Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Е. А. Потапова

ИНФОРМАТИКА

Ассемблер для процессора Intel 8086

Учебное пособие

Томск
«Эль Контент»
2013

УДК 004.431.4Ассемблер(075.8)
ББК 32.973.2-018.1я73
П 640

Рецензенты:
Хабибулина Н. Ю., канд. техн. наук, доцент кафедры компьютерных систем в управлении и проектировании ТУСУРа;
Фофанов О. Б., канд. техн. наук, доцент, зав. кафедрой оптимизации систем управления Федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Национальный исследовательский Томский политехнический университет».

Потапова Е. А.

П 640 Информатика. Ассемблер для процессора Intel 8086 : учебное пособие / Е. А. Потапова. — Томск : Эль Контент, 2013. — 166 с.

ISBN 978-5-4332-0110-1

В пособии рассматриваются основы информатики и вычислительной техники, архитектура и алгоритм работы процессора, системы счисления, способы хранения и обработки информации. Изложены основы машинного языка, языка программирования ассемблера. Приведены алгоритмы и примеры программ на языке ассемблера.

Пособие предназначено для обучения студентов по направлениям «Информатика и вычислительная техника» и «Управление в технических системах», а также может применяться для студентов других специальностей и направлений, связанных с вычислительной техникой.

УДК 004.431.4Ассемблер(075.8)
ББК 32.973.2-018.1я73

ISBN 978-5-4332-0110-0

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Целью данного курса информатики является создание основы (базиса) для изучения и использования вычислительных систем в других курсах.

Вычислительной системой (ВС) *называется система, состоящая из аппаратных и программных средств, предназначенная для выполнения некоторого множества задач по переработке информации. Классификация ВС по составу аппаратных средств приведена на рис. 1.*

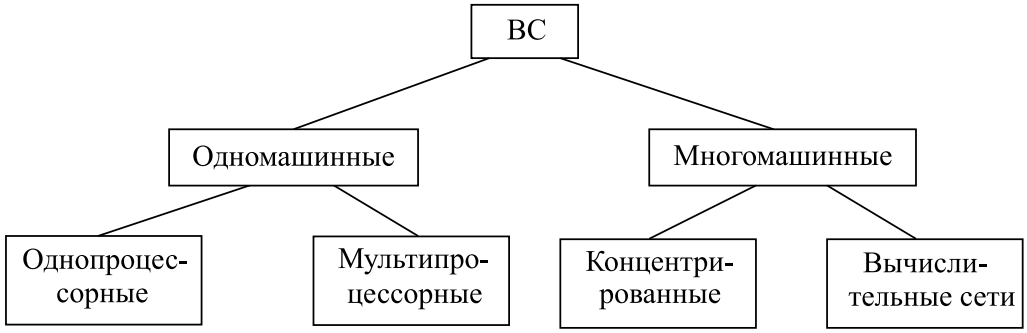


Рис. 1 – Классификация ВС по составу аппаратных средств

Одномашинная однопроцессорная ВС включает одну ЭВМ с одним центральным процессором (ЦП). А *одномашинная мультипроцессорная ВС* — одну ЭВМ с несколькими ЦП. *Многомашинная концентрированная ВС* состоит из нескольких, связанных между собой ЭВМ, расположенных в непосредственной близости одна от другой. *Вычислительной сетью* называется совокупность нескольких территориально разобщенных ЭВМ, связанных каналами передачи данных.

Каждая задача, решаемая ВС, имеет алгоритм решения. *Алгоритм* — правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов. Форма представления алгоритма решения задачи, ориентированная на машинную реализацию, называется *прикладной программой*.

При разработке и выполнении прикладной программы человек-пользователь взаимодействует с аппаратурой ВС не непосредственно, а через *системное программное обеспечение* (рис. 2). В результате пользователь взаимодействует не с реальной, а с *виртуальной* (кажущейся) *ЭВМ*. Например, при выполнении программы на языке ПАСКАЛЬ пользователю кажется, что ЭВМ выполняет операторы этого языка, хотя реальная ЭВМ не может выполнять ничего, кроме программ в машинных кодах.

В данном курсе решается задача обучения основам программирования на языке ассемблера для микропроцессора Intel 80x86 (сокращенно — i8086). Среди всех языков программирования язык ассемблера наиболее близок к языку машинных команд. Поэтому знакомство с ним способствует изучению организации аппаратуры ЭВМ и изучению принципов ее работы. Кроме того, в процессе данного курса решается задача получения навыков построения алгоритмов программ, отвечающих требованиям структурного программирования.

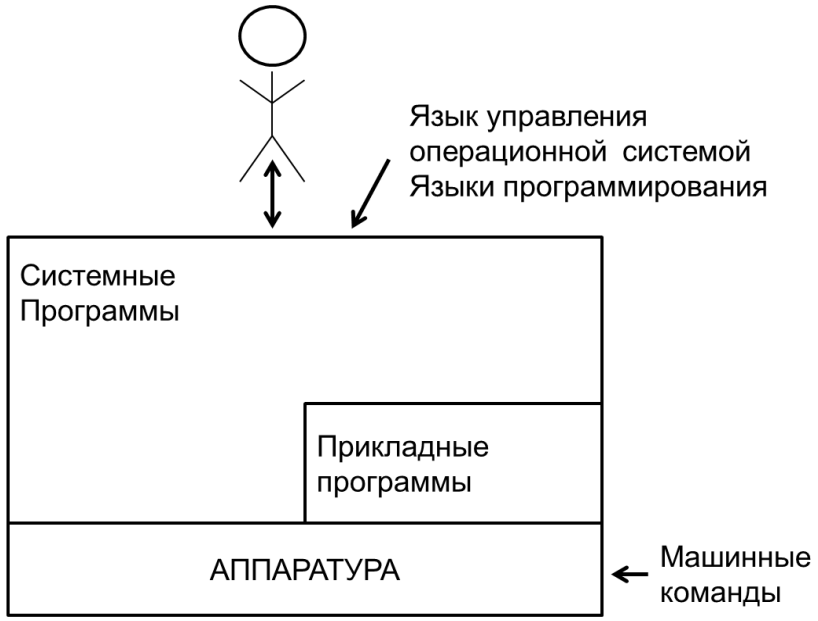
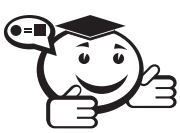


Рис. 2 – Взаимодействие пользователя с виртуальной ЭВМ

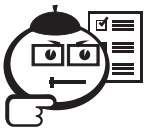
Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.



Эта пиктограмма означает определение или новое понятие.

Эта пиктограмма означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



Эта пиктограмма означает задание. Здесь автор может дать указания для выполнения самостоятельной работы или упражнений, сослаться на дополнительные материалы.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Выводы

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по главе

РАЗДЕЛ I

Выполнение машинных программ

Глава 1

ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ

1.1 Двоичные числа

Чтобы сделать ВС более надежными и простыми, их аппаратура строится из простейших электронных схем, которые могут находиться только в двух состояниях. Одно из них обозначается **0**, а другое — **1**. Такая схема предназначена для длительного или краткого хранения самой мелкой единицы информации — ***бита*** (от «**BI**nary digi**T**» — двоичная цифра).

Пример

Любое число можно представить в виде цепочки битов. Такое представление числа называется *двоичным числом*. Цепочка из восьми битов называется *байтом* (рис. 1.1).

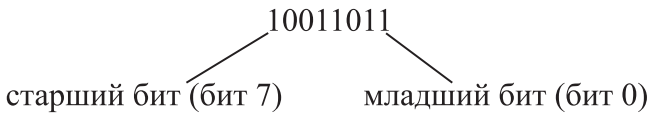


Рис. 1.1 – Пример байта

Величина двоичного числа определяется относительной позицией каждого бита и его значением. Позиционный вес младшего бита $2^0 = 1(10)$. $1(10)$ — единица в десятичной системе счисления. Следующий бит имеет вес $2^1 = 2(10)$. Вес любой позиции получается удвоением веса предыдущей позиции (рис. 1.2).

Для преобразования десятичного числа в двоичное можно использовать один из двух методов — метод деления и метод вычитания. Первый из этих методов

широко используется в программах, выполняющих преобразование чисел из одной системы счисления в другую. Этот метод будет рассмотрен нами в других разделах, при описании соответствующих программ. Сейчас мы будем использовать *метод вычитания*, главное достоинство которого — наглядность. Согласно этому методу для преобразования десятичного числа в двоичное надо сделать ряд вычитаний. Каждое вычитание даст значение одного бита [1].

1. Вычтите из десятичного числа наибольший возможный двоичный вес и запишите 1 в эту позицию бита.
2. Из результата вычтите новый наибольший возможный двоичный вес и запишите 1 в эту новую позицию бита.
3. Так — до получения нулевого результата.

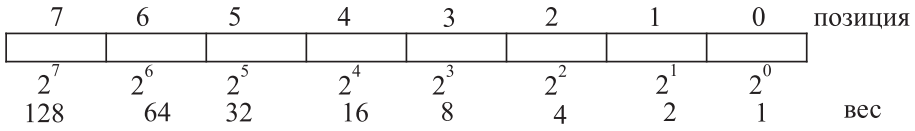


Рис. 1.2 – Веса позиций байта

Пример

Например, преобразование числа 69 в двоичное:

$$\begin{array}{r} 69 \\ -\ 64\ (\text{бит } 6 = 1) \\ \hline 05 \\ -\ 4\ (\text{бит } 2 = 1) \\ \hline 1 \\ -\ 1\ (\text{бит } 0 = 1) \\ \hline 0 \end{array}$$

Записывая 0 в остальные позиции битов (биты 1, 3, 4, 5), получаем окончательный результат: 1000101.

Для выполнения обратного преобразования следует сложить десятичные веса тех позиций, в которых стоит 1:

$$64\ (\text{бит } 6) + 4\ (\text{бит } 2) + 1\ (\text{бит } 0) = 69.$$

Байт может представлять десятичные положительные числа от 0 (00000000) до 255 (11111111). Число 255 может быть получено двумя способами:

1) суммированием весов всех битов байта;
2) по формуле $2^8 - 1$, где 8 — номер первого бита, не вошедшего в состав байта.

Память ЭВМ состоит из блоков памяти по 1024 байта. Число 1024 есть 2^{10}. Число 1024 имеет стандартное обозначение **К**. Следовательно, ЭВМ, имеющая 48К памяти, содержит 48 × 1024 = 49152 байта.

Машинным словом называют битовую строку длиной 16 битов. Одно слово содержит 2 байта (рис. 1.3).

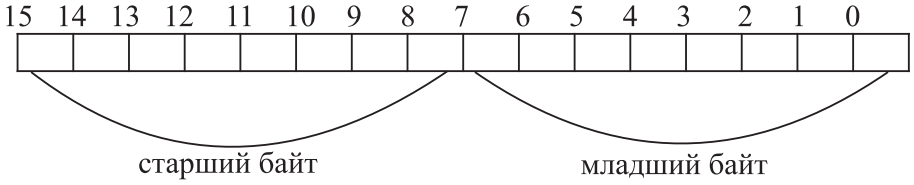


Рис. 1.3 – Структура машинного слова

Каждый бит слова имеет свой вес (рис. 1.4). Просуммировав все веса, найдем максимальное целое число без знака, которое можно записать в одно слово, оно равно $2^{16} - 1 = 65535$.

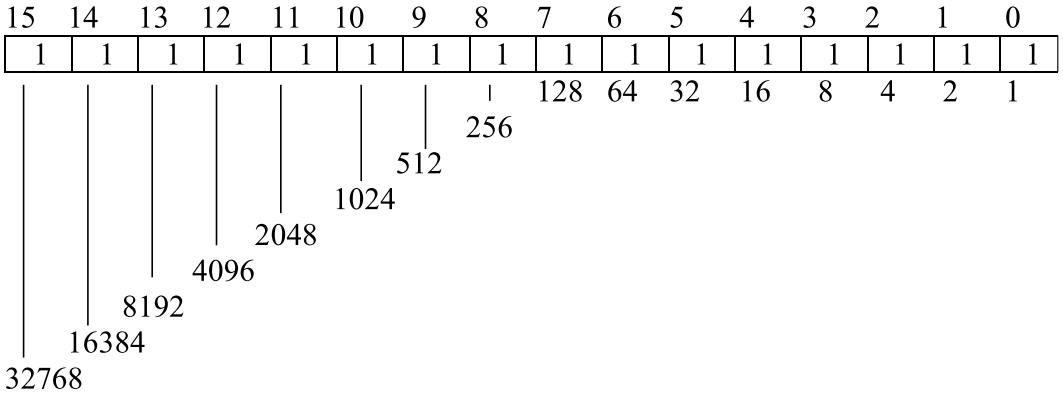


Рис. 1.4 – Веса позиций слова

Двоичное содержимое байта или слова может рассматриваться (интерпретироваться) как число без знака и как число со знаком.

Число без знака занимает все 16 битов слова или 8 битов байта. Оно может быть только положительным.

Пример

Просуммируем два таких числа:

$$\begin{array}{r} 00111100 \\ +\ 00110101 \\ \hline 01110001 \end{array} \qquad \begin{array}{r} 60 \\ +\ 53 \\ \hline 113 \end{array}$$

Если слово (байт) содержит *число со знаком*, то в старшем бите содержится знак (0 есть +, 1 есть –), а оставшиеся 15 или 7 битов содержат само число.

Отрицательное число хранится в *дополнительном коде*. Для получения дополнительного кода числа из его абсолютного значения используется следующее правило:

1) все биты числа (в том числе и знаковый) инвертируются;
2) к полученному числу прибавляется 1.

Пример

Например, получим дополнительный код числа −65.

1) $65(10) = 01000001(2)$
2) $$\begin{array}{r} 10111110 \\ +\quad\quad 1 \\ \hline 10111111 = -65(10) \end{array}$$

Для получения абсолютного значения отрицательного числа повторяют эти же самые два действия. Например:

1) $-65(10) = 10111111$
2) $$\begin{array}{r} 01000000 \\ +\quad\quad 1 \\ \hline 01000001 = 65(10) \end{array}$$

Сумма +65 и −65 должна составить ноль:

$$\begin{array}{r} 01000001\,(+65) \\ +\ 10111111\,(-65) \\ \hline 00000000 \end{array}$$

В данном примере у нас произошли два интересных переноса:

1) в знаковый (7-й) разряд;
2) за пределы байта.

Первая единица переноса обрабатывается как обычно, а вторая теряется. Оба переноса считаются правильными.

Вычитание двух положительных чисел заменяется суммированием первого числа с дополнением второго. Таким образом, как суммирование, так и вычитание выполняет одно и то же аппаратное устройство — сумматор. Кроме этого достоинства, использование дополнения имеет еще одно — число ноль имеет единственное представление, т. е. нет двух нулей — положительного и отрицательного.

Приведем целые числа в окрестностях 0:

+3	00000011
+2	00000010
+1	00000001
0	00000000

−1 11111111
−2 11111110
−3 11111101

Выводы

Отсюда видно, что нулевые биты в отрицательном двоичном числе фактически определяют его величину: рассмотрите весовые значения нулевых битов, как если бы это были единичные биты, сложите эти значения и прибавьте 1.

1.2 Шестнадцатеричные числа

В то время как процессор и другие устройства ЭВМ используют только двоичное представление информации, такое представление очень неудобно для человека, который анализирует содержимое памяти ЭВМ. Введение шестнадцатеричных чисел значительно облегчает эту задачу.

Допустим, что мы хотим проанализировать содержимое четырех последовательных байтов (двух слов). Разделим мысленно каждый байт пополам и запишем для каждого полубайта соответствующее десятичное значение:

0101	1001	0011	0101	1011	1001	1100	1110
5	9	3	5	11	9	12	14

Чтобы не использовать для некоторых полубайтов две десятичные цифры, рассмотрим систему счисления: 10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F. Теперь содержимое тех же самых четырех байтов выглядит более удобно:

59 35 B9 CE

Такая система счисления включает «цифры» от 0 до F, и так как таких цифр 16, то она называется *шестнадцатеричной*. В табл. 1.1 приведено соответствие между двоичными, десятичными и шестнадцатеричными числами от 0 до 15 (10).

Подобно двоичным и десятичным цифрам каждая шестнадцатеричная цифра имеет вес, кратный основанию счисления. Таким образом, каждая цифра имеет вес в 16 раз больше, чем соседняя справа цифра. Крайняя правая цифра имеет вес $16^0 = 1$, следующая $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$.

Например, шестнадцатеричное число 3AF имеет десятичное значение:

$$(3 \cdot 16^2) + (A \cdot 16^1) + (F \cdot 16^0) = (3 \cdot 256) + (10 \cdot 16) + (15 \cdot 1) = 943.$$

Для обозначения шестнадцатеричного числа часто используют букву **H** (или **h**), например: 3AFh. Над шестнадцатеричными числами можно выполнять арифметические операции подобно тому, как они выполняются над десятичными числами.

Таблица 1.1 – Соответствие между двоичными, десятичными и шестнадцатеричными числами

Двоичное	Десятичное	Шестнад.	Двоичное	Десятичное	Шестнад.
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*

Пример

Например, найдем сумму 6Ah и B5h:

$$\begin{array}{r} 6A \\ +\ B5 \\ \hline 11F \end{array}$$

Разность *B*5 – 6*A*:

$$\begin{array}{r} B5 \\ -\ 6A \\ \hline 4B \end{array}$$

Выполнение записанных действий над шестнадцатеричными числами аналогично соответствующим действиям «в столбик» над десятичными числами. Единственное отличие — роль числа 10 при переносе и заеме теперь играет число 16.

1.3 Символьная информация

Для того чтобы хранить в памяти ЭВМ символьную (т. е. буквенно-цифровую) информацию и для того чтобы обрабатывать эту информацию, ее необходимо преобразовать в последовательность битов. Для такого преобразования используются *символьные коды*, среди которых наиболее распространен код ASCII (*American Standard Code for Information Interchange* — Американский стандартный код для обмена информацией).

При использовании данного кода каждый символ представляется в виде одного байта. Например, букве A соответствует двоичный код 01000001 = 41h. Таким образом, одна и та же битовая строка обозначает и букву A, и число 65 (10). Что именно она обозначает, сама битовая строка «не знает». Это определяется тем, как использует (интерпретирует) ее программа.

Контрольные вопросы по главе 1

1) Назовите основные единицы измерения информации.
2) Как перевести число из десятичной системы счисления в двоичную методом вычитания, как проверить правильность перевода?
3) Что такое дополнительный код числа?
4) Как перевести число из десятичной системы счисления в шестнадцатеричную, как проверить правильность перевода?
5) Что такое код ASCII?

Глава 2

ВЫПОЛНЕНИЕ ПРОГРАММ ПРОЦЕССОРОМ I8086

2.1 Структура аппаратных средств

Любая ЭВМ предназначена для выполнения некоторого множества задач по переработке информации. Сущность подобной задачи состоит в том, что имеется некоторая исходная информация, на основе которой требуется получить другую — результирующую информацию. Как следует из предыдущей главы, любая информация содержится в ЭВМ в виде битовой строки.

Каждая задача, решаемая ЭВМ, имеет алгоритм решения.

Алгоритм *— правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов. Форма представления алгоритма решения задачи, ориентированная на машинную реализацию, называется* ***машинной программой****. Совокупность аппаратных средств вычислительной системы (ВС), предназначенных для выполнения машинных программ, часто называют просто* ***аппаратурой****. В процессе любого программирования программист обязательно имеет в своей голове модель той аппаратуры, которая будут использоваться для выполнения программы. Программирование на ассемблере предъявляет особые требования к данной модели, так как в программе приходится учитывать многие особенности используемой аппаратуры.*

Среди однопроцессорных ЭВМ наиболее распространены ***ЭВМ*** *с общей шиной* (рис. 2.1). В данной ЭВМ центральным связывающим звеном между основными

блоками является *общая шина (ОШ)* — группа проводов. ОШ в общем случае есть объединение трех шин:

1) шина управления;
2) шина адреса;
3) шина данных.

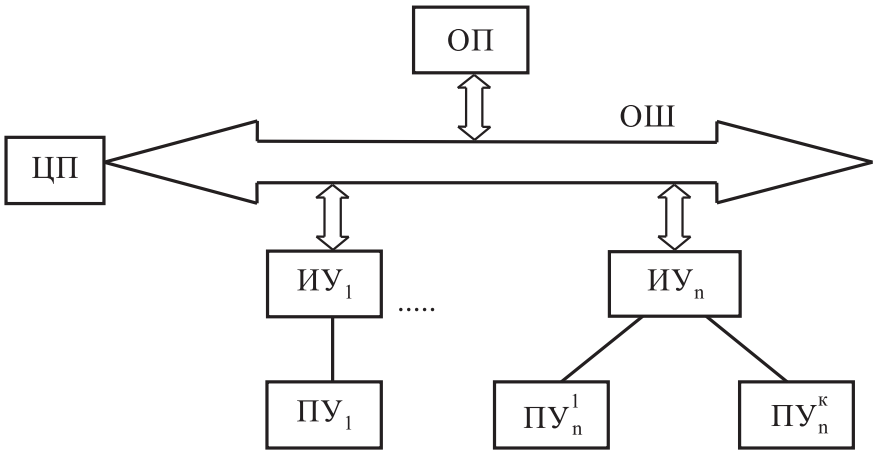


Рис. 2.1 – Структура ЭВМ с общей шиной

Центральный процессор (ЦП) *— «мозг» ЭВМ. Он обеспечивает выполнение прикладных и системных программ.* ***Программа*** *представляет собой последовательность машинных команд (инструкций), каждая из которых требует для своего размещения один, два или большее число байтов* [2]. *На рис. 2.2 приведена структура наиболее типичной машинной инструкции. Здесь* ***КОП*** *— код операции. Это комбинация битов, кодирующая тип операции, которую следует выполнить над операндами (например, суммирование). Операнд 1, операнд 2 — это или сами данные, над которыми выполняется машинная команда, или адреса в памяти (ОП или регистры), где эти данные находятся.*

КОП	Операнд 1	Операнд 2

Рис. 2.2 – Структура машинной инструкции

Оперативная память (**ОП**) предназначена для кратковременного хранения программ и обрабатываемых ими данных. Название обусловлено тем, что операции чтения содержимого ячеек памяти и записи в них нового содержимого производятся достаточно быстро. Иногда используют другое название — ***операционная память***. Это название обусловлено тем, что ЦП может достаточно просто считывать машинные инструкции из ОП и исполнять их. Структура любой ОП

представляет собой линейную последовательность ячеек, которые пронумерованы (рис. 2.3). В зависимости от ЭВМ ячейкой является байт или машинное слово. Номер ячейки называется ***физическим***, или ***реальным***, ***адресом*** этой ячейки. В простейших ЭВМ поле операнда в машинной инструкции содержит этот номер.

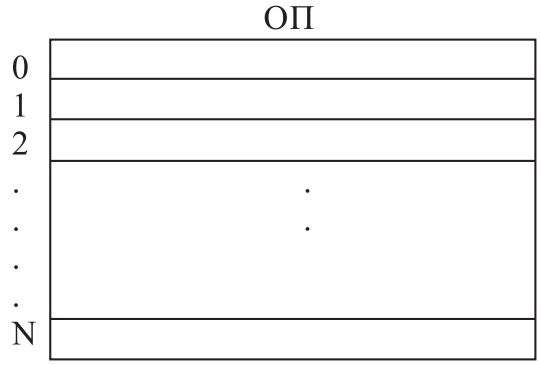


Рис. 2.3 – Структура ОП

Периферийные устройства (ПУ) *— устройства ввода-вывода и устройства внешней памяти. Посредством* ***устройств ввода-вывода*** *ЭВМ «разговаривает» с человеком-пользователем. Сюда относятся: клавиатура, экран (дисплей), телетайп, печатающее устройство и т. д.*

Устройство внешней памяти *предназначено для работы с* ***носителем внешней памяти****. Примером такого устройства является дисковод. Он работает с носителем внешней памяти — магнитным диском.*

Внешняя память (ВП) имеет следующие отличия от ОП:

1) объем ВП во много раз превосходит объем ОП;
2) обмен информацией между ЦП и ВП выполняется во много раз медленнее, чем между ЦП и ОП;
3) ЦП не может выполнять инструкции, записанные в ВП. Для выполнения этих инструкций их необходимо предварительно переписать в ОП;
4) информация на носителе ВП сохраняется и после выключения питания.

Интерфейсное устройство (ИУ) *предназначено для того, чтобы согласовать стандартную для данной ЭВМ структуру ОШ с конкретным типом ПУ, которых существует очень много.*

2.2 Архитектура процессора i8086

На рис. 2.4 приведена архитектура (структура) микропроцессора *i8086*. Рассмотрим назначение его основных модулей.

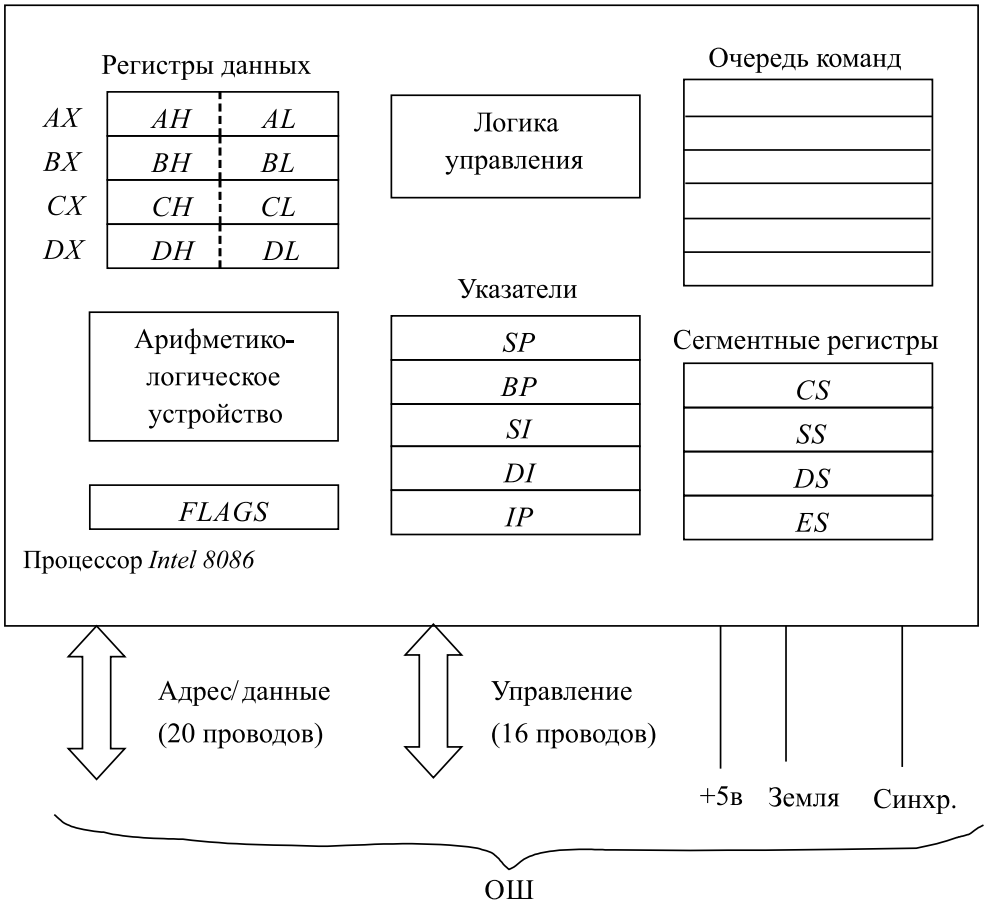


Рис. 2.4 – Структура микропроцессора *i8086*

Блок «*логика управления*» выполняет дешифрирование и исполнение машинных команд. Если очередная команда является арифметической или логической, то она направляется логикой управления для выполнения в *арифметико-логическое устройство*.

Регистры данных предназначены для хранения операндов и результатов операций. *Регистр* — ячейка памяти, скорость обмена с которой намного выше, чем с другими видами памяти: с ОП и, тем более, с ВП. К регистрам данных относятся регистры ***AX***, ***BX***, ***CX*** и ***DX***. Они допускают адресацию не только целых регистров, но и их младшей и старшей половин. Например, допускается использовать два байта в регистре *AX* вместе, а также указывать отдельные байты — ***AL*** (младший) и ***AH*** (старший). Регистры *BX*, *CX* и *DX*, кроме обычных функций, имеют и специальные назначения.

Очередь команд содержит текущую команду на время ее дешифрации (распознавания) и выполнения. Кроме того, для повышения быстродействия в данную очередь заранее считываются из ОП следующие по порядку в программе команды. Длина очереди команд — 6 байт.

В группу регистров «*указатели*» входят указатель команды *IP* и указатель стека *SP*, а также регистры *BP*, *SI* и *DI*. *Указатель команды IP* — очень важный регистр, который содержит относительный адрес (смещение) следующей команды, подлежащей выполнению на ЦП. *Указатель стека SP* также очень важен: он содержит относительный адрес вершины стека. Подробнее регистры *IP* и *SP* будут рассмотрены позже.

Остальные регистры-указатели используются для адресации ячеек какого-то массива относительно его начала. При этом *базовый регистр BP* часто используется для адресации ячеек программного стека относительно его вершины. А *индексные регистры SI* и *DI* обычно используются при циклической обработке элементов массива. При этом для того чтобы обрабатывать одной и той же машинной командой различные ячейки массива, достаточно перед выполнением тела цикла осуществлять изменение содержимого того индексного регистра, который используется для адресации ячеек массива.

Регистр флагов FLAGS отражает текущее состояние ЦП. Структура этого регистра приведена на рис. 2.5.

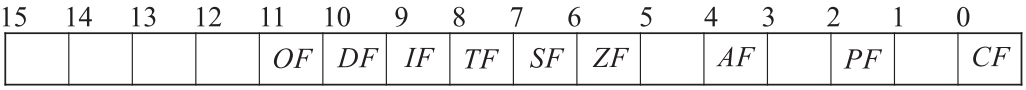


Рис. 2.5 – Регистр флагов микропроцессора *i8086*

Семь битов в *FLAGS* не используются. Остальные биты (флажки) делятся на условные и управляющие. *Условные флажки* отражают результат предыдущей арифметической или логической операции. Это:

1) *SF — флажок знака*. Равен старшему биту результата. Так как в дополнительном коде старший бит отрицательных чисел содержит 1, а у положительных он равен 0, то *SF* показывает знак предыдущего результата;
2) *ZF — флаг нуля*. Устанавливается в 1 при получении нулевого результата и сбрасывается в 0, если результат не равен 0;
3) *PF — флажок паритета*. Устанавливается в 1, если младшие 8 битов результата содержат четное число единиц, в противном случае он сбрасывается в 0;
4) *CF — флажок переноса*. При сложении (вычитании) устанавливается в 1, если возникает перенос (заем) в старший бит (из старшего бита). Обычно данный флаг используется не по прямому назначению, а как признак возврата из подпрограммы: если подпрограмма (процедура или обработчик прерываний) завершилась успешно, то она возвращает *CF* = 0, а если с ошибкой, то *CF* = 1;
5) *AF — флажок вспомогательного переноса*. Устанавливается в 1, если при сложении (вычитании) возникает перенос (заем) из бита 3. Флаг предназначен только для двоично-десятичной арифметики;
6) *OF — флажок переполнения*. Устанавливается в 1, если знаковый бит изменился в той ситуации, когда этого не должно было произойти.

Пример

Пусть, например, машинная команда *add* (здесь и везде далее используются ассемблерные мнемоники машинных команд) выполнила следующее сложение:

$$\begin{array}{r} 0010\,0011\,0100\,0101 \\ +\ 0011\,0010\,0001\,1001 \\ \hline 0101\,0101\,0101\,1110, \end{array}$$

тогда после ее выполнения получаются состояния флажков:

$$SF = 0,\ ZF = 0,\ PF = 0,\ CF = 0,\ AF = 0,\ OF = 0.$$

Если *add* выполнила сложение:

$$\begin{array}{r} 0101\,0100\,0011\,1001 \\ +\ 0100\,0101\,0110\,1010 \\ \hline 1001\,1001\,1010\,0011, \end{array}$$

то флажки принимают состояния:

$$SF = 1,\ ZF = 0,\ PF = 1,\ CF = 0,\ AF = 1,\ OF = 1.$$

Флажки управления влияют на выполнение специальных функций. Эти флажки устанавливаются лишь несколькими специальными машинными командами. Это флажки:

1) *DF — флажок направления*. Он используется при выполнении команд, обрабатывающих цепочки — последовательности ячеек памяти. Если флаг сброшен, цепочка обрабатывается с первого элемента, имеющего наименьший адрес. Иначе цепочка обрабатывается от наибольшего адреса к наименьшему;
2) *IF — флажок разрешения прерываний*. Когда установлен этот флажок, ЦП выполняет маскируемые прерывания. Иначе эти прерывания игнорируются;
3) *TF — флажок трассировки*. Если этот флажок установлен, то после выполнения каждой машинной команды ЦП генерирует внутреннее аппаратное прерывание (прерывание номер 1).

Рассмотренный регистр *FLAGS* чрезвычайно важен не только для понимания логики работы ЦП, но и всей ЭВМ в целом. Это обусловлено тем, что данный регистр фактически является дескриптором ЦП. *Дескриптор* (или *блок управления*) — структура данных, используемая для управления модулем ЭВМ. При этом каждый сколько-нибудь сложный модуль ВС имеет свой блок управления.

Последний блок ЦП образуют сегментные регистры *CS*, *SS*, *DS* и *ES*. Данные регистры используются для адресации ячеек ОП.

2.3 Адресация памяти

Рассмотрим применение сегментных регистров. 20-проводная шина адреса позволяет адресовать до 1 млн ячеек (байтов) ОП, т. к. $2^{20} \approx 1$ млн. Но все регистры в ЦП 16-битные. Ни одного 20-битного нет. Рассмотрим, как получаются 20-битные адреса [3].

Мысленно разобъем ОП на участки по 16 байт, называемые *параграфами* (рис. 2.6). Регистр сегмента кода **CS** содержит номер параграфа, с которого начинается выделенный нашей программе сегмент. Например, это может быть число 0002*h*. Для получения реального адреса начальной ячейки параграфа необходимо его номер (0002*h*) умножить на число 16: $0002h \times 10h = 00020h$.

Указатель команды **IP** содержит относительный адрес (смещение) адресуемой ячейки относительно начала сегмента. Допустим, что это число 100h. Физический адрес R адресуемой ячейки ЦП получает непосредственно перед обращением к ОП путем суммирования содержимого регистра *CS*, умноженного на 16 $(10h)$, с содержимым регистра IP:

$$R = (CS) \times 10h + (IP).$$

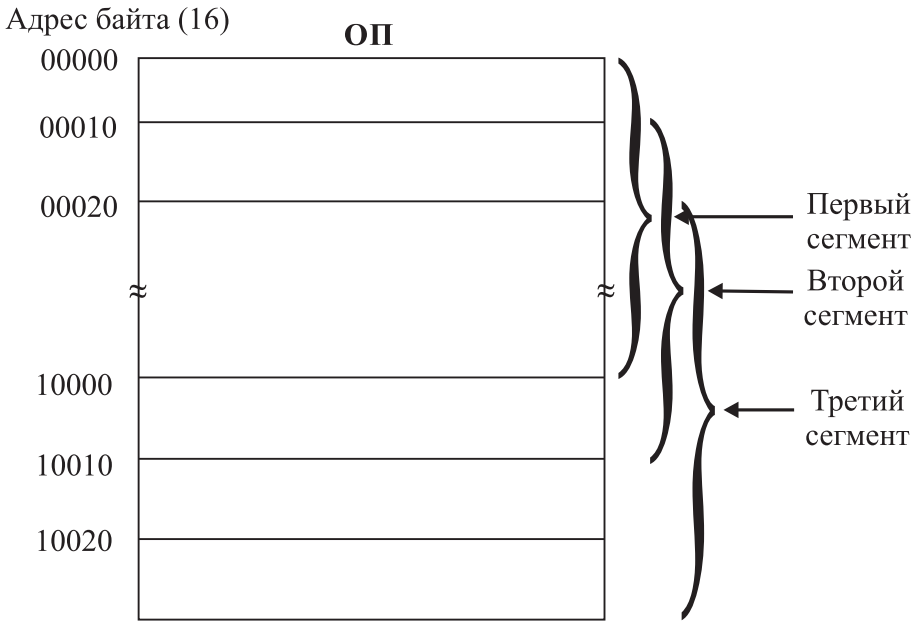


Рис. 2.6 – Разбиение ОП на параграфы

Например, пусть $(CS) = 0002h$, а $(IP) = 0100h$, тогда $R = 2h \times 10h + 100h = 120h$ (рис. 2.7).

В пределах текущего сегмента IP может обращаться к любой ячейке ОП, имеющей смещение относительно начала сегмента $0 \div 2^{16} - 1$, т. е. $0 \div 65535$. Число $2^{16} = 65536 = 10000h$ называется длиной сегмента.

В пределах текущего сегмента *IP* может обращаться к любой ячейке ОП, имеющей смещение относительно начала сегмента $0 \div 2^{16} - 1$, т. е. $0 \div 65535$. Число $2^{16} = 65536 = 10000h = 64K$ называется длиной сегмента.

Имея в своем распоряжении четыре сегментных регистра, программа (а точнее — выполняющий ее ЦП) использует их по-разному: *CS* — для адресации ма-

шинных команд, *SS* — для адресации ячеек стека, *DS* — для основной адресации данных, *ES* — для дополнительной адресации данных. Поэтому в любой момент времени любая машинная программа имеет в своем распоряжении четыре логических сегмента ОП размером 64*K*. Каждый из этих логических сегментов играет строго определенную роль при выполнении команд машинной программы:

- *сегмент кода* — сегмент ОП, номер начального параграфа которого находится в регистре *CS*. Следующая команда программы выбирается только из сегмента кода;
- *сегмент данных* — сегмент ОП, номер начального параграфа которого — в регистре *DS*. Команды, у которых ячейки памяти задаются только одним операндом, имеют дело с этим сегментом данных;
- *дополнительный сегмент данных* — сегмент ОП, номер начального параграфа которого — в регистре *ES*. Команды, у которых ячейки памяти задаются обоими операндами, используют и сегмент данных, и дополнительный сегмент данных;
- *сегмент стека* — сегмент ОП, номер начального параграфа которого в регистре *SS*. Команды, выполняющие операции со стеком, имеют дело с тем стеком, на который «указывает» *SS*.

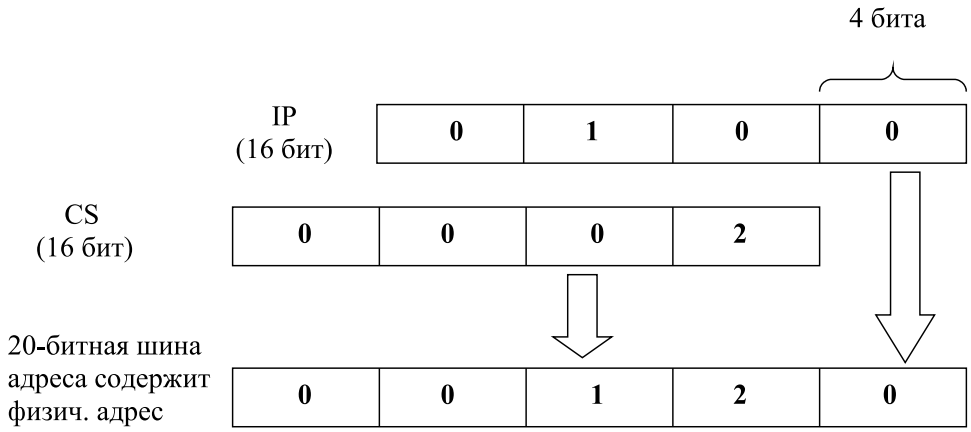


Рис. 2.7 – Получение физического адреса

Перечисленные логические сегменты как бы «высвечиваются» ЦП из одномегабайтового адресного пространства ОП. Причем они могут пересекаться друг с другом или вообще совпадать. Выполняя запись в сегментные регистры (например, с помощью команды *mov*) нового содержимого, программа выполняет замену «высвечиваемых» сегментов.

Для того, чтобы можно было обращаться к любой ячейке логического сегмента, в распоряжении программы имеется регистр, содержащий смещение искомой ячейки относительно начала сегмента. Это:

- для сегмента кода — *IP*;
- для сегмента данных — *BX*, *SI*, *DI*;
- для дополнительного сегмента данных — *BX*, *DI*;
- для сегмента стека — *SP*.

Таким образом, логический адрес любой ячейки ОП представляет собой пару:

$$(S, L) = ((\text{регистр сегмента}), (\text{регистр смещения})),$$

где S — начальный адрес сегмента (номер параграфа); L — смещение ячейки относительно начала сегмента.

Далее будем называть S *адресом-сегментом*, а L — *адресом-смещением*.

Преобразование логического адреса в физический происходит при попадании соответствующей машинной команды на ЦП так, как это показано на рис. 2.4 Так как это преобразование происходит во время выполнения программы, то оно называется *динамическим преобразованием адреса*. (*Статическая* операция производится до начала выполнения программы.)

2.4 Алгоритм работы процессора

На рис. 2.8 приведен алгоритм выполнения машинных команд в ЦП. Он представляет собой упрощенный вариант работы реального процессора *i8086*, в котором некоторые этапы выполняются во времени не последовательно, а параллельно.

Данный алгоритм представляет собой бесконечный цикл, который инициируется сразу же после включения питания. На одной итерации алгоритма ЦП выполняет одну машинную команду. Это выполнение начинается с определения реального адреса команды в ОП. Этот адрес выдается из ЦП на шину адреса. Получив его, ОП помещает следующую команду на шину данных, и ЦП вводит команду в очередь команд. Пока дешифрируется эта команда, определяется ее длина в байтах и производится увеличение содержимого *IP* на эту длину, в результате чего *IP* адресует следующую машинную команду. После этого цикл повторяется.

Последовательная выборка команд из памяти и их выполнение продолжаются до тех пор, пока очередной командой, поступившей из ОП на ЦП, не окажется команда перехода. Команды перехода позволяют изменить естественный порядок следования машинных команд. Они делятся на команды безусловного и команды условного перехода.

Команды безусловного перехода обязательно изменяют естественный порядок выполнения команд. Существуют пять машинных команд безусловных переходов. Все они имеют одну и ту же ассемблерную мнемонику ***jmp*** и один операнд. Эти команды можно разбить на две группы: команды близких и команды дальних переходов.

