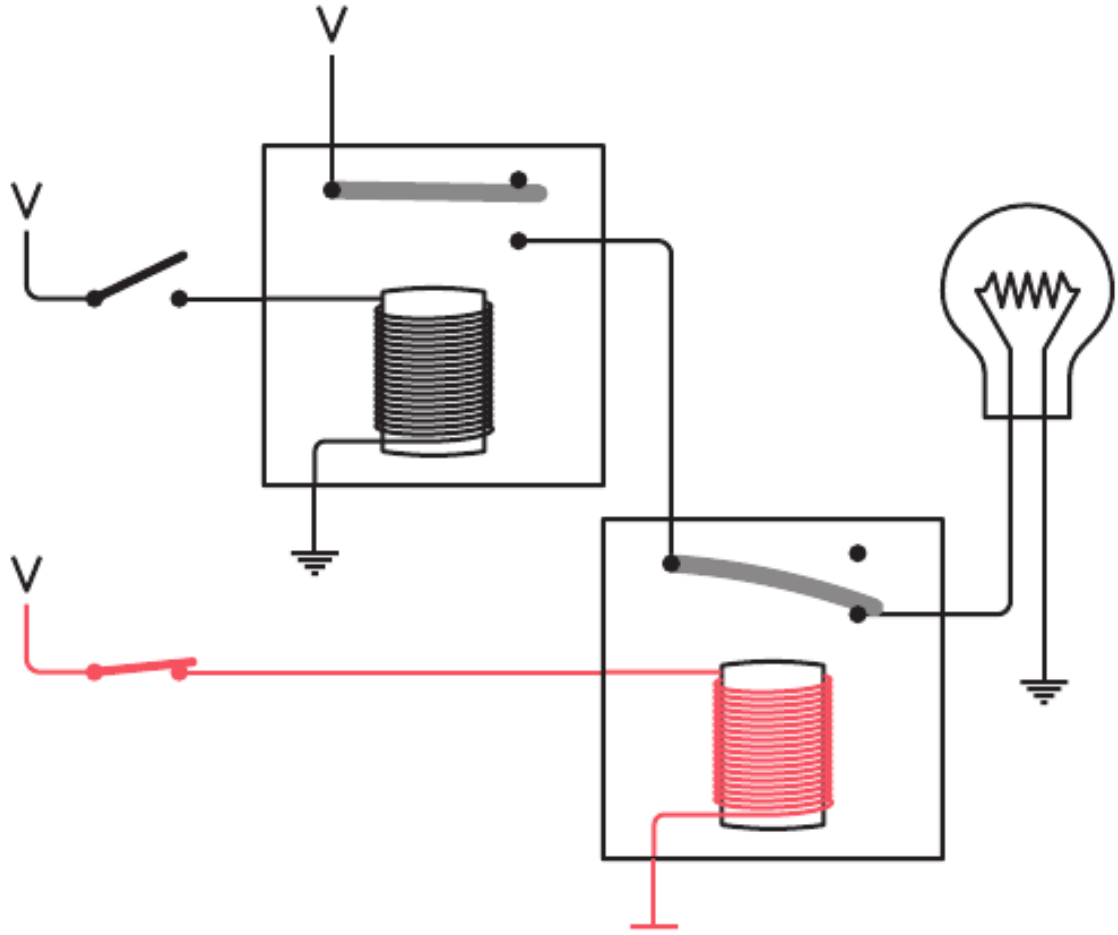
Подобно двум последовательно соединённым переключателям, два реле также можно соединить последовательно:



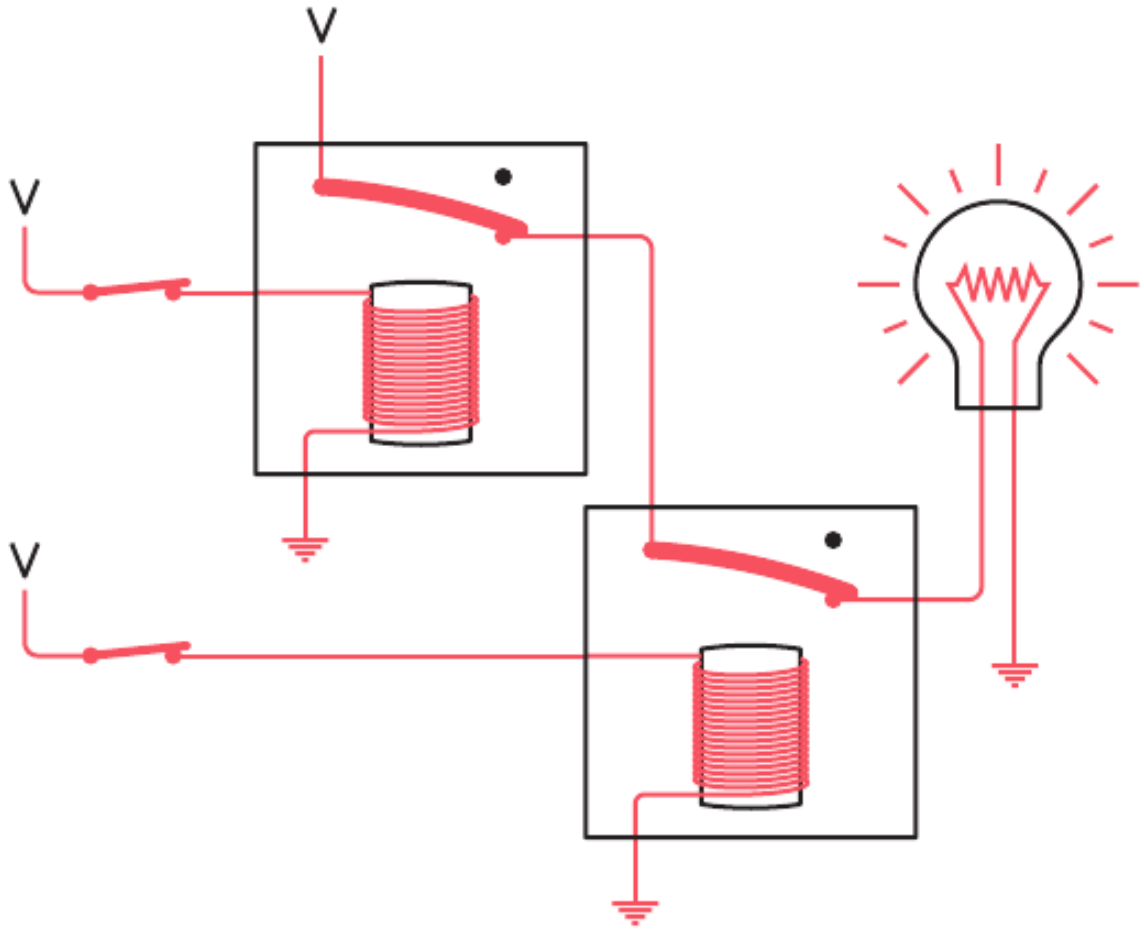
Теперь у нас два переключателя на двух реле. Выход верхнего реле подаёт напряжение на второе реле. Как видите, когда оба переключателя разомкнуты, лампочка не горит. Попробуем замкнуть верхний переключатель:



Лампочка по-прежнему не горит, поскольку нижний переключатель всё ещё разомкнут и соответствующее реле не сработало. Попробуем разомкнуть верхний переключатель и замкнуть нижний:



Лампочка всё равно не горит. Ток не может достичь лампочки, потому что первое реле не сработало. Зажечь лампочку можно лишь одним способом - замкнуть оба переключателя:



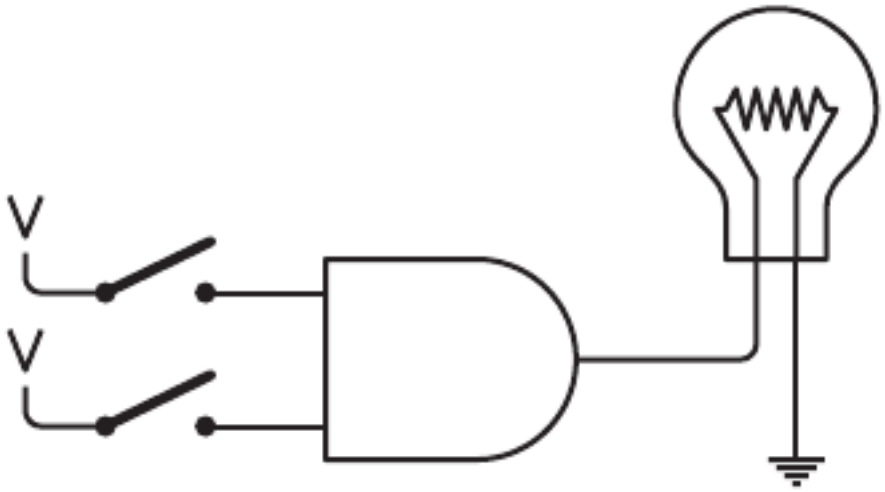
Теперь оба реле сработали, и ток может течь от V через лампочку к земле.

Как и два последовательно соединённых переключателя из главы 6, эти два реле выполняют небольшое логическое упражнение. Лампочка загорается только в том случае, если сработали оба реле. Два последовательно соединённых реле называются *вентилем AND*, потому что выполняют булеву операцию AND.

Чтобы не рисовать реле снова и снова, инженеры-электрики используют специальный символ вентиля AND:

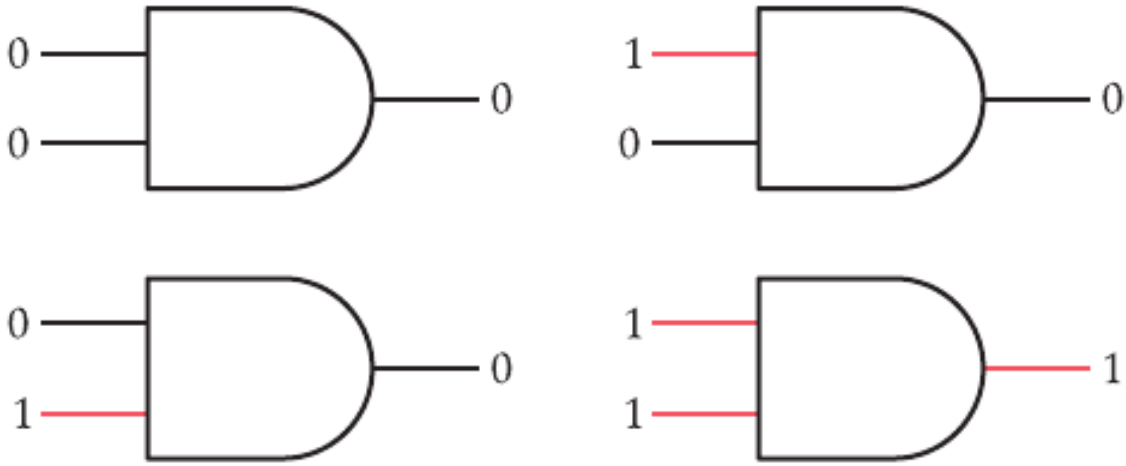
Это первый из шести основных логических вентилей. У вентиля AND два входа и один выход. Обычно его изображают со входами слева и выходом справа. Людям, привыкшим читать слева направо, удобнее так же читать электрические схемы. Но вентиль AND с тем же успехом можно изобразить со входами сверху, справа или снизу.

Предыдущую схему с двумя последовательно соединёнными реле можно записать короче:



Обратите внимание: символ вентиля AND не только заменяет два последовательно соединённых реле, но и подразумевает, что верхнее реле подключено к напряжению, а оба реле - к земле. И снова лампочка загорается лишь тогда, когда замкнуты *и* верхний, *и* нижний переключатель. Поэтому вентиль и называется AND.

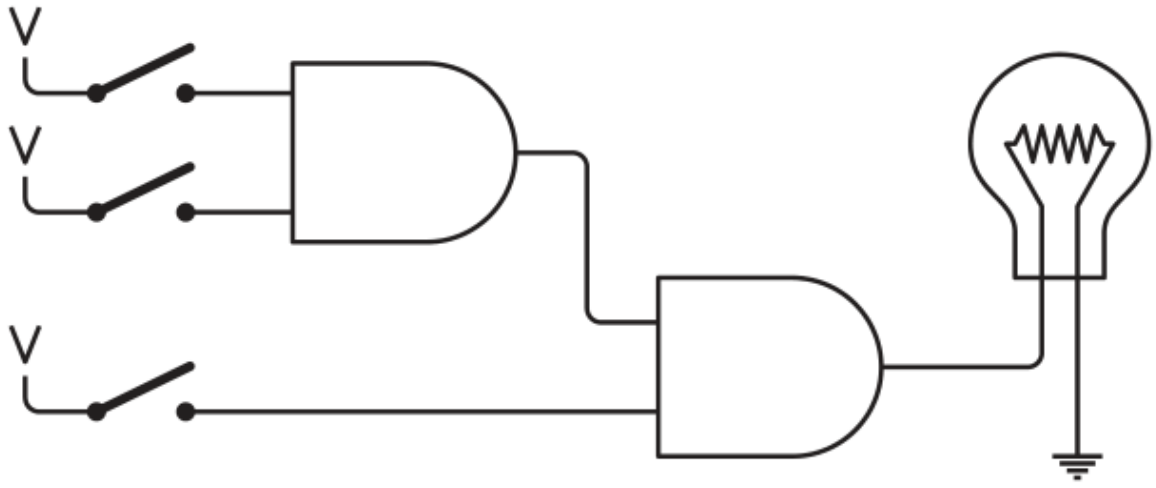
Если отсутствие напряжения считать 0, а наличие напряжения - 1, зависимость выхода вентиля AND от его входов выглядит так:



Как и два последовательно соединённых переключателя, вентиль AND можно описать небольшой таблицей:

AND	**0**	**1**
0	0	0
1	0	1

Входы вентиля AND необязательно соединять с переключателями, а выход - с лампочкой. Например, выход одного вентиля AND может стать входом второго:

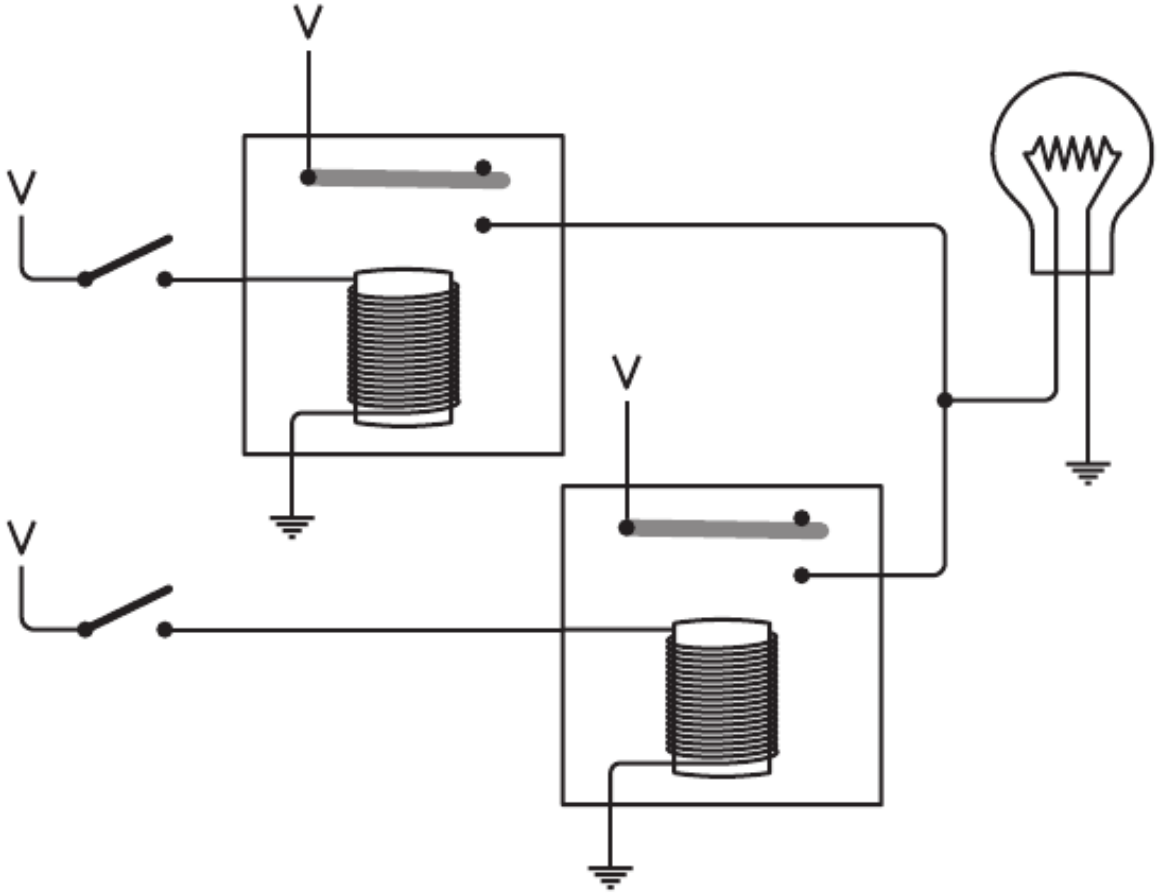


Эта лампочка загорится только при замыкании всех трёх переключателей. Лишь когда замкнуты два верхних переключателя, первый вентиль AND выдаёт напряжение; и лишь когда замкнут также третий переключатель, напряжение выдаёт второй вентиль AND.

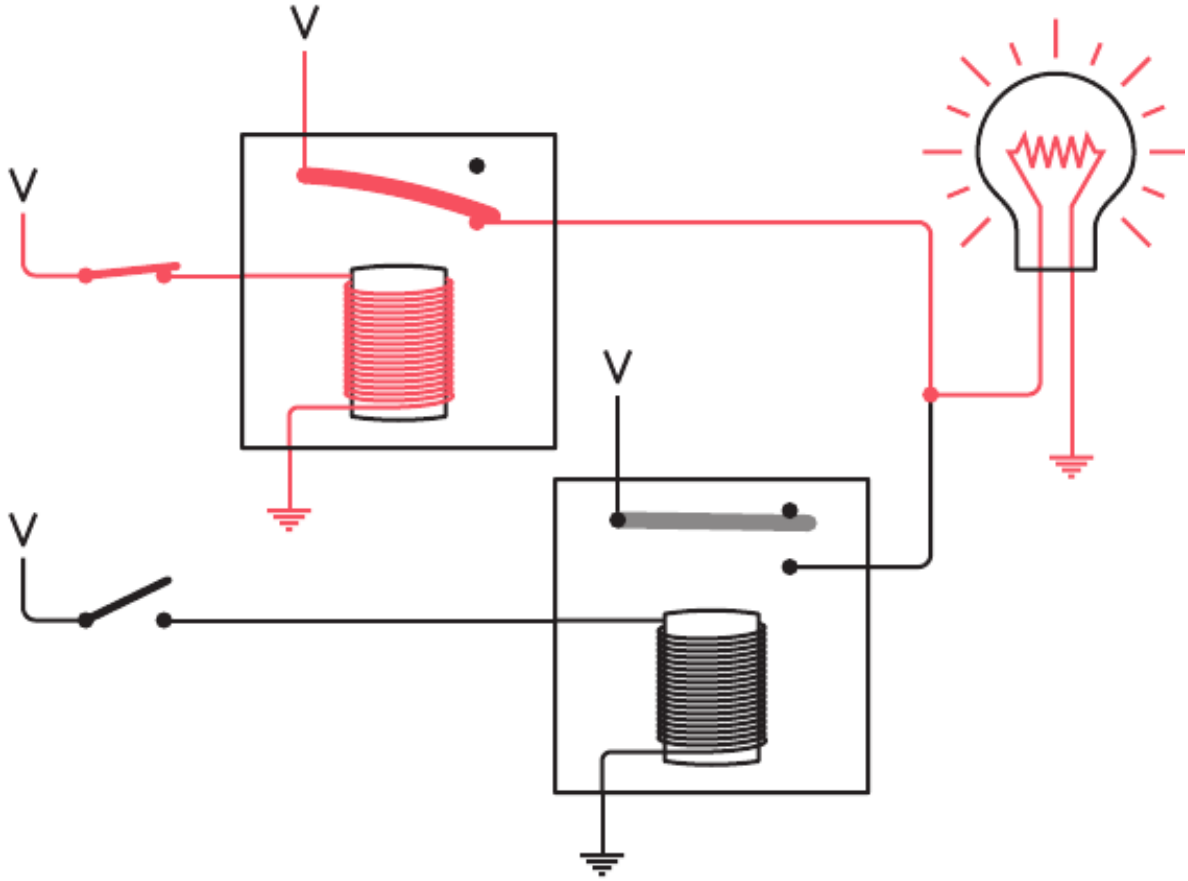
Эту конфигурацию можно выразить и одним символом:

Он называется трёхвходовым вентилем AND. Его выход равен 1 только тогда, когда все входы равны 1. Можно создавать вентили AND с гораздо большим числом входов.

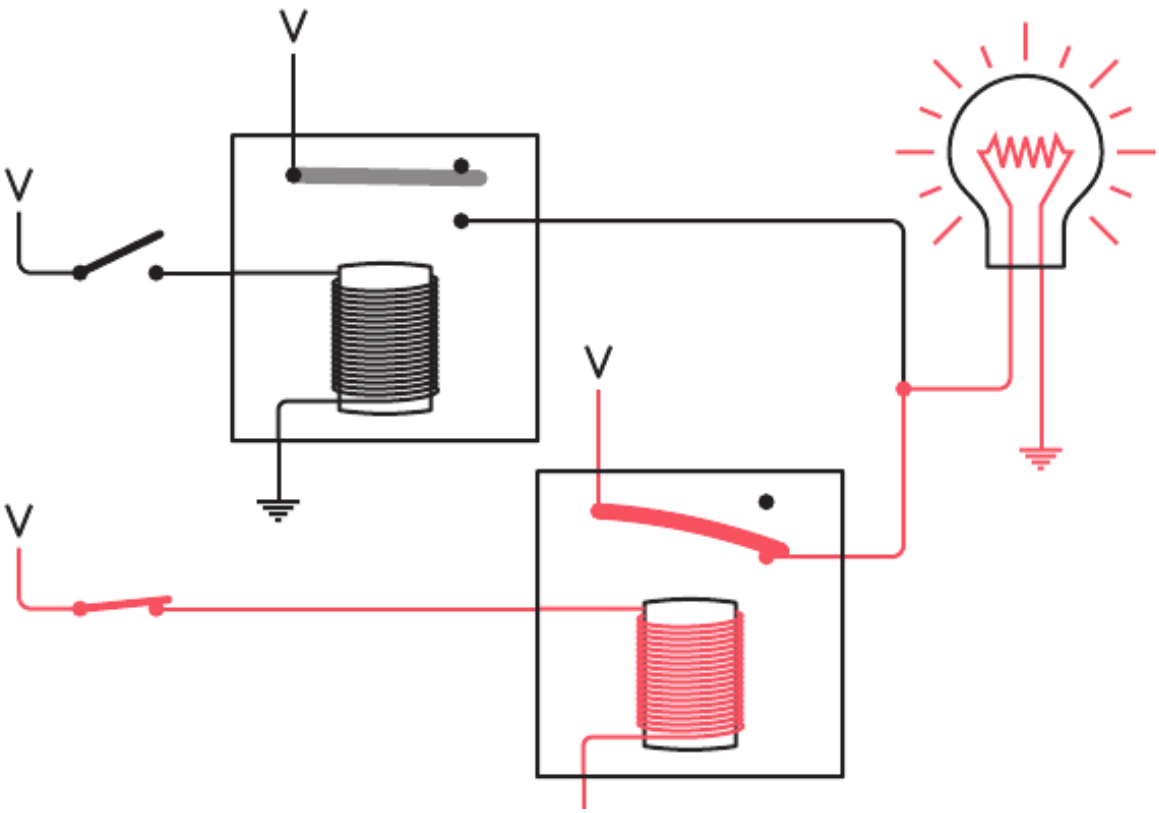
Для следующего логического вентиля требуются два параллельно соединённых реле:



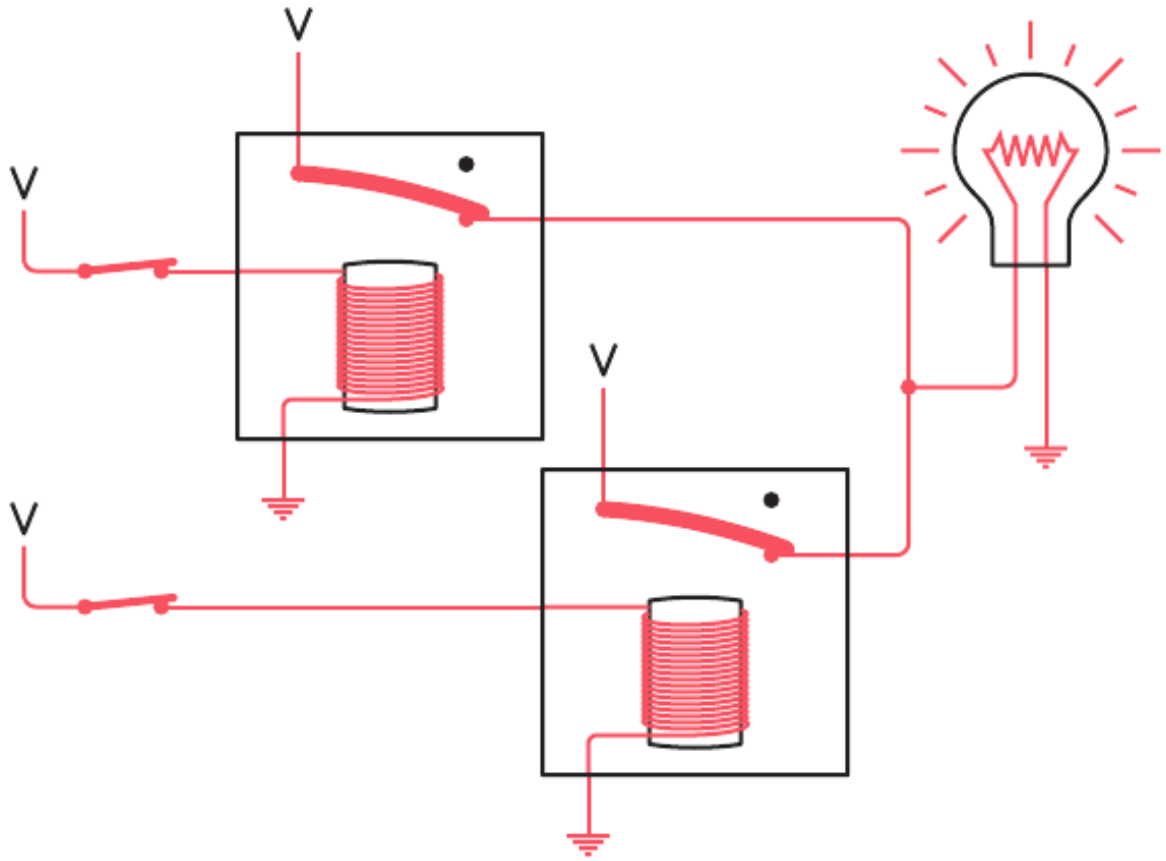
Обратите внимание, что выходы двух реле соединены друг с другом. Этот общий выход питает лампочку. Чтобы зажечь её, достаточно любого из двух реле. Например, если замкнуть верхний переключатель, лампочка загорится. Питание поступает к ней от верхнего реле:



Точно так же, если оставить верхний переключатель разомкнутым, но замкнуть нижний, лампочка загорится:



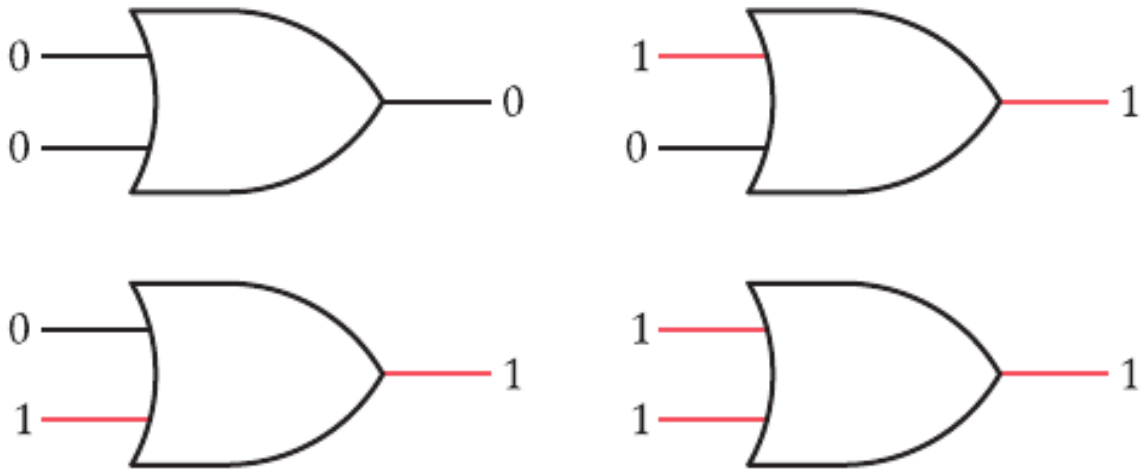
Лампочка также горит, если замкнуты оба переключателя:



Мы создали цепь, в которой лампочка загорается, если замкнут верхний *или* нижний переключатель. Ключевое слово здесь - *или*, поэтому цепь называется *вентилем OR*. Инженеры-электрики обозначают вентиль OR следующим символом:

Он немного похож на символ вентиля AND, но сторона входов закруглена примерно как буква O в слове OR. Возможно, это поможет вам различать эти символы.

Выход вентиля OR выдаёт напряжение, если напряжение есть хотя бы на одном из двух входов. Если снова считать отсутствие напряжения нулём, а наличие - единицей, вентиль OR имеет четыре возможных состояния:

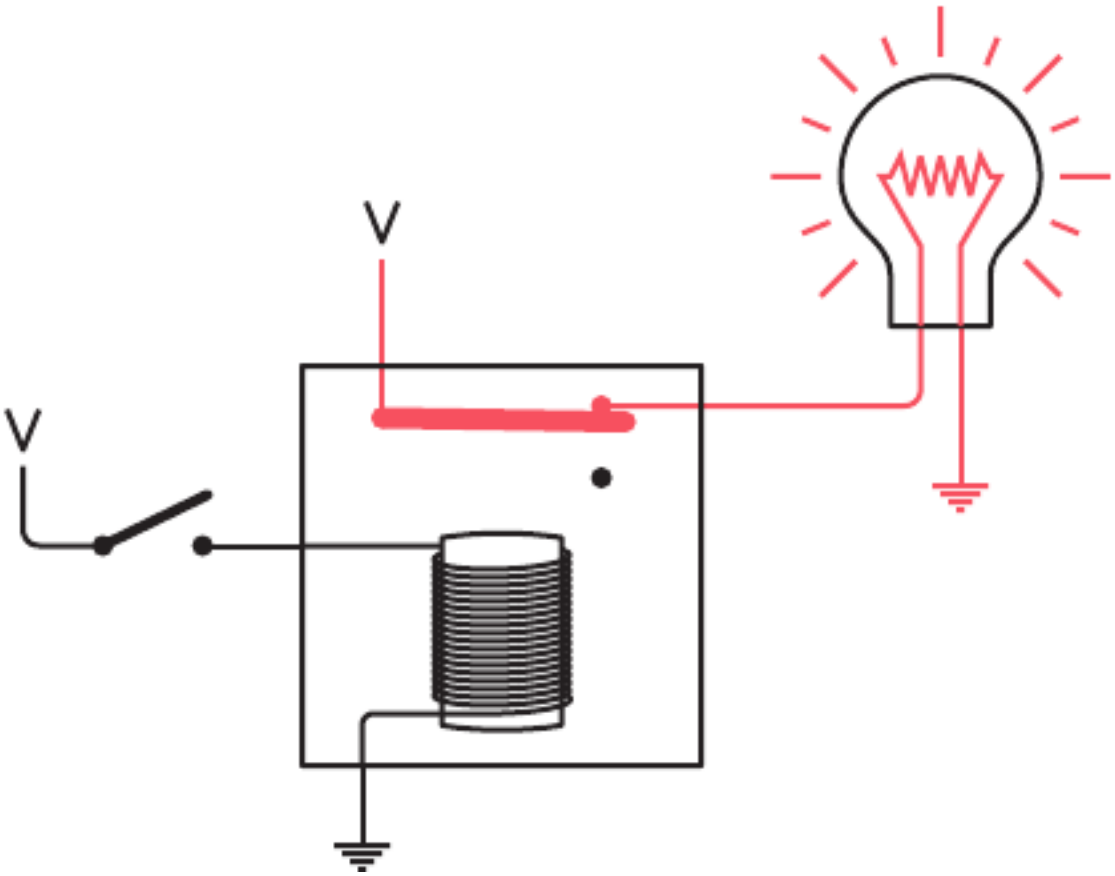


Выход вентиля OR можно свести в таблицу так же, как выход вентиля AND:

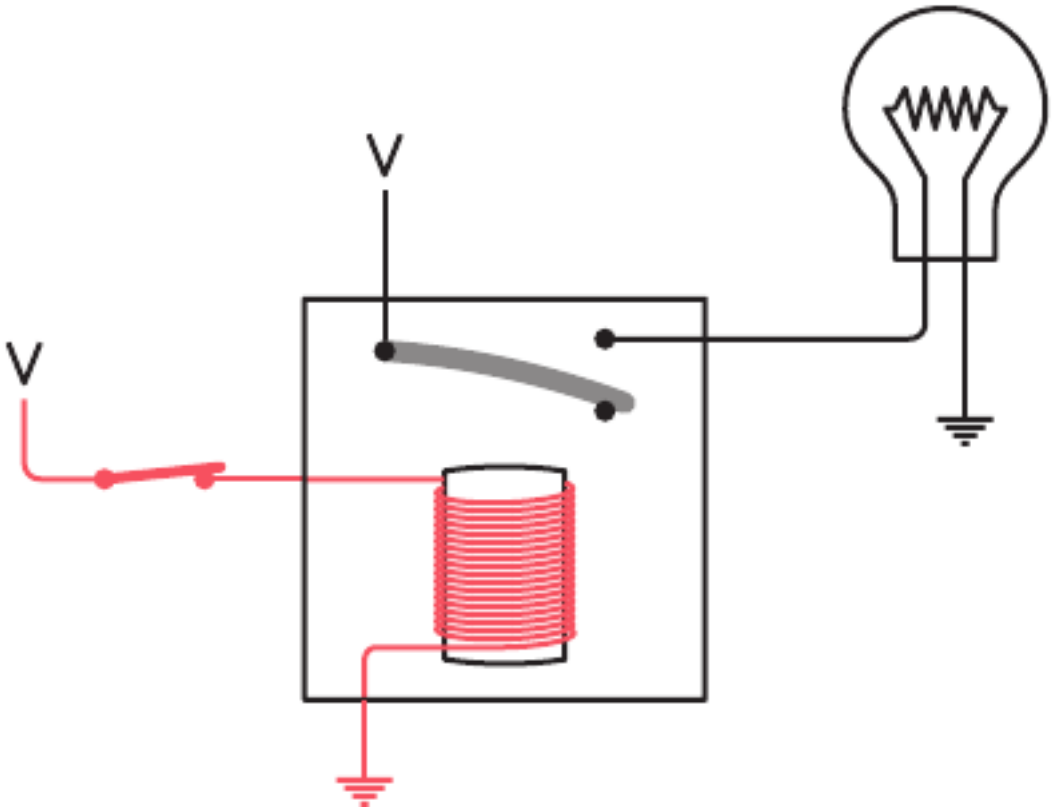
OR	**0**	**1**
0	0	1
1	1	1

У вентилей OR также может быть больше двух входов. Выход такого вентиля равен 1, если единице равен хотя бы один вход; выход равен 0, только если все входы равны 0.

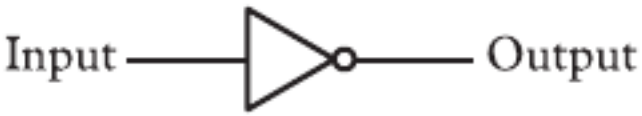
Показанные до сих пор реле называются *реле с переключающим контактом*. В состоянии покоя верхняя поворотная металлическая планка касается одного контакта, а когда электромагнит притягивает её, она касается другого. Нижний контакт называется *нормально разомкнутым* выходом. Именно им мы до сих пор пользовались, но с тем же успехом можно использовать верхний контакт, называемый *нормально замкнутым*. При использовании верхнего контакта выход реле инвертируется. Лампочка горит, когда входной переключатель разомкнут:



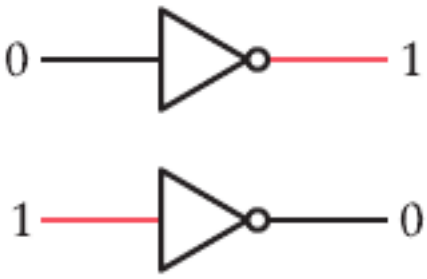
Когда входной переключатель замыкается, лампочка гаснет:



Одно реле, соединённое таким способом, называется *инвертором*. Его обозначает специальный символ:



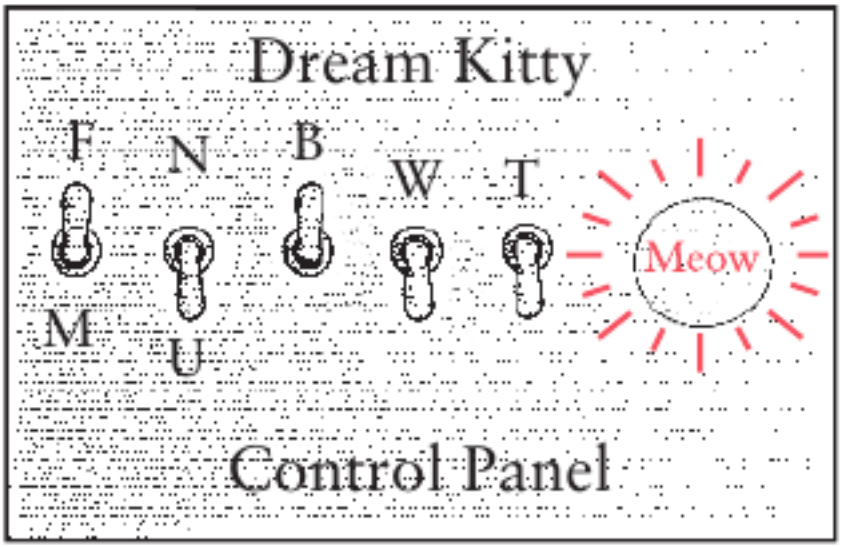
Устройство называется инвертором, поскольку превращает 0, то есть отсутствие напряжения, в 1, то есть напряжение, и наоборот:



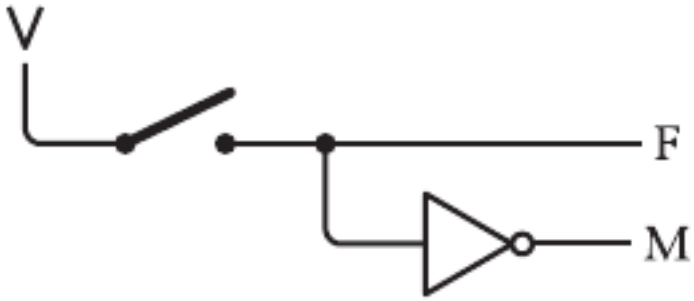
Это реализация булева оператора NOT.

Увидев верхний инвертор, люди иногда спрашивают: «Как на выходе может быть напряжение, если на входе его нет? Откуда берётся это напряжение?» Помните, что инвертор в действительности представляет собой реле, подключённое к источнику напряжения.

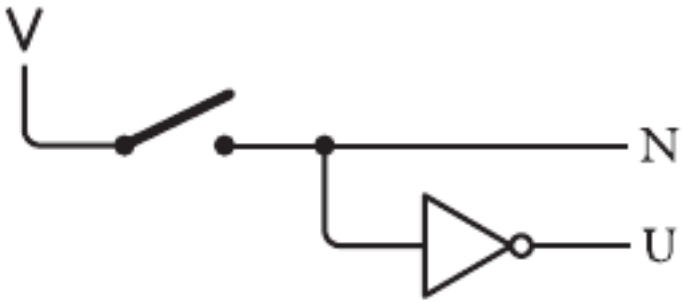
Имея инвертор, вентиль AND и вентиль OR, мы можем начать соединять пульт так, чтобы автоматизировать выбор идеальной кошки. Вот этот пульт ещё раз:



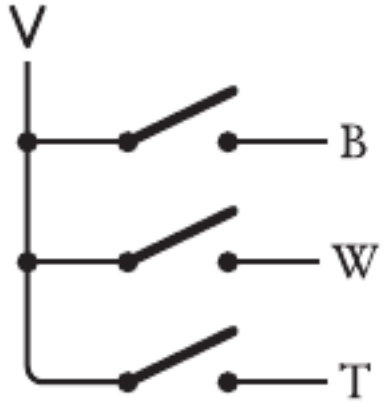
Начнём с переключателей. Первый замкнут для кошки и разомкнут для кота. Таким образом можно получить два сигнала, которые мы назовём F и M:



Когда F равно 1, M равно 0, и наоборот. Аналогично второй переключатель замкнут для кастрированного животного и разомкнут для некастрированного:



Остальные три переключателя выбирают окрас: чёрный, белый или рыжевато-коричневый. Все три подключены к напряжению:

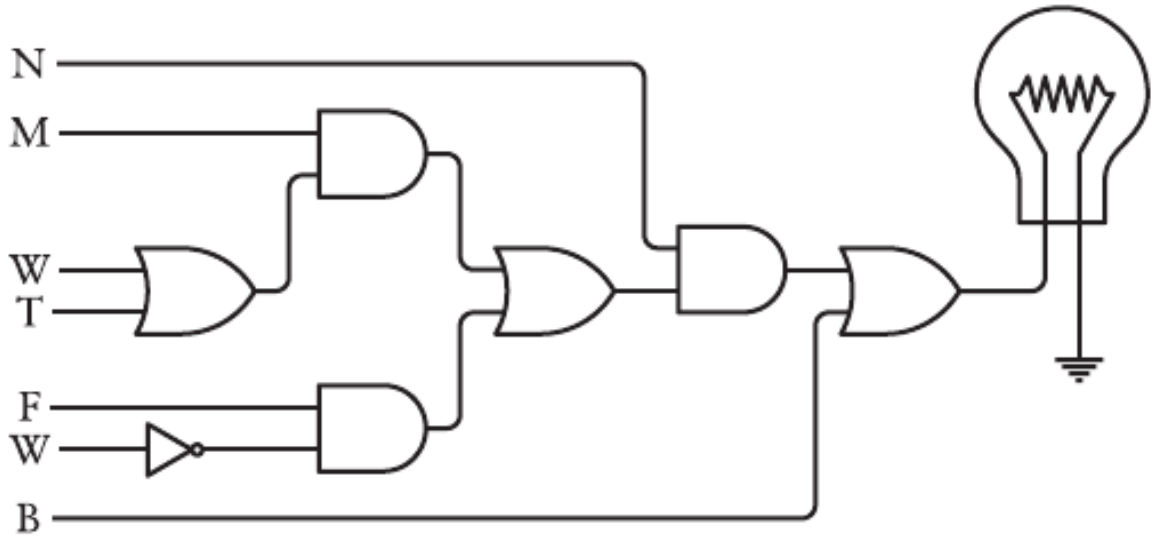


Соединение вентилей и инверторов подчиняется нескольким простым правилам. Выход одного вентиля или инвертора может служить входом для одного или нескольких других вентилей или инверторов. Но соединять друг с другом выходы двух или нескольких вентилей или инверторов нельзя.

Упрощённое выражение выбора кошки имело вид:

```
(N × ((M × (W + T)) + (F × (1 - W)))) + B
```

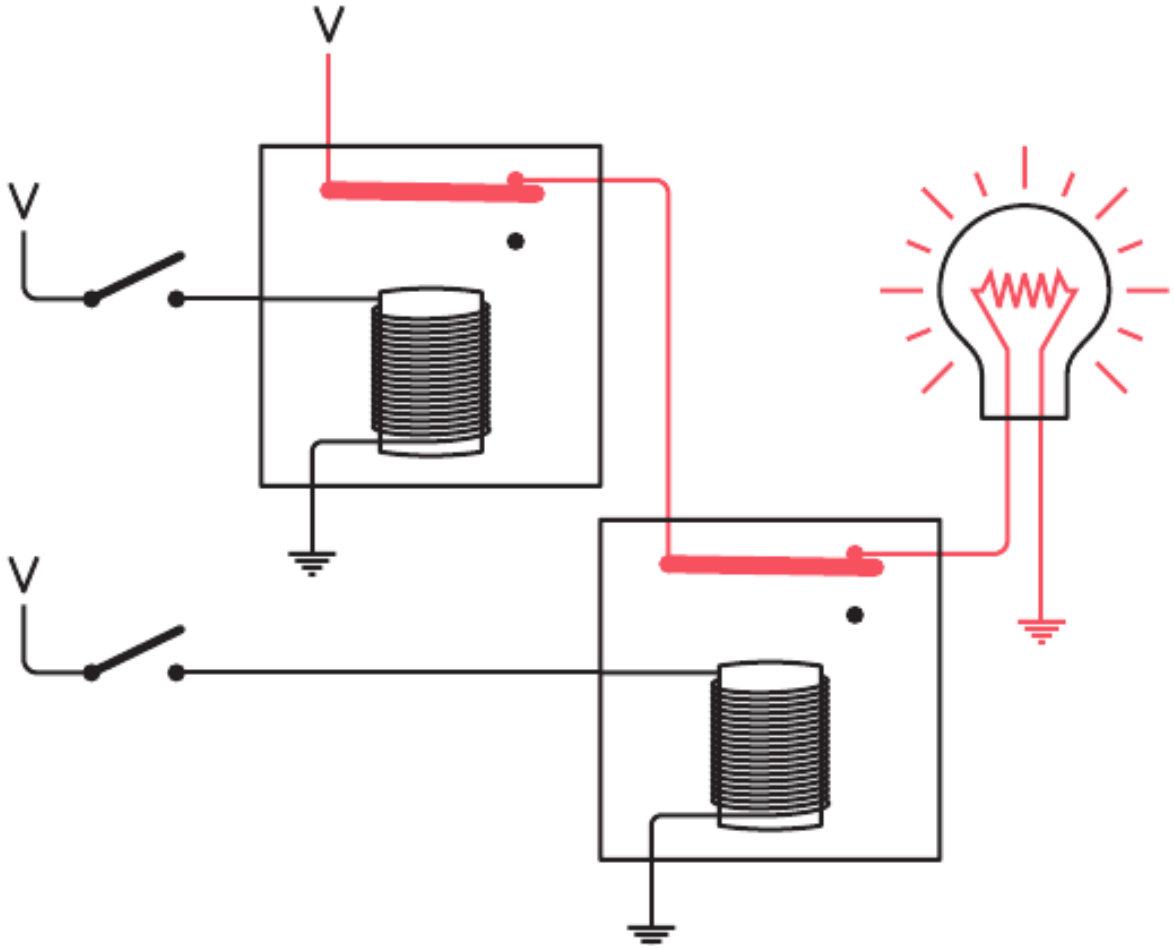
Каждому знаку + в этом выражении должен соответствовать вентиль OR в цепи. Каждому знаку × должен соответствовать вентиль AND.



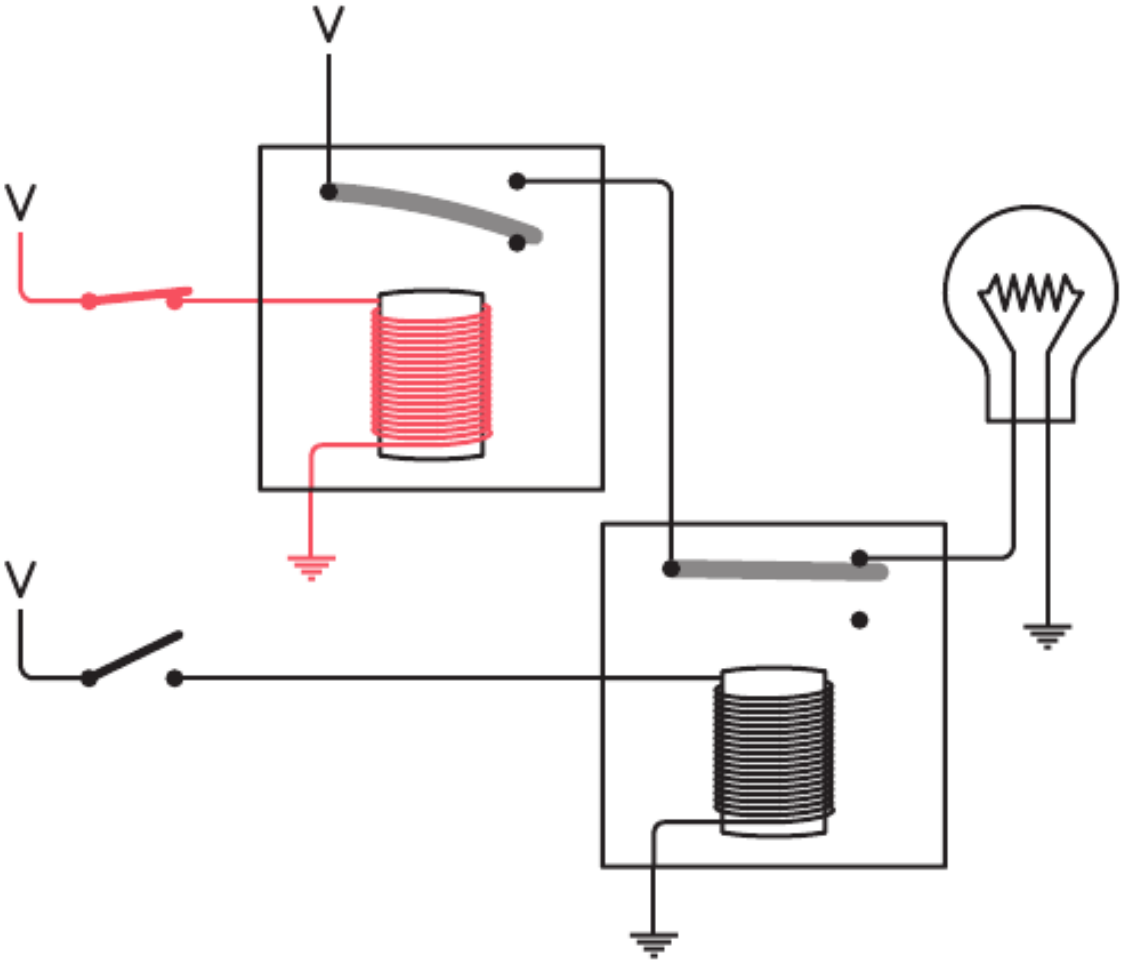
Символы в левой части схемы расположены в том же порядке, в каком появляются в выражении. Эти сигналы поступают из трёх предыдущих схем. Обратите внимание на ещё один инвертор, реализующий часть выражения (1 - W).

Вы можете сказать: «Да здесь прорва реле!» И будете правы. В каждом вентиле AND и OR по два реле, а в каждом инверторе - одно. Боюсь, в следующих главах вы увидите ещё гораздо больше реле. Радуйтесь, что вам не придётся покупать их и соединять дома. Если только вам этого не захочется.

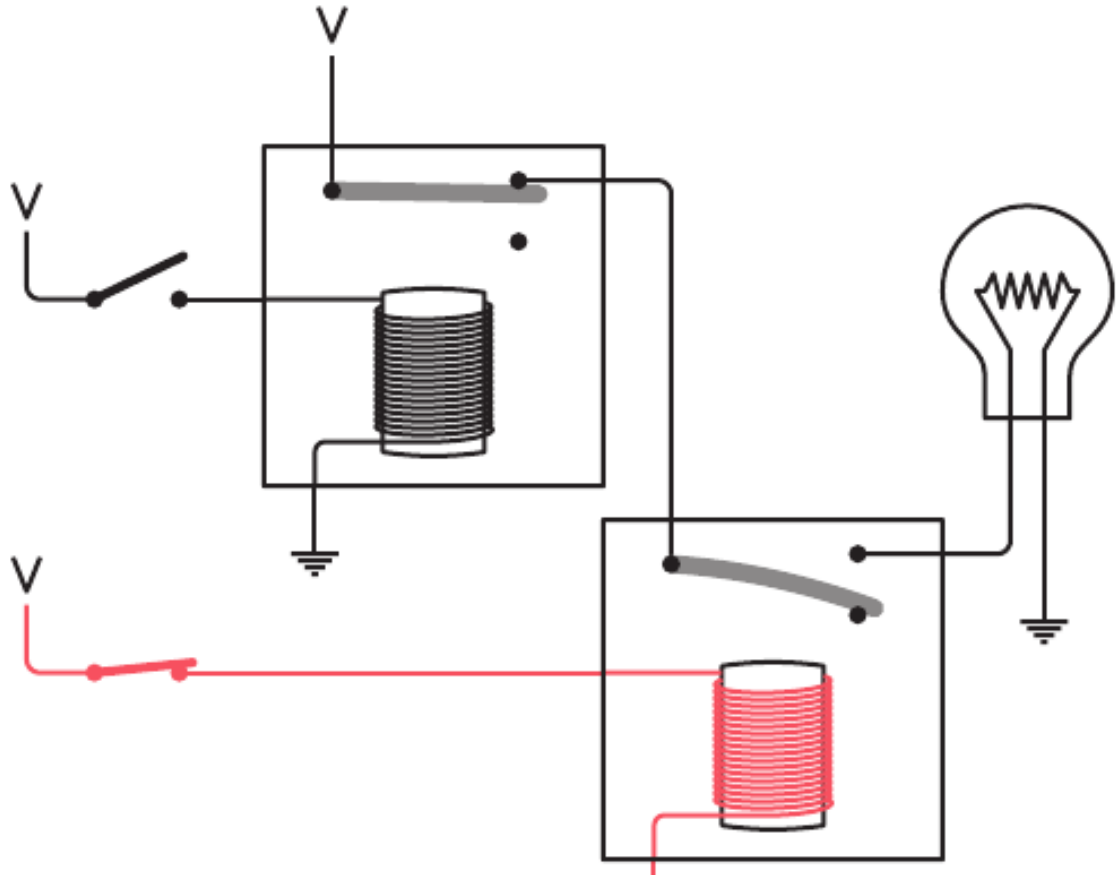
Ранее я упомянул, что существует шесть стандартных логических вентилей. Три вы уже увидели, и теперь пришло время остальных. Первые два используют нормально замкнутый выход реле, тот самый, который применялся в инверторе. На этом выходе присутствует напряжение, когда реле не сработало. Например, в следующей конфигурации выход одного реле питает второе. Когда оба входа выключены, лампочка горит:



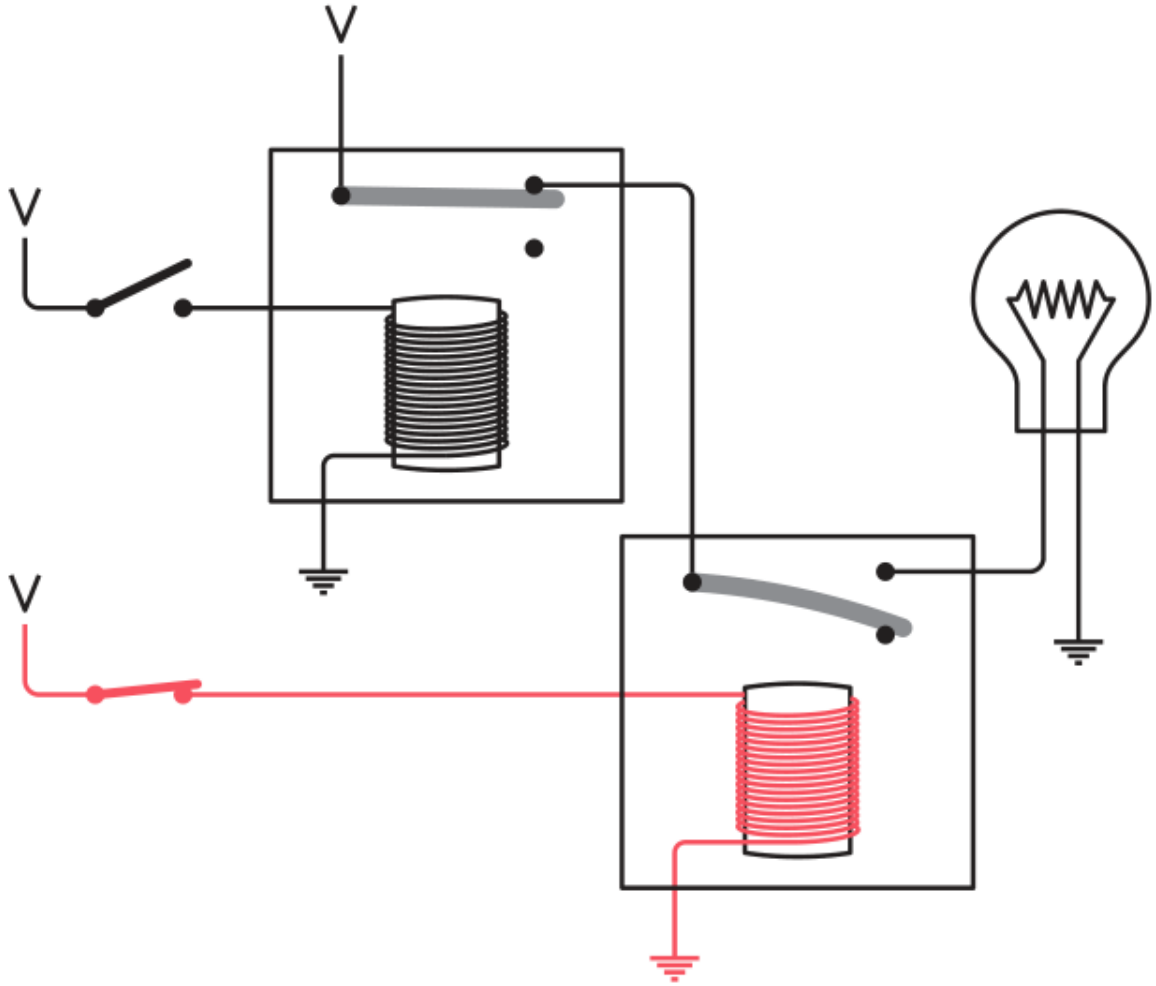
Если замкнуть верхний переключатель, лампочка погаснет:



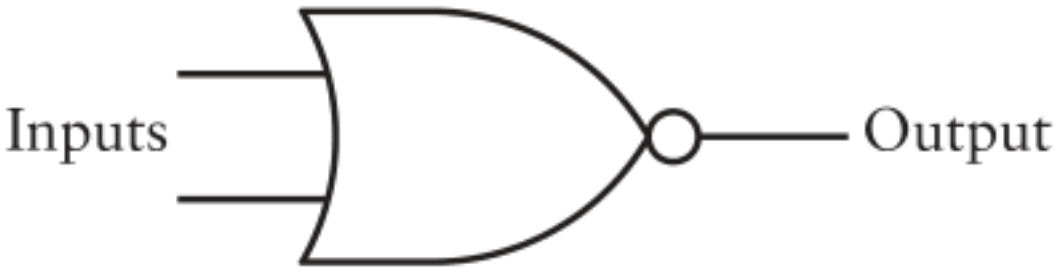
Лампочка гаснет, потому что питание больше не поступает ко второму реле. Точно так же, если верхний переключатель разомкнут, а нижний замкнут, лампочка тоже не горит:



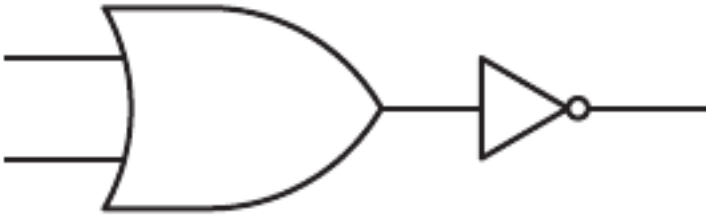
Если замкнуты оба переключателя, лампочка также не горит:



Такое поведение прямо противоположно поведению вентиля OR. Оно называется NOT OR, или короче *NOR*. Вот символ вентиля NOR:



Он совпадает с символом OR, за исключением маленького кружка на выходе. Кружок означает инверсию. NOR эквивалентен вентилю OR, за которым следует инвертор:

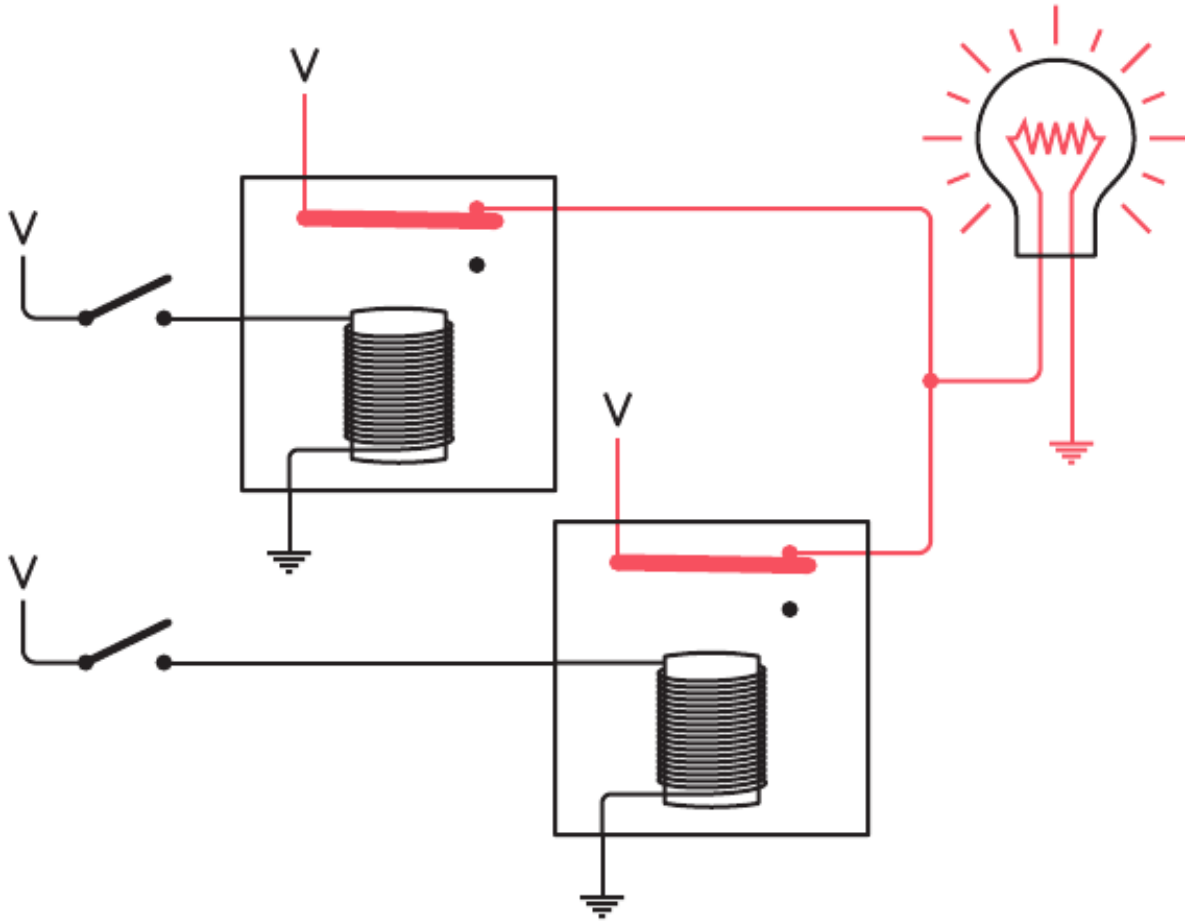


Выход вентиля NOR показан в следующей таблице:

NOR	**0**	**1**
0	1	0
1	0	0

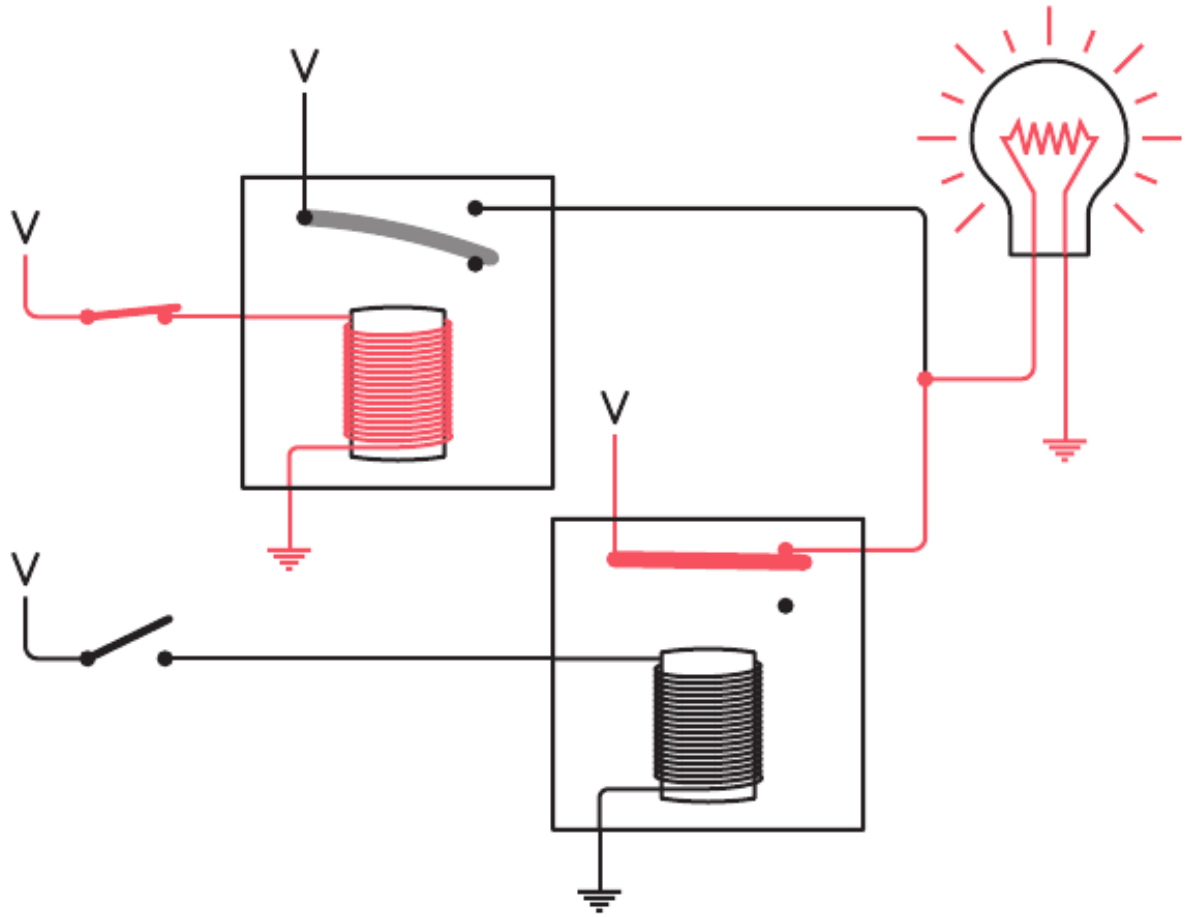
Здесь результаты противоположны результатам вентиля OR, выход которого равен 1, если хотя бы один из двух входов равен 1, и равен 0 только при двух нулевых входах.

Два реле можно соединить и ещё одним способом:

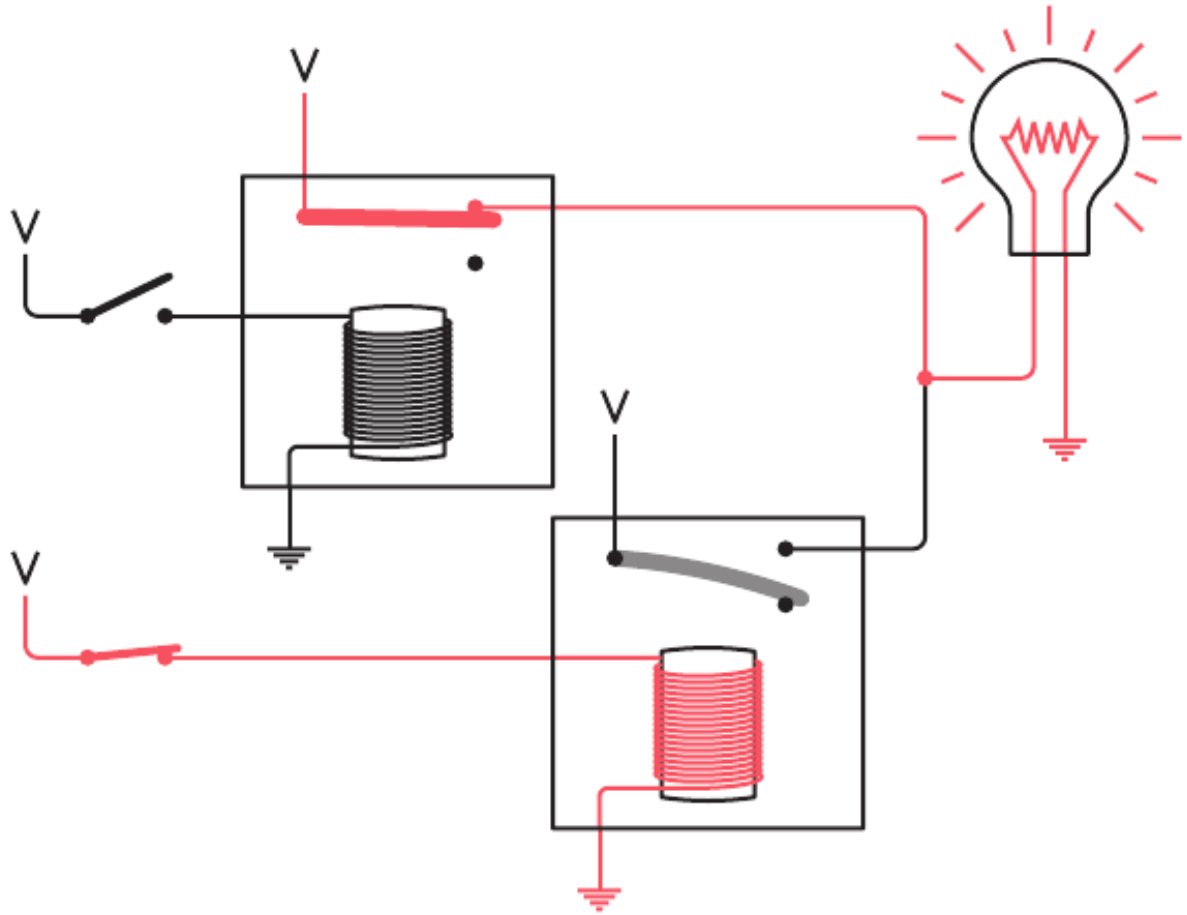


В этой схеме два выхода соединены вместе, как в конфигурации OR, но используются другие контакты. Когда оба переключателя разомкнуты, лампочка горит.

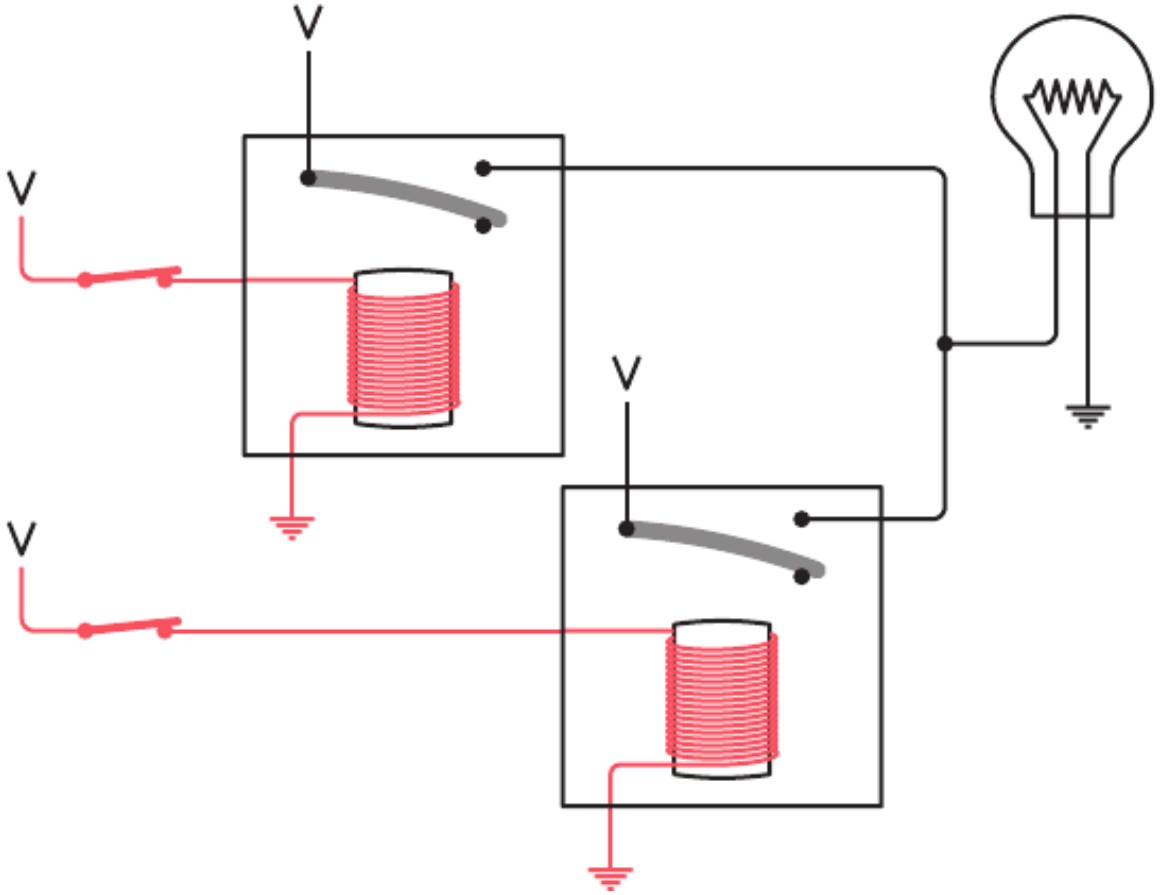
Она продолжает гореть, если замкнут только верхний переключатель, поскольку может получать питание от нижнего реле:



Точно так же лампочка продолжает гореть, если замкнут лишь нижний переключатель, поскольку получает питание от верхнего реле:

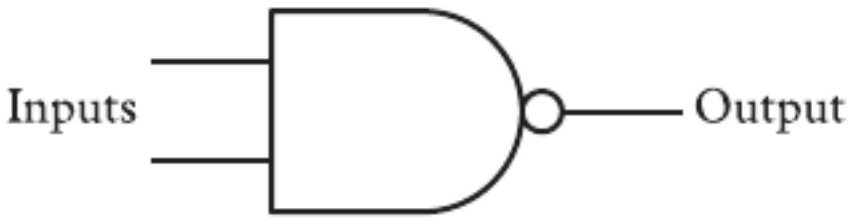


Лампочка гаснет, только когда замкнуты оба переключателя:



Такое поведение в точности противоположно поведению вентиля AND. Оно называется NOT AND, или короче *NAND*. В отличие от NOR, слово NAND было придумано специально для обозначения этого вида логики. Оно появилось в 1958 году.

Вентиль NAND рисуют так же, как AND, но с кружком на выходе, который означает, что выход инвертирован относительно выхода AND:



Вентиль NAND ведёт себя следующим образом:

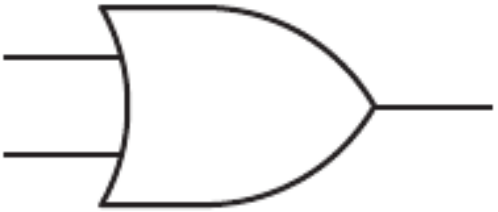
NAND	**0**	**1**
0	1	1
1	1	0

Напомним, что выход вентиля AND равен 1 лишь тогда, когда оба входа равны 1; в остальных случаях выход равен 0. У вентиля NAND всё наоборот.

Итак, мы рассмотрели четыре разных способа соединить реле с двумя входами и одним выходом. Каждая конфигурация ведёт себя немного иначе. Чтобы не рисовать реле снова и снова, мы назвали эти конфигурации логическими вентилями и решили обозначать их теми же символами, которыми пользуются инженеры-электрики. Зависимость выхода конкретного логического вентиля от входов сведена здесь:

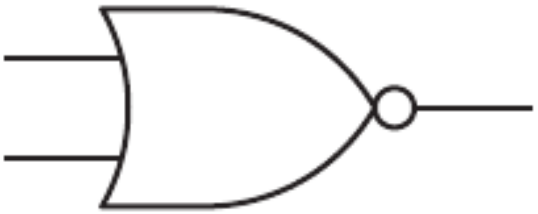
AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1



NAND	0	1
0	1	1
1	1	0

NOR	0	1
0	1	0
1	0	0



AND	**0**	**1**
0	0	0

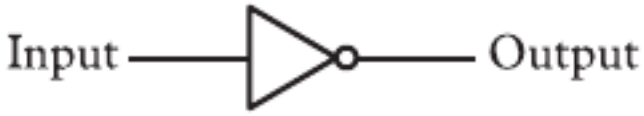
AND	0	1
1	0	1

OR	0	1
0	0	1
1	1	1

NAND	0	1
0	1	1
1	1	0

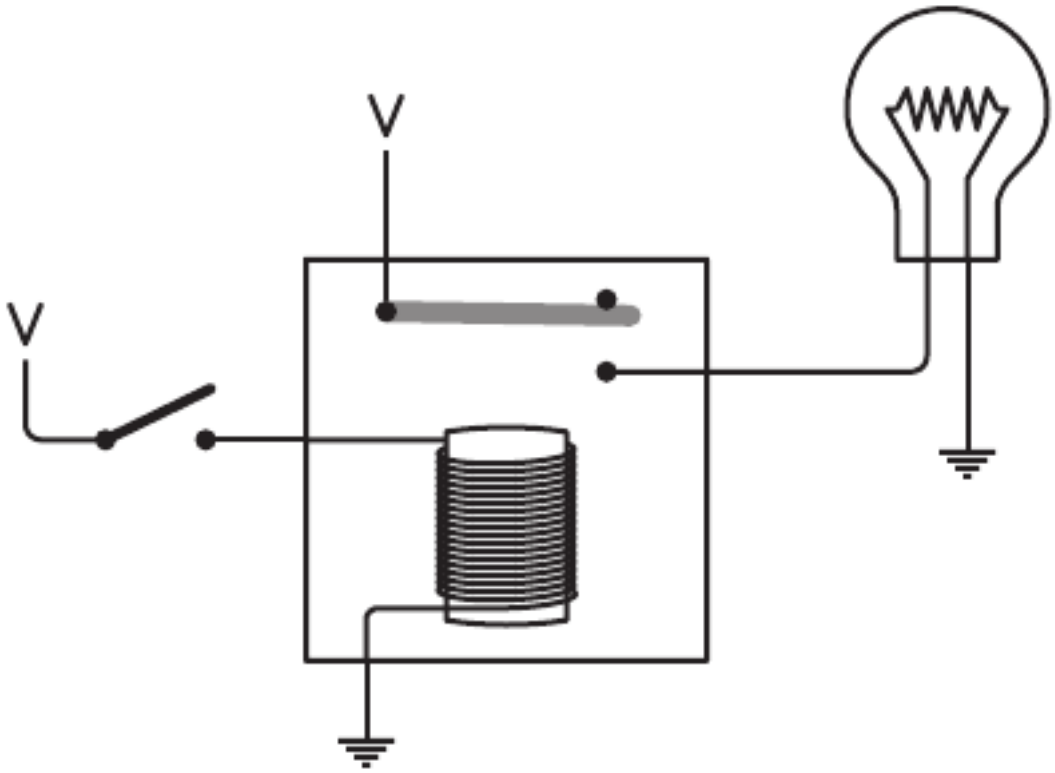
NOR	0	1
0	1	0
1	0	0

Инвертор выглядит так:

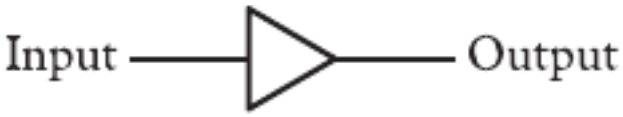


Он превращает сигнал 0 в 1, а сигнал 1 - в 0.

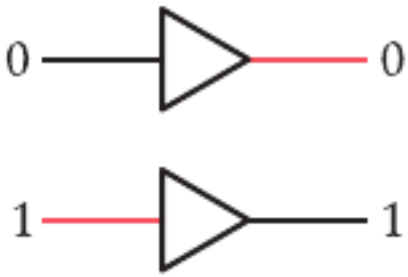
Завершает наш набор инструментов самое обыкновенное реле:



Оно называется *буфером* и обозначается следующим символом:



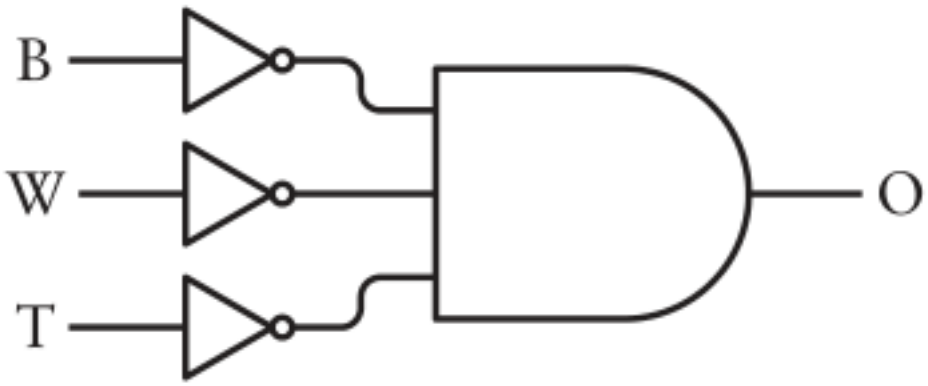
Это тот же символ, что у инвертора, но без маленького кружка. Буфер замечателен тем, что почти ничего не делает. Его выход совпадает со входом:



Но буфер можно использовать при слабом входном сигнале. Вспомните, что именно ради этого много лет назад изобрели реле для телеграфа. В настоящих логических схемах одному выходу иногда приходится обслуживать множество входов. Это называется *разветвлением*, или *fan-out*, и может уменьшить мощность, доступную каждому входу. Буферы помогают повысить эту мощность. Кроме того, буфер может слегка задержать сигнал: реле требуется некоторое время, доля секунды, чтобы сработать.

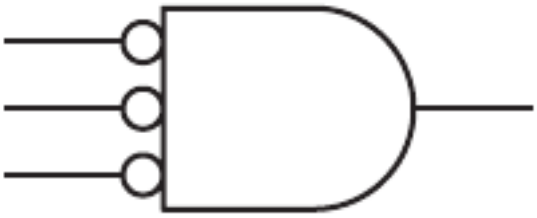
Начиная с этого места в книге схемы реле будут встречаться очень редко. Вместо них последующие цепи будут строиться из буферов, инверторов, четырёх основных двухвходовых логических вентилей и более сложных цепей, собранных из этих вентилей. Разумеется, все эти компоненты сделаны из реле, но теперь нам уже необязательно смотреть на сами реле.

В начале главы был показан небольшой пульт для выбора идеального котёнка. На нём имелись переключатели чёрного, белого и рыжевато-коричневого окраса, но отсутствовал переключатель других окрасов, то есть любого котёнка, который не является ни чёрным, ни белым, ни рыжевато-коричневым. Однако такой сигнал можно создать при помощи трёх инверторов и трёхвходового вентиля AND:



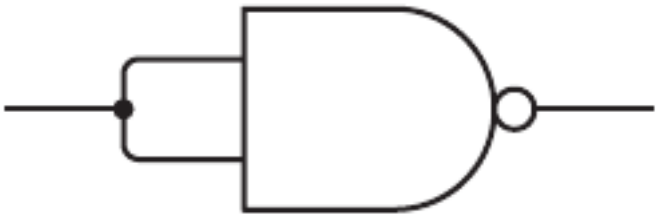
Три входных сигнала инвертируются и поступают на входы вентиля AND. Только когда B, W и T одновременно равны 0, все входы вентиля AND становятся равны 1, отчего и его выход становится равен 1.

Иногда такую конфигурацию рисуют без инверторов:

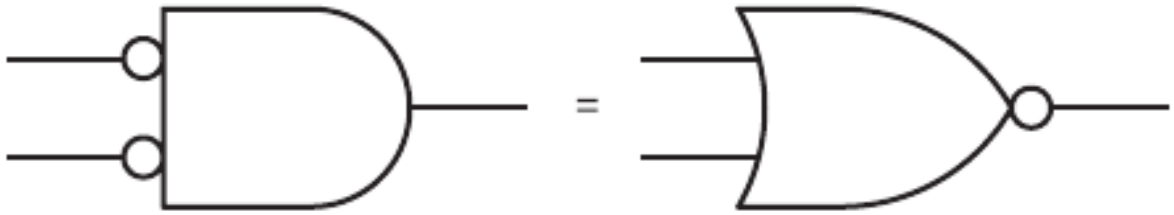


Обратите внимание на маленькие кружки у входов вентиля AND. Они означают, что сигналы в этих точках инвертируются: 0, то есть отсутствие напряжения, становится 1, то есть напряжением, и наоборот.

Если бы из шести показанных логических вентилей пришлось выбрать лишь один, выбирайте NAND или NOR. При помощи NAND либо NOR можно построить все остальные логические вентили. Например, вот как объединить входы вентиля NAND, чтобы получить инвертор:

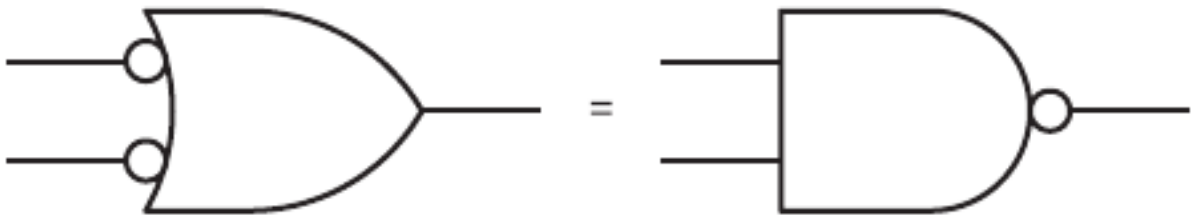


Такой инвертор можно поставить на выходе другого вентиля NAND и получить вентиль AND. Сначала кажется, что из NAND невозможно сделать вентиль OR, однако это возможно. Дело в том, что вентиль AND со всеми инвертированными входами работает в точности так же, как вентиль NOR:



Выход равен 1 только тогда, когда оба входа равны 0.

Аналогично вентиль OR с двумя инвертированными входами эквивалентен вентилю NAND:



Выход равен 0 только тогда, когда оба входа равны 1.

Эти две пары эквивалентных цепей представляют электрическую реализацию *законов де Моргана*. Август де Морган был ещё одним математиком викторианской эпохи, на девять лет старше Джорджа Буля. Его книга *«Формальная логика»* вышла в 1847 году в тот же самый день, как гласит история, что и книга Буля *«Математический анализ логики»*. В действительности к исследованию логики Буля подтолкнула широко обсуждавшаяся в обществе вражда де Моргана с другим британским математиком, в ходе которой звучали обвинения в плагиате. История оправдала де Моргана. Он очень рано понял важность прозрений Буля, бескорыстно поддерживал его и помогал ему, но сегодня, к сожалению, почти забыт, если не считать знаменитых законов.

Законы де Моргана наиболее кратко выражаются так:

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

A и B - два булевых операнда. Черта сверху означает инверсию. В первом выражении A и B инвертируются, а затем объединяются булевым оператором AND. Это то же самое, что объединить два операнда булевым оператором OR, а затем инвертировать результат, то есть применить NOR. Это работает и в обычной речи: если дождь *не* идёт *и* снег *не* идёт, значит, не идёт дождь *или* снег.

Во втором выражении оба операнда инвертируются, а затем объединяются булевым оператором OR. Это то же самое, что объединить операнды булевым оператором AND, а потом инвертировать результат, то есть применить NAND. Если я *не* большой *или* я *не* сильный, значит, я не большой *и* сильный одновременно. Законы де Моргана - важный инструмент для упрощения булевых выражений, а следовательно, и цепей. Исторически именно в этом заключалось подлинное значение работы Клода Шеннона для инженеров-электриков. Но навязчивое упрощение цепей не будет главной заботой этой книги. Предпочтительнее заставить устройство работать, чем добиться, чтобы оно работало самым простым возможным способом.

Следующим крупным проектом станет не что иное, как цифровая суммирующая машина, целиком построенная из логических вентилей. Но этот проект придётся отложить на несколько глав, пока мы вернёмся в начальную школу и научимся считать.

Глава 9. Наши десять цифр

Мысль о том, что язык - всего лишь код, кажется вполне приемлемой. Многие из нас по крайней мере пытались изучать иностранный язык в школе, поэтому готовы признать, что животное, которое по-английски называется *cat*, может также называться *gato*, *chat*, *Katze*, *кошка* или *γάτα*.

Однако числа кажутся менее податливыми влиянию культуры. Независимо от того, на каком языке мы говорим и как произносим числа, почти все люди, с которыми нам доведётся встретиться на этой планете, записывают их одинаково:

```
1   2   3   4   5   6   7   8   9   10
```

Разве математику не называют «универсальным языком» именно по этой причине?

Числа, несомненно, являются самыми абстрактными кодами, с которыми мы регулярно имеем дело. Когда мы видим число

```
3
```

нам необязательно сразу связывать его с чем-либо. Можно представить три яблока или три других предмета, но из контекста мы с такой же лёгкостью поймём, что число обозначает день рождения ребёнка, телевизионный канал, счёт хоккейного матча, количество чашек муки в рецепте пирога или месяц март. Поскольку наши числа изначально столь абстрактны, нам труднее понять, что такое количество яблок

необязательно обозначать символом

```
3
```

Значительная часть этой и следующей главы будет посвящена тому, чтобы убедить себя: такое количество яблок

можно также обозначить записью

```
11
```

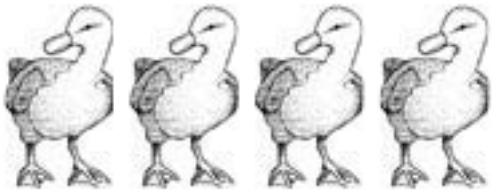
Когда мы дойдём до этого, станет возможно начать представлять числа в электрических цепях, а в конце концов - в компьютерах. Но чем лучше мы понимаем устройство привычных чисел, тем лучше будем подготовлены к этому переходу.

С тех пор как наш вид впервые начал считать, мы помогали себе пальцами. Поэтому большинство цивилизаций строило системы счисления вокруг числа десять. Единственные существенные исключения - несколько систем с основанием пять, двадцать или шестьдесят, но все они тесно связаны с десятью. Древневавилонская система с основанием 60 до сих пор сохраняется в нашем измерении времени секундами и минутами. В нашей системе счисления нет ничего особенного, кроме связи с физиологией человеческой руки. Если бы у людей было восемь или двенадцать пальцев, мы считали бы немного иначе. Не случайно английское слово *digit* означает и палец руки или ноги, и цифру, а слова *five* («пять») и *fist* («кулак») имеют сходные корни.

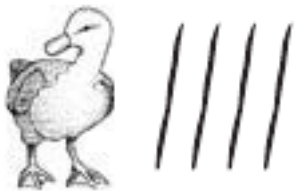
В этом смысле применение системы счисления с *основанием десять*, или *десятичной* системы, от латинского слова «десять», совершенно произвольно. И всё же мы наделяем основанные на десяти числа почти магическим значением и даём им особые названия. Десять лет - десятилетие; десять десятилетий - век; десять веков - тысячелетие. Тысяча тысяч - миллион; тысяча миллионов - миллиард. Все эти числа являются степенями десяти:

```
10^1 = 10
10^2 = 100
10^3 = 1000 (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)
```

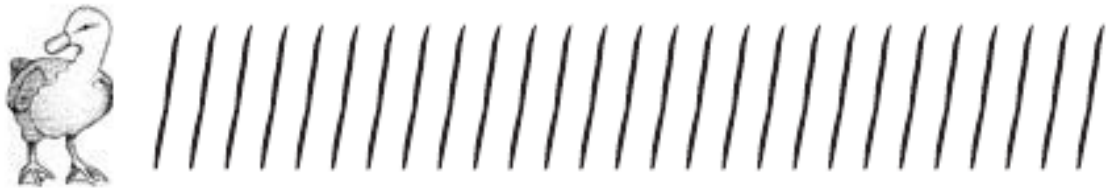
Большинство историков считает, что числа первоначально придумали для подсчёта вещей: людей, имущества и торговых сделок. Например, если у кого-то было четыре утки, это можно было записать рисунками четырёх уток:



В конце концов человек, чьей работой было рисовать уток, подумал: «Почему я должен рисовать четырёх уток? Почему нельзя нарисовать одну и показать, что их четыре, ну, например, царапинами или чем-нибудь ещё?»



А затем настал день, когда у кого-то оказалось 27 уток, и царапины стали выглядеть нелепо:



Кто-то сказал: «Должен существовать способ получше», - и родилась система счисления.

Из всех ранних систем счисления в повседневном употреблении сохранились только римские цифры. Они встречаются на циферблатах часов, в датах на памятниках и статуях, в нумерации некоторых глав и страниц книг, в пунктах планов и, что раздражает сильнее всего, в уведомлениях об авторских правах в фильмах. На вопрос «В каком году снята эта картина?» часто можно ответить, только если успеть расшифровать MCMLIII, пока идут последние кадры титров.

Двадцать семь уток римскими цифрами записываются так:

Идея достаточно проста: X заменяет десять царапин, а V - пять.

До наших дней дошли следующие символы римских цифр:

```
I  V  X  L  C  D  M
```

I означает единицу. Она могла произойти от царапины или одного поднятого пальца. V, возможно являющаяся символом руки, означает пять. Две V образуют X, означающую десять. L - это пятьдесят. Буква C происходит от слова *centum*, по-латыни «сто». D означает пятьсот. Наконец, M происходит от латинского *mille*, то есть «тысяча». Сделав тысячу шагов левой и правой ногой, вы пройдёте примерно одну милю.

Хотя мы, возможно, с этим не согласимся, долгое время считалось, что римские числа легко складывать и вычитать, поэтому в Европе они так долго сохранялись в бухгалтерском учёте. Действительно, при сложении двух римских чисел нужно просто объединить все их символы, а затем упростить результат по нескольким правилам: пять I образуют V, две V образуют X, пять X образуют L и так далее.

Но умножать и делить римские числа трудно. Многие другие ранние системы счисления, например древнегреческая, столь же плохо подходят для сложной работы с числами. Древние греки создали выдающуюся геометрию, которую и сегодня преподают в школах практически без изменений, но своей алгеброй они не прославились.

Современная система счисления известна как *индийско-арабская*, или *индо-арабская*. Она возникла в Индии, но в Европу её принесли арабские математики. Особенно знаменит персидский математик Мухаммад ибн Муса аль-Хорезми, от имени которого произошло слово «алгоритм». Около 820 года н. э. он написал книгу по алгебре, где использовалась индийская система счёта. Её латинский перевод, датируемый примерно 1145 годом н. э., заметно ускорил переход Европы от римских цифр к современной индийско-арабской системе.

Индийско-арабская система счисления отличается от предыдущих тремя особенностями:

• Индийско-арабская система называется *позиционной*: одна и та же цифра обозначает разное количество в зависимости от своего места в числе. *Место* цифр в числе в действительности важнее самих цифр! И в 100, и в 1,000,000 содержится лишь одна единица, но все мы знаем, что миллион намного больше сотни.

• Почти во всех ранних системах счисления есть то, чего *нет* в индийско-арабской: специальный символ для числа десять. В нашей системе отдельного символа для десяти нет.

• С другой стороны, почти во всех ранних системах отсутствует то, что имеется в индийско-арабской и оказывается гораздо важнее символа десяти. Это ноль.

Да, ноль. Скромный ноль, несомненно, является одним из важнейших изобретений в истории чисел и математики. Он поддерживает позиционную запись, поскольку позволяет сразу увидеть разницу между 25, 205 и 250. Кроме того, ноль упрощает многие математические операции, неудобные в непозиционных системах, особенно умножение и деление.

Вся структура индийско-арабских чисел раскрывается в том, как мы их произносим. Возьмём, например, 4825. Мы говорим: «Четыре тысячи восемьсот двадцать пять». Это означает:

```
четыре тысячи
восемь сотен
два десятка и
пять.
```

Компоненты можно записать так:

```
4825 = 4000 + 800 + 20 + 5
```

Разложив число ещё подробнее, получим:

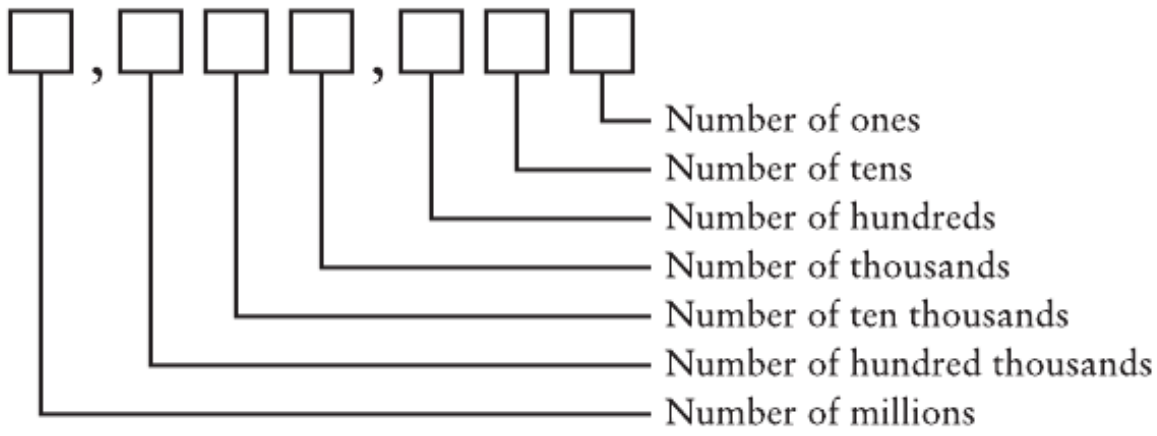
```
4825 = 4 × 1000 +
       8 × 100 +
       2 × 10 +
       5 × 1
```

А при помощи степеней десяти число записывается так:

```
4825 = 4 × 10^3 +
       8 × 10^2 +
       2 × 10^1 +
       5 × 10^0
```

Помните, что любое число в степени 0 равно 1.

Каждая позиция в многозначном числе имеет определённый смысл. Семь показанных ячеек позволяют представить любое число от 0 до 9,999,999:



Поскольку каждой позиции соответствует степень десяти, специальный символ для десяти не нужен: десять представляется единицей в другой позиции и нулём в качестве заполнителя.

Особенно удобно, что дробные величины, записанные цифрами справа от десятичной точки, следуют той же схеме. Число 42,705.684 представляет собой:

```
4 × 10,000 +
2 × 1000 +
7 × 100 +
0 × 10 +
5 × 1 +
6 ÷ 10 +
8 ÷ 100 +
4 ÷ 1000
```

Обратите внимание, что в трёх последних строках используется деление, а не умножение. То же число можно записать без деления:

```
4 × 10,000 +
2 × 1000 +
7 × 100 +
0 × 10 +
5 × 1 +
6 × 0.1 +
8 × 0.01 +
4 × 0.001
```

А при помощи степеней десяти:

```
4 × 10^4 +
2 × 10^3 +
7 × 10^2 +
0 × 10^1 +
5 × 10^0 +
6 × 10^-1 +
8 × 10^-2 +
4 × 10^-3
```

Показатели степеней уменьшаются до нуля, а затем становятся отрицательными числами. Наша система счисления настолько привычна, что мы редко замечаем изящество лежащей в её основе структуры.

Мы знаем, что 3 плюс 4 равно 7. Точно так же 30 плюс 40 равно 70, 300 плюс 400 равно 700, а 3000 плюс 4000 равно 7000. В этом красота индийско-арабской системы. Складывая десятичные числа любой длины, вы следуете процедуре, разбивающей задачу на отдельные шаги. На каждом шаге не требуется ничего сложнее сложения пары однозначных чисел. Именно поэтому когда-то давно вас заставили выучить 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 два слагаемых в верхней строке и левом столбце. Проведите от них линии вниз и вправо, чтобы получить сумму. Например, 4 плюс 6 равно 10.

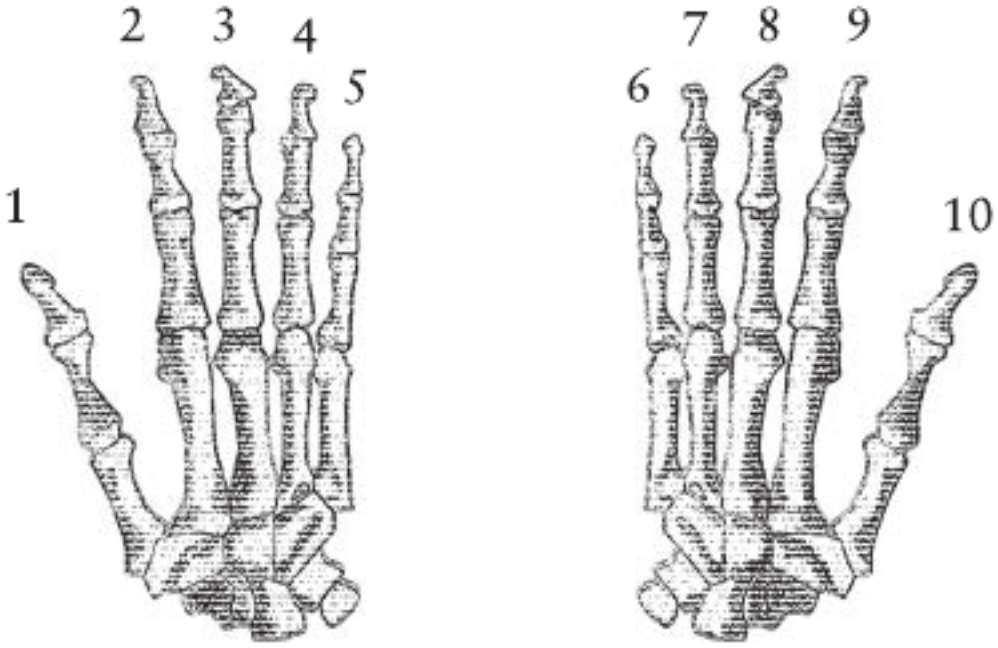
Точно так же при умножении двух десятичных чисел вы следуете несколько более сложной процедуре, которая всё равно разбивает задачу так, что от вас не требуется ничего сложнее сложения или умножения однозначных десятичных чисел. Вероятно, в начальной школе вам также пришлось выучить таблицу умножения:

×	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

Лучшее свойство позиционной записи состоит не просто в том, насколько хорошо она работает, а в том, насколько хорошо она работает для систем счёта *не* с основанием десять. Наша система вовсе не обязательно подходит всем. Одна большая проблема десятичной системы заключается в том, что она не имеет никакого отношения к персонажам мультфильмов. У большинства мультяшных персонажей на каждой руке или лапе только по четыре пальца, поэтому они предпочитают систему счисления с основанием восемь. Любопытно, что многое из наших знаний о десятичной записи применимо и к системе счисления, более подходящей нашим друзьям из мультфильмов.

Глава 10. Альтернативные десятки

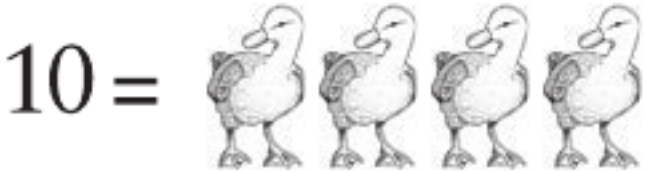
Десять - исключительно важное для людей число. У большинства из нас по десять пальцев на руках и ногах, и мы, несомненно, предпочитаем иметь полный комплект тех и других. Поскольку пальцами удобно считать, люди создали целую систему счисления, основанную на числе десять.



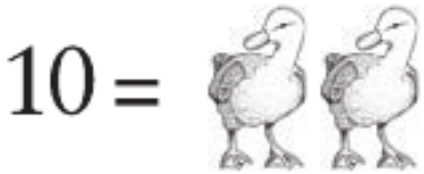
Как говорилось в предыдущей главе, наша обычная система счисления называется системой *с основанием десять*, или *десятичной*. Десятичные числа кажутся нам настолько естественными, что поначалу трудно представить альтернативы. Действительно, увидев 10, мы невольно думаем, что оно обозначает столько уток:

Но единственная причина, по которой запись 10 обозначает столько уток, заключается в том, что уток столько же, сколько у нас пальцев. Если бы у людей было другое количество пальцев, мы считали бы иначе и 10 означало бы что-нибудь другое. Та же самая запись 10 могла бы обозначать столько уток:

Или столько уток:



Или даже столько:



Когда мы дойдём до того, что 10 будет означать всего две утки, то будем готовы рассмотреть, как переключатели, провода и лампочки могут представлять числа и как реле и логические вентили, а следовательно и компьютеры, могут этими числами манипулировать.

Что, если бы у людей, как у персонажей мультфильмов, было только по четыре пальца на каждой руке? Вероятно, нам никогда не пришло бы в голову создавать систему счисления с основанием десять. Вместо этого мы считали бы совершенно нормальным, естественным, разумным, неизбежным, неопровержимым и бесспорно правильным основывать систему счисления на восьми. Такая система называется *восьмеричной*, или системой *с основанием восемь*.

Если бы наша система строилась вокруг восьми, а не десяти, нам не понадобился бы символ

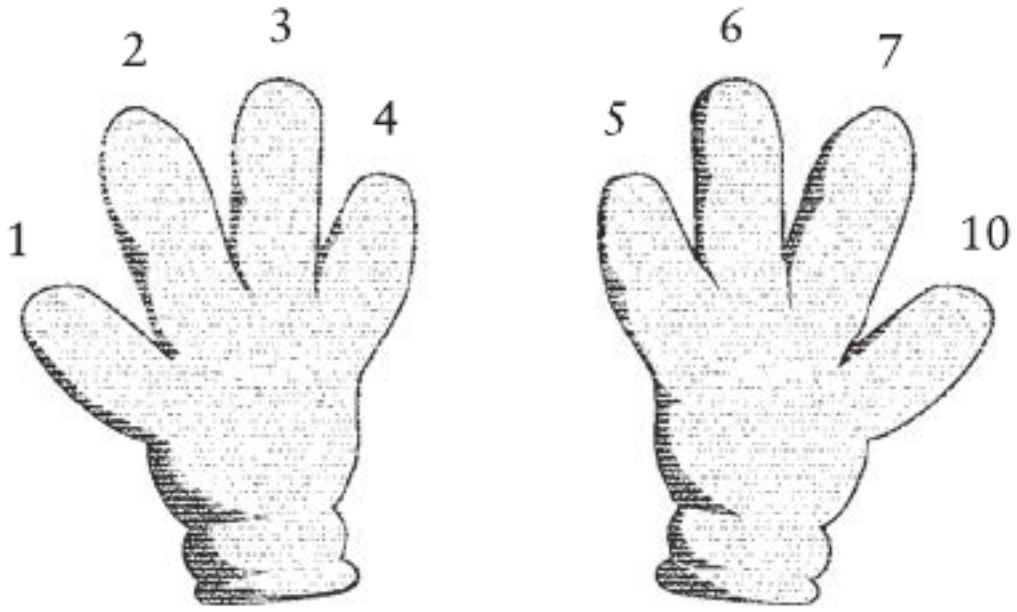
```
9
```

Покажите его любому мультяшному персонажу и услышите: «Что это? Для чего он?» А немного подумав, мы поймём, что нам не понадобился бы и символ

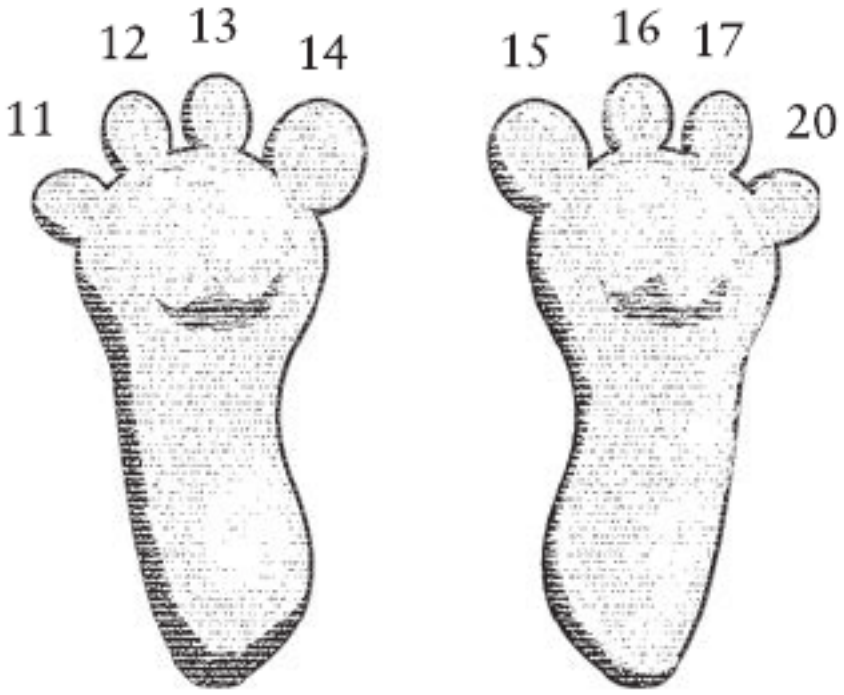
```
8
```

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

В десятичной системе мы считаем 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, а затем 10. В восьмеричной системе счёт идёт 0, 1, 2, 3, 4, 5, 6, 7, а затем что? Символы закончились. Единственный разумный ответ - 10, и он верен. В восьмеричной системе следующее число после 7 - это 10. Но такое 10 означает не количество пальцев у людей, а количество пальцев у мультяшных персонажей.



Продолжить счёт можно на четырёхпалых ногах:



Работая с системами счисления, отличными от десятичной, можно избежать путаницы, если произносить запись вроде 10 как «один ноль». Аналогично 13 произносится как «один три», а 20 - «два ноль». Чтобы выражаться точнее и *действительно избежать неоднозначности*, можно сказать «один три по основанию восемь» или «два ноль в восьмеричной системе».

Хотя пальцы на руках и ногах закончились, мы всё равно можем продолжать считать в восьмеричной системе. В основном это то же самое, что десятичный счёт, только мы пропускаем все числа, содержащие 8 или 9:

```
0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22,
23, 24, 25, 26, 27, 30, 31, 32, 33, 34, 35, 36, 37, 40, 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 произносится «один ноль ноль». Это количество пальцев у мультяшных персонажей, умноженное само на себя.

Почти целая жизнь знакомства с десятичными числами приучила нас ожидать, что определённые последовательности цифр соответствуют конкретным количествам в реальном мире. Считать в другой системе - всё равно что попасть в совершенно иной мир. Вот несколько примеров восьмеричных чисел:

• Белоснежка встречает 7 гномов, как и в десятичной системе.
• У мультяшных персонажей 10 пальцев.
• Бетховен написал 11 симфоний.
• У человека 12 пальцев на руках.
• В году 14 месяцев.

Если вы мысленно переводите эти восьмеричные числа в десятичные, это прекрасно. Такое упражнение полезно. Для двузначного восьмеричного числа, начинающегося с 1, десятичный эквивалент равен 8 плюс вторая цифра. Число месяцев в восьмеричной системе равно 14, а в десятичной - 8 плюс 4, то есть 12. Продолжим:

• В пекарской дюжине 15 предметов.
• В двух неделях 16 дней.
• Праздник «сладких шестнадцати» наступает в 20 лет.
• В сутках 30 часов.
• В латинском алфавите 32 буквы.

Если первая цифра двузначного восьмеричного числа отличается от 1, преобразование в десятичную систему выполняется немного иначе: первую цифру нужно умножить на 8 и прибавить вторую. Число букв в алфавите в восьмеричной системе равно 32, поэтому в десятичной оно равно 3 умножить на 8, то есть 24, плюс 2, что даёт 26.

• В кварте 40 жидких унций.
• В колоде карт 4 умножить на 15, то есть 64 карты.
• На шахматной доске 10 умножить на 10, то есть 100 клеток.

В десятичной системе число клеток шахматной доски равно 8 умножить на 8, то есть 64.

• Длина поля для американского футбола равна 144 ярдам.
• В одиночном женском разряде Уимблдона начинают 200 участниц.
• В восьмиточечном шрифте Брайля 400 символов.

В этом списке есть несколько приятных круглых восьмеричных чисел: 100, 200 и 400. Под *круглым числом* обычно понимают число с несколькими нулями в конце. Два нуля на конце десятичного числа означают, что оно кратно 100, то есть 10 умножить на 10. В восьмеричном числе два конечных нуля также означают кратность 100, но восьмеричное 100 равно десятичному 64. Число участниц Уимблдона равно 128 в десятичной системе, а число символов восьмиточечного Брайля - 256.

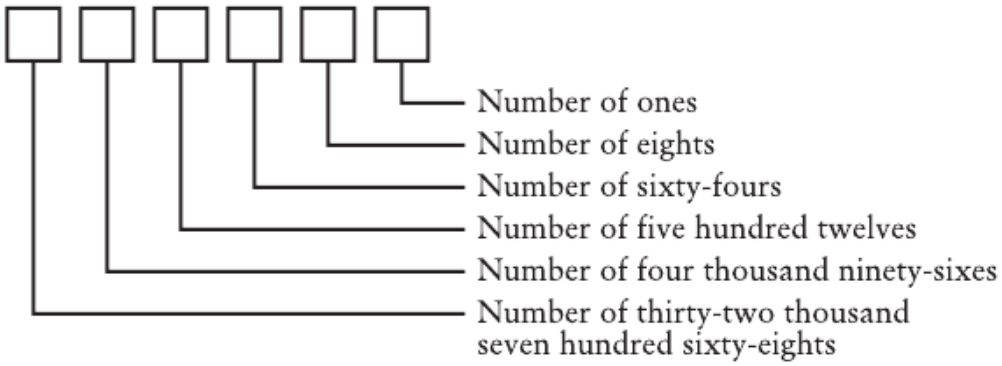
В первых трёх главах книги рассматривалось, как двоичные коды связаны со степенями двойки. Число кодов Морзе из четырёх точек и тире равно 2 в 4-й степени, то есть 16. Число шеститочечных кодов Брайля равно 2 в 6-й степени, то есть 64. Восьмиточечный Брайль увеличивает это число до 2 в 8-й степени, то есть 256. При умножении одной степени двойки на другую результат также является степенью двойки.

Следующая таблица показывает первые 12 степеней двойки и их десятичную и восьмеричную записи:

Степень двойки	Десятичная	Восьмеричная
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

Поскольку восемь является степенью двойки, в восьмеричном столбце много приятных круглых чисел. Это указывает на более тесную связь с двоичными кодами, чем возможна для десятичных чисел.

По структуре восьмеричная система ничем не отличается от десятичной. Различаются лишь детали. Например, каждая позиция восьмеричного числа содержит цифру, умноженную на степень восьми:



Таким образом, восьмеричное число 3725 можно разложить так:

```
3725 = 3000 + 700 + 20 + 5
```

То же число можно выразить отдельными цифрами, умноженными на восьмеричные степени восьми:

```
3725 = 3 × 1000 +
       7 × 100 +
       2 × 10 +
       5 × 1
```

А вот другая форма той же записи:

```
3725 = 3 × 8^3 +
       7 × 8^2 +
       2 × 8^1 +
       5 × 8^0
```

Если выполнить эти вычисления в десятичной системе, получится 2005. Так восьмеричные числа преобразуются в десятичные.

Восьмеричные числа можно складывать и умножать теми же способами, что и десятичные. Единственное настоящее различие состоит в других таблицах сложения и умножения отдельных цифр. Вот таблица сложения восьмеричных чисел:

+	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

Например, 5 + 7 = 14. Более длинные восьмеричные числа складываются так же, как десятичные. Вот небольшое упражнение, внешне ничем не отличающееся от десятичного сложения, хотя числа в нём восьмеричные. Пользуясь приведённой таблицей, сложите цифры каждого столбца:

```
  135
+ 643
```

Сумма цифр в каждом столбце больше восьмеричной 7, поэтому из каждого столбца выполняется перенос в следующий. Результат равен 1000.

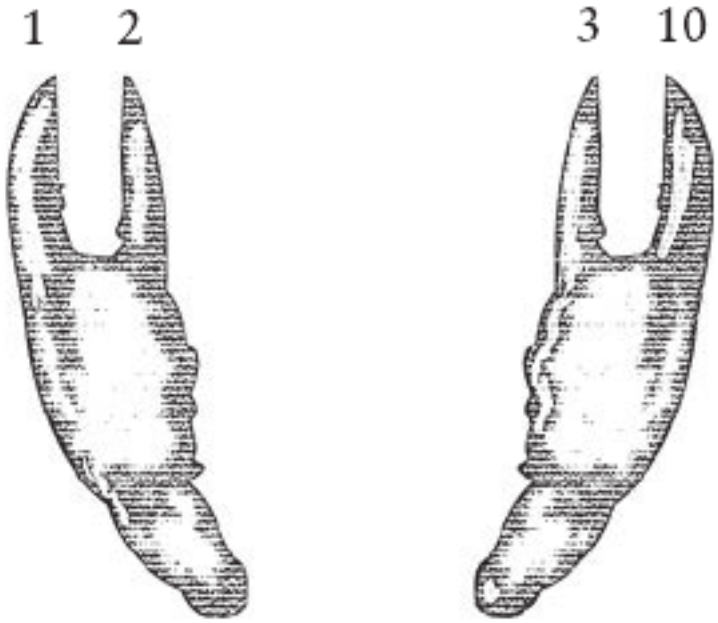
Точно так же 2 умножить на 2 в восьмеричной системе по-прежнему равно 4. Но 3 умножить на 3 не равно 9. Как оно могло бы равняться 9? Вместо этого 3 умножить на 3 равно 11. Полная таблица восьмеричного умножения выглядит так:

×	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

×	0	1	2	3	4	5	6	7
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

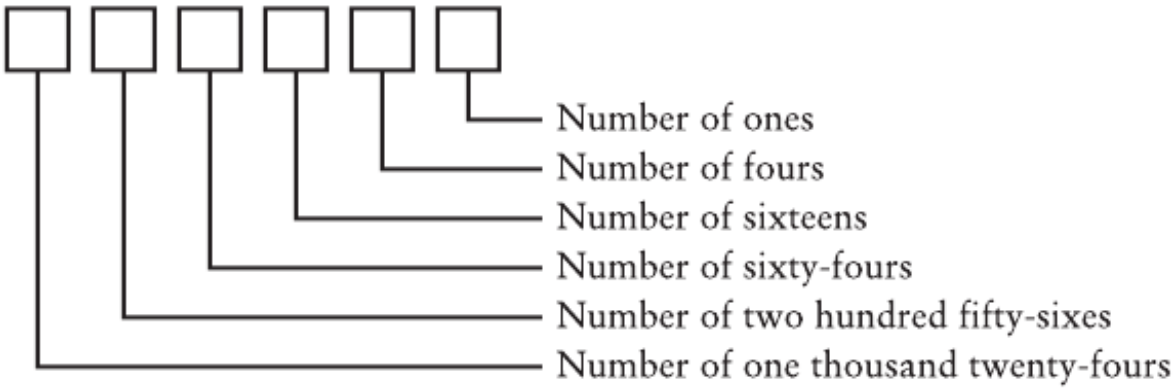
Из таблицы видно, что 4 × 6 равно 30, то есть 24 в десятичной системе.

Восьмеричная система столь же полноценна, как десятичная. Но пойдём дальше. Теперь, создав систему счисления для мультяшных персонажей, разработаем систему, подходящую омарам. В точном смысле пальцев у омаров нет, но у омаров вида *Homarus americanus* на концах двух длинных передних ног есть клешни. Подходящей системой счисления для омаров станет *четверичная*, или система *с основанием четыре*:



В четверичной системе считают так: 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110, 111, 112, 113, 120 и так далее.

Я не стану долго задерживаться на четверичной системе, потому что вскоре мы перейдём к чему-то гораздо более важному. Но здесь видно, что каждая позиция четверичного числа соответствует степени четырёх:



Четверичное число 31232 можно записать так:

```
31232 = 3 × 10000 +
        1 × 1000 +
        2 × 100 +
        3 × 10 +
        2 × 1
```

Каждая цифра умножается на степень четырёх:

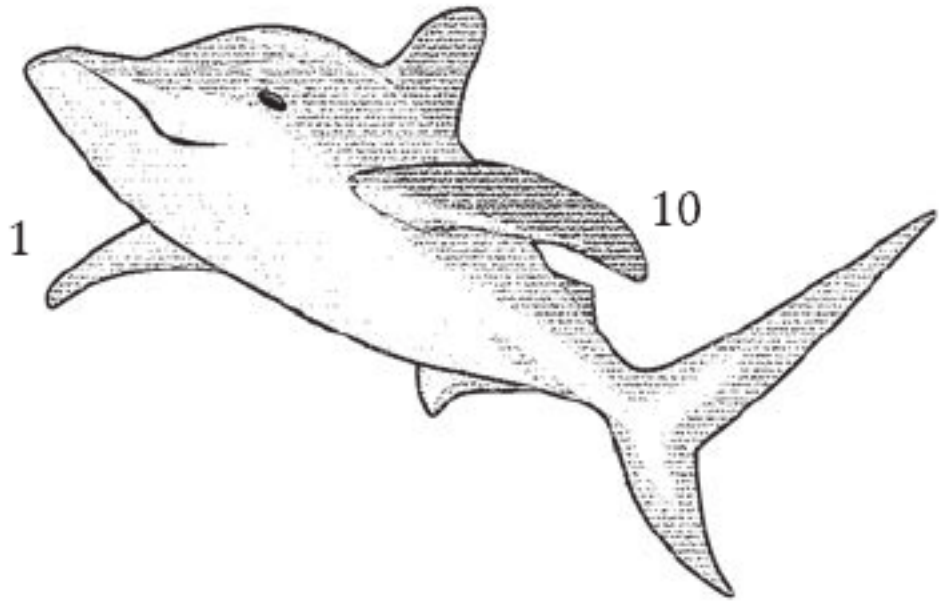
```
31232 = 3 × 4^4 +
        1 × 4^3 +
        2 × 4^2 +
        3 × 4^1 +
        2 × 4^0
```

Выполнив вычисления в десятичной системе, вы обнаружите, что четверичное 31232 равно десятичному 878.

Теперь мы совершим ещё один, на этот раз предельный, скачок. Представим, что мы дельфины и вынуждены считать на двух плавниках. Такая система известна как система *с основанием два*, или *двоичная* система, от латинского выражения «по два». Очевидно, в ней будет всего две цифры - 0 и 1.

Вы уже видели, как 1 и 0 применяются в булевой алгебре для обозначения истины и лжи, «да» и «нет», подходящей и не вполне подходящей кошки. Те же две цифры можно использовать для счёта.

Ноль и единица дают не так уж много возможностей, поэтому к двоичным числам приходится привыкать. Главная проблема в том, что цифры заканчиваются очень быстро. Вот, например, как дельфин считает на плавниках:



Да, в двоичной системе после 1 следует 10. Это поражает, хотя на самом деле не должно удивлять. Какую бы систему счисления мы ни использовали, после исчерпания однозначных чисел первым двузначным числом всегда становится 10. В двоичной системе считают так:

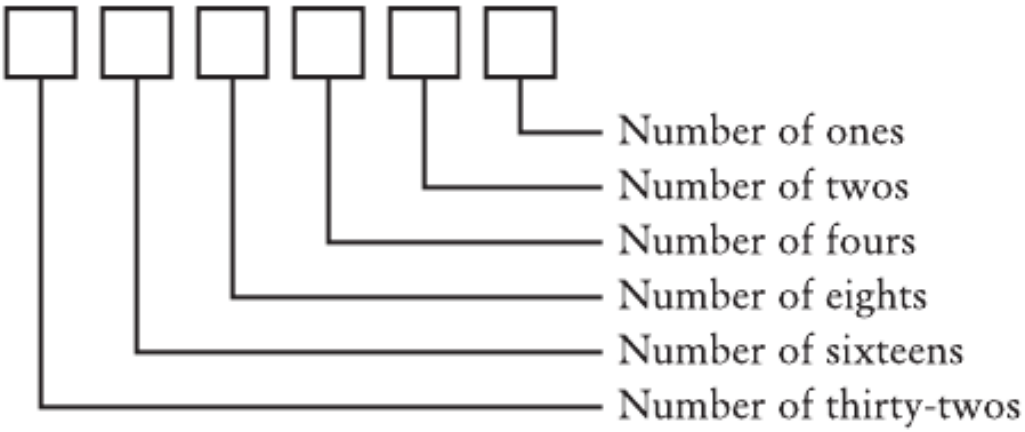
```
0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100,
1101, 1110, 1111, 10000, 10001...
```

Эти числа могут выглядеть большими, но в действительности они невелики. Точнее будет сказать, что двоичные числа очень быстро становятся *длинными*, а не большими:

• У человека 1 голова.
• У дельфина 10 плавников.
• В столовой ложке 11 чайных ложек.
• У квадрата 100 сторон.
• На одной человеческой руке 101 палец.
• У насекомого 110 ног.
• В неделе 111 дней.
• В октете 1000 музыкантов.
• В бейсбольном матче 1001 иннинг.
• В ковбойской шляпе 1010 галлонов.

И так далее.

В многозначном двоичном числе позиции цифр соответствуют степеням двойки:



Таким образом, всякий раз, когда двоичное число состоит из единицы и следующих за ней нулей, оно является степенью двойки, причём показатель степени равен числу нулей. Вот расширенная таблица степеней двойки, демонстрирующая это правило:

Степень двойки	Десятичная	Восьмеричная	Четверичная	Двоичная
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
2^5	32	40	200	100000
2^6	64	100	1000	1000000
2^7	128	200	2000	10000000
2^8	256	400	10000	100000000

Степень двойки	Десятичная	Восьмеричная	Четверичная	Двоичная
2^9	512	1000	20000	1000000000
2^10	1024	2000	100000	10000000000
2^11	2048	4000	200000	100000000000
2^12	4096	10000	1000000	1000000000000

Предположим, нам встретилось двоичное число 101101011010. Его можно записать так:

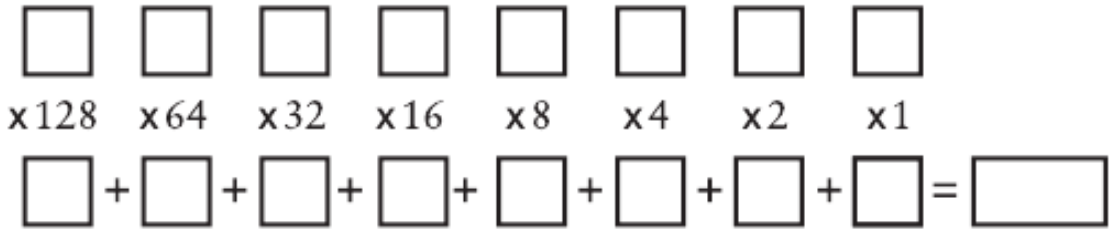
```
101101011010 = 1 × 100000000000 +
               0 × 10000000000 +
               1 × 1000000000 +
               1 × 100000000 +
               0 × 10000000 +
               1 × 1000000 +
               0 × 100000 +
               1 × 10000 +
               1 × 1000 +
               0 × 100 +
               1 × 10 +
               0 × 1
```

То же число можно записать проще, используя степени двойки:

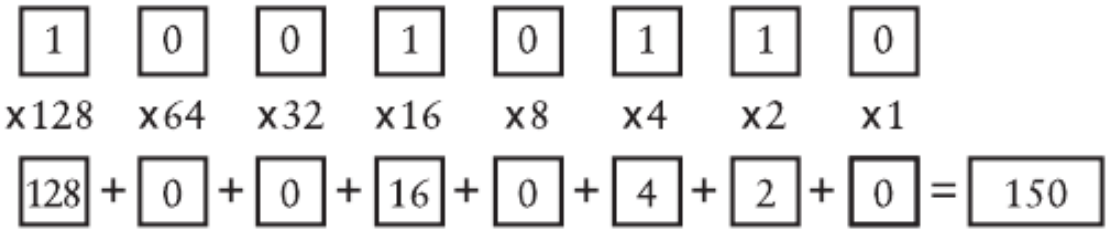
```
101101011010 = 1 × 2^11 +
               0 × 2^10 +
               1 × 2^9 +
               1 × 2^8 +
               0 × 2^7 +
               1 × 2^6 +
               0 × 2^5 +
               1 × 2^4 +
               1 × 2^3 +
               0 × 2^2 +
               1 × 2^1 +
               0 × 2^0
```

Если просто сложить части в десятичной системе, получится 2048 + 512 + 256 + 64 + 16 + 8 + 2, то есть 2906. Это десятичный эквивалент двоичного числа.

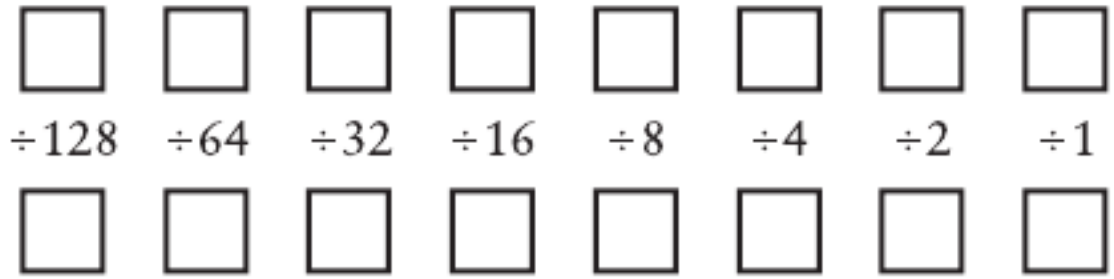
Для более краткого преобразования двоичных чисел в десятичные можно воспользоваться подготовленным шаблоном:



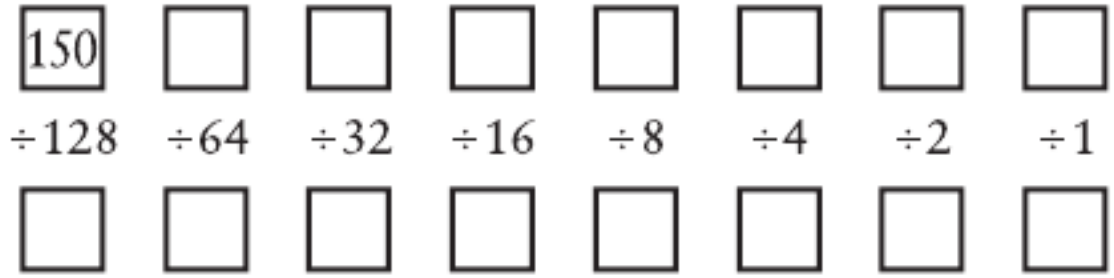
Шаблон позволяет преобразовывать двоичные числа длиной до восьми цифр, но его легко расширить. Поместите двоичные цифры в восемь верхних ячеек, по одной в каждую. Выполните восемь умножений и запишите произведения в нижних ячейках. Сложите содержимое восьми ячеек, чтобы получить окончательный результат. В примере показано, как найти десятичный эквивалент числа 10010110:



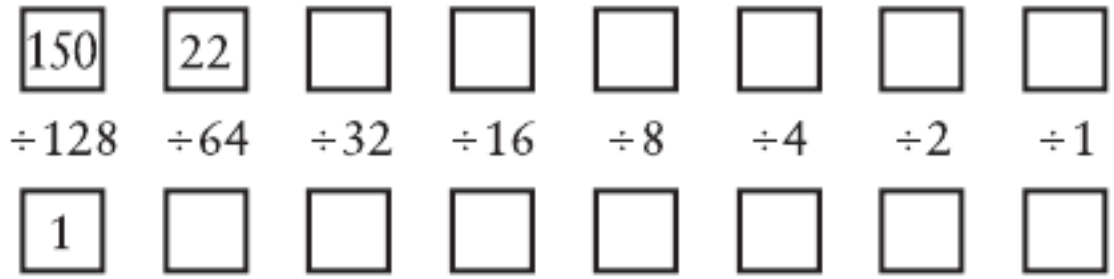
Преобразование десятичного числа в двоичное не столь прямолинейно, но следующий шаблон позволяет переводить в двоичную систему десятичные числа от 0 до 255:



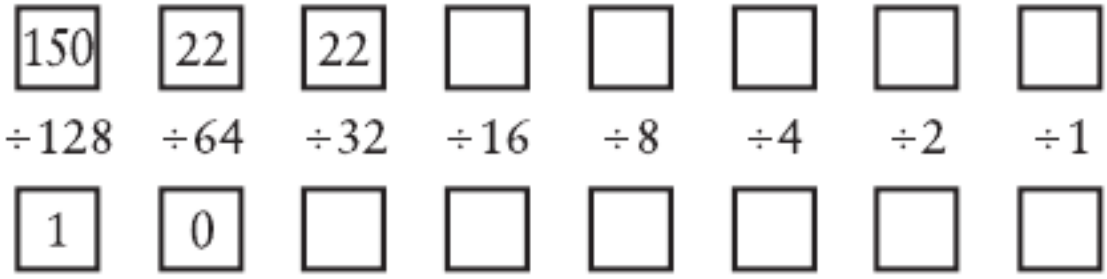
Преобразование сложнее, чем может показаться. Сначала поместите всё десятичное число, не превышающее 255, в левую верхнюю ячейку:



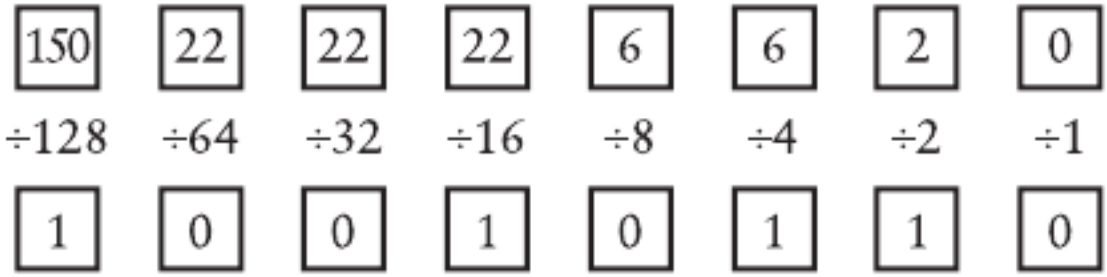
Разделите это число на 128, но только до получения частного и остатка: 150 при делении на 128 даёт 1 с остатком 22. Запишите частное в первую нижнюю ячейку, а остаток - в следующую верхнюю:



Теперь разделите 22 на 64, снова выполнив лишь первый шаг. Поскольку 22 меньше 64, частное равно 0, а остаток - 22. Запишите 0 во вторую нижнюю ячейку и перенесите остаток в следующую верхнюю:



Таким же образом пройдите весь шаблон. Каждое частное будет равно 0 или 1, поэтому после завершения нижние ячейки образуют последовательность двоичных цифр:



Двоичный эквивалент числа 150 - это 10010110.

Преобразования между десятичными и двоичными числами, несомненно, неудобны. Поэтому, если вам когда-нибудь понадобится выполнять их на практике, вы обрадуетесь, узнав, что калькуляторы Windows и macOS имеют режимы программиста, которые сделают это за вас.

Первые цифровые компьютеры не всегда использовали двоичные числа. Некоторые из самых ранних машин проектировались и строились для работы с привычными десятичными числами. Аналитическая машина, которую английский математик Чарльз Бэббидж (1791-1871) начал проектировать в 1830-х годах, хранила десятичные числа посредством положений зубчатых колёс. К сожалению, построить эту машину ему так и не удалось. Некоторые ранние работавшие цифровые компьютеры, например Harvard Mark I, впервые запущенный в 1944 году, и ENIAC 1946 года, тоже были рассчитаны на десятичные числа. Архитектура некоторых компьютеров IBM, выпускавшихся вплоть до 1960-х годов, также основывалась на десятичных числах.

Но сильнее всего цифровую революцию характеризуют именно двоичные кодировки. Простота двоичных чисел, вероятно, яснее всего проявляется в основных операциях сложения и умножения. Эта часть вам *действительно* понравится. Представьте, как быстро вы освоили бы сложение, если бы требовалось запомнить только такую таблицу:

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

Воспользуемся таблицей, чтобы сложить два двоичных числа:

```
   1100101
 + 0110110
 ----------
  10011011
```

Начнём с крайнего правого столбца: 1 плюс 0 равно 1. Второй столбец справа: 0 плюс 1 равно 1. Третий: 1 плюс 1 равно 0, переносим 1. Четвёртый: перенесённая 1 плюс 0 плюс 0 равно 1. Пятый: 0 плюс 1 равно 1. Шестой: 1 плюс 1 равно 0, переносим 1. Седьмой: перенесённая 1 плюс 1 плюс 0 равно 10.

Таблица умножения ещё проще таблицы сложения, поскольку полностью выводится из двух самых основных правил: умножение чего угодно на 0 даёт 0, а умножение любого числа на 1 не изменяет это число.

×	0	1
0	0	0
1	0	1

Вот умножение десятичного числа тринадцать, то есть двоичного 1101, на десятичное одиннадцать, то есть двоичное 1011. Я не стану показывать все шаги, но процесс совпадает с десятичным умножением:

```
      1101
    × 1011
    ------
      1101
     1101
    0000
   1101
   --------
   10001111
```

Десятичный результат равен 143.

Люди, работающие с двоичными числами, часто записывают их с ведущими нулями, то есть нулями слева от первой единицы: например, 0011 вместо просто 11. Значение числа от этого совершенно не меняется; это делается лишь ради внешнего вида. Вот первые 16 двоичных чисел и их десятичные эквиваленты:

Двоичная	Десятичная
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11

Двоичная	Десятичная
1100	12
1101	13
1110	14
1111	15

Ненадолго остановимся и изучим этот список двоичных чисел. Рассмотрите каждый из четырёх вертикальных столбцов нулей и единиц и обратите внимание, как цифры чередуются сверху вниз:

• Крайняя правая цифра чередуется между 0 и 1.
• Следующая справа цифра чередуется между двумя нулями и двумя единицами.
• Следующая цифра чередуется между четырьмя нулями и четырьмя единицами.
• Следующая цифра чередуется между восемью нулями и восемью единицами.

Весьма методично, не правда ли? В действительности настолько методично, что можно создать цепь, автоматически генерирующую последовательности двоичных чисел. Мы доберёмся до неё в главе 17.

Более того, следующие 16 двоичных чисел легко записать, повторив первые 16 и поставив перед ними 1:

Двоичная	Десятичная
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

На это можно взглянуть и иначе. При двоичном счёте крайняя правая цифра, называемая также *младшим разрядом*, чередуется между 0 и 1. Каждый раз, когда она меняется с 1 на 0, цифра, расположенная второй справа, то есть следующий по значимости разряд, также меняется: с 0 на 1 или с 1 на 0. В общем случае каждый раз, когда двоичная цифра меняется с 1 на 0, следующий старший разряд тоже меняется с 0 на 1 либо с 1 на 0.

Двоичные числа очень быстро становятся длинными. Например, двенадцать миллионов в двоичной системе записываются как 101101110001101100000000. Один из способов выразить двоичные числа короче - записать их в восьмеричной системе. Это удобно, поскольку каждой тройке двоичных цифр соответствует одна восьмеричная:

Двоичная	**Восьмеричная**
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Возьмём, например, длинное двоичное число, обозначающее двенадцать миллионов, и разобьём его справа на группы по три цифры:

```
101 101 110 001 101 100 000 000
```

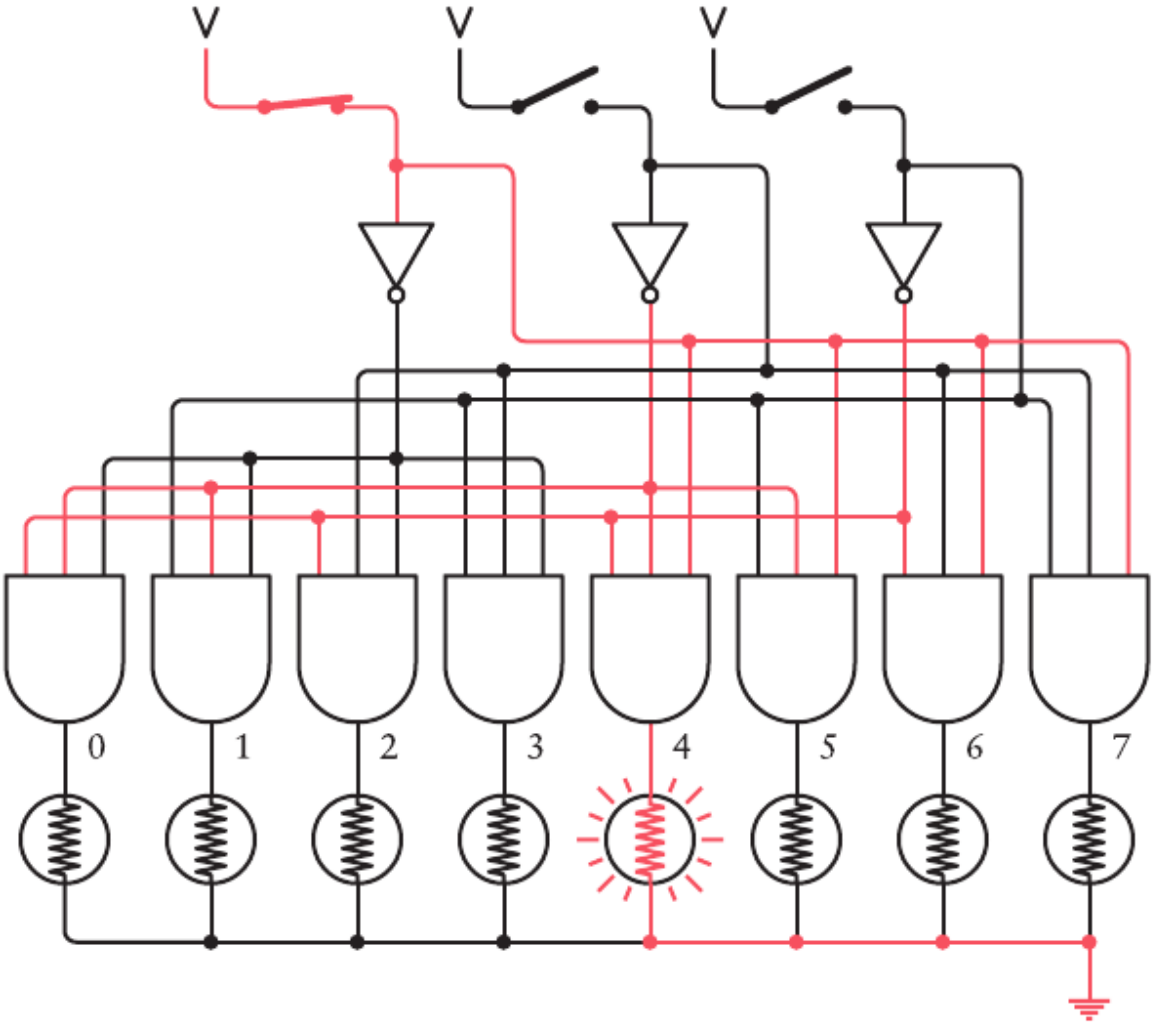
Каждая группа из трёх двоичных цифр соответствует одной восьмеричной:

