23.10.2020 ИС 2к Основы архитектуры

**Тема Память**

Основная память

**Память** - это тот компонент компьютера, в котором хранятся программы и дан-
ные. Также часто встречается термин «запоминающее устройство». Без памяти,
откуда процессоры считывают и куда записывают информацию, не было бы со-
временных цифровых компьютеров.

**Бит**

Основной единицей хранения данных в памяти является двоичный разряд, ко-
торый называется **битом**. Бит может содержать 0 или 1. Эта самая маленькая
единица памяти. (Устройство, в котором хранятся только нули, вряд ли могло
быть основой памяти - необходимы, по крайней мере, два значения.)

Часто говорят, что применение двоичной системы счисления в компьютерах
объясняется ее «эффективностью». При этом имеется в виду (хотя сами говоря-
щие это редко осознают), что хранение цифровой информации может быть осно-
вано на отличиях между разными величинами какой-либо физической характе-
ристики, например напряжения или тока. Чем больше величин нужно различать,
тем меньше отличий между смежными величинами и тем менее надежна память.
В двоичной системе требуется различать всего две величины, следовательно,
это - самый надежный метод кодирования цифровой информации. Если вы не
знакомы с двоичной системой счисления, загляните в приложение А.

Считается, что некоторые компьютеры, например мэйнфреймы IBM, исполь-
зуют и десятичную, и двоичную арифметику. На самом деле здесь применяется
так называемый **двоично-десятичный код**. Для хранения одного десятичного
разряда задействуются 4 бита. Эти 4 бита дают 16 комбинаций для размещения
10 различных значений (от 0 до 9). При этом 6 оставшихся комбинаций не ис-

пользуются. Вот как выглядит число 1944 в двоично-десятичной и в чисто дво-
ичной системах счисления (в обоих случаях используется 16 бит):

+ двоично-десятичное представление - 0001 1001 0100 0100;

+ двоичное представление - 0000011110011000.

В двоично-десятичном представлении 16 бит достаточно для хранения числа
от 0 до 9999, то есть доступно всего 10 000 различных комбинаций, а в двоичном
представлении те же 16 бит позволяют получить 65 536 комбинаций. Именно по
этой причине говорят, что двоичная система эффективнее.

Однако представим, что могло бы произойти, если бы какой-нибудь гениаль-
ный инженер придумал очень надежное электронное устройство, позволяющее
хранить разряды от 0 до 9, разделив диапазон напряжения от 0 до 10 вольт на
10 интервалов. Четыре таких устройства могли бы хранить десятичное число от 0
до 9999, то есть 10 000 комбинаций. А если бы те же устройства использовались
для хранения двоичных чисел, они могли бы содержать всего 16 комбинаций.
Естественно, в этом случае десятичная система была бы более эффективной.

**Адреса памяти**

Память состоит из **ячеек**, каждая из которых может хранить некоторую порцию
информации. Каждая ячейка имеет номер, который называется **адресом**. По
адресу программы могут ссылаться на определенную ячейку. Если память содер-
жит *n* ячеек, они будут иметь адреса от 0 до *n* - 1. Все ячейки памяти содержат
одинаковое число битов. Если ячейка состоит из *k* бит, она может содержать
любую из 2k комбинаций. На рис. 2.8 показаны 3 различных варианта организа-
ции 96-разрядной памяти. Отметим, что соседние ячейки по определению имеют
последовательные адреса.

**Адрес Адрес 1 ячейка Адрес**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0123**4**567891011 |  |  |  |  |  |  |  |  | 0 I II II II II II II 0 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 11 I I I I I I I I I I I I 1 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 2 I I I I I I I I I I I I I 2 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 3 I I I I I I I I I I I I I 3 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | **4** I I I I I I I I I I I I I **4** |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 5l I I I I I I I I I I I I 5 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 6 I I I I I I I I I I I I I |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | 7 I I I I I I I I I I I I I |
|  |  |  |  |  |  |  |  |  12 **бит ^*****6*** |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

**8 бит**

**а**

**Рис. 2.8. Три варианта организации 96-разрядной памяти**

В компьютерах, в которых используется двоичная система счисления (вклю-
чая восьмеричное и шестнадцатеричное представление двоичных чисел), адреса
памяти также выражаются в двоичных числах. Если адрес состоит из *m* бит,
максимальное число адресуемых ячеек составит 2m. Например, адрес для обра-
щения к памяти, изображенной на рис. 2.8, *а,* должен состоять по крайней мере
из 4 бит, чтобы выражать все числа от 0 до 11. При устройстве памяти, показан-
ном на рис. 2.8, *б* и 2.8, *в,* достаточно 3-разрядного адреса. Число битов в адресе
определяет максимальное количество адресуемых ячеек памяти и не зависит от
числа битов в ячейке. 12-разрядные адреса нужны и памяти из 212 ячеек по 8 бит
каждая, и памяти из 212 ячеек по 64 бит каждая.

