Содержание

Введение ... 12

1. Предварительные сведения ... 15

1.1. Что потребуется для работы с ассемблером ... 15

1.2. Представление данных в компьютерах ... 16

1.2.1. Двоичная система счисления ... 17

1.2.2. Биты, байты и слова ... 17

1.2.3. Шестнадцатеричная система счисления ... 19

1.2.4. Числа со знаком ... 19

1.2.5. Логические операции ... 20

1.2.6. Коды символов ... 21

1.2.7. Организация памяти ... 21

2. Процессоры Intel в реальном режиме ... 23

2.1. Регистры процессора ... 23

2.1.1. Регистры общего назначения ... 23

2.1.2. Сегментные регистры ... 25

2.1.3. Стек ... 26

2.1.4. Регистр флагов ... 27

2.2. Способы адресации ... 28

2.2.1. Регистровая адресация ... 28

2.2.2. Непосредственная адресация ... 28

2.2.3. Прямая адресация ... 29

2.2.4. Косвенная адресация ... 29

2.2.5. Адресация по базе со сдвигом ... 30

2.2.6. Косвенная адресация с масштабированием ... 30

2.2.7. Адресация по базе с индексированием ... 31

2.2.8. Адресация по базе с индексированием и масштабированием ... 31

2.3. Основные непривилегированные команды ... 32

2.3.1. Пересылка данных ... 32

2.3.2. Двоичная арифметика ... 40

2.3.3. Десятичная арифметика ... 45
2.3.4. Логические операции ... 48
2.3.5. Сдвиговые операции ... 50
2.3.6. Операции над битами и байтами ... 53
2.3.7. Команды передачи управления ... 55
2.3.8. Строковые операции ... 63
2.3.9. Управление флагами ... 69
2.3.10. Загрузка сегментных регистров ... 72
2.3.11. Другие команды ... 72
2.4. Числа с плавающей запятой ... 77
2.4.1. Типы данных FPU ... 77
2.4.2. Регистры FPU ... 79
2.4.3. Исключения FPU ... 82
2.4.4. Команды пересылки данных FPU ... 83
2.4.5. Базовая арифметика FPU ... 85
2.4.6. Команды сравнения FPU ... 90
2.4.7. Трансцендентные операции FPU ... 92
2.4.8. Константы FPU ... 95
2.4.9. Команды управления FPU ... 95
2.5. Расширение IA MMX ... 100
2.5.1. Регистры MMX ... 100
2.5.2. Типы данных MMX ... 101
2.5.3. Команды пересылки данных MMX ... 101
2.5.4. Команды преобразования типов MMX ... 102
2.5.5. Арифметические операции MMX ... 104
2.5.6. Команды сравнения MMX ... 106
2.5.7. Логические операции MMX ... 107
2.5.8. Сдвиговые операции MMX ... 108
2.5.9. Команды управления состоянием MMX ... 109
2.5.10. Расширение AMD 3D ... 109

3. Директивы и операторы ассемблера ... 112
3.1. Структура программы ... 112
3.2. Директивы распределения памяти ... 114
3.2.1. Псевдокоманды определения переменных ... 114
3.2.2. Структуры ... 115

3.3. Организация программы 116
3.3.1. Сегменты 116
3.3.2. Модели памяти и упрощенные директивы определения сегментов 119
3.3.3. Порядок загрузки сегментов 121
3.3.4. Процедуры 122
3.3.5. Конец программы 123
3.3.6. Директивы задания набора допустимых команд 123
3.3.7. Директивы управления программным счетчиком 124
3.3.8. Глобальные объявления 125
3.3.9. Условное ассемблирование 126
3.4. Выражения 128
3.5. Макроопределения 130
3.5.1. Блоки повторений 131
3.5.2. Макрооператоры 133
3.5.3. Другие директивы, используемые в макроопределениях 134
3.6. Другие директивы 134
3.6.1. Управление файлами 134
3.6.2. Управление листингом 134
3.6.3. Комментарии 135