```
101 101 110 001 101 100 000 000
 5   5   6   1   5   4   0   0
```

Десятичные двенадцать миллионов в восьмеричной системе равны 55615400. В главе 12 вы увидите ещё более краткий способ записи двоичных чисел.

Сократив систему счисления до двух двоичных цифр 0 и 1, мы дошли до предела. Сделать её ещё проще можно лишь возвращением к первобытным царапинам. Но важнее всего то, что двоичные числа позволяют объединить арифметику и электричество. Переключатели, провода и лампочки могут представлять двоичные цифры 0 и 1, а логические вентили позволяют манипулировать этими числами. Вот почему двоичные числа имеют *так много* общего с компьютерами.

Вы только что видели небольшую таблицу соответствия трёхзначных двоичных чисел их восьмеричным эквивалентам. Из переключателей, лампочек и логических вентилей можно построить цепь, выполняющую это преобразование:



На первый взгляд эта цепь, несомненно, выглядит ужасающе сложной, словно кошмарное переплетение автомагистралей в чужом городе, где невозможно прочитать ни одного дорожного знака. Но на самом деле она вполне методична. Маленькие точки показывают места соединения проводов. Если точки нет, провода не соединены, а просто пересекаются.

В верхней части цепи находятся три переключателя, представляющие трёхзначное двоичное число. Для 1 переключатель замкнут, для 0 разомкнут. В примере показано двоичное число 100. Внизу расположены восемь лампочек с метками от 0 до 7. В зависимости от замкнутых переключателей загорается ровно одна из них.

Вероятно, цепь легче понять снизу вверх. Каждую из восьми нижних лампочек питает трёхвходовый вентиль AND. Его выход равен 1 лишь тогда, когда все три входа равны 1. Три входа каждого вентиля AND соответствуют трём переключателям: иногда непосредственно, а иногда после инверсии сигнала одним из трёх инверторов прямо под переключателями. Напомним: если вход инвертора равен 0, выход равен 1, а если вход равен 1, выход равен 0.

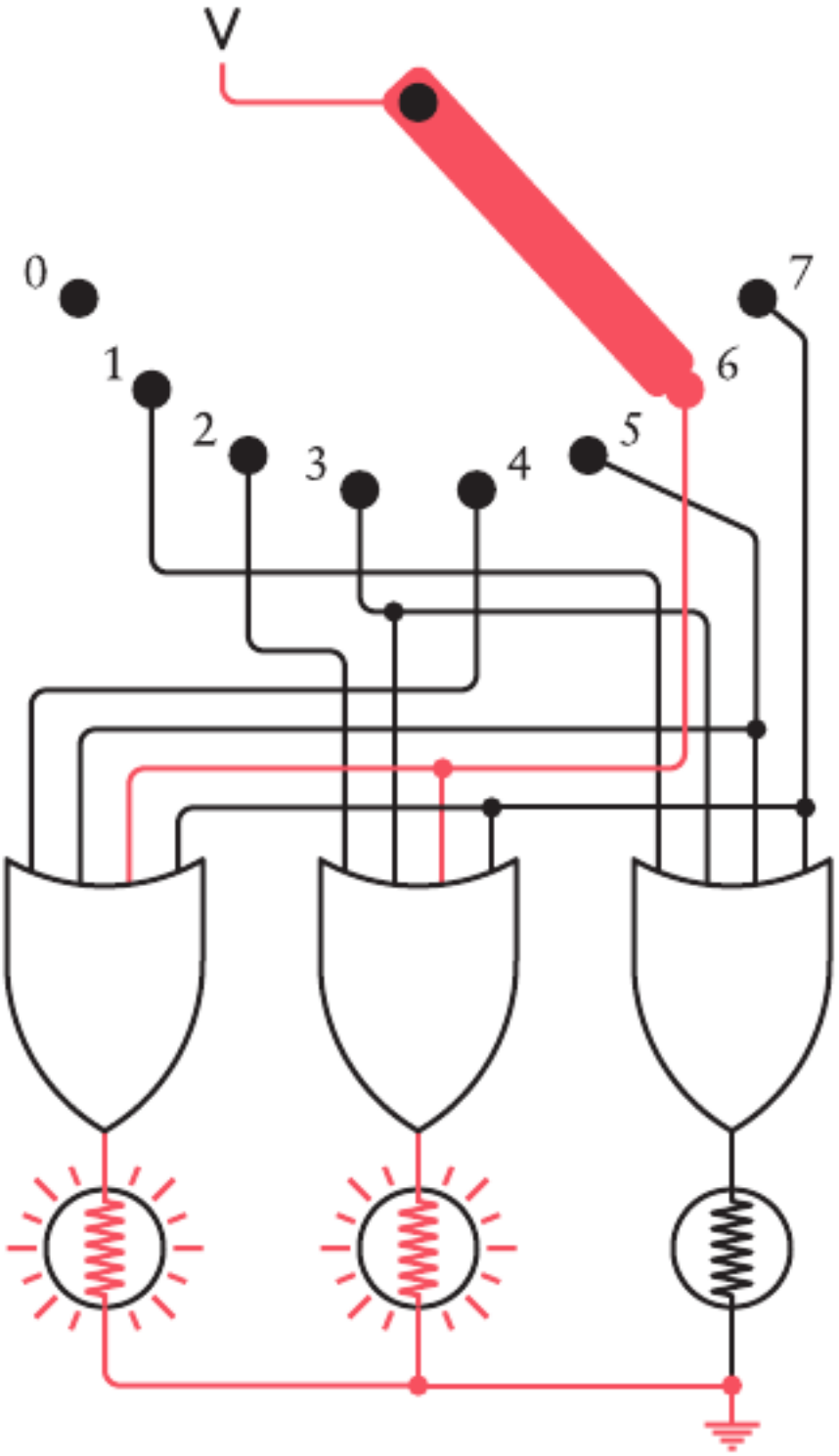
Три верхних переключателя показаны замкнутым, разомкнутым и разомкнутым, что обозначает двоичное число 100. Проследив красные линии, можно увидеть, что старший разряд 1 является

одним из входов вентиля AND, соответствующего восьмеричной цифре 4. Следующая цифра, то есть средний переключатель, инвертируется перед поступлением на тот же вентиль AND. Младшая цифра, правый переключатель, тоже инвертируется перед тем, как стать третьим входом вентиля. Таким образом, все три входа вентиля AND для восьмеричной цифры 4 равны 1, поэтому его выход равен 1.

Аналогично каждый из остальных семи вентилей AND получает другую комбинацию прямых или инвертированных сигналов переключателей.

Это небольшое устройство называется *дешифратором 3 в 8*. Название подразумевает, что трёхзначное двоичное число является *кодом*, представляющим одну из восьми возможностей.

Другая цепь, называемая *шифратором 8 в 3*, решает обратную задачу. Для неё создадим переключатель другого вида, позволяющий выбирать одно из восьми положений. В реальности такой переключатель можно сделать из канцелярских кнопок или гвоздей и полоски металла, вырезанной из консервной банки:



Каждая двоичная цифра внизу отображается лампочкой, которой управляет четырёхвходовый вентиль OR. Выход вентиля OR равен 1, если хотя бы один из четырёх входов равен 1. Когда верхний переключатель выбирает восьмеричную цифру 6, первый и второй вентили OR получают по входной единице, отчего их выходы становятся равны 1 и лампочки показывают двоичные цифры 110. Обратите внимание, что положение 0 переключателя вверху слева ни с чем не

соединено. Восьмеричное число 0 соответствует двоичному 000, поэтому ни одна лампочка не должна гореть.

Примерно в 1947 году американский математик Джон Уайлдер Тьюки (1915-2000) понял, что выражение *binary digit*, «двоичная цифра», вероятно, приобретёт гораздо большее значение в последующие годы по мере распространения компьютеров. Он решил придумать новое, короткое слово вместо громоздких пяти слогов *binary digit*. Он рассматривал варианты *bigit* и *binit*, но в итоге остановился на коротком, простом, изящном и совершенно прекрасном слове *bit* - «бит».

Глава 11. Бит за битом за битом

История, известная по крайней мере с 1950-х годов, рассказывает о человеке, возвращающемся домой после заключения в далёкой тюрьме. Он не знает, будут ли ему рады, поэтому просит подать знак - привязать к ветке дерева кусок ткани. В одной версии мужчина едет поездом к семье и надеется увидеть белую ленту на яблоне. В другой он едет автобусом к жене и ищет жёлтый платок на дубе. В обеих версиях, приехав, мужчина видит дерево, покрытое сотнями таких знаков, и у него не остаётся сомнений, что ему рады.

Историю сделала популярной вышедшая в 1973 году песня-хит *Tie a Yellow Ribbon Round the Ole Oak Tree* («Повяжи жёлтую ленту вокруг старого дуба»). С тех пор вывешивание жёлтой ленты стало также обычаем, когда члены семьи или любимые находятся на войне.

Человек, попросивший о жёлтой ленте, не ждал подробных объяснений или долгих обсуждений. Ему не нужны были никакие «если», «и» или «но». Несмотря на все сложные чувства и эмоциональную историю, ему требовался лишь простой ответ «да» или «нет». Жёлтая лента должна была означать: «Да, хотя ты ужасно ошибся и провёл три года в тюрьме, я всё равно хочу, чтобы ты вернулся ко мне домой». Отсутствие ленты означало: «Даже *не думай* здесь останавливаться».

Это две чёткие, взаимоисключающие альтернативы. Столь же действенным, как жёлтая лента, хотя, возможно, менее удобным для текста песни, оказался бы дорожный знак во дворе: например, «Въезд» или «Движение запрещено».

Или табличка на двери: «Открыто» или «Закрыто».

Или фонарик в окне: включённый либо выключенный.

Если требуется сказать только «да» или «нет», способов существует множество. Для этого не нужно предложение, не нужно слово и даже не нужна буква. Нужен лишь *бит*, то есть всего лишь 0 или 1.

Как вы узнали из двух предыдущих глав, в обычной десятичной системе счисления нет ничего особенно примечательного. Очевидно, мы основываем её на десяти, потому что у нас десять пальцев. С тем же успехом система могла бы основываться на восьми, будь мы мультяшными персонажами, на четырёх, будь мы омарами, или даже на двух, будь мы дельфинами.

В десятичной системе нет ничего особенного, но нечто особенное есть в двоичной, потому что это *простейшая* из возможных систем счисления. Двоичных цифр всего две - 0 и 1. Чтобы получить нечто ещё проще, пришлось бы избавиться от 1, оставив один 0, а с ним почти ничего нельзя сделать.

Слово *bit*, придуманное как сокращение от *binary digit*, «двоичная цифра», несомненно, одно из самых прекрасных слов, созданных в связи с компьютерами. Разумеется, у английского слова *bit* есть обычное значение: «небольшая часть, степень или количество». Оно подходит идеально, поскольку одна двоичная цифра и правда представляет собой очень малую величину.

Иногда изобретённое слово приобретает ещё одно значение. Именно так произошло здесь. Помимо двоичных цифр, которыми дельфины пользуются для счёта, в компьютерную эпоху бит стали считать *основным строительным элементом информации*.

Это смелое утверждение. Конечно, информацию передают не только биты. Её передают буквы, слова, азбука Морзе, шрифт Брайля и десятичные цифры. Особенность бита в том, что он передаёт *совсем немного информации*. Один бит информации - наименьшее возможное количество информации, даже если она столь же важна, как жёлтая лента. Меньше одного бита -

полное отсутствие информации. Но поскольку бит представляет наименьшее возможное её количество, более сложную информацию можно передать несколькими битами.

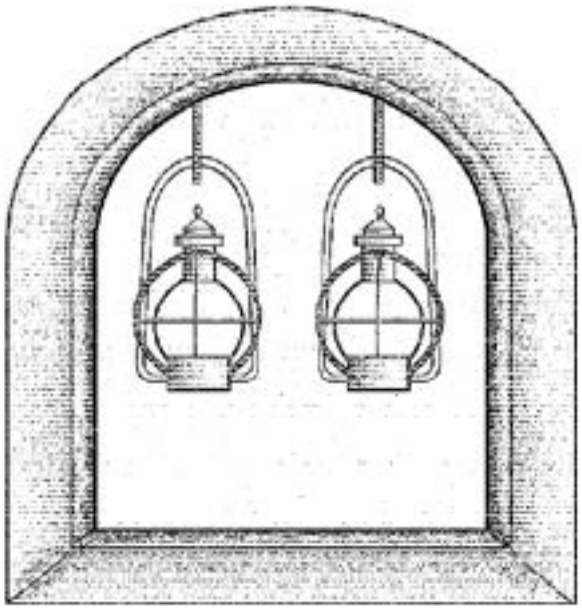
«Послушайте, дети мои, и вы услышите о полуночной скачке Пола Ревира», - писал Генри Уодсворт Лонгфелло. Возможно, его описание того, как Пол Ревир предупредил американские колонии о вторжении британцев, исторически неточно, но поэт привёл наводящий на размышления пример использования битов для передачи важной информации:

> Он другу сказал: «Коль британцы пойдут
> По суше иль морю из города в ночь,
> Под сводом звонницы свет пусть зажгут
> На башне Северной церкви точь-в-точь:
> Один, коль по суше, два, коль морем...»

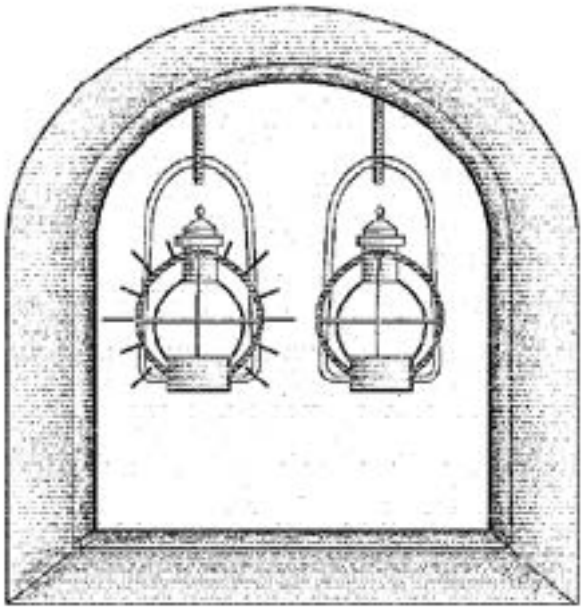
Итак, у друга Пола Ревира два фонаря. Если британцы наступают по суше, он поставит на церковной башне один фонарь. Если они идут морем - оба.

Однако Лонгфелло не упоминает явно все варианты. Он оставляет невысказанной третью возможность: британцы пока не вторгаются. Поэт подразумевает, что об этом сообщит *отсутствие* фонарей на башне.

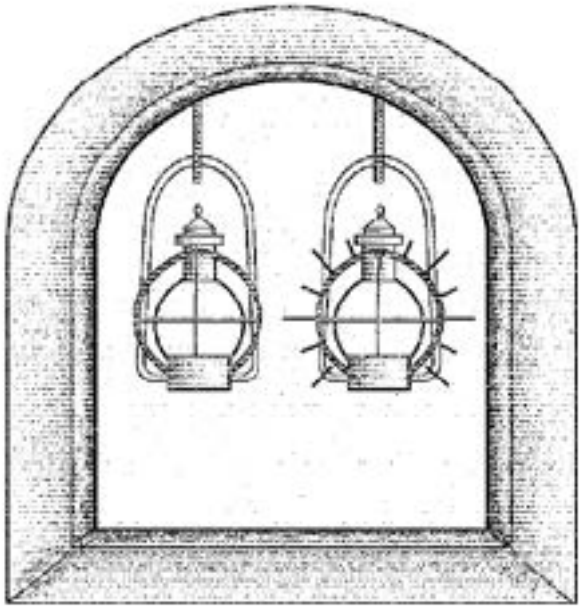
Предположим, что два фонаря постоянно установлены в церковной башне. Обычно они не горят:



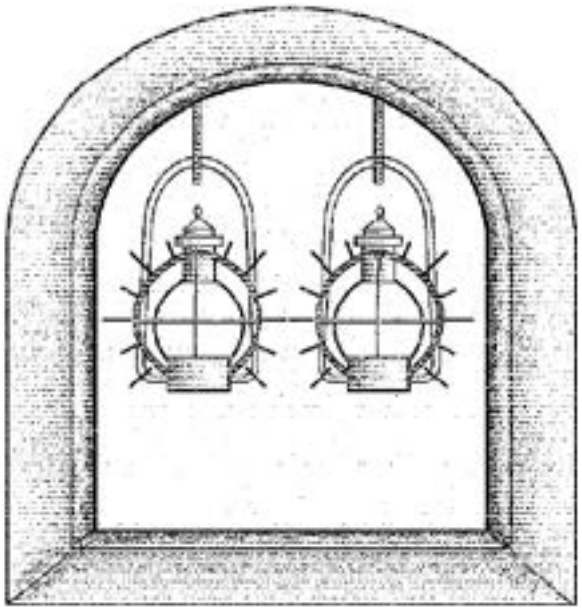
Это означает, что британцы пока не вторгаются. Если горит один из фонарей,



или



британцы идут по суше. Если горят оба,



британцы идут морем.

Каждый фонарь является битом и может быть представлен числом 0 или 1. История жёлтой ленты показывает, что для передачи одного из двух вариантов достаточно одного бита. Если бы Полу Ревиру требовалось знать только о самом вторжении, а не о направлении британцев, хватило бы одного фонаря. При вторжении он горел бы, а в очередной мирный вечер оставался погашенным.

Чтобы передать один из трёх вариантов, нужен ещё один фонарь. Но после появления второго два бита позволяют сообщить об одном из четырёх вариантов:

```
00 = Сегодня британцы не вторгаются.
01 = Они идут по суше.
10 = Они идут по суше.
11 = Они идут морем.
```

То, что Пол Ревир ограничился тремя вариантами, в действительности было весьма изощрённым решением. На языке теории связи он использовал *избыточность*, чтобы ослабить влияние *шума*. В теории связи шумом называется всё, что мешает коммуникации. Плохое мобильное соединение - очевидный пример шума, препятствующего телефонному разговору. Несмотря на шум, телефонное общение обычно остаётся успешным, потому что устная речь обладает большой избыточностью. Чтобы понять сказанное, нам необязательно слышать каждый слог каждого слова.

В случае фонарей на церковной башне шумом можно считать ночную темноту и расстояние от Пола Ревира до башни: и то и другое могло помешать ему различить фонари. Вот ключевой отрывок из стихотворения Лонгфелло:

> И вот, как он смотрит на башни вершину,
> Мерцание, вспышка - свет засиял!
> Он в седло, повернул уже повод коню,
> Но медлит и смотрит, пока не предстал
> Второй огонёк, что на башне пылал!

Совсем не похоже, чтобы Пол Ревир мог точно определить, какой из двух фонарей зажёгся первым.

Главная мысль здесь состоит в том, что информация представляет *выбор из двух или нескольких возможностей*. Разговаривая с другим человеком, мы выбираем каждое произносимое слово из всех слов словаря. Если бы все слова были пронумерованы от 1 до 351 482, мы могли бы с той же точностью беседовать числами вместо слов. Разумеется, обоим собеседникам понадобились бы словари с одинаковой нумерацией и большое терпение.

Обратная сторона этой мысли: любую информацию, которую можно свести к выбору из двух или нескольких возможностей, можно выразить *битами*. Разумеется, существует множество жизненно важных форм человеческого общения, которые не являются выбором из дискретных вариантов. Именно поэтому люди не заводят романтических отношений с компьютерами. Во всяком случае, будем на это надеяться. Если нечто невозможно выразить словами, изображениями или звуками, то эту информацию невозможно закодировать битами. Да и незачем.

Больше десяти лет в конце XX века кинокритики Джин Сискел и Роджер Эберт демонстрировали применение битов в своей телепрограмме *At the Movies*. После подробных рецензий они выносили окончательный вердикт поднятым или опущенным большим пальцем.

Если два больших пальца считать битами, они представляют четыре возможности:

```
00 = Обоим фильм не понравился.
01 = Сискелу не понравился; Эберту понравился.
10 = Сискелу понравился; Эберту не понравился.
11 = Обоим фильм понравился.
```

Первый бит принадлежит Сискелу: он равен 0, если фильм ему не понравился, и 1, если понравился. Аналогично второй бит принадлежит Эберту.

Во времена *At the Movies* друг мог спросить: «Что решили Сискел и Эберт о новом фильме *Невежливая встреча*?» Вместо «Сискел поднял палец, а Эберт опустил» или даже «Сискелу понравилось, Эберту нет» можно было просто ответить: «Один ноль», а после преобразования в четверичную систему - «два». Если друг знает, какой бит принадлежит Сискелу, какой Эберту, что 1 означает палец вверх, а 0 - вниз, ответ будет совершенно понятен. Но вы оба должны знать код.

Изначально можно было бы объявить, что 1 означает палец вниз, а 0 - вверх. Это может показаться нелогичным. Естественно считать 1 чем-то утвердительным, а 0 - противоположностью, но в действительности назначение совершенно произвольно. Единственное требование - все пользователи кода должны знать значения нуля и единицы.

Значение отдельного бита или набора битов всегда определяется контекстом. Смысл жёлтой ленты на конкретном дубе, вероятно, известен лишь тому, кто её повязал, и тому, кто должен увидеть. Измените цвет, дерево или дату - и получится бессмысленный лоскут ткани. Аналогично, чтобы извлечь полезную информацию из жестов Сискела и Эберта, необходимо как минимум знать, о каком фильме идёт речь.

Если во время просмотра *At the Movies* вы записывали фильмы и оценки Сискела и Эберта, то могли добавить ещё один бит для собственного мнения. Третий бит увеличивает число вариантов 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.
```

Преимущество представления этой информации битами состоит в уверенности, что учтены все возможности. Их может быть ровно восемь - ни больше ни меньше. Тремя битами можно сосчитать только от нуля до семи. Других трёхзначных двоичных чисел нет. Как вы узнали в конце предыдущей главы, эти числа можно записать также восьмеричными цифрами от 0 до 7.

Говоря о битах, мы часто указываем определённое *число* битов. Чем их больше, тем больше разных возможностей можно передать.

Разумеется, то же справедливо для десятичных чисел. Например, сколько существует телефонных кодов регионов? Код состоит из трёх десятичных цифр, и если использовать все трёхзначные комбинации, хотя на практике это не так, получится 10^3, или 1000, кодов от 000 до 999. Сколько семизначных телефонных номеров возможно внутри зоны 212? Это 10^7, или 10 000 000. Сколько номеров можно получить с кодом зоны 212 и префиксом 260? Это 10^4, или 10 000.

Аналогично в двоичной системе количество возможных кодов всегда равно двум 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 кодов.

Если известно, сколько кодов требуется, как вычислить необходимое число битов? Иными словами, как пройти предыдущую таблицу в обратном направлении?

Для этого нужен *логарифм по основанию два*. Логарифм - операция, обратная возведению в степень. Мы знаем, что 2 в 7-й степени равно 128. Логарифм 128 по основанию два равен 7. В математической записи утверждение

```
2^7 = 128
```

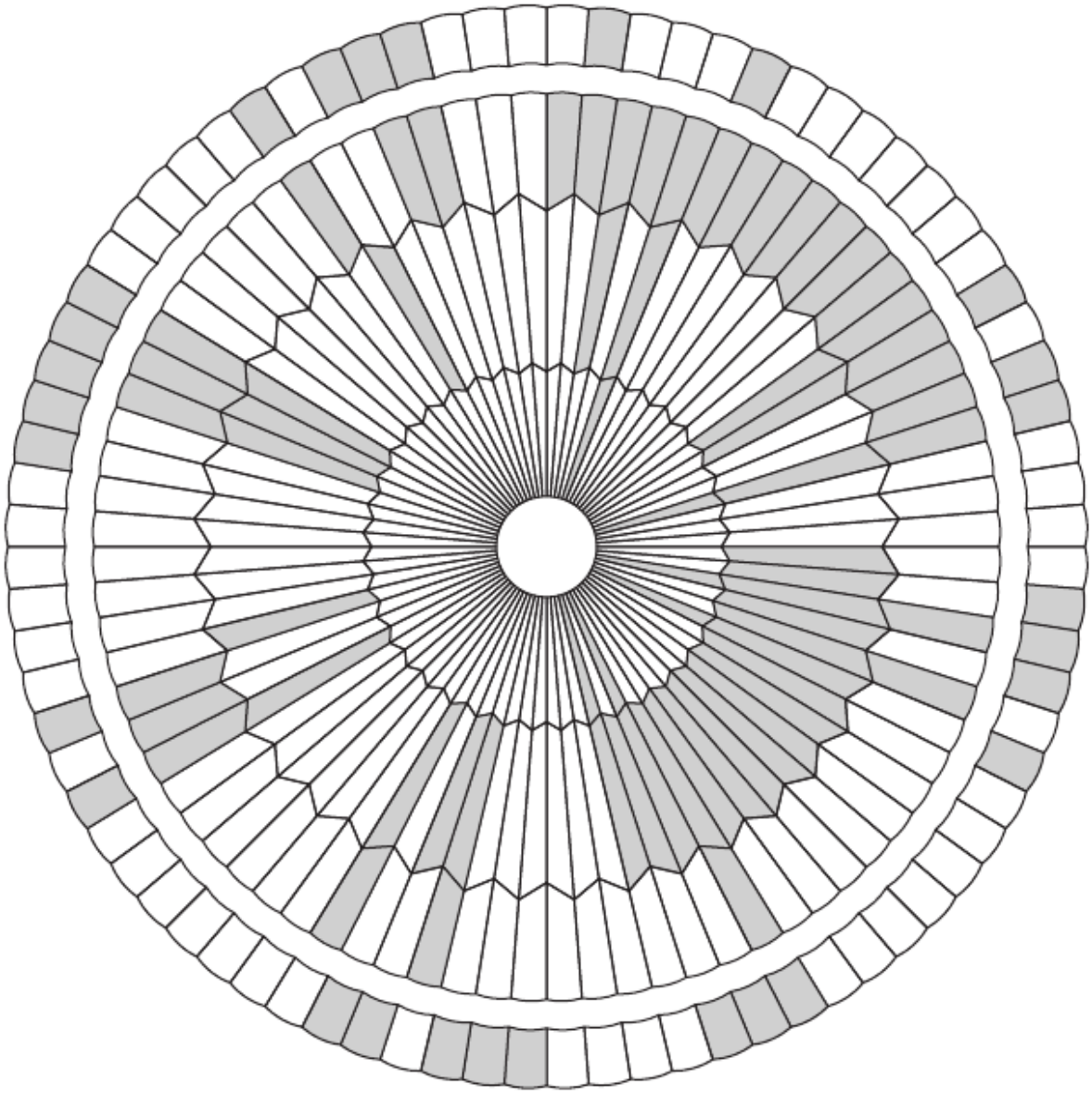
эквивалентно утверждению

```
log₂ 128 = 7
```

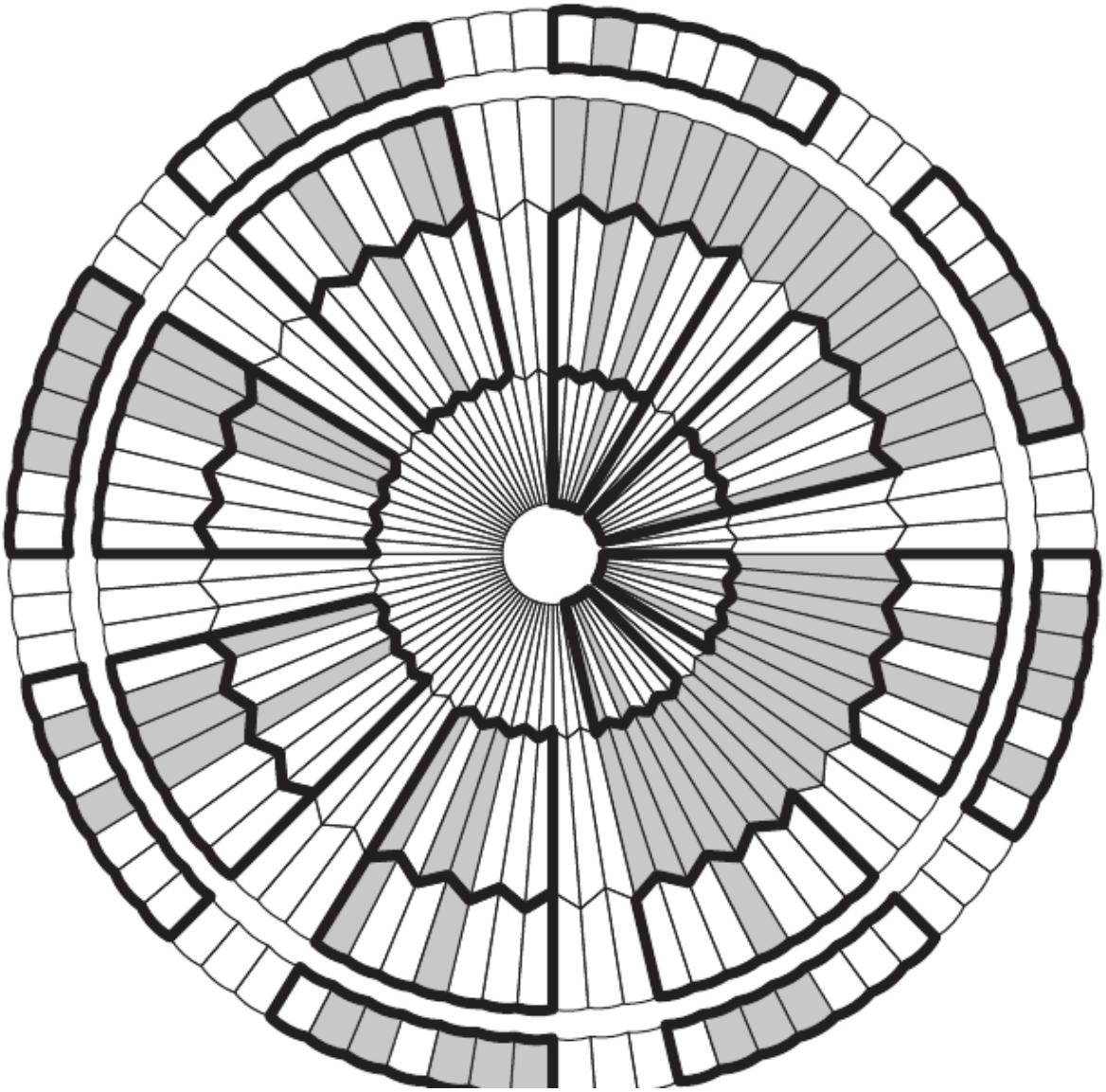
Если логарифм 128 по основанию два равен 7, а логарифм 256 равен 8, чему равен логарифм промежуточного числа, например 200? Примерно 7,64, но знать это необязательно. Чтобы представить битами 200 различных предметов, потребуется 8 битов, как Полу Ревиру понадобились два фонаря для передачи одного из трёх вариантов. Строго математически число битов для трёх вариантов Пола Ревира равно логарифму 3 по основанию два, или примерно 1,6, но на практике ему требовалось 2.

Биты часто скрыты от случайного наблюдателя глубоко внутри электронных устройств. Мы не видим биты, закодированные в компьютерах, текущие по проводам сетей или несущиеся в электромагнитных волнах вокруг точек Wi-Fi и вышек сотовой связи. Но иногда биты находятся прямо перед глазами.

Так было 18 февраля 2021 года, когда марсоход *Perseverance* сел на Марс. Парашют, видимый на снимке с марсохода, был составлен из 320 оранжевых и белых полос ткани, расположенных четырьмя концентрическими кругами:

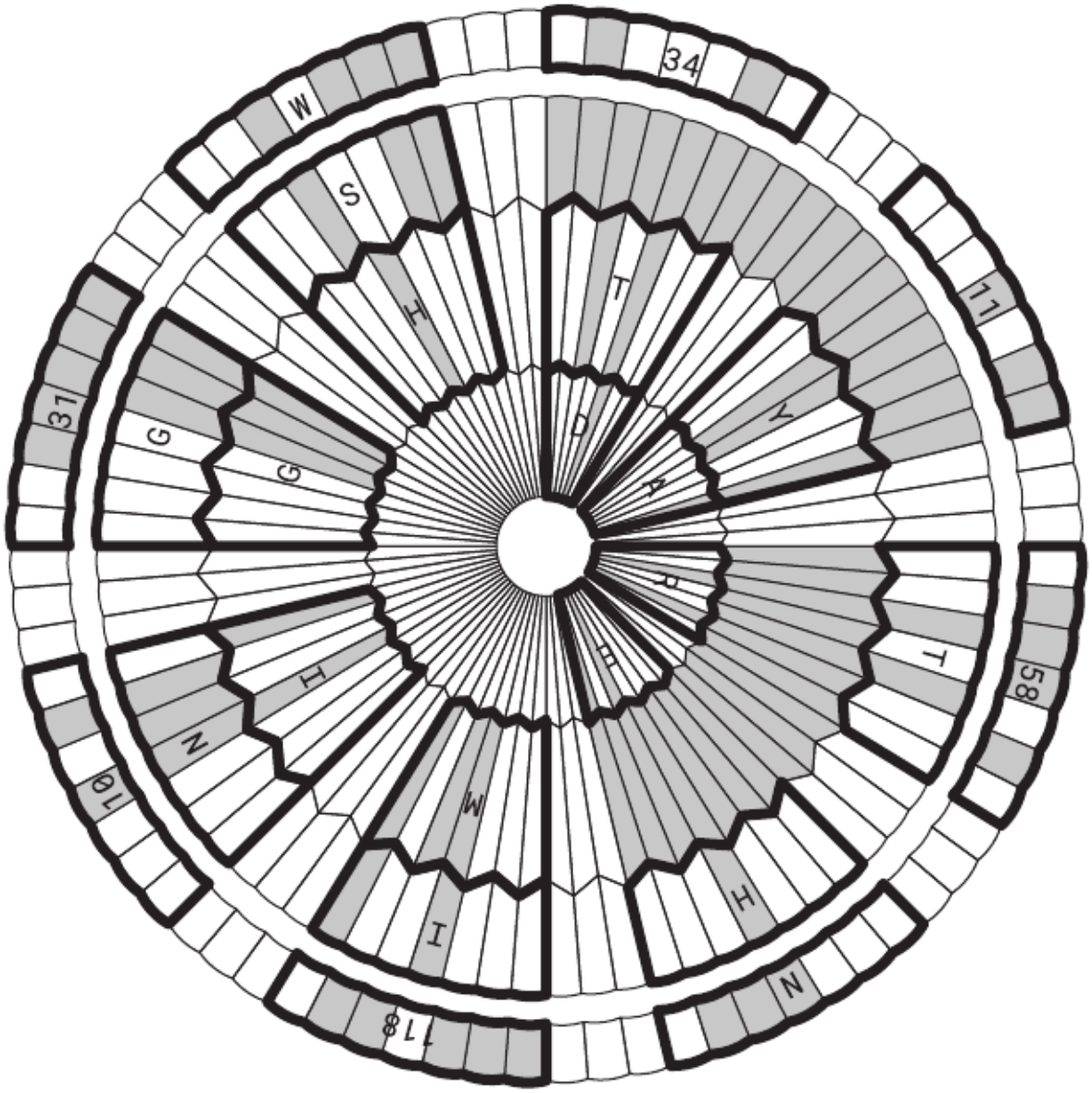


Пользователям Twitter не потребовалось много времени, чтобы расшифровать узор. Ключ - разделить полосы ткани на содержащие оба цвета группы по семь. Такие группы всегда разделены тремя белыми полосами. Области из последовательных оранжевых полос игнорируются. На следующей схеме каждая группа из семи полос обведена толстой чёрной линией:



Каждая группа представляет двоичное число: белая полоса означает 0, оранжевая - 1. Первая группа находится прямо над внутренним кругом. При движении по часовой стрелке эти семь полос кодируют двоичное 0000100, то есть десятичное 4. Четвёртая буква алфавита - D. Следующая по часовой стрелке группа равна 0000001, десятичной 1. Это A. Затем идёт 0010010, десятичное 18. Восемнадцатая буква алфавита - R. Следующая группа - 0000101, десятичное 5, то есть E. Первое слово - DARE.

Теперь перейдём на следующий внешний уровень. Биты равны 0001101, то есть десятичному 13 и букве M. Закончив расшифровку, вы получите три слова - фразу Теодора Рузвельта, ставшую неофициальным девизом Лаборатории реактивного движения NASA.



По внешнему кругу также закодированы числа, сообщающие широту и долготу Лаборатории реактивного движения: 34°11′58″N 118°10′31″W. В применённой простой системе кодирования ничто не отличает буквы от чисел. Числа 10 и 11 в географических координатах могли бы быть буквами J и K. Только контекст говорит, что это числа.

Вероятно, самое распространённое визуальное представление двоичных цифр - повсеместный универсальный товарный код UPC, маленький штрихкод почти на каждой покупаемой нами упаковке. UPC - один из десятков штрихкодов различного назначения. В печатном издании этой книги на задней обложке можно увидеть штрихкод другого типа, кодирующий международный стандартный книжный номер ISBN.

При первом появлении UPC вызвал некоторую паранойю, но в действительности это безобидная маленькая вещь, придуманная для автоматизации расчётов в розничной торговле и учёта запасов, с чем она довольно успешно справляется. До UPC кассовые аппараты супермаркетов не могли печатать детализированные чеки. Теперь это обычное дело.

Для нас важно, что UPC является двоичным кодом, хотя поначалу может таковым не казаться. Интересно расшифровать UPC и разобраться в его устройстве.

В наиболее распространённой форме UPC состоит из 30 вертикальных чёрных полос различной ширины, разделённых промежутками разной ширины, а также нескольких цифр. Например, вот UPC с банки куриного супа с лапшой Campbell's массой 10¾ унции:

Тот же UPC был напечатан в первом издании этой книги. За более чем 20 лет он не изменился!

Возникает соблазн визуально толковать UPC как тонкие и толстые чёрные полосы, узкие и широкие промежутки, и так на него действительно можно смотреть. Чёрные полосы UPC бывают четырёх значений ширины: более толстые в два, три или четыре раза шире самой тонкой. Аналогично широкие промежутки в два, три или четыре раза шире самого узкого.

Но UPC можно рассматривать и как последовательность битов. Помните, что сканер на кассе «видит» не весь штрихкод. Например, он не пытается распознать напечатанные снизу цифры, поскольку для этого потребовалась бы более сложная вычислительная технология - *оптическое распознавание символов*, или OCR. Вместо этого сканер видит лишь тонкий срез всего блока. UPC сделан большим, чтобы кассиру было куда направить сканер. Видимый сканеру срез можно представить так:

Похоже на азбуку Морзе, не правда ли? В действительности первоначальное изобретение сканируемых штрихкодов отчасти вдохновлялось азбукой Морзе.

Сканируя эти данные слева направо, компьютер присваивает первому встретившемуся чёрному отрезку бит 1, а следующему белому промежутку - бит 0. Последующие промежутки и полосы читаются как последовательности из 1, 2, 3 или 4 одинаковых битов в зависимости от их ширины. Соответствие отсканированного штрихкода битам выглядит так:

10100011010110001001100100011010001101000110101010111001011001101101100100111011001101000100101

Таким образом, весь UPC представляет собой последовательность из 95 битов. В данном примере их можно сгруппировать следующим образом:

Биты	Значение
101	Левая защитная комбинация
0001101	
0110001	
0011001	Цифры левой стороны
0001101	
0001101	
0001101	
01010	Центральная защитная комбинация
1110010	
1100110	
1101100	Цифры правой стороны
1001110	
1100110	
1000100	
101	Правая защитная комбинация

Первые 3 бита всегда равны 101. Это *левая защитная комбинация*, позволяющая сканирующему устройству сориентироваться. По ней сканер определяет ширину полос и промежутков, соответствующих отдельным битам. Иначе UPC на всех упаковках пришлось бы печатать строго одного размера.

После левой защитной комбинации следуют шесть групп по 7 битов. Вскоре вы увидите, как каждая группа кодирует одну десятичную цифру от 0 до 9.

Затем идёт пятибитовая центральная защитная комбинация. Наличие этого неизменного узора, всегда равного 01010, является встроенной проверкой ошибок. Если компьютерный сканер не находит центральную комбинацию в положенном месте, он не признаёт UPC прочитанным. Это одна из нескольких мер защиты от повреждённого или плохо напечатанного кода.

За центральной защитной комбинацией следуют ещё шесть групп по 7 битов, а затем правая защитная комбинация, всегда равная 101. Конечная комбинация позволяет сканировать UPC как вперёд, слева направо, так и назад.

Итак, UPC кодирует 12 десятичных цифр. Левая сторона кодирует шесть цифр, для каждой из которых требуется 7 битов. Расшифровать их можно по следующей таблице:

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

Обратите внимание, что каждый семибитовый код начинается с 0 и заканчивается 1. Если сканер встречает слева код, начинающийся с 1 или заканчивающийся 0, он понимает, что либо неправильно прочитал UPC, либо код был изменён. Кроме того, каждый код содержит лишь две группы последовательных единиц. Следовательно, каждой цифре соответствуют две вертикальные полосы UPC.

Рассмотрев коды внимательнее, вы обнаружите, что каждый содержит нечётное число единиц. Это ещё одна форма проверки ошибок и согласованности, называемая *контролем чётности*. Группа битов имеет *чётную чётность*, если содержит чётное число единиц, и *нечётную чётность*, если число единиц нечётно. Таким образом, все эти коды имеют нечётную чётность.

Для расшифровки шести семибитовых кодов справа используется следующая таблица:

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

Эти коды являются противоположностями, или *дополнениями*, предыдущих: там, где был 0, теперь 1, и наоборот. Они всегда начинаются с 1 и заканчиваются 0. Кроме того, в них чётное число единиц, то есть чётная чётность.

Теперь мы готовы расшифровать UPC. По двум предыдущим таблицам можно определить, что на банке супа Campbell's массой 10¾ унции закодированы 12 десятичных цифр:

```
0 51000 01251 7
```

Это *очень* разочаровывает. Как видите, получились ровно те же цифры, которые заботливо напечатаны под UPC. И это вполне разумно: если сканер почему-либо не читает код, кассир может ввести цифры вручную. Вы наверняка видели такое. Вся проделанная работа не потребовалась для расшифровки цифр, и мы даже близко не подошли к раскрытию тайной информации. Но больше в UPC расшифровывать нечего. Тридцать вертикальных полос сводятся всего к 12 цифрам.

Первая из 12 десятичных цифр, в данном случае 0, называется *символом системы счисления*. Ноль означает обычный код UPC. Для товаров с переменной массой, например мяса или овощей, код начинается с 2. Купоны обозначаются цифрой 5.

Следующие пять цифр образуют код производителя. В данном случае 51000 - код компании Campbell Soup Company. Этот код имеют все продукты Campbell. Следующие пять цифр, 01251, являются кодом конкретного продукта этой компании - банки куриного супа с лапшой массой 10¾ унции. Код продукта имеет смысл только вместе с кодом производителя. Суп с лапшой другой компании может иметь иной код, а код 01251 у другого производителя может означать совершенно другой товар.

Вопреки распространённому убеждению, UPC не содержит цену товара. Её необходимо получить из компьютера, используемого магазином совместно с кассовыми сканерами.

Последняя цифра, в данном случае 7, называется *контрольным символом по модулю*. Она обеспечивает ещё одну проверку ошибок. Можете проверить сами. Присвойте первым 11 цифрам, в нашем примере 0 51000 01251, буквы:

```
A BCDEF GHIJK
```

Теперь вычислите:

```
3 × (A + C + E + G + I + K) + (B + D + F + H + J)
```

и вычтите результат из ближайшего большего числа, кратного 10. Для банки куриного супа с лапшой Campbell's получаем:

```
3 × (0 + 1 + 0 + 0 + 2 + 1) + (5 + 0 + 0 + 1 + 5)
= 3 × 4 + 11
= 23
```

Следующее большее число, кратное 10, равно 30, поэтому

```
30 - 23 = 7
```

Именно такой контрольный символ напечатан и закодирован в UPC. Это форма избыточности. Если управляющий сканером компьютер вычислит другой контрольный символ, чем закодированный в UPC, он не признает код действительным.

Обычно для указания десятичной цифры от 0 до 9 достаточно всего 4 битов. UPC использует по 7 битов на цифру. В целом UPC тратит 95 битов, чтобы закодировать лишь 11 полезных десятичных цифр. В действительности слева и справа от защитных комбинаций UPC содержит пустое пространство, эквивалентное девяти нулевым битам с каждой стороны. Поэтому полный UPC требует 113 битов для кодирования 11 десятичных цифр - больше 10 битов на цифру!

Часть этой избыточности необходима для проверки ошибок. Код товара был бы не слишком полезен, если бы покупатель мог легко изменить его фломастером.

Преимущество UPC также в возможности чтения в обоих направлениях. Если первые расшифрованные сканером цифры имеют чётную чётность, то есть чётное число единиц в каждом семибитовом коде, сканер понимает, что читает UPC справа налево. Тогда компьютер применяет к цифрам правой стороны следующую таблицу:

Цифра	Код правой стороны в обратном порядке	Цифра	Код
0	0100111	5	0111001
1	0110011	6	0000101
2	0011011	7	0010001
3	0100001	8	0001001
4	0011101	9	0010111

А для цифр левой стороны - эту:

Цифра	Код левой стороны в обратном порядке	Цифра	Код
0	1011000	5	1000110
1	1001100	6	1111010
2	1100100	7	1101110
3	1011110	8	1110110
4	1100010	9	1101000

Все эти семибитовые коды отличаются от кодов, читаемых слева направо. Неоднозначности нет.

Один из способов вместить в сканируемый код больше информации - перейти к двум измерениям. Вместо строки толстых и тонких полос и промежутков создаётся сетка чёрных и белых квадратов.

Вероятно, самый распространённый двумерный штрихкод - код быстрого отклика QR, впервые разработанный в Японии в 1994 году и теперь применяемый для самых разных целей.

Создать собственный QR-код легко и бесплатно. Для этого существует несколько сайтов. Также широко доступно программное обеспечение, способное сканировать и расшифровывать QR-коды камерой мобильного устройства. Специализированные QR-сканеры используются в промышленности, например для отслеживания грузов и инвентаризации складов.

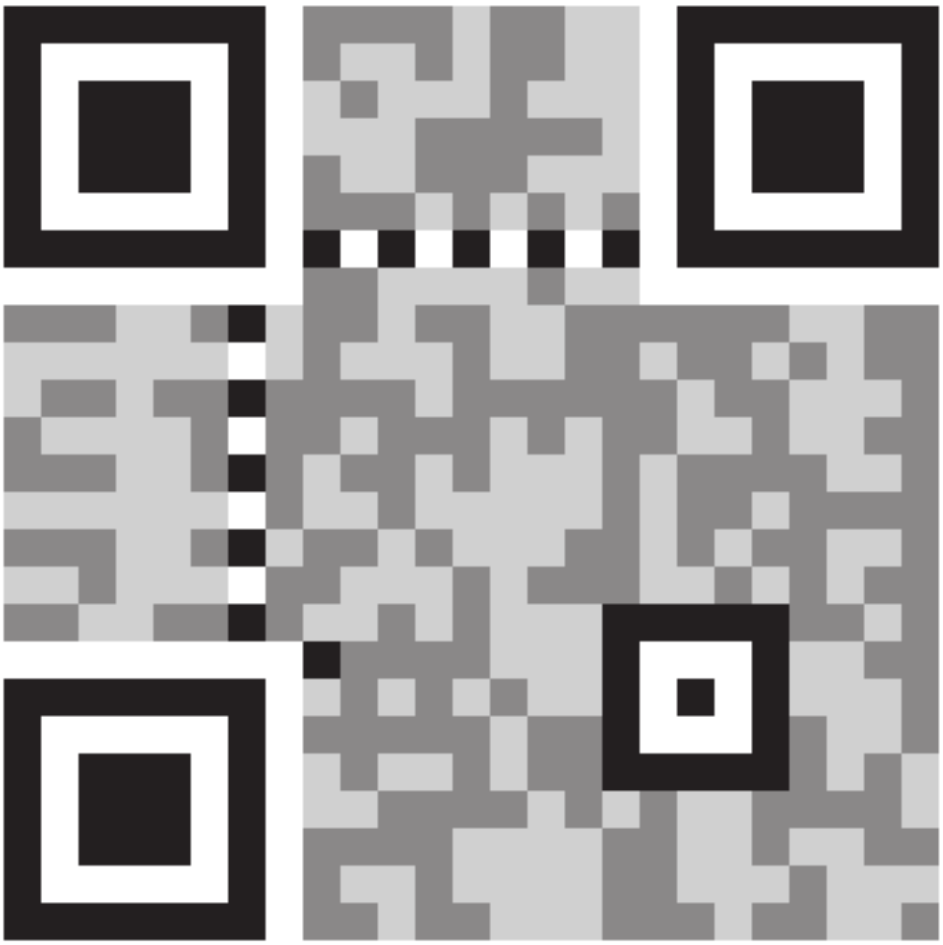
Вот QR-код, в котором закодирован адрес сайта этой книги, CodeHiddenLanguage.com:

Если на мобильном устройстве есть приложение для чтения QR-кодов, можно навести его на изображение и перейти на сайт.

QR-код состоит из сетки квадратов, которые в официальной спецификации называются *модулями*. Этот код имеет по 25 модулей по горизонтали и вертикали, такой размер называется версией 2. Поддерживаются 40 размеров QR-кода; версия 40 имеет по 177 модулей в каждом направлении.

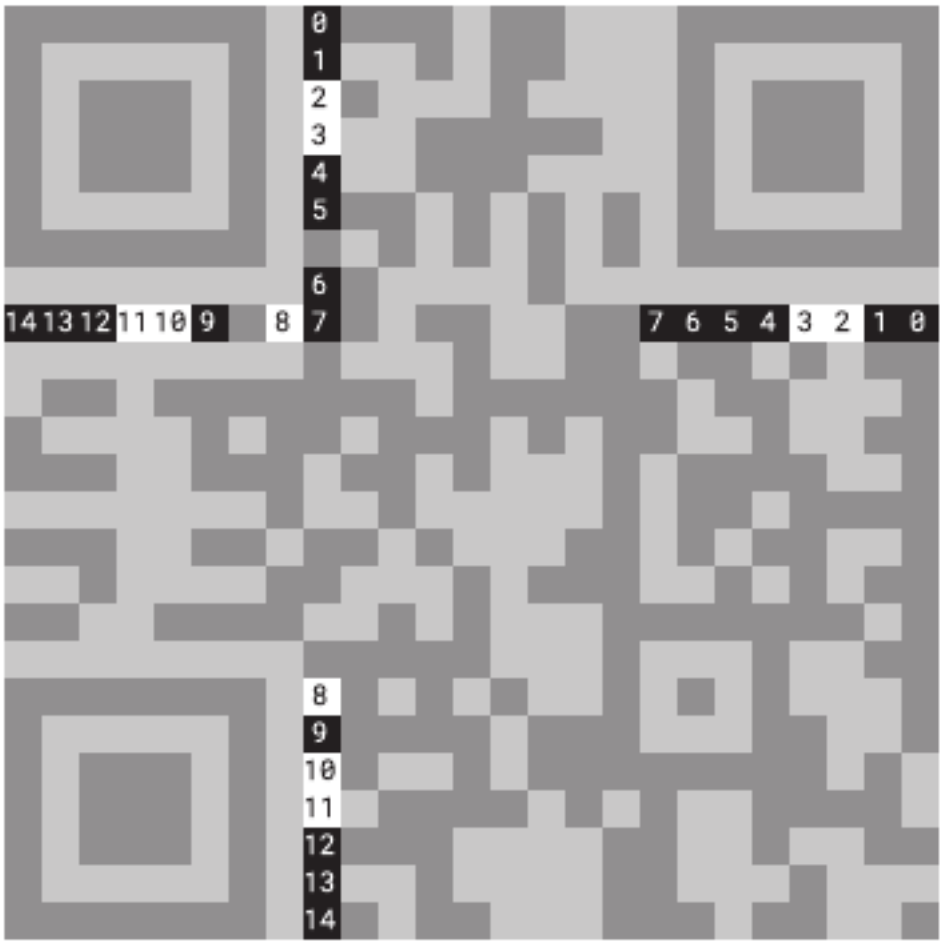
Если считать каждый маленький блок битом, где белый означает 0, а чёрный - 1, сетка такого размера потенциально кодирует 25 × 25, то есть 625 битов. Но реальная ёмкость составляет примерно треть этого значения. Значительная часть информации отведена математически сложной и изощрённой схеме коррекции ошибок. Она защищает QR-код от изменений и помогает восстанавливать данные, отсутствующие в повреждённом коде. Коррекцию ошибок QR-кода я обсуждать не буду.

Наиболее очевидно, что QR-код содержит несколько фиксированных узоров, помогающих сканеру правильно ориентировать сетку. На следующем изображении фиксированные узоры показаны чёрным и белым, а всё остальное - серым:



Три больших квадрата в углах называются *поисковыми узорами*, а меньший квадрат ближе к правому нижнему углу - *узором выравнивания*. Они помогают устройству чтения правильно ориентировать код и компенсировать искажения. Горизонтальная и вертикальная последовательности чередующихся чёрных и белых ячеек возле верхнего и левого краёв называются *синхронизирующими узорами* и служат для определения числа ячеек QR-кода. Кроме того, код должен быть полностью окружён *тихой зоной* - белой рамкой шириной в четыре ячейки.

Программы создания QR-кода предлагают несколько параметров, включая разные системы коррекции ошибок. Информация, необходимая считывающему устройству для коррекции ошибок и других задач, кодируется 15 битами, называемыми *информацией формата*. Эти 15 битов присутствуют в QR-коде дважды. Вот они с метками от 0 до 14 справа и снизу от левого верхнего поискового узора, а затем повторно под правым верхним и справа от левого нижнего узора:



Биты иногда нумеруют таким образом, чтобы показать, как они образуют более длинное значение. Бит с меткой 0 является младшим и находится в крайней правой позиции числа. Бит 14 - старший, он находится слева. Если белые ячейки соответствуют нулевым битам, а чёрные - единичным, полное 15-битовое число выглядит так:

```
111001011110011
```

Почему бит 0 является младшим? Потому что занимает в полном числе позицию, соответствующую 2 в нулевой степени. Если нужно вспомнить, как биты образуют число, посмотрите верхнюю часть страницы 109.

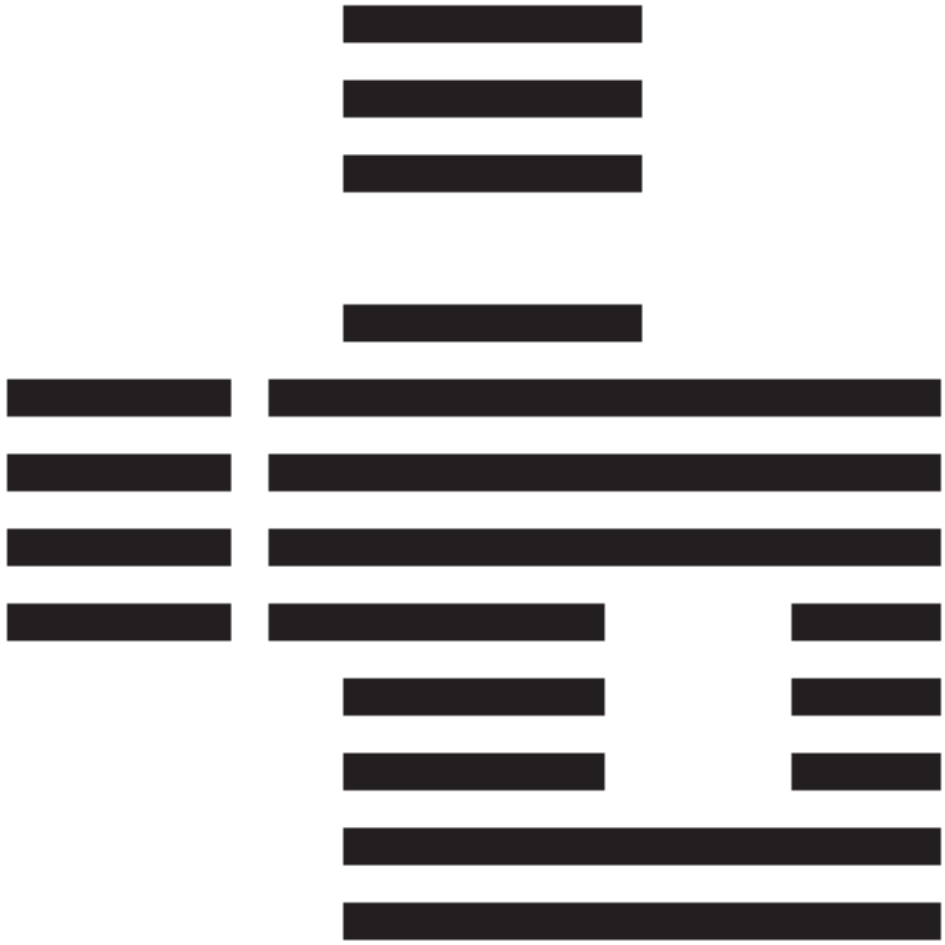
Фактическое числовое значение этого 15-битового числа не важно, потому что оно объединяет три фрагмента информации. Два старших бита задают один из четырёх уровней коррекции ошибок. Десять младших задают десятибитовый код BCH для коррекции ошибок. Аббревиатура BCH происходит от фамилий изобретателей этого вида кода: Боуза, Чоудхури и Хоквингема. Но я обещал не обсуждать коррекцию ошибок QR-кода!

Между двухбитовым уровнем коррекции и десятибитовым кодом BCH находятся три бита, которые *не* используются для коррекции. Здесь они выделены полужирным:

11 **100** 1011110011

Оказывается, устройства чтения QR-кодов лучше всего работают, когда чёрных и белых квадратов примерно поровну. Для некоторых данных это условие не выполняется. Программа создания QR-кода должна выбрать *маску*, выравнивающую количество чёрных и белых квадратов. Маска применяется к QR-коду и меняет цвет выбранных ячеек с белого на чёрный или с чёрного на белый, а представляемые ими биты - с 0 на 1 или с 1 на 0.

Документация QR-кода определяет восемь разных масок, задаваемых восьмью трёхбитовыми последовательностями: 000, 001, 010, 011, 100, 101, 110 и 111. В рассматриваемом QR-коде записано значение 100. Ему соответствует маска из горизонтальных линий, чередующихся через строку:



Каждая ячейка исходного QR-кода, соответствующая белой области маски, остаётся неизменной. Цвет каждой ячейки, соответствующей чёрной области, меняется на противоположный. Обратите внимание: маска не затрагивает фиксированные области и область информации QR-кода. Вот результат применения маски к исходному коду:

Маска не изменяет фиксированные и информационные области. Если сравнить это изображение с исходным QR-кодом, видно, что цвет верхней строки обращён, вторая строка не изменилась, третья снова обращена и так далее.

Теперь можно приступить к настоящим данным. Начнём с четырёх битов в правом нижнем углу. На следующем изображении ячейки пронумерованы от 0 до 3, где 3 - старший бит, а 0 - младший:

Эти четыре бита называются *индикатором типа данных* и показывают, какие данные закодированы в QR-коде. Вот несколько возможных значений:

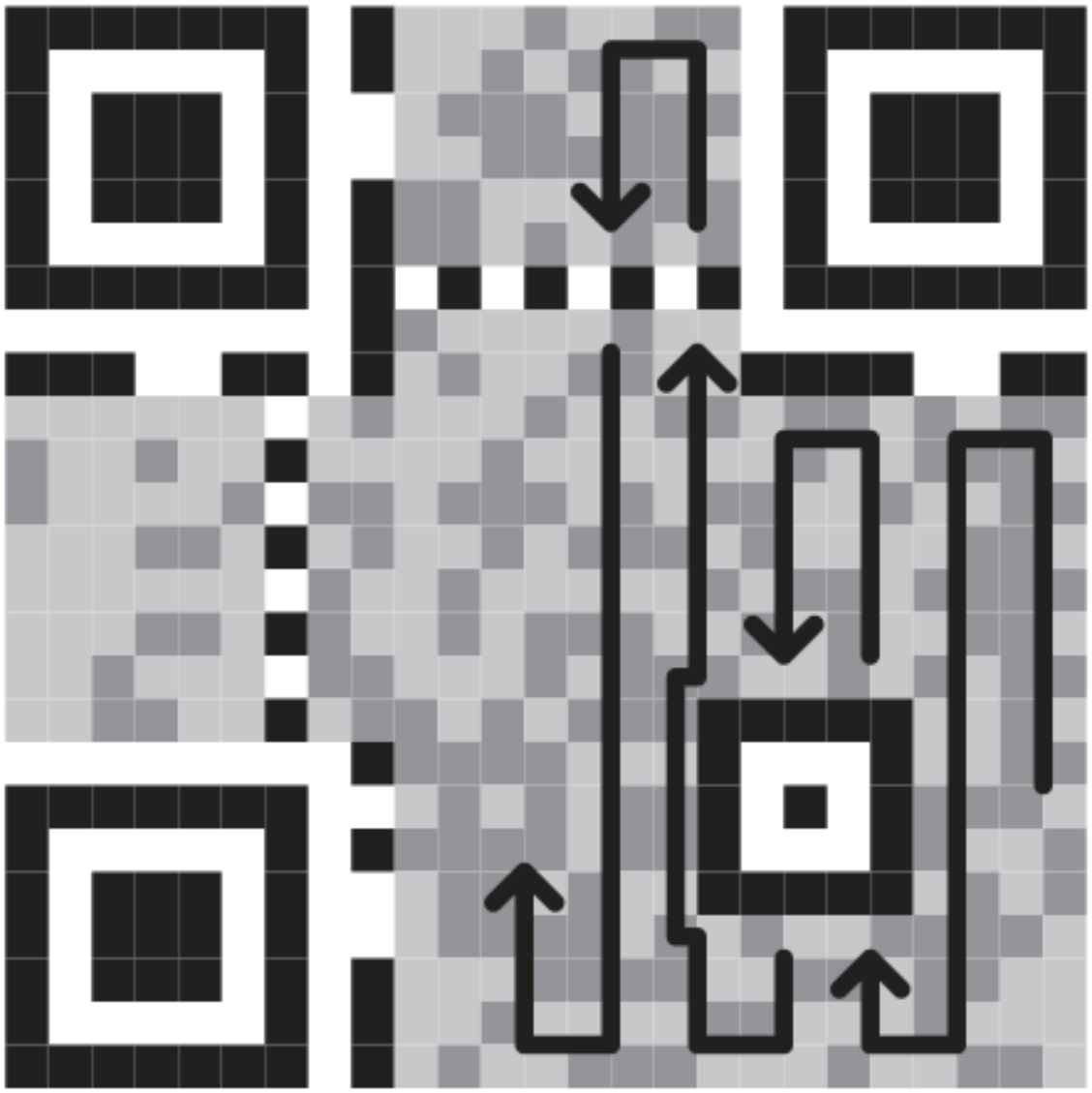
Индикатор типа данных	**Значение**
0001	Только числа
0010	Прописные буквы и числа
0100	Текст, закодированный восьмибитовыми значениями
1000	Японские кандзи

В данном QR-коде значение равно 0100, следовательно, данные состоят из восьмибитовых значений, кодирующих текст.

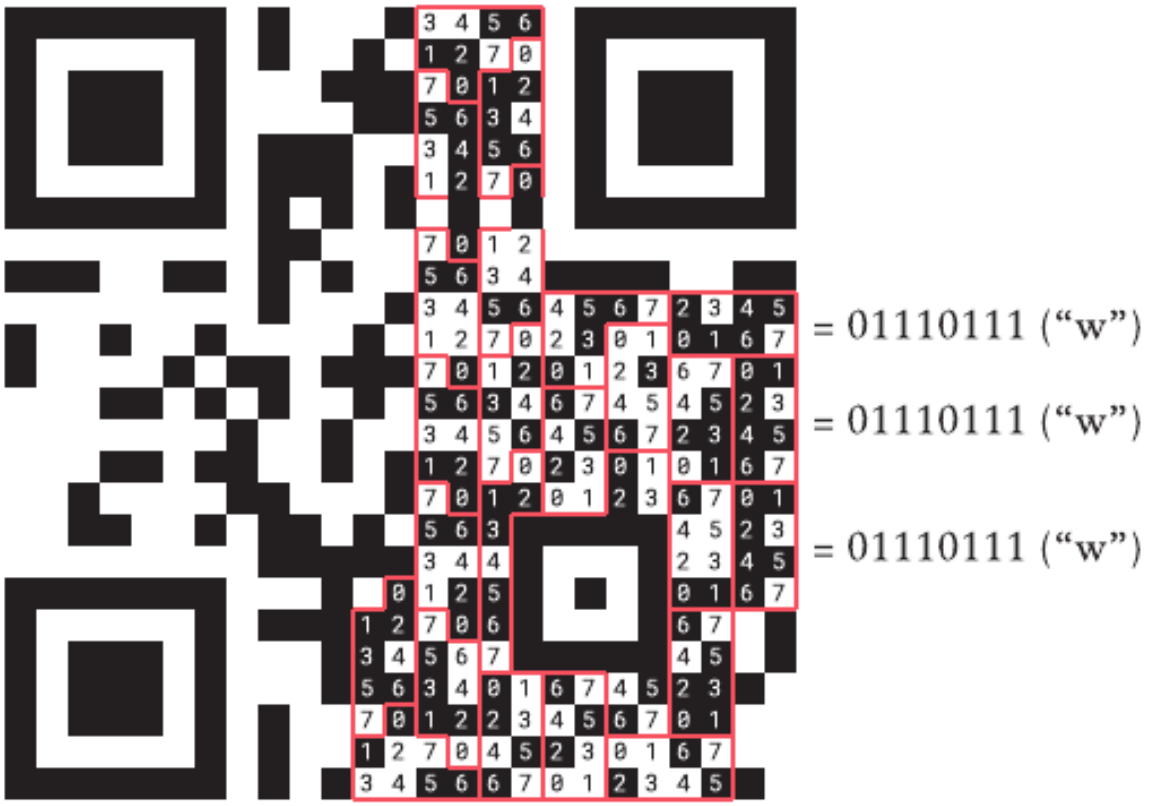
Следующий элемент хранится в восьми ячейках над индикатором типа данных. На этой иллюстрации восемь битов пронумерованы от 0 до 7:

Значение равно 00011010, то есть 26 в десятичной системе. Столько символов закодировано в QR-коде.

Порядок символов систематичен, но странен. Они начинаются прямо над счётчиком символов. Обычно, хотя и не всегда, каждый символ занимает область шириной в две и высотой в четыре ячейки. Символы извиваются по сетке следующим образом:



Не все символы занимают прямоугольники размером две на четыре ячейки. К счастью, официальная спецификация QR очень точно определяет ориентацию битов для непрямоугольной области. На следующем изображении



ячейки каждого из 26 символов обведены красным и пронумерованы от 0 до 7, где 0 обозначает младший бит, а 7 - старший.

Спецификация QR указывает, что текст кодируется восьмибитовыми значениями стандарта ISO/IEC 8859. Это громкое название варианта Американского стандартного кода для обмена информацией ASCII, который будет подробнее рассмотрен в главе 13.

Первый символ равен 01110111 - это код ASCII для *w*. Следующий выше такой же. Следующий символ уходит влево, но это ещё одна *w*. Затем двигайтесь вниз по следующим двум парам столбцов. Следующий символ 00101110 - точка, затем 01000011 - прописная *C*, после неё 01101111 - *o*. Следующий символ переходит через следующую пару строк. Это 01100100 - *d*. Следующий начинается под узором выравнивания и продолжается над ним. Код ASCII 01100101 означает *e*. Продолжая тем же способом, можно прочитать www.CodeHiddenLanguage.com.

Вот и всё. Большая часть оставшегося QR-кода посвящена коррекции ошибок.

Коды UPC и QR на первый взгляд, несомненно, выглядят устрашающе, и людям можно простить предположение, будто они кодируют секретную, а возможно и зловещую информацию. Но для широкого применения такие коды должны быть хорошо документированы и общедоступны. Чем шире они используются, тем ценнее становятся как ещё одно расширение нашего огромного набора средств коммуникации.

Биты повсюду, но ближе к концу обсуждения QR-кода я упомянул «восьмибитовые значения». Для восьмибитовых значений существует специальное слово. Возможно, вы его слышали.

Глава 12. Байты и шестнадцатеричные числа

Отдельные биты могут выражать важные утверждения: да или нет, истина или ложь, зачёт или незачёт. Но чаще несколько битов объединяют для представления чисел, а через них - любых данных, включая текст, звук, музыку, изображения и фильмы. Цепь, складывающая два бита, представляет интерес, но цепь, складывающая много битов, уже становится частью настоящего компьютера.

Чтобы было удобнее перемещать биты и манипулировать ими, компьютерные системы часто объединяют определённое число битов в величину, называемую *словом*. Длина, или размер, слова, то есть число составляющих его битов, чрезвычайно важна для архитектуры компьютера, потому что все данные перемещаются группами по одному или несколько слов.

Некоторые ранние компьютерные системы использовали слова длиной, кратной 6 битам: например, 12, 18 или 24 бита. Такие длины особенно привлекательны по простой причине: их значения легко записываются восьмеричными числами. Как вы помните, восьмеричные цифры 0, 1, 2, 3, 4, 5, 6 и 7 соответствуют трёхбитовым значениям:

Двоичная	Восьмеричная
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Шестибитовое слово можно представить ровно двумя восьмеричными цифрами, а другие размеры слов - 12, 18 и 24 бита - кратны шести. Для 24-битового слова требуется восемь восьмеричных цифр.

Но компьютерная отрасль пошла несколько иным путём. После признания важности двоичных чисел работа со словами длиной 6, 12, 18 или 24 бита, которые не являются степенями двойки, а вместо этого кратны трём, должна была казаться почти извращением.

И тут появляется байт.

Слово *byte* возникло в IBM, вероятно, около 1956 года. Оно произошло от слова *bite* («кусок»), но было написано через *y*, чтобы его никто не спутал со словом *bit*. Первоначально байт означал просто число битов в конкретном тракте данных. Но к середине 1960-х годов, в связи с разработкой большого семейства деловых компьютеров IBM System/360, слово стало означать группу из 8 битов.

Это значение закрепилось: 8 битов в байте теперь являются универсальной единицей цифровых данных.

Как восьмибитовая величина байт принимает значения от 00000000 до 11111111 и может представлять десятичные числа от 0 до 255 либо одну из 2^8, то есть 256, различных возможностей. Восемь оказалось весьма удобным размером порции битов - не слишком малым и не слишком большим. Байт подходит во многих смыслах. Как вы увидите далее, байт идеален для хранения текста, поскольку многие письменные языки мира можно представить менее чем 256

символами. Когда одного байта недостаточно, например для идеограмм китайского, японского и корейского языков, обычно вполне хватает 2 байтов, допускающих 2^16, или 65 536, вариантов. Байт также идеально подходит для оттенков серого на чёрно-белых фотографиях, потому что человеческий глаз различает приблизительно 256 оттенков. Для цвета на видеодисплеях удобно использовать 3 байта, представляющих красную, зелёную и синюю составляющие.

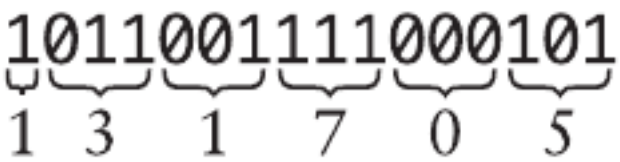
Революция персональных компьютеров началась в конце 1970-х и начале 1980-х годов с восьмибитовых машин. Последующие технические достижения удваивали число используемых компьютером битов: от 16 к 32 и 64 битам, то есть соответственно к 2, 4 и 8 байтам. Для некоторых специальных задач существуют также 128- и 256-битовые компьютеры.

Половина байта, то есть 4 бита, иногда называется *nibble*, а иногда пишется *nybble*, но в разговорах это слово встречается гораздо реже, чем *byte*.

Поскольку внутри компьютеров байты встречаются постоянно, удобно обозначать их значения короче, чем строкой двоичных цифр. Для этого, конечно, можно применять восьмеричную систему. Например, биты байта 10110110 можно разбить справа на тройки и преобразовать каждую группу по приведённой выше таблице:

```
10 110 110
 2   6   6
```

Восьмеричное число 266 короче, чем 10110110, но байты и восьмеричная система принципиально несовместимы. Восемь не делится на три без остатка, поэтому восьмеричная запись 16-битового числа не совпадает с соединёнными восьмеричными записями двух составляющих его байтов:



Чтобы представления многобайтовых величин согласовывались с представлениями отдельных байтов, нужна система счисления, в которой каждый байт делится на равные группы битов.

Можно разделить байт на четыре двухбитовых значения. Это дала бы описанную в главе 10 четверичную систему с основанием четыре. Но запись, вероятно, получилась бы не настолько краткой, как нам хотелось бы.

Либо можно разделить байт на два четырёхбитовых значения. Для этого требуется система счисления с основанием 16.

Основание 16. Этого мы ещё не рассматривали, и не без причины. Система с основанием 16 называется *шестнадцатеричной*, и даже само английское слово *hexadecimal* представляет собой путаницу. Большинство слов с приставкой *hexa*, например *hexagon*, *hexapod* или *hexameter*, относится к шести предметам. Предполагается, что *hexadecimal* означает шестнадцать: шесть плюс десятичное. И хотя мне велели привести текст книги в соответствие с сетевым

Руководством по стилю Microsoft, где ясно сказано: «Не сокращайте до *hex*», все всегда так сокращают, и иногда это буду делать и я.

Название - не единственная странность шестнадцатеричной системы. В десятичной мы считаем так:

```
0 1 2 3 4 5 6 7 8 9 10 11 12...
```

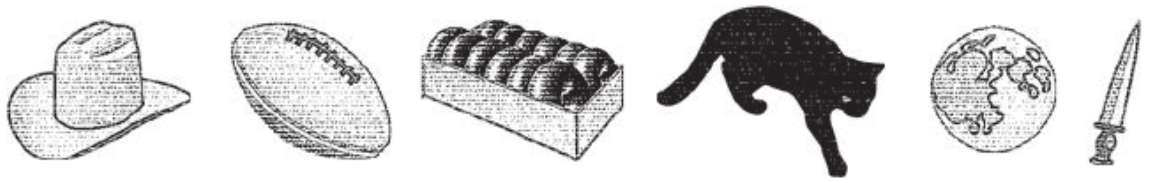
В восьмеричной цифры 8 и 9 уже не нужны:

```
0 1 2 3 4 5 6 7 10 11 12...
```

Но шестнадцатеричная система отличается тем, что ей требуется *больше* цифр, чем десятичной. Счёт выглядел бы примерно так:

```
0 1 2 3 4 5 6 7 8 9 [здесь нужны дополнительные цифры] 10 11 12
```

где 10, произносимое «один ноль», в действительности равно десятичному 16. Но что использовать вместо шести недостающих символов? Откуда их взять? Они не были переданы нам традицией, как остальные цифровые символы, поэтому разумно было бы придумать шесть новых, например:



В отличие от большинства наших цифр, эти символы легко запомнить и связать с представляемыми количествами. Здесь есть десятигаллонная ковбойская шляпа, мяч для американского футбола, где в команде 11 игроков, дюжина пончиков, чёрная кошка, связанная с несчастливым числом 13, полная луна, наступающая примерно через две недели, то есть 14 дней, после новолуния, и кинжал, напоминающий об убийстве Юлия Цезаря в мартовские иды, 15-го числа.

Но нет. К сожалению, а возможно, к вашему облегчению, мы не станем записывать шестнадцатеричные числа мячами и пончиками. Так можно было сделать, но этого не произошло. Вместо этого общепринятая шестнадцатеричная запись гарантирует, что все окончательно запутаются и так и останутся в этом состоянии. Шесть недостающих цифр обозначаются первыми шестью буквами латинского алфавита:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12...
```

Следующая таблица показывает соответствие двоичных, шестнадцатеричных и десятичных записей:

Двоичная	Шестнадцатеричная	Десятичная
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Двоичная	Шестнадцатеричная	Десятичная
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Использовать буквы для чисел неприятно, а когда числа применяются для представления букв, путаница усиливается, но шестнадцатеричная система останется с нами. Она существует по одной-единственной причине: чтобы представлять значения байтов настолько кратко, насколько это разумно возможно. И с этой задачей справляется довольно хорошо.

Каждый байт состоит из 8 битов, или двух шестнадцатеричных цифр от 00 до FF. Байт 10110110 является шестнадцатеричным числом B6, а байт 01010111 - числом 57.

B6 явно шестнадцатеричное из-за буквы, но 57 можно принять за десятичное. Чтобы избежать путаницы, нужен простой способ различать десятичные и шестнадцатеричные числа. Такой способ существует. В действительности разные языки программирования и среды используют около 20 обозначений шестнадцатеричных чисел. В этой книге после числа будет ставиться строчная *h*: например, B6h или 57h.

Вот несколько характерных однобайтовых шестнадцатеричных чисел и их десятичные эквиваленты:

Двоичная	Шестнадцатеричная	Десятичная
00000000	00h	0
00010000	10h	16
00011000	18h	24
00100000	20h	32
01000000	40h	64
01100100	64h	100
10000000	80h	128
11000000	C0h	192
11001000	C8h	200
11100000	E0h	224
11110000	F0h	240
11111111	FFh	255

Подобно двоичным, шестнадцатеричные числа часто записывают с ведущими нулями, чтобы явно указать определённое число цифр. В длинных двоичных числах каждой четвёрке двоичных цифр соответствует одна шестнадцатеричная. Шестнадцатибитовая величина занимает 2 байта и четыре шестнадцатеричные цифры. 32-битовая - 4 байта и восемь шестнадцатеричных цифр.

Из-за широкого применения шестнадцатеричной системы длинные двоичные числа принято разделять дефисами или пробелами через каждые четыре цифры. Например, двоичное число 00100100011010000101011001110 выглядит чуть менее пугающим как 0010 0100 0110 1000 1010 1100 1110 или 0010-0100-0110-1000-1010-1100-1110, а соответствие шестнадцатеричным цифрам становится яснее:

```
0010 0100 0110 1000 1010 1100 1110
  2    4    6    8    A    C    E
```

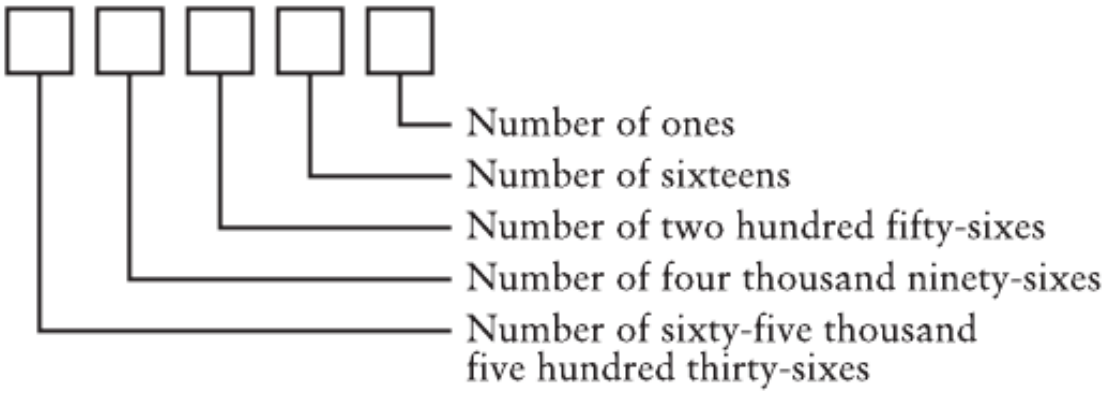
Получилось семизначное шестнадцатеричное число 2468ACE - все чётные шестнадцатеричные цифры подряд. Если группа поддержки скандирует: «2, 4, 6, 8, A, C, E! В информатику поступай скорей!» - ваш колледж, возможно, немного слишком увлечён компьютерами.

Если вы работали с HTML, языком гипертекстовой разметки веб-страниц в интернете, то, возможно, уже знакомы с распространённым применением шестнадцатеричных чисел. Каждая цветная точка, или *пиксель*, компьютерного экрана представляет комбинацию трёх аддитивных основных цветов: красного, зелёного и синего, называемую цветом *RGB*. Интенсивность, или яркость, каждой из трёх составляющих задаётся значением байта, поэтому для конкретного цвета требуется 3 байта. В HTML цвет часто обозначается шестизначным шестнадцатеричным значением со знаком решётки впереди. Например, красный оттенок иллюстраций этой книги имеет значение #E74536: красная составляющая E7h, зелёная 45h, синяя 36h. На HTML-странице тот же цвет можно задать эквивалентными десятичными величинами: `rgb(231, 69, 54)`.

Зная, что для цвета каждого экранного пикселя требуется 3 байта, можно выполнить небольшое вычисление и получить другие сведения. Если экран содержит 1920 пикселей по горизонтали и 1080 по вертикали, стандартное разрешение телевидения высокой чёткости, для хранения изображения требуется 1920 × 1080 × 3, то есть 6 220 800 байтов.

Каждый основной цвет может принимать значения от 0 до 255, поэтому общее число комбинаций равно 256 × 256 × 256, или 16 777 216 уникальных цветов. В шестнадцатеричной системе это 100h × 100h × 100h, то есть 1000000h.

В шестнадцатеричном числе позиции цифр соответствуют степеням 16:



Шестнадцатеричное число 9A48Ch можно записать так:

```
9A48Ch = 9 × 10000h +
         A × 1000h +
         4 × 100h +
         8 × 10h +
         C × 1h
```

То же самое можно записать степенями 16:

```
9A48Ch = 9 × 16^4 +
         A × 16^3 +
         4 × 16^2 +
         8 × 16^1 +
         C × 16^0
```

Или десятичными эквивалентами этих степеней:

```
9A48Ch = 9 × 65,536 +
         A × 4096 +
         4 × 256 +
         8 × 16 +
         C × 1
```

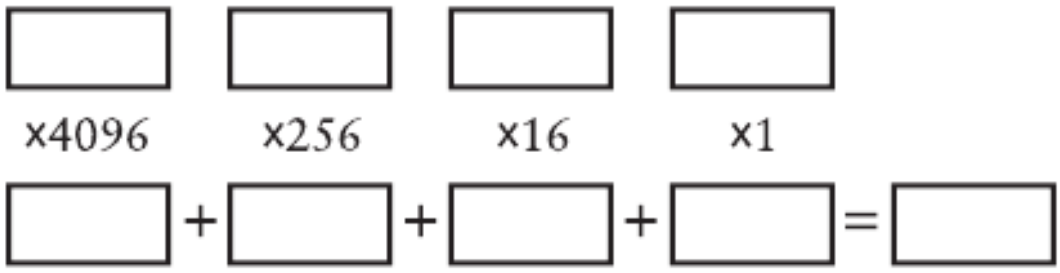
Обратите внимание: отдельные цифры числа 9, A, 4, 8 и C можно писать без указания основания, и неоднозначности не возникает. Отдельная 9 остаётся девяткой и в десятичной, и в шестнадцатеричной системе. А буква A очевидно шестнадцатеричная и равна десятичному 10.

Преобразовав все цифры в десятичные, можно выполнить вычисление:

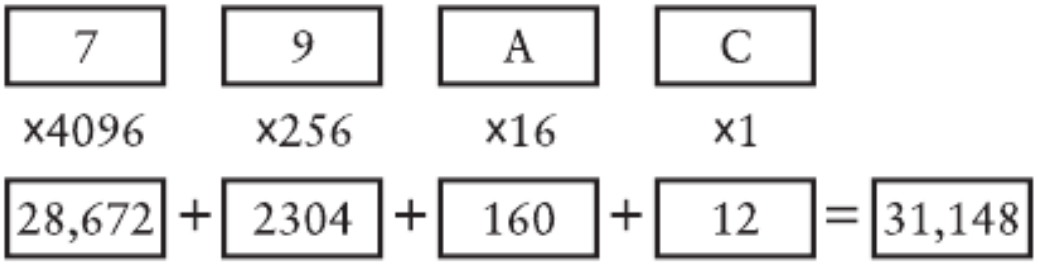
```
9A48Ch = 9 × 65,536 +
         10 × 4096 +
         4 × 256 +
         8 × 16 +
         12 × 1
```

Ответ равен 631 948. Так шестнадцатеричные числа преобразуются в десятичные.

Вот шаблон преобразования любого четырёхзначного шестнадцатеричного числа в десятичное:

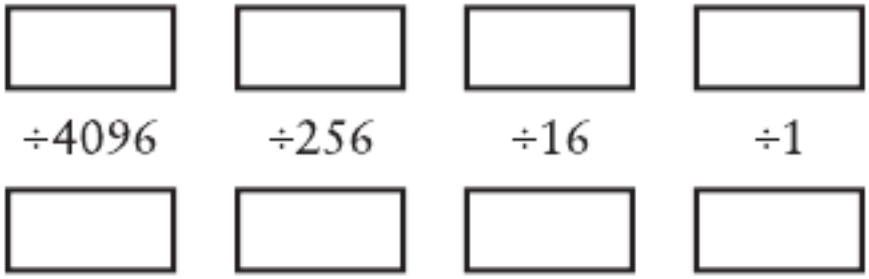


Например, вот преобразование 79ACh. Помните, что шестнадцатеричные цифры A и C равны соответственно десятичным 10 и 12:

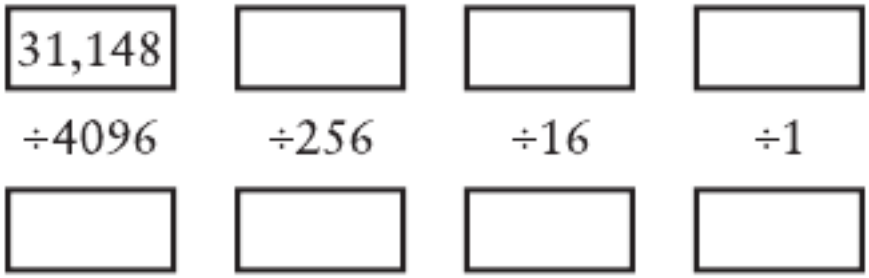


Преобразование десятичных чисел в шестнадцатеричные обычно требует деления. Если число не превышает 255, оно помещается в 1 байт, то есть в две шестнадцатеричные цифры. Чтобы вычислить эти цифры, разделите число на 16 и получите частное и остаток. Например, десятичное число 182 при делении на 16 даёт 11, то есть шестнадцатеричную B, с остатком 6. Шестнадцатеричный эквивалент равен B6h.

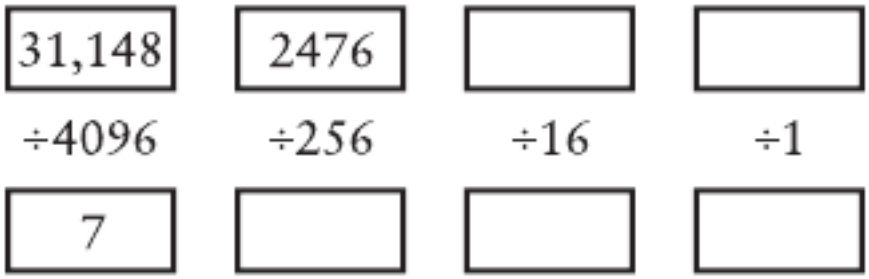
Если преобразуемое десятичное число меньше 65 536, шестнадцатеричный эквивалент займёт не более четырёх цифр. Вот шаблон такого преобразования:



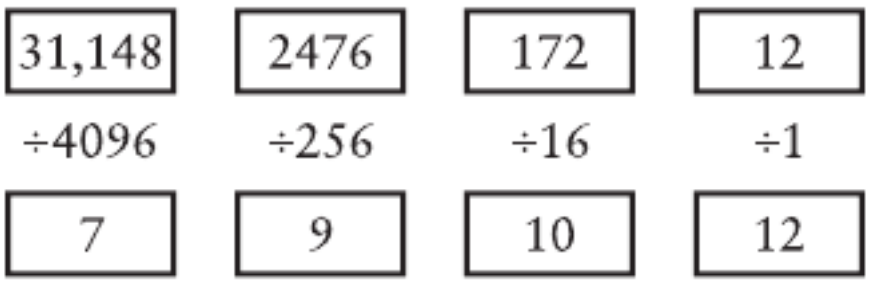
Сначала поместите всё десятичное число в левую верхнюю ячейку:



Разделите число на 4096, но только до получения частного и остатка. Частное помещается в первую нижнюю ячейку, остаток - в следующую верхнюю:

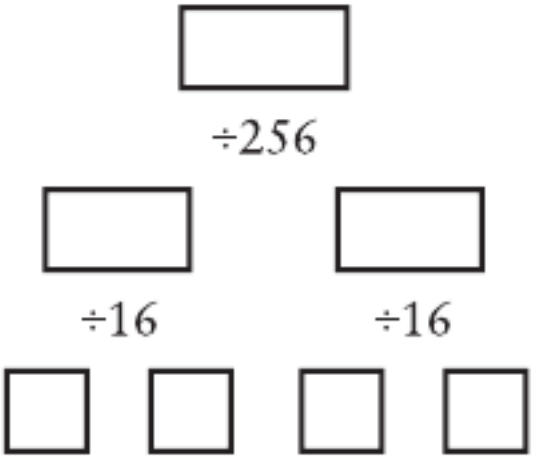


Теперь разделите остаток на 256, снова только до получения частного 9 и нового остатка 172. Продолжите процесс:

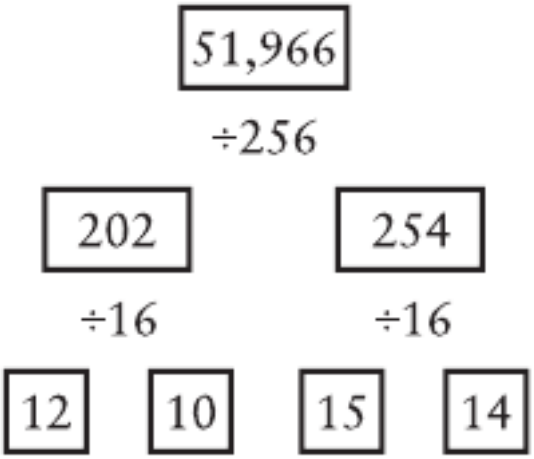


Десятичные числа 10 и 12 соответствуют шестнадцатеричным A и C, поэтому результат равен 79ACh.

Другой способ преобразовать десятичное число до 65 535 в шестнадцатеричное - сначала разделить его на 256 и получить 2 байта. Затем каждый байт разделить на 16. Вот шаблон:



Начните сверху. При каждом делении частное помещается в ячейку слева, а остаток - справа. Например, вот преобразование 51 966:



Шестнадцатеричные цифры равны 12, 10, 15 и 14, то есть CAFE - больше похоже на слово, чем на число! А если вы туда зайдёте, то, возможно, предпочтёте заказать кофе 56 495.

Как и всякому другому основанию, шестнадцатеричной системе соответствует таблица сложения:

+	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

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

Пользуясь таблицей и обычными правилами переноса, можно складывать шестнадцатеричные числа:

```
  4A3378E2
+ 877AB982
----------
  D1AE3264
```

Если вы предпочитаете не выполнять вычисления вручную, калькуляторы Windows и macOS имеют режим программиста, позволяющий считать в двоичной, восьмеричной и шестнадцатеричной системах и преобразовывать числа между ними.

Или можно построить восьмибитовый двоичный сумматор из главы 14.

Глава 13. От ASCII к Unicode

Каждый раз, когда мы касаемся планшета, тычем пальцем в телефон или садимся за ноутбук либо настольный компьютер, мы имеем дело с текстом. Мы либо читаем текст, либо набираем его, либо вырезаем и вставляем из одного места в другое - с веб-страниц в текстовые процессоры, из электронной почты в социальные сети, от замеченных в интернете острот к друзьям, которым отправляем сообщения.

Ничего этого не было бы возможно без стандартизованного способа представлять текстовые символы компьютерными битами и байтами. Кодировка символов - без труда самый жизненно важный компьютерный стандарт. Этот стандарт имеет решающее значение для того, чтобы современная связь могла преодолевать различия между компьютерными системами и приложениями, между производителями аппаратного и программного обеспечения и даже между государственными границами.

И всё же представление текста на компьютерах иногда по-прежнему даёт сбой. В начале 2021 года, когда я приступил к переработке этой главы, я получил от провайдера веб-хостинга письмо с темой

```
Weâ€™ve received your payment, thanks.
```

Вы, несомненно, и сами встречали подобные странности, и выглядят они причудливо, но к концу этой главы вы будете точно знать, как такое может произойти.

Эта книга началась с обсуждения двух систем представления текста двоичными кодами. На первый взгляд азбука Морзе может показаться не вполне двоичным кодом, потому что в ней есть короткие точки, более длинные тире и паузы различной длительности между точками и тире.

Но вспомните, что всё в азбуке Морзе кратно длительности точки: тире длится втрое дольше точки, пауза между буквами длится столько же, сколько тире, а пауза между словами - столько же, сколько два тире. Если точка представляет один бит 1, то тире представляет три идущих подряд бита 1, а паузы - последовательности битов 0. Вот слова «HI THERE» в азбуке Морзе и соответствующие им двоичные цифры:

Азбука Морзе относится к *кодам с переменной битовой длиной*, потому что разным символам требуется разное число битов.

В этом отношении шрифт Брайля гораздо проще. Каждый символ представлен массивом из шести точек, и каждая точка может быть либо выпуклой, либо невыпуклой. Шрифт Брайля совершенно однозначно является 6-битовым кодом, а значит, каждый символ можно представить 6-битовым значением. Есть лишь одна небольшая загвоздка: для представления чисел и прописных букв нужны дополнительные символы Брайля. Возможно, вы помните, что для чисел в шрифте Брайля требуется *код переключения* - символ Брайля, изменяющий значение последующих символов.

Коды переключения встречаются и в другом раннем двоичном коде, изобретённом в связи с печатающим телеграфом в 1870-х годах. Это была работа Эмиля Бодо, служащего французской телеграфной службы, и код до сих пор известен под его именем. Код Бодо использовался вплоть до 1960-х годов - например, компанией Western Union для отправки и приёма текстовых сообщений, называемых *телеграммами*. Даже сегодня можно услышать, как компьютерный старожил называет скорость передачи двоичных данных *скоростью в бодах*.

Код Бодо часто применялся в *телетайпе* - устройстве с клавиатурой, похожей на клавиатуру пишущей машинки, но имеющей лишь 30 клавиш и пробел. Клавиши представляют собой переключатели, которые заставляют двоичный код генерироваться и передаваться по выходному кабелю телетайпа, бит за битом. Телетайп также содержит печатающий механизм. Коды, приходящие по его входному кабелю, приводят в действие электромагниты, печатающие символы на бумаге.

Бодо - 5-битовый код, поэтому в нём существует лишь 32 возможных кода, в шестнадцатеричной записи от 00h до 1Fh. Вот как эти 32 доступных кода соответствуют 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 ничему не назначен. Из оставшегося 31 кода 26 назначены буквам алфавита, а остальные пять обозначены в таблице словами или словосочетаниями, набранными курсивом.

Код 04h - это код пробела, отделяющего слова друг от друга. Коды 02h и 08h называются возвратом каретки и переводом строки. Эта терминология пришла от пишущих машинок: когда вы печатаете на машинке и достигаете конца строки, вы нажимаете рычаг или кнопку, выполняющие два действия. Во-первых, каретка с бумагой перемещается вправо (либо печатающий механизм - влево), чтобы следующая строка началась у левого края бумаги. Это возврат каретки. Во-вторых, машинка поворачивает валик с бумагой так, чтобы следующая строка оказалась под только что законченной. Это перевод строки. В коде Бодо эти два действия представлены отдельными кодами, и принтер телетайпа Бодо реагирует на них при печати.

А где в системе Бодо числа и знаки препинания? Для них предназначен код 1Bh, обозначенный в таблице как «Переключение на цифры». После кода переключения на цифры все последующие коды интерпретируются как числа или знаки препинания, пока код переключения на буквы (1Fh) не вернёт им буквенное значение. Вот коды чисел и знаков препинания:

Шестнадцатеричный код	Цифра или знак Бодо	Шестнадцатеричный код	Цифра или знак Бодо
00		10	3
01	5	11	+
02	Возврат каретки	12	Кто вы?
03	9	13	?
04	Пробел	14	'
05	#	15	6
06	,	16	$
07	.	17	/
08	Перевод строки	18	-
09	)	19	2
0A	4	1A	Звонок
0B	&	1B	Переключение на цифры
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	Переключение на буквы

Эта таблица показывает, как коды использовались в Соединённых Штатах. За пределами США коды 05h, 0Bh и 16h часто применяли для букв с диакритическими знаками, встречающихся в некоторых европейских языках. Код звонка должен был включать слышимый звонок телетайпа. Код «Кто вы?» приводил в действие механизм, с помощью которого телетайп сообщал свои идентификационные данные.

Подобно азбуке Морзе, код Бодо не различает прописные и строчные буквы. Предложение

```
I SPENT $25 TODAY.
```

представляется следующим потоком шестнадцатеричных данных:

```
 I    S  P  E  N  T       $  2  5       T  O  D  A  Y     .
0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08
```

Обратите внимание на три кода переключения: 1Bh непосредственно перед знаком доллара, 1Fh после числа и ещё один 1Bh перед завершающей точкой. Строка заканчивается кодами возврата каретки и перевода строки.

К сожалению, если дважды подряд отправить этот поток данных на принтер телетайпа, он напечатает следующее:

```
I SPENT $25 TODAY.
8 '03,5 $25 TODAY.
```

Что произошло? Последним кодом переключения, полученным принтером перед второй строкой, был код переключения на цифры, поэтому до следующего кода переключения на буквы коды в начале второй строки интерпретируются как числа.

Подобные неполадки - типичные скверные последствия применения кодов переключения. Когда настало время заменить код Бодо чем-то более современным и универсальным, было решено, что лучше обойтись без кодов переключения и определить отдельные коды для строчных и прописных букв.

Сколько битов нужно для такого кода? Если ограничиться английским языком и начать подсчёт символов, то только для прописных и строчных букв латинского алфавита потребуется 52 кода, а для цифр от 0 до 9 - ещё десять. Получается уже 62. Добавьте несколько знаков препинания, и число превысит 64 - предел для 6 битов. Но до превышения 128 символов, которое потребовало бы уже 8 битов, ещё остаётся некоторый запас.

Итак, ответ - 7. Чтобы без кодов переключения представить все символы, обычно встречающиеся в английском тексте, нужно 7 битов.

На смену коду Бодо пришёл 7-битовый код под названием *American Standard Code for Information Interchange* - Американский стандартный код для обмена информацией, сокращённо ASCII и с несколько неожиданным произношением ['аски]. Он был официально утверждён в 1967 году и остаётся важнейшим стандартом всей компьютерной отрасли. За одним существенным исключением (которое я вскоре опишу), встречая на компьютере текст, можно быть уверенным, что ASCII так или иначе имеет к нему отношение.

Будучи 7-битовым кодом, ASCII использует двоичные коды от 0000000 до 1111111, то есть шестнадцатеричные коды от 00h до 7Fh. Скоро вы увидите все 128 кодов ASCII, но я хочу разделить их на четыре группы по 32 кода и сначала пропустить первые 32, потому что концептуально они несколько сложнее остальных. Вторая группа из 32 кодов включает знаки препинания и десять десятичных цифр. В следующей таблице показаны шестнадцатеричные коды от 20h до 3Fh и соответствующие им символы:

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
20	Пробел	30	0
21	!	31	1
22	"	32	2
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	<

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

Обратите внимание, что 20h - это символ пробела, разделяющий слова и предложения.

Следующие 32 кода включают прописные буквы и несколько дополнительных знаков препинания. За исключением знака @ и символа подчёркивания, эти знаки обычно не встречаются на пишущих машинках, но стали стандартными на компьютерных клавиатурах.

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
4B	K	5B	[
4C	L	5C	\
4D	M	5D	]
4E	N	5E	^
4F	O	5F	_

Следующие 32 символа включают все строчные буквы и ещё несколько знаков препинания, опять-таки редко встречающихся на пишущих машинках, но стандартных для компьютерных клавиатур:

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

Шестнадцатеричный код	Символ ASCII	Шестнадцатеричный код	Символ ASCII
6A	j	7A	z
6B	k	7B	{
6C	l	7C	|
6D	m	7D	}
6E	n	7E	~
6F	o		

Обратите внимание, что в этой таблице нет последнего символа, соответствующего коду 7Fh. Вы увидите его совсем скоро.

Текстовую строку

```
Hello, you!
```

можно представить в ASCII следующими шестнадцатеричными кодами:

```
 H  e  l  l  o  ,     y  o  u  !
48 65 6C 6C 6F 2C 20 79 6F 75 21
```

Обратите внимание на запятую (код 2Ch), пробел (код 20h), восклицательный знак (код 21h), а также коды букв. Вот ещё одно короткое предложение:

```
I am 12 years old.
```

И его представление в ASCII:

```
 I     a  m     1  2     y  e  a  r  s     o  l  d  .
49 20 61 6D 20 31 32 20 79 65 61 72 73 20 6F 6C 64 2E
```

Обратите внимание, что число 12 в этом предложении представлено шестнадцатеричными числами 31h и 32h, то есть кодами ASCII для цифр 1 и 2. Когда число 12 является частью потока текста, его *не следует* представлять шестнадцатеричными кодами 01h и 02h или шестнадцатеричным кодом 0Ch. Все эти коды означают в ASCII нечто иное.

Код каждой прописной буквы в ASCII отличается от кода соответствующей строчной буквы на 20h. Благодаря этому компьютерным программам очень легко преобразовывать регистр букв: достаточно прибавить 20h к коду прописной буквы, чтобы получить строчную, и вычесть 20h, чтобы превратить строчную в прописную. (Но даже складывать не требуется. Для преобразования между прописной и строчной буквами достаточно изменить единственный бит. Позже в этой книге вы увидите приёмы выполнения подобных задач.)

Говорят, что 95 только что рассмотренных кодов ASCII относятся к *графическим символам*, потому что имеют визуальное представление. В ASCII также входят 33 *управляющих символа*, которые не имеют визуального представления, а вместо этого выполняют определённые функции. Ради полноты ниже приведены все 33 управляющих символа ASCII, но не беспокойтесь, если большинство из них покажется совершенно непонятным. Когда разрабатывался ASCII, он предназначался главным образом для телетайпов, и многие из этих кодов теперь почти забыты.

Шестнадцатеричный код	Сокращение	Название управляющего символа ASCII
00	NUL	Null (ничего)
01	SOH	Начало заголовка
02	STX	Начало текста
03	ETX	Конец текста
04	EOT	Конец передачи

Шестнадцатеричный код	Сокращение	Название управляющего символа ASCII
05	ENQ	Запрос
06	ACK	Подтверждение
07	BEL	Звонок
08	BS	Возврат на одну позицию
09	HT	Горизонтальная табуляция
0A	LF	Перевод строки
0B	VT	Вертикальная табуляция
0C	FF	Перевод страницы
0D	CR	Возврат каретки
0E	SO	Переключение наружу
0F	SI	Переключение внутрь
10	DLE	Управляющий символ канала данных
11	DC1	Управление устройством 1
12	DC2	Управление устройством 2
13	DC3	Управление устройством 3
14	DC4	Управление устройством 4
15	NAK	Отрицательное подтверждение
16	SYN	Синхронный простой
17	ETB	Конец блока передачи
18	CAN	Отмена
19	EM	Конец носителя
1A	SUB	Символ замены
1B	ESC	Выход
1C	FS	Разделитель файлов, или разделитель информации 4
1D	GS	Разделитель групп, или разделитель информации 3
1E	RS	Разделитель записей, или разделитель информации 2
1F	US	Разделитель единиц, или разделитель информации 1
7F	DEL	Удаление

Смысл состоит в том, что управляющие символы можно смешивать с графическими, выполняя простейшее форматирование текста. Легче всего понять это, если представить устройство - например, телетайп или простой принтер, - которое печатает символы на странице в ответ на поток кодов ASCII. Обычно печатающая головка устройства реагирует на код символа, печатая этот символ и перемещаясь на одну позицию вправо. Наиболее важные управляющие символы изменяют это поведение.

Рассмотрим, например, шестнадцатеричную последовательность символов

```
41 09 42 09 43 09
```

Символ 09 - это код горизонтальной табуляции, сокращённо *табуляция*. Если считать все горизонтальные позиции символов на странице принтера пронумерованными начиная с 0, то код табуляции обычно означает, что следующий символ следует напечатать в ближайшей следующей позиции, кратной 8, вот так:

```
A       B       C
```

Это удобный способ выравнивать текст по столбцам.

Даже сегодня некоторые компьютерные принтеры отвечают на код перевода страницы (0Ch), выбрасывая текущий лист и начиная новый.

На некоторых старых принтерах код возврата на одну позицию можно применять для печати составных символов. Допустим, компьютер, управляющий телетайпом, должен вывести строчную е со знаком грависа: è. Этого можно добиться шестнадцатеричными кодами 65 08 60.

Безусловно, важнейшие управляющие коды - возврат каретки и перевод строки, имеющие тот же смысл, что и сходные коды Бодо. В некоторых старых компьютерных принтерах код возврата каретки перемещал печатающую головку к левому краю страницы, оставляя её на той же строке, а код перевода строки опускал головку на одну строку. Для перехода на новую строку обычно требовались оба кода. Один возврат каретки позволял печатать поверх уже существующей строки, а один перевод строки - перейти к следующей строке, не возвращаясь к левому полю.

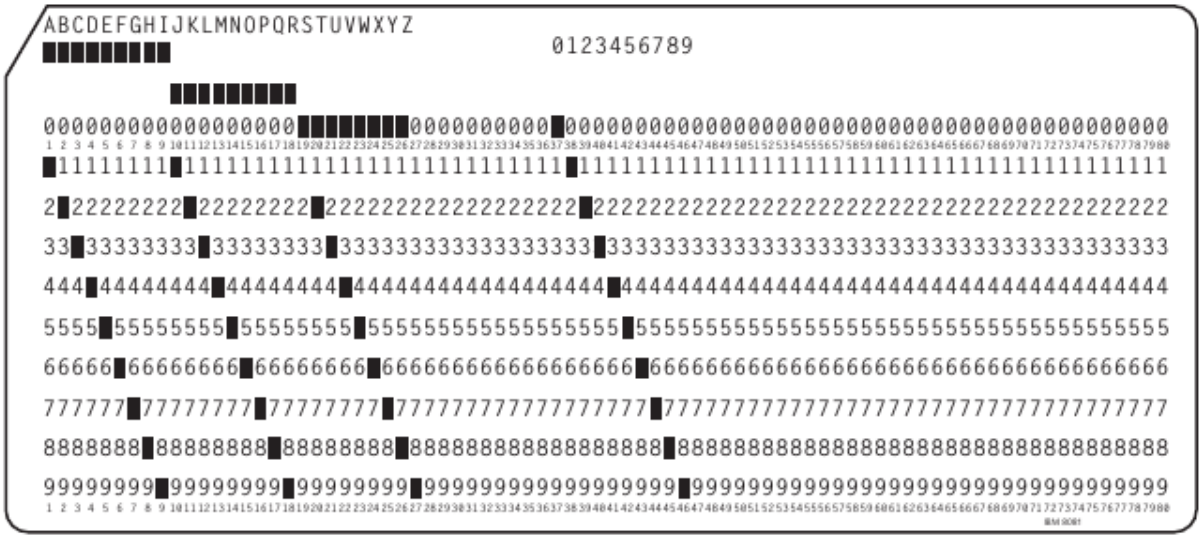
Текст, изображения, музыка и видео могут храниться в компьютере в виде *файлов* - наборов байтов, обозначенных именем. Имена файлов часто состоят из описательного имени, указывающего содержимое файла, и расширения - обычно трёх или четырёх букв, указывающих тип файла. Файлы, состоящие из символов ASCII, часто имеют расширение имени `txt`, означающее «text», то есть «текст». В ASCII нет кодов для курсива, полужирного начертания, различных шрифтов и размеров шрифта. Все эти украшения свойственны так называемому *форматированному тексту*, или *богатому тексту*. ASCII предназначен для *простого текста*. На настольном компьютере с Windows файлы простого текста можно создавать в программе Notepad; в macOS то же делает программа TextEdit (хотя по умолчанию она ведёт себя иначе). Обе программы позволяют выбрать шрифт и его размер, но это влияет только на просмотр текста. Эти сведения не сохраняются вместе с самим текстом.

И Notepad, и TextEdit в ответ на клавишу Enter или Return завершают текущую строку и переходят к началу следующей. Но эти программы также выполняют *перенос по словам*: когда при наборе вы достигаете правого края окна, программа автоматически продолжает ввод на следующей строке, и перенесённый текст на самом деле остаётся частью абзаца, а не превращается в отдельные строки. Клавишу Enter или Return вы нажимаете, чтобы отметить конец этого абзаца и начать новый.

Когда вы нажимаете Enter или Return, Windows Notepad вставляет в файл шестнадцатеричные коды 0Dh и 0Ah - символы возврата каретки и перевода строки. macOS TextEdit вставляет только 0Ah, перевод строки. Так называемая классическая Mac OS (существовавшая с 1984 по 2001 год) вставляла только 0Dh, возврат каретки. Эта несогласованность продолжает вызывать проблемы, когда файл, созданный в одной системе, читают в другой. В последние годы программисты старались уменьшить эти проблемы, но всё равно удивительно - даже постыдно, - что в компьютерной отрасли до сих пор нет стандарта обозначения конца строк или абзацев в файле простого текста.

Вскоре после появления ASCII стал преобладающим текстовым стандартом компьютерного мира, но только не в IBM. В связи с System/360 компания IBM разработала собственный код символов,

известный как *Extended BCD Interchange Code*, или EBCDIC. Это было 8-битовое расширение более раннего 6-битового кода BCDIC, который, в свою очередь, происходил от кодов, применявшихся на перфокартах IBM. Перфокарта такого типа, способная хранить 80 текстовых символов, была представлена IBM в 1928 году и использовалась более 50 лет.



Чёрные прямоугольники - это отверстия, пробитые в карте. У перфокарт есть практическая особенность, влияющая на способ представления символов: если пробить слишком много отверстий, карта может потерять прочность, порваться и заклинить машину.

Символ кодируется на перфокарте сочетанием одного или нескольких прямоугольных отверстий в одном столбце. Сам символ часто печатается возле верхнего края карты. Нижние десять рядов называются *цифровыми рядами* и обозначаются числами: ряд 0, ряд 1 и так далее до ряда 9. Это наследие компьютерных систем, непосредственно работавших с десятичными числами. Два непронумерованных ряда возле верхнего края называются *зонными рядами*: это ряд 11 и самый верхний ряд 12. Ряда 10 нет.

Коды символов EBCDIC представляют собой сочетания зонных и цифровых пробивок. Коды EBCDIC для десяти цифр лежат в диапазоне от F0h до F9h. Коды EBCDIC для прописных букв делятся на три группы: от C1h до C9h, от D1h до D9h и от E2h до E9h. Коды EBCDIC для строчных букв тоже делятся на три группы: от 81h до 89h, от 91h до 99h и от A2h до A9h.

В ASCII все прописные и строчные буквы образуют непрерывные последовательности. Благодаря этому данные ASCII удобно сортировать по алфавиту. Но в последовательностях букв EBCDIC есть разрывы, усложняющие сортировку. К счастью, теперь EBCDIC в основном представляет исторический интерес, и в личной или профессиональной жизни вы вряд ли с ним столкнётесь.

Во время разработки ASCII память стоила очень дорого. Некоторые считали, что для её экономии ASCII должен быть 6-битовым кодом с символом переключения между строчными и прописными буквами. После отказа от этой идеи другие настаивали, что ASCII должен стать 8-битовым, поскольку компьютеры с 8-битовой архитектурой представлялись более вероятными, чем компьютеры с 7-битовой. Конечно, теперь стандартом стали 8-битовые байты, и хотя технически ASCII является 7-битовым кодом, почти всегда он хранится в виде 8-битовых значений.

Соответствие между байтами и символами ASCII, несомненно, удобно: просто подсчитав символы, можно приблизительно понять, сколько компьютерной памяти требуется конкретному текстовому документу. Например, роман Германа Мелвилла *«Моби Дик, или Белый кит»* содержит около 1,25 миллиона символов и потому занимает 1,25 миллиона байтов компьютерного хранилища. Из этих сведений можно вывести и приблизительное число слов: считается, что среднее слово имеет длину пять символов, а если учесть пробел между словами, то в *«Моби Дике»* около 200 тысяч слов.

Файл простого текста с *«Моби Диком»* можно загрузить с сайта проекта «Гутенберг» (`gutenberg.org`), где размещено множество других классических произведений, перешедших в общественное достояние. Хотя проект «Гутенберг» был пионером распространения книг в виде простого текста, он также предлагает те же книги в нескольких форматах электронных книг и в HTML (Hypertext Markup Language, язык гипертекстовой разметки).

Будучи форматом веб-страниц всего интернета, HTML, несомненно, является самым популярным форматом богатого текста. HTML добавляет к простому тексту сложное форматирование с помощью фрагментов *разметки*, или *тегов*. Но интересно, что для разметки HTML использует обычные символы ASCII, поэтому HTML-файл одновременно является обычным файлом простого текста. При просмотре как простой текст HTML выглядит так:

```
This is some <b>bold</b> text, and this is some <i>italic</i> text.
```

Угловые скобки - всего лишь коды ASCII 3Ch и 3Eh. Но если интерпретировать эту строку как HTML, веб-браузер может показать её так:

This is some **bold** text, and this is some *italic* text.

Текст один и тот же, только отображается по-разному.

ASCII, безусловно, важнейший стандарт компьютерной отрасли, но его недостатки были очевидны с самого начала. Главная проблема в том, что American Standard Code for Information Interchange уж слишком американский! Более того, ASCII едва ли подходит даже другим странам, основной язык которых - английский. В ASCII есть знак доллара, но где знак британского фунта? Ещё хуже он справляется с буквами с диакритическими знаками, применяемыми во многих западноевропейских языках, не говоря уже о нелатинских алфавитах Европы - греческом, арабском, еврейском и кириллице - или о происходящих от письма брахми письменностях Индии и Юго-Восточной Азии, включая деванагари, бенгальское, тайское и тибетское письмо. И как 7-битовый код вообще может справиться с *десятками тысяч* идеограмм китайского, японского и корейского языков и примерно десятью тысячами слогов корейского хангыля?

Включить в ASCII все языки мира было бы в 1960-х годах чрезмерно смелой задачей, однако потребности некоторых других стран всё же учитывались, пусть и посредством примитивных решений. Согласно опубликованному стандарту ASCII, десять кодов ASCII (40h, 5Bh, 5Ch, 5Dh, 5Eh, 60h, 7Bh, 7Ch, 7Dh и 7Eh) разрешено переопределять для национального применения. Кроме того, знак номера (#) можно заменить знаком британского фунта (£), а знак доллара ($) - обобщённым знаком валюты (¤). Очевидно, замена символов имеет смысл лишь в том случае, если все, кто работает с содержащим переопределённые коды текстовым документом, знают об изменении.

Поскольку многие компьютерные системы хранят символы как 8-битовые значения, можно создать так называемый *расширенный набор символов ASCII*, содержащий 256, а не 128 символов. В таком наборе первые 128 кодов с шестнадцатеричными значениями от 00h до 7Fh определяются точно так же, как в ASCII, а следующие 128 кодов (от 80h до FFh) могут означать что угодно. Этим способом определяли дополнительные коды символов для букв с диакритическими знаками и нелатинских алфавитов. К сожалению, ASCII расширяли *много* раз и *множеством разных* способов.

Первая версия Microsoft Windows поддерживала расширение ASCII, которое Microsoft называла набором символов ANSI, хотя в действительности Американский национальный институт стандартов его не утверждал. Дополнительные символы с кодами от A0h до FFh - это главным образом полезные знаки и буквы с диакритикой, часто встречающиеся в европейских языках. В следующей таблице старшая шестнадцатеричная тетрада кода символа показана в верхней строке, а младшая - в левом столбце:

Младшая тетрада	**A-**	**B-**	**C-**	**D-**	**E-**	**F-**
-0	Неразрывный пробел	°	À	Ð	à	ð
-1	¡	±	Á	Ñ	á	ñ
-2	¢	²	Â	Ò	â	ò
-3	£	³	Ã	Ó	ã	ó
-4	¤	´	Ä	Ô	ä	ô
-5	¥	µ	Å	Õ	å	õ
-6	¦	¶	Æ	Ö	æ	ö
-7	§	·	Ç	×	ç	÷
-8	¨	¸	È	Ø	è	ø
-9	©	¹	É	Ù	é	ù
-A	ª	º	Ê	Ú	ê	ú
-B	«	»	Ë	Û	ë	û
-C	¬	¼	Ì	Ü	ì	ü
-D	Мягкий перенос	½	Í	Ý	í	ý
-E	®	¾	Î	Þ	î	þ
-F	¯	¿	Ï	ß	ï	ÿ

Символ с кодом A0h определяется как *неразрывный пробел*. Обычно, когда компьютерная программа форматирует текст в строки и абзацы, она разрывает строку на символе пробела, имеющем код ASCII 20h. Код A0h должен отображаться как пробел, но разрывать на нём строку нельзя. Неразрывный пробел можно применить, например, в дате «2 февраля», чтобы число 2 не оказалось в одной строке, а «февраля» - в следующей.

Код ADh определяется как *мягкий перенос*. Это дефис, которым разделяют слоги внутри слова. На печатной странице он появляется только тогда, когда слово необходимо перенести с одной строки на другую.

Набор символов ANSI приобрёл популярность благодаря Windows, но был лишь одним из многочисленных *различных* расширений ASCII, определённых за многие десятилетия. Чтобы различать их, им стали присваивать номера и другие идентификаторы. Набор ANSI из Windows превратился в стандарт Международной организации по стандартизации, известный как ISO-8859-1, или Latin Alphabet No. 1. Когда этот набор символов, в свою очередь, расширили символами для кодов от 80h до 9Fh, он стал называться Windows-1252:

Младшая тетрада	**8-**	**9-**	**A-**	**B-**	**C-**	**D-**	**E-**	**F-**
-0	€		Неразрывный пробел	°	À	Ð	à	ð
-1		Левая одиночная кавычка	¡	±	Á	Ñ	á	ñ
-2	Нижняя одиночная кавычка	Правая одиночная кавычка	¢	²	Â	Ò	â	ò
-3	*ƒ*	Левая двойная кавычка	£	³	Ã	Ó	ã	ó
-4	Нижняя двойная кавычка	Правая двойная кавычка	¤	´	Ä	Ô	ä	ô
-5	…	•	¥	µ	Å	Õ	å	õ
-6	†	Короткое тире	¦	¶	Æ	Ö	æ	ö
-7	‡	Длинное тире	§	·	Ç	×	ç	÷
-8	ˆ	˜	¨	¸	È	Ø	è	ø
-9	‰	™	©	¹	É	Ù	é	ù
-A	Š	š	ª	º	Ê	Ú	ê	ú
-B	‹	›	«	»	Ë	Û	ë	û
-C	Œ	œ	¬	¼	Ì	Ü	ì	ü
-D			Мягкий перенос	½	Í	Ý	í	ý
-E	Ž	ž	®	¾	Î	Þ	î	þ
-F		Ÿ	¯	¿	Ï	ß	ï	ÿ

Число 1252 называется идентификатором *кодовой страницы* - этот термин появился в IBM для различения разных версий EBCDIC. Разные кодовые страницы связывались со странами, которым требовались собственные буквы с диакритическими знаками и даже целые алфавиты, например греческий, кириллица и арабский. Чтобы правильно отобразить символьные данные, необходимо было знать, какая кодовая страница использована. Особенно важным это стало в интернете, где сведения в начале HTML-файла (так называемом *заголовке*) указывают кодовую страницу, применённую при создании веб-страницы.

Для кодирования идеограмм китайского, японского и корейского языков ASCII расширяли и более радикальными способами. В одной популярной кодировке, Shift-JIS (Japanese Industrial Standard), коды от 81h до 9Fh фактически обозначают начальный байт 2-байтового кода символа. Благодаря этому Shift-JIS позволяет закодировать около 6000 дополнительных символов. К сожалению,

Shift-JIS - не единственная система, использующая такой приём. В Азии получили распространение ещё три стандартных *двухбайтовых набора символов* (DBCS).

Существование нескольких несовместимых двухбайтовых наборов - лишь одна из их проблем. Другая состоит в том, что одни символы - в частности, обычные символы ASCII - представлены 1-байтовыми кодами, тогда как тысячи идеограмм представлены 2-байтовыми. Работать с такими наборами символов трудно.

Если всё это кажется вам беспорядком, вы не одиноки. Так не мог бы кто-нибудь, *пожалуйста*, придумать решение?

Исходя из предположения, что предпочтительнее иметь одну однозначную систему кодирования символов, пригодную для всех языков мира, в 1988 году несколько крупных компьютерных компаний объединились и начали разрабатывать альтернативу ASCII под названием *Unicode*. Если ASCII является 7-битовым кодом, то Unicode - 16-битовый. (По крайней мере, такова была первоначальная идея.) В первоначальном замысле каждый без исключения символ Unicode должен был занимать 2 байта, а коды символов от 0000h до FFFFh позволяли представить 65 536 различных символов. Этого сочли достаточным для всех языков мира, которые, вероятно, будут применяться в компьютерной связи, и ещё оставалось место для расширения.

Unicode не начинали с чистого листа. Первые 128 символов Unicode - коды от 0000h до 007Fh - совпадают с символами ASCII. Кроме того, коды Unicode от 00A0h до 00FFh совпадают с описанным выше расширением ASCII Latin Alphabet No. 1. В Unicode включены и другие международные стандарты.

Хотя код Unicode - всего лишь шестнадцатеричное значение, стандартный способ его записи состоит в том, чтобы поставить перед значением прописную букву U и знак плюс. Вот несколько показательных символов Unicode:

Шестнадцатеричный код	Символ	Описание
U+0041	A	Латинская прописная буква A
U+00A3	£	Знак фунта
U+03C0	π	Греческая строчная буква пи
U+0416	Ж	Кириллическая прописная буква Ж
U+05D0	א	Еврейская буква алеф
U+0BEB	௫	Тамильская цифра пять
U+2018	‘	Левая одиночная кавычка
U+2019	’	Правая одиночная кавычка
U+20AC	€	Знак евро
U+221E	∞	Бесконечность

Ещё гораздо больше символов можно найти на сайте Консорциума Unicode `unicode.org`, предлагающем увлекательное путешествие по богатству письменных языков и символики мира. Прокрутите главную страницу до конца и щёлкните Code Charts, чтобы открыть путь к изображениям такого количества символов, какое вы едва ли могли вообразить.

Но переход от 8-битового кода символов к 16-битовому порождает собственные проблемы: разные компьютеры читают 16-битовые значения по-разному. Рассмотрим, например, два байта:

```
20h ACh
```

Некоторые компьютеры прочтут эту последовательность как 16-битовое значение 20ACh - код Unicode для знака евро. Такие компьютеры называют машинами с порядком *от старшего к младшему*, то есть старший байт (большой конец) идёт первым. Другие компьютеры являются машинами с порядком *от младшего к старшему*. (Терминология происходит из «Путешествий Гулливера», где Джонатан Свифт описывает спор о том, с какого конца следует разбивать яйцо всмятку.) Машины с порядком от младшего к старшему прочитают это значение как AC20h, а в Unicode это символ 가 корейского алфавита хангыль.

Чтобы обойти эту проблему, Unicode определяет специальный символ под названием *метка порядка байтов*, или BOM, имеющий код U+FEFF. Его полагается помещать в начало файла с 16-битовыми значениями Unicode. Если первые два байта файла равны FEh и FFh, в файле используется порядок от старшего к младшему. Если они равны FFh и FEh, порядок идёт от младшего к старшему.

К середине 1990-х годов, когда Unicode только начинал получать распространение, возникла необходимость выйти за пределы 16 битов, чтобы включить вымершие письменности, которые всё же нужны для представления исторических материалов, и многочисленные новые символы. Среди этих новых символов оказались популярные и восхитительные знаки, известные как *эмодзи*.

На момент написания этих строк (в 2021 году) Unicode расширился до 21-битового кода со значениями вплоть до U+10FFFF и потенциально поддерживает более 1 миллиона разных символов. Вот лишь несколько символов, которые не удалось бы вместить в 16-битовый код:

Шестнадцатеричный код	Символ	Описание
U+1302C	𓀬	Египетский иероглиф A039
U+1F025	🀥	Фишка маджонга «Хризантема»
U+1F3BB	🎻	Скрипка
U+1F47D	👽	Инопланетянин
U+1F614	😔	Задумчивое лицо
U+1F639	😹	Кошачья мордочка со слезами радости

Включение эмодзи в Unicode может показаться легкомыслием, но только если вы считаете допустимым, чтобы эмодзи, введённый в текстовое сообщение, отображался на телефоне получателя как нечто совершенно иное. Это может привести к недоразумениям, и отношения способны пострадать!

Конечно, потребности людей в отношении Unicode различны. При отображении идеограмм азиатских языков особенно необходимо широко применять Unicode. У других документов и веб-страниц потребности скромнее. Многим вполне достаточно старого доброго ASCII. Поэтому для хранения и передачи текста Unicode определили несколько разных методов. Они называются *форматами преобразования Unicode*, или UTF.

Самый прямолинейный из форматов преобразования Unicode - UTF-32. Все символы Unicode определяются как 32-битовые значения. Четыре байта, необходимые для каждого символа, могут быть заданы как в порядке от младшего к старшему, так и в порядке от старшего к младшему.

Недостаток UTF-32 состоит в том, что он занимает много места. Файл простого текста с *«Моби Диком»* увеличился бы с 1,25 миллиона байтов в ASCII до 5 миллионов байтов в Unicode. А поскольку Unicode использует лишь 21 из 32 битов, на каждый символ впустую расходуется 11 битов.

Один из компромиссов - UTF-16. В этом формате большинство символов Unicode определяется 2 байтами, а символы с кодами выше U+FFFF - 4 байтами. Для этой цели оставили неназначенной область исходной спецификации Unicode от U+D800 до U+DFFF.

Важнейший формат преобразования Unicode - UTF-8, который теперь широко применяется по всему интернету. Недавняя статистика показывает, что UTF-8 используют 97% всех веб-страниц. Едва ли можно желать более универсального стандарта. Все файлы простого текста проекта «Гутенберг» имеют формат UTF-8. Windows Notepad и macOS TextEdit по умолчанию сохраняют файлы в UTF-8.

UTF-8 - компромисс между гибкостью и компактностью. Главное преимущество UTF-8 состоит в обратной совместимости с ASCII. Это означает, что файл, состоящий только из 7-битовых кодов ASCII, сохранённых как байты, автоматически является файлом UTF-8.

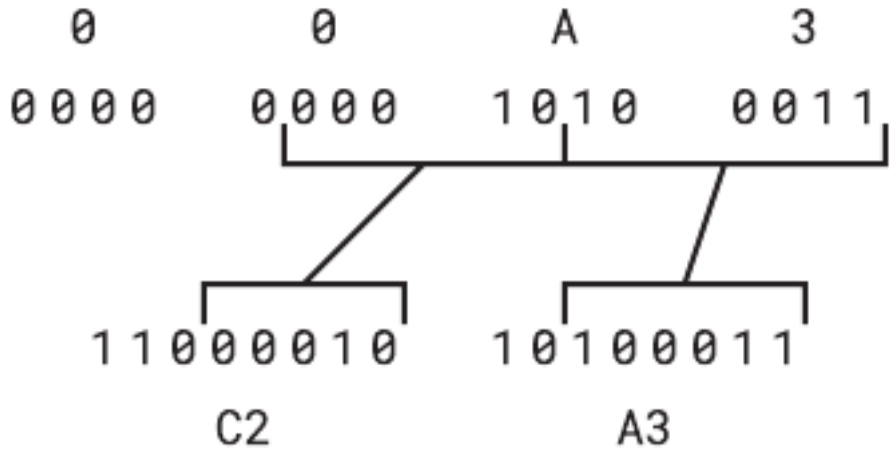
Чтобы такая совместимость была возможна, все остальные символы Unicode хранятся в 2, 3 или 4 байтах в зависимости от их значения. Следующая таблица обобщает работу UTF-8:

Диапазон Unicode	**Число битов**	**Последовательность байтов**
U+0000 - U+007F	7	`0xxxxxxx`
U+0080 - U+07FF	11	`110xxxxx 10xxxxxx`
U+0800 - U+FFFF	16	`1110xxxx 10xxxxxx 10xxxxxx`
U+10000 - U+10FFFF	21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Для диапазонов кодов в первом столбце каждый символ однозначно определяется числом битов, указанным во втором столбце. Затем к этим битам добавляются спереди единицы и нули, как показано в третьем столбце, образуя последовательность байтов. Число знаков x в третьем столбце совпадает с числом во втором.

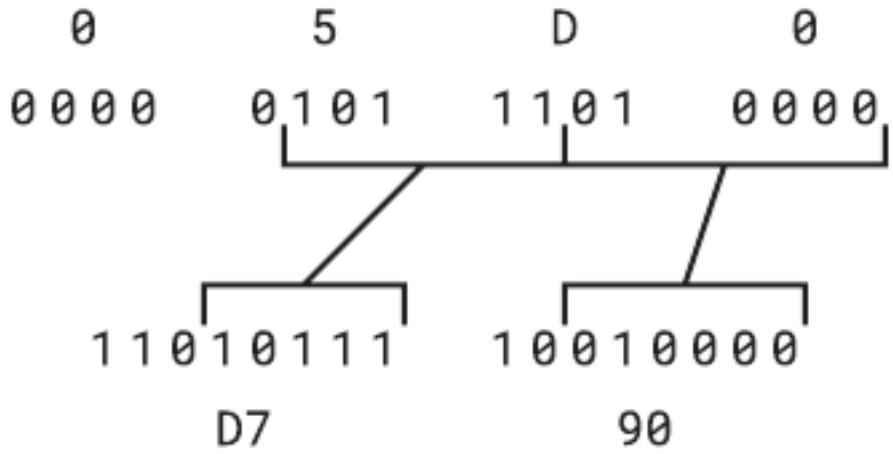
Первая строка таблицы говорит, что если символ входит в исходный набор 7-битовых кодов ASCII, то его кодировка в UTF-8 состоит из бита 0 и следующих за ним 7 битов, то есть совпадает с самим кодом ASCII.

Символам со значениями Unicode U+0080 и выше требуется 2 или более байта. Например, знак британского фунта (£) имеет в Unicode код U+00A3. Поскольку это значение лежит между U+0080 и U+07FF, вторая строка таблицы показывает, что в UTF-8 он кодируется 2 байтами. Для значений этого диапазона при построении 2-байтовой кодировки нужны только младшие 11 битов, как показано здесь:



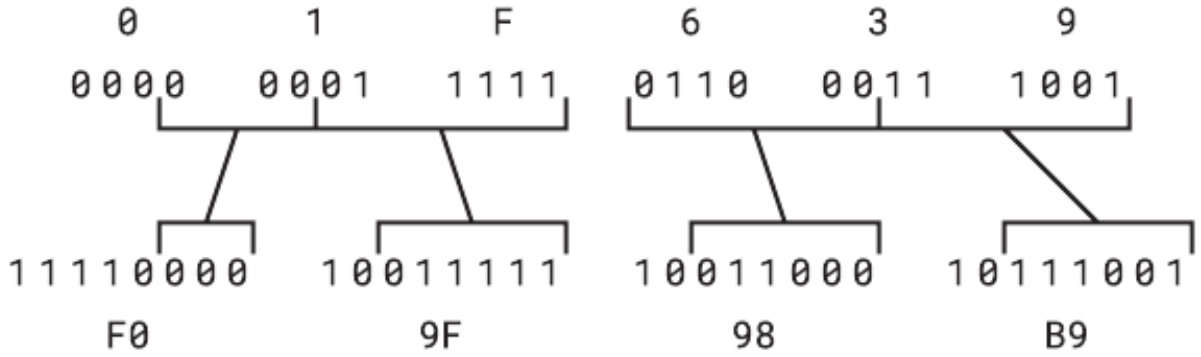
Значение Unicode 00A3 показано в верхней части схемы. Каждой из четырёх шестнадцатеричных цифр соответствует расположенное прямо под ней 4-битовое значение. Мы знаем, что значение не превышает 07FFh, поэтому старшие 5 битов будут нулевыми и их можно отбросить. Перед следующими 5 битами добавляется 110 (как показано в нижней части рисунка), и получается байт C2h. Перед младшими 6 битами добавляется 10, и получается байт A3h. Таким образом, два байта C2h и A3h представляют в UTF-8 знак британского фунта £. Кажется досадным, что для кодирования, по существу, одного байта информации требуется 2 байта, но без этого остальная часть UTF-8 работать не сможет.

Вот другой пример. Еврейская буква א (алеф) имеет в Unicode код U+05D0. Это значение снова лежит между U+0080 и U+07FF, поэтому используется вторая строка таблицы. Процесс тот же, что и для символа £:



Первые 5 битов значения 05D0h можно отбросить; перед следующими 5 битами добавляется 110, а перед младшими 6 битами - 10, в результате чего получаются байты UTF-8 D7h и 90h.

Каким бы невероятным ни казалось само изображение, эмодзи «Кошачья мордочка со слезами радости» представлен кодом Unicode U+1F639, а значит, в UTF-8 он записывается последовательностью из 4 байтов. Следующая схема показывает, как эти 4 байта составляются из 21 бита исходного кода:

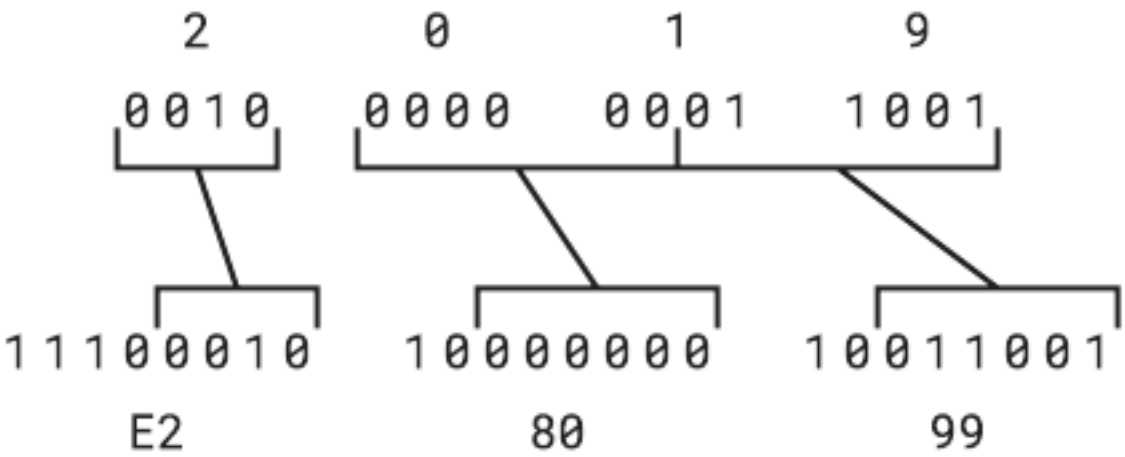


Представляя символы переменным числом байтов, UTF-8 несколько нарушает чистоту и красоту Unicode. В прошлом подобные схемы, применявшиеся вместе с ASCII, вызывали проблемы и путаницу. UTF-8 тоже не полностью защищён от проблем, но определён очень продуманно. При декодировании файла UTF-8 каждый байт можно довольно точно распознать:

• Если байт начинается с нуля, это просто 7-битовый код символа ASCII.
• Если байт начинается с 10, он входит в последовательность байтов, представляющую многобайтовый код символа, но не является первым байтом этой последовательности.

• В противном случае байт начинается как минимум с двух единичных битов и является первым байтом многобайтового кода символа. Общее число байтов этого кода указывает число единичных битов в начале первого байта перед первым нулевым битом. Оно может равняться двум, трём или четырём.

Выполним ещё одно преобразование UTF-8: символ правой одиночной кавычки имеет код U+2019. Здесь нужно обратиться к третьей строке таблицы, поскольку значение лежит между U+0800 и U+FFFF. Представление UTF-8 занимает 3 байта:



Для образования 3 байтов нужны все биты исходного номера Unicode. Перед первыми 4 битами добавляется 1110, перед следующими 6 битами - 10, и перед младшими 6 битами - тоже 10. В результате получается 3-байтовая последовательность E2h, 80h и 99h.

Теперь можно понять проблему с упомянутой в начале главы темой электронного письма:

```
Weâ€™ve received your payment, thanks.
```

Первое слово, очевидно, должно быть «We’ve», но в сокращении используется не старомодный апостроф ASCII (ASCII 27h, или Unicode U+0027), а более изящный символ правой одиночной кавычки Unicode, который, как мы только что увидели, кодируется в UTF-8 тремя байтами E2h, 80h и 99h.

Пока всё в порядке. Но HTML-файл этого письма указывал, что использует набор символов `windows-1252`. Там должно было быть указано `utf-8`, потому что именно так был закодирован текст. Однако поскольку HTML-файл сообщал `windows-1252`, моя почтовая программа применила для интерпретации этих трёх байтов набор символов Windows-1252. Вернитесь к таблице кодов Windows-1252 на странице 162 и убедитесь сами, что три байта E2h, 80h и 99h действительно соответствуют символам â, €, и ™ - в точности тем, что появились в письме.

Тайна раскрыта.

Превратив вычислительную технику в универсальное и многонациональное явление, Unicode стал чрезвычайно важным стандартом. Но, как и всё остальное, он не работает, если применять его неправильно.

Эта страница намеренно оставлена пустой.

Глава 14. Сложение с помощью логических вентилей

Сложение - самая основная арифметическая операция, поэтому, если мы хотим построить компьютер (а именно в этом состоит мой не такой уж скрытый замысел в этой книге), сначала нужно узнать, как построить нечто, складывающее два числа. По существу, сложение - едва ли не *единственное*, что делают компьютеры. Если мы сумеем построить устройство, которое складывает, то значительно приблизимся к созданию устройства, способного с помощью сложения также вычитать, умножать, делить, рассчитывать выплаты по ипотеке, направлять ракеты к Марсу, играть в шахматы и пользоваться социальными сетями, чтобы делиться нашими последними танцевальными движениями, кулинарными подвигами или проделками домашних питомцев.

Суммирующая машина, которую мы построим в этой главе, будет большой, неуклюжей, медленной и шумной - во всяком случае, по сравнению с калькуляторами и компьютерами современной жизни. Самое интересное, что мы собираемся целиком построить её из простых электрических устройств, уже изученных в предыдущих главах: переключателей, лампочек, проводов, батареи и реле, заранее соединённых в различные логические вентили. Все эти компоненты существовали ещё до XX века. И особенно приятно, что нам не придётся в самом деле что-либо сооружать у себя в гостиной: вместо этого мы построим суммирующую машину на бумаге и в своём воображении.

Эта суммирующая машина будет работать исключительно с двоичными числами и лишится некоторых современных удобств. Вы не сможете набрать на клавиатуре числа, которые хотите сложить: вместо этого придётся пользоваться рядом переключателей. Вместо цифрового индикатора для вывода результата у машины будет ряд лампочек.

Но два числа эта машина совершенно точно сложит, причём сделает это почти так же, как складывают числа компьютеры.

Сложение двоичных чисел очень похоже на сложение десятичных. Когда требуется сложить два десятичных числа, например 245 и 673, задача разбивается на более простые шаги. На каждом шаге нужно сложить лишь пару десятичных цифр. В данном примере вы начинаете с 5 плюс 3. Работа идёт гораздо быстрее, если когда-то в жизни вы выучили таблицу сложения.

Большое преимущество двоичных чисел перед десятичными - простота таблицы сложения:

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

Если вы действительно выросли в стае дельфинов и заучили эту таблицу в дельфиньей школе, то могли бы вслух пропищать и просвистеть:

> 0 плюс 0 равно 0.
> 0 плюс 1 равно 1.
> 1 плюс 0 равно 1.
> 1 плюс 1 равно 0, 1 в уме.

Таблицу сложения можно переписать с ведущими нулями, чтобы каждый результат представлял собой 2-битовое значение:

+	0	1
0	00	01
1	01	10

При таком взгляде сложение пары двоичных чисел даёт два бита, называемых *битом суммы* и *битом переноса* (как в выражении «1 плюс 1 равно 0, 1 в уме»). Теперь таблицу двоичного сложения можно разделить на две таблицы. Первая - для бита суммы:

Сумма	0	1
0	0	1
1	1	0

А вторая - для бита переноса:

Перенос	0	1
0	0	0
1	0	1

Рассматривать двоичное сложение таким образом удобно, потому что наша суммирующая машина будет отдельно вычислять сумму и перенос. Чтобы построить двоичную суммирующую машину, нужно спроектировать схему, выполняющую эти операции. Работа исключительно в двоичной системе чрезвычайно упрощает задачу, поскольку все части схемы - переключатели, лампочки и провода - могут представлять двоичные цифры.

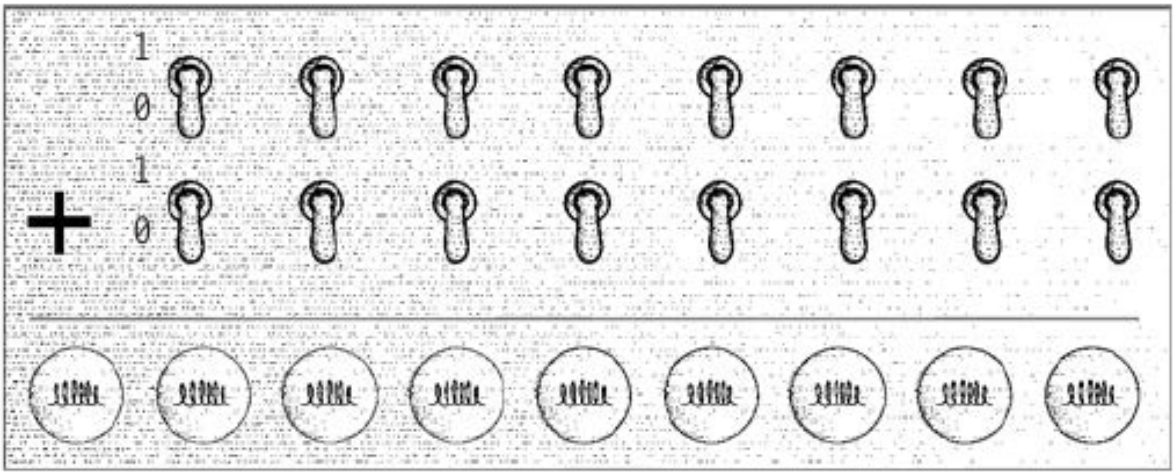
Как и при десятичном сложении, два двоичных числа мы складываем столбец за столбцом, начиная с младшего значащего бита в крайнем правом столбце:

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

Обратите внимание: при сложении третьего справа столбца 1 переносится в следующий столбец. То же происходит в шестом, седьмом и восьмом столбцах справа.

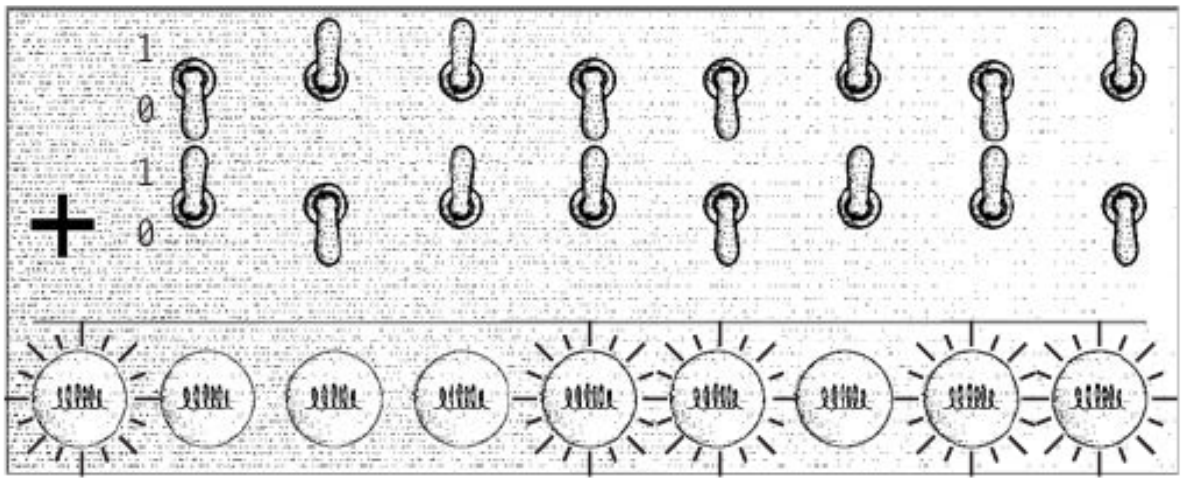
Двоичные числа какого размера мы хотим складывать? Поскольку суммирующую машину мы строим только в своём воображении, можно было бы спроектировать её для очень длинных чисел. Но будем разумны и решим складывать двоичные числа длиной до 8 битов, или 1 байта. Иными словами, мы хотим складывать двоичные числа от 00000000 до 11111111. Эти значения лежат в диапазоне от шестнадцатеричного 00h до FFh, то есть от десятичного 0 до 255. Сумма двух 8-битовых чисел может достигать десятичного 510, шестнадцатеричного 1FEh или двоичного 111111110.

Панель управления нашей двоичной суммирующей машины может выглядеть так:



На панели находятся два ряда по восемь переключателей. Этот набор переключателей служит устройством ввода, и с его помощью мы будем «набирать» два 8-битовых числа. В данном устройстве ввода выключенный переключатель (направленный вниз) означает 0, а включённый (направленный вверх) - 1, как у обычных настенных выключателей в вашем доме. Как всегда, младший значащий бит расположен справа, а старший - слева. Устройство вывода внизу представляет собой ряд из девяти лампочек. Они показывают ответ. Негорящая лампочка означает 0, а горящая - 1. Нужно девять лампочек, потому что сумма двух 8-битовых чисел может оказаться 9-битовым числом. Крайняя левая лампочка загорится только в том случае, если сумма превышает десятичное число 255.

Остальная часть суммирующей машины будет состоять из логических вентилей, соединённых проводами различными способами. Переключатели будут приводить в действие реле логических вентилей, а те, в свою очередь, включат нужные лампочки. Например, если мы захотим сложить 01100101 и 10110110 (два числа из предыдущего примера), то установим соответствующие переключатели так:



Лампочки загораются, показывая ответ 100011011. (Во всяком случае, будем на это надеяться. Ведь машину мы ещё не построили!)

В предыдущей главе я упоминал, что в этой книге буду использовать множество реле. Для создаваемой здесь 8-битовой суммирующей машины требуется не менее 144 реле - по 18 для каждой из восьми складываемых пар битов. Если бы я показал вам завершённую схему целиком, вы определённо пришли бы в ужас. Ни один человек не смог бы разобраться в 144 реле,

странными способами соединённых друг с другом. Поэтому я буду подходить к задаче более простыми последовательными шагами.

Возможно, взглянув на таблицу бита переноса, получаемого при сложении двух 1-битовых чисел, вы сразу заметили связь между логическими вентилями и двоичным сложением:

Перенос	**0**	**1**
0	0	0
1	0	1

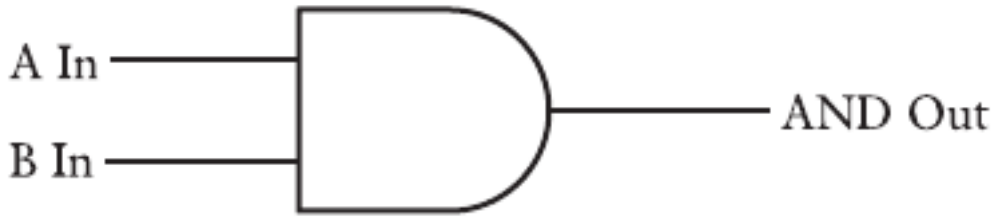
Вероятно, вы поняли, что она совпадает с логической операцией И (AND) и выходом вентиля И, показанного в главе 8:

И	**0**	**1**
0	0	0
1	0	1

Или, если обозначить два входа:

Вход А	**Вход В**	**И**
0	0	0
0	1	0
1	0	0
1	1	1

Вместо множества реле инженеры-электрики обозначают вентиль И так:



Входы слева обозначены А и В - это два складываемых бита. Выход вентиля И справа представляет бит переноса при сложении этих двух двоичных цифр.

Ага! Мы определённо движемся вперёд. Несколько более трудная задача - убедить несколько реле вести себя так:

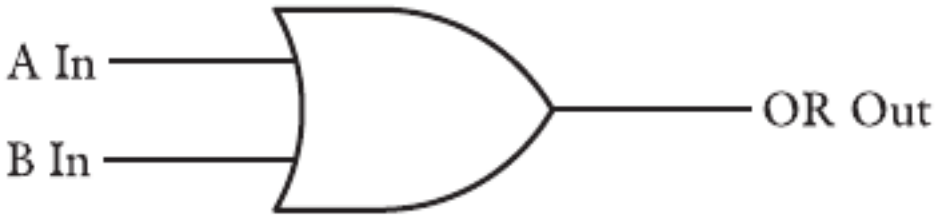
Сумма	**0**	**1**
0	0	1
1	1	0

Это вторая половина задачи сложения пары двоичных цифр. Бит суммы оказывается не столь прямолинейным, как бит переноса, но мы до него доберёмся.

Прежде всего нужно понять, что логическая операция ИЛИ (OR) близка к требуемой, за исключением правой нижней ячейки:

ИЛИ	0	1
0	0	1
1	1	1

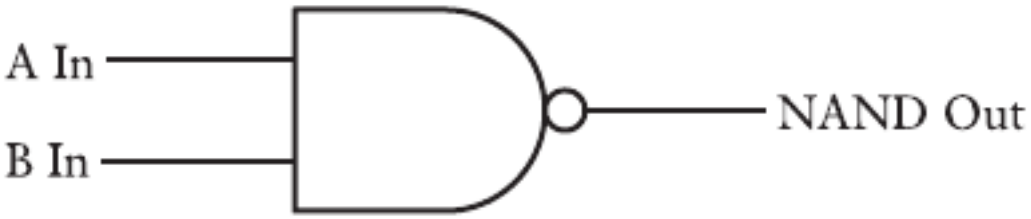
Как вы, возможно, помните из главы 8, вентиль ИЛИ обозначается так:



На нужную нам операцию также похожа логическая операция И-НЕ (NAND, или NOT AND), выход которой противоположен выходу вентиля И. Она совпадает с суммой двух однобитовых чисел, кроме левой верхней ячейки:

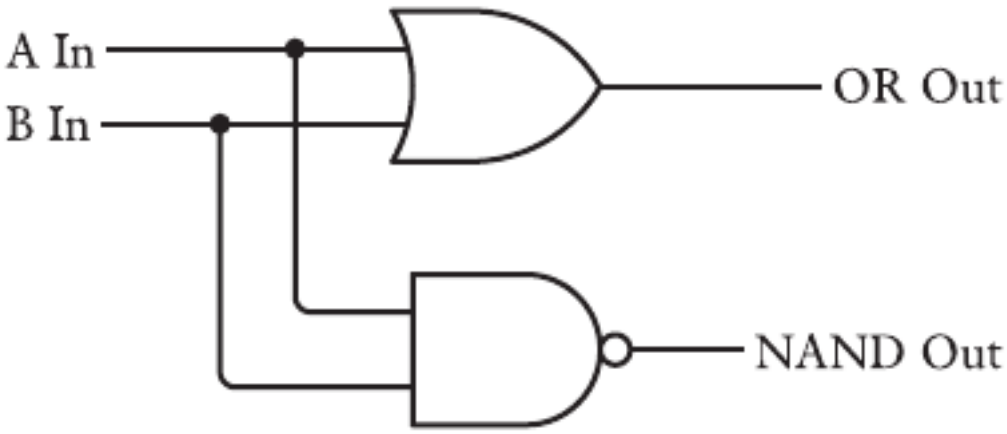
И-НЕ	0	1
0	1	1
1	1	0

Вот как изображается вентиль И-НЕ:



Он совпадает с вентилем И, за исключением маленького кружка справа, показывающего, что выход противоположен результату И.

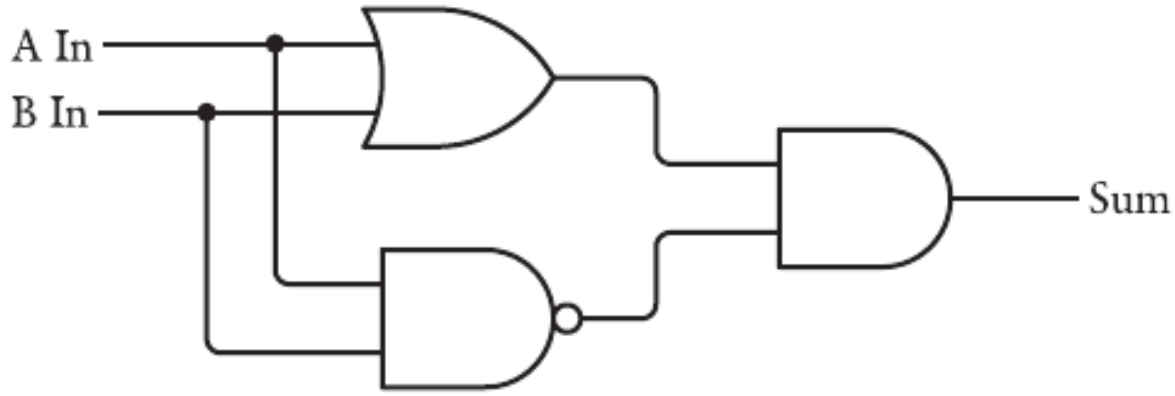
Подключим к одним и тем же входам вентиль ИЛИ и вентиль И-НЕ. Как всегда, маленькие точки показывают места соединения проводов; во всех остальных местах провода лишь пересекаются:



Следующая таблица обобщает выходы этих вентилей ИЛИ и И-НЕ и сравнивает их с тем, что требуется суммирующей машине:

Вход A	Вход B	Выход ИЛИ	Выход И-НЕ	Требуемое значение
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

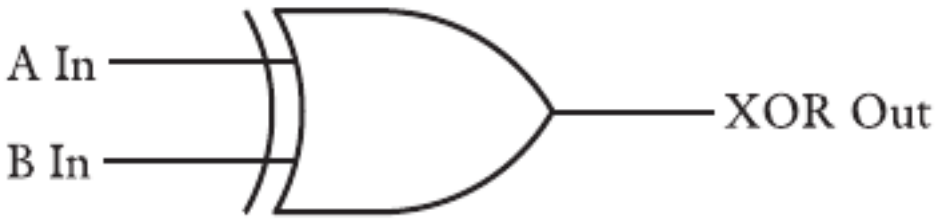
Обратите внимание: требуемое значение равно 1 лишь тогда, когда выходы вентиля ИЛИ *и* вентиля И-НЕ оба равны 1. Это наводит на мысль подать два этих выхода на вход вентиля И:



Обратите внимание, что вся эта схема по-прежнему имеет лишь два входа и один выход. Два входа подаются и на вентиль ИЛИ, и на вентиль И-НЕ. Выходы вентилей ИЛИ и И-НЕ поступают на вентиль И, и это даёт в точности требуемый результат:

Вход A	Вход B	Выход ИЛИ	Выход И-НЕ	Выход И (сумма)
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

У операции, выполняемой этой схемой, есть собственное название. Это вентиль *Исключающее ИЛИ*, или, короче, вентиль XOR. Одни произносят его «экс-ор», а другие называют по буквам: X O R. Он называется вентилем Исключающее ИЛИ, потому что выход равен 1, если вход A равен 1 *или* вход B равен 1, но не оба одновременно. Поэтому вместо изображения вентилей ИЛИ, И-НЕ и И, как выше, можно использовать принятое у инженеров-электриков условное обозначение вентиля XOR:



Он очень похож на вентиль ИЛИ, но со стороны входов у него есть ещё одна изогнутая линия. Поведение вентиля XOR показано здесь:

XOR	**0**	**1**
0	0	1
1	1	0

Вентиль XOR - последний логический вентиль, который я буду подробно описывать в этой книге. (В электротехнике иногда встречается ещё один вентиль. Его называют *вентилем совпадения*, или *вентилем эквивалентности*, поскольку выход равен 1 только тогда, когда два входа одинаковы. Выход вентиля совпадения противоположен выходу XOR, поэтому его условное обозначение совпадает с обозначением XOR, но на выходе добавлен маленький кружок.)

Подведём итог того, что нам уже известно. При сложении двух двоичных чисел получаются бит суммы и бит переноса:

Сумма	**0**	**1**
0	0	1
1	1	0

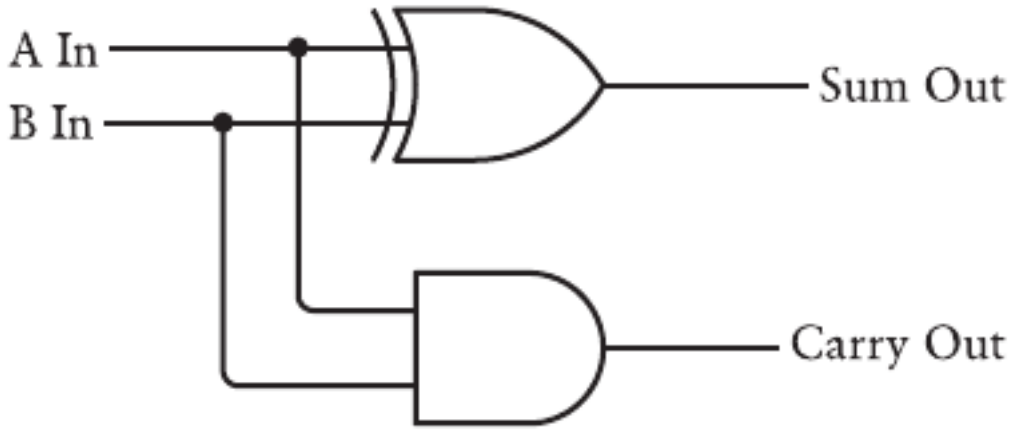
Перенос	**0**	**1**
0	0	0
1	0	1

Для получения этих результатов можно использовать два следующих логических вентиля:

XOR	**0**	**1**
0	0	1
1	1	0

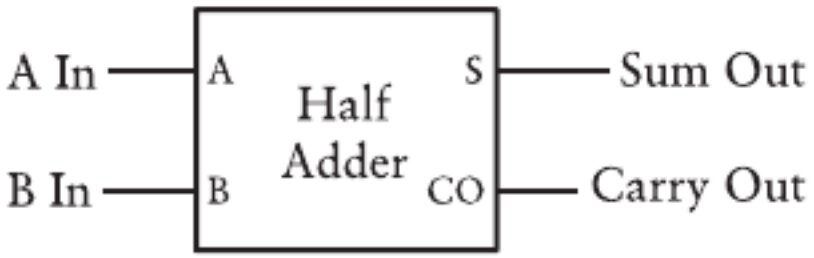
И	**0**	**1**
0	0	0
1	0	1

Сумму двух двоичных цифр даёт выход вентиля XOR, а бит переноса - выход вентиля И. Поэтому вентиль И и вентиль XOR можно объединить, чтобы складывать две двоичные цифры A и B:



Имейте в виду, что эта схема сложнее, чем кажется! В действительности вентиль XOR представляет сочетание вентилей ИЛИ, И-НЕ и И, а каждый из этих вентилей состоит из двух реле. Но понимать устройство становится легче, если скрыть множество подробностей. Этот процесс иногда называют *инкапсуляцией*: сложное нагромождение всякой всячины скрывается внутри более простой оболочки. В любой момент мы можем раскрыть оболочку, если захотим увидеть все детали, но делать это необязательно.

Вот ещё один пример инкапсуляции. Вместо того чтобы снова и снова рисовать вентиль И и вентиль XOR, всю схему можно представить одним блоком, называемым *полусумматором*:



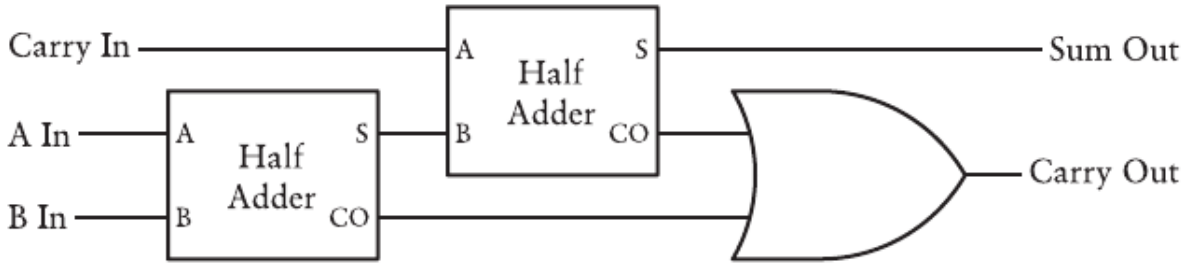
Метки S и CO означают Sum (сумма) и Carry Out (выход переноса). Иногда подобный блок называют *чёрным ящиком*. Определённое сочетание входов приводит к определённым выходам, но реализация скрыта. Однако, поскольку мы знаем, что происходит внутри полусумматора, правильнее называть его *прозрачным ящиком*.

Этот блок не случайно назван полусумматором. Он, конечно, складывает две двоичные цифры и выдаёт бит суммы и бит переноса. Но для двоичных чисел длиной больше 1 бита полусумматор годится только для сложения двух младших значащих битов. Он не способен прибавить возможный бит переноса от предыдущего однобитового сложения. Предположим, например, что мы складываем два следующих двоичных числа:

```
  1111
+ 1111
------
 11110
```

Полусумматор можно применить только для сложения крайнего правого столбца: 1 плюс 1 равно 0, 1 в уме. Во втором столбце справа из-за переноса на самом деле требуется сложить *три* двоичных числа. То же справедливо для всех последующих столбцов. Каждое последующее сложение двух двоичных цифр должно учитывать бит переноса из предыдущего столбца.

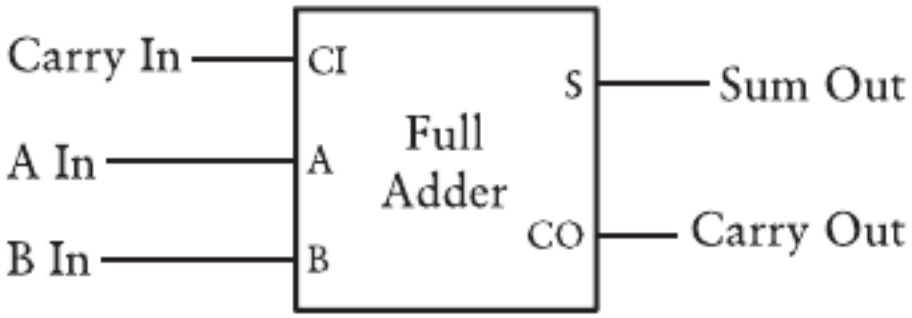
Чтобы сложить три двоичные цифры, нужны два полусумматора и вентиль ИЛИ, соединённые так:



Возможно, не вполне очевидно, почему эта схема работает. Начните со входов A и B первого полусумматора слева. На выходе получаются сумма и перенос. Эту сумму нужно сложить с переносом из предыдущего столбца, который называется Carry In, или входом переноса. Вход переноса и сумма из первого полусумматора подаются на входы второго полусумматора. Сумма второго полусумматора становится окончательной суммой. Два выхода переноса полусумматоров подаются на вентиль ИЛИ. Можно подумать, что здесь нужен ещё один полусумматор, и он,

несомненно, тоже сработал бы. Но если перебрать все варианты, обнаружится, что два выхода переноса полусумматоров *никогда* не равны 1 одновременно. Для их сложения достаточно вентиля ИЛИ, потому что при входах, которые никогда не бывают одновременно равны 1, вентиль ИЛИ действует так же, как XOR.

Вместо того чтобы снова и снова рисовать эту схему, можно просто назвать её *полным сумматором*:

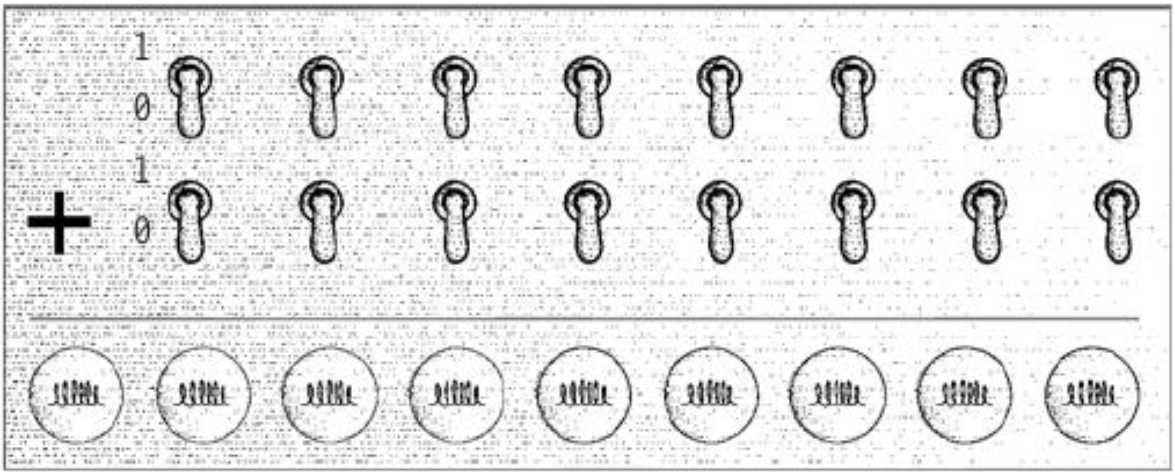


Следующая таблица обобщает все возможные сочетания входов полного сумматора и соответствующие выходы:

Вход A	Вход B	Вход переноса	Выход суммы	Выход 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

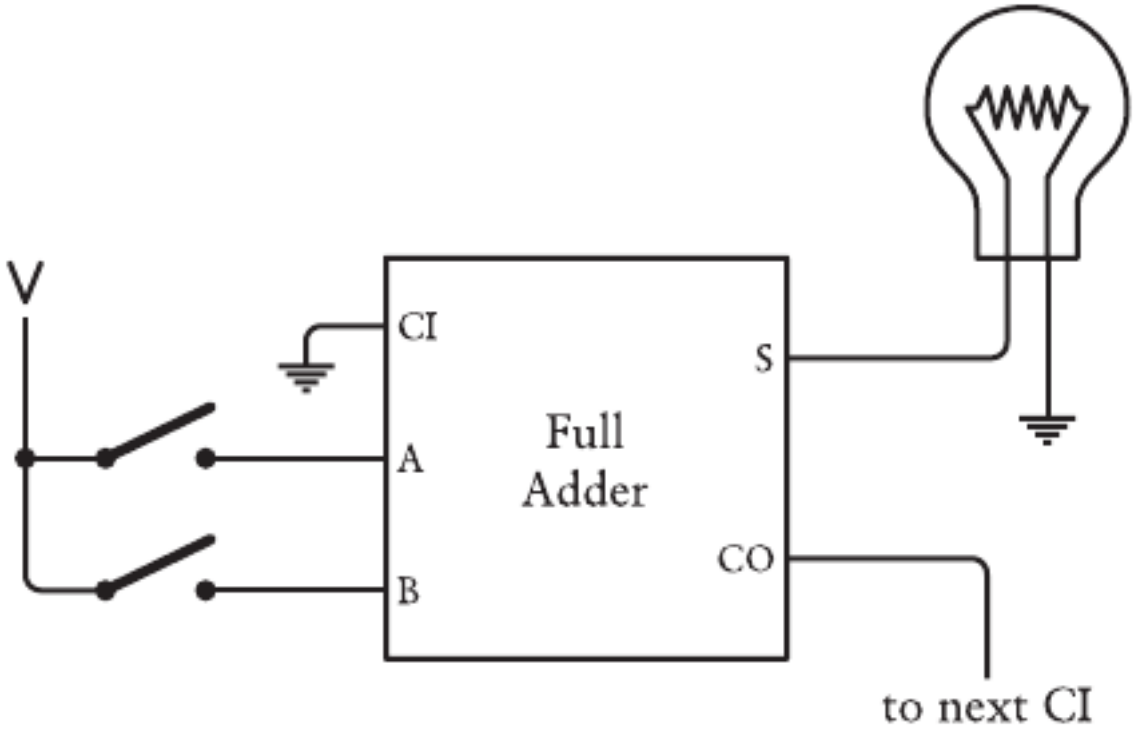
В начале главы я сказал, что для нашего двоичного сумматора потребуется 144 реле. Вот как я это подсчитал: каждому вентилю И, ИЛИ и И-НЕ нужны два реле. Следовательно, вентиль XOR включает шесть реле. Полусумматор состоит из вентиля XOR и вентиля И, поэтому ему нужно восемь реле. Каждый полный сумматор состоит из двух полусумматоров и вентиля ИЛИ, то есть из 18 реле. Для нашей 8-битовой суммирующей машины нужны восемь полных сумматоров. Итого 144 реле.

Вспомните нашу исходную панель управления с переключателями и лампочками:



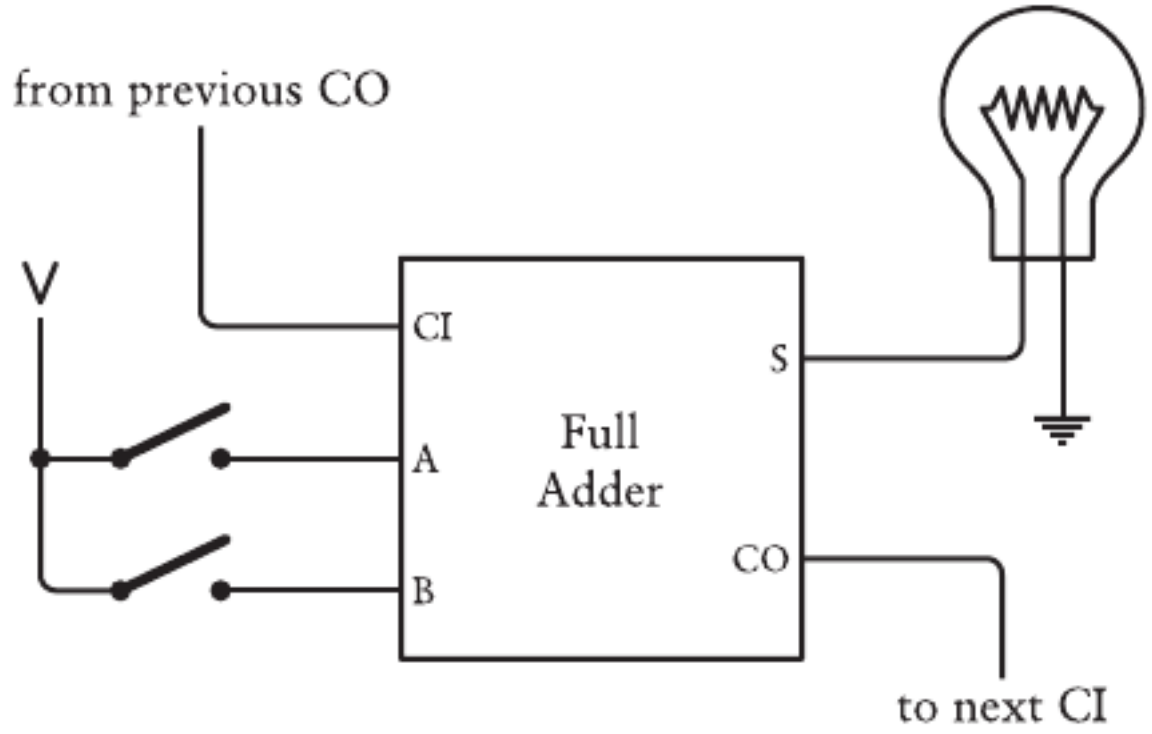
Теперь можно начать соединять эти переключатели и лампочки с восемью полными сумматорами.

Начнём с младшего значащего бита. Сначала соединим два крайних правых переключателя и крайнюю правую лампочку с полным сумматором:



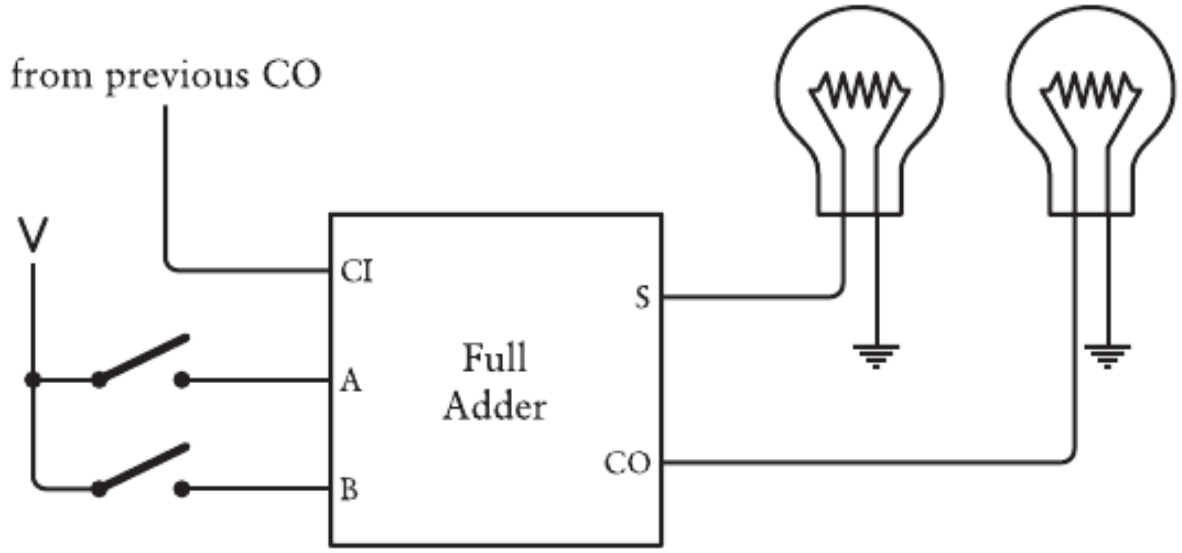
Когда вы начинаете складывать два двоичных числа, первый, крайний правый столбец складываемых цифр отличается от остальных. Отличие состоит в том, что каждый последующий столбец может включать бит переноса из предыдущего, а первый столбец переноса не имеет. Поэтому вход переноса полного сумматора соединён с землёй, что означает бит 0. Разумеется, сложение первой пары двоичных цифр может *породить* бит переноса. Этот выход переноса становится входом следующего столбца.

Для следующих двух цифр и следующей лампочки применяется полный сумматор, соединённый так:



Выход переноса первого полного сумматора подаётся на вход второго. Каждый последующий столбец цифр соединяется таким же образом. Выход переноса каждого столбца становится входом переноса следующего.

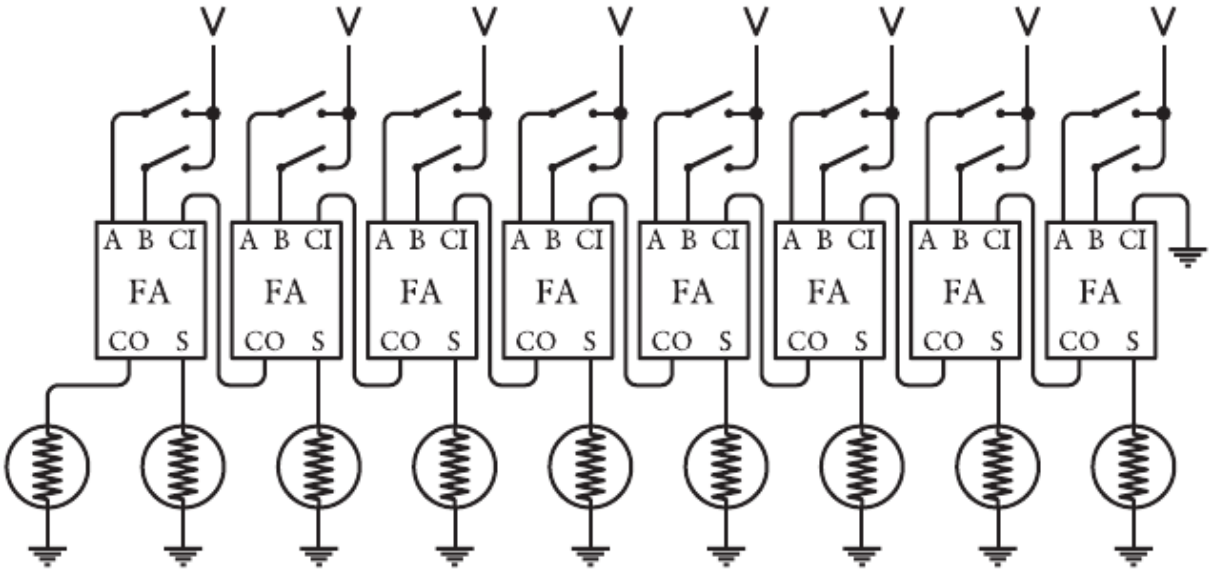
Наконец, восьмая и последняя пара переключателей (крайняя левая на панели управления) соединяется с последним полным сумматором:



Здесь окончательный выход переноса подключён к девятой лампочке.

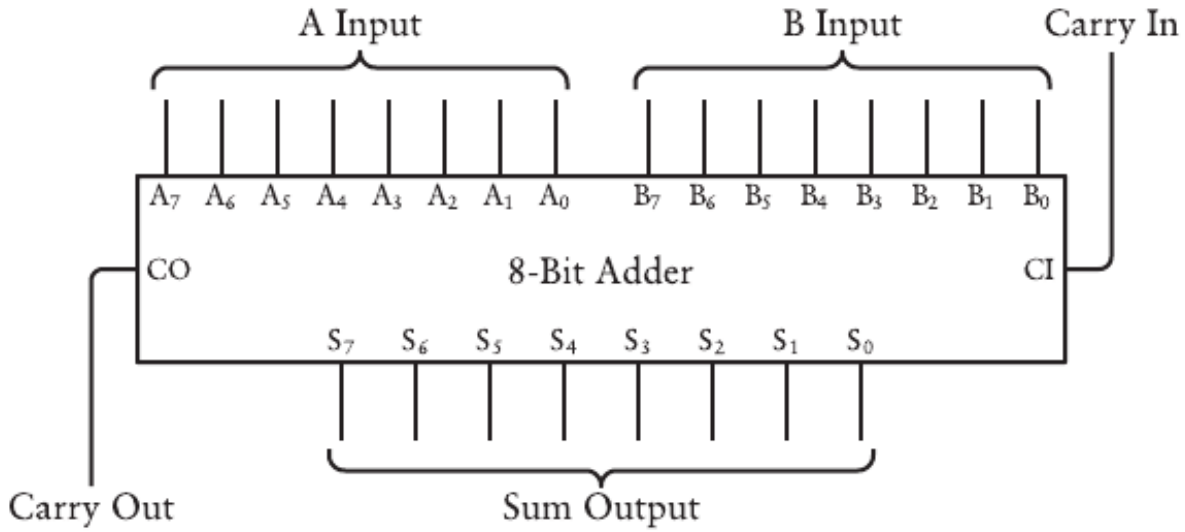
Готово.

Вот другой способ взглянуть на этот набор из восьми полных сумматоров, где каждый выход переноса служит входом переноса следующего:



Порядок полных сумматоров совпадает с порядком переключателей и лампочек на панели управления: младший значащий бит находится справа, а старший - слева, как обычно записывают числа. Обратите внимание, как каждый выход переноса огибает схему и становится входом переноса следующего по значимости бита. Первый вход переноса соединён с землёй (что означает бит 0), а окончательный выход переноса включает девятую лампочку.

Вот полный 8-битовый сумматор, инкапсулированный в одном блоке. Входы обозначены от A_0 до A_7 и от B_0 до B_7. Выходы обозначены от S_0 до S_7 (сумма):



Это распространённый способ обозначать отдельные биты многоразрядного числа. Биты A_0, B_0 и S_0 - *младшие значащие*, или крайние правые. Биты A_7, B_7 и S_7 - *старшие значащие*, или крайние левые. Например, вот как буквы с нижними индексами соответствуют двоичному числу 01101001:

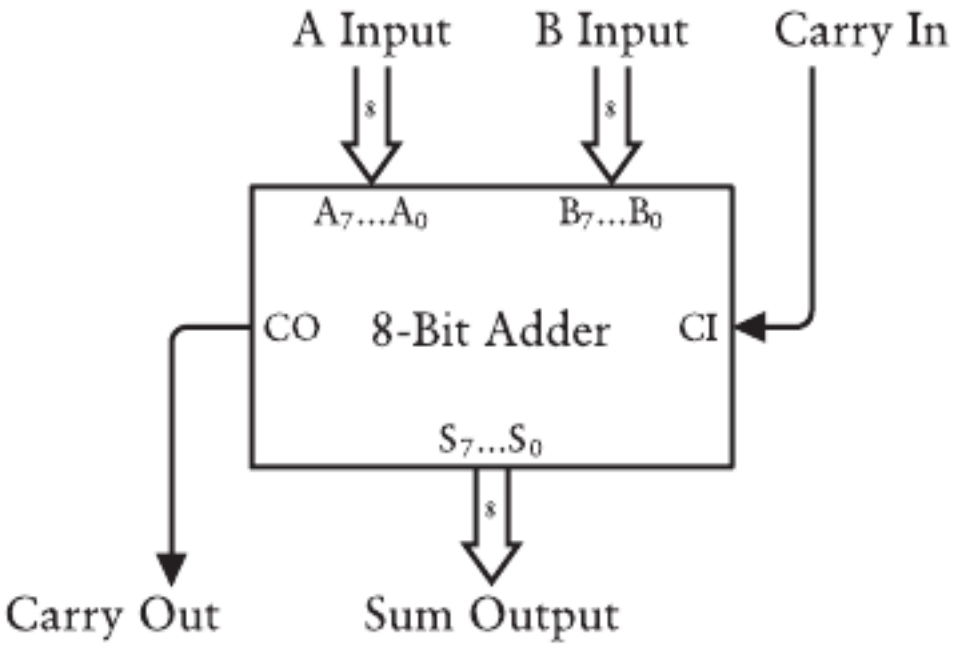
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	0	0	1

Нижние индексы начинаются с 0 и увеличиваются для более значимых цифр, потому что соответствуют показателям степеней 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

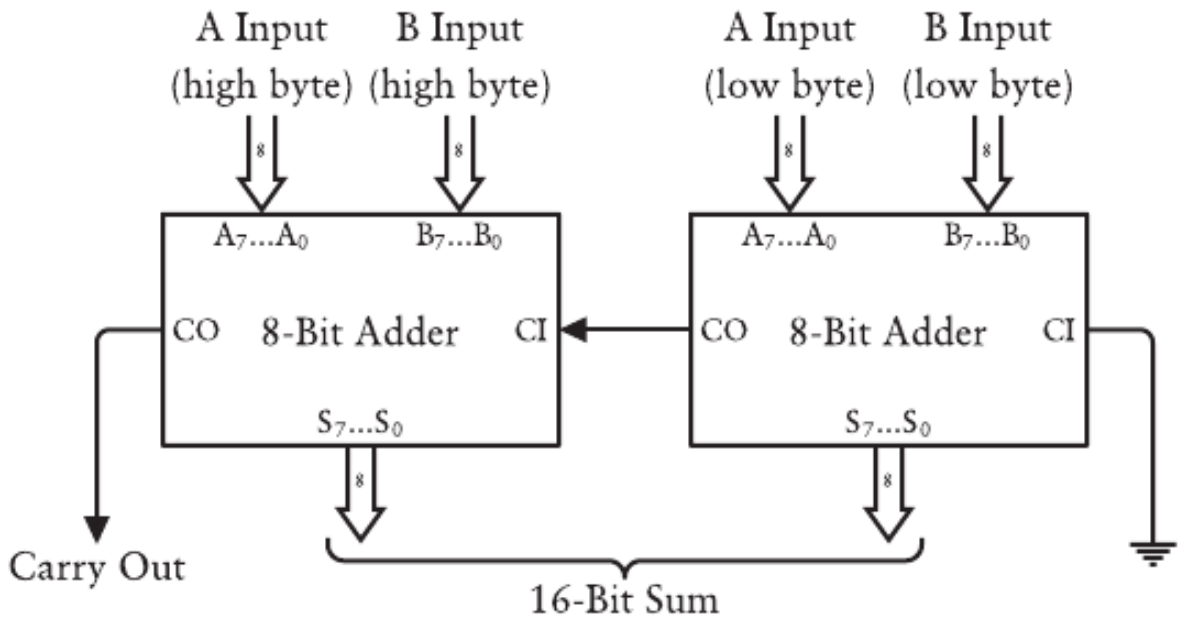
Если умножить каждую степень двойки на расположенную под ней цифру и сложить результаты, получится десятичный эквивалент 01101001: 64 + 32 + 8 + 1, то есть 105.

8-битовый сумматор можно изобразить и так:



Двойные стрелки содержат число 8, показывающее, что каждая из них представляет группу из восьми отдельных сигналов. Это 1-байтовые *тракты данных*. Они также обозначены A_7...A_0, B_7...B_0 и S_7...S_0, что указывает на 8-битовые числа.

Построив один 8-битовый сумматор, можно построить второй. После этого их легко соединить каскадом для сложения двух 16-битовых чисел:



Каждое из двух 16-битовых входных значений разделяется на два байта, называемые *младшим байтом* и *старшим байтом*. Выход переноса сумматора справа соединён со входом переноса сумматора слева. На вход сумматора слева подаются восемь старших цифр двух складываемых чисел, а на его выходе получаются восемь старших цифр результата.

И теперь вы можете спросить: «Неужели компьютеры *действительно* складывают числа именно так?»

Прекрасный вопрос!

Глава 15. Неужели это по-настоящему?

В предыдущей главе вы увидели, как соединить реле, чтобы получить 1-битовый сумматор, а затем объединить восемь таких сумматоров для сложения двух байтов. Вы даже увидели, как эти 8-битовые сумматоры можно соединять каскадом для сложения ещё больших чисел, и, возможно, задумались: неужели компьютеры действительно складывают числа именно так?

Что ж, и да и нет. Одно существенное отличие состоит в том, что современные компьютеры больше не делают из реле. Но когда-то их из реле делали.

В ноябре 1937 года исследователь Bell Labs Джордж Стибиц (1904-1995) принёс домой пару реле, применявшихся в телефонных коммутационных схемах. На кухонном столе он соединил эти реле с батареями, двумя лампочками и двумя переключателями, сделанными из полосок металла, вырезанных из консервных банок. Получился 1-битовый сумматор, точно такой же, как в предыдущей главе. Позднее Стибиц назвал его «Model K», потому что собрал устройство на кухонном столе. Сумматор Model K был тем, что позже назовут «доказательством концепции»: он продемонстрировал, что реле способны выполнять арифметические действия. Bell Labs одобрила продолжение работы, и к 1940 году заработал Complex Number Computer - компьютер комплексных чисел. Он состоял немногим более чем из 400 реле и предназначался для умножения комплексных чисел, то есть чисел, имеющих и действительную, и мнимую часть. (Мнимые числа представляют собой квадратные корни из отрицательных чисел и полезны в научных и инженерных задачах.) Для умножения двух комплексных чисел нужны четыре отдельных умножения и два сложения. Complex Number Computer мог обрабатывать комплексные числа, действительная и мнимая части которых содержали до восьми десятичных цифр. Такое умножение занимало около минуты.

Это был не первый компьютер на реле. Хронологически первым стал компьютер Конрада Цузе (1910-1995), который в 1935 году, будучи студентом инженерного факультета, начал строить машину в квартире родителей в Берлине. Его первая машина, Z1, не использовала реле, а полностью механически имитировала их работу. В машине Z2 реле уже применялись, а программировать её можно было с помощью отверстий, пробитых в старой 35-миллиметровой киноплёнке.

Тем временем, примерно в 1937 году, аспиранту Гарварда Говарду Эйкену (1900-1973) понадобился способ выполнять множество повторяющихся вычислений. Это привело к сотрудничеству Гарварда и IBM, результатом которого стал Automated Sequence Controlled Calculator (ASCC), позднее известный как Harvard Mark I и законченный в 1943 году. Во время работы щёлканье реле этой машины создавало очень характерный звук, который одному человеку напомнил «комнату, полную вяжущих дам». Mark II был крупнейшей релейной машиной и содержал 13 000 реле. Гарвардская вычислительная лаборатория под руководством Эйкена проводила первые учебные курсы по информатике.

Эти релейные компьютеры, называемые также *электромеханическими*, потому что они объединяли электрические и механические устройства, стали первыми работавшими цифровыми компьютерами.

Слово *digital* («цифровой») для описания этих компьютеров придумал в 1942 году Джордж Стибиц, чтобы отличить их от *аналоговых* компьютеров, уже несколько десятилетий находившихся в широком употреблении.

Одним из великих аналоговых компьютеров был дифференциальный анализатор, построенный профессором Массачусетского технологического института Вэниваром Бушем (1890-1974) и его студентами в 1927-1932 годах. Эта машина использовала вращающиеся диски, валы и шестерни для решения дифференциальных уравнений - уравнений, связанных с математическим анализом.

Решением дифференциального уравнения является не число, а функция, и дифференциальный анализатор печатал график этой функции на бумаге.

Историю аналоговых компьютеров можно проследить ещё дальше, до машины для предсказания приливов, спроектированной физиком Уильямом Томсоном (1824-1907), позднее известным как лорд Кельвин. В 1860-х годах Томсон придумал способ анализа подъёма и спада приливов с разложением их закономерностей в ряд синусоидальных кривых разных частот и амплитуд. По словам Томсона, назначение его машины состояло в том, чтобы «заменить мозг латунью в огромном механическом труде вычисления элементарных составляющих полного подъёма и спада прилива». Иными словами, с помощью колёс, шестерён и блоков она складывала составляющие синусоидальные кривые и печатала результат на рулоне бумаги, показывая будущие подъёмы и спады приливов.

И дифференциальный анализатор, и машина для предсказания приливов умели печатать графики, но интересно, что они делали это, не вычисляя чисел, определяющих график! Это характерная черта аналоговых компьютеров.

Не позднее 1879 года Уильям Томсон уже понимал разницу между аналоговыми и цифровыми компьютерами, хотя и пользовался другими терминами. Устройства, подобные его предсказателю приливов, он называл «вычислительными машинами непрерывного действия», отличая их от «чисто арифметических» машин, таких как «грандиозные, но лишь частично осуществлённые замыслы вычислительных машин Бэббиджа».

Томсон имеет в виду знаменитую работу английского математика Чарльза Бэббиджа (1791-1871). В ретроспективе Бэббидж выглядит исторической аномалией: он пытался построить цифровой компьютер задолго до того, как даже аналоговые компьютеры стали обычным явлением!

Во времена Бэббиджа (и ещё долго после них) *компьютером* называли человека, который за плату производил вычисления. Таблицы логарифмов часто применялись для упрощения умножения, а таблицы тригонометрических функций были необходимы в морской навигации и для других целей. Если вы хотели опубликовать новый набор математических таблиц, то нанимали группу вычислителей, поручали им работу, а затем сводили результаты. Разумеется, ошибки могли проникнуть на любом этапе процесса - от исходных вычислений до набора печатной формы для окончательных страниц.

Чарльз Бэббидж был чрезвычайно скрупулёзным человеком и очень огорчался, встречая ошибки в математических таблицах. Примерно в 1820 году у него возникла мысль построить машину, которая составляла бы эти таблицы автоматически, вплоть до подготовки печатной формы.

Первая машина Бэббиджа называлась разностной машиной, поскольку предназначалась для конкретной задачи, связанной с созданием математических таблиц. Было хорошо известно, что для построения таблицы логарифмов необязательно вычислять логарифм каждого отдельного значения. Вместо этого логарифмы можно было рассчитать для избранных значений, а промежуточные числа получить интерполяцией посредством относительно простых вычислений так называемых *разностей*.

Бэббидж спроектировал разностную машину для вычисления этих разностей. Десятичные цифры в ней представлялись шестернями, и машина должна была уметь складывать и вычитать. Но, несмотря на некоторое финансирование британского правительства, её так и не закончили, и в 1833 году Бэббидж отказался от разностной машины.

К тому времени у Бэббиджа появилась ещё более удачная идея машины, которую он назвал аналитической. Её неоднократное проектирование и перепроектирование (при этом несколько небольших моделей и деталей действительно были построены) с перерывами занимало Бэббиджа до самой смерти. Аналитическая машина - самое близкое к цифровому компьютеру из всего, что мог предложить XIX век. В проекте Бэббиджа у неё имелись *склад* (сопоставимый с современным понятием памяти) и *мельница*, выполнявшая арифметические операции. Умножение можно было производить повторным сложением, а деление - повторным вычитанием.

Самая интригующая черта аналитической машины состояла в возможности программировать её перфорированными картами. Эту идею Бэббидж почерпнул у новаторских автоматических ткацких станков, разработанных Жозефом Мари Жаккаром (1752-1834). Жаккардовый станок (около 1801 года) использовал картонные листы с пробитыми отверстиями для управления узорами, вытканными на шёлке. Собственным выдающимся достижением Жаккара стал автопортрет из чёрного и белого шёлка, для которого потребовалось около 10 000 карт.

Бэббидж не оставил нам всестороннего и связного описания того, что пытался сделать с аналитической машиной. Гораздо красноречивее он был, когда писал математическое обоснование чудес или сочинял обличительную речь против уличных музыкантов.

Восполнить этот пробел Бэббиджа пришлось Августе Аде Байрон, графине Лавлейс (1815-1852). Она была единственной законной дочерью поэта лорда Байрона, но мать направила её к математике, чтобы противодействовать тому, что считалось опасным поэтическим темпераментом, который Ада могла унаследовать от отца. Леди Лавлейс училась у логика Огастеса де Моргана (который уже появлялся в главах 6 и 8 этой книги) и была очарована машиной Бэббиджа.

Hulton Archive/Stringer/Getty Images

Когда представилась возможность перевести итальянскую статью об аналитической машине, Ада Лавлейс взялась за эту работу. Её перевод опубликовали в 1843 году, но она добавила ряд примечаний, из-за которых статья стала втрое длиннее первоначальной. Одно из примечаний содержало пример набора инструкций для машины Бэббиджа, и поэтому Лавлейс была не совсем первым программистом (им являлся сам Бэббидж), но первым человеком, *опубликовавшим* компьютерную программу.

Те из нас, кто впоследствии публиковал учебные компьютерные программы в журналах и книгах, могут считать себя детьми Ады.

Возможно, именно Аде Лавлейс мы обязаны самым поэтичным описанием машины Бэббиджа: «Можно сказать, что аналитическая машина ткёт *алгебраические узоры* точно так же, как жаккардовый станок ткёт цветы и листья».

У Лавлейс также был не по годам дальновидный взгляд на вычислительную технику, выходивший за рамки простого расчёта чисел. Всё, что можно выразить числами, могло стать предметом работы аналитической машины:

> Если предположить, например, что фундаментальные отношения звуков определённой высоты в науке о гармонии и музыкальной композиции поддаются такому выражению и приспособлению, машина могла бы сочинять сложные и научные музыкальные произведения любой степени сложности и протяжённости.

Учитывая, что Бэббидж и Сэмюэл Морзе были почти точными современниками и что Бэббидж знал также работы Джорджа Буля, досадно, что он не установил важнейшую связь между телеграфными реле и математической логикой. Только в 1930-х годах изобретательные инженеры начали строить компьютеры из реле. Harvard Mark I стал первым компьютером, печатавшим математические таблицы, и наконец осуществил мечту Бэббиджа более чем через сто лет.

От первых цифровых компьютеров 1930-х годов до наших дней всю историю вычислительной техники можно свести к трём тенденциям: *меньше, быстрее, дешевле*.

Реле - не лучшие устройства для построения компьютера. Поскольку реле являются механическими и работают, сгибая металлические детали, после долгой интенсивной работы они могут сломаться. Реле способно отказать и из-за соринки или клочка бумаги, застрявшего между контактами. В одном знаменитом случае в 1947 году из реле компьютера Harvard Mark II извлекли мотылька. Грейс Мюррей Хоппер (1906-1992), которая присоединилась к коллективу Эйкена в 1944 году и впоследствии получила большую известность в области языков программирования, приклеила мотылька к журналу работы компьютера с подписью: «Первый реальный случай обнаружения жучка».

Возможной заменой реле была электронная лампа (в Великобритании её называли *valve*, «клапан»), разработанная Джоном Амброзом Флемингом (1849-1945) и Ли де Форестом (1873-1961) в связи с радио. К 1940-м годам электронные лампы давно использовались для усиления телефонных сигналов, и почти в каждом доме стоял консольный радиоприёмник, полный светящихся ламп, которые усиливали радиосигналы, делая их слышимыми. Электронные лампы, подобно реле, также можно соединять в вентили И, ИЛИ, И-НЕ и ИЛИ-НЕ.

Неважно, построены ли логические вентили из реле или электронных ламп. Из логических вентилей всегда можно собирать сумматоры и другие сложные компоненты.

Однако у электронных ламп были собственные недостатки. Они дорого стоили, потребляли много электричества и выделяли много тепла. Ещё более серьёзным недостатком было то, что со временем лампы перегорали. Это воспринимали как неизбежный факт жизни. Владельцы ламповых радиоприёмников привыкли периодически менять лампы. Телефонную систему проектировали с большой избыточностью, поэтому потеря одной лампы время от времени не имела большого значения. (Всё равно никто не ждёт от телефонной системы безупречной работы.) Но если лампа перегорает в компьютере, это может быть обнаружено не сразу. Кроме того, компьютер использует *настолько много* электронных ламп, что статистически они могут перегорать каждые несколько минут.

Главным преимуществом электронных ламп перед реле была скорость. В самом лучшем случае реле переключается примерно за тысячную долю секунды, то есть за 1 миллисекунду. Лампа может переключиться примерно за миллионную долю секунды - одну *микросекунду*. Интересно, что скорость не была главным соображением на ранних этапах разработки компьютеров, потому что общая скорость вычислений зависела от того, как быстро машина считывала программу с бумажной или киноплёнки. Пока компьютеры строились таким образом, не имело значения, насколько лампы быстрее реле.

С начала 1940-х годов электронные лампы начали вытеснять реле в новых компьютерах. К 1945 году переход завершился. Если релейные машины называли электромеханическими компьютерами, то электронные лампы стали основой первых *электронных* компьютеров.

В Школе электротехники Мура (Пенсильванский университет) Дж. Преспер Эккерт (1919-1995) и Джон Мокли (1907-1980) спроектировали ENIAC (Electronic Numerical Integrator and Computer). Он использовал 18 000 электронных ламп и был закончен в конце 1945 года. По одному лишь весу (около 30 тонн) ENIAC стал самым большим компьютером из когда-либо построенных (и, вероятно, останется таким навсегда). Однако попытке Эккерта и Мокли запатентовать компьютер помешало конкурирующее притязание Джона В. Атанасова (1903-1995), который ранее спроектировал электронный компьютер, так и не заработавший вполне правильно.

ENIAC привлёк внимание математика Джона фон Неймана (1903-1957). Родившийся в Венгрии фон Нейман (его фамилия произносится «ной ман») с 1930 года жил в Соединённых Штатах. Яркий человек, славившийся умением выполнять сложные арифметические вычисления в уме, фон Нейман был профессором математики в Институте перспективных исследований в Принстоне и занимался всем - от квантовой механики до применения теории игр в экономике.

Фон Нейман помогал проектировать преемника ENIAC - EDVAC (Electronic Discrete Variable Automatic Computer). В частности, в статье 1946 года «Предварительное обсуждение логической конструкции электронного вычислительного устройства», написанной совместно с Артуром У. Бёрксом и Германом Х. Голдстайном, он описал несколько особенностей компьютера, благодаря которым EDVAC стал значительным шагом вперёд по сравнению с ENIAC. ENIAC использовал десятичные числа, но проектировщики EDVAC считали, что внутри компьютер должен применять двоичные числа. Кроме того, у компьютера должно быть как можно больше памяти, и эта память должна использоваться для хранения как программного кода, так и данных во время выполнения программы. (В ENIAC дело обстояло иначе. Программирование ENIAC сводилось к установке переключателей и подключению кабелей.) Этот проект стал известен как *концепция хранимой программы*. Проектные решения оказались настолько важным эволюционным шагом, что сегодня мы говорим об *архитектуре фон Неймана* в компьютерах.

В 1948 году Eckert-Mauchly Computer Corporation (позднее вошедшая в Remington Rand) начала работу над первым коммерчески доступным компьютером - Universal Automatic Computer, или UNIVAC. Его закончили в 1951 году, а первый экземпляр передали Бюро переписи населения. В

1952 году UNIVAC впервые появился в прайм-тайм общенационального телевидения CBS, когда его использовали для предсказания результатов президентских выборов. Ведущий Уолтер Кронкайт назвал его «электронным мозгом». В том же 1952 году IBM анонсировала свою первую коммерческую компьютерную систему 701.

Так началась долгая история корпоративных и государственных вычислений. Какой бы интересной ни была эта история, мы последуем по другой исторической дорожке - той, что уменьшила стоимость и размеры компьютеров и привела их в дома, а началась с почти не замеченного прорыва в электронике в 1947 году.

Bell Telephone Laboratories появилась, когда American Telephone and Telegraph официально отделила свои подразделения научных и технических исследований от остального бизнеса, создав дочернюю компанию 1 января 1925 года. Главной задачей Bell Labs была разработка технологий для совершенствования телефонной системы. К счастью, формулировка была достаточно расплывчатой, чтобы охватить всевозможные направления, но одной очевидной постоянной целью телефонной системы было усиление передаваемых по проводам голосовых сигналов без искажений.

Совершенствованию электронных ламп посвятили значительные исследовательские и инженерные усилия, но 16 декабря 1947 года два физика из Bell Labs, Джон Бардин (1908-1991) и Уолтер Браттейн (1902-1987), собрали усилитель иного типа. Новый усилитель был сделан из пластинки германия - элемента, известного как *полупроводник*, - и полоски золотой фольги. Через неделю они продемонстрировали его своему начальнику Уильяму Шокли (1910-1989). Это был первый *транзистор* - устройство, которое некоторые называют важнейшим изобретением XX века.

Транзистор возник не на пустом месте. Восемью годами ранее, 29 декабря 1939 года, Шокли записал в блокноте: «Сегодня мне пришло в голову, что усилитель, использующий полупроводники вместо вакуума, в принципе возможен». И после демонстрации первого транзистора потребовалось ещё много лет на его совершенствование. Только в 1956 году Шокли, Бардину и Браттейну присудили Нобелевскую премию по физике «за исследования полупроводников и открытие транзисторного эффекта».

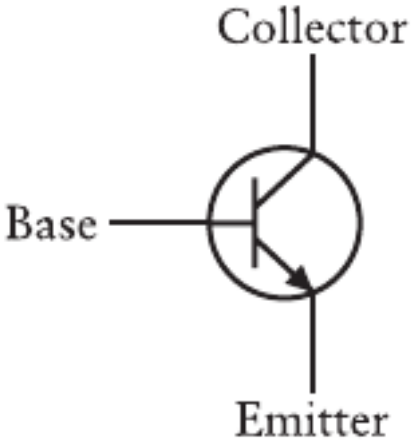
Ранее в этой книге я рассказывал о проводниках и изоляторах. Проводники называются так потому, что хорошо способствуют прохождению электричества. Медь, серебро и золото - лучшие проводники, и не случайно все три находятся в одном столбце периодической таблицы элементов.

Германий и кремний (а также некоторые соединения) называются *полупроводниками* не потому, что проводят ток вдвое хуже проводников, а потому, что их проводимостью можно управлять различными способами. У полупроводников четыре электрона во внешней оболочке, то есть половина максимального числа, которое способна вместить внешняя оболочка. В чистом полупроводнике атомы образуют друг с другом очень устойчивые связи и кристаллическую структуру, подобную алмазу. Такие материалы плохо проводят ток.

Но полупроводники можно *легировать*, то есть соединять с определёнными примесями. Один тип примеси добавляет избыточные электроны сверх необходимых для связи атомов. Такие материалы называются *полупроводниками N-типа* (N от *negative*, «отрицательный»). Другой тип примеси даёт *полупроводник P-типа*.

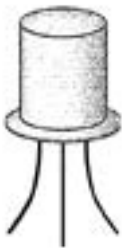
Усилитель из полупроводников можно создать, поместив полупроводник P-типа между двумя полупроводниками N-типа. Такая конструкция называется NPN-транзистором, а три её части - *коллектором*, *базой* и *эмиттером*.

Вот принципиальная схема NPN-транзистора:



Небольшое напряжение на базе может управлять гораздо большим напряжением, проходящим от коллектора к эмиттеру. Если напряжения на базе нет, транзистор фактически выключен.

Обычно транзисторы заключают в маленькие металлические цилиндры диаметром около четверти дюйма, из которых выступают три провода:



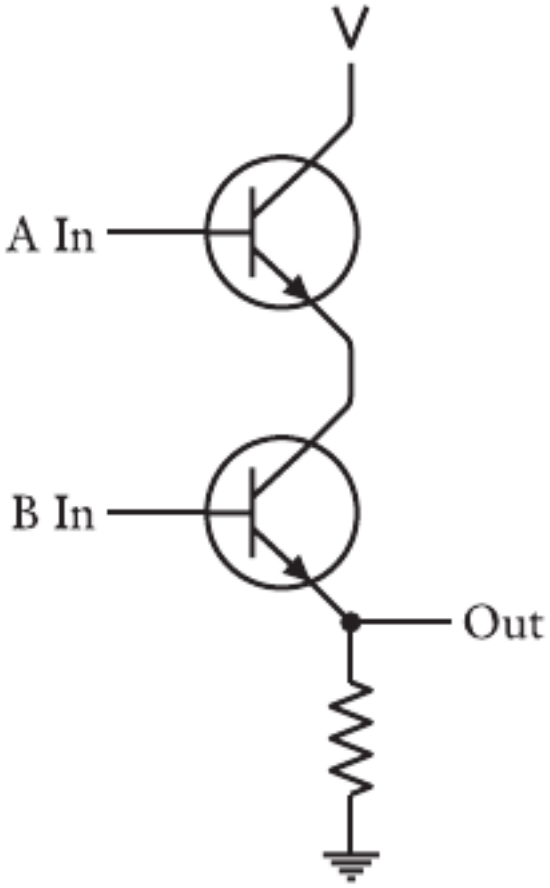
Транзистор положил начало *твердотельной* электронике: транзисторам не нужен вакуум, и они строятся из твёрдых материалов - полупроводников, чаще всего кремния. Помимо значительно меньших размеров по сравнению с электронными лампами, транзисторы потребляют гораздо меньше энергии, выделяют гораздо меньше тепла и служат дольше. Носить в кармане ламповый радиоприёмник было немыслимо. Но транзисторный приёмник мог питаться от небольшой батареи и, в отличие от лампового, не нагревался. Для нескольких счастливчиков, открывавших подарки рождественским утром 1954 года, стало возможным носить транзисторный радиоприёмник в кармане. В первых карманных радиоприёмниках использовались транзисторы Texas Instruments - важной компании полупроводниковой революции.

Однако самым первым коммерческим применением транзистора стали слуховые аппараты. В память о наследии Александра Грейама Белла и его многолетней работе с глухими людьми AT&T разрешила производителям слуховых аппаратов использовать транзисторную технологию без уплаты лицензионных отчислений.

Первый транзисторный телевизор появился в 1960 году, а сегодня ламповые приборы почти исчезли. (Но не полностью. Некоторые аудиофилы и электрогитаристы по-прежнему предпочитают звучание ламповых усилителей их транзисторным аналогам.)

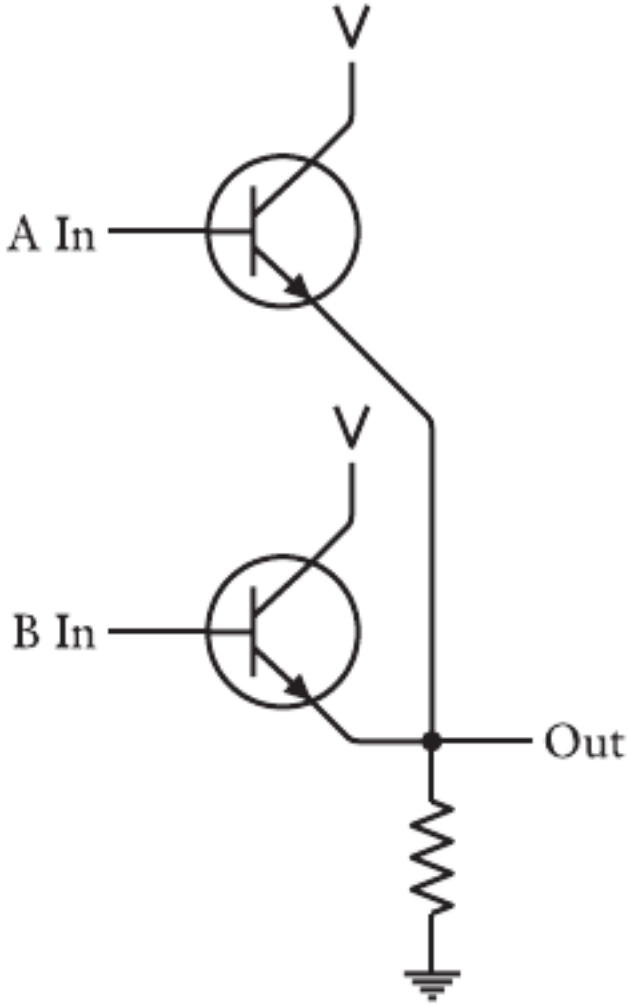
В 1956 году Шокли ушёл из Bell Labs, чтобы основать Shockley Semiconductor Laboratories. Он перебрался в Пало-Альто, штат Калифорния, где вырос. Его компания стала первым предприятием такого рода в этом районе. Со временем там появились другие полупроводниковые и компьютерные компании, и теперь местность к югу от Сан-Франциско неофициально называется Кремниевой долиной.

Первоначально электронные лампы разрабатывались для усиления, но могли служить и переключателями в логических вентилях. То же справедливо для транзистора. Вот вентиль И на транзисторах, устроенный почти так же, как его релейная версия:



Только когда и на входе A, и на входе B присутствует напряжение, оба транзистора проводят ток, и на выходе появляется напряжение. Резистор предотвращает короткое замыкание, когда это происходит.

Соединение двух транзисторов, показанное здесь, создаёт вентиль ИЛИ. Коллекторы обоих транзисторов подключены к источнику напряжения, а эмиттеры соединены:



Всё, что мы узнали о построении логических вентилей и других компонентов из реле, справедливо и для транзисторов. Реле, лампы и транзисторы первоначально разрабатывались главным образом для усиления, но их можно сходным образом соединять в логические вентили, из которых строятся компьютеры. Первые транзисторные компьютеры появились в 1956 году, а через несколько лет от электронных ламп при проектировании новых компьютеров отказались.

Транзисторы, несомненно, делают компьютеры надёжнее, меньше и экономичнее, но не обязательно упрощают их сборку. Транзистор позволяет разместить больше логических вентилей в меньшем пространстве, однако всё равно приходится заботиться обо всех *межсоединениях* этих компонентов. Соединять транзисторы проводами в логические вентили ничуть не легче, чем соединять реле и электронные лампы.

Однако, как вы уже обнаружили, определённые сочетания транзисторов постоянно повторяются. Пары транзисторов почти всегда соединяют в вентили. Вентили часто соединяют в сумматоры либо в декодеры и кодеры, подобные тем, что вы видели в конце главы 10. В главе 17 вы вскоре познакомитесь с чрезвычайно важной конфигурацией логических вентилей, называемой *триггером*, который способен хранить биты, и со *счётчиком*, считающим двоичными числами. Собирать такие схемы было бы гораздо легче, если бы транзисторы заранее соединялись в распространённые конфигурации.

По-видимому, первым эту идею высказал британский физик Джеффри Даммер (1909-2002) в выступлении в мае 1952 года. «Я хотел бы заглянуть в будущее, - сказал он. -

> С появлением транзистора и развитием полупроводников вообще теперь, кажется, можно представить электронное оборудование в виде цельного блока без соединительных проводов. Блок может состоять из слоёв изолирующих, проводящих, выпрямляющих и усиливающих материалов, а электрические функции непосредственно соединяться посредством вырезания участков различных слоёв».

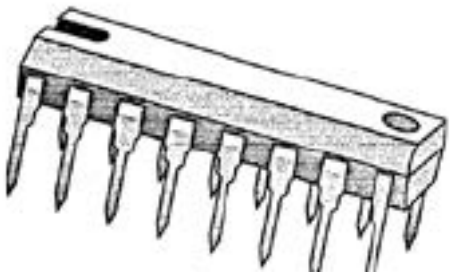
Однако работающего изделия пришлось ждать ещё несколько лет.

Ничего не зная о предсказании Даммера, в июле 1958 года Джек Килби (1923-2005) из Texas Instruments понял, что несколько транзисторов, а также резисторы и другие электрические компоненты можно изготовить из единого куска кремния. Через полгода, в январе 1959 года, почти та же идея пришла Роберту Нойсу (1927-1990). Нойс прежде работал в Shockley Semiconductor Laboratories, но в 1957 году он и ещё семь учёных ушли и основали Fairchild Semiconductor Corporation.

В истории техники одновременные изобретения встречаются чаще, чем можно подумать. Хотя Килби создал устройство на полгода раньше Нойса, а Texas Instruments подала заявку на патент раньше Fairchild, первым патент получил Нойс. Последовали судебные баталии, и лишь через десятилетие их наконец разрешили ко всеобщему удовлетворению. Хотя Килби и Нойс никогда не работали вместе, теперь их считают соавторами изобретения *интегральной схемы*, или *ИС*, обычно называемой *микросхемой*, либо *чипом*.

Интегральные схемы производятся сложным процессом, в котором тонкие пластины кремния послойно и точно легируют и вытравливают в различных областях, формируя микроскопические компоненты. Хотя разработка новой интегральной схемы обходится дорого, она выигрывает от массового производства: чем больше экземпляров выпускается, тем дешевле они становятся.

Сам кремниевый кристалл тонок и хрупок, поэтому его нужно надёжно упаковать, чтобы защитить и обеспечить способ соединения находящихся на нём компонентов с другими микросхемами. Ранние интегральные схемы заключали в корпуса нескольких типов, но самым распространённым стал прямоугольный пластмассовый *корпус с двухрядным расположением выводов* (dual inline package, или DIP) с 14, 16 или даже 40 выводами, выступающими по сторонам:



Это 16-выводная микросхема. Если держать её так, чтобы небольшая выемка находилась слева (как на рисунке), выводы нумеруются от 1 до 16: начиная с нижнего левого, далее вокруг правой стороны и заканчивая выводом 16 вверху слева. Расстояние между выводами с каждой стороны равно точно 1/10 дюйма.

На протяжении 1960-х годов космическая программа и гонка вооружений подпитывали ранний рынок интегральных схем. В гражданской сфере первым коммерческим изделием с интегральной схемой стал слуховой аппарат, продававшийся Zenith в 1964 году. В 1971 году Texas Instruments начала продавать первый карманный калькулятор, а Pulsar - первые цифровые часы. (Разумеется,

ИС в цифровых часах упакована совсем не так, как только что показанный пример.) Затем появилось множество других изделий, в конструкцию которых входили интегральные схемы.

В 1965 году Гордон Э. Мур (тогда работавший в Fairchild, а позднее ставший соучредителем Intel Corporation) заметил, что технология совершенствуется так, что с 1959 года число транзисторов, помещающихся на одном кристалле, ежегодно удваивается. Он предсказал продолжение этой тенденции. В действительности рост оказался несколько медленнее, поэтому закон Мура (как его впоследствии назвали) изменили: теперь он предсказывал удвоение числа транзисторов на кристалле каждые 18 месяцев. Это всё равно поразительно высокая скорость прогресса, объясняющая, почему домашние компьютеры всегда словно устаревают всего за несколько лет. Похоже, закон Мура перестал действовать во втором десятилетии XXI века, но реальность всё ещё близка к предсказанию.

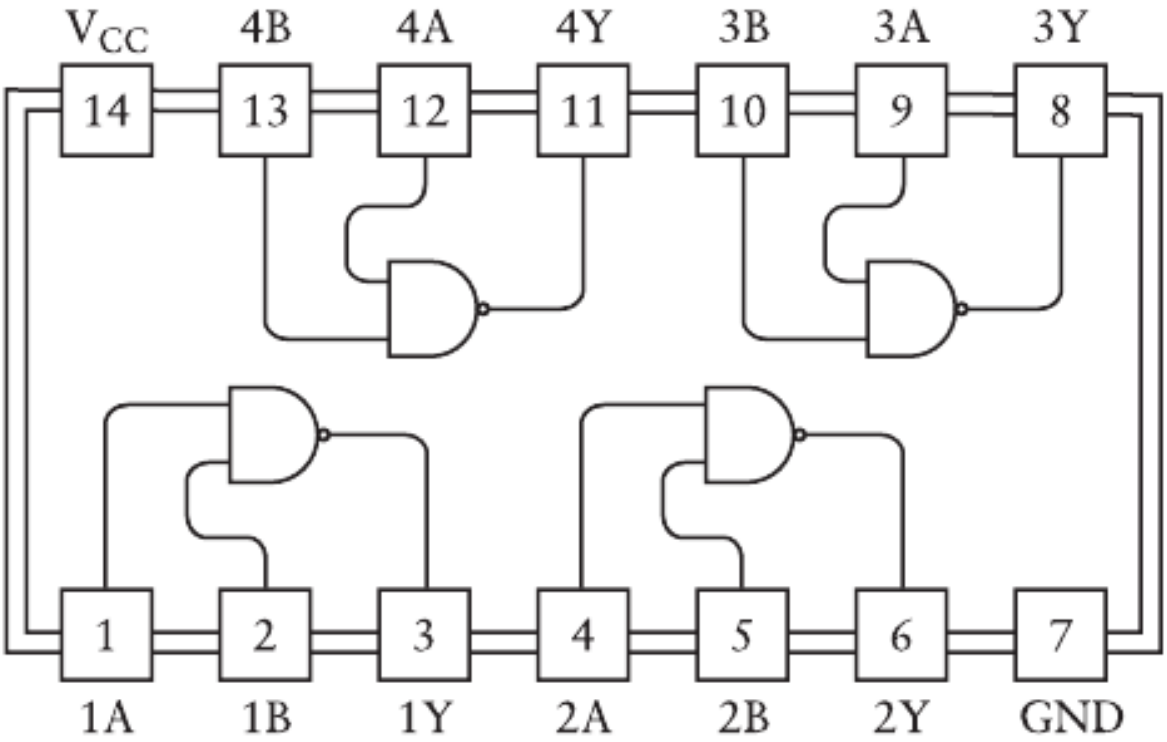
Для изготовления компонентов интегральных схем применялось несколько технологий. Каждую такую технологию иногда называют *семейством* ИС. К середине 1970-х годов преобладали два семейства: TTL (произносится «ти-ти-эл») и CMOS («си-мос»).

TTL означает *transistor-transistor logic* - транзисторно-транзисторная логика. Такие микросхемы предпочитали те, для кого главным соображением была скорость. Микросхемы CMOS (complementary metal-oxide-semiconductor - комплементарная структура металл-оксид-полупроводник) потребляли меньше энергии и лучше переносили изменения напряжения, но уступали TTL в скорости.

Если в середине 1970-х годов вы были инженером по цифровым схемам (то есть проектировали более крупные схемы из ИС), постоянным предметом на вашем столе была книга толщиной 1¼ дюйма, впервые изданная Texas Instruments в 1973 году под названием *The TTL Data Book for Design Engineers*. Это полный справочник по серии интегральных схем TTL 7400 («семьдесят четыре сотни»), продававшихся Texas Instruments и несколькими другими компаниями. Серия называлась так потому, что каждая входившая в неё ИС обозначалась номером, начинавшимся цифрами 74.

Каждая интегральная схема серии 7400 состоит из логических вентилей, заранее соединённых в определённую конфигурацию. Одни микросхемы предоставляют простые заранее соединённые вентили, из которых можно строить более крупные компоненты; другие содержат распространённые компоненты целиком.

Первая ИС серии 7400 имеет собственно номер 7400 и описывается в *TTL Data Book* как «четыре двухвходовых положительных вентиля И-НЕ». Это означает, что данная интегральная схема содержит четыре двухвходовых вентиля И-НЕ. Они называются *положительными* вентилями И-НЕ, потому что входное напряжение 5 вольт (или около того) соответствует логической 1, а нулевое напряжение - 0. У этой микросхемы 14 выводов, и небольшая схема в справочнике показывает, как выводы соответствуют входам и выходам:



Это вид микросхемы сверху (выводы находятся снизу), а маленькая выемка, показанная на странице 193, расположена слева.

Вывод 14 обозначен `V_CC` и эквивалентен символу V, которым я обозначал напряжение. Вывод 7 обозначен GND, то есть *земля*. Каждая интегральная схема, используемая в конкретной цепи, должна быть подключена к общему 5-вольтовому источнику питания и общей земле. Каждый из четырёх вентилей И-НЕ микросхемы 7400 имеет два входа и один выход. Они работают независимо друг от друга.

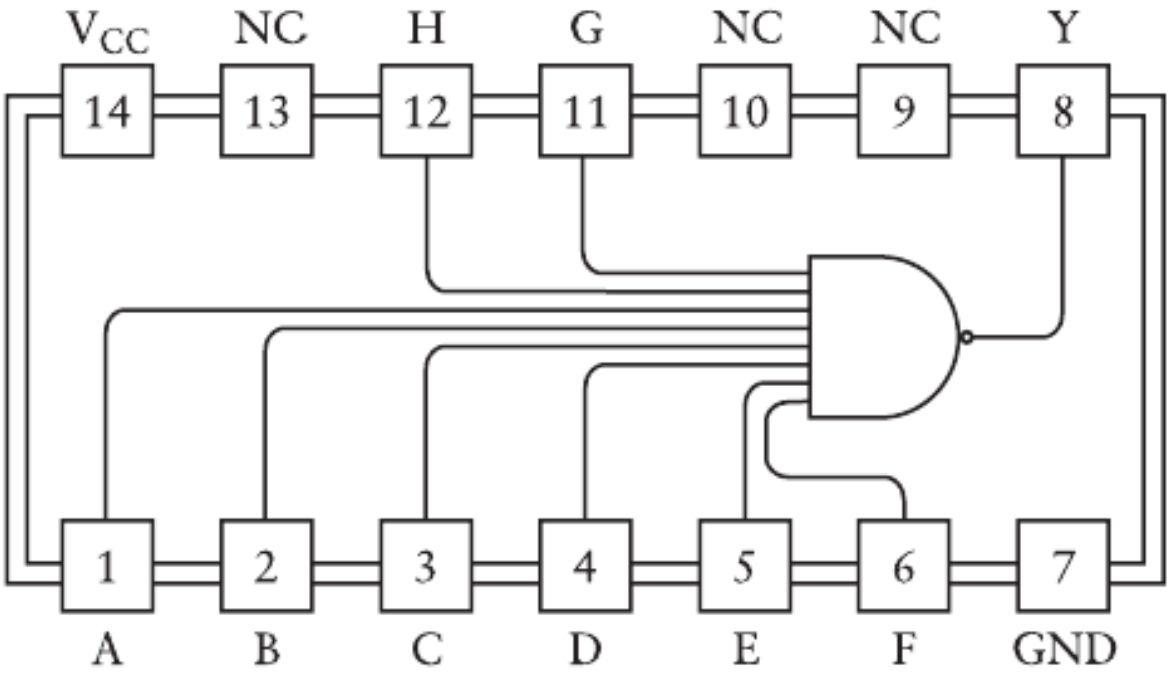
Важная характеристика конкретной интегральной схемы - *время распространения*, то есть время от изменения входов до соответствующего изменения выхода.

Время распространения микросхем обычно измеряется в *наносекундах*, сокращённо нс. Наносекунда - *очень* короткий промежуток. Тысячная доля секунды - миллисекунда. Миллионная доля секунды - микросекунда. Миллиардная доля секунды - наносекунда. Гарантируется, что время распространения вентилей И-НЕ в микросхеме 7400 составляет менее 22 наносекунд. Это 0,000000022 секунды, или 22 миллиардных доли секунды.

Если вы не можете почувствовать длительность наносекунды, вы не одиноки. Но если держать эту книгу на расстоянии 1 фута от лица, наносекунда - это время, за которое свет проходит от страницы до ваших глаз.

И всё же именно наносекунда делает компьютеры возможными. Каждый шаг компьютера представляет собой простейшую основную операцию, и что-либо существенное удаётся выполнить лишь потому, что эти операции происходят очень быстро. Процитируем Роберта Нойса: «После того как вы примиритесь с наносекундой, компьютерные операции станут концептуально довольно простыми».

Продолжим листать *The TTL Data Book for Design Engineers*. В этой книге вы увидите множество знакомых мелочей: микросхема 7402 содержит четыре двухвходовых вентиля ИЛИ-НЕ, 7404 - шесть инверторов, 7408 - четыре двухвходовых вентиля И, 7432 - четыре двухвходовых вентиля ИЛИ, а 7430 представляет собой восьмивходовый вентиль И-НЕ:



Сокращение NC означает *no connection*, то есть «не подключено».

Продвигаясь дальше по *TTL Data Book*, вы обнаружите, что микросхема 7483 представляет собой 4-битовый двоичный полный сумматор, 74151 - селектор данных «8 линий в 1», а 74154 - декодер «4 линии в 16».

Теперь вы знаете, откуда я взял все разнообразные компоненты, которые показывал в этой книге. Я украл их из *The TTL Data Book for Design Engineers*.

Одна из интересных микросхем в этой книге - 74182, называемая генератором ускоренного переноса. Она предназначена для совместного применения с другой микросхемой, 74181, которая выполняет сложение и другие арифметические операции. Как вы видели при построении 8-битового сумматора в главе 14, каждый его бит зависит от переноса из предыдущего разряда. Это называется *последовательным распространением переноса*. Чем больше складываемые числа, тем дольше приходится ждать результат.

Генератор ускоренного переноса призван смягчить эту тенденцию: он содержит схему, специально вычисляющую бит переноса быстрее, чем это мог бы сделать сам сумматор. Разумеется, особая схема требует дополнительных логических вентилей, но сокращает общее время сложения. Иногда схему можно улучшить, перепроектировав так, чтобы удалить логические вентили, однако очень часто схему удаётся ускорить, добавив больше логических вентилей для решения конкретных задач.

Логические вентили - не метафоры и не воображаемые объекты. Они совершенно реальны. Когда-то логические вентили и сумматоры строили из реле, затем реле заменили электронными лампами, лампы - транзисторами, а транзисторы - интегральными схемами. Но лежащие в основе понятия остались в точности прежними.

Глава 16. А как же вычитание?

Убедившись, что реле, лампы или транзисторы действительно можно соединить для сложения двоичных чисел, вы, возможно, спросите: «А как же вычитание?» Не беспокойтесь: подобными вопросами вы не докучаете, а проявляете наблюдательность. Сложение и вычитание в некоторых отношениях дополняют друг друга, но механика этих операций весьма различна. Сложение последовательно движется от крайнего правого столбца цифр к крайнему левому. Каждый перенос из одного столбца прибавляется к следующему. Но при вычитании мы не переносим, а *занимаем*, и это требует принципиально иного механизма - беспорядочного движения туда и обратно.

Рассмотрим, например, типичный пример вычитания с несколькими заимствованиями:

```
 253
-176
----
 ???
```

Если вы похожи на меня, то, увидев в крайнем правом столбце, что 6 больше 3, говорите: «Фу». Нужно занять из следующего столбца слева, а тому столбцу тоже требуется заимствование. Я не стану разбирать неприятные подробности, но если выполнить всё правильно (или допустить те же ошибки, что и я), получится 77:

```
 253
-176
----
  77
```

Как же заставить набор логических вентилей проделать все эти противоестественные рассуждения и получить такой результат?

Мы и не станем пытаться. Вместо этого воспользуемся небольшим приёмом, позволяющим вычитать *без заимствования*. Сначала он может показаться фокусом, но это важнейший первый шаг к пониманию того, как отрицательные числа хранятся в компьютерах.

Вероятно, ещё в начале обучения вам говорили, что вычитание равнозначно прибавлению отрицательного числа. В одном смысле это бесполезная информация, поскольку вычитать от этого не легче. Но она означает, что вычитание можно переписать как сложение положительного числа с отрицательным:

```
-176 + 253 =
```

Теперь я хочу добавить ещё пару чисел - одно положительное и одно отрицательное, - чтобы получилась сумма четырёх чисел:

```
1000 - 176 + 253 - 1000 =
```

Мы прибавляем 1000, а затем вычитаем 1000, поэтому результат не должен измениться. Известно, что 1000 равно 999 плюс 1, поэтому вместо 1000 в начале можно взять 999, а единицу прибавить позже. Последовательность становится ещё длиннее, но остаётся равнозначной:

```
999 - 176 + 253 + 1 - 1000 =
```

Нагромождение внушительное, но начнём обрабатывать его слева направо. Первый шаг - вычитание 176 из 999. Поразительно, но занимать ничего не нужно! Легко вычислить, что 999 минус 176 равно 823:

```
823 + 253 + 1 - 1000 =
```

Результат вычитания числа из последовательности девяток называется *дополнением до девяти*. Дополнение 176 до девяти равно 823. Это работает и в обратную сторону: дополнение 823 до девяти равно 176. И вот что особенно приятно: каким бы ни было число, вычисление дополнения до девяти *никогда не требует заимствования*.

Следующие два шага включают лишь сложение. Сначала сложим 253 и 823 и получим 1076:

```
1076 + 1 - 1000 =
```

Затем прибавим 1 и вычтем 1000:

```
1077 - 1000 = 77
```

Получился прежний ответ, но без единого неприятного заимствования.

Важно следующее: применяя дополнение до девяти для упрощения вычитания, нужно знать, со сколькими цифрами вы работаете. Если одно или оба числа имеют четыре цифры, дополнение до девяти следует вычислять с помощью 9999, а в конце вычитать 10000.

Но что, если большее число вычитается из меньшего? Например:

```
 176
-253
----
 ???
```

Обычно, взглянув на это, вы скажете: «Хм. Из меньшего числа вычитается большее, поэтому надо поменять числа местами, выполнить вычитание и помнить, что результат на самом деле отрицательный». Возможно, вы переставите числа в уме и запишете ответ так:

```
 176
-253
----
 -77
```

Расчёт без заимствования немного отличается от предыдущего примера, но начнём с перестановки двух чисел, добавим 999 в начале и вычтем 999 в конце:

```
999 - 253 + 176 - 999 =
```

Как и прежде, сначала вычтем 253 из 999 и получим дополнение до девяти:

```
746 + 176 - 999 =
```

Теперь прибавим дополнение до девяти к 176:

```
922 - 999 =
```

В предыдущем примере на этом этапе можно было прибавить 1 и вычесть 1000, получив окончательный результат, но здесь такой способ не годится. У нас остаются положительное 922 и отрицательное 999. Если бы 922 было отрицательным, а 999 положительным, можно было бы просто найти дополнение 922 до девяти, равное 77. Но поскольку для вычисления этого дополнения мы поменяли знаки, настоящий ответ равен -77. Это не так прямолинейно, как в первом примере, но заимствование опять не понадобилось.

Тот же приём можно применять к двоичным числам, причём с ними он даже проще. Посмотрим, как это работает.

Исходный пример вычитания:

```
 253
-176
----
 ???
```

После преобразования чисел в двоичные он принимает вид:

```
 11111101
-10110000
---------
 ????????
```

Сначала поменяем числа местами, чтобы задача превратилась в сложение положительного и отрицательного числа. Под двоичными числами показаны десятичные эквиваленты:

```
-10110000 + 11111101 =
     -176 +      253 =
```

Теперь прибавим в начале 11111111 (десятичное 255), а позднее прибавим 00000001 (десятичную 1) и вычтем 100000000 (десятичное 256):

```
11111111 - 10110000 + 11111101 + 00000001 - 100000000 =
      255 -      176 +      253 +        1 -       256 =
```

Первое двоичное вычитание не требует заимствований, потому что число вычитается из 11111111:

```
11111111 - 10110000 + 11111101 + 00000001 - 100000000 =
01001111            + 11111101 + 00000001 - 100000000 =
```

Когда десятичное число вычитают из последовательности девяток, получается дополнение до девяти. Для двоичных чисел результат вычитания из последовательности единиц называется *дополнением до единицы*. Но для его вычисления вовсе не нужно выполнять вычитание. Дополнение 10110000 до единицы равно 01001111, а дополнение 01001111 до единицы - 10110000:

```
1 0 1 1 0 0 0 0
0 1 0 0 1 1 1 1
```

Биты просто инвертируются: каждый бит 0 превращается в дополнении до единицы в 1, а каждый бит 1 - в 0. Поэтому дополнение до единицы часто называют *инверсией*. (Здесь можно вспомнить, что в главе 8 мы построили логический вентиль-инвертор, превращающий 0 в 1, а 1 в 0.)

Теперь задача выглядит так:

```
01001111 + 11111101 + 00000001 - 100000000 =
      79 +      253 +        1 -       256 =
```

Сложим первые два числа:

```
01001111 + 11111101 + 00000001 - 100000000 =
101001100            + 00000001 - 100000000 =
```

Результат имеет 9 битов, но это допустимо. Задача сократилась до:

```
101001100 + 00000001 - 100000000 =
      332 +        1 -       256 =
```

Прибавить 1 совсем просто:

```
101001101 - 100000000 =
      333 -       256 =
```

Остаётся вычесть двоичный эквивалент 256, то есть просто удалить крайний левый бит:

```
01001101
      77
```

К вашему удовольствию, окончательный результат совпадает с ответом, полученным в десятичной системе.

Повторим всё с числами в обратном порядке. Десятичный пример:

```
 176
-253
----
 ???
```

В двоичном виде:

```
 10110000
-11111101
---------
 ????????
```

Как и с десятичными числами, поменяем порядок. Прибавим 11111111 в начале и вычтем 11111111 в конце:

```
11111111 - 11111101 + 10110000 - 11111111 =
      255 -      253 +      176 -       255 =
```

Первый шаг - найти дополнение 11111101 до единицы:

```
11111111 - 11111101 + 10110000 - 11111111 =
00000010            + 10110000 - 11111111 =
```

Прибавим его к следующему числу:

```
10110010 - 11111111 =
      178 -       255 =
```

Теперь из результата каким-то образом нужно вычесть 11111111. При вычитании меньшего числа из большего мы прибавляли 1 и вычитали 100000000. Но здесь так вычесть без заимствования нельзя. Поэтому вместо этого вычтем результат из 11111111:

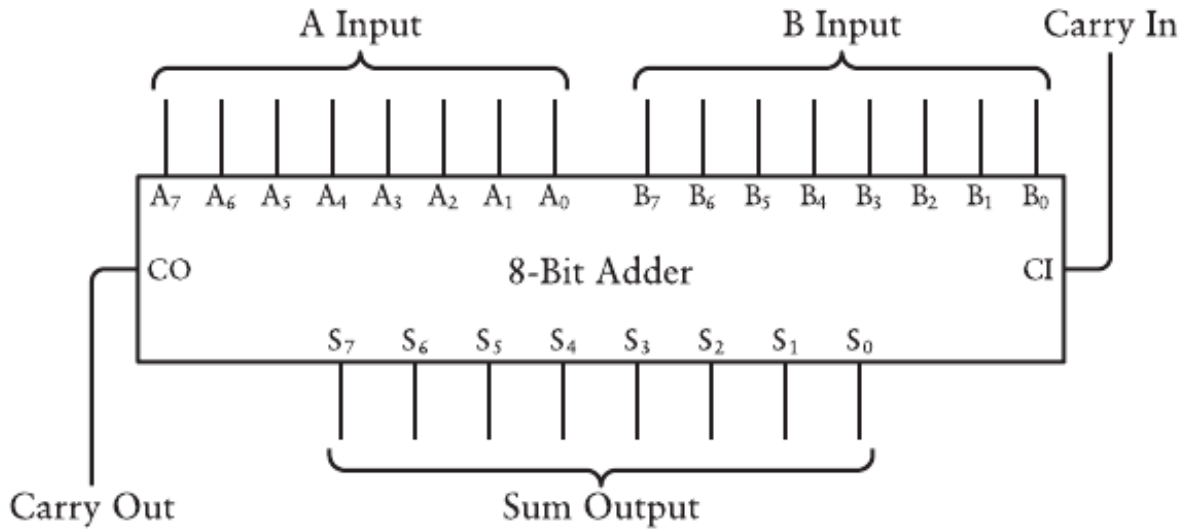
```
11111111 - 10110010 = 01001101
      255 -      178 =       77
```

Такая стратегия опять означает простое инвертирование всех битов результата. Ответ снова представляет двоичный эквивалент 77, но для этой задачи он на самом деле равен -77.

Небольшая странность, возникающая при вычитании большего числа из меньшего, словно шепчет нам, что мы ещё не дошли до конца и пока не вполне решили задачу.

Тем не менее у нас уже есть все знания, необходимые для изменения суммирующей машины из главы 14 так, чтобы она могла не только складывать, но и вычитать. Чтобы устройство не стало *слишком* сложным, новая машина сложения и вычитания будет выполнять лишь те вычитания, результат которых положителен.

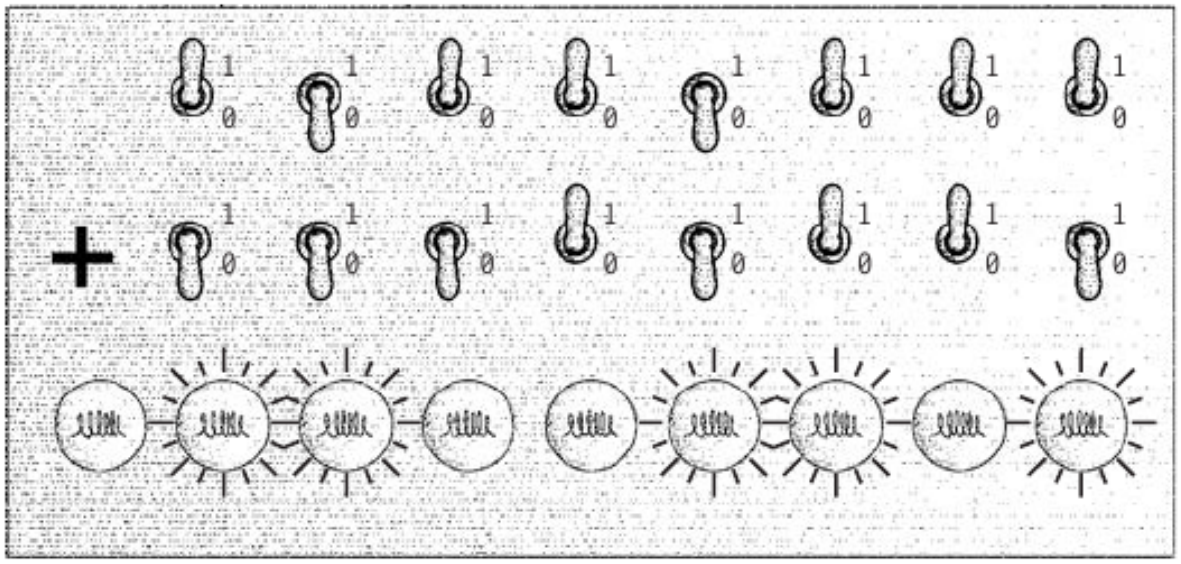
Основой суммирующей машины был 8-битовый сумматор из логических вентилей:



Как вы, вероятно, помните, входы от A_0 до A_7 и от B_0 до B_7 соединялись с переключателями, задававшими два складываемых 8-битовых значения.

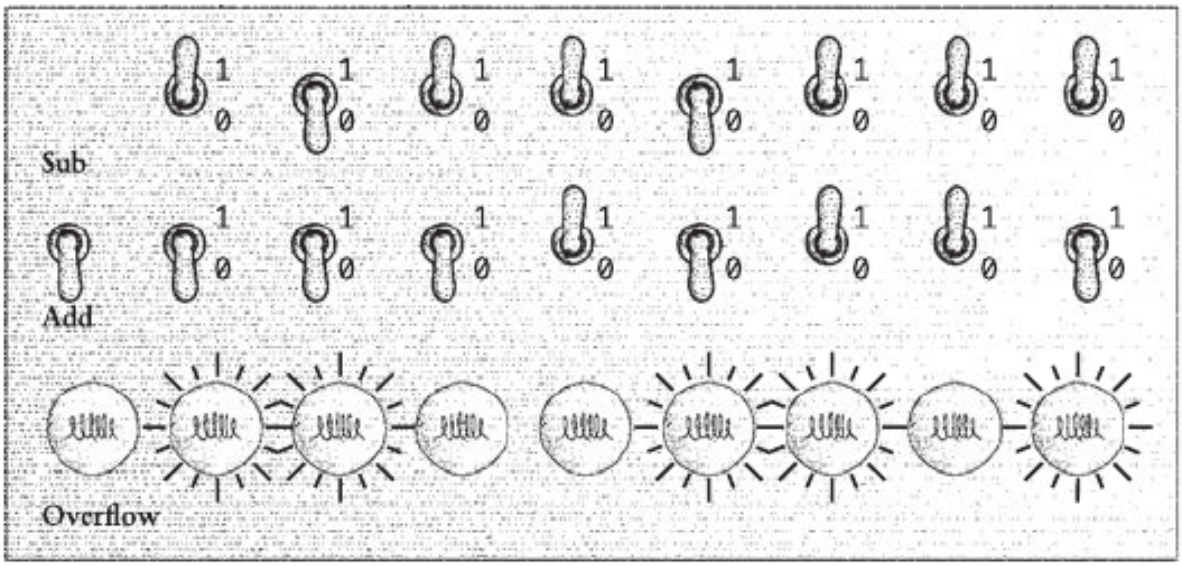
Вход Carry In соединялся с землёй, то есть с битом 0. Выходы от S_0 до S_7 подключались к восьми лампочкам, показывавшим результат сложения. Поскольку сложение могло дать 9-битовое значение, выход Carry Out также подключался к девятой лампочке.

Панель управления выглядела так:



На рисунке переключатели установлены для сложения 183 (10110111) и 22 (00010110), а ряд лампочек показывает результат 205, или 11001101.

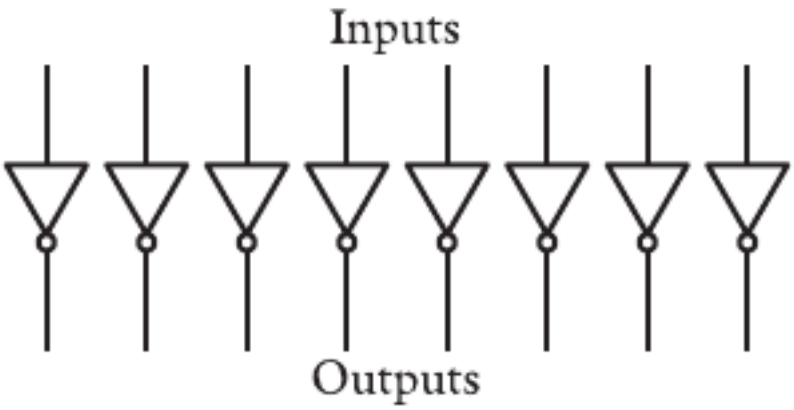
Новая панель управления для сложения и вычитания двух 8-битовых чисел изменена совсем немного. Вместо большого знака плюс на ней есть дополнительный переключатель, указывающий, нужно складывать или вычитать:



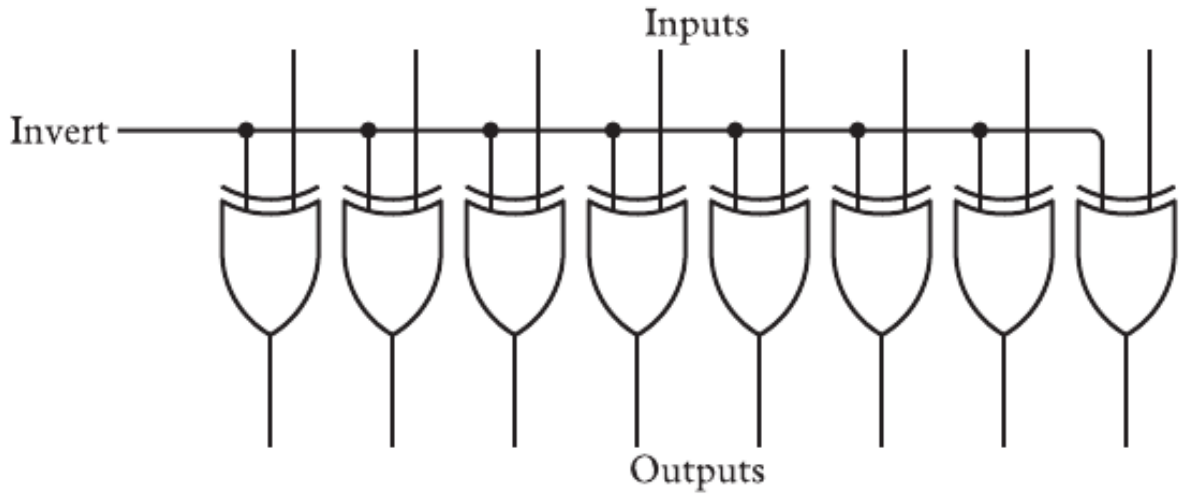
Для сложения этот переключатель выключают, а для вычитания включают, как и подписано. Ещё одно отличие: результат показывают только восемь крайних правых лампочек. Девятая теперь подписана «Overflow», то есть «Переполнение». В программировании этот термин встречается в нескольких контекстах и почти всегда указывает на проблему. Панель предназначена для сложения двух 8-битовых чисел, и во многих случаях результат также будет 8-битовым. Но если его длина равна 9 битам, возникает переполнение: результат выходит за выделенные для его отображения пределы. Переполнение может произойти и при вычитании, если большее число вычитается из меньшего.

Лампочка Overflow означает, что результат отрицателен, но отображение такого результата мы пока должным образом не предусмотрели.

Главное дополнение к суммирующей машине - схема, вычисляющая дополнение 8-битового числа до единицы. Поскольку оно равнозначно инвертированию битов, схема могла бы состоять всего из восьми инверторов:



Проблема в том, что такая схема инвертирует входные биты *всегда*. Мы создаём машину и для сложения, *и* для вычитания, поэтому биты нужно инвертировать лишь при вычитании. Более удачная схема выглядит так:

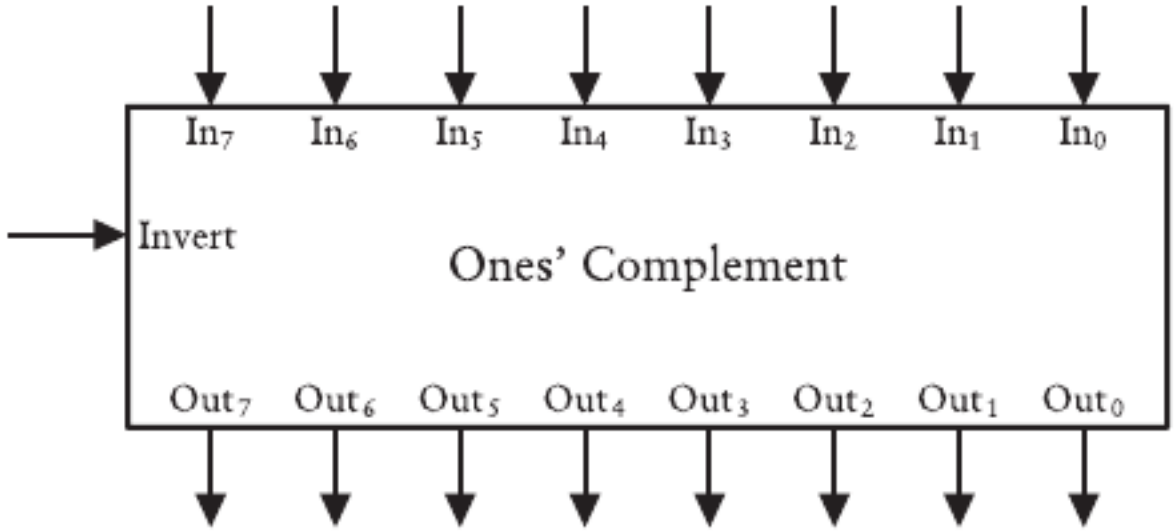


Один сигнал Invert подаётся на каждый из восьми вентилей XOR. Напомним его поведение:

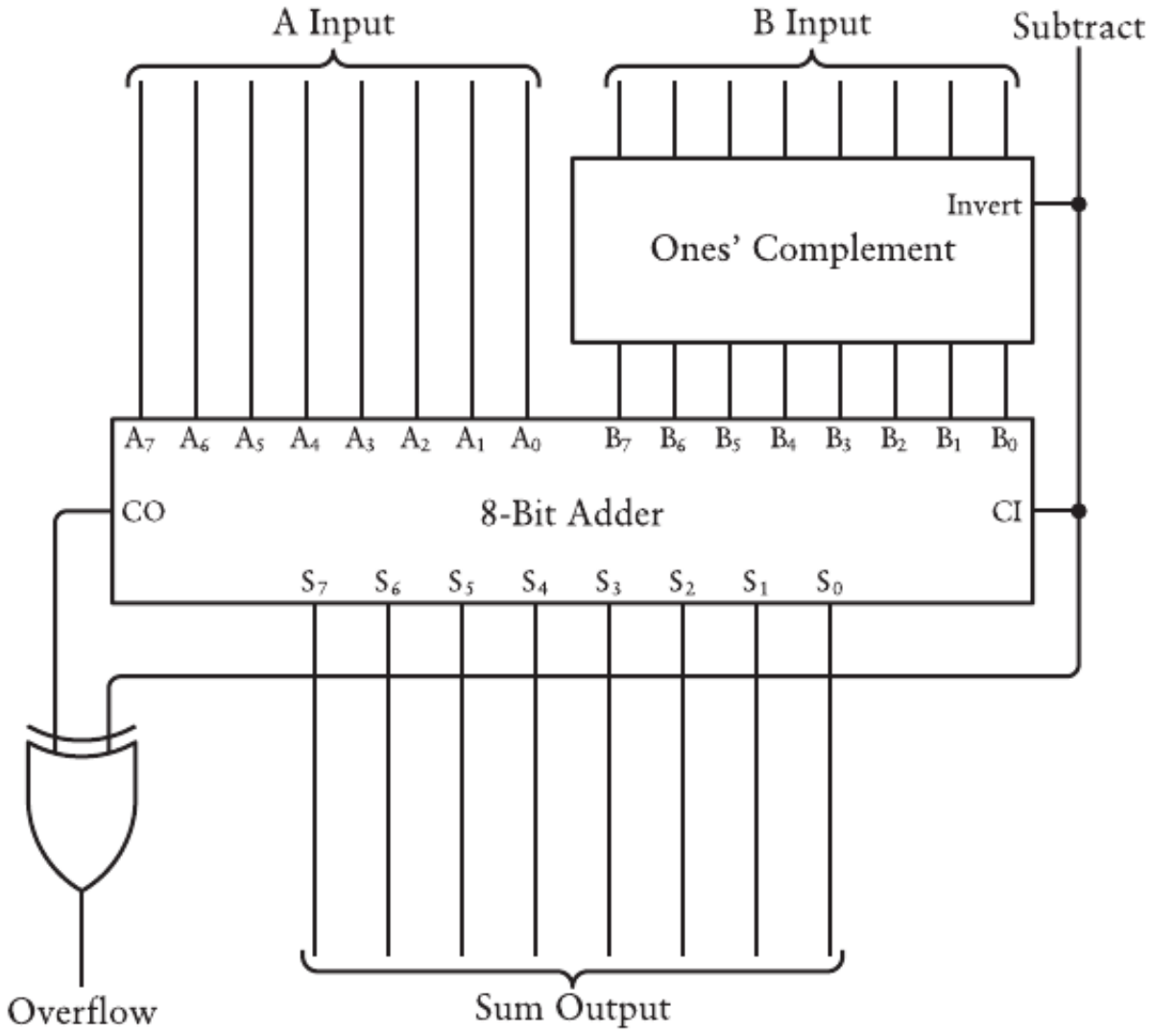
XOR	**0**	**1**
0	0	1
1	1	0

Если Invert равен 0, восемь выходов XOR совпадают с восемью входами. Например, вход 01100001 даёт выход 01100001. Если Invert равен 1, восемь входных сигналов инвертируются: вход 01100001 даёт выход 10011110.

Упакуем восемь вентилей XOR в блок «Дополнение до единицы»:



Теперь блок дополнения до единицы, 8-битовый сумматор и последний вентиль XOR можно соединить так:



Обратите внимание на провод Subtract в правом верхнем углу. Он идёт от переключателя Add/Subtract. При сложении сигнал равен 0, при вычитании - 1. Для сложения вход Invert блока дополнения до единицы равен 0, и блок ничего не изменяет. Вход CI тоже равен 0. Получается обычная схема сложения.

При вычитании все входы B (второй ряд переключателей панели) инвертируются блоком дополнения до единицы до поступления в сумматор. Кроме того, к результату сложения прибавляется 1: для этого вход CI (Carry In) сумматора устанавливается в 1.

Сигнал Subtract и выход CO (Carry Out) сумматора также поступают на вентиль XOR, включающий лампочку Overflow. Если Subtract равен 0, то есть выполняется сложение, лампочка горит при CO = 1. Это означает, что результат сложения больше 255.

Если выполняется вычитание и число B меньше числа A, нормальный выход CO сумматора равен 1. Он представляет 100000000, которое нужно вычесть на последнем шаге. При вычитании лампочка Overflow горит только тогда, когда CO равен 0. Значит, мы пытаемся вычесть большее число из меньшего. Показанная машина не рассчитана на отображение отрицательных чисел.

Теперь вы наверняка рады, что спросили: «А как же вычитание?»

В этой главе я говорил об отрицательных числах, но ещё не показывал, как выглядят отрицательные двоичные числа. Можно предположить, что традиционный знак минус используется так же, как в десятичной системе: например, -77 записывается как -1001101. Так делать можно, но одна из целей двоичной записи - представлять *всё* нулями и единицами, даже такой крошечный знак, как минус.

Конечно, можно просто выделить ещё один бит для знака: 1 для отрицательного числа и 0 для положительного, оставив остальное без изменений. Это работало бы, но существует более совершенное решение, ставшее стандартом представления отрицательных чисел в компьютерах. Главная причина его распространения - возможность без лишних хлопот складывать отрицательные и положительные числа. Основной недостаток - необходимость заранее определить число цифр для всех возможных значений.

Обычная запись положительных и отрицательных чисел позволяет им продолжаться бесконечно. Мы представляем 0 серединой бесконечного потока положительных чисел в одном направлении и отрицательных в другом:

```
... -1 000 000 -999 999 ... -3 -2 -1 0 1 2 3 ... 999 999 1 000 000 ...
```

Но предположим, что бесконечное множество не нужно и заранее известно, что любое число лежит в определённом диапазоне.

Рассмотрим текущий банковский счёт - одно из мест, где люди иногда встречают отрицательные числа. Допустим, на нём никогда нет целых 500 долларов, а банк разрешает овердрафт до 500 долларов. Баланс всегда лежит между $499 и -$500. Допустим также, что вы никогда не вносите сразу $500, не выписываете чек более чем на $500, работаете только с долларами и не учитываете центы.

Таким образом, диапазон равен от -500 до 499 - всего 1000 чисел. Это позволяет тремя десятичными цифрами без знака минус представить *все* необходимые значения. Положительные числа от 500 до 999 не нужны, поскольку максимальное требуемое положительное число - 499. Поэтому трёхзначные числа 500-999 могут обозначать отрицательные:

```
-500 означает 500.
-499 означает 501.
-498 означает 502.
...
-2 означает 998.
-1 означает 999.
0 означает 000.
1 означает 001.
2 означает 002.
...
497 означает 497.
498 означает 498.
499 означает 499.
```

Иными словами, каждое трёхзначное число, начинающееся с 5, 6, 7, 8 или 9, на самом деле отрицательно. Вместо чисел, расходящихся от нуля в двух направлениях:

```
-500 -499 -498 ... -4 -3 -2 -1 0 1 2 3 4 ... 497 498 499
```

их можно записать так:

```
500 501 502 ... 996 997 998 999 000 001 002 003 004 ... 497 498 499
```

Получается своего рода круг. Наименьшее отрицательное число 500 словно продолжает наибольшее положительное 499. Число 999 (то есть -1) на единицу меньше нуля. Обычно 999 + 1 даёт 1000, но при работе только с тремя цифрами результат равен 000.

Такая запись называется *дополнением до десяти*. Чтобы преобразовать трёхзначное отрицательное число, вычтите его из 999 и прибавьте 1. Иными словами, дополнение до десяти равно дополнению до девяти плюс один. Для записи -255 вычтем 255 из 999, получим 744 и прибавим 1: получится 745.

При использовании дополнения до десяти числа вообще не вычитают. Всё сводится к сложению.

Допустим, баланс счёта равен $143, и вы выписываете чек на $78. Обычно новый баланс вычисляется так:

```
 143
- 78
----
  65
```

Это вычитание с двумя заимствованиями. В дополнении до десяти -78 записывается как 999 - 078 + 1, то есть 922:

```
 143
+922
----
1065
```

Игнорируем переполнение и снова получаем $65. Если затем выписать чек на $150, нужно прибавить -150, равное в дополнении до десяти 850:

```
 65
+850
----
915
```

Результат начинается с 9, следовательно, это отрицательное число -$85.

Эквивалентная двоичная система называется *дополнением до двух* и является стандартным способом представления положительных и отрицательных чисел в компьютерах.

Будем работать с байтами, то есть 8-битовыми числами от 00000000 до 11111111. До сих пор они соответствовали десятичным 0-255. Но для выражения отрицательных чисел каждое 8-битовое число, начинающееся с 1, считается отрицательным:

Двоичное	**Десятичное**
10000000	-128
10000001	-127
10000010	-126
10000011	-125
...	...
11111101	-3
11111110	-2
11111111	-1
00000000	0
00000001	1

Двоичное	Десятичное
00000010	2
...	...
01111100	124
01111101	125
01111110	126
01111111	127

Диапазон теперь ограничен значениями от -128 до +127. Старший (крайний левый) бит называется *знаковым*: для отрицательных чисел он равен 1, для положительных - 0.

Чтобы вычислить дополнение до двух, сначала найдите дополнение до единицы, затем прибавьте 1: инвертируйте все цифры и прибавьте 1. Десятичное 125 равно 01111101. Для записи -125 инвертируем биты и получаем 10000010, затем прибавляем 1 и получаем 10000011. Обратное преобразование выполняется так же.

Главное преимущество - запись положительных и отрицательных чисел без знака минус. Ещё важнее возможность свободно складывать их по обычным правилам. Сложим двоичные эквиваленты -127 и 124:

```
  10000001
 +01111100
 ---------
  11111101
```

Результат равен десятичному -3.

Следует остерегаться переполнения, когда результат сложения больше 127. Например, сложим 125 с самим собой:

```
  01111101
 +01111101
 ---------
  11111010
```

Старший бит суммы равен 1, поэтому результат приходится трактовать как отрицательное число -6. Сумма двух положительных чисел, конечно, не может быть отрицательной: результат явно неверен.

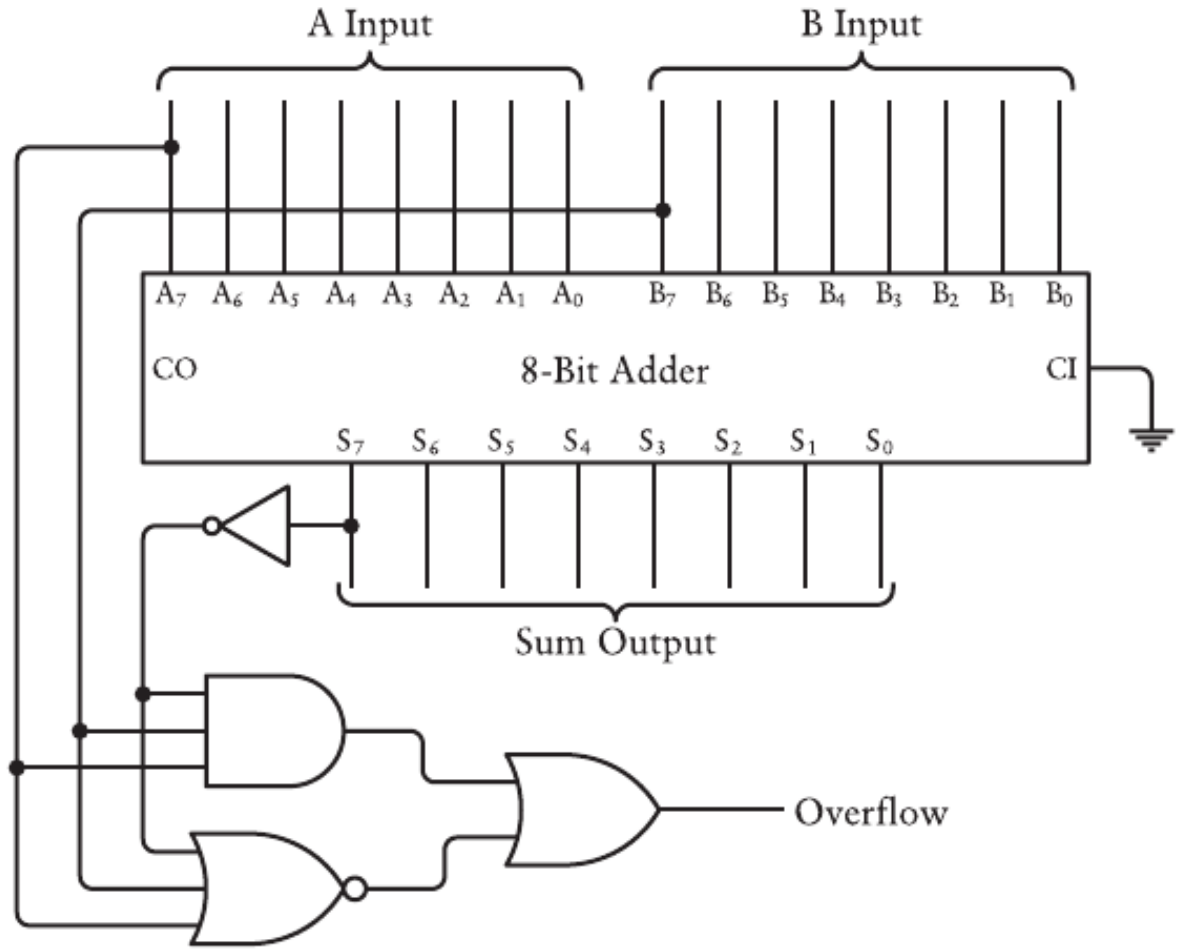
Сходное происходит при сложении -125 с самим собой:

```
  10000011
 +10000011
 ---------
 100000110
```

Мы заранее ограничились 8 битами, поэтому крайний левый бит нужно отбросить. Оставшиеся восемь равны положительному числу 6.

В общем случае сумма чисел в дополнении до двух недействительна, если знаковые биты двух операндов одинаковы, а знак результата отличается. Сумма положительного и отрицательного числа всегда действительна, поскольку всегда попадает в диапазон от -128 до 127.

Вот изменённая машина для сложения двух 8-битовых чисел в дополнении до двух:



8-битовый сумматор уже знаком. Несколько вентилей добавлены для обнаружения переполнения. Старшие биты представляют знак: 1 для отрицательного числа, 0 для положительного. На входе это A_7 и B_7, а знак суммы - S_7. Перед подачей на вентили И и ИЛИ-НЕ знак суммы инвертируется.

Вентиль И обнаруживает переполнение отрицательных чисел. Если знаки A и B оба равны 1, а знак суммы равен 0, что-то явно пошло не так: сумма двух отрицательных чисел настолько отрицательна, что не помещается в выделенные 8 битов.

Вентиль ИЛИ-НЕ обнаруживает переполнение положительных чисел. Если знаки A и B равны 0, а знак суммы - 1, два положительных числа дали настолько большое значение, что оно представлено отрицательным. Все три входа ИЛИ-НЕ равны 0, поэтому выход равен 1 и указывает переполнение.

В начале главы двоичные числа напрямую соответствовали десятичным: 8-битовое число лежало от 0 до 255. Такие числа называются *беззнаковыми*, потому что всегда положительны.

Дополнение до двух позволяет работать со *знаковыми* двоичными числами - положительными или отрицательными. Для 8 битов диапазон равен от -128 до 127. Чисел столько же (256), но диапазон другой.

Формальный математический термин для таких чисел - *целые*: они могут быть положительными или отрицательными, но не имеют дробной части. В реальных задачах 8-битовых целых часто недостаточно, поэтому программисты используют 16-битовые (2 байта), 32-битовые (4 байта) и даже 64-битовые (8 байтов) целые. Каждый размер может быть знаковым или беззнаковым:

Размер	**Беззнаковый диапазон**	**Знаковый диапазон**
8 битов	0 - 255	-128 - 127
16 битов	0 - 65 535	-32 768 - 32 767
32 бита	0 - 4 294 967 295	-2 147 483 648 - 2 147 483 647
64 бита	0 - 18 446 744 073 709 551 615	-9 223 372 036 854 775 808 - 9 223 372 036 854 775 807

Диапазоны основаны на степенях двойки. Например, 16 битов представляют 2 в степени 16, то есть 65 536 разных чисел: от 0 до 65 535 либо от -32 768 до 32 767.

Сами биты не сообщают, знаковое число или беззнаковое. Если кто-то скажет: «У меня есть 8-битовое число 10110110. Каков десятичный эквивалент?», сначала нужно спросить: «Оно знаковое или беззнаковое? Это может быть -74 или 182».

Биты - лишь нули и единицы. Сами по себе они *ничего* о себе не сообщают. Смысл определяется контекстом использования.

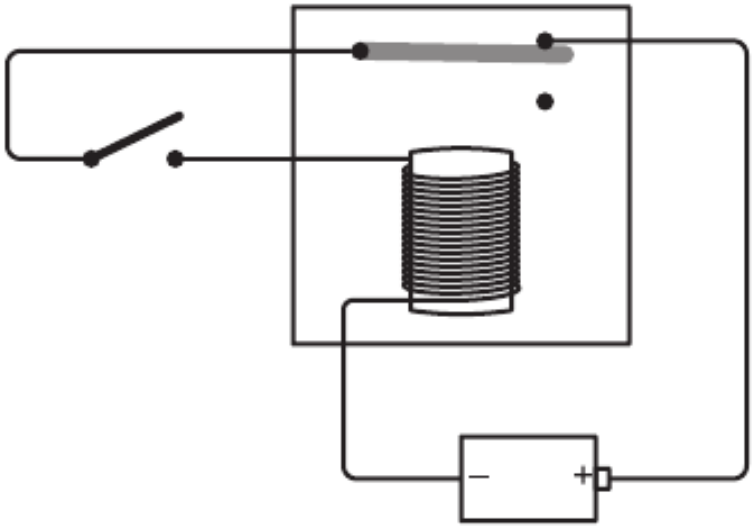
Эта страница намеренно оставлена пустой.

Глава 17. Обратная связь и триггеры

Всем известно, что электричество способно приводить предметы в движение. Даже беглого взгляда на обычный дом достаточно, чтобы увидеть электродвигатели в таких устройствах, как часы, вентиляторы, кухонные комбайны и всё, что вращает диск. Электричество также управляет колебаниями громкоговорителей, наушников и внутриканальных наушников, благодаря чему из множества наших устройств льются музыка и речь. И даже если снаружи не видно ни одной электрической машины, электродвигатель всё равно отвечает за запуск старинных двигателей внутреннего сгорания, работающих на ископаемом топливе.

Но, пожалуй, самый простой и изящный пример того, как электричество заставляет предметы двигаться, дают устройства, которые быстро исчезают по мере того, как их вытесняют электронные аналоги. Я говорю об удивительно старомодных электрических зуммерах и звонках.

Рассмотрим реле, соединённое с выключателем и батареей вот так:

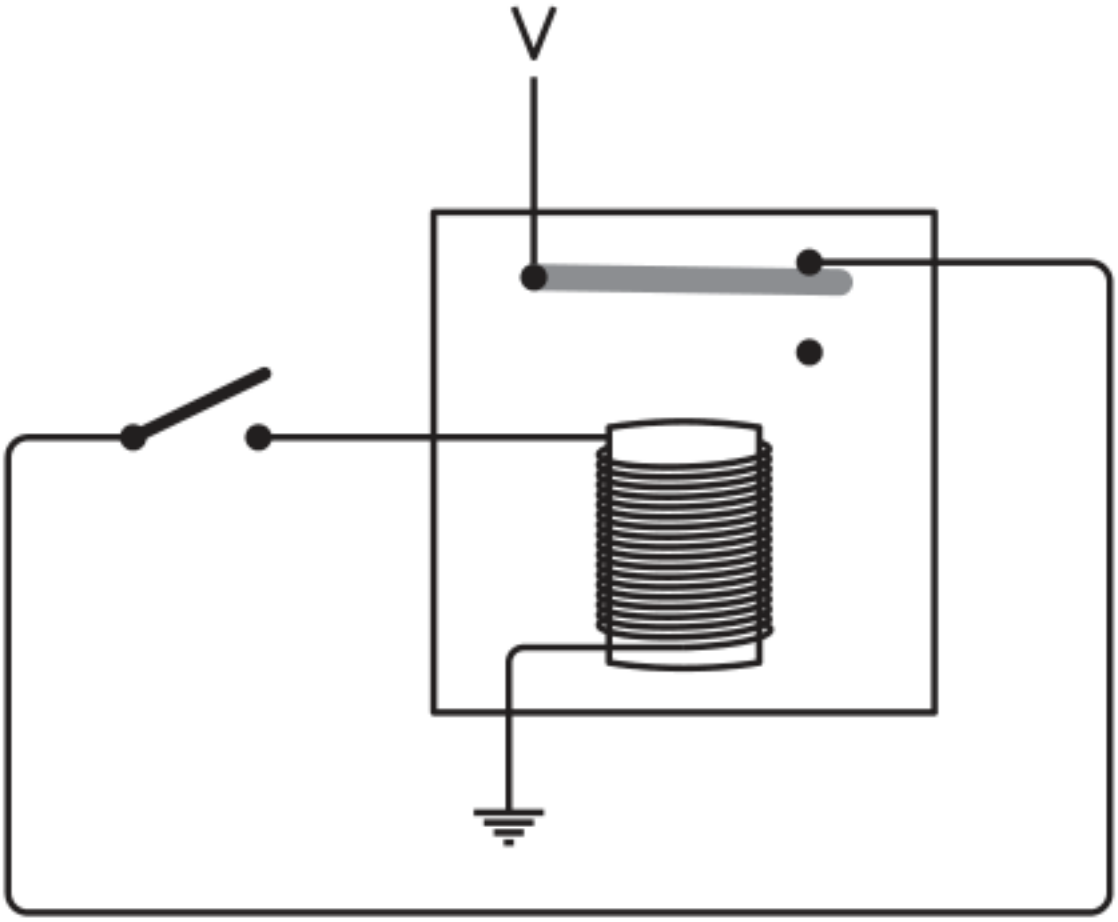


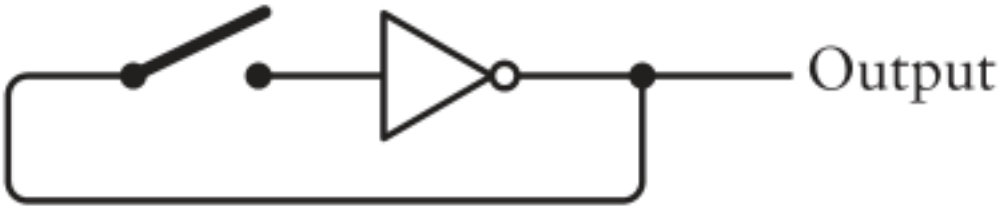
Если эта схема кажется вам немного странной, вам не мерещится. До сих пор реле подключалось у нас иначе. Обычно его вход отделён от выхода. Здесь же всё соединено в один большой контур.

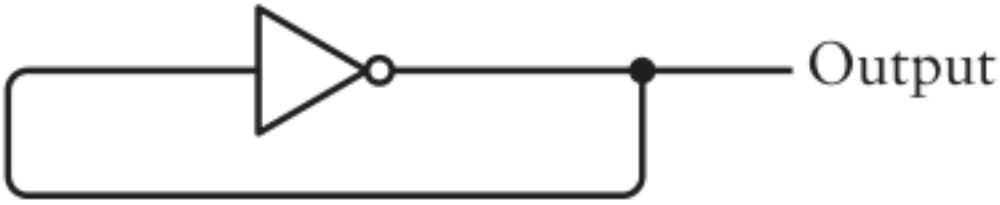
Если замкнуть выключатель, цепь замыкается. Электромагнит притягивает вниз гибкий контакт. Но когда контакт меняет положение, цепь размыкается, электромагнит теряет магнитные свойства, а гибкий контакт возвращается вверх.

Однако при этом цепь снова замыкается. Пока выключатель замкнут, металлический контакт ходит туда и обратно, попеременно замыкая и размыкая цепь и, скорее всего, издавая повторяющийся и, возможно, раздражающий звук. Если контакт производит скрежет, это зуммер. Если прикрепить к нему молоточек и поставить рядом металлический гонг, получится электрический звонок.

Зуммер на основе такого реле можно соединить несколькими способами. Вот ещё один вариант, использующий обычные обозначения напряжения и земли:







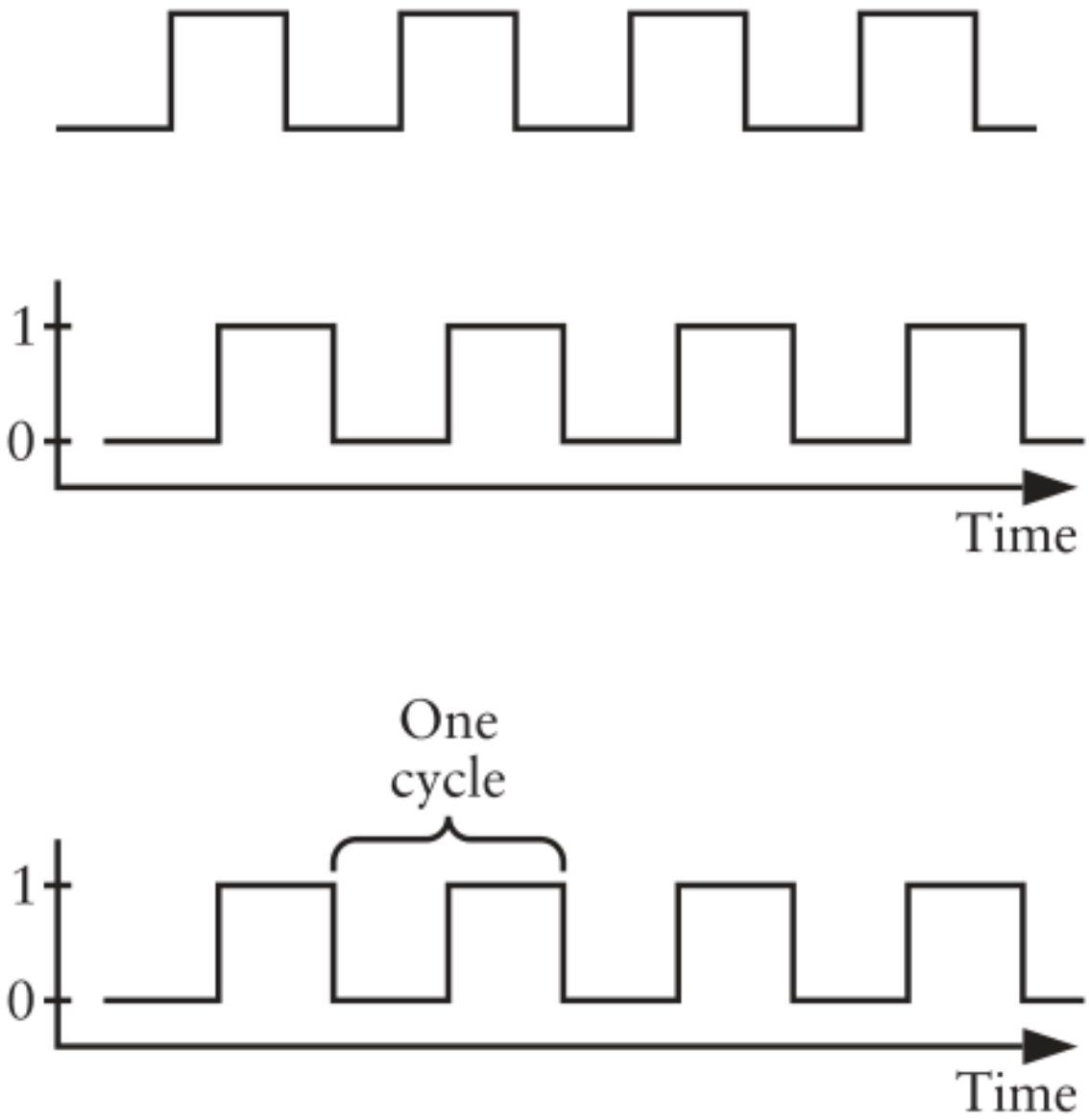
В таком виде вы, возможно, узнаете инвертор из главы 8. Схему можно изобразить ещё проще. Напомним: выход инвертора равен 1, если вход равен 0, и равен 0, если вход равен 1. Замыкание выключателя в этой схеме заставляет реле или транзистор внутри инвертора попеременно размыкаться и замыкаться. Можно также соединить инвертор без выключателя, чтобы процесс шёл непрерывно.

Каков выход этой схемы? Он быстро чередуется между наличием и отсутствием напряжения. Иначе говоря, *выход быстро чередуется между 0 и 1*.

Такая схема называется *генератором*. Она принципиально отличается от всего, что мы видели до сих пор. Состояние всех прежних схем изменял человек, замыкая или размыкая выключатель. Генератор в человеке не нуждается: он, по существу, работает сам.

Сам по себе генератор, конечно, кажется не слишком полезным. Но далее в этой и нескольких следующих главах мы увидим, что схема, соединённая с другими схемами, становится важнейшей частью автоматизации. Во всех компьютерах имеется какой-либо генератор, заставляющий остальные части двигаться синхронно. Компьютерные генераторы устроены несколько сложнее: в них применяют кристаллы кварца, соединённые так, чтобы они колебались очень устойчиво и очень быстро.

Выход генератора чередуется между 0 и 1. Обычно это представляют диаграммой такого вида:

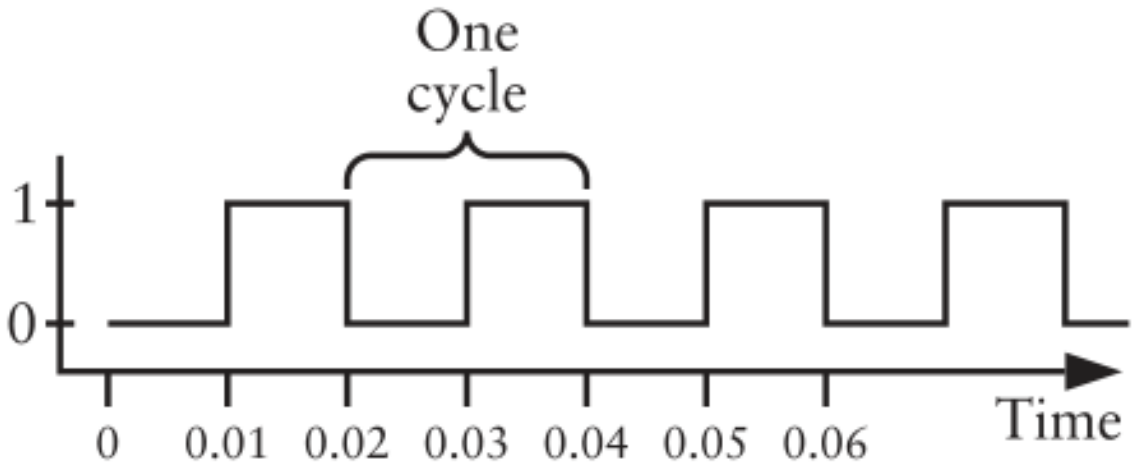


Это разновидность графика. По горизонтальной оси отложено время, а по вертикальной показано, равен ли выход 0 или 1. Всё, что здесь сказано, сводится к следующему: с течением времени выход генератора регулярно меняется между 0 и 1. Поэтому генератор иногда называют *тактовым генератором*, или просто *тактовым сигналом*: подсчитав число колебаний, можно в некотором смысле измерять время.

Насколько быстро он работает? Сколько раз в секунду выход меняется с 0 на 1? Это, разумеется, зависит от конструкции генератора. Легко представить большое тяжёлое реле, медленно и с грохотом переключающееся туда и обратно, и маленькое лёгкое реле, гудящее очень быстро. Транзисторный генератор способен колебаться миллионы или миллиарды раз в секунду.

Один *цикл* генератора определяется как интервал, за который его выход изменяется и затем возвращается в исходное состояние.

Время одного цикла называется *периодом* генератора. Предположим, что период некоторого генератора равен 0,02 секунды. Горизонтальную ось можно разметить в секундах, начиная с некоторого произвольно выбранного момента, обозначенного 0:



Частота генератора равна единице, делённой на период. В нашем примере период равен 0,02 секунды, поэтому частота составляет 1 ÷ 0,02, то есть 50 *циклов в секунду*. Пятьдесят раз в секунду выход генератора изменяется и возвращается обратно.

Выражение «циклов в секунду» довольно понятно, как «миль в час», «фунтов на квадратный дюйм» или «калорий на порцию». Однако теперь им почти не пользуются. В память о Генрихе Рудольфе Герце (1857-1894), первым передавшем и принявшем радиоволны, вместо него употребляют слово *герц*. Сначала такое употребление распространилось в Германии в 1920-х годах, а в других странах было принято в последующие десятилетия.

Итак, можно сказать, что частота нашего генератора равна 50 герцам, или, сокращённо, 50 Гц.

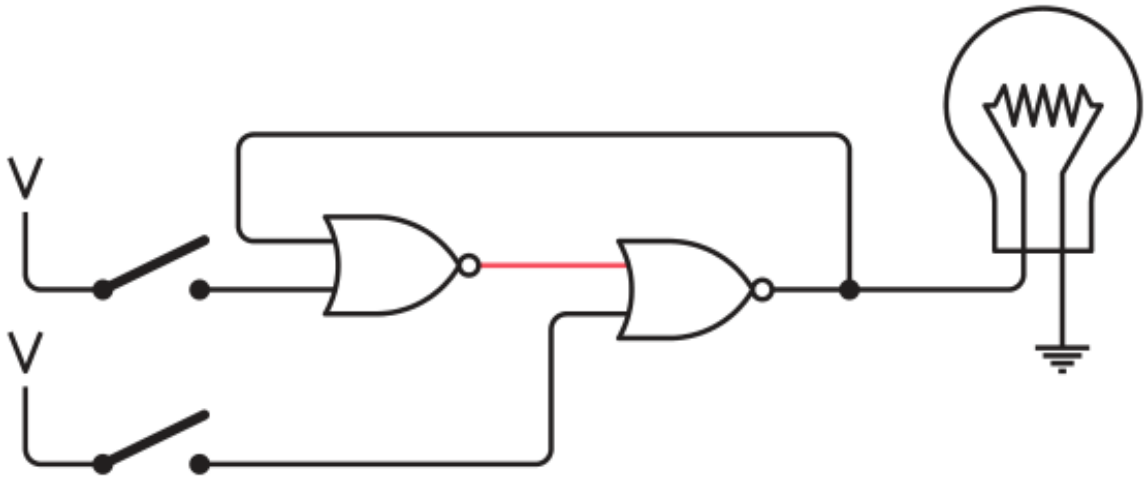
Скорость конкретного генератора мы, конечно, просто угадали. К концу этой главы мы сумеем построить устройство, позволяющее действительно измерить её.

Для начала рассмотрим пару вентилей ИЛИ-НЕ, соединённых необычным образом. Напомним, что выход вентиля ИЛИ-НЕ равен 1 лишь тогда, когда ни на одном входе нет напряжения:

ИЛИ-НЕ	**0**	**1**
0	1	0
1	0	0

Вот схема с двумя вентилями ИЛИ-НЕ, двумя выключателями и лампочкой:

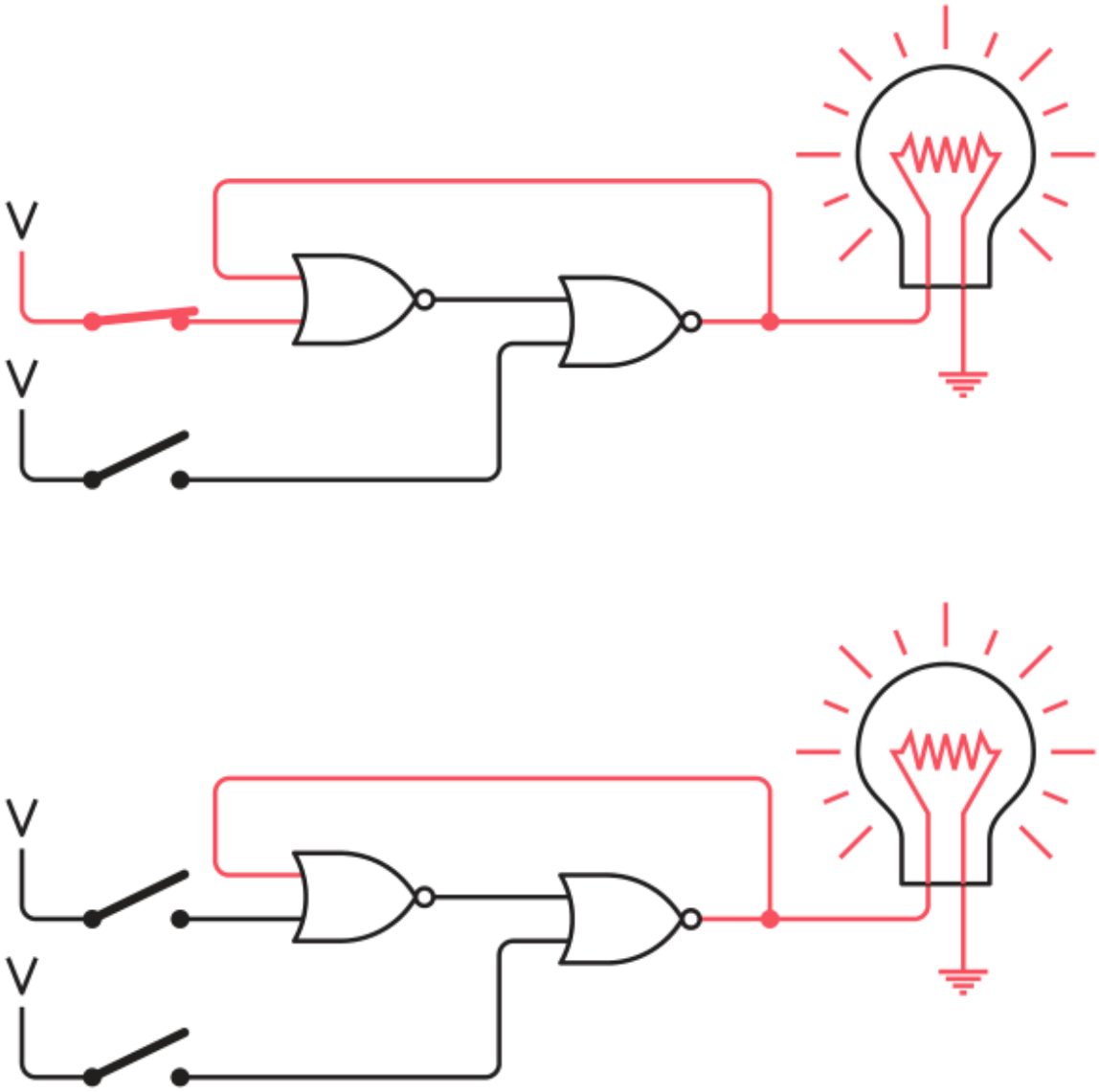
NOR	0	1
0	1	0
1	0	0



Обратите внимание на странно перекрученную проводку: выход левого вентиля ИЛИ-НЕ служит входом правого вентиля, а выход правого служит входом первого. Это разновидность *обратной связи*. Как и в генераторе, выходы замыкаются по кругу и становятся входами. Такая особенность будет характерна для большинства схем этой главы.

Есть простое правило: можно замкнуть либо верхний, либо нижний выключатель, но не оба одновременно. Дальнейшее рассуждение опирается на это правило.

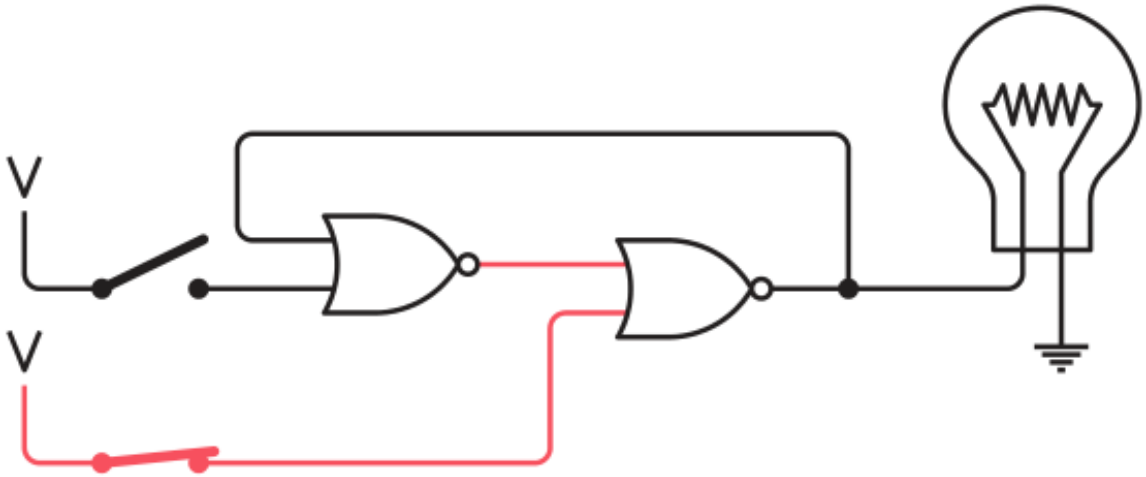
В исходном состоянии по цепи идёт ток от выхода левого вентиля ИЛИ-НЕ. Это происходит потому, что оба его входа равны 0. Теперь замкнём верхний выключатель. Выход левого вентиля становится 0, поэтому выход правого вентиля становится 1 и лампочка загорается:

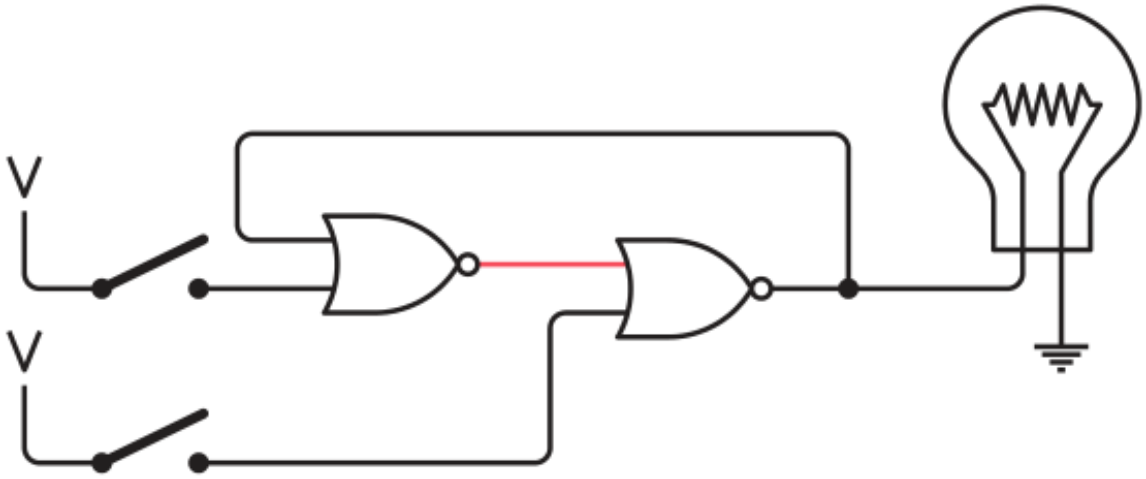


Теперь верхний выключатель можно разомкнуть. Происходит удивительное: поскольку выход вентиля ИЛИ-НЕ равен 0, если хотя бы один его вход равен 1, выход левого вентиля остаётся прежним, а лампочка продолжает гореть.

Это странно, не правда ли? Оба выключателя разомкнуты, как на первоначальном рисунке, но теперь лампочка горит. Такая ситуация определённо отличается от всего, что встречалось нам прежде. Обычно выход схемы зависит исключительно от её входов. Здесь это, по-видимому, не так. Более того, теперь верхний выключатель можно сколько угодно замыкать и размыкать: свет не изменится. Этот выключатель больше не действует, потому что выход левого вентиля остаётся равным 0.

Теперь замкнём нижний выключатель. Поскольку один из входов правого вентиля ИЛИ-НЕ становится равен 1, его выход становится равен 0 и лампочка гаснет. Выход левого вентиля ИЛИ-НЕ становится равен 1:





После этого нижний выключатель можно разомкнуть, а лампочка останется погашенной. Мы вернулись к исходному состоянию. Теперь нижний выключатель можно замыкать и размыкать без дальнейшего воздействия на лампочку.

Итак:

• замыкание верхнего выключателя включает лампочку, и после его размыкания она продолжает гореть;

• замыкание нижнего выключателя выключает лампочку, и после его размыкания она остаётся погашенной.

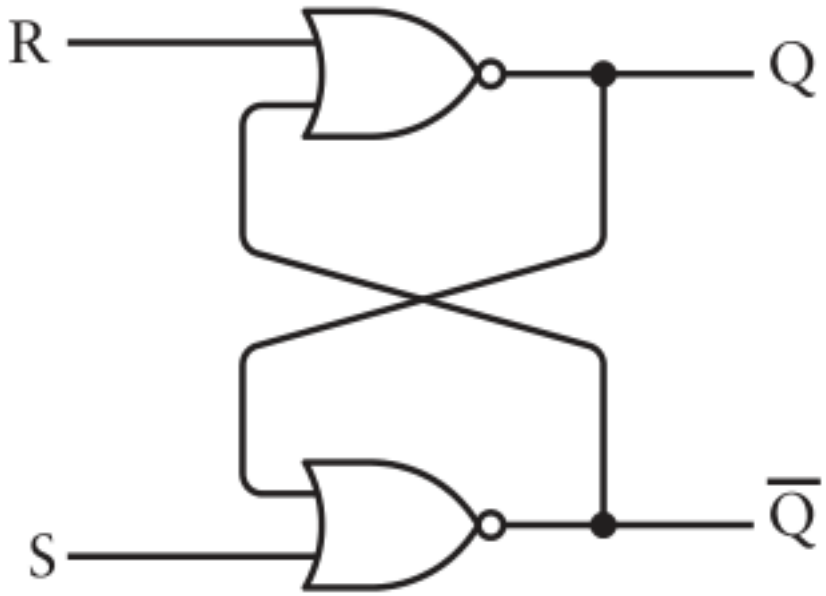
Странность схемы состоит в том, что иногда при обоих разомкнутых выключателях лампочка горит, а иногда не горит. Можно сказать, что у схемы есть *два устойчивых состояния*, когда оба выключателя разомкнуты. Такая схема называется *триггером* - словом, которое также означает вьетнамские сандалии и один из приёмов политиков. Триггер датируется 1918 годом и был изобретён английскими физиками Уильямом Генри Экклзом (1875-1966) и Ф. У. Джорданом (1881-1941).

Триггер *сохраняет информацию*. Он запоминает! Правда, он помнит только то, какой выключатель замыкался последним, но это уже существенно. Если вы встретите такую схему в дороге и увидите, что лампочка горит, можно заключить: последним замыкался верхний выключатель; если лампочка не горит, последним замыкался нижний.

Триггер очень похож на качели-балансир. У качелей два устойчивых состояния, а в шатком среднем положении они никогда надолго не остаются. По одному взгляду на качели всегда можно сказать, какую сторону опустили последней.

Хотя это пока не очевидно, триггеры являются важнейшими инструментами. Они добавляют схеме память, дают ей историю того, что происходило прежде. Представьте попытку считать, если бы вы ничего не могли запомнить: вы не знали бы, до какого числа дошли и какое число должно идти следующим. Подобным образом схема, считающая, - а далее в этой главе вы увидите такую схему - нуждается в триггерах.

Триггеры бывают нескольких разновидностей. Простейшая из только что рассмотренных называется *R-S-триггером*, или триггером Reset-Set, то есть «сброс-установка». Два вентиля ИЛИ-НЕ обычно рисуют более симметрично и обозначают так:



Выход, который мы использовали для лампочки, традиционно называется Q. Кроме него есть второй выход, $\overline{Q}$, произносимый «Q с чертой»; он противоположен Q. Если Q равно 0, то $\overline{Q}$ равно 1, и наоборот. Два входа называются S, то есть Set, «установка», и R, то есть Reset, «сброс». Эти глаголы можно понимать как «установить Q в 1» и «сбросить Q в 0». Когда S равен 1, что соответствует замыканию верхнего выключателя в прежней схеме, Q становится равным 1, а $\overline{Q}$ - 0. Когда R равен 1, что соответствует замыканию нижнего выключателя, Q становится равным 0, а $\overline{Q}$ - 1. Когда оба входа равны 0, выход показывает, что было сделано с Q последним: установка или сброс.

Вход S	**Вход R**	**Выход Q**	**Выход $\overline{Q}$**
1	0	1	0
0	1	0	1
0	0	Q	$\overline{Q}$
1	1	недопустимо	недопустимо

Эта таблица называется *таблицей функций*, *логической таблицей* или *таблицей истинности*. Она показывает все возможные сочетания входов и соответствующие им выходы.

Поскольку у R-S-триггера всего два входа, существует четыре их сочетания. Им соответствуют четыре строки таблицы под заголовками.

Обратите внимание на предпоследнюю строку, где S и R оба равны 0. Выходы обозначены Q и $\overline{Q}$. Это значит, что они сохраняются такими, какими были до того, как оба входа стали равны 0. Последняя строка сообщает, что одновременная единица на S и R *недопустима*, или незаконна. Схема от этого не выйдет из строя, однако оба выхода окажутся равны 0, что нарушает представление о $\overline{Q}$ как о противоположности Q. При проектировании схем с R-S-триггером следует избегать случая, когда S и R одновременно равны 1.

R-S-триггер часто изображают маленьким прямоугольником с двумя входами и двумя подписанными выходами.

Как первый пример схемы, которая словно «помнит», на каком из двух входов последним было напряжение, R-S-триггер, безусловно, интересен. Но гораздо полезнее была бы схема, запоминающая состояние определённого сигнала в определённый момент времени.

Подумаем, как должна вести себя такая схема, прежде чем пытаться её построить. У неё должно быть два входа. Один назовём Data, «данные». Как любой цифровой сигнал, Data может быть равен 0 или 1. Другой вход назовём Hold That Bit - это цифровой эквивалент слов «запомни эту мысль». Обычно Hold That Bit равен 0, и тогда сигнал Data не влияет на схему. Когда Hold That Bit равен 1, схема отражает значение Data. Затем Hold That Bit возвращается к 0, и схема запоминает последнее значение Data. Последующие изменения Data на неё не действуют.

Иными словами, нужна схема со следующей таблицей функций:

Data	Hold That Bit	Q
0	1	0
1	1	1
0	0	Q
1	0	Q

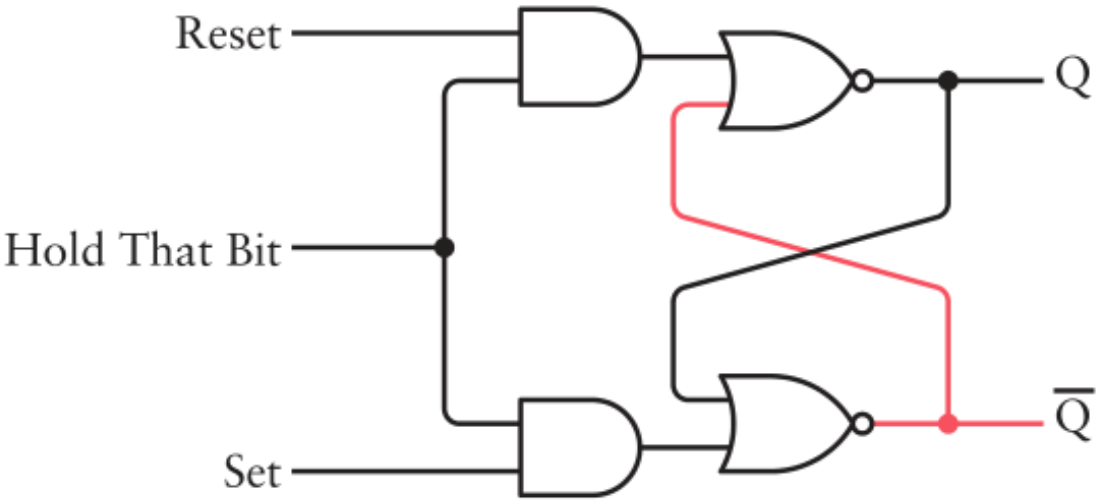
В первых двух случаях, когда Hold That Bit равен 1, выход Q совпадает с Data. В двух следующих случаях, когда Hold That Bit равен 0, Q остаётся прежним независимо от Data. Таблицу можно немного упростить:

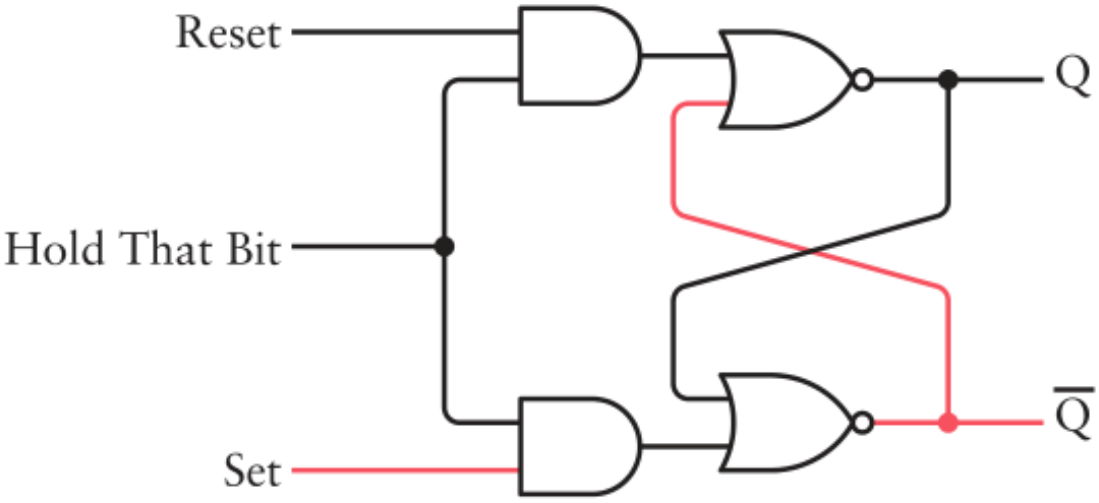
Data	Hold That Bit	Q
0	1	0
1	1	1
X	0	Q

X означает «не имеет значения». При Hold That Bit, равном 0, безразлично, чему равен Data: выход Q остаётся прежним.

Чтобы получить сигнал Hold That Bit из уже имеющегося R-S-триггера, добавим на входе два вентиля И, как на следующей схеме:

Inputs		Output
Data	Hold That Bit	Q
0	1	0
1	1	1
X	0	Q





Пока входа Data здесь нет, но мы вскоре его добавим. Напомним, что выход вентиля И равен 1 только при двух единицах на входах. Значит, сигналы Reset и Set никак не действуют на остальную схему, если Hold That Bit не равен 1.

Сначала выход Q равен 0, а $\bar{Q}$ имеет противоположное значение 1. Пока Hold That Bit равен 0, сигнал Set на выходы не влияет. То же верно для сигнала Reset.

Только при Hold That Bit, равном 1, схема действует как обычный R-S-триггер. Тогда выход верхнего вентиля И совпадает с Reset, а выход нижнего - с Set.

Но цели мы ещё не достигли. Нам нужны только два входа, а не три. Как этого добиться?

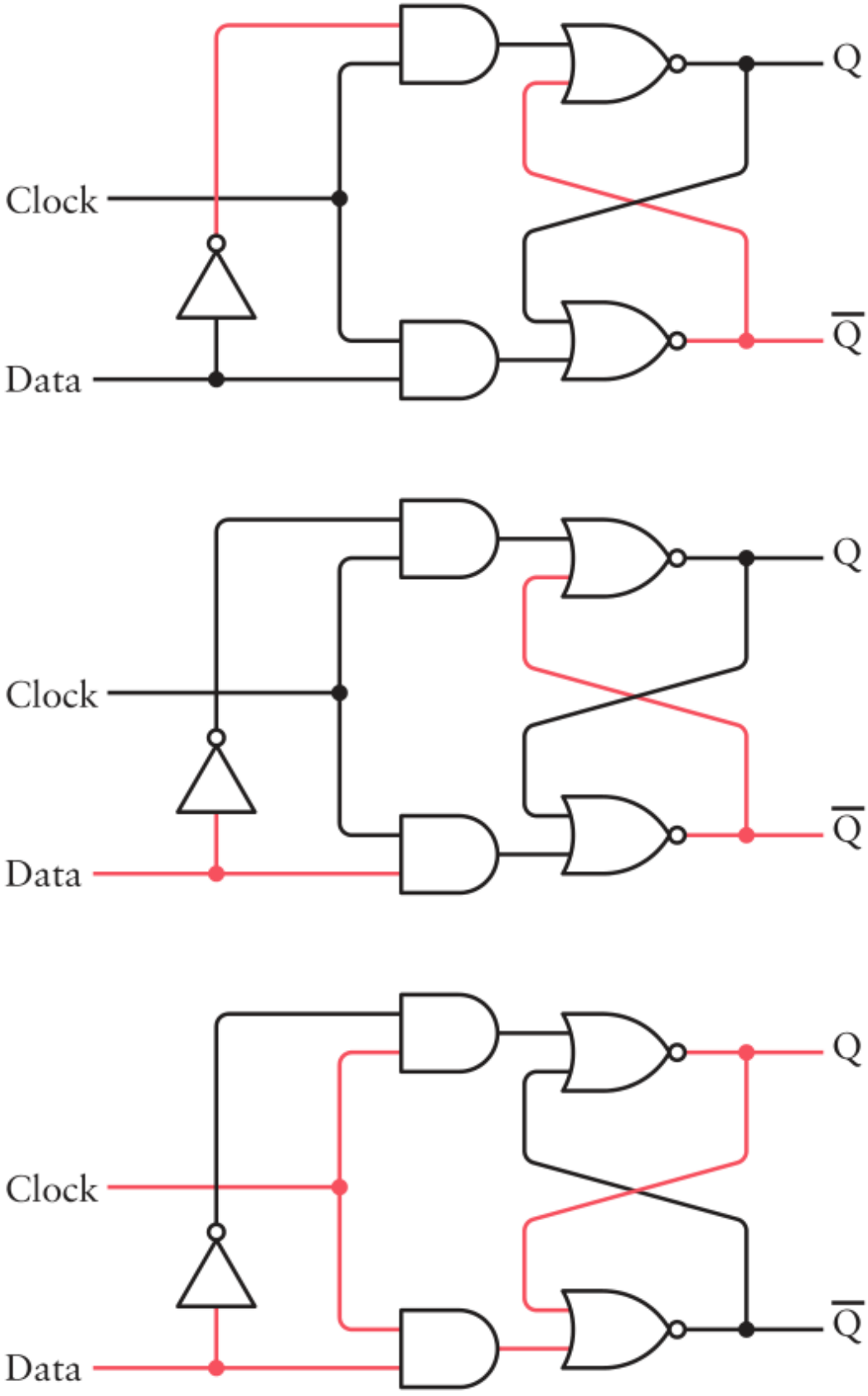
Если вспомнить исходную таблицу R-S-триггера, случай S = R = 1 был запрещён, и его надо избегать. Кроме того, устанавливать оба сигнала в 0 имеет смысл лишь тогда, когда выход не должен меняться. В нашей схеме это уже достигается нулём на входе Hold That Bit.

Следовательно, разумно делать Set и Reset противоположными друг другу: если Set равен 1, Reset равен 0; если Set равен 0, Reset равен 1.

Внесём в схему два изменения. Входы Set и Reset можно заменить одним входом Data. Он эквивалентен прежнему Set. Затем добавим инвертор, чтобы получить Reset.

Второе изменение - дадим сигналу Hold That Bit более традиционное имя *Clock*, «тактовый сигнал». Это может показаться немного странным, ведь настоящего тактового генератора здесь нет, но вскоре вы увидите, что иногда вход Clock действительно может тикать туда и обратно между 0 и 1 через регулярные промежутки. Пока же Clock просто указывает, когда нужно сохранить вход Data.

Вот переработанная схема. Вход Data заменяет Set у нижнего вентиля И, а инвертор превращает его в сигнал, заменяющий Reset у верхнего вентиля:



Сначала оба входа равны 0. Выход Q равен 0, а значит, $\overline{Q}$ равен 1. Пока Clock остаётся равным 0, Data никак не влияет на схему.

Но когда Clock становится равным 1, схема отражает значение Data.

Теперь выход Q совпадает с Data, а $\overline{Q}$ ему противоположен. Затем Clock может вернуться к 0. Схема запоминает значение Data в тот последний момент, когда Clock был равен 1, независимо от дальнейших изменений Data. Например, Data может снова перейти к 0, не влияя на выход.

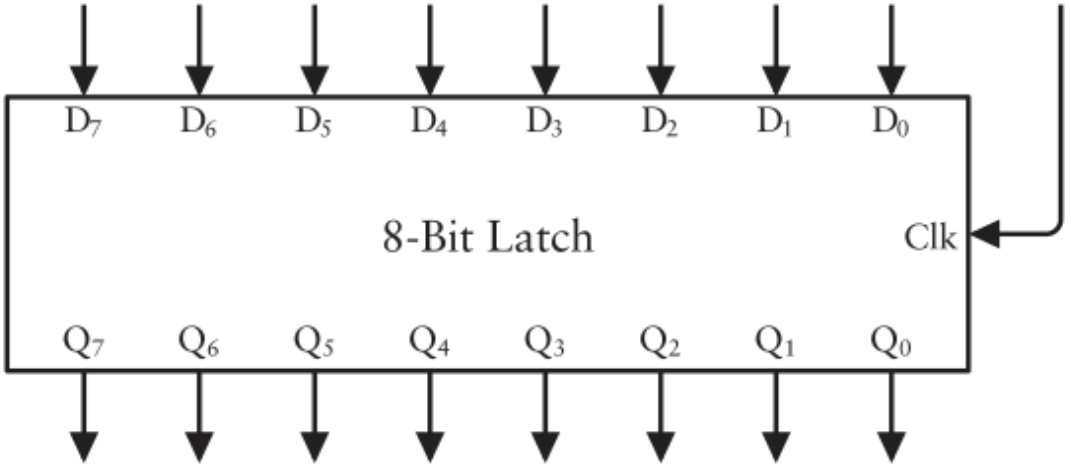
Эта схема называется *D-триггером с управлением уровнем*. Буква D означает Data. Выражение «с управлением уровнем» означает, что триггер сохраняет значение входа Data, когда вход Clock находится на определённом *уровне*, в данном случае 1. Позже мы рассмотрим другой вариант.

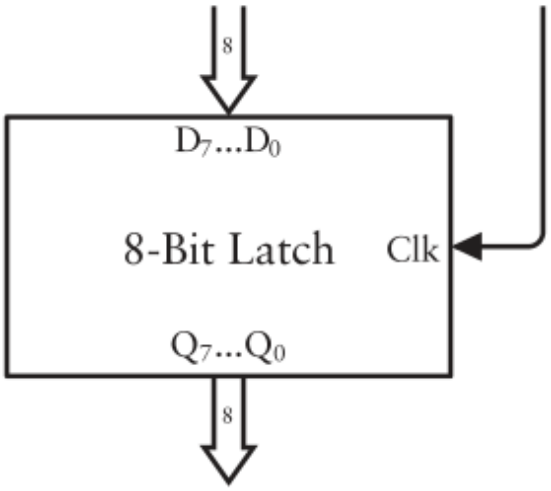
В таблице функций Data можно сократить до D, а Clock - до Clk:

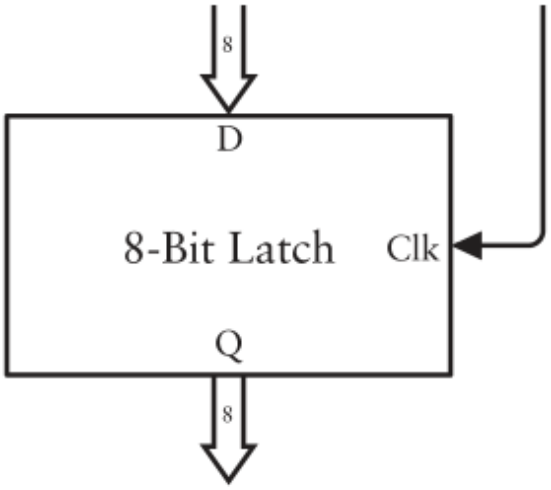
D	**Clk**	**Q**	**$\overline{Q}$**
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

Эта схема также называется *D-защёлкой с управлением уровнем*. Защёлка цепляется за один бит данных и удерживает его для дальнейшего использования. Схему можно также назвать однобитной памятью. В главе 19 я покажу, как множество таких триггеров соединяются и дают много битов и байтов памяти.

Пока попробуем сохранить всего один байт данных. Можно собрать восемь D-триггеров с управлением уровнем и объединить все их входы Clock в один сигнал. Получится такое устройство:







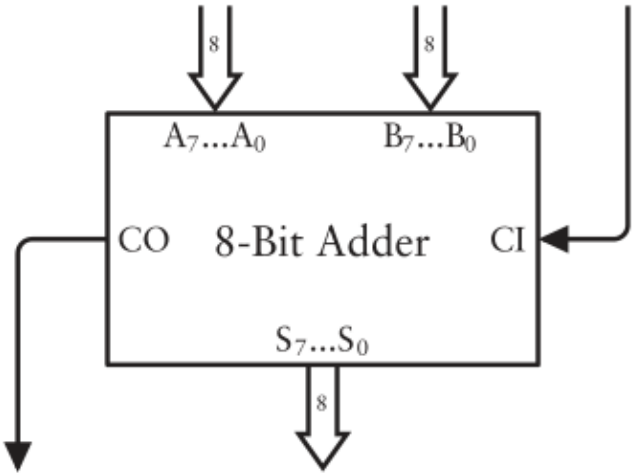
Эта защёлка способна сохранять целый байт сразу. Восемь входов сверху обозначены D_0-D_7, восемь выходов снизу - Q_0-Q_7. Вход справа называется Clock. Обычно Clock равен 0. Когда Clock становится равным 1, восьмибитное значение на входах D переносится на выходы Q. Когда Clock возвращается к 0, восьмибитное значение остаётся там до следующего перехода Clock в 1. Выходы $\overline{Q}$ каждой защёлки игнорируются.

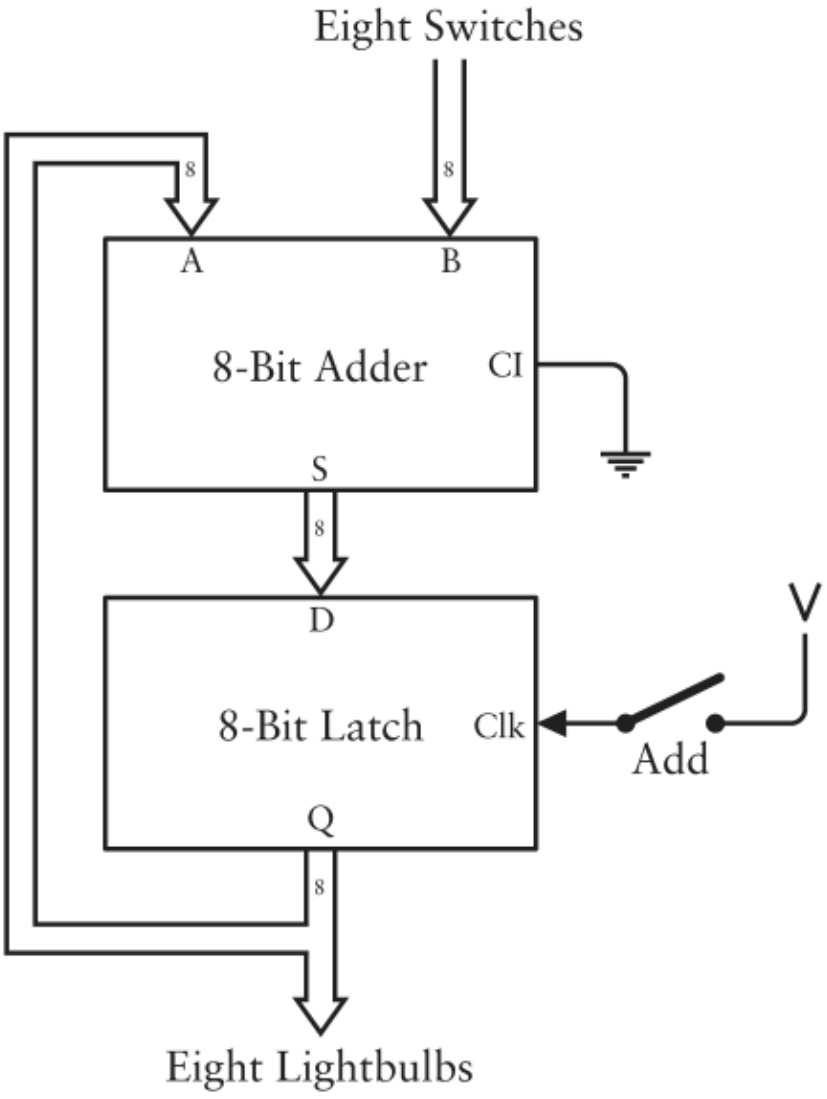
Восьмибитную защёлку можно изобразить и как путь данных, объединив восемь входов Data и восемь выходов Q. Схему можно упростить ещё сильнее, подписав входы просто D и Q.

Ближе к концу главы 14 восемь одноразрядных сумматоров были соединены, чтобы складывать целые байты:

В той главе восемь входов A и восемь входов B соединялись с выключателями, вход CI (Carry In, вход переноса) - с землёй, а восемь выходов S (Sum, сумма) и выход CO (Carry Out, выход переноса) - с лампочками.

Защёлку и сумматор можно использовать как модульные строительные блоки в более сложных схемах. Например, выход одного ряда из восьми выключателей можно сохранить в восьмибитной защёлке, чтобы выход защёлки стал одним из входов сумматора. Затем можно соединить выход сумматора с входом защёлки. Получится устройство, которое можно назвать *накапливающим сумматором*: оно хранит текущую сумму нескольких чисел.





Обратите внимание: выключатель Add управляет входом Clock защёлки.

Помимо сокращения числа выключателей вдвое, такая конфигурация позволяет складывать не только два числа, не вводя заново промежуточные результаты. Сначала выход защёлки состоит из

нулей, и это также вход A сумматора. На первом ряду выключателей устанавливают число и переключают Add: замыкают выключатель, а затем размыкают. Это число сохраняется защёлкой и появляется на лампочках. Затем на выключателях набирают второе число и снова переключают Add. Число на выключателях прибавляется к прежнему итогу, а новый итог появляется на лампочках. Можно продолжать вводить числа и переключать Add.

К сожалению, схема работает не совсем так, как хотелось бы. Она могла бы работать, если бы сумматор был построен из медленных реле и выключатель Add удавалось переключить очень быстро, чтобы сохранить результат сумматора в защёлке. Но пока Add замкнут, любое изменение входов Data защёлки немедленно проходит к выходам Q, оттуда возвращается к сумматору, где значение снова прибавляется к состоянию выключателей, затем опять попадает в защёлку - и так по кругу.

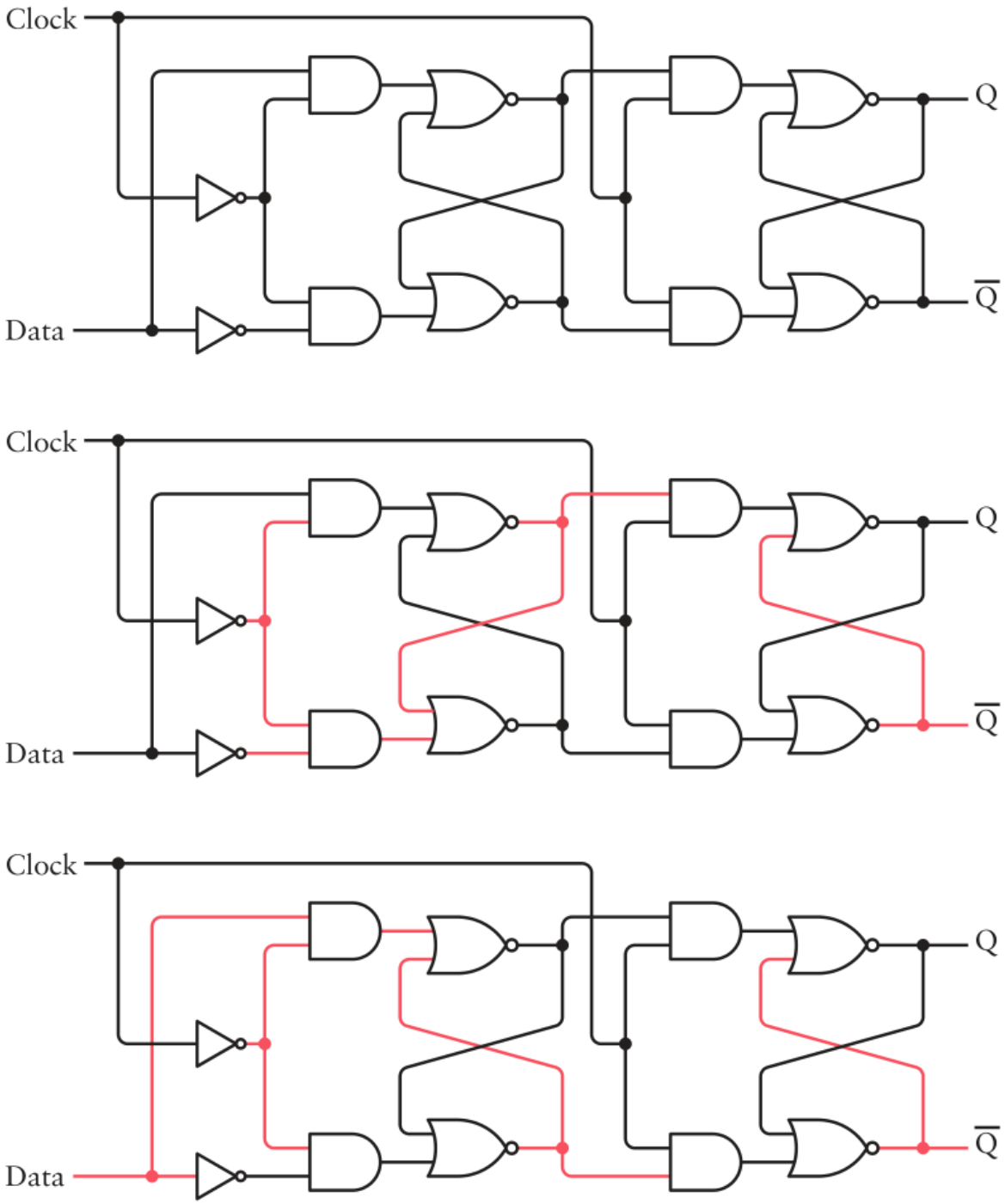
Возникает так называемый *бесконечный цикл*. Причина в том, что применённый D-триггер управляется уровнем. Чтобы значение Data сохранилось, Clock должен изменить свой *уровень* с 0 на 1. Но всё время, пока Clock равен 1, Data может изменяться, и эти изменения отражаются на выходах.

Для некоторых задач D-триггера с управлением уровнем вполне достаточно. Но для накапливающего сумматора он не годится. Нужна защёлка, разрешающая данным пройти только тогда, когда Clock меняется с 0 на 1, либо, как возможный вариант, с 1 на 0. Такой переход называется *фронтом*, потому что графически выглядит как резкая кромка.

Переход 0 -> 1 иногда называют *положительным переходом*, или *положительным фронтом*, а переход 1 -> 0 - *отрицательным переходом*, или *отрицательным фронтом*.

Рассмотренный D-триггер с управлением уровнем фиксирует данные, пока Clock равен 1. В отличие от него, *D-триггер с запуском по положительному фронту* фиксирует данные лишь тогда, когда Clock совершает переход с 0 на 1. Как и у триггера с управлением уровнем, при Clock = 0 изменения Data не действуют на выходы. Но у триггера с запуском по положительному фронту изменения Data не действуют на выход и при Clock = 1. Data влияет на выходы только в самое мгновение перехода Clock с 0 на 1.

Поначалу может показаться, что реализовать такую идею трудно. Но есть хитрость: D-триггер с запуском по фронту строится из двух ступеней D-триггеров с управлением уровнем, соединённых так:



Вход Clock управляет и первой, и второй ступенью, но на первую он поступает через инвертор. Поэтому первая ступень действует точно как D-триггер, только данные сохраняются в ней при Clock = 0. Выходы первой ступени служат входами второй, а там сохраняются при Clock = 1. В итоге Data сохраняется только тогда, когда Clock меняется с 0 на 1.

Рассмотрим подробнее. В исходном состоянии Data и Clock равны 0, а выход Q равен 0. Затем Data становится равным 1, пока Clock по-прежнему равен 0. Из-за инвертированного входа Clock первая ступень меняется.

Вторая ступень остаётся неизменной, поскольку её неинвертированный вход Clock равен 0. Теперь изменим Clock на 1. Это заставляет измениться вторую ступень, и выход Q становится равным 1.

Отличие в том, что теперь вход Data может измениться, например вернуться к 0, не влияя на выход Q. Выходы Q и $\overline{Q}$ способны измениться только в тот момент, когда Clock переходит с 0 на 1.

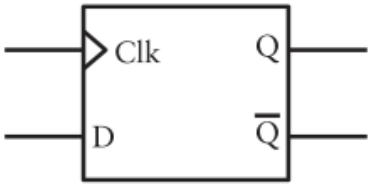
Таблица функций D-триггера с запуском по фронту требует нового обозначения - стрелки вверх (↑). Она обозначает сигнал, совершающий переход с 0 на 1:

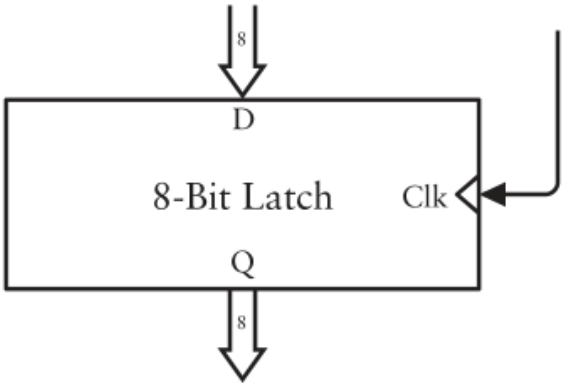
D	**Clk**	**Q**	**$\overline{Q}$**
0	↑	0	1
1	↑	1	0
X	0	Q	$\overline{Q}$

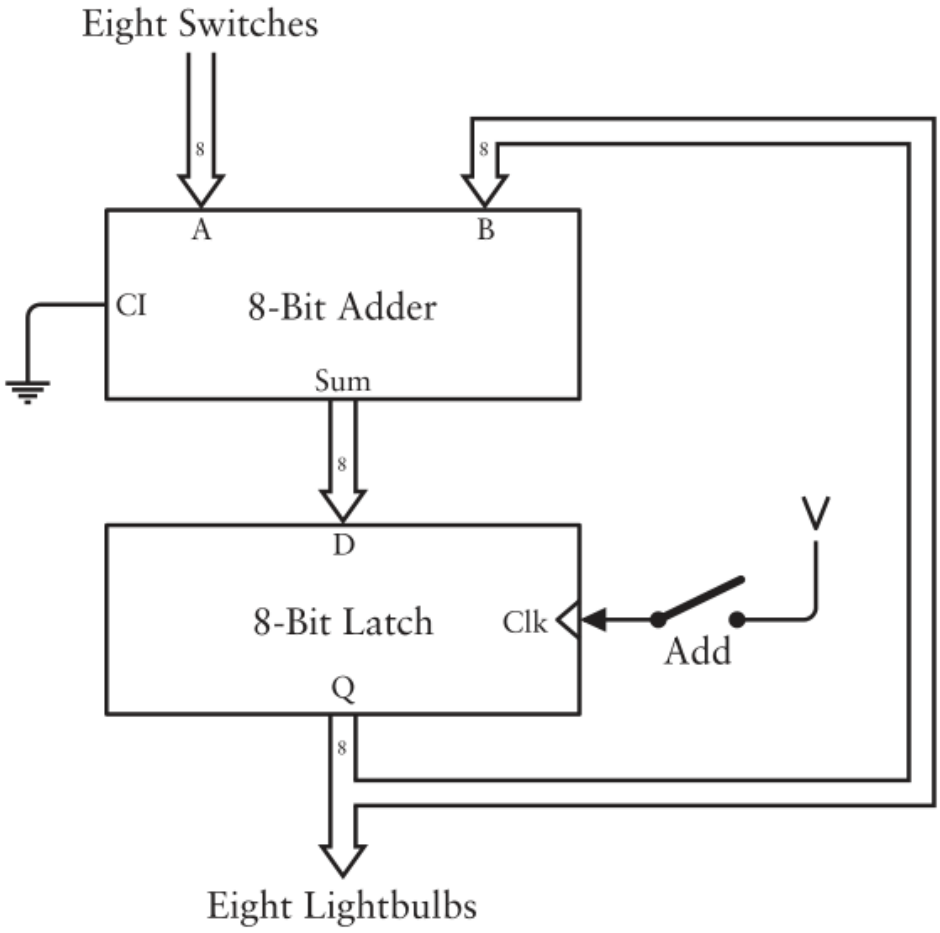
Стрелка вверх указывает, что выход Q становится равным Data, когда Clock совершает положительный переход, то есть переход с 0 на 1. Схематическое обозначение триггера выглядит так:

Маленький угловой знак на входе Clk показывает, что триггер запускается по фронту. Подобным образом новая сборка из восьми триггеров с запуском по фронту обозначается маленьким углом на входе Clock.

Такая защёлка с запуском по фронту идеально подходит для накапливающего сумматора:



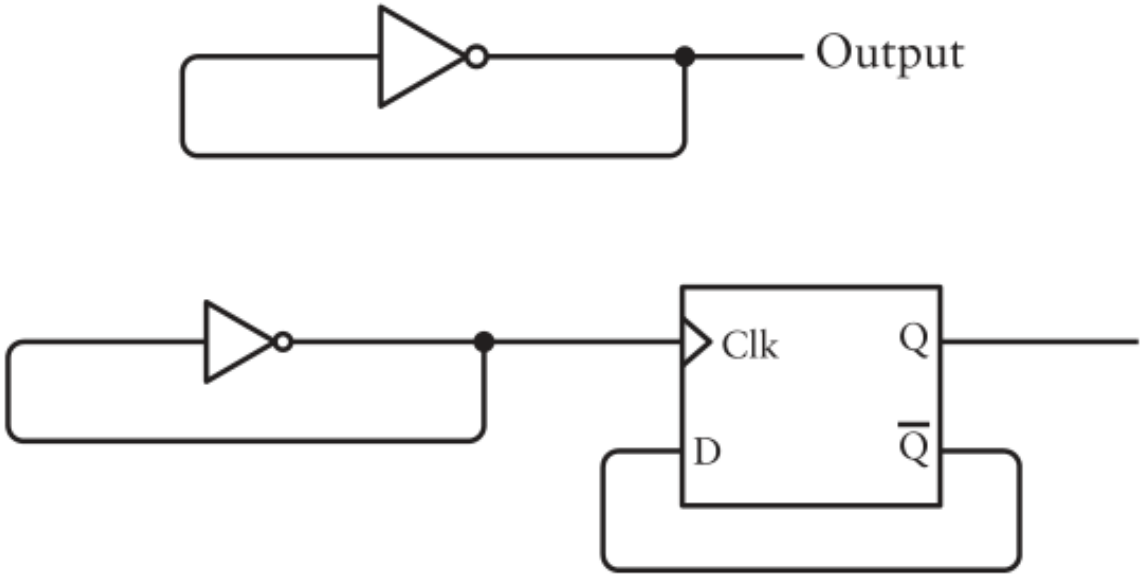




Этот накапливающий сумматор не слишком хорошо обращается с сигналом Carry Out. Если сумма двух чисел превышает 255, выходной перенос просто игнорируется, и лампочки показывают сумму, меньшую, чем следовало бы. Возможное решение - сделать сумматор и защёлку девятиразрядными или по крайней мере более широкими, чем наибольшая ожидаемая сумма. Но отложим решение этой задачи.

Другая проблема: нельзя очистить сумматор, чтобы начать новый текущий итог. Однако для этого есть косвенный способ. В предыдущей главе вы узнали о дополнении до единицы и дополнении до двух, и здесь можно воспользоваться этими понятиями. Если текущий итог равен, например, 10110001, установите на выключателях его дополнение до единицы, 01001110, и сложите. Итог станет 11111111. Затем установите только 00000001 и снова сложите. Все восемь лампочек погаснут, и сумматор будет очищен.

Теперь исследуем ещё одну схему с D-триггером, запускаемым по фронту. Вспомним генератор, построенный в начале главы. Его выход чередуется между 0 и 1. Соединим выход генератора со входом Clock D-триггера, а выход $\overline{Q}$ - со входом D:



Выход триггера сам является его входом. Обратная связь поверх обратной связи! На практике это могло бы стать проблемой. Генератор, собранный из реле или другого переключающего элемента, меняется с предельно возможной для него скоростью. Его выход подключён к элементам триггера, а они могут не успевать за генератором. Чтобы избежать такой трудности, предположим, что генератор гораздо медленнее применённых здесь триггеров.

Чтобы увидеть происходящее, составим таблицу функций, показывающую последовательные изменения. Выполним небольшую хитрость и проследим всё шаг за шагом.

Начнём с Clock = 0 и Q = 0. Тогда $\overline{Q}$ равен 1 и соединён со входом D:

D	Clk	Q	$\overline{Q}$
1	0	0	1

Когда Clock меняется с 0 на 1, выход Q становится равен входу D:

D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0

Теперь Clock равен 1. Но поскольку $\overline{Q}$ стал равен 0, вход D также изменится на 0:

D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0

Clock возвращается к 0, не влияя на выходы:

D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0

Теперь Clock снова становится равным 1. Поскольку D равен 0, Q становится равным 0, а $\overline{Q}$ - 1:

D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

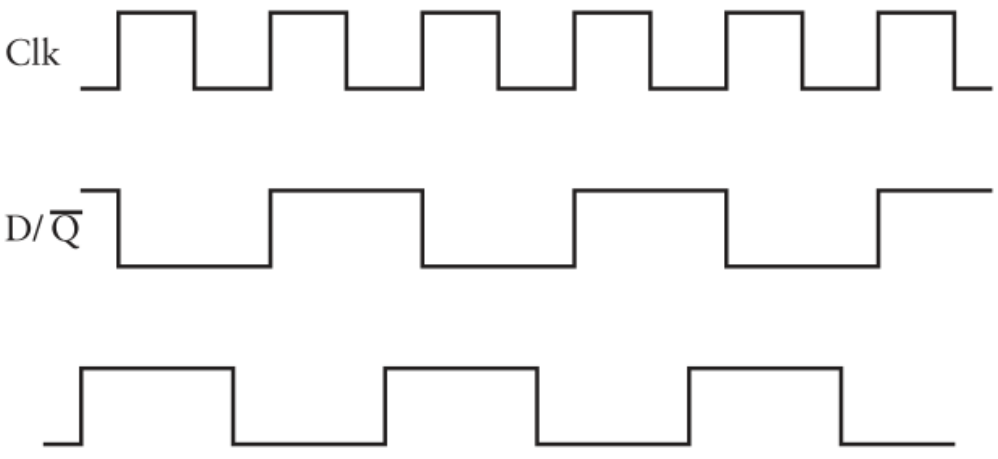
Тогда вход D также становится равным 1:

D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1

Происходящее можно сформулировать очень просто: при каждом переходе Clock с 0 на 1 выход Q меняется либо с 0 на 1, либо с 1 на 0. Это ещё яснее на временной диаграмме:

Inputs		Outputs	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

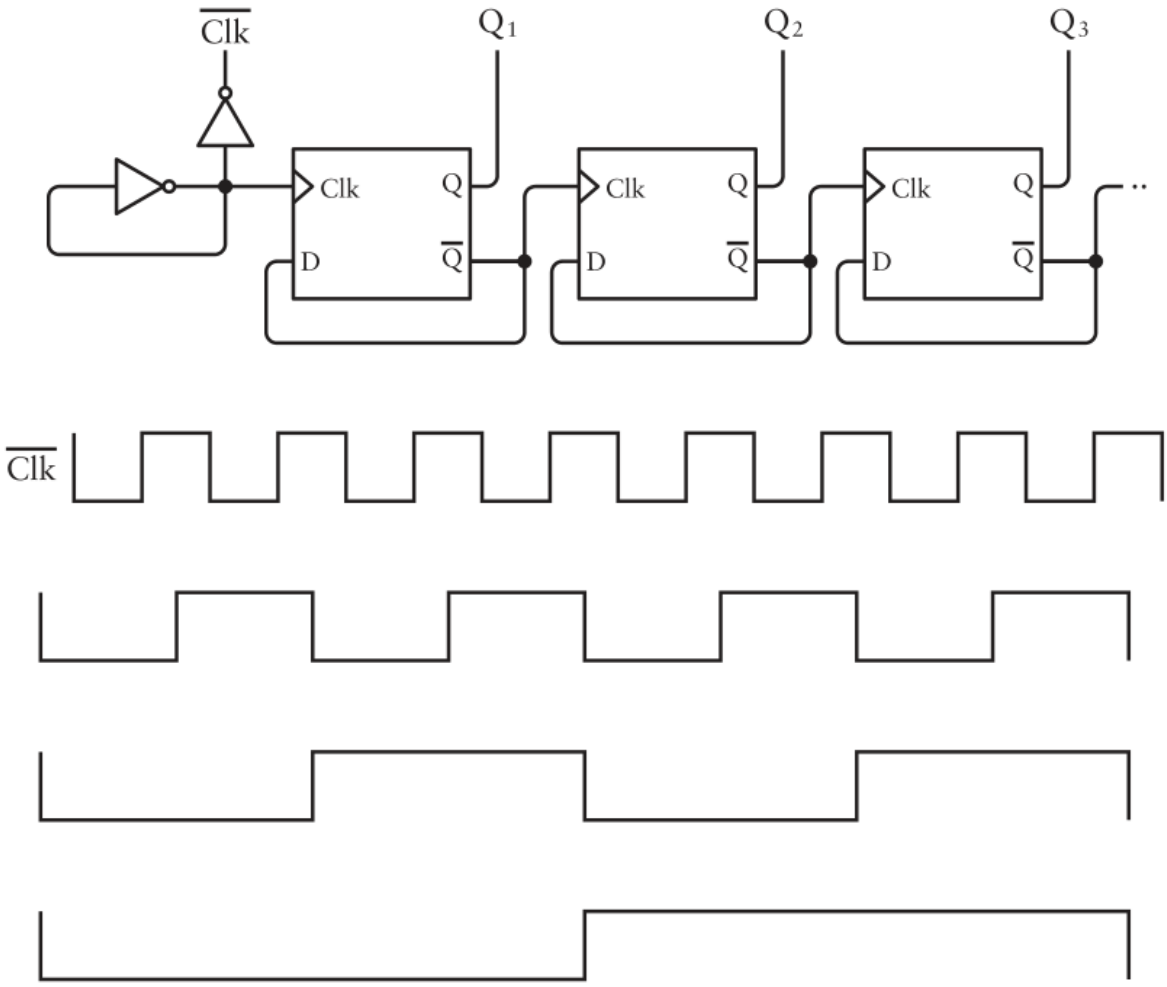
Inputs		Outputs	
D	Clk	Q	$\overline{Q}$
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1
1	1	0	1



Когда Clock переходит с 0 на 1, значение D, совпадающее с $\overline{Q}$, передаётся в Q, тем самым меняя Q и D для следующего перехода Clock с 0 на 1.

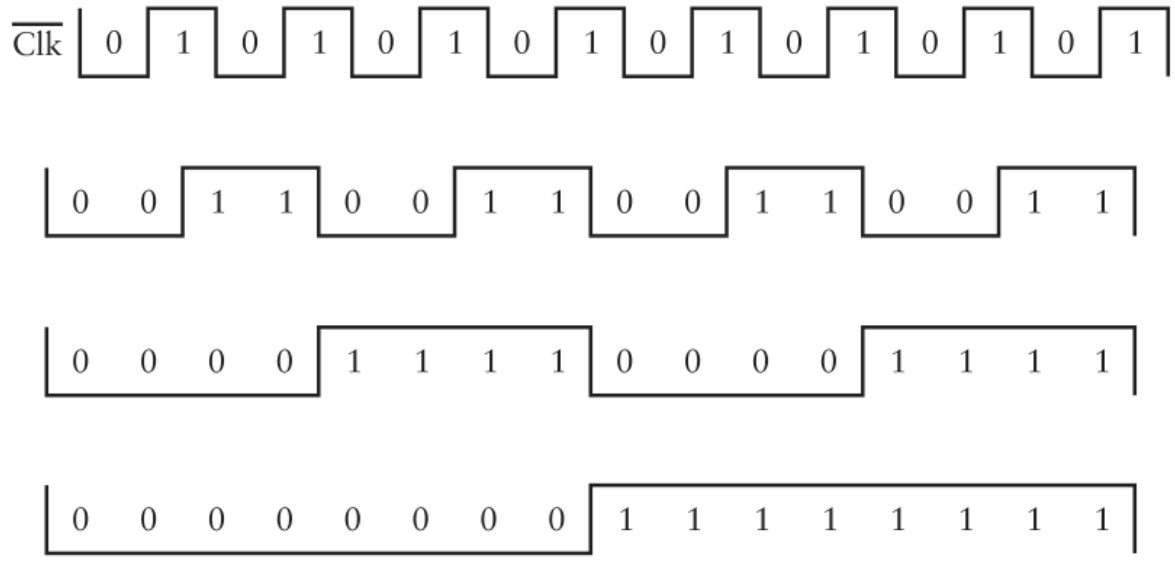
Ранее я говорил, что скорость, с которой сигнал колеблется между 0 и 1, называется частотой и измеряется в герцах, сокращённо Гц, то есть в циклах в секунду. Если генератор имеет частоту 20 Гц, частота выхода Q вдвое меньше, то есть 10 Гц. Поэтому схема, в которой выход Q возвращается на вход D триггера, называется *делителем частоты*.

Разумеется, выход одного делителя частоты можно подать на вход Clock другого, чтобы снова разделить частоту пополам. Вот ряд всего из трёх каскадно соединённых триггеров, хотя его можно продолжать сколько угодно:



Рассмотрим четыре сигнала, обозначенные сверху схемы. Я признаю, что начал и закончил диаграмму в удобном месте, но в этом нет обмана: схема снова и снова повторяет этот рисунок. Не узнаёте ли вы в нём что-нибудь знакомое?

Вот подсказка. Обозначим сигналы нулями и единицами:

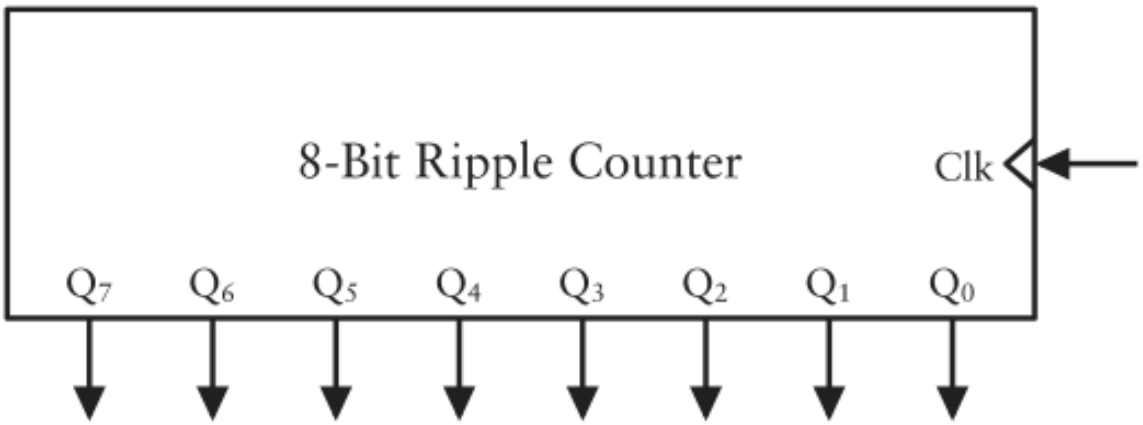


Теперь видно? Попробуйте мысленно повернуть диаграмму на 90 градусов по часовой стрелке и читать четырёхбитные числа поперёк. Каждому соответствует десятичное число от 0 до 15:

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

Значит, эта схема занимается не чем иным, как *счётом в двоичных числах*. Чем больше триггеров мы добавим, тем выше она сможет считать. В главе 10 я отмечал, что в последовательности возрастающих двоичных чисел каждый столбец цифр чередуется между 0 и 1 с частотой вдвое меньшей, чем столбец справа. Здесь схема воспроизводит это. При каждом положительном переходе Clock выходы счётчика *увеличиваются на единицу*.

Соединим восемь триггеров и заключим их в прямоугольник:



Это называется *пульсационным счётчиком*, потому что выход каждого триггера становится входом Clock следующего. Изменения последовательно распространяются по ступеням, словно рябь, и крайние триггеры могут переключаться с небольшой задержкой. Более сложные счётчики бывают *синхронными*: все их выходы меняются одновременно.

Я обозначил выходы Q_0-Q_7 по их положению в цепочке. Q_0, выход первого триггера, находится справа. Если подключить к этим выходам лампочки, можно считывать восьмибитное число.

Ранее я упоминал, что мы найдём способ определять частоту генератора. Вот он. Если подключить генератор ко входу Clock восьмиразрядного счётчика, счётчик покажет, сколько циклов прошёл генератор. Когда итог достигает 11111111, то есть 255 в десятичной системе, он возвращается к 00000000. Это иногда называется *переворотом*, или *циклическим переходом*.

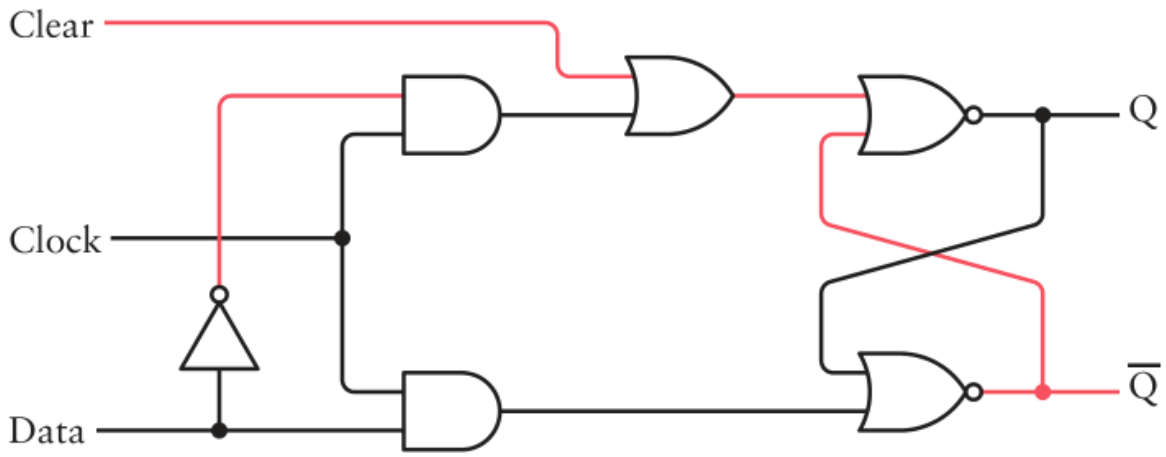
Вероятно, проще всего измерить частоту генератора так: подключить восемь лампочек к выходам восьмиразрядного счётчика, дождаться, пока все выходы станут 0 и ни одна лампочка не будет гореть, и запустить секундомер. Остановить его, когда все лампочки снова погаснут. Это время 256 циклов генератора. Допустим, оно равно 10 секундам. Тогда частота генератора составляет 256 ÷ 10, или 25,6 Гц.

В реальности генераторы на вибрирующих кристаллах гораздо быстрее: от нижней границы около 32 000 Гц, или 32 килогерц (кГц), до миллиона циклов в секунду, то есть мегагерца (МГц), и выше, доходя даже до миллиарда циклов в секунду, то есть гигагерца (ГГц).

Один распространённый кварцевый генератор имеет частоту 32 768 Гц. Это число выбрано не случайно. После последовательности делителей частоты получаются 16 384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2 и 1 Гц. В этот момент он способен считать секунды в цифровых часах.

Практическая проблема пульсационных счётчиков состоит в том, что они не всегда начинают с нуля. При включении питания выход Q отдельных триггеров может оказаться равен 1 или 0. Обычное усовершенствование - вход Clear, устанавливающий Q в 0 независимо от входов Clock и Data.

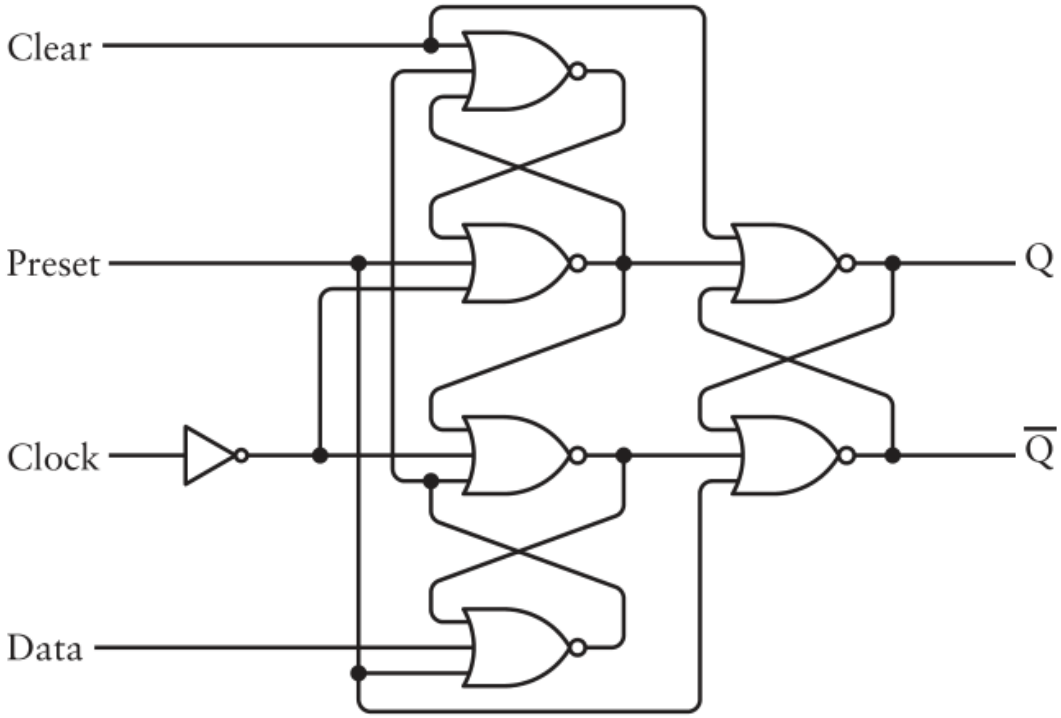
Для простого D-триггера с управлением уровнем добавить Clear довольно легко: нужен лишь дополнительный вентиль ИЛИ. Обычно Clear равен 0, но когда он становится равен 1, выход Q становится равен 0:



Этот сигнал принудительно устанавливает Q в 0 независимо от других входов, фактически очищая триггер.

Для триггера с запуском по фронту сигнал Clear сложнее. Если мы добавляем Clear, то стоит рассмотреть и вход Preset, «предварительная установка». Clear устанавливает выход Q в 0 независимо от Clock и Data, а Preset устанавливает Q в 1. При создании цифровых часов сигналы Clear и Preset пригодились бы, чтобы установить начальное время.

Вот *D-триггер с запуском по фронту, предварительной установкой и очисткой*, построенный целиком из шести трёхвходовых вентилей ИЛИ-НЕ и одного инвертора. Недостаток простоты он восполняет симметрией:



Входы Preset и Clear имеют преимущество перед Clock и Data. Обычно Preset и Clear равны 0. Когда Preset равен 1, Q становится равным 1, а $\overline{Q}$ - 0. Когда Clear равен 1, Q становится равным 0, а $\overline{Q}$ - 1. Как и входы Set и Reset у R-S-триггера, Preset и Clear не должны одновременно быть равны 1.

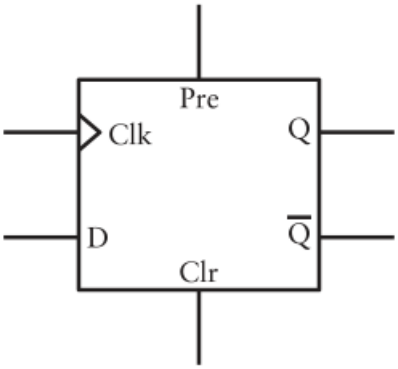
Во всех остальных случаях схема ведёт себя как обычный D-триггер с запуском по фронту:

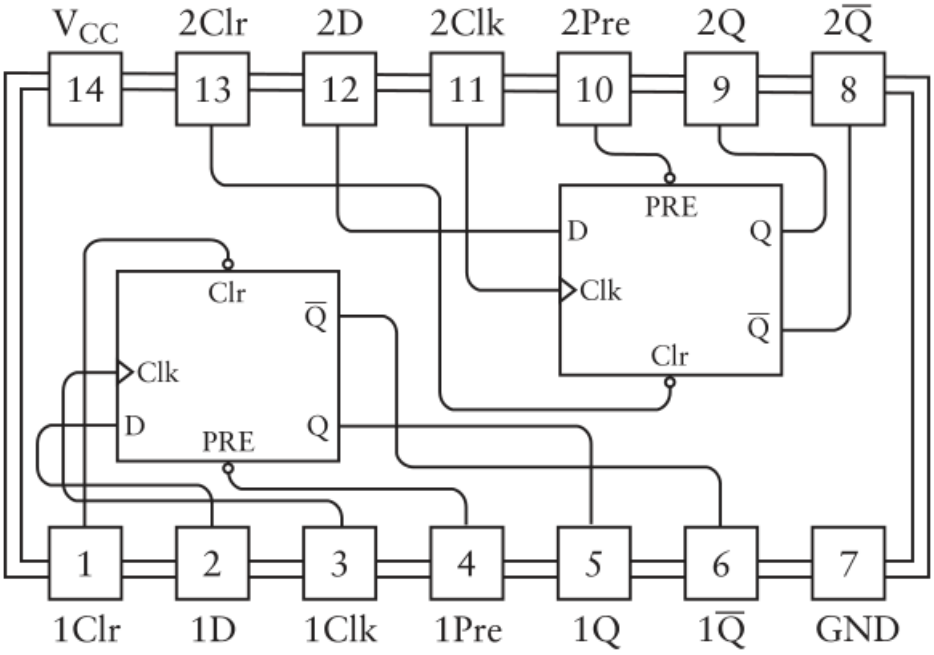
Pre	**Clr**	**D**	**Clk**	**Q**	**$\overline{Q}$**
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	$\overline{Q}$

Схематическое обозначение D-триггера с запуском по фронту, предварительной установкой и очисткой имеет отдельные входы Pre и Clr.

В главе 15 я описал несколько примеров интегральных схем семейства TTL, то есть транзисторно-транзисторной логики. При работе с TTL, когда нужен один из таких триггеров, собирать его из вентилей не требуется. Микросхема 7474 описывается как «сдвоенный D-триггер с запуском по положительному фронту, предварительной установкой и очисткой»; вот как она показана в *TTL Data Book for Design Engineers*:

Inputs				Outputs	
Pre	Clr	D	Clk	Q	$\overline{Q}$
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	$\overline{Q}$





Итак, мы убедили телеграфные реле и транзисторы складывать, вычитать и считать двоичные числа. Мы также увидели, как триггеры способны сохранять биты и байты. Это первый шаг к построению важнейшего компонента компьютеров, известного как *память*.

Но сначала немного развлечёмся.

Глава 18. Давайте построим часы!

Какой увлекательный проект - построить часы! Представьте большие старомодные напольные часы с затейливо вырезанным деревянным корпусом и стеклянной дверцей, сквозь которую виден тяжёлый качающийся маятник. За украшенным металлическим циферблатом находится сложная система шестерёнок, отсчитывающая время с помощью хитроумного механизма, называемого спуском. Его тиканье разносится по дому и каждый час вызывает торжественный бой.

Но нет. Мы будем строить совсем не такие часы. Часы этой главы будут цифровыми: часы, минуты и секунды на них показываются числами, а не вращающимися стрелками. Более того, первая версия даже не будет показывать привычные десятичные цифры, а станет отображать время в двоичном виде мигающими лампочками.

Знаю, знаю: двоичное представление времени кажется ужасным! Но это необходимый первый шаг к привычным десятичным цифрам. При этом двоичные числа, которыми мы воспользуемся, в действительности представляют собой нечто среднее между чисто двоичной и десятичной записью.

Сначала рассмотрим числа, из которых складывается время. Если показывать и секунды, и минуты, требуются шесть десятичных цифр, например:

```
12:30:47
```

Это 12 часов 30 минут 47 секунд, то есть примерно половина первого ночи или дня. Индикатор AM или PM устранил бы неоднозначность.

В двоичном виде это время можно было бы представить двоичными эквивалентами чисел 12, 30 и 47:

```
1100 : 11110 : 101111
```

Не знаю, как вы, а я не хотел бы видеть время в таком виде. Пока я закончу переводить эти двоичные числа в десятичные, часы вполне могут уйти на минуту вперёд.

Поэтому поступим иначе. Представим каждую десятичную цифру отдельно в двоичном виде. Тогда время 12:30:47 будет показано двоичными числами для 1, 2, 3, 0, 4 и 7:

```
0001 0010 : 0011 0000 : 0100 0111
```

Теперь в уме приходится преобразовывать шесть четырёхразрядных двоичных чисел, но все десятичные значения лежат между 0 и 9, поэтому преобразование намного легче. Наблюдая, как на таких часах меняются секунды, можно быстро научиться читать эти числа.

У такого представления есть название: *двоично-десятичный код*, или BCD. В BCD каждая цифра десятичного числа кодируется четырьмя двоичными разрядами:

BCD	Десятичная цифра
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

BCD	Десятичная цифра
1000	8
1001	9

Раньше подобные таблицы продолжались после 1001 числами 1010, 1011, 1100, 1101, 1110 и 1111 - двоичными эквивалентами чисел от 10 до 15. В BCD эти дополнительные сочетания недопустимы. Код заканчивается на 1001; остальные комбинации битов не используются.

Это ещё один пример того, что сами биты ничего о себе не сообщают. Встретив число 10011001 без контекста, нельзя определить его смысл. Как беззнаковое целое оно равно 153. Как знаковое число в дополнительном коде из главы 16 оно равно -103. А как BCD оно равно 99.

Во внутреннем устройстве компьютеров BCD применяется редко, потому что усложняет основные арифметические операции, например сложение и вычитание. Но при выводе десятичных чисел BCD часто служит промежуточной ступенью.

Первые часы этой главы не обязательно сразу покажут правильное время, но секунды будут возрастать от 00 до 59.

Затем минуты будут идти от 00 до 59, а после них - часы. Решение показывать время в BCD означает, что каждую из шести десятичных цифр можно вычислять отдельно, начиная с секунд. Допустимые диапазоны таковы:

• младшая цифра секунд: 0-9;

• старшая цифра секунд: 0-5;

• младшая цифра минут: 0-9;

• старшая цифра минут: 0-5;

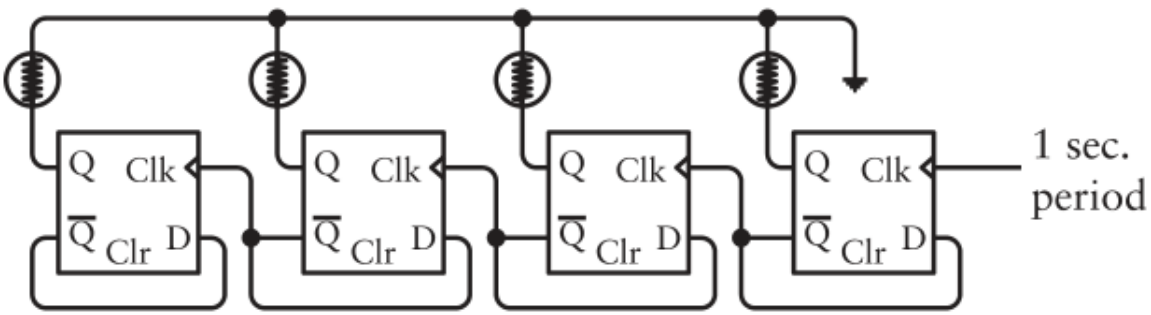
• младшая цифра часов: 0-9;

• старшая цифра часов: только 0 или 1.

Младшая цифра секунд постоянно возрастает от 0 до 9. Достигнув 9, она *переворачивается*, или сбрасывается, в 0, а старшая цифра секунд увеличивается на 1: от 0 к 1, затем к 2, 3, 4 и наконец к 5. После 59 секунд следующим значением становится 00, а минуты увеличиваются на 1.

Для каждой из шести цифр нужна отдельная схема, которая воздействует на следующую.

Начнём с младшей цифры секунд. Можно соединить в ряд четыре триггера с запуском по фронту, как в пульсационном счётчике на странице 235 главы 17. К каждому выходу Q подключается лампочка:



В главе 17 триггеры соединялись слева направо, здесь - справа налево. Скоро станет ясно, что так лампочки показывают удобочитаемые двоичные числа.

Крайний правый вход получает сигнал генератора частотой 1 герц, то есть один цикл в секунду. Период этого генератора - время одного цикла - равен единице, делённой на частоту, то есть 1 секунде. Каждую секунду генератор переходит из 0 в 1 и обратно.

Выходы каждого триггера Q и $\overline{Q}$ противоположны. Выход $\overline{Q}$ соединён с D. При переходе Clock с 0 на 1 значение D становится выходом Q. Когда Q переходит с 0 на 1, он изменяет состояние следующего слева триггера.

У первого, правого, триггера Q одну секунду равен 0, следующую - 1: лампочка секунду не горит, затем секунду горит. Период равен двум секундам, а частота делится пополам.

Второй справа триггер снова делит частоту пополам: его лампочка горит две секунды и не горит две секунды. И так далее. В результате четыре мигающие лампочки считают секунды в двоичном виде:

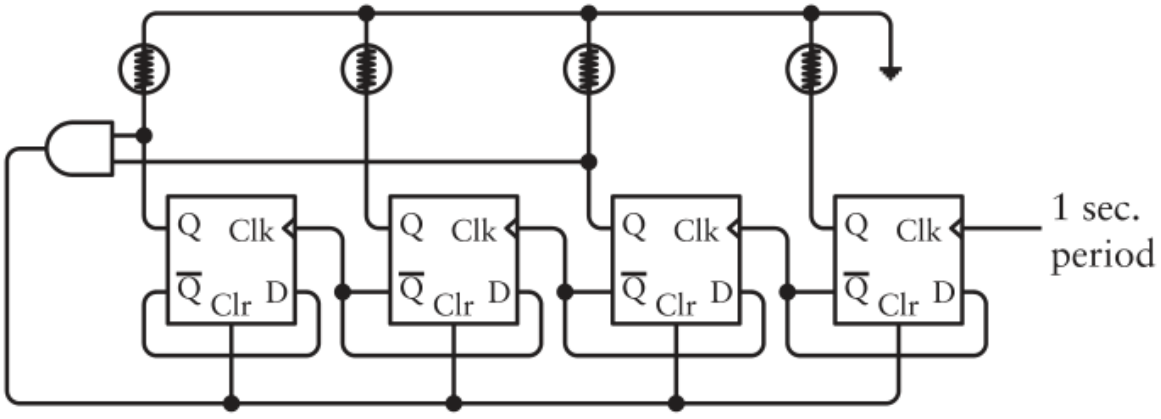
```
0000
0001
0010
0011
0100
...
1110
1111
0000
...
```

Счёт идёт от 0000 до 1111 и возвращается к 0000, завершая цикл каждые 16 секунд.

Но нам нужно не это. Лампочки должны считать от 0000 до 1001, то есть выполнять цикл за 10 секунд. После 1001, десятичной 9, требуется вернуться к 0.

К счастью, у выбранных триггеров есть вход Clear, обозначенный Clr снизу. Если он равен 1, выход Q становится равен 0 независимо от остальных входов. Все входы Clear можно одновременно установить в 1, и показанное число вернётся к 0000.

Когда это надо сделать? Значение 1001, или 9, допустимо, но следующее значение 1010, или 10, уже недопустимо. Поэтому, как только четыре триггера выдадут 1010, нужно очистить их до нуля. Для этого достаточно вентиля И, соединённого с двумя выходами Q:



В реальной схеме транзисторы триггеров настолько быстры, что перехода от 1010 к 0000 видно не будет. Как только выходы Q станут равны 1010, выход вентиля И станет равен 1, триггеры очистятся.

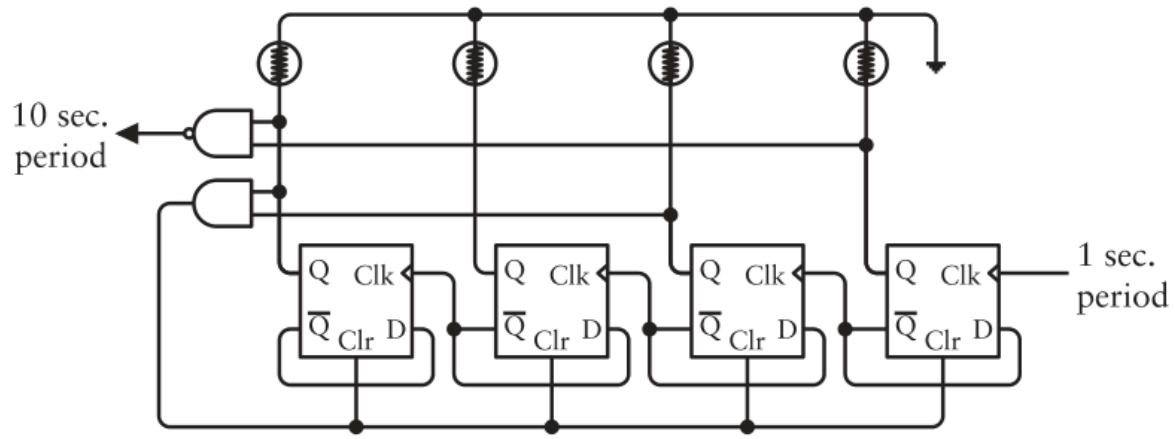
Зрительно получится плавный переход от 1001 к 0000:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
0000
...
```

Если использование выходов триггеров для очистки самих триггеров вызывает у вас лёгкое беспокойство, оно не совсем беспочвенно. Существуют более правильные, но и более сложные способы. В настоящих двоичных часах можно также применить интегральные *декадные счётчики*, которые считают от 0000 до 1001, а затем аккуратно возвращаются к 0000.

Теперь секунды считаются от 0 до 9. Скоро появится схема старшей цифры секунд, считающая от 0 до 5. Когда четыре лампочки младшей цифры переходят от 1001 к 0000, старшая цифра должна увеличиться на 1. Значит, в этот момент нужен сигнал, переходящий с 0 на 1.

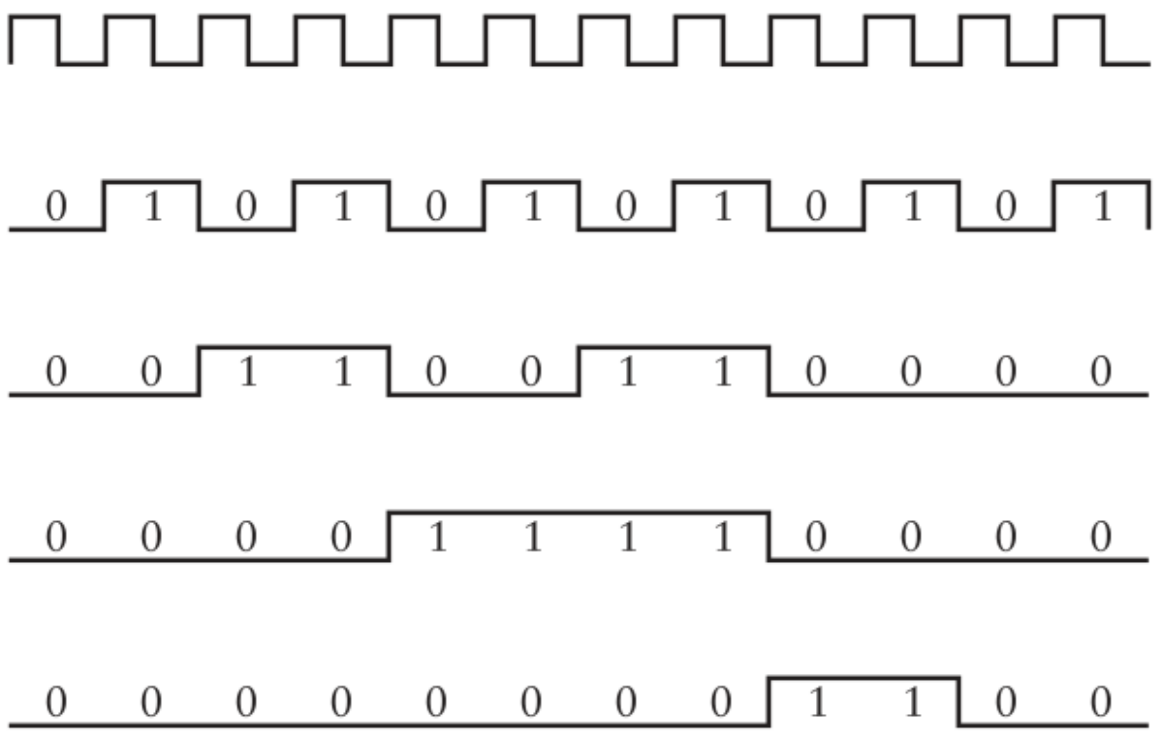
Можно было бы использовать выход вентиля И, но поступим немного иначе и добавим вентиль И-НЕ:



Выход И-НЕ противоположен выходу И. Обычно он равен 1 и становится 0 лишь при двух единицах на входах. Входы этого вентиля подключены так, что оба равны 1 при числе 1001, то есть 9.

При числе 1001 выход И-НЕ становится равен 0, а затем возвращается к 1. Это происходит раз в 10 секунд, и переход 0 -> 1 можно подать на следующий триггер с запуском по фронту.

Временная диаграмма показывает сигнал с периодом 1 секунда, выходы Q четырёх триггеров справа налево и сигнал с периодом 10 секунд:



Если повернуть диаграмму на 90 градусов по часовой стрелке, видно, как четыре триггера считают от 0000 до 1001 и возвращаются к началу.

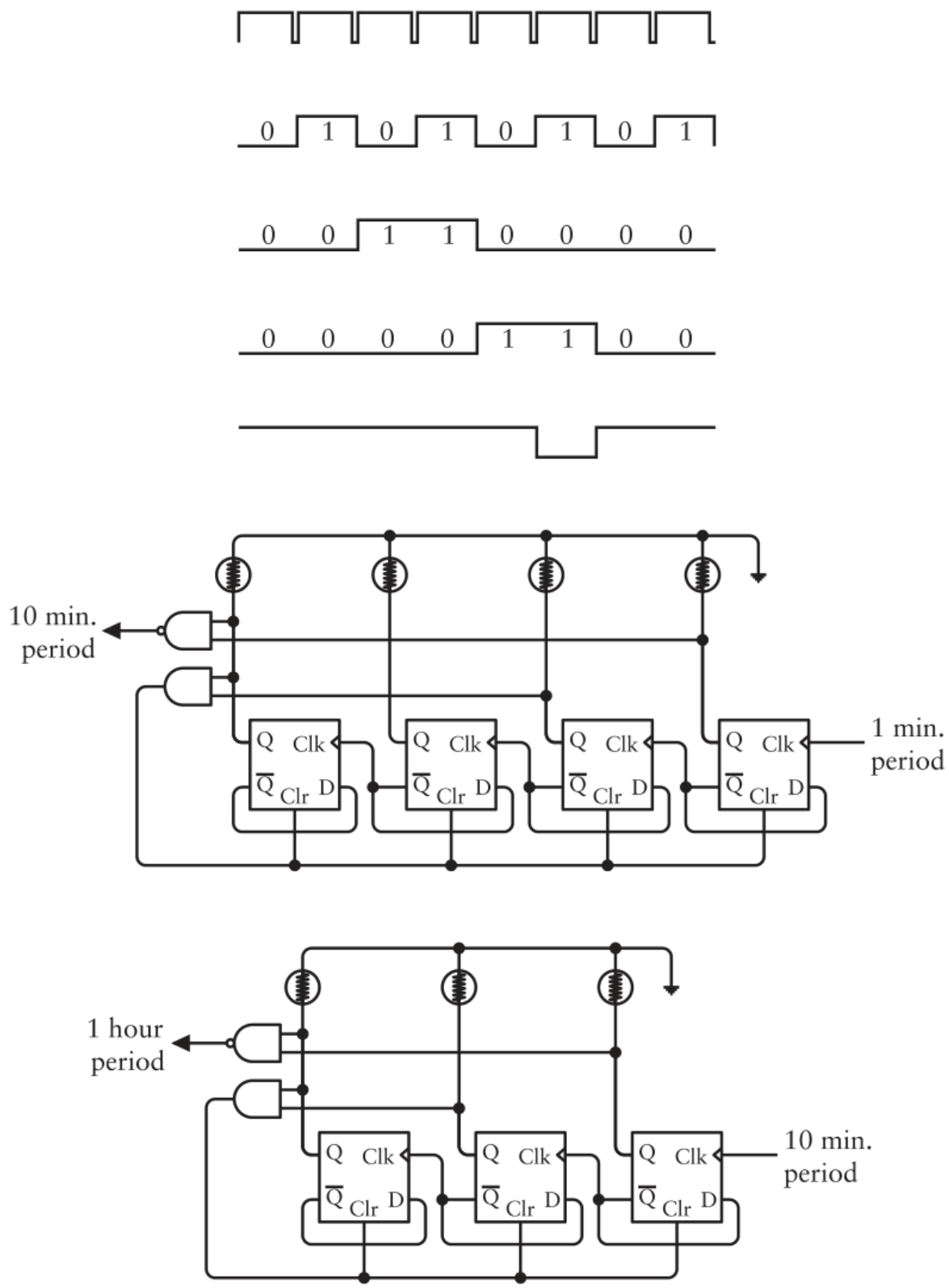
Выход вентиля И-НЕ служит входом следующей ступени часов, считающей старшую цифру секунд: 0, 1, 2, 3, 4 и 5. Для неё достаточно трёх триггеров, но при достижении 110, то есть 6, их нужно очистить:

Выход И-НЕ этой схемы становится равен 0 при числе 101, то есть 5. Вместе с предыдущими четырьмя триггерами часы теперь считают от 000 0000 до 101 1001, то есть до 59, после чего все семь триггеров возвращаются к 0.

И снова, повернув временную диаграмму на 90 градусов по часовой стрелке, можно увидеть счёт от 000 до 101 и возврат к началу.

Теперь имеется сигнал с периодом 1 минута, и можно считать минуты. Ещё четыре триггера соединяются точно как первые четыре, чтобы считать младшую цифру минут от 0 до 9:

Сигнал с периодом 10 минут становится входом ещё одной группы из трёх триггеров для старшей цифры минут, устроенной так же, как старшая цифра секунд:



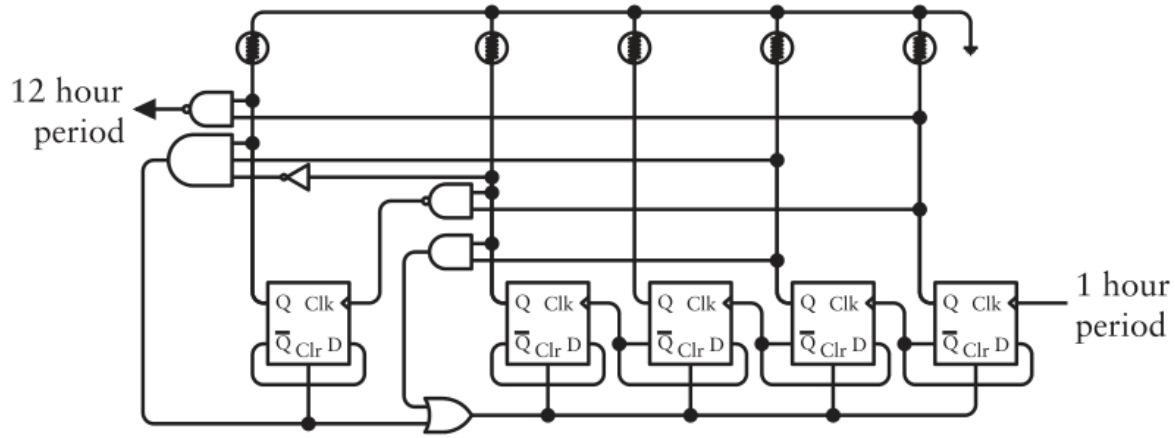
И вот, когда кажется, что построение целых двоичных часов уже выходит на финишную прямую, у вас могут появиться дурные предчувствия.

В 24-часовых часах значения идут от 0 до 23, но в англоязычных странах обычно применяют 12-часовой формат, и он создаёт проблему. Часы, как секунды и минуты, состоят из двух цифр, однако отсчёт начинается не с нуля. По соглашению полночь и полдень обозначаются числом 12, а следующий час - числом 1.

Пока проигнорируем эту проблему. Допустим, что значения идут 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 и снова 0, а 00:00:00 означает полночь или полдень.

Но у часов есть ещё одна особенность. Для секунд и минут очистка младшей и старшей цифр независима: младшая очищается при достижении 1010, старшая - при 110. У часов младшая цифра также должна очищаться при 1010 - это переход от 9:59:59 к 10:00:00. Но обе цифры нужно очистить, когда старшая равна 1, а младшая - 0010. Это переход от 11:59:59 к полуночи или полудню, временно представленным как 00:00:00.

Значит, обе цифры часа нужно рассматривать вместе. Пять триггеров показывают, как очистить их при двух разных условиях:

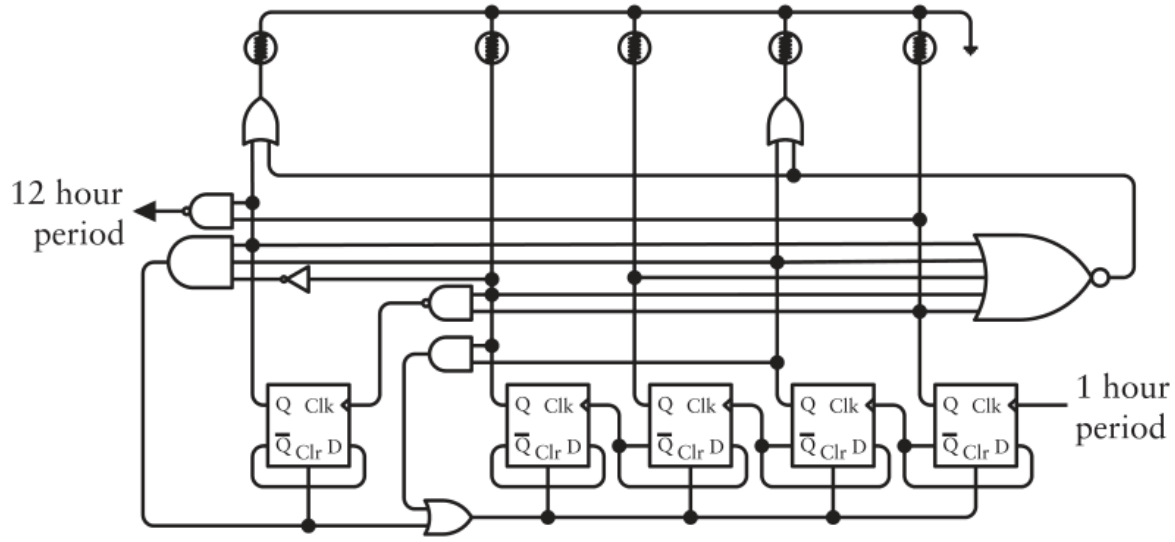


Четыре правых триггера соединены почти как для младших цифр секунд и минут. Вентиль И очищает их при числе 1010, а И-НЕ в тот же момент выдаёт сигнал, переходящий с 0 на 1.

Другой трёхвходовый вентиль И вверху слева определяет, что старшая цифра равна 1, а младшая - 0010, то есть общее BCD-значение равно 12. Можно подумать, что именно 12 и надо показать, но тогда возникнет проблема с переходом к 1. Вместо этого трёхвходовый И очищает все пять триггеров, и отображаемый час становится равен 0.

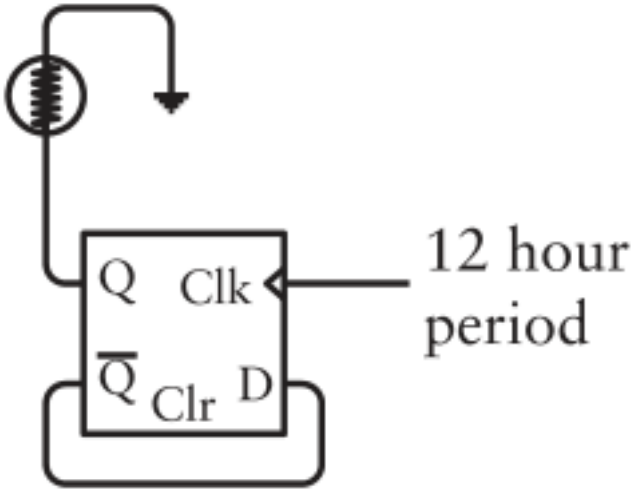
Схема успешно показывает часы от 0 до 11. Остаётся одна проблема: когда все выходы Q равны 0, на индикаторе должно быть не 0, а 12, то есть 1 0010.

Это делается с помощью пятивходового вентиля ИЛИ-НЕ справа:



Выход ИЛИ-НЕ противоположен выходу ИЛИ и равен 1 только тогда, когда все пять входов равны 0. Он подаётся на два вентиля ИЛИ у двух разрядов. Поэтому при выходе триггеров 0 0000 лампочки показывают 1 0010, то есть 12.

О вентиле И-НЕ слева ещё не говорилось. Его выход обычно равен 1, но становится равен 0, когда часы показывают 1 0001, то есть 11. Когда час перестаёт быть одиннадцатым, выход снова переходит в 1. Его можно подать на другой триггер, служащий индикатором AM/PM:



Полные двоичные часы, объединяющие все рассмотренные компоненты, доступны на сайте CodeHiddenLanguage.com.

Как мы видели, каждая из шести цифр использует И-НЕ для формирования тактового сигнала следующей цифры. Выход такого вентиля обычно равен 1, если только оба входа не равны 1. Отсюда возникает особенность при первом запуске часов: тогда каждый И-НЕ имеет выход 1 и запускает Clock первого триггера следующей ступени. Начальное время устанавливается в:

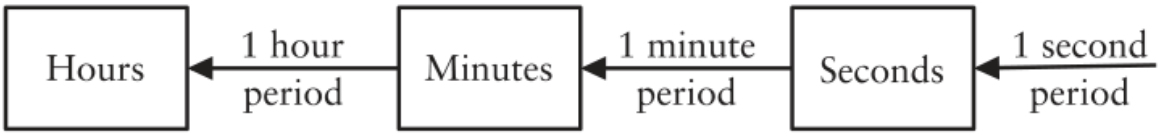
```
1:11:10
```

Это прекрасно, если часы включены ровно в этот момент. В противном случае хотелось бы установить текущее время.

Некоторые цифровые часы получают время из интернета, от спутников GPS или специальных радиосигналов. В моделях с ручной установкой порой встречается множество кнопок, которыми нужно манипулировать в столь сложной последовательности, что без подробной инструкции не обойтись.

Взаимодействие с человеком всегда непросто, поэтому реализуем нечто очень простое.

Секунды, минуты и часы двоичных часов соединены справа налево:



Добавим два выключателя. Первый вручную увеличивает минуты, второй - часы. Решение не идеально: если показано 2:55, а нужно установить 1:50, кнопку минут придётся нажать 55 раз, а кнопку часов - 11 раз. Зато всё предельно просто, и подробная инструкция не нужна.

Минуты обычно увеличиваются, когда секунды достигают 59 и возвращаются к 00. Сигнал «период 1 минута» обычно равен 1, но равен 0, когда старшая цифра секунд равна 5. Аналогично сигнал «период 1 час» обычно равен 1, но равен 0, когда старшая цифра минут равна 5.

При нажатии выключателей эти сигналы надо изменять. Если сигнал «период 1 минута» равен 1, как почти всегда, нажатие должно сделать его равным 0, а отпускание - вернуть к 1. Если же сигнал равен 0, что бывает при секундах от 50 до 59, нажатие должно перевести его в 1, а отпускание - обратно в 0.

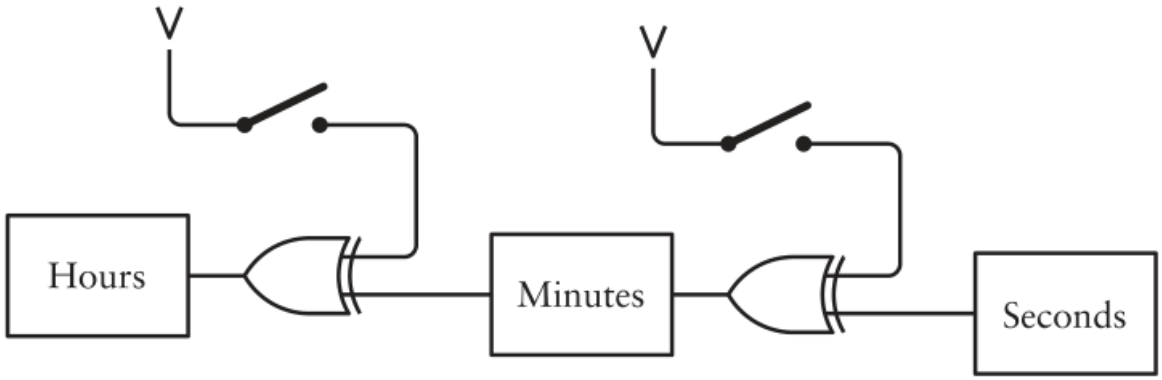
Иначе говоря, выключатели ручной установки должны превращать сигналы «период 1 минута» и «период 1 час» в противоположные.

Это одно из применений исключающего ИЛИ, XOR, уже использованного в главе 14 для сложения. Его выход совпадает с ИЛИ, кроме случая двух единиц:

XOR	**0**	**1**
0	0	1
1	1	0

Помимо важнейшей роли в сложении, XOR способен инвертировать сигнал. Когда один вход равен 0, выход совпадает с другим входом. Когда один вход равен 1, выход противоположен другому входу.

Благодаря XOR выключатели ручной установки добавляются очень просто:



Мигание секунд и минут двоичных часов может действовать гипнотически. Начиная с 1970-х годов двоичные часы с мигающими лампочками производились как необычные сувениры, нередко по цене, совершенно не соответствовавшей простоте схемы. Но для желающих изучать двоичные числа или хотя бы BCD они имеют образовательную ценность.

Для предпочитающих обычные десятичные цифры есть альтернативы. Одна из самых красивых в технологически ретроспективном смысле называется *индикатором с холодным катодом*. Это стеклянная трубка, заполненная главным образом неоном. Внутри находятся наложенные друг на друга проволочки в форме цифр; каждая соединена с одним из десяти выводов снизу, начиная с 0 слева:

Не показан ещё один вывод земли, а также соединённая с ним проволочная сетка, окружающая все проволочные цифры.

Если подать напряжение на один из выводов, неон вокруг соответствующей цифры начинает светиться.

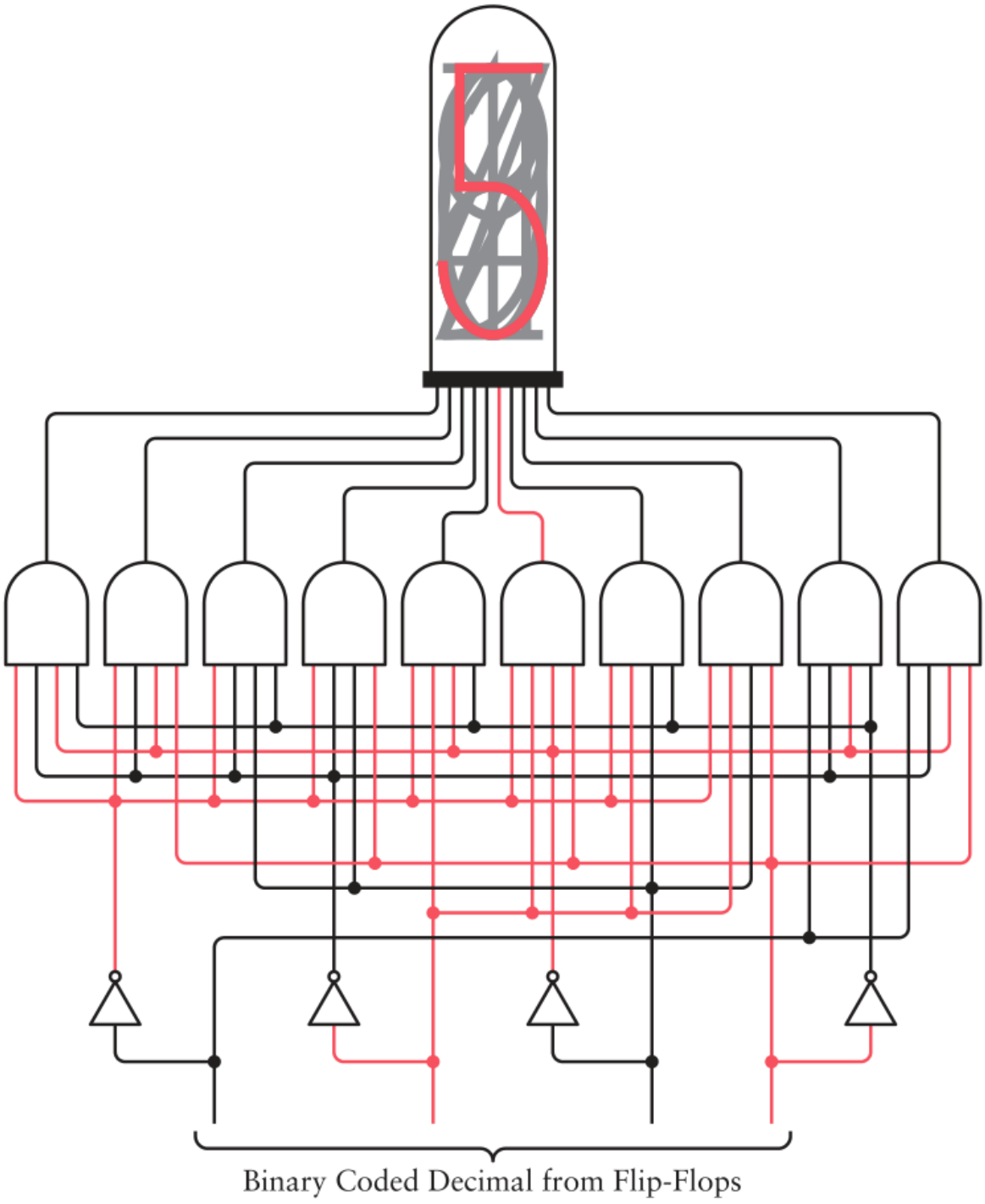
В 1955 году корпорация Burroughs представила такую индикаторную лампу и дала ей имя мифологической водяной нимфы - *Nixie*.

Для каждой из шести цифр времени потребуется отдельная лампа. В принципе пользоваться Nixie легко: нужно лишь создать схему, подающую питание на один из десяти выводов, чтобы зажечь нужную цифру. На практике это несколько сложнее, поскольку требуется больше мощности, чем обычно дают транзисторы интегральных схем. Необходимый ток обеспечивают специальные *драйверы* для ламп Nixie.

Цифровая схема должна преобразовывать BCD-числа с триггеров в отдельные сигналы десяти выводов. При числе 0000 нужен сигнал первого вывода для 0; при 0001 - вывода цифры 1; при 1001 - последнего вывода для 9.

Подобную схему мы видели в конце главы 10 на странице 114: восьмеричное число превращалось в сигналы для включения одной из восьми лампочек. Такая схема называется *декодером*. Теперь она немного расширена для BCD и называется *BCD-декодером*.

Знаю, эта схема выглядит безумной, но построена она совершенно методично:



BCD-число от четырёх триггеров поступает снизу. Красные провода показывают текущее число 0101, то есть 5. Каждый из четырёх сигналов инвертируется; различные сочетания исходных и инвертированных сигналов подаются на десять четырёхвходовых вентилей И.

Для вентиля И, соответствующего 5 и расположенного чуть правее центра, входы таковы:

• младший, самый правый, бит BCD;

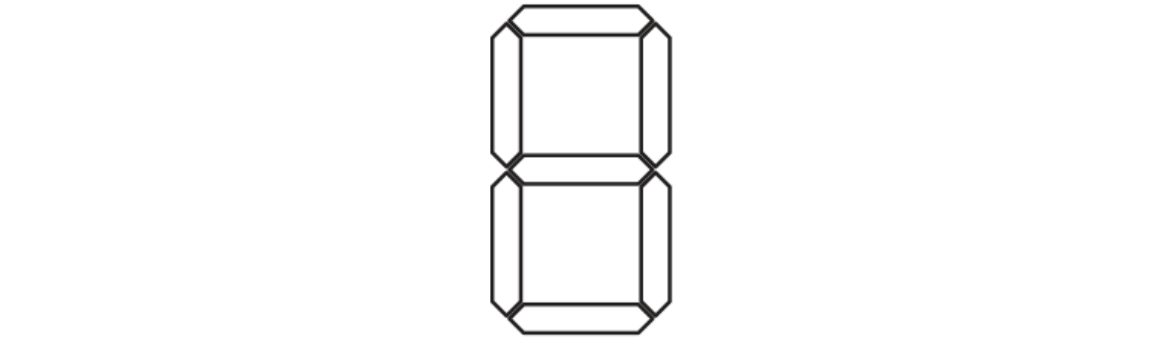
• следующий по значимости бит BCD, инвертированный;

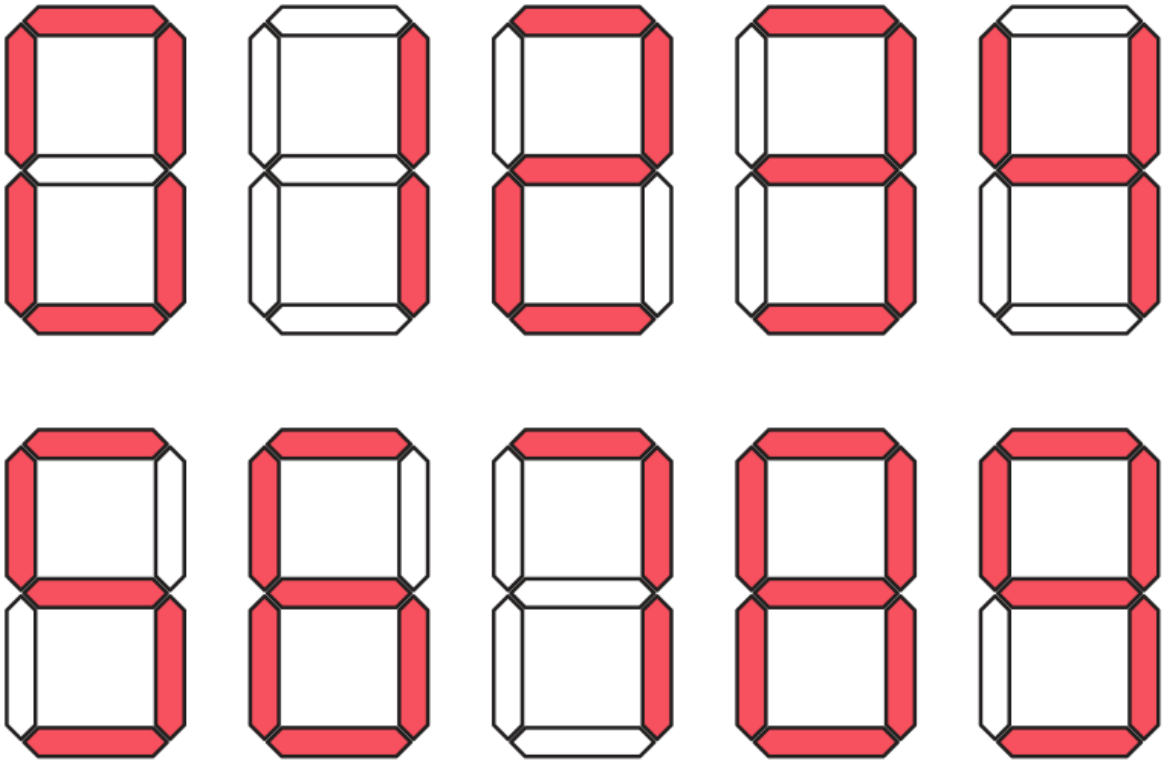
• следующий по значимости бит BCD;

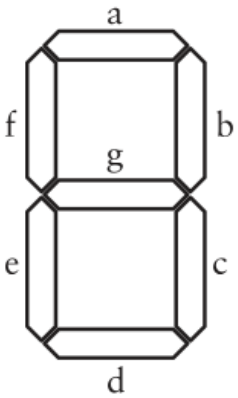
• старший, самый левый, бит BCD, инвертированный.

Все четыре входа равны 1 только для BCD-числа 0101.

Более распространённый способ показа десятичных чисел - *семисегментный индикатор*. Он состоит из семи продолговатых светящихся элементов, расположенных по простой схеме:







Обычно сзади имеется семь выводов, по одному на сегмент, и восьмой вывод земли. Напряжения на разных сочетаниях семи выводов зажигают сегменты, образующие нужную десятичную цифру.

Для удобства соединения каждому сегменту присвоена буква.

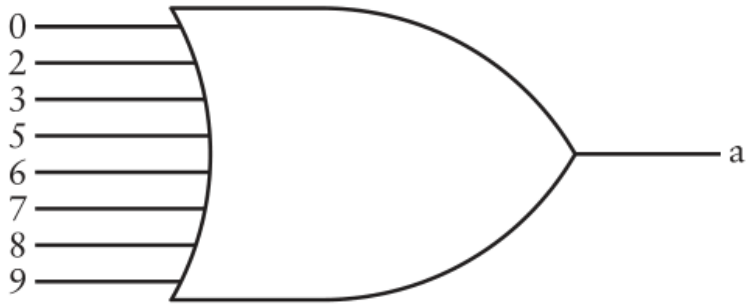
Таблица показывает, какие сегменты должны гореть для каждой цифры от 0 до 9:

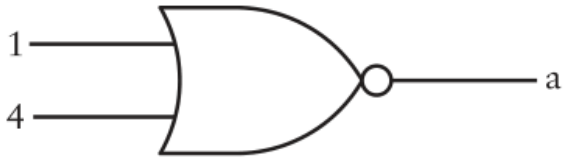
Цифра	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Сигналы для десятичных цифр уже имеются: это выходы вентилей И BCD-декодера, применённого для лампы Nixie.

Сначала рассмотрим сегмент a. Он должен гореть для цифр 0, 2, 3, 5, 6, 7, 8 и 9. Значит, выходы восьми соответствующих вентилей И можно подать на восьмивходовый ИЛИ:

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1





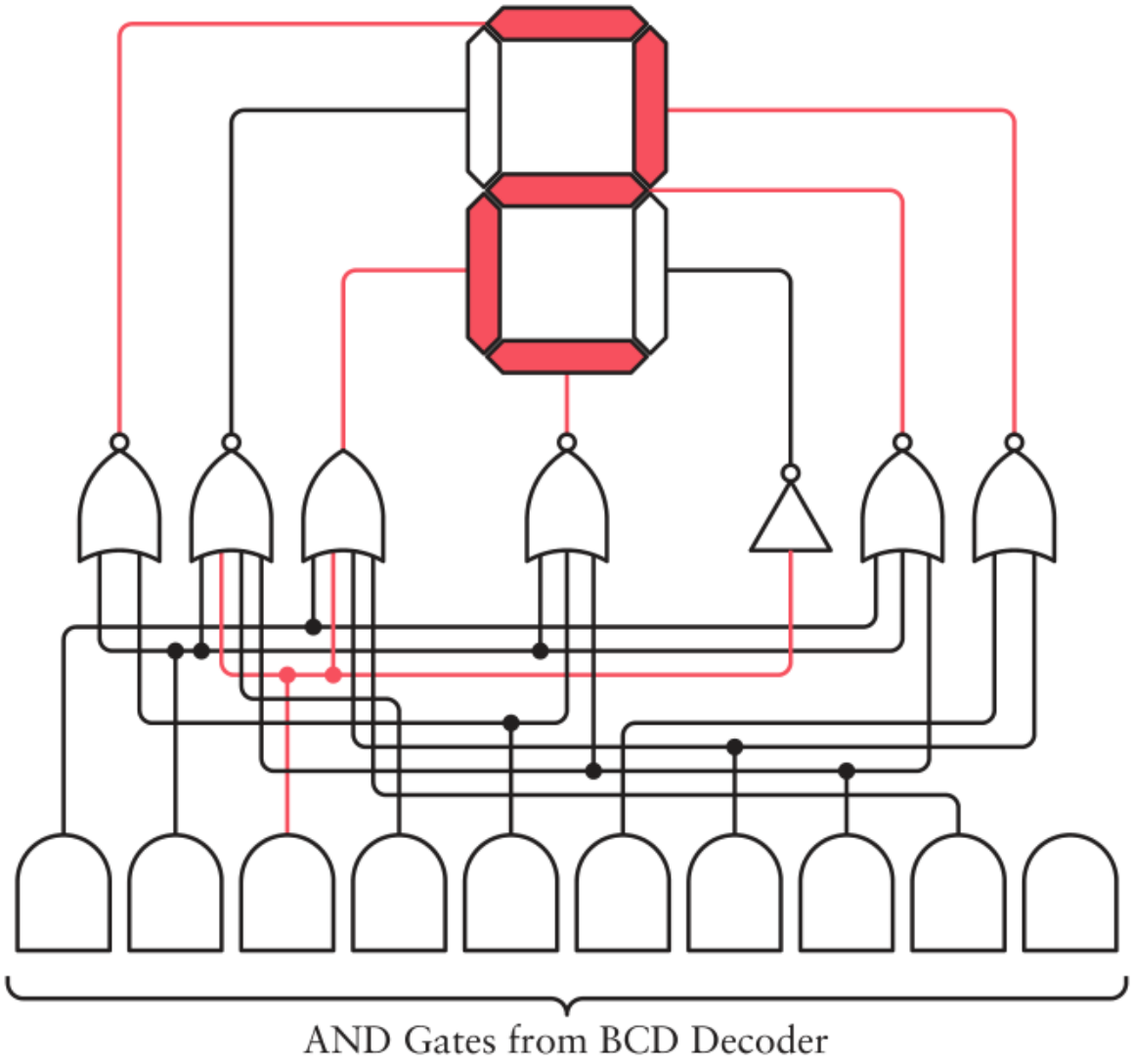
То же самое можно сделать для сегментов b-g.

Но можно найти более простой путь. Сегмент a горит для 0, 2, 3, 5, 6, 7, 8 и 9, то есть не горит лишь для 1 и 4. Поэтому выходы двух вентилей И, соответствующих 1 и 4, можно подать на обычный двухвходовый ИЛИ-НЕ.

Выход этого ИЛИ-НЕ равен 1, кроме случаев сигналов 1 и 4. Для этих двух цифр верхний сегмент не зажигается.

Два подхода не вполне одинаковы. Иногда семисегментный индикатор требуется полностью погасить. Для этого ни один из десяти сигналов BCD-декодера не должен быть равен 1. Восьмивходовый ИЛИ правильно отработает такую ситуацию, а двухвходовый ИЛИ-НЕ продолжит зажигать верхний сегмент.

Если индикатор всегда показывает цифру, семисегментный декодер можно построить из BCD-декодера так:



Для пяти сегментов применяются ИЛИ-НЕ, для двух остальных - ИЛИ и инвертор.

Обратите внимание: крайний правый вентиль И вообще никуда не подключён. Он относится к цифре 9, но все сегменты девятки и без него зажигаются через ИЛИ-НЕ и инвертор.

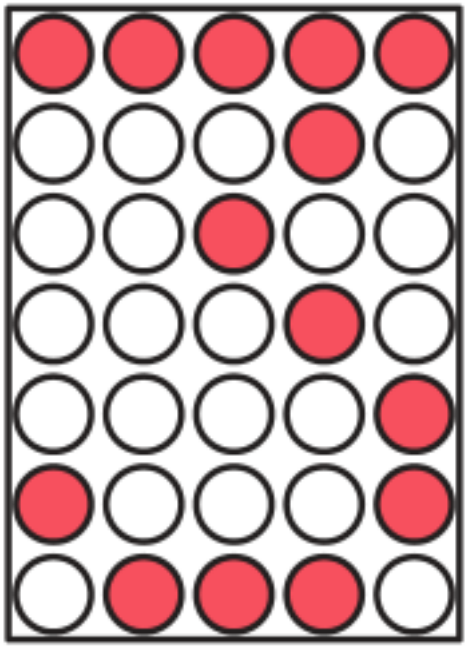
Семисегментный индикатор можно приспособить и для дополнительных цифр шестнадцатеричных чисел, однако придётся отличать шестнадцатеричную B от 8, а D от 0. Одно решение смешивает верхний и нижний регистр: A, b, C, d, E и F.

Для всех 26 букв потребовалось бы ещё несколько, в том числе диагональных, сегментов. Распространены 14- и 16-сегментные индикаторы.

Можно подумать, что для каждой отображаемой цифры нужна своя схема декодирования. Так сделать можно, но существуют способы уменьшить число компонентов. Приём под названием *мультиплексирование* позволяет нескольким цифрам совместно использовать один декодер. Его

входы быстро переключаются между разными источниками, а выходы одновременно идут ко всем индикаторам. Синхронно с переключением декодера заземляется только один индикатор. В каждый момент горит лишь одна цифра, но переключение происходит настолько быстро, что обычно незаметно.

Ещё один способ показывать числа и буквы - *точечная матрица*, сетка из круглых лампочек по горизонтали и вертикали. Самая маленькая сетка, способная показать все цифры, знаки препинания и латинские буквы без диакритики, имеет пять точек в ширину и семь в высоту. Это матрица 5 на 7; здесь она показывает цифру 3:



Можно предположить, что этими лампочками следует независимо управлять, как сегментами предыдущего индикатора. Но это неудобно: лампочек 35, и для индивидуального управления нужна слишком большая схема. Применяют другой подход.

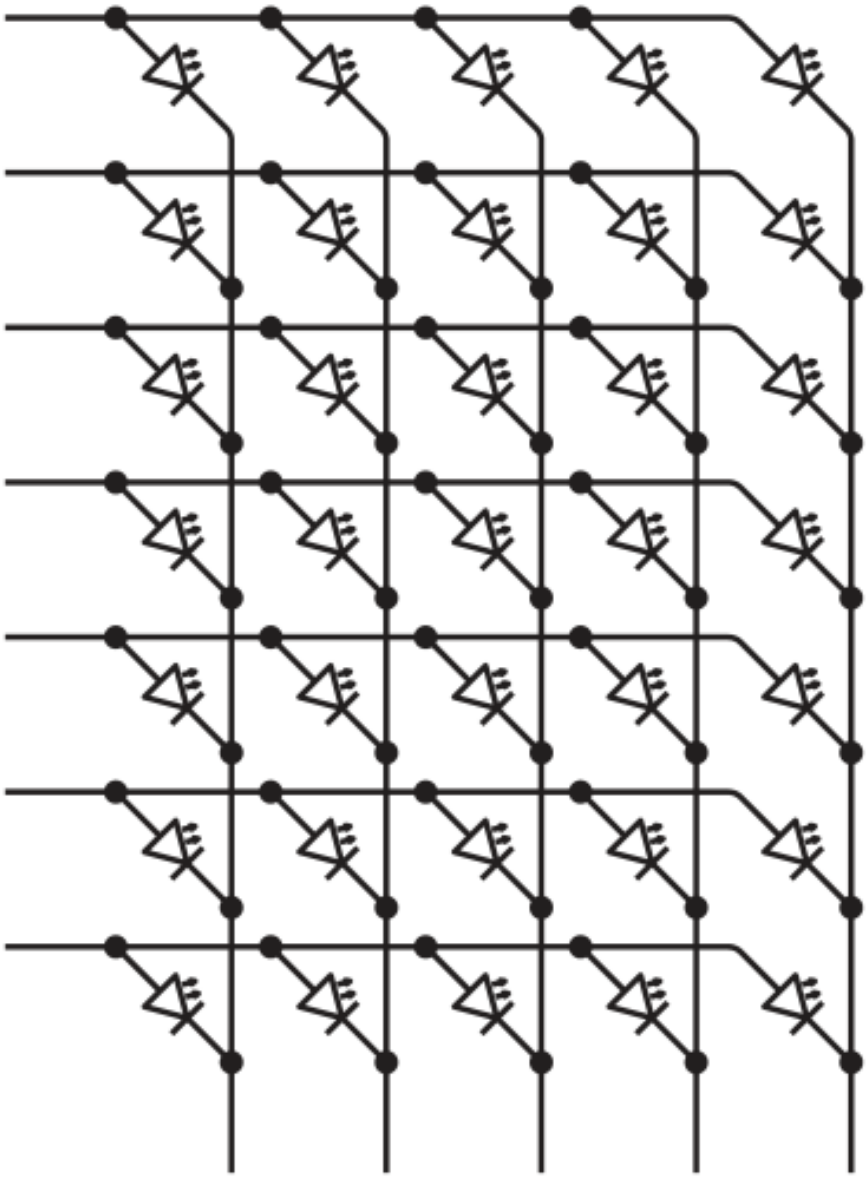
Эти 35 лампочек являются *светодиодами*, или LED. Диоды - небольшие электрические компоненты со специальным условным обозначением.

Диод пропускает ток только в одном направлении, здесь слева направо. Вертикальная черта справа символизирует преграду для тока, который иначе пошёл бы справа налево.

Светодиод - это диод, испускающий фотоны при прохождении тока. Наши глаза воспринимают их как свет. Светодиод обозначается как обычный диод с маленькими стрелками, изображающими лучи.

За последние десятилетия светодиоды стали ярче и дешевле, и теперь они повсеместно освещают дома, расходуя меньше электричества и выделяя меньше тепла, чем другие лампы.

Тридцать пять светодиодов матрицы 5 на 7 соединены так:



Каждый LED находится на пересечении строки и столбца. В каждой строке входы диодов объединены, а в каждом столбце объединены выходы. Можно, наоборот, объединить входы по столбцам, а выходы по строкам; существенной разницы в работе нет.

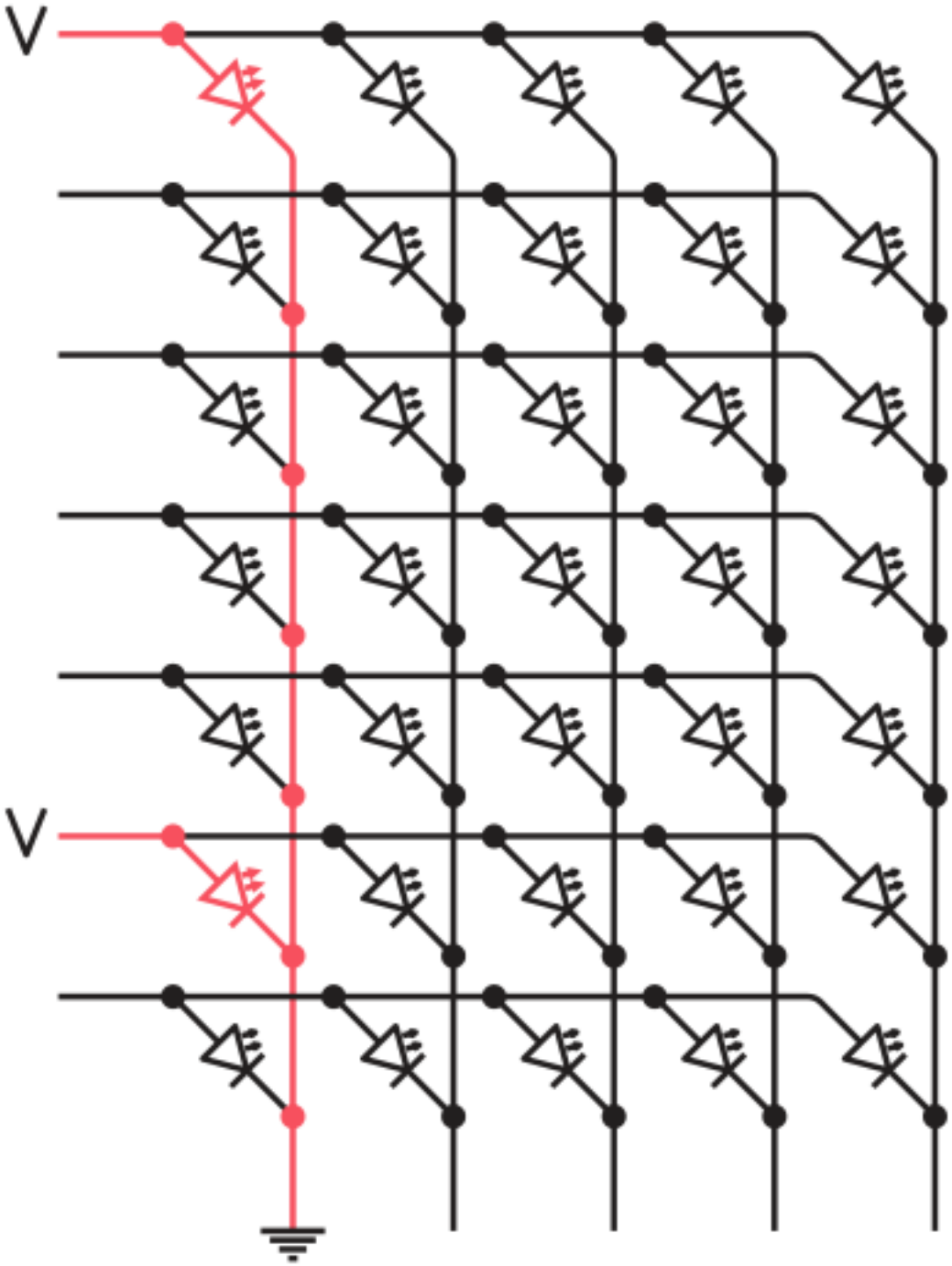
Такая организация уменьшает число соединений с 35 до 12: семь строк и пять столбцов.

Недостаток в том, что одновременно можно зажигать лишь одну строку или один столбец. Сначала это кажется ужасным ограничением, но есть хитрость: если показывать строки и столбцы матрицы последовательно и очень быстро, будет казаться, что весь рисунок горит одновременно.

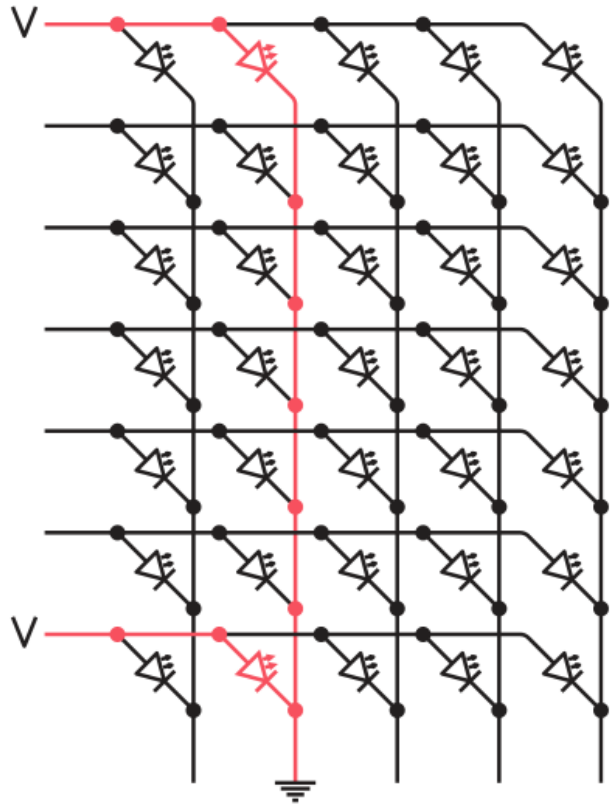
Вернёмся к цифре 3. В крайнем левом столбце горят две точки: верхняя и предпоследняя снизу. Для этого подадим напряжение на две соответствующие строки, а первый столбец соединим с землёй:

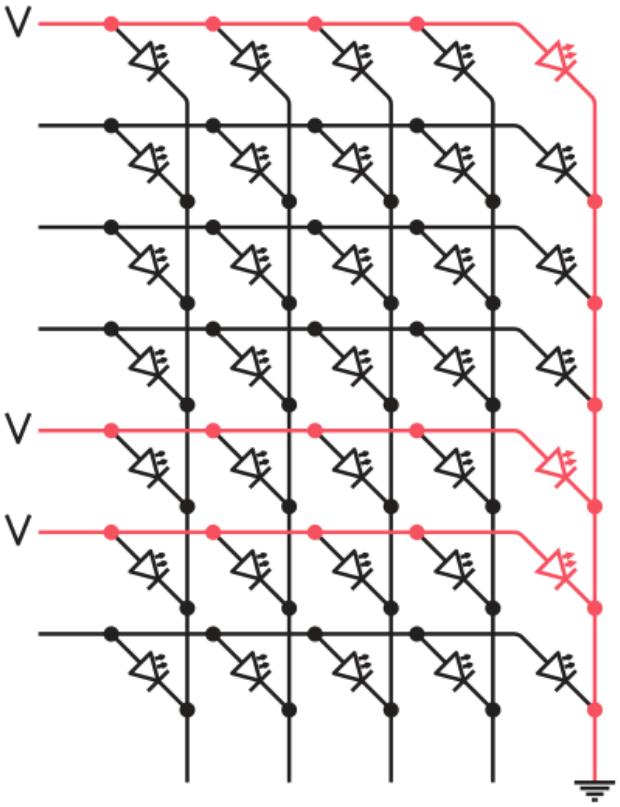
Если проследить все возможные пути от напряжения к земле, видно, что диоды преграждают все пути, кроме ведущих через две нужные лампочки.

Во втором столбце цифры 3 горят верхняя и нижняя точки. Подадим напряжение на эти две строки, а землю - на второй столбец:



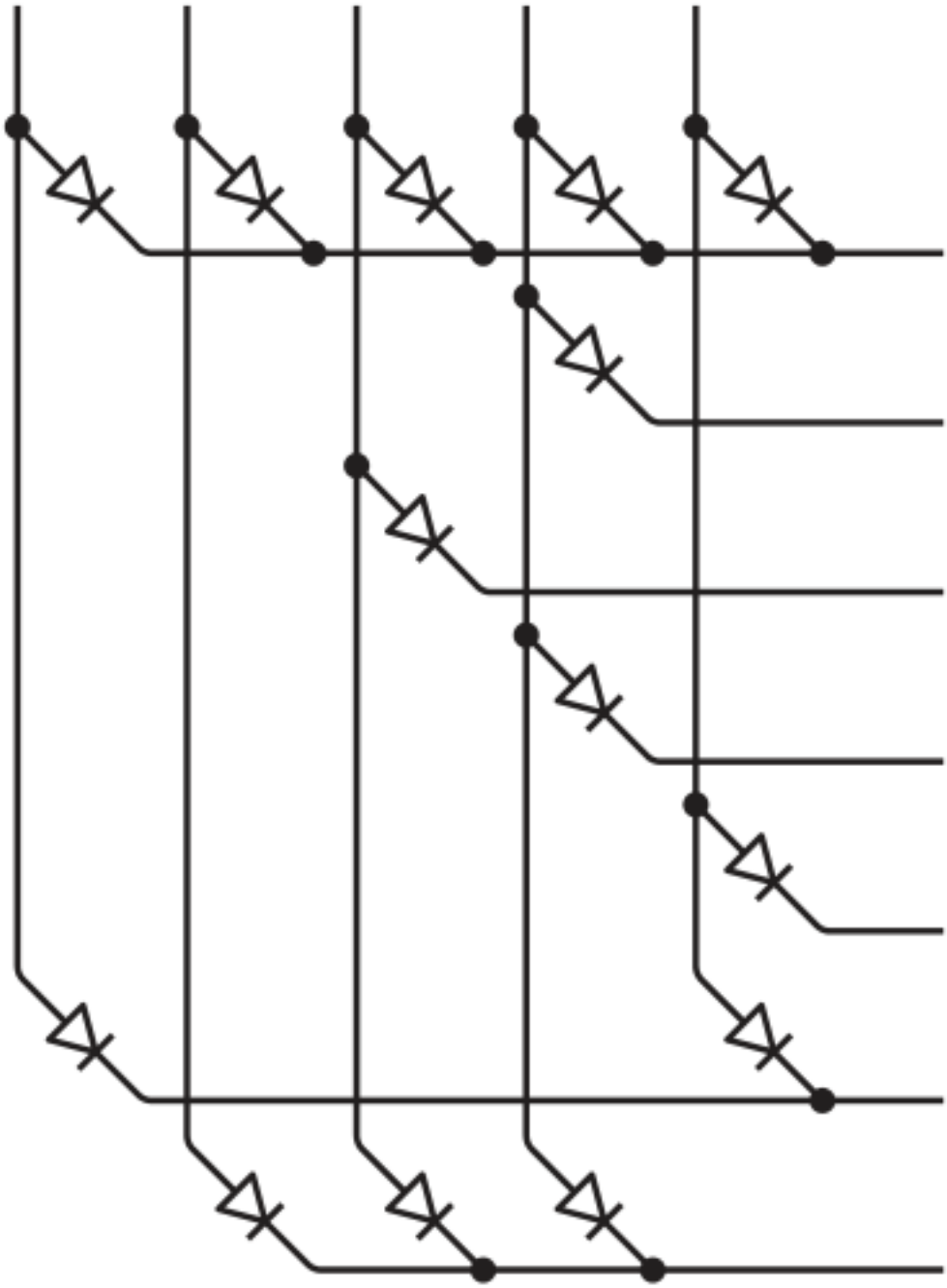
Остальные столбцы работают аналогично. В крайнем правом столбце цифры 3 должны гореть три точки. Напряжение подаётся на эти строки, а земля - на соответствующий столбец:





Теперь нужно автоматизировать подачу напряжения на строки матрицы и подключение земли к одному из столбцов.

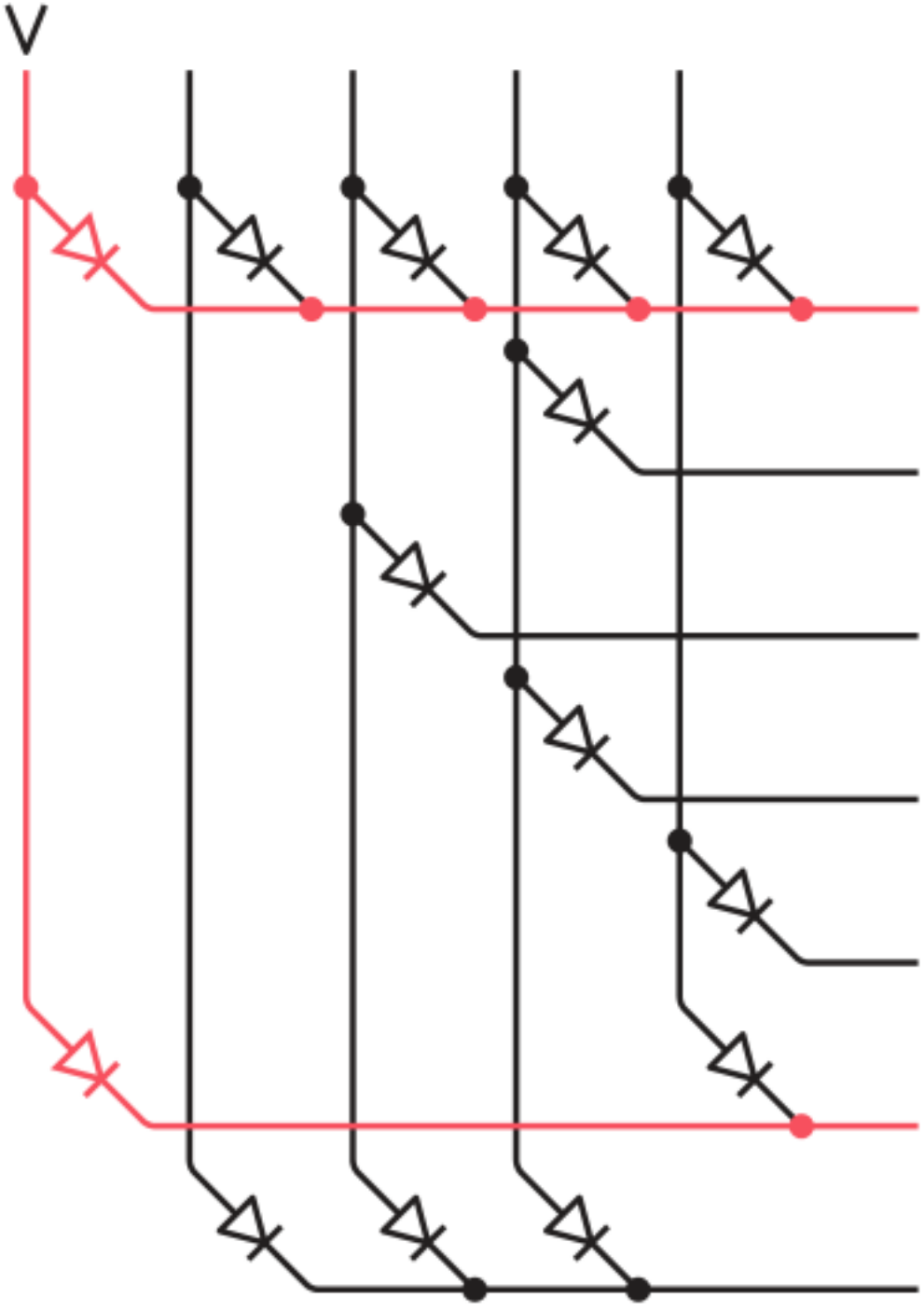
На помощь снова приходят диоды, но теперь обычные, а не светоизлучающие. Вот как можно соединить диоды в форме цифры 3:



Видите здесь тройку? Диоды точно соответствуют горящим точкам показанной ранее цифры 3. Повторю: это не светодиоды, а обычные диоды. Цифра 3, по существу, закодирована в их соединениях.