Команда близкого перехода выполняет безусловный переход в пределах текущего сегмента кода. Это делается путем замещения содержимого *IP* (т. е. внутрисегментного адреса следующей по порядку команды) адресом, задаваемым самой командой перехода.

Команда дальнего перехода выполняет безусловный переход на ячейку ОП, расположенную за пределами текущего сегмента кода. Это делается путем замещения содержимого не только *IP*, но и *CS* значениями, содержащимися в самой команде перехода. Следствием этого является замена логического сегмента кода, и поэтому до тех пор, пока на ЦП не поступит следующая команда дальнего перехода, все последующие команды будут выбираться из нового сегмента.

Кроме команд *jmp* безусловный переход выполняют команды *call*, *ret*, *int*, *iret*. Эти команды мы рассмотрим позже, а пока лишь отметим, что мнемоникам *call* и *ret* соответствуют по две машинные команды, одна из которых выполняет близкий, а другая — дальний безусловный переход. Команды *int* и *iret* выполняют только дальние переходы.

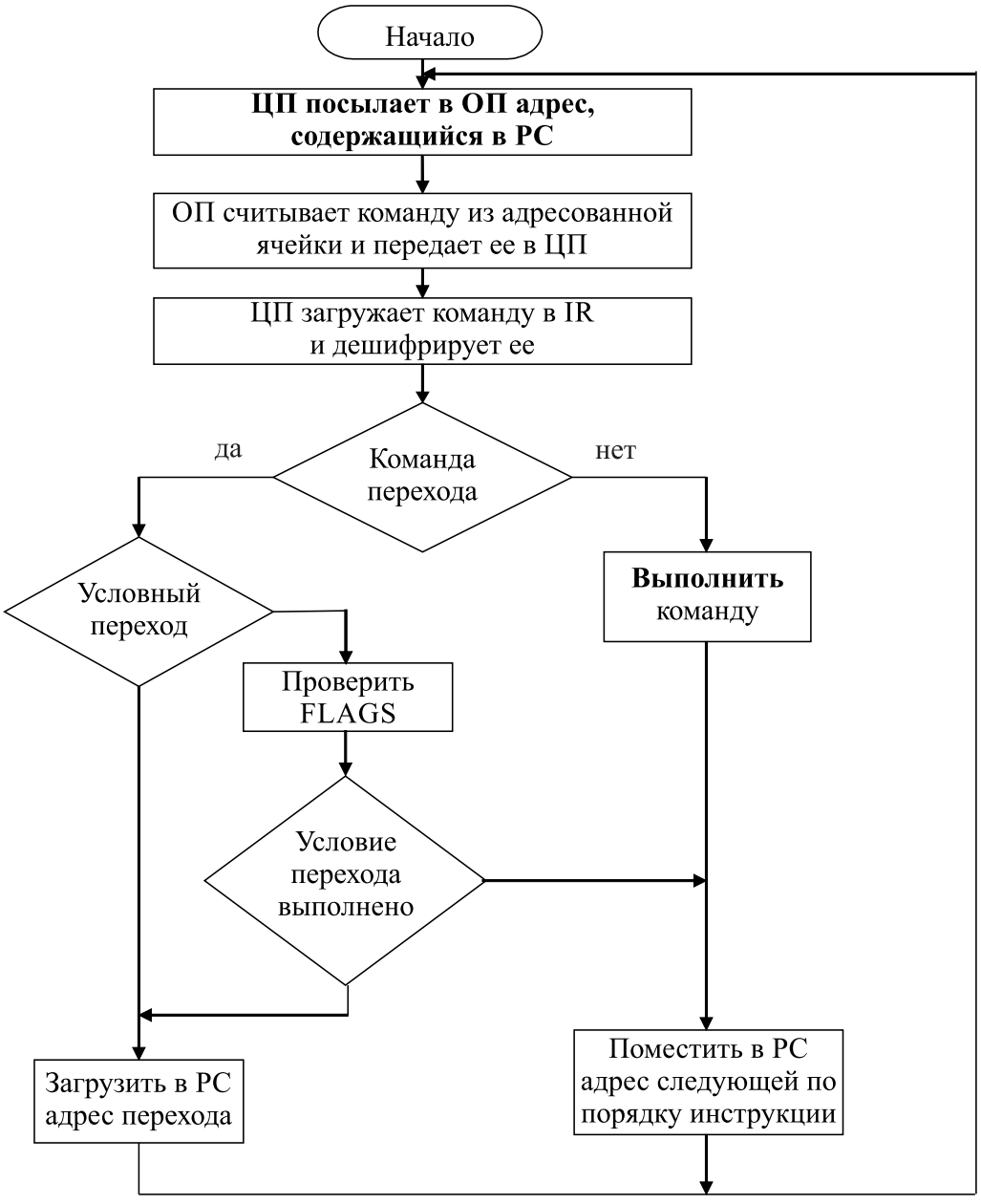


Рис. 2.8 – Алгоритм работы ЦП

Команды условных переходов замещают или не замещают содержимое *IP* в зависимости от результатов предыдущих команд, отраженных в регистре *FLAGS*. Например, если после команды *sub* (вычитание) в программе находится команда *jz* (переход по нулю), то переход осуществляется в том случае, если результат вычитания нулевой, и поэтому *ZF* (флаг нуля в *FLAGS*) установлен. Если же *ZF* = 0 (ненулевой результат вычитания), переход не производится. Так как ко-

манды условных переходов могут выполнять только очень близкие переходы (не далее 128 байтов), то для реализации больших переходов (в том числе и за пределы сегмента кода), каждый такой оператор дополняется оператором безусловного перехода.

Контрольные вопросы по главе 2

1) Рассмотрите структуру центрального процессора. Для чего служат его основные блоки?
2) Что такое регистр флагов? Какую роль он играет?
3) Как осуществляется адресация памяти? Как получить реальный адрес, если известно содержание регистра *CS* и регистра *IP*?
4) Рассмотрите алгоритм работы центрального процессора. Какие команды позволяют осуществить безусловные переходы?
5) Рассмотрите алгоритм работы центрального процессора. Влияет ли регистр *FLAGS* на работу процессора?

Глава 3

ПРОГРАММИРОВАНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

После того, как были рассмотрены логические основы организации аппаратуры ЭВМ, перейдем к построению простейших машинных программ, выполняемых с помощью этой аппаратуры. Программы будут выполняться в среде операционной системы *DOS*. При этом для того, чтобы «не отдаляться» от аппаратуры, при построении своих программ в первой части пособия мы будем использовать единственную системную программу — *Debug*.

Debug является отладчиком, то есть программой, предназначенной для оказания помощи программистам в поиске ошибок в программах, исходные тексты которых написаны на ассемблере или каких-то других языках программирования. С помощью этой программы производится анализ и заполнение ячеек регистровой и оперативной памяти, осуществляется пошаговое выполнение программы. Преследуя цель лучше понять механизм выполнения машинных программ, мы не будем пока использовать для записи программ даже язык ассемблера, ограничившись лишь использованием вместо цифровых КОПов машинных команд соответствующих ассемблерных мнемоник [4].

Рассмотрим происхождение слова *Debug*. «*Bugs*» (дословно «насекомые») в переводе со слэнга программистов означает «ошибки в программе». Используя *Debug* для пошагового запуска программы и наблюдая, как программа работает на каждом этапе, мы можем найти ошибки и исправить их. Этот процесс называется отладкой («*debugging*»), отсюда и произошло название программы *Debug*.

3.1 Чтение и заполнение регистров

Прежде, чем запустить *Debug*, необходимо запустить ту операционную систему (ОС), в среде которой будут выполняться и *Debug*, и получаемые с помощью него наши программы. В качестве такой ОС везде далее используется любая система из семейства *DOS*. Эти ОС имеют схожие пользовательские и программные интерфейсы. Примером *DOS* является свободно распространяемая операционная система *FREEDOS*. Так как эта ОС является самостоятельной системой, выполняемой на «голой» аппаратуре, то для ее запуска из среды какой-то другой ОС, например из одной из *WINDOWS* или из *UNIX*, предварительно следует запустить программный иммитатор аппаратуры ЭВМ, например *BOCHS*.

В результате запуска *DOS* на черном экране появится ее приглашение для ввода нами команды, например, как на рис. 3.1.

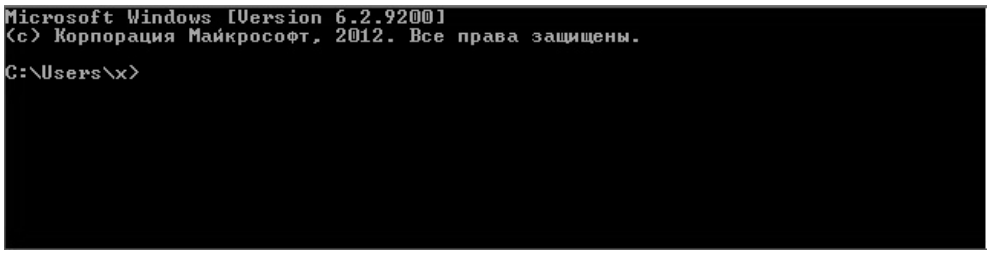
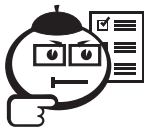


Рис. 3.1 – Приглашение DOS

Данное приглашение означает, что в данный момент времени текущим логическим диском является *C:*, а текущим каталогом — Users\x на этом логическом диске. Далее на экране находится само приглашение — символ «>».



Запустите *Debug*, набрав его название после приглашения *DOS*, например, как на рис. 3.2.

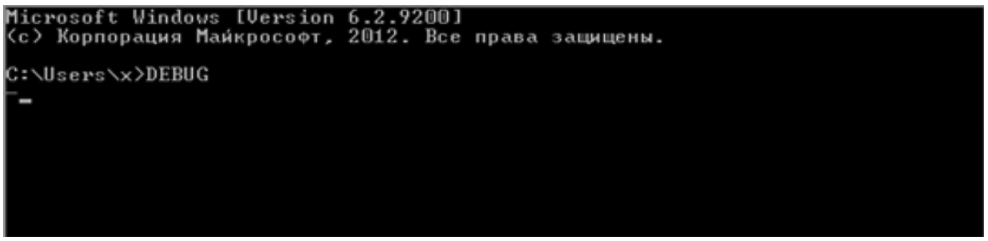


Рис. 3.2 – Работа программы Debug

Debug можно вызвать и с помощью программы *DOS NAVIGATOR* или с помощью другой аналогичной утилиты, имеющейся на Вашей ЭВМ. Для этого надо найти в каталоге файлов файл *DEBUG.COM*, установить на него курсор-маркер и нажать клавишу <*Enter*>.

Дефис «_», который Вы видите в качестве ответа на Вашу команду, — это приглашение программы *Debug*. Это означает, что *Debug* ждет Вашей команды. Чтобы

покинуть *Debug* и вернуться в *DOS*, напечатайте «***Q***» («*Quit*») около дефиса и нажмите «*Enter*», как на рис. 3.3.

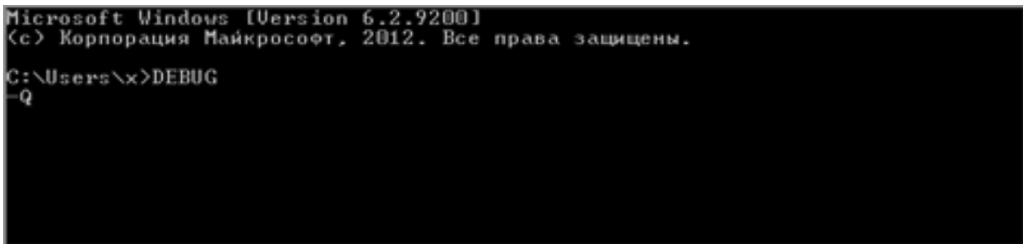
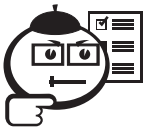


Рис. 3.3 – Команда Q Debug



. . . . . .

Попробуйте выйти и затем обратно вернуться в *Debug*:
_Q
C:\>DEBUG

. . . . . .

Мы начнем использование *Debug* с того, что попросим его показать содержимое регистров ЦП (микропроцессора *i8086*) с помощью команды ***R*** (от «*Register*»), рис. 3.4.

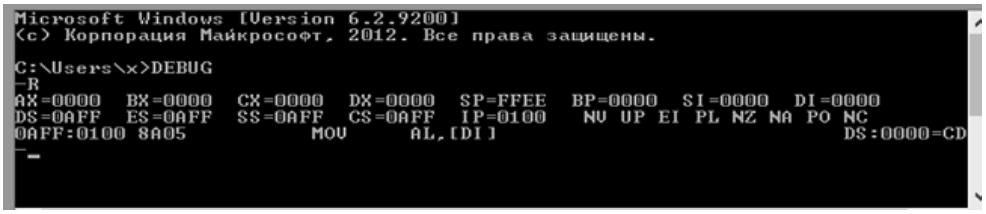


Рис. 3.4 – Команда R Debug

Возможно, на своем экране вы увидите другие числа во второй и третьей строках. А сейчас обратим внимание на первые четыре регистра, *AX*, *BX*, *CX* и *DX*, о значениях которых *Debug* сообщил, что они все равны 0000. Это регистры общего назначения. Остальные регистры *SP*, *BP*, *SI*, *DI*, *DS*, *ES*, *SS*, *CS*, *IP* являются регистрами специального назначения. Так как каждый из 13 регистров *i*8086 является словом и имеет длину 16 бит, то его содержимое представлено на экране в виде четырехзначного шестнадцатеричного числа.

Команда *Debug R* не только высвечивает регистры. Если указать в команде имя регистра, то *Debug* поймет, что мы хотим взглянуть на содержимое именно этого регистра и может быть изменить его. Например, мы можем изменить содержимое *AX*:

_R AX
AX = 0000
:3A7

Теперь можно убедиться (рис. 3.5) в том, что в регистре *AX* содержится 3*A*7*h*:

_R

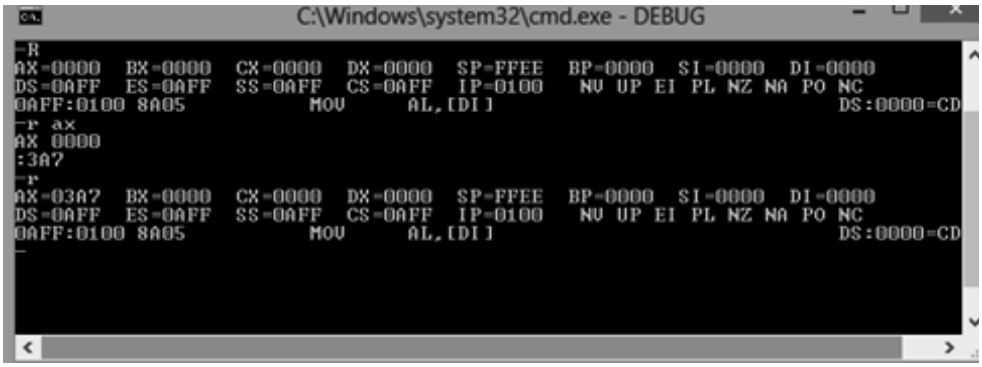


Рис. 3.5 – Содержимое регистра AX

Так и есть. Итак, мы можем помещать шестнадцатеричное число в регистр с помощью команды *R*, указывая имя регистра и вводя его новое значение после двоеточия.

3.2 Сложение двух чисел

Теперь перейдем к написанию и выполнению с помощью *Debug* программы на машинном языке. Простейшая такая программа состоит всего из одной машинной команды. Допустим, что эта команда выполняет сложение двух чисел, предварительно записанных в регистры.

ADD AX, BX

Для того чтобы данная машинная команда была исполнена ЦП, необходимо выполнить четыре действия:

1) выбрать место в ОП, куда поместить (записать) машинную команду;
2) выполнить запись команды на выбранное место;
3) сообщить ЦП о том, где расположена наша машинная команда;
4) запустить ЦП для того, чтобы он исполнил команду.

Debug рекомендует использовать область (сегмент) памяти, который имеет длину 65536 байт (64 Кбайт). Его адрес высчитывается, как значение CS, умноженное на 16. На практике такое умножение выполняется очень просто — к номеру параграфа в шестнадцатеричном коде (он и содержится в *CS*) справа приписывается 0. Чаще всего берут смещение 100*h*. Ячейки сегмента с меньшими внутрисегментными адресами резервируют для служебной информации, помещаемой туда *DOS*.

Итак, мы хотим сложить числа 3*A*7*h* и 92*Ah*. Запишем эти числа соответственно в регистры *AX* и *BX*, воспользовавшись командой *R Debug*. Для суммирования содержимого *AX* и содержимого *BX* будем использовать машинную команду:

D801

Данная машинная команда имеет длину два байта. Старший байт команды — *D*8, а младший — 01.

После того, как мы выбрали место для размещения нашей машинной команды в ОП, ***запишите*** ее туда с помощью команды *Debug* ***E*** (от «*ENTER*»), предназначенной для исследования и изменения памяти, рис. 3.6:

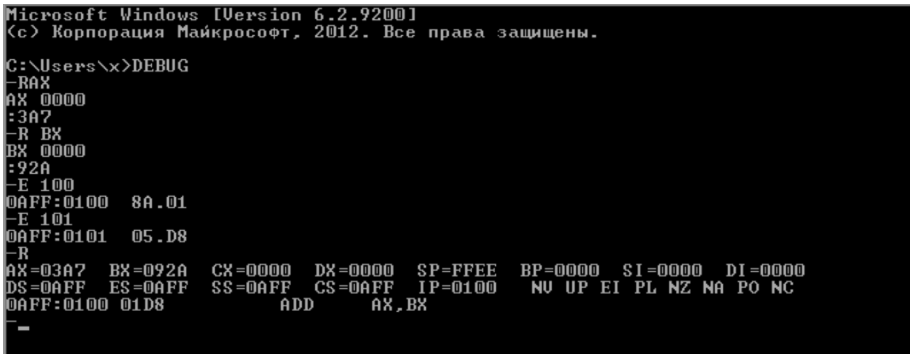


Рис. 3.6 – Команда E Debug

В результате числа 01*h* и *D*8*h* расположены по адресам *0AFF:0100* и *0AFF:0101*. Числа 8*Ah* и 05*h* представляют собой старое содержимое указанных ячеек ОП, которое осталось от ранее выполнявшихся программ. Номер начального параграфа сегмента, который вы увидите, возможно, будет другим, но это различие не будет влиять на нашу программу. Обратите внимание, что младший байт команды (01*h*) записан нами в ячейку с меньшим адресом, а старший байт (*D*8*h*) — с большим адресом. Заметим, что для записи нескольких соседних байтов мы можем использовать всего одну команду *E*, разделяя пробелом уже записанный байт от следующего:

_E 100

*0AFF:0100 8A.**01** 05.**D8***

Прежде чем идти дальше, *проверьте* результат наших предыдущих действий с помощью команды *R*.

Последняя строка, выданная *Debug*, содержит:

1) логический адрес в ОП машинной команды;
2) шестнадцатеричный код этой команды, причем младший байт команды изображен слева, а старший справа (содержимое регистров показывается наоборот — старший байт слева, а младший справа);
3) ее мнемоническое представление.

Почему именно нашу команду показал *Debug*? Ответ заключается в том, что *Debug* высвечивает ту команду, младший байт которой имеет адрес: *(CS):(IP)*.

Напомним, что указатель команды *IP* содержит внутрисегментное смещение младшего байта той машинной команды, которая будет исполняться следующей на ЦП. В процессе выполнения машинной программы ее команды сами могут изменять содержимое *IP*. А пока для этой цели мы будем использовать *Debug*. После своего запуска *Debug* всегда записывает в *IP* 100*h*. В процессе выполнения нашей машинной программы это значение будет меняться. Пользуясь командой *R Debug (R IP)*, мы всегда можем записать в *IP* требуемое нам значение.

Теперь регистры и ОП готовы для исполнения нашей машинной команды. ***Попросите*** *Debug* ее выполнить, используя команду ***T*** (от «*Trace*»), которая выполняет одну команду за шаг, а затем показывает содержимое регистров (рис. 3.7). После каждого запуска *IP* будет указывать на следующую команду, в нашем случае будет указывать на 102*h*. Мы не помещали никакой команды в 102*h*, поэтому

в последней строке распечатки мы увидим команду, оставшуюся от предыдущей программы:

Рис. 3.7 – Команда T Debug

Регистр *AX* теперь содержит число *CD*1*h*, которое является суммой 3*A*7*h* и 92*Ah*. А регистр *IP* указывает на адрес 102*h*, так что в последней строке распечатки регистров мы видим команду, расположенную в памяти по адресу *l*02*h*, а не по адресу *l*00*h*.

Указатель команды *IP* вместе с регистром *CS* всегда указывает на следующую машинную команду, которую нужно выполнить процессору. Если мы опять напечатаем «*T*», то выполнится следующая команда. Но не делайте этого сейчас — ваш процессор может «зависнуть».

Если нужно выполнить введенную машинную команду еще раз, то есть сложить 92*Ah* и *CD*1*h* и сохранить новый ответ в *AX*, то надо указать процессору, где найти следующую команду и чтобы этой следующей командой оказалась та же «*add ax,bx*», расположенная по адресу *l*00*h*. Изменить значение регистра *IP* на *l*00*h* можно, используя команду *R* (рис. 3.8).

```
-R IP
  IP 0102
: 100
```

После этого:

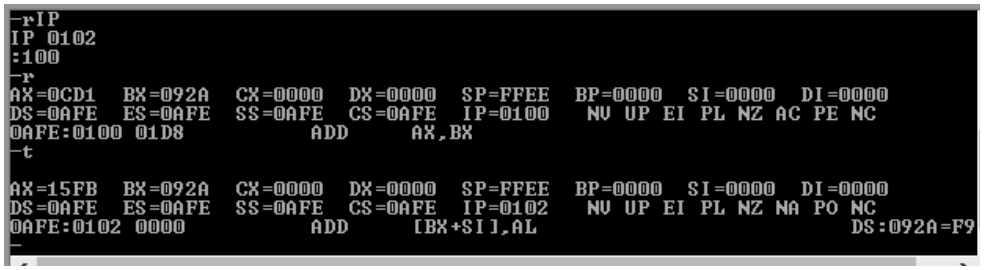
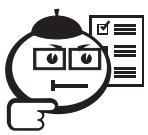


Рис. 3.8 – Изменение адреса текущей команды



Попробуйте еще раз ввести команду *Debug T* и убедитесь, что регистр *AX* содержит число 15*FBh*.

Следовательно, перед тем, как использовать команду *T*, необходимо проверить регистр *IP* и соответствующую его значению машинную команду, располагаемую

в нижней части распечатки (листинга), выдаваемой командой *R*. В результате вы будете уверены, что процессор выполнит требуемую команду.

До этого арифметические действия совершались над шестнадцатебитными словами. Но рассматриваемый процессор может выполнять действия и над восьмибитными байтами.

Каждый регистр общего назначения может быть разделен на два байта — старший байт (первые две шестнадцатеричные цифры) и младший байт (следующие две шестнадцатеричные цифры). Название каждого из полученных регистров складывается из первой буквы названия регистра (от «*A*» до «*D*»), стоящей перед *X* в слове, и буквы *H* для старшего байта или буквы *L* для младшего. Например, *DL* и *DH* — регистры длиной в байт, а *DX* — длиной в слово. На две части делятся только регистры общего назначения: AX (AH, AL), BX (BH,BL), DX (DH,DL), CX (CH,CL).

Проверим байтовую арифметику на машинной команде *add*. ***Введите*** два байта 00*h* и *C*4*h*, начиная с адреса 0100*h*(*E* 100). Внизу листинга регистров вы увидите команду «*ADD AH,AL*», которая суммирует два байта регистра *AX* и записывает результат в старший байт *AH*.

Затем ***загрузите*** в *AX* число 0102*h (R AX)*. Таким образом, вы поместите 01*h* в регистр *AH* и 02*h* в регистр *AL*. Установите регистр *IP* в 100*h*, выполните команду *T*, и вы увидите, что регистр *AX* теперь содержит 0302. Результат сложения 01*h* и 02*h* будет 03*h*, и именно это значение находится в *AH*.

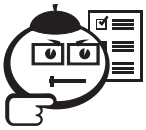
3.3 Вычитание двух чисел

Напишем машинную команду для вычитания *BX* из *AX*, так что после двух вычитаний в регистре *AX* появится результат 3*A*7*h*. Тогда мы вернемся к той точке, с которой начали. ***Запишите*** с помощью команды *E* команду вычитания в ОП:

_E 100

*0AFF:0100 01.**29** D8.**D8***

Листинг регистров (не забывайте установить *IP* в 100*h*) должен теперь показать команду «***sub*** *ax,bx*», которая вычитает содержимое регистра *BX* из регистра *AX* и размещает результат в *AX*.

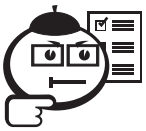


Выполните эту машинную команду с помощью команды *Debug T*. *AX* должен содержать *CD*1. Измените *IP* так, чтобы он указывал на эту машинную команду, и выполните ее опять (не забывайте сначала проверить команды внизу листинга регистров), *AX* теперь должен содержать 03*A*7*h*.

Используйте машинную команду *sub*, чтобы подтвердить свои знания о представлении отрицательных чисел. Вычтем из 0 (в регистре *AX*) единицу (в *BX*). В результате *AX* должен содержать *FFFFh* (–1).

3.4 Умножение двух чисел

Команда умножения называется «***mul***», а машинный код для умножения *AX* на *BX* — *E*3*F*7*h*. Так как умножение двух 16-битных чисел может дать 32-разрядный ответ, то машинная команда *mul* сохраняет результат в двух регистрах — *DX* и *AX*. Старшие 16 бит помещаются в регистре *DX*, а младшие — в *AX*. Эта комбинация регистров записывается как *DX:AX*.



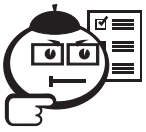
Введите с помощью *Debug* команду умножения *E*3*F*7*h* по адресу 0100*h* и установите *AX* = 7*C*4*Bh* и *BX* = 100*h*. Вы увидите команду в листинге регистров как «*mul bx*», без всяких ссылок на регистр *AX*. При умножении слов процессор *i8086* всегда умножает регистр, имя которого вы указываете в машинной команде, на регистр *AX* и сохраняет ответ в паре регистров *DX:AX*.

Перед запуском команды умножения перемножим 100*h* и 7*C*4*Bh* вручную. Три цифры 100 имеют в шестнадцатеричной системе такой же эффект, как и в десятичной. Так что умножение на 100*h* просто добавит два нуля справа от шестнадцатеричного числа. Таким образом, 100*h* × 7*C*4*B* = 7*C*4*B*00*h*. Этот результат слишком длинен для того, чтобы поместиться в одном слове, поэтому мы разбиваем его на два слова 007*Ch* и 4*B*00*h*.

Используйте *Debug* для запуска машинной команды умножения. Вы увидите, что *DX* содержит слово 007*Ch*, а *AX* содержит слово 4*B*00*h*.

3.5 Деление двух чисел

При делении сохраняется как результат, так и остаток от деления.



Поместите машинную команду *F*3*F*7*h* по адресу 0100*h* (и 101*h*). Как и команда *mul, div* использует пару регистров *DX:AX*, не сообщая об этом, так что все, что мы видим, — это «*div bx*». Загрузите в регистры значения: *DX* = 007*Ch* и *AX* = 4*Bl*2*h*; регистр *BX* по-прежнему должен содержать 0100*h*.

Подсчитаем результат вручную: 7*C*4*B*12*h*/100*h* = 7*C*4*Bh* с остатком 12*h*. После выполнения команды деления по адресу 0100*h* мы получим для *AX* = 7*C*4*Bh* результат нашего деления и для *DX* = 0012*h* остаток.

Контрольные вопросы по главе 3

1) Как запустить *Debug*, выйти из системы, просмотреть содержимое регистров?
2) В какой системе счисления представлены числа в *Debug*?
3) Как изменить значения регистров?
4) Как занести инструкцию (команду) в память?
5) Как произвести арифметические операции — сложение, вычитание, умножение, деление?

Глава 4

ВЫВОД СИМВОЛОВ НА ЭКРАН

В предыдущей главе мы выполняли с помощью *Debug* запись в ОП и последующее выполнение всего одной машинной команды, производящей над своими операндами какую-то арифметическую операцию. Теперь напишем программу, состоящую из нескольких команд. Первая команда программы будет получать управление из *Debug*. Последняя команда нашей программы должна возвращать управление обратно в *Debug*. В процессе своего выполнения программа будет обращаться за помощью к *DOS* с целью вывода на экран строки символов. Для того, чтобы вызвать из прикладной программы системную подпрограмму *DOS* или *BIOS*, нужно разместить в этой программе машинную команду программного прерывания. Термин «прерывание» означает, что выполнение нашей программы прерывается (приостанавливается) на время, необходимое для выполнения требуемой системной программы. Команда программного прерывания обозначается как ***int*** (от «*Interrupt*» — прерывание). Команда *int* для функций *DOS* имеет вид «*int 21h*», в машинном коде 21*CDh*.

4.1 Вывод одного символа

Примером функции *DOS*, выполнение которой мы можем запросить из программы с помощью команды «*int 21h*», является вывод символа на экран. Для того, чтобы различать функции *DOS*, которых много, используется регистр *AH*. При выводе одного символа в него помещается 02*h*. В регистр *DL* заносится код *ASCII* выводимого символа. *В табл. 4.1* приведены отображаемые (видимые на экране) коды *ASCII*.

Допустим, что мы хотим вывести символ *A*, тогда в регистр *DL* мы должны поместить число 41*h*. ***Подготовьте*** регистры и память для последующего выполнения команды *INT 21*. Для этого в регистры *AX* и *DX* запишем с помощью *Debug* числа 0200*h* и 0041*h*, а по адресу 0100*h* в ОП запишем 21*CDh*. После этого можно перейти к выполнению машинной команды программного прерывания. Для этого

не рекомендуется использовать команду *T Debug*. Дело в том, что в результате выполнения «*int 21*» начинает выполняться системная подпрограмма вывода символа, состоящая из многих машинных команд. Пошаговое выполнение этой подпрограммы вам скоро наскучит. Но если вы не доведете его до конца, то ваш компьютер «зависнет». Но если вы все-таки протрассируете несколько шагов, можно выйти из *Debug* с помощью команды *Q*, которая ликвидирует беспорядок. (При выполнении трассировки обратите внимание на то, что изменилось первое число, являющееся составляющей адреса. Это обусловлено тем, что единственная машинная команда нашей программы и подпрограмма *DOS* находятся в разных сегментах ОП.)

Таблица 4.1 – Коды ASCII

Символ ASCII	16-рич. код	Символ ASCII	16-рич. код	Символ ASCII	16-рич. код	Символ ASCII	16-рич. код
	20	8	38	P	50	h	68
!	21	9	39	Q	51	i	69
“	22	:	3A	R	52	j	6A
#	23	;	3B	S	53	k	6B
$	24	<	3C	T	54	l	6C
%	25	=	3D	U	55	m	6D
&	26	>	3E	V	56	n	6E
‘	27	?	3F	W	57	o	6F
(	28	@	40	X	58	p	70
)	29	A	41	Y	59	q	71
*	2A	B	42	Z	5A	r	72
+	2B	C	43	[	5B	s	73
,	2C	D	44	\	5C	t	74
-	2D	E	45	]	5D	u	75
.	2E	F	46	^	5E	v	76
/	2F	G	47	_	5F	w	77
0	30	H	48	`	60	x	78
1	31	I	49	a	61	y	79
2	32	J	4A	b	62	z	7A
3	33	K	4B	c	63	{	7B
4	34	L	4C	d	64	|	7C
5	35	M	4D	e	65	}	7D
6	36	N	4E	f	66	~	7E
7	37	O	4F	g	67	DEL	7F

В этом случае намного удобнее использовать команду *Debug* ***G*** (от «*GO*»), после которой пишется адрес-смещение, на котором мы хотим остановиться:

_G 102

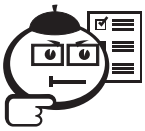
A

AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000 DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC 3970:0102 BBE5 MOV SP,BP

DOS вывел на экран букву *A* и возвратил затем управление в нашу программу. (Машинная команда, размещенная по адресу 102*h*, осталась от другой программы, поэтому последняя строка вашего листинга может выглядеть по-другому.)

4.2 Команда завершения программы

Машинная команда «*int 20h*» сообщает *DOS* о том, что мы хотим выйти из нашей программы и чтобы управление опять вернулось в *DOS*. В нашем случае эта команда вернет управление в *Debug*, так как мы запускаем нашу программу не непосредственно из *DOS*, а из *Debug*.



Введите команду 20*CDh*, начиная с адреса *l*00*h*, а затем проделайте следующее (не забудьте проверить команду «*int 20h*» с помощью команды *R DEBUG*):

```
_G 102
Program terminated normally
_R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20 INT 20
_G
Program terminated normally
_R
```

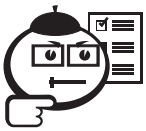
Результат команды *G* аналогичен результату команды *G 102*. Любая из этих команд *Debug* выполняет всю программу (сейчас она состоит всего из одной команды — «*int 20h*») и затем возвращается к началу. Когда мы начали выполнение, *IP* был установлен в *l00h*, т. к. мы заново запустили *Debug*. После выполнения *G IP* опять содержит *l*00*h*.

Нужно помещать машинную команду «*int 20h*» в конец любой программы для того, чтобы красиво передать управление *DOS* (или *Debug*). Для начала поместим ее после команды «*int 21h*» и получим программу из двух команд, выполняющую вывод символа на экран. Для этого, начиная с адреса 100*h*, ***введите*** одну за другой две машинные команды — 21*CDh* и 20*CDh*.

Когда у нас была только одна машинная команда, то мы могли «пролистать» ее командой *R Debug*, но теперь у нас две команды. Чтобы увидеть их, воспользуемся командой *Debug* ***U*** (от «*Unassemble*» — разассемблирование):

```
_U 100
3970:0100 CD21 INT 21
3970:0102 CD20 INT 20
```

Далее идут еще 12 строк листинга, содержащие команды, оставшиеся в памяти от предыдущих программ.



Поместите в регистр *AH* значение 02*h*, а в регистр *DL* код любого символа, например код символа *F* — 46*h*. Затем введите команду *G*, чтобы увидеть символ на экране:

_G
F
Program terminated normally

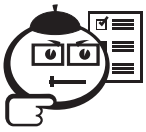
До этого мы вводили команды программы в виде чисел, например 21*CDh*. Но это слишком тяжелая работа, и избавиться от нее помогает команда *Debug **A*** (от «*Assemble*» — ассемблирование). Эта команда помогает вводить мнемонические (человекочитаемые) машинные команды. Применим команду *A* для ввода нашей программы:

_A 100
3970:0100 INT 21
3970:0102 INT 20
3970:0104

Команда *A* сообщает *Debug* о том, что мы хотим ввести машинные команды в мнемонической форме, а число 100 в команде означает, что ввод машинных команд начинается с ячейки 100*h*.

4.3 Пересылка данных между регистрами

До сих пор мы записывали требуемые числа в регистры с помощью команды *R Debug*. Но обычно это делает команда самой программы — ***mov***. Эта же команда выполняет пересылку чисел между регистрами.



Поместите 1234*h* в *AX* (12*h* в регистр *AH* и 34*h* в *AL*) и *ABCDh* в *DX* (*ABh* в *DH* и *CDh* в *DL*). С помощью команды *A* введите машинную команду «*mov ah,dl*». Эта команда пересылает (копирует) число из *DL* в *AH*. *AL* при этом не используется. Если вы протрассируете эту строку, то увидите, что *AX* = *CD*34*h* и *DX* = *ABCDh*. Изменился только *AH*. Теперь он содержит копию числа из *DL*.

Команда *mov* пересылает число из второго регистра в первый, и по этой причине мы пишем *AH* перед *DL*. Машинный код данной команды *D*488*h*. Существуют другие формы этой же команды *mov*. Они имеют другие машинные коды и выполняют другие операции пересылки. Например, следующая команда (*C*389*h*) выполняет пересылку не байтов, а слов между двумя регистрами *AX* и *BX*:

3970:0100 89C3 MOV BX, AX

Следующая форма команды *mov* записывает значение числа в регистр, не используя другой регистр-источник:

3970:0100 B402 MOV AH, 02

Эта команда загружает число 02*h* в регистр *AH*. Старший байт команды 02*h* является числом, которое мы хотим загрузить. ***Запишите*** эту команду в ОП и выполните ее.

Сложим все части вместе и построим длинную программу. Она будет печатать звездочку *, выполняя все операции сама, не требуя от нас установки регистров (*AH* и *DL*). Программа использует команды *mov* для того, чтобы установить регистры *AH* и *DL* перед выполнением команды «*int 21h*», выполняющей вызов функции *DOS*:

Пример

```
15AC:0100    B402      mov ah, 02
15AC:0102    B22A      mov dl, 2a
15AC:0104    CD21      int 21
15AC:0106    CD20      int 20
```

Введите программу и проверьте ее командой «*U 100*». Убедитесь, что *IP* указывает на ячейку *l00h*. Запустите программу командой *G*. В итоге на экране должен появиться символ *.

Теперь у нас есть законченная программа. Запишем ее на диск в виде *com*-файла для того, чтобы мы могли запускать ее прямо из *DOS*, просто набрав ее имя. Так как у программы пока нет имени, то мы должны его присвоить.

Команда *Debug* ***N*** (от «*Name*») присваивает файлу имя перед записью на диск. ***Напечатайте***:

```
_N Writestr.com
```

Эта команда не запишет файл на диск — она только назовет его *Writestr.com*.

Далее мы должны сообщить *Debug* о том, сколько байт занимает программа, для того чтобы он знал размер файла. Если вы посмотрите на разассемблированный листинг программы, то увидите, что каждая машинная команда в нем занимает два байта (в общем случае это не выполняется). У нас четыре команды, следовательно, программа имеет длину восемь байт.

Полученное число байт надо куда-то записать. Для этого *Debug* использует пару регистров *BX:CX*, и поэтому, поместив 8*h* в *CX*, мы сообщим *Debug* о том, что программа имеет длину в восемь байт. *BX* должен быть предварительно установлен в ноль.

После того, как мы установили имя и длину программы, мы можем записать ее на диск с помощью команды *Debug* ***W*** (от «*Write*»):

```
_W
Writing 0008 bytes
```

Теперь на диске есть программа *Writestr.com*, а мы с помощью *Q* покинем *Debug* и посмотрим на нее. ***Используйте*** команду *DOS dir*, чтобы увидеть справочную информацию о файле:

```
C:\>dir Writestr.com
Volume in drive C has no label
```

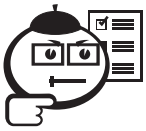
Directory of C:
WRITESTR.COM 8 6-30-93 10:05a
1 File (S) 18432 bytes free

Листинг директории сообщает, что *Writestr.com* находится на диске «*C:*» и его длина составляет восемь байт. Чтобы загрузить программу, наберите *writestr* в ответ на приглашение *DOS* и нажмите <*Enter*>. Вы увидите *.

Если мы хотим запустить свою *com*-программу не из *DOS*, а из *Debug*, то запуск *Debug* следует выполнить вместе с требуемым загрузочным модулем. Пример такого запуска: *Debug Writestr.com*. После этого с данной программой можно работать так, как будто мы создали ее только что с помощью *Debug*, а не считали с диска. Для сохранения скорректированной программы на диске следует выполнить те же операции, что и для нового файла.

4.4 Вывод на экран строки символов

Функция номер 02*h* для прерывания «*int 21h*» печатает один символ на экране. Другая функция, номер 09*h*, выводит на экран целую строку и прекращает вывод, когда находит символ «***$***».



. . . . . .

Поместим строку в память, начиная с ячейки 200*h*, чтобы эта строка не перепуталась с кодом самой программы. ***Введите*** следующие числа, используя команду *E 200*:

48 65 6C 6C 6F 2C 20 44
4F 53 20 68 65 72 65 2E
24

. . . . . .

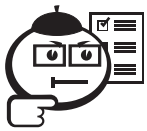
Последнее число 24*h* является *ASCII*-кодом для символа *$*, и оно сообщает *DOS*, что это конец строки символов. Теперь посмотрим, что сообщает эта строка, запустив следующую программу:

```
15AC:0100    B409      mov ah,09
15AC:0102    BA0002    mov dx, 0200
15AC:0105    CD21      int 21
15AC:0107    CD20      int 20
```

200*h* — адрес строки, которую мы ввели, а загрузка 200*h* в регистр *DX* сообщает *DOS* о том, где ее искать. ***Проверьте*** программу командой *U* и затем запустите ее командой *G*:

_G
Hello, DOS here.
Program terminated normally

Команда *Debug* ***D*** (от «*Dump*») дампирует (выводит содержимое) памяти на экран. Это похоже на действия, совершаемые командой *U* при распечатке машинных команд. Подобно *U* поместите после *D* адрес, чтобы сообщить *Debug*, откуда начинать дамп.



Наберите команду «*D 200*». Она выведет содержимое участка памяти, в котором хранится только что введенная строка.

Пример

```
_D 200
15AC:0200 48 65 6C 6C 6F 2C 20 44–4F 53 20 68 65 72 65 2E Hello, DOS here.
15AC:0210 24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8 $J.U..0.……..
```

После каждого числа, обозначающего адрес (как *15AC:0200* в примере), мы видим 16 пар шестнадцатеричных чисел, вслед за которыми записаны 16 *ASCII*-символов для этих пар (байтов). Например, в первой строке записаны символы, которые вы ввели. Символ *$* является первым символом в следующей строке, остальная часть строки представляет собой беспорядочный набор символов.

Точка «.» в окне *ASCII* означает, что это может быть как точка, так и специальный символ, например греческая буква «*pi*». Команда *Debug D* выдает только 96 из 256 символов символьного набора *IBM PC*, поэтому точка используется для обозначения остальных 160 символов. Часть специальных символов представляет собой прописные и строчные буквы русского алфавита. Соответствующие шестнадцатеричные коды приведены в табл. 4.1.

