Глава 3

Введение в язык ассемблера

В этой главе...

- Представление данных
- Основы языка ассемблера
- Разработка программы на языке ассемблера
- Работа в DOS под Windows NT
- Инструментальные средства
- Пример простой программы
- Резюме
- Контрольные вопросы

Язык ассемблера помогает раскрыть все секреты аппаратного и программного обеспечения. С его помощью можно получить представление о том, как аппаратная часть взаимодействует с операционной системой и как прикладные программы обращаются к операционной системе. Большинство программистов работают с языками высокого уровня, где отдельное утверждение преобразовывается во множество процессорных команд. Ассемблер — язык машинного уровня; каждая команда непосредственно интерпретируется в команду процессора, что дает основание считать его языком низкого уровня.

Наиболее часто язык ассемблера используется для непосредственного управления операционной системой или для прямого доступа к аппаратуре. Он необходим также при оптимизации критических блоков в прикладных программах с целью повышения их быстродействия.

Представление данных

Поскольку общение с компьютером происходит на машинном уровне, необходимо иметь представление о том, как сохраняется и обрабатывается информация в компьютерах. Для этого используются электрические элементы, которые могут принимать только два состояния: включено и выключено. При сохранении данных в устройствах хранения, последовательность электрических или магнитных зарядов также интерпретируется как состояние включено или выключено, что и составляет содержимое записанной информации.

Двоичные числа

Компьютер сохраняет команды и данные в оперативной памяти как последовательность заряженных или разряженных ячеек. Образно можно представить состояние каждой ячейки как переключатель с двумя состояниями: включено и выключено или истина и ложь. Такие ячейки идеально подходят для хранения двоичных чисел, которые используют *базовое число* 2, и поэтому отдельные биты могут принимать только два состояния — 0 или 1. Ячейки памяти, соответствующие единице, имеют повышенный заряд, а соответствующие нулю — почти разряжены. На рис. 3.1 условно показано соответствие переключателей и двоичных чисел.

Включено	Выключено	Включено	Включено	Выключено	Выключено	Включено	Выключено
1	0	1	1	0	0	1	0

Рис. 3.1. Соответствие переключателей и двоичных чисел

Первые компьютеры на самом деле имели наборы механических переключателей, которыми управляли вручную. На смену им пришли электромеханические переключатели, и только позже стали использоваться транзисторы. Сначала тысячи, а сейчас и миллионы электронных переключателей размещаются в микропроцессорном чипе.

Биты, байты, слова, двойные и учетверенные слова

Каждый разряд в двоичном числе называется *битом*. Восемь битов составляют *байт* — отдельно адресуемый элемент памяти в большинстве компьютеров. Байт может содержать простую машинную команду, символ или число. Следующим по размеру элементарным понятием является *слово*. В процессорах Intel слово составляет 16 бит (2 байта).

Размер слова не является жестко определенным. Компьютеры, в которых применяются процессоры Intel, используют 16-, 32- или 64-разрядные операнды, поэтому длина слова определяется в 16, 32 или 64 бит, т.е. существуют слова, двойные слова и учетверенные слова, как показано на рис. 3.2.

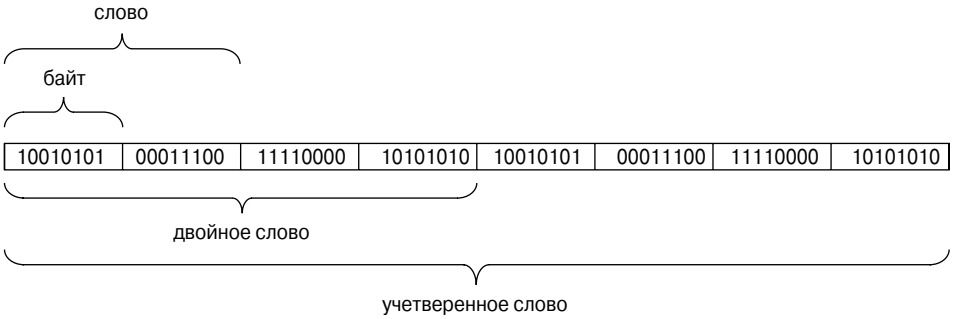

Рис. 3.2. Размеры слов

В табл. 3.1 представлены диапазоны значений в зависимости от количества разрядов, составляющих байты, слова, двойные слова и учетверенные слова. Наибольшее значение в диапазоне определяется как 2^b-1, где b — число битов.

Таблица 3.1. Размеры в битах и диапазон целых чисел

Тип слова	Биты	Диапазон
байт без знака	8	0–255
слово без знака	16	0–65 535
двойное слово без знака	32	0–4 294 967 295
учетверенное слово без знака	64	0–18 446 744 073 709 551 615

Команды и данные

В языках высокого уровня команды и данные имеют существенное различие, однако в машине они все представлены одинаково, как наборы нулей и единиц. Например, следующая последовательность двоичных разрядов может включать первые три символа алфавита, сохраненные в строковой переменной, или может быть машинной командой.

```
01000001010000101000011
```

Именно поэтому программисты, использующие язык ассемблера, должны разделять данные и команды, чтобы процессор "выполнял" переменные и воспринимал команды как переменные.

Числовые системы

Каждая числовая система имеет *основание системы счисления*, или *базовое число* — максимальное значение, которое может быть присвоено отдельной цифре. В табл. 3.2 приведены разрешенные значения для различных систем счисления. Во всех последующих главах при отображении записей в памяти, значений регистров и адресов будут использоваться шестнадцатеричные числа, для которых основанием системы счисления является число 16. Для компактного отображения значений больше 9 используются *шестнадцатеричные символы* от A до F, соответствующие десятичным значениям от 10 до 15.

Когда записывают двоичное, восьмеричное или шестнадцатеричное число, к нему добавляют определенный символ, представленный строчной буквой. Например, шестнадцатеричное число 45 должно быть записано как `45h`, восьмеричное 76 — как `76o`, а двоичное 11010011 необходимо записать как `11010011b`. Таким образом ассемблер распознает числовые константы в исходной программе.

Таблица 3.2. Цифры в различных числовых системах

Система	Базовое число	Разрешенные значения
Двоичная	2	0 1
Восьмеричная	8	0 1 2 3 4 5 6 7
Десятичная	10	0 1 2 3 4 5 6 7 8 9
Шестнадцатеричная	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Правила представления числовых данных

Очень важно знать и понимать правила представления данных в памяти и отображения их на экране. Для примера воспользуемся десятичным числом 65. Сохраненное в памяти как один байт оно будет представлено в двоичном виде как `01000001`. Отладочная программа, вероятнее всего, будет его отображать как `41`, т.е. как шестнадца-

теричное значение. Но если это число послать в память видеоадаптера как символ, то на экране увидим букву A. Это происходит потому, что в соответствии с кодировкой ASCII для символа A выбрано значение 01000001. Таким образом, интерпретация данного значения зависит от определенных условий, которые и придают ему смысл.

- ***двоичное число*** — сохраняется в памяти как последовательность битов, готовых к использованию в расчетах. Целые двоичные числа сохраняются по 8, 16 или 32 разряда.
- ***символы стандартного набора ASCII*** — могут быть представлены в памяти подобно числовому значению, например как 123 или 65. Для отображения символов может быть использован любой числовой формат, как показано в табл. 3.3.

Таблица 3.3. Представление буквы "A" в различных форматах

Формат	Значение
Двоичный символ ASCII	01000001
Восьмеричный символ ASCII	101
Десятичный символ ASCII	65
Шестнадцатеричный символ ASCII	41

Преобразование двоичных чисел в десятичные

Довольно часто приходится переводить двоичные числа в соответствующие десятичные. В табл. 3.4 показано соответствие двоичных и десятичных чисел от 2^0 до 2^{15}. Каждая ячейка представляет степень числа 2.

Чтобы перевести двоичное число в десятичное, просуммируйте десятичные эквиваленты всех позиций двоичного числа, в которых находится единица. Пример преобразования числа 00001001 показан на рис 3.3.

Таблица 3.4. Значения разрядов двоичного числа

2^n	Десятичное значение	2^n	Десятичное значение
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

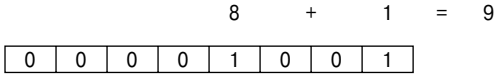

Рис. 3.3. Преобразование двоичного числа в десятичное

Шестнадцатеричные числа

Большие двоичные числа почти невозможно прочитать, поэтому используют шестнадцатеричные числа, которые удобно преобразовывать и которые довольно хорошо воспринимаются при просмотре листингов. Их используют и в языке ассемблера, и в отладчиках для отображения двоичных данных и машинных команд. Каждое шестнадцатеричное число заменяет четыре двоичных бита, а два шестнадцатеричных числа представляют байт.

На рис. 3.4 показано представление двоичного числа `000101100000011110010100` в шестнадцатеричном виде `160794h`.

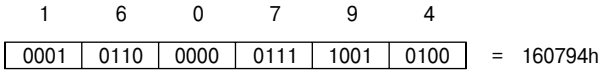

Рис. 3.4. Соответствие двоичного и шестнадцатеричного чисел

Одно шестнадцатеричное число может принимать значения от `0` до `15`, поэтому наравне с числами `0–9` для отображения значений от `10` до `15` используют символы от `A` до `F`: `A=10`, `B=11`, `C=12`, `D=13`, `E=14`, `F=15`. В табл. 3.5 показано, как последовательность четырех битов переводится в десятичное и шестнадцатеричное значение.