Такая конфигурация называется *диодной матрицей*. Она хранит информацию, а именно расположение точек, которые надо зажечь для цифры 3. Поэтому диодная матрица считается разновидностью *памяти*. Поскольку её содержимое нельзя изменить, не перепаяв диоды, точнее назвать её разновидностью *постоянной памяти*, или ROM.

Диодная ROM-матрица помогает вывести цифру 3 на точечный светодиодный индикатор. Сверху проходят провода, соответствующие пяти столбцам индикатора. На следующем рисунке напряжение подано на первый слева вертикальный провод:

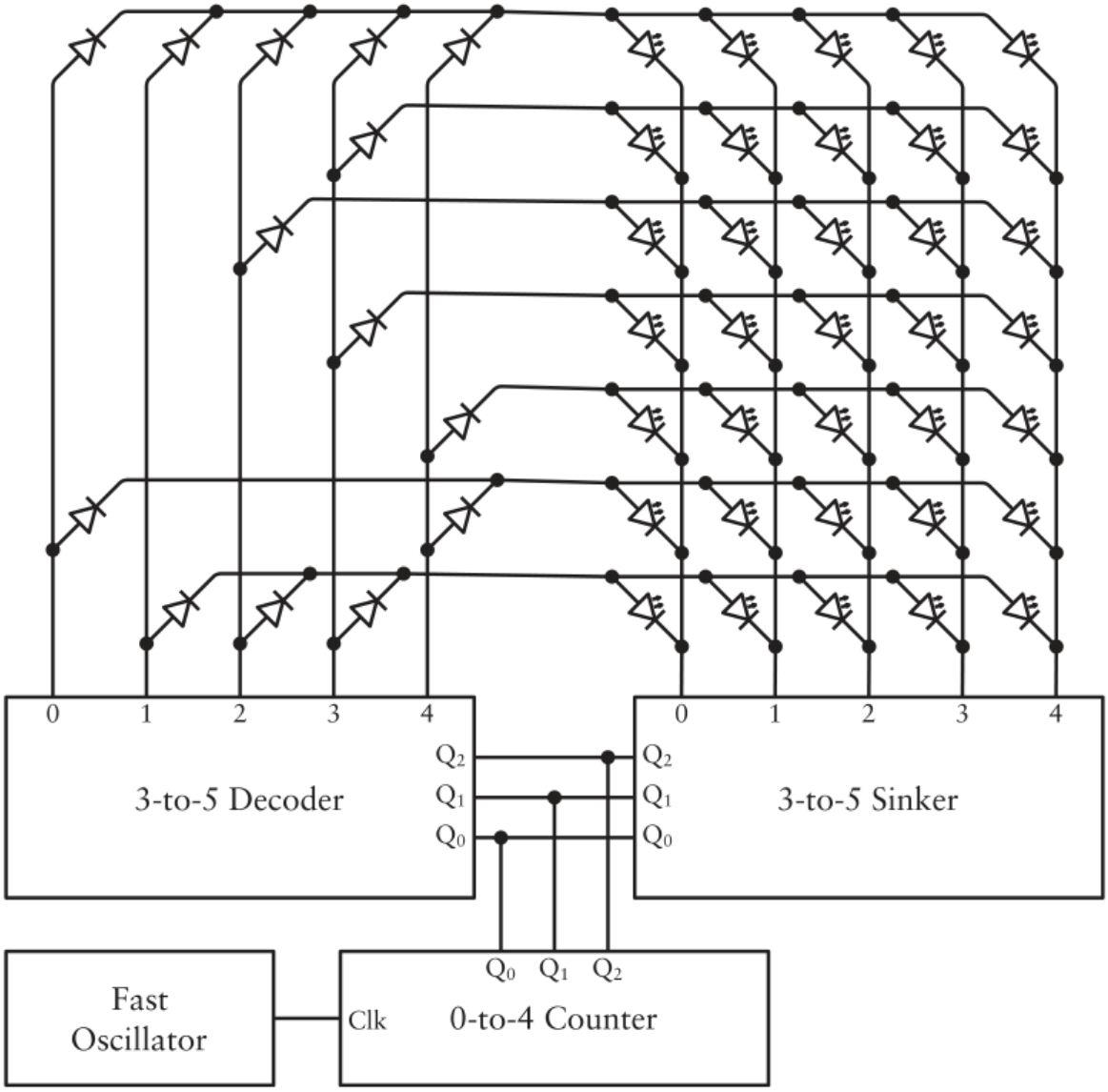


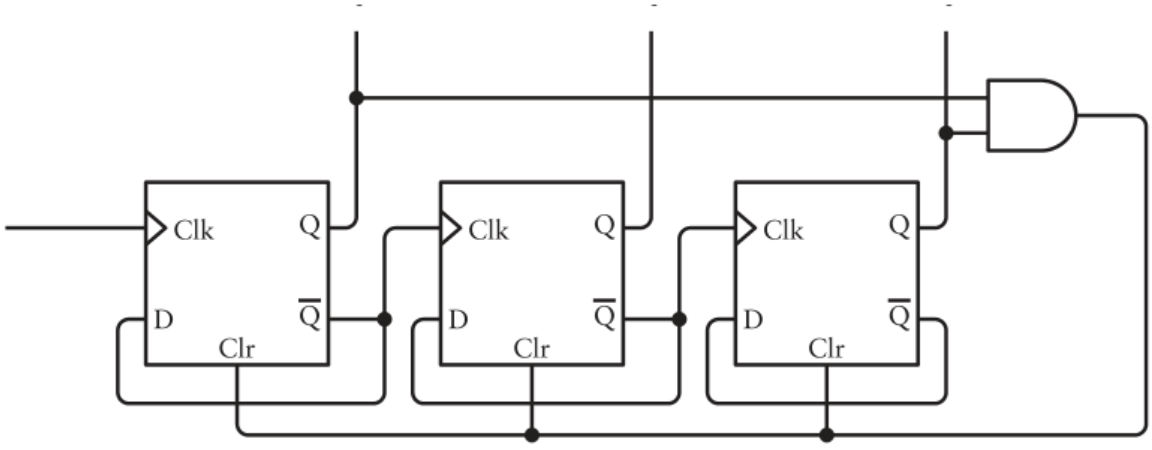
Благодаря расположению диодов справа появляются два напряжения. Они соответствуют точкам, которые должны гореть в первом столбце цифры 3.

Последовательно и быстро подавая напряжение на остальные столбцы, можно получить все комбинации напряжений, необходимые точечной матрице.

Схема соединяет диодное ПЗУ с точечным индикатором и вспомогательными компонентами. Матрица ROM немного повёрнута относительно предыдущего рисунка, но функционально ничем не отличается.

Начнём изучение схемы с нижнего левого угла. Нужен очень быстрый генератор, достаточно быстрый, чтобы включать и выключать точки незаметно для зрительной системы человека. Его сигнал поступает на счётчик из трёх триггеров:

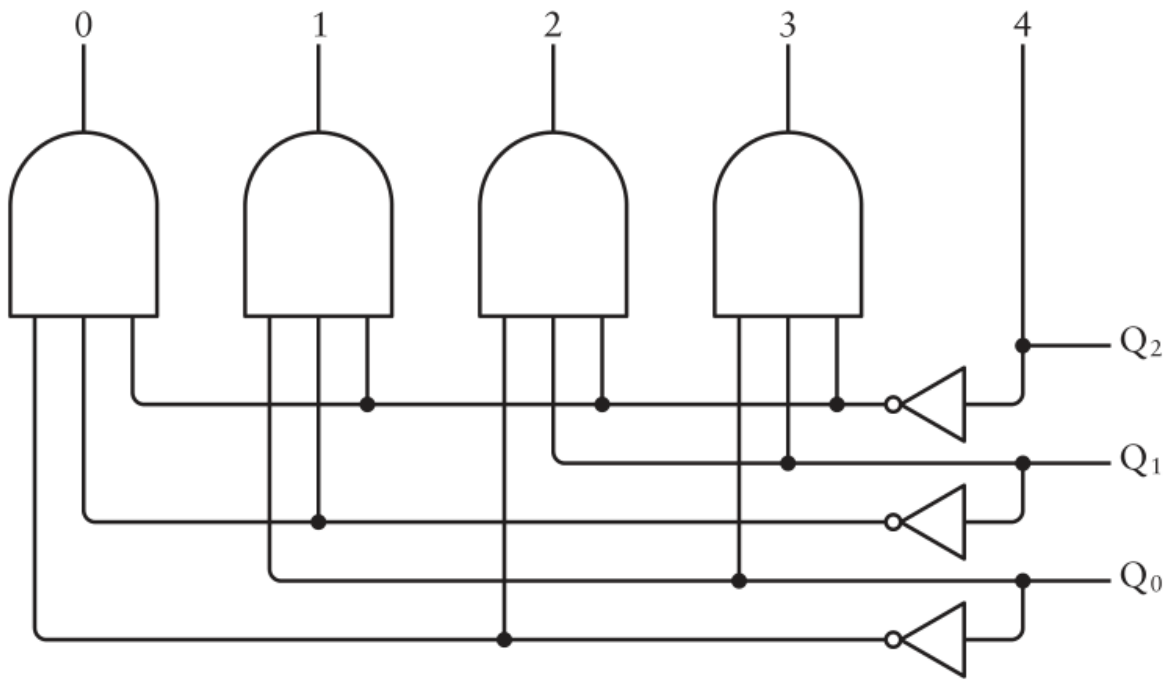




Такие схемы нам уже встречались.

Счётчик похож на применённые в часах, но считает лишь от 0 до 4: в двоичном виде 000, 001, 010, 011 и 100. При достижении 101 вентиль И очищает три триггера до 0.

Эти двоичные числа поступают в декодер 3 в 5 слева:



Это сокращённая версия декодера 3 в 8 со страницы 114 главы 10 и BCD-декодера из этой главы. Она сокращена потому, что нужно преобразовать только трёхразрядные числа 000-100 в один из пяти сигналов.

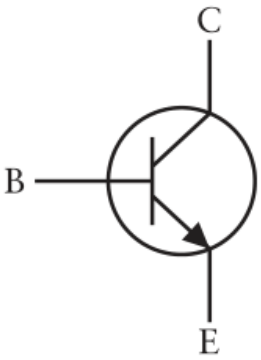
При счёте от 000 до 100 активными по очереди становятся выходы декодера 0, 1, 2, 3, 4, затем в новом цикле снова 0. Для выхода 4 вентиль И не требуется: Q_2 равен 1 только при двоичном числе 100, то есть 4.

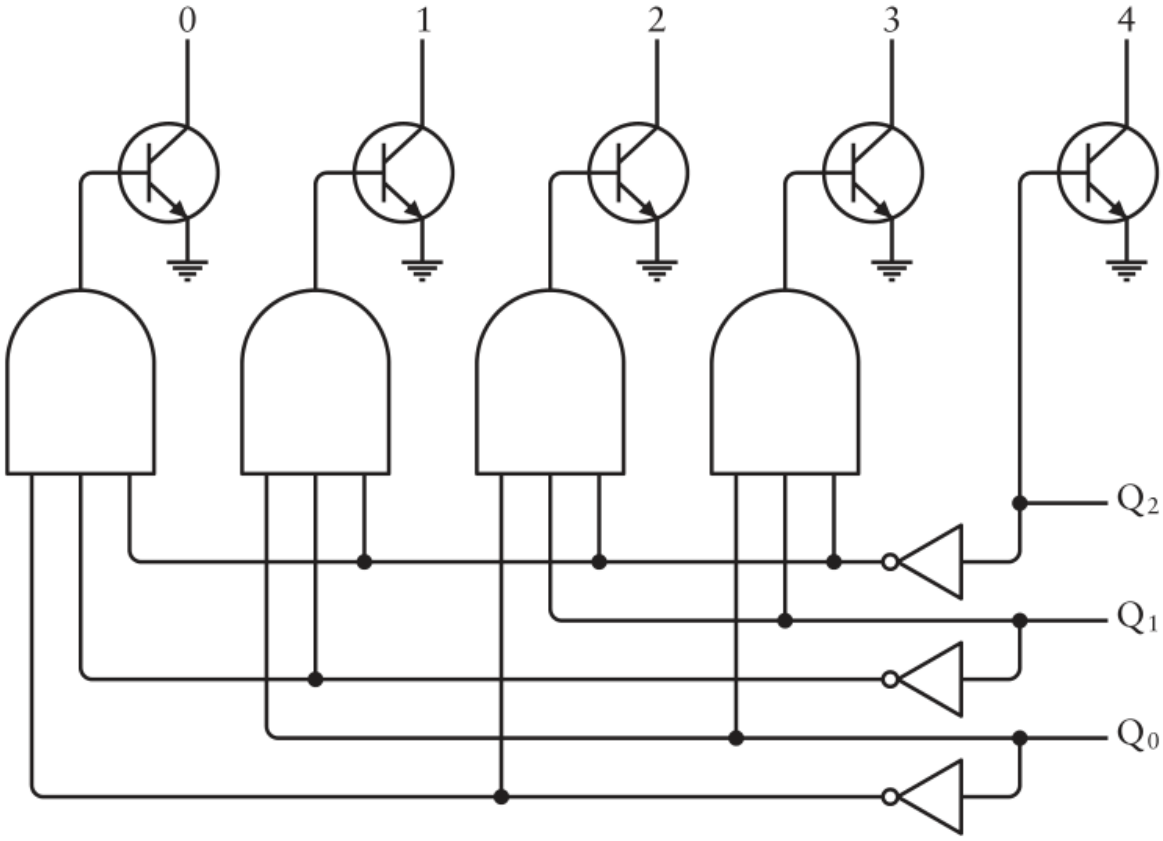
Пять выходов 0-4 соответствуют пяти столбцам диодного ПЗУ слева большой схемы. ПЗУ подаёт напряжения на семь строк точечного индикатора справа. В настоящей схеме между диодной и светодиодной матрицами были бы установлены резисторы для ограничения тока и защиты LED от перегорания.

Затем токи спускаются в загадочный блок «поглотитель 3 в 5» - единственный новый компонент схемы.

Нам требуется нечто необычное. Синхронно с пятью сигналами, идущими вверх через диодную матрицу, один из пяти столбцов светодиодной матрицы нужно соединять с землёй. До сих пор мы строили схемы и вентили, подающие напряжение. Такую схему можно назвать *источником тока*. Теперь нужно обратное: устройство, *поглощающее* ток, то есть автоматически соединяющее цепь с землёй.

Воспользуемся транзисторами из главы 15. Здесь показан один из них, по крайней мере его условное обозначение:





Три буквы означают базу, коллектор и эмиттер. Ток, поданный на базу, позволяет току течь от коллектора к эмиттеру, поэтому эмиттер можно соединить с землёй.

Поглотитель 3 в 5 внизу большой схемы чрезвычайно похож на декодер 3 в 5. Для обоих можно применить одну и ту же логическую схему. Единственное отличие: выходы верхних вентилей И и вход Q_2 соединены с базами пяти транзисторов.

Токи, спускающиеся через пять столбцов матричного индикатора, поочерёдно соединяются этими транзисторами с землёй.

Теперь у нас есть законченная схема, показывающая на матрице цифру 3. Конечно, нужно не совсем это. Как и для Nixie и семисегментного индикатора, требуется показывать цифры, поступающие от часов.

Чтобы выводить цифры 0-9, диодную матрицу нужно расширить рисунками остальных девяти цифр, а затем добавить ещё один уровень выбора с помощью BCD-декодера.

Анимированная версия этой схемы, показывающая цифры от 0 до 9, доступна на сайте CodeHiddenLanguage.com.

Но сколько бы удовольствия ни доставлял показ анимированных цифр, соответствующих двоичным числам, цель этой книги состоит не в том, чтобы построить часы.

Глава 19. Ансамбль памяти

Каждое утро, пробуждаясь ото сна, мы заполняем пробелы с помощью памяти. Мы вспоминаем, где находимся, что делали вчера и что собираемся делать сегодня. Воспоминания могут нахлынуть потоком или просачиваться по капле, а через несколько минут какие-то провалы ещё сохранятся («Забавно, не помню, чтобы я ложился спать в носках»), но в целом нам обычно удаётся заново собрать свою жизнь и обрести достаточно непрерывности, чтобы начать новый день.

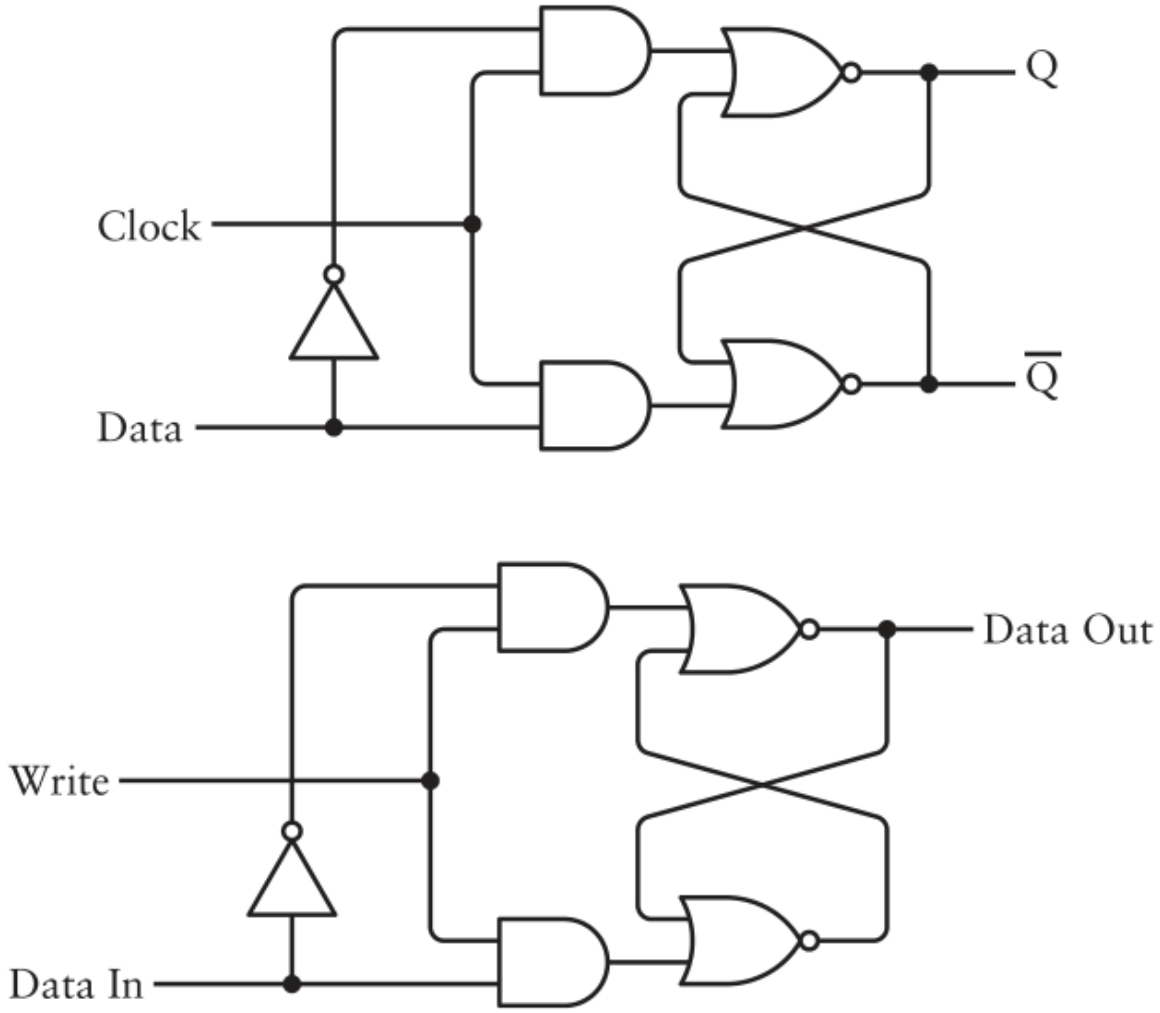
Разумеется, человеческая память устроена не очень упорядоченно. Попробуйте вспомнить что-нибудь из школьной геометрии - и, скорее всего, начнёте думать о том дне, когда объявили учебную пожарную тревогу как раз перед тем, как учитель собирался объяснить значение QED.

Человеческая память также не безошибочна. Вероятно, письменность и была изобретена специально, чтобы компенсировать её недостатки.

Мы *пишем*, а позднее *читаем*. Мы *сохраняем*, а позднее *извлекаем*. Мы *записываем*, а позднее *обращаемся*. Назначение памяти - сохранять информацию неизменной между двумя этими событиями. Всякий раз, когда мы что-нибудь сохраняем, используется та или иная память. Только за минувшее столетие носителями информации служили бумага, пластиковые диски, магнитная лента и различные виды компьютерной памяти.

Даже телеграфные реле, собранные сначала в логические вентили, а затем в триггеры, могут хранить информацию. Как мы видели, триггер способен сохранить 1 бит. Это совсем немного, но уже начало. Научившись хранить 1 бит, легко сохранить 2, 3 или больше.

На странице 224 главы 17 мы встретили D-триггер с управлением уровнем, построенный из инвертора, двух вентилей И и двух ИЛИ-НЕ:



Когда Clock равен 1, выход Q совпадает с Data. Когда Clock переходит в 0, Q удерживает последнее значение Data. Последующие изменения Data не влияют на выходы, пока Clock снова не станет равен 1.

В главе 17 этот триггер участвовал в нескольких схемах, но здесь он почти всегда будет применяться одним способом - для хранения 1 бита. Поэтому переименуем входы и выходы в соответствии с назначением.

Это всё тот же триггер, но Q теперь называется Data Out, а Clock, который в главе 17 начинал как Hold That Bit, - Write. Подобно тому как мы записываем сведения на бумаге, сигнал Write заставляет записать, или сохранить, Data In в схеме. Обычно Write равен 0, и Data In не влияет на выход. Чтобы сохранить бит, мы устанавливаем Write в 1, а затем снова в 0.

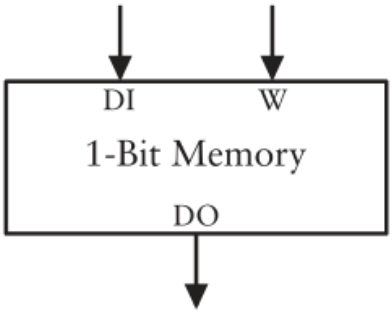
Таблица с сокращениями DI, W и DO выглядит так:

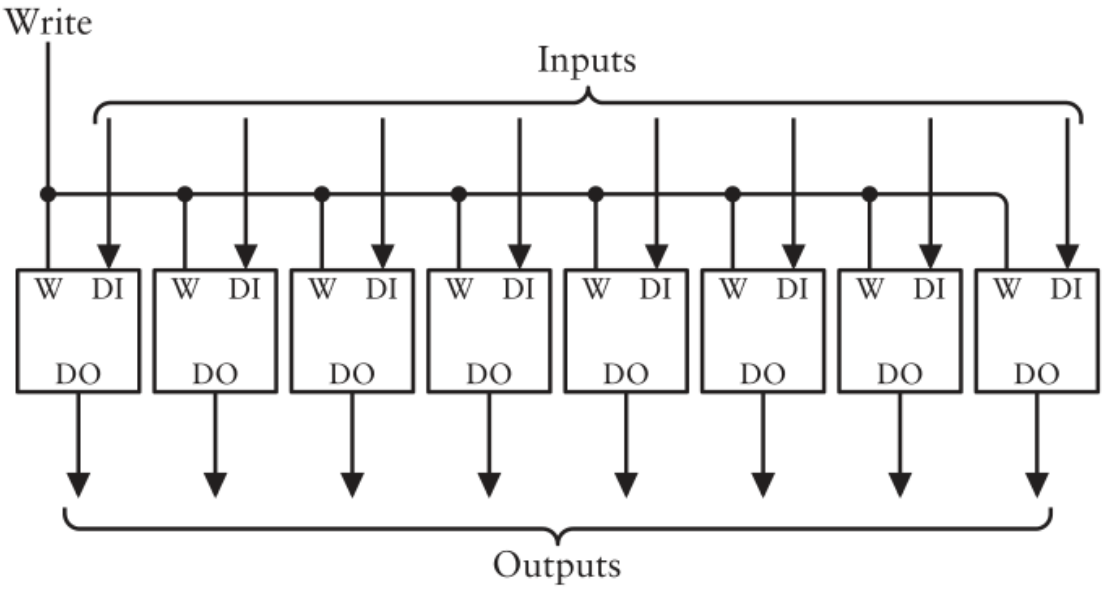
DI	W	DO
0	1	0
1	1	1
X	0	DO

Как упоминалось в главе 17, такая схема также называется защёлкой, потому что цепляется за данные. Но в этой главе мы будем называть её *памятью*. Однобитную память можно обозначать прямоугольником, не рисуя все компоненты. Расположение входов и выходов значения не имеет.

Один бит - совсем немного, но целый байт легко собрать из восьми битов памяти. Нужно лишь соединить восемь сигналов Write:

Inputs		Output
DI	W	DO
0	1	0
1	1	1
X	0	DO





У этой восьмибитной памяти восемь входов, восемь выходов и один общий вход Write, обычно равный 0. Чтобы сохранить байт, следует перевести Write в 1 и обратно в 0. Схему также можно изображать одним прямоугольником.

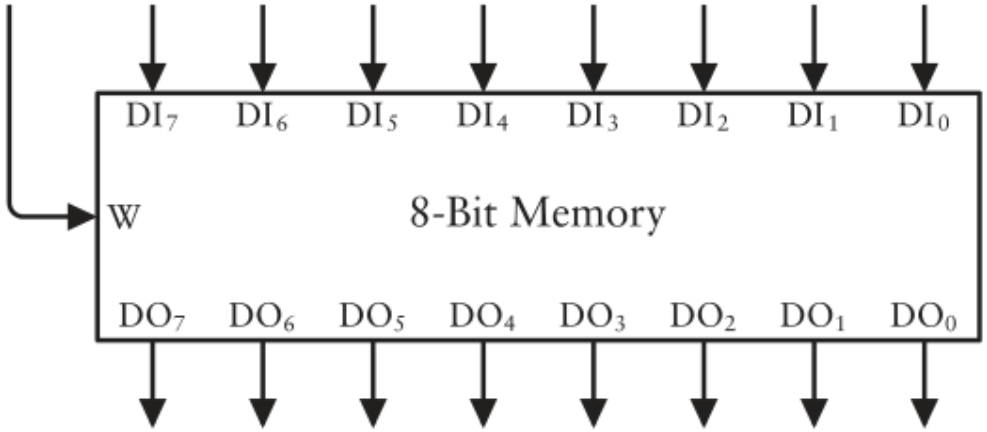
Как обычно, индексы различают восемь битов: 0 означает младший, 7 - старший. Для согласованности с однобитной памятью вход и выход можно показывать восьмиразрядными трактами данных.

Есть и другой, менее прямолинейный способ собрать восемь триггеров. Предположим, нужен только один сигнал Data In и один Data Out, но значение Data In требуется сохранять в восемь разных моментов дня или ближайшей минуты, а позднее считывать эти восемь значений через единственный Data Out.

Иначе говоря, вместо одного восьмибитного значения нужно сохранить восемь отдельных однобитных.

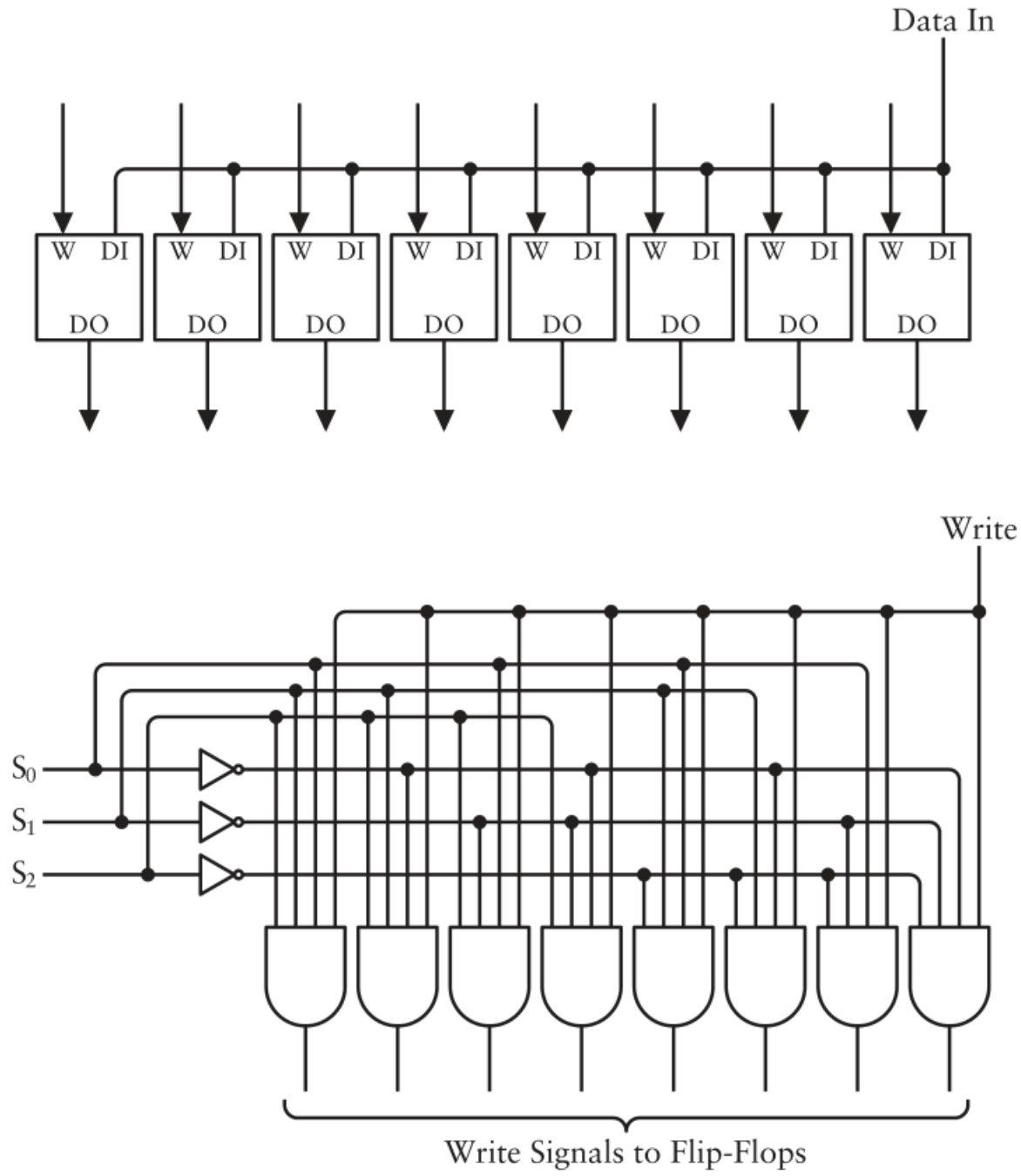
Схема становится сложнее, но число внешних соединений уменьшается. Для восьмибитной памяти их 17, а для восьми отдельных однобитных значений - всего 6.

По-прежнему нужны восемь триггеров, но теперь все входы Data соединены, а сигналы Write разделены:



Общий Data In не означает, что все триггеры хранят одинаковое значение. Их сигналы Write разделены, поэтому конкретный триггер сохраняет Data In только в момент, когда соответствующий Write становится равен 1.

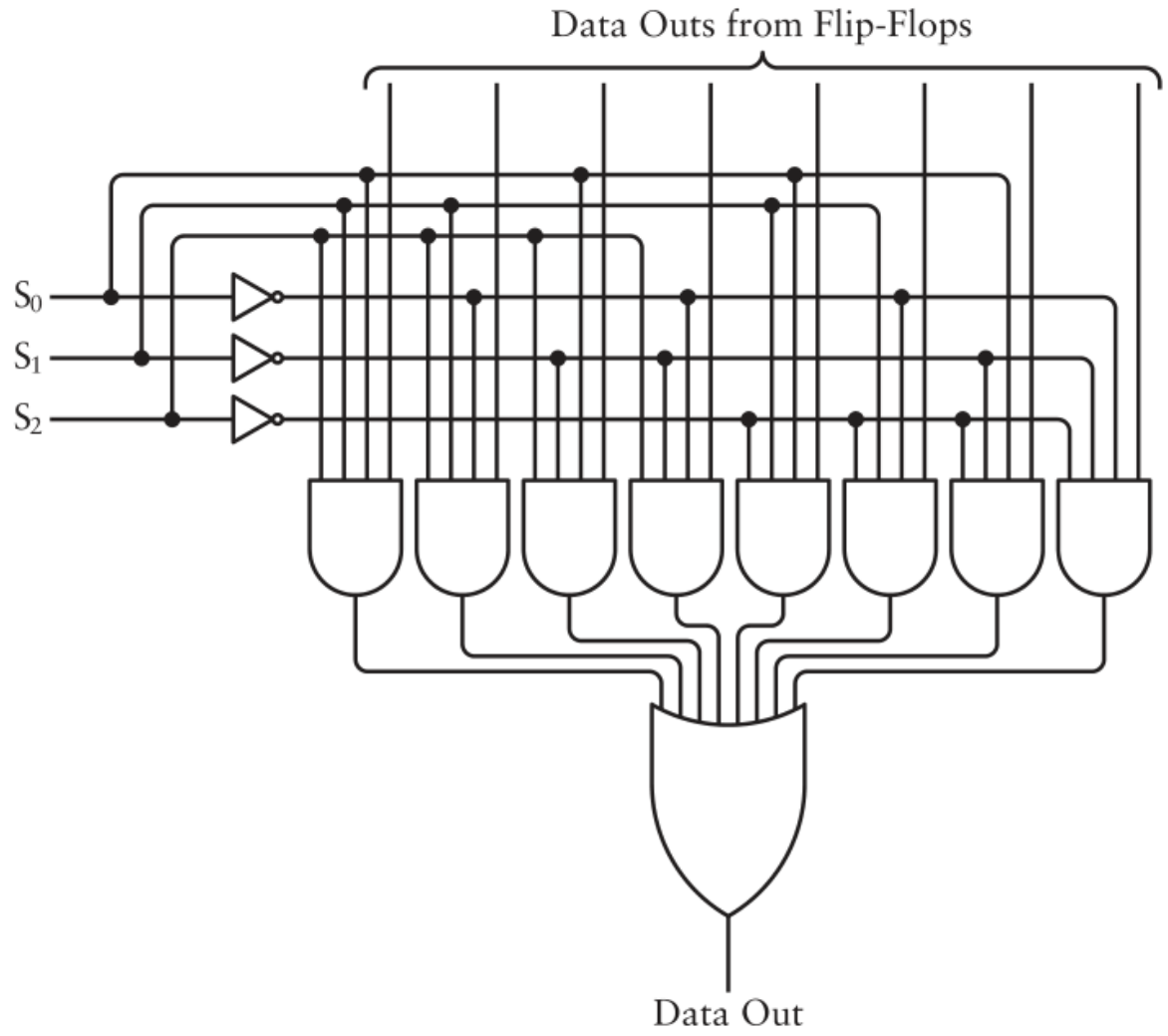
Вместо управления восемью отдельными Write можно использовать один Write и выбирать управляемый триггер декодером 3 в 8:



Подобные схемы уже встречались. На странице 114 в конце главы 10 три выключателя задавали восьмеричное число, а каждый из восьми вентилей И был соединён с лампочкой. В зависимости от заданного числа загоралась одна и только одна лампочка. В главе 18 такие схемы помогали отображать цифры часов.

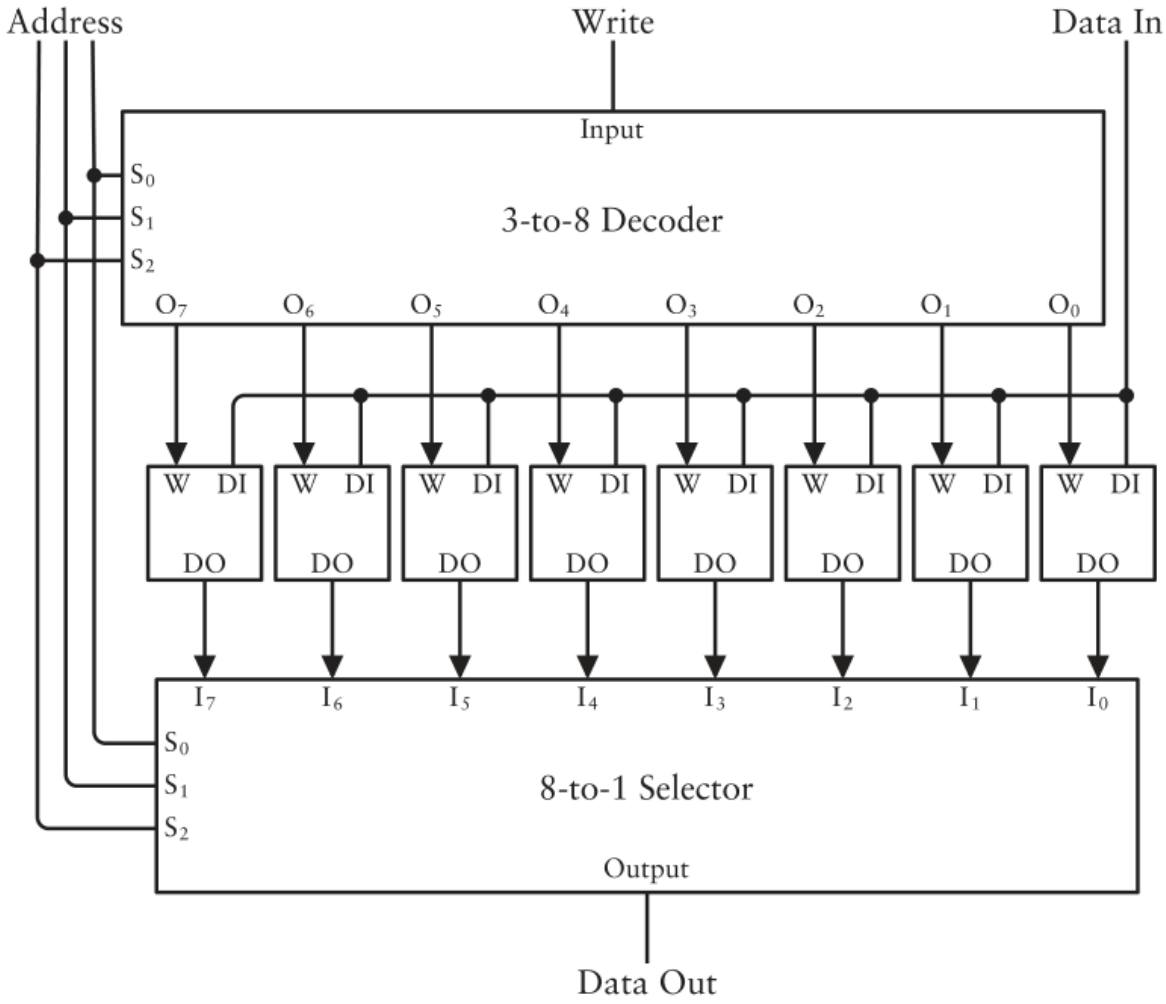
Сигналы S_0, S_1 и S_2 означают Select, «выбор». На каждый вентиль И поступает каждый из этих сигналов либо его инверсия. Этот декодер 3 в 8 универсальнее варианта из главы 10, потому что вместе с S_0-S_2 используется Write. При Write = 0 выходы всех вентилей И равны 0. При Write = 1 ровно один выход равен 1 в зависимости от S_0-S_2.

Выходы Data Out восьми триггеров можно подать на *селектор 8 в 1*, выбирающий один из них:



И здесь три сигнала Select и их инверсии поступают на восемь вентилей И. В зависимости от S_0-S_2 лишь один вентиль способен выдать 1. Но на эти вентили также поступают Data Out триггеров, поэтому выход выбранного вентиля совпадает с соответствующим Data Out. Восьмивходовый ИЛИ формирует окончательный выбранный сигнал.

Декодер 3 в 8 и селектор 8 в 1 объединяются с восемью триггерами:



Три сигнала Select у декодера и селектора одинаковы. Теперь они названы *Address*, «адрес», потому что образуют число, указывающее местонахождение бита в памяти. Это подобно почтовому адресу, только возможны лишь восемь трёхбитных значений: 000, 001, 010, 011, 100, 101, 110 и 111.

На стороне входа Address определяет, какой триггер получит Write и сохранит Data In. На стороне выхода тот же Address управляет селектором 8 в 1, выбирающим выход одной защёлки:

Например, установите Address = 010, Data In = 0 или 1, затем переведите Write в 1 и обратно в 0. Это называется *записью в память*: значение Data In сохраняется по адресу 010.

Измените Address. Вернитесь на следующий день. Если питание не выключалось, снова задайте 010 - и Data Out покажет то, что было записано. Это называется *чтением из памяти* или *обращением к памяти*. По тому же адресу можно записать новое значение, переключив Write.

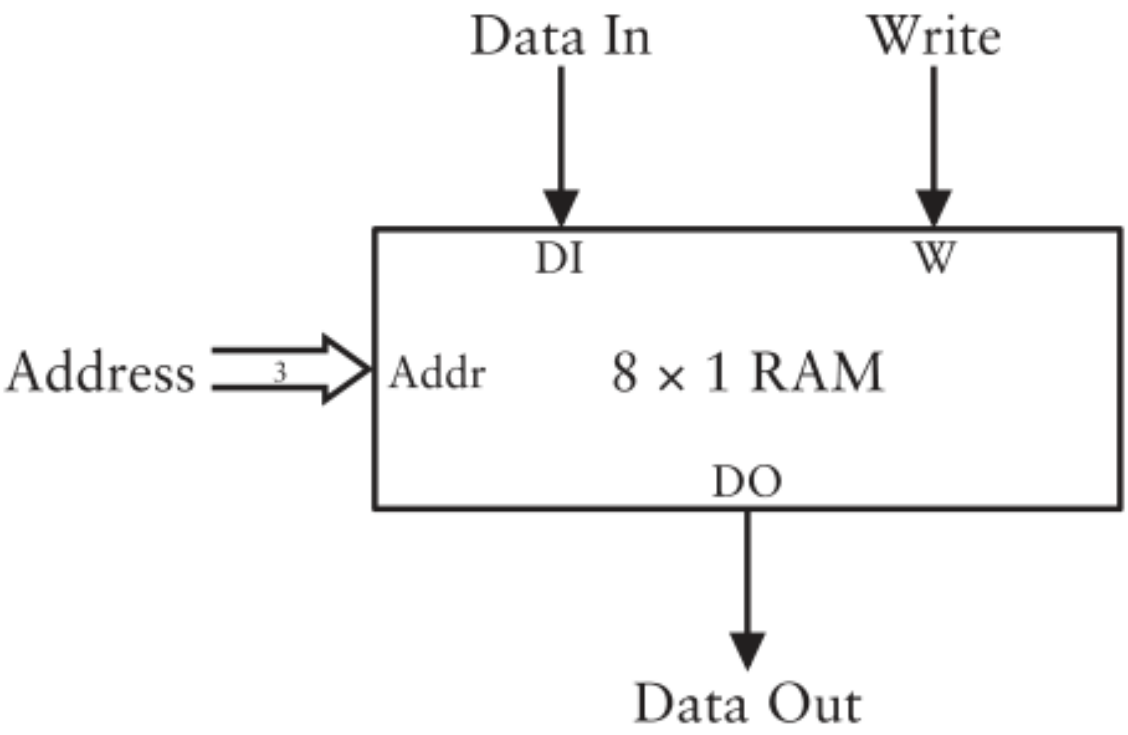
В любой момент Address можно установить в одно из восьми состояний, сохраняя восемь различных однобитных значений. Такая конфигурация иногда называется памятью чтения-записи.

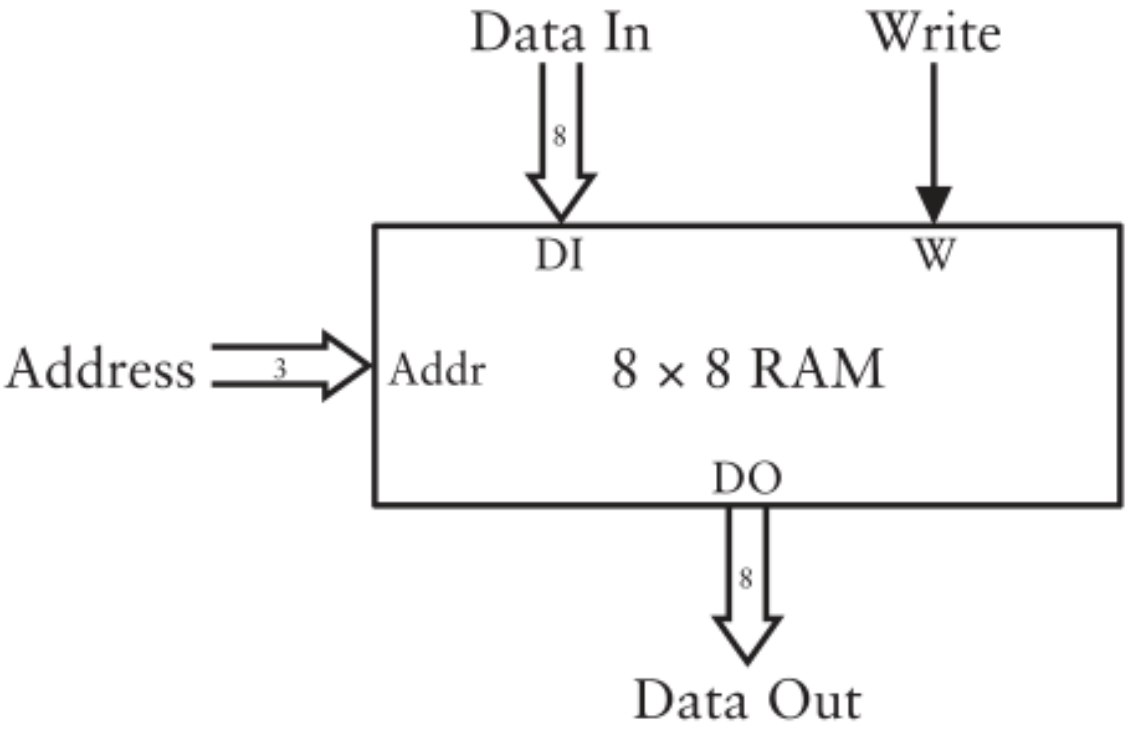
Она позволяет сохранить значения, то есть записать их, а позднее узнать эти значения, то есть прочитать. Поскольку Address в любой момент можно свободно изменить на одно из восьми значений, эта память чаще называется *памятью с произвольным доступом*, или RAM.

Не всякая память допускает произвольный доступ. В конце 1940-х годов, когда память из электронных ламп ещё была непрактична, а транзистор не был изобретён, применялись другие технологии. В одной необычной системе биты хранились в длинных трубках с ртутью. Импульсы распространялись от одного конца к другому, подобно волнам на пруду, и читать их можно было только последовательно. Разные виды памяти на линиях задержки применялись до 1960-х годов.

Построенная RAM хранит восемь отдельных однобитных значений. Такая конфигурация называется *массивом RAM* и обозначается 8 x 1, «восемь на один». Каждое из восьми значений имеет длину 1 бит. Общая ёмкость равна произведению: 8 x 1 = 8 бит.

Большие массивы можно собирать из меньших. Если взять восемь массивов 8 x 1 и соединить все Address и все Write, получится массив 8 x 8:

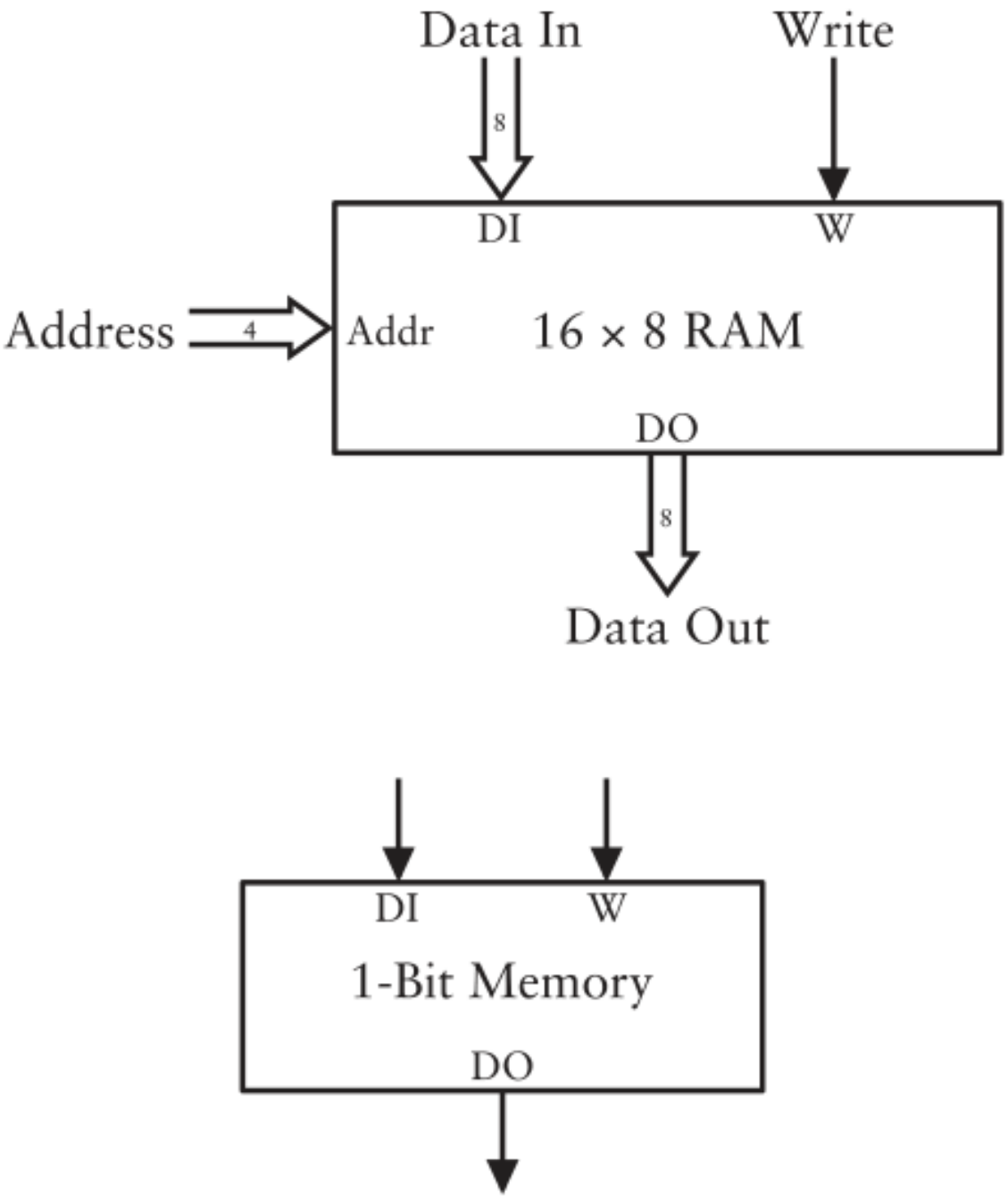




Теперь Data In и Data Out имеют ширину 8 бит. Массив хранит восемь отдельных байтов, каждый из которых выбирается трёхбитным адресом.

Однако при сборке из восьми массивов 8 x 1 вся логика декодирования и выбора дублируется. Более того, декодер 3 в 8 и селектор 8 в 1 во многом похожи: оба используют восемь четырёхвходовых вентилей И, выбираемых тремя сигналами Address. В реальной памяти эти вентили используются совместно.

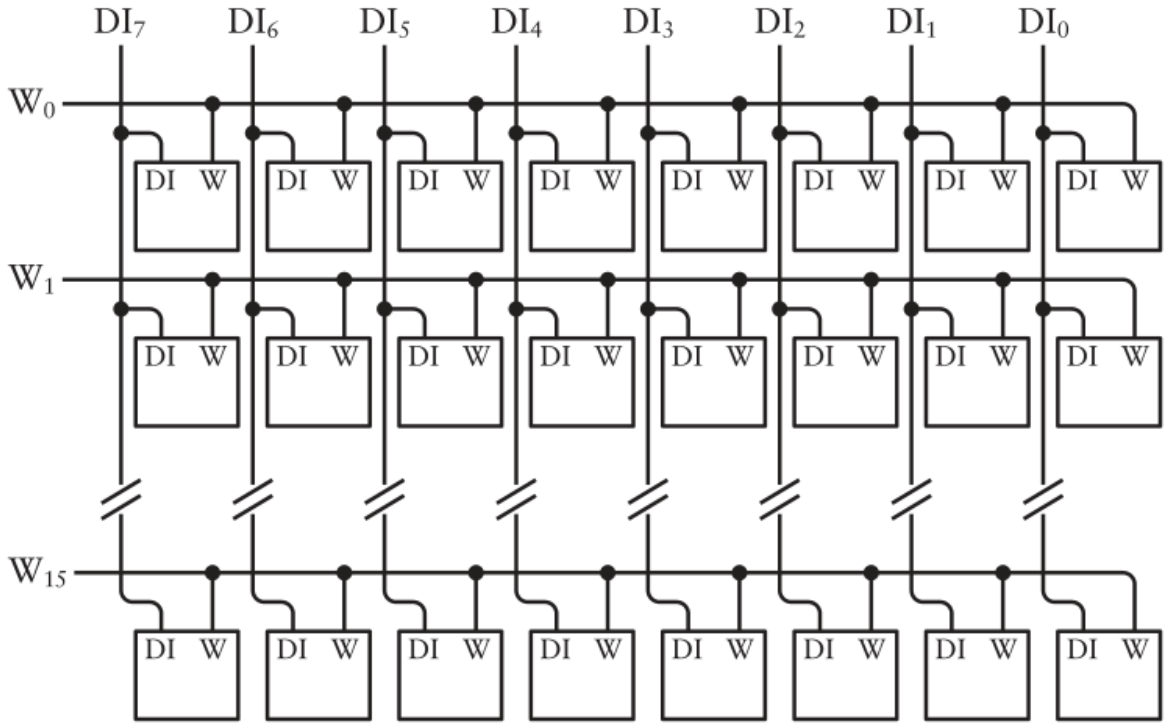
Попробуем построить RAM эффективнее. Вместо 8 x 8, хранящего 8 байтов, удвоим память и сделаем массив 16 x 8 на 16 байтов:



Для адресации 16 байтов нужен четырёхбитный адрес. Общая ёмкость равна 16 x 8 = 64 бита, поэтому понадобятся 64 триггера. Полная схема не поместится на страницах книги, и далее она будет показана по частям.

Однобитную память можно обозначать ячейкой с входами Data In и Write и выходом Data Out.

Один бит памяти иногда называют *ячейкой памяти*. Разместим 64 ячейки сеткой из 8 столбцов и 16 строк. Каждая строка из восьми ячеек представляет байт; 16 строк, из которых показаны только три, предназначены для 16 байтов:

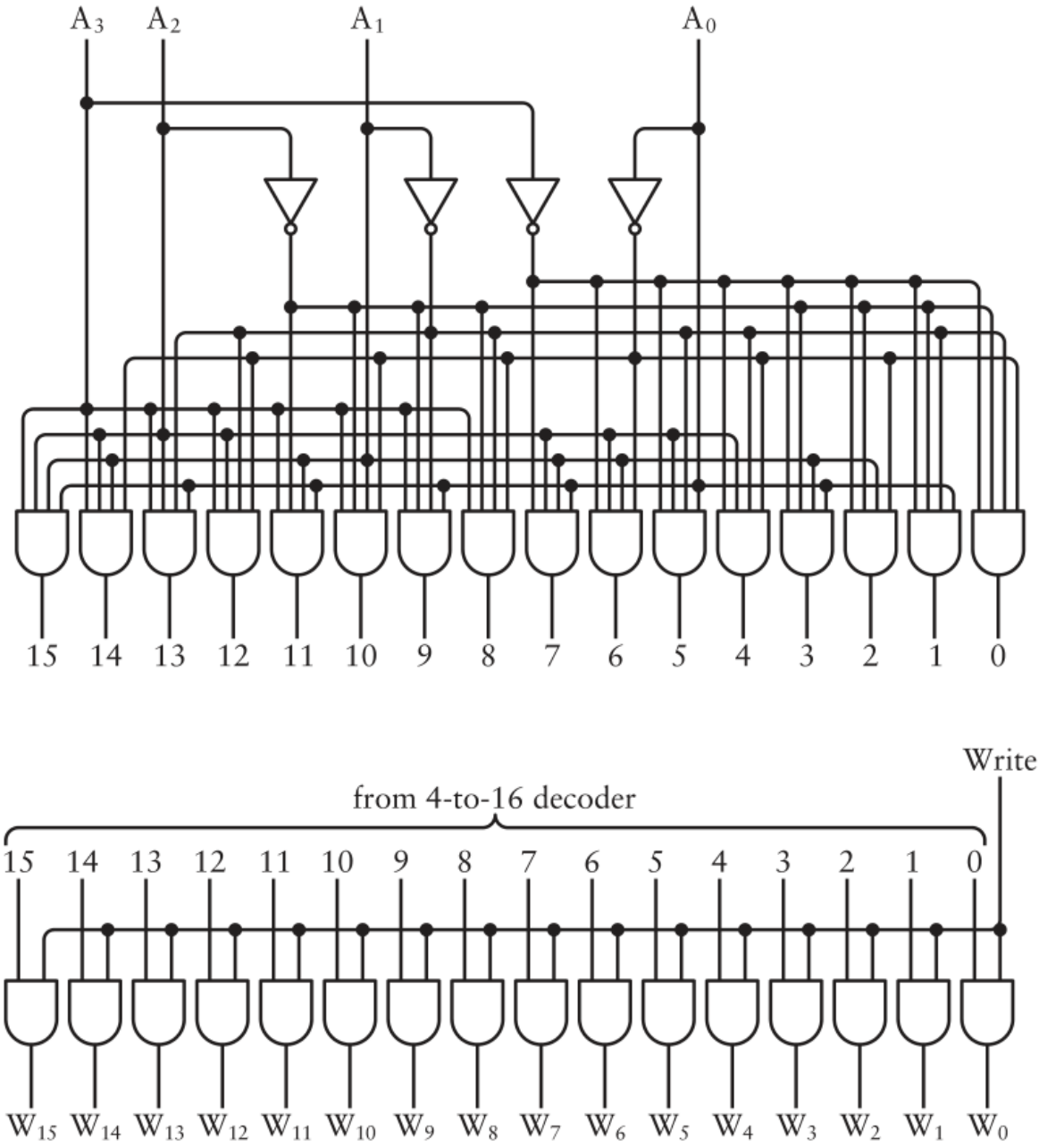


Пока не будем рассматривать Data Out. В каждом байте сигналы Write соединены, потому что байт записывается целиком. Слева они обозначены W_0-W_{15} и соответствуют 16 возможным адресам.

Data In соединены иначе. В каждой строке старший бит байта расположен слева, младший - справа. Соответствующие биты всех байтов объединены. То, что все байты получают одинаковые Data In, неважно: записан будет лишь байт, у которого Write равен 1.

Для выбора одного из 16 байтов нужен четырёхбитный адрес, поскольку четыре бита дают 16 значений. Требуется преобразовать этот адрес в нужный сигнал Write. Это назначение декодера 4 в 16.

Перед нами самый сложный декодер в книге:

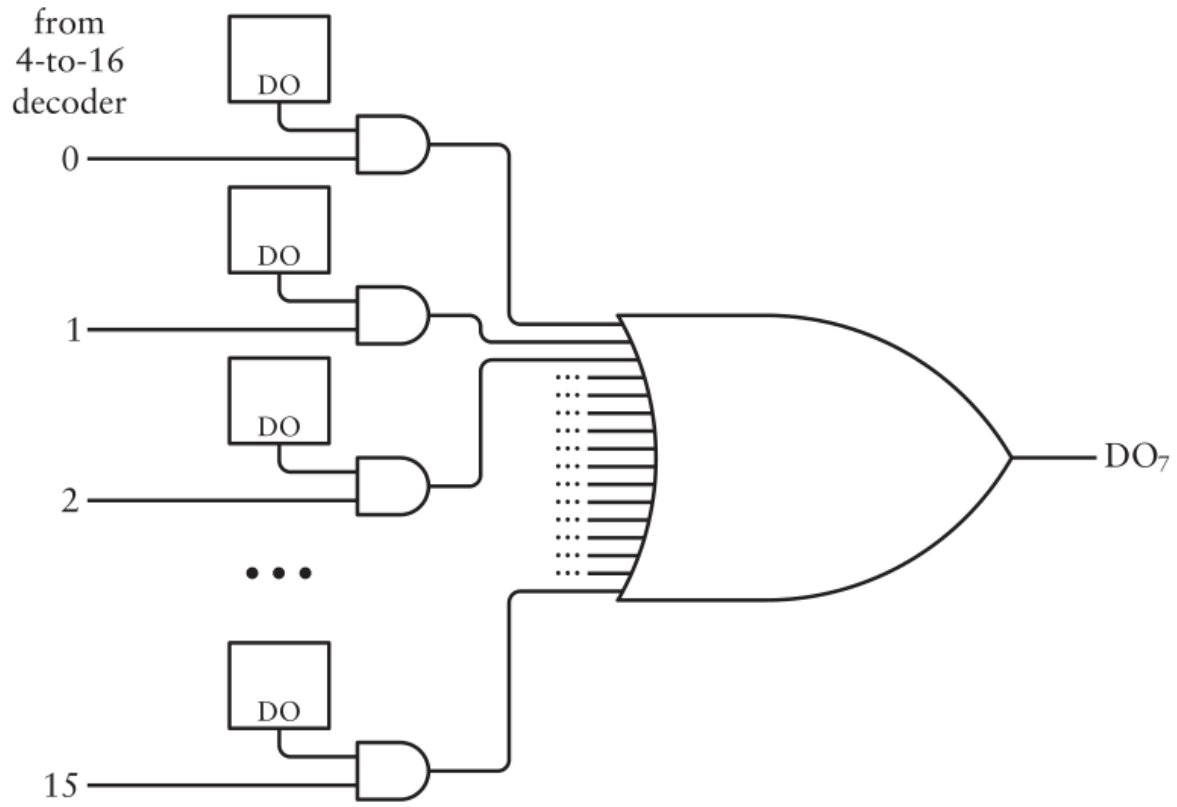


Каждый из 16 вентилей И имеет четыре входа, соответствующие четырём сигналам Address или их инверсиям. Выходы подписаны числами, совпадающими со значениями четырёх адресных битов.

Декодер помогает сформировать Write для 16 байтов массива 16 x 8. Каждый выход его вентиля И идёт на ещё один вентиль И вместе с общим Write. Именно эти сигналы записывают байт Data In в ячейки со страницы 276.

Со входами покончено. Остались Data Out всех 64 ячеек. Здесь сложность в том, что каждый из восьми столбцов надо обрабатывать отдельно. Сокращённая схема для крайнего левого столбца объединяет Data Out 16 ячеек с 16 выходами декодера 4 в 16 и выбирает одну ячейку.

Слева показаны 16 выходов декодера 4 в 16. Каждый служит входом вентиля И; второй вход - Data Out одной из 16 ячеек первого столбца. Выходы 16 вентилей И поступают на огромный 16-входовый ИЛИ. Результат DO_7 является старшим битом выходного байта:



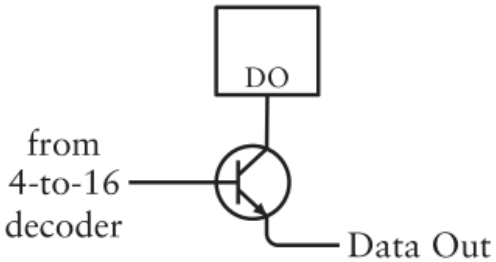
Хуже всего то, что схему пришлось бы повторить для каждого из восьми битов байта. К счастью, есть лучший способ.

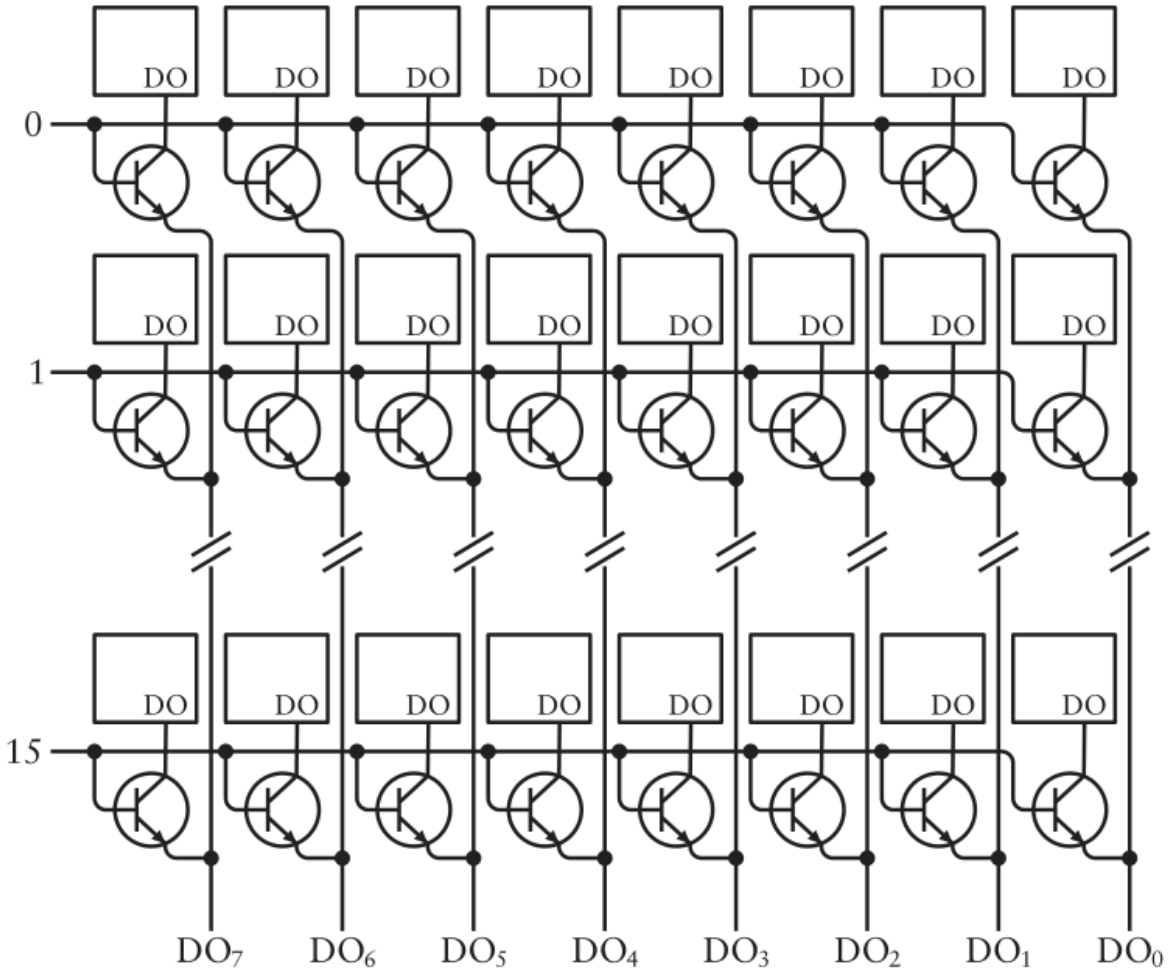
В любой момент только один из 16 выходов декодера равен 1, то есть напряжению. Остальные равны 0, то есть земле. Следовательно, только один вентиль И способен выдать 1, и то лишь если Data Out выбранной ячейки равен 1. Огромный ИЛИ нужен только для определения, равен ли единице *хотя бы один* вход.

От него можно было бы отказаться, просто соединив выходы всех вентилей И. Обычно соединять выходы логических вентилей нельзя: напряжение может оказаться напрямую соединено с землёй, вызвав короткое замыкание. Но это можно сделать с помощью транзистора.

Если сигнал декодера равен 1, Data Out с эмиттера транзистора совпадает с DO ячейки - напряжением или землёй. Если сигнал декодера равен 0, транзистор ничего не пропускает, и на его выходе нет *ничего*: ни напряжения, ни земли. Поэтому выходы целого ряда транзисторов можно соединить без короткого замыкания.

Сокращённый массив снова показан только с выходными соединениями. Слева находятся выходы декодера, снизу - восемь итоговых Data Out. Не показаны маленькие резисторы, обеспечивающие на этих линиях 1 или 0:





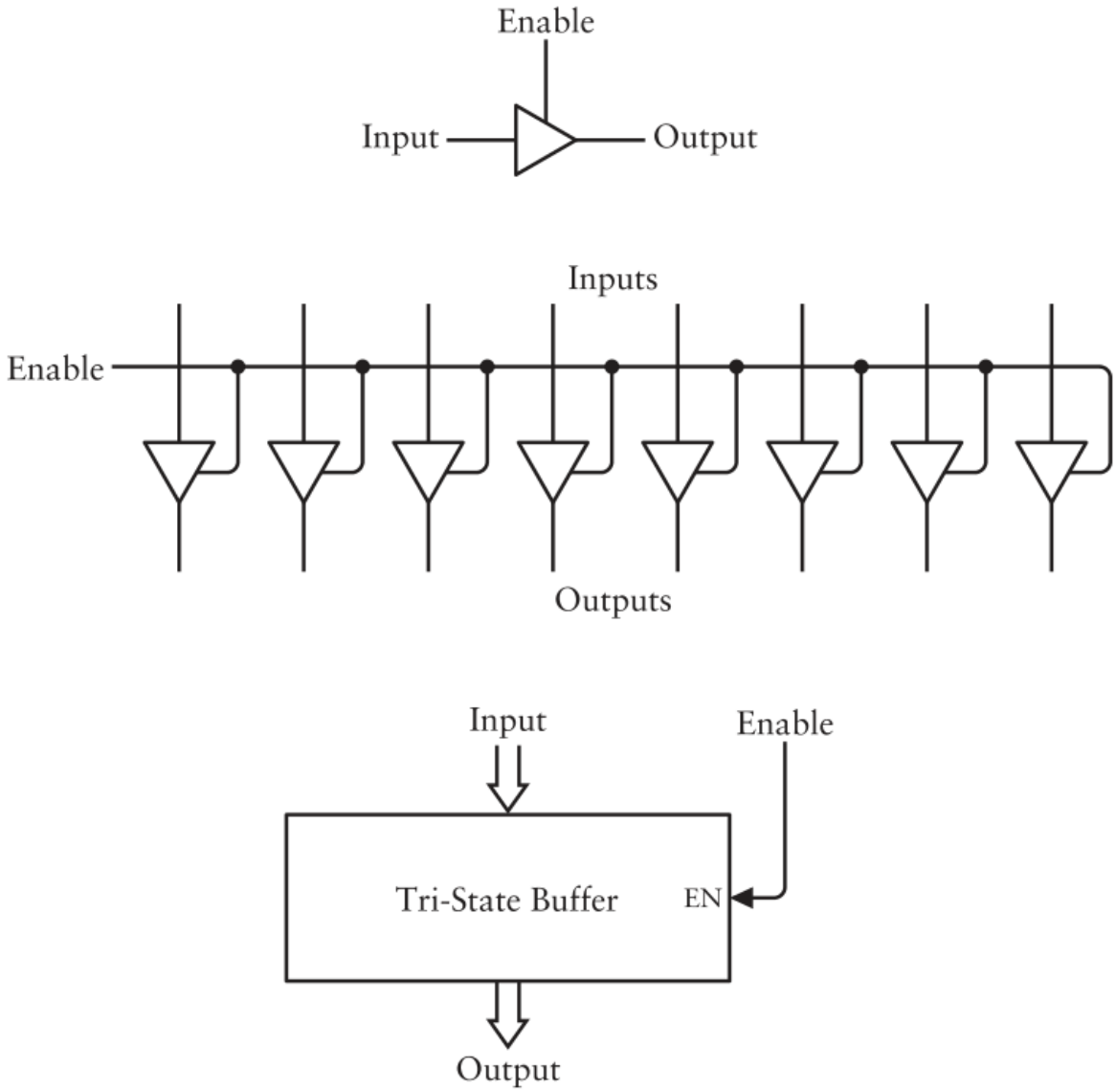
Полная схема массива 16 x 8 доступна на CodeHiddenLanguage.com.

Эти транзисторы лежат в основе *трёхсостоятельного буфера*. Его выход может находиться в одном из трёх состояний: земля, означающая логический 0; напряжение, означающее логическую 1; либо полное отсутствие сигнала - ни земли, ни напряжения, словно выход ни к чему не подключён.

Условное обозначение похоже на буфер с дополнительным сигналом Enable. При Enable = 1 Output совпадает с Input. Иначе выход, как говорят, «плавает», будто никуда не подключён.

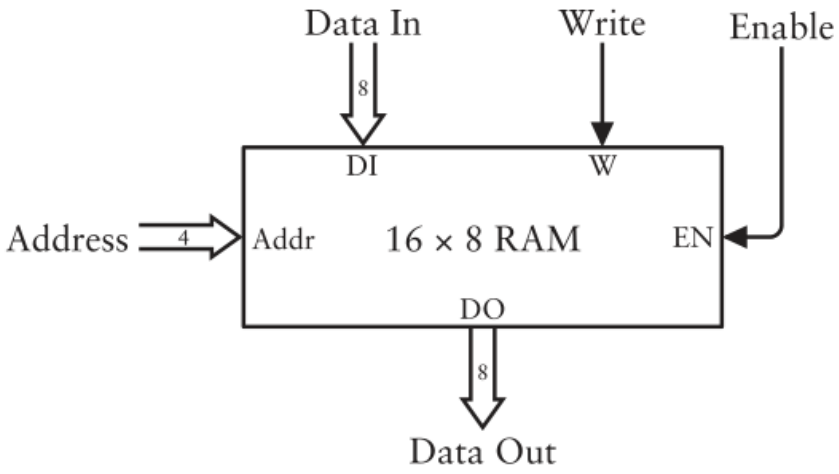
Трёхсостоятельный буфер позволяет нарушить правило, запрещающее соединять выходы вентилей. Выходы нескольких таких буферов можно объединить без короткого замыкания, если в каждый момент разрешён только один.

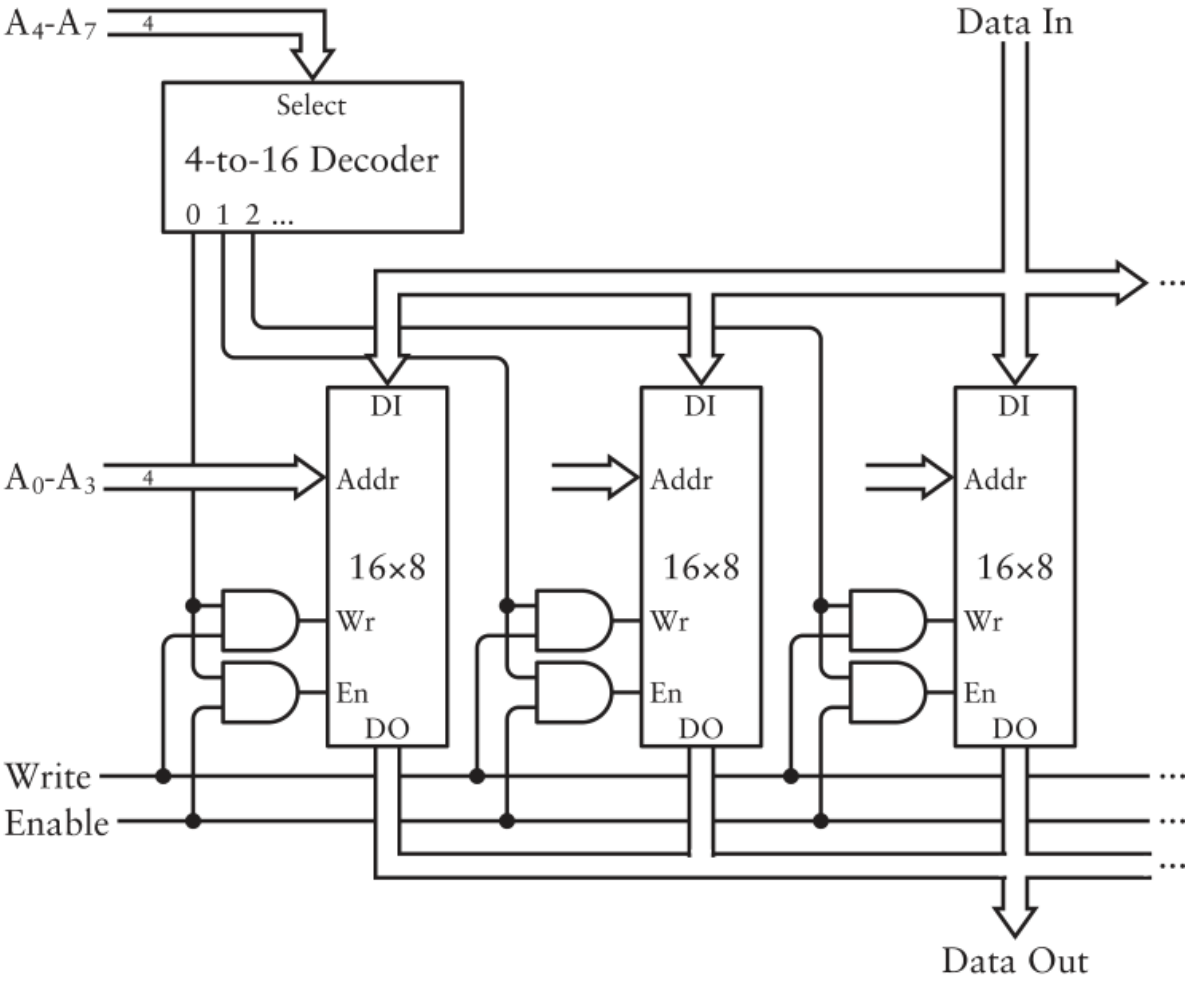
Обычно они полезнее в сборке, обрабатывающей целый байт одним сигналом Enable:



Если в последующих схемах не хватит места для полного названия, прямоугольник будет подписан Tri-State или TRI.

Мы увидели, как трёхсостоятельные буферы выбирают один из 16 байтов массива. Добавим самому массиву 16 x 8 вход Enable:



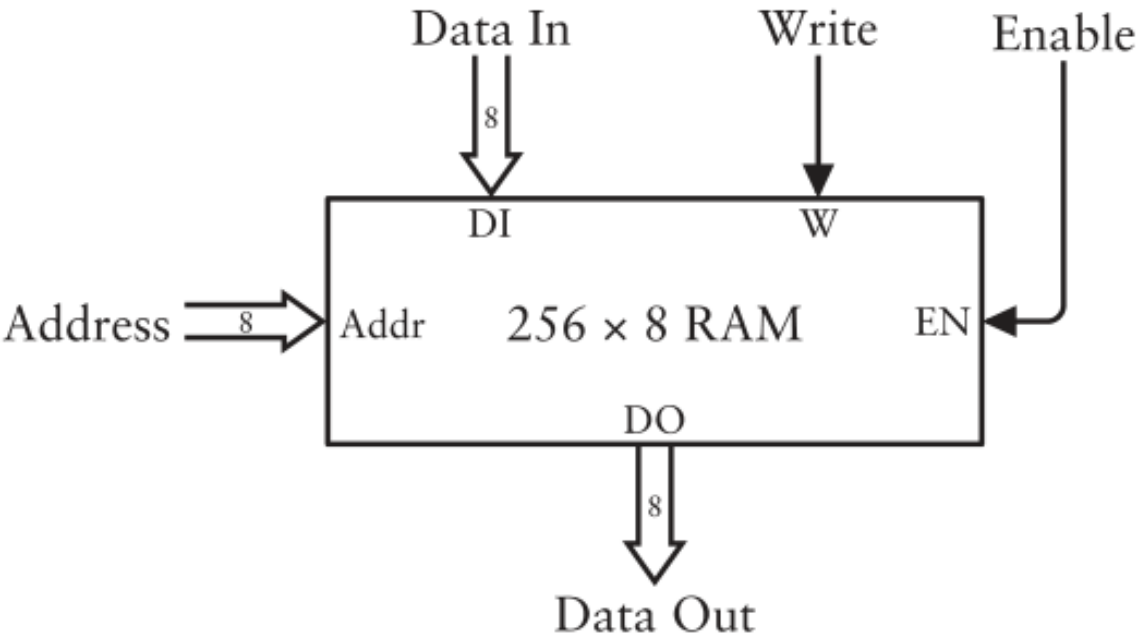


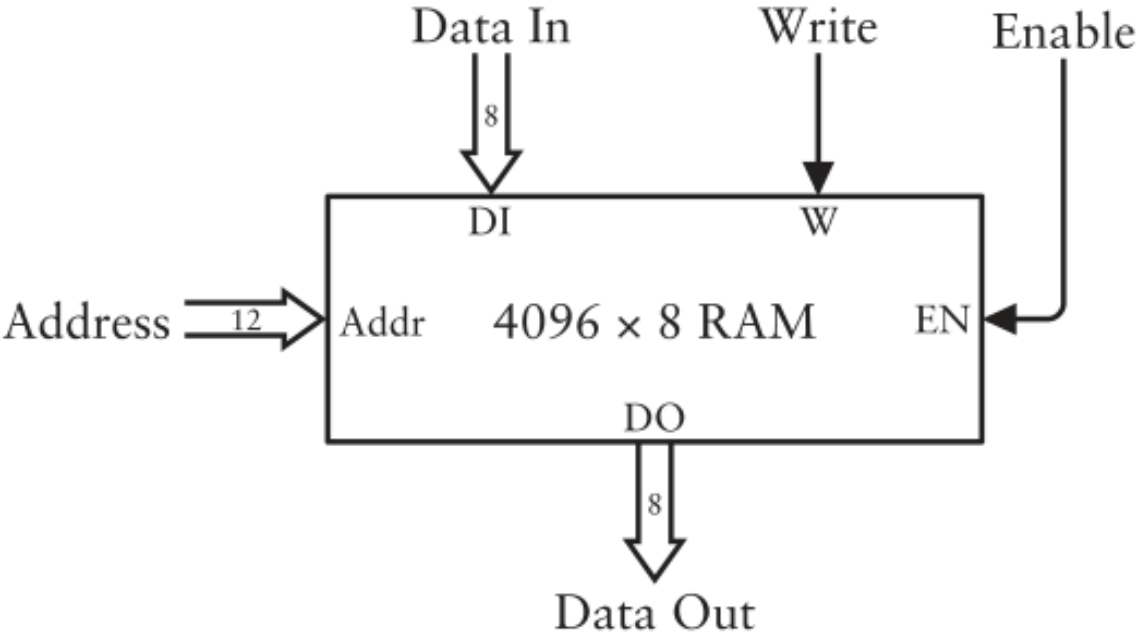
Если Enable равен 1, Data Out представляет байт по указанному адресу. При Enable = 0 на Data Out нет сигнала.

Мы построили схему на 16 байтов. Теперь удвоим её. Нет, учетверим. Нет, увеличим в восемь раз. Нет-нет, сразу в 16 раз!

Для этого нужны 16 массивов 16 x 8.

На рисунке показаны только три массива из 16. Их входы Data In общие. Data Out безопасно соединены благодаря трёхсостоятельным буферам. Есть два набора четырёхбитных адресов: A_0-A_3 адресуют байты внутри всех 16 массивов, а A_4-A_7 поступают как Select на декодер 4 в 16. Он определяет, какой массив получит Write и Enable:

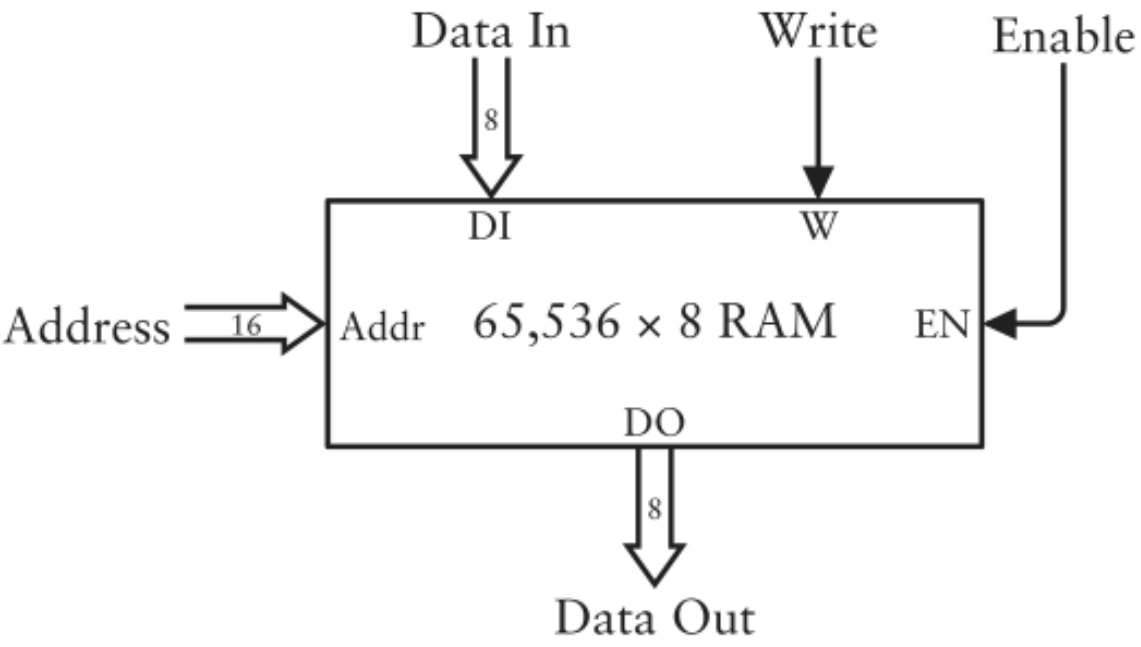




Ёмкость выросла в 16 раз до 256 байтов. Схему можно заключить в прямоугольник. Адрес теперь восьмибитный. Массив на 256 байтов напоминает почтовое отделение с 256 ящиками, в каждом из которых лежит своё однобайтовое значение, возможно, но не обязательно более полезное, чем рекламная почта.

Повторим. Возьмём 16 массивов 256 x 8 и ещё один декодер 4 в 16, управляемый следующими четырьмя адресными битами. Ёмкость снова увеличивается в 16 раз, до 4096 байтов. Адрес теперь имеет ширину 12 бит.

Ещё раз: потребуются 16 массивов 4096 x 8 и очередной декодер 4 в 16. Адрес вырастет до 16 бит, а ёмкость составит 65 536 байтов:



Продолжать можно и дальше, но здесь мы остановимся.

Число хранимых значений непосредственно связано с числом адресных битов. Без Address хранится одно значение. Четыре адресных бита дают 16 значений, шестнадцать - 65 536:

```
Число значений в массиве RAM = 2^(число входов Address)
```

RAM на 65 536 байтов называют также памятью на 64 килобайта, 64K или 64KB. Сначала это может озадачить: по какой странной арифметике 65 536 превращается в 64 килобайта?

Значение 2^{10} равно 1024, и его обычно называют одним килобайтом. Приставка «кило», от греческого *khilioi*, «тысяча», чаще используется в метрической системе: килограмм равен 1000 граммов, километр - 1000 метров. Но здесь килобайт равен 1024, а не 1000 байтов.

Метрическая система основана на степенях 10, двоичные числа - на степенях 2, и этим двум рядам не сойтись. Степени 10: 10, 100, 1000, 10 000, 100 000 и так далее. Степени 2: 2, 4, 8, 16, 32, 64 и так далее. Ни одна целая степень 10 не равна целой степени 2.

Иногда они всё же близки. Число 1000 довольно близко к 1024:

```
2^10 ≈ 10^3
```

Ничего магического здесь нет. Одна степень 2 приблизительно равна одной степени 10, и эта маленькая особенность позволяет удобно говорить «килобайт», имея в виду 1024 байта.

Не следует говорить, что массив 64K хранит 64 тысячи байтов. Он хранит больше: 65 536. Чтобы звучать осведомлённо, говорят «64K», «64 килобайта» или «шестьдесят пять тысяч пятьсот тридцать шесть».

Каждый дополнительный адресный бит удваивает память:

Объём	Байты	Степень
1 КБ	1 024	$2^{10} \approx 10^{3}$
2 КБ	2 048	2^{11}
4 КБ	4 096	2^{12}
8 КБ	8 192	2^{13}
16 КБ	16 384	2^{14}
32 КБ	32 768	2^{15}
64 КБ	65 536	2^{16}
128 КБ	131 072	2^{17}
256 КБ	262 144	2^{18}
512 КБ	524 288	2^{19}
1 024 КБ	1 048 576	$2^{20} \approx 10^{6}$

Числа килобайтов слева также являются степенями 2. По той же логике 1024 килобайта называют *мегабайтом*, сокращённо МВ. Греческое *megas* означает «большой». Удвоение продолжается:

Объём	Байты	Степень
1 МБ	1 048 576	$2^{20} \approx 10^{6}$
2 МБ	2 097 152	2^{21}
4 МБ	4 194 304	2^{22}
8 МБ	8 388 608	2^{23}
16 МБ	16 777 216	2^{24}
32 МБ	33 554 432	2^{25}
64 МБ	67 108 864	2^{26}
128 МБ	134 217 728	2^{27}
256 МБ	268 435 456	2^{28}
512 МБ	536 870 912	2^{29}
1 024 МБ	1 073 741 824	$2^{30} \approx 10^{9}$

Греческое *gigas* означает «гигант», поэтому 1024 мегабайта называют *гигабайтом*, GB.

А *терабайт*, ТВ, от греческого *teras*, «чудовище», равен 2^{40} байтам, приблизительно 10^{12}, или 1 099 511 627 776 байтам. Килобайт - примерно тысяча байтов, мегабайт - миллион, гигабайт - миллиард, терабайт - триллион.

Поднимаясь в области, где бывали немногие: *петабайт* равен 2^{50} байтам, или 1 125 899 906 842 624 байтам, приблизительно 10^{15}, квадриллиону. *Эксабайт* равен 2^{60} байтам, или 1 152 921 504 606 846 976 байтам, приблизительно 10^{18}, квинтиллиону.

Для ориентира: настольные компьютеры времени первого издания этой книги, 1999 года, обычно имели 32, 64 или иногда 128 МБ RAM. Во время написания второго издания, в 2021 году, обычными стали 4, 8 или 16 ГБ. Не путайтесь: пока речь не шла о хранилищах, сохраняющих данные без питания, включая жёсткие и твердотельные диски SSD; здесь обсуждается только RAM.

В разговоре числа сокращают. Имеющий 65 536 байтов скажет: «У меня 64K, и я гость из 1980 года». Имеющий 33 554 432 байта скажет: «У меня 32 мега». А владелец 8 589 934 592 байтов: «У меня 8 гигов, и речь не о концертах».

Иногда говорят о килобитах и мегабитах, но применительно к памяти это редко. Почти всегда объём памяти измеряется байтами. Килобиты и мегабиты чаще встречаются при передаче данных по проводу или воздуху, например в описании широкополосного соединения: «килобит в секунду», «мегабит в секунду».

Теперь вы умеете мысленно строить RAM любого размера, но мы остановились на 65 536 байтах.

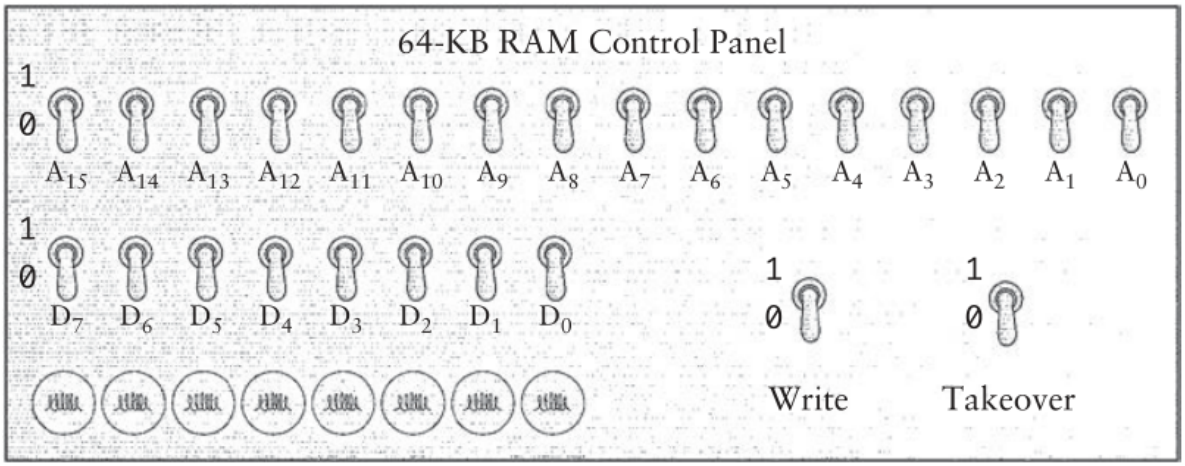
Почему 64 КБ, а не 32 или 128? Потому что 65 536 - красивое круглое число: 2^{16}. У этого массива шестнадцатибитный адрес, ровно два байта. В шестнадцатеричной записи адреса идут от 0000h до FFFFh.

В персональных компьютерах около 1980 года 64 КБ действительно были обычны, но память была устроена не совсем так. Память на триггерах точнее называется статической памятью с произвольным доступом, SRAM. К 1980 году стала преобладать динамическая RAM, DRAM. Каждая ячейка DRAM требует лишь одного транзистора и одного конденсатора. Конденсатор содержит два разделённых проводника и способен хранить электрический заряд, но не бесконечно. Поэтому заряды DRAM обновляются тысячи раз в секунду.

И SRAM, и DRAM являются *энергозависимой памятью*: для сохранения данных постоянно требуется электричество. При отключении питания память забывает всё, что знала.

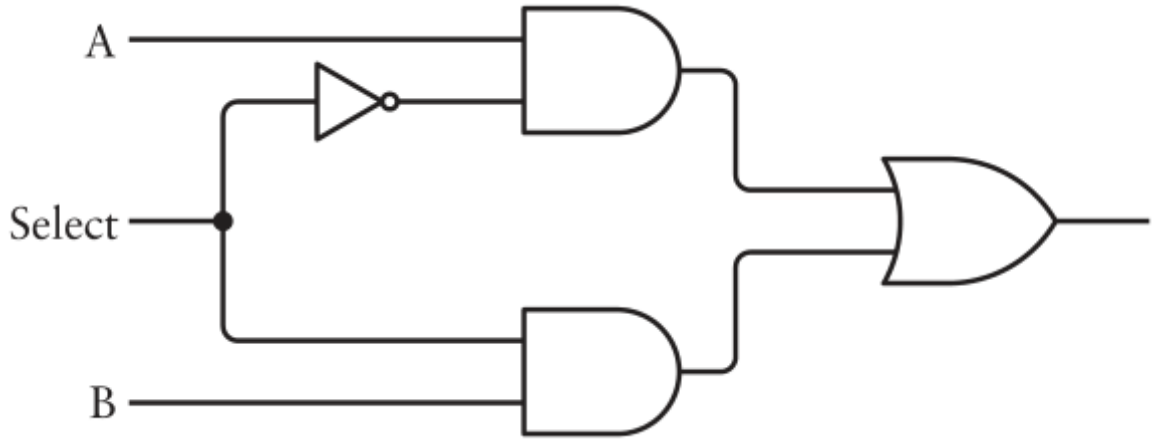
Полезно иметь панель управления, позволяющую распоряжаться 64 КБ: записывать значения и просматривать их.

У такой панели 16 выключателей адреса, восемь выключателей задаваемого байта, отдельный Write и восемь лампочек для отображения байта:



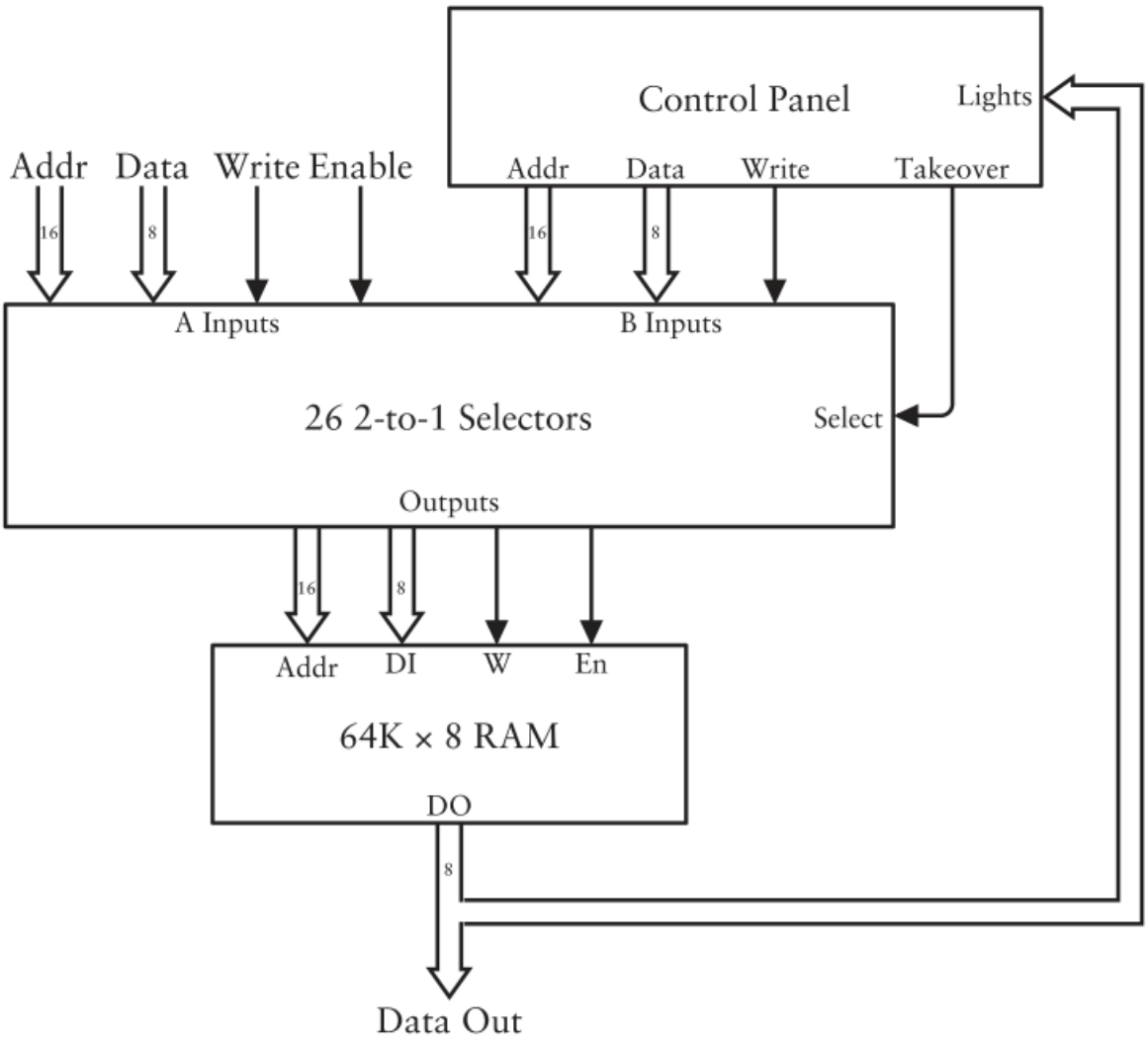
Все выключатели показаны в положении 0. Добавлен также Takeover, «перехват управления». Он позволяет другим схемам пользоваться той же памятью. При Takeover = 0 остальные органы панели ничего не делают. При Takeover = 1 панель получает исключительный контроль над памятью.

Реализация Takeover требует набора селекторов 2 в 1, довольно простых по сравнению с большими декодерами этой главы:



При Select = 0 выход ИЛИ совпадает с A. При Select = 1 выбирается B.

Нужно 26 селекторов 2 в 1: шестнадцать для Address, восемь для Data In и ещё два для Write и Enable:



Когда Takeover разомкнут, Address, Data In, Write и Enable массива 64K x 8 поступают от внешних сигналов, показанных вверху слева. Когда Takeover замкнут, Address, Data In и Write поступают с выключателей панели, а Enable устанавливается в 1. В обоих случаях Data Out памяти возвращается к восьми лампочкам панели и, возможно, идёт куда-нибудь ещё.

При замкнутом Takeover шестнадцатью адресными выключателями можно выбрать любой из 65 536 адресов. Лампочки показывают хранимый там байт. Восемью выключателями Data задаётся новое значение, а Write записывает его в память.

Массив 64K x 8 с панелью позволяет держать под рукой любые 65 536 восьмибитных значений. При этом оставлена возможность другой схеме пользоваться сохранёнными значениями и записывать новые.

Если этот сценарий кажется невероятным, взгляните на обложку знаменитого январского номера *Popular Electronics* за 1975 год, где рассказывалось о первом домашнем компьютере Altair 8800:

Retro AdArchives/Alamy Stock Photo

Передняя сторона компьютера - панель, состоящая из одних выключателей и лампочек. Если сосчитать длинный ряд выключателей внизу, окажется, что их шестнадцать.

Совпадение? Не думаю.

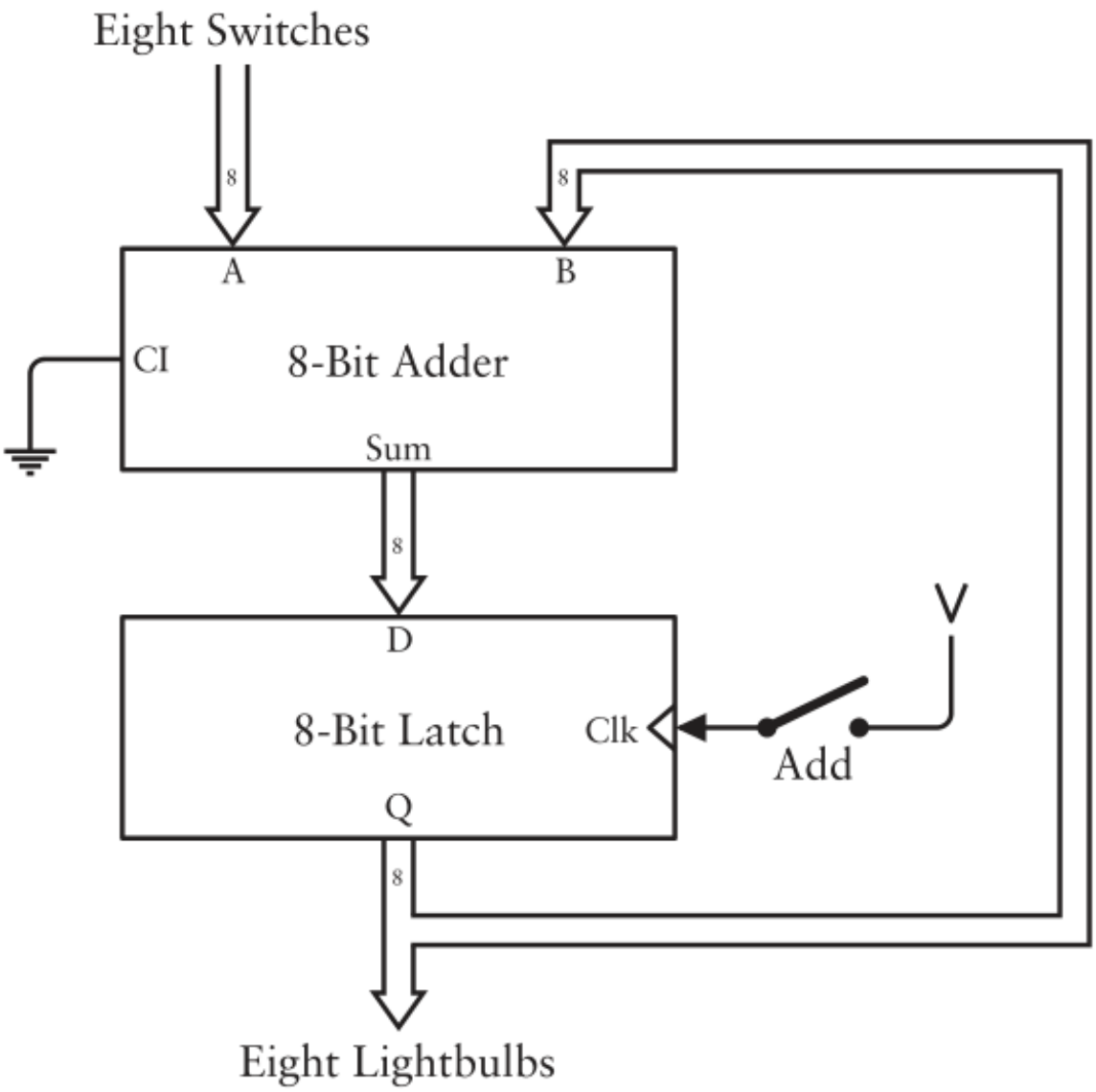
Глава 20. Автоматизация арифметики

Человеческий вид часто проявляет удивительную изобретательность и трудолюбие, одновременно оставаясь глубоко ленивым. Совершенно ясно: люди не любят работать. Наша неприязнь к труду настолько сильна, а изобретательность настолько остра, что мы готовы потратить бесчисленные часы на проектирование и постройку хитроумных устройств, способных сэкономить несколько минут рабочего дня. Мало какая фантазия сильнее радует человека, чем картина отдыха в гамаке, пока только что построенная новомодная машина стрижёт газон.

Чертежей автоматической газонокосилки здесь, боюсь, не будет. Но в этой главе мы начнём создавать всё более сложные машины, автоматизирующие сложение и вычитание. Звучит не слишком потрясающе, однако постепенно они станут настолько универсальными, что смогут решать почти всякую задачу, использующую сложение, вычитание и булеву логику, а таких задач очень много.

Вместе с совершенством приходит сложность, поэтому временами будет трудно. Никто не осудит вас, если вы бегло просмотрите мучительные подробности. Иногда захочется взбунтоваться и пообещать никогда больше не обращаться к электронике с математическими задачами. Но потерпите: в итоге мы изобретём машину, которую законно можно назвать *компьютером*.

Последний сумматор встречался на странице 231 главы 17. Он включал восьмибитную защёлку с запуском по фронту, накапливающую текущий итог чисел, введённых восемью выключателями:



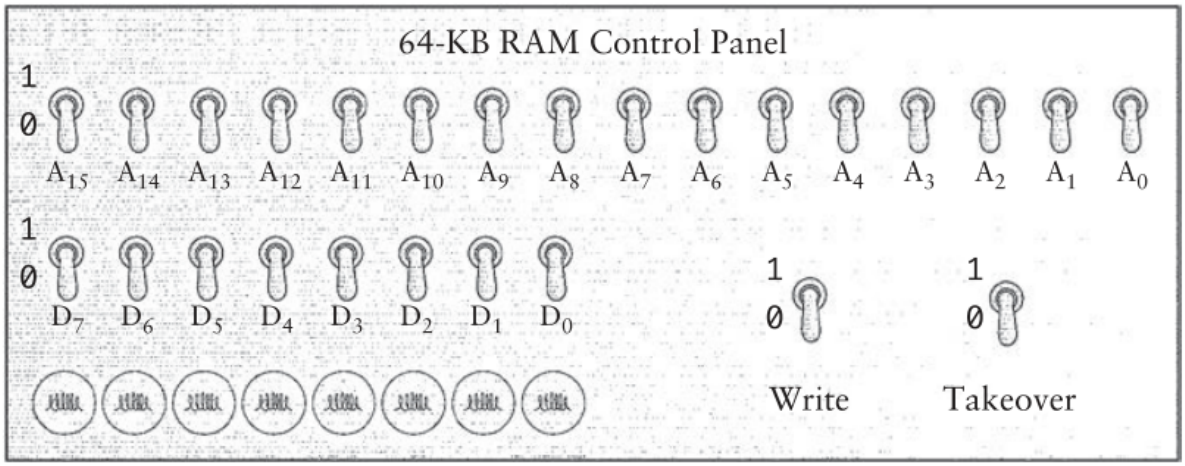
Напомним, восьмибитная защёлка хранит восьмибитное значение. Сначала её содержимое и выходы равны нулю. Выключателями вводится первое число. Сумматор прибавляет его к нулевому выходу защёлки, поэтому результат совпадает с введённым числом. Нажатие Add сохраняет его в защёлке и включает соответствующие лампочки. Поскольку защёлка запускается фронтом, новое значение не сохраняется, пока Add не будет отпущен и нажат снова.

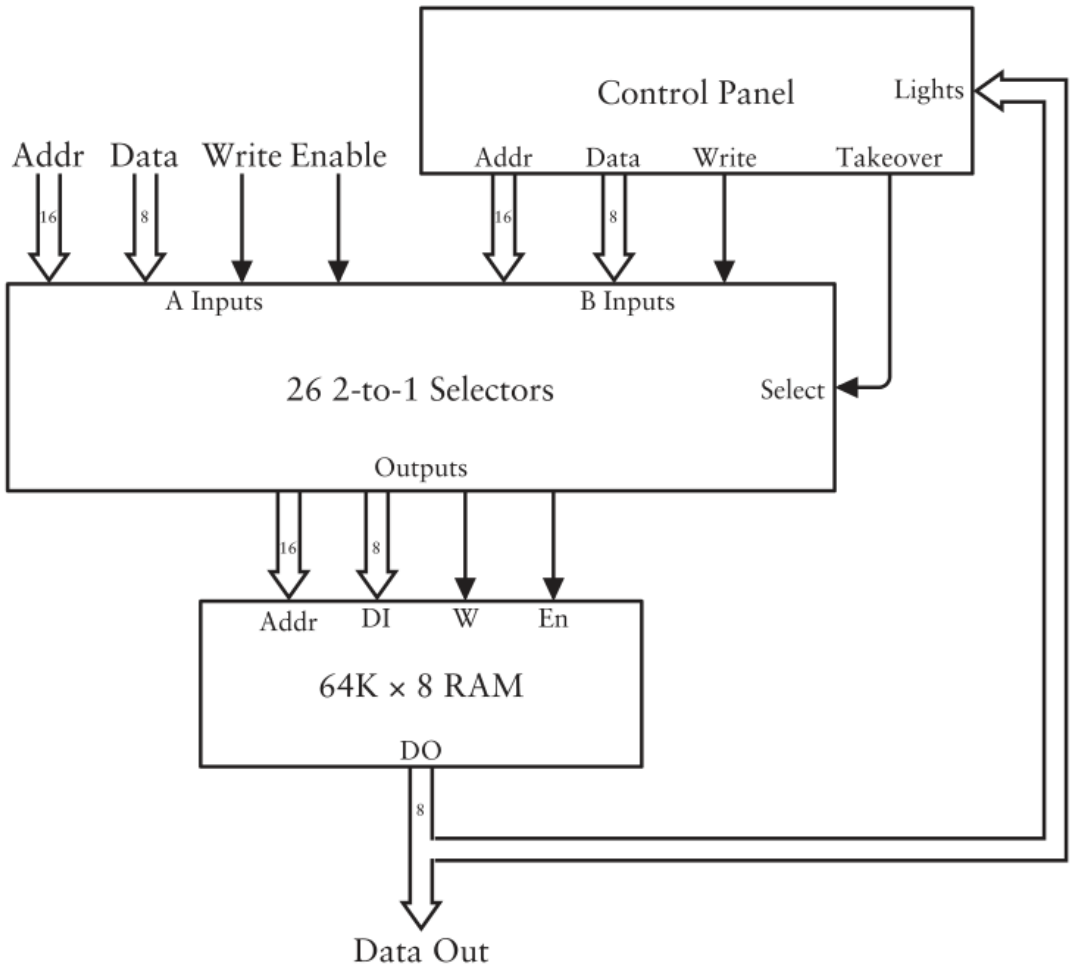
Затем на выключателях устанавливается второе число. Сумматор складывает его с числом в защёлке. Новое нажатие Add сохраняет и показывает итог. Так можно сложить целый ряд чисел и видеть текущую сумму. Ограничение очевидно: восемь лампочек не показывают результат больше 255.

Защёлку, накапливающую текущий итог, часто называют *аккумулятором*. Но аккумулятор не обязан только накапливать: обычно это защёлка, сначала хранящая одно число, а затем результат его арифметического или логического объединения с другим.

Главная проблема такой машины тоже очевидна. Допустим, надо сложить список из 100 байтов. Вы упорно вводите каждое число и получаете сумму, а затем обнаруживаете, что несколько исходных значений были неверны. К тому же вы начинаете сомневаться, не допустили ли ошибки при наборе. Приходится повторять всё сначала.

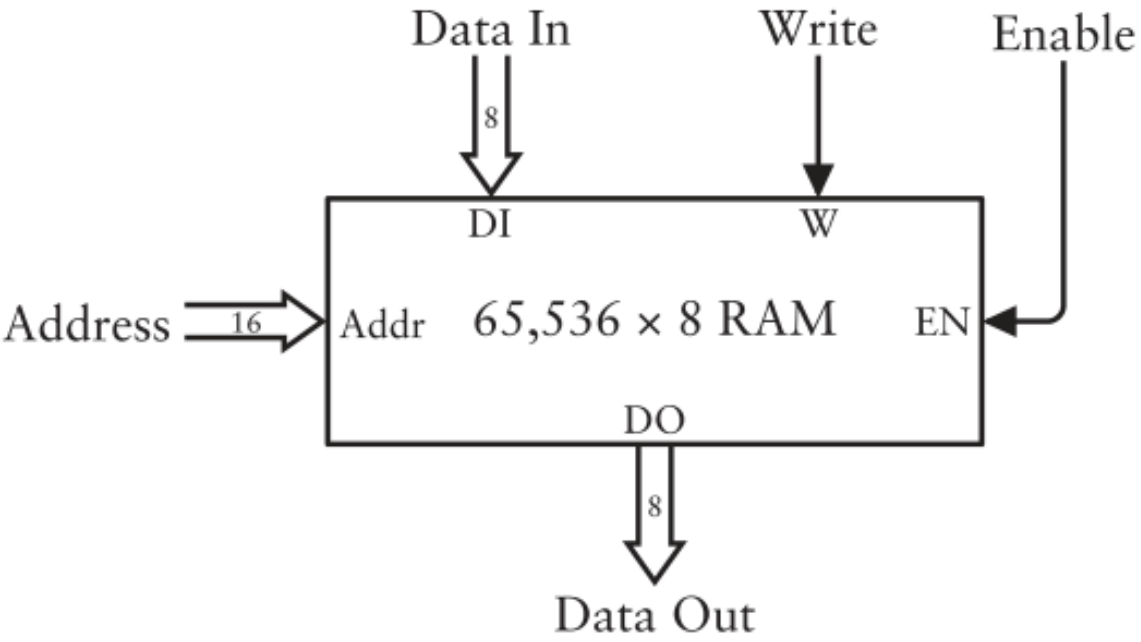
Возможно, есть решение. В предыдущей главе мы построили массив RAM на 64 КБ и панель с выключателями и лампочками. Переключатель Takeover буквально позволяет перехватить всё чтение и запись этой памяти:

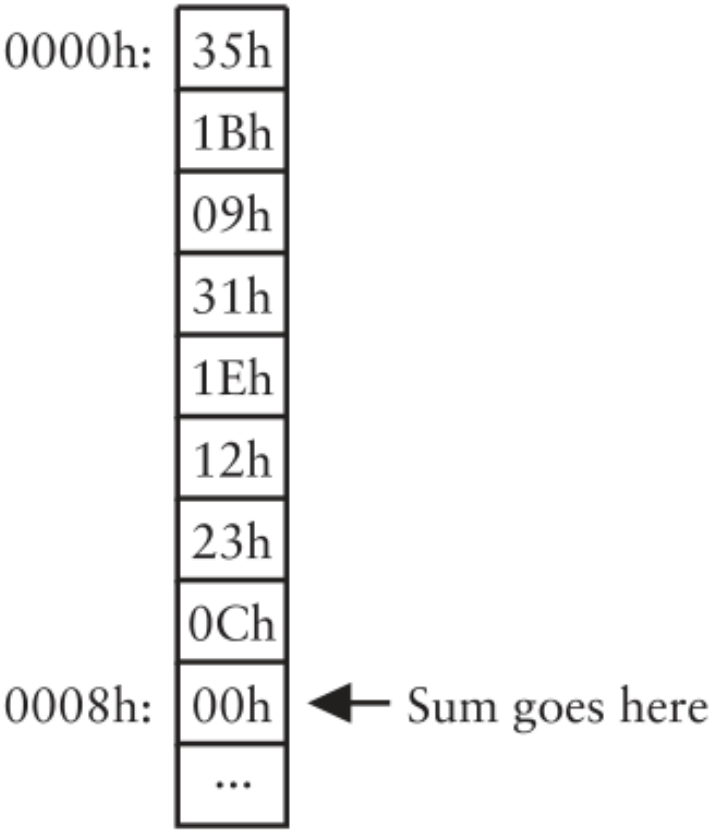




Если бы 100 байтов были сначала введены в RAM, а не прямо в сумматор, проверить их и исправить ошибки было бы гораздо легче.

Чтобы упростить дальнейшие рисунки, массив 64K x 8 будет показываться отдельно, без панели и 26 селекторов, необходимых для перехвата чтения и записи:





Наличие панели или эквивалентного средства записи и чтения подразумевается. Иногда не будет показан и Enable: тогда можно считать, что трёхсостоятельные буферы Data Out разрешены.

Предположим, требуется сложить восемь байтов: 35h, 1Bh, 09h, 31h, 1Eh, 12h, 23h и 0Ch. Калькулятор Windows или macOS в режиме программиста покажет сумму E9h, но мы построим оборудование, которое вычислит её самостоятельно.

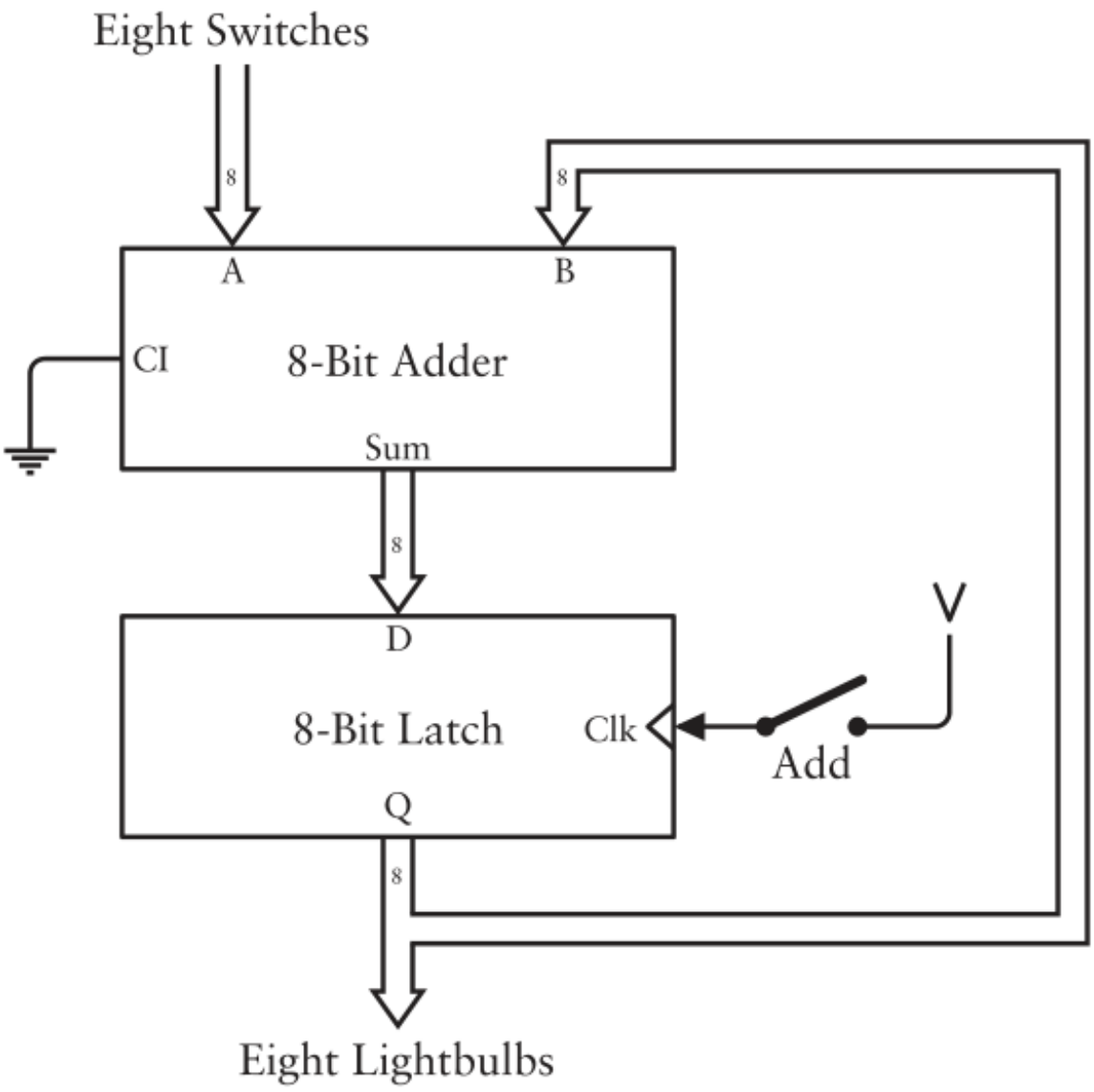
Через панель эти байты записываются в RAM начиная с адреса 0000h.

Их расположение последовательно:

Адрес	**Байт**
0000h	35h
0001h	1Bh
0002h	09h
0003h	31h
0004h	1Eh
0005h	12h
0006h	23h
0007h	0Ch
0008h	00h, сюда должна попасть сумма

Отныне участок памяти будет изображаться как ряд прямоугольников. Каждый содержит байт, адрес указан слева. Не все адреса требуется подписывать, потому что они идут последовательно. Справа расположены комментарии. В данном случае нужно сложить первые восемь байтов, а сумму записать в первую ячейку со значением 00h, то есть по адресу 0008h.

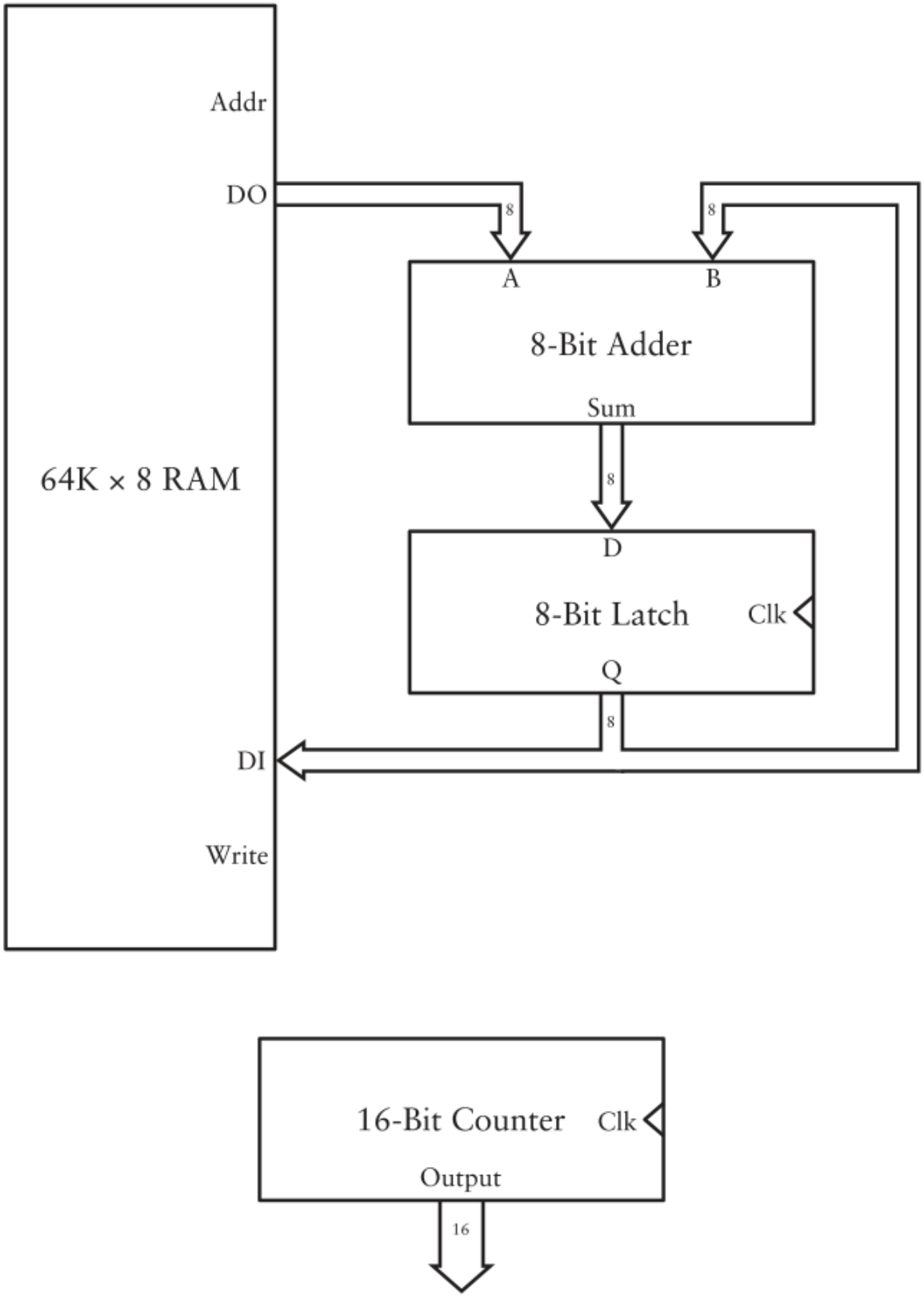
Можно хранить и 100 чисел - по адресам 0000h-0063h. Теперь надо соединить RAM с накапливающим сумматором из главы 17:



Выключатели и лампочки сумматора больше не нужны, потому что они есть на панели RAM. Выключатели заменяются сигналами Data Out памяти.

Комментарии справа на рисунке памяти не являются её содержимым. Они лишь поясняют человеку назначение байтов: первые восемь следует сложить, а итог поместить в первую свободную нулевую ячейку. Адреса можно восстанавливать по положению прямоугольников, поскольку каждый следующий отличается от предыдущего на единицу.

Вместо лампочек выход защёлки направляется на Data In памяти:



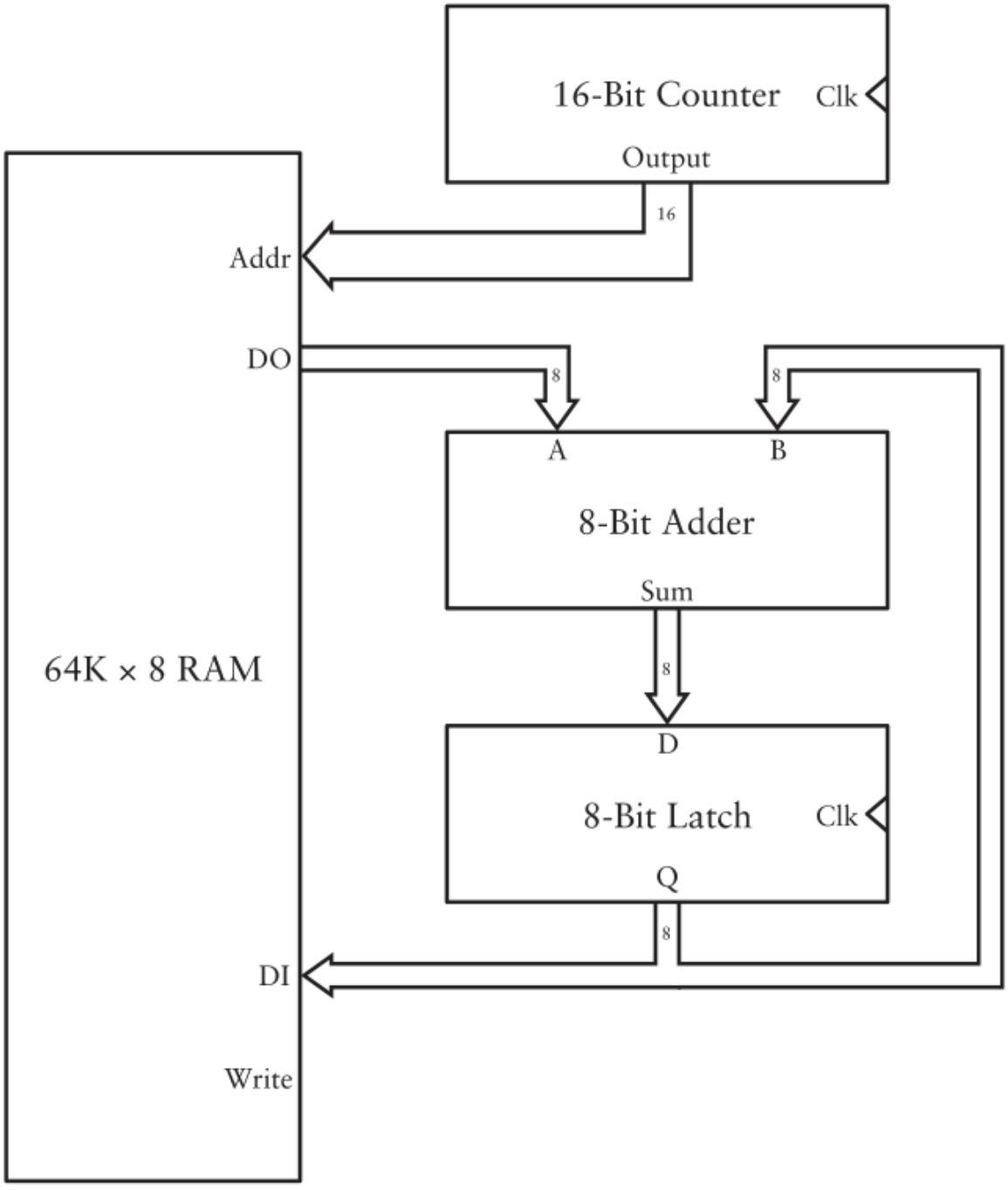
Нескольких частей всё ещё не хватает. Не показано, что подключено к Clock защёлки, необходимому для сохранения суммы, и к Write RAM, необходимому для записи окончательного результата. У памяти также отсутствует шестнадцатибитный адрес.

Address должен последовательно увеличиваться: 0000h, 0001h, 0002h, 0003h и так далее. Это работа счётчика из каскада триггеров, подобного схеме на странице 237 главы 17.

Сигналы Data Out памяти идут прямо на один вход восьмибитного сумматора. На втором находится текущий итог с защёлки. Sum одновременно возвращается на вход защёлки и на Data In RAM, хотя запись в каждое из этих мест должна происходить в свой строго определённый момент.

Широкий тракт от счётчика символизирует 16 бит вместо 8. Счётчик подаёт Address на RAM.

Эту машину назовём *автоматическим накапливающим сумматором*:



Добавив адресный счётчик, мы ввели ещё один отсутствующий Clock. Но основные восьми- и шестнадцатибитные тракты уже определены. Остались три сигнала:

На рисунке можно проследить весь путь данных. Шестнадцатибитный счётчик выбирает адрес, RAM выдаёт восьмибитный байт, сумматор складывает его с байтом защёлки, Sum возвращается в защёлку, а её выход одновременно служит текущим итогом и данными, которые впоследствии можно записать в RAM. Ни один из этих трактов сам по себе не определяет, *когда* должно произойти действие.

• Clock счётчика;

• Clock защёлки;

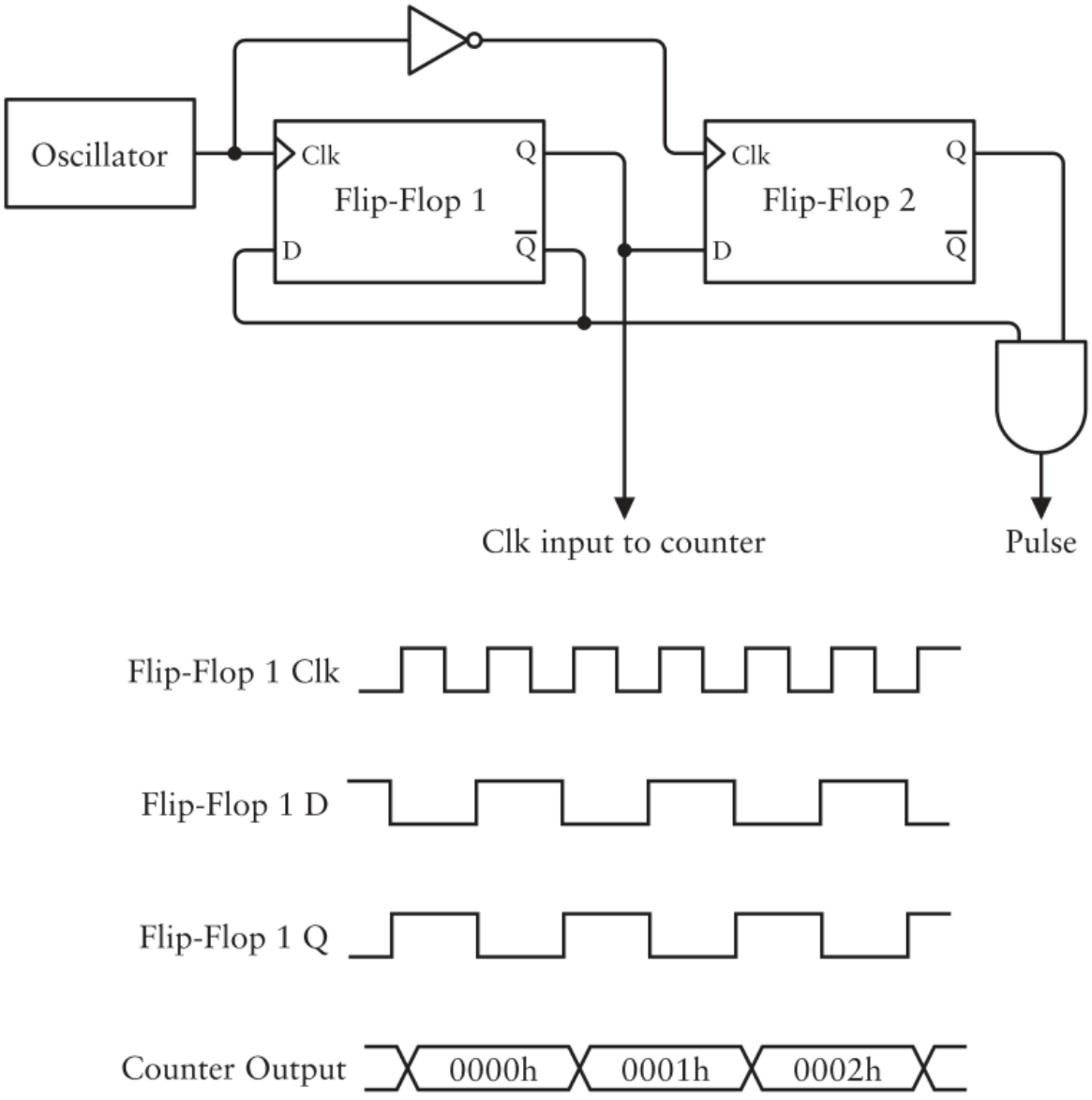
• Write памяти с произвольным доступом.

Такие сигналы вместе называют *управляющими*. Они часто оказываются самой сложной частью схемы и должны быть согласованы и синхронизированы.

Clock счётчика увеличивает адрес от 0000h к 0001h, затем к 0002h и так далее. Адрес выбирает байт памяти, который поступает в сумматор вместе с выходом защёлки. Затем Clock защёлки должен сохранить новую сумму. В реальности доступ к памяти и сложение требуют времени, поэтому Clock защёлки должен возникать после Clock счётчика, а следующий Clock счётчика - после Clock защёлки.

Иными словами, каждый цикл разделяется на фазы. Сначала выбирается следующий адрес. Затем нужно подождать, пока RAM выдаст байт, а сумматор распространит сигналы через свои вентили и сформирует устойчивую сумму. Только после этого разрешено защёлкнуть результат. Следующий адрес также нельзя менять слишком рано.

Соединим два триггера:

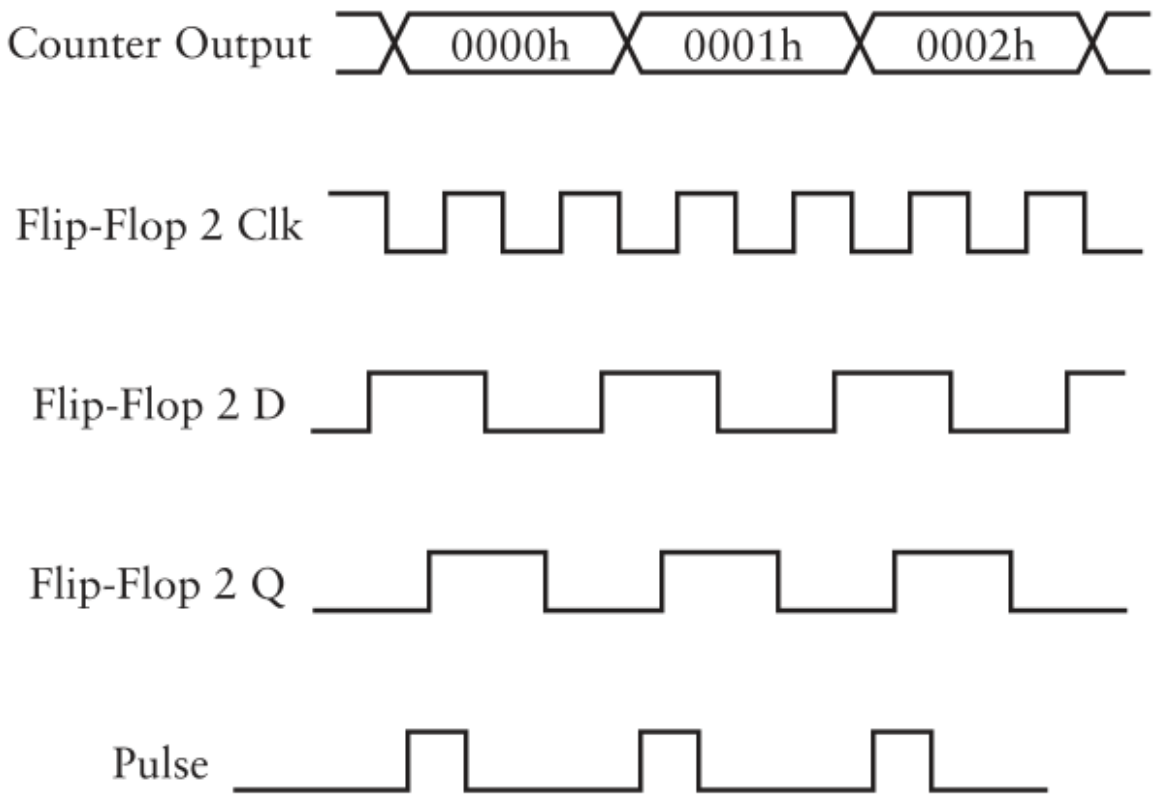


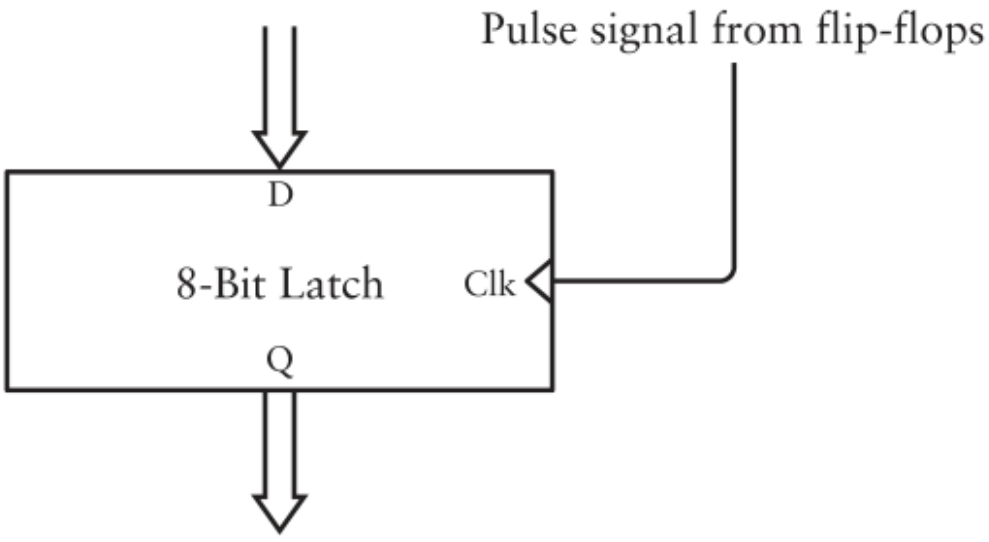
Генератор слева просто чередует 0 и 1. Это может быть быстрый кварцевый генератор или обычная кнопка.

Первый триггер делит частоту пополам, как в конце главы 17. Его Q становится Clock счётчика, который увеличивается при каждом переходе 0 -> 1. Нижняя строка временной диаграммы показывает изменение выхода счётчика.

На временной диаграмме Q первого триггера чередуется вдвое медленнее генератора. Счётчик меняет значение лишь на положительных фронтах этого Q: 0000h, 0001h, 0002h и далее. Между фронтами адрес остаётся неизменным.