4. Основы программирования для MS-DOS 136
4.1. Программа типа COM 137
4.2. Программа типа EXE 139
4.3. Вывод на экран в текстовом режиме 141
4.3.1. Средства DOS 141
4.3.2. Средства BIOS 144
4.3.3. Прямая работа с видеопамятью 149
4.4. Ввод с клавиатуры 151
4.4.1. Средства DOS 151
4.4.2. Средства BIOS 159
4.5. Графические видеорежимы 162
4.5.1. Работа с VGA-режимами 162
4.5.2. Работа с SVGA-режимами 167

4.6. Работа с мышью 179
4.7. Другие устройства 185
4.7.1. Системный таймер 185
4.7.2. Последовательный порт 192
4.7.3. Параллельный порт 196
4.8. Работа с файлами 198
4.8.1. Создание и открытие файлов 198
4.8.2. Чтение и запись в файл 201
4.8.3. Закрытие и удаление файла 203
4.8.4. Поиск файлов 204
4.8.5. Управление файловой системой 208
4.9. Управление памятью 211
4.9.1. Обычная память 211
4.9.2. Область памяти UMB 212
4.9.3. Область памяти HMA 213
4.9.4. Интерфейс EMS 214
4.9.5. Интерфейс XMS 215
4.10. Загрузка и выполнение программ 220
4.11. Командные параметры и переменные среды 227

5. Более сложные приемы программирования 232
5.1. Управляющие структуры 232
5.1.1. Структуры IF... THEN... ELSE 232
5.1.2. Структуры CASE 233
5.1.3. Конечные автоматы 234
5.1.4. Циклы 235
5.2. Процедуры и функции 236
5.2.1. Передача параметров 236
5.2.2. Локальные переменные 242
5.3. Вложенные процедуры 243
5.3.1. Вложенные процедуры со статическими ссылками 243
5.3.2. Вложенные процедуры с дисплеями 245

5.4. Целочисленная арифметика повышенной точности ... 246
5.4.1. Сложение и вычитание ... 246
5.4.2. Сравнение ... 247
5.4.3. Умножение ... 248
5.4.4. Деление ... 249
5.5. Вычисления с фиксированной запятой ... 250
5.5.1. Сложение и вычитание ... 250
5.5.2. Умножение ... 251
5.5.3. Деление ... 251
5.5.4. Трансцендентные функции ... 251
5.6. Вычисления с плавающей запятой ... 256
5.7. Популярные алгоритмы ... 261
5.7.1. Генераторы случайных чисел ... 261
5.7.2. Сортировки ... 265
5.8. Перехват прерываний ... 269
5.8.1. Обработчики прерываний ... 270
5.8.2. Прерывания от внешних устройств ... 274
5.8.3. Повторная входимость ... 278
5.9. Резидентные программы ... 281
5.9.1. Пассивная резидентная программа ... 282
5.9.2. Мультиплексорное прерывание ... 288
5.9.3. Выгрузка резидентной программы из памяти ... 304
5.9.4. Полурезидентные программы ... 321
5.9.5. Взаимодействие между процессами ... 326
5.10. Программирование на уровне портов ввода-вывода ... 335
5.10.1. Клавиатура ... 335
5.10.2. Последовательный порт ... 339
5.10.3. Параллельный порт ... 345
5.10.4. Видеоадаптеры VGA ... 347
5.10.5. Таймер ... 363
5.10.6. Динамик ... 368
5.10.7. Часы реального времени и CMOS-память ... 369

5.10.8. Звуковые платы ... 373
5.10.9. Контроллер DMA ... 381
5.10.10. Контроллер прерываний ... 389
5.10.11. Джойстик ... 395
5.11. Драйверы устройств в DOS ... 397
5.11.1. Символьные устройства ... 400
5.11.2. Блочные устройства ... 409

