Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра компьютерных систем в управлении
и проектировании (КСУП)

**Е.А. Потапова, В. П. Коцубинский**

# ИНФОРМАТИКА

## Ассемблер для процессора i8086

Учебное пособие

**2013**

**Е.А. Потапова, В. П. Коцубинский**
Информатика. Ассемблер для процессора i8086: Учебное пособие, второе издание. - Томск: ТУСУР, каф. КСУП 2013. - 93 с.

# СОДЕРЖАНИЕ

1. Введение . . . . . 5
2. Представление информации . . . . . 6
2.1. Двоичные числа . . . . . 6
2.2. Шестнадцатеричные числа . . . . . 9
2.3. Символьная информация . . . . . 11
3. Аппаратные средства простой ЭВМ . . . . . 11
3.1. Общая структура ЭВМ . . . . . 11
3.2. Работа центрального процессора . . . . . 13
3.3. Архитектура микропроцессора Intel 80x86 . . . . . 14
4. Списки . . . . . 18
4.1. Основные понятия . . . . . 18
4.2. Несвязанные списки . . . . . 20
4.3. Связанные списки . . . . . 22
5. Аппаратная поддержка взаимодействия между программными модулями . . . . . 23
5.1. Работа со стеком . . . . . 24
5.2. Процедуры . . . . . 25
5.3. Программные прерывания . . . . . 27
6. Основные операторы ассемблера . . . . . 28
6.1. Типы операторов . . . . . 28
6.2. Операторы обработки данных . . . . . 29
6.2.1. Арифметические операторы . . . . . 29
6.2.2. Логические операторы . . . . . 32
6.2.3. Операторы передачи данных . . . . . 33
6.2.4. Структура FLAGS и операции над ним . . . . . 35
6.2.5. Операторы сдвига . . . . . 37
6.2.6. Цепочечные (строковые) операторы . . . . . 39
6.3. Адресация данных . . . . . 41
6.4. Определение данных . . . . . 43
6.4.1. Метки . . . . . 43
6.4.2. Определение байтов . . . . . 44
6.4.3. Определение слов . . . . . 44
6.4.4. Определение констант . . . . . 45
6.4.5. Структуры . . . . . 45
6.5. Операторы передачи управления . . . . . 47
6.5.1. Операторы условных переходов . . . . . 47
6.5.2. Операторы безусловных переходов . . . . . 49
6.5.3. Операторы циклов . . . . . 51
6.5.4. Операторы процедур . . . . . 52
6.5.5. Другие операторы передачи управления . . . . . 53

6.6. Вспомогательные псевдооператоры . . . . . . . . . . . . . . . . . . . . . 54
6.7. Макрооператоры . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
7. Проектирование программы . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.1. Введение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.2. Результаты проектирования . . . . . . . . . . . . . . . . . . . . . . . . . . 58
7.3. Структурное программирование . . . . . . . . . . . . . . . . . . . . . . . 61
7.4. Методы проектирования программ . . . . . . . . . . . . . . . . . . . . . . 65
8. Кодирование и отладка . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
8.1. Кодирование управляющих структур . . . . . . . . . . . . . . . . . . . . 66
8.2. Этапы преобразования программы . . . . . . . . . . . . . . . . . . . . . . 68
8.3. Кодирование виртуальных сегментов . . . . . . . . . . . . . . . . . . . . 73
8.4. Влияние на программу типа загрузочного модуля . . . . . . . . . . . . . 77
8.5. Замена логических сегментов памяти . . . . . . . . . . . . . . . . . . . . 80
8.6. Комментирование программы . . . . . . . . . . . . . . . . . . . . . . . . . 82
8.7. Отладка программы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
9. Организация информации во внешней памяти . . . . . . . . . . . . . . . 84
9.1. Файлы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
9.2. Таблица размещения файлов . . . . . . . . . . . . . . . . . . . . . . . . . . 85
9.3. Файловая структура системы . . . . . . . . . . . . . . . . . . . . . . . . . . 86
10. Системные программы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
10.1. Классификация системных программ . . . . . . . . . . . . . . . . . . . . 88
10.2. Утилиты . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
10.3. Лингвистические процессоры . . . . . . . . . . . . . . . . . . . . . . . . . 90
10.4. Подпрограммы управления аппаратурой . . . . . . . . . . . . . . . . . . 91