В табл. 2.1 приведены данные о количестве битов в ячейках памяти некоторых
коммерческих компьютеров.

**Таблица 2.1.** Число битов в ячейке памяти некоторых моделей
коммерческих компьютеров

|  |  |
| --- | --- |
| **Компьютер** | **Число битов в ячейке** |
| Burroughs B1700 | 1 |
| IBM PC | 8 |
| DEC PDP-8 | 12 |
| IBM 1130 | 16 |
| DEC PDP-15 | 18 |
| XDS 940 | 24 |
| Electrologica X8 | 27 |
| XDS Sigma 9 | 32 |
| Honeywell 6180 | 36 |
| CDC 3600 | 48 |
| CDC Cyber | 60 |

Ячейка - минимальная адресуемая единица памяти. В последние годы прак-
тически все производители выпускают компьютеры с 8-разрядными ячейками,
которые называются **байтами** (также иногда встречается термин «**октет**»). Байты
группируются в **слова**. В компьютере с 32-разрядными словами на каждое слово
приходится 4 байт, а в компьютере с 64-разрядными словами - 8 байт. Такая
единица, как слово, необходима, поскольку большинство команд производят
операции над целыми словами (например, складывают два слова). Таким об-
разом, 32-разрядная машина содержит 32-разрядные регистры и команды для
манипуляций с 32-разрядными словами, тогда как 64-разрядная машина имеет
64-разрядные регистры и команды для перемещения, сложения, вычитания
и других операций над 64-разрядными словами.

**Упорядочение байтов**

Байты в слове могут нумероваться слева направо или справа налево. На первый
взгляд может показаться, что между этими двумя вариантами нет разницы, но

мы скоро увидим, что выбор имеет большое значение. На рис. 2.9, *а* изобра-
жена область памяти 32-разрядного компьютера, в котором байты пронумеро-
ваны слева направо (как у компьютеров SPARC или мэйнфреймов IBM). На
рис. 2.9, *б* показана аналогичная область памяти 32-разрядного компьютера
с нумерацией байтов справа налево (как у компьютеров Intel). В первой из
этих систем нумерация начинается с высшего порядка, в связи с чем она от-
носится к категории компьютеров **с прямым порядком следования байтов** (big
endian) - в противоположность системам **с обратным порядком следования
байтов** (little endian) (рис. 2.9, а). Между прочим, эти термины («big endian»
и «little endian») заимствованы из «Путешествий Гулливера» Свифта - он, как
мы помним, иронизировал по поводу спора политиков о том, с какого конца
нужно разбивать яйца. Впервые они были введены в научный оборот в вирту-
озной статье Коэна (1981).

**Обратный порядок**

**Прямой порядок**

**Байты**

**-32-разрядное слово—>-**

**Адрес**

О

**4**

8

12

**Байты**

**-32-разрядное слово—»-**

**а *б***

**Рис. 2.9.** Память с прямым порядком следования байтов (а);
память с обратным порядком следования байтов (б)

Важно понимать, что в обеих системах 32-разрядное целое число (напри-
мер, 6) представлено битами 110 в трех крайних правых битах слова, а остальные
29 бит представлены нулями. Если байты нумеруются слева направо, биты 110
находятся в байте 3 (или 7, или 11 и т. д.). Если байты нумеруются справа на-
лево, биты 110 находятся в байте 0 (или 4, или 8 и т. д.). В обеих системах слово,
содержащее это целое число, имеет адрес 0.

Если компьютеры содержат только целые числа, никаких сложностей не воз-
никает. Однако многие прикладные задачи требуют использования не только
целых чисел, но и цепочек символов и других типов данных. Рассмотрим, на-
пример, простую запись данных о персонале, состоящую из строки (имя сотруд-
ника) и двух целых чисел (возраст и номер отдела). Строка завершается одним
или несколькими нулевыми байтами, призванными заполнить слово целиком.
На рис. 2.10, *а* для записи «Jim Smith, 21 год, отдел 260» (1 X 256 + 4 = 260)
представлена схема с нумерацией байтов справа налево, а на рис. 2.10, *б* - с ну-
мерацией байтов слева направо.

Оба эти представления хороши и внутренне непротиворечивы. Проблемы
начинаются тогда, когда один из компьютеров пытается переслать запись на
другой компьютер по сети. Предположим, что машина с нумерацией байтов
слева направо пересылает запись на компьютер с нумерацией байтов справа

**Обратный**

**порядок**

**следования**

**байтов**

**Прямой**

**порядок**

**следования**

**байтов**

|  |  |  |  |
| --- | --- | --- | --- |
|  | м | 1 | J |
| т | 1 | м | S |
| 0 | 0 | 0 | н |
| 0 | 0 | 0 | 21 |
| 0 | 0 | 1 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| J | I | м |  |
| S | м | I | т |
| н | 0 | 0 | 0 |
| 0 | 0 | 0 | 21 |
| 0 | 0 | 1 | 4 |