Теперь запишем программу, выводящую строку на экран, на диск. Программа начинается со строки 100*h*, и из выполненного дампа памяти можно видеть, что символ, следующий за знаком $, заканчивающим нашу строку, расположен по адресу 211*h*. Если посчитать разность 211h–100h, получим 111*h* — столько байт достаточно сохранить на диске, чтобы программа работала корректно, то есть сохраняем не только программный код, но и те данные, которые были занесены начиная с адреса 200*h*. ***Сохраните*** 111h в регистре *CX*, опять установив *BX* в ноль. Используйте команду *N*, чтобы дать имя программе (добавьте расширение *com*, чтобы запускать программу прямо из *DOS*), и затем командой *W* запишите программу и данные в дисковый файл.

Контрольные вопросы по главе 4

1) Что такое прерывание? Для чего оно используется?
2) Как вывести на экран один символ?
3) Как вывести на экран строку символов?
4) Как рассчитать, сколько байт будет занимать программа?
5) Как создать и сохранить программу с расширением .com?

Глава 5

ВЫВОД НА ЭКРАН ДВОИЧНЫХ ЧИСЕЛ

Приступим к решению задачи отображения на экран двоичных чисел, содержащихся в ячейках памяти, называемых регистрами. Напомним, что в ячейке любой памяти (в регистровой, ОП, ВП) любая информация содержится в виде последовательности битов. Поэтому отображение этой информации на экран в виде двоичного числа имеет практический смысл.

В ходе решения указанной задачи нам потребуется рассмотреть новые машинные команды, а также произвести знакомство с новыми понятиями. При этом мы начнем с более близкого знакомства с одним из битов рассмотренного ранее регистра *FLAGS* — флага переноса.

5.1 Флаг переноса

Если выполнить сложение чисел 1 и *FFFFh*, то получим 10000*h*. Это число не может быть записано в шестнадцатибитное слово, т. к. в нем помещаются только четыре шестнадцатеричные цифры. Единица в результате называется *переполнением*. Она записывается в специальную ячейку, называемую флагом переноса *CF* (от «*Carry Flag*»). Флаг содержит число, состоящее из одного бита, т. е. содержит или единицу, или ноль. Если флаг содержит единицу, то говорят, что он «установлен», а если ноль — «сброшен». Напомним, что флаг *CF* является одним из шестнадцати битов в регистре флагов *FLAGS*.

Выполните загрузку чисел 1 и *FFFFh* в регистры *BX* и *AX* и запишите в память команду «*add ax,bx*». После этого протрассируйте эту команду. В конце второй строки распечатки, полученной с помощью команды *R Debug*, вы увидите восемь пар букв. Последняя пара выглядит как *CY* (от «*Carry Ye*» — перенос есть), т. е. флаг переноса установлен.

Установите *IP* в 100*h* и прибавьте единицу к нулю в *AX*, повторив трассировку команды сложения. Флаг переноса переустанавливается в каждой операции сложения, и так как на этот раз переполнения не будет, то флаг будет сброшен.

С помощью команды *R* проверьте, что в качестве состояния флага *CF* листинг содержит *NC* («от *No Carry*» — нет переноса).

5.2 Циклический сдвиг

Допустим, что нам надо выполнить вывод на экран двоичного числа. За шаг мы выводим только один символ, и нам надо произвести выборку всех битов двоичного числа, одного за другим, слева направо. Например, пусть требуемое число есть 10000000*b*. Если мы сдвинем весь этот байт влево на одну позицию, помещая единицу во флаг переноса и добавляя ноль справа, а затем повторим этот процесс для каждой последующей цифры, то во флаге переноса будут по очереди содержаться все цифры нашего двоичного числа.

Команда ***rcl*** (от «*Rotate Carry Left*» — циклический сдвиг влево с переносом) сдвигает крайний левый бит во флаг переноса (в примере это 1), в то время как бит, находившийся до этого во флаге переноса, сдвигается в крайне правую позицию (т. е. в нулевой бит). В процессе сдвига все остальные биты сдвигаются влево. После определенного количества циклических сдвигов (17 для слова, 9 для байта) биты возвращаются на их начальные позиции, и вы получаете исходное число. На рис. 5.1 показано наглядное представление работы команды *rcl*.

Рис. 5.1 – Команда циклического сдвига влево через перенос *rcl*

Выполните с помощью *Debug* размещение по адресу 100*h* команды «*rcl bl,1*», которая циклически сдвигает байт в *BL* влево на один бит, используя флаг переноса. Поместите в регистр *BX* число *B*7*h* (10110111*b*) и протрассируйте эту команду несколько раз, например, как на рис. 5.2.

Обратите внимание, что после того, как команда *RCL BL,1* будет выполнена первый раз, регистр *BX* будет содержать число 6*Eh* (01101110*b*), а адрес *IP* будет иметь значение 102*h*. Чтобы еще раз выполнить команду *RCL BL,1*, нужно изменить значение регистра *IP* на 100*h*.

Убедитесь, что после 9 циклов регистр *BX* содержит опять *B*7*h*.

Как вывести на экран двоичное значение флага переноса? Из таблицы кодов *ASCII* видно, что символ «0» есть 30*h*, а символ «1» есть 31*h*. Таким образом, сложение флага переноса и 30*h* дает символ «0», когда флаг сброшен, и символ «1», когда он установлен. Для выполнения такого сложения удобно использовать команду ***adc*** (от «*Add with Carry*» — сложение с переносом). Эта команда складывает три числа: два числа, как и команда *add*, а также один бит из флага переноса.

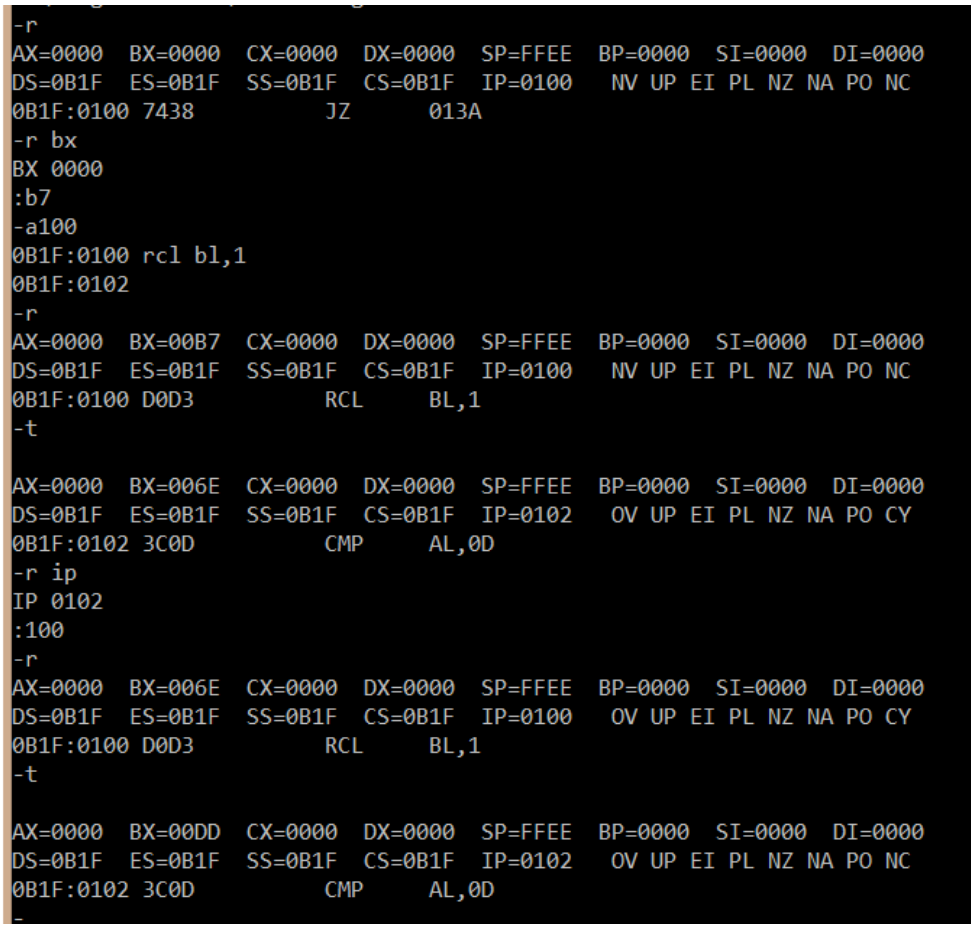
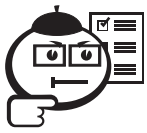


Рис. 5.2 – Трассировка команды RCL BL,1



Поместите в память после команды «*rcl bl,1*» команду «*adc dl,30*», которая выполнит сложение содержимого *DL* (0), 30*h* и флага переноса, поместив результат в *BL*. Записав далее команды, обеспечивающие вывод символа на экран и завершение программы, получим программу, выполняющую вывод на экран старшего бита регистра *BL*:

```
mov     dl, 00
rcl     bl, 1
adc     dl, 30
mov     ah, 02
int     21
int     20
```

Выполните эту программу для обоих значений старшего бита *BL*. Для записи в *BX* используйте команду *R Debug*.

5.3 Организация циклов

Если мы хотим вывести на экран все биты *BL*, то мы должны повторить операции циклического сдвига и распечатки флага переноса *CF* восемь раз (число битов в *BL*). Неоднократное повторение одних и тех же операций называется *циклом*.

Соответствующий алгоритм приведен на рис 5.3. На первом этапе переменной *CX* присваивается значение 8. Именно столько раз мы хотим повторить выполнение этапов, образующих «тело цикла». Переменная *CX* называется *счетчиком повторений*. Тело цикла образуют этапы «Получение кода *ASCII* для бита» и «Вывод символа». После того, как тело цикла выполнено, значение *CX* уменьшается на 1. На следующем этапе алгоритма значение *CX* сравнивается с 0. Если это значение ненулевое, то делается возврат для повторения тела цикла. Иначе — выполняется этап алгоритма, расположенный после цикла. На рис. 5.3 это этап «Завершение алгоритма».

Для организации циклов используются специальные машинные команды. Одной из них является команда ***loop***. Она записывается в конце цикла, т. е. после тех команд, выполнение которых следует повторить. Данная команда имеет один операнд — *адрес* первой из повторяемых машинных команд. Счетчик повторений цикла содержится в регистре *CX*. Данный регистр используется потому, что буква *C* в названии регистра *CX* означает «счетчик» (от «*Count*»). *CX* может использоваться также как регистр общего назначения.

Выполнение команды *loop* сводится к следующему.

Во-первых, она вычитает из содержимого регистра *CX* единицу.

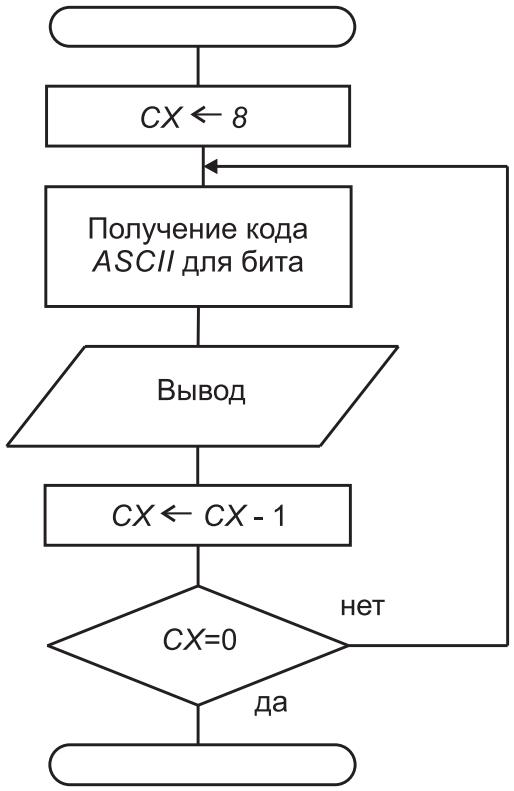
Во-вторых, она сравнивает полученное содержимое регистра *CX* с нулем, и если оно не равно 0, то делает переход по адресу, заданному в качестве операнда команды *loop*.

В-третьих, если переход не делается, то на ЦП начинает выполняться машинная команда, расположенная в программе сразу же за командой *loop*. Таким образом, наличие данной команды обеспечивает реализацию двух этапов алгоритма, приведенного на рис. 5.3.

Пример

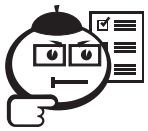
В качестве примера применения команды *loop* приведем программу, выполняющую вывод на экран четырех звездочек:

```
100     mov     ah,02
102     mov     dl,2a
104     mov     cx,4
107     int     21
109     loop    107
10B     int     20
```



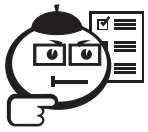
CX – счетчик повторений

Рис. 5.3 – Алгоритм вывода на экран содержимого байта



Выполните трассировку данной программы, наблюдая за содержимым регистров *IP* и *CX*. При этом нужно вспомнить, что не следует использовать команду *T Debug* для команды *int*. При достижении этой команды следует набрать команду *Debug «G d»*, где *d* — адрес в памяти команды, следующей за *int*. При достижении команды *«int 20»* вводится команда *G Debug*.

При применении команды «*G d*» необходимо знать адрес останова *d*. Исключить трассировку при достижении машинной команды *int* проще, если использовать команду *Debug* ***P*** (от «*Proceed*» — переходить, продолжать). Эта команда является удобным средством обхода команд *int*, вызывающих подпрограммы *DOS*.



Выполните написание и ввод в память программы вывода двоичного содержимого байта (содержится в регистре *BL*) на экран (алгоритм на рис. 5.3).

5.4 Отладка программы

Отладка программы включает поиск ошибок (тестирование программы) и их исправление. Пока наши программы достаточно просты, и каждая из них включает всего одну подпрограмму. Что касается отладки программ, состоящих из нескольких подпрограмм, такая отладка может рассматриваться как последовательность отладок подпрограмм [5].

Тестирование программы выполняется при различных значениях ее входных данных. Если очередной прогон программы показал наличие в ней ошибки, то производится ее поиск. Как раз для такого поиска и предназначена трассировка программы.

Если программа длинная, то ее пошаговая трассировка не очень удобна. В этом случае сначала желательно локализовать ошибку, определив содержащий ее фрагмент программы. Далее этот фрагмент исследуется более подробно. Для локализации ошибки мы выбираем несколько точек останова. Для выбора этих адресов, разбивающих программу на фрагменты, используется листинг программы, а также ее блок-схема. Далее с помощью команды «*G d*» производится анализ работы программы в каждой из точек останова. Если в очередной точке останова результаты работы программы неверны, то данная точка завершает искомый фрагмент.

Выполните отладку введенной ранее в память программы вывода двоичного содержимого байта. Тестирование программы проведите с помощью команды *G*, предварительно загружая с помощью команды *R* в регистр *BX* различные пары шестнадцатеричных цифр. При этом заметим, что *Debug* не имеет команд для работы с однобайтовыми регистрами и поэтому *BL* заполняется как часть *BX*. В случае обнаружения ошибки выполните пошаговую трассировку или используйте точки останова.

Контрольные вопросы по главе 5

1) Что такое флаг переноса, для чего он используется?
2) Как использовать циклический сдвиг с занесением во флаг переноса?
3) Как осуществить организацию циклов, какой оператор для этого используется?
4) Для чего используется регистр *CX* при организации циклов?
5) Что такое отладка программы, как она осуществляется?

Глава 6

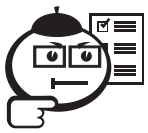
ВЫВОД НА ЭКРАН ЧИСЕЛ В ШЕСТНАДЦАТЕРИЧНОЙ ФОРМЕ

В ходе рассмотрения предыдущей главы мы научились выводить на экран двоичное содержимое регистров. Аналогично делается вывод на экран и двоичного содержимого ячеек ОП. Но гораздо удобнее для человека увидеть содержимое этих же ячеек не в двоичной, а в шестнадцатеричной системе. Напомним, что шестнадцатеричное число фактически есть сокращенная форма записи двоичного числа. При этом каждая шестнадцатеричная цифра соответствует четырем двоичным цифрам, то есть описывает содержимое четырех битов памяти.

6.1 Флаги состояния

Кроме флага переноса *CF* существуют другие флаги состояния. Рассмотрим три из них, которые описывают результат последней арифметической операции.

Допустим, что мы выполнили машинную команду вычитания *sub*. Одним из флагов состояния, устанавливаемых в зависимости от результата этой команды, является *флаг нуля* («*Zero Flag*»). Если результат команды *sub* есть 0, то флаг нуля будет установлен в 1. На распечатке регистров это значение обозначается как *ZR* (от «*Zero*» — нуль). Если результат арифметической операции не равен нулю, то флаг нуля сбрасывается в 0 — *NZ* (от «*Not Zero*» — не ноль).



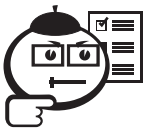
Введите в память команду «*sub ax,bx*». Протрассируйте ее с одинаковыми и с разными числами в регистрах *AX* и *BX*, наблюдая за состоянием флага нуля (*ZR* или *NZ*).

Флаг знака («*Sign Flag*») принимает значение 1 (на распечатке регистров *NG* — от «*Negative*»), если результат предыдущей арифметической операции отрицатель-

ный. Если результат неотрицательный (ноль или положительный), то флаг знака принимает значение 0 (на распечатке *PL* — «*Plus*»). ***Выполните*** трассировку команды «*sub ax,bx*» при разных содержимых *AX* и *BX*, и наблюдая за флагом знака.

Флаг переполнения устанавливается в 1 в том случае, если знаковый бит изменился в той ситуации, когда этого не должно было произойти. Например, если мы сложим два положительных числа 7000*h* и 6000*h*, то получим отрицательное число *D*000*h* (представление в дополнительном коде числа −2288). Это ошибка, т. к. результат переполняет слово. На листинге регистров переполнение обозначается как *OV* («*Overflow*» — переполнение). Если предыдущая арифметическая команда не дала переполнения, то флаг сбрасывается в 0. На распечатке регистров это значение обозначается как *NV* («*No Overflow*»). ***Проверьте*** установку флага переполнения, протрассировав команду *sub* или *add*.

Использование команды *sub* для сравнения двух чисел неудобно, т. к. эта команда производит изменение первого из чисел. Другая команда, ***cmp*** (от «*Compare*» — сравнение), производит сравнение двух чисел без их изменения. Результат сравнения используется только для установки флагов.



Загрузите в регистры *AX* и *BX* одинаковые числа, например *F*5*h*, и протрассируйте команду «*cmp ax,bx*». При этом убедитесь, что установлен флаг нуля (*ZR*), но оба регистра сохранили свое значение — *F*5*h*.

6.2 Команды условного перехода

Напомним, что флаги состояния устанавливаются для того, чтобы можно было менять ход выполнения программы в зависимости от текущей ситуации. Анализ флагов состояния и соответствующие переходы в программе выполняют команды, называемые командами условного перехода.

Команда ***jz*** (от «*Jump if Zero*» — перейти, если ноль) проверяет флаг нуля, и если он установлен (*ZR*), то выполняется переход на новый адрес. Таким образом, если мы вслед за командой *sub* напишем, например, *«jz 15a»*, то нулевой результат вычитания означает, что ЦП начнет выполнять не следующую по порядку команду, а команду, находящуюся по адресу 15*A*.

Противоположной по отношению к *jz* является команда *jnz* («*Jump if Not Zero*» — перейти, если не ноль). В следующей простой программе из числа вычитается единица до тех пор, пока в результате не получится ноль:

```
100     sub     al,01
102     jnz     100
104     int     20
```

Поместите небольшое число в *AL* и протрассируйте программу, чтобы увидеть, как работает условное ветвление. При достижении последней команды введите команду *G Debug*.

Команда условного перехода ***ja*** (от «*Jump if Above*» — перейти, если больше) осуществляет переход на указанный в команде адрес, если по результатам преды-

дущей команды флаг переноса *CF* сброшен (на листинге *CF = NC*). Флаг нуля *ZF* также должен быть сброшен. Данная команда обычно записывается сразу за командой сравнения (*cmp*) двух беззнаковых чисел. Если первое сравниваемое число больше второго, то команда *ja* осуществляет переход.

6.3 Вывод на экран одной шестнадцатеричной цифры

Любое число между 0 и *Fh* соответствует одной шестнадцатеричной цифре. Переведя выбранное число в *ASCII*-символ, его можно вывести на экран. *ASCII*-символы от 0 до 9 имеют значения от 30*h* до 39*h*; символы от *A* до *F*, однако, имеют значения от 41*h* до 46*h*. В кодовой таблице между символом 9 и символом *Ah* расположены 7 не нужных нам для представления шестнадцатеричных чисел символов (:, ; и т. д.). В результате переход в *ASCII* будет затруднен из-за наличия двух групп чисел (от 0 до 9 и от *Ah* до *Fh*), так что мы должны обрабатывать отдельно каждую группу. На рис. 6.1 приведена блок-схема программы, выполняющей вывод на экран одной шестнадцатеричной цифры.

Пример

Текст программы вывода шестнадцатеричной цифры:

```
100     mov     dl,bl
102     cmp     dl,09
105     ja      10c
107     add     dl,30
10A     jmp     10f
10C     add     dl,37
10F     mov     ah,02
111     int     21
113     int     20
```

Для передачи значения шестнадцатеричной цифры на вход программы используется регистр *BL*. Команда *cmp* вычитает два числа $\left((DL) - 9h\right)$, чтобы установить флаги, но она не изменяет регистр *DL*. Поэтому, если содержимое *DL* больше чем 9, команда «*ja 10c*» осуществляет переход к команде по адресу 10*C*.

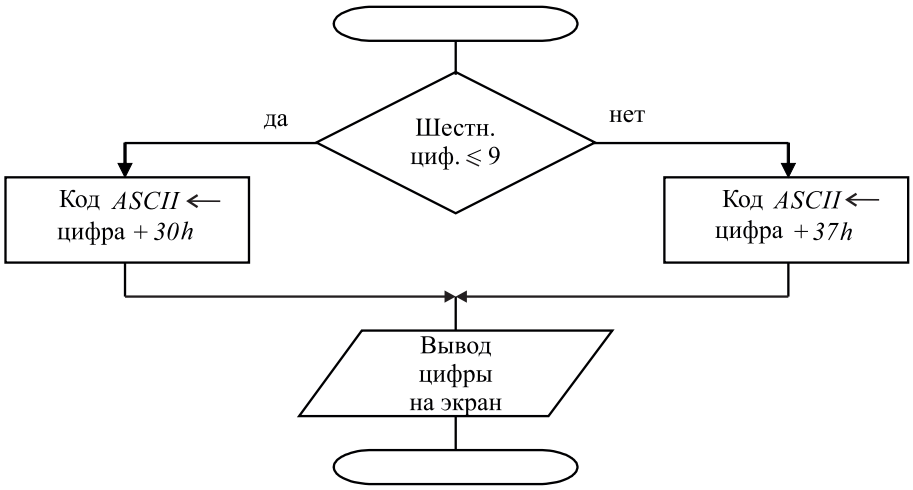
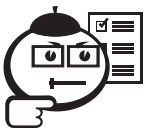


Рис. 6.1 – Алгоритм программы вывода одной шестнадцатеричной цифры



Запишите приведенную выше программу в ОП и протрассируйте ее, предварительно записав в *BL* шестнадцатеричное число, состоящее из одной цифры. Не забывайте использовать или команду *G Debug* с указанием точки останова, или команду *P*, когда запускаете машинную команду *int*. Затем проверьте правильность работы программы, используя команду *G*, предварительно загружая в *BX* граничные данные: 0; 9; *Ah* и *Fh*.

Общее правило: при тестировании любой программы рекомендуется проверить граничные данные.

6.4 Вывод старшей цифры двузначного шестнадцатеричного числа

Одна шестнадцатеричная цифра занимает четыре бита (четыре бита часто называют полубайтом или тетрадой). Двузначное шестнадцатеричное число занимает восемь бит (один байт). Это может быть, например, регистр *BL*. При выводе числа на экран сначала выводится старшая цифра, а затем — младшая. Соответствующий укрупненный алгоритм приведен на рис. 6.2. При этом детальный алгоритм каждого из двух этапов «Вывод цифры на экран» приведен ранее на рис. 6.1. Рассмотрим этап «Выделение старшей цифры». В результате него содержимое старшего полубайта должно быть переписано (сдвинуто) в младший полубайт.

Несмотря на то, что нам надо выполнить сдвиг вправо, вспомним команду *rcl*, которая циклически сдвигает байт или слово влево через флаг переноса. Ранее мы использовали команду «*rcl bl,1*», в которой единица есть сообщение для ЦП о том, что надо сдвинуть содержимое *BL* на один бит. Мы можем осуществить циклический сдвиг более чем на один бит, но мы не можем написать команду «*rcl bl,2*». Для циклического сдвига необходимо поместить счетчик сдвигов в регистр

CL, который используется здесь так же, как регистр *CX* применялся командой *loop* при определении числа повторений цикла.

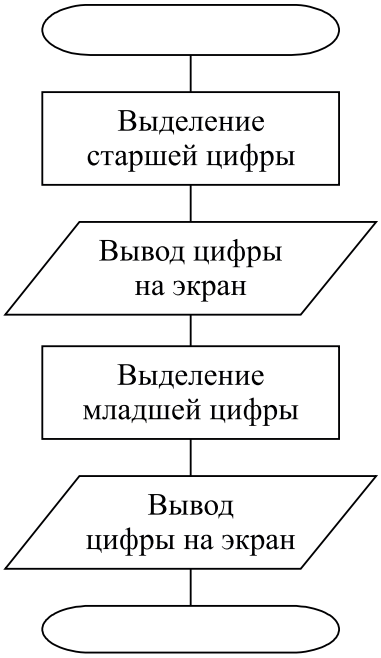
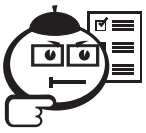


Рис. 6.2 – Алгоритм вывода двузначного шестнадцатеричного числа

Так как не имеет смысла осуществлять циклический сдвиг более чем 16 раз, то для записи числа сдвигов вполне подойдет восьмибитовый регистр *CL*.

Для сдвига старшего полубайта вправо на четыре бита будем использовать команду сдвига ***shr*** («*Shift Right*» — логический сдвиг вправо). Данная команда не только выполняет сдвиг вправо, но и записывает в освобождающиеся старшие биты нули. В этом проявляется разница между терминами «логический» и «циклический», так как команда циклического сдвига записывает в освобождающиеся биты содержимое флага переноса. Что касается выталкиваемых младших битов байта (или слова), то они по очереди записываются во флаг переноса аналогично циклическому сдвигу.



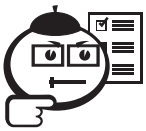
Загрузите числа 4 в *CL* и 5*Dh* в *DL*, а затем введите и протрассируйте следующую команду сдвига:

100 shr dl,cl

DL должен теперь содержать число 05*h*. То есть этот регистр содержит в своем младшем полубайте старшую цифру числа 5*Dh*.

Реализацию этапа «Выделение старшей цифры» осуществляют команды:

mov dl,bl
mov cl,04
shr dl,cl



Поместите эти команды в ОП, дополнив их командами этапа «Вывод цифры на экран». При этом не забудьте скорректировать адреса переходов. (Можно записать программу со старыми адресами, а затем скорректировать команды переходов.) Выполните программу, предварительно загрузив в регистр *BL* любую пару шестнадцатеричных цифр.

6.5 Вывод младшей цифры двузначного шестнадцатеричного числа

Для вывода младшей цифры достаточно обнулить старший полубайт в восьмибитовом регистре, содержащем пару шестнадцатеричных цифр.

Обнуление любых битов в байте или в слове удобно выполнить, используя команду ***and*** (логическое «И»). Данная команда побитно сравнивает два заданных в ней байта (слова). Если в обоих байтах соответствующий бит имеет значение «1», то и в результирующий байт на место соответствующего бита записывается 1. Если хотя бы один из сравниваемых битов имеет значение «0», то и результирующий бит принимает нулевое значение. Результирующий байт записывается на место первого из сравниваемых байтов. Например, команда «*and bl,cl*» последовательно выполняет операцию *and* сначала над битами 0 регистров *BL* и *CL*, затем над битами 1, битами 2 и т. д. и помещает результат в *BL*.

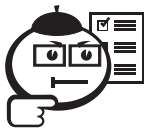
Пример

Выполняя операцию *and* над 0*Fh* и каким-либо байтом, мы можем обнулить старший полубайт этого байта:

$$\begin{array}{rr} & 1011\,0101 \\ AND & 0000\,1111 \\ \hline & 0000\,0101 \end{array}$$

Следующие две команды реализуют этап «Выделение младшей цифры»:

```
mov dl,bl
and dl,0f
```



Запишите в память программу для вывода младшей цифры. Протестируйте эту программу, загружая в *BL* различные пары шестнадцатеричных цифр. Далее запишите в память всю программу вывода на экран двузначного шестнадцатеричного числа и протестируйте ее. (Не забудьте при этом скорректировать адреса переходов во второй части программы, а также исключить первую команду «*int 20*».)

Контрольные вопросы по главе 6

1) Что такое флаги состояния, какие значения они могут принимать, как используются?
2) Какие команды условных переходов Вы знаете?
3) Как осуществить вывод на экран одной шестнадцатеричной цифры?
4) Как вывести на экран старшую цифру двузначного шестнадцатеричного числа?
5) Как вывести на экран младшую цифру двузначного шестнадцатеричного числа?

Глава 7

СПИСКИ И ПРОЦЕДУРЫ

7.1 Несвязанные списки

Важное место при разработке программ занимают операции с информационными структурами. К таким структурам относятся *списки*. Они очень широко используются для работы с информацией, находящейся как в ОП, так и в ВП.

Каждый список состоит в общем случае из множества более мелких информационных структур, называемых *записями*. Записи, входящие в список, содержат «родственную» информацию. Примером списка является список, содержащий сведения о служащих организации. Одна запись такого списка содержит сведения об одном сотруднике организации.

В свою очередь, запись состоит из одного или нескольких *полей* (рис. 7.1). Содержанием поля может быть, например, число или набор символов. Для приведенного выше примера поля записи могут содержать фамилию, имя, отчество, год рождения и т. д. Далее будем считать, что все записи, входящие в один и тот же список, имеют одинаковую (фиксированную) длину. Кроме того, для удобства будем считать, что все записи списка имеют одинаковую *структуру*, т. е. одинаковый состав и порядок полей.

Поле 1	Поле 2	.....	Поле *M*

Рис. 7.1 – Структура записи

В списке записи расположены в определенном ***логическом порядке***. В зависимости от этого порядка все списки делятся на линейные и нелинейные. ***Линейным списком*** называется список, логическое расположение записей в котором наглядно описывается прямой линией (рис. 7.1). Пример списка, который не является линейным, приведен на рис. 7.2.

Логическое расположение записей в списке в общем случае отличается от их *физического размещения* в памяти (оперативной или внешней). Это значит, что две

записи, являющиеся в списке соседними, могут занимать участки памяти, расположенные далеко друг от друга.

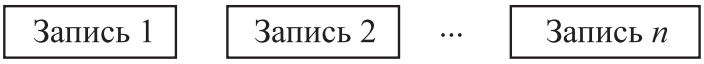


Рис. 7.2 – Линейный список

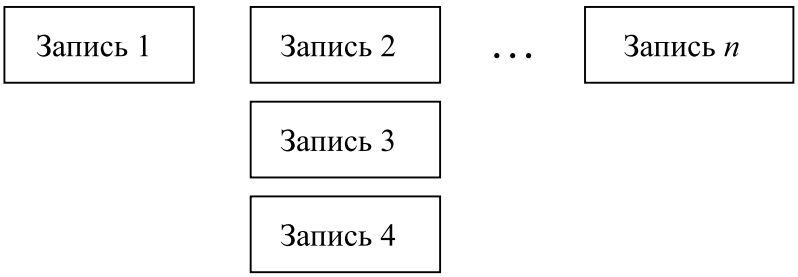


Рис. 7.3 – Пример нелинейного списка

Логический и физический порядки размещения записей совпадают для линейных списков, называемых *несвязанными*. Для несвязанного списка характерно то, что, зная адрес памяти, которую занимает данная запись, мы без труда можем найти в памяти логически соседние к ней записи. Допустим пока, что мы имеем дело с несвязанными линейными списками. В зависимости от состава допустимых операций над списком будем различать следующие типы линейных списков:

1) линейный список общего вида;
2) стек;
3) очередь.

Линейный список общего вида допускает наибольшее число операций:

1) получение доступа к k-й записи списка, чтобы проанализировать или изменить содержимое ее полей;
2) включение новой записи непосредственно перед записью k;
3) исключение записи k из списка;
4) объединение двух или более линейных списков в один;
5) определение числа записей в списке;
6) поиск записи в списке с заданным значением некоторого поля записи;
7) сортировка записей списка в порядке возрастания или убывания значения некоторого поля записи.

Для удобства выполнения перечисленных операций над линейным списком общего вида вводятся две вспомогательные переменные (*переменная* — небольшая область в ОП или в ВП, или это — регистр):

- ***S*** — содержит адрес в памяти, где расположена первая запись списка;
- ***F*** — содержит адрес в памяти последней записи списка.

Переменные S и F часто называют указателями соответственно на начало и конец списка. Пользуясь этими переменными, нетрудно входить в список, как с начала, так и с конца (рис. 7.4).

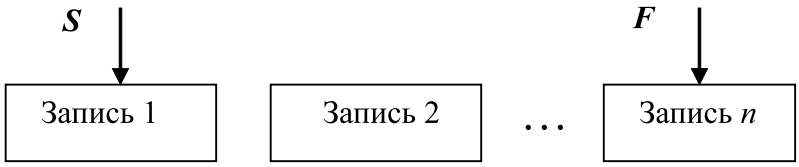


Рис. 7.4 – S и F — указатели на начало и конец линейного списка

Стек — линейный список, над которым допустимы только две операции:

1) включение новой записи в начало списка;
2) исключение записи, стоящей первой от начала списка.

При работе со стеком выполняется правило «последним пришел, первым ушел». Это аналогично стопке подносов в столовой: поднос, который был положен на стопку последним, первым будет с нее снят. Так как включения и исключения из стека производятся только в его начале, то нужна только одна переменная S. На рис. 7.5 и 7.6 приведены примеры включения и исключения записи.

Исходный стек

Результат

Рис. 7.5 – Включение записи k в стек

Очередь — линейный список, над которым допустимы только две операции:

1) включение новой записи в конец очереди;
2) исключение записи, стоящей в начале очереди.

На рис. 7.7 и 7.8 приведены примеры включения и исключения записи.

Исходный стек

Результат

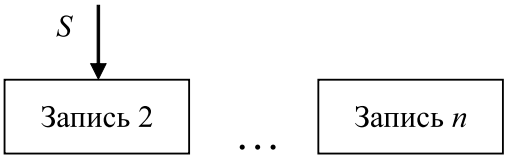
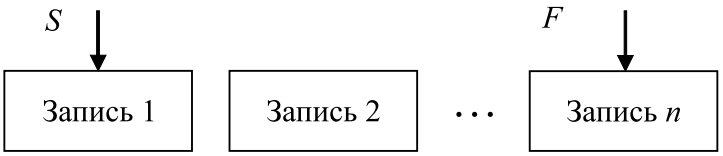


Рис. 7.6 – Исключение записи из стека

Исходная очередь:



Результат:

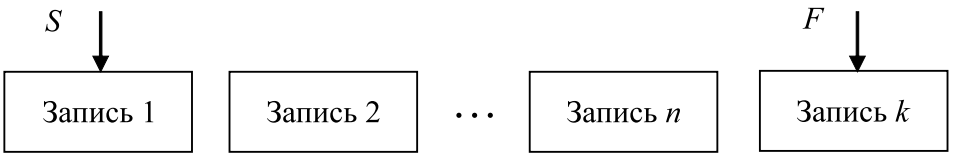
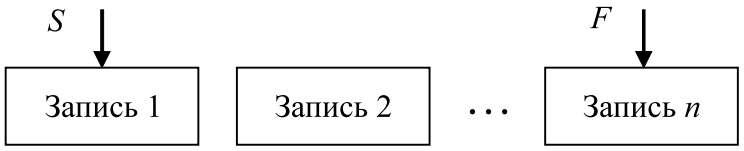


Рис. 7.7 – Включение записи k в несвязанную очередь

Исходная очередь:



Результат:

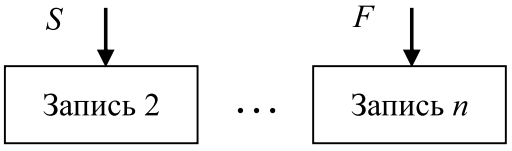


Рис. 7.8 – Исключение записи из несвязанной очереди

7.2 Связанные списки

Снимем теперь допущение, что записи, расположенные по соседству в линейном списке, занимают и соседние места в памяти. Подобное несоответствие между логическим и физическим расположением записей допускается в *связанных списках*. Связанные линейные списки бывают одно и двухсвязанные. В *односвязанном списке* каждая запись имеет одно специальное поле, содержащее указатель на соседнюю запись в списке (рис. 7.9). *Указатель* — это адрес соседней записи, пользуясь которым можно найти эту соседнюю запись в памяти. Переменной F может и не быть, т. к. двигаться справа налево мы все равно не сможем. Единственное применение данной переменной — добавление новой записи в конец списка. Указатель ⊥ есть пустой указатель. Часто он кодируется помещением в поле указателя нуля.

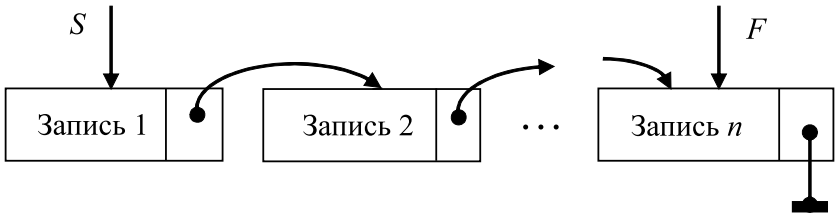


Рис. 7.9 – Пример линейного односвязанного списка

В *двухсвязанном списке* каждая запись имеет два поля указателей на обе соседние записи (рис. 7.10).

Для связанного линейного списка общего вида определены те же операции, что и для несвязанного. Часто применяются связанные стеки и очереди. Исходя из определений этих списков связанный стек может быть только односвязным, а очередь как одно- так и двухсвязанной. Операции над этими списками аналогичны операциям над соответствующими несвязанными списками.

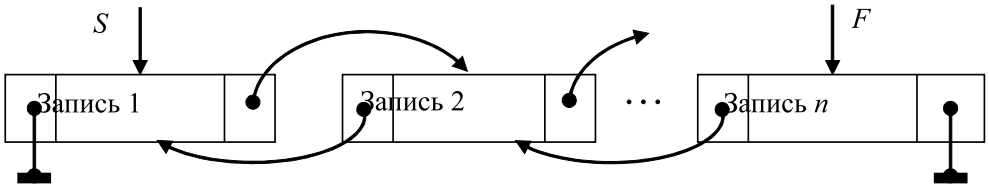


Рис. 7.10 – Двухсвязанный линейный список

В отличие от несвязанных связанные списки могут быть нелинейными. Важнейшим видом нелинейных списков является *дерево* (рис. 7.11). Запись 1 называется *корнем дерева*. Записи, у которых нет *сыновей*, называются *листьями*. Это — 1.1, 1.2.1, 1.2.2.1, 1.2.2.2, 1.2.3. Совокупность записей, начиная от корня и кончая каким-то листом, называется *путем*. Пример пути: «1, 1.2, 1.2.1».

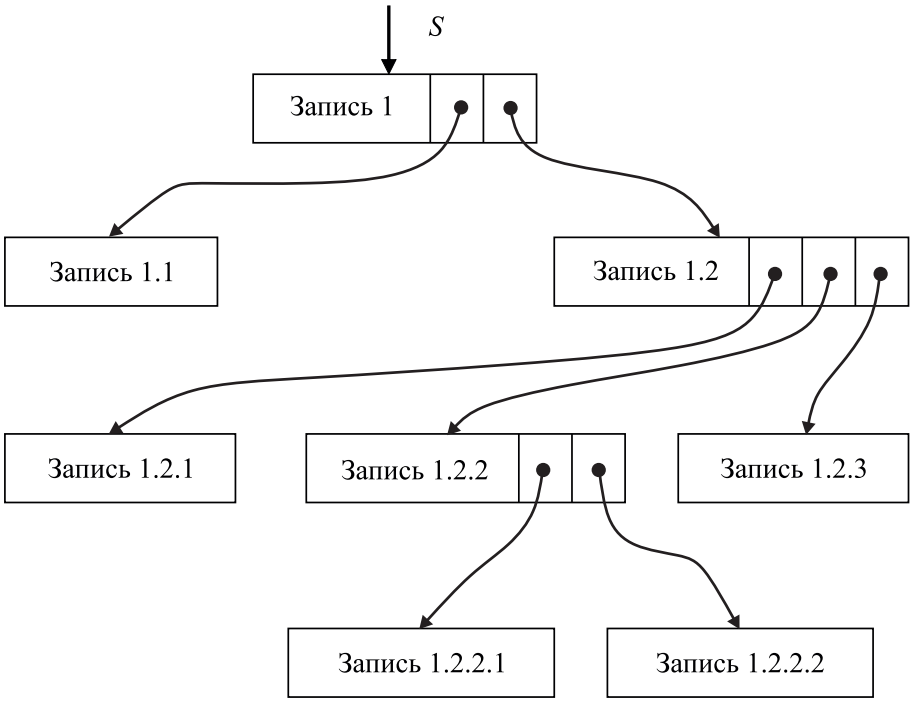


Рис. 7.11 – Пример дерева

7.3 Программные стеки

В принципе программа может обрабатывать любые списки, но несвязанный стек — единственный вид списков, для работы с которым ЦП имеет специальные регистры и машинные команды.

Допустим, что для нашей программы операционная система (ОС) выделила область (сегмент) ОП, начиная с параграфа 20*h* (рис. 7.11). Регистр *CS* содержит число 20*h* и представляет собой указатель на начало этой области. Одновременно с назначением области ОП для размещения программы ОС назначает сегмент для размещения стека, который будет обслуживать нашу программу и который поэтому называется *программным стеком*.

Где находится стек в ОП? Для того чтобы выполнять операции со стеком, достаточно знать адрес в памяти вершины стека. Этот адрес всегда содержится в паре регистров:

1) регистр сегмента стека — ***SS***;
2) указатель стека — ***SP*** (от «*Stack Pointer*» — указатель стека).