# 1. 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 .

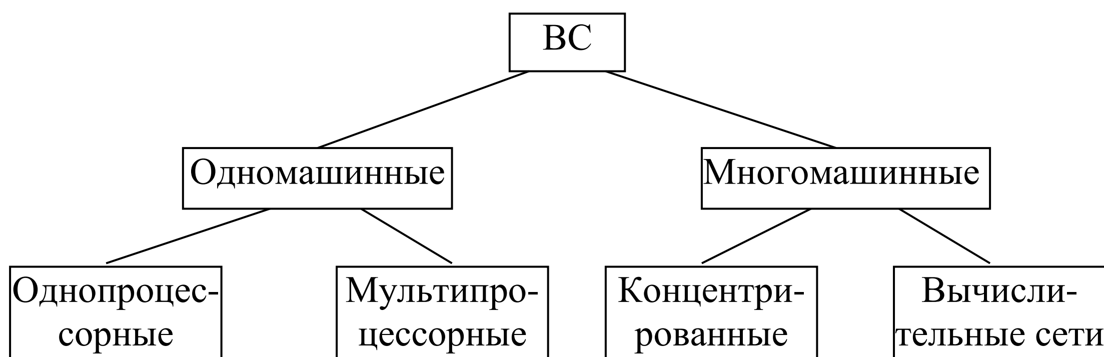



Рис. 1. Классификация ВС по составу аппаратных средств

***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.

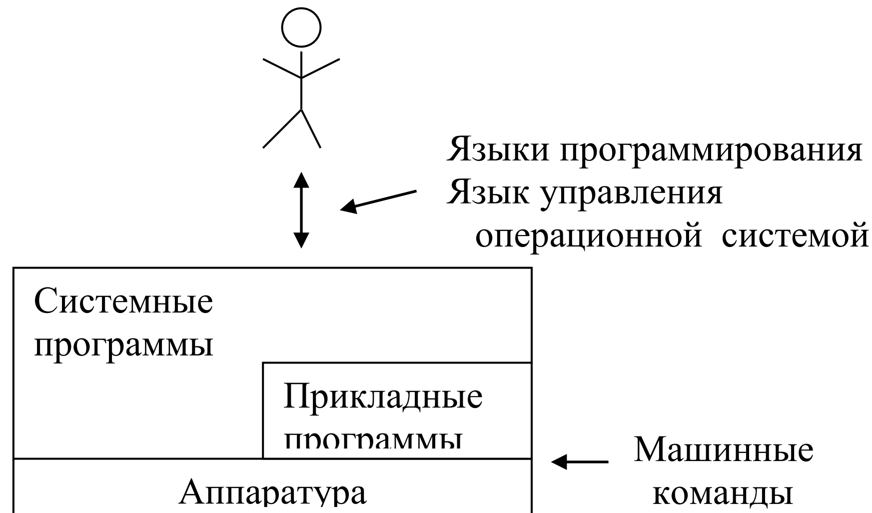



Рис. 2. Взаимодействие пользователя с виртуальной ЭВМ

# 2. ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ

## 2.1. 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).

Любое число можно представить в виде цепочки битов. Такое представление числа называется ***двоичным числом***. Цепочка из восьми битов называется ***байтом*** (рис. 3).

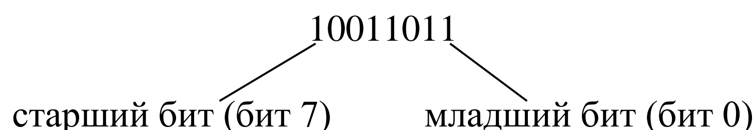



Рис. 3. Пример байта

Величина двоичного числа определяется относительной позицией каждого бита и его значением. Позиционный вес младшего бита $2^o$ = 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. 4).

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | позиция |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | вес |

Рис. 4. Веса позиций байта

Для преобразования десятичного числа в двоичное надо сделать ряд вычитаний. Каждое вычитание даст значение одного бита. Прежде всего, вычтите из десятичного числа наибольший возможный двоичный вес и запишите 1 в эту позицию бита. Затем из результата вычтите новый наибольший возможный двоичный вес и запишите 1 в эту новую позицию бита. Так - до получения нулевого результата. Например, преобразование числа 50 в двоичное:

```
_50
 32  (бит 5 = 1)
_18
 16  (бит 4 = 1)
 _2
  2  (бит 1 = 1)
  0
```

Записывая 0 в остальные позиции битов (биты 0,2,3) получаем окончательный результат: 110010.

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

32 (бит 5) + 16 (бит 4) + 2 (бит 1) = 50

Байт может представлять десятичные положительные числа от 0 (00000000) до 255 (11111111). Память ЭВМ состоит из блоков памяти по 1024 байта. Число 1024 есть $2^{10}$. Число 1024 имеет стандартное обозначение **К**. Следовательно, ЭВМ, имеющая 48 К памяти, содержит 48 x 1024 = 49152 байта.