Таблица 3.5. Двоичные, десятичные и шестнадцатеричные эквиваленты

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

Каждая позиция шестнадцатеричного числа представляет степень числа `16`, что используется при вычислении десятичного значения числа, как показано в табл. 3.6.

Таблица 3.6. Степени числа 16

16^n	Десятичное	16^n	Десятичное
16^0	1	16^4	65 536
16^1	16	16^5	1 048 576
16^2	256	16^6	16 777 216
16^3	4096	16^7	268 435 456

Для преобразования шестнадцатеричного значения в десятичное необходимо умножить значение каждого разряда на соответствующий десятичный эквивалент, а потом их просуммировать. На рис. 3.5 приведен пример преобразования числа `3BA4h`. Берется наи-

большее значение 3 и умножается на десятичный эквивалент позиции — 4096. Следующее число B умножается на 256, A умножается на 16 и последнее 4 умножается на 1. Все просуммировав, получим соответствующее десятичное число 15268.

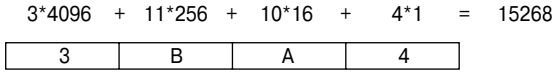

Рис. 3.5. Преобразование шестнадцатеричного числа 3BA4 в десятичное

Числа со знаком

Двоичные числа могут быть как со знаком, так и без знака. Числа без знака используют все восемь битов для получения значения (например, 11111111 = 255). Просуммировав значения всех битов для преобразования в десятичное число, получим максимально возможное значение, которое может хранить байт без знака (255). Для слова без знака это значение будет составлять 65535. Байт со знаком использует только семь битов для получения значения, а старший восьмой бит зарезервирован для знака, при этом 0 соответствует положительному значению, а 1 — отрицательному. На представленном ниже рис. 3.6 показано отображение положительного и отрицательного числа 10.

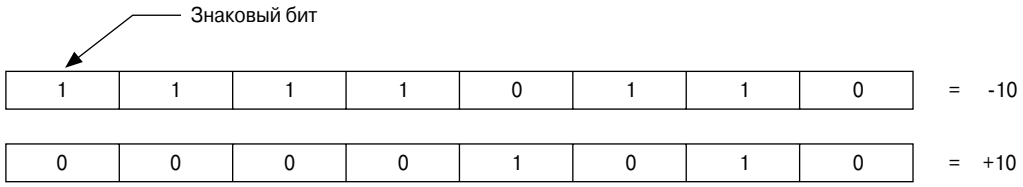

Рис. 3.6. Отображение положительного и отрицательного числа 10

Дополнение до двух

Чтобы не усложнять процессор, отдельный блок для реализации операции вычитания не делают; эту операцию выполняет блок суммирования. Перед суммированием отрицательные числа преобразовываются в дополнительные. Это такое число, которое в сумме с исходным числом дает 0. Например, десятичное –6 будет дополнением к 6, так как 6+(-6)=0. Таким образом, вместо операции вычитания A–B процессор суммирует с положительным числом A дополнительное к B: A+(-B). Вместо того чтобы вычесть 4 из 6, процессор просто складывает –4 и 6.

При работе с двоичными числами для дополнительного числа используется термин *дополнение до двух* (встречается также определение *двоичное дополнение*). Например, для двоичного значения 0001 двоичным дополнением до двух будет 1111. Такое число получается из исходного числа после изменения всех единиц на нули, а нулей на единицы (инверсия) и прибавления к полученному числу единицы, как показано ниже. Инвертирование битов в двоичном числе обозначается NOT(n), а полученное значение называется *дополнением до единицы*, или *первичным дополнением*. Поэтому двоичное дополнение может быть представлено выражением NOT(n)+1.

число	0001
инверсированное число	1110
добавить 1	1111

Если сложить N и дополнение до двух к N, получим 0: 0001+1111=0000. Операция получения дополнения до двух полностью обратима. Например, для отрицательного числа -10 дополнением до двух будет 10.

```
 11110110        = -10
 00001001          инверсия бит
+00000001          добавить 1
 00001010        = +10
```

Еще несколько примеров преобразования чисел приведено в табл. 3.7 (для дополнения до двух используем аббревиатуру NEG(n)).

Таблица 3.7. Преобразование чисел

Десятичное	Двоичное	NEG(n)	Десятичное
+2	00000010	11111110	–2
+16	00010000	11110000	–16
+127	01111111	10000001	–127

Максимальные и минимальные значения

Число со знаком из n разрядов может использовать только n-1 бит для получения значения. Например, знаковый байт использует только семь битов (от 0 до 127). В табл. 3.8 показаны максимальные и минимальные значения для байт, слов, двойных и учетверенных слов со знаком. Наименьшие значения (-128, -32768, -2147483648) являются недопустимыми. Нетрудно убедиться, что двоичное дополнение до -128 (10000000) будет также 10000000.

Таблица 3.8. Целые числа со знаком и без знака

Тип хранения	Биты	Диапазон
байт со знаком	7	от –128 до +127
слово со знаком	15	от –32 768 до +32 767
двойное слово со знаком	31	от –2 147 483 648 до +2 147 483 647
учетверенное слово со знаком	63	от –9 223 372 036 854 775 808 до +9 223 372 036 854 775 807

Хотя процессор выполняет вычисления без учета знака числа, в программе знак операнда необходимо обязательно указывать. Сложение операндов со знаком +16 и -23 будет выглядеть в командах ассемблера следующим образом.

```
MOV AX,+16
ADD AX,-23
```

В двоичном выражении число 16 будет выглядеть как 00010000, а -23 — как 11101001. Когда процессор складывает эти числа, он получает 11111001. Это двоичное число соответствует десятичному -7, как показано в примере.

```
 00010000          16
+11101001         -23
=11111001          -7
```

Таким образом, сложение чисел со знаком получается корректным. Но в данном случае двоичное число можно интерпретировать и как десятичное число без знака 249. Именно поэтому программист должен отслеживать эти величины и четко представлять, какой тип имеет данное значение.

Хранение символов

Компьютеры могут хранить только двоичные значения, но нам необходимо работать не только с численными значениями, но и с символами, такими как "A" или "$". Для этого компьютер использует схему кодирования символов, которая позволяет преобразовывать символы в числа и наоборот. Наиболее известная система кодирования для компьютеров обозначается аббревиатурой ASCII (American Standard Code for Information Interchange). В ASCII каждому символу присваивается уникальный код, включая контрольные символы, используемые при печати и передаче данных между компьютерами. Стандартный ASCII-код использует только 7 разрядов в диапазоне 0-127. Значения от 0 до 31 заняты служебными кодами, используемыми при печати, передаче информации и выводе на экран. В обычном режиме они не отображаются на экране. Остальные значения, допустимые в байте, — дополнительные, их применяют для расширения символьного ряда. В операционной системе MS DOS значения 128-255 используются для получения графических символов и греческих букв. В операционной системе Windows существует множество наборов символов, и в каждом из них дополнительным значениям соответствуют различные символы.

Строка символов представляет в памяти последовательность байт. Например, числовым кодам строки "ABC123" будет соответствовать последовательность значений 41h, 42h, 43h, 31n, 32h и 33h.

Таблица кодов ASCII приведена в справочном разделе. Чтобы найти шестнадцатеричное значение нужного символа, используйте соответствующие значения второй строки и второй колонки, на пересечении которых находится символ. Старший разряд числа находится в строке, младший — в колонке. Например, чтобы найти шестнадцатеричное значение символа "a", посмотрим на соответствующее значение в строке. Это будет 6, соответствующее значение в колонке будет 1. Таким образом, получаем шестнадцатеричное значение 61h.

Хранение чисел

Как наиболее эффективно сохранять числа в памяти? Это зависит от того, как эти числа будут использоваться. Если числа используются для вычислений, должно быть применено двоичное представление числа, и наоборот, лучше хранить коды ASCII, если данные значения будут использоваться для отображения символов на экране. Например, число 123 можно сохранить в памяти двумя способами: как последовательность кодов ASCII для чисел 1, 2 и 3 или как один байт со значением 123, как показано на рис. 3.7.

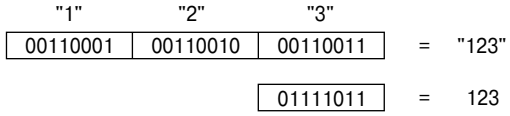

Рис. 3.7. Хранение строки "123" и числа 123 в памяти

Двоичное содержимое памяти всегда можно просмотреть, но только по значению нельзя определить, что оно представляет. Предположим, два байта памяти имеют значения 01000001 и 01000010. Но что это такое? Данные, код или текст? Это невозможно узнать, пока не идентифицирован определенный участок памяти. Программа

должна отслеживать состояние данных и тип представления, чтобы избежать конфликтов. Языки высокого уровня не позволят использовать переменные вместо команд, чего нельзя сказать о языке ассемблера. Ограничения языка помогают избежать серьезных ошибок, но в языке ассемблера почти нет ограничений, и программист должен учитывать все, даже незначительные детали.