6. Программирование в защищенном режиме ... 414
6.1. Адресация в защищенном режиме ... 414
6.2. Интерфейс VCPI ... 418
6.3. Интерфейс DPMI ... 420
6.3.1. Переключение в защищенный режим ... 421
6.3.2. Функции DPMI управления дескрипторами ... 422
6.3.3. Передача управления между режимами в DPMI ... 424
6.3.4. Обработчики прерываний ... 426
6.3.5. Пример программы ... 428
6.4. Расширители DOS ... 431
6.4.1. Способы объединения программы с расширителем ... 432
6.4.2. Управление памятью в DPMI ... 433
6.4.3. Вывод на экран через линейный кадровый буфер ... 435

7. Программирование для Windows 95 и Windows NT ... 442
7.1. Первая программа ... 442
7.2. Консольные приложения ... 446
7.3. Графические приложения ... 451
7.3.1. Окно типа MessageBox ... 451
7.3.2. Окна ... 452
7.3.3. Меню ... 457
7.3.4. Диалоги ... 462
7.3.5. Полноценное приложение ... 467
7.4. Динамические библиотеки ... 483
7.5. Драйверы устройств ... 489

8. Ассемблер и языки высокого уровня 492
8.1. Передача параметров 492
8.1.1. Конвенция Pascal 492
8.1.2. Конвенция C 493
8.1.3. Смешанные конвенции 495
8.2. Искажение имен 495
8.3. Встроенный ассемблер 496
8.3.1. Встроенный ассемблер в Pascal 496
8.3.2. Встроенный ассемблер в C 496

9. Оптимизация 498
9.1. Высокоуровневая оптимизация 498
9.2. Оптимизация на среднем уровне 498
9.2.1. Оптимизация циклов 499
9.3. Низкоуровневая оптимизация 501
9.3.1. Общие принципы низкоуровневой оптимизации 501
9.3.2. Особенности архитектуры процессоров Pentium и Pentium MMX 505
9.3.3. Особенности архитектуры процессоров Pentium Pro и Pentium II 507

10. Процессоры Intel в защищенном режиме 511
10.1. Регистры 511
10.1.1. Системные флаги 511
10.1.2. Регистры управления памятью 513
10.1.3. Регистры управления процессором 513
10.1.4. Отладочные регистры 515
10.1.5. Машинно-специфичные регистры 517
10.2. Системные и привилегированные команды 517
10.3. Вход и выход из защищенного режима 525
10.4. Сегментная адресация 527
10.4.1. Модель памяти в защищенном режиме 527
10.4.2. Селектор 528

10.4.3. Дескрипторы ... 528
10.4.4. Пример программы ... 530
10.4.5. Нереальный режим ... 535
10.5. Обработка прерываний и исключений ... 537
10.6. Страничная адресация ... 548
10.7. Механизм защиты ... 555
10.7.1. Проверка лимитов ... 556
10.7.2. Проверка типа сегмента ... 557
10.7.3. Проверка привилегий ... 557
10.7.4. Выполнение привилегированных команд ... 558
10.7.5. Защита на уровне страниц ... 559
10.8. Управление задачами ... 559
10.8.1. Сегмент состояния задачи ... 560
10.8.2. Переключение задач ... 561
10.9. Режим виртуального 8086 ... 568
10.9.1. Прерывания в V86 ... 568
10.9.2. Ввод-вывод в V86 ... 569

11. Программирование на ассемблере в среде UNIX ... 570
11.1. Синтаксис AT&T ... 571
11.1.1. Основные правила ... 571
11.1.2. Запись команд ... 572
11.1.3. Адресация ... 574
11.2. Операторы ассемблера ... 574
11.2.1. Префиксные, или унарные операторы ... 575
11.2.2. Инфиксные, или бинарные операторы ... 575
11.3. Директивы ассемблера ... 575
11.3.1. Директивы определения данных ... 575
11.3.2. Директивы управления символами ... 576
11.3.3. Директивы определения секций ... 577
11.3.4. Директивы управления разрядностью ... 578
11.3.5. Директивы управления программным указателем ... 578
11.3.6. Директивы управления листингом ... 579

11.3.7. Директивы управления ассемблированием 579
11.3.8. Блоки повторения 579
11.3.9. Макроопределения 580
11.4. Программирование с использованием libc 581
11.5. Программирование без использования libc 583