Реальный адрес ячейки, являющейся вершиной стека, получается аппаратно путем суммирования содержимого *SS*, умноженного на 16, с содержимым *SP*.

Содержимое регистров *SS* и *SP* вы многократно наблюдали ранее, получая листинг регистров по команде *R* или какой-то другой команде *Debug*. Первоначальную запись в данные регистры выполняет *DOS*, руководствуясь следующим. Во-первых, в *SS* записывается тот же номер параграфа, что и в регистр сегмента кодов *CS*. Это означает, что для стека отводится тот же сегмент ОП, что и для программы (см. рис. 7.12).

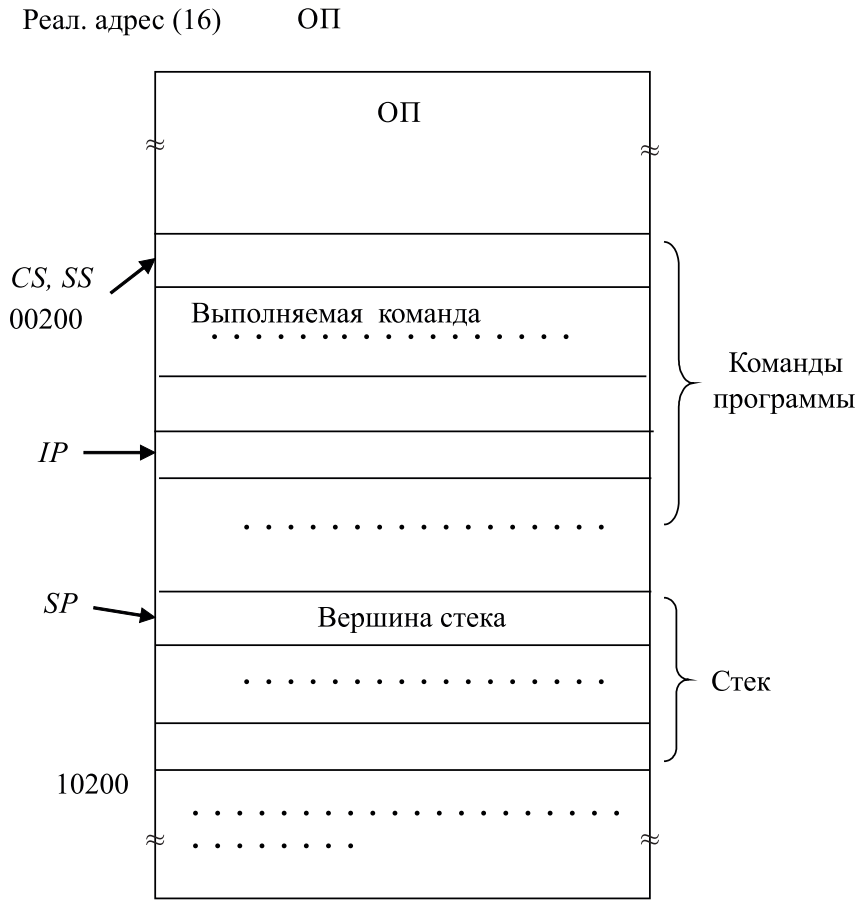


Рис. 7.12 – Пример размещения программы в ОП

Первоначально в указатель стека *DOS* записывает максимально возможное число без знака (*FFFF*) или чуть меньшее число, в результате чего вершиной стека является последняя ячейка сегмента стека (и сегмента программы), называемая «дном» стека. «Растет» стек в отличие от программы в сторону не больших, а меньших адресов. При добавлении слова данных в стек содержимое *SP* уменьшается на 2, а при исключении слова из стека — увеличивается на 2.

Программа может выполнить включение слова данных в стек, используя машинную команду «***push*** *b*», где *b* — адрес ячейки памяти (ОП или регистровой памяти), содержимое которой следует включить в стек. Исключение слова данных из стека выполняет команда «***pop*** *b*», где *b* — адрес ячейки памяти, в которую следует считать слово данных из стека.

Проверьте с помощью команды *R Debug* содержимое регистров *SS* и *SP*. Выполните команды «*push ax*» и «*pop ax*», наблюдая за содержимым регистра *SP*.

В заключение заметим, что многоцелевое использование одного и того же стека делает необходимым повышенное внимание при работе с ним: каждое записанное в стек слово должно быть вовремя извлечено оттуда.

7.4 Процедуры

За исключением очень небольших программ, разработка программы предполагает ее представление в виде совокупности относительно независимых частей (модулей), называемых *подпрограммами*. Применение подпрограмм предоставляет следующие преимущества:

1) существенное уменьшение трудоемкости программирования за счет возможности выполнять разработку программы не целиком, а по частям;
2) существенное сокращение памяти для размещения программы, так как выделив в подпрограмму многократно повторяющийся участок кода программы, достаточно выделить память лишь для одной подпрограммы, инициируя ее из различных мест программы;
3) уменьшение трудоемкости программирования за счет того, что одну и ту же подпрограмму можно использовать не в одной, а в нескольких программах.

Машинные подпрограммы бывают двух типов: процедуры и обработчики прерываний. Любой обработчик прерываний можно инициировать (запустить) из своей программы, поместив в нее машинную команду «*int n*», где *n* — номер прерывания.

Процедура — список машинных команд, который можно вызывать из различных мест программы. Переход к процедуре называется *вызовом*, а соответствующий переход назад называется *возвратом*. Вызов процедуры выполняет машинная команда «***call*** *b*», где *b* — адрес, по которому находится первая команда процедуры. Возврат осуществляет команда ***ret*** (рис. 7.13). Возврат после каждого вызова осуществляется к команде, которая находится в памяти сразу за командой *call*.

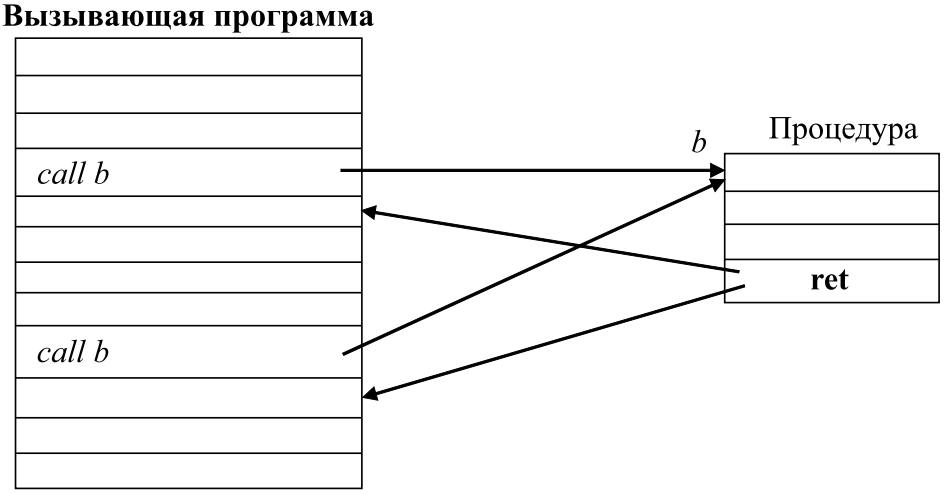


Рис. 7.13 – Вызов процедуры и возврат из нее

Подобно команде *jmp*, команды *call* и *ret* выполняют или близкие переходы, когда вызывающая программа и процедура находятся в одном сегменте кода, или дальние переходы — программа и процедура находятся в разных сегментах. В первом случае адрес перехода *b* представляет собой число — смещение первой команды процедуры относительно начала сегмента (новое значение регистра *IP*). Во

втором случае это пара чисел: (*CS*, *IP*). Так как пока наши программы небольшие, то далее речь будет идти только о близком вызове процедур.

Допустим, что для вызова процедуры мы будем использовать команду «*call 200h*», где 200*h* — адрес, по которому находится первая команда процедуры. Первая команда нашей программы находится по адресу *l*00*h*. Располагая процедуру по адресу 200*h*, мы стремимся убрать ее подальше от основной программы. Если мы запишем список команд процедуры (тело процедуры) по другому адресу, то этот адрес следует записать вместо адреса 200*h*.

Пример

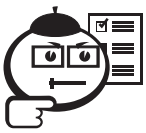
Следующая программа вызывает десять раз процедуру, которая каждый раз печатает один символ, начиная от *A* и заканчивая *J*:

```
100     mov     dl,41
102     mov     cx,000a
105     call    0200
108     loop    0105
10A     int     20
```

Текст процедуры:

```
200     mov     ah,02
202     int     21
204     inc     dl
206     ret
```

В тексте процедуры содержится новая команда ***inc***, которая увеличивает содержимое своего операнда (в примере — регистр *DL*) на единицу. Команда *ret* возвращает управление в то место основной программы, откуда процедура была вызвана. Следующей командой, выполняемой после команды *ret*, будет та команда основной программы, которая следует за командой *call*. В примере это команда *loop*.



Введите основную программу и процедуру в память. С помощью команды *G* выполните программу, а затем протрассируйте ее с целью детального изучения работы. При этом особое внимание уделите изменению регистра *IP*.

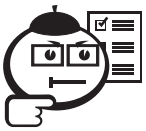
При вызове процедуры и возврате из нее необходимо выполнить следующие три требования:

1) при вызове процедуры необходимо запомнить адрес следующей команды, чтобы можно было впоследствии осуществить возврат в нужное место вызывающей программы;
2) используемые процедурой регистры необходимо запомнить до изменения их содержимого, а перед самым выходом из процедуры восстановить;

3) процедура должна иметь возможность выполнять обмен данными с вызывающей ее программой.

Первое требование реализуют команды *call* и *ret*. При выполнении команды вызова процедуры *call* программа «перескакивает» в другую область ОП (например, по адресу 200*h*), где находится первая команда тела вызываемой процедуры. Так как при выполнении команды процедуры *ret* мы возвращаемся в основную программу, то необходимо где-то хранить адрес возврата (адрес команды, следующей за командой *call*). В качестве места хранения адреса возврата используется программный стек. При близком вызове команда *call* помещает адрес следующей за ней команды (находится в *IP*) в программный стек, а *ret* извлекает этот адрес из стека и помещает его в указатель команды *IP*. При дальнем вызове *call* помещает в стек, а *ret* извлекает оттуда не одно, а два слова — содержимое регистров *CS* и *IP*.

Заметим, что вызов процедуры не требует от нас использования команд *push* и *pop*, т. к. работа со стеком в данном случае производится «автоматически» — команда *call* помещает адрес возврата в стек, а команда *ret* извлекает его оттуда.



Введите в память (если они там не сохранились) предыдущие программу и процедуру. Протестировав программу, наблюдайте за содержимым указателя стека *SP* до и после исполнения команд *call* и *ret*. Сразу же после любого выполнения команды *call* найдите в стеке адрес возврата (108), используя команду *U Debug*.

Свойство стека «пришел первым, ушел последним» идеально подходит для реализации вложенных вызовов процедур (рис. 7.14), когда одна вызываемая процедура вызывает другую процедуру, которая, в свою очередь, вызывает третью, и т. д. При этом размещение в программном стеке адреса возврата вызывающей процедуры производится раньше, а его выборка позже, чем соответствующие операции с адресом возврата вызываемой процедуры, что соответствует правилу «последним пришел, первым ушел».

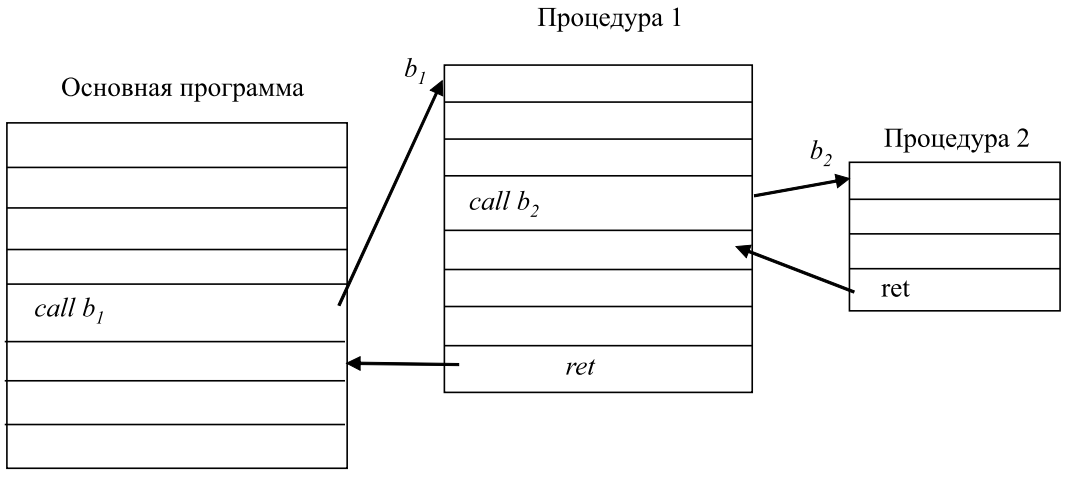
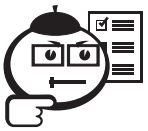


Рис. 7.14 – Вложенный вызов процедур

Допустим, что программа состоит из главной программы и трех процедур. Главная программа имеет вид:

```
100     call    200
103     int     20
```

По адресу 200 находится процедура, которая печатает букву *A* и вызывает вторую процедуру, записанную по адресу 300, которая выводит *B*. Вторая процедура вызывает третью процедуру, первая команда которой находится по адресу 400. Эта третья процедура выполняет только вывод на экран буквы *C*.



Введите данную программу в память и протрассируйте ее, наблюдая за изменением регистра *SP* и содержанием стека. Обязательно найдите в стеке адреса возврата, когда вызваны все три процедуры.

Требование восстановления содержимого регистров, используемых в вызывающей программе, также выполняется с помощью стека. В начале процедуры ее команды *push* помещают в стек содержимое запоминаемых регистров, а в конце ее команды *pop* извлекают из стека запомненные слова и помещают их в регистры. Порядок регистров при восстановлении обратный их порядку при запоминании. Пример процедуры, в которой производится такое восстановление содержимого регистров:

Пример

```
200     push    cx
201     push    dx
202     mov     dl,0a
205     call    300
208     inc     dl
20C     pop     dx
20D     pop     cx
20E     ret
```

Обратите внимание, что команды *pop* расположены по отношению к командам *push* в обратном порядке, так как *pop* удаляет слово, помещенное в стек последним, а старое значение *CX* находится в стеке «глубже» старого значения *DX*.

Сохранение и восстановление регистров *CX* и *DX* позволяет произвольно изменять их значения внутри процедуры (которая начинается с адреса 200*h*), в то же время значения регистров, используемых процедурами, вызывающими эту, сохранены. Следовательно, мы можем использовать эти регистры для хранения *локальных переменных* — переменных, которые применяются внутри процедур, не влияя на значения переменных, используемых вызывающей программой.

В дальнейшем обязательно выполняйте правило: все регистры, содержимое которых меняется внутри процедуры, за исключением регистров, используемых

для передачи выходных данных процедуры в вызвавшую ее программу, в конце процедуры должны восстанавливать свои прежние значения. При несоблюдении этого правила отладка ваших программ затянется на долгие недели, а возможно, и месяцы.

Рассмотрим теперь обмен данными между процедурой и вызывающей ее программой. Данные, передаваемые процедуре при ее вызове, называются *входными параметрами процедуры*. А данные, которые передаются процедурой в вызывающую ее программу, называются *выходными параметрами*. Информационный обмен между программой и процедурой обеспечивается путем использования областей памяти одного или нескольких следующих видов:

1) регистры данных;
2) программный стек;
3) другие области ОП.

Если передаваемых данных немного, то достаточно регистров данных, большее количество данных может быть передано с помощью стека. Если же этих данных слишком много, то они помещаются в область ОП, а начальный адрес этой области записывается или в регистр данных, или в стек.

Любой язык программирования позволяет записывать *виртуальные программные процедуры*. В зависимости от языка программирования они называются процедурами, функциями или подпрограммами. В любом случае текст любого из перечисленных модулей преобразуется соответствующим транслятором в код машинной программной процедуры.

Контрольные вопросы по главе 7

1) Для чего используются списки?
2) В чем отличие связанных списков от несвязанных списков, линейных списков от нелинейных списков?
3) Что такое стек, как определить реальный адрес ячейки, которая является вершиной стека?
4) Что такое процедура, какой оператор используется для вызова процедуры на исполнение, для возврата из процедуры в программу?
5) В чем особенность использования вложенных процедур?

Глава 8

ПРОГРАММНЫЕ ПРЕРЫВАНИЯ

Прерыванием называется навязанное принудительно центральному процессору прекращение выполнения текущей программы и переход им на выполнение подпрограммы, которая называется *обработчиком прерываний*.

Для обеспечения прерываний в ОП выделяется специальная область — *область векторов прерываний*. В IBM PC эта область занимает первые 1024 байта ОП, и их никогда нельзя использовать для других целей (рис. 8.1). Каждый *вектор прерываний* занимает два слова (4 байта) ОП и соответствует своему типу прерывания. Содержимым вектора является адрес в ОП первой ячейки обработчика прерываний.

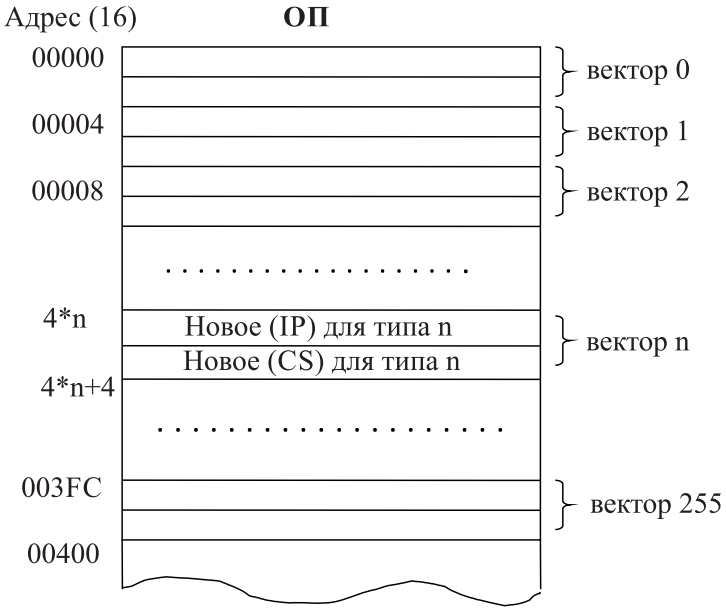


Рис. 8.1 – Область векторов прерываний

В зависимости от причины прерывания разделяются на программные и аппаратные. Причиной *внешнего аппаратного прерывания* является сигнал по шине

управления (эта шина входит в состав ОШ), переданный в ЦП от ПУ, которое требует к себе внимания со стороны программ ЦП. Причиной *внутреннего аппаратного прерывания* является сигнал от одной из аппаратных схем самого ЦП.

Причиной *программного прерывания* является попадание на ЦП машинной инструкции **INT** *n*, где *n* — номер прерывания. При выполнении этой инструкции в стек помещаются:

1) содержимое *FLAGS*;
2) содержимое *CS*;
3) содержимое *IP*.

Последние два слова представляют собой адрес возврата в прерванную программу. Далее в регистры *IP* и *CS* загружаются значения из соответствующего вектора прерываний.

Таким образом, следующей после INT исполняемой на ЦП инструкцией будет первая инструкция обработчика прерываний. В конце программы обработчика прерываний стоит инструкция возврата из прерывания **RTI**, которая выталкивает из стека прежние содержимые *IP* и *CS*, а также *FLAGS*.

Программные прерывания являются единственным способом обращения из прикладной программы к системным программам с целью получения помощи. Примером такого прерывания является 21-е прерывание. При этом в регистр *AH* программа должна записать *номер функции*, уточняющий тип услуги, запрашиваемой у операционной системы.

Глава 9

ВВОД С КЛАВИАТУРЫ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

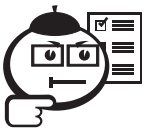
В предыдущей главе были рассмотрены теоретические вопросы, связанные с использованием в программе стека и процедур. Теперь перейдем к практическому рассмотрению этих вопросов в процессе решения важной задачи ввода с клавиатуры шестнадцатеричных чисел.

9.1 Ввод одной шестнадцатеричной цифры

Для того, чтобы ввести с клавиатуры в программу *ASCII*-символ, можно воспользоваться командой программного прерывания «*int 21h*» с функцией номер 01. Вызванная в результате данной команды подпрограмма *DOS* помещает *ASCII*-символ, соответствущий нажатой клавише, в регистр *AL*.

Введите команду *«int 21h»* по адресу 100*h*, поместите номер функции 01 в регистр *AH*, а затем запустите команду с помощью «*G 102*» либо *P*. В результате *DOS* переходит в состояние ожидания нажатия вами клавиши (на экране вы видите мерцающий курсор). Нажмите любую клавишу, соответствующую шестнадцатеричной цифре (0–*F*). Убедитесь, что в результате регистр *AL* содержит соответствующий код *ASCII*.

При преобразовании *ASCII*-символа, который содержится в регистре *AL*, в шестнадцатеричную цифру решается задача, обратная той, которую мы решали при выводе цифры на экран. На рис. 9.1 приведена блок-схема программы, которая выполняет ввод цифры с клавиатуры в регистр *AL*.



Запишите текст программы ввода шестнадцатеричной цифры и поместите его в память. Для программирования условия можно использовать не только уже знакомую нам команду условного перехода *ja* (перейти, если больше), но и обратную ей команду ***jbe*** (перейти, если меньше или равно). Обе команды используются после сравнения без знаковых величин, каковыми коды *ASCII* и являются.

Так как результат данной программы содержится в регистре *AL*, то этот регистр необходимо проанализировать прежде, чем исполнится команда «*int 20*» (*Debug* восстанавливает регистры после этой команды). Поэтому для запуска программы используйте команду Debug «*G d*», где *d* — смещение команды «*int 20*».

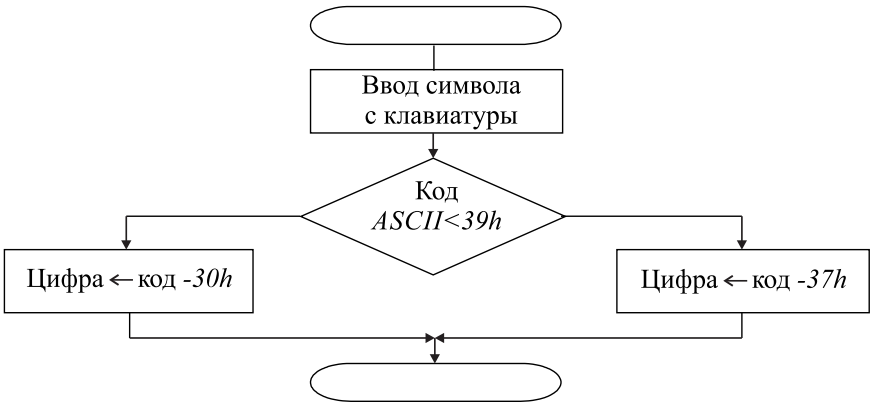
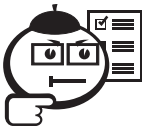


Рис. 9.1 – Алгоритм ввода шестнадцатеричной цифры



Выполните программу несколько раз, нажимая не только клавиши, соответствующие шестнадцатеричным цифрам, но и другие клавиши. При этом убедитесь, что программа реагирует на неправильное нажатие клавиши точно так же, как и на правильное. В дальнейшем этот недостаток будет устранен.

9.2 Ввод двузначного шестнадцатеричного числа

Такой ввод можно осуществить следующим образом. Введем старшую цифру числа, записав ее в четыре младших бита регистра *DL*. Далее умножим регистр *DL* на 16, в результате чего цифра переместится в старшие четыре бита *DL*. После этого введем с клавиатуры младшую цифру числа и, просуммировав *AL* с *DL*, получим в *DL* все двузначное шестнадцатеричное число. Соответствующий алгоритм приведен на рис. 9.2.

Запишите программу ввода с клавиатуры в регистр *DL* двузначного шестнадцатеричного числа. (Для сдвига регистра *DL* влево используйте рассмотренную

ранее команду *shl*.) Введите данную программу в память и протрассируйте ее при различных парах чисел, вводимых с клавиатуры.

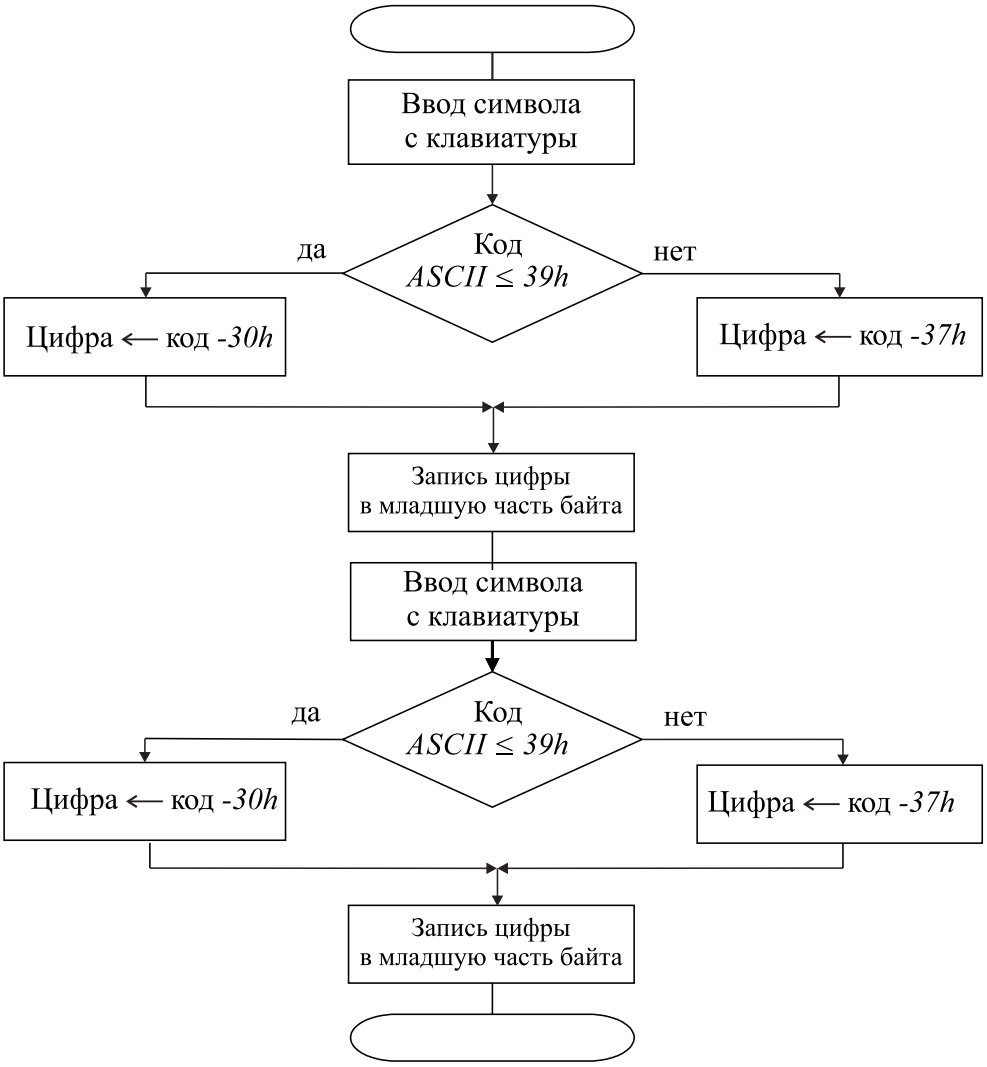


Рис. 9.2 – Алгоритм ввода двузначного шестнадцатеричного числа

Необходимо убедиться, что программа правильно работает при граничных условиях. Это пары чисел: 00; 09; 0*A*; 0*F*; 90; *A*0; *F*0. Используйте точку останова для запуска программы без выполнения команды *«int 20h»*. (Для ввода шестнадцатеричных чисел используйте только заглавные буквы.)

9.3 Более совершенный ввод шестнадцатеричных цифр

Записанные ранее программы ввода одно- и двузначного шестнадцатеричных чисел реагировали на нажатие «нецифровых» клавиш точно так же, как и «цифровых». Теперь мы получим программу, которая не будет реагировать на нажатие «нецифровых» клавиш.

Для этого в состав программы введем процедуру ввода шестнадцатеричной цифры. Эта процедура возвратит управление в главную программу только тогда, когда она получит с клавиатуры правильную шестнадцатеричную цифру. При нажатии «нецифровой» клавиши ее код на экран не выводится и управление в программу не возвращается.

Ранее для ввода символа с клавиатуры использовалась функция 01 программного прерывания 21*h*. Эта функция не только вводит символ с клавиатуры, но и выводит его на экран. Подобный вывод символа вовремя его ввода называется «эхом» символа. Теперь мы будем использовать функцию 08 21-го прерывания, которая не выводит «эхо» символа.

На рис. 9.3 приведена блок-схема процедуры ввода шестнадцатеричной цифры, а на рис. 9.4 — сама эта процедура. (Процедура возвращает цифру в регистре *BL*.) ***Изучите*** внимательно алгоритм и найдите реализацию его этапов в программе. Во-первых, обратите внимание, что управляющей структурой верхнего уровня является цепочка из цикла ПОКА и этапа «Вывод цифры на экран». В качестве условия повторения цикла выступает равенство единице флага ошибки *CF*.

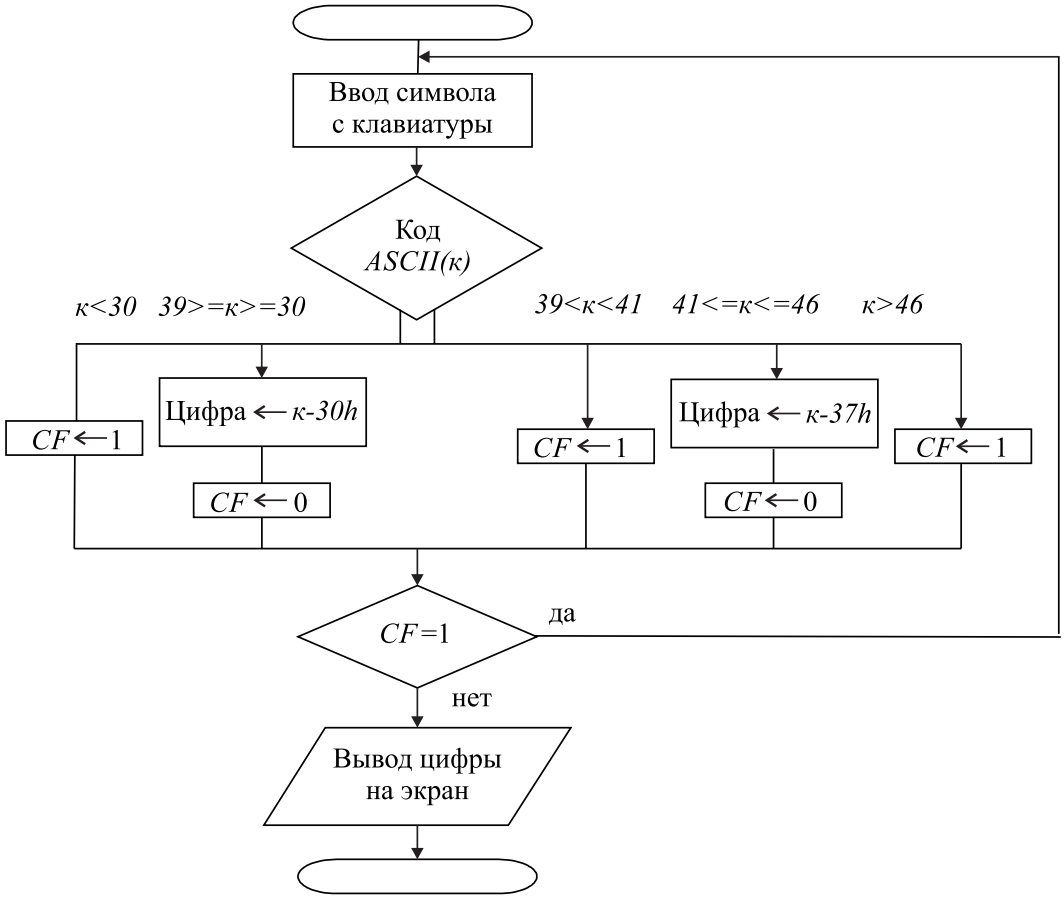


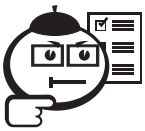
Рис. 9.3 – Алгоритм ввода одной шестнадцатеричной цифры

Имя *CF* флага ошибки обусловлено тем, что он реализуется в программе с помощью одноименного флага переноса. При этом флаг переноса используется совсем не для того, для чего он был создан. Данный прием широко распространен, и мы также будем его использовать.

Существуют специальные команды для работы с флагом *CF*. Команда ***stc*** устанавливает флаг (*CF* = 1), а ***clc*** сбрасывает его (*CF* = 0). Команда условного перехода ***jc*** выполняет переход при *CF* = 1, а команда ***jnc*** — при *CF* = 0.

```
200     push  ax
201     push  dx
202     mov   ah,8
204     int   21
206     mov   dl,al
208     cmp   al,30
20A     jb    222
20C     cmp   al,39
20E     ja    215
210     sub   al,30
212     clc
213     jmp   223
215     cmp   al,41
217     jb    222
219     cmp   al,46
21B     ja    222
21D     sub   al,37
21F     clc
220     jmp   223
222     stc
223     jc    204
225     mov   bl,al
227     mov   ah,2
229     int   21
22B     pop   dx
22C     pop   ax
22D     ret
```

Рис. 9.4 – Процедура ввода шестнадцатеричной цифры



. . . . . .

Введите данную процедуру в память. Кроме того, запишите в память следующую программу для проверки процедуры:

```
100     call 200
103     int 20
```

. . . . . .

Протрассируйте программу, используя команду *P Debug* для перехода через команды *int*. Курсор появится в левой части экрана и будет ждать ввода символа. Напечатайте символ «*K*», который не является правильным символом. Ничего не произойдет. Теперь напечатайте один из заглавных шестнадцатеричных символов. Вы должны увидеть шестнадцатеричную цифру в регистре *BL*, а также на экране. Испытайте эту процедуру на граничных условиях: «\» (символ, стоящий перед нулем), «0», «9», «:» (символ, стоящий после 9) и т. д.

Теперь, когда у нас есть процедура ввода одной шестнадцатеричной цифры, программа, считывающая двузначное шестнадцатеричное число в регистр *DL* и обрабатывающая ошибки, стала достаточно простой. Ее алгоритм приведен на рис. 9.5.

Рис. 9.5 – Алгоритм ввода двузначного шестнадцатеричного числа

Контрольные вопросы по главе 9

1) Какая функция для прерывания INT21 используется для ввода с клавиатуры?
2) Для чего используется проверка правильности введенных символов?
3) Сколько раз будет использована процедура ввода шестнадцатеричной цифры при вводе значения в регистр?
4) Как используется флаг ошибки в алгоритме ввода одной цифры?
5) Как ввести четырехзначное шестнадцатеричное число в регистр?

РАЗДЕЛ II

Ассемблерные программы в среде DOS

Глава 10

СИСТЕМНЫЕ ПРОГРАММЫ

До сих пор для написания программ использовалась программа *Debug*. Такое использование сложно и трудоемко. Причиной этого является то, что *Debug* предназначен не для получения текстов программ, а для отладки программ. Теперь мы перейдем от процесса написания программ на машинном языке, которым мы фактически занимались, к написанию программ на языке ассемблера. Для этого нам потребуется помощь некоторых новых системных программ. Прежде чем рассматривать эти программы, полезно рассмотреть общие свойства и различия системных программ.

10.1 Функции системных программ

Несмотря на то, что наличие аппаратуры и прикладных программ является минимально-достаточным условием для решения задач по переработке информации, ЭВМ, состоящие только из этих двух подсистем, на практике не используются. Обязательной подсистемой любой ЭВМ являются также системные программы. Благодаря им и пользователи ЭВМ, и их прикладные программы имеют дело не с реальной («голой») аппаратурой, а взаимодействуют с *виртуальной* (кажущейся) *ЭВМ*.

На рис. 10.1 показано наиболее укрупненное представление ЭВМ, в том числе наиболее важные системные интерфейсы.

Интерфейс пользователя ЭВМ с аппаратурой представляет собой клавиши мыши и клавиатуры, поверхность экрана, а также различные кнопки на системном блоке и на периферийных устройствах.

Интерфейс между пользователем и системными программами представляет собой *язык управления операционной системой*, называемый также *командным языком*, или *языком директив*. Одна команда этого языка задаёт выполнение одной обрабатывающей программы (прикладной программы, утилиты или лингвистиче-

ского процессора). Интерфейс пользователя и прикладной программы представляет собой язык диалога этой программы.

Интерфейс между аппаратурой и программами представляет собой язык машинных команд. *Системный программный интерфейс* — множество машинных команд, называемых *системными вызовами*, посредством которых прикладные программы обращаются за помощью к системным программам. При выполнении прикладной программы в среде *DOS* в качестве системных вызовов используются исключительно команды *int*.

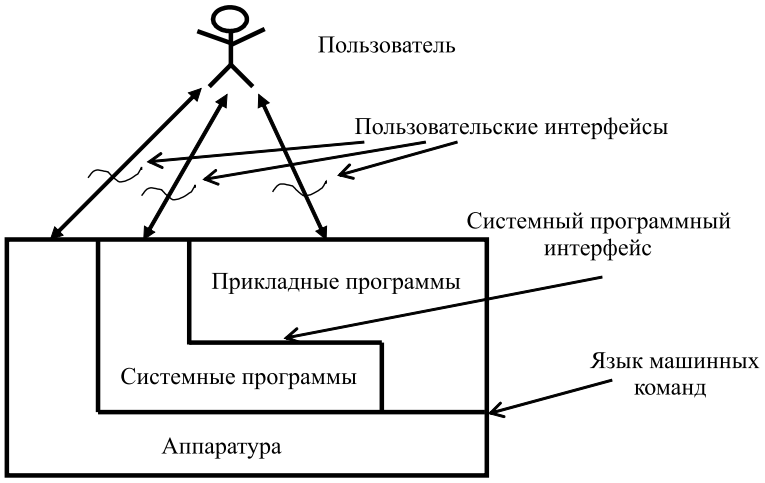


Рис. 10.1 – Укрупненное представление ЭВМ

Совокупность интерфейсов, используемых пользователем или прикладной программой, представляет собой соответствующую виртуальную машину. Например, прикладная программа «выполняется» на виртуальной ЭВМ, предоставляемой языком машинных команд, а также совокупностью системных вызовов. При этом каждый системный вызов может рассматриваться как команда виртуальной ЭВМ, которой в действительности соответствуют многие десятки или сотни машинных команд, образующие вызываемую системную подпрограмму [6].

На рис. 10.2 приведена классификация системных программ. *Обрабатывающие системные программы* отличаются от управляющих программ как по своим функциям, так и по способу их инициирования (запуска). Основные функции обрабатывающих программ:

1) перенос информации. Перенос может выполняться между различными устройствами или в пределах одного устройства. При этом под устройствами понимаются: ОП, устройства ВП, устройства ввода-вывода;

2) преобразование информации. То есть после считывания информации с устройства обрабатывающая программа преобразует эту информацию, а уж затем записывает ее на это же или на другое устройство.

В зависимости от того, какая из этих двух функций является основной, обрабатывающие системные программы делятся на утилиты и лингвистические процессоры. Основной функцией *утилиты* является перенос информации, а основная функция *лингвистического процессора* — перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его пред-

ставления, ориентированная на программиста, преобразуется в форму, ориентированную на ЦП. Лингвистические процессоры делятся на трансляторы и интерпретаторы.

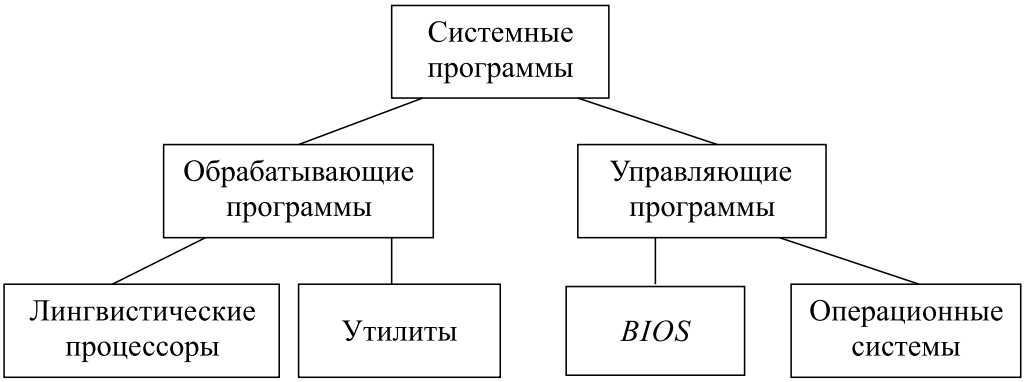


Рис. 10.2 – Классификация системных программ

В результате работы *транслятора* алгоритм, записанный на языке программирования (*исходная виртуальная программа*), преобразуется в алгоритм, записанный на машинном языке. (На самом деле, как будет показано позже, машинная программа является результатом совместной работы нескольких лингвистических процессоров.)

Трансляторы делятся на компиляторы и ассемблеры. Исходная программа для *транслятора-ассемблера* записывается на *языке-ассемблере*. Один оператор данного языка транслируется в одну машинную команду. Исходная программа для *компилятора* записывается на ***языке*** *программирования высокого уровня*. Каждый оператор такого языка транслируется в несколько машинных команд. Примерами языков высокого уровня являются Паскаль и Си.

Интерпретатор в отличие от транслятора не выдаёт машинную программу целиком. Выполнив перевод очередного оператора исходной программы в соответствующую совокупность машинных команд, интерпретатор обеспечивает их выполнение. Затем преобразуется тот исходный оператор, который должен выполняться следующим по логике алгоритма, и т. д. Примером интерпретатора является *интерпретатор командного языка* (*ИК*). Другие названия этого модуля: *командный процессор*, *командная оболочка*. ИК представляет собой лишь «видимую», сравнительно небольшую часть операционной системы. На самом деле, чтобы интерпретировать, т. е. выполнить (преобразовав в совокупность машинных команд), очередную команду пользователя, ИК инициирует многие другие модули ОС. Одни из них ищут текст требуемой прикладной программы на диске, другие выделяют необходимые аппаратные ресурсы, третьи загружают программу в ОП и инициируют её. Попутно заметим, что так как одна команда пользователя задаёт выполнение целой программы (прикладной, утилиты или системы программирования), то язык управления ОС может рассматриваться как язык программирования очень высокого уровня.