Основы языка ассемблера

Уже было сказано о том, что программы можно писать прямо в машинных кодах, используя числовые значения, но на языке ассемблера делать это значительно удобнее. Понятные аббревиатуры команд позволяют легко запомнить их назначение и писать программы, которые затем можно читать и модернизировать. Однако поскольку ассемблер — язык низкого уровня, то для выполнения простейшего математического выражения требуется несколько команд или даже несколько десятков команд. Поэтому обычно почти невозможно анализировать логику программы, используя только мнемонику языка ассемблера.

Команды языка ассемблера

Команды языка ассемблера представляют взаимно однозначное соответствие с машинными инструкциями. В простейшем варианте они состоят из мнемокода команды с последующими операндами. Все это непосредственно преобразуется в машинные команды. Команды могут либо иметь, либо не иметь операндов, как показано на примере ниже.

```
CLC         ; мнемокод
INC AX      ; мнемокод с одним операндом
MOV AX,BX   ; мнемокод с двумя операндами
```

Любую команду можно сопроводить комментарием, отделяя его от команды точкой с запятой “;”.

Ассемблер является языком низкого уровня, потому что его команды, по сути, машинные, т.е. команды языка ассемблера имеют взаимно однозначное соответствие с машинными командами. И наоборот, одно утверждение в языке высокого уровня, таком как Delphi, обычно транслируется в несколько машинных команд.

Операнд может быть регистром, переменной, ячейкой памяти или непосредственным значением, как показано в табл. 3.9.

Таблица 3.9. Представление операндов

Операнд	Описание
`10`	(непосредственное значение)
`count`	(переменная)
`AX`	(регистр)
`[0200]`	(ячейка памяти)

Константы и выражения

Цифровой литерал является комбинацией цифр и дополнительных символов — знаков, десятичных точек и экспонент:

- `5`;
- `5,5`;
- `26,E+05`.

Целочисленные константы могут оканчиваться дополнительным буквенным символом, который является указателем базы системы счисления: h – шестнадцатеричная, q (или o) — восьмеричная, d — десятичная, b — двоичная. Если дополнительного буквенного символа нет, то по умолчанию принимается десятичная система счисления. Буквенный символ может быть строчным или заглавным. В табл. 3.10 приведено несколько примеров целочисленных констант.

Если число не сопровождается дополнительным буквенным символом, то по умолчанию принимается, что для данного числа используется десятичная система счисления.

Таблица 3.10. Представление целочисленных констант

Константа	Система счисления
1Ah	шестнадцатеричная
26	десятичная
1101b	двоичная
36q	восьмеричная
2BH	шестнадцатеричная
42Q	восьмеричная
36D	десятичная
47d	десятичная
0F6h	шестнадцатеричная

Если шестнадцатеричная константа начинается с буквы, то перед ней должна ставиться цифра 0. Хотя дополнительный символ может быть и заглавной буквой, рекомендуется использовать строчные буквы для унификации записи.

Константное выражение состоит из комбинации цифровых литералов, операторов и определенных символьных констант. Значение константного выражения определяется во время трансляции программы и не может меняться во время выполнения программы. Ниже приведено несколько примеров константных выражений, включающих только цифровые литералы:

- 5;
- 26,5;
- 4 * 20;
- -3 * 4/6;
- -2,301E+04.

Символические константы являются именами константных выражений.

```
rows = 5
columns = 10
tablePos = rows * columns
```

Обратите внимание на то, что хотя это утверждение и выглядит подобно выражению времени выполнения в языках высокого уровня, но определяется оно во время трансляции.

Символы или символьные константы. Константами могут быть отдельные символы или строки символов, заключенные в двойные или одинарные кавычки. Внутренние кавычки допускаются, как показано в следующих примерах:

- 'ABC';
- 'X';
- "This is a test";
- '4096';
- "This isn't a test";
- 'Say "hello" to Bill.'.

При этом необходимо хорошо представлять, что символьная константа "4096" занимает в памяти четыре байта, так как каждая цифра определяется в соответствии с кодом ASCII и занимает один байт, о чем уже говорилось ранее.

Утверждения

В языке ассемблера *утверждение* состоит из имени, мнемокода, операндов и комментариев. Утверждения бывают двух типов: команды и директивы. *Команды* — это утверждения, выполняемые в программе, а *директивы* — утверждения для информирования ассемблера о том, как создавать выполняемый код. Общая форма утверждения выглядит так:

```
[имя] [мнемокод] [операнды] [; комментарии]
```

Утверждение имеет свободную форму записи. Это означает, что его можно записывать с любой колонки и с произвольным количеством пробелов между операндами. Утверждение должно быть записано на одной строке и не заходить за 128-ю колонку. Можно продолжить запись со следующей строки, но при этом первая строка должна заканчиваться символом "\" (обратная косая черта), как показано в примере ниже.

```
longArrayDefinition DW l000h, 1020h, 1030h \
1040h, 1050h, 1060h, 1070h, 1080h
```

Повторим, что команда — это утверждение, которое выполняется процессором во время работы программы. Команды могут быть нескольких типов: передачи управления, передачи данных, арифметические, логические и ввода-вывода. Команды транслируются ассемблером прямо в машинные коды. Ниже приведен фрагмент листинга со всеми используемыми категориями команд.

```
CALL MySub  ; Передача управления.
MOV AX,5    ; Передача данных.
ADD AX,20   ; Арифметическая.
JZ next1    ; Логическая  ; (переход, если установлен флаг нуля).;
IN AL,20    ; Ввод-вывод (чтение из аппаратного порта).
```

Директива — это утверждение, которое выполняется ассемблером во время трансляции исходной программы в машинные коды. Например, директива DB заставляет ассемблер выделить память для однобайтовой переменной, названной count, и поместить туда значение 50.

```
count DB 50
```

Следующая директива .STACK заставляет ассемблер зарезервировать пространство памяти для стека.

```
.STACK 4096
```

Имена

Имена определяют метки, переменные, символы или ключевые слова. Они могут состоять из символов, приведенных в табл. 3.11.

Таблица 3.11. Допустимые для имен символы

Символы	Описание
A ... Z, a ... z	Буквы
0 ... 9	Цифры
?	Знак вопроса
_	Подчеркивание
@	Знак амперсанда
$	Знак доллара

Для имен есть следующие ограничения.

- Максимальное количество символов — 247 (в MASM).
- Заглавные и строчные буквы не различаются.
- Первым символом могут быть @, _ или $. Последующими могут быть эти же символы, буквы или цифры. Избегайте использования в начале имени символа "@", так как многие предопределенные имена начинаются именно с него.
- Выбранные программистом имена не должны совпадать со словами, зарезервированными в языке ассемблера.

Переменные — это данные какой-либо программы, которым присвоены имена.

```
count1 DB 50        ; Переменная (место в памяти).
```

Метка является именем, которое размещается в пространстве кодов. Метки отмечают те строки программы, на которые необходимо делать переход из других мест. Метка может стоять в начале пустой строки или за ней могут находиться команды. В приведенном ниже фрагменте листинга метки указывают на определенные строки в программе.

```
Label1:  MOV AX,0
         MOV BX,0
Label2:
         JMP Label1  ; Переход на Label1.
```

Ключевые слова всегда имеют предопределенный смысл в языке ассемблера. Это могут быть команды или директивы, например MOV, PROC, TITLE, ADD, AX или END. Ключевые слова не могут использоваться программистом для каких-либо других целей, например как имена. Если использовать ключевое слово ADD как метку, это будет синтаксической ошибкой.

```
ADD: MOV AX,10  ; Синтаксическая ошибка!
```

Разработка программы на языке ассемблера

Хотя внешне программы, написанные на языке ассемблера, сильно отличаются от программ, созданных на языке высокого уровня, тем не менее технология их разработки одинакова. Однако следует учесть, что разработка программ на языке ассемблера требует большего внимания и аккуратности. При этом необходимо последовательно выполнить следующие этапы.

- Поставить задачу и составить проект программы. На этом этапе нередко составляются блок-схемы — эскиз выполняемых программой действий.

- Ввести команды программы в компьютер с помощью редактора. При сложной логике программы удобно предварительно написать комментарии на обычном языке с описанием предполагаемых действий, а затем вставлять между ними соответствующие команды языка ассемблера. В качестве редактора можно использовать любой текстовый редактор, который создает файлы с расширением `.txt`. Подойдет даже простейший Notepad.
- Оттранслировать программу с помощью ассемблера. Если ассемблер обнаружит ошибки, исправить их в редакторе и оттранслировать программу заново.
- Преобразовать результат работы ассемблера в исполняемый модуль с помощью компоновщика.
- Выполнить программу.
- Проверить результаты. Если они не соответствуют ожидаемым, найти ошибки с помощью отладчика. Данный этап называется отладкой и обычно занимает бо́льшую часть времени, затрачиваемого на разработку программы.

Если программа простая и короткая, то ее разработка не займет много времени. Однако сложные программы требуют значительного времени на каждом этапе, поэтому необходимо тщательно планировать процесс разработки уже на этапе проектирования, иначе этап отладки может никогда не закончиться.