**а *б***

**Передача**

**с преобразованием
прямого порядка Передача**

**следования с перестановкой**

**байтов в обратный байтов**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **М** | **I** | **J** |
| **т** | **I** | **М** | S |
| **0** | **0** | **0** | **н** |
| **21** | **0** | **0** | **0** |
| **4** | **1** | **0** | **0** |

в

|  |  |  |  |
| --- | --- | --- | --- |
| **J** | **I** | **м** |  |
| S | **м** | **I** | **т** |
| **н** | **0** | **0** | **0** |
| **0** | **0** | **0** | **21** |
| **0** | **0** | **1** | **4** |

г

О

4

**8**

12

**16**

**Рис. 2.10.** Запись данных о сотруднике для машины с прямым порядком следования
байтов (а); та же запись для машины с обратным порядком следования байтов (б);
результат передачи записи с машины с прямым порядком следования байтов на
машину с обратным порядком следования байтов (в); результат перестановки байтов

в предыдущем случае ***(г*** )

налево по одному байту, начиная с байта 0 и заканчивая байтом 19. Для про-
стоты будем считать, что биты не инвертируются при передаче. Таким образом,
байт 0 переносится из первой машины на вторую в байт 0 и т. д., как показано
на рис. 2.10, *в.*

Компьютер, получивший запись, имя печатает правильно, но возраст полу-
чается 21 X 224, и номер отдела тоже искажается. Такая ситуация возникает, по-
скольку при передаче записи порядок следования букв в слове меняется так, как
нужно, но при этом порядок следования байтов в целых числах тоже меняется,
что приводит к неверному результату.

Очевидное решение проблемы - использование программы, которая бы
инвертировала байты в слове после создания копии. Результат такой операции
представлен на рис. 2.11, г. Мы видим, что числа стали правильными, однако
строка превратилась в «MIJTIMS», при этом буква «Н» вообще расположилась
отдельно. Строка переворачивается потому, что компьютер сначала считывает
байт 0 (пробел), затем байт 1 (M) и т. д.

Простого решения не существует. Есть один способ, но он неэффективен.
(Нужно перед каждой единицей данных помещать заголовок, информирую-
щий, какой тип данных следует за ним - строка, целое и т. д. Это позволит
компьютеру-получателю производить только необходимые преобразования.)
Ясно, что отсутствие стандарта упорядочивания байтов является серьезной про-
блемой при обмене информацией между разными машинами.

**Код исправления ошибок**

В памяти компьютера из-за всплесков напряжения и по другим причинам время
от времени могут возникать ошибки. Для борьбы с ними используются специ-
альные коды, умеющие обнаруживать и исправлять ошибки. В этом случае
к каждому слову в памяти особым образом добавляются дополнительные биты.
Когда слово считывается из памяти, эти дополнительные биты проверяются, что
и позволяет обнаруживать ошибки.

Чтобы понять, как обращаться с ошибками, необходимо внимательно изучить,
что представляют собой эти ошибки. Предположим, что слово состоит из *m* бит
данных, к которым мы дополнительно прибавляем *r* бит (контрольных разрядов).
Пусть общая длина слова составит *n* бит (то есть *n* = *m* + r). Единицу из *n* бит,
содержащую *m* бит данных и *r* контрольных разрядов, часто называют **кодовым
словом**.

Для любых двух кодовых слов, например 10001001 и 10110001, можно опре-
делить, сколько соответствующих битов в них отличаются. В данном примере
таких бита три. Чтобы определить количество отличающихся битов, нужно над
двумя кодовыми словами произвести логическую операцию ИСКЛЮЧАЮЩЕЕ
ИЛИ и сосчитать число битов со значением 1 в полученном результате. Число
битовых позиций, по которым отличаются два слова, называется **интервалом
Хэмминга** [Hamming, 1950]. Если интервал Хэмминга для двух слов равен d,
значит, достаточно *d* одноразрядных ошибок, чтобы превратить одно слово
в другое. Например, интервал Хэмминга для кодовых слов 11110001 и 00110000
равен 3, поскольку для превращения первого слова во второе достаточно трех
одноразрядных ошибок.

Память состоит из m-разрядных слов, следовательно, существуют 2m вариан-
тов сочетания битов. Кодовые слова состоят из *n* бит, но из-за способа подсчета
контрольных разрядов допустимы только 2*m* из 2*n* кодовых слов. Если в памяти
обнаруживается недопустимое кодовое слово, компьютер знает, что произошла
ошибка. При наличии алгоритма подсчета контрольных разрядов можно соста-
вить полный список допустимых кодовых слов и из этого списка найти два слова,
для которых интервал Хэмминга будет минимальным. Это интервал Хэмминга
для полного кода.