Утилиты — обрабатывающие системные программы, которые выполняют:

1) перенос данных с одного периферийного устройства на другое или перенос данных в пределах одного устройства. Примеры: программа копирования

данных на магнитном диске; программа ввода данных с клавиатуры терминала на диск; программа распечатки информации на диске;

2) программы изменения расположения данных. Это различные программы сортировки;

3) программы для изменения представления данных. Сюда относятся редакторы, которые осуществляют редактирование виртуальных программ и других текстов, а также программы перекодировки данных для согласования программ, использующих разные кодировки, или для обеспечения секретности.

Примером сложной утилиты является ***DOS Navigator***. Эта утилита переносит с диска на экран информацию, содержащуюся в любом каталоге любого логического диска. По запросу пользователя она выводит на экран файловую структуру любого логического диска. Кроме того, она предоставляет пользователю удобный язык управления операционной системой *DOS* за счет того, что она переносит имя исполняемого файла программы из позиции экрана, отмеченной пользователем с помощью курсора-маркера, в то место памяти, откуда это имя может взять интерпретатор команд ОС. *Курсор-маркер* — светящийся прямоугольник, генерируемый не аппаратурой экрана (как настоящий курсор), а получаемый программно.

В отличие от лингвистических процессоров утилиты используются не только программистами, но и *пользователями-непрограммистами*. Эта наиболее многочисленная категория пользователей ЭВМ работает на виртуальных машинах, предоставляемых готовыми прикладными программами, а также утилитами. При этом заметим, что запуск любой обрабатывающей системной программы (утилиты или лингвистического процессора) аналогичен запуску прикладной программы. Более того, с точки зрения других системных программ, а также аппаратуры, системные обрабатывающие программы ничем не отличаются от обычных прикладных программ.

Основные функции *управляющих программ*:

1) оказание помощи прикладным и системным обрабатывающим программам в использовании ими ресурсов ЭВМ. При этом различают информационные, программные и аппаратные ресурсы. Данная функция реализуется во всех системах;

2) обеспечение *однопользовательской мультипрограммности* — одновременное выполнение нескольких прикладных и (или) системных обрабатывающих программ в интересах одного пользователя. Эта функция реализуется лишь в мультипрограммных системах. В однопользовательских однопрограммных системах эта функция отсутствует;

3) обеспечение *многопользовательской мультипрограммности* — одновременное выполнение нескольких обрабатывающих программ (прикладных и системных) в интересах нескольких пользователей. Данная функция реализуется лишь в многопользовательских системах.

Управляющие системные программы делятся на две группы: программы *BIOS* и программы операционной системы. *BIOS* — базовая система ввода-вывода. Сюда относятся системные программы, находящиеся в ПЗУ (постоянное запоминающее

устройство). Эти программы выполняют многие функции обмена с периферийными устройствами, участвуя, таким образом, в выполнении первой из перечисленных выше функций управляющих программ.

Операционная система (ОС) — множество управляющих программ, предназначенных для выполнения всех трех перечисленных выше функций. Примером однопользовательской однопрограммной ОС является *DOS*. Примерами однопользовательских мультипрограммных ОС являются различные *WINDOWS*. Операционная система *UNIX* является примером многопользовательской системы.

При рассмотрении любого командного языка, а этим мы займемся чуть позже, невозможно обойтись без понятия файла. В отличие от многих других объектов, управляемых ОС, файлы «видимы» для пользователя и используются им при формировании своих команд для ОС.

10.2 Файлы

Вся информация, обрабатываемая ВС, содержится в ней на устройствах: ОП, ВП, устройства ввода-вывода. При этом на любом из устройств информация хранится в виде длинной битовой строки, т. е. в виде последовательности нулей и единиц. Длина одной такой битовой строки может составлять многие сотни Гбит ($1Г = (1024)^3$). Так как работать с такой длинной строкой чрезвычайно неудобно, то она разделяется на поименованные части разной длины, называемые файлами. Более точное определение: *файл* — часть пространства носителя ВП (разрывная или непрерывная), которой присвоено имя, уникальное для данной ЭВМ.

Информация на носителе делится на части-файлы по смысловому принципу. Например, один файл может содержать текст исходной программы, второй — ее объектный модуль, а третий — загрузочный модуль. Кроме того, в большинстве современных ОС каждое устройство ввода-вывода также считается файлом. Что касается ОП, то информация на ней делится не на файлы, а на сегменты.

Применительно к файлу существуют три пользовательских имени. Наиболее короткое из них — *простое имя файла*. Это имя дает файлу пользователь при его создании. Ограничения на выбор простого имени файла определяются типом используемой ОС. В достаточно старых операционных системах используется короткое простое имя файла. Например, в *DOS* (за исключением новых версий) длина простого имени не может превышать 12 символов. При этом до восьми символов имеет *собственно имя файла*, до трех символов — *расширение имени файла*, один символ — разделительная точка. Применение расширений позволяет давать «родственным» файлам «родственные» имена, что весьма удобно для пользователя. Обычно расширение указывает на тип файла и учитывается многими системными программами. Например, если текст ассемблерной программы записывается текстовым редактором в файл *Abc.asm*, то объектный модуль помещается транслятором в файл *Abc.obj*, а загрузочный модуль записывается редактором связей в файл *Abc.com* или *Abc.exe*. В новых версиях *DOS* и в различных *WINDOWS* длина простого имени файла может достигать 255 символов.

Что касается символов, которые можно использовать для записи имен файлов, то к их составу старые версии *DOS* также предъявляют гораздо более жесткие требования, в соответствии с которыми в состав символов могут входить строчные

и прописные латинские буквы, цифры, а также некоторые служебные символы: -; _; $; #; &; @; !; %; ~; ^; (;); {; }. Придерживаясь далее этих требований, мы обеспечим выполнимость своих программ в среде любой *DOS*. Попутно заметим, что если в состав имени входят строчные (малые) буквы, то *DOS* (как и *WINDOWS*) всегда воспринимает их как прописные (большие).

Файл является достаточно крупной единицей информации. Для удобства работы с ним битовая строка, образующая файл, делится на части, называемые *записями файла*. При этом файл может рассматриваться как несвязанный линейный список, расположенный на конкретном носителе информации. Заметим, что разные программы могут использовать разное разбиение одного и того же файла на записи.

В реальной ВС одновременно существуют сотни или даже тысячи файлов. Для того, чтобы ориентироваться в этом «море», и ОС, и ее пользователь прибегают к помощи *файловой структуры системы*. На верхнем уровне этой структуры находятся логические диски. *Логический диск* — магнитный или оптический диск целиком или его часть, имеющий имя, уникальное для данной ЭВМ. Имя логического диска — буква с двоеточием, например *a:* или *c:*. Каждый логический диск имеет отдельную *файловую структуру логического диска* в виде дерева, пример которой приведен на рис. 10.3.

Для построения древовидной структуры применяются специальные файлы, называемые каталогами (в *WINDOWS* каталоги называются *папками*). *Каталог* — служебный файл, содержащий сведения о других файлах (в том числе, возможно, и о других каталогах). Файл-каталог состоит из 32-байтовых записей, каждая из которых содержит информацию об одном файле. Поля записи содержат сведения о файле: имя, размер, начальный адрес, атрибуты, дату и время последней модификации. При этом поле начального адреса в такой записи играет роль указателя на соответствующий файл (каталог). Две записи каталога всегда используются особо. В одной из них содержится адрес размещения на диске самого каталога, а во второй — адрес «родительского» каталога.

Так как в каталоге могут быть «зарегистрированы» не только файлы данных, но и каталоги, то появляется возможность связать все файлы системы в единую *иерархическую (древовидную) структуру*. При этом корнем дерева является *корневой каталог*. На следующем уровне дерева находятся те файлы и каталоги, сведения о которых содержатся в корневом каталоге. Аналогично, каталоги первого уровня дерева «порождают» файлы и каталоги второго уровня и т. д. Корневой каталог расположен в фиксированной области диска и состоит из 32-байтовых записей, структура которых аналогична структуре записей обычного каталога. Корневой каталог имеет имя «\» (обратный слеш). Простое имя любого другого каталога отличается от имени обычного файла данных тем, что оно не может иметь расширения имени.

Большим достоинством любой древовидной файловой структуры является то, что она позволяет пользователю не заботиться об уникальности простых имен файлов. Это объясняется тем, что ОС работает не с этими пользовательскими именами файлов, а с путями. Единственное ограничение: все файлы, связанные с одним каталогом, должны иметь разные простые имена. *Имя-путь файла*, называемое также *абсолютным именем файла*, представляет собой последовательность всех

имен, начиная с корневого каталога и кончая простым именем файла. При этом имя каждого промежуточного каталога в имени-пути завершается символом «\». Например, на рис. 9.2 три файла имеют одинаковое простое имя *a1*, но абсолютные имена у них разные: *\user1\a1*, *\user2\a1*, *\user3\a1*.

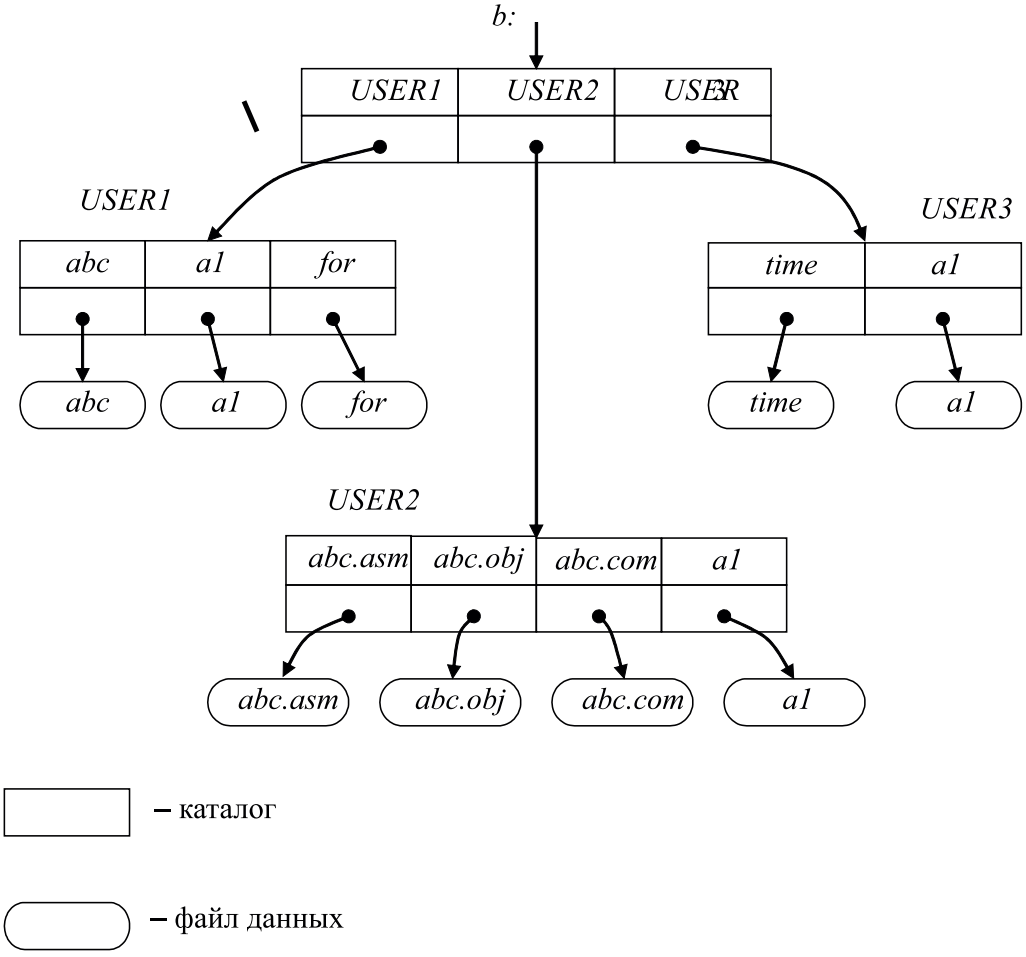


Рис. 10.3 – Пример файловой структуры логического диска

Так как ЭВМ имеет в общем случае несколько логических дисков, то имя каждого такого диска может рассматриваться в качестве переменной *S* (см. п. 7.2), позволяющей ОС однозначно выбрать среди нескольких древовидных файловых структур требуемую. Один из логических дисков ОС считает *текущим логическим диском*. Смена текущего диска выполняется командой пользователя для ОС (в *DOS* для этого достаточно набрать имя требуемого логического диска.) Если пользователь хочет задать имя файла, расположенного на логическом диске, отличном от текущего, то он добавляет имя логического диска к имени-пути файла. Пример: *b:\USER2\abc.asm*.

Следует отметить, что ОС «помнит» не только текущий логический диск, но и *текущий каталог* на этом диске. Поэтому если искомый файл «зарегистрирован» в текущем каталоге, то его можно задать для ОС не с помощью имени-пути, а используя его простое имя. ОС сама получит имя-путь файла, соединив имя-путь

каталога с простым именем файла. С помощью команды ОС пользователь может сменить текущий каталог.

Если адресуемый файл является «потомком» текущего каталога, то в качестве имени этого файла можно использовать «смещение» относительно текущего каталога. Такое пользовательское имя файла называется *относительным именем*. Например, если текущим каталогом является \, то записанное выше имя-путь файла *\user1\a1* может быть заменено чуть более коротким именем *user1\a1*. Обратите внимание на отсутствие в начале этого имени символа «\». Его наличие всегда говорит о том, что записано полное имя-путь.

Принято использовать для обозначения текущего каталога символ «.», а для обозначения родительского каталога (по отношению к текущему каталогу) — «..». Например, если текущим каталогом является *\user1*, то для задания файла *\user3\time* пользователь может использовать относительное имя *.. \user3\time*. Так как это относительное имя оказалось даже длиннее имени-пути файла на целых два символа, то никакого практического смысла в его использовании нет. Подобный смысл появляется лишь при использовании многоуровневых файловых структур.

Контрольные вопросы по главе 10

1) Для чего используются системные программы?
2) Назначение различных системных программ, на какие классы они подразделяются?
3) Что такое файлы?
4) Приведите пример файловой структуры логического диска.
5) Что такое имя файла, расширение?

Глава 11

ПРОСТЫЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ

11.1 Общая структура простых ассемблерных программ

Основным отличием программы на языке ассемблера от соответствующей машинной программы, которую мы вводим в ЭВМ с помощью *Debug*, является наличие псевдооператоров. В отличие от исполнительного оператора, который преобразуется транслятором-ассемблером в одну машинную команду, псевдооператор ни в какие машинные команды не транслируется. Такой оператор представляет собой указание транслятору и нигде, кроме самого транслятора, не используется.

Общая структура простых ассемблерных программ, которая будет для нас достаточна на ближайшее время, имеет вид:

[org 100h]

.....

int 20h

Посмотрим внимательно на данную структуру. По горизонтали она разделена на две части. Левая (пока пустая) часть предназначена для записи идентификаторов (меток) программных объектов, к которым относятся сегменты, процедуры, исполнительные операторы и элементы данных. В правой части листинга находятся псевдооператоры и исполнительные операторы, а также операнды этих операторов. Идентификатор отделяется от соответствующего оператора минимум одним пробелом.

Простейшая программа состоит всего из одного сегмента кодов, в котором находятся команды программы. При этом псевдооператор *«[org 100h]»* сообщает транслятору о том, что самый первый исполнительный оператор нашей программы должен быть помещен в выделенный программе сегмент ОП со смещением 100*h* относительно начала сегмента. Мы и раньше использовали это смещение, вводя

машинные программы с помощью *Debug*. Следует обратить внимание на символ *h* после шестнадцатеричного числа 100. Использование этого символа после шестнадцатеричных чисел обязательно. Так как транслятор-ассемблер в отличие от *Debug* «обычной» считает не шестнадцатеричную, а десятичную систему счисления.

В конце программы будем помещать хорошо знакомый нам исполнительный оператор «*int 20h*», выполняющий возврат из программы туда, откуда она была запущена.

Глава 12

ОСНОВНЫЕ ОПЕРАТОРЫ АССЕМБЛЕРА

12.1 Типы операторов

Любой алгоритмический язык программирования, в том числе и ассемблер, имеет операторы следующих типов [3].

Исполнительные операторы. Данные операторы преобразуются транслятором в машинные инструкции. Один исполнительный оператор ассемблера преобразуется в одну машинную инструкцию. А один исполнительный оператор языка высокого уровня транслируется в несколько машинных инструкций. Все исполнительные операторы языка программирования делятся на операторы обработки данных и операторы передачи управления. Операторы обработки данных влияют на содержимое ячеек памяти (ячейки ОП, регистры, флаги), а операторы передачи управления изменяют ход выполнения программы.

Псевдооператоры определения данных. В отличие от исполнительного оператора псевдооператор ни в какие машинные инструкции не транслируется, а представляет собой указание транслятору со стороны программиста. Псевдооператор определения данных требует от транслятора выделить область памяти заданной длины. Кроме того, он может попросить транслятор поместить в выделенную область какие-то первоначальные данные. Впоследствии на этапе выполнения сама программа может менять содержимое этой области.

Другие псевдооператоры. Они информируют транслятор о структуре программы, помогая транслятору и редактору связей правильно преобразовать исполнительные операторы в машинные инструкции.

Макрооператоры. Каждый такой оператор заменяется транслятором на несколько обычных операторов языка программирования (в том числе, возможно, и псевдооператоров).

Комментарии. Это любые сообщения в исходной программе, предворяемые специальным символом. В рассматриваемом языке ассемблера это символ «;».

Комментарии игнорируются транслятором и никак не влияют на текст машинной программы.

12.2 Операторы обработки данных

Операторы обработки данных делятся:

1) на арифметические операторы;
2) логические операторы;
3) операторы передачи данных;
4) операторы манипуляций флажками;
5) операторы сдвигов;
6) цепочечные (строковые) операторы.

Далее рассматриваются эти типы операторов, а также адресация данных и псевдооператоры определения данных.

12.2.1 Арифметические операторы

Арифметические инструкции процессора i8086 делятся на *двоичные* и *двоично-кодированные десятичные инструкции*. Второй из этих классов используется редко, т. к. применяемый в нем способ кодирования данных неэффективен по затратам памяти.

В свою очередь, двоичные арифметические инструкции разделяются на *знаковые* и *беззнаковые*. Первые из них выполняют операции как над положительными так и над отрицательными двоичными числами, в то время как беззнаковые инструкции имеют дело только с положительными числами. Рассмотрим основные типы операторов ассемблера, выполняющие операции над двоичными числами.

Операторы сложения:

1. **ADD** (*сложить*) суммирует два операнда (слова или байты). Результат записывается на место первого операнда. Примеры:

```
ADD     AX, Mem     ; (AX) + (Mem) → AX , Mem — слово в ОП
ADD     Mem, AX     ; (Mem) + (AX) → Mem
ADD     AL, 40      ; (AL) + 40 → AL
ADD     Mem, 0Fh    ; (Mem) + 0Fh → Mem
```

Запрещается суммировать содержимое двух ячеек ОП, а также записывать в качестве первого операнда непосредственное значение.

2. **ADC** (*сложить с переносом*) суммирует два операнда (слова или байты), а также флаг переноса CF. Результат помещается на место первого операнда.

Совместное применение инструкций ADD и ADC позволяет выполнить суммирование двух чисел даже тогда, когда результат не вмещается в 16 битов. Например, следующие два оператора складывают 32-битовое число, находящееся в регистрах CX и DX, с 32-битовым числом, находящимся в регистрах AX и BX. Результат записывается в регистры AX и BX:

```
ADD     AX, CX      ; Суммируются младшие 16 битов
ADC     BX, DX      ; Суммируются старшие 16 битов, а также
                    ; перенос от предыдущего суммирования
```

3. **INC** (*инкремент*) увеличивает операнд на 1. Пример:

```
INC     AX      ; (AX) + 1 → AX
```

Операторы вычитания:

1. **SUB (*вычесть*)** выполняет вычитание второго операнда из первого операнда (операнды — байты или слова). Результат помещается в качестве первого операнда. Примеры:

```
SUB     AX, CX      ; (AX)–(CX) → AX
SUB     AX, Mem     ; (AX)–(Mem) → AX
SUB     Mem, AX     ; (Mem)–(AX) → Mem
SUB     AL, 10      ; (AL)–10 → AL
```

Запрещается брать в качестве обоих операндов ячейки ОП, а также задавать в качестве первого операнда непосредственное значение.

2. **DEC** (*декремент*) уменьшает операнд на 1. Пример:

```
DEC     AX      ; (AX)–1 → AX
```

3. **NEG (*изменить знак*)** вычитает из нулевого значения значение операнда. Результат записывается на место операнда. Пример:

```
NEG     AX      ; –(AX) → AX
```

4. **CMP** (*сравнить два операнда*) выполняет вычитание 1-го и 2-го операндов, но в отличие от оператора SUB результат вычитания никуда не записывается, а лишь используется для установки флажков. Примеры:

```
CMP     AX, BX      ; (AX)–(BX)
CMP     Mem, AH     ; (Mem)–(AH)
CMP     AL, 10      ; (AL)–10
```

Операторы умножения:

1. **MUL** (*умножить*) выполняет умножение двух беззнаковых чисел (слов или байтов). Единственный операнд содержит один из сомножителей и представляет собой регистр общего назначения или ячейку памяти размером в байт или слово. В качестве второго сомножителя используется содержимое регистра AL (в операциях над байтами) или регистра AX (в операциях над словами).

16-битовое произведение байтов помещается в регистры AH (старший байт) и AL (младший байт). 32-битовое произведение слов помещается в регистры DX (старшее слово) и AX (младшее слово). Примеры:

```
MUL     BX          ; Умножить BX на AX без знака
MUL     Mem_wor     ; Умножить содержимое ячейки на AX без знака
MUL     DL          ; Умножить DL на AL без знака
```

2. **IMUL** (*умножить целые числа*) выполняет умножение двух знаковых чисел (слов или байтов). Правило размещения сомножителей и результата аналогично оператору MUL.

Операторы деления:

1. **DIV** (*разделить*) выполняет деление чисел без знака. Единственный операнд представляет собой регистр общего назначения или ячейку памяти (байт или слово) и содержит делитель. Делимое должно иметь двойной размер; оно извлекается из регистров AH и AL (при делении на байт) или из регистров DX и AX (при делении на слово).

Результат возвращается следующим образом. Если делитель представляет собой байт, то частное возвращается в регистре AL, а остаток в регистре AH. Ес-

ли делитель представляет собой слово, то частное возвращается в регистре AX, а остаток в регистре DX. Примеры:

```
DIV     BX              ; Разделить DX:AX на BX, без знака
DIV     Mem_byte        ; Разделить AH:AL на байт ОП, без знака
```

2. **IDIV** (разделить целые числа) выполняет деление чисел со знаком. Правило записи делимого, делителя и результата аналогично DIV.

12.2.2 Логические операторы

Они используются для установки и сброса битов в слове или в байте. В качестве операндов могут выступать два регистра, регистр с ячейкой ОП или непосредственное значение с регистром или ячейкой ОП.

AND (*логическое* И) сравнивает два операнда (слова или байты) побитно. Результат записывается на место первого операнда. Если оба из сравниваемых битов равны 1, то результат равен 1, во всех остальных случаях результат равен 0. Например, пусть (AL) = 10101111, а (BH) = 11100000, тогда AND AL, BH запишет в AL 10100000. Примеры:

```
AND     AX, BX ; Операция AND над двумя регистрами
AND     AX, Mem_wor ; Операция AND над регистром и словом ОП
AND     Mem_byte, AL ; Операция AND над байтом ОП и регистром
AND     BL, 11110000b ; Обнуление младших 4-х битов регистра BL
AND     Mem_byte, 00001111b ; Обнуление старших 4-х битов байта ОП
```

OR (*логическое* ИЛИ) сравнивает два операнда (слова или байты) побитно. Если хотя бы один из сравниваемых битов равен 1, то результат равен 1, если оба сравниваемых бита равны 0, то результат равен 0. Например, пусть (AL) = = 10101111, а (BH) = 11100000, тогда OR AL, BX запишет в AL 11101111. Примеры:

```
OR      BL, 11110000b ; Установка в 1 старших 4-х и сохранение
                      ; значений 4-х младших битов регистра BL
OR      Mem_byte, 00001111b ; Сохранение значений 4-х старших
                            ; и установка в 1 4-х младших битов байта ОП
```

XOR (*исключающее* ИЛИ) сравнивает два операнда (слова или байты) побитно. Если один из сравниваемых битов равен 0, а другой 1, то результат есть 1, если оба сравниваемых бита одинаковы (оба — 0 или оба — 1), то результат есть 0.

TEST (*проверить*) выполняет логическое побитовое умножение своих двух операндов (байтов или слов), но в отличие от AND результат никуда не записывается, а лишь используется для установки флажков. Оба операнда остаются без изменения. Например, оператор:

TEST BL, 11110000b установит флаг нуля в 1 только тогда, когда ни один из четырех старших битов в BL не равен 1.

NOT (НЕ) инвертирует все биты в операнде: вместо 0 пишет 1, а вместо 1 — 0.

12.2.3 Операторы передачи данных

Они осуществляют обмен данными между регистрами и ячейками памяти. **MOV** (*переслать*) пересылает содержимое второго операнда (слово или байт) в качестве содержимого первого операнда. Можно пересылать байт или слово между регистром и ячейкой памяти или между двумя регистрами. А также можно помещать непосредственное значение в регистр или в ячейку памяти. Примеры:

```
MOV     AX, Table     ; Пересылка из ячейки ОП в регистр
MOV     Table, AX     ; и наоборот
MOV     BL, AL        ; Пересылка между регистрами
MOV     CL, -30       ; Загрузка константы в регистр
MOV     Mem, 25h      ; или в ячейку ОП
```

Следует отметить, что с помощью данного оператора нельзя выполнять пересылку данных из одной ячейки ОП в другую ячейку ОП. Такая пересылка может быть осуществлена двумя операторами MOV с использованием регистра общего назначения.

С помощью оператора MOV можно загружать в регистр общего назначения (но не в регистр сегмента) номер начального параграфа сегмента, задав в качестве второго операнда имя сегмента. Например:

```
MOV     AX, Data_seg      ; Data_seg — имя сегмента данных
```

Для загрузки в регистр сегмента (кроме CS) номера начального параграфа сегмента требуются два оператора MOV. Например:

```
MOV     AX, Data_seg      ; Загрузка в AX
MOV     DS, AX            ; Пересылка в регистр DS
```

Что касается регистра сегмента кода CS, то он не может быть использован в качестве первого операнда в операторе MOV. Кроме того, нельзя пересылать содержимое одного регистра сегмента в другой. Такая пересылка может быть осуществлена двумя операторами MOV с использованием регистра общего назначения.

XCHG (*обменять*) производит обмен содержимого двух регистров или регистра и ячейки ОП. Следующие два фрагмента программ делают одно и то же — меняют местами содержимое двух байтов в ОП — Opr1 и Opr2:

```
а) MOV     AL, Opr1     ; (Opr1) → AL
   MOV     BL, Opr2     ; (Opr2) → BL
   MOV     Opr2, AL     ; (AL) → Opr2
   MOV     Opr1, BL     ; (BL) → Opr1
б) MOV     AL, Opr1     ; (Opr1) → AL
   XCHG    AL, Opr2     ; (Opr2) → AL
   MOV     Opr1, AL     ; (AL) → Opr1
```

В варианте б) потребовалось на одну инструкцию и на один регистр меньше, чем в а).

PUSH (*поместить слово в стек*) помещает содержимое регистра или ячейки ОП размером в 16-битовое слово на вершину стека. Результат выполнения данного оператора (как и оператора POP) был рассмотрен в гл. 5. Примеры:

```
PUSH    AX        ; (AX) → стек
PUSH    Mem       ; (Mem) → стек
```

POP (*извлечь слово из стека*) выбирает слово из вершины стека и помещает его в ячейку памяти или в регистр. Пример:

```
POP     AX      ; Слово из вершины стека → AX
```

LEA (*загрузить адрес*) пересылает адрес (смещение) ячейки памяти в любой 16-битовый регистр данных, регистр-указатель или индексный регистр. Оператор имеет два операнда. В качестве первого из них записывается регистр, а в качестве второго — метка ячейки ОП. Пример:

```
LEA     BX, Mem     ; Смещение ячейки Mem → BX
```

12.2.4 Структура FLAGS и операции над ним

Большинство операторов не только выполняют действия над своими операндами, но и выполняют действия над флажками — битами регистра флагов FLAGS (рис. 12.1).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Рис. 12.1 – Регистр флагов микропроцессора i8086

Семь битов в FLAGS не используются. Остальные биты (флажки) делятся на условные и управляющие. *Условные флажки* отражают результат предыдущей арифметической или логической операции. Это:

1) **SF** — *флажок знака*. Равен старшему биту результата. Так как в дополнительном коде старший бит отрицательных чисел содержит 1, а у положительных он равен 0, то SF показывает знак предыдущего результата;
2) **ZF** — *флаг нуля*. Устанавливается в 1 при получении нулевого результата и сбрасывается в 0, если результат не равен 0;
3) **PF** — *флажок паритета*. Устанавливается в 1, если младшие 8 битов результата содержат четное число единиц: в противном случае он сбрасывается в 0;
4) **CF** — *флажок переноса*. При сложении (вычитании) устанавливается в 1, если возникает перенос (заем) из старшего бита (в старший бит);
5) **AF** — *флажок вспомогательного переноса*. Устанавливается в 1, если при сложении (вычитании) возникает перенос (заем) из бита 3. Флаг предназначен только для двоично-десятичной арифметики;
6) **OF** — *флажок переполнения*. Устанавливается в 1, если знаковый бит изменился в той ситуации, когда этого не должно было произойти.

Пример

Пусть, например, команда ADD выполнила следующее сложение:

$$\begin{array}{r} 0010\,0011\,0100\,0101 \\ +\ 0011\,0010\,0001\,1001 \\ \hline 0101\,0101\,0101\,1110 \end{array}$$

тогда после ее выполнения получаются состояния флажков:

$$SF = 0,\ ZF = 0,\ PF = 0,\ CF = 0,\ AF = 0,\ OF = 0$$

Если ADD выполнила сложение:

$$\begin{array}{r} 0101\,0100\,0011\,1001 \\ +\ 0100\,0101\,0110\,1010 \\ \hline 1001\,1001\,1010\,0011 \end{array}$$

то флажки принимают состояния:

$$SF = 1,\ ZF = 0,\ PF = 1,\ CF = 0,\ AF = 1,\ OF = 1$$

Флажки управления влияют на выполнение специальных функций. Эти флажки устанавливаются лишь несколькими специальными инструкциями. Это флажки:

1) **DF** — *флажок направления*. Он используется при выполнении инструкций, обрабатывающих цепочки — последовательности ячеек памяти. Если флаг сброшен, цепочка обрабатывается с первого элемента, имеющего наименьший адрес. Иначе — цепочка обрабатывается от наибольшего адреса к наименьшему;
2) **IF** — *флажок разрешения прерываний*. Когда установлен этот флажок, ЦП выполняет маскируемые прерывания. Иначе эти прерывания игнорируются;
3) **TF** — *флажок трассировки*. Если этот флажок установлен, то после выполнения каждой машинной инструкции ЦП генерирует внутреннее аппаратное прерывание (прерывание номер 1).

Существуют семь операторов, которые предназначены только для манипуляций флажками FLAGS. Они не имеют операндов и позволяют изменять CF, DF и IF. Это:

1) **STC** — устанавливает флаг переноса CF;
2) **CLC** — сбрасывает CF;
3) **CMC** — инвертирует CF;
4) **STD** — устанавливает флажок направления DF;
5) **CLD** — сбрасывает DF;
6) **STI** — устанавливает флажок разрешения прерываний IF;
7) **CLI** — сбрасывает IF.

Следующие операторы выполняют пересылку содержимого регистра FLAGS:

1) **LAHF** — пересылает младший байт FLAGS в регистр AH;
2) **SAHF** — пересылает содержимое регистра AH в FLAGS;
3) **PUSHF** — записывает содержимое FLAGS в стек;
4) **POPF** — выбирает слово из вершины стека и помещает его в регистр FLAGS.

Нетрудно заметить, что эти операторы позволяют изменять содержимое FLAGS.

12.2.5 Операторы сдвига

Такие операторы перемещают все биты первого операнда (байта или слова) влево или вправо на число, заданное вторым операндом. Вторым операндом может быть только 1 или регистр CL.

Для всех восьми операторов сдвига флаг переноса CF является как бы расширением сдвигаемого операнда: в CF загружается значение бита, выдвинутого за пределы операнда.

Операторы сдвига разделяются на логические и арифметические. *Логический оператор сдвига* рассматривает знаковый бит операнда как обычный бит, а *арифметический оператор сдвига* обрабатывает бит знака особо.

SHL — *логический сдвиг влево*. Этот оператор сдвигает число без знака (рис. 12.2). При каждом сдвиге в освободившийся нулевой бит заносится 0. Например, пусть (AL) = 10110100b, CF = 0, тогда:

```
SHL     AL, 1       ; 01101000 → AL, 1 → CF
```

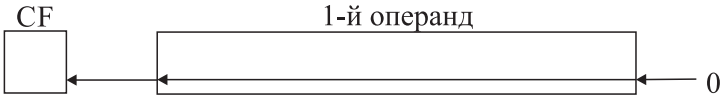


Рис. 12.2 – Логический сдвиг влево SHL

Одним из применений оператора SHL является умножение беззнаковых чисел на степень числа 2. Например, пусть (CL) = 2, тогда:

```
SHL     AX, CL      ; Умножение числа без знака в AX на 4
```

По сравнению с обычным умножением время выполнения в 6–8 раз меньше.

SAL — *арифметический сдвиг влево*. Этот оператор сдвигает число со знаком. Действие аналогично SHL. При этом содержимое знакового бита не сохраняется, но оно переписывается в флажок OF. Например, пусть (AL) = 10110100*b*, CF = 0, OF = 0, тогда:

```
SAL     AL, 1       ; 01101000 → AL, 1 → CF, 1 → OF
```

SHR — *логический сдвиг вправо*. Этот оператор сдвигает число без знака. При каждом сдвиге операнда в освободившийся старший бит (бит 7 для байта и бит 15 для слова) заносится 0 (рис. 12.3). Например, пусть (AL) = 10110100b, CF = 1, тогда:

```
SHR     AL, 1       ; 01011010 → AL, 0 → CF
```

Одним из применений оператора SHR является деление беззнаковых чисел на степень числа 2. Например, пусть (CL) = 2, тогда:

```
SHR     AX, CL      ; Деление беззнакового числа в AX на 4
```

Рис. 12.3 – Логический сдвиг вправо SHR

SAR — *арифметический сдвиг вправо*. Данный оператор сдвигает число со знаком. При сдвиге в старшие освобождающиеся биты дублируется знак операнда (рис. 12.4). Например, пусть (AL) = 10110100b, CF = 1, тогда:

```
SAR     AL, 1       ; 11011010 → AL, 0 → CF
```

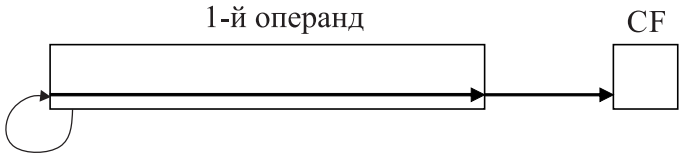


Рис. 12.4 – Арифметический сдвиг вправо SAR

Одним из применений оператора SAR является деление чисел со знаком на степень числа 2. Например, пусть (CL) = 3, тогда:

```
SAR     AX, CL      ; Деление числа со знаком в AX на 8
```

ROL — *циклический сдвиг влево*. При выполнении данного оператора (как и любого другого циклического оператора) вышедший за пределы операнда бит входит в него с противоположного конца (рис. 12.5). Например, пусть (AL) = 10110100b, CF = 0, тогда:

```
ROL     AL, 1       ; 01101001 → AL, 1 → CF
```

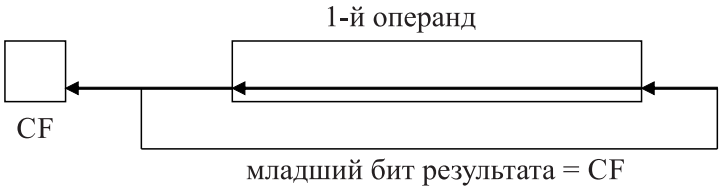


Рис. 12.5 – Циклический сдвиг влево ROL

ROR — *циклический сдвиг вправо* (рис. 12.6). Например, пусть (AL) = 10110100b, CF = 1, тогда:

```
ROR     AL, 1       ; 01011010 → AL, 0 → CF
```

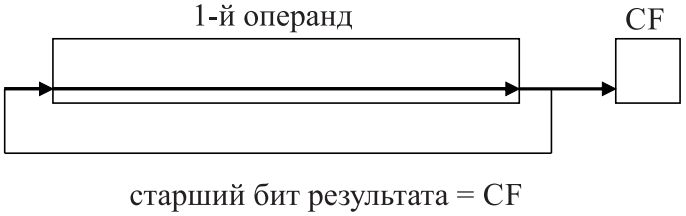


Рис. 12.6 – Циклический сдвиг вправо ROR

RCL — *циклический сдвиг влево через перенос* (рис. 12.7). Например, пусть (AL) = 10110100*b*, CF = 1, тогда:

```
RCL     AL, 1      ; 01101001 → AL, 1 → CF
```

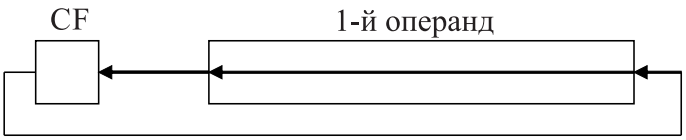


Рис. 12.7 – Циклический сдвиг влево через перенос RCL

RCR — *циклический сдвиг вправо через перенос* (рис. 12.8). Например, пусть (AL) = 10110100b, CF = 1, тогда:

```
RCR     AL, 1      ; 11011010 → AL, 0 → CF
```

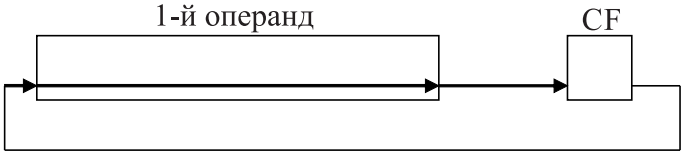


Рис. 12.8 – Циклический сдвиг вправо через перенос RCR

12.2.6 Цепочечные (строковые) операторы

Такие операторы предназначены для того, чтобы обрабатывать одной инструкцией последовательности из нескольких байт или слов. В результате программа становится короче.

Существуют пять групп цепочечных операторов. Из них далее рассматриваются лишь операторы пересылки строк. Это следующие операторы.

MOVSB — переслать цепочку байт. Оператор не имеет операндов. Ему может предшествовать префикс **REP**.

При отсутствии префикса оператор копирует всего один байт, расположенный в сегменте данных, в байт, принадлежащий дополнительному сегменту. Как задать байт в сегменте длиной 64 Кбайта? Для этого используются индексные регистры SI и DI. Регистр SI содержит номер (индекс) байта в сегменте данных, а DI — номер байта в дополнительном сегменте. В результате выполнения оператора происходит не только копирование байта, но и изменение на единицу содержимого регистров SI и DI. Направление изменения (увеличение или уменьшение) определяется флагом направления DF в регистре FLAGS. Если DF = 0, то адреса увеличиваются, а если 1, то уменьшаются.

Добавление к оператору MOVSB (как и к любому другому строковому оператору пересылки) префикса повторения команды REP многократно усиливает мощность строковой инструкции. При этом оператор MOVSB выполняется столько раз, каково содержимое регистра CX. При каждом выполнении содержимое CX уменьшается на единицу. Как только это содержимое станет равно нулю, то выполняется следующий за MOVSB оператор. Применение оператора MOVSB с префиксом REP требует выполнения следующих пяти шагов:

1) обнулить флаг DF (оператором CLD) или установить его (оператором STD) в зависимости от того, будет ли пересылка осуществляться от младших адресов к старшим или наоборот;

2) загрузить смещение адреса строки-источника в регистр SI;
3) загрузить смещение адреса строки-приемника в регистр DI;
4) загрузить число пересылаемых байтов в регистр CX;
5) выполнить оператор MOVSB с префиксом REP.

Пример

Первый пример. Следующий фрагмент программы копирует 100 байтов из строки Source, находящейся в сегменте данных, в строку Dest, находящуюся в дополнительном сегменте.

```
        CLD              ; 0 → DF
        LEA SI, Source   ; Смещение адреса Source →SI
        LEA DI, ES:Dest  ; Смещение адреса Dest →DI
        MOV CX, 100      ; Число пересылаемых байтов
REP MOVSB                ; Копирование цепочки байтов
```

Пример

Второй пример. Следующая процедура выполняет копирование цепочки 256 байтов, находящейся в сегменте данных, в цепочку этой же длины, но расположенную в дополнительном сегменте.

```
;
; Копирование 256-байтов
; - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; Входы : SI — содержит смещение адреса цепочки-источника
; DI — содержит смещение адреса цепочки-приемника
;
        PUSH CX          ; (CX) → стек
        PUSHF            ; (FLAGS) → стек
        CLD              ; 0 → DF
        MOV CX, 256      ; Счетчик байтов
REP MOVSB                ; Копирование цепочки байтов
        POPF             ; Восстановление FLAGS
        POP CX           ; Восстановление CX
        RET              ; Возврат из процедуры
```

Так как флаг направления может использоваться и в программе, вызывающей нашу процедуру, то надо временно сохранить его в стеке. Для этого используются инструкции PUSHF и POPF, выполняющие запись в стек и извлечение оттуда содержимого регистра FLAGS.

MOVSW — пересылка цепочки слов. Данный оператор копирует цепочку слов из сегмента данных в цепочку, расположенную в дополнительном сегменте. Детали применения данного оператора аналогичны оператору MOVSB.