12.Заключение 587

Приложение 1. Таблицы символов 588
1. Символы ASCII 588
2. Управляющие символы ASCII 589
3. Кодировки второй половины ASCII 590
4. Коды символов расширенного ASCII 593

Приложение 2. Команды Intel 80x86 596
1. Общая информация о кодах команд 596
1.1. Общий формат команды процессора Intel 596
1.2. Значения полей кода команды 596
1.3. Значения поля ModRM 598
1.4. Значения поля SIB 599
2. Общая информация о скоростях выполнения 599
3. Префиксы 601
4. Команды процессоров Intel 8088 – Pentium II 602

Список используемых сокращений 624

Глоссарий 627

Алфавитный указатель 630

Введение

Первый вопрос, который задает себе человек, впервые услышавший об этом языке программирования,– а зачем он, собственно, нужен? Особенно теперь, когда все пишут на C/C++, Delphi или других языках высокого уровня? Ведь очень многое можно создать на C, но ни один язык, даже такой популярный, не может претендовать на то, чтобы на нем можно было написать действительно «все».

Итак, на ассемблере пишут:

- *все, что требует максимальной скорости выполнения:* основные компоненты компьютерных игр, ядра операционных систем реального времени и просто критические участки программ;
- *все, что взаимодействует с внешними устройствами:* драйверы, программы, работающие напрямую с портами, звуковыми и видеоплатами;
- *все, что использует полностью возможности процессора:* ядра многозадачных операционных систем, DPMI-серверы и вообще любые программы, переводящие процессор в защищенный режим;
- *все, что полностью использует возможности операционной системы:* вирусы и антивирусы, защиты от несанкционированного доступа, программы, обходящие эти защиты, и программы, защищающиеся от этих программ;
- *и многое другое*. Стоит познакомиться с ассемблером поближе, как оказывается, что многое из того, что обычно пишут на языках высокого уровня, лучше, проще и быстрее написать на ассемблере.

Как же так? – спросите вы, прочитав последний пункт. – Ведь всем известно, что ассемблер – неудобный язык, и писать на нем долго и сложно! Давайте перечислим мотивы, которые обычно выдвигаются в доказательство того, что ассемблер не нужен.

Говорят, что ассемблер трудно выучить. Любой язык программирования трудно выучить. Легко выучить C или Delphi после Паскаля, потому что они похожи. А попробуйте освоить Lisp, Forth или Prolog, и окажется, что ассемблер в действительности даже проще, чем любой совершенно незнакомый язык программирования.

Говорят, что программы на ассемблере трудно понять. Разумеется, на ассемблере легко написать неудобочитаемую программу... точно так же, как и на любом другом языке! Если вы знаете язык и если автор программы не старался ее запутать, то понять программу будет не сложнее, чем если бы она была написана на Бейсике.

Говорят, что программы на ассемблере трудно отлаживать. Программы на ассемблере легко отлаживать – опять же при условии, что вы знаете язык. Более того, знание ассемблера часто помогает отлаживать программы на других языках, потому что оно дает представление о том, как на самом деле функционирует компьютер и что происходит при выполнении команд языка высокого уровня.

Говорят, что современные компьютеры такие быстрые, что ассемблер больше не нужен. Каким бы быстрым ни был компьютер, пользователю всегда хочется большей скорости, иначе не наблюдалось бы постоянного спроса на еще более быстрые компьютеры. И самой быстрой программой на данном оборудовании всегда будет программа, написанная на ассемблере.

Говорят, что писать на ассемблере сложно. В этом есть доля правды. Очень часто авторы программ на ассемблере «изобретают велосипеды», программируя заново элементарные процедуры типа форматированного вывода на экран или генератора случайных чисел, в то время как программисты на C просто вызывают стандартные функции. Библиотеки таких функций существуют и для ассемблера, но они не стандартизованы и не распространяются вместе с компиляторами.