Возможности проверки и исправления ошибок определенного кода зависят
от его интервала Хэмминга. Для обнаружения *d* одноразрядных ошибок необ-
ходим код с интервалом *d* + 1, поскольку *d* ошибок не могут превратить одно
допустимое кодовое слово в другое допустимое кодовое слово. Соответственно,
для исправления *d* одноразрядных ошибок необходим код с интервалом 2*d* + 1,
поскольку в этом случае допустимые кодовые слова настолько сильно отлича-
ются друг от друга, что даже если произойдет *d* изменений, изначальное кодовое
слово окажется ближе к ошибочному, чем любое другое кодовое слово, поэтому
его без труда можно будет выявить.

В качестве простого примера кода с обнаружением ошибок рассмотрим код,
в котором к данным присоединяется один **бит четности**. Бит четности выбира-
ется таким образом, чтобы число битов со значением 1 в кодовом слове было
четным (или нечетным). Интервал Хэмминга для этого кода равен 2, поскольку
любая одноразрядная ошибка приводит к кодовому слову с неправильной чет-
ностью. Другими словами, достаточно двух одноразрядных ошибок для перехода
от одного допустимого кодового слова к другому допустимому слову. Такой код
может использоваться для обнаружения одиночных ошибок. Если из памяти счи-
тывается слово с неверной четностью, поступает сигнал об ошибке. Программа
выполняться не сможет, но зато не выдаст неверных результатов.

В качестве простого примера кода исправления ошибок рассмотрим код с че-
тырьмя допустимыми кодовыми словами: 0000000000, 0000011111, 1111100000
и 1111111111.

Интервал Хэмминга для этого кода равен 5. Это значит, что он может исправ-
лять двойные ошибки. Если появляется кодовое слово 0000000111, компьютер
знает, что изначально это слово выглядело как 0000011111 (если произошло не
более двух ошибок). При появлении трех ошибок (например, слово 0000000000
меняется на 0000000111) этот метод не подходит.

Представим, что мы хотим разработать код с *m* бит данных и *r* контрольных
разрядов, позволяющий исправлять все одноразрядные ошибки. Каждое из 2mдопустимых слов имеет *n* недопустимых кодовых слов, которые отличаются от
допустимого одним битом. Они образуются инвертированием каждого из *n* бит
в n-разрядном кодовом слове. Следовательно, каждое из 2m допустимых слов
требует *n* + 1 возможных сочетаний битов, приписываемых этому слову *(n* воз-
можных ошибочных вариантов и один правильный). Поскольку общее число
различных сочетаний битов равно 2n, то (n + 1) 2m < 2n. Так как *n* = *m* + r, то

(m + *r* + 1) < 2r. Эта формула дает нижний предел числа контрольных разрядов,
необходимых для исправления одиночных ошибок. В табл. 2.2 показано необхо-
димое количество контрольных разрядов для слов разного размера.

**Таблица 2.2.** Число контрольных разрядов для кода, способного исправлять
одиночные ошибки

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер исходно-го слова** | **Количество контроль-ных разрядов** | **Общий размерслова** | **Процент увеличе-ния слова** |
| 8 | 4 | 12 | 50 |
| 16 | 5 | 21 | 31 |
| 32 | 6 | 38 | 19 |
| 64 | 7 | 71 | 11 |
| 128 | 8 | 136 | 6 |
| 256 | 9 | 265 | 4 |
| 512 | 10 | 522 | 2 |

Этого теоретического нижнего предела можно достичь, используя метод
Ричарда Хэмминга (1950). Но прежде чем обратиться к указанному алгоритму,
давайте рассмотрим простую графическую схему, которая четко иллюстрирует
идею кода исправления ошибок для 4-разрядных слов. Диаграмма Венна на
рис. 2.11 содержит 3 круга, *A, B* и C, которые вместе образуют семь секторов.
Давайте закодируем в качестве примера слово из 4 бит 1100 в секторы AB, *ABC,
AC* и *BC,* по одному биту в каждом секторе (в алфавитном порядке). Подобное
кодирование иллюстрирует рис. 2.11, *а.*

Далее мы добавим бит четности к каждому из трех пустых секторов, чтобы
сумма битов в каждом из трех кругов, *A*, *B*, и *C*, получилась четной, как пока-
зано на рис. 2.11, *б.* В круге *А* находится 4 числа: 0, 0, 1 и 1, которые в сумме
дают четное число 2. В круге *B* находятся числа 1, 1, 0 и 0, которые также при
сложении дают четное число 2. Аналогичная ситуация и для круга C. В данном
примере получилось так, что все суммы одинаковы, но вообще возможны случаи
с суммами 0 и 4. Рисунок соответствует кодовому слову, состоящему из 4 бит
данных и 3 бит четности.

Основная память

**101**

**А**

***а 6 в***

**Рис. 2.11.** Кодирование числа 1100 *(а);* добавляются биты четности *(б);*ошибка в секторе AC (в)

Предположим, что бит в секторе *AC* изменился с 0 на 1, как показано на
рис. 2.11, *в.* Компьютер обнаруживает, что круги *A* и *C* являются нечетными.
Единственный способ исправить ошибку, изменив только один бит, - возвраще-
ние значения 0 биту в секторе *AC*. Таким способом компьютер может исправлять
одиночные ошибки автоматически.