12.3 Адресация данных

Многие исполнительные операторы ассемблера (арифметические, логические, передачи данных, строковые операторы) обрабатывают какие-то данные, являющиеся операндами этих операторов. Эти данные могут находиться в следующих местах:

1) в регистрах;
2) в ОП — в поле машинной инструкции, их обрабатывающей;
3) в ОП — вне сегмента кодов.

Регистровая адресация. Обрабатываемое данное находится в регистре. В следующем операторе оба операнда используют такую адресацию:

```
MOV     AX, BX
```

Непосредственная адресация. Пример оператора, второй операнд которого содержит непосредственные данные:

```
MOV     AX, 2
```

Непосредственно адресуемый операнд помещается в поле машинной инструкции, занимая в зависимости от типа команды один или два байта. Например, для приведенного выше оператора ассемблера соответствующая машинная инструкция имеет вид:

B80200,

где B8h — КОП, а два байта данных (02h и 00h) следуют за ним (младщий байт слова расположен первым в памяти). Данные, задаваемые в операторе непосредственно, можно только читать, но менять их нельзя.

Все остальные типы адресации предназначены для работы с данными, находящимися в ОП вне сегмента кодов. Основным местом, предназначенным для хранения таких данных, является сегмент данных (рис. 12.9). Местоположение любого байта или слова данных однозначно задается смещением L относительно начала сегмента данных. С учетом того, что регистр сегмента данных DS содержит начальный параграф сегмента данных, реальный адрес соответствующей ячейки в ОП: $R = (DS) \times 16 + L$.

Различные способы (режимы) адресации отличаются друг от друга тем, как они задают смещение L. Вот некоторые из них.

Прямая адресация. 16-битное смещение L является частью исполнительного оператора. Пример:

```
MOV     Max, AL,
```

где Max — метка ячейки в сегменте данных. В поле машинной инструкции находится 16-битное слово, содержащее L для байта, помеченного в исходной программе как Max.

Регистровая косвенная адресация. Искомое смещение L находится или в базовом регистре BX или в индексном регистре (DI или SI). Регистр задается в опера-

торе ассемблера, заключенным в квадратные скобки. Например, следующий оператор

```
MOV     [BX], DX
```

пересылает содержимое регистра DX в ячейку (слово) ОП, адрес которой находится в регистре BX.

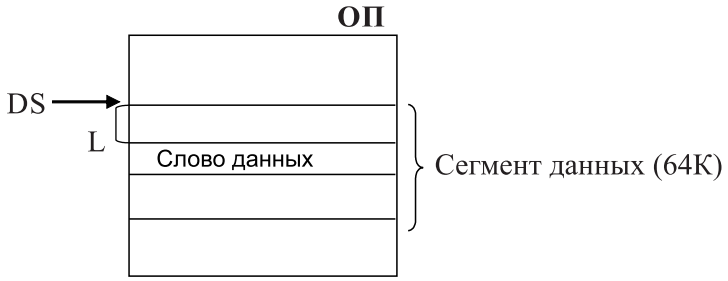


Рис. 12.9 – Расположение сегмента данных в ОП

Относительная регистровая адресация. Это «гибрид» прямой и регистровой косвенной адресации для получения L. К заданному в операторе смещению прибавляется содержимое базового или индексного регистра. Смещение можно задать двумя способами: а) меткой; б) непосредственно, т. е. числом. Примеры:

```
а) MOV     Max [BX], DX
```

содержимое регистра DX записывается в слово ОП, смещение L для которого получается путем суммирования смещения для метки MAX с содержимым регистра BX;

```
б) MOV     [10+BX], DX
```

содержимое DX записывается в ячейку ОП, имеющую L = 10 + (BX).

Базовая индексная адресация. Смещение L равно сумме содержимых базового регистра BX и индексного (SI или DI), заданных в операнде. Пример:

```
MOV     AL, [BX][SI]
```

содержимое байта ОП, имеющего L = (BX) + (SI), переписывается в регистр AL.

Относительная базовая индексная адресация. Смещение L равно сумме трех слагаемых: а) заданного в операторе смещения; б) содержимого базового регистра BX; в) содержимого индексного регистра SI или DI. Примеры:

```
а) MOV     AL, Max[BX][DI]
           L = смещение для Max + (BX) + (DI).
б) MOV     AL, [10+BX][DI]
           L = 10 + (BX) + (DI).
```

12.4 Определение данных

Начальное состояние сегмента данных к моменту начала выполнения программы задается программистом с помощью псевдооператоров определения данных. С помощью них присваиваются метки различным элементам сегмента данных и, возможно, загружаются в них первоначальные значения.

12.4.1 Метки

При построении нами машинных программ с помощью *Debug* команды переходов содержали адреса тех команд программы, на которые переход делался. Допустим, что мы решили добавить в программу новые команды. В этом случае от нас требуется изменить многие ранее записанные адреса переходов, что чрезвычайно неудобно.

При написании программы на ассемблере вместо адресов перехода мы используем метки тех операторов, на которые делается переход. То же самое относится и к процедурам: первые операторы процедур мы помечаем метками, которые теперь являются именами соответствующих процедур и используются в операторах вызова этих процедур. Рассмотрим правило записи меток.

Максимальная длина метки — 31 символ. Это могут быть следующие символы:

1) латинские буквы — от *A* до *Z* и от *a* до *z*;
2) цифры — от 0 до 9;
3) специальные символы — знак вопроса «?»; знак «@»; точка «.»; подчеркивание «_»; доллар «*$*».

Первым символом в метке должна быть буква или специальный символ. Точка может использоваться в метке только в качестве первого символа. В отличие от операционных систем *WINDOWS*, *DOS*, а также многих других программ транслятор-ассемблер *NASM* различает заглавные и строчные буквы в именах меток. Для него это совершенно разные символы. Примеры меток: *count*, *Page*25, $*E*10. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

Метки бывают локальными и глобальными. *Глобальная метка* может использоваться во всем исходном файле. Имя такой метки не может начинаться с точки. Имя *локальной метки* начинается с точки. Областью действия такой метки является фрагмент исходной программы, заключенный между двумя ближайшими глобальными метками. Вне этого фрагмента может быть задана другая локальная метка с точно таким же именем.

Особенно полезно применение локальных меток внутри процедур. Дело в том, что в разных процедурах часто используются идентичные метки. В процессе сборки исходной программы транслятором (такая сборка производится при обработке рассматриваемого позже псевдооператора *include*) он выведет сообщения об ошибках только в том случае, если идентичные метки являются глобальными. Если эти метки оформлены как локальные, то никаких ошибок не будет, так как это совершенно различные метки. Заметим, что использование идентичных меток внутри процедур очень удобно для программиста, позволяя присваивать схожим фрагментам процедур одинаковые имена. Пример: использование метки «*.Exit*» для заключительного фрагмента процедуры.

Некоторые из буквенных слов зарезервированы транслятором-ассемблером и не могут быть использованы в качестве меток. Сюда относятся имена регистров (например, *AX*), мнемоники исполнительных команд (например, *add*), указатель на точку входа (*..start*) и псевдооператоры, например *«end»*.

12.4.2 Определение байтов

DB — определение байта. Данный псевдооператор просит транслятор выделить один или несколько байтов ОП и сообщает ему, что первому из этих байтов присваивается указанная метка. Кроме того, возможно, от транслятора требуется записать в эти байты первоначальное содержимое. С помощью одного псевдооператора DB можно определить один байт или ***массив*** байтов [3]. Основные варианты:

а) инициализация (первоначальная запись) не требуется:

```
F1db    DB    ?              ; Один байт
F2db    DB    ?,?,?          ; Массив из трех байтов
F3db    DB    3 DUP (?)      ; –//–
```

б) размещение в последовательности байтов символьной строки (в один байт записывается код ASCII одного символа):

```
F4db    DB   'H', 'E', 'L', 'L', 'O' ; В массиве из пяти байтов
                                     ; HELLO
F5db    DB   'HELLO'                ; –//–
F6db    DB   "HELLO"                ; –//–
```

в) размещение в байте десятичной константы:

```
F7db    DB    32
F8db    DB    10 DUP (0) ; Массив из десяти байтов с нулями
```

г) шестнадцатеричная константа:

```
F9db    DB    20h
```

д) двоичная константа:

```
F10db    DB    01011001b
```

е) смешанные данные:

```
F11db    DB    0, ?, ?, ?, 0
F12db    DB    'TABL1', 10h, 20h, 30h
```

12.4.3 Определение слов

DW — определение слова. Основные варианты:

а) без инициализации:

```
F1dw    DW    ?              ; Одно слово
F2dw    DW    10 DUP (?)     ; Массив из 10 слов
```

б) шестнадцатеричная константа:

```
F3dw    DW    0FFF0h
```

в) двоичная константа:

```
F4dw    DW    0101010101010101b
```

г) адресная константа:

```
F5dw    DW    F10DB     ; В слове смещение L для байта с меткой
                        ; F10DB
```

DD — определение двойного слова. Резервируется область ОП длиной в два слова (четыре байта). Максимальное число без знака 0FFFFFFFFh.

DQ — определение последовательности четырех слов (восьми байтов).

12.4.4 Определение констант

EQU — память не резервируется, а лишь задается инициализирующее значение. Например, пусть в сегменте данных имеется псевдооператор:

```
Times    EQU    10,
```

тогда в каком бы исполнительном операторе или псевдооператоре ни использовалось слово Times, транслятор-ассемблер подставит вместо него 10. Например, он преобразует

```
Fielda    DB    TIMES DUP (?)
```

в оператор:

```
Fielda    DB    10 DUP (?)
```

Другой пример:

```
Countr    EQU    05
          MOV    CX, Countr
```

транслятор сделает замену оператора MOV:

```
MOV    CX, 05.
```

В следующем примере переопределяется имя регистра CX:

```
Countr    EQU    CX
```

= — применение данного псевдооператора схоже с оператором EQU. Отличие: выражение справа может быть только числовым. Примеры:

```
Times = 10
Countr = 5
```

12.4.5 Структуры

В отличие от простых данных, определяемых с помощью псевдооператоров DB, DW, DD, DQ, ***структура*** представляет собой сложный тип данных, т. к. ее элементы (поля) имеют разную длину. Программист определяет структуру с помощью псевдооператоров **STRUC** и **ENDS**:

```
<имя> STRUC
. . . . . ; Описание полей структуры
<имя> ENDS
```

Здесь описание полей представляет собой последовательность псевдооператоров определения данных (DB, DW, DD и DQ). Их операнды определяют размер полей и задают, если нужно, их начальные значения. Пример описания структуры:

```
WINDOW    STRUC
Height    DW ?     ; Высота окна
Width     DW ?     ; Ширина окна
Color     DB ?     ; Цвета символов и фона
Text      DD ?     ; Дальний указатель (сегмент и смещение)
                   ; на буфер, содержащий выводимый текст
WINDOW    ENDS
```

Где расположить описание структуры? Здесь можно использовать два варианта. В первом из них описание структуры располагается в начале того же файла, где эта структура будет использоваться. Недостаток такого варианта очевиден, т. к. в каждом файле, где предполагается использовать структуру, ее придется описывать заново. Во втором варианте описание структуры выносится в отдельный файл

с расширением .inc (текстовый файл, полученный с помощью любого текстового редактора). А в начале каждого ассемблерного файла, где предполагается использовать структуру, записывается псевдооператор **INCLUDE** (*включить*) с указанием имени включаемого файла. Например:

```
INCLUDE     Window.inc
```

Вне зависимости от того, каким из двух способов определена структура, она может использоваться в программе следующим образом. Во-первых, имя структуры фактически является типом данных, пользуясь которым можно требовать от транслятора создание любого числа экземпляров данной структуры в памяти. Например, для создания трех экземпляров структуры WINDOW достаточно записать:

```
Wind1       WINDOW <>
Wind2       WINDOW <>
Wind3       WINDOW <>
```

Для инициализации экземпляра структуры, т. е. для задания первоначальных значений каких-то ее полей, поля структуры выделяются запятыми внутри угловых скобок. При этом требуемое содержимое поля записывается в его позицию. Например:

```
Wind1       WINDOW <10,20,,>
```

Здесь высота и ширина окна инициализируются значениями 10 и 20, а цвет и выводимый в окно текст первоначально не задаются.

Если экземпляр структуры определен в программе, с его полями можно работать так же, как с обычными переменными. Единственное отличие — имя (метка) поля экземпляра структуры составное. Оно состоит из имени экземпляра структуры и имени поля структуры, разделенных символом «.». Например, следующие операторы выполняют запись в поле Text экземпляра Wind1 адреса сообщения, находящегося в поле памяти с меткой Buff:

```
MOV     Wind1.Text, OFFSET Buff     ; Запись смещения
MOV     Wind1.Text+2, SEG Buff      ; Запись сегмента
```

12.5 Операторы передачи управления

Основные типы операторов передачи управления:

1) операторы условных переходов;
2) операторы безусловных переходов;
3) операторы циклов;
4) операторы процедур;
5) операторы программных прерываний;
6) операторы останова и холостого хода.

12.5.1 Операторы условных переходов

Формат оператора условного перехода:

```
КОП     α,
```

где α — метка, или метка $+(-)$ выражение, вычисление которого дает константу. Операнд α указывает на тот оператор в программе, на который делается переход в случае выполнения предусмотренных в операторе перехода условий. Такими условиями являются значения одного, двух или трех флажков.

Машинная инструкция, соответствующая оператору условного перехода, имеет длину 2 байта. В первом байте находится КОП, а во втором — «расстояние» между содержимым IP и искомым адресом. Это «расстояние» есть число со знаком, т. к. переходы могут делаться как вперед по программе, так и назад.

В одном байте можно разместить число со знаком (в дополнительном коде) от −128 до +127. Это приводит к тому, что оператор условного перехода может использоваться лишь для небольших переходов. Для выполнения больших переходов, в том числе и в другие программные сегменты, оператор данного типа дополняется операторами безусловного перехода.

Пример

Подсчитаем содержимое второго байта в операторе перехода в следующем фрагменте:

```
0050 Again:    INC CX
0052           ADD AX, [BX]
0054           JZ Again
0056 Next:     MOV Result, CX
```

В момент выполнения JZ (IP)=0056h. Следовательно, второй байт JZ должен содержать −6. В дополнительном коде это FAh.

Некоторые операторы условного перехода:

1) **JZ** (или **JE**) — перейти, если нуль или равно. Условием перехода является ZF = 1;
2) **JNZ** (или **JNE**) — перейти, если не нуль или не равно. Условие перехода: ZF = 0.

Следующие операторы перехода записывают в программу только после операторов, выполняющих действие над беззнаковыми данными. Эти операторы перехода не учитывают ни флаг знака SF, ни флаг переполнения OF. Для них важен флаг переноса CF:

1) **JA** (или **JNBE**) — перейти, если больше. Условие: $(CF = 0)\&(ZF = 0)$;
2) **JAE** (или **JNB**) — перейти, если больше или равно. Условие: CF = 0;
3) **JB** (или **JNAE**) — перейти, если меньше. Условие: CF = 1;
4) **JBE** (или **JNA**) — перейти, если не больше. Условие: $(CF = 1)V(ZF = 1)$.

Следующие операторы перехода записывают в программу только после операторов, выполняющих действия над знаковыми данными. Для них важны флаг знака SF и флаг переполнения OF:

1) **JG** (или **JNLE**) — перейти, если больше. Условие: $(SF = OF)\&(ZF = 0)$;
2) **JGE** (или **JNL**) — перейти, если больше или равно. Условие: SF = OF;

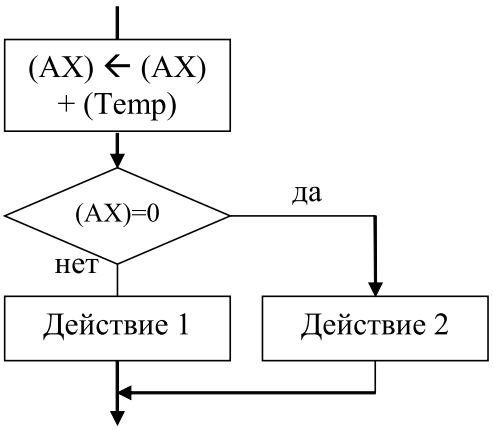
3) **JL** (или **JNGE**) — перейти, если меньше. Условие: SF ≠ OF;
4) **JLE** (или **JNG**) — перейти, если меньше или равно. Условие: (SF ≠ OF) V V(ZF = 1).

Пример

Каждому условному оператору перехода соответствует противоположный по смыслу оператор. Например, приведенные на рис. 12.10 два фрагмента программ эквивалентны.

a)

```
        ADD   AX, Temp
        JZ    Ac_2
        ____________
        ____________   Дейс. 1
        ____________
        JMP   Exit
Ac_2:   ____________
        ____________   Дейс. 2
        ____________
Exit:   . . . . . .
```



b)

```
        ADD   AX, Temp
        JNZ   Ac_1
        ____________
        ____________   Дейс. 2
        ____________
        JMP   Exit
Ac_1:   ____________
        ____________   Дейс. 1
        ____________
        ____________
Exit:   . . . . . .
```

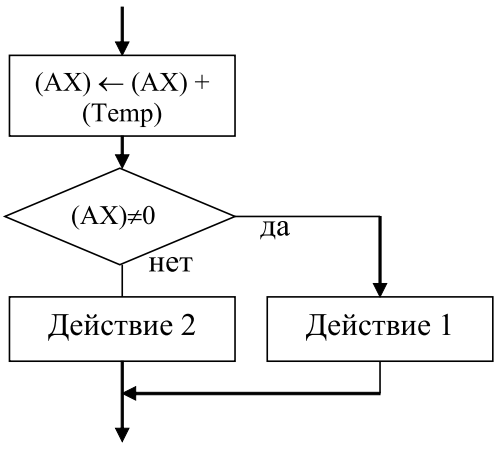


Рис. 12.10 – Фрагменты программ *a* и *b* эквивалентны

Несмотря на логическую эквивалентность двух фрагментов программ, продолжительность их выполнения скорее всего будет различной. Дело в том, что время выполнения условного оператора в случае перехода в четыре раза больше времени выполнения этого же оператора, если переход не делается.

Поэтому из двух альтернативных фрагментов желательно использовать тот, в котором вероятность перехода по условному оператору меньше.

12.5.2 Операторы безусловных переходов

Такой оператор заставляет ЦП извлечь новую инструкцию не из следующей ячейки ОП, а из какой-то другой, известной еще до начала выполнения программы. Существуют пять машинных инструкций безусловных переходов. Все они имеют одну и ту же ассемблерную мнемонику **JMP** и один операнд. Это:

1) внутрисегментный прямой короткий переход;
2) внутрисегментный прямой переход;
3) внутрисегментный косвенный переход;
4) межсегментный прямой переход;
5) межсегментный косвенный переход.

Во *внутрисегментном переходе* оператор перехода находится в том же сегменте, что и оператор, на который делается переход. А в *межсегментном переходе* — в разных сегментах.

В операторах *прямого перехода* операндом является метка того оператора, на который делается переход. При внутрисегментном переходе данная метка имеет тип **NEAR** (близкий). Этот тип задается путем записи после метки символа «:». При межсегментном переходе метка перехода имеет тип **FAR** (дальний).

Внутрисегментный прямой короткий переход используется для переходов от +127 до −127 байтов. Инструкция перехода имеет длину 2 байта. В первом байте КОП, а во втором число со знаком, представляющее собой «расстояние» до искомого оператора. Это число идентично соответствующему числу в операторе условного перехода.

Для указания транслятору, что переход короткий, используется слово **SHORT**. Например:

```
JMP     SHORT     A90
. . . . . .
A90:
```

Если внутрисегментный переход не короткий, слово SHORT опускается. Машинная инструкция в этом случае занимает 3 байта — один для КОП и 2 — для «расстояния» перехода.

Если переход в программе осуществляется «назад», то слово SHORT можно не писать вовсе. Так как при записи инструкции JMP транслятор уже «знает» расстояние до искомого оператора. При переходе вперед это расстояние транслятору неизвестно, и при отсутствии слова SHORT он всегда записывает трехбайтовую инструкцию перехода.

Внутрисегментный косвенный переход задается путем записи в качестве операнда оператора JMP не метки, а адреса данных. По этому адресу (в регистре или в слове ОП) записано требуемое содержимое IP. Допустим, имеется оператор:

```
JMP     BX
```

Если в момент исполнения соответствующей машинной инструкции (BX) = = 1ABh, то ЦП запишет в IP это число 1ABh и извлечет следующую машинную инструкцию из ОП по физическому адресу (CS) × 16 + 1ABh.

12.5.3 Операторы циклов

Цикл — многократное повторение группы операторов, называемых *телом цикла*, до тех пор, пока выполняется некоторое условие (рис. 12.11).

Типичное условие цикла: повторить тело цикла заданное число раз. Число повторений обычно заносится в регистр CX. Реализация такого цикла с помощью условного оператора JNZ:

```
        MOV     CX, N
Begin:
        . . . . . .          } Тело цикла
        DEC     CX
        JNZ     Begin
```

Это же самое можно сделать с помощью специального оператора цикла **LOOP**:

```
        MOV     CX, N
Begin:
        . . . . . .          } Тело цикла
        LOOP    Begin
```

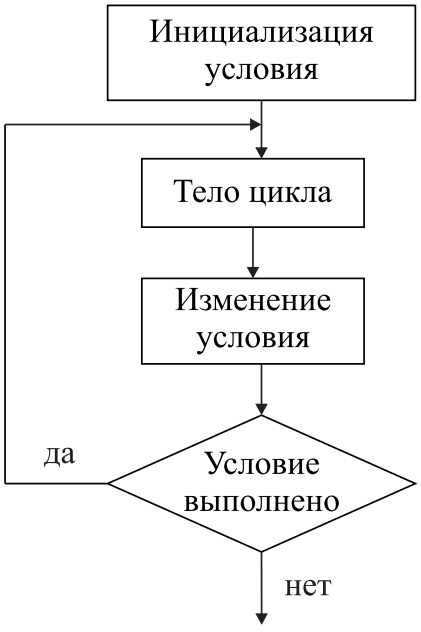


Рис. 12.11 – Блок-схема цикла

Оператор LOOP уменьшает содержимое регистра CX на 1 и выполняет переход, если (CX) ≠ 0. Следующий фрагмент программы выполняет сложение M слов, начинающихся с адреса Array. Результат записывается в слово с именем Total:

Пример

```
        MOV     CX, M
        MOV     AX, 0
        MOV     SI, AX
L1:     ADD     AX, Array[SI]
        ADD     SI, 2           ; Инкремент индекса на 2
        LOOP    L1
        MOV     Total, AX
```

Существуют еще два оператора циклов — **LOOPZ** (или **LOOPE**) и **LOOPNZ** (или **LOOPNE**). Условием повторения для LOOPZ является (CX) ≠ 0 & ZF = 1. То есть кроме ненулевого содержимого счетчика CX требуется, чтобы был установлен флаг нуля. Условием повторения для LOOPNZ является (CX) ≠ 0 & ZF = 0. Подобный оператор обычно используется для поиска в массиве заданного элемента.

Пример

Пусть метка Ascii присвоена первой ячейке массива из L символов. Требуется найти в этом массиве пробел (код ASCII пробела — 20h). Если пробела нет, требуется перейти на оператор с меткой Not. Соответствующий фрагмент программы:

```
        MOV     CX, L
        MOV     SI, -1          ; Инициализировать индекс
        MOV     AL, 20h         ; Код пробела → AL
Next:   INC     SI              ; Инкремент индекса
        CMP     AL, Ascii[SI]   ; Проверка на пробел
        LOOPNE  Next            ; Если не пробел, то цикл
        JNZ     Not
```

Машинная инструкция, соответствующая оператору цикла, имеет длину 2 байта. В первом байте находится КОП, а во втором — «расстояние» до инструкции, помеченной меткой перехода. Подобно инструкции условного перехода это «расстояние» может находиться в пределах от −128 до +127 байт.

12.5.4 Операторы процедур

Ранее в п. 5.2 рассматривались машинные инструкции, выполняющие вызов процедуры и возврат из нее. В языке ассемблера этим инструкциям соответствуют операторы **CALL** и **RET**. Основное отличие оператора CALL от соответствующей инструкции заключается в том, что в качестве операнда записывается не адрес первой инструкции процедуры, а его заменитель — имя (метка) процедуры. Пример:

```
CALL    Read_hex,
```

где Read_hex — имя процедуры.

Для задания имени процедуры используется псевдооператор **PROC**, которому всегда соответствует псевдооператор **ENDP**, записываемый в конце процедуры, например:

```
Read_hex    PROC NEAR
. . . . . .             ; Операторы процедуры
RET                     ; Возврат из процедуры
Read_hex    ENDP
```

При этом слово **NEAR** (ближний) имеет тот же смысл, что и для оператора безусловного перехода, а именно — вызов процедуры Read_hex производится из того же сегмента ОП, в котором расположена данная процедура. Если процедура может вызываться и из других сегментов, то ей присваивается тип **FAR**.

12.5.5 Другие операторы передачи управления

Это следующие операторы.

INT, IRET — операторы программного прерывания и возврата из него. Данные операторы не имеют смысловых отличий от соответствующих машинных инструкций, рассмотренных в п. 5.3. Единственное, на что требуется обратить внимание, это тип системы счисления операнда в операторе INT. Так как «родной» для ассемблера является десятичная система, то любое шестнадцатеричное число должно записываться по правилам, принятым в ассемблере. Это:

1) если старшей «цифрой» числа является буква, то перед ней записывается 0;
2) сразу за числом записывается буква h.

Пример

```
INT     21h.
```

12.6 Вспомогательные псевдооператоры

Вспомогательные псевдооператоры — дополняют основные операторы (исполнительные и псевдооператоры) с целью уточнения их операндов. Примером такого оператора является рассмотренный в п. 6.5 псевдооператор SHORT. Рассмотрим некоторые другие вспомогательные операторы.

SEG и **OFFSET** — вычисляют значения номера параграфа и смещения, образующих логический адрес программного объекта, заданного в ассемблерной программе своей меткой. Например, операторы

```
MOV     AX, SEG Table
MOV     BX, OFFSET Table
```

загрузят номер параграфа и смещение адреса переменной Table в регистры AX и BX соответственно.

$ — вычисляет адрес (смещение) той точки программы, где находится данный оператор $. Например, при трансляции операторов

```
Mess    DB      'Hello'
Messl   EQU     $-Mess
```

транслятор определит число символов в строке Mess и присвоит его константе Messl.

PTR — изменяет тип операнда. Например, если тип операнда BYTE, то его можно заменить на WORD. А если тип операнда WORD, то его можно заменить на BYTE. Например, с помощью этого оператора можно получить доступ к отдельным байтам в таблице слов. Пусть таблица определена следующим образом:

```
Table DW 100 DUP(?)
```

тогда оператор

```
First_byte EQU BYTE PTR Table
```

присвоит имя First_byte первому байту таблицы Table. Можно присвоить имя любому другому байту этой таблицы, например:

```
Fifth_byte EQU First_byte + 4
```

С помощью оператора PTR можно заменить тип метки NEAR на тип FAR, и наоборот. Например, если программа содержит оператор

```
Start:  MOV     CX, 100
```

то метка Start имеет тип NEAR, что позволяет ссылаться на нее инструкциям перехода, находящимся в том же сегменте памяти. Для того, чтобы на это же место программы могли делать переходы инструкции JMP, расположенные в других сегментах ОП, необходимо задать в программе альтернативную метку типа FAR:

```
Far_start EQU FAR PTR Start
```

LABEL — позволяет задать новое имя и новый тип участку программы, расположенному непосредственно за псевдооператором LABEL. В следующем примере переменная Dd1 имеет тип DWORD (двойное слово). С помощью оператора LABEL ей присваивается второе имя Dw1 и тип WORD:

```
Dw1     LABEL WORD      ; Dw1 — имя первого слова переменной
                        ; Dd1, которое содержит 5678h
Dd1     DD              12345678h
```

Так как псевдооператор LABEL не требует от транслятора выделения нового участка памяти, а лишь задает альтернативное имя и тип этого участка, то он может

рассматриваться как «конкурент» псевдооператора PTR. Например, следующие два оператора выполняют одно и то же, а именно — загружают в регистр AX число 5678h:

```
MOV     AX, Dw1
MOV     AX, WORD PTR Dd1
```

12.7 Макрооператоры

Макрооператор — оператор, заменяющий в программе на ассемблере последовательность обычных операторов (исполнительных операторов и псевдооператоров). То есть вместо того, чтобы записывать в программе несколько операторов, мы помещаем в нее единственный макрооператор. Структура макрооператора:

<имя макрооператора> [список фактических параметров]

Пример макрооператора:

OUT_STR Buff

Здесь макрооператор OUT_STR предназначен для вывода на экран строки символов, расположенной в области памяти с именем Buff.

В начале своей работы транслятор-ассемблер выполняет для каждого макрооператора *макрорасширение* — заменяет макрооператор на соответствующую совокупность операторов ассемблера. После завершения макрорасширений текст программы не содержит макрооператоров и представляет собой обычную ассемблерную программу.

Для того чтобы транслятор мог выполнить макрорасширение, имя макрооператора должно быть определено в *макроопределении*. Структура макроопределения:

<имя макрооператора> **MACRO** [список формальных параметров]

. . . . . ; Тело макроопределения

ENDM

Пример макроопределения:


```
OUT_STR MACRO Str
;
; Вывод строки на экран
; - - - - - - - - - - - - - - - - - - - - -
; Вход: Str — выводимая строка
;
    PUSH    AX
    PUSH    DX
    MOV     AH, O9h
    MOV     DX, OFFSET Str
    INT     21h
    POP     DX
```

```
        POP     AX
        ENDM
```

. . . . . .

При выполнении макрорасширения для OUT_STR транслятор заменит этот макрооператор на тело макроопределения, причем он заменит формальный параметр Str в теле макроопределения на фактический параметр Buff. (В примере такая замена выполняется в единственном операторе MOV.)

Где расположить макроопределение? Здесь можно использовать два варианта. В первом из них макроопределение располагается в начале того же файла, где оно будет использоваться. Недостаток: в каждом файле, где предполагается использовать аналогичный макрооператор, его придется описывать заново. Во втором варианте макроопределение выносится в отдельный файл (текстовый ASCII-файл, полученный с помощью любого текстового редактора). А в начале каждого ассемблерного файла, где предполагается использовать макроопределение, записывается псевдооператор INCLUDE с указанием имени включаемого файла. Например:

```
INCLUDE     Outstr.inc
```

Обычно в один включаемый файл помещают несколько макроопределений и, возможно, описания структур (которые фактически являются разновидностью макроопределений). При выполнении транслятором псевдооператора INCLUDE весь включаемый файл подсоединяется к тексту программы, содержащей INCLUDE, но на текст машинной программы наличие этого файла не влияет.

Полезно выполнить сравнение макроопределений с подпрограммами. Оба эти типа модулей позволяют программисту инициировать выполнение многих операторов с помощью единственного оператора (CALL или INT для подпрограммы и макрооператор для макроопределения). Но механизм их реализации совершенно различен. В то время, как тело макроопределения будет записано транслятором всюду, где он встретит соответствующий макрооператор, тело подпрограммы находится в памяти в единственном экземпляре, а операторы CALL (или INT) выполняют передачу управления этому экземпляру. Отсюда можно сделать более обоснованным выбор между этими типами модулей.

Так как каждое выполнение подпрограммы связано с «накладными расходами» в виде двух инструкций (CALL и RET или INT и IRET), то вклад этих инструкций в продолжительность выполнения тем больше, чем меньше подпрограмма. С другой стороны, чем длиннее тело макроопределения, тем больше затраты памяти на его «тиражирование». Следовательно, короткие повторяющиеся последовательности операторов лучше оформлять в виде макроопределений, а длинные — в виде подпрограмм.

Контрольные вопросы по главе 12

1) Какие арифметические операторы ассемблера Вы знаете?
2) Какие логические операторы ассемблера Вы знаете?
3) Какие операторы переходов Вы знаете?
4) Какие операторы сдвигов Вы знаете?
5) Какие операторы определения данных Вы знаете?

Глава 13

ПРИМЕР ПРОГРАММЫ НА АССЕМБЛЕРЕ

В качестве примера запишем на ассемблере ту программу *Writestr*, которую мы уже создали с помощью *Debug*. Напомним текст машинной программы:

Пример

```
100     mov ah,02
102     mov dl,2a
104     int 21
106     int 20
```

Соответствующий текст программы на ассемблере:

```
[org 100h]
mov     ah,2h
mov     dl,2ah
int     21h
int     20h
```

Следует отметить, что если в программе есть шестнадцатеричные числа типа *ACh*, то чтобы транслятор не запутался с ними (он может принять их за имя), всякое шестнадцатеричное число, начинающееся с буквы, следует предварять нулем. Например: 0*ACh*.

13.1 Подготовка программы к выполнению

Текст программы на ассемблере или на любом другом языке программирования называется *исходной программой*. Для того чтобы преобразовать этот текст в машинную программу, нам нужна помощь не только со стороны транслятора, но и от некоторых других системных программ.

Во-первых, с помощью текстового редактора мы получаем исходный файл программы. Имя исходного файла должно иметь расширение «*.asm*». Можно использовать любой редактор, который выдает результирующий текст в коде *ASCII*. Примерами такого редактора является *Edit*, запускаемый из командной строки *DOS*, или текстовый редактор в *DOSNavigator*.

Создайте исходный файл *Writestr.asm*, поместив в него приведенную выше программу на ассемблере. Убедитесь, что это именно *ASCII*-файл. Для этого, находясь в *DOS*, напечатайте:

C:\> TYPE Writestr.asm

Вы должны увидеть тот же текст, который ввели в текстовом редакторе. Если вы увидите в вашей программе странные символы, то для ввода текста программ следует использовать другой текстовый редактор. Теперь давайте начнем ассемблировать программу *Writestr*:

C:\> NASM Writestr.asm -o Writestr.com

Ответного сообщения транслятора в случае успешной трансляции не будет:

C:\>

В результате транслятор-ассемблер создал файл, называющийся *Writestr.com*, который вы найдете на диске. Конечная часть команды «*-o Writestr.com*» используется для задания имени результирующему файлу (*Writestr.com*). При отсутствии этой части результирующий файл получит имя *Writestr*.

Напечатайте «*Writestr.com*», чтобы запустить *com*-файл, и убедитесь, что Ваша программа функционирует правильно (напоминаем, что она должна печатать звездочку на экране).

Теперь введем созданный *com*-файл в *Debug* и разассемблируем его, чтобы увидеть получившуюся машинную программу:

```
C:\DEBUG Writestr.com
_U
1593      :0100 B402      mov ah,02
1593      :0102 B22A      mov dl,2a
1593      :0104 CD21      int 21
1593      :0106 CD20      int 20
```

13.2 Комментарии

Программа на ассемблере может содержать любые сообщения, информирующие программиста о содержании текста программы. Такое сообщение называется *комментарием*. Комментарии могут находиться на любых строках программы. На каждой строке, где есть комментарий, ему предшествует точка с запятой (;). Подобно псевдооператорам комментарии не транслируются ни в какие машинные

команды. Но в отличие от псевдооператоров они не интересуют и сам транслятор, существуя лишь для человека, который читает программу.

Добавим комментарии в записанную ранее программу на ассемблере:

```
    [org 100h]
;
;   Вывод звездочки на экран
;   - - - - - - - - - - - - - - - - - - - -
;
    mov ah,2h       ; Функция вывода символа
    mov dl,2ah      ; Символ * в DL
    int 21h         ; Вывод символа
    int 20h         ; Возврат в DOS
```

Теперь можно легко понять смысл программы. Обратите внимание на комментирование процедуры. Каждая процедура должна обязательно иметь вводные комментарии, которые содержат:

1) словесное описание функций процедуры;
2) перечень и способ передачи каждого входного и выходного параметра процедуры;
3) перечень процедур, вызываемых в данной процедуре;
4) перечень переменных (областей памяти), которые используются процедурой, с указанием для каждой переменной способа ее использования. Допустимые способы использования: чтение; запись; чтение-запись.

Для приведенной выше простой процедуры вводный комментарий содержит лишь описание функций процедуры.

Что касается текущих комментариев, поясняющих назначение отдельных операторов программы и их групп, то к ним нет жестких требований. Желательно, чтобы были прокомментированы основные управляющие структуры программы (цепочки, ветвления и циклы). Кроме того, должны быть прокомментированы вызовы подпрограмм, т. е. операторы *call* и *int*. Каждый такой оператор инициирует десятки или сотни машинных команд и поэтому заслуживает пояснения.

Следует помнить, что исходная программа без комментариев не имеет какой-либо коммерческой ценности. Это просто-напросто черновик автора программы. Более того, по истечению нескольких месяцев даже автору требуются значительные усилия на то, чтобы понять смысл программы.

13.3 Еще один пример программы

Ранее в п. 7.4 мы создали небольшую программу, которая выводила на экран буквы от *A* до *J*.


Пример

Вот эта машинная программа:

```
100     mov dl,41
102     mov cx,000a
105     call 0200
108     loop 0105
10A     int 20
200     mov ah,02
202     int 21
204     inc dl
206     ret
```

Перепишем теперь эту программу на языке ассемблера:

```
        [org 100h]
;
;       Вывод 10 символов от A до J
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Вызовы: Write_char
;
Start:
        mov dl,'A'          ; В DL код буквы A
        mov cx,10           ; В счетчике символов -10
.Loop:  call Write_char     ; Вывод символа
        loop .Loop          ; Переход к следующему символу
        int 20h             ; Возврат в DOS
;
;       Вывод символа на экран и увеличение кода символа на 1
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: DL содержит код символа
;       Выходы: DL содержит код нового символа
;
Write_char:
        push ax
        mov ah,02       ; Функция вывода символа
        int 21h         ; Вывод символа на экран
        inc dl          ; Код следующего символа
        pop ax
        ret             ; Возврат из процедуры
```

Обратите внимание, что в первом операторе программы второй операнд представляет собой символ, заключенный в кавычки (кавычки могут быть как одиночными, так и двойными). Это означает, что операнд представляет собой код *ASCII*-символа, заключенного в кавычки.

Отметим попутно еще одну весьма полезную деталь ассемблера. Данный язык позволяет записывать в качестве операнда арифметическое выражение, элементами которого являются константы. Например, оператор «*mov dl,'A'-10*» помещает в регистр *DL* код *ASCII*-символа «*A*», уменьшенный на 10.

Введите эту программу в файл *Printaj.asm* и получите файл *Printaj.com*. Затем выполните программу. Добившись правильной ее работы, используйте *Debug* для разассемблирования программы. Посмотрите, как транслятор распределил память для основной программы и для процедуры. Если мы раньше перестраховывались, располагая процедуру подальше (по адресу 200*h*), то теперь транслятор расходует память экономно, не оставляя промежутков между процедурами.

13.4 Вывод на экран двузначного шестнадцатеричного числа

В гл. 6 подобная программа была разработана нами с помощью *Debug*. Теперь мы запишем ее на языке ассемблера, добавив небольшую процедуру, которая выводит один символ на экран. Выделение этой, на первый взгляд, лишней процедуры обусловлено желанием в дальнейшем вносить изменения в операцию вывода символа, совершенно не влияя на остальную часть программы.

На рис. 13.1 приведено *дерево подпрограмм* для данной программы. Эта структура показывает, какие подпрограммы (процедуры) входят в состав программы и как эти процедуры связаны между собой по управлению. При этом самой верхней (т. е. главной процедурой) является *Test_write_byte_hex*. Данная процедура предназначена для тестирования процедуры *Write_byte_hex*, выполняющей вывод на экран двузначного шестнадцатеричного числа, содержащегося в байте. В свою очередь, процедура *Write_byte_hex* вызывает (инициирует) процедуру *Write_digit_hex*, выполняющую вывод на экран шестнадцатеричной цифры. В ходе своей работы данная процедура вызывает процедуру *Write_char*, выполняющую вывод символа на экран.

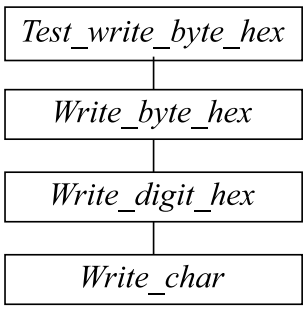


Рис. 13.1 – Дерево подпрограмм для программы вывода двузначного шестнадцатеричного числа

В гл. 6 приведены блок-схемы для процедур *Write_byte_hex* и *Write_digit_hex*.

Пример

В следующей программе тексты этих процедур опущены:

```
        [org 100h]
;
;       Тестирование процедуры Write_byte_hex
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Вызовы: Write_byte_hex
;
Test_write_byte_hex:
        mov dl,3fh              ; Тестировать с 3Fh
        call Write_byte_hex     ; Вывод числа
        int 20h                 ; Возврат в DOS
;
;       Вывод двузначного шестнадцатеричного числа
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: DL содержит выводимое число
;       Вызовы: Write_digit_hex
;
Write_byte_hex:
        . . . . . .     ; Тело процедуры Write_byte_hex
        ret
;
;       Вывод шестнадцатеричной цифры
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: BL содержит шестнадцатеричную цифру
;       Вызовы: Write_char
Write_digit_hex:
        . . . . . .     ; Тело процедуры Write_digit_hex
        ret
;
;       Вывод символа на экран
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: DL содержит код символа, выводимый на экран
;
        Write_char:
        push ax
        mov ah,2        ; Функция вывода символа
        int 21h         ; Вывод символа на экран
        pop ax
        ret
```

Контрольные вопросы по главе 13

1) Что такое исходная программа на ассемблере, как её создать?
2) Как получить машинную программу?
3) Для чего используются комментарии в программе?
4) Как дерево подпрограмм помогает понять структуру программы?
5) Как вывести на экран двузначное шестнадцатеричное число?