Говорят, что программы на ассемблере не переносятся. Действительно, в этом заключается самая сильная и самая слабая сторона ассемблера. С одной стороны, благодаря этой особенности программы на ассемблере используют возможности компьютера с наибольшей полнотой; с другой стороны, эти же программы не будут работать на другом компьютере. Стоит заметить, что и другие языки часто не гарантируют переносимости – та же программа на C, написанная, например, под Windows 95, не скомпилируется ни на Macintosh, ни на SGI.

Далеко не все, что говорят об ассемблере, является правдой, и далеко не все, кто говорят об ассемблере, на самом деле знают его. Но даже ярые противники согласятся с тем, что программы на ассемблере – самые быстрые, самые маленькие и могут то, что не под силу программам, созданным на любом другом языке программирования.

Эта книга рассчитана на читателей с разным уровнем подготовки – от начинающих, которые хотят познакомиться с ассемблером серьезно или желают лишь написать пару программ, выполняющих необычные трюки с компьютером, до профессиональных программистов, которые тоже могут найти здесь интересные разделы. Почти все, что надо знать об ассемблере, где-нибудь да объяснено, а также объяснено многое из того, что не заботит большинство программистов. С одной стороны, чтобы написать

простую программу, не нужно знать язык и устройство процессора в совершенстве, но, с другой стороны, по-настоящему серьезная работа потребует и серьезной подготовки. Уровень сложности в этой книге возрастает от начала к концу, но в первой ее половине отдельные абзацы будут помечены специальным знаком  , который означает, что данный абзац лучше пропустить при чтении, если вы знакомитесь с ассемблером впервые. Впрочем, если у вас есть время и желание выучить ассемблер с нуля,– читайте все по порядку. Если же вам хочется немедленно приступить к написанию программ, начните сразу с главы 4.1, но будьте готовы к тому, что иногда придется возвращаться к предыдущим главам за более подробным описанием тех или иных команд. И наконец, если вам уже доводилось программировать на ассемблере,– выбирайте то, что интересно.

1. Предварительные сведения

1.1. Что потребуется для работы с ассемблером

Прежде всего вам потребуется *ассемблер*. Здесь самое время сказать, что на самом деле язык программирования, которым мы собираемся заниматься, называется «язык ассемблера» (assembly language). Ассемблер – это программа, которая переводит текст с языка, понятного человеку, в язык, понятный процессору, то есть говорят, что она переводит язык ассемблера в машинный код. Однако сначала в повседневной речи, а затем и в литературе слово «ассемблер» стало также и названием самого языка программирования. Понятно, что, когда говорят «программа на ассемблере», имеют в виду язык, а когда говорят «макроассемблер версии 6.13», имеют в виду программу. Вместе с ассемблером обязательно должна быть еще одна программа – компоновщик (linker), которая и создает исполнимые файлы из одного или нескольких объектных модулей, полученных после запуска ассемблера. Помимо этого для разных целей могут потребоваться дополнительные вспомогательные программы – компиляторы ресурсов, расширители DOS и тому подобное (см. табл. 1).

Трудно говорить о том, продукция какой из этих трех компаний однозначно лучше. С точки зрения удобства компиляции TASM лучше подходит для создания 16-битных программ для DOS, WASM – для 32-битных программ для DOS, MASM – для Windows. С точки зрения удобства программирования развитость языковых средств растет в ряду WASM – MASM – TASM. Все примеры программ в этой книге построены так, что можно использовать любой из этих компиляторов.

Разумеется, существуют и другие компиляторы, например бесплатно распространяемый в сети Internet NASM или условно бесплатный A86, но пользоваться ими проще, если вы уже знаете турбо– или макроассемблер. Бесплатно распространяемый GNU ассемблер, gas, вообще использует совершенно непохожий синтаксис, который будет рассмотрен в главе 11, рассказывающей о программировании для UNIX.