Программа, написанная в кодах ассемблера, называется *исходной программой*, а ее преобразованный вид в команды микропроцессора именуется *объектной программой*. Таким образом, функцией ассемблера является преобразование исходной программы, доступной восприятию человеком, в объектную программу, понятную микропроцессору.

Операционная система может хранить программу в любом подходящем месте памяти и освобождает разработчика от необходимости думать, куда ее поместить. Но чтобы этим воспользоваться, надо преобразовать оттранслированную программу в вид, позволяющий ее перемещение. Такие программы называются *перемещаемыми*. Они создаются с помощью *компоновщика* — программы LINK, которая обязательно входит в комплект поставки ассемблера.

Обычно *объектным модулем* называется файл, содержащий результат трансляции программы ассемблером. А файл, содержащий перемещаемую версию оттранслированной программы, называется *исполняемым модулем*. Таким образом, функцией компоновщика LINK является создание исполняемого модуля из объектного модуля.

Компоновщик необходим также при написании большой программы. Невозможно написать сложную программу как единое целое, поэтому такие программы пишут по частям, которые потом можно собрать вместе с помощью компоновщика. При этом можно использовать модули, написанные другими программистами, или ранее написанные и отлаженные модули. Если есть набор подходящих модулей, то разработка сложной программы может занять не так уж и много времени. Надо только объединить уже существующие и вновь написанные модули и получить один исполняемый модуль — что и сделает компоновщик.

Компоновщик должен вызываться для любой написанной программы, даже если она состоит только из одного объектного модуля. Одномодульные программы компоновщик сразу преобразует в перемещаемый модуль. Если программа состоит из двух или большего количества модулей, то компоновщик сначала объединяет их, а затем преобразовывает результат в перемещаемый модуль.

Завершенную программу можно вызвать для выполнения двумя способами:

- набрать ее имя в качестве команды DOS;
- выполнить ее под управлением программы DEBUG.

Обычно программу следует вызывать из операционной системы только в том случае, если есть уверенность в ее безошибочной работе. Пока она не будет полностью отлажена, необходимо вызывать программу только под управлением отладчика DEBUG.

С помощью отладчика DEBUG можно управлять процессом выполнения программы. Наряду с другими функциями, DEBUG позволяет отображать и изменять значения переменных, останавливать выполнение программы в заданной точке или выполнять программу по шагам. Таким образом, DEBUG является основным инструментом для поиска и исправления ошибок в программе.

На рис. 3.8 показаны этапы разработки программ с помощью ассемблера. В скобках для каждого модуля указаны расширения файлов, в которых модули сохраняются на диске.

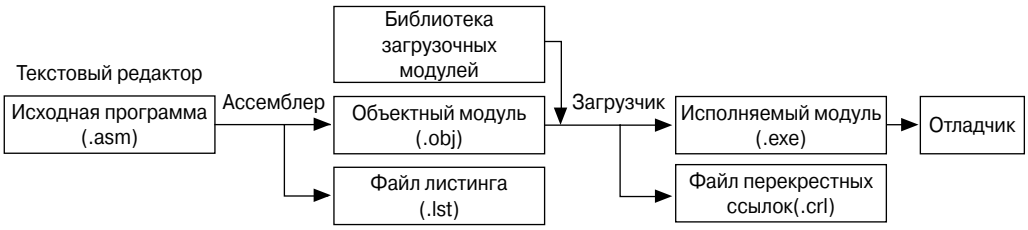

Рис. 3.8. Этапы разработки программ на ассемблере

Разработка программы методом “сверху вниз”

При создании программы естественным желанием является начать последовательно вводить команды, реализуя логику программы по мере их ввода. Такой метод может сработать в случае простой или короткой программы, но обычно это приводит к ошибкам и программам, которые трудно понять, а впоследствии еще труднее модифицировать. Благодаря удобным возможностям редактирования текстов, предоставляемым программами обработки текстов и редакторами, можно воспользоваться более легким, удобным и эффективным методом разработки программ. Это метод разработки “сверху вниз”.

Разработка методом “сверху вниз” означает, что вначале создается набросок программы в виде текста на обычном языке, а затем этот набросок постепенно дополняется деталями. Набросок должен представлять собой ряд строк, в которых описаны действия программы. Например, при разработке программы, которая выполняет одну из нескольких функций по выбору пользователя, набросок может выглядеть следующим образом.

```
;  Изобразить меню возможных функций
;  Запросить у пользователя выбор из меню
;  Прочитать ответ пользователя
;  Проверить допустимость ответа
;  Если ответ допустим, выполнить требуемую функцию
```

Как вы уже знаете, точки с запятой означают, что эти строки представляют собой не команды, а комментарии, которые необходимы только разработчику. Ассемблер пропускает все до конца строки после точки с запятой.

Затем, используя этот текст, производится вставка необходимых команд между строками. Так как каждая строка описывает относительно небольшую задачу, проще всего выполнить каждую задачу в отдельности, проверяя решение перед тем, как двинуться дальше. Иначе говоря, начните со вставки первой группы команд (в нашем примере с команд для изображения меню), затем запишите полученную программу на диск и выполните все последующие этапы (трансляцию, загрузку и выполнение). Вы-

полнение такой частичной программы покажет, правильно ли она работает. Если она работает неправильно, отладьте ее и попробуйте еще раз. После того как первая часть отлажена, перейдите ко второй части, затем к третьей и т.д.

Может показаться, что это очень медленный метод разработки программы, но только так можно разрабатывать программы, которые содержат мало ошибок и которые впоследствии можно модифицировать. При этом достигаются следующие цели:

- четко просматривается логика программы;
- производится документирование программы с помощью комментариев, впоследствии программу будет легко модифицировать;
- обеспечивается правильность работы каждой части до того, как произойдет переход к разработке следующей части программы.

Работа в DOS под Windows NT

Хотя термины “Windows 3x” и “Windows 9x” связывают с операционными системами, эти графические интерфейсы пользователя являются только оболочкой, сооруженной над DOS. Довольно часто эти системы останавливаются, и связано это с тем, что если одна программа для Windows (или программа DOS, работающая под Windows) дает сбой, то зависает и вся система. Приходится перезагружать компьютер, и, вероятнее всего, часть готовой работы будет потеряна. И все же, благодаря внутренним конструктивным доработкам, в Windows 98 отказов стало меньше, пользователи, работающие с этими системами, ограничены архитектурой DOS, которая не позволяет создать оболочку, устойчивую к неисправностям.

Однако все еще работают сотни миллионов копий DOS и десятки тысяч приложений, созданных для этих систем. Довольно часто программа состоит только из версии DOS, что относится как к большим приложениям, так и к незаметным утилитам. Поэтому технология DOS сохранена и в новейших операционных системах, в которых она используется совсем по-другому. Операционные системы Windows NT (Windows 2000 и Windows XP) совсем не используют DOS. Программа DOS является только отдельной исполняемой программой, как и многие другие. При инициализации Windows система DOS не загружается даже на заднем плане. Технология NT имеет операционную систему с ядром, которое отвечает за большую часть того, что умеет DOS, например ввод-вывод с помощью клавиатуры и экрана, работа с загружаемыми драйверами устройств, обработка запросов на ввод-вывод с диска. Windows NT может эмулировать определенные операционные системы с использованием модулей, которые называются подсистемами среды, и DOS — одна из таких операционных систем, которая работает как программа под управлением Windows NT. Подсистема среды DOS предоставляет весь системный сервис, как это обычно делает DOS, но эти функции интегрированы в Windows NT, а не расположены отдельно от нее.

Windows NT имеет специальное окно для DOS, которое называется *командная консоль*, и работа в этом окне очень похожа на сеанс MS DOS в Windows 3x или Windows 9x. Интерпретатор команд содержит богатый набор команд, размеры окна можно изменять, окно имеет полосу прокрутки и не ограничено 24 строками, как в DOS.

Эмуляцию DOS обеспечивает 32-разрядное приложение с именем `cmd.exe`, которое является расширенной версией MS DOS и не только обеспечивает совместимость с MS DOS, но и позволяет запускать приложения Windows, OS/2 и POSIX в режиме командной строки.

Сеанс DOS, который создается при запуске приложения DOS из командного окна, можно сконфигурировать с помощью загружаемых драйверов устройств, TSR и т.д.

При запуске программы DOS из Windows NT создается виртуальная машина DOS, благодаря которой программа DOS работает как бы на отдельном компьютере.

Windows NT создает отдельную виртуальную машину для каждого запускаемого приложения DOS. Каждая такая машина имеет весь сервис, необходимый для работы как с 16-разрядными, так и с 32-разрядными вызовами DOS в соответствии с требованиями DOS 6. Этой виртуальной машине выделяется 16 Мбайт памяти. При необходимости могут поддерживаться популярные диспетчеры памяти. Есть и ограничения. В целях безопасности и защиты ядра приложения DOS должны быть изолированы. Для этого подсистема среды DOS перехватывает все процессы ввода-вывода, проверяет их, а затем направляет данные по назначению. Этот процесс обслуживается перехватчиками ввода-вывода, которые, в свою очередь, передают данные программе NT Executive для доставки по назначению.