Глава 14

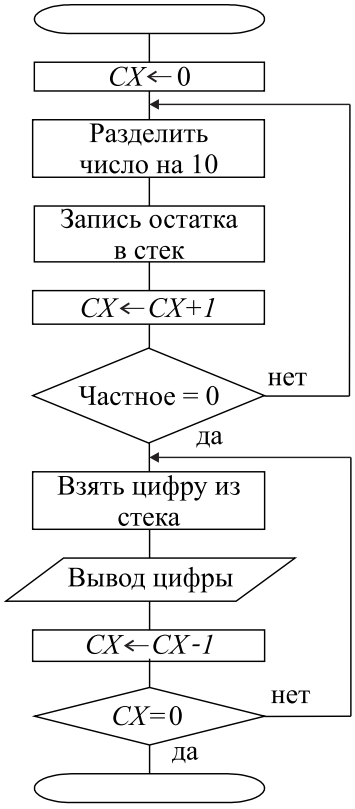
ВЫВОД НА ЭКРАН ДЕСЯТИЧНЫХ И ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

В процессе разработки практически важных процедур, выполняющих вывод на экран десятичных и шестнадцатеричных чисел, производится изучение нескольких новых исполнительных операторов ассемблера, расширяются навыки работы со стеком, а также производится знакомство с представлением исходной программы в виде нескольких файлов.

14.1 Получение алгоритма

Назовем процедуру, выполняющую вывод на экран десятичного представления слова данных, содержащего двоичное число без знака, как *Write_word_dec*. Пусть это слово данных передается на вход процедуры в регистре *DX*. Обсудим алгоритм процедуры *Write_word_dec*.

Предполагаемый результат работы процедуры состоит в том, что сначала на экран будет выведена старшая десятичная цифра числа, затем вторая слева цифра и т. д. Для этого следует иметь десятичное представление числа, хранящегося у нас в двоичном виде. Вспомним, что при делении числа на 10 остаток есть младшая десятичная цифра числа. Если полученное частное разделим на 10, то получим вторую справа десятичную цифру. Подобное деление можно продолжать до тех пор, пока очередное частное не будет меньше 10 и, следовательно, оно и будет равно старшей цифре числа. Если бы порядок получения десятичных цифр из числа совпал бы с порядком их вывода на экран, то задача была бы уже решена. Но у нас эти порядки противоположны. Поэтому мы опять обратимся за помощью к стеку. Вспомним его свойство «кто пришел первым, тот уйдет последним». Поэтому если мы поместим в стек младшую десятичную цифру числа (она получена первой), то она будет извлечена из стека для вывода на экран последней. Блок-схема соответствующего алгоритма приведена на рис. 14.1.



CX – счетчик десятичных цифр в числе

Рис. 14.1 – Алгоритм вывода десятичного числа

14.2 Дерево подпрограмм

Кроме самой процедуры *Write_word_dec*, выполняющей вывод на экран десятичного представления слова данных, наша программа включает и другие процедуры (рис. 14.2). Во-первых, это главная подпрограмма *Test_write_word_dec*, используемая для тестирования *Write_word_dec*. Во-вторых, это процедура вывода шестнадцатеричной цифры *Write_digit_hex* (она пригодна и для вывода десятичной цифры), а также процедура вывода символа *Write_char*.

Главная подпрограмма *Test_write_word_dec* тестирует процедуру *Write_word_dec* с помощью числа 12345 (которое транслятор переводит в слово 3039*h*).

Пример

```
Test_write_word_dec:
        mov dx,12345
        call Write_word_dec
        int 20h                 ; Возврат в DOS
```

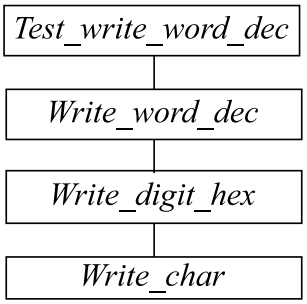


Рис. 14.2 – Дерево подпрограмм для программы вывода десятичного числа

14.3 Запись на ассемблере

При кодировании процедуры *Write_word_dec* можно использовать два следующих приема программирования. Хотя выгода от их применения небольшая, они широко используются на практике.

Целью первого приема является обнуление ячейки (слова или байта). Для этого используется логическая команда ***xor*** — «исключающее *ИЛИ*». Например, команда *«xor ax,ax»* обнуляет регистр *AX*. Она сравнивает оба операнда побитно, и если один из битов равен 1, то и в соответствующий бит результата записывается 1. Если оба сравниваемых бита содержат 0 или оба содержат 1, то в результирующий бит записывается 0. Например:

$$\begin{array}{rr} & 1011\,0101 \\ XOR & 1011\,0101 \\ \hline & 0000\,0000 \end{array}$$

Второй прием используется для того, чтобы проверить на равенство 0 слово или байт. Вместо команды «*cmp ax,*0» можно записать команду «*or ax,ax*», используя далее или команду условного перехода ***jne*** (от «*jump if not equal*» — перейти, если не равно) или ***jnz*** (перейти, если не нуль).

Логическая команда ***or*** («*ИЛИ*») побитно сравнивает оба операнда и записывает 1 в бит результата тогда, когда хотя бы в одном из двух сравниваемых битов есть 1. Данная команда устанавливает флаг нуля только тогда, когда все биты результата содержат 0.

Операция *OR*, выполненная по отношению к одному и тому же числу, дает в результате это же число:

$$\begin{array}{rr} & 1011\,0101 \\ OR & 1011\,0101 \\ \hline & 1011\,0101 \end{array}$$

Команда *or* также полезна при установке одного бита в 1. Например, мы можем установить бит 3:

$$\begin{array}{rr} & 1011\,0101 \\ OR & 0000\,1000 \\ \hline & 1011\,1101 \end{array}$$

Внесите изменения в полученный на предыдущей работе исходный файл *Video_io.asm*. Во-первых, с помощью текстового редактора удалите процедуру *Test_write_byte_hex* и на ее место запишите тестовую процедуру *Test_write_word_dec*.

Во-вторых, получите текст на ассемблере процедуры *Write_word_dec*, выполняющей вывод на экран содержимого 2-байтового регистра *DX* в виде десятичного числа. При этом для кодирования этапов алгоритма «*CX*←0» и «Частное = 0» следует использовать описанные выше два приема. Процедуру *Write_word_dec* добавьте в конец файла *Video_io.asm*.

Внеся изменения, ***проделайте*** с *Video_io.asm* все необходимые шаги, чтобы получить *com*-файл. После этого протестируйте программу. В случае ошибки проверьте исходный файл. Если это не поможет, то для поиска ошибки используйте *Debug*.

Выполняя тестирование, будьте осторожны при проверке граничных условий. Первое граничное условие 0 не представляет трудности. Другое граничное условие — 65535 (*FFFFh*), которое вам лучше проверять с помощью *Debug*. ***Загрузите*** *Video_io.com* в *Debug*, напечатав «*Debug Video_io.com*», и замените 12345 (*3039h*) по адресу 101*h* и 102*h* на 65535 (*FFFFh*). Напомним, что *Write_word_dec* работает с беззнаковыми числами.

14.4 Многофайловая исходная программа

Для выполнения тестирования процедуры *Write_word_dec* нам пришлось одну тестовую процедуру в файле *Video_io.asm* заменить другой, которая после завершения тестирования также больше не нужна. Очень неудобно вносить изменения в файл с готовыми (или почти готовыми процедурами) с целью добавления в него процедур временного назначения. Выход заключается в размещении тестирующей процедуры в отдельном исходном файле. Когда данная процедура становится ненужной, соответствующий файл уничтожается. Другой причиной размещения исходной программы в нескольких файлах является большой объем программы.

Далее будем понимать под *файловой структурой исходной программы* перечень файлов, содержащих текст программы на языке программирования. А также перечень процедур, входящих в состав каждого файла. Кроме процедур, файлы могут содержать *переменные* — области данных, которым в исходной программе присвоены символьные имена. Примеры применения переменных встретятся нам в последующих работах.

Следует отметить, что в отличие от дерева подпрограмм файловая структура не влияет на машинную программу. Данная структура создается для исходной программы с целью обеспечения удобства в программировании. Нескольким программам, имеющим разные файловые структуры, может соответствовать одна и та же результирующая машинная программа.

При выполнении данной работы мы добавили в файл *Video_io.asm* короткую тестовую процедуру *Test_write_word_dec*. Теперь уберем ее оттуда и поместим в собственный отдельный файл *Test.asm*. В результате получим файловую структуру программы, приведенную на рис. 14.3.

Для того чтобы транслятор-ассемблер объединил несколько исходных файлов в единый текст исходной программы, каждый «родительский» исходный файл должен содержать псевдооператоры ***%include*** («включить»), в качестве операндов которых записаны имена «дочерних» файлов, заключенные в кавычки. Встретив опе-

ратор *include*, транслятор заменяет его на текст «дочернего» исходного файла, указанного в этом операторе.

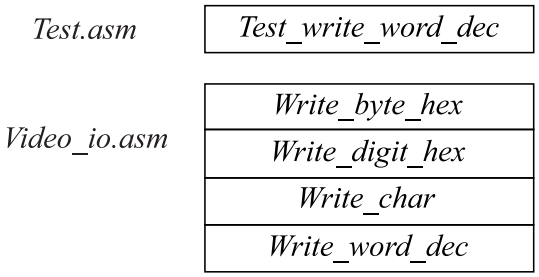
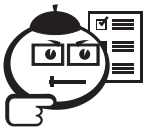


Рис. 14.3 – Файловая структура исходной программы

Пример

Рассмотрим эти псевдооператоры на примере файла *Test.asm*:

```
        [org l00h]
Test_write_word_dec:
        mov dx,12345
        call Write_word_dec
        int 20h                    ; Возврат в DOS
        %include 'Video_io.asm'    ; Включение файла Video_io.asm
```



Запишите файл *Test.asm* на диск и внесите в файл *Video_io.asm* следующие изменения:

1) удалите из *Video_io.asm* процедуру *Test_write_word_dec*, т. к. мы поместили ее в файл *Test.asm*;
2) удалите из *Video_io.asm* выражение *[org l00h]*, т. к. мы перенесли его в файл *Test.asm*, который теперь содержит главную процедуру программы.

Контрольные вопросы по главе 14

1) В чем заключается алгоритм вывода на экран десятичного числа?
2) Можно ли с помощью этого алгоритма вывести на экран число, например в двоичном виде?
3) Что такое файловая структура исходной программы?
4) Как транслятор-ассемблер объединяет несколько исходных файлов в одну программу?
5) Что такое многофайловая исходная программа?

Глава 15

ДАМПИРОВАНИЕ ШЕСТНАДЦАТИ БАЙТОВ

Разработку программы, выполняющей дампирование, целесообразно начать с создания процедуры, выполняющей вывод на экран содержимого шестнадцати байтов. Именно столько информации удобно разместить на одной строке экрана. Назовем данную процедуру *Disp_line*. Для каждого из 16-ти байтов она выводит на экран соответствующее шестнадцатеричное представление, разделяя два соседних числа пробелами.

Пример

Ниже приведен файл *Disp_sec.asm*, который содержит начальную версию процедуры *Disp_line*:

```
            [org l00h]
            jmp     Disp_line
;   Данные программы
Sector      db      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h     ; Образец
            db      18h, 19h, 1Ah, 1Bh,1Ch,1Dh,1Eh,1Fh        ; текста
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;     Процедура дампирует 16 байт памяти в одну строку
;     шестнадцатеричных чисел
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; Вызовы:     Write_hex, Write_char
; Чтение:     Sector
;
Disp_line:
            xor     bx,bx                   ; Обнуление BX
```

```
        mov     cx,16                   ; Счетчик байтов
.M:     mov     dl, [Sector+bx]         ; Получить один байт
        call    Write_byte_hex          ; Вывод шестн. числа
        mov     DL, ' '                 ; Вывод на экран
        call    Write_char              ; пробела
        inc     bx                      ; Возврат за следующим
        loop    .M                      ; байтом
        int     20h                     ; Возврат в DOS
        %include    'Video_io.asm'      ; Подсоединение процедур
```

. . . . . .

В начале программы находится оператор безусловного перехода, осуществляющий переход через область данных, расположенную после оператора перехода, на начало области исполнительных операторов программы. Для исполнительных операторов новым является использование относительной регистровой адресации в операторе:

Hex_loop: mov dl, [Sector+bx]

Байт программы, отмеченный меткой *Sector*, имеет смещение 259 (100*h* + 3) относительно самого первого байта сегмента памяти, выделенного программе (команда *jmp* имеет длину три байта). Тогда при выполнении на ЦП машинной команды *mov*, соответствующей записанному оператору, реальный адрес второго операнда будет определен по формуле:

$$R = (DS) * 16 + 259 + (BX).$$

Задав нулевое содержимое регистра *BX*: (*BX*) = 0, мы получим адрес байта с меткой *Sector*, в который транслятором было помещено число *l*0*h*. Меняя содержимое регистра *BX* от 0 до 15, мы можем с помощью приведенного выше оператора *mov* записать в регистр *DL* содержимое любого из 16-ти байтов, проинициализированных по нашей просьбе транслятором.

Запишите файл *Disp_sec.asm*, а затем получите файл *Disp_sec.com*. Если после запуска программы вы не увидите:

10 11 12 13 14 15 16 17 18 19 IA 1B 1C 1D 1E 1F,

то вернитесь назад и найдите ошибку.

15.1 Дампирование 256 байтов памяти

После того, как мы сделали первую версию процедуры вывода на экран шестнадцатеричного представления 16-и байтов памяти, перейдем к изготовлению программы, выполняющей дамп 256 байтов памяти.

Число 256 обусловлено следующим. Во-первых, дальнейшей целью наших лабораторных работ является создание программы, позволяющей выполнять редактирование текстовой информации, находящейся в сегменте памяти. Во-вторых, объем сегмента равен 65536 байт и 256 есть наибольшее число, которое одновременно отвечает двум требованиям:

1) является делителем числа 65536;
2) содержимое 256 байтов одновременно умещается на экране. Далее будем называть такой объем памяти *сектором*. Поэтому назовем процедуру, выполняющую дамп 256 байтов, как *Disp_sector*.

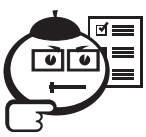
До сих пор мы делали вывод символов, которые размещались на одной строке экрана. При выполнении дампа 256 байтов нам потребуются 16 строк. Поэтому потребуется выполнять переход с одной строки экрана на следующую строку. Для осуществления такого перехода достаточно «вывести» на экран два управляющих символа кода *ASCII*. Первый символ имеет код 0*Dh* и называется «возврат каретки». В результате его «вывода» последующий вывод на экран начнется с самой левой позиции строки. Второй управляющий символ имеет код 0*Ah* и называется «перевод строки». Его «вывод» и реализует перевод строки экрана.

Назовем процедуру, выполняющую перевод строки, как *Send_crlf* (*crlf* означает «*Carriage Return — Line Feed*.» — «Возврат каретки — Перевод строки»). Текст файла *Cursor.asm*, содержащего эту процедуру:

```
                %define    cr      13              ; Возврат каретки
                %define    lf      10              ; Перевод строки
;
;                Перевод строки экрана
;               - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
Send_crlf:
                push     ax
                push     dx
                mov      ah,2       ; Функция вывода
                mov      dl,cr      ; Выводимый символ
                int      21h        ; Вывод символа
                mov      dl,lf
                int      32h ; - - - // - - -
                pop      dx
                pop      ax
                ret
```

Псевдооператор *%define* указывает транслятору, что имя *cr* эквивалентно числу 13, а имя *lf* эквивалентно числу 10. Поэтому везде в программе, где транслятор встретит данное имя, он заменит его указанным числом. Применение данного псевдооператора позволяет программисту записывать в программе вместо чисел более удобные их символьные обозначения. Например, вместо 13 записывать *cr*.

На рис. 15.1 приведены дерево подпрограмм и файловая структура программы вывода на экран 256 байтов из переменной *Sector*. Переменная *Address* изображена пунктиром, т. к. в первой версии программы она не используется.



Введите файл *Cursor.asm* и скорректируйте файл *Disp_sec.asm* в соответствии с его новым вариантом.

```
                [org l00h]
                jmp Disp_sector
;       Данные программы
Sector          times 16        db 10h          ; Первая строка байтов
                times 16        db 11h
        . . . . . .
                times 16        db 1Fh          ; Последняя строка байтов
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
;       Отображает на экран сектор (256 байт)
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Вызовы: Disp_line, Send_crlf
;
Disp_sector:
                xor     dx,dx           ; Начало Sector
                mov     cx,16           ; Число строк 16
.M:             call            Disp_line ; Вывод строки
                call    Send_crlf       ; Перевод строки
                add     dx,16           ; Номер следующего байта
                loop    .M              ; Проверка числа строк
                int     20h             ; Выход в DOS
;
;       Процедура дампирует 16 байт памяти в одну строку шестнадцат. чисел
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы:      DX — номер первого байта строки в Sector
;       Вызовы:     Write_char, Write_hex
;       Читается : Sector
;
Disp_line:
                push    bx
                push    cx
                push    dx
                mov     bx,dx                   ; В BX номер первого байта
                mov     cx, 16                  ; Счетчик байтов
.M:             mov     dl, Sector[bx]          ; Получить один байт
                call    Write_byte_hex          ; Вывод шестнад. числа
                mov     dl, ' '                 ; Вывод на экран
                call    Write_char              ; пробела
                inc     bx                      ; Возврат за
                loop    .M                      ; следующим байтом
                pop     dx
                pop     cx
                pop     bx
                ret
                %include        'Vide_io.asm' ; Подсоединение процедур
                %include        'Cursor.asm'  ; –//–
```

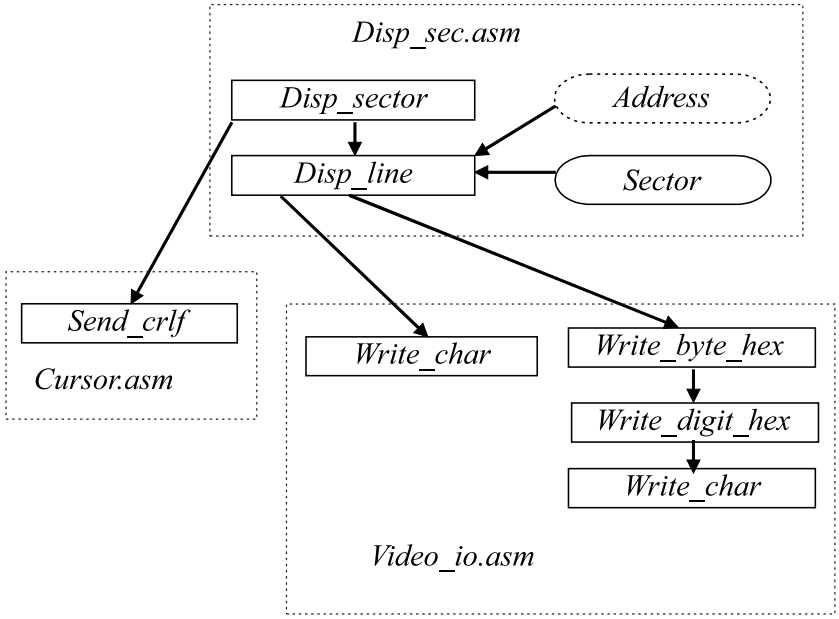
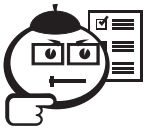


Рис. 15.1 – Дерево подпрограмм и файловая структура программы вывода на экран

Обратите внимание на изменения, которые мы внесли в процедуру *Disp_line*. Теперь она имеет входной параметр, передаваемый в регистре *DX*. Это номер байта в поле *Sector*, начиная с которого следует вывести на экран 16 байтов. Есть и другие изменения в данной процедуре, не требующие пояснений.



Выполните ассемблирование файла *Disp_sec*.

Пример

Выполнив программу, вы должны получить изображение на экране:

10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
. . .
1 F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F

15.2 Очистка экрана

Прежде чем вывести на экран сектор, наша программа должна позаботиться об очистке экрана. Для выполнения этой операции мы обратимся за помощью к системной программе, называемой *BIOS*.

Операционная система, например *DOS*, использует подпрограммы *BIOS* для выполнения своих операций обмена с внешними устройствами. Разработчик прикладной программы может применять подпрограммы *BIOS* точно так же, как и подпрограммы *DOS*, т. е. с помощью команд *int*. Подпрограммы *BIOS* применяются в двух случаях:

1) соответствующая функция *DOS* отсутствует;

2) требуется повысить скорость выполнения функции.

Для очистки экрана будем использовать функцию *BIOS* — «Прокрутка экрана вверх». Она вызывается оператором «*int 10h*» (функция 6). Данная функция требует задания следующих входных параметров:

(*AL*) — число строк, которые должны быть стерты внизу окна. Обычная прокрутка стирает одну строку. Нуль означает, что нужно стереть все окно;

(*CH*, *CL*) — строка и столбец левого верхнего угла окна;

(*DH*, *DL*) — строка и столбец правого нижнего угла окна;

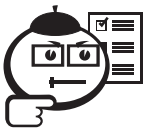
(*BH*) — атрибуты (например, цвет), используемые для этих строк.

Таким образом, функция номер 6 десятого прерывания требует довольно много входной информации, даже если все сводится только к очищению экрана. Отметим, что она обладает мощными способностями: может очистить любую прямоугольную часть экрана — окно («*Window*»).

Пример

Текст процедуры *Clear_screen*, выполняющей очистку экрана:

```
;           Очистка экрана
;           - - - - - - - - - - - - - - - - - - -
Clear_screen:
        push    ax
        push    bx
        push    cx
        push    dx
        xor     al, al      ; Очистить все окно
        xor     cx, cx      ; Верхний левый угол в (0,0)
        mov     dh, 24      ; Нижняя строка экрана -24
        mov     dl, 79      ; Правая граница в 79 столбце
        mov     bh, 7       ; Применить нормальные атрибуты
        mov     ah, 6       ; Очистить
        int     10h         ; окно
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
```



Запишите процедуру *Clear_screen* в файл *Cursor.asm* и протестируйте ее. Затем скорректируйте файл *Disp_sec.asm*, записав в начале процедуры *Disp_sector* оператор вызова процедуры очистки экрана.

Запустив программу *Disp_sec.com*, можно убедиться, что экран очищается на весьма короткое время. Это обусловлено тем, что после завершения *Disp_sec.com* управление возвращается в *DOS*, которая восстанавливает на экране свою информацию. Для избежания этого в конце процедуры *Disp_sector* поместите оператор, выполняющий ввод символа с клавиатуры. Наличие такого оператора позволит наблюдать на экране неискаженное изображение сектора до тех пор, пока не будет нажата любая клавиша. Впоследствии у нас отпадет потребность в подобном «торможении» нашей программы.

Контрольные вопросы по главе 15

1) Что такое дампирование памяти?
2) Как определить реальный адрес данных в секторе?
3) Для чего используются циклы?
4) Для чего используется очистка экрана?
5) Насколько важным является сохранение значений регистров, например в стеке?

Глава 16

ПЕРЕПИСКА СЕКТОРА ПАМЯТИ

Разработанная в предыдущей главе программа выполняет вывод на экран единственного 256-байтового сектора, заполняемого транслятором до начала выполнения программы. Теперь пришла пора заняться выводом на экран любого сектора, входящего в состав того сегмента ОП, который выделен нашей программе. Напомним, что размер одного сегмента ОП — 64*K*, что соответствует 256 сегментам по 256 байт.

16.1 Функции переписки сектора

В полученной ранее программе мы использовали для хранения редактируемого сектора (256 байт) область (переменную) *Sector*. В принципе можно было бы не использовать эту область, а напрямую работать с требуемым сектором памяти. Но в этом случае любая ошибка в редактировании мгновенно отразилась бы на оригинале. Поэтому мы будем копировать текущий сектор памяти в область *Sector*, а после выполнения редактирования осуществим обратное копирование.

На рис. 16.1 приведено дерево подпрограмм для программы, выполняющей различные виды функций переписки сектора. Набор этих функций определяется потребностями пользователя разрабатываемой программы. Каждая из функций программно реализуется одной из следующих процедур:

- *Write_sector* — переписывает содержимое переменной *Sector* в область памяти, начальный адрес которой находится в переменной *Address*;
- *Init_sector* помещает в переменную *Sector* начальный сектор сегмента памяти нашей программы. Кроме того, эта процедура помещает 0 в переменную *Address*;
- *Prev_sector* помещает в *Sector* предыдущий сектор памяти, а в переменную *Address* записывает начальный адрес этого сектора;

- *Next_sector* помещает в переменные *Sector* и *Address* соответственно содержимое и начальный адрес следующего сектора памяти;
- *N_sector* загружает в область *Sector N*-й сектор сегмента программы. При этом значение N $(0 \leqslant N \leqslant 255)$, умноженное на 256, записывается в переменную *Address*.

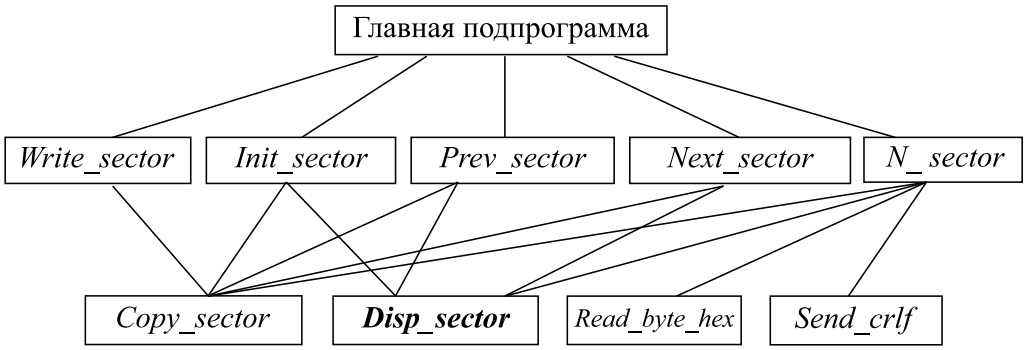


Рис. 16.1 – Дерево подпрограмм для программы, выполняющей функции переписки сектора

16.2 Копирование сектора

Как видно из рис. 16.1, все процедуры, выполняющие функции по переписке сектора, пользуются услугами процедуры *Copy_sector*, которая записывает содержимое заданного сектора памяти в качестве содержимого другого заданного сектора. *Copy_sector* имеет два входных параметра: регистр *SI* содержит адрес (внутрисегментное смещение) сектора-источника, *DI* — адрес-смещение сектора-приемника.

Пример

При написании процедуры *Copy_sector* мы могли бы ограничиться уже известными нам операторами, но мы привлечем для этого ряд новых операторов:

```
;               Копирует сектор (256 байт)
;               - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: SI — начальный адрес сектора-источника
;               DI — начальный адрес сектора-приемника
;
Copy_sector:
                push    cx
                push    f               ; Сохранить флаг направления DF
                cld                     ; Сбросить этот флаг
                mov     cx, 256         ; В счетчике число байт
        rep     movsb                   ; Пересылка цепочки байт
```

```
popf                    ; Восстановить флаг DF
pop     cx
ret
```

. . . . . .

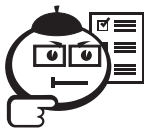
Строковый (цепочечный) оператор ***movsb*** занимает центральное место в *Copy_sector*. Взятый без префикса ***rep***, он выполняет:

1) пересылку байта из ячейки памяти с адресом в регистре *SI* в ячейку памяти, адрес которой находится в регистре *DI*;
2) изменяет на единицу адреса в регистрах *SI* и *DI*.

Направление изменения (увеличение или уменьшение) адресов в *SI* и *DI* определяется значением флага направления *DF* в регистре флагов. Если *DF* = 0, то адреса увеличиваются, а если *DF* = 1, то уменьшаются. Сброс флага *DF* выполняет специальный оператор ***cld***, а установку — оператор ***std***. Так как флаг направления может использоваться и в процедуре, вызывающей нашу процедуру, то целесообразно сохранить его перед изменением в стеке, а затем восстановить его оттуда. Для этого используются операторы ***pushf*** и ***popf***, выполняющие запись в стек и извлечение оттуда регистра флагов.

Префикс повторения команды ***rep*** многократно усиливает мощность строковой команды. При этом оператор *movsb* выполняется столько раз, каково содержимое регистра *CX*. При каждом выполнении содержимое *CX* уменьшается на единицу. Как только это содержимое станет равным нулю, то выполняется следующий за *movsb* оператор.

Перед вызовом процедуры *Copy_sector* необходимо в вызывающей ее программе загрузить в регистры *SI* и *DI* начальные адреса секторов памяти. Для этого можно использовать оператор *mov*. Например, в результате оператора «*mov si,Sector*» адрес самого первого байта области *Sector* будет помещен в регистр *SI*.



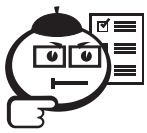
. . . . . .

Запишите процедуру *Copy_sector* в файл *Disp_sec.asm*.

. . . . . .

16.3 Алгоритмы процедур

На рис. 16.2 приведён алгоритм процедуры *Init_sector*, на рис. 16.3 алгоритм *Prev_sector*, а на рис. 16.4 алгоритм *N_sector*. Что касается алгоритма процедуры *Next_sector*, то он очень похож на алгоритм *Prev_sector*. Отличие состоит в том, что номер текущего сектора *N* сравнивается не с 0, а с числом *FFh*.



. . . . . .

Запишите тексты процедур *Write_sector*, *Init_sector*, *Prev_sector*, *Next_sector*, *N_sector* в файл *Disp_sec.asm*.

. . . . . .

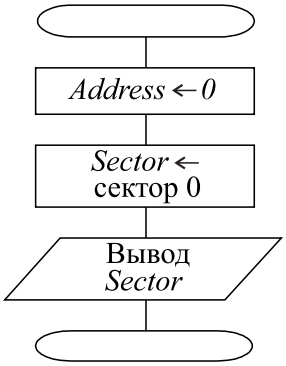
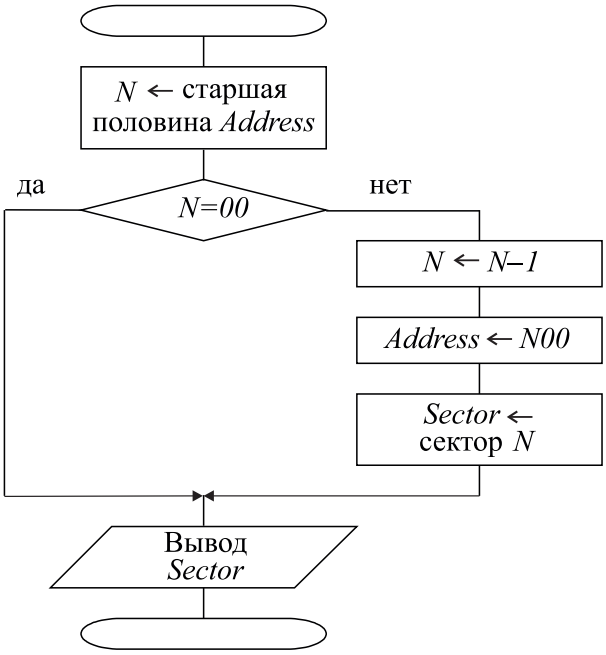


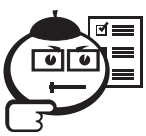
Рис. 16.2 – Алгоритм процедуры *Init_sector*



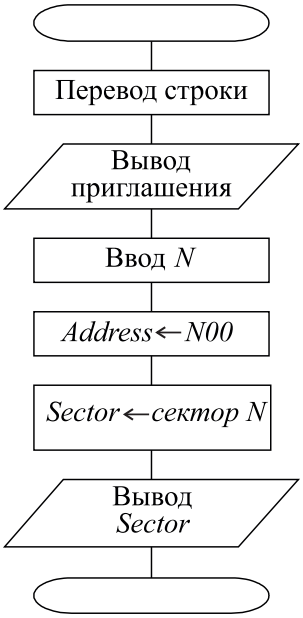
N – номер текущего сектора

Рис. 16.3 – Алгоритм процедуры *Prev_sector*

Реализация этапа «Ввод *N*» в процедуре *N_sector* осуществляется путём вызова процедуры *Read_byte_hex*, выполняющей ввод с клавиатуры двузначного шестнадцатеричного числа. *Read_byte_hex*, в свою очередь, вызывает процедуру *Read_digit_hex*, выполняющую ввод шестнадцатеричной цифры. Алгоритмы обеих процедур были рассмотрены нами ранее в гл. 9.



Запишите тексты этих процедур в новый файл *Kbd_io.asm*.



N – номер сектора (двузначное шестнадцатеричное число)

Рис. 16.4 – Алгоритм процедуры *N_sector*

Контрольные вопросы по главе 16

1) Для чего используется дерево подпрограмм?
2) Как применяются операторы pushf, popf, для чего используются?
3) Какое значение имеет флаг DF?
4) Как ввести номер сектора?
5) Как вывести переписанный сектор на экран?

Глава 17

ДИСПЕТЧЕР КОМАНД

В процессе выполнения работы решается задача разработки диспетчера — программного модуля, обеспечивающего диалог с пользователем и выполняющего координацию работы других модулей программы (текстового редактора). Применяемый при этом подход может быть использован для построения диспетчеров в других программных системах.

17.1 Ввод команд

Почти любая полноценная программа является интерактивной, т. е. способной вести диалог со своим пользователем. Естественно, что наш редактор текстовой информации также должен быть интерактивным. Он должен воспринимать команды пользователя, набираемые на клавиатуре, и выводить на экран ответные сообщения. При этом принято называть сообщения пользователя (например, команды) *входными сообщениями*, а сообщения программы — *выходными*.

Распознавание команд пользователя поручим программному модулю (процедуре), называемому *диспетчером команд*. Данный модуль занимает центральное место в программной системе и координирует работу других модулей.

Допустим, что наш редактор выполняет всего шесть команд, каждой из которых соответствует своя управляющая клавиша:

<*F1*> — вывести на экран предыдущий сектор (256 байт) ОП;

<*F2*> — записать скорректированный сектор в память;

<*F3*> — вывести на экран следующий сектор ОП;

<*F4*> — вывести на экран начальный сектор ОП;

<*F5*> — вывести на экран сектор N, где $0 \leqslant N \leqslant 256$;

<*F10*> — закончить работу редактора.

Особенностью этих и многих других управляющих клавиш является то, что им соответствует не обычный, а расширенный код *ASCII*. В расширенном коде каждое из 256 кодовых значений может иметь наряду с обычной интерпретацией второе

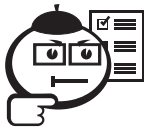
смысловое значение. Например, код *ASCII 3Bh* обозначает символ «;», но этот же код соответствует и клавише *<F1>*. Аналогично код 44*h* соответствует и символу *D*, и клавише *<F10>*. Таким образом, в расширенном коде *ASCII* предельное количество «символов» не 256, а 512.

Как разделить обычный и расширенный коды *ASCII*, чтобы не было путаницы? Допустим, что для ввода символа мы обращаемся к *DOS* с помощью команды *«int 21h»*. В результате будет вызвана подпрограмма *DOS*, ожидающая нажатия клавиши. Если мы нажмем «обычную» клавишу, то данная подпрограмма передаст нашей программе в регистре *AL* соответствующий код *ASCII*. Если же мы нажмем управляющую клавишу, например *<F1>*, то в *AL* будет возвращен 0. Если мы выполним команду *«int 21h»* повторно, то в *AL* будет возвращен код *ASCII*, соответствующий управляющему символу.

Ниже приведен текст процедуры *Read_byte*, выполняющей ввод с клавиатуры любого символа — обычного или расширенного. При этом код *ASCII* символа возвращается в регистре *AL*, а в регистре *AH* указывается тип символа (1 — обычный символ, −1 — символ расширенного кода). Обратите внимание, что вывод «эха» символа не производится.

Пример

```
;              Считывает код символа с клавиатуры
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;     Выход: AL — код ASCII символа
;              AH — тип символа (1 — обычный символ; −1 — символ
;              расширенного кода)
;
Read_byte:
               mov     ah,7        ; Ввод символа
               int     21h         ; без эха
               or      al,al       ; Расширенный код ?
               jz      Extended    ; Да
               mov     ah,1        ; Обычный символ jmp Exit
Extended:      mov     ah,7        ; Ввод расширенного
               int     21h         ; кода
               mov     ah,0ffh     ; AH ← −1
Exit:     ret
```



Поместите процедуру *Read_byte* в файл *Kbd_io.asm*, а затем выполните ее отладку, используя *Debug*. Для этого получите файл *Kbd_io.com* и наберите команду *DOS*:

```
DEBUG Kbd_io.com
```

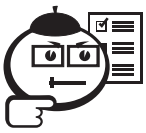
Для запуска программы используйте команду *Debug*: *«G i»*, где *i* — адрес в листинге команды *ret*. Нажимая различные клавиши, проверьте правильность заполнения процедурой регистра *AX*.

17.2 Алгоритм диспетчера

На рис. 17.1 приведена блок-схема процедуры *Dispatcher*, выполняющей совместно с рассматриваемой далее процедурой *Command* функции диспетчера команд. Для того, чтобы обеспечить структурность алгоритма, мы, как и в одной из предыдущих работ, используем флаг переноса *CF* и операции над ним.

Кодирование процедуры *Dispatcher* не представляет особого труда. Три этапа ее алгоритма реализуются путем вызова ранее разработанных процедур. Этап «Вывод приглашения» реализуется путем вывода на экран строки символов или, даже, всего одного-двух символов, однозначно указывающих на то, что редактор ожидает команд пользователя. Два этапа — «Редактирование символа» и «Выполнение команды» реализуются путем вызова соответственно процедур *Edit_byte* и *Command*, которых у нас пока нет.

Процедура *Edit_byte* не имеет выходных параметров и имеет единственный входной параметр: регистр *DL* содержит новый код *ASCII* редактируемого символа. Разработка данной процедуры выходит за рамки данной работы и поэтому мы вынуждены заменить ее «заглушкой». Процедура — заглушка имеет то же имя и те же параметры, что и настоящая процедура. Но в отличие от нее «заглушка» не имеет или почти не имеет внутренних операторов.



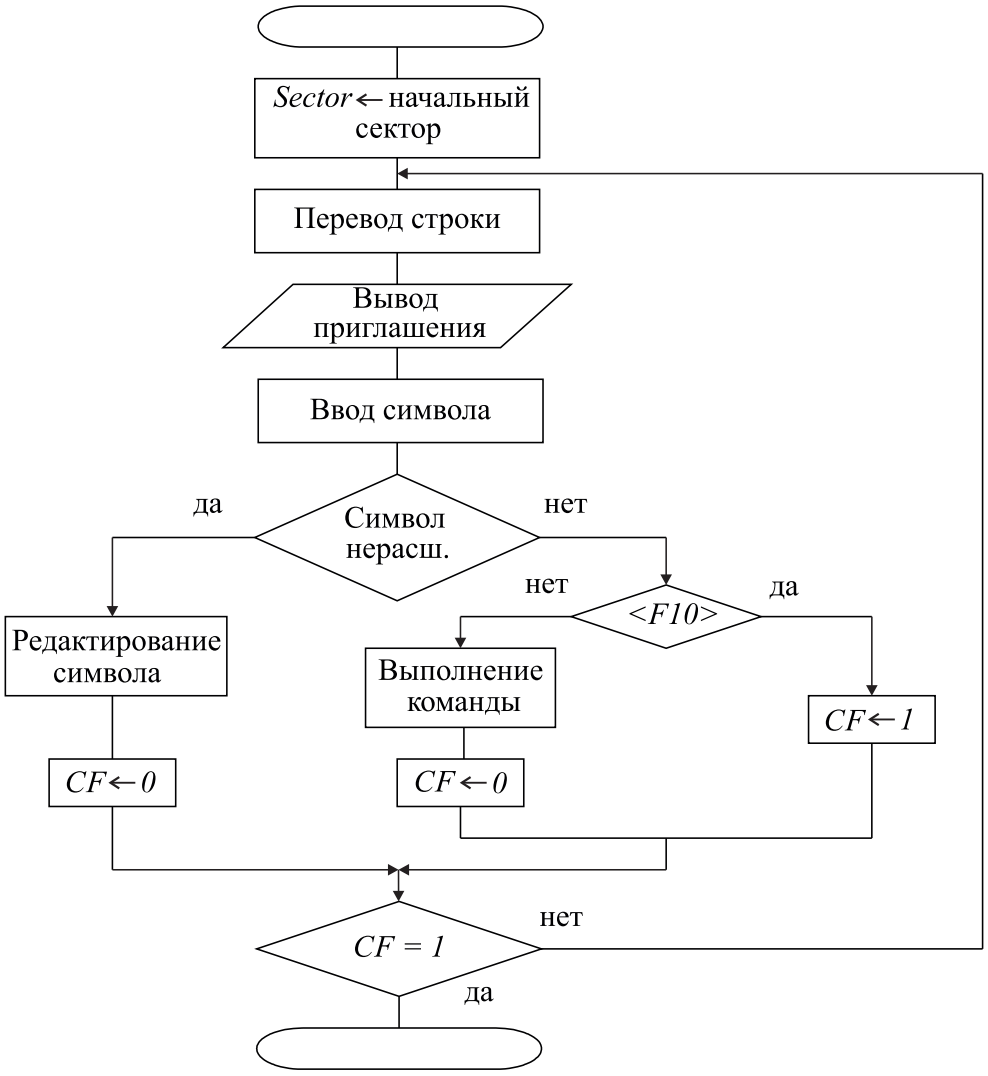
Запишите процедуру-заглушку *Edit_byte* в файл *Disp_sec.asm*.

17.3 Выполнение команды

Этап «Выполнение команды» реализуется путем вызова процедуры *Command*. Данная процедура получает в регистре *DL* код *ASCII*, соответствующий команде, и в зависимости от этого кода вызывает процедуру, выполняющую заданное командой действие.

На рис. 17.2 приведена блок-схема процедуры *Command*. Центральной особенностью ее алгоритма является использование таблицы переходов. В данной таблице для каждой разрешенной команды пользователя отводятся три байта. В первом байте находится код *ASCII*, соответствующий команде, а в двух других байтах — начальный адрес (внутрисегментное смещение) соответствующей процедуры. В последнем байте таблицы находится 0.

Процедура *Command* совместно с процедурой *Dispatcher* обеспечивают выполнение алгоритма диспетчера команд. Поэтому обе процедуры, а также таблицу переходов *Table* мы поместим в один и тот же файл *Dispatch.asm*.