Clock второго триггера противоположен Clock первого, а D получает Q первого. Поэтому Q второго смещён на один цикл относительно Q первого. В диаграмму для сравнения включён выход счётчика:





Вентиль И объединяет Q первого и Q второго триггеров. Его выход мы будем называть *Pulse*, «импульс». Pulse становится Clock защёлки.

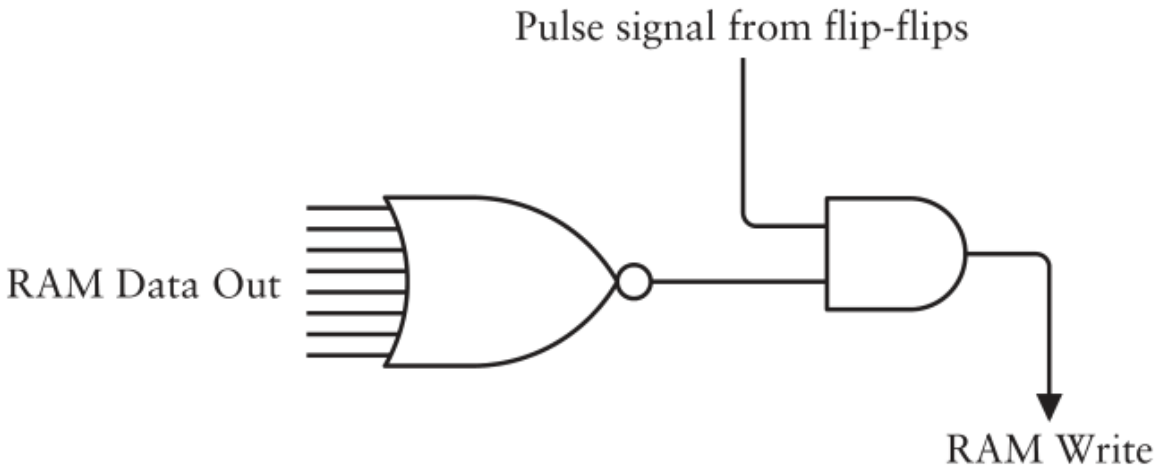
Из-за смещения двух Q импульс возникает лишь в коротком промежутке после того, как счётчик уже перешёл к новому адресу. На диаграмме видно, что Pulse никогда не совпадает с изменением Counter Output. Именно эта задержка создаёт время, необходимое памяти и сумматору.

Нужно дать счётчику достаточно времени для адресации памяти, а данным из памяти - для сложения с прежней суммой до сохранения в защёлке. Всё должно стабилизироваться, чтобы избежать сбоев. Это условие выполнено: пока Pulse равен 1, выход счётчика постоянен.

Остаётся Write памяти. Накопленная сумма должна быть записана в первую ячейку, содержащую 00h. Её можно обнаружить, подав Data Out RAM на восьмивходовый ИЛИ-НЕ.

Восемь нулей означают, что каждый вход ИЛИ-НЕ равен 0, и только тогда его выход становится равен 1. Вентиль И пропускает этот признак на RAM Write лишь одновременно с Pulse, так что запись происходит один раз в синхронизированный момент.

Выход ИЛИ-НЕ равен 1, когда все отдельные Data Out равны 0. Его можно объединить с Pulse:



Интерактивная версия полного автоматического сумматора доступна на CodeHiddenLanguage.com.

Пока машина ничем не останавливается. Пока генератор чередует 0 и 1, счётчик продолжает обращаться к памяти. Если последующие байты равны 00h, схема запишет готовую сумму и туда. В конце счётчик достигнет FFFFh, перевернётся в 0000h и начнёт сложение заново, прибавляя все байты к уже вычисленной сумме.

Для управления полезна кнопка Clear. Она может очистить адресный счётчик, защёлку и триггеры управляющих сигналов, остановив их работу. После этого можно ввести новые значения и снова запустить сложение.

Без Clear генератор не знает, что найден первый нулевой байт и итог уже записан. Он продолжает перебирать адреса, поэтому каждый последующий нулевой байт тоже получает сумму. После переворота адреса машина не просто повторяет прежний расчёт: она начинает с суммы, уже оставшейся в аккумуляторе, и прибавляет к ней содержимое памяти ещё раз.

Но важнейшая проблема - ограничение байтами 00h-FFh, то есть 0-255. Наши восемь значений дают E9h, или 233. Если добавить девятый байт 20h, получится 109h. Это уже два байта: восьмибитный сумматор выдаст только 09h, и именно оно попадёт в память. Carry Out сообщит о превышении FFh, но машина его игнорирует.

Предположим, сумматор должен проверять поступления на банковский счёт.

В США денежная сумма записывается, например, как $1.25. Для хранения в байте её умножают на 100 и получают 125 центов, то есть 7Dh. Значит, байт ограничивает сумму всего лишь $2.55.

Два байта дают диапазон 0000h-FFFFh, 0-65 535, или до $655.35. Но нужны и отрицательные суммы, например при овердрафте. В дополнительном коде положительный предел шестнадцатибитного значения равен 7FFFh, 32 767, а отрицательный - 8000h, -32 768. Денежный диапазон составит -$327.68...$327.67.

Попробуем три байта. В дополнительном коде диапазон 800000h-7FFFFFh равен -8 388 608...8 388 607, то есть -$83 886.08...$83 886.07. Для большинства текущих счетов это безопаснее.

Как научить автоматическую машину складывать трёхбайтовые значения? Простое решение - расширить память до 24 бит и построить 24-битные сумматор и защёлку. Но, возможно, массив 64K x 8 и восьмибитный сумматор уже построены и заменить их трудно.

В восьмибитной памяти 24-битное значение хранится в трёх последовательных ячейках. Но в каком направлении?

Это не косметическая подробность. При обработке число будет извлекаться по одному байту, и порядок адресов определяет, какой байт сумматор увидит первым. Если первым идёт старший, переносы от младших частей нельзя использовать естественным последовательным образом без дополнительного хранения и управления.

Сумма $10 000.00 равна 1 000 000 центов, или 0F4240h, то есть байтам 0Fh, 42h и 40h. Их называют старшим, средним и младшим. Возможны два порядка хранения.

Трёхбайтовый формат даёт 24 бита, однако один из них в дополнительном коде занят знаком. Поэтому максимальная положительная сумма не FFFFFFh, а 7FFFFFh, а минимальная отрицательная начинается с 800000h. Выбранный денежный диапазон всё ещё конечен, но значительно практичнее двухбайтового.

Какой порядок принят? Каков освящённый временем отраслевой стандарт? К сожалению, оба. Одни компьютеры делают так, другие наоборот.

Эти способы называются *big-endian* и *little-endian*. Различие упоминалось в главе 13 при обсуждении Unicode. Термины происходят из сатирического романа Джонатана Свифта «Путешествия Гулливера», где лилипуты долго спорили, с какого конца разбивать яйцо. В компьютерной отрасли это не столько спор, сколько фундаментальное различие, с которым научились жить.

На первый взгляд big-endian разумнее: байты идут в обычном порядке записи. Little-endian выглядит задом наперёд, начиная с младшего байта.

Для значения 0F4240h два порядка в памяти выглядят так:

Смещение адреса	Big-endian	Little-endian
0	0Fh, старший	40h, младший
1	42h, средний	42h, средний
2	40h, младший	0Fh, старший

Но при последовательном сложении многобайтовых чисел удобнее начать с младшего байта: его сложение может дать перенос для следующего, более старшего.

Поэтому далее байты хранятся в little-endian, младший первым. Это относится только к памяти; в обычной шестнадцатеричной записи старший байт по-прежнему будет первым.

Будем говорить о «вкладах» и «снятиях», словно машина вычисляет банковский остаток, хотя столь же хорошо это могут быть доходы и расходы или активы и обязательства.

Начнём с вкладов $450.00 и $350.00:

```
   00 AF C8
+ 00 88 B8
-----------
   01 38 80
```

При сложении справа налево каждая пара байтов может создать перенос для следующей.

Теперь снимем $500.00, то есть 00C350h:

```
   01 38 80
- 00 C3 50
```

Как объяснялось в главе 16, вычитаемое превращается в дополнительный код и складывается. Инвертируем биты 00C350h, получая FF3CAFh, затем прибавляем 1: FF3CB0h.

```
   01 38 80
+ FF 3C B0
-----------
   00 75 30
```

В десятичном виде это 30 000 центов, или $300. Снимем ещё $500:

```
   00 75 30
+ FF 3C B0
-----------
   FF B1 E0
```

Результат отрицателен. Для определения величины снова инвертируем биты, получая 004E1Fh, и прибавляем 1: 004E20h, то есть 20 000. Остаток равен -$200.00.

Теперь поступает $2000.00, или 030D40h:

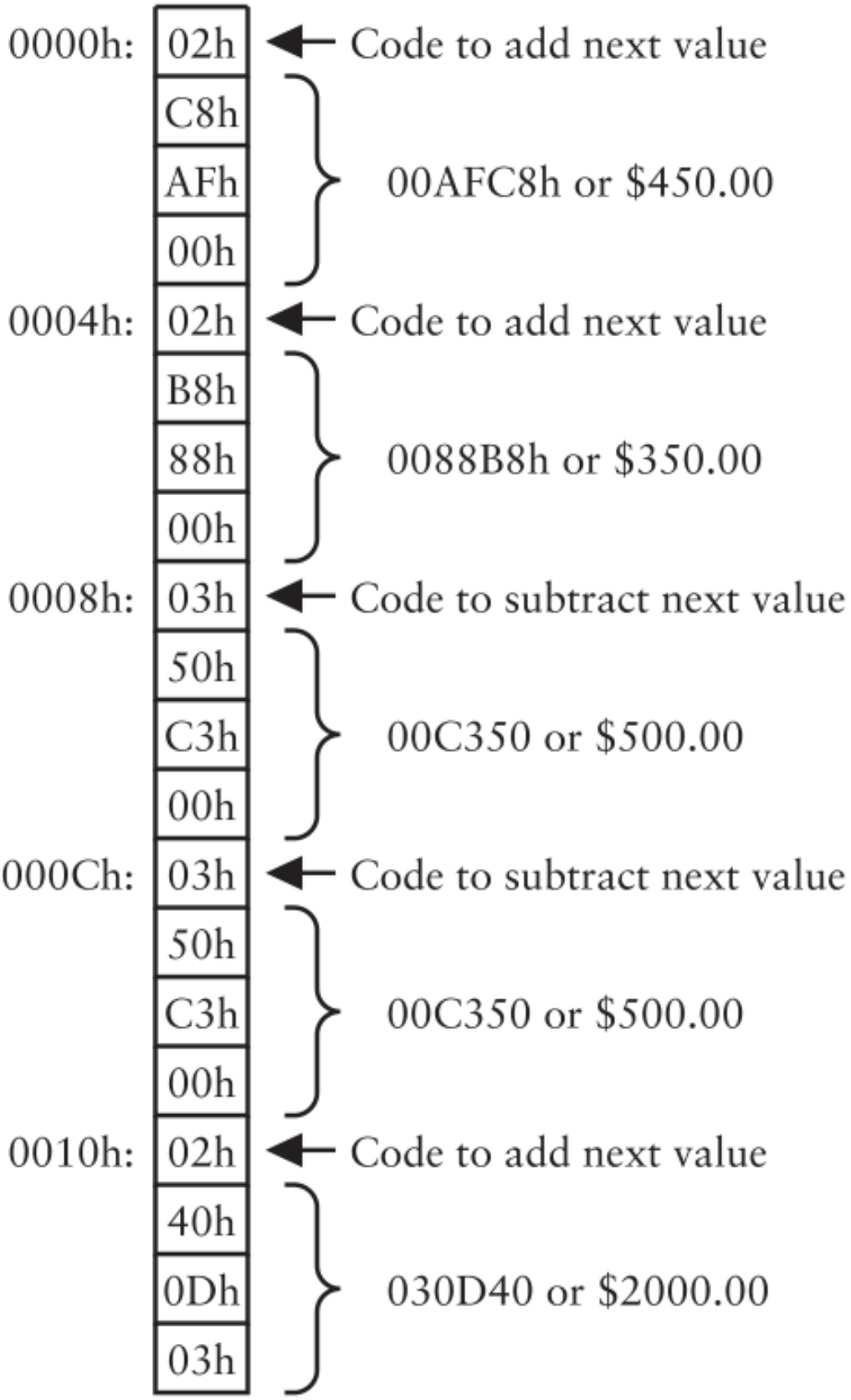
```
   FF B1 E0
+ 03 0D 40
-----------
   02 BF 20
```

Это 180 000 центов, или $1800.00. Новая машина должна складывать и вычитать трёхбайтовые значения из памяти и записывать результат обратно.

Снятие $500 должно храниться как байты 00, C3 и 50, не в дополнительном коде. Дополнение должна вычислять машина. Но если все значения положительны, как отличать вклады от снятий?

Перед каждым числом нужен код: один означает «прибавить следующие три байта», другой - «вычесть».

В памяти нашего примера это выглядит так:



Код 02h означает сложение следующего трёхбайтового значения, 03h - вычитание. Коды в некоторой степени произвольны, но не полностью.

Поскольку числа хранятся в little-endian, первый байт каждого операнда является младшим. Начало памяти можно прочитать так:

Адрес	Значение	Назначение
0000h	02h	прибавить следующее число
0001h-0003h	C8 AF 00	$450.00
0004h	02h	прибавить следующее число
0005h-0007h	B8 88 00	$350.00
0008h	03h	вычесть следующее число
0009h-000Bh	50 C3 00	$500.00
000Ch	03h	вычесть следующее число
000Dh-000Fh	50 C3 00	$500.00
0010h	02h	прибавить следующее число
0011h-0013h	40 0D 03	$2000.00

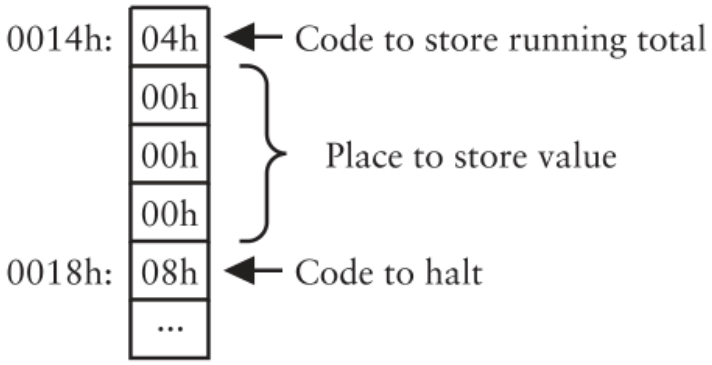
Следующий код предписывает записать текущий итог в три последующие ячейки, а последний код прекращает выполнение, чтобы машина не продолжала читать пустую память.

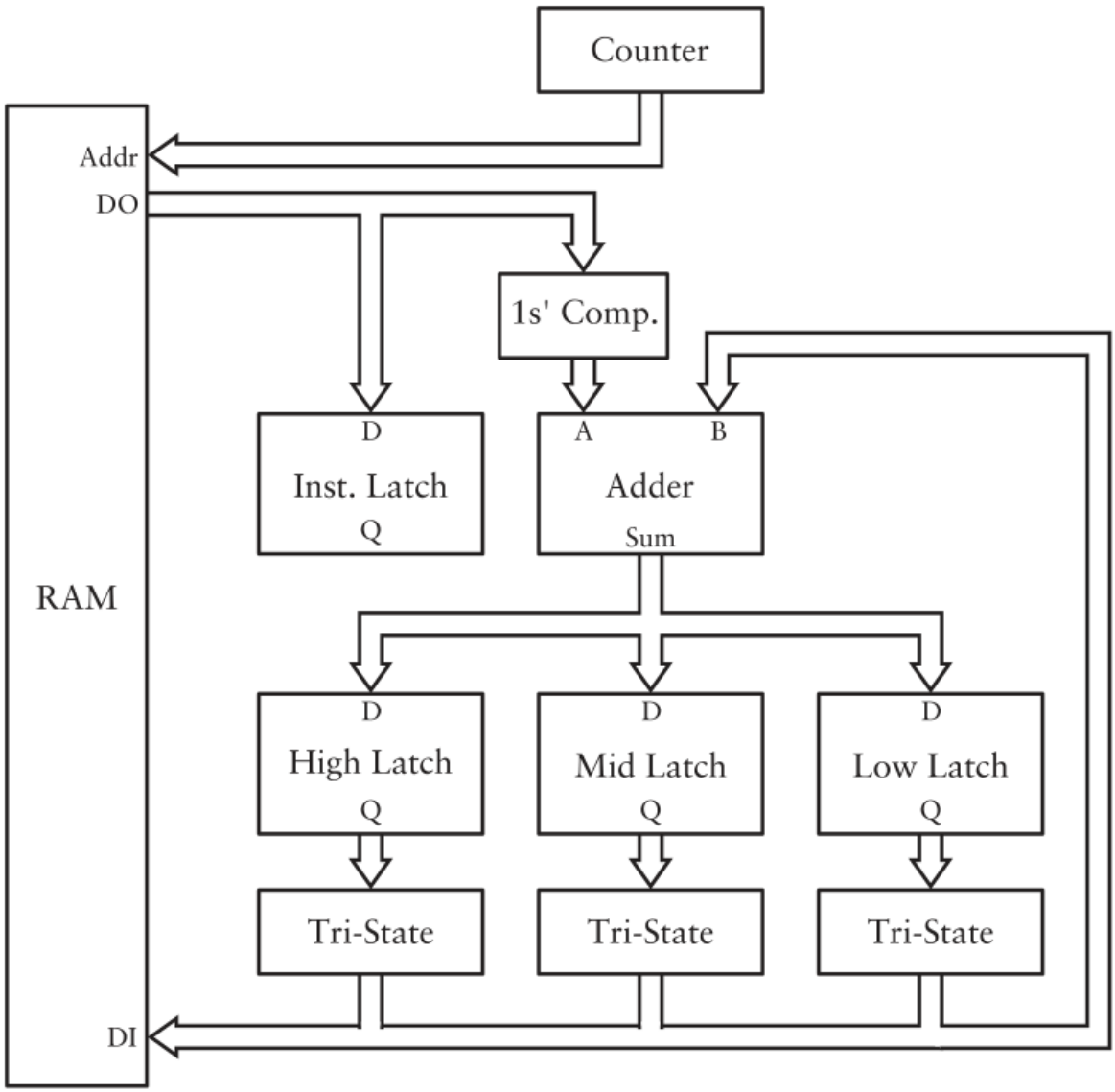
Такие значения называются *кодами инструкций*, *кодами операций* или *опкодами*. Они предписывают читающей память машине, что делать, а та выполняет сложение, вычитание или другую операцию.

Содержимое памяти теперь делится на *код* и *данные*: за каждым кодовым байтом следуют три байта данных.

Добавим код, записывающий текущий итог в следующие три ячейки, и код остановки машины.

Назовём новую машину *трёхбайтовым аккумулятором*. Как автоматический сумматор, она последовательно обращается к RAM 64K x 8 и накапливает итог восьмибитным сумматором. Но защёлок теперь четыре: одна хранит код инструкции, ещё три - текущий итог:





Чтобы не перегружать рисунок, многочисленные управляющие сигналы не показаны. Остаток главы во многом посвящён именно им. Не показаны и входы, к которым они подключаются: Clock счётчика и защёлок, Enable трёхсостоятельных буферов, Write RAM.

Как и прежде, шестнадцатибитный счётчик даёт адрес памяти, а его Clock приходит от уже рассмотренных двух триггеров.

Четыре защёлки: Inst. Latch хранит код по адресам 0000h, 0004h, 0008h и так далее; High, Mid и Low хранят три байта текущего итога. На их входы идёт Sum сумматора. Выходы проходят через три отдельных Tri-State с собственными Enable. В каждый момент разрешён только один, поэтому их выходы можно соединить и направить на Data In RAM и вход B сумматора.

Защёлка инструкции используется именно для кодов, встречающихся через каждые четыре адреса. Остальные три защёлки предназначены соответственно для старшего, среднего и младшего байтов текущего итога. Выход Sum восьмибитного сумматора подключён сразу ко входам всех трёх, но фактически значение сохраняет только та защёлка, на Clock которой приходит управляющий импульс.

Каждый блок Tri-State представляет описанный в предыдущей главе трёхсостоятельный буфер. Его Enable на рисунке опущен. При Enable = 1 выходы повторяют входы: для входного 0 получается 0, для входной 1 - 1. При Enable = 0 выходы не равны ни 0, ни 1, ни напряжению, ни земле: они вообще ни к чему не подключены. Это и есть третье состояние.

Каждый из трёх буферов имеет отдельный Enable, и в любой момент единице равен лишь один из них. Поэтому выходы можно объединить без столкновения напряжения с землёй. Управляющие сигналы выбирают, какая защёлка в данный момент подаёт свой байт одновременно на Data In памяти и вход B сумматора.

Три Enable зависят от двух младших адресных битов. Память организована строго: байт инструкции, затем младший, средний и старший байты числа. Поэтому:

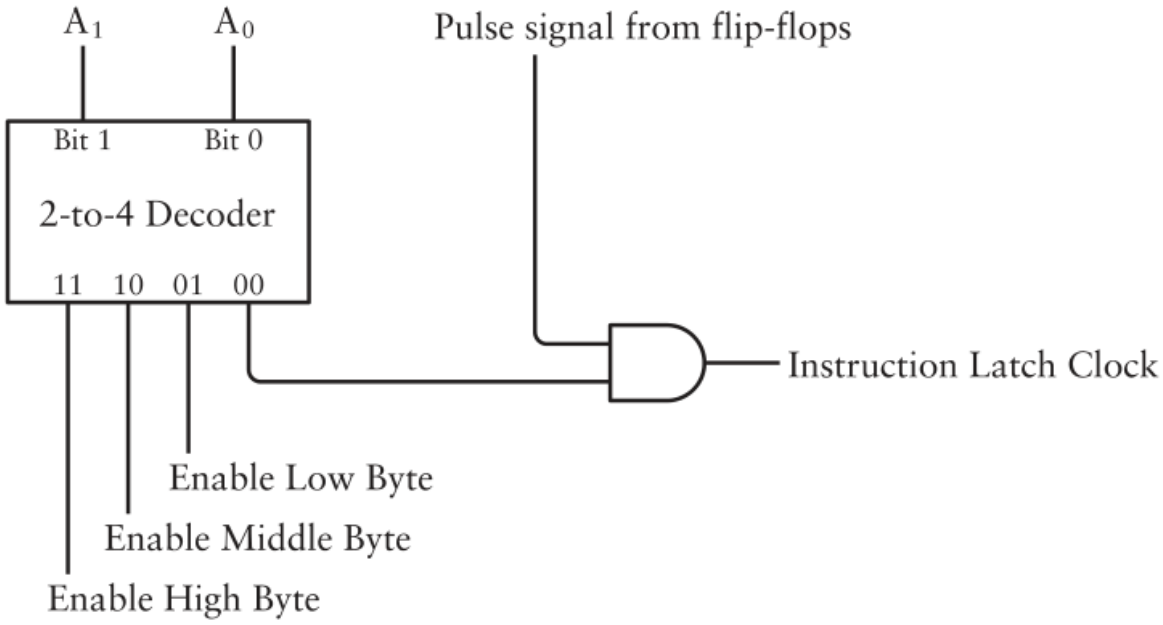
• адрес, заканчивающийся на 00, содержит код инструкции;

• 01 содержит младший байт числа;

• 10 содержит средний байт;

• 11 содержит старший байт.

Три Enable формируются декодером 2 в 4 по A_0 и A_1:



Показан и Clock защёлки инструкции: он возникает, когда $A_1A_0 = 00$ и Pulse = 1. Инструкция остаётся в защёлке, пока читаются следующие три байта.

Clock остальных трёх защёлок устроен несколько сложнее и будет показан далее. Пока проследим работу машины шаг за шагом. Все защёлки вначале очищены и не содержат значений.

Предположим, все защёлки очищены. Счётчик выдаёт 0000h; байт 02h сохраняется как инструкция. Затем адрес 0001h выбирает младший байт первого числа. Он идёт в сумматор. Блок 1s' Comp. пока ничего не делает. A_1A_0 = 01, поэтому выбран буфер Low. Поскольку Low очищен, B сумматора равен 00h. Sum совпадает с младшим байтом памяти и сохраняется в Low.

Инструкция 02h остаётся защёлкнутой всё время, пока счётчик проходит адреса 0001h, 0002h и 0003h. Поэтому управляющая логика знает, что все эти байты нужно прибавить, хотя код читался только один раз.

При адресе 0002h средний байт складывается с 00h защёлки Mid и сохраняется в ней. При 0003h то же происходит со старшим байтом и High.

Адрес 0004h содержит следующую инструкцию 02h, то есть Add. Адрес 0005h содержит младший байт второго числа и подаёт его на A сумматора.

На этом втором проходе соответствующие защёлки уже не нулевые: в них находится первое трёхбайтовое число. Поэтому каждый очередной байт из памяти складывается с соответствующим байтом текущего итога, а Sum заменяет прежнее содержимое нужной защёлки. Та же последовательность повторяется для каждой следующей группы из четырёх байтов.

Разрешён буфер Low, поэтому его значение - младший байт первого числа - поступает на B. Два байта складываются, а результат сохраняется обратно в Low. Процесс продолжается со средним и старшим байтами, затем со следующим числом.

Определены четыре инструкции:

• 02h - прибавить следующее трёхбайтовое число;

• 03h - вычесть следующее трёхбайтовое число;

• 04h - записать трёхбайтовый текущий итог в память;

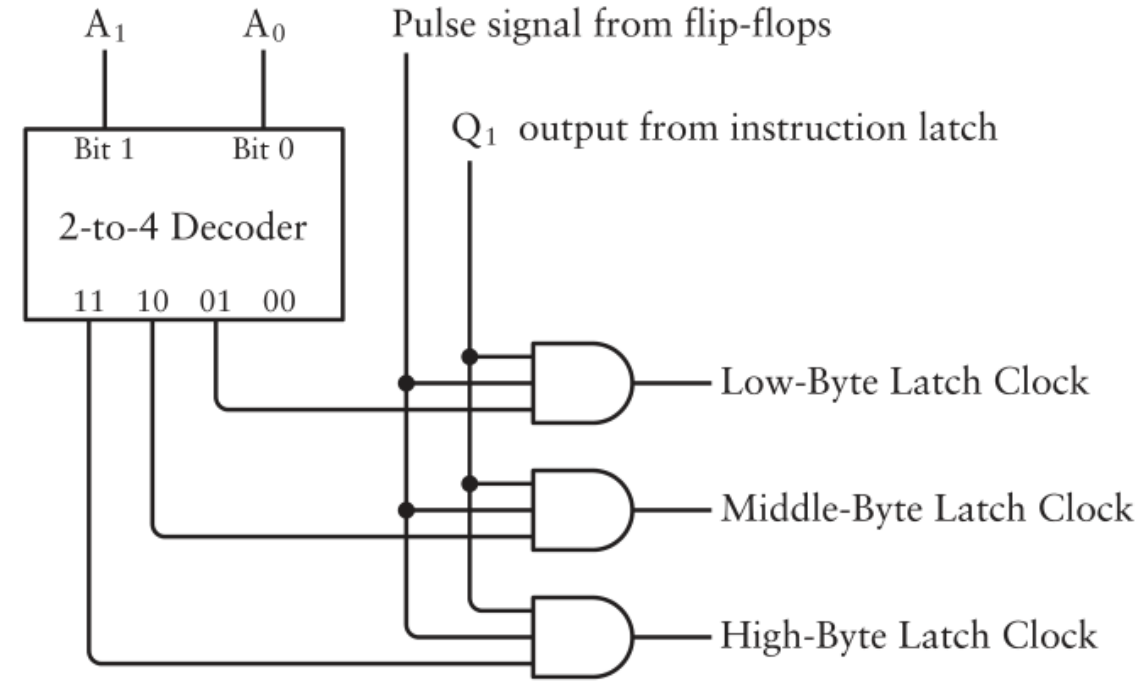
• 08h - остановить машину.

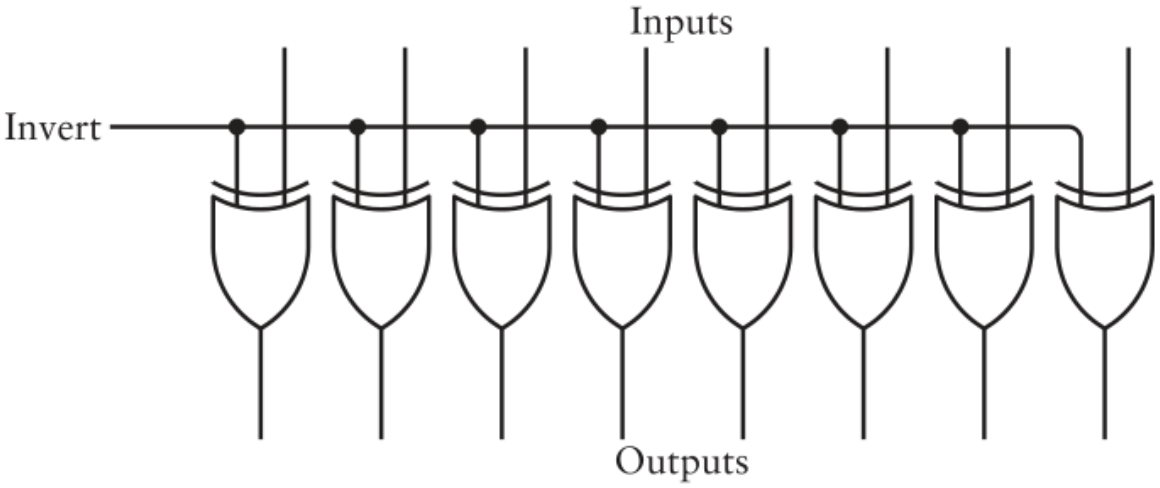
В защёлке инструкции нужны лишь четыре бита. Код остаётся там, пока читаются следующие три байта, а отдельные его биты управляют частями машины.

Код	**Q_3**	**Q_2**	**Q_1**	**Q_0**	**Операция**
02h	0	0	1	0	Add
03h	0	0	1	1	Subtract
04h	0	1	0	0	Write
08h	1	0	0	0	Halt

Например, Q_1 равен 1 для Add и Subtract. Поэтому он участвует в формировании Clock трёх защёлок данных вместе с декодером 2 в 4 и Pulse.

Когда A_1A_0 = 00, фиксируется инструкция. Затем при 01, 10 и 11 последовательно читаются три байта данных. Если инструкция Add или Subtract, Q_1 равен 1 и три вентиля И по очереди создают Clock защёлок:





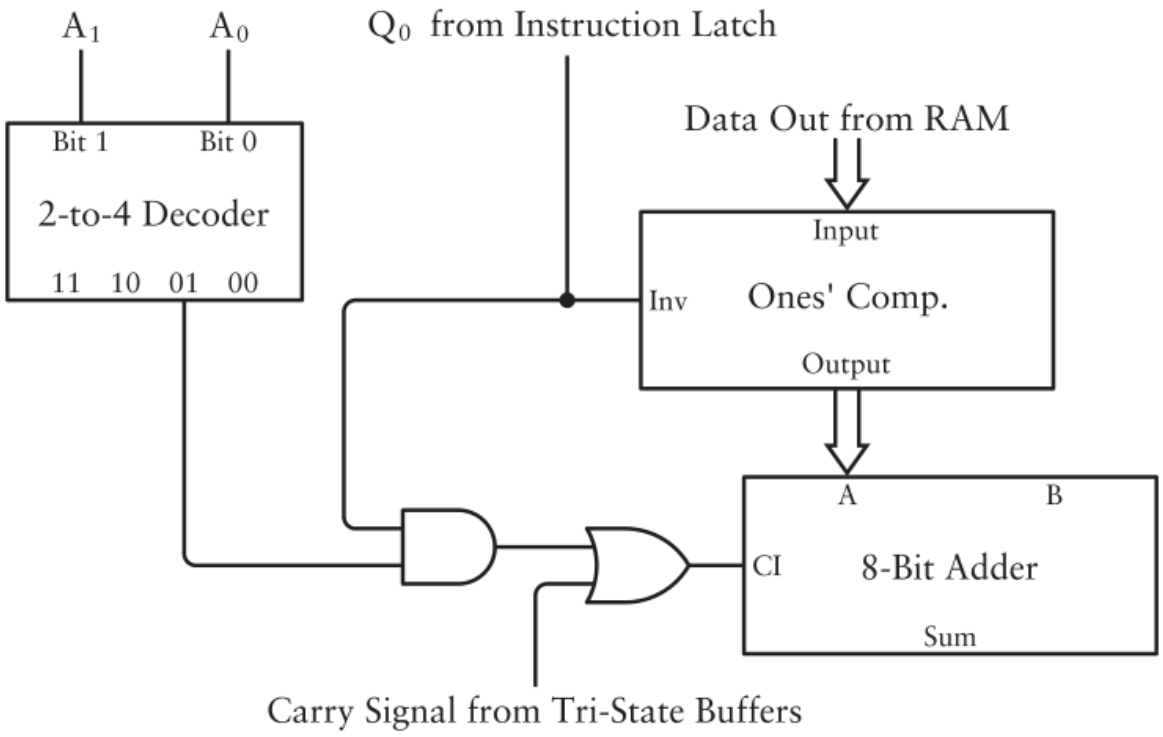
Выбор кодов 02h и 03h вместо, скажем, 01h и 02h был намеренным: Add и Subtract имеют общий бит, напрямую управляющий Clock защёлок.

Таким образом, один и тот же Q_1 остаётся активным на протяжении чтения всех трёх байтов, а конкретную защёлку выбирают A_1A_0. С Pulse эти условия образуют короткий синхронный такт: сначала для Low при 01, затем для Mid при 10 и для High при 11.

Q_0 равен 1 только для вычитания. Тогда число из следующих трёх байтов нужно дополнить до двух: сначала инвертировать все биты, то есть получить дополнение до единицы, затем прибавить 1. Схема инвертирования нам знакома из главы 16.

Входы блока поступают от Data Out RAM, выходы - на A восьмибитного сумматора. Invert можно подключить прямо к Q_0 инструкции.

Дополнение до двух равно дополнению до единицы плюс 1. Единица подаётся через Carry In сумматора, но только для первого из трёх байтов:



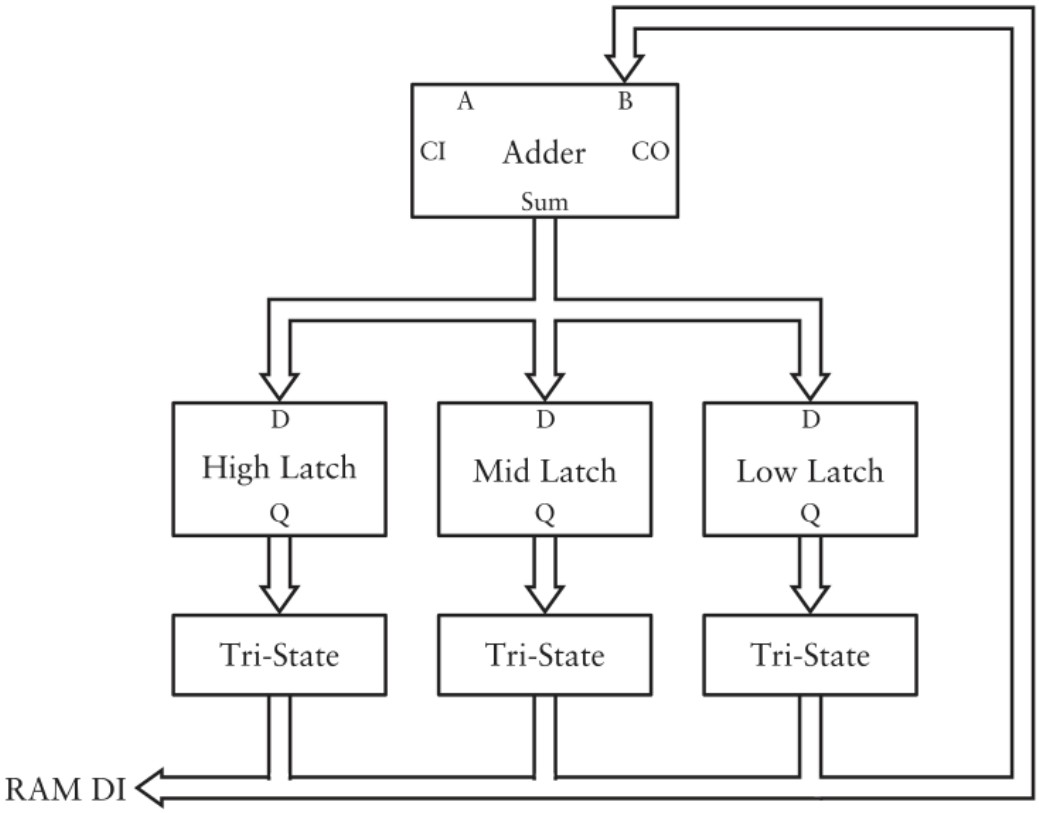
При $Q_0 = 1$ выполняется вычитание: все байты RAM инвертируются блоком Ones' Comp., а Carry In устанавливается в 1 только для первого байта. Выход 01 декодера активен лишь для него.

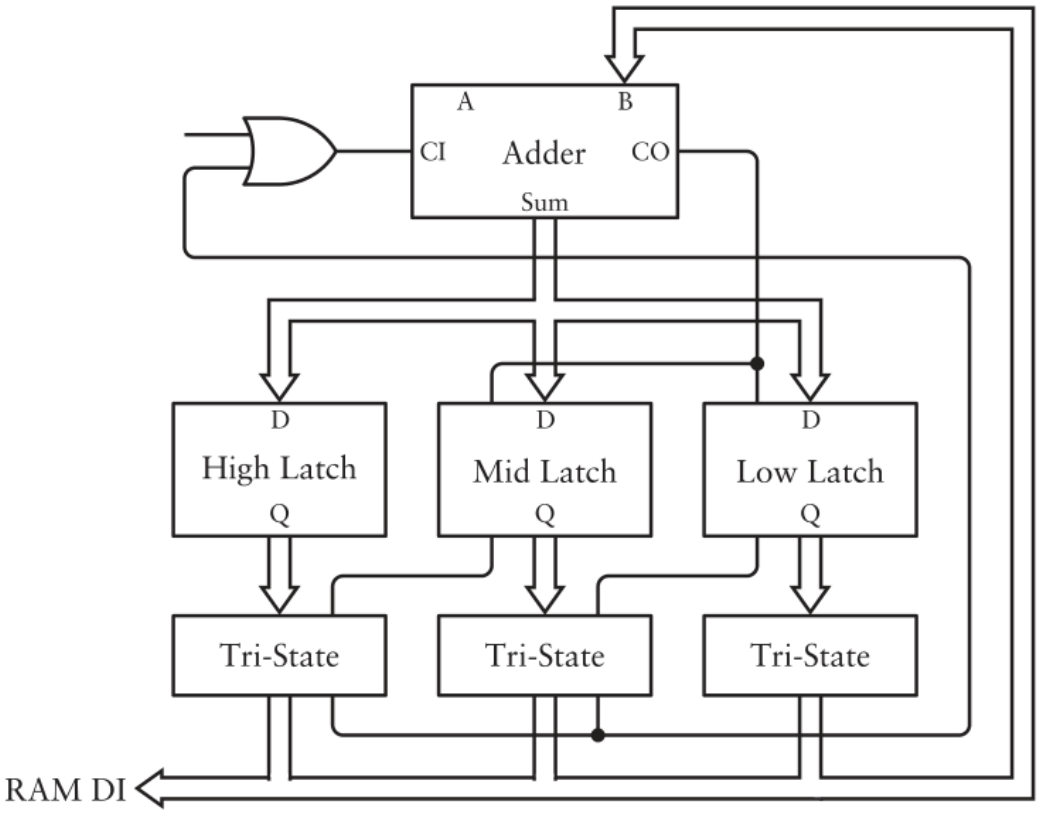
Когда инструкция является сложением, Q_0 равен 0, поэтому блок дополнения пропускает биты без изменения, а добавочная единица не создаётся. При вычитании Q_0 остаётся равным 1 для всех трёх байтов, инвертируя каждый из них, но единица дополнительного кода вводится только в самое начало многобайтового числа.

ИЛИ нужен потому, что Carry In может потребоваться и для второго или третьего байта из-за переноса предыдущего сложения. До сих пор проблема переносов полностью игнорировалась; теперь придётся встретиться с ней лицом к лицу.

Трёхбайтовый аккумулятор содержит три защёлки и три трёхсостоятельных буфера для текущего итога.

Но не совсем такие, как показано ранее. Чтобы вместить перенос, две защёлки должны хранить девять бит: восемь битов суммы и Carry Out сумматора. Два буфера также становятся девятибитными, чтобы перенос от младших байтов использовался при сложении средних, а перенос от средних - при сложении старших:



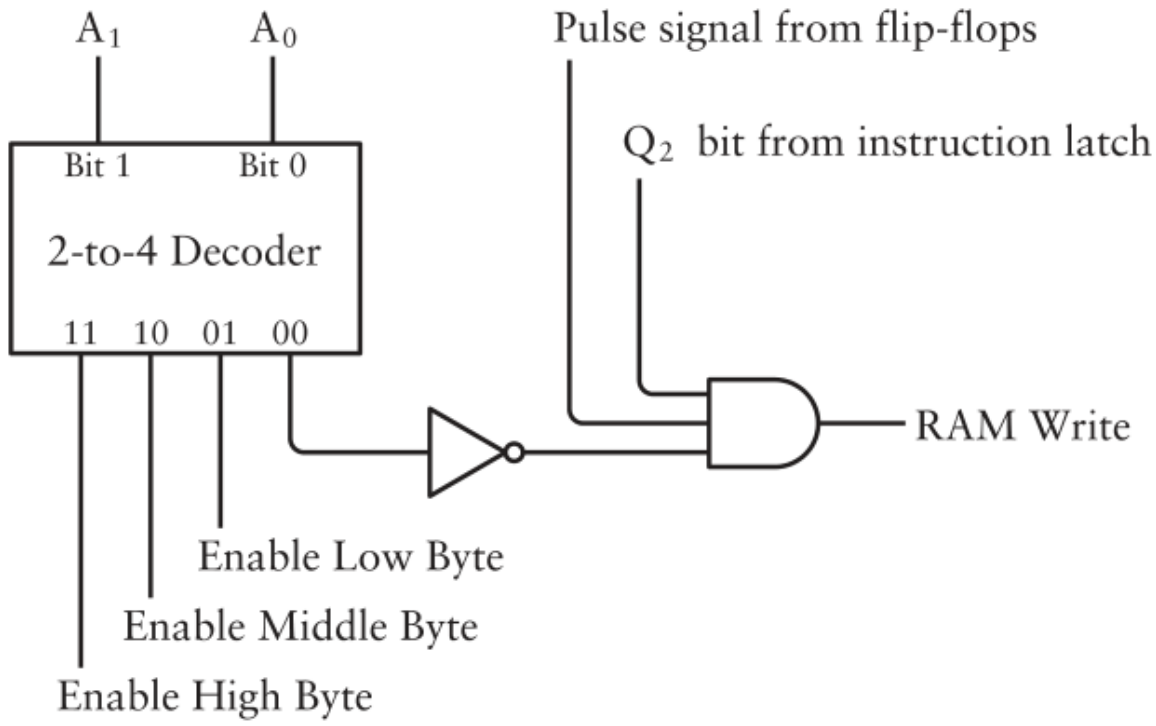


Carry Out сумматора сохраняется в защёлках Low и Mid, но затем соответствующие биты идут в буферы Mid и High. Это выглядит странно, но объясняется просто.

При сложении младших байтов Carry In равен 0 для сложения и 1 для вычитания. Результат может создать Carry Out, который сохраняется в Low, однако использовать его нужно при сложении *средних* байтов, поэтому он поступает через буфер Mid. Аналогично перенос среднего сложения используется для старшего.

Это объясняет кажущееся перекрёстным соединение: девятый бит Low физически относится не к следующему чтению Low, а к немедленно следующей операции над Mid. Девятый бит Mid, в свою очередь, становится Carry In операции High. Для старшей защёлки сохранять ещё один перенос уже не требуется, поскольку результат ограничен тремя байтами.

Сложение и вычитание теперь полностью обработаны. Осталась инструкция 04h, записывающая три байта в память. Она определяется Q_2:

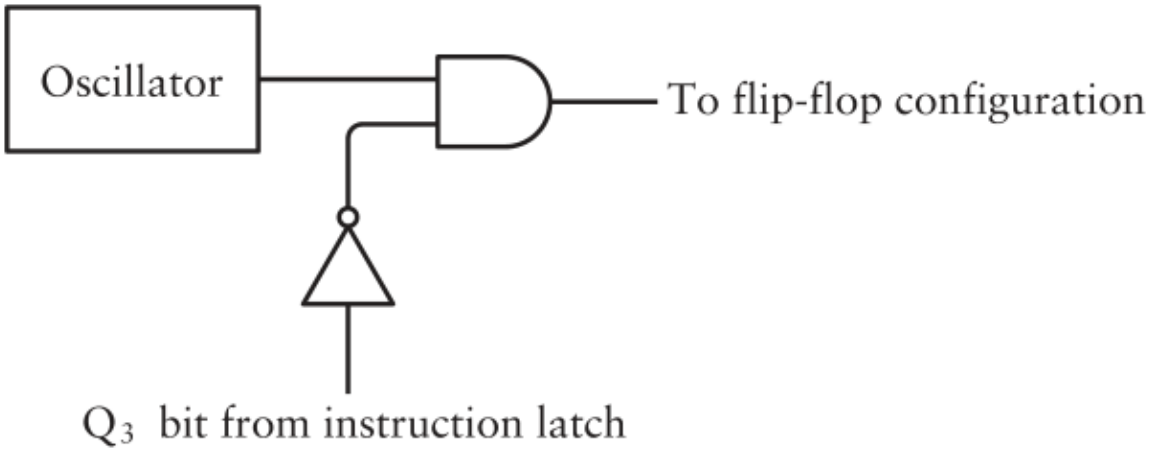


RAM Write создаётся только при A_1A_0 = 01, 10 или 11, то есть для трёх байтов данных; поэтому используется инвертор. Последовательно разрешаются три буфера. Если Q_2 = 1 и Pulse = 1, три байта по очереди записываются в память.

При A_1A_0 = 00 находится сама инструкция, и запись результата туда запрещена. После неё адреса 01, 10 и 11 выбирают Low, Mid и High. Те же буферы, которые раньше подавали текущий итог на сумматор, теперь по очереди выводят его на Data In RAM. Pulse гарантирует, что Write возникает в устойчивую фазу.

Последняя инструкция 08h означает Halt. Когда Q_3 равен 1, нужно остановить генератор, управляющий всей машиной.

Удобно добавить Clear, очищающий триггеры, счётчик и все защёлки перед новым вычислением:



Интерактивная версия полного трёхбайтового аккумулятора доступна на CodeHiddenLanguage.com.

Теперь очевидно, почему четыре кода выбраны именно так. Один бит обозначает сложение или вычитание, второй - вычитание, третий - запись результата, четвёртый - остановку. При кодах 01h, 02h, 03h и 04h потребовалась бы дополнительная схема декодирования.

Понятно и решение хранить числа в трёх, а не четырёх байтах. Благодаря этому A_0 и A_1 напрямую управляют защёлками. При четырёх байтах коды располагались бы по адресам 0000h, 0005h, 000Ah, 000Fh, 0014h и так далее, усложняя правильную маршрутизацию.

Теперь можно определить два слова из названия книги: *аппаратное обеспечение* и *программное обеспечение*. Трёхбайтовый аккумулятор ясно показывает различие. Аппаратное обеспечение - вся схема, программное - коды и данные в памяти. Оно «мягкое», потому что легко изменяется: ошибочное число или перепутанный код можно быстро исправить. Схему менять намного труднее, даже если она лишь моделируется программно, а не собрана из проводов и транзисторов.

Но машина показывает и тесную связь двух сторон. Коды и числа хранятся в триггерах памяти, а составляющие их биты превращаются в сигналы, взаимодействующие с остальным оборудованием. На самом основном уровне и hardware, и software - электрические сигналы, взаимодействующие с логическими вентилями.

Если использовать такую машину для финансов малого предприятия, успех бизнеса может заставить нервничать.

Доход или расход легко превысит трёхбайтовую ёмкость, а расширить машину трудно: предел встроен в аппаратную схему.

Трёхбайтовый аккумулятор придётся признать тупиком. Но знания не пропали: мы обнаружили нечто чрезвычайно важное.

Можно построить машину, реагирующую на коды в памяти. У нашей было четыре кода. Если инструкция занимает байт, можно определить до 256 разных операций. Каждая может быть простой, но их сочетания способны выполнять сложные задачи.

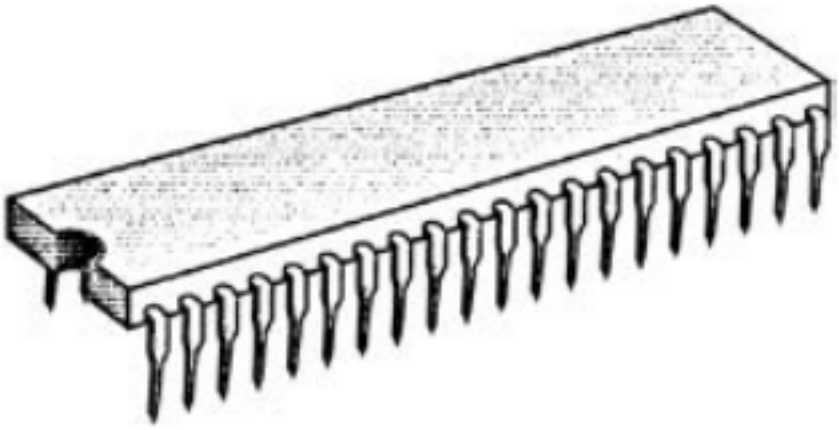
Ключ в том, что простые задачи реализуются аппаратно, а сложные - программно, последовательностями кодов инструкций.

В 1970 году постройка универсальной машины была бы огромной работой. К 1980 году её уже можно было купить в виде микросхемы - *микропроцессора*, обращающегося к 64 КБ памяти и понимающего почти 256 инструкций.

Первый «компьютер на кристалле» появился в ноябре 1971 года: Intel 4004, четырёхбитный процессор с 2250 транзисторами и адресацией 4 КБ. К середине 1972 года вышел первый восьмибитный Intel 8008 с адресацией 16 КБ.

Их универсальности и памяти ещё не хватало для персональных компьютеров с клавиатурой и дисплеем. В основном они предназначались для *встроенных систем*, работая вместе с другой цифровой логикой, управляя механизмами или выполняя специальные задачи.

В апреле 1974 года появился Intel 8080 - восьмибитный процессор примерно с 4500 транзисторами и адресацией 64 КБ. Он помещался в 40-выводной корпус:



8080 был готов к большим делам. Именно он работал в Altair 8800 с обложки *Popular Electronics* в конце главы 19 и стал предком шестнадцатибитных Intel в первом IBM PC, выпущенном в августе 1981 года.

Тем временем микропроцессоры создавала Motorola. Выпущенный в 1974 году Motorola 6800 также был восьмибитным, адресовал 64 КБ и помещался в 40-выводной корпус. Упрощённую версию 6800 компания MOS Technology выпустила в 1975 году под названием MOS 6502. Стив Возняк использовал её во влиятельном компьютере Apple II, вышедшем в июне 1977 года.

Intel 8080 и Motorola 6800 были во многом похожи, однако реализовали совершенно разные коды инструкций. Отличался и порядок байтов. Motorola 6800 был big-endian и хранил многобайтовые значения старшим байтом вперёд. Intel 8080 был little-endian и начинал с младшего.

В следующей главе мы попытаемся заглянуть внутрь Intel 8080, построив его подобие. Будут применяться только уже знакомые основные компоненты: вентили, триггеры, сумматоры, защёлки и трёхсостоятельные буферы.

Этот амбициозный проект не будет доведён до полного аналога. Наша версия окажется слабее настоящей. Но мы продвинемся достаточно далеко, чтобы получить исключительно глубокое понимание происходящего внутри компьютера.

Эта страница намеренно оставлена пустой.

Глава 21. Арифметико-логическое устройство

Современный компьютер представляет собой сложное собрание множества компонентов, но приблизительно их можно разделить на три категории:

• память;

• центральный процессор, CPU;

• устройства ввода и вывода, I/O, часто называемые периферийными.

В главе 19 мы узнали, как устроена память с произвольным доступом и как к каждому байту обращаются по адресу. Глава 20 показала, что память хранит числа, а находящиеся там коды управляют схемой, обрабатывающей эти числа. В общем случае содержимым памяти также бывают текст, изображения, музыка, фильмы и всё, что представимо цифровым способом, нулями и единицами. Коды инструкций вместе часто называют *кодом*, всё остальное - *данными*. Иными словами, память содержит код и данные.

Компьютер включает несколько устройств I/O. Их набор сильно зависит от того, стоит ли компьютер на столе, складывается ли и носится под мышкой, помещается ли в кармане или сумочке, либо скрыт в микроволновке, роботе-пылесосе или автомобиле.

Самые заметные устройства настольного компьютера - экран, клавиатура, мышь и, возможно, принтер. У ноутбука вместо мыши бывает сенсорная панель, а телефон объединяет все эти функции на одном экране. У каждого есть и устройство массового хранения.

На настольном компьютере это может быть жёсткий диск, в ноутбуке SSD, в телефоне флеш-память, иногда дополненная внешними накопителями.

Другие устройства менее очевидны: схемы воспроизведения звука и музыки; подключения к интернету через Ethernet или Wi-Fi; приёма GPS, сообщающего положение и направление; датчики силы тяжести и движения, определяющие ориентацию телефона относительно Земли.

Однако предмет этой и трёх следующих глав - CPU, который называют «сердцем», «душой» или «мозгом» компьютера в зависимости от предпочитаемой метафоры.

Трёхбайтовый аккумулятор главы 20 состоял из счётчика для доступа к памяти, сумматора и защёлок. Схема управлялась кодами из памяти: складывала и вычитала числа, а затем записывала текущий итог.

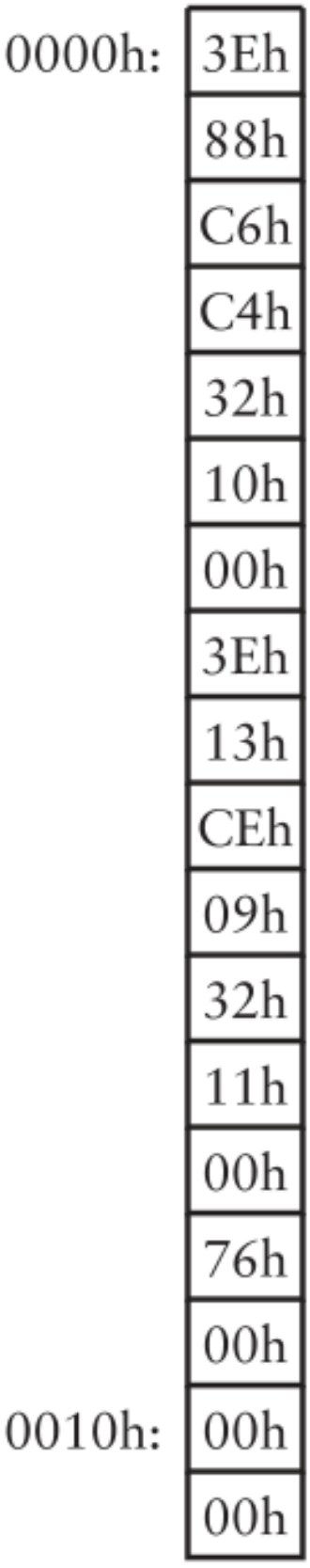
CPU очень похож на него, но обобщён для множества кодов и потому намного универсальнее.

Процессор, который мы начнём строить, работает с байтами и потому является восьмибитным CPU. Но он адресует 64 КБ RAM, для чего нужен шестнадцатибитный, двухбайтовый адрес. Хотя основная единица работы - байт, в ограниченной степени CPU должен обрабатывать шестнадцатибитные адреса.

Пусть этот CPU не существует материально, теоретически он сможет читать код и данные и выполнять множество арифметических и логических задач. По типам обработки он эквивалентен любому цифровому компьютеру, каким бы сложным тот ни был.

Переход от 8 к 16, 32 и 64 битам не дал процессорам принципиально новых ***видов*** задач. Они выполняют те же задачи быстрее. Иногда скорость решает всё: например, при декодировании видеопотока. Восьмибитный CPU способен сделать те же вычисления, но, вероятно, слишком медленно для нормального воспроизведения.

Хотя восьмибитный CPU выполняет математические и логические операции над байтами, он может работать и с многобайтовыми числами. Допустим, нужно сложить шестнадцатибитные 1388h и 09C4h, то есть 5000 и 2500. В память для обработки записываются такие байты:



Эта смесь кодов и данных пока вряд ли понятна, поскольку коды инструкций ещё не известны. На аннотированном рисунке показано их назначение.

Последовательность инструкций называется *компьютерной программой*. Эта простая программа складывает 1388h и 09C4h. Сначала CPU складывает младшие байты 88h и C4h и записывает результат по адресу 0010h. Затем складывает старшие 13h и 09h вместе с возможным переносом первого сложения, сохраняя сумму по 0011h. После этого CPU останавливается. Шестнадцатибитный результат находится в 0010h и 0011h.

Откуда взялись коды 3Eh, C6h, 32h, CEh и 76h? Пока процессор не построен, их можно было придумать. Но это настоящие коды Intel 8080, использованного в MITS Altair 8800, который широко считают первым коммерчески успешным персональным компьютером. Первый IBM PC применял не 8080, а Intel 8088 - следующее поколение того же семейства.

В этой и следующих главах Intel 8080 служит моделью, и только моделью. Наш CPU реализует лишь подмножество его возможностей. У 8080 было 244 кода инструкций, наша машина получит чуть больше половины.

Тем не менее станет ясно, что происходит в самом сердце, душе или мозге компьютера.

Коды инструкций, коды операций и опкоды также называют *машинными кодами*, потому что их непосредственно использует машина - схема CPU. Показанный пример является программой в машинном коде.

Все коды 8080 однобайтовые, но некоторым нужны один или два дополнительных байта. Коды 3Eh, C6h и CEh всегда сопровождаются ещё одним байтом, поэтому вся инструкция двухбайтовая. После 32h идут два байта адреса памяти, и это трёхбайтовая инструкция. Другим, например 76h для остановки, дополнительные байты не нужны. Разная длина заметно усложнит CPU.

Дополнительный байт после 3Eh, C6h или CEh не является самостоятельной инструкцией: он составляет часть той же команды и задаёт непосредственно используемое значение. Аналогично два байта после 32h вместе образуют шестнадцатибитный адрес. CPU должен знать длину текущего опкода, чтобы правильно определить адрес следующей инструкции, а не принять данные или часть адреса за новый код.

Предыдущий пример - не лучший способ складывать шестнадцатибитные числа: код и данные перемешаны. Часто их лучше держать в разных областях памяти; это станет понятнее в следующей главе.

CPU состоит из нескольких частей. Остаток главы посвящён самой фундаментальной - *арифметико-логическому устройству*, ALU. Оно складывает, вычитает и выполняет несколько иных полезных операций.

В восьмибитном CPU ALU выполняет лишь восьмибитное сложение и вычитание. Но числа часто имеют 16, 24, 32 и больше битов. Их приходится обрабатывать байтами от младшего к старшему, и каждая следующая операция должна учитывать перенос предыдущей.

Само ALU не обязано становиться шире всякий раз, когда программа встречает большое число. Одно и то же восьмибитное оборудование многократно обрабатывает отдельные байты. Универсальность возникает благодаря последовательности операций и сохранению состояния между ними.

Поэтому ALU обязано уметь:

• сложить два байта;

• сложить два байта с возможным предыдущим переносом;

• вычесть один байт из другого;

• вычесть с возможным предыдущим переносом, то есть заёмом.

Сократим названия: Add, Add with Carry, Subtract и Subtract with Borrow. Две операции с переносом используют сохранённый бит предыдущего сложения или вычитания. Он равен 0 или 1 в зависимости от результата, поэтому ALU должно сохранять Carry для следующей операции. Как обычно, перенос значительно усложняет арифметику.

Для первой пары младших байтов прежний Carry не используется, поскольку до неё ещё не было части того же многобайтового вычисления. Для каждой следующей пары он уже обязателен: иначе единица, вышедшая за старший разряд предыдущего байта, потеряется.

Для двух 32-битных, четырёхбайтовых чисел сначала складываются младшие байты. Полученный перенос назовём *флагом Carry*, поскольку он сигнализирует о переносе. Затем следующая пара складывается с этим флагом. Всего нужны:

• Add;

• Add with Carry;

• Add with Carry;

• Add with Carry.

При вычитании вычитаемое превращается в дополнительный код: биты инвертируются, а для первого байта через вход переноса прибавляется 1. Поэтому нужны:

• Subtract;

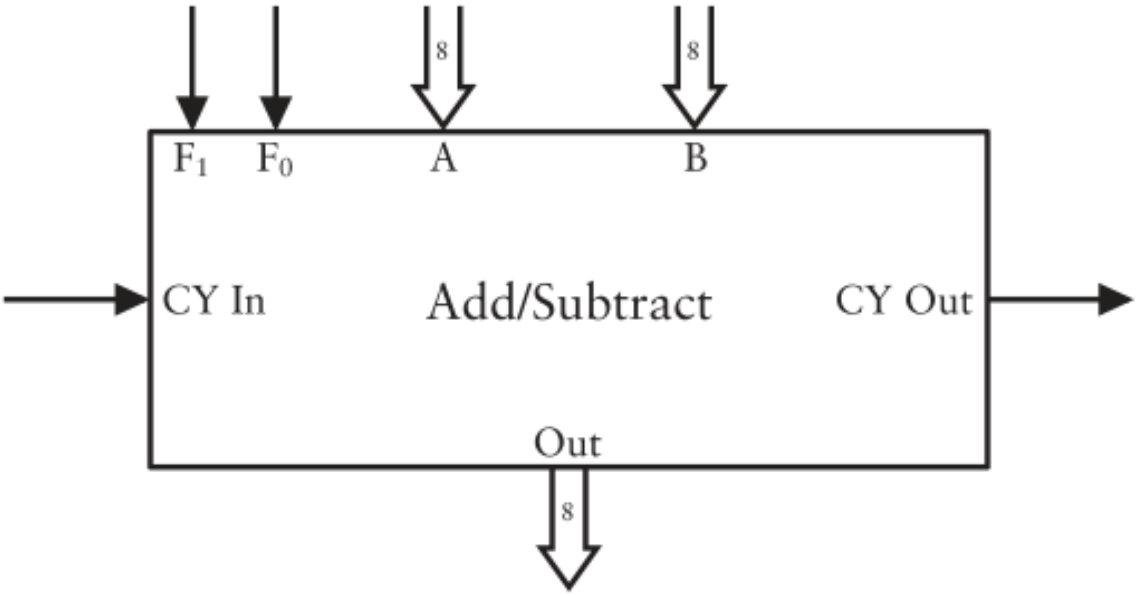
• Subtract with Borrow;

• Subtract with Borrow;

• Subtract with Borrow.

Название Borrow, «заём», использует привычную терминологию вычитания. Электронная схема по-прежнему работает с однобитным состоянием переноса. Первая операция создаёт правильное дополнение до двух, а следующие передают это состояние между всё более старшими байтами.

Заключим схему сложения и вычитания в прямоугольник:



F_1	F_0	Operation
0	0	Add
0	1	Add with Carry
1	0	Subtract
1	1	Subtract with Borrow

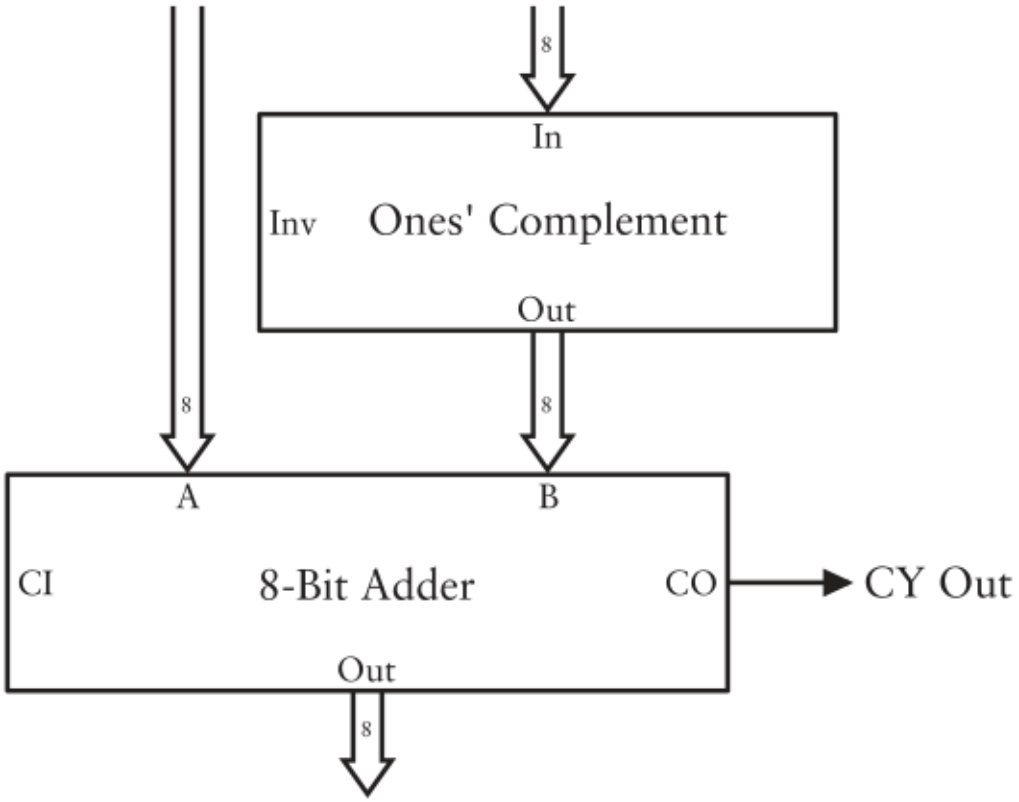
Два восьмибитных входа дают восьмибитный результат, но есть отличия от прежних сумматоров. Обычно применялись CI и CO. Здесь CY означает флаг Carry. CY Out совпадает с Carry Out сумматора, но CY In - сохранённый флаг предыдущей операции, и он не обязательно совпадает с непосредственным Carry In сумматора.

Новые входы F_0 и F_1 обозначают функцию:

F_1	F_0	Операция
0	0	Add
0	1	Add with Carry
1	0	Subtract
1	1	Subtract with Borrow

Устройство будет работать с кодами в памяти. Если спроектировать их разумно, два бита кода можно напрямую использовать как F_0 и F_1, подобно битам кодов Add и Subtract в предыдущей главе.

Большая часть модуля уже знакома:



F_1	F_0	Function	Inv	CI
0	0	Add	0	0
0	1	Add with Carry	0	CY
1	0	Subtract	1	1
1	1	Subtract with Borrow	1	CY

Ones' Complement инвертирует вход при Inv = 1 - это первый шаг получения дополнительного кода для вычитания. Carry Out сумматора становится CY Out.

Остаётся сформировать Inv и CI:

F_1	F_0	Функция	Inv	CI
0	0	Add	0	0
0	1	Add with Carry	0	CY
1	0	Subtract	1	1
1	1	Subtract with Borrow	1	CY

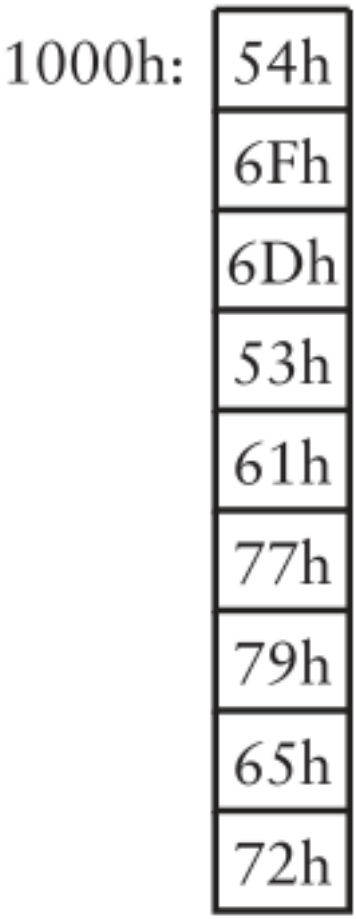
Inv совпадает с F_1. CI сложнее. Для Subtract он равен 1, чтобы при первом байте прибавить единицу к дополнению до единицы. Если F_0 = 1, CI берётся из сохранённого CY предыдущей операции. Всё это реализуется показанной логикой.

Интерактивная версия полного Add/Subtract доступна на CodeHiddenLanguage.com.

Что ещё должно делать ALU кроме сложения и вычитания? Если вы ответили «умножать и делить», придётся разочароваться. После всей сложности сложения представьте логическую сложность умножения и деления. Такие схемы возможны, но выходят за скромные цели книги. Intel 8080 тоже не реализовывал эти операции, поэтому не будет их и у нашего CPU. Однако к концу главы 24 у него появятся основные средства для программного умножения.

Вместо этого подумаем о втором слове в названии ALU - *логическое*. В книге оно обычно означает булевы операции. Чем они полезны?

Предположим, в памяти с некоторого адреса лежат ASCII-коды текста, который нужно перевести в нижний регистр:



Коды заглавных букв идут от 41h до 5Ah, строчных - от 61h до 7Ah.

Соответствующие прописные и строчные буквы отличаются на 20h. Если буква заведомо заглавная, к её коду можно прибавить 20h. Для T код 54h становится 74h, кодом t:

```
  01010100
+ 00100000
----------
  01110100
```

Но ко всем буквам прибавлять нельзя. 6Fh, строчная o, превратится в 8Fh, который не является ASCII-кодом:

```
  01101111
+ 00100000
----------
  10001111
```

Сравним A = 41h и a = 61h:

```
A: 01000001
a: 01100001
```

А также Z = 5Ah и z = 7Ah:

```
Z: 01011010
z: 01111010
```

Для всех букв отличается только третий бит слева. Чтобы получить нижний регистр, его нужно установить в 1. Если буква уже строчная, бит уже равен 1 и ничего не меняется.

Значит, разумнее не складывать, а применить побитовое ИЛИ с 20h. Таблица ИЛИ:

ИЛИ	**0**	**1**
0	0	1
1	1	1

Результат равен 1, если хотя бы один операнд равен 1.

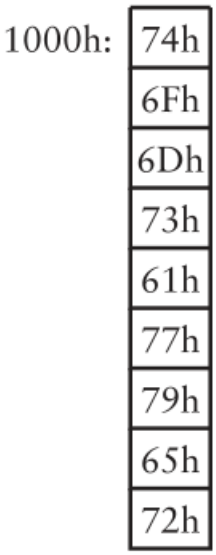
Для заглавной T:

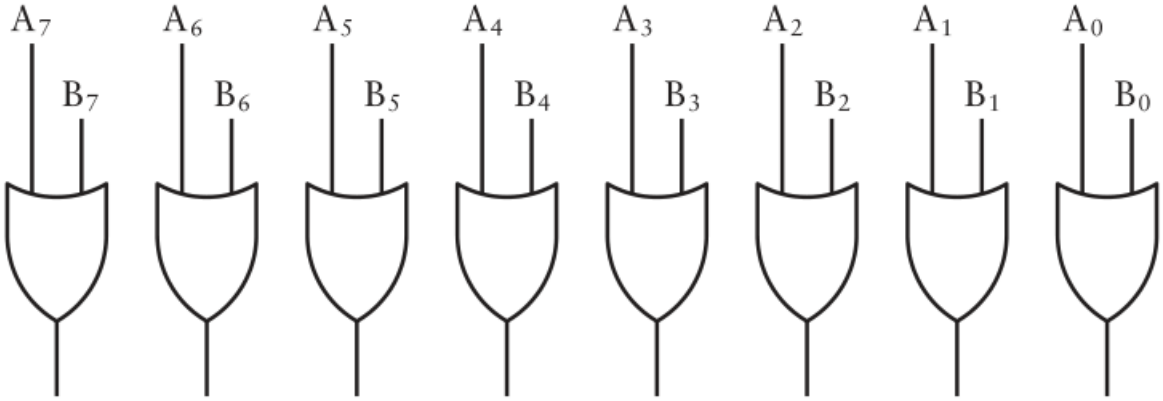
```
   01010100
OR 00100000
-----------
   01110100
```

Преимущество: строчные буквы не меняются. Для o:

```
   01101111
OR 00100000
-----------
   01101111
```

Применив ИЛИ с 20h к каждому коду в памяти, можно перевести весь текст в нижний регистр:





Операция называется *побитовым ИЛИ*, потому что ИЛИ выполняется между каждой парой соответствующих битов. Она полезна не только для регистра текста, поэтому добавим её в ALU.

Все восемь пар обрабатываются одновременно. Каждый выходной бит зависит только от двух входных битов того же разряда; переносов между разрядами, как при сложении, нет. Маска 00100000 изменяет только нужную позицию, а нули во всех остальных оставляют исходные биты без изменений.

Для восьми соответствующих пар битов двух байтов A и B схема выполняет ИЛИ, после чего её можно заключить в блок.

Теперь преобразуем текст в верхний регистр. Здесь бит надо сбросить в 0, поэтому требуется И. Таблица:

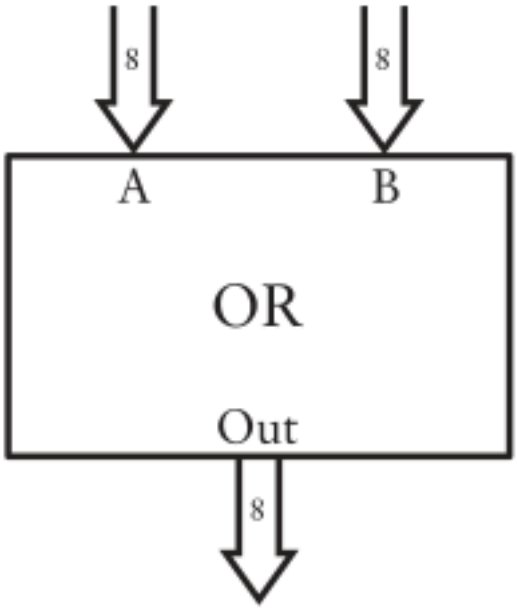
И	**0**	**1**
0	0	0
1	0	1

Вместо ИЛИ с 20h применяется И с DFh, то есть 11011111, инверсией 20h. Строчная o преобразуется так:

```
    01101111
AND 11011111
------------
    01001111
```

6Fh становится 4Fh, кодом заглавной O. Если буква уже заглавная, И с DFh на неё не влияет:

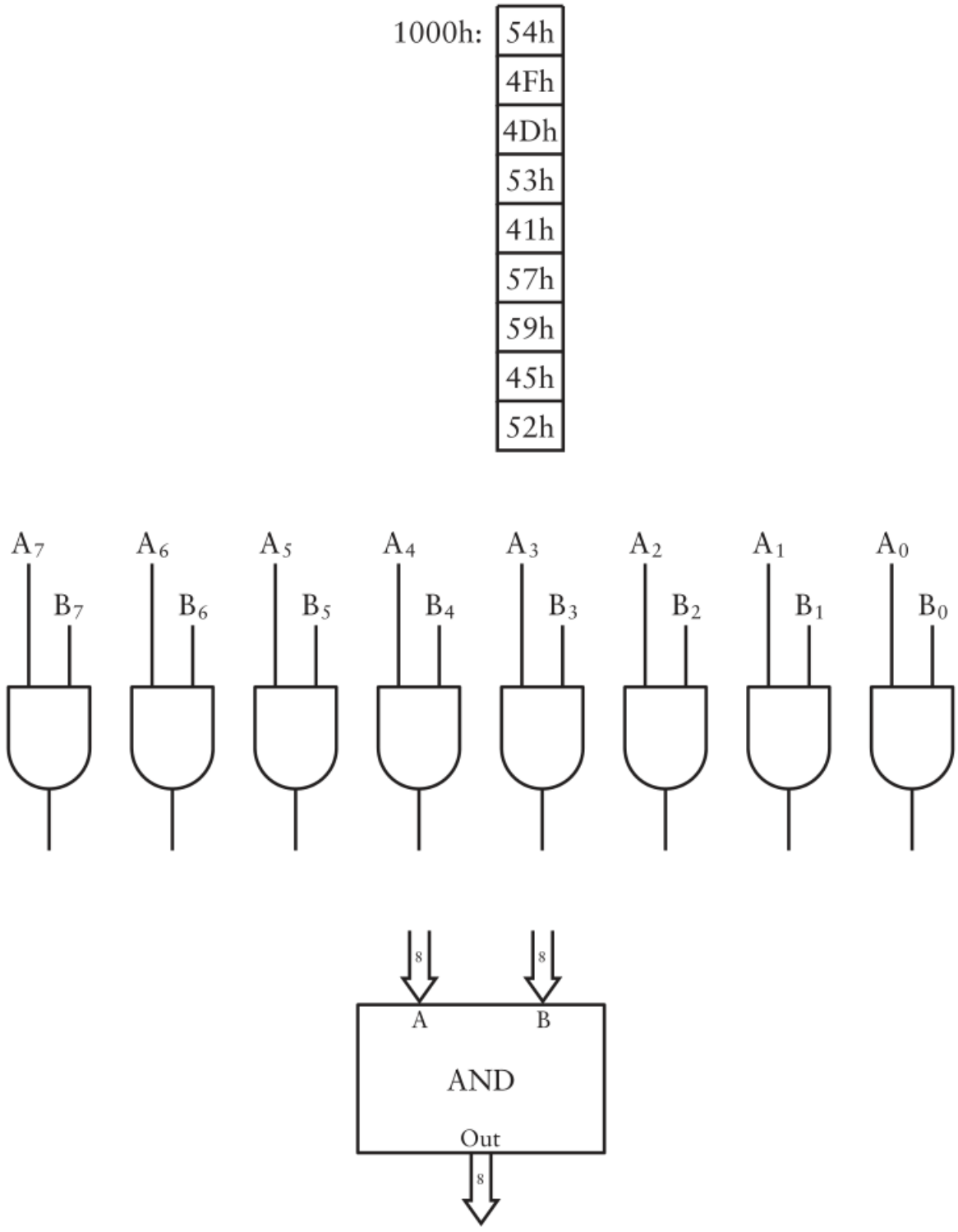
```
    01010100
AND 11011111
------------
    01010100
```



AND	0	1
0	0	0
1	0	1

На всём тексте И с DFh переводит каждую букву в верхний регистр.

В ALU пригодятся восемь вентилей И для побитовой операции над двумя байтами:



Побитовое И также позволяет проверять отдельные биты. Например, чтобы узнать регистр ASCII-буквы, выполните И с 20h. Результат 20h означает строчную букву, 00h - заглавную.

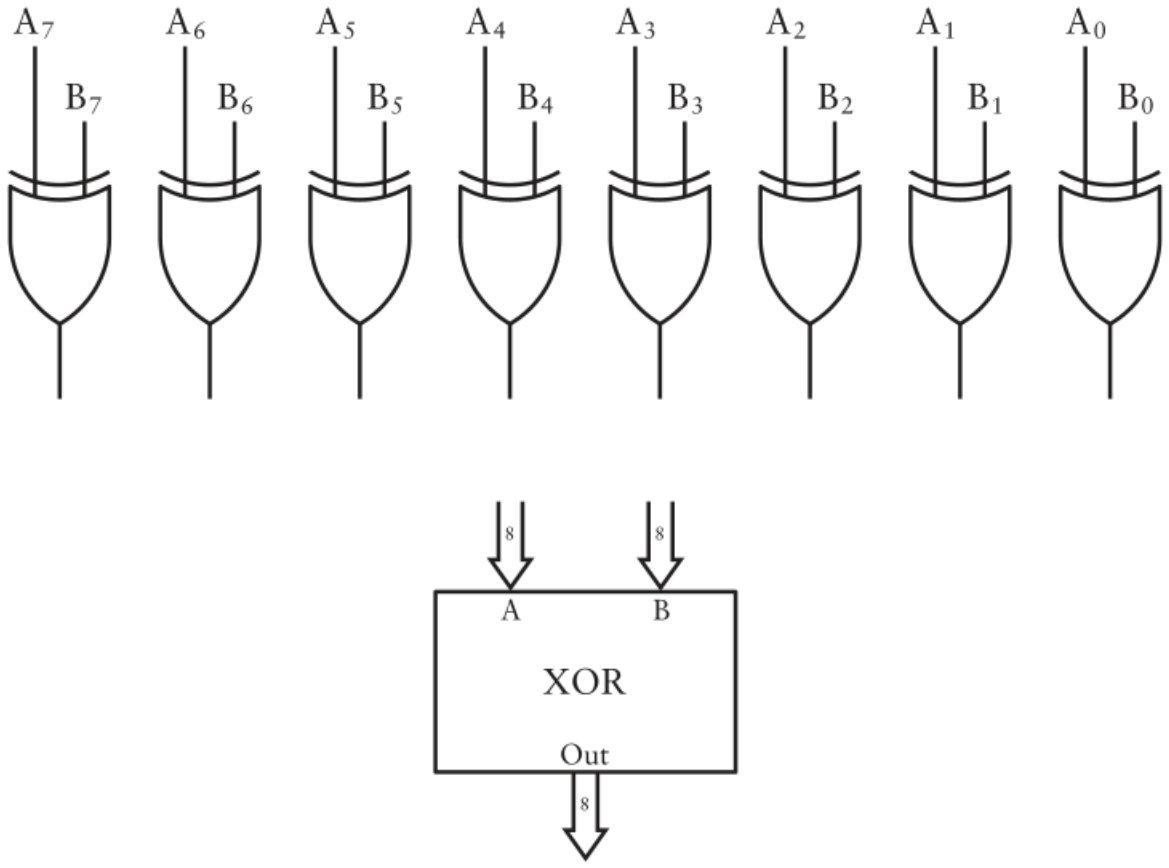
Величина 20h здесь служит *маской*. Единица маски сохраняет проверяемый бит, а нули принудительно обнуляют остальные. После AND остаётся только интересующий третий слева бит, и результат легко сравнить с 00h или 20h.

Ещё одна полезная операция - исключающее ИЛИ, XOR:

XOR	0	1
0	0	1
1	1	0

Ряд из восьми XOR выполняет операцию между двумя байтами:

XOR	0	1
0	0	1
1	1	0



XOR полезен для инвертирования. Если применить XOR с 20h к ASCII-кодам строки `TomSawyer`, все заглавные буквы станут строчными, а строчные - заглавными. XOR с FFh инвертирует все биты значения.

Для Add/Subtract были определены F_1 и F_0. Всему ALU нужны три функциональных бита.

F_2	F_1	F_0	**Операция**
0	0	0	Add
0	0	1	Add with Carry
0	1	0	Subtract
0	1	1	Subtract with Borrow
1	0	0	Bitwise AND
1	0	1	Bitwise XOR
1	1	0	Bitwise OR
1	1	1	Compare

Коды выбраны не произвольно: их подразумевают настоящие инструкции Intel 8080. Кроме И, XOR и ИЛИ добавлена операция Compare, которую мы скоро рассмотрим.

В начале главы программа использовала C6h и CEh, выполняющие операцию с байтом, следующим в памяти. C6h - обычное сложение, CEh - сложение с переносом. Такие инструкции называют *непосредственными*. В 8080 они входят в семейство:

Инструкция	**Опкод**
Add Immediate	C6h
Add with Carry Immediate	CEh
Subtract Immediate	D6h
Subtract with Borrow Immediate	DEh
AND Immediate	E6h
XOR Immediate	EEh
OR Immediate	F6h
Compare Immediate	FEh

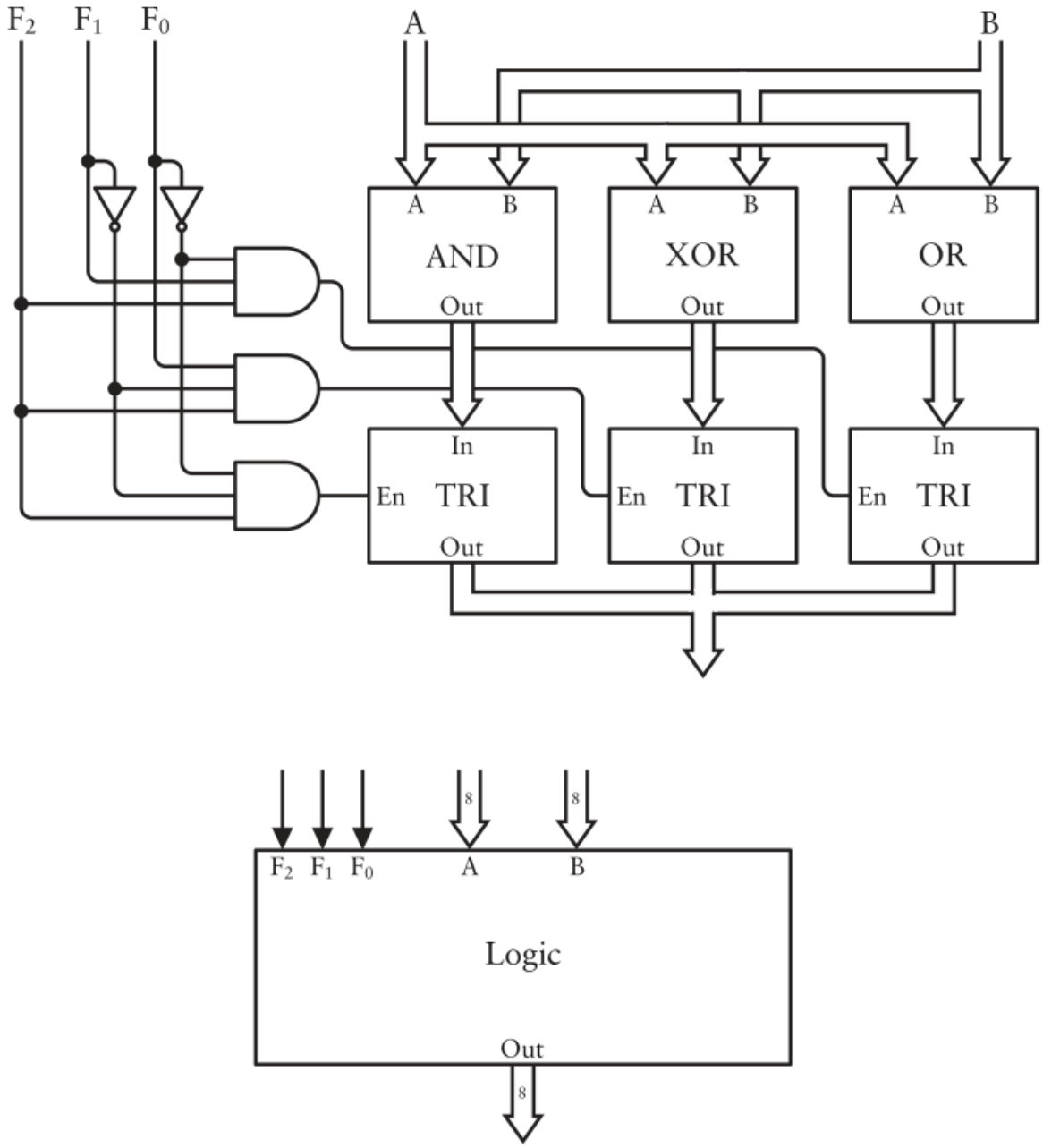
Общий двоичный вид:

```
11 F2 F1 F0 110
```

F_2	F_1	F_0	Operation
0	0	0	Add
0	0	1	Add with Carry
0	1	0	Subtract
0	1	1	Subtract with Borrow
1	0	0	Bitwise AND
1	0	1	Bitwise XOR
1	1	0	Bitwise OR
1	1	1	Compare

Instruction	Opcode
Add Immediate	C6h
Add with Carry Immediate	CEh
Subtract Immediate	D6h
Subtract with Borrow Immediate	DEh
AND Immediate	E6h
XOR Immediate	EEh
OR Immediate	F6h
Compare Immediate	FEh

F_2, F_1 и F_0 используются схемой, объединяющей блоки AND, XOR и OR:



Входы A и B одновременно поступают во все три блока, и каждый выполняет свою операцию. Но выбрать нужно только один выход. Для этого служат три восьмибитных трёхсостоятельных буфера TRI. Они выбирают один результат или ни одного по F_0-F_2. Если $F_2 = 0$ либо все три F равны 1, ни один логический выход не выбран.

Заключив схему в блок, получаем логический компонент ALU.

При $F_2F_1F_0 = 111$ выполняется Compare. Что это означает?

Иногда нужно определить, меньше, больше или равно одно число другому. По существу это вычитание B из A. Нулевой результат означает равенство. Иначе установленный Carry показывает, что B больше A; неустановленный - что A больше B.

Compare совпадает с Subtract, кроме важного отличия: результат никуда не сохраняется, сохраняется только Carry.

Так программа узнаёт отношение операндов, не разрушая A результатом вычитания. Следующая инструкция сможет проверить флаги и выбрать дальнейшее действие.

Но для равенства необходимо знать, был ли результат нулевым. Значит, нужен ещё *флаг Zero*, сохраняемый вместе с Carry.

Добавим и *флаг Sign*. Он устанавливается, если старший бит результата равен 1. В дополнительном коде это означает отрицательное число; 0 означает положительное.

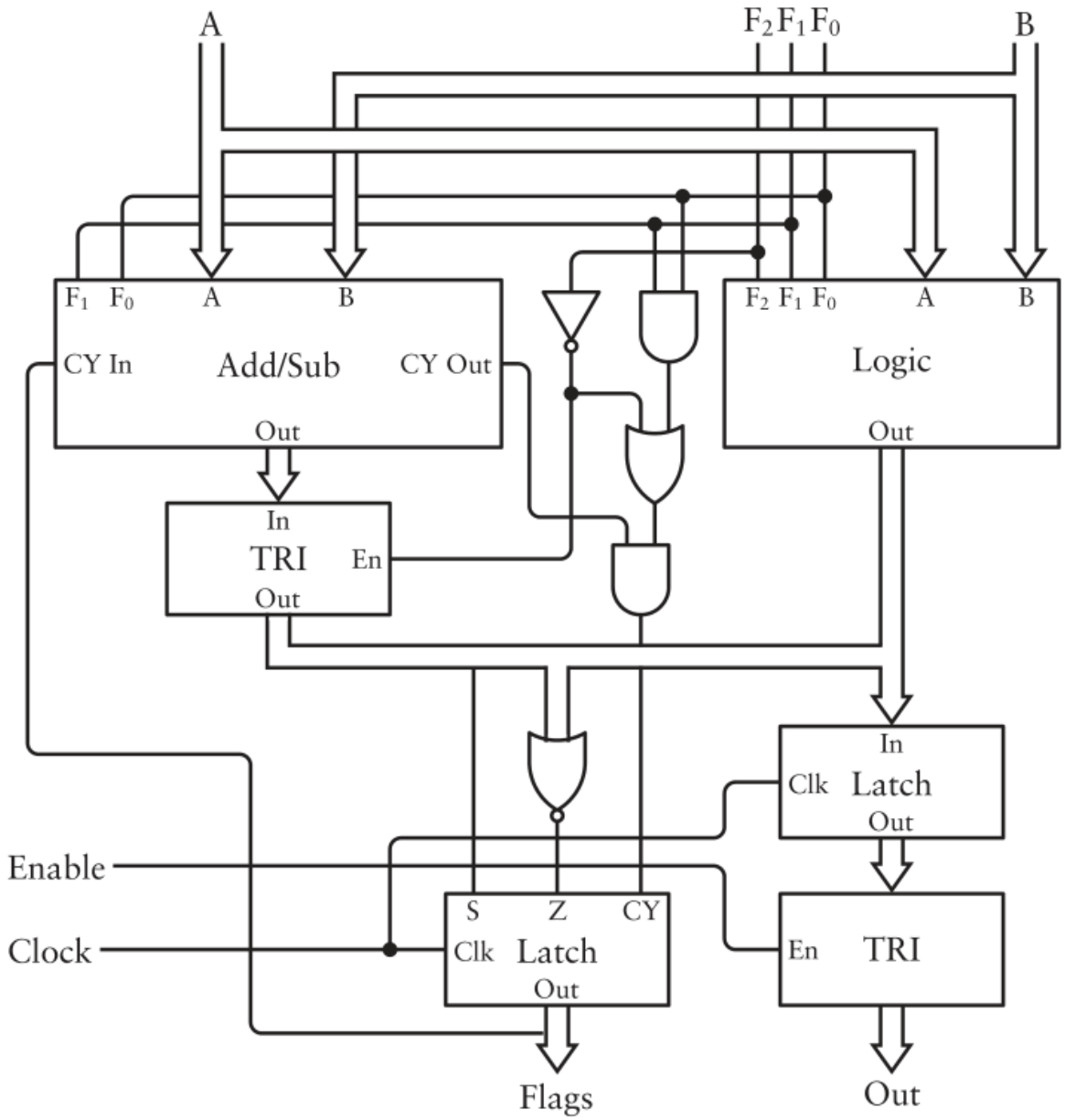
Intel 8080 на самом деле определяет пять флагов. Мы не будем реализовывать Auxiliary Carry, показывающий перенос из младших четырёх битов в старшие. Он нужен инструкции Decimal Adjust Accumulator, преобразующей аккумулятор из двоичного вида в BCD. Не будет и Parity, равного 1 при чётном числе единичных битов результата. Его легко построить из семи XOR, но он менее полезен.

Auxiliary Carry относится к границе между двумя шестнадцатеричными цифрами байта: между младшей и старшей тетрадами. Decimal Adjust Accumulator использует его для коррекции BCD-результата. Parity ничего не говорит о величине числа; он сообщает лишь, чётно ли количество единиц во всех восьми разрядах.

Для некоторых программ Compare важнее сложения. Например, поиск текста на веб-странице требует сравнивать символы страницы с символами искомой строки.

Такое сравнение повторяется снова и снова: программа сопоставляет первый символ, затем следующий и продолжает, пока не найдёт полное совпадение или различие. Флаги позволяют решить, продолжить сравнение, перейти к следующей позиции либо объявить строку найденной.

Полное ALU объединяет Add/Subtract и Logic с довольно запутанной вспомогательной схемой:



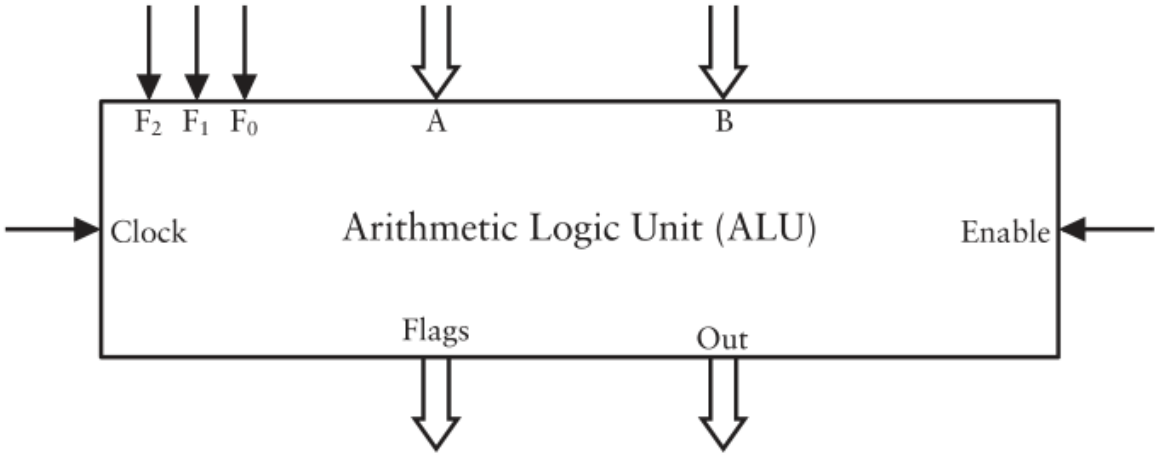
Два блока TRI являются трёхсостоятельными буферами. Logic разрешает выход только для сочетаний F, выбирающих AND, OR или XOR. Буфер Add/Subtract разрешён, когда $F_2 = 0$, то есть при сложении или вычитании.

Внизу две защёлки получают общий Clock всего ALU. Ещё один TRI управляется общим Enable. Правый нижний буфер выдаёт объединённый результат арифметического или логического блока.

Большая часть вентилей обслуживает Carry, CY. Его следует сохранять, если $F_2 = 0$, то есть выполнялось сложение или вычитание, либо если F_1 и F_0 равны 1, что обозначает Compare.

Три флага поступают в центральную нижнюю защёлку. Восьмивходовый ИЛИ-НЕ определяет, равен ли весь результат нулю, формируя Zero, Z. Старший бит Data Out является Sign, S. Хотя флагов три, при выходе ALU они рассматриваются как три бита байта. Carry возвращается наверх как CY In модуля Add/Subtract.

Спрячем всю сложную логику в простом блоке:



Арифметико-логическое устройство готово.

Хотя ALU чрезвычайно важно, CPU нуждается не только в способе выполнять арифметические и логические операции. Нужно вводить числа в ALU, сохранять результаты и перемещать их. Это следующий шаг.

Эта страница намеренно оставлена пустой.

Глава 22. Регистры и шины

Многие повседневные операции компьютера состоят в перемещении «вещей», а под вещами я, конечно, понимаю байты. Мы сталкиваемся с этим при загрузке и сохранении файла, потоковом воспроизведении музыки и фильмов, видеосвязи. Если байты движутся недостаточно быстро, звук или изображение замирает или искажается. Это знакомо всем.

На более низком уровне байты перемещаются и внутри CPU: из памяти в CPU, затем в ALU. Результаты ALU иногда снова поступают в него для новых операций, прежде чем вернуться в память.

Такое движение не столь эффектно, как вычисления ALU, но не менее необходимо.

При перемещении байты хранятся в наборе защёлок. Трёхбайтовый аккумулятор главы 20 имел четыре восьмибитные защёлки: одну для кода инструкции и три для данных.

Процессору на основе Intel 8080 понадобится больше четырёх. Сначала сосредоточимся на семи особых восьмибитных защёлках, которыми непосредственно управляют инструкции CPU. Они называются *регистрами*, и их основное назначение - хранить байты во время обработки ALU.

Все семь важны, но особенно важен *аккумулятор*. У ALU два входа A и B.

В Intel 8080 первый вход ALU всегда получает значение аккумулятора, а результат ALU всегда сохраняется обратно в аккумулятор. Оттуда его можно перенести в другой регистр или память.

Семь регистров обозначаются буквами. Аккумулятор также называется A. Четыре других - B, C, D и E. Последние два не F и G: они часто используются вместе как шестнадцатибитный адрес памяти, поэтому называются H и L, high byte и low byte. Итак: A, B, C, D, E, H и L.

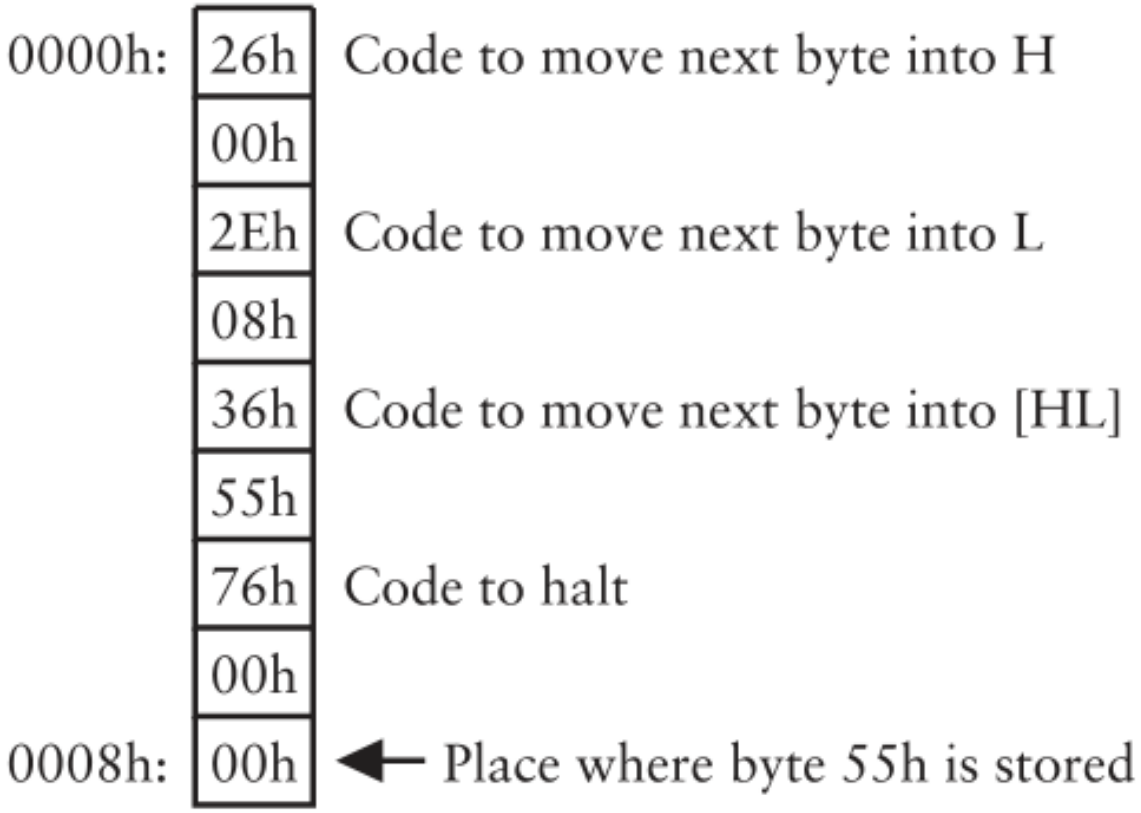
Код 3Eh, ранее описанный как «переместить следующий байт в CPU», точнее означает перенос следующего байта памяти в аккумулятор A. Он входит в семейство:

Код	Значение
06h	следующий байт в B
0Eh	следующий байт в C
16h	следующий байт в D
1Eh	следующий байт в E
26h	следующий байт в H
2Eh	следующий байт в L
36h	следующий байт в память по адресу [HL]
3Eh	следующий байт в A

Числовой порядок не совпадает с алфавитным, но так уж устроено.

Код 36h отличается: следующий байт сохраняется не в защёлке, а по шестнадцатибитному адресу, образованному H и L и обозначенному [HL].

Небольшая программа использует 26h, 2Eh и 36h:



Первая инструкция переносит 00h в H, вторая - 08h в L. Вместе HL образуют адрес 0008h. Затем 36h сохраняет следующий байт 55h по адресу [HL]. Код 76h останавливает CPU.

Использование H и L для формирования адреса называется *косвенной адресацией* и оказывается очень полезным.

В восьми кодах виден шаблон:

```
00 DDD 110
```

DDD обозначает получателя:

DDD	Регистр или память
000	B
001	C
010	D
011	E
100	H
101	L
110	[HL], или M
111	A

Код 110 означает память по адресу HL. Мы предпочитаем [HL], но документация Intel называет её M, словно M - ещё один регистр.

Однобайтовый процессор теоретически допускает 256 опкодов. Intel 8080 определяет 244, оставляя 12 значений свободными. Уже встречались два семейства по восемь кодов, а также 32h для записи по следующему адресу и 76h для остановки.

Следующая таблица содержит 64 кода:

Источник	ADD	ADC	SUB	SBB	ANA	XRA	ORA	CMP
B	80h	88h	90h	98h	A0h	A8h	B0h	B8h
C	81h	89h	91h	99h	A1h	A9h	B1h	B9h
D	82h	8Ah	92h	9Ah	A2h	AAh	B2h	BAh
E	83h	8Bh	93h	9Bh	A3h	ABh	B3h	BBh
H	84h	8Ch	94h	9Ch	A4h	ACh	B4h	BCh
L	85h	8Dh	95h	9Dh	A5h	ADh	B5h	BDh
M	86h	8Eh	96h	9Eh	A6h	AEh	B6h	BEh
A	87h	8Fh	97h	9Fh	A7h	AFh	B7h	BFh

	Arithmetic or Logical Operation							
Source	ADD	ADC	SUB	SBB	ANA	XRA	ORA	CMP
B	80h	88h	90h	98h	A0h	A8h	B0h	B8h
C	81h	89h	91h	99h	A1h	A9h	B1h	B9h
D	82h	8Ah	92h	9Ah	A2h	AAh	B2h	BAh
E	83h	8Bh	93h	9Bh	A3h	ABh	B3h	BBh
H	84h	8Ch	94h	9Ch	A4h	ACh	B4h	BCh
L	85h	8Dh	95h	9Dh	A5h	ADh	B5h	BDh
M	86h	8Eh	96h	9Eh	A6h	AEh	B6h	BEh
A	87h	8Fh	97h	9Fh	A7h	AFh	B7h	BFh

Это более четверти всех инструкций 8080 и ядро арифметики и логики. Сокращения означают Add, Add with Carry, Subtract, Subtract with Borrow, AND, XOR, OR и Compare. Это *мнемоники*: короткие слова для запоминания операций и записи программ.

Каждая операция сочетается с одним из семи регистров или M. Вместо «прибавить E к аккумулятору» или «83h» можно сказать:

```
ADD E
```

Сумма E и аккумулятора сохраняется в аккумуляторе.

Вместо «выполнить XOR аккумулятора и байта [HL]» или «AEh» пишут:

```
XRA M
```

Результат снова сохраняется в аккумуляторе.

Эти обозначения официально называются *инструкциями языка ассемблера*. Термин возник в начале 1950-х и связан со сборкой программы. `XRA M` и AEh обозначают одну и ту же инструкцию в разных формах.

Все 64 кода имеют вид:

```
10 FFF SSS
```

FFF определяет функцию ALU из главы 21, SSS - источник по уже приведённой таблице.

Семейство переноса следующего байта называется move immediate, MVI:

Инструкция	**Опкод**
MVI B,data	06h
MVI C,data	0Eh
MVI D,data	16h
MVI E,data	1Eh
MVI H,data	26h
MVI L,data	2Eh
MVI M,data	36h
MVI A,data	3Eh

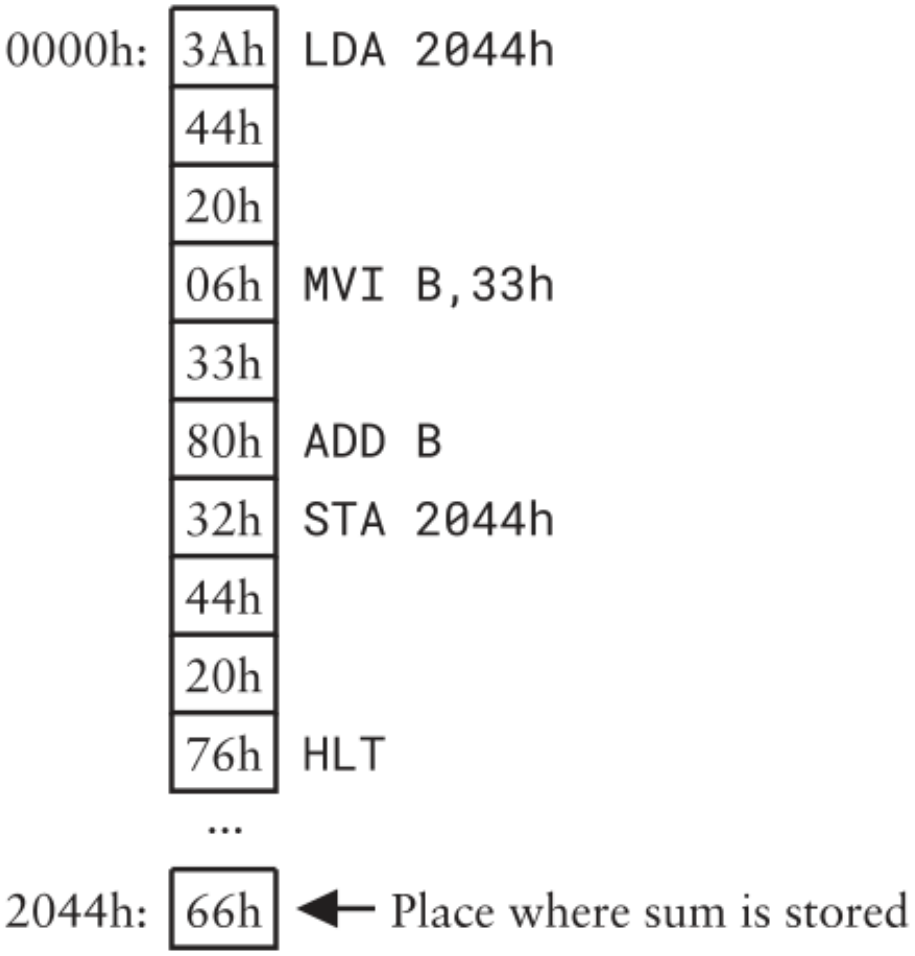
Опкод теперь показан последним, подчёркивая первичность записи на ассемблере. `data` - байт после опкода.

Код 32h сохраняет аккумулятор по адресу, находящемуся после опкода. Парный код 3Ah загружает байт по такому адресу:

Инструкция	**Опкод**
STA addr	32h
LDA addr	3Ah

STA означает store accumulator, LDA - load accumulator. `addr` - шестнадцатибитный адрес, заданный двумя байтами после кода. HLT соответствует 76h.

Программа показывает соответствие машинных байтов и ассемблера:



LDA загружает из адреса 2044h значение 66h. MVI помещает 33h в B. ADD прибавляет B к аккумулятору, получая 99h. STA записывает сумму обратно по 2044h, заменяя 66h на 99h.

Непосредственные арифметические и логические инструкции имеют официальные мнемоники:

Инструкция	**Опкод**
ADI data	C6h
ACI data	CEh
SUI data	D6h
SBI data	DEh
ANI data	E6h
XRI data	EEh
ORI data	F6h
CPI data	FEh

Они используют байт непосредственно после опкода: add immediate, add with carry immediate и так далее. Операция всегда включает аккумулятор, и результат возвращается туда же.

Предыдущую программу можно упростить: вместо загрузки 33h в B и последующего ADD B выполнить `ADI 33h`.

Intel 8080 определяет 63 инструкции переноса между регистрами, из [HL] в регистр или из регистра в [HL]:

Opcodes for `MOV destination,source`								
	Destination							
Source	B	C	D	E	H	L	M	A
B	40h	48h	50h	58h	60h	68h	70h	78h
C	41h	49h	51h	59h	61h	69h	71h	79h
D	42h	4Ah	52h	5Ah	62h	6Ah	72h	7Ah
E	43h	4Bh	53h	5Bh	63h	6Bh	73h	7Bh
H	44h	4Ch	54h	5Ch	64h	6Ch	74h	7Ch
L	45h	4Dh	55h	5Dh	65h	6Dh	75h	7Dh
M	46h	4Eh	56h	5Eh	66h	6Eh		7Eh
A	47h	4Fh	57h	5Fh	67h	6Fh	77h	7Fh

Полная таблица устроена так: столбцы задают получателя, строки - источник.

Источник \ Получатель	**B**	**C**	**D**	**E**	**H**	**L**	**M**	**A**
B	40h	48h	50h	58h	60h	68h	70h	78h
C	41h	49h	51h	59h	61h	69h	71h	79h
D	42h	4Ah	52h	5Ah	62h	6Ah	72h	7Ah
E	43h	4Bh	53h	5Bh	63h	6Bh	73h	7Bh
H	44h	4Ch	54h	5Ch	64h	6Ch	74h	7Ch
L	45h	4Dh	55h	5Dh	65h	6Dh	75h	7Dh
M	46h	4Eh	56h	5Eh	66h	6Eh	-	7Eh
A	47h	4Fh	57h	5Fh	67h	6Fh	77h	7Fh

Они обозначаются MOV и записываются с получателем, затем источником. Код 69h:

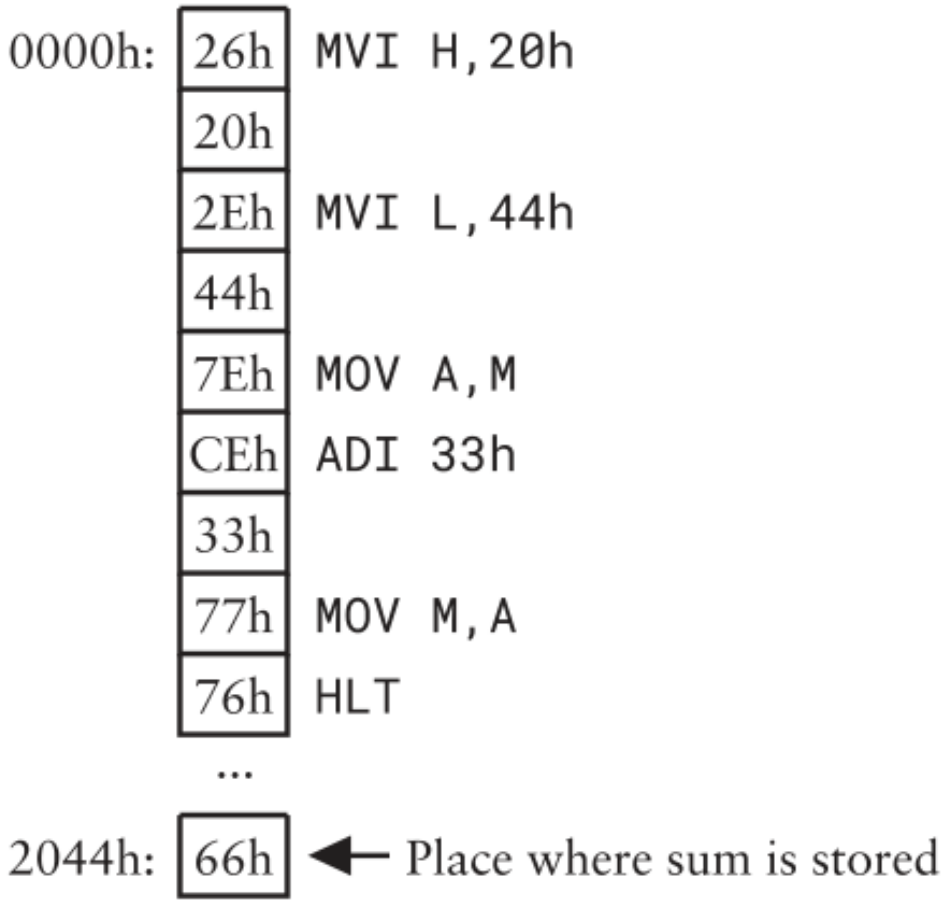
```
MOV L,C
```

означает L <- C. Прежнее содержимое L заменяется значением C; C не изменяется, и после операции оба равны.

Семь инструкций вроде `MOV C,C` фактически ничего не делают. `MOV M,M` нет: соответствующий код 76h занят HLT.

Хотя такие самопереносы ничего не изменяют, их коды естественно возникают из общего двоичного шаблона. Исключение 76h разрушает только одну клетку таблицы и отдаёт особенно удобный код инструкции остановки.

Программу сложения значения с байтом по адресу 2044h можно записать иначе:



Этот вариант показывает удобство HL. Регистры один раз устанавливаются в 2044h. Первый MOV переносит значение [HL], 66h, в аккумулятор. К нему прибавляется 33h. Второй MOV записывает аккумулятор обратно в [HL] без повторного указания адреса.

Все 63 MOV имеют вид:

```
01 DDD SSS
```

DDD - получатель, SSS - источник:

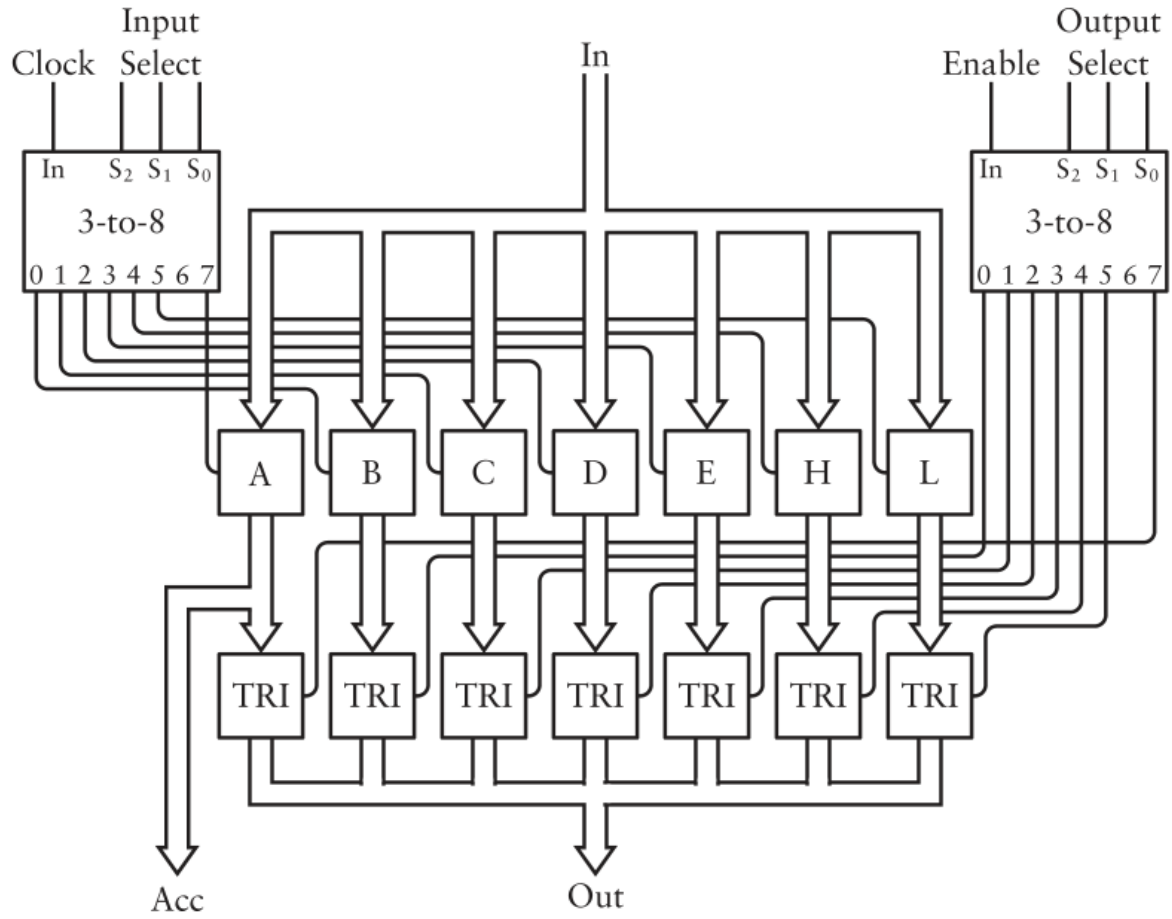
Код	Регистр или память
000	B
001	C
010	D
011	E
100	H
101	L
110	M
111	A

Один путь проектирования CPU - сначала выбрать инструкции, затем определить необходимую схему. Именно так мы поступаем: берём подмножество 8080 и строим оборудование.

Для инструкций с семью регистрами CPU должен сохранять байты в семи защёлках и извлекать их по трёхбитным кодам. Код 110 пока игнорируется как особый случай памяти. Остальные семь подаются на декодеры 3 в 8.

Устройство решает две независимые задачи. По коду получателя оно открывает Clock ровно одной защёлки и записывает входной байт. По коду источника оно разрешает ровно один выход, не создавая конфликта с остальными регистрами.

Схема содержит семь защёлок и семь трёхсостоятельных буферов. Один декодер направляет входной байт в нужную защёлку, другой разрешает буфер выбранного выхода:



Это *массив регистров*, важнейшая схема главы. Верхний восьмибитный тракт In несёт сохраняемый байт. Семь буквенных блоков - восьмибитные защёлки с Clock слева.

Хотя общий In соединён со всеми защёлками, байт сохраняется только там, куда левый декодер направил тактовый импульс. Остальные удерживают прежние значения. Все выходы тоже физически объединены через буферы, но электрически с Out соединён лишь один разрешённый TRI.

Слева и справа находятся декодеры 3 в 8 с S_2-S_0, значения которых совпадают с кодами регистров. Поэтому выход 6 не используется: 110 означает память, а не защёлку.

Левый декодер управляет Clock: в зависимости от S_2-S_0 сохраняет In в одном регистре. Под каждой защёлкой находится TRI с Enable. Правый декодер разрешает один буфер, и его байт появляется на Out.

У декодеров одинаковая кодировка, но их Select не обязаны совпадать. Поэтому можно читать один регистр и записывать другой. Для `MOV L,C` правый декодер выбирает C как источник, а левый - L как получателя.

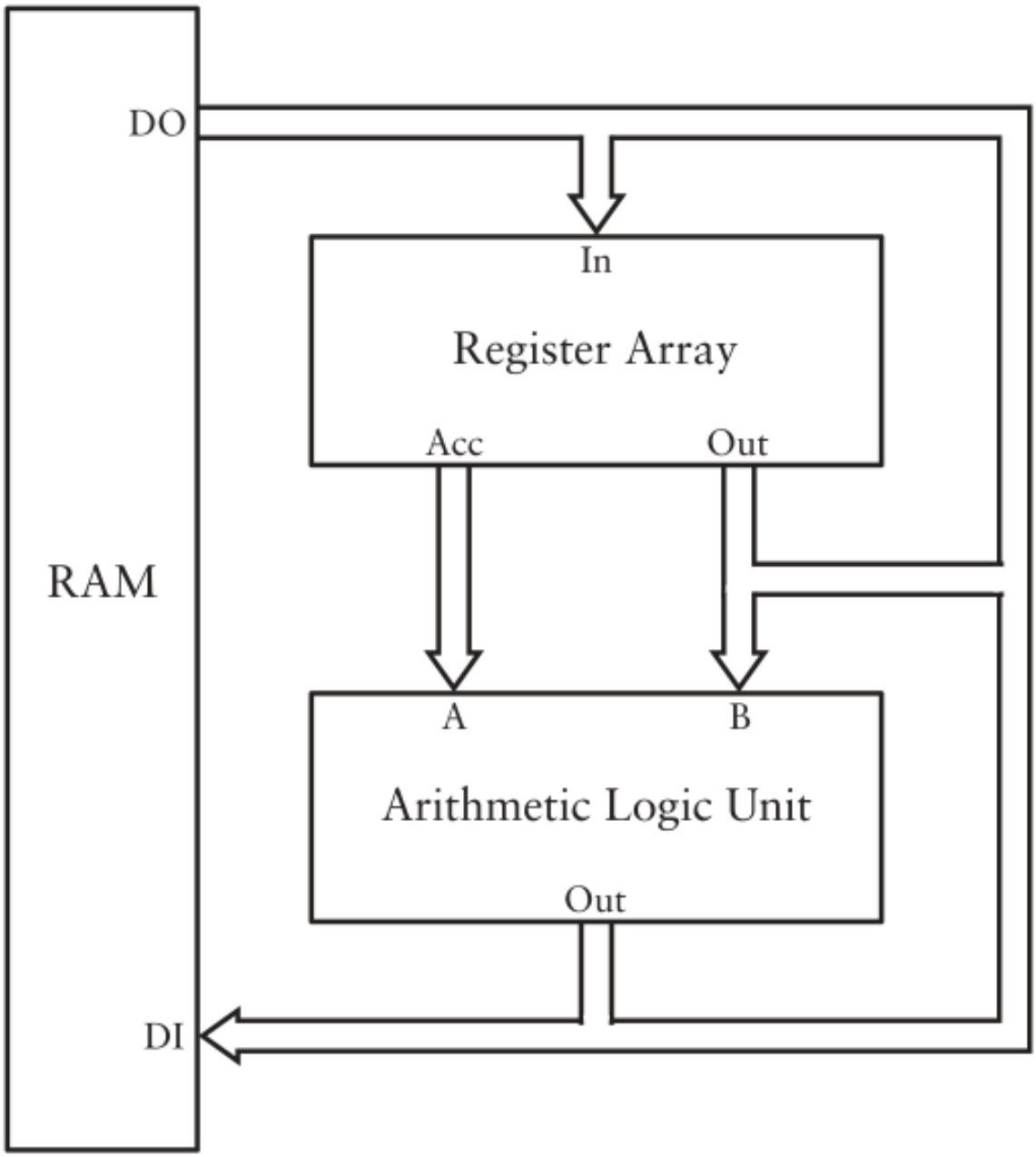
Аккумулятор обрабатывается особо: его значение всегда доступно на отдельном выходе Acc.

Отдельный путь нужен потому, что вход A арифметико-логического устройства всегда связан с аккумулятором. CPU не должен занимать общую шину только для подачи A в ALU: значение Acc присутствует там постоянно.

Вход массива может получать байт из памяти, другого регистра или ALU. Выход можно сохранить в памяти, другом регистре или направить в ALU.

Массив сам не определяет происхождение и дальнейшую судьбу байта. Он лишь умеет принять значение с общего тракта и предоставить сохранённое значение. Источник и получатель задаются управляющей схемой всего CPU.

Упрощённая блок-схема не даёт потерять лес за деревьями:



На ней показаны только главные восьмибитные пути. Кроме отдельного Acc -> A ALU, все остальные входы и выходы объединены. Даже Data Out RAM соединён с Data In.

Это возможно, потому что каждый выход проходит через TRI и в каждый момент разрешён лишь один. Его байт доступен всем: может быть записан в память через Write, сохранён в одном из регистров либо стать результатом ALU.

Общее соединение называется *шиной данных*. Когда TRI разрешён, его байт присутствует по всей шине и доступен любому подключённому компоненту.

Шина не является отдельным хранилищем. Это восемь общих проводников, по которым в конкретный момент распространяется значение одного источника. Чтобы получатель сохранил байт, его Clock должен прийти после стабилизации сигналов.

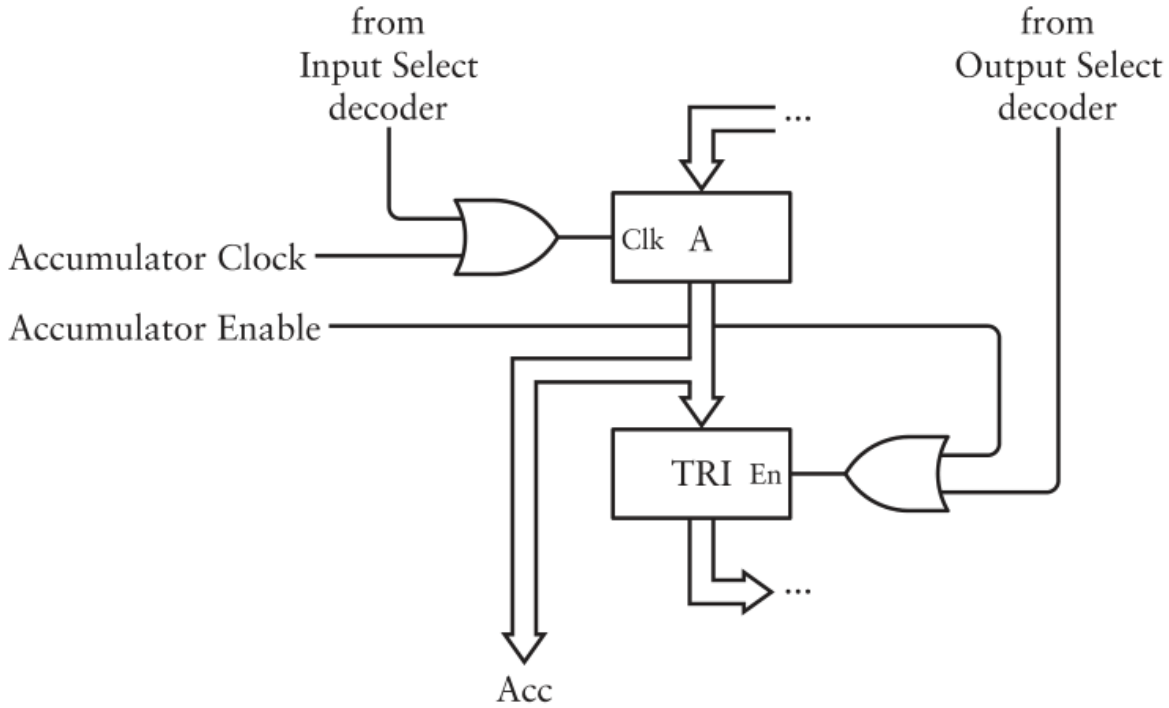
Если случайно разрешить два источника с различными битами, один может подать напряжение, а другой соединить ту же линию с землёй. Поэтому управление TRI является и логической, и электрической необходимостью.

Эта шина восьмибитная. Для шестнадцатибитного адреса будет отдельная *шина адреса*, поскольку адрес тоже может поступать из разных источников.

Теперь неприятные подробности. Массив хорош для MOV между регистрами: один декодер выводит источник на шину, второй сохраняет в получателе. Но он не подходит для STA и LDA, переносящих данные между аккумулятором и памятью. Все арифметические и логические инструкции тоже сохраняют результат в A.

Для `STA addr` аккумулятор должен появиться на шине без обычного кода Output Select, чтобы RAM получила Data In. Для `LDA addr`, наоборот, A должен сохранить Data Out памяти без выбора через Input Select. Результат ALU требует такого же независимого такта записи.

Поэтому A должен независимо от декодеров принимать значение и выводить его. Для этого добавляется немного логики только к защёлке и TRI аккумулятора:

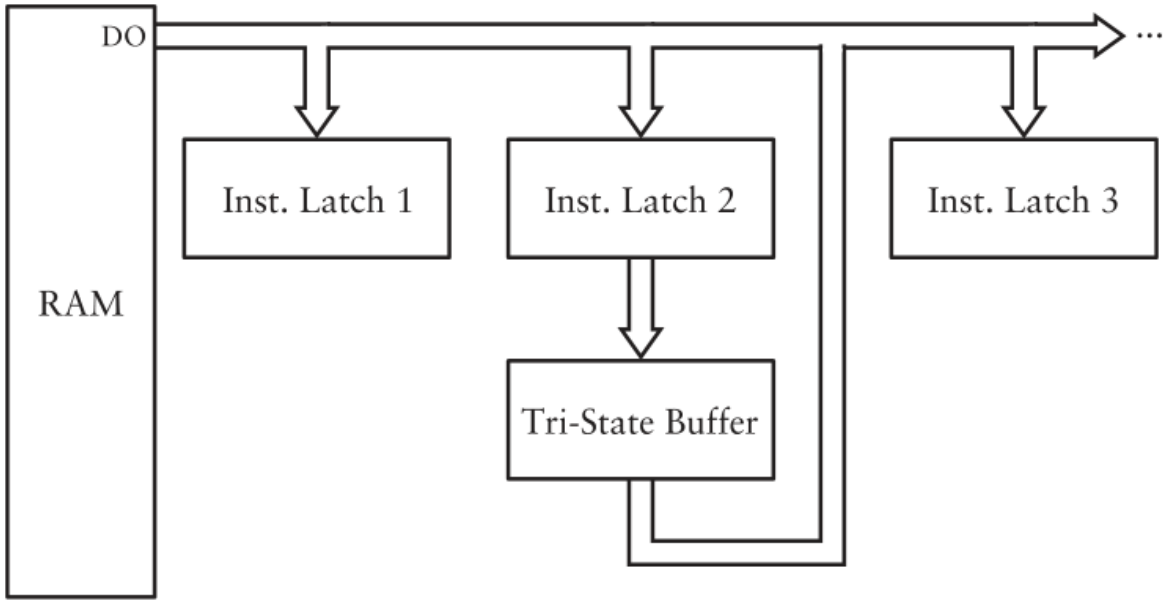


Два дополнительных сигнала слева входят в два ИЛИ. Они позволяют сохранить байт шины в A независимо от Input Select и вывести A на шину независимо от Output Select.

Обычные сигналы декодеров при этом продолжают работать. ИЛИ лишь добавляют второй способ открыть Clock защёлки A и Enable её буфера. Поэтому аккумулятор всё ещё может участвовать в MOV как обычный регистр, но одновременно получает особые пути для LDA, STA и ALU.

Для адресации через HL массив тоже придётся расширить, но это изменение серьёзнее, поэтому пока займёмся остальными необходимыми частями.

К шине данных подключаются ещё три восьмибитные защёлки для байтов инструкции:



Опкод всегда сохраняется в Instruction Latch 1. После этого его биты создают остальные управляющие сигналы CPU, что будет показано в следующей главе.

Три защёлки не означают, что каждая инструкция обязательно имеет три байта. Однобайтовой достаточно первой. В двухбайтовой используется первая и вторая, а трёхбайтовая занимает все три. Управляющая схема должна знать длину кода и подавать Clock на нужное число защёлок.

Instruction Latch 2 используется для инструкций с дополнительным байтом: MVI и непосредственных арифметических, например ADI. Для ADI этот байт прибавляется к аккумулятору, поэтому Latch 2 должен выводиться на шину через TRI.

Для MVI тот же выход TRI помещает непосредственный байт на шину, после чего Input Select и Clock нужного регистра сохраняют его. Для ADI этот байт с шины поступает на вход B ALU, тогда как аккумулятор уже находится на входе A.

Instruction Latches 2 и 3 вместе обслуживают трёхбайтовые STA и LDA: второй и третий байты образуют шестнадцатибитный адрес.

Кроме шины данных CPU нужна шестнадцатибитная шина адреса для 64 КБ. Адрес поступает из трёх источников:

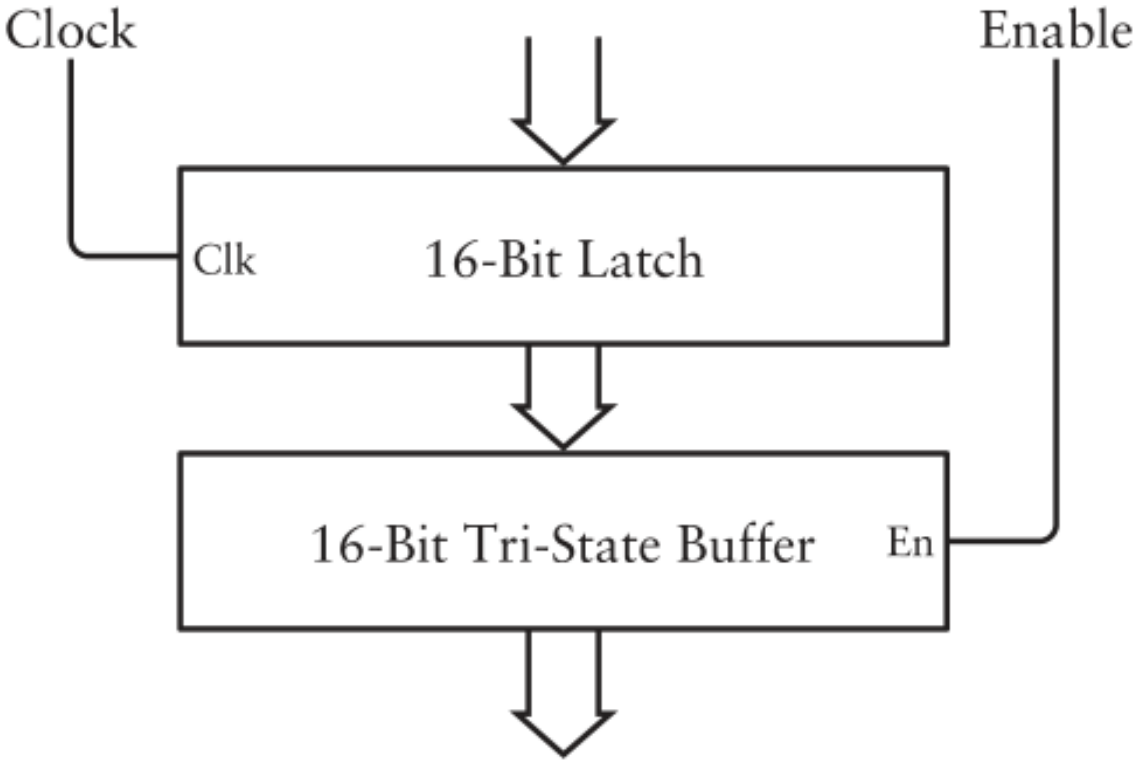
• *счётчик команд*, шестнадцатибитное значение адреса инструкции; начинается с 0000h и обычно растёт до HLT;

• два байта после STA или LDA;

• пара HL, например в `MOV A,M`.

В трёхбайтовом аккумуляторе память перебирал счётчик. В CPU обычного счётчика не будет, потому что некоторые инструкции могут установить адрес следующей команды. Поэтому program counter - шестнадцатибитная защёлка, обычно увеличиваемая на 1 после чтения каждого байта инструкции.

Если бы использовался простой необратимый счётчик, процессор мог бы двигаться только вперёд по последовательным адресам. Защёлка позволяет позднее загрузить совершенно новый адрес, что необходимо для переходов, циклов и вызовов программ.



Вход и выход шире прежних, потому что имеют 16 бит, и оба подключены к шине адреса.

Любое значение шины адреса сохраняется в program counter через Clock, а Enable выводит содержимое защёлки на шину.

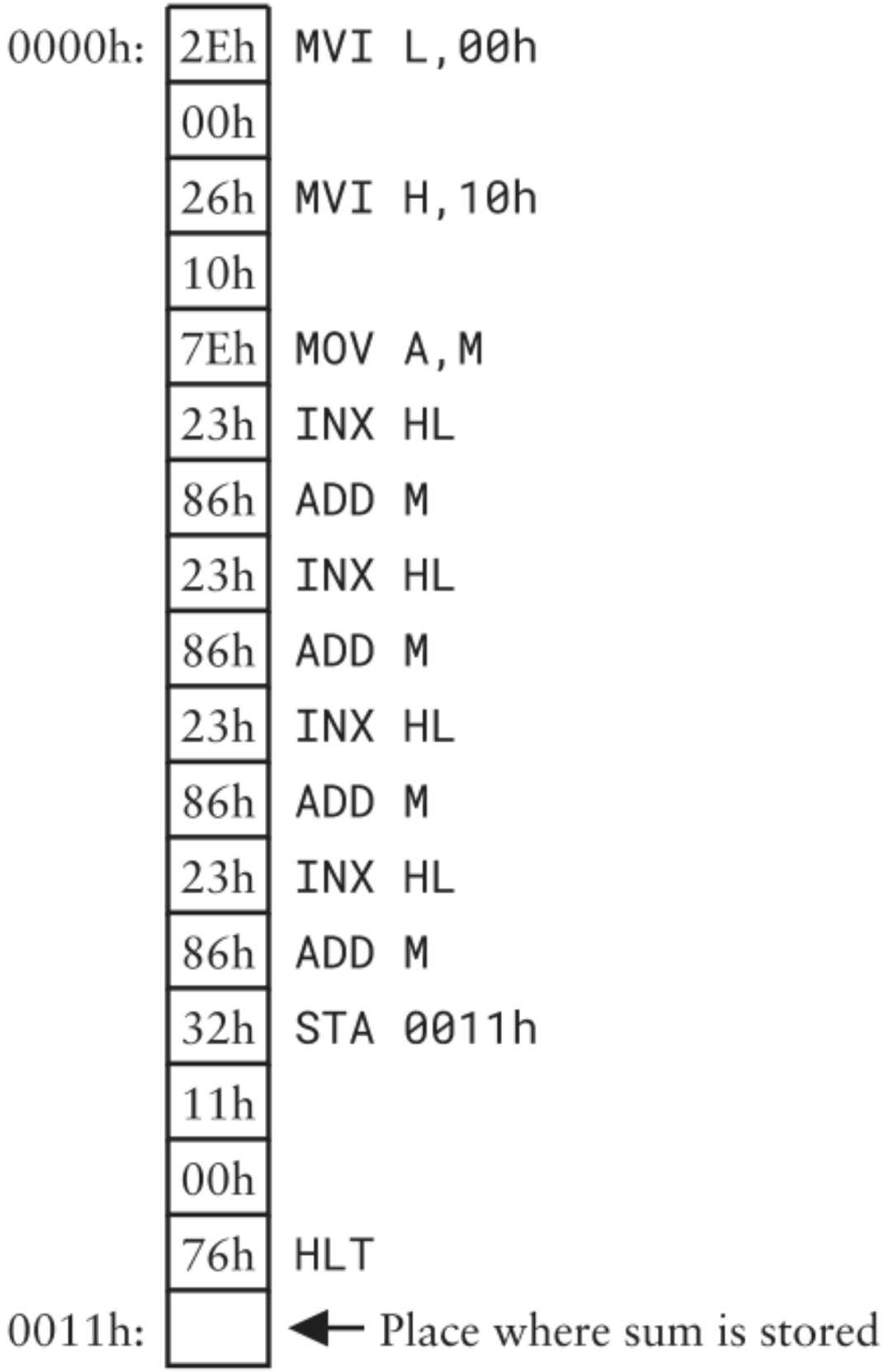
Эти действия происходят по очереди. Сначала Enable счётчика команд выставляет адрес следующего байта на шине и RAM. После чтения прежнее значение переносится через шину в инкрементор, увеличивается и по Clock возвращается в защёлку.

Для STA/LDA защёлки 2 и 3 тоже соединяются с шиной адреса через шестнадцатибитный TRI. Для MOV с M к ней должны подключаться H и L, хотя первоначальный массив регистров связывал их только с шиной данных.

Добавим две инструкции:

Инструкция	Описание	Опкод
INX HL	увеличить HL	23h
DCX HL	уменьшить HL	2Bh

INX прибавляет 1 к шестнадцатибитной паре HL, DCX вычитает 1. Особенно полезен INX. Если пять байтов лежат с адреса 1000h и их надо сложить, достаточно один раз установить HL, а затем увеличивать после каждого чтения:



Без INX пришлось бы после каждого байта снова загружать H и L новым непосредственным адресом. При последовательном массиве данных одна короткая инструкция заменяет повторяющиеся загрузки и делает программу компактнее.

Флаги ALU показывают ноль, знак и перенос, но INX и DCX их не изменяют.