Любые традиционно написанные программы DOS, которые выполняют ввод-вывод с помощью стандартных системных вызовов DOS, будут работать под Windows NT без проблем. А программы, выполняющие запись непосредственно на устройство, для которого драйверы не разрешают прямого доступа (например, драйверы жестких дисков), будут прерваны программой контроля безопасности, что приведет к сообщению об ошибке и завершению опасной программы.

Если программа пытается записывать и читать из портов COM и LPT, с экрана или клавиатуры, то Windows NT выполнит это безупречно, однако другие виды прямого управления памятью, диском или устройством разрешены не будут.

Большинство программ DOS, которые используют драйвер мыши или клавиатуры, будут работать, поскольку строки ввода драйвера мыши и клавиатуры эмулируются. Эти устройства эмулируются, так как очень многие программы DOS используют их.

Операционная система DOS использует командную строку для ввода отдельных команд. Перечень всех доступных команд можно получить, если ввести команду `help`. Список всех команд будет приведен в табл. 3.13.

Инструментальные средства

Раньше уже упоминались слова “трансляция”, “загрузка”, “отладка” и другие, связанные с этапами работы ассемблера, но о самом ассемблере пока ничего не говорилось. В этом разделе будут даны начальные сведения о наиболее популярных ассемблерах TASM и MASM.

Ассемблер фирмы Borland (TASM)

Ассемблер фирмы Borland, как и любой другой, поставляется с полным набором программ, необходимых для компиляции исходного файла, получения выполняемого файла, загрузки и редактирования. Последняя версия ассемблера Borland Turbo Assembler 5.0 реализует следующие функциональные особенности:

- объектно-ориентированную технику программирования;
- поддержку 32-разрядной модели и кадра стека;
- полную поддержку процессоров Intel 386, Intel 486 и Pentium;
- использование директив упрощенной сегментации;
- поддержку таблиц;
- гибкую систему макросов.

К преимуществам данного транслятора следует также отнести высокую скорость компиляции. Используя данный ассемблер, можно программировать для Windows. Так как фирма Borland закрыла свое направление для языков C/C++, она отказалась и от TASM как от отдельного продукта. И теперь новые версии TASM приходят только в составе таких продуктов, как Delphi и C++ Builder.

Фирма Borland разработала отличный ассемблер, в котором есть возможности, недоступные в других трансляторах, например объектно-ориентированная техника программирования. Так как TASM удобно использовать при написании программ для операционной системы Windows, которая состоит из сообщений, исключений и классов, то эта возможность TASM оказалась как нельзя кстати.

Ассемблер фирмы Microsoft (MASM)

Последняя версия ассемблера Microsoft Macro Assembler 6.15 реализует следующие возможности:

- поддержку 32-разрядной модели памяти и кадра стека;
- полную поддержку всех процессоров вплоть до Pentium II;
- директивы упрощенной сегментации;
- поддержку таблиц;
- директивы языков верхнего уровня.

В отличие от фирмы Borland, Microsoft поддерживала свой продукт и выпускала его как отдельный пакет, так и в составе таких программных пакетов, как Microsoft Quick C, Microsoft Visual Studio и др.

Использование этого ассемблера при написании программ для Windows будет более привычным для тех, кто начал использовать MASM еще с операционной системой DOS и продолжает его применять с Windows. В этом трансляторе предусмотрено все: от разработки простых оконных приложений для Windows до создания виртуальных драйверов устройств VxD (все подробно описано в документации и файлах помощи). В MASM нельзя применять объектно-ориентированный стиль программирования, но можно использовать дополнительные макросы, такие как `.IF`, `.WHILE`, `.REPEAT`, `.CONTINUE`, `.BREAK`. (В TASM также можно обнаружить эти макросы, но они плохо документированы.)

Еще одно преимущество данного ассемблера — новый пакет MASM32, в котором собрано все, что нужно программисту при написании программ для Win32.

Пример простой программы

В листинге ниже приведена простая программа, которая отображает на экране традиционное приветствие “`Hello, world!`”. В этой программе использованы основные особенности приложений на языке ассемблера. В первой строке использована директива `Title`, остальные символы строки трактуются как комментарий, как и все символы во второй строке. Исходный код этой программы написан на языке ассемблера и должен быть оттранслирован в машинные коды перед запуском программы. Эта программа совместима как с ассемблером Microsoft, так и с ассемблером Borland.

Сегменты являются строительными блоками программы. *Сегмент кодов* определяет место, где хранятся коды программы, *сегмент данных* включает все переменные, а *сегмент стека* включает исполнительный стек. Стек — это специальное пространство в памяти, которое обычно используется программой при вызове и возврате подпрограмм.

Программа “Hello World”

```
TITLE Hello World Program (hello.asm)
; Эта программа отображает слова "Hello, world!"
.MODEL small
.STACK 100h
.DATA
```

```
  message DB "Hello, world!",0dh,0ah,'$'
.CODE
main PROC
  MOV AX,@data
  MOV DS,AX
  MOV AH,9
  MOV DX,offset message
  INT 21h
  MOV AX,4C00h
  INT 21h
main ENDP
END main
```

Кратко рассмотрим основные строки программы.

- Директива `.MODEL small` сообщает ассемблеру, что в данной программе необходимо использовать не более 64 Кбайт памяти для кодов и не более 64 Кбайт для данных. Директива `.STACK` устанавливает размер пространства для стека емкостью `100h` (256) байт. Директива `.DATA` отмечает начало сегмента данных, где сохраняются переменные. Здесь под именем `message` сохраняется строка "Hello, world!", за которой следуют два служебных символа перехода на новую строку (`0dh,0ah`). Символ "`$`" используется как символ конца строки для подпрограмм, которые будут считывать эту строку.
- Директива `.CODE` отмечает начало сегмента кодов, где должны находиться выполняемые команды. Директива `PROC` объявляет начало процедуры. В этой программе объявлена процедура с именем `main`.
- Первые две команды процедуры `main` копируют адрес сегмента данных (`@data`) в регистр `DS`. Команда `MOV` всегда сопровождается двумя операндами: первый указывает, куда поместить данные, а второй — откуда эти данные взять.
- Затем в процедуре `main` на экран выводится строка символов. При этом вызывается функция DOS, которая непосредственно выводит на экран строку символов, начиная с адреса, который указан в регистре `DX`. Номер этой функции предварительно должен быть помещен в регистр `AH`.
- Последние две команды процедуры `main` (`MOV AX,4C00h` и `INT 21h`) заканчивают программу и передают управление операционной системе.
- Утверждение `main ENDP` использует директиву `ENDP`, которая отмечает конец процедуры, причем процедуры не могут "перекрывать" друг друга.
- В самом конце находится директива `END`, заканчивающая программу, которая должна быть оттранслирована. Следующая за ней метка `main` определяет точку входа программы, т.е. то место, в котором процессор начинает выполнять коды программы.

В табл. 3.12 приведен список наиболее часто используемых директив ассемблера.

Таблица 3.12. Стандартные директивы ассемблера

Директива	Описание
`END`	Окончание трансляции программы
`ENDP`	Конец процедуры
`PAGE`	Устанавливает формат листинга
`PROC`	Начало процедуры
`TITLE`	Название листинга

Окончание табл. 3.12

Директива	Описание
.CODE	Отмечает начало сегмента кодов
.DATA	Отмечает начало сегмента данных
.MODEL	Устанавливает режим памяти
.STACK	Устанавливает размер стека

После того как программа "Hello World" написана в текстовом редакторе, ее необходимо сохранить на диске под именем `hello.asm`. После этого ее можно транслировать.

Работа с ассемблером Microsoft

Как уже не раз упоминалось, операционная система DOS сохранена для работы с множеством все еще существующих программ, написанных под DOS. Ассемблер Microsoft является одной из таких программ. Для этой программы нет соответствующего оконного интерфейса под Windows, и приходится работать в менее удобной среде DOS. Поэтому для начала работы с ассемблером необходимо запустить интерпретатор команд DOS, для чего выбирается команда Start⇨All Programs⇨Accessories⇨Command Prompt в системном меню. Затем необходимо установить пути для удобного запуска программ ассемблера. Для этого существует команда `PATH` из набора команд DOS. Рассмотрим подробнее работу в окне DOS.

Работа с DOS

Для работы с DOS необходимо запустить интерпретатор команд DOS. Для операционной системы Windows XP это делается так, как описано в предыдущем разделе. Получим окно, показанное на рис. 3.9.

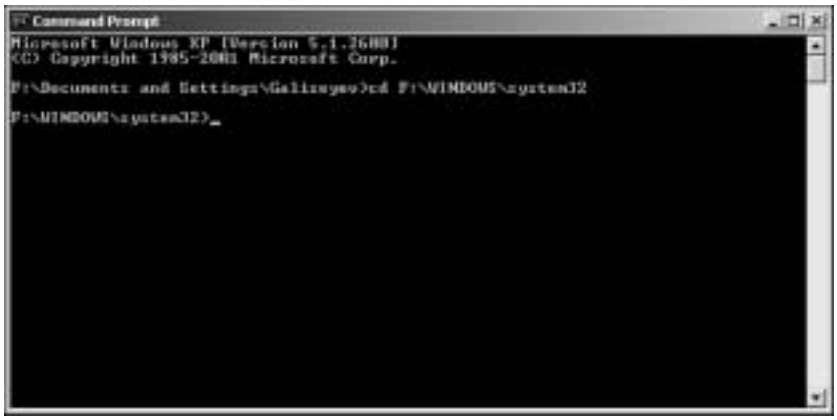

Рис. 3.9. Окно интерпретатора команд DOS

Теперь необходимо перейти в каталог, где находится программа интерпретатора команд DOS — `cmd.exe`. Это каталог `..\WINDOWS\system32`. Здесь не указан том, на котором находится этот каталог, так как у каждого каталога он может быть разный. Для перехода к данному каталогу необходимо выполнить команду `cd`.