А теперь посмотрим, как может использоваться алгоритм Хэмминга при соз-
дании кодов исправления ошибок для слов любого размера. В коде Хэмминга
к слову, состоящему из *m* бит, добавляются *r* бит четности, при этом образуется
слово длиной *m* + *r* бит. Биты нумеруются с единицы (а не с нуля), причем
первым считается крайний левый. Все биты, номера которых - степени двойки,
являются битами четности; остальные используются для данных. Например,
к 16-разрядному слову нужно добавить 5 бит четности. Биты с номерами 1, 2,
4, 8 и 16 - биты четности, все остальные - биты данных. Всего слово содержит

21 бит (16 бит данных и 5 бит четности). В рассматриваемом примере мы будем

использовать проверку на четность (выбор произвольный).

Каждый бит четности позволяет проверять определенные битовые позиции.
Общее число битов со значением 1 в проверяемых позициях должно быть чет-
ным. Ниже указаны позиции проверки для каждого бита четности:

+ бит 1 проверяет биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21;

+ бит 2 проверяет биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19;

+ бит 4 проверяет биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21;

+ бит 8 проверяет биты 8, 9, 10, 11, 12, 13, 14, 15;

+ бит 16 проверяет биты 16, 17, 18, 19, 20, 21.

В общем случае бит *b* проверяется битами *b1, b2,* ..., *b,,* такими что *b1* + *b2* +
+ ... + *b,* = b. Например, бит 5 проверяется битами 1 и 4, поскольку 1 + 4 = 5.
Бит 6 проверяется битами 2 и 4, поскольку 2 + 4 = 6 и т. д.

Рисунок 2.12 иллюстрирует построение кода Хэмминга для 16-разрядного
слова 1111000010101110. Соответствующим 21-разрядным кодовым словом
является 001011100000101101110. Чтобы понять, как происходит исправление
ошибок, рассмотрим, что произойдет, если бит 5 изменит значение (напри-
мер, из-за резкого скачка напряжения). В результате вместо кодового слова
001011100000101101110 получится 001001100000101101110. Будут проверены
5 бит четности. Вот результаты.

+ неправильный бит четности 1 (биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 со-
держат пять единиц);

+ правильный бит четности 2 (биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19 содержат
шесть единиц);

+ неправильный бит четности 4 (биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21 содержат
пять единиц);

+ правильный бит четности 8 (биты 8, 9, 10, 11, 12, 13, 14, 15 содержат две
единицы);

+ правильный бит четности 16 (биты 16, 17, 18, 19, 20, 21 содержат четыре
единицы).

Общее число единиц в битах 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21 должно
быть четным, поскольку в данном случае используется проверка на четность.
Неправильным должен быть один из битов, проверяемых битом четности 1
(а именно 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21). Бит четности 4 тоже неправильный.
Это значит, что изменил значение один из следующих битов: 4, 5, 6, 7, 12, 13, 14,
15, 20, 21. Ошибка должна быть в бите, который содержится в обоих списках.
В данном случае общими являются биты 5, 7, 13, 15 и 21. Поскольку бит чет-
ности 2 правильный, биты 7 и 15 исключаются. Правильность бита четности 8
исключает наличие ошибки в бите 13. Наконец, бит 21 также исключается, по-
скольку бит четности 16 правильный. В итоге остается бит 5, в котором и кроется
ошибка. Поскольку этот бит имеет значение 1, он должен принять значение 0.
Именно таким образом исправляются ошибки.

**Слово 1111000010101110 в памяти**

folfol 1 fol 1 1 1 [ol О О О О 1 0 1 РП 0 1 1 1 о

**1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21**

**Биты
четности**

**Рис. 2.12.** Построение кода Хэмминга для слова 1111000010101110 добавлением
к битам данных пяти контрольных разрядов

Чтобы найти неправильный бит, сначала нужно подсчитать все биты четности.
Если они правильные, ошибки нет (или есть, но ошибка не однократная). Если
обнаружились неправильные биты четности, нужно сложить их номера. Сумма,
полученная в результате, даст номер позиции неправильного бита. Например,
если биты четности 1 и 4 неправильные, а 2, 8 и 16 правильные, то ошибка
произошла в бите 5 (1 + 4).

**Кэш-память**

Процессоры всегда работали быстрее, чем память. Поскольку процессоры и па-
мять совершенствуются параллельно, это несоответствие сохраняется. Поскольку
на микросхему можно помещать все больше и больше транзисторов, разработчи-
ки процессоров создают конвейерные и суперскалярные архитектуры, что еще
больше увеличивает быстродействие процессоров. Разработчики памяти обычно

используют новые технологии для увеличения емкости, а не быстродействия,
что делает разрыв еще большим. На практике такое несоответствие в скорости
работы приводит к тому, что когда процессор обращается к памяти, проходит
несколько машинных циклов, прежде чем он получит запрошенное слово. Чем
медленнее работает память, тем дольше процессору приходится ждать, тем боль-
ше циклов проходит.