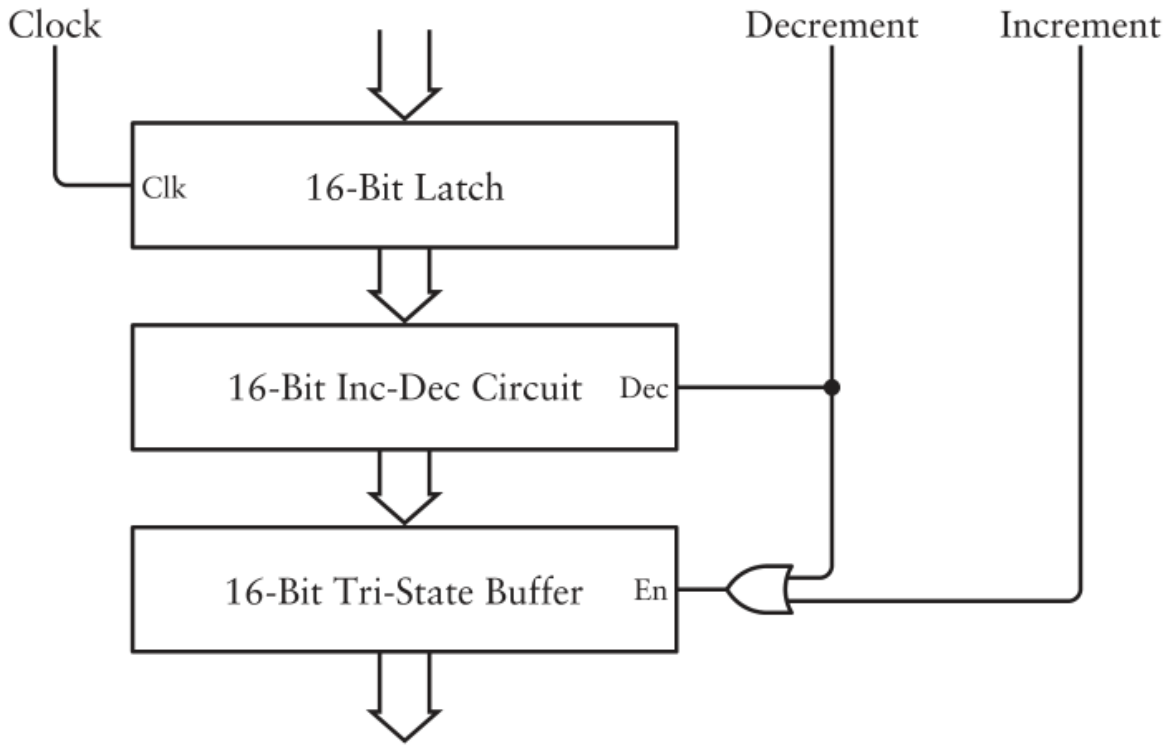
Intel 8080 также имеет INX/DCX для BC и DE, но они менее полезны и не войдут в наш CPU. Не будут реализованы и восьмибитные INR/DCR для семи регистров и M.

Для INX/DCX нужна схема шестнадцатибитного увеличения и уменьшения. Program counter также надо увеличивать после чтения каждого байта инструкции.

Одна и та же аппаратная часть может обслуживать оба назначения: временно принять значение program counter или HL, изменить его на единицу и вернуть результат исходной защёлке. Это ещё один пример совместного использования оборудования разными инструкциями.

Инкрементор-декрементор проще сумматора: он прибавляет или вычитает только 1. В восьмибитной версии I с индексами - входы, O - выходы, а Dec = 0 означает увеличение, Dec = 1 - уменьшение. Набор XOR и AND заключим в компонент с шестнадцатибитной защёлкой и TRI.

Назовём его *инкрементором-декрементором*:



Как у program counter, шестнадцатибитные вход защёлки и выход TRI подключены к шине адреса. Increment или Decrement разрешает TRI, причём первый выдаёт значение плюс 1, второй - минус 1.

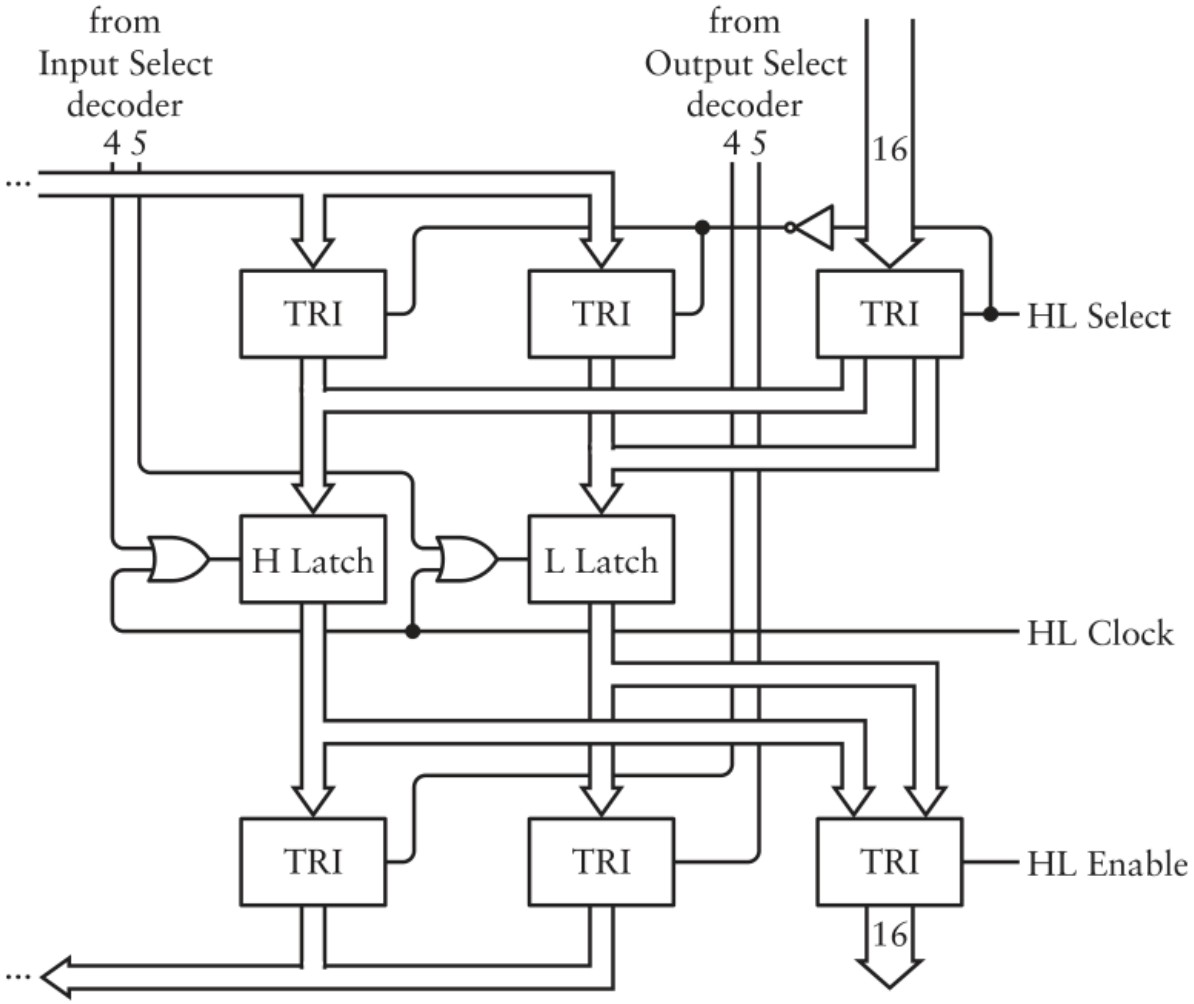
Сначала исходное значение с шины сохраняется во внутренней шестнадцатибитной защёлке. Затем выбранный сигнал включает комбинационную схему и буфер. На шине появляется уже изменённое значение, которое можно защёлкнуть обратно в program counter или HL.

Шина адреса в первую очередь адресует RAM, но также перемещает шестнадцатибитные значения между частями CPU.

Например, program counter адресует инструкцию. После чтения его значение переносится в инкрементор, увеличивается и сохраняется обратно. Для `MOV A,M` память адресует HL; следующая `INX HL` переносит HL в инкрементор, увеличивает и возвращает.

Таким образом, шина адреса временно перестаёт быть только «адресом RAM» и превращается в общий шестнадцатибитный транспорт CPU. В каждый момент её назначение определяется тем, какой TRI разрешён и какая защёлка получает Clock.

Проблему массива регистров больше нельзя откладывать: H и L должны выводиться на шестнадцатибитную шину. Исправление:



Шестнадцатибитные вход и выход справа соединяются с шиной адреса, позволяя сохранять и извлекать пару HL.

Обычные восьмибитные пути H и L при этом сохраняются. Регистры по-прежнему доступны по отдельности через шину данных и MOV, а новый тракт рассматривает их как одну шестнадцатибитную пару. Сигналы должны выбрать один из этих двух режимов без конфликтов.

Добавлены:

• HL Select выбирает, поступают ли входы H и L с обычного входа массива или с шестнадцатибитного входа;

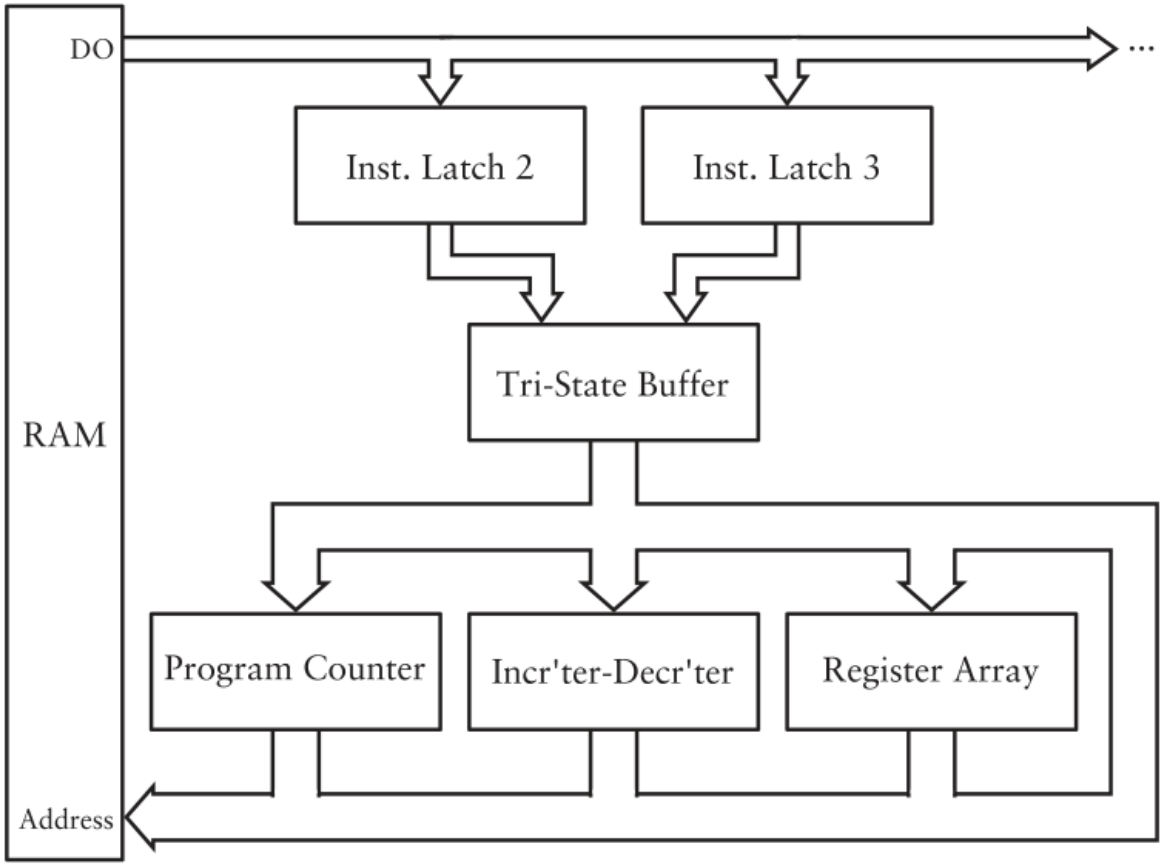
• HL Clock через два ИЛИ позволяет сохранить значение с шины адреса;

• HL Enable разрешает новый шестнадцатибитный TRI, выводящий объединённые H и L на шину адреса.

Какой беспорядок! Но никто не обещал, что строить компьютер легко.

Интерактивная версия полного массива регистров доступна на CodeHiddenLanguage.com.

Соединим шестнадцатибитные компоненты с шиной адреса:



Вверху показана часть шины данных, входящая в Instruction Latches 2 и 3. Их выходы объединяются TRI и подключаются к шине адреса. Широкая шина огибает нижние компоненты как их вход и выход и адресует RAM.

Как и прежняя блок-схема шины данных, эта схема не показывает важные сигналы: Enable буферов и Clock защёлок.

Именно эти правильно согласованные и синхронизированные сигналы заставляют CPU выполнять инструкции. Они являются пульсирующей жизненной силой компьютера и заслуживают отдельной главы. Она следующая.

Эта страница намеренно оставлена пустой.

Глава 23. Управляющие сигналы CPU

Старая инженерная поговорка гласит: последние 10% проекта требуют 90% работы. Как бы удручающе это ни звучало, об этом стоит помнить. Мы далеко продвинулись в постройке компьютера, но ещё не закончили. Оставшееся не составляет 90% работы, однако финиш может быть дальше, чем кажется.

Проектируемый CPU основан на Intel 8080. ALU и массив регистров из двух предыдущих глав являются его важнейшими частями. ALU выполняет арифметические и логические операции над байтами. Массив содержит защёлки регистров A, B, C, D, E, H и L. Ещё три защёлки сохраняют опкод и один или два следующих байта некоторых инструкций.

Компоненты соединены друг с другом и RAM двумя шинами: восьмибитной шиной данных для байтов и шестнадцатибитной шиной адреса. Есть program counter, хранящий адрес RAM, и инкрементор-декрементор для изменения шестнадцатибитного адреса.

Шины служат основными соединениями между всеми этими частями. Восьмибитная шина данных переносит байты от одного компонента к другому, а шестнадцатибитная шина адреса задаёт адрес ячейки памяти. Но одних шин недостаточно: компоненты также связаны сложным набором *управляющих сигналов*. Они так называются потому, что управляют совместной работой компонентов при исполнении инструкций, хранящихся в памяти.

Большинство сигналов относится к двум типам:

• помещает значение на одну из шин;

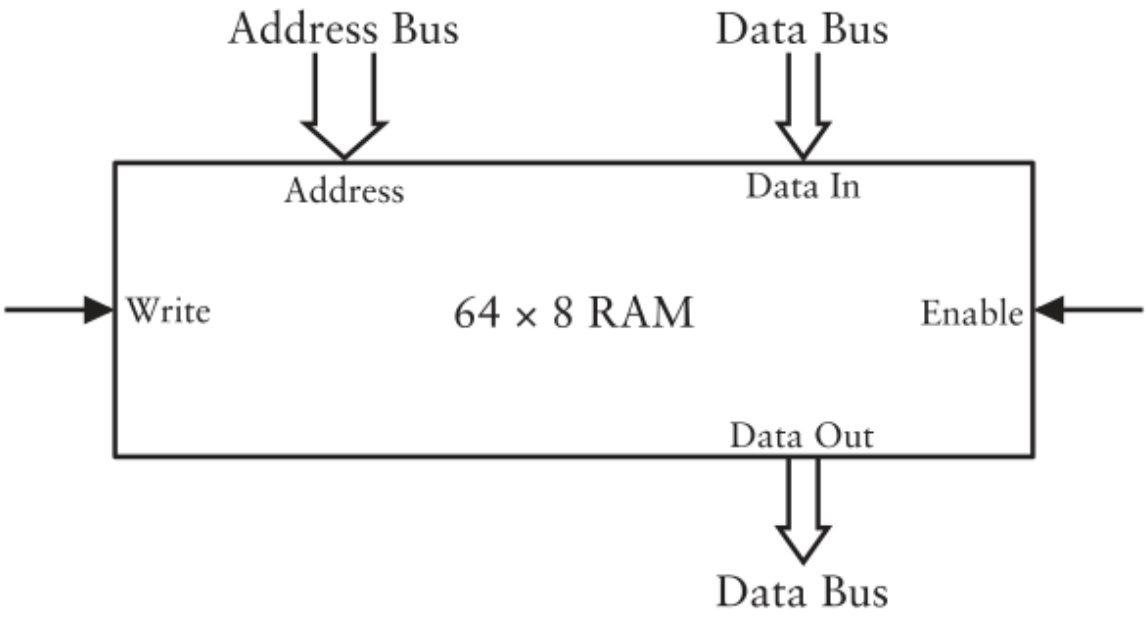
• сохраняет значение с одной из шин.

Эта глава целиком о том, как значения попадают на шину и сходят с неё. Сигналы первого типа подключаются к Enable трёхсостоятельных буферов, связывающих выходы с шиной. Сигналы второго обычно управляют Clock защёлок. Исключение - запись байта шины данных в память через RAM Write.

Именно синхронизация позволяет CPU выполнять инструкции. Так восьми- и шестнадцатибитные значения перемещаются между компонентами и памятью. Так коды в памяти управляют оборудованием, объединяя hardware и software. Можно представить кукловода и труппу марионеток в безупречно поставленном танце арифметики и логики. Управляющие сигналы - нити.

Иными словами, значение недостаточно просто поместить на шину: в согласованный момент другой компонент должен его принять. Благодаря этой точной синхронизации коды, находящиеся в памяти, действительно управляют аппаратурой компьютера. Именно здесь аппаратное и программное обеспечение соединяются в единое целое, на что намекает название книги.

Рассмотрим шесть основных компонентов, их шины и сигналы. Address RAM соединён с шиной адреса, Data In и Data Out - с шиной данных:



Write записывает байт шины данных в память по адресу на адресной шине; Enable включает трёхстабильный буфер RAM Data Out, и содержимое адресованной ячейки появляется на шине данных. К этому массиву памяти можно подключить пульт управления, позволяющий человеку записывать байты в память и просматривать их.

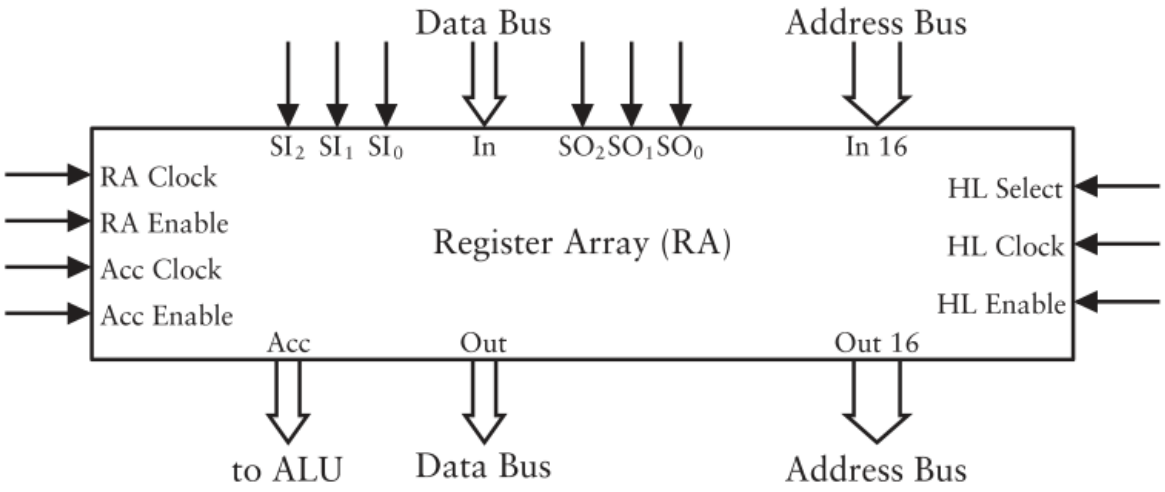
Самый сложный компонент - массив регистров, RA.

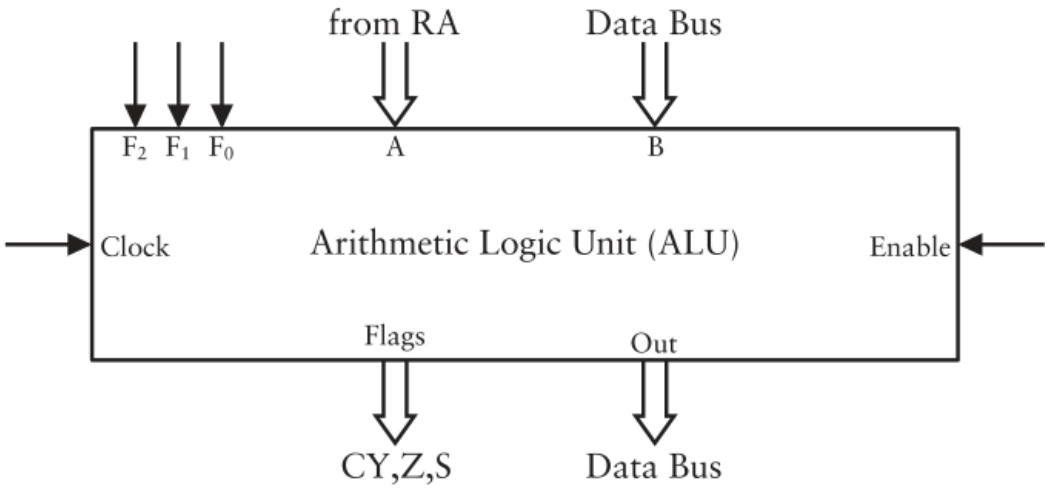
У RA два набора Select сверху. SI выбирает регистр, сохраняющий байт шины данных; RA Clock определяет момент сохранения. SO вместе с RA Enable выводит выбранный регистр на шину.

Как было показано в предыдущей главе, массив регистров усложнён в двух отношениях. Во-первых, есть два дополнительных сигнала аккумулятора, который далее иногда сокращённо обозначается Acc: Accumulator Clock независимо сохраняет значение с шины данных в аккумуляторе, а Accumulator Enable включает трёхстабильный буфер и независимо выводит Acc на шину.

Во-вторых, H и L соединены также с шиной адреса. HL Select выбирает шину адреса в качестве входа пары регистров; HL Clock сохраняет находящееся на ней шестнадцатибитное значение в H и L; HL Enable включает трёхстабильный буфер и выводит содержимое H и L на адресную шину.

ALU имеет F_0-F_2 для выбора сложения, вычитания, сравнения или логической функции:

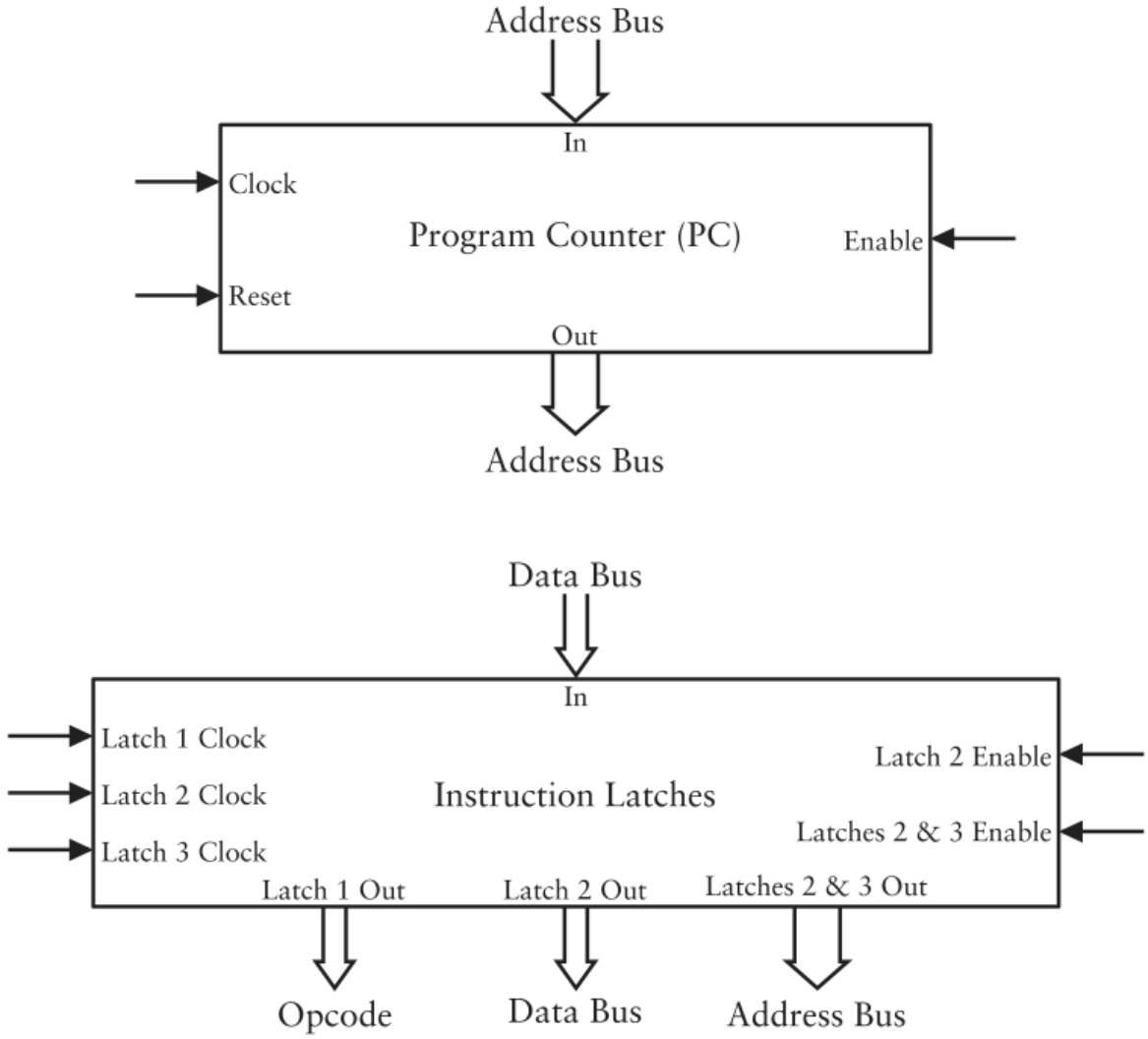




Вход B и выход Out ALU соединены с шиной данных, но вход A напрямую получает выход Acc массива регистров. Устройство несколько усложняется необходимостью сохранять флаги Carry (CY), Zero (Z) и Sign (S), значения которых устанавливаются в соответствии с выполненной арифметической или логической операцией.

Сигнал Clock ALU сохраняет результат операции в одной защёлке, а флаги - в другой. Сигнал Enable включает трёхстабильный буфер, помещающий сохранённый результат ALU на шину данных.

Ещё одна шестнадцатибитная защёлка хранит program counter, PC:



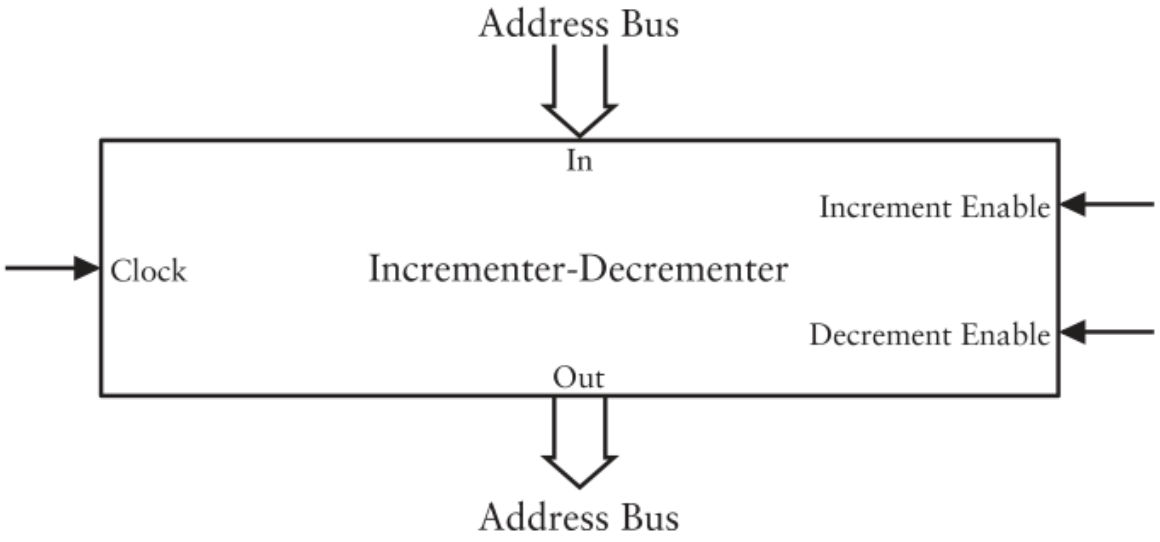
PC Clock сохраняет шестнадцатибитное значение с шины адреса в защёлке, Enable включает трёхстабильный буфер и выводит содержимое PC на эту шину, а Reset устанавливает все биты защёлки в 0, чтобы начать обращение к байтам памяти с адреса 0000h.

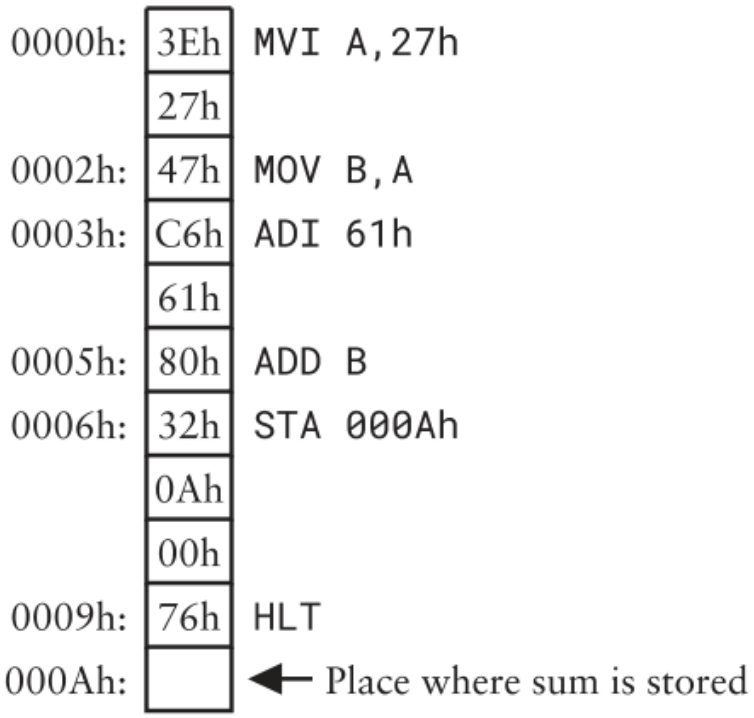
Три восьмибитные защёлки сохраняют до трёх байтов инструкции.

Некоторые инструкции состоят только из опкода, другие имеют ещё один или два байта. Три Clock сохраняют до трёх байтов.

Первый байт всегда опкод. Latch 2 Enable выводит второй байт на шину данных. Если после опкода идут два байта адреса, Latches 2 & 3 Enable выводит их на шину адреса.

Последний компонент увеличивает или уменьшает шестнадцатибитное значение, сокращённо Inc-Dec:





Clock сохраняет значение с адресной шины во внутренней защёлке инкрементора-декрементора. Два сигнала Enable справа включают на шину адреса либо увеличенное, либо уменьшенное значение.

Чтобы почувствовать необходимую координацию, рассмотрим программу из шести инструкций разной длины.

Программа переносит 27h в аккумулятор. `MOV B,A` копирует его в B. Затем 61h прибавляется к Acc, получая 88h. ADD B доводит значение до AFh. STA сохраняет его по адресу 000Ah. HLT останавливает CPU.

Для исполнения этих инструкций CPU использует счётчик команд, адресующий память и переносящий байты инструкций в соответствующие защёлки. Для первой инструкции `MVI A,27h` PC сначала устанавливается в 0000h, чтобы обратиться к первой инструкции в памяти. Её обработка требует пяти шагов. Каждый шаг помещает что-либо на адресную шину и где-либо это сохраняет, либо делает то же самое с шиной данных, либо выполняет обе операции одновременно.

Шаг 1 адресует RAM через PC и сохраняет 3Eh в Latch 1:

• PC Enable выводит 0000h на шину адреса;

• RAM Data Out Enable выводит 3Eh на шину данных;

• Inc-Dec Clock сохраняет 0000h;

• Instruction Latch 1 Clock сохраняет 3Eh.

Шаг 2 увеличивает PC:

• Increment Enable выводит 0001h на шину адреса;

• PC Clock сохраняет 0001h.

Теперь, когда первый байт инструкции сохранён в Latch 1, его можно использовать для управления последующими шагами. В данном случае шаги 3 и 4 повторяют первые два, но обращаются к байту по адресу 0001h и сохраняют 27h в Latch 2.

Шаги, читающие байты инструкции из памяти, вместе называются *выборкой инструкции*. Их назначение состоит в обращении к байтам инструкции в памяти и сохранении этих байтов в защёлках инструкции. После выборки `MVI A,27h` значение 27h находится в Latch 2, и теперь его нужно перенести в аккумулятор.

Шаг 5 - *исполнение*:

• Latch 2 Enable выводит 27h на шину данных;

• Accumulator Clock сохраняет его.

Во всех пяти шагах на каждой шине не более одного значения, после чего оно сохраняется в другом месте.

Вторая инструкция `MOV B,A` однобайтовая, поэтому выборка требует двух шагов. Исполнение:

• RA Enable выводит источник на шину данных;

• RA Clock сохраняет его в получателе.

Почему не упомянуты A и B? Коды MOV имеют вид:

```
01 DDD SSS
```

DDD и SSS из Latch 1 напрямую идут на два Select RA, поэтому достаточно общих Enable и Clock.

DDD обозначает регистр-получатель, а SSS - регистр-источник. Поскольку код операции уже сохранён в Instruction Latch 1, эти две трёхбитные группы сами выбирают A и B. Поэтому в микрокоманде исполнения нет необходимости отдельно называть конкретные регистры.

Следующая инструкция `ADI 61h` имеет вид:

```
11 FFF 110
```

FFF обозначает функцию, выполняемую инструкцией: Add, Add with Carry, Subtract, Subtract with Borrow, AND, XOR, OR или Compare. У ALU имеется соответствующий трёхбитный вход Function. Поэтому три функциональных бита кода операции из Instruction Latch 1 можно направить прямо на вход выбора функции ALU.

После выборки двух байтов ADI нужны два шага исполнения.

Первый:

• Latch 2 Enable выводит 61h на шину данных;

• ALU Clock сохраняет результат и флаги.

Второй:

• ALU Enable выводит результат;

• Accumulator Clock сохраняет его.

Итак, на первом шаге защёлка второго байта помещает непосредственное значение 61h на шину данных, а ALU защёлкивает результат операции и новые значения флагов. Отдельный второй шаг необходим именно для переноса уже вычисленного результата: выход ALU включается на шину данных, после чего аккумулятор принимает находящийся там байт.

Следующая инструкция ADD B также требует двух шагов исполнения. Сначала RA Enable помещает содержимое регистра B на шину данных, а ALU Clock сохраняет результат сложения и флаги. Второй шаг полностью совпадает со вторым шагом ADI: ALU Enable выводит результат на шину, а Accumulator Clock сохраняет его в аккумуляторе.

STA требует шести шагов выборки, потому что два байта после опкода сохраняются в Latches 2 и 3. Исполнение:

• Latches 2 & 3 Enable выводит адрес на шестнадцатибитную шину;

• Accumulator Enable выводит Acc на шину данных;

• RAM Write записывает байт.

Таким образом, второй и третий байты STA вместе образуют адрес памяти. Они одновременно появляются на шестнадцатибитной адресной шине, тогда как аккумулятор помещает сохраняемый байт на восьмибитную шину данных. RAM Write связывает эти два значения, записывая байт именно по указанному адресу.

HLT уникально останавливает CPU; реализация будет позже.

Описанные шаги также называют *циклами*, подобно циклам стирки, полоскания и отжима в стиральной машине. Более точный термин - *машинные циклы*. В строящемся CPU каждый байт инструкции извлекается из памяти за один цикл, за которым всегда следует ещё один цикл увеличения счётчика команд. Поэтому для одно-, двух- и трёхбайтовых инструкций выборка занимает соответственно два, четыре и шесть машинных циклов.

Исполнение требует одного или двух циклов. Далее перечислены первые циклы всех введённых инструкций.

Инструкция	Шина адреса, первый цикл	Шина данных, первый цикл
MOV r,r	-	RA Enable; RA Clock
MOV r,M	HL Enable	RAM Data Out Enable; RA Clock
MOV M,r	HL Enable	RA Enable; RAM Write
MVI r,data	-	Latch 2 Enable; RA Clock
MVI M,data	HL Enable	Latch 2 Enable; RAM Write
LDA	Latches 2 & 3 Enable	RAM Data Out Enable; Acc Clock
STA	Latches 2 & 3 Enable	Acc Enable; RAM Write
ADD r...	-	RA Enable; ALU Clock
ADD M...	HL Enable	RAM Data Out Enable; ALU Clock
ADI data...	-	Latch 2 Enable; ALU Clock
INX/DCX HL	HL Enable	Inc-Dec Clock

Многоточие означает семейства. ADD включает ADC, SUB, SBB, ANA, XRA, ORA и CMP; ADI включает ACI, SUI, SBI, ANI, XRI, ORI и CPI.

Последние четыре строки требуют второго цикла исполнения.

Инструкция	Шина адреса, второй цикл	Шина данных, второй цикл
ADD r...	-	ALU Enable; Acc Clock
ADD M...	-	ALU Enable; Acc Clock
ADI data...	-	ALU Enable; Acc Clock
INX HL	Increment Enable; HL Select; HL Clock	-
DCX HL	Decrement Enable; HL Select; HL Clock	-

Чтобы выполнить все действия из двух таблиц, код операции надо декодировать и преобразовать в управляющие сигналы для компонентов CPU и RAM. Эти сигналы включают трёхстабильные буферы, управляют входами Clock различных защёлок, входом Write RAM и несколькими другими входами. Остаток главы показывает, как это делается: процесс требует нескольких этапов и двух разных стратегий.

Шаблоны кодов:

Инструкция	Код
MOV r,r	`01 DDD SSS`
MOV r,M	`01 DDD 110`
MOV M,r	`01 110 SSS`
HLT	`01110110`
MVI r,data	`00 DDD 110`
MVI M,data	`00110110`
ADD/ADC/SUB/SBB/ANA/XRA/ORA/CMP r	`10 FFF SSS`
те же с M	`10 FFF 110`
ADI/ACI/SUI/SBI/ANI/XRI/ORI/CPI data	`11 FFF 110`

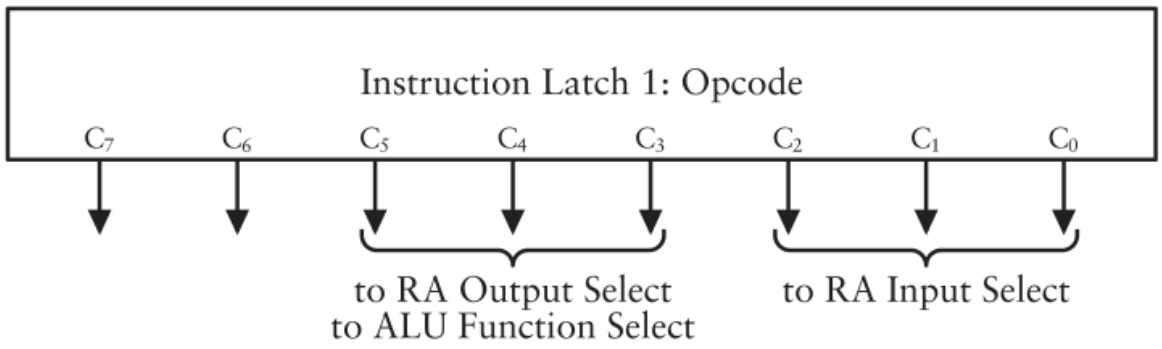
Инструкция	Код
INX HL	`00100011`
DCX HL	`00101011`
LDA addr	`00111010`
STA addr	`00110010`

SSS и DDD обозначают:

Код	Регистр
000	B
001	C
010	D
011	E
100	H
101	L
111	A

Комбинация 110 отсутствует в таблице регистров, потому что она означает память, адресуемую регистрами HL. В арифметических и логических инструкциях биты FFF означают функцию и выбирают одну из восьми арифметических или логических операций.

Простая часть управления - прямые соединения Latch 1 с Select RA и ALU:



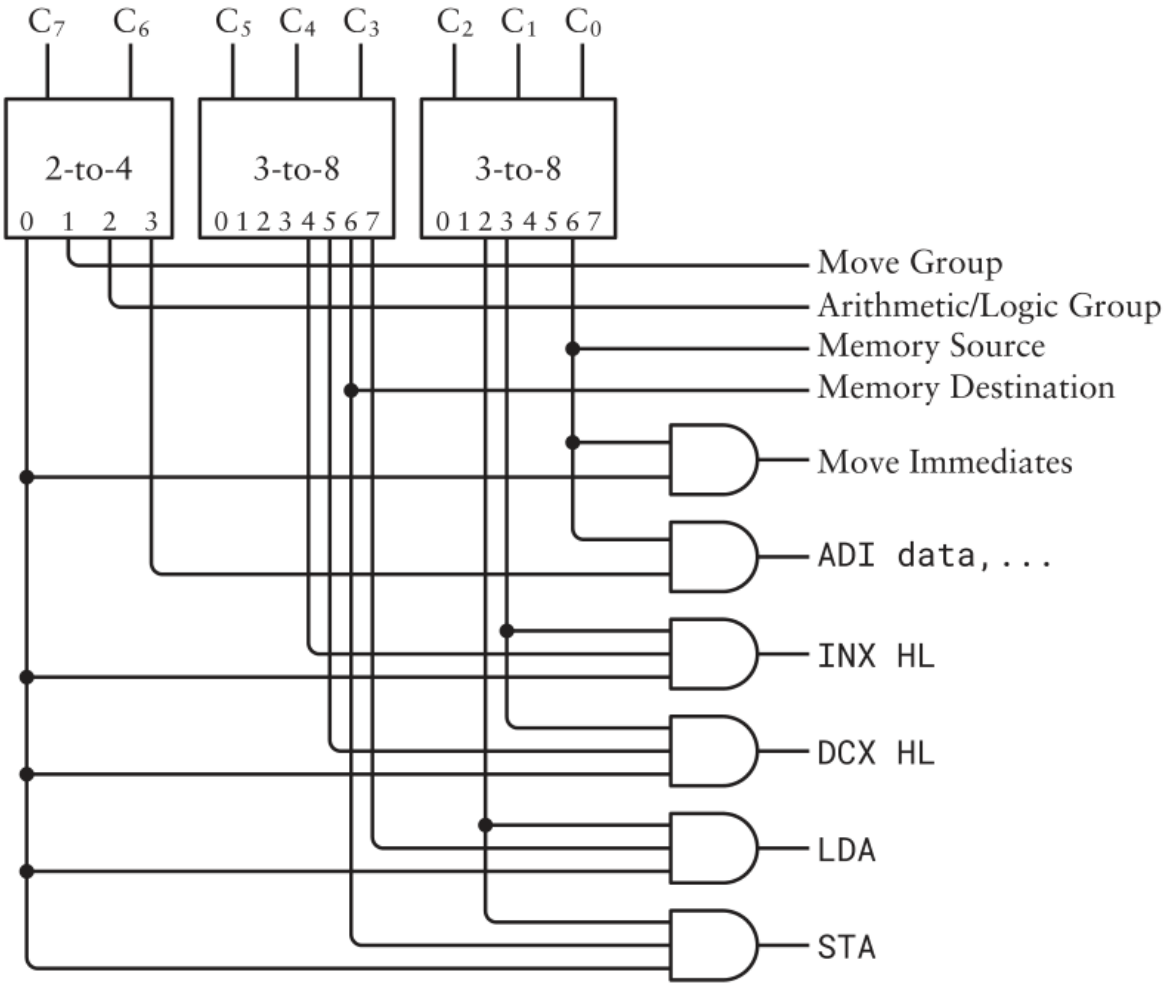
Буква C означает Code. Выходные биты C_0, C_1 и C_2 этой защёлки идут непосредственно на Input Select массива регистров, а C_3, C_4 и C_5 - одновременно на Output Select массива регистров и Function Select ALU. Это один из способов воспользоваться закономерностями построения кодов операций.

Можно заметить и другие закономерности. Все коды с началом 01 - инструкции MOV, кроме 76h, обозначающего HLT. Все регистровые арифметические и логические инструкции, в отличие от их непосредственных вариантов ADI, ACI и других, начинаются с 10. На первом этапе декодирования выходные биты Latch 1 подаются на три дешифратора: один дешифратор 2 в 4 и два дешифратора 3 в 8. Они создают дополнительные сигналы, часть которых соответствует отдельным инструкциям, а часть - целым группам инструкций.

Move Group соответствует началу 01, Arithmetic/Logic Group - 10.

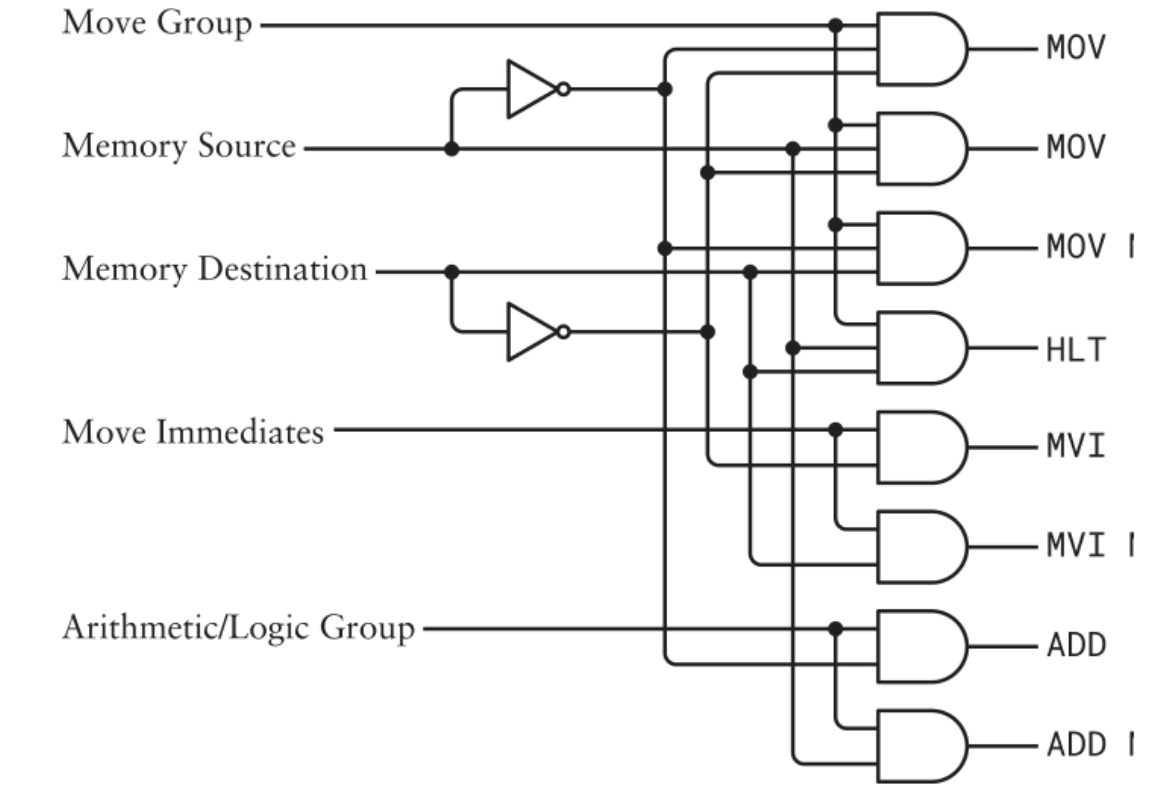
Важно отличать MOV, пересылающие байты между регистрами, от MOV, перемещающих значения между регистром и памятью. Инструкции с памятью распознаются по значению 110 в поле источника или получателя. Сигналы Memory Source и Memory Destination предыдущей схемы показывают соответственно, когда биты источника или получателя равны 110. Наконец, инструкции Move Immediate распознаются по тому, что начинаются с 00 и заканчиваются последовательностью 110.

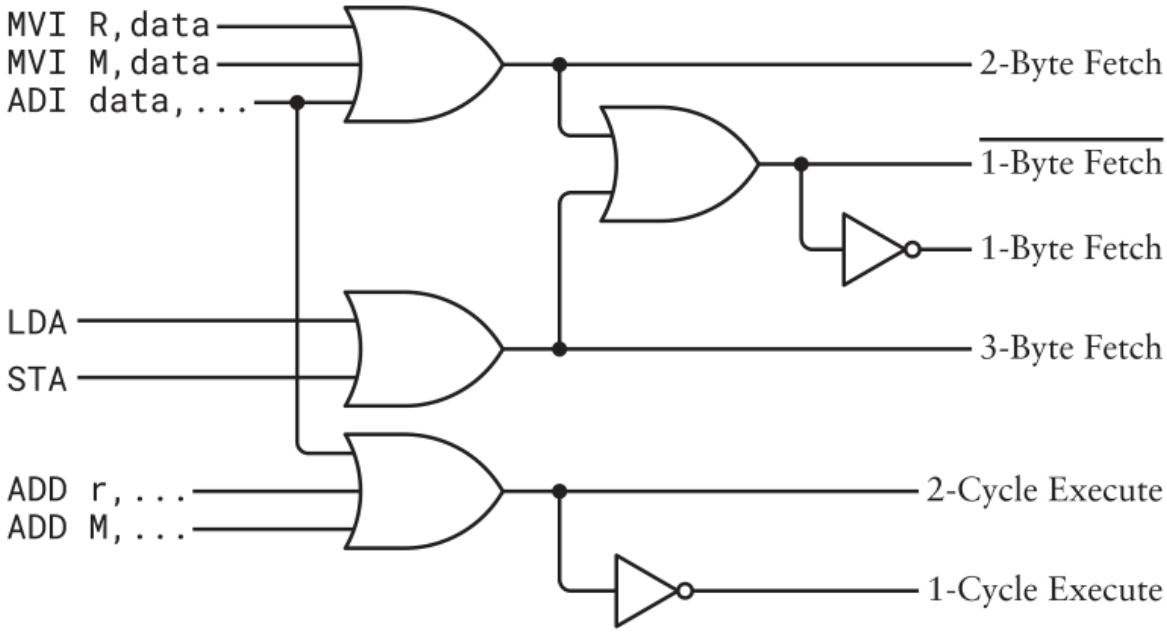
Пять верхних сигналов дополнительно декодируются:



Теперь каждая инструкция или группа сходных инструкций представлена отдельным сигналом. Эти сигналы становятся доступны, когда код операции сохранён в защёлке Instruction Latch 1, и их можно дальше использовать для управления обработкой инструкции.

Следующий шаг - по коду операции определить, сколько дополнительных байтов инструкции надо извлечь из памяти и сколько машинных циклов потребуется для её исполнения:



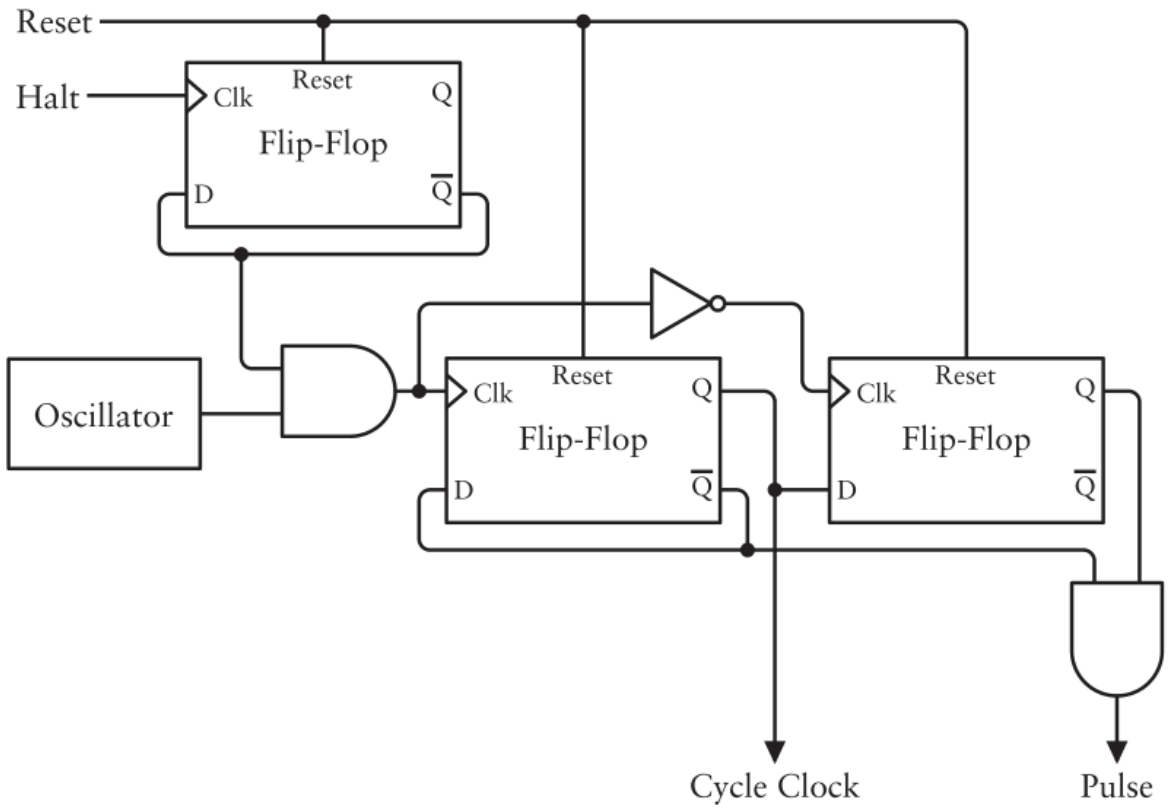


Логическая схема объединяет уже полученные сигналы отдельных инструкций и групп. На её выходах появляются признаки 1-, 2- или 3-байтовой выборки, а также одного или двух циклов исполнения. Так код операции начинает определять не только выполняемое действие, но и временную последовательность работы процессора.

Что произойдёт, если код операции не соответствует ни одной из этих инструкций? Например, до сих пор не упоминалась своеобразная инструкция 8080 NOP, произносимая как "no op", сокращение от "no operation", то есть "нет операции". Её код равен 00h.

Если ни один из входных сигналов слева на предыдущей схеме не равен 1, все выходы элементов ИЛИ равны 0. Тогда сигналы справа указывают на выборку одного байта и исполнение за один цикл. Именно так вписывается в эту схему NOP.

Основной ритм CPU задаёт усовершенствованный вариант небольшой схемы, впервые появившейся на странице 296 в главе 20:



Слева расположен генератор, выдающий сигнал, который, как правило очень быстро, чередуется между 0 и 1. Именно он заставляет CPU работать. Это своего рода сердцебиение процессора.

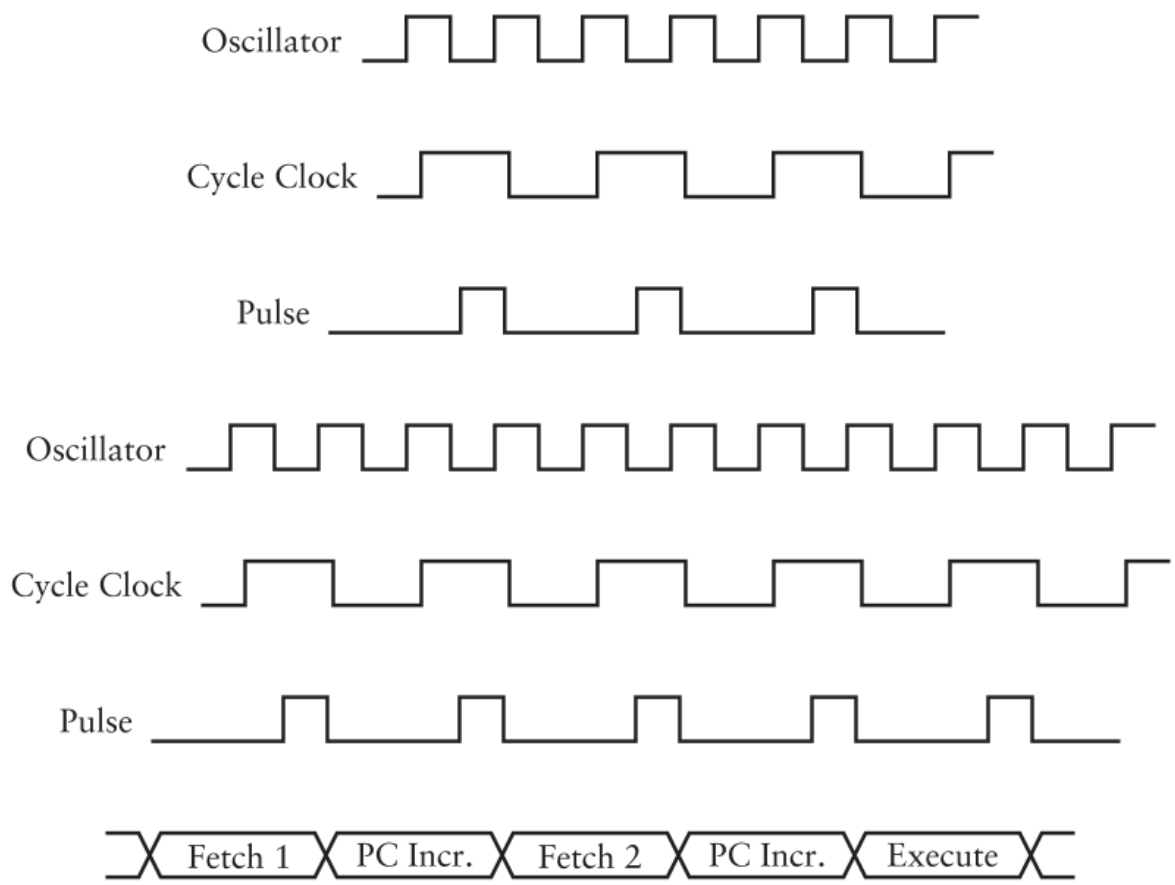
Показанный наверху сигнал Reset приходит извне CPU. Обычно им управляет человек, использующий компьютер, чтобы снова запустить CPU с самого начала. В нормальном состоянии Reset равен 0, но когда он становится равен 1 - например, при нажатии кнопки Reset, - CPU останавливается и всё возвращается в исходное состояние.

В этой схеме Reset сбрасывает три триггера: все выходы Q становятся равны 0, а все инверсные выходы Q - 1. Когда Reset возвращается к 0, триггеры снова могут работать нормально и CPU начинает исполнять инструкции.

После сброса CPU инверсный выход Q верхнего триггера равен 1. Он является одним из двух входов элемента И, который позволяет генератору управлять входами Clock двух нижних триггеров.

Верхний сигнал Halt показывает, что исполнена инструкция HLT. Она переводит инверсный выход Q верхнего триггера в 0 и тем самым фактически прекращает передачу импульсов генератора в CPU. Вернуть процессор из остановленного состояния можно сигналом Reset.

Если CPU не остановлен, два нижних триггера создают сигналы Cycle Clock и Pulse, показанные на временной диаграмме:



Каждый период Cycle Clock соответствует одному машинному циклу. Всякий раз, когда Cycle Clock переходит с низкого уровня на высокий, то есть с 0 на 1, начинается новый машинный цикл.

Например, первая инструкция приведённой ранее небольшой программы - Move Immediate, или MVI. Для неё требуются пять машинных циклов:

1. Выбрать из памяти код операции.
2. Увеличить счётчик команд.
3. Выбрать байт, следующий за кодом операции.
4. Снова увеличить счётчик команд.
5. Исполнить инструкцию.

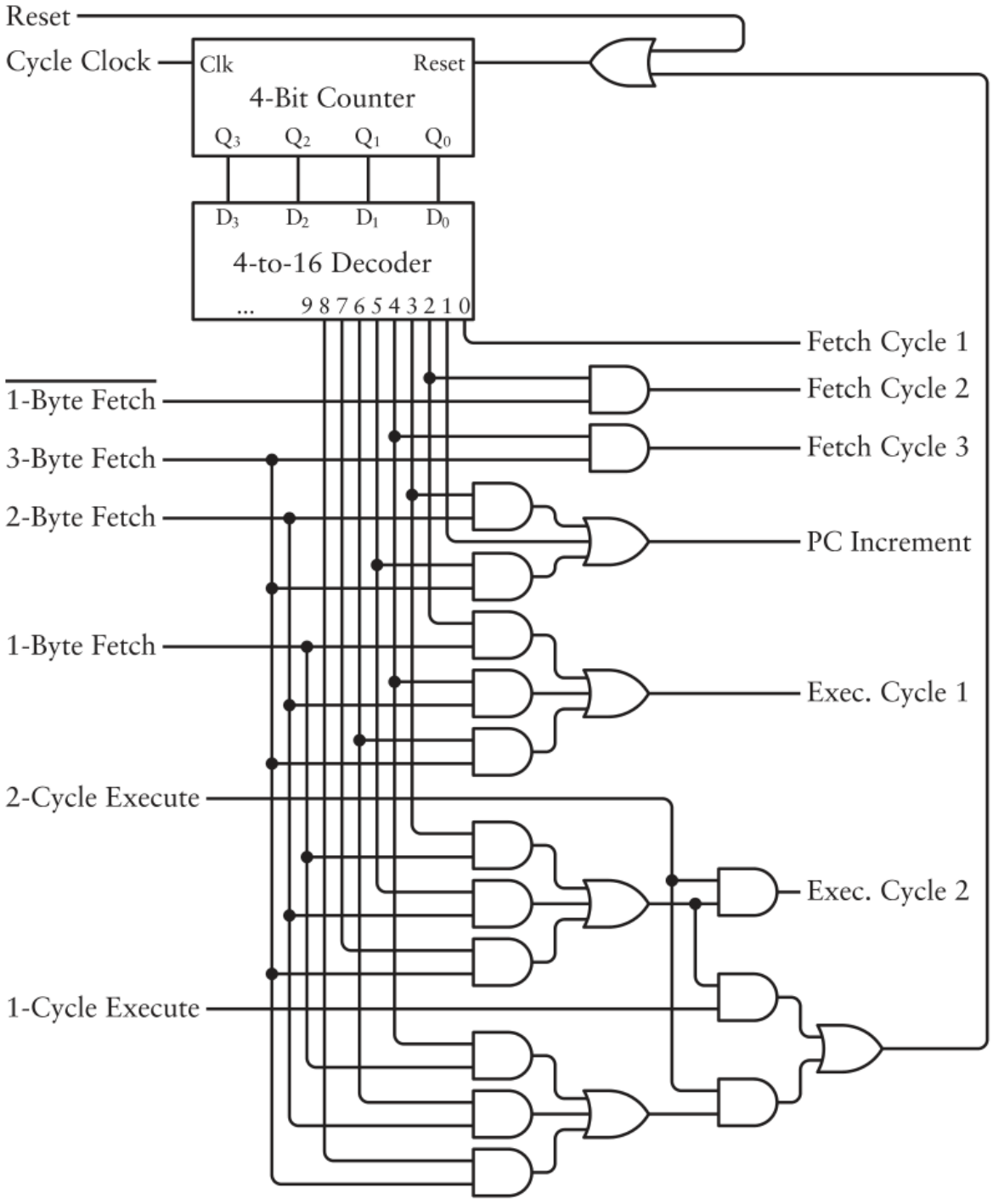
На диаграмме эти пять циклов обозначены сокращёнными названиями.

Каждому циклу соответствует включение различных трёхстабильных буферов, вследствие чего на шинах адреса и данных появляются разные значения. Например, во время Fetch 1 на шину адреса включён счётчик команд, а на шину данных - выход RAM Data Out. Сигнал Pulse управляет входом Clock защёлки Instruction Byte 1 и входом Clock инкрементора-декрементора.

Во время цикла увеличения PC выход инкрементора-декрементора находится на шине адреса, а Pulse сохраняет это увеличенное значение в счётчике команд.

Ранее была показана схема, определяющая, состоит ли инструкция из одного, двух или трёх байтов и требует ли она одного или двух циклов исполнения.

Теперь из кода операции нужно получить сигналы, указывающие тип текущего цикла: первая, вторая или третья выборка, увеличение PC либо первый или второй цикл исполнения. Эту работу выполняет следующая довольно сложная схема:



Входы расположены слева, выходы - справа. Некоторые входы и выходы имеют похожие имена, поэтому сначала схема может показаться несколько запутанной. Например, вход 2-Byte Fetch сообщает, что длина инструкции составляет два байта. Выход Fetch Cycle 2 сообщает совсем другое: в данный момент выбирается второй байт инструкции.

Верхний сигнал Reset - тот же сигнал, что и на предыдущей схеме. Его включает человек, чтобы запустить CPU с начала. Кроме того, 4-разрядный счётчик может быть сброшен сигналом, сформированным в нижней части самой схемы.

Cycle Clock увеличивает счётчик. Поскольку счётчик 4-разрядный, он перебирает двоичные числа от 0000 до 1111, то есть десятичные числа от 0 до 15. Выходы счётчика идут прямо на расположенный ниже дешифратор 4 в 16. Двоичное число могло бы быть преобразовано в 16 последовательных выходов, но здесь используются только первые девять. Каждый выход обозначает новый машинный цикл: цикл выборки, цикл увеличения счётчика команд - на схеме он сокращён до PC Increment - либо цикл исполнения.

При продвижении выходов дешифратора от 0 к 1, 2, 3 и далее формируются такие сигналы:

0. Fetch Cycle 1.
1. Program Counter Increment.
2. Fetch Cycle 2, но только если инструкция не является однобайтовой.
3. Program Counter Increment для двух- или трёхбайтовой инструкции.
4. Fetch Cycle 3, но только если сигнал 3-Byte Fetch равен 1.
5. Program Counter Increment для трёхбайтовой инструкции.

Fetch Cycle 1 и первый Program Counter Increment создаются всегда. После этого код операции уже выбран, загружен в Instruction Latch 1 и частично декодирован, поэтому доступны все входные сигналы в левой части схемы.

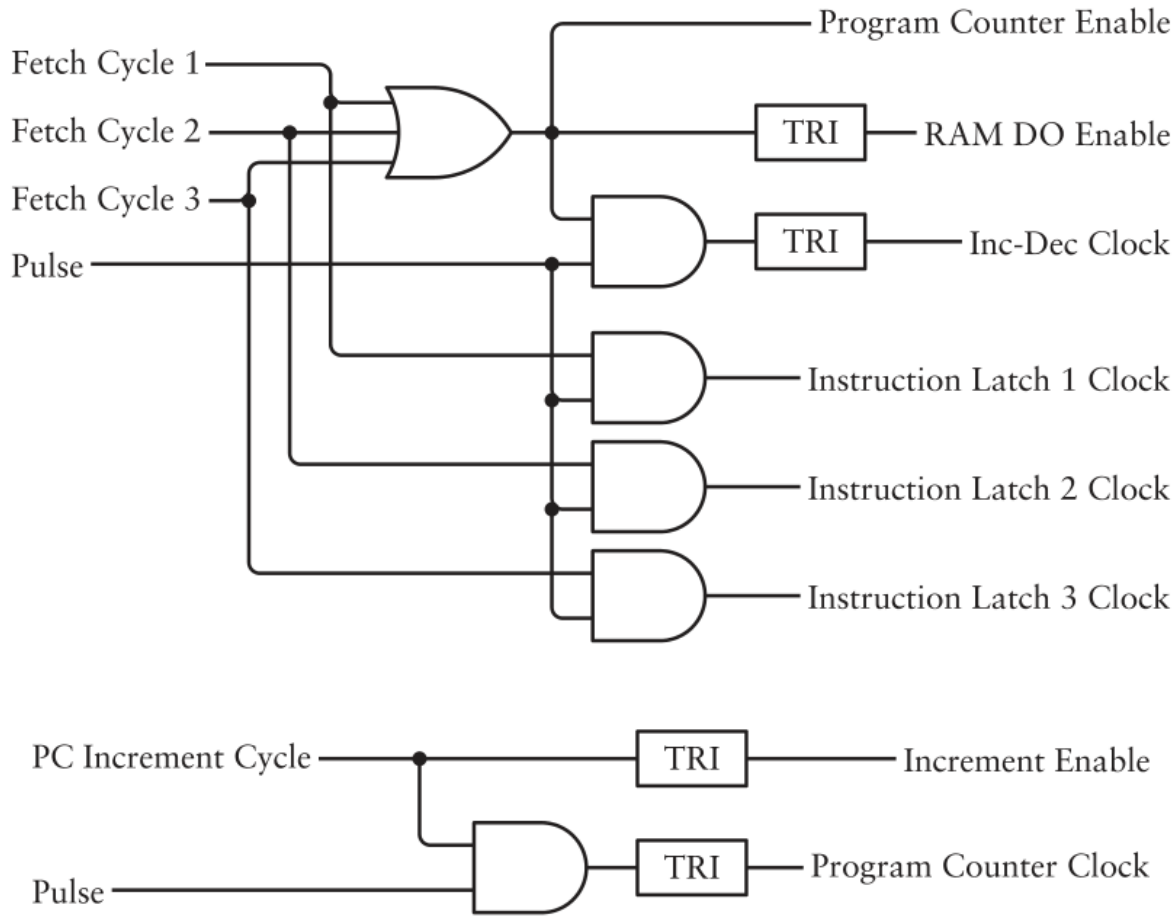
Для одной инструкции требуется не более трёх циклов выборки, за каждым из которых следует цикл увеличения PC, и не более двух циклов исполнения. Итого максимум восемь циклов, соответствующих выходам дешифратора с 0 по 7.

Запутанная логика нужна, чтобы учесть все сочетания многобайтовых выборок и нескольких циклов исполнения. Например, выходной сигнал Execution Cycle 1 может соответствовать третьему циклу для однобайтовой инструкции, пятому - для двухбайтовой и седьмому - для трёхбайтовой.

Сложнее всего логика сброса внизу. Сброс счётчика может произойти уже в четвёртом цикле для инструкции с одной выборкой и одним циклом исполнения или только в девятом цикле для инструкции с тремя выборками и двумя циклами исполнения.

Во время каждого из трёх циклов выборки счётчик команд включён на 16-разрядную шину, а RAM Data Out - на 8-разрядную. Pulse сохраняет значение с шины адреса в защёлке инкрементора-декрементора, а значение с шины данных - в одной из трёх защёлок инструкции.

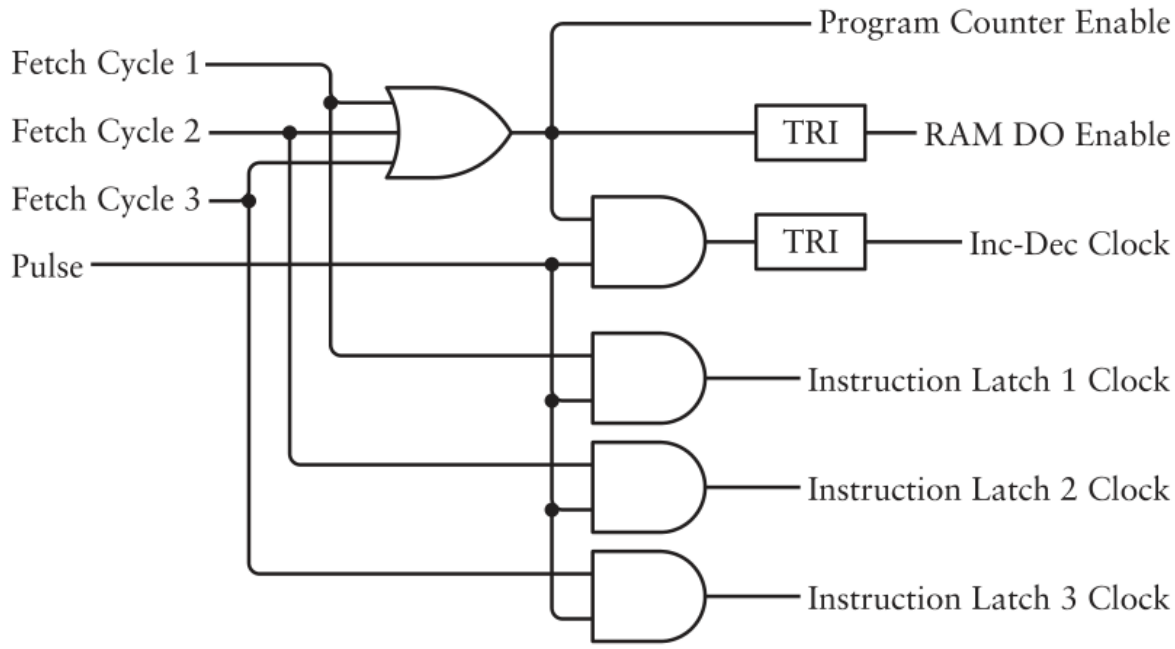
Именно для этого предназначена следующая схема. Она создаёт все сигналы циклов выборки инструкции, но не циклов увеличения PC:

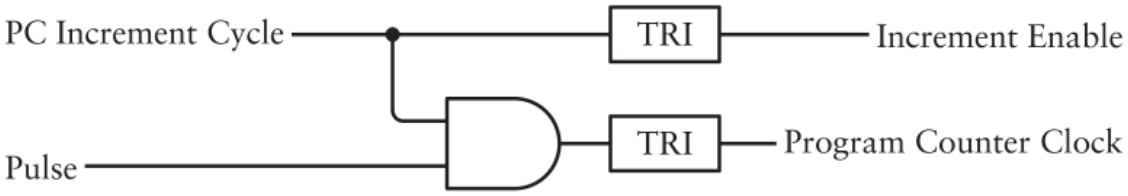


Независимо от того, первая это выборка, вторая или третья, счётчик команд включается на шину адреса, а RAM Data Out - на шину данных. Во всех трёх случаях Pulse всегда управляет входом Clock защёлки инкрементора-декрементора. В каждом цикле выборки Pulse также управляет входом Clock соответствующей защёлки инструкции.

Обратите внимание на трёхстабильные буферы в двух цепях. Они нужны потому, что другие схемы, которые появятся далее, тоже могут управлять сигналом Enable трёхстабильного буфера RAM Data Out и сигналом Clock защёлки инкрементора-декрементора. Сигнал слева от каждого такого буфера служит одновременно его входом и сигналом Enable.

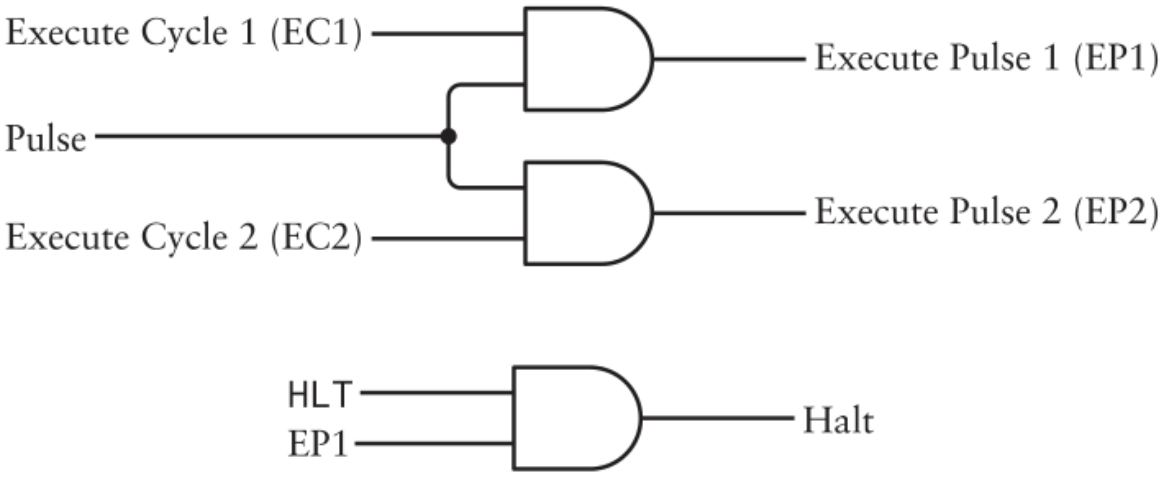
Сигналы, необходимые для цикла увеличения PC, полностью формируются отдельной схемой:





Итак, все сигналы циклов выборки инструкции и циклов увеличения PC уже созданы. Остались сигналы циклов исполнения. Они сложнее, поскольку зависят от конкретной исполняемой инструкции.

У большой схемы на странице 370 есть два выходных сигнала Exec. Cycle 1 и Exec. Cycle 2. Эти два цикла исполнения можно сокращённо назвать EC1 и EC2:

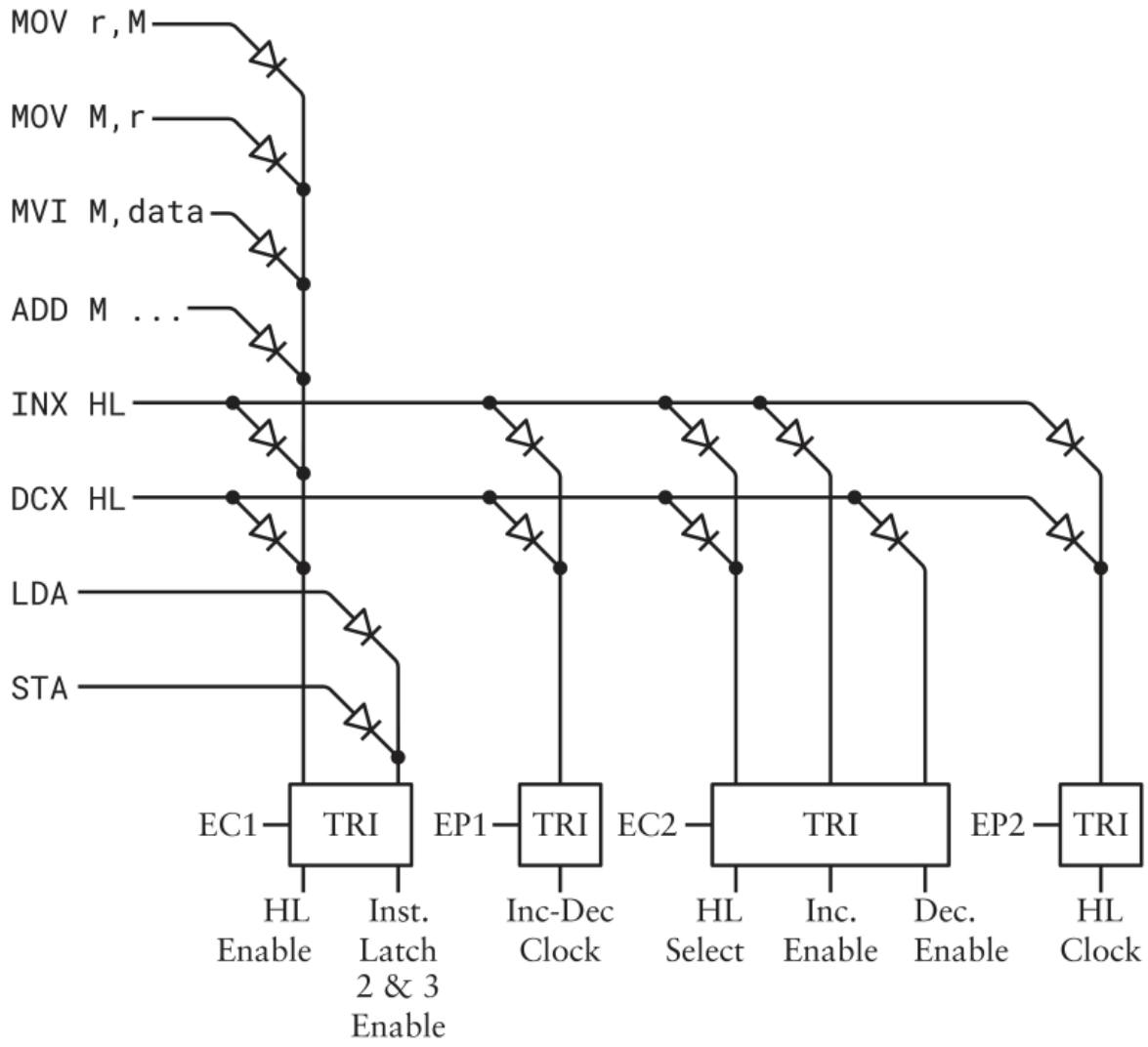


Каждый из них объединяется с Pulse, образуя два дополнительных сигнала Execute Pulse, сокращённо EP1 и EP2.

Одна инструкция обрабатывается совсем просто. Это HLT, останавливающая CPU. Сигнал HLT слева приходит с дешифратора инструкций на странице 367, а выходной Halt справа направляется к схеме с генератором на странице 368.

Связь остальных инструкций с управляющими сигналами, которые для них надо сформировать, довольно сложна. Чтобы не строить громоздкую систему логических вентилей, лучше воспользоваться несколькими диодными матрицами ROM, подобными тем, которые встречались в главе 18.

Первая диодная матрица ROM формирует все сигналы Enable и Clock, относящиеся к 16-разрядной шине адреса, как для первого, так и для второго цикла исполнения:



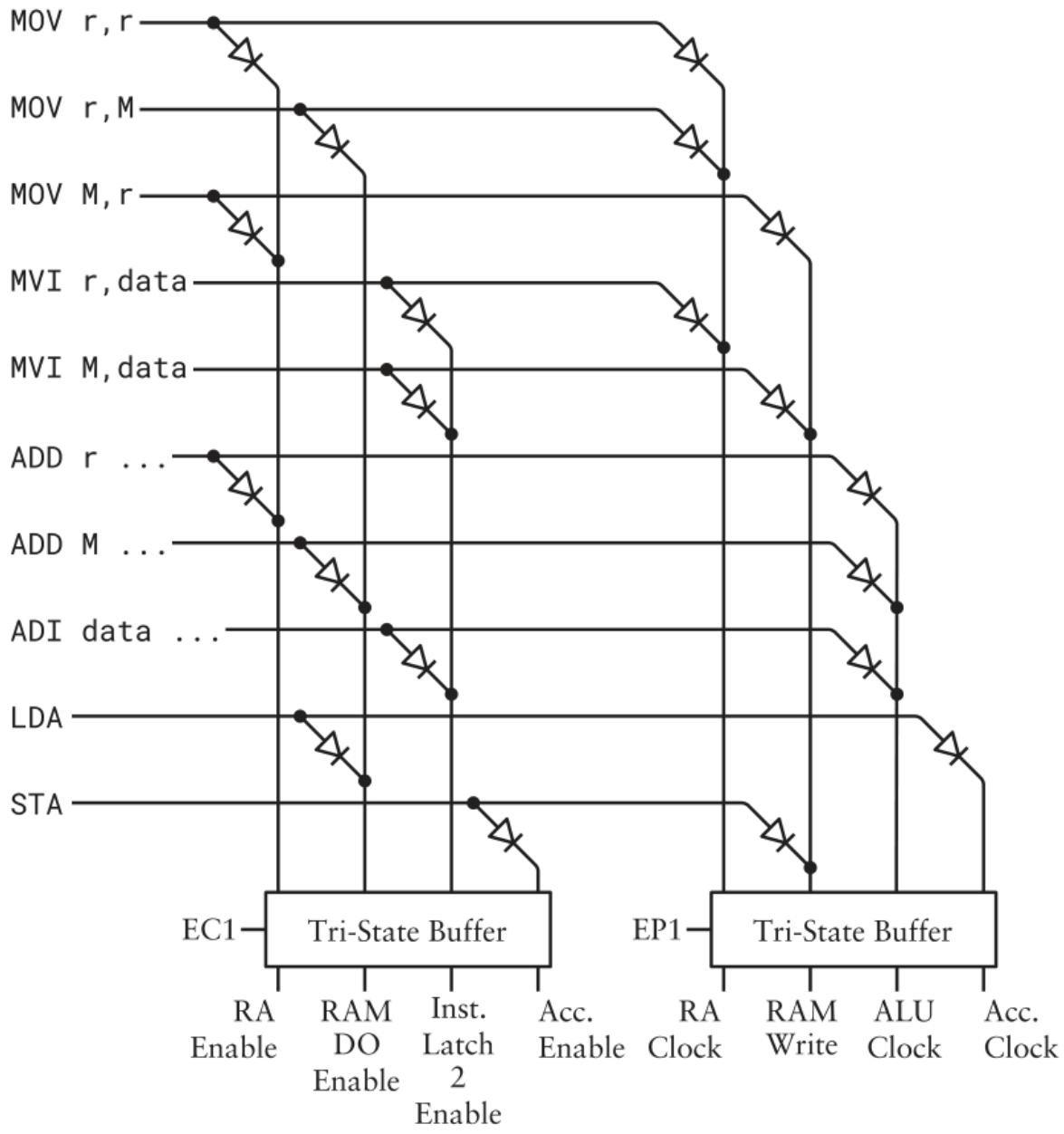
Помните, что нижние выходы относятся только к шине адреса. Сигналы 8-разрядной шины данных появятся чуть позже. Эта схема соответствует столбцу "16-Bit Address Bus" таблиц на страницах 363 и 364.

Левый нижний трёхстабильный буфер включается сигналом Execution Cycle 1 и определяет, какое значение находится на шине адреса во время первого цикла исполнения. Для всех MOV, MVI и арифметических инструкций, обращающихся к памяти по адресу из регистров HL, на шину включается именно HL. Кроме того, HL включается для инструкций INX и DCX, увеличивающих и уменьшающих этот регистр. Но для LDA и STA адрес памяти, по которому загружается или сохраняется байт, поступает из Instruction Latches 2 и 3.

Для INX и DCX сигнал Execution Pulse 1 сохраняет значение HL в защёлке инкрементора-декрементора.

INX и DCX - единственные две инструкции, использующие шину адреса во втором цикле исполнения. Они помещают на неё увеличенное или уменьшенное значение HL. Затем Execution Pulse 2 сохраняет новое значение HL в регистрах H и L.

Диодная ROM-матрица 8-разрядной шины данных несколько сложнее. Она разделена на две схемы, соответствующие двум циклам исполнения. Вот схема первого цикла:



Она реализует столбец "8-Bit Data Bus" таблицы на странице 363. Два нижних трёхстабильных буфера включаются сигналами Execution Cycle 1 и Execution Pulse 1. Первый буфер определяет, что находится на шине данных, а второй - куда это значение сохраняется.

Три типа инструкций MOV в верхней части схемы содержат получателя и источник. И тем и другим может быть любой регистр либо память по адресу из HL. Если источником служит регистр, на шину данных включается массив регистров, обозначенный RA. Если источник - память, включается RAM Data Out. При этом адрес RAM задаётся через 16-разрядную шину, а матрица ROM адресной шины устанавливает на ней значение HL.

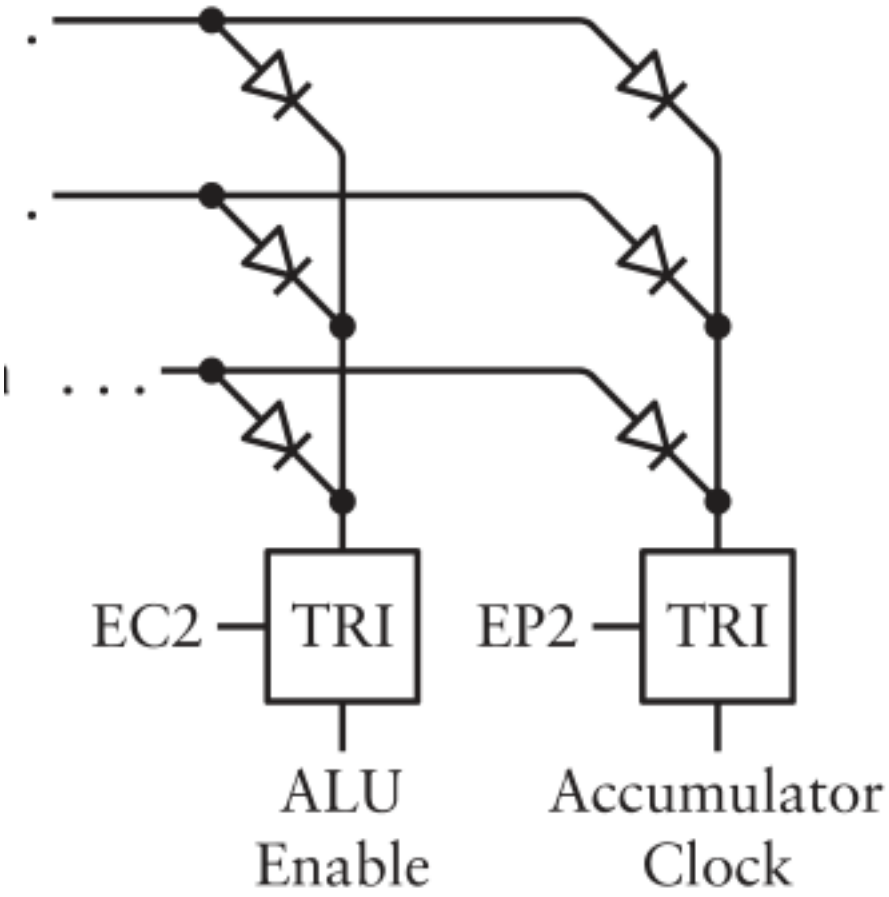
Если получатель - регистр, второй трёхстабильный буфер управляет входом Clock массива регистров. Если получатель - память, RAM Write сохраняет значение в памяти.

Для двух разновидностей MVI, то есть Move Immediate, на шину данных включается содержимое Instruction Latch 2. Затем это значение либо сохраняется в массиве регистров, либо записывается в память.

Все арифметические и логические инструкции представлены на схеме инструкциями ADD и ADI, то есть Add Immediate. В зависимости от конкретной инструкции на шину данных включается массив регистров, RAM Data Out или Instruction Latch 2. В любом случае значение защёлкивается в арифметико-логическом устройстве. Для завершения этим инструкциям нужен второй цикл исполнения.

Для LDA, то есть Load Accumulator, и STA, то есть Store Accumulator, ROM-матрица адресной шины обеспечивает адресацию RAM содержимым Instruction Latches 2 и 3. При LDA на шину данных включается RAM Data Out, а значение сохраняется в аккумуляторе. При STA на шину включается аккумулятор, а его значение сохраняется в памяти.

Во втором цикле исполнения арифметических и логических инструкций снова используется шина данных. Соответствующая диодная ROM-матрица гораздо проще:



Для этих инструкций на шину данных включается выход ALU, после чего результат сохраняется в аккумуляторе. Именно это было указано в столбце "8-Bit Data Bus" таблицы на странице 364.

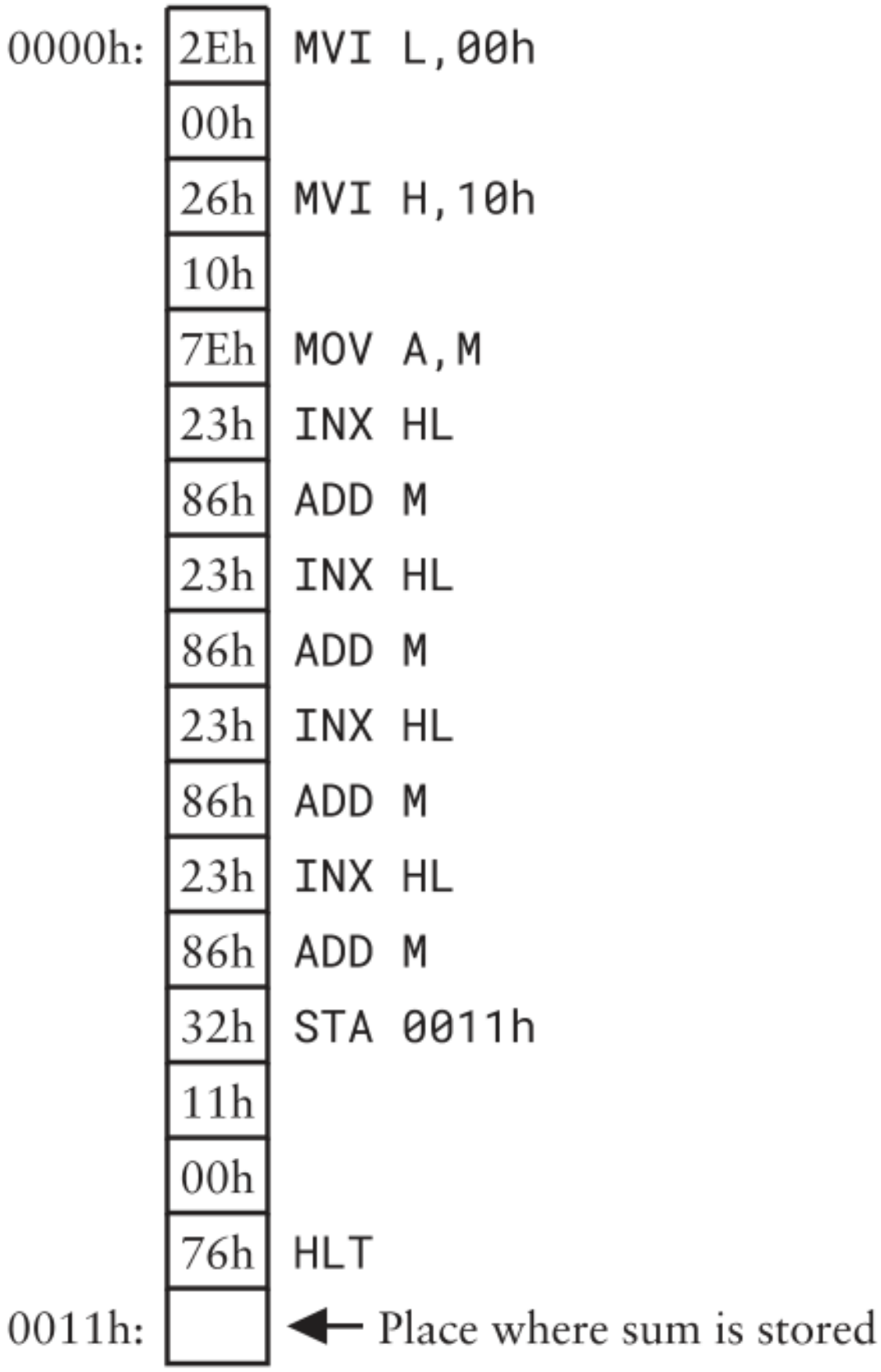
На этом построение подмножества микропроцессора 8080, продолжавшееся три главы, завершено. Рабочая имитация доступна на сайте CodeHiddenLanguage.com.

Инженеры, проектирующие компьютеры, часто тратят много времени на их ускорение. Одни варианты цифровых логических схем работают быстрее других. Очень часто увеличение скорости требует добавления логических вентилей.

Если бы автор хотел ускорить описанный CPU, прежде всего он занялся бы выборкой инструкций. Для каждой выборки сейчас требуется отдельный машинный цикл, единственная цель которого - увеличить счётчик команд. Стоило бы включить эту логику в сам цикл выборки и выполнять обе операции одновременно. Вероятно, для этого понадобился бы отдельный инкрементор. Такое усовершенствование вдвое сократило бы время загрузки инструкций из памяти.

Даже небольшие изменения приносят огромную пользу. Если CPU предназначен для миллионов компьютеров, каждый из которых потенциально исполняет миллионы инструкций в секунду, устранение лишних машинных циклов даёт громадный выигрыш каждому пользователю.

Рассмотрим простую программу для этого CPU. Допустим, в памяти, начиная с адреса 1000h, хранятся пять байтов и требуется сложить их:



Первые две инструкции задают значения регистров H и L. Затем программа через HL обращается к байтам, накапливает сумму и после каждого обращения увеличивает HL.

В программе заметно повторение: за INX четыре раза следует ADD. Для этого примера оно не слишком обременительно. Но что, если требуется сложить 20 значений? Или сто?

А если складывать надо не байты, а 16- или 32-разрядные значения, для накопления суммы которых требуется больше инструкций?

Можно ли избежать такого повторения? Может ли существовать инструкция, которая заставляет повторяться другие последовательности инструкций? Как такая инструкция должна выглядеть? И как она будет работать?

Эта тема настолько важна, что ей посвящена целая глава.

Глава 24. Циклы, переходы и вызовы

Наша жизнь полна повторений. Мы отсчитываем дни по естественным ритмам вращения Земли, обращения Луны вокруг Земли и Земли вокруг Солнца. Каждый день отличается от других, но наша жизнь часто строится вокруг стандартных дел, которые изо дня в день остаются похожими.

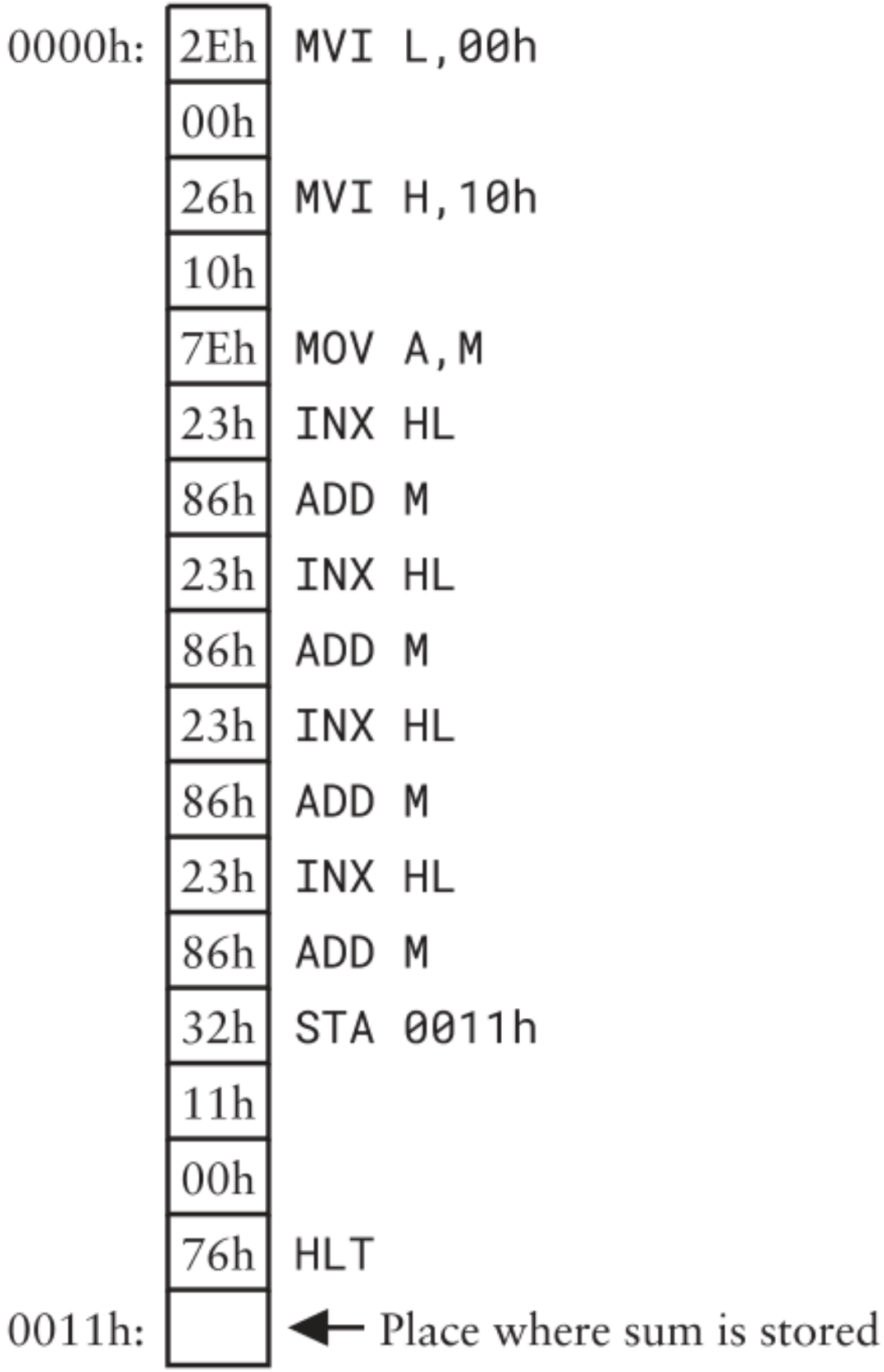
В некотором смысле повторение составляет и самую суть вычислений. Никому не нужен компьютер, чтобы сложить два числа. Во всяком случае, будем на это надеяться! Но сложить тысячу или миллион чисел - вот работа для компьютера.

Связь вычислений с повторением была очевидна уже очень давно. В знаменитом обсуждении Аналитической машины Чарльза Бэббиджа, написанном в 1843 году, Ада Лавлейс отмечала:

> Ради краткости и ясности повторяющаяся группа называется циклом. Под циклом операций следует понимать любой набор операций, повторяемый более одного раза. Он в равной мере является циклом независимо от того, повторяется ли всего дважды или неопределённое число раз: циклом его делает сам факт повторения. Во многих аналитических задачах встречается повторяющаяся группа из одного или нескольких циклов, то есть цикл цикла или цикл циклов.

В современной терминологии такие циклы часто называют *loops*. То, что Лавлейс называла циклом цикла, теперь называется *вложенным циклом*.

CPU, который строился на протяжении нескольких последних глав, в этом отношении кажется неполным. В конце предыдущей главы была показана небольшая программа, складывающая пять байтов, которые хранятся в памяти начиная с адреса 1000h.



Для адресации памяти используется пара регистров HL. Первая команда MOV читает первый байт из памяти в аккумулятор, а последующие инструкции ADD прибавляют к аккумулятору остальные четыре байта. После чтения каждого байта значение пары HL увеличивается инструкцией INX. Наконец, STA сохраняет результат в памяти.

Как расширить эту программу, чтобы она складывала сто или тысячу байтов? Просто продолжать добавлять пары инструкций INX и ADD, пока их количество не совпадёт с числом слагаемых? Такое решение кажется неправильным. Оно плохо приспосабливается к другим потребностям и не является *обобщённым* решением.

Гораздо удобнее была бы новая инструкция, позволяющая повторять определённую последовательность инструкций - в данном случае INX и ADD. Но как она должна выглядеть?

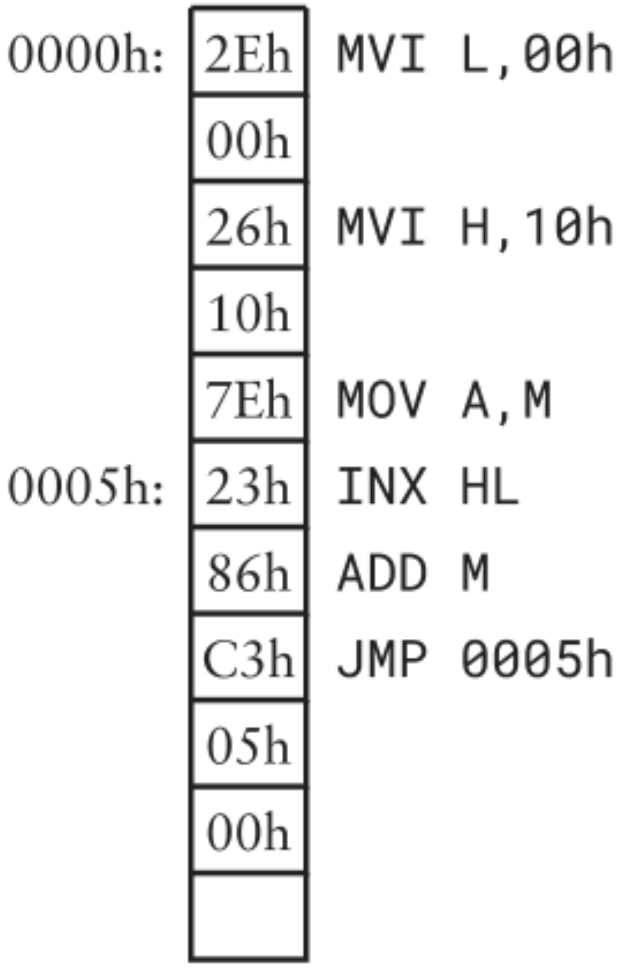
Сначала может показаться, что подобная инструкция настолько отличается от уже существующих, что ради неё придётся полностью переделать CPU. Однако отчаиваться пока рано.

В нормальном режиме после выборки каждого байта инструкции счётчик команд увеличивается. Именно поэтому CPU переходит от одной инструкции к следующей. Инструкция, создающая цикл, тоже должна как-то изменять счётчик команд, но уже иным способом.

Ранее встречались LDA и STA, после которых следуют два байта, вместе образующие шестнадцатибитный адрес памяти. Представим похожую инструкцию с двумя следующими за ней байтами, но эти байты не используются для адресации памяти. Вместо этого они защёлкиваются в счётчике команд. Такая инструкция изменяет обычный порядок исполнения, поскольку заставляет счётчик команд фактически *перепрыгнуть* на другой адрес.

Назовём её JMP, от слова jump - "переход". Именно так она называется в микропроцессоре Intel 8080. В Motorola 6809 сходная инструкция называлась BRA, от branch - "ветвление".

После JMP следуют два байта, образующие шестнадцатибитный адрес. В следующем примере этот адрес равен 0005h:



Каждый раз после исполнения INX и ADD инструкция JMP продолжает исполнение с адреса 0005h, то есть снова с INX и ADD. Это и есть цикл.

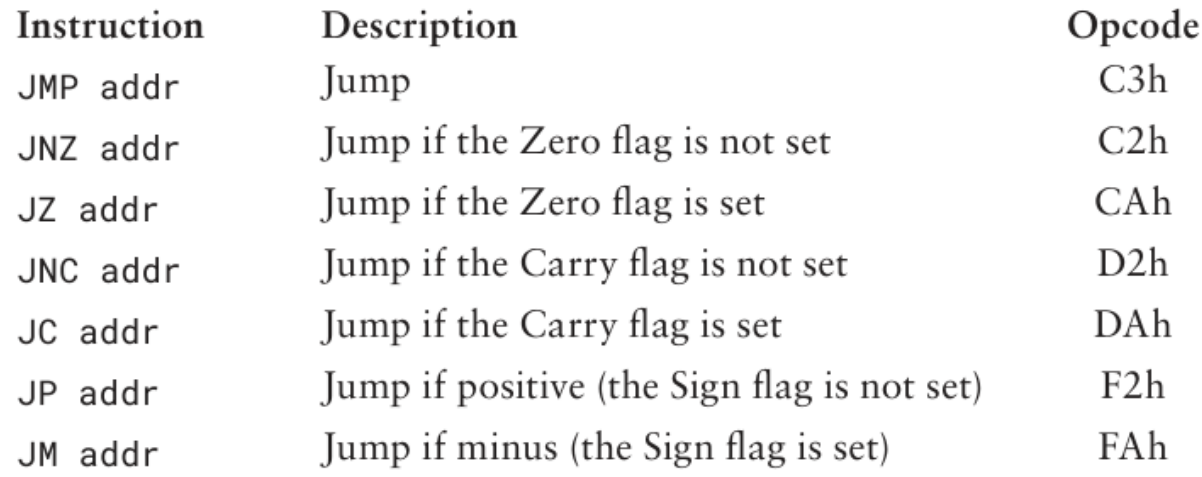
Добавить JMP в CPU на удивление легко. Но прежде стоит признать проблему: программа с JMP будет работать вечно. Остановить цикл невозможно, поэтому он называется *бесконечным циклом*. HL продолжает увеличиваться, а байт по очередному адресу продолжает прибавляться к сумме в аккумуляторе. В конце концов HL достигает FFFFh, последнего адреса памяти. После следующего увеличения он переполняется и превращается в 0000h, а программа начинает прибавлять к аккумулятору байты собственных инструкций.

Циклы чрезвычайно важны в программировании, но не менее важно выполнять цикл *иногда, а не всегда*.

Есть ли в CPU что-либо, способное решить, должен ли произойти переход? Да. Напомним, что ALU из главы 21 сохраняет в защёлке несколько флагов. Carry показывает, возник ли перенос; Zero - равен ли результат нулю; Sign - равен ли старший бит результата 1, что для числа в дополнительном коде означает отрицательное значение.

Можно представить инструкцию, выполняющую переход только при установленном Zero, или, наоборот, только когда Zero не установлен. На самом деле можно определить целое небольшое семейство:

Инструкция	Описание	Код
JMP addr	Перейти	C3h
JNZ addr	Перейти, если Zero не установлен	C2h
JZ addr	Перейти, если Zero установлен	CAh
JNC addr	Перейти, если Carry не установлен	D2h
JC addr	Перейти, если Carry установлен	DAh
JP addr	Перейти при положительном результате, когда Sign не установлен	F2h
JM addr	Перейти при отрицательном результате, когда Sign установлен	FAh



Instruction	Description	Opcode
JMP addr	Jump	C3h
JNZ addr	Jump if the Zero flag is not set	C2h
JZ addr	Jump if the Zero flag is set	CAh
JNC addr	Jump if the Carry flag is not set	D2h
JC addr	Jump if the Carry flag is set	DAh
JP addr	Jump if positive (the Sign flag is not set)	F2h
JM addr	Jump if minus (the Sign flag is set)	FAh

Эти инструкции и коды операций не выдуманы: они реализованы в Intel 8080, служащем образцом для создаваемого подмножества CPU. Обозначение addr в первом столбце означает двухбайтовый адрес памяти, следующий за кодом операции.

JMP называется *безусловным переходом*. Он заставляет CPU изменить обычный ход исполнения независимо от значений флагов ALU. Остальные инструкции - *условные переходы*: они изменяют счётчик команд, только когда определённые флаги ALU установлены или сброшены. В 8080 есть ещё два условных перехода по флагу Parity, упомянутому в главе 21, но в нашем CPU этот флаг не реализован.

Посмотрим, как условный переход работает в программе, складывающей 200 байтов из памяти, начиная с адреса 1000h.

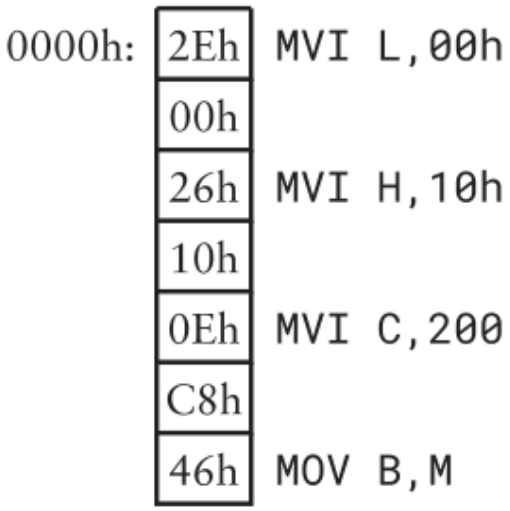
Хитрость состоит в том, чтобы хранить в одном из регистров значение, называемое *счётчиком*. Сначала оно равно 200 - количеству складываемых байтов. После обращения к каждому байту и его прибавления счётчик уменьшается. В любой момент он показывает, сколько байтов осталось обработать. Когда он достигает нуля, работа окончена.

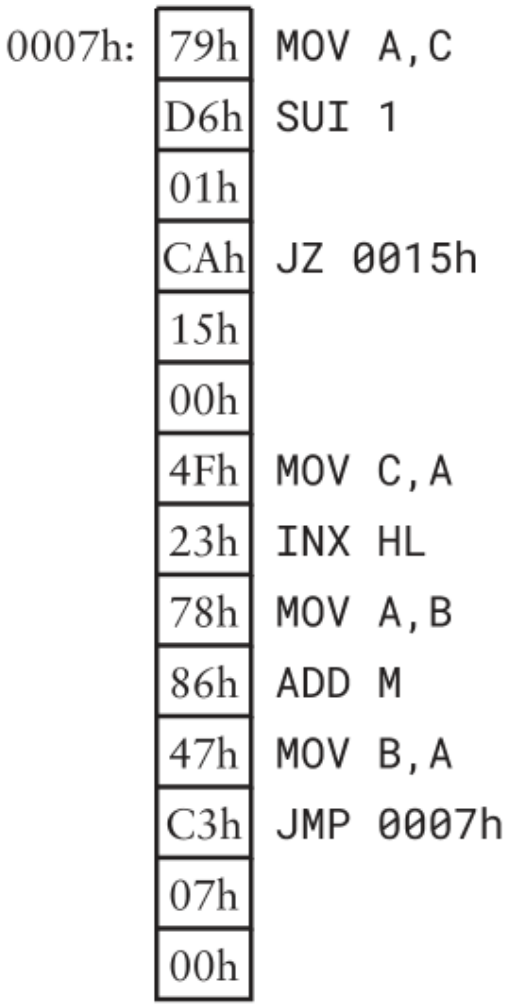
Программе приходится жонглировать двумя арифметическими операциями: поддерживать текущую сумму байтов и уменьшать счётчик после прибавления каждого нового байта.

Здесь возникает небольшая проблема. Как известно, все арифметические и логические операции используют аккумулятор. Поэтому программе приходится переносить байты из регистров в аккумулятор, выполнять арифметику, а затем возвращать новые значения в регистры.

Решим хранить текущую сумму в регистре B, а счётчик - в C. Для любой арифметической операции эти значения переносятся в аккумулятор, а перед следующей итерацией возвращаются соответственно в B или C.

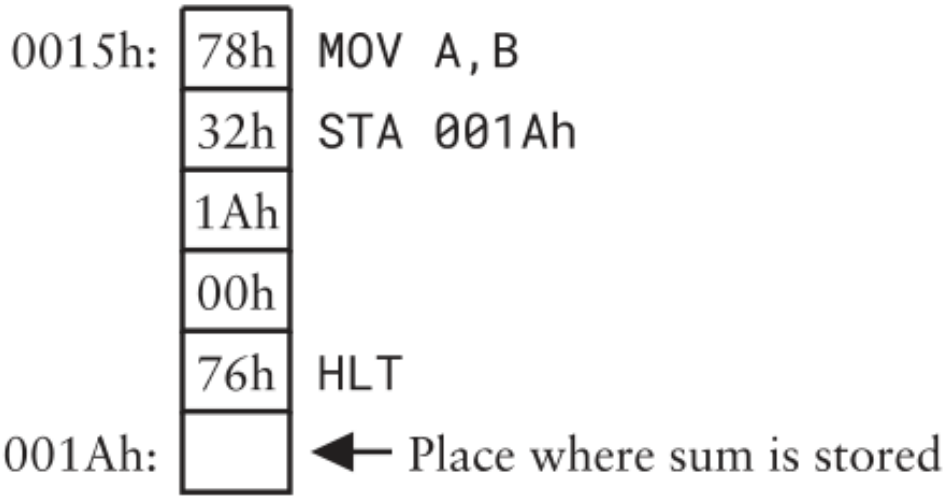
Поскольку программа длиннее предыдущих примеров, она разделена на три части. Первая часть компьютерной программы обычно называется *инициализацией*:

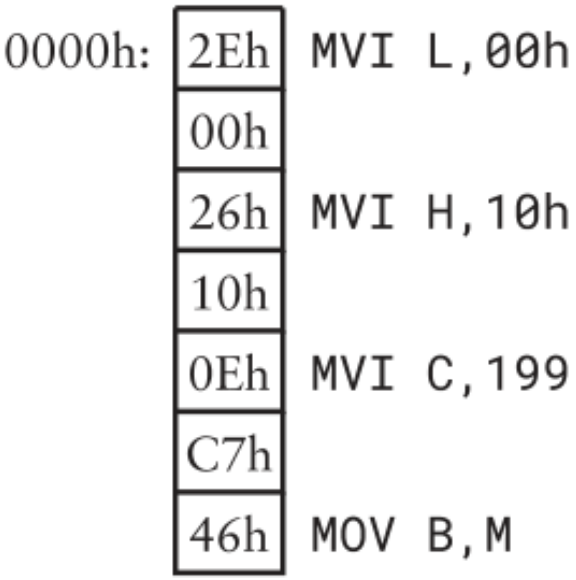




Здесь составное шестнадцатибитное значение пары HL устанавливается равным 1000h - адресу первого складываемого числа. Регистр C получает десятичное 200, то есть C8h, - число значений, которые надо сложить. Наконец, в B помещается первое число из списка.

Во второй части программы находятся повторяемые инструкции.





Сначала значение счётчика копируется в аккумулятор. SUI вычитает из него 1. При первом прохождении 200 превращается в 199. Если результат равен нулю - чего при первом прохождении, конечно, ещё не происходит, - JZ переходит по адресу 0015h, следующему сразу после этого блока. Такой выход называется *выходом из цикла*.

В противном случае значение аккумулятора, равное 199 при первом прохождении, возвращается в C. Затем HL увеличивается инструкцией INX. Текущая сумма из B переносится в A, к ней прибавляется значение из памяти по адресу HL, а новая сумма копируется обратно в B. После этого безусловная JMP возвращает исполнение к началу цикла.

Каждое прохождение этого участка кода обычно называется *итерацией*. В конце концов в C остаётся 1. После вычитания единицы получается ноль, и JZ переходит по адресу 0015h.

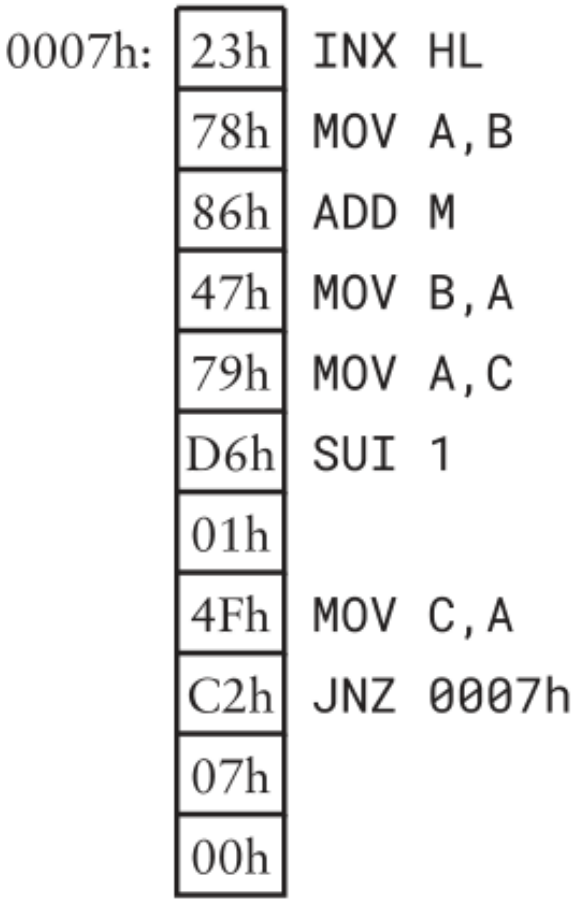
В B теперь находится окончательная сумма всех 200 чисел. Она переносится в аккумулятор для STA, которая сохраняет значение в памяти. Затем программа останавливается.

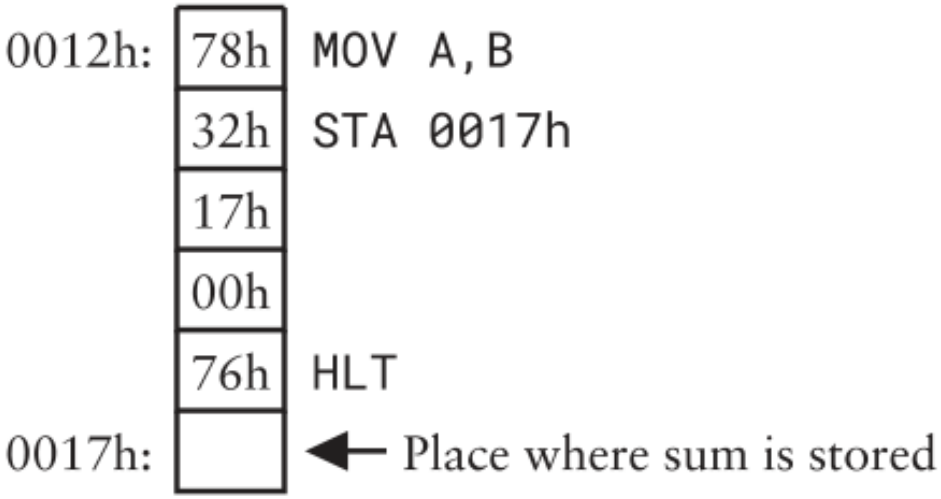
Программу легко изменить, если числа находятся в другом месте памяти или их количество отличается от 200. Все изменяемые данные задаются в самом начале. При написании программы всегда полезно думать о том, как её, возможно, придётся менять в будущем.

Компьютерную программу очень редко можно написать только одним способом. Существует немного иной вариант этого примера, использующий всего одну инструкцию перехода. Его начало почти совпадает с первым вариантом.

Единственное различие состоит в том, что C устанавливается равным 199, а не 200. Причина станет ясна чуть позже.

Средняя часть программы перестроена. Теперь она начинается с увеличения HL и прибавления следующего значения списка:





После прибавления следующего значения содержимое счётчика C переносится в A, уменьшается на 1, и новое значение возвращается в C. Затем JNZ переходит к началу цикла, если результат SUI *не* равен нулю.

Если результат SUI равен нулю, исполнение продолжается с инструкции сразу после JNZ. Заключительная часть программы сохраняет накопленную сумму в памяти и останавливает CPU.

Удаление одной инструкции перехода сокращает программу на три байта, хотя делает её немного сложнее. Почему C следует установить равным 199, а не 200? Потому что счётчик изменяется и проверяется *после* прибавления значения из памяти. Если список содержит лишь два числа, оба будут прочитаны до первой проверки JNZ. Поэтому C надо было бы инициализировать единицей, а не двойкой. Для списка всего из одного байта эта программа вообще не работает. Понятно почему?

При определении числа итераций легко ошибиться. Подобные ошибки настолько распространены, что получили собственное название: *ошибки на единицу*, или *off-by-one errors*.

Иногда количество чисел заранее неизвестно, зато известно, что последнее значение списка равно 00h. Оно сообщает программе, что список закончился. Такое значение называют *сторожевым*, или *sentinel*. В этом случае значение из памяти можно сравнить с 00h инструкцией Compare и по результату выйти из цикла.

В следующем варианте со сторожевым значением адреса памяти больше не показываются. Вместо них используются слова, называемые *метками*. Они выглядят как обычные слова, но по-прежнему обозначают ячейки памяти. После метки ставится двоеточие:

```
Start:    MVI L,00h
          MVI H,10h
          MVI B,00h
Loop:     MOV A,M
          CPI 00h
          JZ End
          ADD B
          MOV B,A
          INX HL
          JMP Loop
End:      MOV A,B
          STA Result
          HLT
Result:
```

После загрузки очередного значения из памяти в аккумулятор инструкцией `MOV A,M` команда `CPI 00h` сравнивает его с нулём. Если A равен 00h, устанавливается Zero и JZ переходит к End. Иначе значение прибавляется к текущей сумме в B, а HL увеличивается для следующей итерации.

Метки избавляют от необходимости вручную определять адреса инструкций, хотя вычислить эти адреса всегда возможно. Если программа начинается с 0000h, первые три инструкции занимают по два байта, поэтому Loop соответствует адресу 0006h. Следующие семь инструкций занимают в общей сложности 12 байтов, следовательно End находится по адресу 0012h, а Result - по адресу 0017h.

Условный переход - очень важная возможность CPU, но его значение, вероятно, ещё больше, чем можно предположить.

В 1936 году 24-летний выпускник Кембриджского университета Алан Тьюринг взялся за задачу математической логики, поставленную немецким математиком Давидом Гильбертом и известную как *Entscheidungsproblem*, или "проблема разрешения".

```
Start:    MVI L,00h
          MVI H,10h
          MVI B,00h
Loop:     MOV A,M
          CPI 00h
          JZ End
          ADD B
          MOV B,A
          INX HL
          JMP Loop
End:      MOV A,B
          STA Result
          HLT
Result:
```

Проблема разрешения спрашивала: существует ли процесс, способный определить разрешимость произвольного высказывания математической логики, то есть установить, истинно оно или ложно?

Отвечая на этот вопрос, Алан Тьюринг избрал крайне необычный подход. Он предположил существование простой вычислительной машины, действующей по простым правилам. В действительности он её не строил: это был воображаемый компьютер. Но, помимо доказательства отрицательного ответа на Entscheidungsproblem, Тьюринг заложил несколько фундаментальных понятий цифровых вычислений, значение которых вышло далеко за пределы этой задачи математической логики.

Изобретённая им воображаемая вычислительная машина теперь известна как *машина Тьюринга*. По вычислительным возможностям она функционально эквивалентна всем цифровым компьютерам, построенным с тех пор. Тем, кто захочет познакомиться с оригинальной статьёй Тьюринга, описывающей эту машину, может помочь книга автора *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*.

Разные цифровые компьютеры работают с разной скоростью, имеют доступ к разным объёмам оперативной и долговременной памяти и снабжены разным оборудованием. Но по вычислительной силе они функционально эквивалентны. Все они способны решать задачи одного типа, поскольку обладают одной особой возможностью: условным переходом, зависящим от результата арифметической операции.

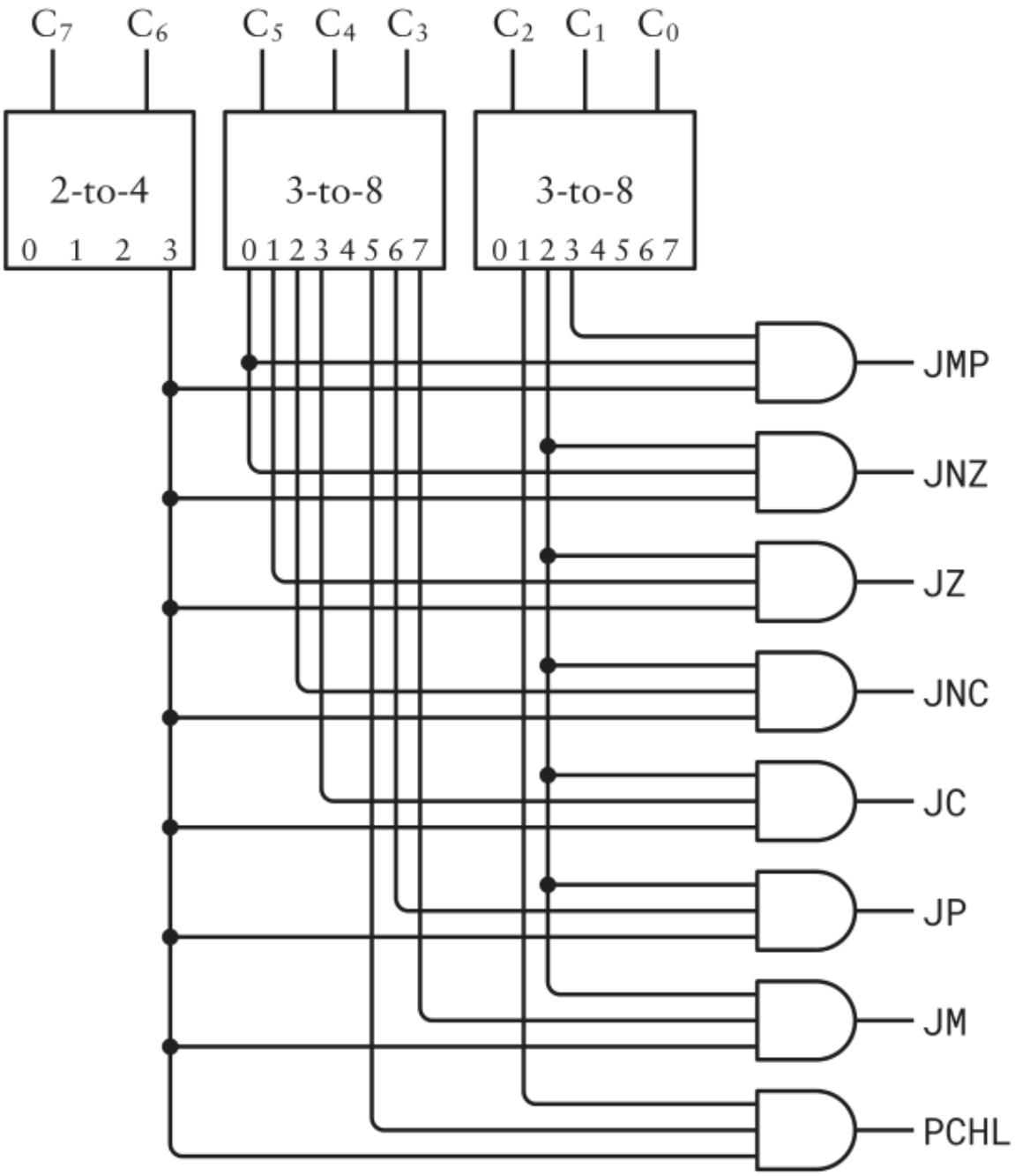
Все языки программирования, поддерживающие условный переход или его эквивалент, фундаментально равносильны. Такие языки называют *полными по Тьюрингу*. Почти все языки программирования удовлетворяют этому условию, но языки разметки, например применяемый на веб-страницах HyperText Markup Language (HTML), полными по Тьюрингу не являются.

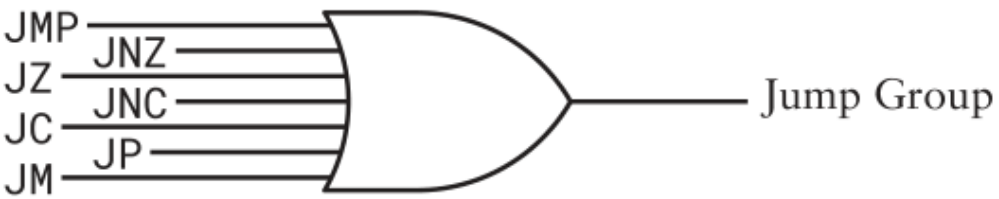
Кроме перечисленных инструкций перехода полезна ещё одна, использующая значение HL:

Инструкция	Описание	Код
PCHL	Скопировать HL в счётчик команд	E9h

Семь инструкций перехода и PCHL сравнительно легко добавить во временную схему предыдущей главы. Вспомним схему со страницы 366 главы 23: входы трёх её дешифраторов соответствуют восьми битам кода операции.

Различные сочетания выходов этих дешифраторов формируют сигналы, когда код операции соответствует инструкции перехода.



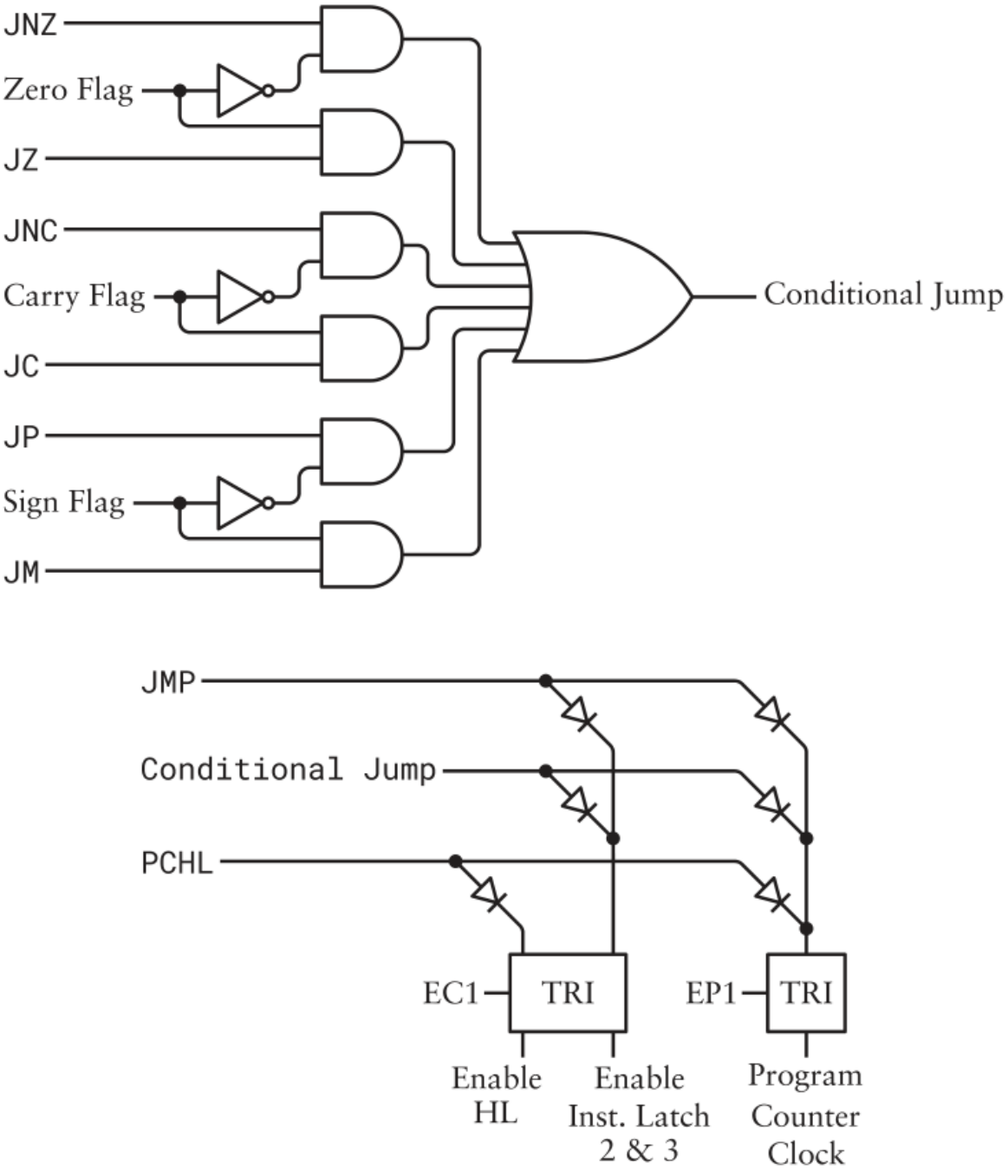


Все инструкции перехода, кроме PCHL, можно объединить в группу семивходовым элементом ИЛИ. Полученный Jump Group встраивается в схему со страницы 367 главы 23, которая определяет, сколько байтов надо выбрать из памяти и сколько циклов требуется для исполнения.

Jump Group указывает, что из памяти выбираются три байта: код операции и двухбайтовый адрес. PCHL имеет длину только один байт. Все эти инструкции исполняются за один цикл и используют только шину адреса.

Для исполнения переходов создадим сигнал, показывающий, должен ли состояться условный переход. Для этого сигналы декодированной инструкции объединяются с флагами ALU из главы 21. JNZ соединяется с инверсией Zero, JZ - с Zero, JNC - с инверсией Carry, JC - с Carry, JP - с инверсией Sign, а JM - с Sign. Результаты объединяются сигналом Conditional Jump.

Теперь эти сигналы довольно просто включить в диодную ROM-матрицу со страницы 374 главы 23:



JMP и условные переходы включают Instruction Latches 2 и 3 на шину адреса, тогда как PCHL включает на неё HL. Во всех случаях этот адрес сохраняется в счётчике команд.

Интерактивная версия расширенного CPU доступна на сайте CodeHiddenLanguage.com.

Почти всякая содержательная работа компьютерной программы связана с повторением и поэтому прекрасно подходит для цикла. Хороший пример - умножение. В главе 21 автор обещал показать, как заставить этот CPU умножать; теперь настало время выполнить обещание.

Начнём с простейшего случая: умножения двух байтов, например 132 на 209, или в шестнадцатеричной записи 84h на D1h. Эти два числа называются *множителем* и *множимым*, а результат - *произведением*.

В общем случае произведение двух однобайтовых чисел занимает два байта. В нашем примере результат легко вычислить как 27 588, или 6BC4h, но поручим вычисление CPU.

Ранее H и L использовались для шестнадцатибитного адреса памяти. Однако их можно использовать как обычные восьмибитные регистры или рассматривать пару HL как хранилище двухбайтового значения. Здесь в HL будет находиться произведение.

Сначала B получает множимое, C - множитель, а H и L обнуляются:

```
Start: MVI B,D1h   ; Поместить множимое в B
       MVI C,84h   ; Поместить множитель в C
       MVI H,00h   ; Инициализировать HL нулём
       MVI L,00h
```

Небольшие пояснения справа после точки с запятой называются *комментариями*. Использование точки с запятой перед комментарием встречается уже в оригинальной документации Intel по CPU 8080.

Сначала рассмотрим простой способ умножения - повторное сложение. Множитель прибавляется к HL столько раз, каково значение множимого.

Первым делом проверяется, не равно ли множимое в B нулю. Если равно, умножение окончено:

```
Loop:  MOV A,B     ; Проверить, равен ли B нулю
       CPI 00h
       JZ Done     ; Если да, всё закончено
```

Иначе множитель из C прибавляется к паре HL. По существу это шестнадцатибитное сложение: C прибавляется к L, а затем к H прибавляется ноль и возможный перенос от первого сложения:

```
       MOV A,L     ; Прибавить C к HL
       ADD C
       MOV L,A
       MOV A,H
       ACI 00h
       MOV H,A
```

Затем множимое B уменьшается, показывая, что к HL осталось прибавить на одно число меньше, и программа возвращается к Loop:

```
       MOV A,B     ; Уменьшить B
       SBI 01h
       MOV B,A
       JMP Loop    ; Повторить вычисление
```

```
Start:  MVI B,D1h   ; Set B to multiplicand
        MVI C,84h   ; Set C to multiplier
        MVI H,00h   ; Initialize HL to zero
        MVI L,00h

Loop:   MOV A,B     ; Check whether B is zero
        CPI 00h
        JZ Done     ; All finished if that's the case

            MOV A,L     ; Add C to HL
            ADD C
            MOV L,A
            MOV A,H
            ACI 00h
            MOV H,A

          MOV A,B     ; Decrement B
          SBI 01h
          MOV B,A
          JMP Loop    ; Repeat calculation
```

Когда выполняется предыдущий переход к Done, умножение завершено, а HL содержит произведение:

```
Done:  HLT        ; Результат находится в HL
```

Это не лучший способ перемножить два байта, зато он прост для понимания. Решения такого типа иногда называют подходом *грубой силы*. Скорость умножения здесь вообще не учитывается. Код даже не сравнивает два числа, чтобы использовать в качестве числа итераций меньшее из них. Несколько дополнительных инструкций позволили бы выполнить 132 сложения числа 209 вместо 209 сложений числа 132.

Можно ли умножать лучше? Вспомним десятичное умножение столбиком:

```
   132
 x 209
  1188
 264
27588
```

Под чертой находятся 132, умноженное на 9, и 132, умноженное на 2 со сдвигом на две позиции влево, что фактически означает умножение на 200. Произведение 132 на 0 можно вовсе не записывать, поскольку оно равно нулю. Вместо 209 или 132 сложений требуется сложить всего два числа.

Как выглядит то же умножение в двоичной системе? Множитель 132 равен `10000100`, а множимое 209 - `11010001`.

Для каждого бита множимого `11010001`, начиная справа, этот бит умножается на множитель `10000100`. Если бит равен 1, результатом служит множитель, сдвинутый влево на соответствующее число позиций. Если бит равен 0, результат равен нулю и его можно не учитывать.

Под чертой оказываются лишь четыре экземпляра множителя `10000100`, поскольку в множимом `11010001` только четыре единицы.

```
  132
× 209
 1188
264
27588
```

```
      10000100
    × 11010001
 ---------------
      10000100
     10000100
    10000100
10000100
---------------
1101011111000100
```

Такой подход сокращает число сложений до абсолютного минимума. Для умножения шестнадцати- или тридцатидвухразрядных чисел это становится ещё важнее.

Но способ выглядит сложнее: надо проверять, какие биты множимого равны 1, а какие 0.

Для проверки применяется инструкция 8080 ANA, то есть AND with Accumulator. Она выполняет поразрядную операцию И над двумя байтами. Операция называется поразрядной, потому что для каждой позиции результат равен 1 только тогда, когда соответствующие биты обоих байтов равны 1; во всех остальных случаях получается 0.

Поместим множимое D1h в D:

```
MVI D,D1h
```

Чтобы определить, равен ли младший бит D единице, надо выполнить ANA между D и 01h. Сначала значение 01h помещается в E:

```
MVI E,01h
```

ALU работает только с аккумулятором, поэтому одно из чисел сначала переносится туда:

```
MOV A,D
ANA E
```

Результат И равен 1, если младший, самый правый бит D равен 1, и 0 в противном случае. Значит, при нулевом младшем бите D устанавливается флаг Zero, позволяющий выполнить условный переход.

Следующий бит проверяется уже маской 02h, а остальные - масками 04h, 08h, 10h, 20h, 40h и 80h. Каждая из них вдвое больше предыдущей: удвоение 01h даёт 02h, затем 04h, затем 08h и так далее.

E начинается с 01h. Его можно удвоить, прибавив к самому себе:

```
MOV A,E
ADD E
MOV E,A
```

Теперь E равно 02h. После ещё одного исполнения трёх инструкций получится 04h, затем 08h и так далее. По существу операция постепенно сдвигает единичный бит от младшей позиции к старшей, от 01h до 80h.

Чтобы прибавлять множитель к результату, его тоже придётся сдвигать. Тогда он больше не помещается в восьмибитный регистр и должен рассматриваться как шестнадцатибитное значение.

```
MVI D,D1h

MVI E,01h

MOV A,D
ANA E

MOV A,E
ADD E
MOV E,A
```

Поэтому множитель сначала сохраняется в C, а B устанавливается равным 0. Регистры B и C можно считать парой BC, которая хранит шестнадцатибитный множитель и сдвигается для шестнадцатибитных сложений.

Улучшенный алгоритм начинается такой инициализацией:

```
Start: MVI D,D1h   ; Множимое
       MVI C,84h   ; Хранить множитель в BC
       MVI B,00h
       MVI E,01h   ; Проверяемый бит
       MVI H,00h   ; HL хранит двухбайтовый результат
       MVI L,00h
```

Цикл сначала проверяет очередной бит множимого:

```
Loop:  MOV A,D
       ANA E       ; Проверить, равен бит 0 или 1
       JZ Skip
```

Если бит равен 1, результат проверки не нулевой и пара BC прибавляется к HL. Восьмибитные регистры обрабатываются отдельно. Для младших байтов используется ADD, а для старших - ADC, учитывающая перенос:

```
       MOV A,L     ; Прибавить BC к HL
       ADD C
       MOV L,A
       MOV A,H
       ADC B
       MOV H,A
```

На настоящем Intel 8080, а не на построенном подмножестве, эти шесть инструкций можно заменить одной `DAD BC`, удобно прибавляющей BC к HL. DAD входит в число инструкций 8080, работающих с шестнадцатибитными значениями.