После этого можно ввести команду `help` — появится перечень всех доступных команд, который приведен в табл. 3.13.

Таблица 3.13. Перечень команд DOS для Windows XP

Команда	Описание
ASSOC	Отображает или модифицирует связанные с программами расширения
AT	Таблица команд и программ для запуска на компьютере
ATTRIB	Отображает или изменяет атрибуты файла
BREAK	Устанавливает или сбрасывает проверку комбинации клавиш
CACLS	Отображает или модифицирует список контроля доступа файлов (ACL)
CALL	Вызов одной командной программы из другой
CD	Отображает или изменяет текущий каталог
CHCP	Отображает или устанавливает номер активной кодовой страницы
CHDIR	Отображает или изменяет текущий каталог
CHKDSK	Проверяет диск и отображает отчет
CHKNTFS	Отображает или модифицирует проверку диска во время загрузки
CLS	Очистка экрана
CMD	Запускает новый экземпляр интерпретатора команд Windows
COLOR	Устанавливает цвета символов и фона
COMP	Сравнивает содержимое двух файлов
COMPACT	Отображает или изменяет сжатие файлов на разделах NTFS
CONVERT	Конвертирует тома FAT в NTFS. Нельзя конвертировать текущий том
COPY	Копирует один или несколько файлов
DATE	Отображает или устанавливает дату
DEL	Стирает один или несколько файлов
DIR	Отображает список файлов и подкаталогов
DISKCOMP	Сравнивает содержимое двух флоппи-дисков
DISKCOPY	Копирует содержимое флоппи-диска на другой диск
DOSKEY	Редактирование командной строки, повторение команд и создание макросов
ECHO	Отображение сообщений или выключение и включение отображения вводимых символов на экране (эхо)
ENDLOCAL	Окончание изменений локализации Windows в командных файлах
ERASE	Удаление одного или нескольких файлов
EXIT	Закрытие интерпретатора команд
FC	Сравнивает содержимое нескольких файлов и отображает различие между ними
FIND	Поиск текстовой строки в одном или нескольких файлах
FINDSTR	Поиск строк в файлах
FOR	Запуск указанной команды для каждого файла из набора нескольких файлов
FORMAT	Форматирование диска для работы с Windows
FTYPE	Отображение или модификация типов файлов, используемых в перечне соответствий типов и расширений
GOTO	Делает переход к помеченной строке в командных файлах
GRAFTABL	Разрешает системе Windows отображать расширенный набор символов в графическом режиме
HELP	Используется для получения справочной информации

Окончание табл. 3.13

Команда	Описание
IF	Используется для создания условного утверждения в командных файлах
LABEL	Создает, изменяет или удаляет метки томов на диске
MD	Создает каталог
MKDIR	Создает каталог
MODE	Конфигурирует системные устройства
MORE	Отображает часть выходных данных на экран за один раз (для получения продолжения необходимо еще раз выполнить эту команду)
MOVE	Перемещает один или несколько файлов из одного каталога в другой
PATH	Отображает или устанавливает путь поиска файлов для операционной системы
PAUSE	Приостанавливает обработку команд в командных файлах и отображает сообщение
POPD	Восстанавливает предыдущее значение текущего каталога, сохраненного командой PUSHD
PRINT	Распечатывает текстовый файл
PROMPT	Изменяет вид запроса
PUSHD	Сохраняет текущий каталог перед его изменением
RD	Удаляет каталог
RECOVER	Восстанавливает доступную информацию с запорченных дисков
REM	Отмечает комментарии в командных файлах или файле CONFIG.SYS
REN	Переименовывает один или несколько файлов
RENAME	Переименовывает один или несколько файлов
REPLACE	Заменяет файлы
RMDIR	Удаляет каталог
SET	Отображает, устанавливает или удаляет переменные окружения Windows
SETLOCAL	Начинает локализацию окружения Windows в командных файлах
SHIFT	Сдвигает позицию заменяемых параметров в командных файлах
SORT	Сортировка
START	Запуск указанной команды или программы в отдельном окне
SUBST	Создание виртуального диска
TIME	Отображает или устанавливает системное время
TITLE	Устанавливает имя окна для интерпретатора команд
TREE	Графически отображает структуру каталогов
TYPE	Отображает содержимое текстового файла
VER	Отображает версию Windows
VERIFY	Устанавливает режим проверки файлов на корректность записи на диск
VOL	Отображает метку тома и серийный номер
XCOPY	Копирует файлы и каталоги

Как видите, команд операционной системы DOS не так уж и много и запомнить их не составляет особого труда. Но даже этого делать не надо, те несколько команд, которые необходимы для запуска и работы с ассемблером, будут рассмотрены подробно ниже, а аналоги всех команд есть в системе Windows, так как режим DOS просто

эмулируется системой Windows и в конечном итоге производится обращение к процедурам Windows. В Windows 98 и в более ранних версиях можно работать и непосредственно в DOS, так как эти системы являются только надстройками над DOS, но начиная с версии Windows NT все обстоит по-другому. Здесь системы DOS в чистом виде уже нет и вся работа производится только через процедуры Windows.

Чтобы получить более подробное описание каждой команды, необходимо набрать в командной строке имя команды и дополнить ее символами "/?". Например, если набрать `cd /?`, то получим следующую справку.

```
Отображает или изменяет имя текущего каталога.
CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]
Где две точки (..) указывают, что необходимо перейти к каталогу, включающему
данный каталог.
Если набрать CD без параметров, то отобразится обозначение текущего тома и ка-
талог.
Если набрать CD с обозначением тома, то отобразится текущий каталог.
Параметр  /D используется для переключения текущего тома на указанный том с не-
обходимым каталогом.
Команда CHDIR используется в тех случаях, когда имена файлов или каталогов со-
держат пробелы.
```

А вот командные файлы, которые существовали еще в первых системах DOS, могут принести пользу даже при работе в системе Windows. Рассмотрим подробнее командные файлы.

Командный файл

Как уже отмечалось ранее, в среде DOS работать довольно неудобно, так как приходится вручную набирать довольно длинные команды и пути файлов, часто приводящие к ошибкам. Командные файлы созданы для облегчения работы в среде DOS, но и в среде Windows они тоже приносят довольно ощутимую пользу.

Командный файл — это обычный текстовый файл с последовательностью команд, которые необходимо выполнить. Командный файл можно написать в обычном текстовом редакторе и сохранить с расширением `.bat`. Файлы с таким расширением операционная система трактует как командные и вызывает для их выполнения интерпретатор команд, а не текстовый редактор.

В командный файл можно включать все команды DOS, а также команды условного перехода `for`, `goto` и `if`, которые позволяют реализовывать различную последовательность выполнения команд в зависимости от наличия определенных условий. Еще несколько команд позволяют контролировать ввод-вывод и вызывать другие командные файлы.

Контролировать процесс выполнения можно по возвращаемым приложениями кодам ошибок, которые могут быть равны `0` (ошибок нет) или `1` или бо́льшие значения при наличии ошибок.

Командный файл может иметь параметры, которые дописываются в командный файл при его запуске на выполнение. Для подстановки параметров в команды используются переменные от `%0` до `%9`. Если используется переменная `%0`, то вместо ее при запуске подставляется имя командного файла, а переменные от `%1` до `%9` заменяются соответствующими аргументами. Для доступа к аргументу за пределами `%9` используется команда `shift`. Переменная `%*` ссылается на все аргументы, за исключением `%0`.

Например, для копирования содержимого каталога `Folder1` в каталог `Folder2` можно создать командный файл `Mybatch.bat`, содержимое которого будет представлять одну строку

```
xcopy %1\*.* %2
```

и при вызове этого файла как

```
Mybatch.bat C:\folder1 D:\folder2
```

вместо переменной %1 будет подставлен каталог Folder1, а вместо переменной %2 — каталог Folder2.

Тот же самый результат можно получить, если в среде DOS выполнить команду

```
xcopy C:\folder1\*.* D:\folder2
```

С параметрами командного файла можно применять модификаторы. Модификаторы используют информацию о текущем диске и каталоге для расширения параметров. При использования модификаторов сначала поставьте символ процента (%), затем тильду (~), а за ней требуемый модификатор.

Все возможные модификаторы перечислены в табл. 13.14.