Как мы уже отмечали, есть два пути решения проблемы. Самый простой из
них - начать считывать информацию из памяти и при этом продолжать выпол-
нение команд, но если какая-либо команда попытается использовать слово до
того, как оно считано из памяти, процессор должен приостановить работу. Чем
медленнее работает память, тем чаще будет возникать такая ситуация и тем боль-
ше окажется время простоя процессора. Например, если одна команда из пяти
обращается к памяти и время обращения составляет 5 циклов, время выполнения
увеличивается вдвое по сравнению с мгновенным обращением к памяти. Но если
время обращения к памяти составляет 50 циклов, то время выполнения увели-
чивается уже в 11 раз (5 циклов на выполнение команд + 50 циклов ожидания
данных из памяти).

Другое решение проблемы - сконструировать машину, которая не приостанав-
ливает работу, но следит, чтобы программы-компиляторы не использовали слова до
того, как они считаны из памяти. Однако это не так просто осуществить на прак-
тике. Часто при обработке команды загрузки машина не может выполнять другие
действия, поэтому компилятор вынужден вставлять пустые команды, которые не
производят никаких операций, но при этом занимают место в памяти. В действи-
тельности при таком подходе задержка возникает на уровне программного, а не
аппаратного обеспечения, но снижение производительности при этом такое же.

На самом деле эта проблема не технологическая, а экономическая. Инженеры
знают, как создать память, которая работает так же быстро, как процессор.
Однако ее приходится помещать прямо на микросхему процессора (поскольку
информация через шину поступает очень медленно). Размещение памяти боль-
шого объема на микросхеме процессора увеличивает его размеры, а следователь-
но, и стоимость. Впрочем, даже если бы стоимость не имела значения, все равно
существуют ограничения на размеры создаваемых процессоров. Таким образом,
приходится выбирать между быстрой памятью небольшого объема и медленной
памятью большого объема. (Мы, естественно, предпочли бы иметь быструю па-
мять большого объема и к тому же дешевую.)

Интересно отметить, что существуют технологии, объединяющие небольшую
и быструю память с большой и медленной, что позволяет по разумной цене
получить память и с высокой скоростью работы, и большой емкости. Память
небольшого объема с высокой скоростью работы называется **кэш-памятью** (от
французского слова «cacher» - «прятать»1; читается «кашэ»). Далее мы кратко
опишем, как используется кэш-память и как она работает. Более подробное
описание вы найдете в главе 4.

Основная идея кэш-памяти проста: в ней находятся слова, которые чаще
всего используются. Если процессору нужно какое-нибудь слово, сначала он

1 В английском языке слово «cash» получило значение «наличные (карманные) деньги», то есть то,
что под рукой. А уже из него и образовался термин «кэш», который относят к сверхоперативной
памяти. — *Примеч. науч. ред.*

обращается к кэш-памяти. Только в том случае, если слова там нет, он обраща-
ется к основной памяти. Если значительная часть слов находится в кэш-памяти,
среднее время доступа значительно сокращается.

Таким образом, успех или неудача зависит от того, какая часть слов находится
в кэш-памяти. Давно известно, что программы не обращаются к памяти наугад.
Если программе нужен доступ к адресу А, то скорее всего после этого ей по-
надобится доступ к адресу, расположенному поблизости от А. Практически все
команды обычной программы (за исключением команд перехода и вызова про-
цедур) вызываются из последовательных областей памяти. Кроме того, большую
часть времени программа тратит на циклы, когда ограниченный набор команд
выполняется снова и снова. Точно так же при работе с матрицами программа,
скорее всего, будет многократно обращаться к одной и той же матрице, прежде
чем перейдет к чему-либо другому.

Ситуация, когда при последовательных обращениях к памяти в течение не-
которого промежутка времени используется только небольшая ее область, на-
зывается **принципом локальности**. Этот принцип составляет основу всех систем
кэш-памяти. Идея состоит в том, что когда определенное слово вызывается из
памяти, оно вместе с соседними словами переносится в кэш-память, что позво-
ляет при очередном запросе быстро обращаться к следующим словам. Общее
устройство процессора, кэш-памяти и основной памяти иллюстрирует рис. 2.13.
Если слово считывается или записывается *k* раз, компьютеру требуется сделать
одно обращение к медленной основной памяти и *k* - 1 обращений к быстрой
кэш-памяти. Чем больше *k,* тем выше общая производительность.