Далее BC удваивается, то есть фактически сдвигается влево перед следующим возможным сложением. Этот код исполняется независимо от того, была ли BC прибавлена к HL:

```
Skip:  MOV A,C     ; Удвоить BC, множитель
       ADD C
       MOV C,A
       MOV A,B
       ADC B
       MOV B,A
```

Затем удваивается E, содержащий проверяемый бит. Если результат не нулевой, программа возвращается к Loop для следующей итерации.

```
Start:  MVI D,D1h   ; Multiplicand
        MVI C,84h   ; Store multiplier in BC
        MVI B,00h

        MVI E,01h   ; Bit tester
        MVI H,00h   ; Use HL for 2-byte result
        MVI L,00h

Loop:   MOV A,D
        ANA E       ; Test whether bit is 0 or 1
        JZ Skip

           MOV A,L     ; Add BC to HL
           ADD C
           MOV L,A
           MOV A,H
           ADC B
           MOV H,A

Skip:   MOV A,C     ; Double BC, the multiplier
        ADD C
        MOV C,A
        MOV A,B
        ADC B
        MOV B,A

    MOV A,E     ; Удвоить E, проверяемый бит
    ADD E
    MOV E,A
    JNZ Loop
```

Код после метки Loop исполняется ровно восемь раз. После восьми удвоений восьмибитный E переполняется и становится равным нулю. Умножение завершено:

```
Done:  HLT        ; Результат находится в HL
```

Умножение двух шестнадцатибитных или двух тридцатидвухразрядных значений, конечно, сложнее и требует больше регистров. Когда регистров для промежуточных значений не хватает, можно временно использовать область памяти. Небольшой блок памяти такого назначения обычно называют *рабочей областью*, или *scratchpad memory*.

Цель упражнения не в том, чтобы напугать читателя или отговорить его от профессии программиста. Оно показывает, что набор логических вентилей, реагирующих на хранящиеся в памяти коды, действительно способен объединять простейшие операции для выполнения сложных задач.

В настоящем программировании умножение гораздо проще благодаря так называемым языкам высокого уровня, которые будут обсуждаться в главе 27. Волшебство программного обеспечения состоит в том, что другие люди уже выполнили трудную работу и нам не приходится повторять её.

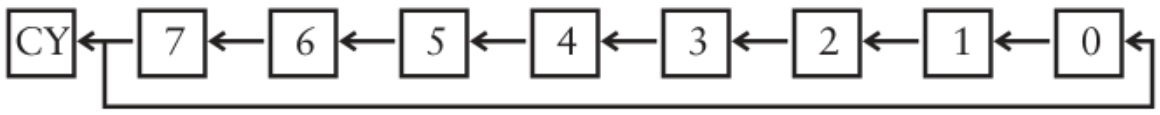
В машинном коде умножение требует сдвига битов. До сих пор он выполнялся прибавлением значения к самому себе. Настоящий Intel 8080 предлагает более удобный способ: четыре инструкции rotate сдвигают биты без необходимости складывать регистр с самим собой.

Инструкция	Описание	Код
RLC	Циклический сдвиг влево	07h
RRC	Циклический сдвиг вправо	0Fh
RAL	Сдвиг влево через перенос	17h
RAR	Сдвиг вправо через перенос	1Fh

Все они работают со значением аккумулятора и изменяют Carry.

RLC сдвигает биты аккумулятора влево. Старший бит одновременно устанавливает Carry и становится новым младшим битом.

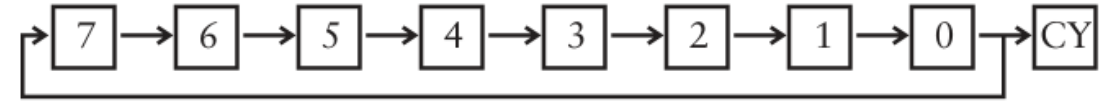
Instruction	Description	Opcode
RLC	Rotate left	07h
RRC	Rotate right	0Fh
RAL	Rotate left through carry	17h
RAR	Rotate right through carry	1Fh

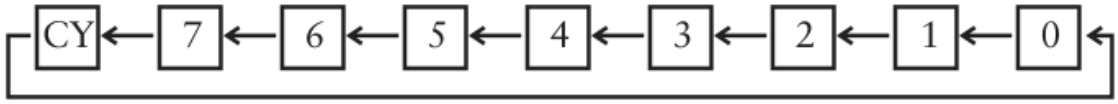


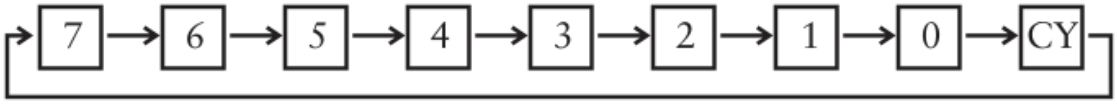
RRC действует сходным образом, но сдвигает биты вправо. Младший бит одновременно устанавливает Carry и становится новым старшим битом.

RAL напоминает удвоение аккумулятора, но новым младшим битом становится прежнее значение Carry. Это удобно при сдвиге многобайтового значения.

RAR похожа на RAL, но сдвигает биты вправо, пропуская их через Carry.







Хотя инструкции циклического сдвига полезны в некоторых ситуациях, они не обязательны, поэтому автор не станет добавлять их в строящийся CPU.

Переходы и циклы позволяют многократно исполнять группу инструкций. Однако часто нужен более гибкий способ запуска группы инструкций. Например, однажды написанную группу может потребоваться вызывать из разных частей программы. Такой группой мог бы быть универсальный алгоритм умножения.

Подобные группы называют *функциями*, *процедурами*, *подпрограммами* или просто *программами*.

Intel 8080 реализует подпрограммы инструкцией CALL. Её синтаксис похож на JMP, потому что вслед за кодом операции идёт адрес памяти:

```
CALL addr
```

Как и JMP, CALL переходит по этому адресу и продолжает исполнение оттуда. Но перед переходом CALL сохраняет напоминание о том, откуда был совершён вызов, а именно адрес инструкции, следующей за CALL. Вскоре станет ясно, где именно хранится этот адрес.

Инструкция RET, сокращение от return, тоже похожа на JMP, но переходит по адресу, ранее сохранённому CALL. Подпрограмма часто завершается инструкцией RET.

Инструкции вызова и возврата 8080 имеют следующие коды:

Инструкция	Описание	Код
CALL	Перейти к подпрограмме	CDh
RET	Вернуться из подпрограммы	C9h

Intel 8080 поддерживает также условные вызовы и условные возвраты, но они используются гораздо реже, чем CALL и RET.

Рассмотрим практический пример. Допустим, программа должна вывести значение байта, например 5Bh. В главе 13 было показано, как ASCII представляет буквы, цифры и знаки. Но значение байта 5Bh нельзя вывести кодом ASCII 5Bh: это код символа левой квадратной скобки. Вместо этого байт 5Bh надо преобразовать в два кода ASCII:

• 35h - код символа `5`;

• 42h - код символа `B`.

Так значение байта отображается понятным человеку способом, по крайней мере человеку, знакомому с шестнадцатеричной системой.

Сначала байт разделяется на две части: верхние четыре и нижние четыре бита, которые иногда называют *полубайтами*, или *nibbles*. В нашем примере 5Bh разделяется на 05h и 0Bh.

Затем каждое четырёхбитное значение преобразуется в ASCII. Значения от 0h до 9h соответствуют кодам 30h-39h для цифр `0-9`. Значения Ah-Fh соответствуют кодам 41h-46h для букв `A-F`.

Небольшая подпрограмма преобразует четырёхбитное значение в аккумуляторе в ASCII:

Instruction	Description	Opcode
`CALL`	Jumps to a subroutine	CDh
`RET`	Returns from a subroutine	C9h

```
1532h: FEh  Digit:   CPI 0Ah
       0Ah
       DAh           JC Number
       39h
       15h
       C6h           ADI 07h
       07h
1539h: C6h  Number:  ADI 30h
       30h
       C9h           RET
```

```
1532h  Digit:   CPI 0Ah
                 JC Number
                 ADI 07h
1539h  Number:  ADI 30h
                 RET
```

Здесь снова показаны адреса памяти, поскольку они важны для понимания происходящего. Подпрограмма начинается по адресу 1532h, но в этом адресе нет ничего особенного: автор просто решил разместить её именно там.

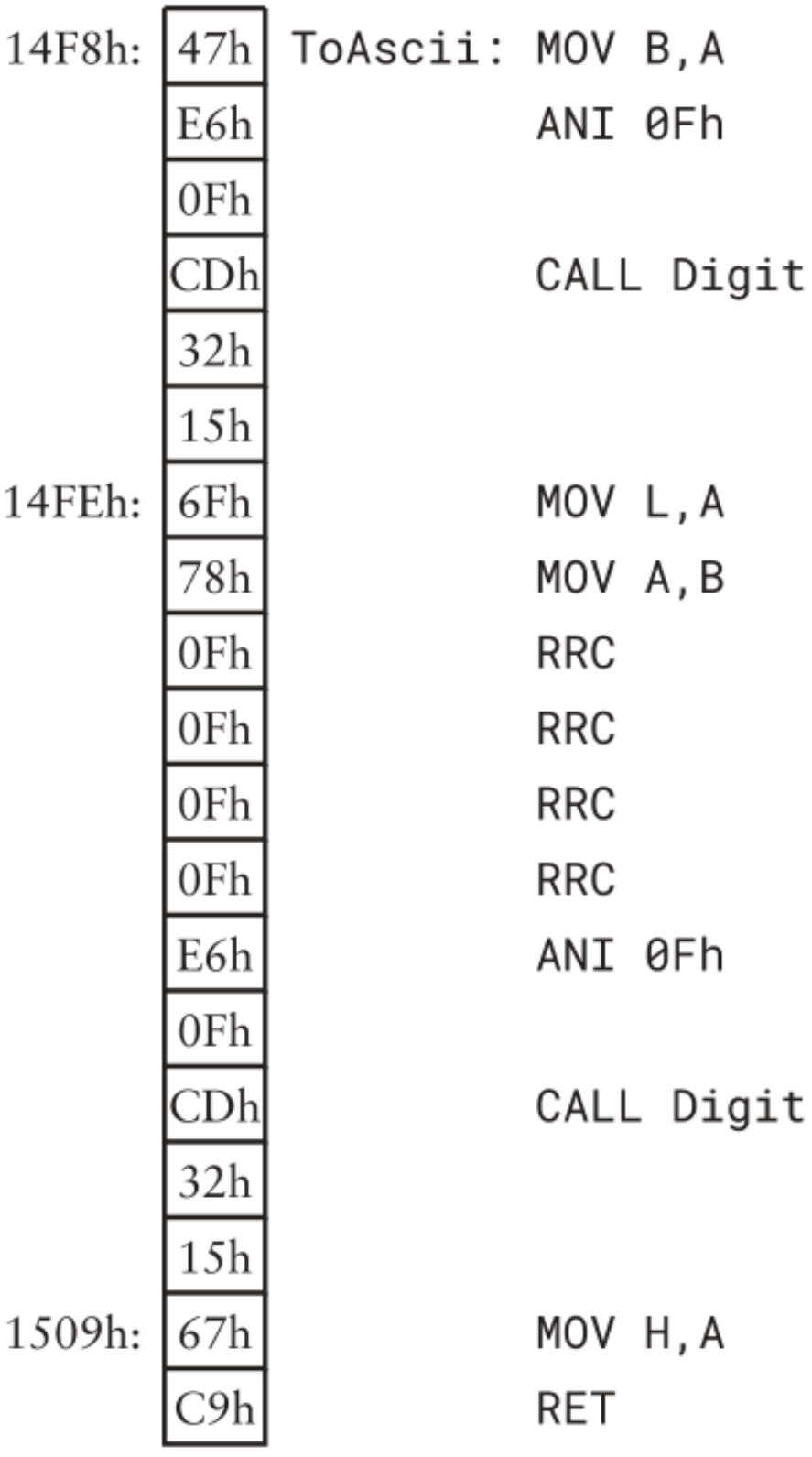
Предполагается, что аккумулятор содержит преобразуемое значение. Такое входное значение часто называют *аргументом* или *параметром* подпрограммы.

Сначала Compare Immediate устанавливает флаги ALU так, словно произошло вычитание. Если аккумулятор содержит, например, 05h, вычитание 0Ah требует заёма, поэтому устанавливается Carry. Поскольку Carry установлен, JC переходит к метке Number. Там к аккумулятору прибавляется 30h, и получается 35h - код ASCII цифры 5.

Если же аккумулятор содержит 0Bh, при вычитании 0Ah заём не нужен. CPI не устанавливает Carry, поэтому переход не происходит. Сначала к аккумулятору прибавляется 07h: 0Bh плюс 07h даёт 12h. Затем вторая ADI добавляет 30h, получая 42h - код ASCII буквы B. Сложение двух констант - небольшая хитрость, позволяющая использовать вторую ADI как для букв, так и для цифр.

В обоих случаях следующей исполняется RET, завершающая подпрограмму.

Но требовалась подпрограмма, превращающая целый байт в два кода ASCII. Вторая подпрограмма дважды вызывает Digit: сначала для младшего, затем для старшего полубайта. В начале исходный байт находится в аккумуляторе, а результаты сохраняются в H и L. Подпрограмма называется ToAscii и начинается по адресу 14F8h.


14F8h: 47h ToAscii: MOV B,A
E6h ANI 0Fh
0Fh
CDh CALL Digit
32h
15h
14FEh: 6Fh MOV L,A
78h MOV A,B
0Fh RRC
0Fh RRC
0Fh RRC
0Fh RRC
E6h ANI 0Fh
0Fh
CDh CALL Digit
32h
15h
1509h: 67h MOV H,A
C9h RET

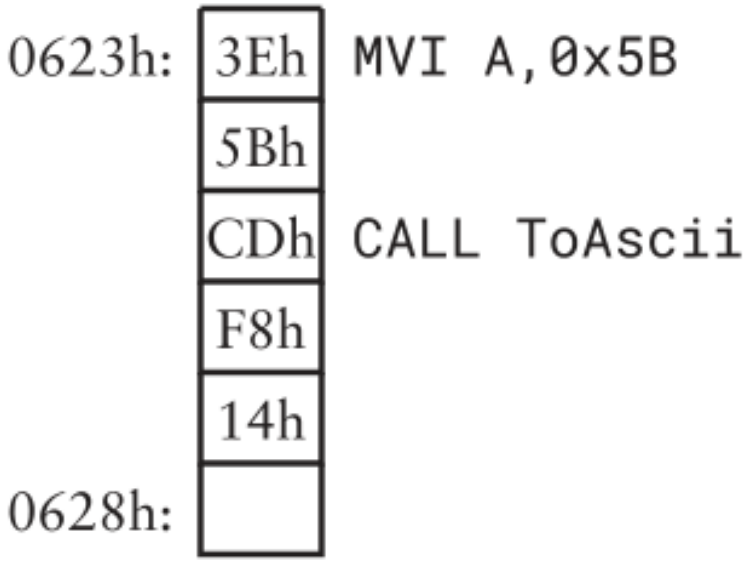
```
14F8h  ToAscii: MOV B,A
                ANI 0Fh
                CALL Digit
                MOV L,A
                MOV A,B
                RRC
                RRC
                RRC
                RRC
                ANI 0Fh
                CALL Digit
                MOV H,A
                RET
```

Сначала исходный байт сохраняется в B. ANI, то есть AND Immediate, выполняет поразрядное И с 0Fh, оставляя только нижние четыре бита. Затем CALL вызывает Digit по адресу 1532h, а полученный код сохраняется в L.

Исходный байт возвращается из B. Четыре RRC сдвигают старший полубайт вниз, в младшие четыре позиции. После ещё одной ANI вызывается Digit во второй раз. Её результат сохраняется в H, после чего ToAscii завершается RET.

Посмотрим на работу вызова. Где-то в другой части программы может находиться небольшой фрагмент с CALL подпрограммы ToAscii по адресу 14F8h:

```
0623h  MVI A,5Bh
0625h  CALL ToAscii
0628h  ...
```



Когда исполнение продолжится по адресу 0628h, H и L будут содержать коды ASCII двух цифр значения 5Bh.

Как работают CALL и RET?

При CALL адрес сохраняется в совершенно особом месте, благодаря чему после завершения подпрограммы можно продолжить исполнение. Это место называется *стеком*. Стек представляет собой область памяти, расположенную как можно дальше от всего остального. В восьмибитном CPU вроде Intel 8080 стек находится в самом конце памяти.

Intel 8080 содержит шестнадцатибитный регистр, называемый *указателем стека*. После сброса 8080 он инициализируется адресом 0000h. Программа может изменить его инструкциями SPHL, устанавливающей указатель из HL, или LXI SP, загружающей непосредственный адрес. Но в примере оставим значение по умолчанию 0000h.

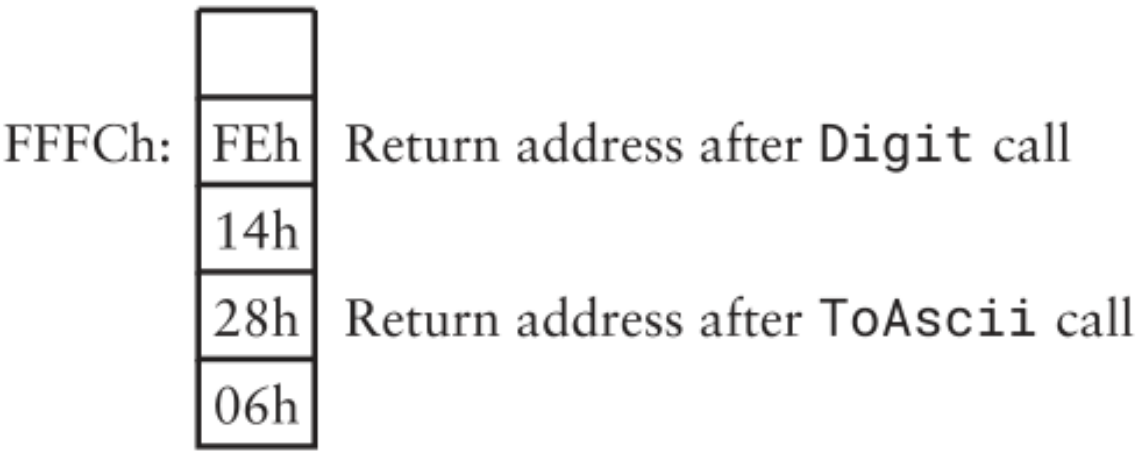
При исполнении `CALL ToAscii` последовательно происходит следующее:

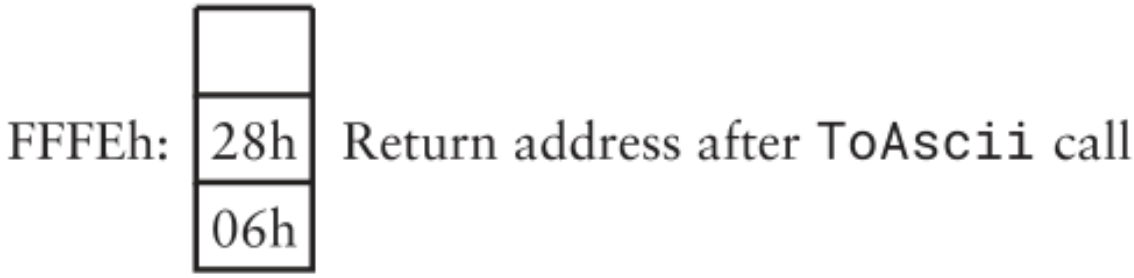
1. Указатель стека уменьшается. Из исходного 0000h получается FFFFh - максимальное шестнадцатибитное значение, адресующее последний байт памяти.
2. Старший байт адреса после CALL - адреса 0628h, являющегося текущим значением счётчика команд, - сохраняется в памяти по адресу указателя стека. Это байт 06h.
3. Указатель стека снова уменьшается и становится равен FFFEh.
4. Младший байт адреса после CALL сохраняется по адресу указателя стека. Это байт 28h.
5. Адрес из CALL, равный 14F8h, загружается в счётчик команд. CPU фактически переходит к подпрограмме ToAscii.

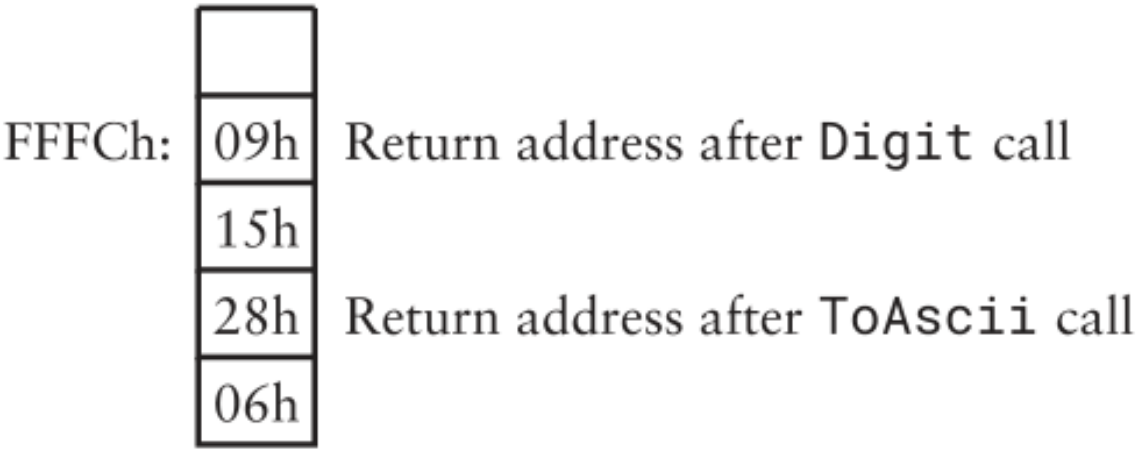
Верхняя область RAM теперь содержит адрес возврата 0628h, а указатель стека равен FFFEh. CALL оставила маленькую дорожку из хлебных крошек, по которой можно вернуться домой.

ToAscii начинает исполняться и сама вызывает Digit. Адрес инструкции ToAscii, следующей за этим CALL, равен 14FEh.

Во время вызова Digit адрес 14FEh помещается в стек, который теперь содержит два адреса возврата:





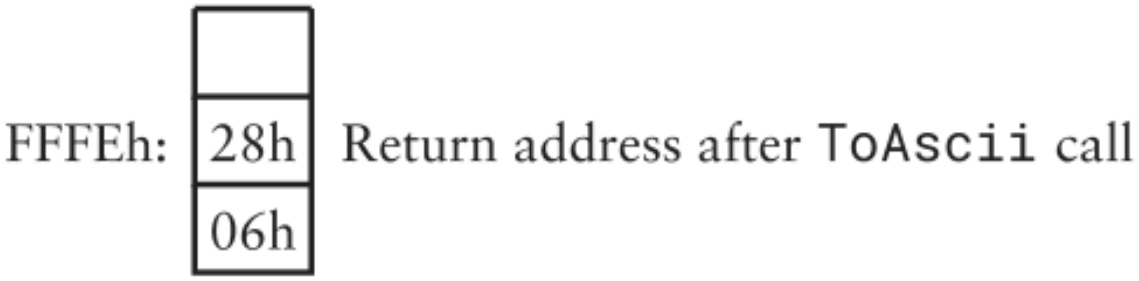


Указатель стека становится равен FFFCh, и начинается исполнение Digit. Когда Digit доходит до RET, выполняются следующие действия:

1. Читается байт памяти по адресу указателя стека - FEh.
2. Указатель стека увеличивается.
3. Читается следующий байт по адресу указателя стека - 14h.
4. Указатель стека снова увеличивается.
5. Два байта загружаются в счётчик команд, фактически вызывая переход к адресу 14FEh внутри ToAscii, то есть возврат в вызвавшую Digit подпрограмму.

Стек возвращается к состоянию перед первым вызовом Digit, а указатель снова равен FFFEh. Адрес 14FEh всё ещё физически остаётся в памяти, но больше не имеет значения. Следующий вызов Digit запишет поверх него новый адрес возврата.

После второго вызова Digit в стек помещается адрес следующей инструкции ToAscii. Когда Digit снова исполняет RET, она переходит к адресу 1509h. Затем RET в самой ToAscii извлекает из стека адрес 0628h и переходит к нему - к инструкции после исходного вызова ToAscii.



Так работает стек.

Формально стек относится к хранилищам типа Last-In-First-Out, или LIFO: значение, добавленное последним, извлекается первым. Стек часто представляют стопкой тарелок в столовой, поддерживаемой пружиной. Тарелки кладут сверху и снимают в порядке, обратном добавлению.

Добавление в стек называется *вталкиванием*, или *push*, а удаление - *выталкиванием*, или *pop*. В Intel 8080 есть несколько инструкций PUSH и POP, сохраняющих регистры в стеке и позднее восстанавливающих их:

Инструкция	Описание	Код
PUSH BC	Сохранить B и C в стеке	C5h
PUSH DE	Сохранить D и E в стеке	D5h
PUSH HL	Сохранить H и L в стеке	E5h
PUSH PSW	Сохранить Program Status Word в стеке	F5h
POP BC	Восстановить B и C из стека	C1h
POP DE	Восстановить D и E из стека	D1h
POP HL	Восстановить H и L из стека	E1h
POP PSW	Восстановить Program Status Word из стека	F1h

PSW означает Program Status Word. Ничего нового здесь нет: в одном байте находится аккумулятор, в другом - флаги ALU.

PUSH и POP позволяют удобно сохранить регистры при вызове подпрограммы. Вызывающий код может поместить регистры в стек перед CALL и извлечь после него. Тогда подпрограмма свободно использует эти регистры, не нарушая работу вызывающего кода. Либо сама подпрограмма выполняет PUSH в начале и POP перед RET.

PUSH и POP должны быть сбалансированы так же, как CALL и RET. Если подпрограмма дважды выполнит PUSH, только один раз POP, а затем RET, переход произойдёт совсем не туда, куда ожидалось.

Ошибочный код может слишком много раз извлечь данные из стека, и указатель начнёт адресовать начало памяти вместо её конца. Эта проблема называется *опустошением стека*, или *stack underflow*, и способна привести к тому, что содержимое стека перезапишет программу.

Связанная проблема возникает, когда в стек помещается слишком много данных: он разрастается и, вероятно, тоже перезаписывает код. Это *переполнение стека*, или *stack overflow*. То же название носит популярный интернет-форум, где программисты ищут ответы на технические вопросы.

CALL и RET не обязательны для полноты CPU по Тьюрингу, но на практике они чрезвычайно удобны, а кто-то назовёт их незаменимыми. Подпрограммы являются основными организационными элементами программ на языке ассемблера и играют важную роль во многих других языках программирования.

Автор не добавляет CALL, RET, PUSH и POP в CPU, проектировавшийся в последних главах. Для них нужна более универсальная архитектура. Нетрудно представить общую реализацию: на шину адреса добавляется новая шестнадцатибитная защёлка указателя стека, похожая на защёлку счётчика команд. Это простая часть.

Но во время CALL счётчик команд надо поместить в стек, а во время RET извлечь обратно. Для этого оба байта счётчика команд должны появляться на шине данных, чего нынешняя конструкция не предусматривает.

Хотя стековые инструкции не добавлены в строящийся CPU, на сайте CodeHiddenLanguage.com доступен полный эмулятор 8080.

За последние главы мы увидели, как восьмибитный микропроцессор вроде Intel 8080 исполняет коды инструкций. Intel выпустила 8080 в 1974 году, и по сравнению со всеми последующими разработками он теперь выглядит довольно примитивным. По мере перехода CPU к шестнадцати-, тридцатидвух- и даже шестидесятичетырёхразрядной обработке они становились гораздо сложнее.

Тем не менее все CPU в основе работают одинаково: исполняют инструкции, извлекают байты из памяти, выполняют над ними арифметические и логические операции и сохраняют результаты обратно в память.

Пришло время выяснить, что ещё нужно для настоящего компьютера.

Глава 25. Периферийные устройства

Центральный процессор, безусловно, важнейшая часть компьютера, но его необходимо дополнить другим оборудованием. Компьютеру требуется оперативная память RAM, содержащая и машинные инструкции для процессора, и данные, к которым эти инструкции обращаются. RAM энергозависима: при отключении питания её содержимое исчезает. Поэтому полезна также долговременная массовая память, способная сохранять код и данные без питания.

Компьютеру нужен способ загрузить инструкции в RAM и способ изучить результаты работы программы. Современные компьютеры оснащены микрофонами, камерами и динамиками, а также радиопередатчиками и приёмниками для связи с Wi-Fi, устройствами Bluetooth и спутниками Global Positioning System (GPS).

Всё это называют *устройствами ввода* и *устройствами вывода*. Вместе их обычно обозначают сокращением I/O, от input/output, а в более общем смысле называют *периферийными устройствами*.

Вероятно, самая заметная периферия - видеодисплей: именно на него человек обычно смотрит, пользуясь настольным компьютером, ноутбуком, планшетом или телефоном. Возможно, читатель смотрит на дисплей и сейчас, читая эту книгу.

Все распространённые дисплеи создают изображение из строк и столбцов *пикселей* - маленьких цветных точек, которые можно разглядеть через увеличительное стекло. Общее число строк и столбцов обычно называют *разрешением дисплея*.

Например, стандартное разрешение телевидения высокой чёткости HDTV записывается как 1920 x 1080: 1920 пикселей по горизонтали и 1080 по вертикали, всего около двух миллионов пикселей, каждый из которых может иметь собственный цвет. Для компьютерных дисплеев это почти стало минимальным разрешением.

Пиксели зажигаются не все одновременно. Содержимое экрана хранится в специальном блоке памяти, а отдельные пиксели обновляются последовательно: слева направо по верхней строке, затем по следующим строкам вниз. Чтобы экран не мерцал, процесс идёт очень быстро, и весь дисплей обычно обновляется не реже 60 раз в секунду. Управляющая этим схема называется *видеоадаптером*.

Сколько памяти требуется для экрана 1920 x 1080?

Каждый из примерно двух миллионов пикселей имеет определённый цвет, представляющий собой сочетание красного, зелёного и синего основных цветов, то есть RGB. Художники могут пользоваться другой системой основных цветов, но в видеодисплеях применяются именно эти три. Изменение интенсивности отдельных компонентов создаёт все отображаемые цвета.

Интенсивность каждого основного цвета обычно задаётся одним байтом: 00h означает полное отсутствие компонента, FFh - максимальную интенсивность. Получается 256 уровней красного, 256 зелёного и 256 синего, то есть `256 x 256 x 256 = 16 777 216` цветов. В соответствии с принципом, что в компьютерах можно улучшить всё, некоторые компании увеличивают цветовой диапазон и разрешение, используя более восьми бит на компонент.

При работе с HTML цвета веб-страниц могут задаваться шестизначными шестнадцатеричными значениями со знаком решётки. Вот 16 стандартных цветов спецификации HTML 4.01 1999 года:

Цвет	Значение	Цвет	Значение
Чёрный	`#000000`	Зелёный	`#008000`
Серебристый	`#C0C0C0`	Лаймовый	`#00FF00`
Серый	`#808080`	Оливковый	`#808000`
Белый	`#FFFFFF`	Жёлтый	`#FFFF00`
Тёмно-бордовый	`#800000`	Тёмно-синий	`#000080`
Красный	`#FF0000`	Синий	`#0000FF`
Пурпурный	`#800080`	Бирюзовый	`#008080`
Фуксия	`#FF00FF`	Голубой	`#00FFFF`

Остальные цвета задаются другими значениями. После # идут три пары шестнадцатеричных цифр: уровень красного от 00h до FFh, затем зелёного и синего. Если все три компонента равны 00h, получается чёрный; если FFh - белый. Равные значения трёх компонентов дают оттенки серого.

Для экрана 1920 x 1080 каждому из двух миллионов пикселей нужны три байта RGB. Всего получается около шести миллионов байтов, или шесть мегабайт.

В предыдущих главах доступная CPU память RAM рассматривалась как единый монолитный блок. В действительности память кода и данных обычно разделяет адресное пространство с памятью видеодисплея. Благодаря этому компьютер очень быстро обновляет изображение простым записыванием байтов в RAM, что позволяет создавать скоростную графическую анимацию.

Построенный ранее восьмибитный CPU имеет шестнадцатибитный адрес и может адресовать 64 килобайта. Очевидно, шесть мегабайт видеопамяти в 64 килобайта не помещаются. Можно было бы поочерёдно подменять доступные CPU фрагменты памяти, но это заметно замедлило бы работу.

Именно поэтому дисплеи высокого разрешения стали возможны лишь после удешевления памяти и появления более мощных CPU, способных проворнее с ней работать. Тридцатидвухразрядный CPU обращается к памяти порциями по 32 бита, поэтому видеопамять часто организуют по четыре байта на пиксель, хотя для RGB нужны только три. Тогда экран 1920 x 1080 занимает восемь мегабайт, а не шесть.

Видеопамять обычно расположена в порядке обновления экрана: сначала первая строка, начиная с левого пикселя; для каждого пикселя следуют три байта красного, зелёного и синего и один неиспользуемый байт. Чтобы нарисовать на экране текст или графику, программа должна определить, какие пиксели изменить в графической памяти.

Компьютерная графика часто применяет математические средства аналитической геометрии. Весь дисплей или его прямоугольную область можно считать координатной системой, где каждый пиксель - точка с горизонтальной и вертикальной координатами `(x, y)`.

Например, точка `(10, 5)` находится на десять пикселей правее левого края и на пять пикселей ниже верхнего. Диагональ от неё до `(15, 10)` окрашивает точки `(10, 5)`, `(11, 6)`, `(12, 7)`, `(13, 8)`, `(14, 9)` и `(15, 10)`. Другие линии и кривые сложнее, но для них существует множество программных инструментов.

Текст - частный случай графики. Каждый символ конкретного шрифта задаётся набором прямых и кривых и дополнительной информацией, называемой *хинтами*, которая помогает отрисовать текст максимально разборчиво.

Трёхмерная графика гораздо сложнее: в ней различные способы закраски передают действие света и тени. Современным программам часто помогает графический процессор GPU, выполняющий часть тяжёлых математических вычислений для 3D.

В эпоху первых персональных компьютеров дисплеи высокого разрешения были недоступны. Первым графическим адаптером IBM PC стал Color Graphics Adapter (CGA). Он поддерживал три графических режима: 160 x 100 пикселей и 16 цветов при одном байте на пиксель; 320 x 200 и четыре цвета при двух битах на пиксель; 640 x 200 и два цвета при одном бите на пиксель.

В любом режиме требовалось только 16 000 байтов памяти. Например, `320 x 200 x 1/4 байта = 16 000`.

Некоторые ранние дисплеи вообще не выводили графику и ограничивались текстом. Это ещё один способ сэкономить память, лежавший в основе Monochrome Display Adapter (MDA), второго дисплея ранних IBM PC.

MDA показывал 25 строк по 80 символов одного цвета - зелёного на чёрном фоне. Каждый символ задавался восьмибитным кодом ASCII и сопровождался байтом атрибутов, управлявшим яркостью, инверсным изображением, подчёркиванием или миганием. Содержимое экрана занимало `25 x 80 x 2 = 4000` байтов. В схеме видеоадаптера ROM преобразовывала каждый символ ASCII в строки и столбцы пикселей.

Как внутри CPU есть шины для обмена данными между компонентами, так сам CPU часто подключён к внешним шинам, переносящим данные между CPU, памятью и периферией.

Видеопамять занимает обычное адресное пространство CPU. То же могут делать и другие устройства - это называется *memory-mapped I/O*, или вводом-выводом, отображённым в память. Но CPU может определить отдельную шину периферии и специальные средства для работы с устройствами I/O.

Среди 244 инструкций Intel 8080, на основе которого строился предыдущий CPU, есть две команды IN и OUT:

Инструкция	Описание	Код
IN port	Прочитать байт из входного порта	DBh
OUT port	Записать байт в выходной порт	D3h

За каждой следует восьмибитный *номер порта*, похожий на адрес памяти, но шириной восемь бит и предназначенный специально для устройств I/O. IN читает порт и сохраняет результат в аккумуляторе. OUT записывает содержимое аккумулятора в порт. Специальный сигнал 8080 показывает, обращается ли CPU к RAM, как обычно, или к порту I/O.

Рассмотрим клавиатуру настольного компьютера или ноутбука. Каждая клавиша - простой переключатель, замыкающийся при нажатии, и у каждой есть уникальный код. Допустим, клавиатура доступна через порт 25h. Тогда программа выполняет:

```
IN 25h
```

После этого аккумулятор содержит код нажатой клавиши.

Можно предположить, что это ASCII-код клавиши. Но заставлять оборудование самостоятельно определять ASCII непрактично и нежелательно.

Например, клавиша A может означать ASCII 41h или 61h в зависимости от того, удерживается ли Shift, определяющий верхний или нижний регистр буквы. Кроме того, на клавиатуре есть функциональные клавиши, стрелки и множество других клавиш, которым символы ASCII вообще не соответствуют. Небольшая программа способна определить, какой ASCII-код, если он существует, соответствует конкретному нажатию.

Но как программа узнаёт, что клавиша нажата? Один способ - очень часто проверять клавиатуру. Такой подход называется *опросом*, или *polling*. Лучше, если сама клавиатура сообщит CPU о нажатии. В общем случае устройство I/O сообщает CPU о событии с помощью *прерывания* - особого сигнала, поступающего в CPU.

Для прерываний в 8080 предусмотрены восемь инструкций restart:

Инструкция	Описание	Код
RST 0	Вызвать адрес 0000h	C7h
RST 1	Вызвать адрес 0008h	CFh
RST 2	Вызвать адрес 0010h	D7h
RST 3	Вызвать адрес 0018h	DFh
RST 4	Вызвать адрес 0020h	E7h
RST 5	Вызвать адрес 0028h	EFh
RST 6	Вызвать адрес 0030h	F7h
RST 7	Вызвать адрес 0038h	FFh

Каждая сохраняет текущий счётчик команд в стеке и переходит соответственно к 0000h, 0008h, 0010h и далее. RST 0 по существу похожа на сброс CPU, а по остальным адресам могут находиться инструкции перехода или вызова.

Вот как это работает. У 8080 есть внешний сигнал прерывания. Когда периферийное устройство, например физическая клавиатура, устанавливает его, оно одновременно помещает на шину данных байт одной из restart-инструкций. По соответствующему адресу памяти находится код обслуживания данного устройства I/O.

Такой режим называется *вводом-выводом по прерываниям*. CPU не должен постоянно опрашивать устройства: он занимается другой работой, пока устройство сигналом прерывания не сообщит о новом событии. Именно так клавиатура сообщает о нажатии.

Прерывания полезны также для мыши настольного компьютера или ноутбука, сенсорной панели ноутбука и сенсорного экрана планшета или телефона.

Кажется, будто мышь непосредственно связана с дисплеем: человек двигает её по столу вверх, вниз, влево или вправо, и указатель повторяет движение на экране. Но эта связь лишь иллюзия.

Мышь передаёт электрические импульсы, указывающие направление движения. Программное обеспечение обязано перерисовывать указатель в новых местах. Кроме движения мышь сообщает о нажатии и отпускании кнопок и о вращении колеса прокрутки.

Сенсорный экран обычно представляет собой слой поверх видеодисплея, способный обнаруживать изменение электрической ёмкости при касании пальца. Он сообщает положение одного или нескольких пальцев в тех же координатах `(x, y)`, которые программа использует для графики.

Программа получает события касания, отпускания и движения пальца по экрану. Эти сведения позволяют прокручивать изображение или перетаскивать графический объект. Можно интерпретировать и жесты двумя пальцами, например сведение и разведение пальцев для изменения масштаба.

Внутри компьютера всё цифровое, всё представлено числами. Но реальный мир часто аналоговый: свет и звук воспринимаются непрерывными, а не набором отдельных чисел.

Для преобразования аналоговых данных реального мира в числа и обратно созданы два устройства:

• аналого-цифровой преобразователь, ADC;

• цифро-аналоговый преобразователь, DAC.

На вход ADC подаётся напряжение, непрерывно меняющееся между двумя уровнями, а на выходе получается двоичное число, представляющее это напряжение. Выходы ADC часто имеют ширину восемь или шестнадцать бит. Например, восьмибитный ADC может выдавать 00h при нулевом напряжении, 80h при 2,5 вольта и FFh при 5 вольтах.

DAC действует в обратную сторону: получает восьми- или шестнадцатибитное двоичное число и выдаёт соответствующее ему напряжение.

В видеодисплеях DAC преобразуют цифровые значения пикселей в напряжения, управляющие интенсивностью света красного, зелёного и синего компонентов.

Цифровые камеры используют массив активных пиксельных датчиков APS. Под действием света каждый датчик создаёт напряжение, которое ADC превращает в число. Результатом становится *bitmap* - прямоугольный массив пикселей определённых цветов. Как и в видеопамяти, пиксели сохраняются последовательно: строка за строкой сверху вниз, внутри строки слева направо.

Bitmap может быть огромным. Камера телефона автора создаёт изображения 4032 x 3024 пикселя. Но для воспроизведения изображения нужны не все исходные данные, поэтому инженеры и математики разработали способы уменьшить число байтов. Это называется *сжатием*.

Простой способ сжатия bitmap - *кодирование длин серий*, или RLE. Если подряд идут десять пикселей одного цвета, достаточно сохранить сам пиксель и число 10. Метод хорошо работает только для изображений с большими одноцветными областями.

Более сложная и всё ещё распространённая схема - Graphics Interchange Format (GIF), произносимое автором как "джиф", подобно названию марки арахисового масла, хотя согласны с этим не все. Формат разработала в 1987 году прежняя онлайновая служба CompuServe.

GIF применяет LZW, названный по фамилиям создателей Lempel, Ziv и Welch. Алгоритм ищет шаблоны пикселей с разными значениями, а не только последовательности одинаковых. GIF также поддерживает простую анимацию из нескольких изображений.

Более совершенный Portable Network Graphics (PNG) появился в 1996 году. Он фактически преобразует соседние значения пикселей в разности между ними. Эти разности обычно меньше исходных чисел и эффективнее сжимаются.

GIF или PNG не обязательно меньше несжатого bitmap. Если процесс уменьшает одни изображения, другие неизбежно увеличивает. Такое случается с картинками, содержащими множество цветов или мелких деталей.

Для них полезны иные методы. Формат JPEG, произносимый "джей-пег", появился в 1992 году и стал чрезвычайно популярен для фотографий реального мира. Современные камеры телефонов создают JPEG, готовые к отправке или переносу на компьютер.

JPEG означает Joint Photographic Experts Group. В отличие от предыдущих методов, он основан на психовизуальных исследованиях и использует особенности человеческого зрения. В частности, JPEG может отбрасывать резкие переходы цвета, уменьшая объём данных для воспроизведения изображения. Для этого применяется довольно сложная математика.

Недостаток JPEG - необратимость: после сжатия нельзя точно восстановить оригинал. GIF и PNG обратимы, при них ничего не теряется. Поэтому GIF и PNG называют *сжатием без потерь*, а JPEG - *сжатием с потерями*. Информация исчезает, и в крайних случаях возникают заметные искажения.

Компьютеры часто имеют микрофон, воспринимающий звук реального мира, и динамик, создающий звук.

Звук - это вибрация. Вибрируют голосовые связки, туба, падающее в лесу дерево; они заставляют двигаться молекулы воздуха. Воздух попеременно давит и тянет, сжимается и разрежается сотни или тысячи раз в секунду. Затем он колеблет барабанные перепонки, и человек ощущает звук.

Микрофон отвечает на колебания электрическим током, напряжение которого изменяется подобно звуковым волнам.

Таким же аналогом звуковых волн были небольшие холмы и впадины на поверхности цилиндра из оловянной фольги, с помощью которого Томас Эдисон в 1877 году записывал и воспроизводил звук в первом фонографе, а позднее - холмы и впадины канавок виниловых пластинок, любимых современными аудиофилами и поклонниками ретротехники.

Но компьютеру напряжение надо *оцифровать*, то есть превратить в числа. Это ещё одна работа для ADC.

Цифровой звук громко заявил о себе на потребительском рынке в 1983 году с появлением компакт-диска CD, ставшего крупнейшим успехом потребительской электроники. Philips и Sony разработали CD для хранения 74 минут цифрового звука на одной стороне диска диаметром 12 сантиметров. По легенде, 74 минуты выбрали, чтобы на одном диске помещалась Девятая симфония Бетховена.

Звук CD кодируется импульсно-кодовой модуляцией PCM. Несмотря на внушительное название, идея проста: напряжение звуковой волны с постоянной частотой преобразуется в цифровые значения и сохраняется. При воспроизведении DAC снова превращает числа в электрический ток.

Скорость преобразования напряжения в числа называется *частотой дискретизации*. В 1928 году Гарри Найквист из Bell Telephone Laboratories показал, что она должна быть не меньше удвоенной максимальной частоты, которую требуется записать и воспроизвести.

Обычно считается, что человек слышит от 20 до 20 000 Гц. Частота CD чуть больше удвоенного максимума - 44 100 отсчётов в секунду.

Число бит на отсчёт определяет *динамический диапазон* CD - разницу между самым громким и самым тихим воспроизводимым звуком. Здесь всё несколько сложнее. Электрический ток колеблется по аналогии со звуковой волной, а достигаемые пики представляют её *амплитуду*. Воспринимаемая интенсивность звука пропорциональна удвоенной амплитуде.

Бел, название которого составляет три четверти фамилии Александра Грэма Белла, означает десятикратное увеличение интенсивности. Децибел - одна десятая бела. Примерно один децибел соответствует минимальному изменению громкости, которое способен заметить человек.

Шестнадцать бит на отсчёт дают динамический диапазон 96 децибел - приблизительно разницу между порогом слышимости, ниже которого звук не слышен, и болевым порогом, выше которого хочется закрыть уши. CD использует 16 бит на отсчёт.

Одна секунда CD содержит 44 100 двухбайтовых отсчётов. Для стерео это число удваивается: 176 400 байтов в секунду, или 10 584 000 байтов в минуту. Теперь понятно, почему цифровая запись не была обычным явлением до 1980-х. Полные 74 минуты стереозвука требуют 783 216 000 байтов. Поздние CD немного увеличили ёмкость.

Хотя значение CD в последние годы уменьшилось, принципы цифрового звука не изменились. Для домашней записи не всегда нужно качество CD, поэтому доступны меньшие частоты: 22 050, 11 025 и 8000 Гц. Можно использовать восьмибитный отсчёт и вдвое сократить данные, записывая моно.

Как и bitmap, звуковые файлы удобно сжимать, уменьшая занимаемое место и время передачи. Популярный метод MP3 возник как часть технологии сжатия кино MPEG, то есть Moving Picture Experts Group. MP3 выполняет сжатие с потерями, но опирается на психоакустический анализ и удаляет данные, почти не влияющие на восприятие музыки.

Bitmap, сжатые GIF, PNG или JPEG, и звук MP3 могут находиться в памяти, особенно пока программа работает с ними, но обычно они сохраняются в виде файлов на устройстве хранения.

RAM, построенная на реле, лампах или транзисторах, теряет содержимое при отключении питания. Поэтому полноценному компьютеру нужно долговременное хранилище.

Старый способ - пробивать отверстия в бумаге или картоне, как на перфокартах IBM. В ранних небольших компьютерах программы и данные сохранялись на рулонах перфоленты и позднее снова загружались в память. Следующим шагом стали аудиокассеты, популярные в 1980-х и для музыки. Они были уменьшенной версией магнитных лент больших компьютеров.

Но лента неудобна для хранения и поиска: быстро перейти к произвольному месту нельзя, перемотка занимает время.

Геометрически для быстрого доступа лучше подходит диск. Он вращается вокруг центра, а одна или несколько головок на рычагах перемещаются от внешнего края к внутреннему. К любой области можно обратиться быстро. Биты записываются намагничиванием небольших участков.

Первые компьютерные дисковые накопители создала IBM в 1956 году. Random Access Method of Accounting and Control (RAMAC) содержал 50 металлических дисков диаметром два фута и хранил пять мегабайт.

На персональных компьютерах стали популярны отдельные листы пластика с покрытием внутри защитного картонного или пластикового корпуса. Их называли *гибкими дисками* или *дискетами*. Диаметр последовательно уменьшался с 8 до 5,25 и 3,5 дюйма.

Дискеты вынимались из накопителя, поэтому переносили данные между компьютерами и служили важным средством распространения коммерческих программ. Теперь они почти исчезли, оставив после себя лишь небольшое изображение 3,5-дюймовой дискеты в значке Save многих приложений.

Жёсткий диск, всё ещё встречающийся внутри некоторых персональных компьютеров, обычно содержит несколько металлических дисков, постоянно встроенных в накопитель. Он быстрее дискеты и хранит больше данных, но сами диски нельзя легко извлечь.

Сегодня хранилищем чаще служит твердотельный накопитель SSD, встроенный в компьютер, планшет или телефон, либо флеш-память переносного USB-накопителя.

Устройства массовой памяти должны размещать файлы разного размера, поступающие из разных источников. Поэтому хранилище делится на области фиксированного размера, называемые *секторами*. У дискет и жёстких дисков сектор часто имел размер 512 байтов. У SSD распространены сектора по 512 и 4096 байтов.

Каждый файл занимает один или несколько секторов. При размере сектора 512 байтов файлу меньше 512 байтов нужен один сектор, но оставшееся место нельзя использовать для другого файла. Файлу из 513 байтов требуются два сектора, а файлу размером один мегабайт - 4096 секторов.

Сектора одного файла не обязаны идти подряд: они могут быть разбросаны по всему накопителю. После удаления файла сектора освобождаются для других данных. Новые файлы используют доступные сектора, но те не обязательно образуют непрерывную группу.

Отслеживанием всего этого, включая сохранение и извлечение файлов, занимается чрезвычайно важная программа - *операционная система*.

Глава 26. Операционная система

Наконец-то мы, хотя бы в воображении, собрали нечто похожее на полноценный компьютер. В нём есть центральный процессор CPU, оперативная память RAM, клавиатура, видеодисплей, чья память является частью RAM, и какое-либо устройство массового хранения. Всё оборудование готово, и мы с волнением смотрим на выключатель, который подаст питание и вдохнёт в машину жизнь. Возможно, проект напоминает труды Виктора Франкенштейна, собиравшего своего монстра, или Джеппетто, мастерившего деревянную куклу по имени Пиноккио.

Но чего-то всё ещё не хватает, и это не сила молнии и не чистота желания, загаданного звезде. Включите новый компьютер и расскажите, что видно.

Загоревшийся экран показывает чистый случайный мусор. Графический адаптер выводит бессвязные разноцветные точки, текстовый - случайные символы. Именно этого и следовало ожидать. Полупроводниковая память при отключённом питании теряет содержимое, а при первом включении оказывается в случайном, непредсказуемом состоянии.

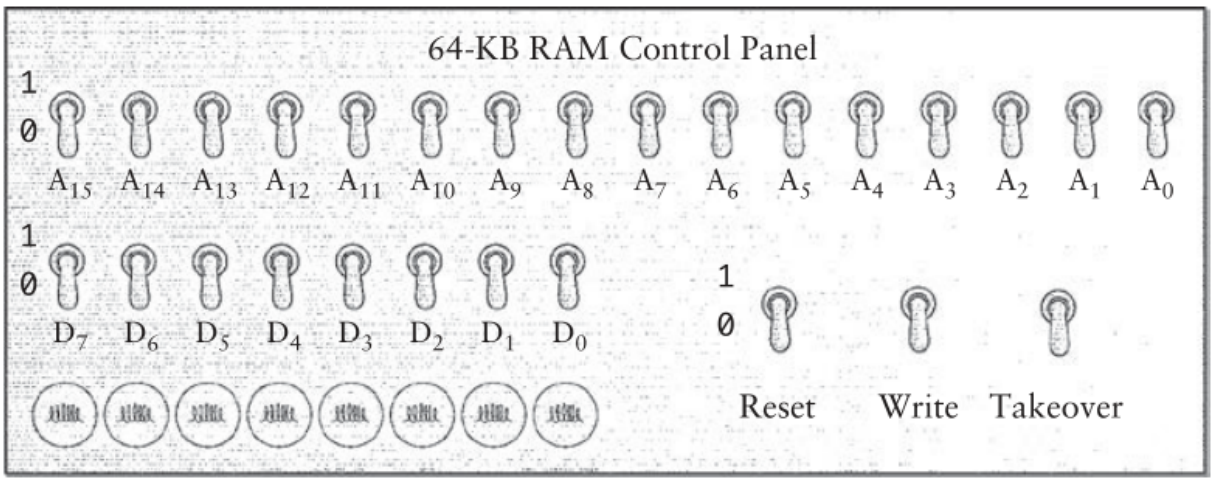
Вся RAM микропроцессора заполнена случайными байтами. Микропроцессор начинает исполнять их так, словно это машинный код. Ничего опасного не произойдёт, компьютер не взорвётся, но и пользы от такой работы не будет.

Причина предсказуема: в полупроводниковой памяти нет заранее заданного начального состояния. Цветные точки или случайные символы экрана - лишь видимое проявление тех произвольных байтов, которые CPU тем временем принимает за инструкции.

Не хватает *программного обеспечения*. При включении или сбросе микропроцессор не ждёт указаний человека, а сразу начинает выполнять машинный код с заранее определённого адреса. Для Intel 8080 это адрес 0000h, поэтому именно содержимое этой ячейки определяет первое осмысленное действие компьютера.

В правильно сконструированном компьютере по этому адресу должна находиться машинная инструкция, скорее всего первая из многих, которую CPU исполнит после включения.

Но как она туда попадёт? Загрузка программ в только что спроектированный компьютер - одна из самых запутанных частей проекта. Один способ - панель управления, похожая на использованную в главе 19 для записи байтов в RAM и последующего чтения:



В отличие от прежней панели здесь есть переключатель Reset, соединённый со входом Reset CPU. Пока он включён, микропроцессор ничего не делает. После выключения переключателя процессор начинает исполнять код с адреса 0000h.

Для работы с панелью сначала включают Reset, сбрасывая микропроцессор и останавливая исполнение. Затем включают Takeover, чтобы перехватить шины адреса и данных. Переключатели A0-A15 задают шестнадцатибитный адрес памяти, а лампы D0-D7 показывают восьмибитное содержимое этой ячейки.

Чтобы записать новый байт, его устанавливают переключателями D0-D7 и включают, а затем выключают Write. Закончив ввод байтов, отключают Takeover и Reset, после чего микропроцессор исполняет программу.

Так вводят первые программы машинного кода в компьютер, собранный с нуля. Да, это невыносимо трудоёмко. Ошибки неизбежны. Волдыри на пальцах и превращение мозга в кашу - профессиональные риски.

Каждый адрес и каждый байт приходится задавать вручную. Лампы позволяют проверить уже записанное содержимое, но не избавляют от необходимости тщательно следить за адресом, данными и последовательностью переключений. Единственная ошибка в одном бите может изменить инструкцию или её операнд.

Но всё окупается, когда видеодисплей начинает показывать результаты небольших программ. Одной из первых станет подпрограмма преобразования чисел в ASCII.

Если результат программы равен 4Bh, нельзя просто записать это значение в видеопамять: на экране появится буква K, которой соответствует ASCII 4Bh. Нужно вывести два символа ASCII.

Это 34h, ASCII-код 4, и 42h, ASCII-код B. Подобный код уже встречался: подпрограмма ToAscii на странице 398 главы 24.

Одной из первых задач наверняка станет избавление от нелепой панели управления. Для этого нужен *обработчик клавиатуры* - программа, читающая введённые символы, сохраняющая их в памяти и одновременно выводящая на экран. Перенос символов с клавиатуры на экран иногда называют *эхом*: он создаёт иллюзию прямого соединения клавиатуры и дисплея.

Обработчик можно расширить до исполнения простых *команд*, то есть дать ему полезную работу. Код, обрабатывающий команды, называется *командным процессором*. Сначала ограничимся тремя командами, определяемыми первой буквой строки:

• W - Write, записать;

• D - Display, показать;

• R - Run, запустить.

Обработчик выполняет команду после нажатия Enter, сообщающего об окончании ввода.

До Enter обработчик лишь собирает и отображает символы введённой строки. После Enter командный процессор рассматривает первый символ как название операции, а следующие группы ASCII-цифр - как шестнадцатеричные адреса и байты. Таким образом, панельные переключатели заменяются текстовым языком управления компьютером.

Строка с W означает запись байтов в память:

```
W 1020 35 4F 78 23 9B AC 67
```

Командный процессор записывает шестнадцатеричные байты 35h, 4Fh и остальные в память начиная с адреса 1020h. Для этого обработчик клавиатуры должен преобразовать коды ASCII в байты - выполнить действие, обратное ToAscii.

Каждая пара набранных шестнадцатеричных символов должна превратиться в одно восьмибитное значение. Сам адрес 1020 тоже вводится как четыре символа ASCII и преобразуется в шестнадцатибитный адрес перед записью.

Строка с D означает вывод байтов памяти:

```
D 1030
```

Процессор отвечает отображением байтов, хранящихся начиная с 1030h. Команда Display позволяет исследовать содержимое памяти.

Строка с R означает запуск:

```
R 1000
```

Она говорит: "Запустить программу, хранящуюся начиная с адреса 1000h". Командный процессор может поместить 1000h в пару HL.

Таким образом, три команды вместе воспроизводят основные возможности прежней панели: Write вводит программу, Display проверяет память, а Run передаёт введённому коду управление.

Затем он исполняет PCHL, загружающую счётчик команд из HL и фактически переходящую по этому адресу.

Работающий обработчик клавиатуры с командным процессором - важная веха. После неё больше не приходится терпеть унижение панелью управления. Вводить данные с клавиатуры проще, быстрее и элегантнее.

Однако весь введённый код по-прежнему исчезает при отключении питания. Поэтому его стоит сохранить в постоянной памяти ROM. В первые годы микропроцессоров вроде Intel 8080 появилась возможность программировать микросхемы ROM дома. Programmable Read-Only Memory (PROM) программируется один раз. Erasable Programmable Read-Only Memory (EPROM) можно программировать повторно после полного стирания ультрафиолетом.

ROM с обработчиком клавиатуры займёт адресное пространство с 0000h, ранее принадлежавшее RAM. Сама RAM останется, но переместится на более высокие адреса.

Командный процессор важен не только как быстрый способ вводить байты: теперь компьютер стал *интерактивным*.

Пользователь больше не задаёт машине заранее фиксированную последовательность действий переключателями. Он вводит команды, получает ответы и выбирает следующий шаг на основании увиденного результата.

Имея командный процессор в ROM, можно экспериментировать с записью данных из памяти на диск и их чтением обратно. Хранить программы и данные на диске гораздо надёжнее, чем в RAM, где они исчезнут при сбое питания, и гибче, чем в ROM.

Со временем в командный процессор добавятся новые команды. Например, S может означать Store - сохранить область памяти в определённой группе дисковых секторов, а L - Load, загрузить содержимое этих секторов в память.

Придётся отслеживать, что и в каких секторах хранится, вероятно, с блокнотом и карандашом. Нельзя просто сохранить код, находившийся по одному адресу, а затем загрузить по другому и ожидать работы: все Jump и Call будут содержать старые адреса.

Кроме того, программа может быть длиннее дискового сектора, и тогда её надо разместить в нескольких секторах. Часть секторов уже занята другими программами и данными, поэтому свободные места для длинной программы могут идти не подряд.

Рано или поздно ручной канцелярский учёт расположения данных на диске станет слишком обременительным. Значит, пришло время *файловой системы*.

Файловая система - программа, организующая данные в *файлы*. Файл представляет собой набор связанных данных, занимающий один или несколько секторов. Главное - у каждого файла есть *имя*, помогающее запомнить его содержимое.

Диск можно представить картотечным шкафом, где на каждом деле есть небольшая вкладка с именем.

Файловая система почти всегда входит в более крупный набор программ, называемый *операционной системой*. Создаваемые обработчик клавиатуры и командный процессор могли бы постепенно превратиться в ОС. Но вместо долгого эволюционного пути рассмотрим настоящую систему и разберёмся, что она делает.

Исторически важнейшей ОС для восьмибитных микропроцессоров была CP/M. Сначала сокращение означало Control Program/Monitor, позднее - Control Program for Microcomputers. Гэри Килдалл (1942-1994), будущий основатель Digital Research Incorporated (DRI), написал её в середине 1970-х для Intel 8080.

CP/M хранилась на диске, большая часть которого оставалась для пользовательских файлов. Её файловая система проста, но удовлетворяет двум главным требованиям.

Во-первых, каждый файл на диске имеет имя, также сохранённое на диске. Во-вторых, файл не обязан занимать последовательные сектора. По мере создания и удаления файлов свободное место часто становится фрагментированным, поэтому способность хранить большой файл в разрозненных секторах очень полезна. Таблица соответствия файлов секторам тоже находится на диске.

В CP/M имя состоит из двух частей. Первая - *имя файла* длиной до восьми символов. Вторая - *тип* или *расширение* длиной до трёх символов. Некоторые типы стандартны: TXT означает текстовый файл, содержащий только читаемые человеком коды ASCII; COM, сокращение от command, - файл с машинными инструкциями 8080, то есть программу.

Соглашение получило название 8.3, произносимое "восемь точка три": до восьми букв перед точкой и три после неё. Современные файловые системы сняли эти ограничения, но общий принцип именования остался распространённым.

В компьютерах CP/M небольшой код в ROM назывался *bootstrap loader*, или загрузчиком начальной загрузки: он словно вытягивал остальную ОС за собственные ремешки ботинок. Загрузчик читает первый сектор дискеты в память и запускает его. Тот сектор содержит код, загружающий остальную CP/M. Весь процесс называется *booting*, загрузкой операционной системы; термин используется до сих пор.

CP/M организована иерархически. На нижнем уровне находится Basic Input/Output System, BIOS, произносимое "бай-ос". BIOS содержит код прямой работы с оборудованием, включая чтение и запись секторов. Каждый производитель компьютера с CP/M поставлял собственную BIOS для конкретного набора аппаратуры.

Следующий уровень - Basic Disk Operating System, BDOS, произносимое "би-дос". Его главная задача - организовать дисковые сектора, обслуживаемые BIOS, в файлы.

Завершив загрузку в память, CP/M запускает Console Command Processor (CCP), который выводит приглашение:

```
A>
```

В компьютере с несколькими накопителями A обозначает первый диск, с которого загружена CP/M. Приглашение означает, что можно набрать команду и нажать Enter.

Большинство команд работает с файлами: DIR, от directory, перечисляет их; ERA удаляет; REN переименовывает; TYPE показывает содержимое. Имя, которое CP/M не распознаёт как встроенную команду, считается именем программы на диске.

CP/M содержит и подпрограммы, с помощью которых программы читают клавиатуру, выводят символы на дисплей, сохраняют данные в дисковый файл и загружают его обратно в память.

Программам CP/M не надо напрямую обращаться к аппаратуре: BDOS использует BIOS. Поэтому одна программа CP/M может работать на любом компьютере с CP/M, ничего не зная о его внутреннем оборудовании. Этот принцип называется *независимостью от устройств* и был решающим для развития коммерческого ПО. Позднее такие программы стали называть *приложениями*, или *apps*.

Набор подпрограмм операционной системы называется *интерфейсом программирования приложений*, API. В идеальном мире разработчику приложения достаточно знать API, а не его реализацию и оборудование, к которому он обращается. На практике дополнительные знания иногда полезны.

Для пользователя операционная система - это *пользовательский интерфейс*, UI. В CP/M им был интерфейс командной строки CLI, реализованный CCP. Для программиста ОС одновременно является API - набором подпрограмм, доступных приложению.

Подпрограммы CP/M имели общую точку входа по адресу 0005h. Программа обращалась к одной из них вызовом:

```
CALL 0005h
```

или просто:

```
CALL 5
```

Это называлось интерфейсом "Call 5". Конкретная функция задавалась значением регистра C.

Несколько примеров функций:

Регистр C	**Функция CP/M Call 5**
01h	Console Input - прочитать символ с клавиатуры
02h	Console Output - вывести символ на дисплей
09h	Print String - вывести строку символов
15h	Open File - открыть существующий файл
16h	Close File - закончить работу с файлом
20h	Read Sequential - последовательно читать байты файла
21h	Write Sequential - последовательно записывать байты файла
22h	Make File - создать новый файл

Часто функции нужны дополнительные сведения. Например, когда C равен 09h, пара DE содержит адрес символов ASCII, выводимых на экран. Знак доллара $ отмечает конец строки.

Что в действительности делает CALL 5? CP/M помещает по адресу 0005h инструкцию JMP. Она переходит в часть BDOS, которая проверяет C и вызывает соответствующую подпрограмму.

Когда-то CP/M была очень популярной системой для 8080 и остаётся исторически важной. Она сильно повлияла на шестнадцатибитную QDOS, Quick and Dirty Operating System, написанную Тимом Патерсоном из Seattle Computer Products для шестнадцатибитных Intel 8086 и 8088.

Позднее QDOS переименовали в 86-DOS и лицензировали Microsoft. Под названием MS-DOS, Microsoft Disk Operating System, система была лицензирована IBM для первого IBM Personal Computer, представленного в 1981 году. Для IBM PC существовала и шестнадцатибитная CP/M-86, но стандартом быстро стала MS-DOS.

MS-DOS, на компьютерах IBM называвшаяся PC-DOS, лицензировалась и другим производителям IBM PC-совместимых машин.

Как следует из названия, MS-DOS прежде всего дисковая операционная система, как и Apple DOS 1978 года для Apple II. Помимо записи файлов на диск и последующего чтения, она предлагала немногое.

Теоретически приложения должны обращаться к оборудованию только через интерфейсы ОС. Но программисты 1970-х и 1980-х часто обходили ОС, особенно при работе с дисплеем. Прямая запись в видеопамять была быстрее, а для графических приложений средства ОС вообще не годились. Многим нравилось, что ранние системы "не мешали" и позволяли программам работать с предельной скоростью аппаратуры.

Первым признаком того, что домашние компьютеры будут сильно отличаться от больших и дорогих собратьев, вероятно, стало приложение VisiCalc. Дэн Бриклин (род. 1951) и Боб Фрэнкстон (род. 1949) разработали его для Apple II и выпустили в 1979 году. VisiCalc давал на экране двумерное представление электронной таблицы.

До него spreadsheet был широким листом бумаги со строками и столбцами для последовательности вычислений. VisiCalc заменил бумагу видеодисплеем: пользователь перемещался по таблице, вводил числа и формулы, а после изменения всё пересчитывалось.

Удивительно, что приложение вроде VisiCalc нельзя было воспроизвести на больших компьютерах. Ему требовалось очень быстро обновлять экран, поэтому оно напрямую писало в RAM видеодисплея Apple II, являвшуюся частью адресного пространства микропроцессора. Большие компьютеры были устроены и работали иначе.

Чем быстрее компьютер отвечает на клавиатуру и изменяет экран, тем теснее взаимодействие пользователя с машиной. Большинство программ первого десятилетия персональных компьютеров, до конца 1980-х, писали прямо в видеопамять.

IBM установила аппаратный стандарт, которому следовали другие производители, поэтому разработчики могли обходить ОС и обращаться к оборудованию, не опасаясь несовместимости. Если бы у всех клонов PC были разные интерфейсы дисплея, производителям ПО было бы слишком трудно поддерживать все варианты.

Но с ростом числа приложений возникли проблемы. Самые успешные занимали весь экран и создавали сложный клавиатурный UI, причём каждое понимало интерфейс по-своему. Навыки работы с одной программой не переносились на другую. Программы плохо сосуществовали: для перехода обычно приходилось завершить одну и запустить следующую.

Совсем другое видение персональных вычислений несколько лет развивалось в Palo Alto Research Center (PARC), основанном Xerox в 1970 году отчасти для создания продуктов, которые позволили бы компании войти в компьютерную индустрию.

Первым крупным проектом PARC стал Alto, спроектированный и построенный в 1972-1973 годах. По меркам того времени это была впечатляющая машина. Напольный системный блок имел шестнадцатибитную обработку, два диска по 3 МБ и 128 КБ памяти с расширением до 512 КБ. Alto появился раньше однокристальных шестнадцатибитных микропроцессоров, поэтому CPU пришлось собрать примерно из 200 интегральных схем.

Одной из необычных частей Alto был дисплей. По размеру и форме экран напоминал лист бумаги: около восьми дюймов шириной и десяти высотой. Графический режим имел 606 пикселей по горизонтали.

По вертикали было 808 пикселей, всего 489 648. На пиксель приходился один бит памяти, поэтому он мог быть чёрным или белым. Видеопамять занимала 64 КБ адресного пространства процессора.

Записывая в неё, программа рисовала изображения или текст разных шрифтов и размеров. Дисплей перестал лишь повторять набранное на клавиатуре: он стал плотным двумерным массивом информации и более непосредственным источником пользовательского ввода.

Alto имел и небольшое устройство - *мышь*, катавшуюся по столу и снабжённую тремя кнопками. Её изобрёл инженер Дуглас Энгельбарт (1925-2013) в Stanford Research Center. Двигая мышь, пользователь Alto перемещал указатель и взаимодействовал с объектами на экране.

В оставшиеся годы 1970-х программы Alto приобрели интересные черты. Несколько программ помещались в окна и одновременно отображались на одном экране. Графика позволила ПО выйти за пределы текста и отражать воображение пользователя. Графические объекты, кнопки, меню и маленькие картинки, называемые *значками*, или *icons*, стали частью UI. Мышь выбирала окна и запускала действия объектов.

Это ПО вышло за рамки пользовательского интерфейса к близости с пользователем. Оно расширяло компьютер за пределы простого счёта и, если процитировать название статьи Энгельбарта 1963 года, было предназначено "для усиления человеческого интеллекта".

Alto положил начало *графическому пользовательскому интерфейсу*, GUI, часто произносимому "гуи". Значительная часть новаторской концептуальной работы приписывается Алану Кею (род. 1940). Но Xerox не продавала Alto - машина стоила бы больше 30 000 долларов, - и прошло более десяти лет, прежде чем её идеи воплотились в успешном потребительском продукте.

В 1979 году Стив Джобс с группой Apple Computer посетил PARC и был впечатлён увиденным. Но компьютер с графическим интерфейсом появился у Apple только спустя более трёх лет: неудачная Lisa в январе 1983 года. Годом позже вышел гораздо более успешный Macintosh.

Первый Macintosh имел Motorola 68000, 64 КБ ROM с операционной системой, 128 КБ RAM, дисковод для 3,5-дюймовых дискет по 400 КБ, клавиатуру, мышь и дисплей 512 x 342 пикселя с диагональю лишь девять дюймов.

Всего получалось 175 104 пикселя. Каждому соответствовал один бит, то есть чёрный или белый цвет, поэтому видеопамяти требовалось около 22 КБ.

Аппаратура первого Macintosh была элегантной, но едва ли революционной. Mac отличала операционная система, в 1984 году обычно называвшаяся system software, позднее Mac OS, а теперь macOS.

Однопользовательская текстовая система вроде CP/M, MS-DOS или Apple DOS невелика, а большая часть API обслуживает файлы. Графическая система вроде macOS гораздо крупнее и содержит сотни функций API. Каждая имеет имя, описывающее действие.

Текстовая MS-DOS предоставляет несколько простых функций вывода текста на экран в манере телетайпа. Графическая macOS должна позволять приложениям рисовать. Теоретически достаточно одной функции установки цвета пикселя с заданными горизонтальной и вертикальной координатами, но это неэффективно и создаёт очень медленную графику.

Разумнее предоставить полноценную графическую систему с API для линий, прямоугольников, кривых и текста. Линии бывают сплошными, штриховыми или точечными. Прямоугольники и эллипсы заполняются узорами. Текст выводится разными шрифтами и размерами, жирным или подчёркнутым. Графическая система сама превращает объекты в набор точек дисплея.

Это существенно сокращает работу каждого приложения. Вместо собственных алгоритмов для каждой линии, кривой, буквы и заполненной фигуры программа вызывает именованную функцию ОС и передаёт ей координаты, размеры, цвет и прочие параметры. Ответственность за конкретное размещение пикселей остаётся у общей графической подсистемы.

Приложения графической ОС используют одинаковые API для рисования и на экране, и на принтере. Текстовый процессор показывает документ почти таким же, каким он будет напечатан. Это свойство называется WYSIWYG, произносимое "визивиг", сокращение от What You See Is What You Get - "что видишь, то и получишь". Выражение вошло в компьютерный жаргон благодаря комику Флипу Уилсону и его персонажу Джеральдине.

Привлекательность GUI отчасти в том, что интерфейсы разных приложений похожи и используют уже накопленный опыт пользователя. Поэтому ОС предоставляет API для кнопок, меню и других компонентов UI.

GUI обычно считают удобной средой для пользователя, но он столь же важен как лучшая среда для программистов: современный интерфейс можно создать, не изобретая колесо заново.

Общие функции интерфейса обеспечивают и единообразие поведения. Кнопки, меню и другие элементы в разных программах выглядят и реагируют похоже именно потому, что приложения строят их через одни и те же средства операционной системы.

Ещё до Macintosh несколько компаний разрабатывали графические ОС для IBM PC и совместимых компьютеров. В некотором смысле Apple было проще, поскольку она одновременно проектировала аппаратуру и ПО. Системе Macintosh требовалось поддерживать только один дисковод, один тип дисплея и два принтера.

Графическая ОС для PC должна была поддерживать множество видов оборудования.

Кроме того, хотя IBM PC появился лишь в 1981 году, пользователи уже привыкли к любимым приложениям MS-DOS и не хотели от них отказываться. Поэтому считалось очень важным, чтобы новая графическая ОС запускала и программы MS-DOS, и специально созданные графические приложения. Macintosh не запускал ПО Apple II главным образом из-за другого микропроцессора.

В 1985 году Digital Research, компания CP/M, выпустила GEM, Graphical Environment Manager; VisiCorp, продвигавшая VisiCalc, - VisiOn; Microsoft - Windows 1.0, быстро признанную вероятным победителем "войн окон".

Однако значительное число пользователей Windows получила только после выхода версии 3.0 в мае 1990 года. В конце концов она стала доминирующей ОС настольных компьютеров и ноутбуков. Несмотря на внешнее сходство Macintosh и Windows, API этих систем совершенно различны.

У телефонов и планшетов иная история. В графических интерфейсах телефонов, планшетов и больших ПК много общего, но API отличаются. Сейчас на рынке телефонов и планшетов доминируют системы Android и Apple.

Хотя большинству пользователей это почти не видно, наследие и влияние UNIX остаётся мощным. UNIX разработали в начале 1970-х в Bell Telephone Laboratories главным образом Кен Томпсон (род. 1943) и Деннис Ритчи (1941-2011), обладатели одних из лучших бород в компьютерной индустрии.

Забавное имя - игра слов. UNIX задумывалась как менее тяжеловесная версия прежней Multics, от Multiplexed Information and Computing Services, которую Bell Labs совместно создавала с MIT и GE.

Среди увлечённых программистов UNIX, вероятно, самая любимая ОС. Большинство систем пишется для определённых компьютеров, а UNIX проектировалась как *переносимая*, то есть приспосабливаемая к разным машинам.

Bell Labs тогда была подразделением American Telephone & Telegraph и подчинялась судебным постановлениям, ограничивавшим телефонную монополию AT&T. Сначала AT&T запрещалось продавать UNIX, и компания была обязана лицензировать её другим.

Начиная с 1973 года UNIX широко лицензировалась университетам, корпорациям и государственным организациям. В 1983 году AT&T снова разрешили заниматься компьютерами, и она выпустила собственную UNIX.

В результате единой версии UNIX нет. Существует множество вариантов с разными именами для разных компьютеров и поставщиков. Многие оставили на UNIX свои отпечатки.

Тем не менее добавление новых частей направляет распространённая "философия UNIX". Один из её принципов - использовать текстовые файлы как общий знаменатель. Множество небольших программ командной строки UNIX, называемых *утилитами*, читает текстовый файл, обрабатывает его и записывает другой. Утилиты соединяются в цепочки, выполняющие разные виды обработки текста.

Самым интересным развитием UNIX последних лет стали Free Software Foundation (FSF) и проект GNU, основанные Ричардом Столлманом (род. 1953). GNU произносится не как название животного, а с отчётливым `G` в начале, и означает "GNU's Not UNIX". Разумеется, GNU не UNIX.

GNU предназначен для совместимости с UNIX, но распространяется так, чтобы не позволить ПО стать проприетарным. Проект создал множество UNIX-совместимых утилит и инструментов, а также Linux - ядро UNIX-совместимой ОС.

Linux, написанный главным образом финским программистом Линусом Торвальдсом (род. 1969), стал очень популярен. Android основан на ядре Linux, крупные суперкомпьютеры используют Linux почти исключительно, и Linux широко распространён на интернет-серверах.

Но интернет - тема последней главы этой книги.

Глава 27. Программирование

Все компьютеры исполняют машинный код, но программировать в машинном коде - всё равно что есть зубочисткой. Кусочки настолько малы, а процесс настолько трудоёмок, что ужин длится бесконечно. Байты машинного кода выполняют мельчайшие и простейшие мыслимые вычислительные задачи: загружают число из памяти в процессор, прибавляют другое, сохраняют результат обратно. Трудно представить, как из них получается целая трапеза.

По крайней мере, мы уже покинули первобытную эпоху начала предыдущей главы, когда двоичные данные вводились переключателями панели. Затем появились простые программы, позволяющие через клавиатуру и дисплей вводить и исследовать шестнадцатеричные байты машинного кода. Это гораздо лучше, но улучшения на этом не заканчиваются.

Байтам машинного кода соответствуют короткие мнемоники MOV, ADD, JMP, HLT и другие, позволяющие обозначать команды чем-то отдалённо похожим на английский. Вместе с мнемоникой записываются операнды, уточняющие действие.

Например, байт 46h процессора 8080 переносит в B байт по адресу памяти, заданному шестнадцатибитной парой HL. Краткая запись выглядит так:

```
MOV B,M
```

Здесь M означает memory, память. Полный набор мнемоник с дополнительными средствами образует язык программирования, называемый *языком ассемблера*. На нём писать программы намного проще, чем в машинном коде. Единственная проблема: CPU не понимает язык ассемблера непосредственно.

В первые дни работы с таким примитивным компьютером программист, вероятно, долго писал бы программы на бумаге. Лишь убедившись, что замысел может сработать, он *ассемблировал бы вручную*: по таблице или справочнику преобразовал операторы ассемблера в байты машинного кода и ввёл их в память.

Особенно трудны переходы и вызовы. Для ручного JMP или CALL надо знать точный двоичный адрес назначения, который зависит от размещения всех остальных инструкций. Гораздо лучше поручить преобразование компьютеру. Но как?

Адрес метки нельзя окончательно вычислить, пока неизвестна длина всех предшествующих инструкций. Если позднее добавить или удалить хотя бы одну команду, адреса следующих меток изменятся, и соответствующие байты каждого перехода или вызова придётся исправлять заново. Именно эта механическая работа особенно хорошо подходит компьютеру.

Сначала можно написать *текстовый редактор* - программу для ввода строк и сохранения их в файл. К сожалению, сам редактор пришлось бы ассемблировать вручную. Затем с его помощью создаются текстовые файлы с инструкциями ассемблера.

Потребуется вручную собрать и другую программу - *ассемблер*. Он читает текстовый файл с инструкциями, преобразует их в машинный код и сохраняет в другом файле. Содержимое результата загружается в память и исполняется.

Таким образом, исходная программа остаётся удобным для человека текстом, а второй файл содержит форму, понятную CPU. Редактор отвечает только за ввод и изменение текста; ассемблер - за перевод, вычисление адресов и выдачу байтов.

В CP/M для 8080 большая часть инструментов уже готова. Редактор ED.COM создаёт и изменяет текстовые файлы. Современные простые аналоги - Notepad в Windows и TextEdit в macOS.

Допустим, создан файл `PROGRAM1.ASM`. Расширение ASM означает, что в нём программа на языке ассемблера:

```
       ORG 0100h
       LXI DE,Text
       MVI C,9
       CALL 5
       RET
Text:  DB 'Hello!$'
       END
```

Здесь есть новые операторы. ORG, от Origin, не соответствует инструкции 8080. Он указывает, что следующий оператор начинается по адресу 0100h, куда CP/M загружает программы.

LXI, Load Extended Immediate, загружает шестнадцатибитное значение в DE. Это одна из нескольких инструкций 8080, не реализованных в построенном CPU. Здесь значением служит метка Text.

Метка расположена ниже перед DB, Data Byte, ещё одним новым оператором. После DB могут идти несколько байтов через запятые или, как здесь, текст в одинарных кавычках.

MVI переносит 9 в C. CALL 5 обращается к CP/M, которая смотрит значение C и переходит к нужной функции. Функция выводит строку, начинающуюся по адресу из DE, и заканчивает на знаке доллара. Поэтому последняя строка текста содержит $. Использовать доллар как конец строки странно, но так работает CP/M.

RET завершает программу и возвращает управление CP/M - это один из нескольких способов закончить программу CP/M. END отмечает конец исходного файла ассемблера.

Итак, есть текстовый файл из семи строк. Следующий шаг - сборка. CP/M содержит ассемблер ASM.COM, запускаемый из командной строки:

```
ASM PROGRAM1.ASM
```

ASM исследует `PROGRAM1.ASM` и создаёт `PROGRAM1.COM` с соответствующим машинным кодом. На самом деле процесс включает ещё один этап, несущественный для объяснения.

Файл COM содержит 16 байтов:

```
11 09 01 0E 09 CD 05 00 C9 48 65 6C 6C 6F 21 24
```

Первые три байта - LXI, следующие два - MVI, затем три байта CALL и один RET. Последние семь - ASCII-коды пяти букв `Hello`, восклицательного знака и доллара.

Программа запускается командой:

```
PROGRAM1
```

Операционная система загружает её в память и исполняет. На экране появляется:

```
Hello!
```

Ассемблер читает программу на ассемблере, часто называемую *файлом исходного кода*, и создаёт файл машинного кода - *исполняемый файл*.

В общей картине ассемблер сравнительно прост, поскольку мнемоники почти однозначно соответствуют машинным инструкциям. Он разделяет каждую строку на мнемонику и аргументы и сравнивает маленькие слова, буквы и числа с поддерживаемым списком.

Этот процесс называется *синтаксическим разбором*, или *parsing*. В машинном коде он включает множество CMP с последующими условными переходами. Сравнения определяют машинную инструкцию для каждого оператора.

PROGRAM1.COM начинается с 11h - кода LXI. Далее идут 09h и 01h, образующие адрес 0109h. Ассемблер вычисляет его сам: если LXI находится по 0100h, как после загрузки CP/M, то строка Text начинается по 0109h. Пользователь ассемблера обычно не беспокоится о конкретных адресах частей программы.

Адрес записан младшим байтом прежде старшего, поэтому в файле идут 09h и 01h. Ассемблер одновременно считает размеры LXI, MVI, CALL и RET и знает, что после первых девяти байтов машинных инструкций начинается строка. Метка Text заменяется вычисленным числом автоматически.

Первому автору первого ассемблера, конечно, пришлось собрать его вручную. Новый, возможно улучшенный ассемблер для того же компьютера можно написать на ассемблере и собрать прежней программой. После этого новый ассемблер способен ассемблировать сам себя.

Для каждого нового микропроцессора нужен новый ассемблер. Но сначала его можно написать на существующем компьютере, пользуясь его инструментами. Это *кросс-ассемблер*: программа работает на компьютере A, но создаёт код для компьютера B.

Ассемблер устраняет наименее творческую часть работы - ручное преобразование, - однако у языка остаются две большие проблемы.

Первая очевидна: программирование крайне утомительно. Работа ведётся на уровне CPU, и приходится заботиться о каждой мелочи.

Вторая - непереносимость. Программа Intel 8080 не работает на Motorola 6800, её надо переписать на ассемблере 6800. Это легче исходной разработки, поскольку организационные и алгоритмические задачи уже решены, но работы всё равно много.

Мнемоники, регистры, способы адресации и даже длины команд у процессоров различаются. Поэтому недостаточно заменить несколько кодов операций: приходится приспособить последовательность шагов к архитектуре другого CPU.

Большая часть вычислений математическая, а математика на ассемблере выражается неуклюже. Гораздо удобнее привычная алгебраическая запись:

```
Angle = 27.5
Hypotenuse = 125.2
Height = Hypotenuse x Sine(Angle)
```

Если это часть программы, каждая строка называется *оператором*. Имена Angle, Hypotenuse и Height - *переменные*, поскольку получают разные значения. Знак равенства означает *присваивание*: Angle становится 27,5, Hypotenuse - 125,2. Sine - функция.

Где-то существует код, вычисляющий тригонометрический синус угла и возвращающий значение.

Эти числа не похожи на целые, обычные для ассемблера: у них есть десятичная точка и дробная часть. В компьютерной терминологии это *числа с плавающей точкой*.

Если подобные операторы находятся в текстовом файле, можно написать программу на ассемблере, которая прочитает файл и преобразует выражения в машинный код вычисления. Почему бы и нет?

Так возникает *язык программирования высокого уровня*. Ассемблер считается низкоуровневым, потому что близок к аппаратуре. Высокоуровневыми называют все остальные языки, хотя одни выше других.

Если бы президент компании мог сказать компьютеру: "Подсчитай все прибыли и убытки за год, составь годовой отчёт, напечатай пару тысяч экземпляров и разошли акционерам", это был бы действительно очень высокий уровень. Реальные языки до такого идеала не доходят.

Уровень языка определяется не достоинством и не сложностью задач, а расстоянием от конкретных аппаратных команд. Чем выше язык, тем больше подробностей выбора регистров, адресов и отдельных машинных шагов скрывает его реализация.

Человеческие языки возникли из тысяч лет сложных влияний, случайных изменений и приспособлений. Даже искусственный эсперанто выдаёт происхождение из живых языков. Компьютерные языки проектируются гораздо сознательнее. Изобретать их привлекательно, потому что язык определяет способ, которым человек даёт указания компьютеру.

В первом издании книги приводилась оценка 1993 года: с начала 1950-х было придумано и реализовано более тысячи высокоуровневых языков. На конец 2021 года Online Historical Encyclopedia of Programming Languages, hopl.info, насчитывала 8945.

Но определить язык и его *синтаксис*, позволяющий выразить все нужные действия, недостаточно. Нужен *компилятор* - программа, превращающая высокоуровневые операторы в машинный код.

Как ассемблер, компилятор читает исходный файл посимвольно и разбивает его на короткие слова, знаки и числа. Но компилятор намного сложнее. Ассемблеру помогает почти однозначное соответствие оператора машинной инструкции; один высокоуровневый оператор обычно превращается во множество машинных команд. Проектированию компиляторов посвящены целые книги.

Компилятор должен понимать структуру выражений, типы значений, порядок операций, функции, условия и циклы, а затем выбрать подходящую последовательность машинных инструкций. Он также сообщает о синтаксических ошибках, когда исходный текст не соответствует правилам языка.

Высокоуровневые языки имеют достоинства и недостатки. Главное достоинство - обычно их проще изучать.

На них легче программировать, а программы яснее и короче. Они часто *переносимы*, то есть не зависят от конкретного процессора. Программисту не обязательно знать внутреннее устройство машины. Для запуска на разных CPU всё равно нужны компиляторы, создающие машинный код каждого процессора: исполняемые файлы остаются специфичными.

С другой стороны, хороший программист на ассемблере часто пишет более быстрый и эффективный код, чем компилятор. Исполняемый файл высокоуровневой программы может быть больше и медленнее функционально равной ассемблерной. В последние годы различие менее очевидно: микропроцессоры усложнились, а компиляторы научились лучше оптимизировать код.

Компромисс состоит в удобстве и переносимости с одной стороны и полном контроле над процессором с другой. Высокоуровневый исходный текст компилируется заново для каждого CPU, но остаётся почти тем же; вручную оптимизированный ассемблер тесно связан с одной машиной.

Высокоуровневый язык делает процессор удобнее, но не мощнее. Некоторые языки не поддерживают обычные операции CPU, например сдвиг и проверку битов, поэтому такие задачи могут усложняться.

В первые годы домашних компьютеров большинство приложений писали на ассемблере. Теперь он используется редко и для особых целей. Конвейерное исполнение нескольких инструкций одновременно усложнило процессоры и ассемблерное программирование, а компиляторы стали совершеннее. Большие объёмы памяти и накопителей тоже сняли необходимость умещать код в крошечной RAM и на маленькой дискете.

На сложном современном CPU очевидная вручную написанная последовательность не всегда оказывается самой быстрой: конвейеры и другие аппаратные средства меняют стоимость команд. Оптимизирующий компилятор способен учитывать такие особенности систематически.

Разработчики ранних компьютеров пытались задавать задачи алгебраически, но первым действительно работающим компилятором обычно считают Arithmetic Language version 0, A-0, созданный для UNIVAC Грейс Мюррей Хоппер (1906-1992) в Remington-Rand в 1952 году.

Доктор Хоппер ввела термин "compiler". Ещё в 1944 году она работала с Говардом Эйкеном над Mark I, а в восьмидесятилетнем возрасте продолжала деятельность в компьютерной отрасли, занимаясь связями с общественностью в Digital Equipment Corporation (DEC).

Старейший высокоуровневый язык, используемый до сих пор, хотя сильно пересмотренный, - FORTRAN. Имена многих ранних языков писались заглавными буквами, потому что были своего рода сокращениями.

FORTRAN соединяет первые три буквы FORmula и первые четыре TRANslation. IBM разработала его для серии 704 в середине 1950-х. Много лет он считался главным языком учёных и инженеров, имел развитую поддержку плавающей точки и даже комплексных чисел - сочетаний действительной и мнимой частей.

COBOL, COmmon Business Oriented Language, - другой старый язык, всё ещё используемый прежде всего финансовыми организациями. Его создал комитет представителей американской промышленности и Министерства обороны США начиная с 1959 года под влиянием ранних компиляторов Хоппер.

COBOL отчасти проектировали так, чтобы руководители, даже не программируя, могли читать код и проверять его работу. В действительности это происходит редко.

Чрезвычайно влиятельный, но сегодня почти не используемый, кроме любителей, ALGOL означает ALGOrithmic Language и одновременно носит имя второй по яркости звезды Персея.

Международный комитет разработал его в 1957-1958 годах. ALGOL стал прямым предком множества популярных универсальных языков последующих десятилетий и первым воплотил идею, позднее названную *структурным программированием*. До сих пор говорят об "ALGOL-подобных" языках.

ALGOL утвердил конструкции, общие почти для всех языков. С ними связаны *ключевые слова*, обозначающие операции. Несколько операторов объединяются в *блоки*, исполняемые при определённом условии или заданное число раз.

`if` исполняет оператор или блок при логическом условии, например когда height меньше 55. `for` исполняет блок многократно, обычно увеличивая переменную. *Массив* - набор однотипных значений, например названий городов. Программы организуются в блоки и функции.

Версии FORTRAN, COBOL и ALGOL существовали для домашних компьютеров, но ни один не повлиял на маленькие машины так, как BASIC.

BASIC, Beginner's All-purpose Symbolic Instruction Code, разработали в 1964 году Джон Кемени и Томас Курц с математического факультета Дартмутского колледжа для системы разделения времени.

Большинство студентов Дартмута не изучало математику или инженерное дело, поэтому от них нельзя было ожидать работы со сложностью компьютеров и тяжёлым синтаксисом. Студент вводил в терминал операторы BASIC с номерами, задающими порядок строк.

Первая программа в первом опубликованном руководстве BASIC выглядела так:

```
10 LET X = (7 + 8) / 3
20 PRINT X
30 END
```

Многие реализации BASIC были *интерпретаторами*, а не компиляторами. Компилятор читает исходный файл и создаёт исполняемый машинный код. Интерпретатор читает код и выполняет его непосредственно, не создавая исполняемый файл.

Интерпретатор написать проще, но интерпретируемая программа обычно медленнее скомпилированной. BASIC рано появился на домашних компьютерах: в 1975 году приятели Билл Гейтс (род. 1955) и Пол Аллен (род. 1953) написали интерпретатор для Altair 8800 и тем самым дали старт Microsoft Corporation.

Pascal унаследовал структуру ALGOL и некоторые особенности COBOL. Его спроектировал в конце 1960-х швейцарский профессор информатики Никлаус Вирт (род. 1934).

Среди первых программистов IBM PC особенно популярен был Turbo Pascal, выпущенный Borland International в 1983 году по выгодной цене 49,95 доллара. Его написал датский студент Андерс Хейлсберг (род. 1960).

Turbo Pascal включал *интегрированную среду разработки*, IDE: текстовый редактор и компилятор объединялись в одной программе, заметно ускоряя работу. На мэйнфреймах IDE были известны давно, но Turbo Pascal ознаменовал их приход на маленькие компьютеры.

Pascal сильно повлиял на Ada, язык Министерства обороны США, названный в честь Августы Ады Байрон, рассказчицы об Аналитической машине Бэббиджа из главы 15.

И наконец, любимый многими C, созданный в 1969-1973 годах главным образом Деннисом Ритчи в Bell Telephone Laboratories. Почему C называется C? Он произошёл от языка B, упрощённой версии BCPL, Basic CPL, который в свою очередь произошёл от CPL, Combined Programming Language.

Большинство языков старается скрыть остатки ассемблера вроде адресов памяти, но C этого не делает. В нём есть *указатель* - по существу адрес памяти. Умелому программисту указатели удобны, но для остальных опасны: они позволяют перезаписать важные области памяти и часто становятся источником ошибок. Алан Холуб назвал книгу о C *Enough Rope to Shoot Yourself in the Foot*.

C стал предком серии языков, более безопасных, чем C, и добавивших возможность работать с *объектами* - программными сущностями, объединяющими код и данные в строго определённую структуру.

Самые известные из них: C++, созданный датским информатиком Бьёрном Страуструпом (род. 1950) в 1985 году; Java, разработанный Джеймсом Гослингом (род. 1955) в Oracle Corporation в 1995 году; C#, первоначально спроектированный Андерсом Хейлсбергом в Microsoft в 2000 году.

Во время написания книги одним из самых употребительных языков был другой потомок C - Python, созданный нидерландским программистом Гвидо ван Россумом (род. 1956) в 1991 году. Но читатель 2030-х или 2040-х, возможно, уже знаком с языками, которые сегодня ещё не изобретены.

Разные высокоуровневые языки заставляют думать по-разному. Некоторые новые языки сосредоточены на манипулировании функциями, а не переменными. Их называют *функциональными*. Программисту, привыкшему к обычным *процедурным* языкам, сначала они кажутся странными, но альтернативные решения способны полностью изменить подход к задачам. Независимо от языка CPU исполняет всё тот же машинный код.

ПО может сглаживать различия между CPU и их машинными кодами. Эмуляторы позволяют запускать старые программы и древние игры на современных машинах. Когда Билл Гейтс и Пол Аллен писали BASIC для Altair 8800, они проверяли его на эмуляторе Intel 8080, созданном ими для мэйнфрейма DEC PDP-10 Гарвардского университета.

Java и C# могут компилироваться в промежуточный код, похожий на машинный, а во время исполнения превращаться в настоящий машинный код. Проект LLVM стремится создать виртуальную связь между любым высокоуровневым языком и любым набором инструкций CPU.

В этом магия программного обеспечения. При достаточных памяти и скорости любой цифровой компьютер способен сделать всё, что может другой. Таков вывод из работ Алана Тьюринга о вычислимости 1930-х годов.

Но Тьюринг показал и существование алгоритмических задач, навсегда недоступных цифровому компьютеру. Одна имеет поразительное следствие: нельзя написать программу, определяющую, правильно ли работает другая произвольная программа. Значит, абсолютной уверенности в собственном ПО быть не может.

Именно поэтому всестороннее тестирование и отладка так важны при разработке.

Один из самых успешных языков под влиянием C - JavaScript, созданный Бренданом Айком (род. 1961) в Netscape и впервые появившийся в 1995 году. Он даёт веб-страницам интерактивные возможности сверх простого текста и bitmap, которыми управляет HTML. На момент написания почти 98% десяти миллионов самых популярных сайтов использовали JavaScript.

Все распространённые браузеры понимают JavaScript. Поэтому начать писать программы на настольном компьютере или ноутбуке можно без загрузки и установки дополнительных инструментов.

Хотите поэкспериментировать? Достаточно создать в Windows Notepad или macOS TextEdit файл HTML с JavaScript, сохранить и открыть в любимом браузере: Edge, Chrome или Safari.

В Windows запустите Notepad, при необходимости найдя его поиском меню Start. В macOS откройте TextEdit через Spotlight Search и нажмите New Document. TextEdit по умолчанию создаёт rich text с форматированием, но нужен простой текст: в меню Format выберите Make Plain Text. В разделе Spelling and Grammar меню Edit отключите проверку и исправление орфографии.

Простой текст важен потому, что браузеру нужны сами символы HTML и JavaScript, а не скрытые сведения о шрифтах и оформлении. Автоматическая проверка тоже опасна: она может "исправить" слово, являющееся точным ключевым словом программы.

Введите:

```
<html>
    <head>
        <title>My JavaScript</title>
    </head>
    <body>
        <p id="result">Program results go here!</p>
        <script>
            // JavaScript programs go here
        </script>
    </body>
</html>
```

HTML строится из *тегов*, окружающих части файла. Весь документ начинается `<html>` и заканчивается `</html>`. Внутри раздел `<head>` содержит `<title>`, отображаемый в заголовке страницы. `<body>` содержит абзац `<p>` с текстом `Program results go here!`.

В `<body>` есть и раздел `<script>`, где будет JavaScript. Пока программа состоит из строки с двумя косыми чертами. `//` означает *комментарий*: всё до конца строки предназначено человеку и игнорируется при исполнении.

Теги образуют вложенную структуру: head и body находятся внутри html, title внутри head, а p и script внутри body. Закрывающий тег с косой чертой завершает область, открытую соответствующим начальным тегом.

Отступы не обязательны, многое можно поместить в одну строку. Но ради здравого смысла оставьте `<script>` и `</script>` на отдельных строках.

Сохраните файл через Save в удобном месте, например на Desktop, под именем `MyJavaScriptExperiment.html` или похожим. Расширение после точки принципиально: оно должно быть `html`. TextEdit попросит подтвердить выбор, и его надо подтвердить.

Не закрывайте редактор: файл ещё будет изменяться.

Найдите сохранённый файл и дважды щёлкните. Windows или macOS откроет его в браузере. Заголовок должен быть `My JavaScript`, а слева сверху - `Program results go here!`. Если нет, проверьте ввод.

Расширение `.html` сообщает системе и браузеру, что содержимое следует интерпретировать как веб-страницу, а не показывать как обычный текст. Поэтому нельзя позволить редактору незаметно добавить расширение rich-text или `.txt`.

Рабочий цикл эксперимента таков: впишите JavaScript между `<script>` и `</script>`, снова сохраните файл, перейдите в браузер и обновите страницу кнопкой с круговой стрелкой. Новая версия запускается двумя действиями: сохранить и обновить.

Редактор и браузер играют разные роли. В редакторе изменяется исходный код, а браузер заново читает файл, разбирает HTML и исполняет JavaScript. Обновление без сохранения запустит старую версию, а сохранение без обновления не изменит уже открытую страницу.

Первая программа:

```
let message = "Hello from my JavaScript program!";
document.getElementById("result").innerHTML = message;
```

Это два оператора в разных строках, каждый заканчивается точкой с запятой.

В первом `let` - ключевое слово JavaScript, а `message` - переменная. `let` задаёт её значение, которое позднее можно изменить. Имя не обязано быть message: подойдёт `msg` или другое имя, начинающееся с буквы и не содержащее пробелов и знаков пунктуации. Здесь в переменной хранится строка символов в кавычках.

Кавычки обозначают границы строки и не входят в отображаемое сообщение. Значение хранится в памяти программы, а имя позволяет обращаться к нему без знания физического адреса, что наглядно отличает высокий уровень от ассемблера.

Второй оператор сложнее, но нужен для взаимодействия JavaScript с HTML. `document` обозначает веб-страницу. `getElementById` ищет элемент с именем `result`, то есть тег `<p>`. `innerHTML` помещает содержимое message между `<p>` и `</p>`, словно оно изначально было набрано там.

Оператор длинный и громоздкий, потому что JavaScript должен гибко обращаться к любому элементу страницы и изменять его.

Компиляторы и интерпретаторы придирчивее к написанию, чем старомодные учителя английского. JavaScript различает регистр, поэтому `innerHTML` надо набрать именно так: `InnerHTML` и `innerHtml` не сработают. По той же причине в TextEdit отключалось исправление: превращение `let` в `Let` сломает программу.

Имена `document`, `getElementById` и `innerHTML` являются частями точно определённого интерфейса браузера. Изменение регистра превращает их в другие, неизвестные программе имена, и требуемого действия не происходит.

Сохраните новую версию и обновите страницу. Сообщение появится слева сверху. Если нет - проверьте работу.

Чтобы не удалять прежнюю программу, заключите её в специальные символы:

```
/*
let message = "Hello from my JavaScript program!";
document.getElementById("result").innerHTML = message;
*/
```

Всё между `/*` и `*/` считается комментарием. Как многие языки семейства C, JavaScript имеет многострочные комментарии `/* ... */` и однострочные `//`.

Следующая программа выполняет арифметику:

```
let a = 535.43;
let b = 289.771;
let c = a * b;
document.getElementById("result").innerHTML = c;
```

Умножение обозначается звёздочкой, потому что обычный знак умножения отсутствует в ASCII.

Последний оператор совпадает с предыдущим примером, но теперь inner HTML тега `<p>` получает переменную c - произведение чисел. JavaScript неважно, строка это или число: он выполнит необходимое для отображения.

Первые два оператора помещают в переменные числа с дробными частями. Третий читает оба значения, умножает их и сохраняет результат в c. Только четвёртый связывает вычисление с видимой веб-страницей.

Одна из важнейших высокоуровневых возможностей - *цикл*. В ассемблере он создавался JMP и условными переходами. Некоторые языки имеют похожий `goto`, но его использование обычно не рекомендуется. Программа с множеством переходов быстро становится неуправляемой.

Технический термин - *спагетти-код*: переходы словно перепутываются друг с другом. Поэтому в JavaScript вообще нет goto.

Современные языки организуют циклы без беспорядочных переходов. Сумму чисел от 1 до 100 можно найти так:

```
let total = 0;
let number = 1;
while (number <= 100)
{
    total = total + number;
    number = number + 1;
}
document.getElementById("result").innerHTML = total;
```

Пустые строки лишь визуально разделяют части. Сначала *инициализация* задаёт начальные значения. Цикл состоит из `while` и блока в фигурных скобках. Пока number меньше или равно 100, блок исполняется: number прибавляется к total и увеличивается на 1. Когда number превышает 100, выполняется оператор после закрывающей скобки и выводит результат.

Условие проверяется перед каждой итерацией. Если бы number изначально было больше 100, блок не выполнился бы ни разу. Фигурные скобки показывают, какие два оператора повторяются как единое целое.

С точки зрения алгебры две строки выглядят странно:

```
total = total + number;
number = number + 1;
```

Как total может равняться самому себе плюс number? В JavaScript = обозначает не равенство, а *присваивание*: переменная слева получает вычисленное справа значение. Проверка равенства использует два знака ==.

Сначала считывается старое значение total, к нему прибавляется number, затем результат записывается обратно под именем total. Поэтому противоречия с алгебраическим равенством нет.

Унаследованные от C сокращения:

```
total += number;
number += 1;
```

+= означает прибавить правую часть к переменной слева.

Увеличение на единицу встречается настолько часто, что записывается:

```
number++;
```

Оба действия можно даже объединить:

```
total += number++;
```

Сначала number прибавляется к total, затем увеличивается. Но запись может быть неясной менее опытному читателю, поэтому её лучше избегать.

Другой вариант использует `for`:

```
let total = 0;
for (let number = 1; number <= 100; number++)
{
    total += number;
}
document.getElementById("result").innerHTML = total;
```

В `for` три части разделены точками с запятой. Первая задаёт number равным 1. Блок выполняется, пока истинна вторая часть, то есть number не больше 100. После каждого исполнения действует третья часть, увеличивающая number. Поскольку в блоке только один оператор, фигурные скобки можно убрать.

Инициализация выполняется один раз, условие проверяется перед каждой итерацией, а увеличение - после блока. `for` собирает три управляющих элемента цикла в одной строке и делает его границы очевиднее.

Следующая программа перебирает числа 1-100 и выводит их квадратные корни:

```
for (let number = 1; number <= 100; number++)
{
    document.getElementById("result").innerHTML +=
        "The square root of " + number + " is " +
            Math.sqrt(number) + "<br />";
}
```

В блоке один длинный оператор, записанный в трёх строках. Первая оканчивается +=, поэтому новые данные добавляются к inner HTML тега <p>, а текст растёт с каждой итерацией.

Добавляемое значение сочетает текст и числа. Встроенная функция `Math.sqrt` вычисляет квадратный корень. Тег HTML `<br />` создаёт перевод строки.

Знак + здесь выполняет две роли. Между числами он означает арифметическое сложение, а при участии строк соединяет фрагменты текста. Браузер преобразует number и результат Math.sqrt в символы и склеивает их с поясняющими словами.

После завершения получится длинный список, и страницу, вероятно, придётся прокручивать.

Следующий пример реализует знаменитый алгоритм поиска простых чисел - *решето Эратосфена*. Эратосфен (276-194 до н. э.) был хранителем легендарной Александрийской библиотеки и известен точным вычислением окружности Земли.

Простые числа - целые числа, делящиеся без остатка только на себя и 1. Первое простое - 2, единственное чётное; далее 3, 5, 7, 11, 13, 17, 19, 23, 29 и так далее.

Метод начинает со списка положительных целых от 2. Поскольку 2 простое, вычёркиваются все кратные 2, кроме самой двойки. Затем, поскольку 3 простое, вычёркиваются кратные 3. Четвёрка уже исключена. Следующее простое - 5, и исключаются его кратные. Оставшиеся числа простые.

Алгоритм не пытается делить каждое число на все возможные делители. Он последовательно помечает заведомо составные числа. Булев массив играет роль листа бумаги: true означает "пока считается простым", false - "вычеркнуто".

Программа использует *массив*. Он похож на переменную и имеет имя, но хранит много элементов, каждый выбирается *индексом* в квадратных скобках после имени.

Массив primes содержит 10 000 логических значений. В JavaScript они равны ключевым словам `true` или `false`, знакомым ещё с главы 6.

Создание массива и установка всех элементов в true:

```
let primes = [];
for (let index = 0; index < 10000; index++)
{
    primes.push(true);
}
```

Более короткая, но менее очевидная запись:

```
let primes = new Array(10000).fill(true);
```

В первом варианте пустой массив растёт вызовами `push`, по одному элементу за итерацию. Во втором `new Array(10000)` сразу создаёт нужное число мест, а `fill(true)` заполняет их одинаковым начальным значением.

Основное вычисление содержит два `for`, один внутри другого - второй *вложен* в первый. Для индексов используются короткие i1 и i2. Имя переменной может включать цифры, но обязано начинаться с буквы:

```
for (let i1 = 2; i1 <= 100; i1++)
{
```

Продолжение вложенных циклов:

```
    if (primes[i1])
    {
        for (let i2 = 2; i2 < 10000 / i1; i2++)
        {
            primes[i1 * i2] = false;
        }
    }
}
```

Внешний цикл увеличивает i1 от 2 до 100, квадратного корня из 10 000. `if` выполняет внутреннюю часть, только если элемент массива равен true, то есть i1 простое. i2 начинается с 2; произведения i1 и i2 равны 2 x i1, 3 x i1, 4 x i1 и так далее. Они составные, поэтому соответствующие элементы становятся false.

Может показаться странным ограничить i1 сотней, а i2 величиной `10000 / i1`, но этого достаточно, чтобы охватить все простые до 10 000.

Вывод результата:

```
for (let index = 2; index < 10000; index++)
{
    if (primes[index])
    {
        document.getElementById("result").innerHTML +=
            index + " ";
    }
}
```

Если JavaScript заинтересовал вас, не продолжайте пользоваться Notepad или TextEdit. Есть гораздо лучшие инструменты, которые сообщат об опечатке и других ошибках.

На CodeHiddenLanguage.com в разделе этой главы доступны простые JavaScript-программы с подробными учебными комментариями.

Люди спорят, является ли программирование искусством или наукой. Университетские курсы называются Computer Science, но существует знаменитая серия Дональда Кнута *The Art of Computer Programming*. В программировании есть и наука, и искусство, но в действительности это нечто иное. Физик Ричард Фейнман писал: "Информатика похожа на инженерное дело - она целиком о том, чтобы заставить что-либо что-либо делать".

Часто это борьба в гору. Ошибиться в программе очень легко, а поиск ошибки может занять много времени. Отладка сама по себе является искусством, или наукой, или инженерным подвигом.

Показанное - лишь пресловутая верхушка айсберга JavaScript. Но история учит осторожности рядом с айсбергами. Иногда сам компьютер делает неожиданное. Попробуйте:

```
let a = 55.2;
let b = 27.8;
let c = a * b;
document.getElementById("result").innerHTML = c;
```

Программа выводит `1534.5600000000002`. Это выглядит неверно и действительно неверно: точный результат равен 1534,56. Что произошло?

Числа с плавающей точкой исключительно важны. В 1985 году Institute of Electrical and Electronics Engineers (IEEE) установил стандарт, признанный также American National Standards Institute (ANSI). ANSI/IEEE Std 754-1985 называется *IEEE Standard for Binary Floating-Point Arithmetic*.

Для стандарта он невелик, всего 18 страниц, но подробно задаёт удобное кодирование чисел с плавающей точкой. Это один из важнейших компьютерных стандартов, применяемый практически всеми современными компьютерами и программами.

IEEE определяет два основных формата: одинарная точность по четыре байта и двойная по восемь. Некоторые языки разрешают выбирать, а JavaScript использует исключительно двойную точность.

Стандарт основан на научной записи, где число состоит из *значащей части*, или *мантиссы*, умноженной на 10 в целой степени - *экспоненте*:

```
42 705,7846 = 4,27057846 x 10^4
```

Форма *нормализована*, потому что слева от запятой мантиссы только одна цифра.

IEEE представляет числа так же, но в двоичной системе. До сих пор двоичные числа книги были целыми, однако двоичная запись допускает дроби. Рассмотрим:

```
101.1101
```

Точку здесь нельзя называть десятичной: это *двоичная точка*. Слева целая часть, справа дробная. Как показано в главе 10, позиции соответствуют степеням 2; справа от точки - отрицательным степеням.

Преобразование `101.1101` в десятичное:

```
1 x 2^2  +
0 x 2^1  +
1 x 2^0  +
1 x 2^-1 +
1 x 2^-2 +
0 x 2^-3 +
1 x 2^-4
```

Отрицательные степени вычисляются последовательным делением единицы на 2:

```
1 x 4      +
0 x 2      +
1 x 1      +
1 x 0,5    +
1 x 0,25   +
0 x 0,125  +
1 x 0,0625
```

Итого десятичный эквивалент равен 5,8125.

Как в нормализованной десятичной записи слева одна цифра, так в нормализованной двоичной значащей части слева от точки только один бит:

```
101.1101 = 1.011101 x 2^2
```

Следовательно, слева от точки нормализованного двоичного числа с плавающей точкой всегда находится единственная 1.

Двойная точность IEEE занимает восемь байтов, то есть 64 бита:

Поле	Размер
`s` - знак	1 бит
`e` - экспонента	11 бит
`f` - дробь значащей части	52 бита

Поскольку ведущая 1 нормализованной мантиссы известна заранее, она не хранится. Записываются только 52 бита дробной части, но точность считается 53-битной.

Одиннадцатибитная экспонента принимает значения 0-2047. Это *смещённая экспонента*: для получения настоящей знаковой степени вычитается число, называемое *смещением*.

Для двойной точности смещение равно 1023. Число, представленное знаком `s`, экспонентой `e` и дробью `f`, вычисляется так:

```
(-1)^s x 1.f x 2^(e - 1023)
```

`(-1)^s` - хитрый математический способ сказать: при s = 0 число положительное, поскольку любое число в нулевой степени равно 1; при s = 1 оно отрицательное, поскольку `(-1)^1 = -1`.

`1.f` означает единицу, двоичную точку и 52 бита дроби. Всё умножается на 2 в степени, равной сохранённой одиннадцатибитной экспоненте минус 1023.

Некоторые детали пока пропущены. В описанной форме нельзя представить даже ноль. IEEE обрабатывает его особым образом и допускает отрицательный ноль для очень малых отрицательных значений, положительную и отрицательную бесконечность и NaN, Not a Number. Эти случаи - важная часть стандарта.

Пример `101.1101` хранится с 52-битной дробью:

```
0111 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Пробелы через четыре бита добавлены только для чтения. Смещённая экспонента равна 1025:

```
1.011101 x 2^(1025 - 1023) = 1.011101 x 2^2
```

Если не считать нуля, наименьшее по модулю положительное или отрицательное число двойной точности:

```
1.0000000000000000000000000000000000000000000000000000 x 2^-1022
```

После точки 52 нуля. Наибольшее:

```
1.1111111111111111111111111111111111111111111111111111 x 2^1023
```

Десятичный диапазон приблизительно от `2,2250738585072014 x 10^-308` до `1,7976931348623158 x 10^308`. Десять в степени 308 - единица с 308 нулями.

Пятьдесят три бита значащей части, включая не хранящуюся единицу, дают примерно 16 десятичных цифр точности, но пределы существуют. Числа `140 737 488 355 328,00` и `140 737 488 355 328,01` сохраняются одинаково и в программе неразличимы.

Другая проблема: подавляющее большинство десятичных дробей не представляется точно. Например, 1,1 хранится с 52-битной дробью:

```
0001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010
```

Полное двоичное число:

```
1.0001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010
```

Начало его перевода в десятичную систему:

```
1 + 2^-3 + 2^-4 + 2^-7 + 2^-8 + 2^-11 + ...
```

то есть:

```
1 + 0,0625 + 0,03125 + 0,00390625 + 0,001953125 + 0,000244140625 + ...
```

В итоге получается не 1,1, а:

```
1,1000000000000008881...
```

Арифметика над неточно представленными числами тоже даёт неточные результаты. Поэтому JavaScript получает `1534.5600000000002` при умножении 55,2 на 27,8.

Мы привыкли считать числа непрерывным множеством без промежутков, но компьютер по необходимости хранит дискретные значения. Теоретическую основу математики цифровых компьютеров даёт *дискретная математика*.

Дополнительная сложность плавающей точки - вычисление корней, степеней, логарифмов и тригонометрических функций. Но все они сводятся к четырём операциям: сложению, вычитанию, умножению и делению.

Например, синус вычисляется разложением:

```
sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...
```

x задаётся в радианах, которых 2π в 360 градусах. Восклицательный знак означает факториал, произведение целых от 1 до указанного числа. Например, `5! = 1 x 2 x 3 x 4 x 5`. Степень каждого члена тоже получается умножением. Остальное - деление, сложение и вычитание.

Страшнее всего многоточие: оно предлагает продолжать вычисление вечно. На практике, если ограничиться диапазоном от 0 до π/2, из которого выводятся остальные значения синуса, вечность не нужна. Примерно дюжина членов даёт точность 53 бит двойного формата.

Компьютеры должны облегчать жизнь, и необходимость каждому писать множество процедур плавающей точки противоречила бы цели. Но в этом красота ПО: однажды написанные для машины процедуры используют все.

Плавающая арифметика настолько важна в науке и инженерии, что всегда имела высокий приоритет. На ранних компьютерах её процедуры становились одной из первых программных задач после создания нового типа машины. Языки обычно поставляют целые библиотеки математических функций; уже встречалась `Math.sqrt` JavaScript.

Разумно и специальное оборудование, выполняющее плавающие вычисления напрямую. Первым коммерческим компьютером с необязательной аппаратурой плавающей точки стал IBM 704 в 1954 году.

Все числа IBM 704 занимали 36 бит. Для плавающей точки это 27 бит значащей части, 8 бит экспоненты и один знак. Аппаратура выполняла сложение, вычитание, умножение и деление, а остальные функции оставались программными.

На рабочий стол аппаратная плавающая точка пришла в 1980 году с Intel 8087 Numeric Data Coprocessor - интегральной схемой, которую теперь называют *математическим сопроцессором* или *устройством плавающей точки*, FPU.

8087 назывался сопроцессором, потому что не работал самостоятельно, а только вместе с первыми шестнадцатибитными Intel 8086 и 8088. В то время 8087 считался самой сложной созданной интегральной схемой. Позднее математические сопроцессоры встроили в сам CPU.

Современные программисты пользуются числами с плавающей точкой так, словно они просто часть компьютера. Так оно и есть.

Эта страница намеренно оставлена пустой.

Глава 28. Мировой мозг

В 1936 и 1937 годах английский писатель Герберт Джордж Уэллс прочёл серию публичных лекций на довольно необычную тему. Уэллсу было уже немного за семьдесят. Знаменитые фантастические романы *Машина времени*, *Остров доктора Моро*, *Человек-невидимка* и *Война миров* вышли ещё в 1890-х и принесли ему известность.

Но Уэллс превратился в публичного интеллектуала, глубоко размышлявшего о социальных и политических вопросах и делившегося мыслями с обществом.

Лекции 1936-1937 годов были опубликованы в 1938 году под названием *World Brain*. Уэллс предлагал энциклопедию, но не коммерческое издание для продажи от двери к двери. Эта *Всемирная энциклопедия* должна была заключить в себе мировое знание способом, ранее невозможным.

Европа переживала опасное время. Память о Великой войне двадцатилетней давности оставалась свежей, но континент уже стремительно приближался к новому всеохватывающему конфликту. Оптимист и утопист Уэллс считал науку, рациональность и знание лучшими инструментами, способными направить мир в будущее.

Его энциклопедия должна была содержать господствующие понятия общественного устройства, основные сведения всех областей знания, точную и достаточно подробную картину Вселенной, общую историю мира и надёжную полную систему ссылок на первичные источники.

Короче говоря, она представляла бы "общее толкование реальности" и "умственное объединение".

Энциклопедию пришлось бы постоянно обновлять вслед за расширением знаний. В ходе развития она стала бы своеобразным умственным распределительным центром, куда знания и идеи поступают, сортируются, обобщаются, усваиваются, проясняются и сравниваются. Это было бы материальное начало настоящего *Мирового мозга*.

Первые зародышевые цифровые компьютеры только строились в 1930-х, и Уэллс едва ли о них знал. Поэтому он представлял энциклопедию книгами: двадцатью, тридцатью или сорока томами. Но он был знаком с новой технологией микрофильма.

Уэллс предполагал, что вскоре появятся микроскопические библиотеки, где будет сохранена фотография каждой важной книги и документа мира, легко доступная исследователю. Любой студент в любой точке мира сможет в удобное время сесть у проектора в своём кабинете и изучить точную копию любой книги или документа.

Какая картина будущего!

Менее чем через десятилетие, в 1945 году, инженер и изобретатель Вэнивар Буш предложил сходное, но немного более развитое видение.

Буш уже вошёл в историю вычислений. Начиная с 1927 года он и его студенты электротехнического факультета Massachusetts Institute of Technology построили дифференциальный анализатор - важный аналоговый компьютер, решавший дифференциальные уравнения. К началу 1930-х он стал деканом инженерного факультета и вице-президентом MIT.

Некролог New York Times 1974 года назвал Буша "образцом инженера - человеком, который добивался результата", будь то технические задачи или государственная бюрократия.

Во Вторую мировую войну Буш координировал более 30 000 учёных и инженеров, включая руководство Манхэттенским проектом, создавшим первую атомную бомбу. Несколько десятилетий он выступал за участие учёных и инженеров в государственной политике.

В конце войны Буш написал знаменитую статью *As We May Think* для Atlantic Monthly за июль 1945 года. В ретроспективе она кажется пророческой. Сокращённая версия вышла в сентябрьском Life с фантастическими иллюстрациями.

Как Уэллс, Буш сосредоточился на информации и трудности поспевать за её ростом.

Он писал о растущей горе исследований и о том, что расширение специализации всё сильнее затягивает нас в трясину. Исследователь ошеломлён результатами и выводами тысяч коллег, которые не успевает понять и тем более запомнить. Проблема не столько в чрезмерности публикаций, сколько в том, что их объём намного превысил способность использовать накопленные сведения. Совокупный человеческий опыт растёт с невероятной скоростью, а средство поиска нужного элемента в этом лабиринте остаётся таким же, как во времена парусников с прямым вооружением.

Буш видел быстро развивающиеся технологии, способные помочь будущему учёному. Он представлял камеру на лбу, срабатывающую при необходимости что-то зафиксировать; говорил о микрофильме, факсимильной передаче документов и машинах, записывающих человеческую речь и превращающих её в текст.

Все эти средства расширяли саму запись человеческого опыта: позволяли быстрее фиксировать, копировать и передавать сведения. Однако увеличение объёма не решало задачу нахождения нужного материала. Чем больше становился архив, тем острее требовался способ проходить через него осмысленными путями.

Но в конце статьи оставалась проблема: запись можно неизмеримо расширить, а обращаться даже к нынешнему объёму уже почти невозможно. Большинство сведений организовано и индексировано по алфавиту, чего явно недостаточно.

Человеческий разум действует иначе - по ассоциации. Удерживая один объект, он мгновенно перескакивает к следующему, предложенному связью мыслей в запутанной сети путей клеток мозга.

Буш придумал "механизированное личное дело и библиотеку" - сложный письменный стол, хранящий микрофильм и обеспечивающий быстрый доступ. Он назвал машину *memex*.

Большинство материалов memex приобретается готовым на микрофильме: книги, изображения, журналы, газеты и деловая переписка. Предусмотрен и прямой ввод. На прозрачную поверхность сверху кладутся рукописные заметки, фотографии и записки; нажатие рычага фотографирует их в следующую свободную область памяти memex.

Главное - к документам можно добавлять примечания и комментарии и соединять их *ассоциативным индексированием*.

Связывание двух объектов - основная возможность memex. Множество соединённых объектов образует путь, который можно просматривать быстро или медленно рычагом, похожим на перелистывание книги. Это словно физические материалы из далёких друг от друга источников собраны и переплетены в новую книгу.

Путь не обязан следовать алфавиту или заранее принятой классификации. Его создаёт сам пользователь, соединяя материалы в порядке собственных ассоциаций. Другой человек может получить готовый путь, поместить его в свой memex и добавить новые ответвления.

Появятся новые энциклопедии с готовой сетью ассоциативных путей, которые можно поместить в memex и развивать.

Буш предвидел и ленивое удобство не помнить всё: пользователь "снова получает привилегию забывать множество вещей, не нужных прямо сейчас, с уверенностью, что найдёт их, если они окажутся важны".

В 1965 году, через два десятилетия после статьи Буша, компьютерное воплощение мечты становилось возможным. Визионер Тед Нельсон (род. 1937) взялся осовременить memex в статье *Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate*, опубликованной в материалах конференции Association for Computing Machinery ACM '65.

Нельсон писал, что структуры файлов для личных данных и поддержки творчества должны принципиально отличаться от применяемых в деловой и научной обработке. Они обязаны допускать сложную индивидуальную организацию, полную изменяемость, неопределённые альтернативы и подробную внутреннюю документацию.

Ссылаясь на memex, Нельсон утверждал: "Аппаратура готова". Для амбициозной и привлекательной структуры ему понадобилось новое слово - *hypertext*.

Гипертекст - корпус письменного или графического материала, связанный настолько сложно, что его нельзя удобно представить на бумаге. Он может содержать резюме, карты содержания и связей, примечания, дополнения и сноски исследователей.

Правильно спроектированная и управляемая система могла бы иметь огромный образовательный потенциал: расширить выбор учащегося, чувство свободы, мотивацию и интеллектуальное понимание. Она могла бы расти бесконечно, постепенно включая всё больше письменных знаний мира. Внутренняя файловая структура должна поддерживать рост, изменения и сложную организацию информации.

В работах Уэллса, Буша и Нельсона видно, что некоторые люди размышляли об интернете задолго до его технической осуществимости.

Связь компьютеров на больших расстояниях - труднейшая задача. Сам интернет возник из исследований Министерства обороны США в 1960-х. Advanced Research Projects Agency Network (ARPANET) заработала в 1971 году и заложила многие интернет-концепции.

Вероятно, важнейшая - *коммутация пакетов*: данные делятся на небольшие *пакеты*, снабжённые информацией, называемой *заголовком*.

Допустим, компьютер A хранит текстовый файл размером 30 000 байтов, а компьютер B каким-либо образом с ним соединён и запрашивает файл. A делит данные на 20 частей по 1500 байтов. Заголовок каждого пакета указывает источник A, получателя B, имя файла и номер части, например 7 из 20.

B подтверждает получение каждого пакета и собирает файл заново. Если какой-то пакет потерян, B запрашивает повторную копию.

Пакеты могут поступать не обязательно по порядку. Номера в заголовках позволяют B поставить часть 7 именно между частями 6 и 8 независимо от последовательности прихода. Подтверждение сообщает A, какие части благополучно достигли назначения.

Заголовок может содержать *контрольную сумму* - число, стандартным способом вычисленное по всем байтам пакета. B повторяет вычисление и сравнивает результат. Несовпадение означает повреждение при передаче, и пакет запрашивается снова.

Контрольная сумма не исправляет повреждённые биты сама, но обнаруживает, что полученный пакет отличается от отправленного. Повторная передача заменяет только ошибочную часть проверенной копией.

Пакетная коммутация имеет преимущества перед отправкой целого файла. Соединение одновременно разделяют другие компьютеры со своими пакетами: одна большая передача не захватывает систему целиком. При ошибке повторяется только повреждённый пакет, а не весь файл.

Чередование пакетов разных разговоров позволяет общей линии обслуживать множество пользователей. Каждый поток получает часть пропускной способности, а сеть не резервирует весь канал одному файлу на всё время передачи.

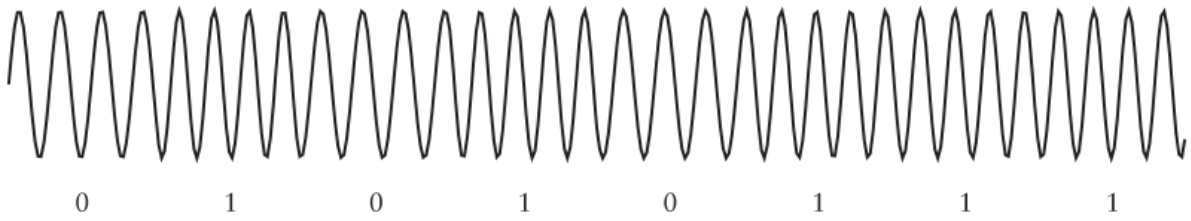
В книге цифровые сведения передавались по проводам: ток означает 1, отсутствие тока - 0. Но провода схем были короткими. Для больших расстояний нужны другие стратегии.

Первые дальние цифровые соединения использовали телефонные линии, поскольку они уже существовали и были доступны. Телефонная система создавалась для человеческой речи и передаёт звуковые колебания примерно 300-3400 Гц.

Двоичные нули и единицы можно превратить в звуковую волну *модуляцией* - изменением аналогового аудиосигнала так, чтобы он кодировал цифровые данные.

Один из первых модуляторов, Bell 103, выпускался AT&T с 1962 года и влиял на технику до 1990-х. Он работал в *полном дуплексе*, одновременно передавая и принимая данные. На одном конце линии находилась вызывающая станция, на другом отвечающая. Скорость составляла 300 бит в секунду.

Bell 103 применял *частотную манипуляцию*, frequency-shift keying (FSK). Вызывающая станция кодировала 0 частотой 1070 Гц, а 1 - 1270 Гц. Так выглядит восьмибитный ASCII-код буквы W:



Промежутки между циклами нулевых битов чуть шире, поскольку частота ниже. Отвечающая станция работала аналогично, но использовала 2025 и 2225 Гц. Для простой проверки ошибок часто добавлялся бит чётности.

Устройство, модулирующее тон для двоичных данных, также *демодулирует* входящий тон обратно в нули и единицы. Поэтому оно называется *модулятором-демодулятором*, или *модемом*.

Bell 103 передавал 300 бит/с и назывался устройством на 300 *бод*. Единица названа в честь Эмиля Бодо из главы 13. Бод измеряет число символов в секунду и иногда совпадает с бит/с, но не всегда.

Например, FSK с четырьмя частотами для последовательностей 00, 01, 10 и 11 при 1000 изменениях тона в секунду имеет скорость 1000 бод, но переносит 2000 бит/с.

Модем на 300 бод при соединении издавал громкий шум, который фильмы и телепередачи используют как звуковой символ домашних компьютеров 1980-1990-х.

Поздние модемы цифровых телефонных линий достигли 56 килобит/с и назывались 56K. В некоторых местах они используются до сих пор. Более быстрые соединения дают digital subscriber line (DSL), коаксиальный кабель и спутники. Они несут радиочастотные колебания, а сложные методы модуляции кодируют больше цифровых данных и обеспечивают намного более высокую скорость.

Для значительной части межконтинентальных соединений и связи между прибрежными районами используется другая среда. Под морем лежит множество волоконно-оптических кабелей: тонких волокон стекла или пластика, несущих инфракрасный свет. Свет обычно не изгибается, но отражается от внутренней поверхности волокна, поэтому кабель может гнуться.

В кабель объединяются сотни волокон, каждое несёт отдельную передачу; некоторые сами переносят несколько сигналов. Информация кодируется импульсами света: он очень быстро выключается и включается, где выключено - 0, включено - 1. Так достигаются скорости современного интернета.

Критична и топология. Можно было бы построить один гигантский компьютер и подключить к нему все остальные. Это упростило бы интернет, но удалённые пользователи испытывали бы большие задержки, а сбой центральной машины отключил бы весь мир.

Интернет децентрализован, обладает большой избыточностью и не имеет единой точки отказа. Очень большие компьютеры, хранящие данные, называются *серверами*, а меньшие, обращающиеся к ним, - *клиентами*.

Избыточность означает наличие альтернативных связей. Если один маршрут или узел не работает, пакет может пройти другим путём. Архитектура не гарантирует отсутствия отдельных отказов, но не позволяет единственному центральному компьютеру обрушить всю сеть.

Клиент не соединяется с сервером напрямую. Доступ предоставляет *интернет-провайдер*, ISP, которому обычно платят ежемесячно. При мобильном интернете оператор связи одновременно является ISP.

Кроме проводов, кабелей и радиоволн интернет связывают *маршрутизаторы*, создающие путь между клиентом и сервером. Домашний маршрутизатор может быть частью модема или Wi-Fi-хаба и содержать разъёмы Ethernet для физического подключения компьютеров и принтера.

Маршрутизаторы глобальной сети сложнее. Каждый связан с другими, те - со следующими, образуя запутанную сеть. В них есть собственные CPU, поскольку они хранят таблицы маршрутов или алгоритмическую политику выбора лучшего пути пакета.

Каждый маршрутизатор рассматривает заголовок пришедшего пакета и решает только следующий шаг. Ему не обязательно непосредственно соединяться с конечным сервером или знать весь физический путь: достаточно выбрать подходящего соседнего маршрутизатора.

Другое повсеместное устройство - *сетевой интерфейсный контроллер*, NIC. У каждого NIC есть уникальный постоянный аппаратный идентификатор - адрес Media Access Control, MAC. Он состоит из 12 шестнадцатеричных цифр, иногда записанных шестью парами.

NIC соединяет конкретное устройство с локальной сетью. MAC относится к аппаратному интерфейсу, поэтому разные интерфейсы одного компьютера получают разные адреса, даже если принадлежат одной машине.

Каждая единица оборудования в интернете имеет свой MAC. У настольного компьютера или ноутбука обычно несколько адресов: для Ethernet, Wi-Fi и Bluetooth, связывающего близкие устройства радиоволнами. Их можно увидеть в настройках компьютера. У модема и Wi-Fi-хаба MAC часто напечатаны на наклейках. Телефон, вероятно, имеет адреса Wi-Fi и Bluetooth.

Двенадцать шестнадцатеричных цифр позволяют предположить, что адреса закончатся нескоро: на каждого человека планеты приходится больше 30 000 вариантов.

Интернет поддерживает разные службы, включая электронную почту и обмен файлами, но большинство людей взаимодействует с ним через World Wide Web, в основном изобретённую английским учёным Тимом Бернерсом-Ли (род. 1955) в 1989 году.

Создавая Web, он принял слово "hypertext", придуманное Тедом Нельсоном, когда Бернерсу-Ли было десять лет.

Основной документ Web называется страницей, или веб-страницей, и состоит из текста Hypertext Markup Language (HTML). Немного HTML встречалось в предыдущей главе. Теги `<p>` обозначают абзац, `<h1>` - заголовок верхнего уровня, `<img>` - bitmap.

Один из важнейших тегов - `<a>`, anchor. Он заключает *гиперссылку*, обычно короткий фрагмент текста с особым оформлением, часто подчёркнутый. Щелчок или касание загружает другую страницу. Так страницы соединяются.

Ссылки могут вести к разным частям большого документа, как оглавление книги, указывать источники или дополнительные сведения, позволяя всё глубже погружаться в тему.

За несколько десятилетий Web вырос невероятно. Ни Уэллс, ни Буш не могли представить возможностей онлайновых исследований, покупок и развлечений, радости от видео с кошками и извращённой привлекательности ожесточённых политических споров с незнакомцами.

Доинтернетная компьютерная революция теперь кажется незавершённой. Интернет стал её вершиной и кульминацией, а успех революции следует оценивать по тому, насколько интернет сделал мир лучше. Эта тема требует умов, способных мыслить глубже автора.

Страницы идентифицируются Uniform Resource Locator, URL. Одна страница сайта книги имеет адрес:

```
https://www.CodeHiddenLanguage.com/Chapter27/index.html
```

Он состоит из доменного имени `www.CodeHiddenLanguage.com`, каталога `Chapter27` и HTML-файла `index.html`. В начале указан *протокол*. `http` означает Hypertext Transfer Protocol.

`https` - защищённый вариант HTTP. Протоколы описывают, как браузер получает страницы с сайта.

Существует также Uniform Resource Identifier, URI, имеющий почти ту же форму, но способный служить уникальным идентификатором, а не ссылкой на веб-страницу.

Любое приложение современного компьютера может попросить ОС начать *HTTP-запрос*. Достаточно передать URL строкой, например `https://www.CodeHiddenLanguage.com`. Через некоторое, желательно короткое, время приложение получает *HTTP-ответ* с запрошенной страницей.

При неудаче приходит код ошибки. Если клиент запрашивает несуществующий файл вроде `https://www.CodeHiddenLanguage.com/FakePage.html`, ответом будет знакомый код 404: страница не найдена.

Между запросом и ответом происходит сложное взаимодействие клиента и отвечающего сервера.

Приложение видит удобную операцию "получить URL", тогда как операционная система и сетевое ПО устанавливают соединение, формируют пакеты, следят за их доставкой, принимают ответ и передают собранные данные приложению.

URL - лишь удобный человеку псевдоним настоящего идентификатора сайта, *адреса Internet Protocol*, например `50.87.147.75`. Адрес IPv4 - 32-битное число, IPv6 использует 128 бит.

Чтобы получить IP сайта, браузер обращается к Domain Name System (DNS) - огромному справочнику, сопоставляющему доменные имена IP-адресам.

Именно DNS позволяет людям запоминать слова вместо тридцатидвух- или стодвадцативосьмибитных чисел. До отправки запроса по сети доменное имя должно быть разрешено в адрес, понятный маршрутизаторам.

IP сайта фиксирован. Клиентский компьютер тоже имеет IP, вероятно назначенный провайдером. Всё больше бытовых приборов получают локально доступные IP; это примеры *интернета вещей*, IoT.

Посылая HTTP-запрос, клиент общается с сервером через набор протоколов TCP/IP: Transmission Control Protocol и Internet Protocol. Они делят файл на пакеты и добавляют заголовки.

Заголовок содержит IP источника и назначения, не меняющиеся на пути через маршрутизаторы, а также исходный и конечный MAC. MAC-адреса изменяются при переходе пакета от одного маршрутизатора к другому.

IP описывает конечные точки всего путешествия, а MAC - ближайший участок текущей локальной передачи. На каждом новом звене формируется оболочка с аппаратными адресами следующей пары устройств, тогда как IP продолжает указывать первоначального клиента и сервер.

Большинство маршрутизаторов имеет таблицу или политику выбора следующего наиболее эффективного узла для запроса от клиента к серверу и ответа обратно. Маршрутизация пакетов - вероятно, самая сложная часть интернета.

При первом посещении сайта книги пользователь, вероятно, вводит:

```
CodeHiddenLanguage.com
```

Особый регистр букв не нужен: доменные имена регистронезависимы.

Браузер сам добавляет `https` к HTTP-запросу. Имя файла не указано. Сервер хранит список файлов по умолчанию; для этого сайта первым идёт `default.html`. Ввод домена эквивалентен:

```
CodeHiddenLanguage.com/default.html
```

Это домашняя страница. Браузер позволяет посмотреть HTML напрямую командой вроде View Page Source.

Получив `default.html`, браузер начинает *parsing*, описанный в главе 27: посимвольно читает HTML, распознаёт теги и размещает элементы. На уровне CPU это множество CMP и условных переходов. Работу выполняет HTML engine, вероятно написанный на C++. Для отрисовки браузер использует графические средства ОС.

HTML-файл не является готовым изображением страницы. Движок должен построить структуру документа, определить вложенность элементов, вычислить их размеры и положение, а затем попросить графическую систему нарисовать текст, изображения и прочие объекты.

При разборе обнаруживается ссылка на `style.css` - текстовый файл Cascading Style Sheets, описывающий оформление. Браузер делает ещё один HTTP-запрос.

CSS отделяет содержание от внешнего вида: HTML сообщает, что является заголовком, абзацем или ссылкой, а таблица стилей задаёт шрифты, цвета, интервалы и расположение.

Ниже HTML ссылается на `Code2Cover.jpg`, изображение обложки книги. Следующий запрос получает JPEG.

Далее перечислены главы со ссылками на другие страницы. Пока браузер не загружает их, а лишь показывает гиперссылки.

При выборе Chapter 6 отправляется запрос к `https://www.codehiddenlanguage.com/Chapter06`. Файл снова не указан. Сервер проверяет список: `default.html` в папке Chapter06 отсутствует, но следующим идёт `index.html`, и он возвращается.

Браузер разбирает новую страницу. Она тоже ссылается на `style.css`, но файл уже *кэширован*, то есть сохранён для будущего использования и не загружается повторно.

В `index.html` есть несколько `<iframe>`, ссылающихся на другие HTML-файлы. Они тоже загружаются, а их разделы `<script>` перечисляют JavaScript-файлы.

Таким образом, одна видимая страница может потребовать целой последовательности запросов. Главный HTML служит картой зависимостей: вслед за ним браузер получает стили, изображения, вложенные документы и программы.

JavaScript-файлы загружаются, разбираются и исполняются.

Раньше браузер *интерпретировал* JavaScript по мере разбора. Теперь JavaScript engines *компилируют* код, но не весь сразу, а по мере необходимости. Это *just-in-time compilation*, JIT.

Сайт CodeHiddenLanguage.com передаёт интерактивную графику на настольный компьютер, но сами страницы HTML статичны. Сервер способен выдавать и динамическое содержимое: получив определённый URL, он выполняет любые необходимые действия, на лету создаёт HTML и возвращает клиенту.

К URL иногда добавляются строки запросов: обычно после вопросительного знака, с разделением параметров амперсандами. Сервер разбирает и интерпретирует их.

Серверы также поддерживают стиль URL REST, representational state transfer, связанный с передачей файлов данных от сервера клиенту.

Эти возможности называются *server-side*, поскольку работают программы сервера, тогда как JavaScript - клиентский язык. Клиентская программа JavaScript может взаимодействовать с серверной.

Веб-программирование предлагает ошеломляющее множество вариантов, что видно по разнообразию сайтов.

По мере переноса обработки и данных на серверы их совокупность стали называть *облаком*. Чем больше личных данных хранится в облаке, тем менее важны конкретные компьютеры, на которых их создают или открывают. Вычислительный опыт становится ориентированным на пользователя, а не на оборудование.

Что подумали бы об интернете Уэллс и Буш?

Оба оптимистично считали жизненно важным улучшить доступ к знаниям и мудрости мира. С этим трудно спорить. Но столь же очевидно, что доступ не переносит цивилизацию автоматически в золотой век. Сегодня люди скорее ещё сильнее подавлены объёмом информации, чем чувствуют способность ею управлять.

Интернет, отражающий множество людей, характеров, убеждений и интересов, безусловно представляет собой какой-то Мировой мозг. Но это не "общее толкование реальности", которого хотел Уэллс. Почти рядом с подлинным знанием существуют тревожные проявления псевдонауки и теорий заговора.

Уэллсу, вероятно, понравилась бы идея Google Books, созданного сканированием и оцифровкой книг и журналов разных библиотек. Не защищённые авторским правом книги доступны полностью.

Но создатели Google Books словно забыли о каталожных карточках и заставили пользователя полагаться на несовершенный поиск.

Из-за этого фундаментального недостатка найти в Google Books конкретный материал порой мучительно трудно.

Почти полная противоположность - JSTOR, Journal Storage, собрание академических журналов с исключительно тщательной организацией и каталогизацией. Сначала JSTOR был закрытым ресурсом, но после постыдного случая с преследованием и трагическим самоубийством программиста, стремившегося сделать материалы свободными, он стал гораздо доступнее обществу.

Для читающих традиционную западную нотную запись International Music Score Library Project (imslp.org) является для нот тем же, чем Google Books для книг. Это огромное хранилище оцифрованных партитур, вышедших из-под охраны авторского права, к счастью хорошо каталогизированных и индексированных.

Если сопоставить современность с идеями Буша и Нельсона о собственной сети документов, чего-то не хватает. Google Books, JSTOR и IMSLP сопротивляются произвольным связям, которые они представляли. Текстовые процессоры и электронные таблицы позволяют хранить ссылки на источники, но не очень гибко.

Ближе всего к Всемирной энциклопедии Уэллса, очевидно, Wikipedia. Идея энциклопедии, редактируемой пользователями, легко могла привести к хаосу и бесполезности. Но под внимательным и добросовестным руководством Джимми Уэйлса (род. 1966) она стала самым необходимым сайтом интернета.

В *World Brain* Уэллс писал, что энциклопедия для всего человечества не может допускать ограничивающих догм без одновременной корректирующей критики. Её необходимо ревностно охранять от непрерывного вторжения узкой пропаганды.

Она будет иметь общий оттенок того, что многие назовут скептицизмом. Почитаемый миф следует рассматривать как миф, а не как символическое представление некой высшей истины. Видения, проекты и теории надо отличать от твёрдо установленного факта.

Энциклопедия неизбежно выступит против национальной мании величия и сектантских предположений. Она будет за мировое сообщество, а не безразлична к нему, поскольку должна стать его важнейшей частью.

Если это называется предвзятостью, Всемирная энциклопедия будет предвзятой. Она неизбежно склоняется к организации, сравнению, строительству и творчеству. Это по существу созидательный проект, который должен стать главной силой, направляющей рост нового мира.

Цели амбициозны, и впечатляет, насколько близко Wikipedia подошла к их выполнению.

Вероятно, большинство из нас не разделяет оптимизма Уэллса, будто одно присутствие корпуса знаний способно направить мир к лучшему будущему. Иногда говорят: если построить, они придут. Как бы ни хотелось верить, гарантии нет. Человеческая природа редко соответствует ожиданиям.

И всё же каждый из нас должен делать всё, что может.