Таблица 3.14. Модификаторы для командных файлов

Модификатор	Описание
%~1	Удаляет все фрагменты, ограниченные кавычками ("")
%~f1	Дополняет аргумент полным путем
%~d1	Дополняет аргумент буквенным символом текущего диска
%~p1	Дополняет аргумент путем
%~n1	Дополняет аргумент именем файла
%~x1	Дополняет аргумент расширением файла
%~s1	Использует только короткие имена
%~a1	Дополняет аргумент атрибутами файла
%~t1	Дополняет аргумент датой и временем создания файла
%~z1	Дополняет аргумент размером файла
%~$PATH:1	Ищет каталоги, перечисленные в переменной окружения PATH и дополняет аргумент полным путем первого найденного каталога

Подготовка к запуску ассемблера

Теперь, когда вы познакомились с командами DOS и командными файлами, сделаем так, чтобы было удобно запускать файлы на компиляцию, загрузку и отладку.

Для начала нужно обязательно указать пути, по которым операционная система будет находить каталог с установленными модулями ассемблера. Например, если ассемблер находится в каталоге E:\MASM615\, то необходимо выполнить следующую команду:

```
path E:\MASM615;%path%
```

После выполнения этой команды операционная система уже будет знать, в каких каталогах искать исполняемый файл, если не указан полный путь, а введено только имя файла. Здесь используется модификатор %path% для сохранения всех ранее введенных каталогов.

А для того, чтобы сделать активным каталог с вашими программами, необходимо выполнить команду cd. Например:

```
cd /d H:\Gennady\Work\Assembl\Book1\Programs
```

Чтобы не набирать текст каждый раз, создайте командный файл и включите в него эти строки, после чего нужно будет только запустить командный файл, и все выполнится автоматически.

Теперь можно оттранслировать разработанную ранее программу `Hello`, которая должна находиться в активном каталоге. Для этого необходимо вызвать транслятор. Командная строка для вызова транслятора имеет следующий синтаксис:

```
masm /options source(.asm),[out(.obj)],[list(.lst)], [cref(.crf)][;]
```

Здесь `options` — это опции, или дополнительные элементы настройки; `out` — имя выходного файла; `list` — имя файла листинга; `cref` — имя файла перекрестных ссылок. Так как имена заключены в квадратные скобки, то их указывать не обязательно. Если будут установлены соответствующие опции, то эти файлы все равно будут созданы, но с именем исходного файла. Все допустимые опции перечислены в табл. 3.15.

Таблица 3.15. Опции транслятора MASM

Опции	Описание действия	Преобразование для ML
`/A`	Располагает сегменты в алфавитном порядке	Не поддерживается. Эмулируется с помощью директивы `.ALPHA`
`/B`	Устанавливает размеры внутреннего буфера	Игнорируется
`/C`	Создает таблицу перекрестных ссылок в листинге	Преобразуется в `/FR`
`/D`	Создает листинг первого прохода	Преобразуется в `/Fl /Sf`
`/Dsymbol`	Определяет символ	Передается без модификации
`/E`	Эмулирует команды вычислений с плавающей запятой	Преобразуется в `/FPi`
`/H`	Создает список аргументов командной строки	Вызывает `QH` для `MASM.EXE`
`/I`	Устанавливает пути включаемых файлов	Передается без модификации
`/L`	Создает стандартный листинг	Преобразуется в `/Fl`
`/LA`	Полный листинг	Преобразуется в `/Fl` и `/Sa`
`/ML`	Различает строчные и заглавные символы в именах	Преобразуется в `/Cp`
`/MU`	Преобразует все символы имен в строчные	Преобразуется в `/Cu`
`/MX`	Сохраняет написание нелокальных имен (идентификаторы, объявленные с директивами `PUBLIC`, `EXTERN` или `COMM`)	Преобразуется в `/Cx`
`/N`	Подавляет выдачу таблиц в листинге	Преобразуется в `/Sn`

Учитывая все сказанное выше, дадим команду на трансляцию рассмотренного выше файла `Hello` в следующем виде:

```
masm /la Hello
```

При успешной трансляции на экране появится следующее сообщение:

```
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.
  Invoking: ML.EXE /I. /Zm /c /Fl /Sa hello.asm
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000.  All rights reserved.
 Assembling: hello.asm
```

Полностью весь процесс настройки путей в DOS и обращение к транслятору показаны на рис. 3.10.

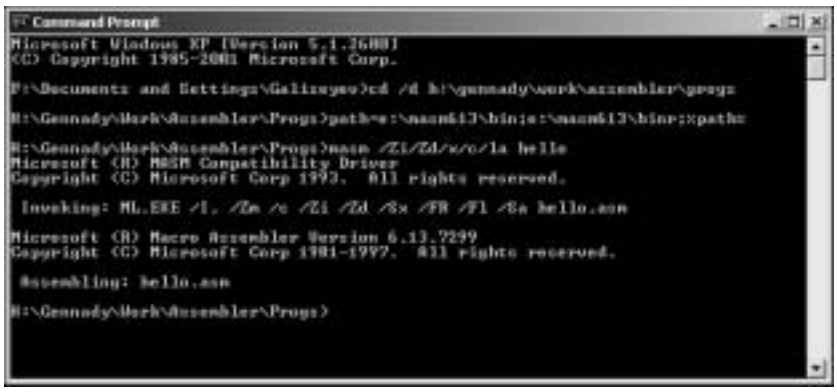

Рис. 3.10. Работа с ассемблером MASM в окне DOS

Обратите внимание, что программа `MASM.EXE` вызывает программу `ML.EXE`, т.е. компиляцию выполняет программа `ML`, которая является универсальной программой, производящей как компиляцию, так и загрузку разрабатываемой программы. С опциями программы `ML` познакомимся далее.

После трансляции в указанном каталоге появятся файлы с расширением `.obj` и `.lst`. Полностью файл листинга приведен ниже.

```
Microsoft (R) Macro Assembler Version 6.15.8803  05/31/04 12:20:51
Hello World Program (hello.asm)                     Page 1 - 1
      TITLE Hello World Program (hello.asm)
      ; Эта программа отображает слова "Hello, world!"
                     .MODEL small
                     .STACK 100h
 0000                .DATA
 0000 48 65 6C 6C 6F 2C    message DB "Hello, world!",0dh,0ah,'$'
       20 77 6F 72 6C 64
       21 0D 0A 24
 0000                .CODE
 0000                main PROC
 0000  B8 ---- R       MOV AX,@data
 0003  8E D8           MOV DS,AX
 0005  B4 09           MOV AH,9
 0007  BA 0000 R       MOV DX,offset message
 000A  CD 21           INT 21h
 000C  B8 4C00         MOV AX,4C00h
 000F  CD 21           INT 21h
 0011                main ENDP
                     END main
Microsoft (R) Macro Assembler Version 6.15.8803  05/31/04 12:20:51
Hello World Program (hello.asm)                      Symbols 2 - 1
Segments and Groups:
                N a m e                 Size   Length  Align  Combine Class
DGROUP . . . . . . . . . . . . .  GROUP
_DATA  . . . . . . . . . . . . .  16 Bit  0010   Word   Public  'DATA'
STACK  . . . . . . . . . . . . .  16 Bit  0100   Para   Stack   'STACK'
_TEXT  . . . . . . . . . . . . .  16 Bit  0011   Word   Public  'CODE'
Procedures,  parameters and locals:
          N a m e           Type    Value  Attr
main . . . . . . . . . .    P Near  0000  _TEXT  Length= 0011 Private
Symbols:
                N a m e                 Type     Value    Attr
@CodeSize  . . . . . . . . . . .  Number   0000h
@DataSize  . . . . . . . . . . .  Number   0000h
@Interface . . . . . . . . . . .  Number   0000h
@Model . . . . . . . . . . . . .  Number   0002h
@code  . . . . . . . . . . . . .  Text     _TEXT
```

```
@data  . . . . . .  Text     DGROUP
@fardata?  . . . . . .  Text     FAR_BSS
@fardata . . . . . .  Text     FAR_DATA
@stack . . . . . .  Text     DGROUP
message  . . . . . .  Byte     0000    _DATA
     0 Warnings
     0 Errors
```

В этом листинге приведена вся информация о созданной программе, начиная с исходных команд и их машинных эквивалентов и заканчивая распределением имен, памяти и сегментов. В дальнейшем мы будем неоднократно обращаться к файлам листингов.

Синтаксические ошибки

Очень немного найдется программистов, которые сразу смогут написать даже небольшую программу без ошибок. Поэтому ассемблер проверяет написанные команды и выводит на экран все строки, в которых есть ошибки, причем с их объяснением. Например, если в программе `Hello` первую команду `MOV` ошибочно набрать как `MIV`, то появится следующее сообщение:

```
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.
 Invoking: ML.EXE /I. /Zm /c /Fl /Sa hello.asm
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.
 Assembling: hello.asm
hello.asm(9) : error A2008: syntax error : ax
```

Иными словами, компилятор зафиксирует ошибку перед операндом `ax` в строке номер 9.

Загрузка программы

После компиляции будет получен объектный файл с расширением `.obj`, который необходимо преобразовать в исполняемый модуль. На данном этапе используется программа-компоновщик, которая применяет объектный файл (в нашем случае `hello.obj`) в качестве входного и создает выполняемый файл, называя его `hello.exe`. Командная строка для компоновщика имеет следующий формат:

```
link [objs],[exefile],[mapfile],[libs],[deffile]
```

Здесь `link` — имя программы компоновщика `link.exe`; `objs` — имя объектного файла; `exefile` — имя выполняемого файла; `mapfile` — имя файла распределения памяти; `libs` — библиотечные файлы; `deffile` — файл определений. Можно также ввести команду `link`, после чего будет создан диалоговый режим, в котором необходимо ввести ответы на все запросы. Если не требуется что-то вводить, просто нажмите клавишу <Enter>, как показано на рис. 3.11.