Шина

**Рис. 2.13.** Кэш-память по логике вещей должна находиться между процессором
и основной памятью. В действительности существует три возможных варианта

размещения кэш-памяти

Мы можем провести и более строгие вычисления. Пусть *c* - время досту-
па к кэш-памяти, *m* - время доступа к основной памяти и *h* - **коэффициент
кэш-попаданий** (hit ratio), который показывает соотношение числа обращений
к кэш памяти и общего числа всех обращений к памяти. В нашем примере
*h* = *(k* - 1)/k. Некоторые авторы выделяют **коэффициент кэш-промахов** (miss
ratio), равный 1 - *h*.

Таким образом, мы можем вычислить среднее время доступа:

*Среднее время доступа = c +* (1 - h) *m.*

Если *h ^* 1, то есть все обращения делаются только к кэш-памяти, то время
доступа стремится к c. С другой стороны, если *h* 0, то есть каждый раз нужно
обращаться к основной памяти, то время доступа стремится к *c* + m: сначала
требуется время *c* для проверки кэш-памяти (в данном случае безуспешной),

а затем время *m* для обращения к основной памяти. В некоторых системах об-
ращение к основной памяти может начинаться параллельно с исследованием
кэш-памяти, чтобы в случае кэш-промаха цикл обращения к основной памяти
уже начался. Однако эта стратегия требует способности останавливать процесс
обращения к основной памяти в случае кэш-попадания, что усложняет разра-
ботку подобного компьютера.

Основная память и кэш-память делятся на блоки фиксированного размера
с учетом принципа локальности. Блоки внутри кэш-памяти обычно называют
**строками кэша** (cache lines). При кэш-промахе из основной памяти в кэш-память
загружается вся строка, а не только необходимое слово. Например, если строка
состоит из 64 байт, обращение к адресу 260 влечет за собой загрузку в кэш-
память всей строки (байты с 256 по 319) на случай, если через некоторое время
понадобятся другие слова из этой строки. Такой путь обращения к памяти более
эффективен, чем вызов каждого слова по отдельности, потому что однократ-
ный вызов *k* слов происходит гораздо быстрее, чем вызов одного слова *k* раз.
Кроме того, превышение строками кэша размера слова означает, что их будет
меньше, а следовательно, потребуется меньше непроизводительных затрат ре-
сурсов. Наконец, многие компьютеры (даже 32-разрядные) способны передавать
64 и 128 бит параллельно за один цикл шины.

Кэш-память очень важна для высокопроизводительных процессоров. Однако
здесь возникает ряд вопросов. Первый вопрос - объем кэш-памяти. Чем боль-
ше объем, тем лучше работает память, но тем дороже она стоит. Второй во-
прос - размер строки кэша. Кэш-память объемом 16 Кбайт можно разделить на
1024 строки по 16 байт, 2048 строк по 8 байт и т. д. Третий вопрос - механизм
организации кэш-памяти, то есть то, как она определяет, какие именно слова
находятся в ней в данный момент. Устройство кэш-памяти мы рассмотрим под-
робно в главе 4.

Четвертый вопрос - должны ли команды и данные находиться вместе в об-
щей кэш-памяти. Проще всего разработать **объединенную кэш-память** (unified
cache), в которой будут храниться и данные, и команды. В этом случае вызов
команд и данных автоматически уравновешивается. Однако в настоящее время
существует тенденция к использованию **разделенной кэш-памяти** (split cache),
когда команды хранятся в одной кэш-памяти, а данные - в другой. Такая архи-
тектура также называется **гарвардской** (Harvard architecture), поскольку идея
использования отдельной памяти для команд и отдельной памяти для данных
впервые воплотилась в компьютере Marc III, который был создан Говардом
Айкеном (Howard Aiken) в Гарварде. Современные разработчики пошли по этому
пути, поскольку сейчас широко распространены конвейерные архитектуры, а при
конвейерной организации обращения и к командам, и к данным (операндам)
должны осуществляться одновременно. Разделенная кэш-память позволяет осу-
ществлять параллельный доступ, а общая - нет. К тому же, поскольку команды
обычно не меняются во время выполнения программы, содержание кэша команд
не приходится записывать обратно в основную память.

Наконец, пятый вопрос - количество блоков кэш-памяти. В настоящее вре-
мя очень часто кэш-память первого уровня располагается прямо на микросхеме
процессора, кэш-память второго уровня - не на самой микросхеме, но в корпусе
процессора, а кэш-память третьего уровня - еще дальше от процессора.

**Сборка модулей памяти и их типы**

Со времен появления полупроводниковой памяти и до начала 90-х годов все
микросхемы памяти производились, продавались и устанавливались в виде от-
дельных микросхем. Эти микросхемы вмещали от 1 Кбит до 1 Мбит информации
и выше. В первых персональных компьютерах часто оставлялись пустые разъемы,
чтобы покупатель в случае необходимости мог вставить дополнительные микро-
схемы памяти.