Во всех программах встречаются ошибки. Если вы собираетесь не только попробовать примеры из книги, но и написать что-то свое, то вам рано или поздно обязательно потребуется отладчик. Кроме поиска ошибок отладчики иногда применяют и для того, чтобы исследовать работу существующих программ. Безусловно, самый мощный отладчик на сегодняшний день – SoftICE от NuMega Software. Это фактически единственный

Таблица 1. Ассемблеры и сопутствующие программы

	Microsoft	Bî rland	Watcom
DOS, 16 бит	masm или ml, link (16 бит)	tasm tlink	wasm wlink
DOS, 32 бита	masm или ml, link (32 бита) и dosx link (16 бит) и dos32	tasm tlink wdosx или dos32	wasm wlink dos4gw, pmodew, zrdx или wdosx
Windows EXE	masm386 или ml link (32 бита) rc	tasm tlink32 brcc32	wasm wlink wrc
Windows DLL	masm386 или ml link (32 бита)	tasm tlink32 implib	wasm wlink wlib

отладчик для Windows 95/NT, позволяющий исследовать все – от ядра Windows до программ на C ++, поддерживающий одновременно 16- и 32-битный код и многое другое. Другие популярные отладчики, распространяемые вместе с соответствующими ассемблерами,– Codeview (MS), Turbo Debugger (Borland) и Watcom Debugger (Watcom).

Еще одна особенность ассемблера, отличающая его от всех остальных языков программирования,– возможность дизассемблирования. То есть, имея исполнимый файл, с помощью специальной программы *(дизассемблера)* почти всегда можно получить исходный текст на ассемблере. Например, можно дизассемблировать BIOS вашего компьютера и узнать, как выполняется переключение видеорежимов, или драйвер для DOS, чтобы написать такой же для Windows. Дизассемблер не необходим, но иногда оказывается удобно иметь его под рукой. Лучшие дизассемблеры на сегодняшний день – Sourcer от V Communications и IDA.

И наконец, последняя необязательная, но крайне полезная утилита – шестнадцатеричный редактор. Многие такие редакторы (hiew, proview, lview, hexit) тоже имеют встроенный дизассемблер, так что можно, например, открыв в таком редакторе свою программу, посмотреть, как скомпилировался тот или иной участок программы, поправить какую-нибудь команду ассемблера или изменить значения констант и тут же, без перекомпиляции, запустить программу, чтобы посмотреть на результат изменений.

1.2. Представление данных в компьютерах

Для того чтобы освоить программирование на ассемблере, неизбежно приходится знакомиться с двоичными и шестнадцатеричными числами.

Таблица 2. Перевод числа из десятичной системы в двоичную

	Остаток:	Разряд:
150/2 = 75	0	1
75/2 = 37	1	0
37/2 = 18	1	0
18/2 = 9	0	1
9/2 = 4	1	0
4/2 = 2	0	1
2/2 = 1	0	1
1/2 = 0	1	0
Результат:		10010110b

В некоторых случаях в тексте программы можно обойтись и обычными десятичными числами, но без понимания того, как на самом деле хранятся данные в памяти компьютера, очень трудно использовать логические и битовые операции, упакованные форматы данных и многое другое.

1.2.1. Двоичная система счисления

Практически все существующие сейчас компьютерные системы, включая Intel, используют для всех вычислений двоичную систему счисления. В их электрических цепях напряжение может принимать два значения, и эти значения назвали нулем и единицей. Двоичная система счисления как раз и использует только эти две цифры, а вместо степеней десяти, как в обычной десятичной системе, здесь используют степени двойки. Чтобы перевести двоичное число в десятичное, надо сложить двойки в степенях, соответствующих позициям, где в двоичном стоят единицы. Например:

$$10010110b=1*2^7+0*2^6+0*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0=128+16+4+2=150$$

Чтобы перевести десятичное число в двоичное, можно, например, просто делить его на 2, записывая 0 каждый раз, когда число делится на два, и 1, когда не делится (табл. 2).

Чтобы отличать двоичные числа от десятичных, в ассемблерных программах в конце каждого двоичного числа ставят букву «b».

1.2.2. Биты, байты и слова

Минимальная единица информации называется *битом*. Бит может принимать только два значения – обычно 0 и 1. На самом деле эти значения совершенно необязательны – один бит может принимать значения «да» и «нет», показывать присутствие и отсутствие жесткого диска, является

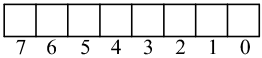

Рис. 1. Байт

ли персонаж игры магом или воином – важно лишь то, что бит имеет только два значения. Но далеко не все величины принимают только два значения, а значит, для их описания нельзя обойтись одним битом.