Рис. 3.11. Диалоговый режим для команды `link`

После работы компоновщика будут созданы выполняемый файл с расширением .exe и файл распределения памяти с расширением .map, который в нашем случае будет совсем небольшим. Листинг файла распределения памяти приведен ниже.

```
Start   Stop    Length  Name            Class
00000H  00010H  00011H  _TEXT           CODE
00012H  00021H  00010H  _DATA           DATA
00030H  0012FH  00100H  STACK           STACK
Origin    Group
0001:0    DGROUP
Program entry point at 0000:0000
```

Запуск программы

Для запуска программы наберите в командной строке имя выполняемой программы hello.

Отладчик

Основной инструмент, с которым приходится работать при создании программ на ассемблере, — отладчик. В дальнейшем будут рассматриваться небольшие примеры программ на языке ассемблера, и лучшим способом для их изучения является использование отладчика. Отладчик — это программа, позволяющая отображать на экране значения необходимых переменных, получать состояние всех регистров и ячеек памяти при пошаговом выполнении программы, вносить изменения в программу, указывать точки останова и многое другое. Это необходимо при проверке написанных на языке ассемблера программ.

Существует несколько великолепных отладчиков, среди которых можно выделить легкую в использовании программу cv.exe для DOS — CodeView, поставляемую вместе с ассемблером компании Microsoft, которая позволяет видеть исходные коды программы, блоки памяти и состояние регистров процессора. То же самое делает отладчик Borland Turbo Debugger. Подробное описание отладчика Microsoft приведено в справочном разделе.

Простая программа

Для знакомства с отладчиком напишем на языке ассемблера небольшую программу sum, приведенную ниже, которая складывает три числа и сохраняет сумму в памяти. Оттранслируем и выполним загрузку программы, затем начнем работу с отладчиком.

```
TITLE Программа суммирования (sum.asm)
.MODEL small        ; Объявляем модель памяти.
.STACK 100h         ; Объявляем сегмент стека размером 100h.
.DATA               ; Объявляем сегмент данных.
sum DW ?            ; Объявляем переменную sum.
.CODE               ; Объявляем сегмент кодов.
main PROC           ; Точка входа в программу.
    MOV  AX,5       ; Поместить в регистр AX значение 5.
    ADD  AX,10      ; Сложить содержимое регистра AX с 10.
    ADD  AX,15      ; Сложить содержимое регистра AX с 15.
    MOV  sum, AX    ; Сохранить содержимое регистра AX в sum.
    MOV  AX,4C00h   ; Номер функции DOS.
    INT  21h        ; Выход из программы. Передача управления в DOS.
main ENDP           ; Конец процедуры (для компилятора).
END main            ; Конец программы (для компилятора).
```

Каждая строка в процедуре main этой программы начинается с кода команды (MOV, ADD или INT) с соответствующими операндами. Если после этого поставить точку с запятой, то до конца строки все будет восприниматься как комментарий, не влияющий на ход выполнения программы. Комментарии можно расставлять и в отладчике. Пошаговая диаграмма выполнения показана на рис. 3.12.

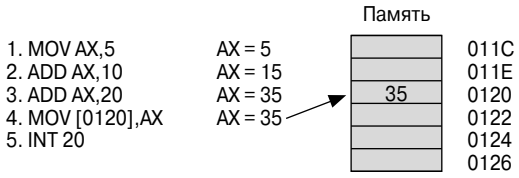

Рис. 3.12. Диаграмма выполнения простой программы

Команда `MOV` заставляет процессор переместить или скопировать значение исходного операнда в принимающий оператор, поэтому в первой строке число `5` перемещается в регистр `AX`. Во второй строке происходит суммирование числа `10` и содержимого регистра `AX`, что делает значение регистра равным `15`. В третьей строке также складываются содержимое регистра `AX` с числом `20`. В регистре `AX` уже будет находиться число `35`. Далее это число копируется в ячейку памяти по адресу, определенному для перемененной `sum`. Команды в пятой и шестой строках необходимы для выхода из программы и передачи управления в DOS.

Программу необходимо написать в текстовом редакторе и сохранить в файле с именем `sum.asm`, после чего ее нужно откомпилировать, а затем для получения исполняемого файла вызвать компоновщик. Но так как эту программу в дальнейшем будем использовать с отладчиком, то необходимо запускать компоновщик с опцией `/co`, т.е. команда вызова компоновщика будет такой:

```
link /co sum
```

После получения выполняемого файла можно запустить отладчик и поработать с этой программой, для чего необходимо ввести команду: `cv sum`. В окне DOS появится графический интерфейс отладчика, как показано на рис. 3.13.

Рис. 3.13. Графический интерфейс отладчика

Работа с отладчиком не представляет особых трудностей. На панели меню внизу окна перечислены оперативные клавиши управления с соответствующими пояснениями. Нажатие клавиши <Alt> открывает доступ к основному набору команд, расположенному на верхней панели меню.

Функции оперативных клавиш с нижней панели меню перечислены в табл. 3.16.

Таблица 3.16. Функции оперативных клавиш

Клавиша		Alt
<F1>	Помощь	
<F2>	Точка останова	Останов по условию
<F3>	Окно модуля	Закрыть окно
<F4>		Возврат команды
<F5>	Размер окна	Окно пользователя
<F6>	Следующий оператор	Возврат исправлений
<F7>	Трассировка	Команда
<F8>	Выполнение по шагам	
<F9>	Запуск	Переход
<F10>	Главное меню	Оперативное меню

С помощью отладчика можно видеть состояние всех регистров, флагов, памяти и текущих команд. Для этого необходимо выполнить команду View⇨CPU из основного меню отладчика (рис. 3.14).

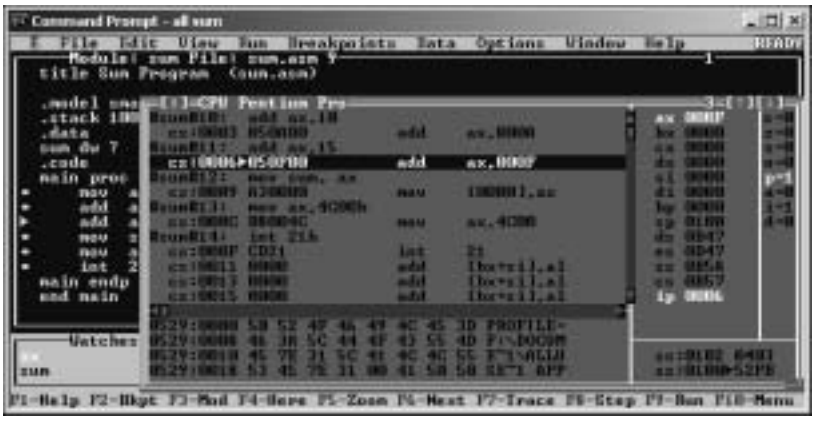

Рис. 3.14. Окно CPU отладчика

В окне CPU можно видеть как команды ассемблера, так и коды машинных команд. Все это не только значительно помогает в отладке программы, но и удобно при изучении языка ассемблера и в процессе исследования операционной системы и аппаратной части компьютера.

В дальнейшем по мере необходимости будут объясняться команды отладчика и описываться приемы работы с ним. Более полное описание команд и директив отладчика, а также работа с ним приведены в справочном разделе.

Резюме

В этой главе даны общие сведения о языке ассемблера, используемых форматах данных и принципах разработки программ на языке ассемблера. Описана работа с ассемблером и основные этапы разработки программ с использованием ассемблера. От-

мечена важность этапа отладки программ и приведены базовые сведения о работе с отладчиком. Для лучшего понимания материала вниманию читателя представлены простейшие примеры разработки и отладки программ.

Контрольные вопросы

1. Существует ли взаимно однозначное соответствие между командами языка ассемблера и машинными кодами?
2. Можно ли написать программу в машинных кодах?
3. Сколько бит находится в байте, слове, двойном и учетверенном слове?
4. Подсчитайте диапазон значений для слова без знака?
5. Что такое команды и что такое данные?
6. Что такое основание системы счисления?
7. Почему удобно использовать шестнадцатеричную систему счисления для отображения данных?
8. Как представлены в памяти числа без знака и со знаком?
9. Что такое дополнение до двух?
10. Как сохраняется в памяти строка символов? Как подсчитать размер занимаемой памяти для отдельной строки?
11. Как сохраняется в памяти числовое значение? Как подсчитать размер занимаемой памяти для числового значения?
12. Что такое константа? Чем константа отличается от переменной?
13. Какие типы утверждений используются в языке ассемблера?
14. Как разрабатывается программа на языке ассемблера?
15. Назовите основные этапы получения выполняемой программы.
16. Вспомните основные опции транслятора.
17. Для чего нужен отладчик?
18. Каковы особенности работы DOS в операционной системе Windows NT?
19. Можно ли использовать все возможности языка ассемблера при работе в DOS под управлением Windows NT?
20. Чем отличаются ассемблеры Microsoft и Borland?