Sector – буфер для редактирования сектора

CF – признак завершения (*0* – продолжить, *1* – окончить работу)

Рис. 17.1 – Алгоритм процедуры *Dispatcher*

Пример

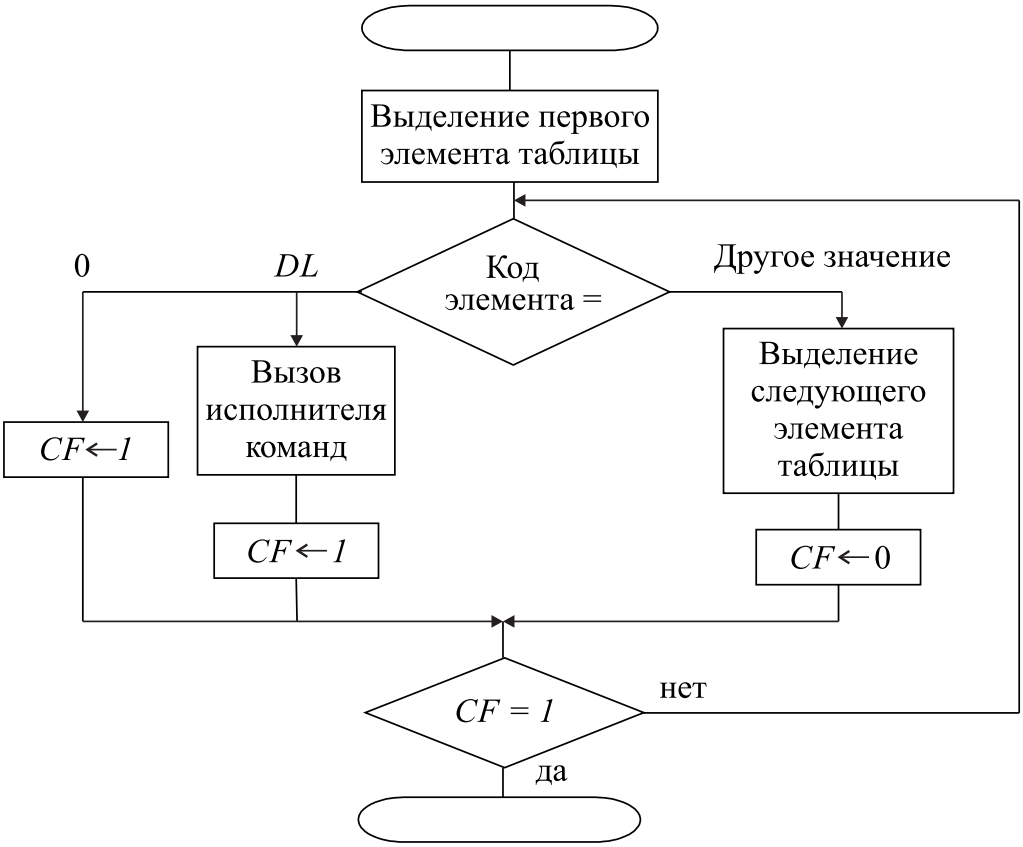
Заключительный фрагмент этого файла, содержащий модули *Command* и *Table*:

```
        [org 100h]
;
;       Координирует выполнение модулей редактора
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
Dispatcher:
    . . . . . . . . ; Тело процедуры Dispatcher
        ret
```

```
;
;          Интерпретатор команд
;          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Входы: DL — код ASCII, соответствующий команде
;       Чтение: Table — таблица переходов
;
Command:
                push    bx
                mov     bx,Table            ; BX←адрес таблицы
M1:             cmp BYTE PTR [BX], 0        ; Конец таблицы ?
                je      Exit                ; Да, кода нет в таблице
                cmp     dl,[bx]             ; Это вход в таблицу?
                je      Dispatch            ; Да, выполнить команду
                add     bx,3                ; Нет, переход к
                clc     ; следующему
                jmp     M2                  ; элементу таблицы
Dispatch:       inc     bx                  ; Адрес процедуры
                call    [bx]                ; Вызов процедуры
Exit:           stc     ; CF←1
M2:             jnc                 M1 ; Повторение для нового элемента таблицы
                pop     bx
                ret
;       Таблица содержит разрешенные расширенные ASCII-коды и
;       адреса процедур, выполняющих соответствующие команды
Table           db      3bh         ; <F1>
                dw      Prev_sector
                db      3ch         ; <F2>
                dw      Copy_sector
                db      3dh         ; <F3>
                dw      Next_sector
                db      3eh         ; <F4>
                dw      Init_sector
                db      3fh         ; <F5>
                dw      N_sector
                db      0           ; Конец таблицы
```

. . . . . .

Рассмотрим особенности этих модулей, обусловленные применением новых псевдооператоров *PTR* и *OFFSET*. Псевдооператор *PTR* используется для уточнения типа данного, обрабатываемого инструкцией программы. Обратите внимание, что таблица *Table* содержит и байты, и слова. Так как мы работаем с двумя типами данных, то мы должны указать транслятору-ассемблеру, какой тип данных используется при применении инструкций *CMP* или *CALL*. В случае если инструкция будет написана следующим образом: *CMP [BX], 0*, то ассемблер не поймет, что мы хотим сравнивать — слова или байты.



DL – код команды
CF – признак завершения (*0* – продолжить, *1* – завершить работу)

Рис. 17.2 – Алгоритм процедуры *Command*

Но если мы запишем инструкцию в виде: *CMP BYTE PTR[BX], 0*, то ассемблеру станет ясно, что *BX* указывает на байт и что мы хотим сравнивать байты. Аналогично инструкция *CMP WORD PTR[BX], 0* будет сравнивать слова. С другой стороны, инструкция *CMP AL, [BX]* проблем не вызывает, т. к. *AL* — это байтовый регистр и ассемблер понимает без разъяснений, что мы хотим сравнивать байты.

Кроме того, как Вы помните, инструкция *CALL* может быть либо типа *NEAR*, либо типа *FAR*. Для задания адреса «близкого» вызова («*NEAR CALL*») требуется одно слово, в то время как для «дальнего» вызова требуется два слова. Например, инструкция *CALL WORD PTR[BX]* посредством «*WORD PTR*» говорит транслятору, что [*BX*] указывает на одно слово, поэтому ассемблер будет генерировать близкий вызов и использовать то слово, на которое указывает [*BX*], как адрес, который мы записали в *Table*. (Для дальнего вызова, который использует адрес, состоящий из двух слов, мы должны были бы использовать инструкцию: *CALL DWORD PTR[BX]*, где *DWORD* означает «*Double Word*» — двойное слово.)

Для того, чтобы получить адреса процедур, вызываемых диспетчером и содержащихся в таблице, мы применим новый псевдооператор *OFFSET*. Строка

DW OFFSET CGROUP: Prev_sector

сообщает транслятору о необходимости применить смещение процедуры *Prev_sector*. Это смещение подсчитывается относительно начала группы *CGROUP*, и именно поэтому необходимо поставить «*CGROUP:*» перед именем процедуры. Если бы мы не поместили там *CGROUP*, то ассемблер подсчитывал бы адрес *Prev_sector* относительно начала сегмента кода, а это не совсем то, что нужно. (В данном конкретном случае *CGROUP* не необходим, т. к. в нашей программе сегмент кода загружается первым. Тем не менее для ясности мы будем везде писать «*CGROUP:*».)

Применение для реализации диспетчера команд таблицы переходов значительно увеличивает его гибкость, т. е. пригодность к модификации. В самом деле, мы легко можем добавить в нашу программу новые команды, набираемые на клавиатуре. Для этого достаточно записать процедуру, выполняющую новую команду, в подходящий файл и поместить новую точку входа в *Table*.

Контрольные вопросы по главе 17

1) Что такое расширенный код ASCII, когда он применяется?
2) Для чего используется оператор PTR?
3) Для чего используется оператор OFFSET?
4) Может ли диспетчер содержать другие команды?
5) Как поместить новую команду в диспетчер команд?

Глава 18

РАЗДЕЛЬНАЯ ТРАНСЛЯЦИЯ ПРОГРАММЫ

18.1 Получение прикладной программы

Преобразование виртуальной прикладной программы, записанной на языке программирования, в реальную машинную программу выполняет системная обрабатывающая программа, называемая лингвистическим процессором. На самом деле для такого преобразования требуются три лингвистических процессора — транслятор, редактор связей и загрузчик. В принципе, их можно объединить в единую программу, которая, естественно, также будет лингвистическим процессором. Именно так устроены современные *среды программирования*, позволяющие «выполнять» виртуальные программы. Кроме перечисленных программ, они могут включать также текстовый редактор для набора программ, а также отладчик. Вне зависимости от того, используются ли перечисленные лингвистические процессоры и утилиты по отдельности или в виде модулей среды программирования, общая схема получения и выполнения прикладной программы остается прежней (рис. 18.1).

На первом этапе в диалоге с текстовым редактором программист вводит в память ВС текст виртуальной программы, называемой также *исходной программой*. Для удобства программиста исходная программа может быть записана в виде не одного, а нескольких текстовых файлов, называемых *исходными модулями*. Программист сам определяет, как разбить свою программу на модули. Более того, он может программировать свои исходные модули на разных языках.

Исходный модуль является входными данными для транслятора того языка, на котором написана исходная программа. В результате одного выполнения транслятора исходный программный модуль преобразуется в *объектный программный модуль*. При этом частично решается задача получения последовательности машинных команд, отображающих алгоритм модуля. Транслятор окончательно записывает коды операций машинных команд, а также проставляет в эти команды

номера используемых регистров. Что касается символьных имен, то они обрабатываются транслятором по-разному.

Рассмотрим преобразование адресов транслятором. *Адрес* — место в ОП, которое займет соответствующий программный объект — команда или данное. Любой язык программирования позволяет программисту использовать в программе не адреса, а их заменители — *символьные имена* (*метки*). Основные типы меток:

1) метки операторов;
2) имена переменных;
3) имена процедур (имя процедуры заменяет для программиста адрес, по которому первая команда процедуры находится в ОП).

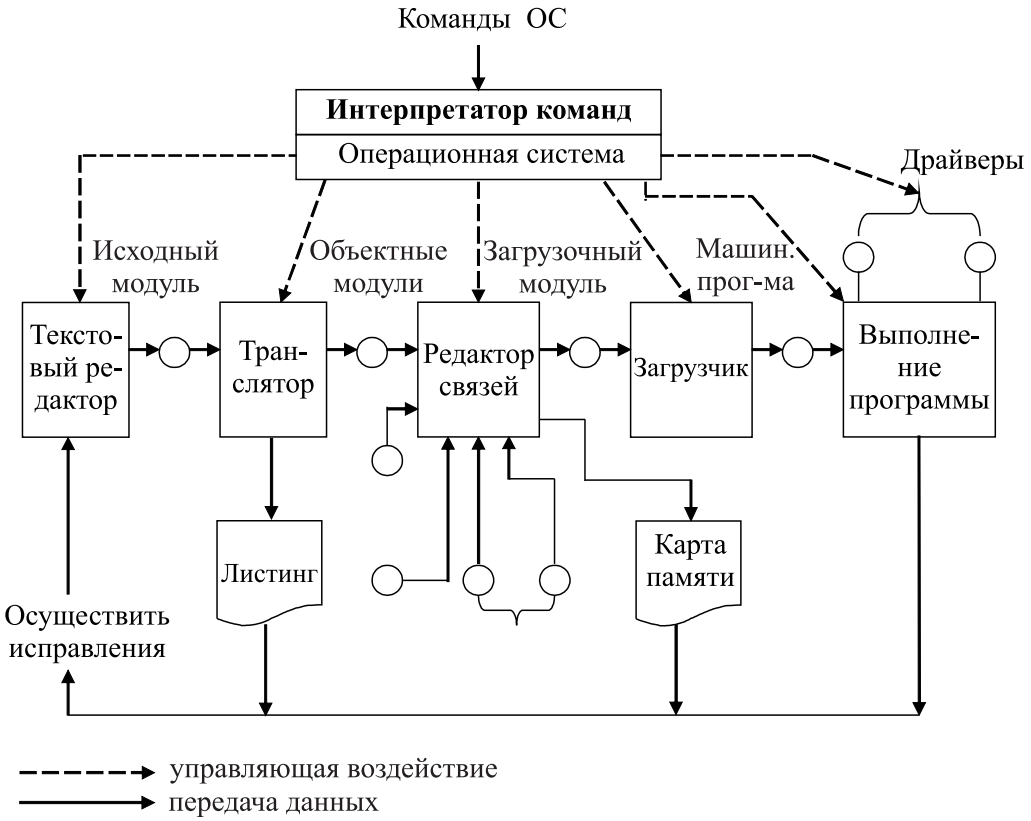


Рис. 18.1 – Общая схема создания и выполнения программы

Все символьные имена в исходном модуле делятся на *внешние* и *внутренние*. Символьное имя называется внутренним, если выполняются два условия:

1) соответствующий программный объект (оператор, данное или процедура) находится в этом исходном модуле;
2) данный программный объект используется (вызывается) только внутри данного исходного модуля.

Все внешние метки делятся на входные и выходные. *Внешние выходные метки* определены внутри данного исходного модуля, а используются вне его. Это:

1) имена процедур, входящих в состав данного исходного модуля, но которые могут вызываться в других исходных модулях;

2) имена переменных, которые определены в данном исходном модуле, а используются вне его.

Внешние входные метки определены вне данного исходного модуля, а используются в нем. Это:

1) имена процедур, не входящих в данный исходный модуль, но используемых в нем;
2) имена переменных, которые используются в исходном модуле, но определены вне его.

При получении объектного модуля вместо внутренних меток и внешних выходных меток транслятор проставляет в те машинные команды, которые используют соответствующие адреса, или смещение относительно текущего содержимого указателя команд IP, или смещение относительно начала сегмента ОП, в котором находится соответствующий программный объект. В первом из этих вариантов расчет смещения относительно начала сегмента памяти производится в момент выполнения данной машинной команды на ЦП. Что касается внешних входных меток, то обработать их транслятор не может. Он ничего не знает о размещении соответствующих программных объектов в памяти, так как имеет в своем распоряжении единственный исходный модуль, в котором этих объектов нет. Дальнейшее преобразование программы выполняет системная программа, называемая редактором связей.

Редактор связей (компоновщик) связывает («сшивает») все объектные модули программы в единый *загрузочный модуль*. При получении загрузочного модуля редактор связей записывает объектные модули один за другим в ОП. Поэтому он «знает», где расположен в памяти каждый программный объект. Следовательно, он может заменить все внешние метки, оставленные транслятором, на соответствующие численные адреса (адреса-сегменты и адреса-смещения). В конце своей работы редактор связей записывает загрузочный модуль в файл на магнитном диске и выдает программисту сообщение, называемое *картой памяти*. Эта карта описывает распределение памяти машинной программе.

В операционной системе MS-DOS допускается существование двух типов загрузочных модулей — типа .com и типа .exe (тип модуля совпадает с расширением имени файла, содержащего загрузочный модуль). Загрузочный модуль типа **.com** применяется для создания небольших машинных программ (объемом не более 64K), а типа **.exe** — для создания любых программ. Основное различие между этими загрузочными модулями заключается в том, что **.com** — файл содержит готовую машинную программу, не требующую «доводки» при загрузке в память, а **.exe** — файл содержит «заготовку» машинной программы, а также служебную информацию, используемую загрузчиком для настройки программы.

Как говорилось ранее, пользователь осуществляет запуск прикладных и системных обрабатывающих программ с помощью интерпретатора команд ОС. (ИК для MS-DOS существует в виде отдельного загрузочного модуля Command.com.) В диалоге с ним пользователь сообщает имя загрузочного модуля требуемой программы. В ответ интерпретатор команд вызывает системную программу — *загрузчик*, передав ему имя загрузочного модуля.

Загрузчик переписывает загрузочный модуль из ВП в ОП и, возможно, настраивает некоторые адреса в полях машинных команд. *Настройка* — изменение адреса в зависимости от фактического расположения машинной программы в ОП. Нетрудно предположить, что при загрузке .exe-файла настройка производится, а для .com-файла — не производится.

18.2 Префикс программного сегмента

Что касается машинной программы, записанной загрузчиком в ОП, то, кроме собственно инструкций программы и обрабатываемых ею данных, она содержит вспомогательную информацию, называемую *префиксом программного сегмента* (PSP). Длина PSP 256 (100h) байтов, и эта структура данных находится в начале любой машинной программы, вне зависимости от того, получена ли программа загрузкой .com- или .exe-файла. Информация в PSP используется операционной системой при выполнении запросов со стороны программы, а также может быть использована самой программой. Структура PSP приведена на рис. 18.2.

Относит. адрес	
00h	Команда INT 20h
02h	Верхняя граница памяти программы
04h	Зарезервировано
05h	Вызов диспетчера функций
0Ah	Векторы прерываний 22h-24h
16h	Адрес-сегмент PSP родительской программы
18h	Таблица логич. номеров файлов данных
2Ch	Адрес-сегмент блока окружения
2Eh	Зарезервировано
5Ch	2 блока управления файлами
80h	Хвост команды

Рис. 18.2 – Структура *PSP*

Для наглядного знакомства со структурой PSP удобно использовать отладчик Debug. Допустим, что мы ранее получили загрузочный модуль программы с именем Prob.com. Тогда для анализа PSP вызовем Debug следующей командой MS-DOS:

Debug Prob.com parametr,

где вместо слова parametr может быть записана любая символьная строка (в коде ASCII), содержащая входные параметры нашей прикладной программы. Эти

параметры совершенно не интересуют ни Debug, ни тем более MS-DOS, а используются лишь для передачи в программу какой-то исходной информации. (Кстати, если рассматривать в качестве программы сам Debug, то в качестве его параметра выступает имя исследуемой программы, то есть Prob.com).

После того, как мы запустили Debug описанным выше способом, с помощью команды Debug — **d 0** получим на экране дамп начальной части PSP. В результате на экран будут выведены первые 128 байтов PSP. Для вывода на экран каждых следующих 128 байтов достаточно просто ввести команду d.

В первых двух байтах PSP находится код машинной команды INT 20h. (Чтобы убедиться в этом, введем команду Debug — **u 0**.) Команда программного прерывания выполняет возврат в ОС при завершении программы. Сама же эта команда получает управление следующим образом.

Во-первых, сразу после загрузки программы в память загрузчик помещает в ее стек 16-битное нулевое слово (0000h). Во-вторых, если в конце главной подпрограммы прикладной программы стоит команда RET, а тип главной подпрограммы — NEAR, то при попадании RET на ЦП в качестве адреса возврата из стека будет выбран 0 и, следовательно, следующей исполняемой командой будет INT 20h. Следует отметить, что это не единственный способ возвращения из программы в ОС. Например, вместо RET в главной подпрограмме можно записать оператор INT 20h. В этом случае управление из программы возвращается в ОС непосредственно, минуя PSP. Другой способ возврата — запись в конце главной подпрограммы команды INT 21h (функция 4Ch). Этот способ является наиболее предпочтительным, так как он позволяет возвращать из программы в ОС код возврата (в регистре AL), информирующий операционную систему о причине завершения программы.

В следующих двух байтах PSP находится верхняя граница (адрес-сегмент) ОП. Обычно она равна величине A000h. Мы еще вернемся к этой величине, когда в дальнейшем будем рассматривать распределение ОП.

В пяти байтах PSP, начиная с 05h, находится вызов диспетчера функций MS-DOS. Данный вызов представляет собой команду дальнего вызова процедуры. С помощью команды Debug — **u 5** мы можем получить на экране, например следующее:

CALL F01D:FEF0,

где F01D — адрес-сегмент, а FEF0 — адрес-смещение первой команды диспетчера функций MS-DOS. (Соответствующий реальный адрес интересен тем, что он находится за пределами первого мегабайта. Мы вернемся к нему в вопросе о распределении памяти.)

Для того, чтобы обратиться из программы к MS-DOS (с целью получения помощи), достаточно поместить в нее команду CALL 5. В результате следующей командой, исполняемой на ЦП, будет команда вызова диспетчера функций. (Данный способ обращения к MS-DOS менее предпочтителен по сравнению с командой программного прерывания INT 21h.)

В двенадцати байтах PSP, начиная с 0Ah, содержатся копии векторов прерываний с номерами 22h, 23h, 24h. Эти прерывания MS-DOS использует для управления выполнением программы. При этом прерывание 22h используется для вызова той подпрограммы, которая должна получить управление при завершении при-

кладной программы. Прерывание 23h происходит при нажатии клавиш <Ctrl>&<C>. (MS-DOS использует для обработки этой комбинации специальный обработчик.) Причинами прерывания 24h являются аппаратные ошибки при работе с ПУ. Некоторые из этих ошибок:

1) попытка записи на защищенный диск;
2) нет бумаги на принтере;
3) неизвестное устройство. Хранение копий векторов перечисленных прерываний обусловлено тем, что в случае если прикладная программа изменяет содержимое векторов (с целью выполнить свою обработку прерываний), то после ее завершения MS-DOS восстанавливает прежнее содержимое векторов, используя копии из PSP.

По адресу 16h находится адрес-сегмент PSP родительской программы. Назначение этого поля будет понятно из п. 3.3. Следующее поле PSP начинается по адресу 18h и имеет длину 20 байтов. В нем размещена таблица логических номеров файлов, которая будет рассмотрена в следующей главе.

Начиная с ячейки 2Ch, два байта PSP содержат адрес-сегмент блока окружения программы. *Блок окружения* — совокупность переменных окружения. *Переменная окружения* — символьная строка в коде ASCII, имеющая структуру:

<имя переменной>=<первое значение>; . . .; <последнее значение>00*h*

После последней переменной окружения записывается не один, а два нулевых байта (0000h). Пример переменной окружения:

PATH = C:\WINDOWS; C:\WINDOWS\COMMAND00h

Данная переменная задает те каталоги (директории), в которых могут находиться используемые программой файлы. В том случае, если на вход программы поступает не имя-путь файла, а лишь завершающая часть этого имени, то для получения имени-пути программа должна последовательно просмотреть заданные значения переменной окружения PATH. В случае обнаружения в очередном каталоге искомого файла имя-путь каталога соединяется с именем файла в единое имя-путь файла.

Допустим, что с помощью команды Debug — **d 0** мы прочитали значение адреса-сегмента блока окружения, равное 2065. Тогда для получения на экране содержимого блока окружения следует воспользоваться командой: **d 2065:0**. Первоначальное содержимое своего блока окружения программа получает «в наследство» от той программы, которая создает данную программу, используя системный вызов **EXEC** (INT 21h, функция 4Bh). Данный вызов будет рассматриваться позже. А сейчас лишь заметим, что программа может не только читать свои переменные окружения, но и вносить в них изменения.

По адресу 5Ch находятся два блока управления файлами, используемые при работе с линейными файловыми структурами. Так как эти структуры сейчас полностью вытеснены иерархическими файловыми структурами, соответствующее поле PSP может использоваться для хранения любой другой информации. Например, учитывая, что подобно блоку окружения данные блоки управления передаются «по наследству», родительская программа может помещать в них исходные данные для дочерней программы. Другой способ передачи исходных данных в программу заключается в использовании «хвоста» команды.

Хвост команды — остаток команды ОС (после имени программы), запустившей прикладную программу. Иными словами, хвост содержит параметры команды, а также пробелы, которые были набраны в командной строке. В приведенном выше примере хвостом является слово parametr, предваряемое одним пробелом. Для размещения хвоста используется поле в PSP, начиная с ячейки 81h. Длина хвоста содержится в байте 80h. (Убедитесь в этом, используя Debug.)

Пример

Следующая простая .com-программа выводит на экран свои параметры (хвост команды).

```
;
;           Программа выводит на экран свои параметры
;           - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
            ASSUME    CS:_Text
_Text SEGMENT     PUBLIC     'CODE'
            ORG       80h      ; Следующий байт имеет адрес-смещение 80h
Lparam      DB        ?        ; Длина хвоста команды
Param       DB        ?        ; Первый байт хвоста
            ORG       100h
Start:      XOR       CX, CX          ; Обнуление CX
            MOV       CL, Lparam      ; CL←Длина хвоста
            CMP       CX, 0           ; Хвост отсутствует ?
            JZ        Exit            ; Если да
            MOV       AH, 0Eh         ; 0Eh — функция вывода символа
            MOV       BH, 0           ; Нулевая страница экрана
            XOR       SI, SI          ; SI←0
Next:       MOV       AL, Param[SI] ; Вывод следующего
            INT       10h             ; символа
            INC       SI              ;
            LOOP      Next
Exit:       MOV       AX, 4C00h       ; Возврат в DOS с
            INT       21h             ; кодом завершения 0

_Text       ENDS
END         Start
```

В данной программе для вывода символа на экран используется системный вызов BIOS — **INT 10h** (функция **0Eh**). Предварительно программа помещает код выводимого символа в регистр AL.

18.3 Программа типа com

Такая программа получается в результате загрузки .com-файла, которая сводится к простому переписыванию содержимого этого файла с диска в ОП.

На рис. 18.3 приведено содержимое сегментных регистров и регистров-указателей IP и SP после загрузки .com-программы в момент передачи ей управления. (Собственно передача управления заключается в записи в CS и IP тех значений, которые соответствуют первой исполняемой команде программы). В этот момент все сегментные регистры содержат одно и то же — начальный адрес (а точнее — номер начального параграфа) области ОП, в которую загружена программа. Так как программа начинается с PSP, то, следовательно, номер начального параграфа PSP и содержится в сегментных регистрах CS, DS, ES и SS.

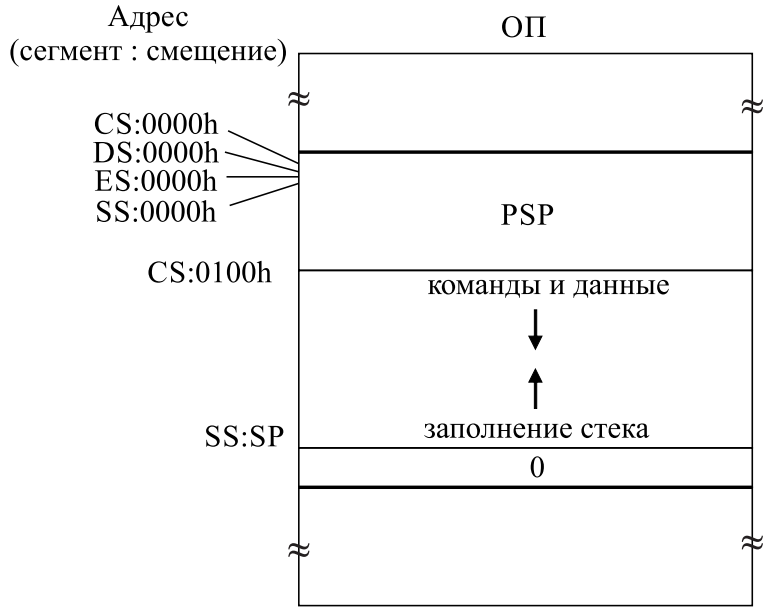


Рис. 18.3 – Результат загрузки com-программы

Первоначальное содержимое указателя команд IP задает адрес-смещение первой исполняемой команды программы. Для .com-программ это всегда число 100h, так как первая исполняемая команда начинается сразу же после завершения PSP. Что касается указателя стека SP, то он содержит адрес-смещение вершины стека, в который при загрузке программы уже было помещено одно нулевое слово. По мере того, как в стек будут записываться новые слова данных, его вершина будет приближаться к области памяти, занимаемой командами и данными программы. В случае наложения стека на другие данные программы дальнейшее ее продолжение будет или ошибочным, или вообще невозможным.

Исходная программа на ассемблере, ориентированная на получение .com-программы, имеет следующие особенности:

- первому исполнительному оператору программы предшествует псевдооператор **ORG 100h**, который обеспечивает требуемое смещение соответствующей машинной команды относительно начала программы;

- операнд псевдооператора END в конце исходной программы в качестве точки входа задает первый исполнительный оператор программы;
- исходная программа не имеет виртуального сегмента стека. Отсутствие такого сегмента не означает, что в машинной программе не будет стека (любая машинная программа обязательно имеет стек), а лишь говорит о том, что стек будет создан автоматически, то есть без участия программиста;
- исходная программа не должна содержать исполнительных операторов, выполняющих действия с сегментными адресами. (Такие адреса должны настраиваться загрузчиком в зависимости от размещения машинной программы в ОП .com-файла) Например, недопустимы следующие операторы:

```
MOV     AX, _Data       ; _Data — сегмент данных
MOV     SI, SEG Buff    ; SI←адрес-сегмент переменной Buff
```

Пример

Пример исходной программы, ориентированной на получение загрузочного модуля типа .com:

```
; Программа выводит сообщение «Здравствуйте» на экран
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
cr      EQU 0Dh                         ; Код ASCII возврата каретки
lf      EQU 0Ah                         ; Код ASCII перевода строки
ASSUME      CS:_Text
_Text       SEGMENT PUBLIC 'CODE'
ORG         100h
Start:      JMP     Begin               ; Переход через данные
Msg         DB 'Здравствуйте', cr, lf, '$'  ; Выводимое сообщение
Begin:      MOV     DX, OFFSET Msg      ; DX←адрес сообщения
MOV         AH, 9                       ; Вывод строки
INT         21h                         ; на экран
MOV         AX, 4C00h                   ; Возврат в DOS с
INT         21h                         ; кодом завершения 0
_Text       ENDS
END         Start
```

Эта исходная программа включает единственный виртуальный сегмент — сегмент кода _Text. Данные, обрабатываемые программой, размещены в сегменте _Text так, чтобы они не могли попасть на ЦП в качестве исполняемых команд, что неизбежно привело бы к сбою в работе процессора. Считается хорошим тоном при написании .com-программы на ассемблере располагать ее данные в начале программы, так как это упрощает трансляцию. Первый исполнительный оператор JMP «перепрыгивает» через эти данные на исполнительную часть программы.

Для вывода строки на экран программа использует системный вызов **INT 21h** (функция **9**). Перед применением вызова программа помещает в регистры DS и DX соответственно адрес-сегмент и адрес-смещение для младшего байта выводимой строки. (В нашей программе DS уже содержит требуемый адрес и поэтому не загружается.) Выводимая строка обязательно должна заканчиваться символом «$». Сама строка может содержать любые другие символы, в том числе и управляющие. В программе используются управляющие символы 0Dh (возврат каретки) и 0Ah (перевод строки).

18.4 Программа типа exe

В отличие от .com-программы, загружаемой всегда в один сегмент ОП (объемом 64K), программа типа .exe может занимать несколько таких сегментов. Следствием этого является наличие в программе команд, выполняющих запись в регистры адресов-сегментов, требующих настройки во время загрузки.

В результате, получение .exe-программы требует не простого копирования соответствующего файла в ОП, а предполагает преобразование этого файла загрузчиком. В результате такого преобразования, во-первых, записывается PSP программы, а во-вторых, настраиваются некоторые адреса в полях машинных команд. Наличие первой из этих функций обусловлено тем, что преобразуемый .exe-файл вообще не содержит PSP. Вместо него в начале этого файла находится *заголовок*, размер которого кратен 512 байтам, содержащий управляющую информацию, используемую загрузчиком для получения .exe-программы.

Свою работу по созданию .exe-файла редактор связей начинает с того, что последовательно записывает в свою память код (команды), данные и стек будущей программы. Порядок расположения этих частей программы, в отличие от .com-файла, может быть любым. Их содержимое редактор связей считывает из доступных ему объектных модулей программы. Выполняя размещение программы в памяти, редактор связей обращается с ее адресами так, как будто программа загружена в ОП, начиная с условного адреса 00000h. Поэтому считается, что часть программы (код, данные или стек), размещаемая первой, имеет начальный логический адрес 0000h:0000h. Если, например, вторая часть программы начинается с условного реального адреса 000E7h, то ему соответствует начальный логический адрес 000Eh:0007h. Применение таких условных адресов позволяет редактору связей выполнить запись численных значений для всех внешних адресов, оставшихся не проставленными после трансляции.

Исключение составляют команды, выполняющие запись в регистры значений адресов-сегментов. К таким командам относятся, во-первых, команды дальнего перехода JMP и CALL (они выполняют запись адреса-сегмента в регистр CS), а во-вторых, команды MOV, выполняющие запись адресов-сегментов в регистры данных (из регистров данных другие команды MOV переписывают значения в регистры сегментов DS, ES или CS). Например, следующие два оператора исходной программы выполняют запись в регистр DS того адреса-сегмента, который соответствует точке исходной программы с меткой Msg:

```
MOV     DX, SEG Msg
MOV     DS, DX
```

Первый оператор MOV транслируется в трех-, а второй — в двухбайтовую машинную команду. При этом второй и третий байты первой команды используются для размещения адреса-сегмента, загружаемого в регистр DX.

Так как адреса-сегменты зависят от фактического размещения будущей программы в ОП, которое редактор связей «не знает», то он не может проставить численные значения адресов-сегментов в поля машинных команд. Тем не менее он выполняет значительную подготовку для будущей простановки этих значений загрузчиком. Для этого, во-первых, редактор связей помещает в эти поля условные адреса-сегменты, полученные в предположении, что программа загружается в память, начиная с нулевого адреса.

Во-вторых, редактор связей создает *таблицу настройки*, занимающую почти весь объем заголовка .exe-файла. Эта таблица состоит из 4-х байтовых полей, каждое из которых соответствует одному адресу-сегменту, используемому какой-то командой программы, и который требует настройки в зависимости от фактического размещения программы в памяти. В качестве содержимого этого поля редактор связей записывает условный логический адрес (сегмент и смещение) той ячейки (двух байтов) программы, которая содержит адрес-сегмент, требующий настройки.

Сама настройка адресов-сегментов программы производится загрузчиком сразу же после размещения программы (в том числе и PSP) в ОП. Для этого загрузчик последовательно просматривает таблицу настройки, считывая из нее указатель на очередную последовательность из двух байтов программы, требующую настройки. А затем он прибавляет к их содержимому (то есть к условному адресу-сегменту) номер того параграфа ОП, начиная с которого программа загружена фактически.

После того, как программа загружена, загрузчик передает ей управление. На рис. 18.4 приведено содержимое сегментных регистров и указателя стека SP в момент передачи управления .exe-программе. В этот момент сегментные регистры данных содержат номер начального параграфа PSP. Благодаря этому программа может использовать полезную информацию, содержащуюся в PSP. (Запомнив где-то первоначальное значение этих регистров, программа обычно записывает в них новое значение, указывающее на начало области данных программы.) Что касается CS, то в него загрузчик помещает начальный номер параграфа не PSP, а области кода программы. Аналогично содержимое SS указывает на границу области стека программы.

Исходная программа на ассемблере, ориентированная на получение .exe-программы, обязательно имеет кроме виртуальных сегментов кода и данных виртуальный сегмент стека. Порядок записи этих сегментов в программе может быть различным. В качестве точки входа в программу (операнд оператора END) может быть задан любой исполнительный оператор программы (рис. 18.4).

Пример

Следующая .exe-программа выполняет то же самое, что и предыдущая .com-программа.

```
;       Программа выводит сообщение «Здравствуйте» на экран
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
;
cr      EQU 0Dh                          ; Код ASCII возврата каретки
lf      EQU 0Ah                          ; Код ASCII перевода строки
ASSUME      CS:_Text, DS:_Data, SS:_Stack
_Text       SEGMENT PUBLIC 'CODE'        ; Сегмент кода
Start:      MOV     AX, _Data            ; Сделаем сегмент
MOV         DS, AX                       ; данных адресуемым
MOV         DX, OFFSET Msg               ; DXINT адрес сообщения
MOV         AH, 9                        ; Вывод строки
INT         21h                          ; на экран
MOV         AX, 4C00h                    ; Возврат в DOS с
INT         21h                          ; кодом завершения 0
_Text       ENDS
_Data       SEGMENT PUBLIC 'DATA'        ; Сегмент данных
Msg         DB      'Здравствуйте', cr, lf, '$'; Выводимая строка
_Data       ENDS
_Stack      SEGMENT STACK 'STACK'        ; Сегмент стека
DB          20 DUP (?)
_Stack      ENDS
END         Start
```

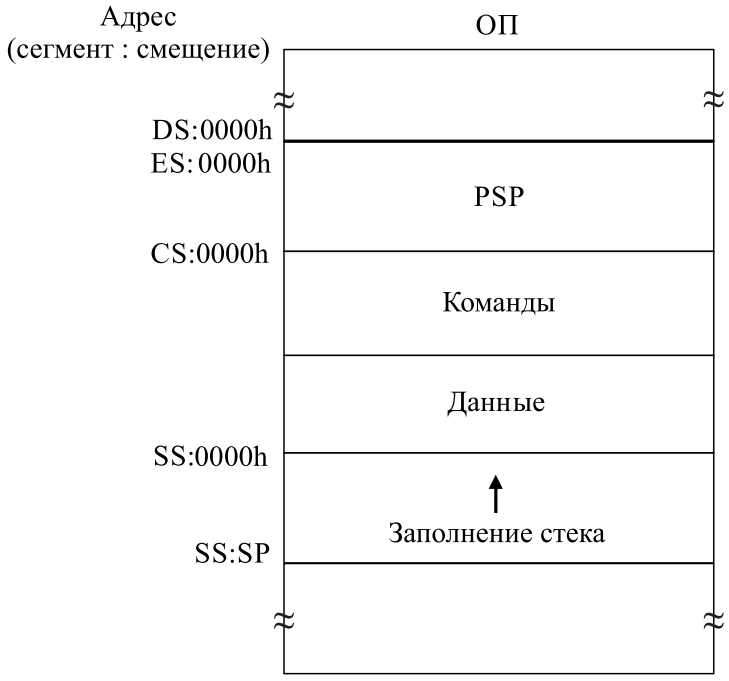


Рис. 18.4 – Результат загрузки exe-программы

В данной исходной программе три виртуальных сегмента – сегмент кода _Text, сегмент данных _Data и сегмент стека _Stack. Обратите внимание, что в начале программы в регистр DS загружается адрес-сегмент, соответствующий началу данных программы. Это делается для того, чтобы фактическое размещение данных

в памяти никак не зависило от размещения в ней PSP. (Для .com-программы адрес-смещение данных вычисляется относительно начала PSP.) Обратите внимание, что запись значения в сегментный регистр *DS* производится не непосредственно, а через рабочий регистр *AX*.

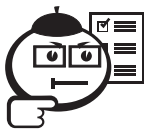
Еще одним отличием исходных *exe*- и *com*-программ является то, что в *exe*-программе никак не задается расположение сегмента кода в памяти. Это полностью остается на усмотрение редактора связей и загрузчика.

Допустим, что записанная выше исходная *exe*-программа имеет имя *test1.asm*. Тогда для получения соответствующего объектного модуля можно использовать следующую команду *DOS*:

nasm test1.asm -f obj -o test1.obj

Команда *DOS* для получения exe-файла:

alink test1.obj -oEXE -o test1.exe



Получите *exe*-файл для приведенной выше программы, а затем выполните его.

Контрольные вопросы по главе 18

1) Как происходит преобразование адресов транслятором?
2) В чем отличие загрузочного модуля типа .com от загрузочного модуля типа .exe?
3) Что такое префикс программного сегмента?
4) Как создать программу типа .com?
5) Как создать программу типа .exe?

ЗАКЛЮЧЕНИЕ

В рамках изучения курса «Ассемблер для процессора Intel 8086» вы освоили основы программирования на языке ассемблера для микропроцессора Intel 8086, получили навыки работы с системой debug, познакомились с ассемблером NASM. Изучили такие понятия, как вычислительная система, системы счисления, операционные системы. Имеете представление о структуре аппаратных средств, архитектуре процессора, адресации памяти, алгоритме работы процессора, файловой структуре хранения информации.

Научились конструировать довольно сложные программы, познакомились с методами структурного программирования.

Полученные знания являются базисом для освоения дисциплин направлений «Информатика и вычислительная техника», «Управление в технических системах», помогут вам в дальнейшем освоении компьютерных дисциплин, программировании на языках высокого уровня. Данное пособие может применяться для обучения по другим направлениям и специальностям, связанным с вычислительной техникой.

ЛИТЕРАТУРА

[1] Одиноков В. В. Информатика. Ассемблер для процессора i8086 : учеб. пособие / В. В. Одиноков. — Томск, ТУСУР, 2000. — 93 с.

[2] Информатика. Базовый курс : учебник для вузов / В. С. Симонович [и др.] ; ред. В. С. Симонович. — 2-е изд. — СПб. : Питер, 2007. — 639 с.

[3] Абель Питер. Ассемблер и программирование для IBM PC / Питер Абель. — СПб. : Корона-Век, 2009. — 736 с.

[4] Одиноков В. В. Программирование на ассемблере : учеб. пособие / В. В. Одиноков, В. П. Коцубинский. — М. : Горячая линия-Телеком, 2011.

[5] Острейковский В. А. Информатика : учебник для вузов / В. А. Острейковский. — М. : Высшая школа, 2001. — 512 с.

[6] Юров В. И. Assembler : учебник для вузов / В. И. Юров. — СПб. : Питер, 2001. — 624 с.

ГЛОССАРИЙ

Алгоритм — набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий.

Арифметико-логическое устройство — часть процессора, выполняет операции, связанные с обработкой информации.

Ассемблер — язык программирования низкого уровня, приближен к машинному языку.

Байт — единица измерения информации, равна 8 бит.

Бит — минимальная единица измерения информации, может принимать значения 0 или 1.

Инверсия — логическое отрицание.

Процессор — электронный блок либо интегральная схема (микропроцессор), исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера или программируемого логического контроллера.

Процедура — относительно независимая часть программы, представляет собой список машинных команд, которые можно вызвать из любого места программы.

Регистр — ячейка пвмяти процессора, предназначенная для быстрого выполнения операций процессором Система счисления — символический метод записи чисел, представление чисел с помощью письменных знаков. Количество символов в системе счисления называется её основанием.

Система счисления — символический метод записи чисел, представление чисел с помощью письменных знаков. Количество символов в системе счисления называется её основанием.

Стек — область памяти, предназначенная для хранения информации. Свойство стека — «первый пришёл — последний ушёл».

Файл — область памяти на внешнем устройстве, которая имеет имя и расширение, предназначена для хранения информации.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Учебное издание

Потапова Евгения Андреевна

ИНФОРМАТИКА.
АССЕМБЛЕР ДЛЯ ПРОЦЕССОРА INTEL 8086

Учебное пособие

Корректор Осипова Е. А.
Компьютерная верстка Лигай Т. А.

Подписано в печать 10.10.13. Формат 60x84/8.
Усл. печ. л. 19,53. Тираж 100 экз. Заказ

Издано в ООО «Эль Контент»
634029, г. Томск, ул. Кузнецова д. 11 оф. 17
Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.