Единица информации размером восемь бит называется *байтом*. Байт – это минимальный объем данных, который реально может использовать компьютерная программа. Даже чтобы изменить значение одного бита в памяти, надо сначала считать байт, содержащий его. Биты в байте нумеруют справа налево, от нуля до семи, нулевой бит часто называют младшим битом, а седьмой – старшим.

Так как всего в байте восемь бит, байт может принимать до $2^8 = 256$ разных значений. Байт используют для представления целых чисел от 0 до 255 (тип unsigned char в C), целых чисел со знаком от –128 до +127 (тип signed char в C), набора символов ASCII (тип char в C) или переменных, принимающих менее 256 значений, например для представления десятичных чисел от 0 до 99. Следующий по размеру базовый тип данных – *слово*. Размер одного слова в процессорах Intel – два байта.

Биты с 0 по 7 составляют *младший байт* слова, а биты с 8 по 15 – *старший*. В слове содержится 16 бит, а значит, оно может принимать до $2^{16} = 65\,536$ разных значений. Слова используют для представления целых чисел без знака со значениями 0 – 65 535 (тип unsigned short в C), целых чисел со знаком со значениями от –32 768 до +32 767 (тип short int в C), адресов сегментов и смещений при 16-битной адресации. Два слова подряд образуют *двойное слово*, состоящее из 32 бит, а два двойных слова составляют одно *учетверенное слово* (64 бита). Байты, слова и двойные слова – основные типы данных, с которыми мы будем работать.

Еще одно важное замечание: в компьютерах, использующих процессоры Intel, все данные хранятся так, что младший байт находится по младшему адресу, так что слова записываются задом наперед, то есть сначала (по младшему адресу) записывают последний (младший) байт, а потом (по старшему адресу) записывают первый (старший) байт. Если из программы всегда обращаться к слову как к слову, а к двойному слову как к двойному слову, это не оказывает никакого влияния. Но если вы хотите прочитать первый (старший) байт из слова в памяти, придется увеличить адрес на 1. Двойные и учетверенные слова записываются так же – от младшего байта к старшему.

Рис. 2. Слово

1.2.3. Шестнадцатеричная система счисления

Главное неудобство двоичной системы счисления – это размеры чисел, с которыми приходится обращаться. На практике с двоичными числами работают, только если необходимо следить за значениями отдельных бит, а когда размеры переменных превышают хотя бы четыре бита, используется шестнадцатеричная система. Эта система хороша тем, что она гораздо более компактна, компактнее десятичной, и тем, что перевод в двоичную систему и обратно происходит очень легко. В шестнадцатеричной системе используется 16 «цифр»: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, и номер позиции цифры в числе соответствует степени, в которую надо возвести число 16, так что:

```
96h = 9 * 16 + 6 = 150
```

Перевод в двоичную систему и обратно осуществляется крайне просто – вместо каждой шестнадцатеричной цифры подставляют соответствующее четырехзначное двоичное число:

```
9h = 1001b, 6h = 0110b, 96h = 10010110b
```

В ассемблерных программах при записи чисел, начинающихся с A, B, C, D, E, F, в начале приписывается цифра 0, чтобы нельзя было спутать такое число с названием переменной или другим идентификатором. После шестнадцатеричных чисел ставится буква «h» (см. табл. 3).

1.2.4. Числа со знаком

Легко использовать байты, слова или двойные слова для представления целых положительных чисел – от 0 до 255, 65 535 или 4 294 967 295 соответственно. Чтобы использовать те же самые байты или слова для представления отрицательных чисел, существует специальная операция, известная как дополнение до двух. Для изменения знака числа выполняют инверсию, то есть заменяют в двоичном представлении числа все единицы нулями и нули единицами, а затем прибавляют 1.