В настоящее время распространен другой подход. Группа микросхем (обычно
8 или 16) монтируется на одну крошечную печатную плату и продается как один
блок. Он называется **SIMM** (Single Inline Memory Module - **модуль памяти
с односторонним расположением выводов**) или **DIMM** (Dual Inline Memory
Module - **модуль памяти с двухсторонним расположением выводов**). На платах
SIMM устанавливается один краевой разъем с 72 контактами; при этом скорость
передачи данных за один тактовый цикл составляет 32 бит. Модули DIMM, как
правило, снабжаются двумя краевыми разъемами (по одному на каждой стороне
платы) с 120 контактами; таким образом, общее количество контактов достигает
240, а скорость передачи данных возрастает до 64 бит за цикл. В настоящее время
наиболее распространенными являются DDR3 DIMM - третья версия двухско-
ростных модулей памяти. Типичный модуль DIMM изображен на рис. 2.14.

**Рис. 2.14.** 4-гигабайтный модуль DIMM с восемью 256-мегабайтными микросхемами
с каждой стороны. Другая сторона выглядит аналогично.

Обычно модули DIMM содержат 8 микросхем по 256 Мбайт каждая. Таким
образом, весь модуль вмещает 2 Гбайт информации. Во многих компьютерах
предусматривается возможность установки четырех модулей; следовательно,
при использовании модулей по 2 Гбайт общий объем памяти достигает 8 Гбайт
(и более при использовании модулей большей емкости).

В портативных компьютерах обычно используется модуль DIMM меньшего
размера, который называется **SO-DIMM** (Small Outline DIMM). Модули SIMM
и DIMM могут содержать бит четности или код исправления ошибок, однако,
поскольку вероятность возникновения ошибок в модуле составляет примерно
одну ошибку за 10 лет, в большинстве обычных компьютеров схемы обнаружения
и исправления ошибок не применяются.

**Вспомогательная память**

Каков бы ни был объем основной памяти, ее все равно будет мало. Такова уж
наша природа, мы всегда хотим сохранить в памяти компьютера больше данных,
чем она может вместить. С развитием технологий людям приходят в голову такие
вещи, которые раньше считались совершенно фантастическими. Например, мож-

но вообразить, что Библиотека Конгресса решила представить в цифровой форме
и продать полный текст со всеми иллюстрациями всех хранящихся в ней изданий
(«Все человеческие знания всего за $299,95 доллара»). В среднем каждая книга
содержит 1 Мбайт текста и 1 Мбайт упакованных иллюстраций. Таким образом,
для размещения 50 млн книг понадобится 1014 байт или 100 Тбайт памяти. Для
хранения всех существующих художественных фильмов (50 000) необходимо
примерно столько же места. Такой объем информации в настоящее время не-
возможно разместить в основной памяти и вряд удастся это сделать в будущем
(по крайней мере, в ближайшие несколько десятилетий).

**Иерархическая структура памяти**

Иерархическая структура памяти является традиционным решением проблемы
хранения больших объемов данных (рис. 2.15). На самом верху иерархии нахо-
дятся регистры процессора. Доступ к регистрам осуществляется быстрее всего.
Дальше идет кэш-память, объем которой сейчас составляет от 32 Кбайт до не-
скольких мегабайт. Затем следует основная память, объем которой в настоящее
время лежит в диапазоне от 1 Гбайт до сотен гигабайт. Затем идут магнитные
диски и твердотельные накопители для долгосрочного хранения данных. Нижний
уровень иерархии занимают накопители на магнитной ленте и оптические диски
для хранения архивов.

По мере продвижения сверху вниз по иерархии меняются три параметра. Во-
первых, увеличивается время доступа. Доступ к регистрам занимает несколько
наносекунд, доступ к кэш-памяти - немного больше, доступ к основной памя-
ти - несколько десятков наносекунд. Дальше идет большой разрыв: доступ к дис-
кам происходит по крайней мере в 10 раз медленнее для твердотельных дисков
и в сотни раз медленнее для магнитных дисков. Время доступа к магнитным
лентам и оптическим дискам вообще может измеряться в секундах (поскольку
эти накопители информации еще нужно взять и поместить в соответствующее
устройство).

Во-вторых, растет объем памяти. Регистры могут содержать в лучшем слу-
чае 128 байт, кэш-память - десятки мегабайт, основная память - гигабайты,
магнитные диски - терабайты. Магнитные ленты и оптические диски хранятся
автономно от компьютера, поэтому их совокупный объем ограничивается только
финансовыми возможностями владельца.

В третьих, увеличивается количество битов, которое вы получаете за один
доллар. Стоимость объема основной памяти измеряется в долларах за мега-
байт1, твердотельных накопителей - в долларах за гигабайт, магнитных дисков
и лент - в центах за гигабайт или еще дешевле.

Регистры, кэш-память и основную память мы уже рассмотрели. магнитные диски и твердотельные накопителе мы уже прошли раньше. Накопители на магнитных лентах мы рассматривать не будем, поскольку используются они редко; к тому же о них практически нечего сказать.