Например, пусть используются переменные типа слова:

```
150 = 0096h = 0000 0000 1001 0110b
инверсия дает: 1111 1111 0110 1001b
+1 = 1111 1111 0110 1010b = 0FF6Ah
```

Таблица 3. Двоичные и шестнадцатеричные числа

Десятичное	Двоичное	Шестнадцатеричное
0	0000b	00h
1	0001b	01h
2	0010b	02h
3	0011b	03h
4	0100b	04h
5	0101b	05h
6	0110b	06h
7	0111b	07h
8	1000b	08h
9	1001b	09h
10	1010b	0Ah
11	1011b	0Bh
12	1100b	0Ch
13	1101b	0Dh
14	1110b	0Eh
15	1111b	0Fh
16	10000b	10h

Проверим, что полученное число на самом деле –150: сумма с +150 должна быть равна нулю:

```
+150 + (-150) = 0096h + FF6Ah = 10000h
```

Единица в 16-м разряде не помещается в слово, и значит, мы действительно получили 0. В этом формате старший (7-й, 15-й, 31-й для байта, слова, двойного слова соответственно) бит всегда соответствует знаку числа 0 – для положительных и 1 – для отрицательных. Таким образом, схема с использованием дополнения до двух выделяет для положительных и отрицательных чисел равные диапазоны: –128 – +127 – для байта, –32 768 – +32 767 – для слов, –2 147 483 648 – +2 147 483 647 – для двойных слов.

1.2.5. Логические операции

Один из широко распространенных вариантов значений, которые может принимать один бит,– это значения «правда» и «ложь», используемые в логике, откуда происходят так называемые «логические операции» над битами. Так, если объединить «правду» и «правду» –

Таблица 4. Логические операции

И	ИЛИ	Исключающее ИЛИ	Отрицание
0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

получится «правда», а если объединить «правду» и «ложь» – «правды» не получится. В ассемблере нам встретятся четыре основные операции – И (AND), ИЛИ (OR), исключающее ИЛИ (XOR) и отрицание (NOT), действие которых приводится в таблице 4.

Все эти операции побитовые, поэтому, чтобы выполнить логическую операцию над числом, надо перевести его в двоичный формат и выполнить операцию над каждым битом, например:

```
96h AND 0Fh = 10010110b AND 00001111b = 00000110b = 06h
```

1.2.6. Коды символов

Для представления всех букв, цифр и знаков, появляющихся на экране компьютера, обычно используется всего один байт. Символы, соответствующие значениям от 0 до 127, то есть первой половине всех возможных значений байта, были стандартизованы и названы символами ASCII (хотя часто кодами ASCII называют всю таблицу из 256 символов). Сюда входят некоторые управляющие коды (символ с кодом 0Dh – конец строки), знаки препинания, цифры (символы с кодами 30h – 39h), большие (41h – 5Ah) и маленькие (61h – 7Ah) латинские буквы. Вторая половина символьных кодов используется для алфавитов других языков и псевдографики, и набор и порядок символов в ней различаются в разных странах и даже в пределах одной страны. Например, для букв одного только русского языка существует пять разных вариантов размещения во второй половине таблицы символов ASCII. Эти таблицы символов приведены в приложении 1. Существует также стандарт, использующий слова для хранения кодов символов, известный как UNICODE или UCS–2, и даже двойные слова (UCS–4), но мы пока не будем на нем останавливаться.

1.2.7. Организация памяти

Память с точки зрения процессора представляет собой последовательность байт, каждому из которых присвоен уникальный адрес. Он может принимать значения от 0 до $2^{32}-1$ (4 гигабайта). Программы же могут работать с памятью как с одним непрерывным массивом (модель памяти flat) или как с несколькими массивами (сегментированные

модели памяти). Во втором случае для задания адреса любого байта требуется два числа – адрес начала массива и адрес искомого байта внутри массива. Помимо основной памяти программы могут использовать *регистры* – специальные ячейки памяти, расположенные физически внутри процессора, доступ к которым осуществляется не по адресам, а по именам. Но здесь мы вплотную подходим к рассмотрению собственно работы процессора, подробнее о чем – в следующей главе.