***Машинным словом*** будем называть битовую строку длиной 16 битов. Одно слово содержит 2 байта (рис. 5).

Каждый бит слова имеет свой вес (рис. 6). Просуммировав все веса, найдем максимальное целое число без знака, которое можно записать в одно слово, оно равно $2^{16} - 1 = 65535$.

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. Просуммируем два таких числа:

```
  00111100           60
+ 00110101         + 53
  01110001          113
```

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.

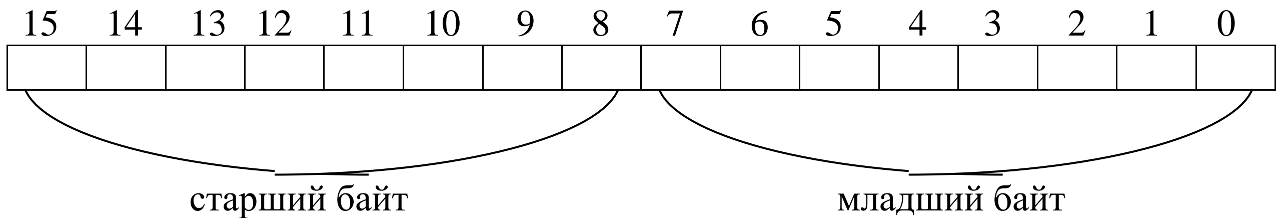



Рис. 5. Структура машинного слова

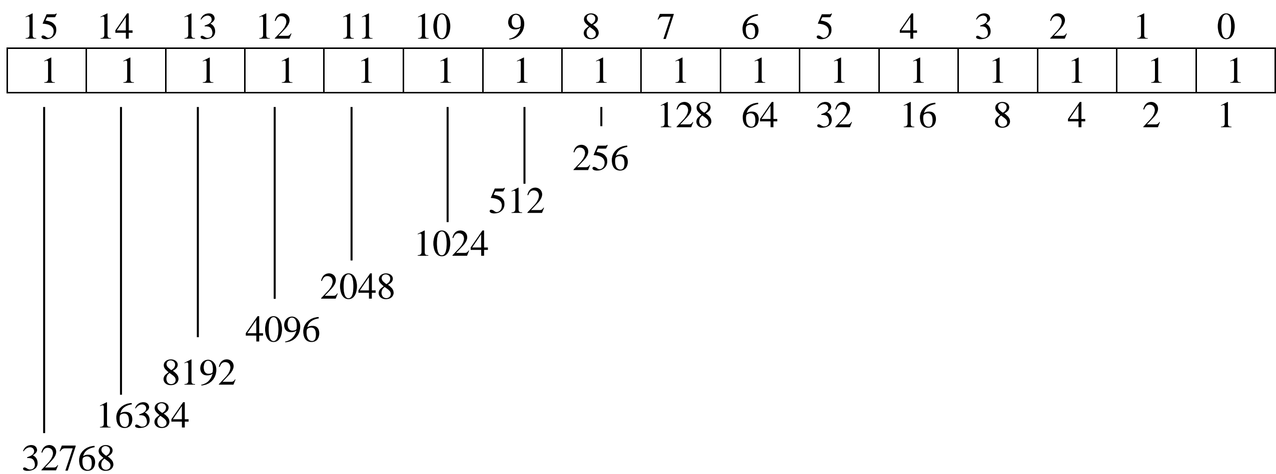



Рис. 6. Веса позиций слова

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. Например, получим дополнительный код числа –65.

65(10) = 01000001(2)

```
1)        10111110
2)      +        1
          10111111 = -65(10)
```

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

```
        -65(10)= 10111111
1)               01000000
2)             +        1
                 01000001 = 65(10)
```

Сумма +65 и –65 должна составить ноль:

```
   01000001  (+65)
 + 10111111  (-65)
   00000000
```

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
```

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.

## 2.2. 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***. На рис. 7 приведено соответствие между двоичными, десятичными и шестнадцатеричными числами от 0 до 15(10).

| Двоичн. | Десят. | Шестн. | Двоичн. | Десят. | Шестн. |
|---|---|---|---|---|---|
| 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 |

Рис. 7. Соответствие между двоичными, десятичными и шестнадцатеричными числами

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. Например, найдем сумму 6Ah и B5h:

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

Разность B5 – 6A:

$$\begin{array}{r} _\ B5 \\ \underline{6A} \\ 4B \end{array}$$

### 2.3. 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**.

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

## 3. АППАРАТНЫЕ СРЕДСТВА ПРОСТОЙ ЭВМ

### 3.1. Общая структура ЭВМ

Среди однопроцессорных ЭВМ наиболее распространены **ЭВМ *с общей шиной*** (рис. 8). 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.

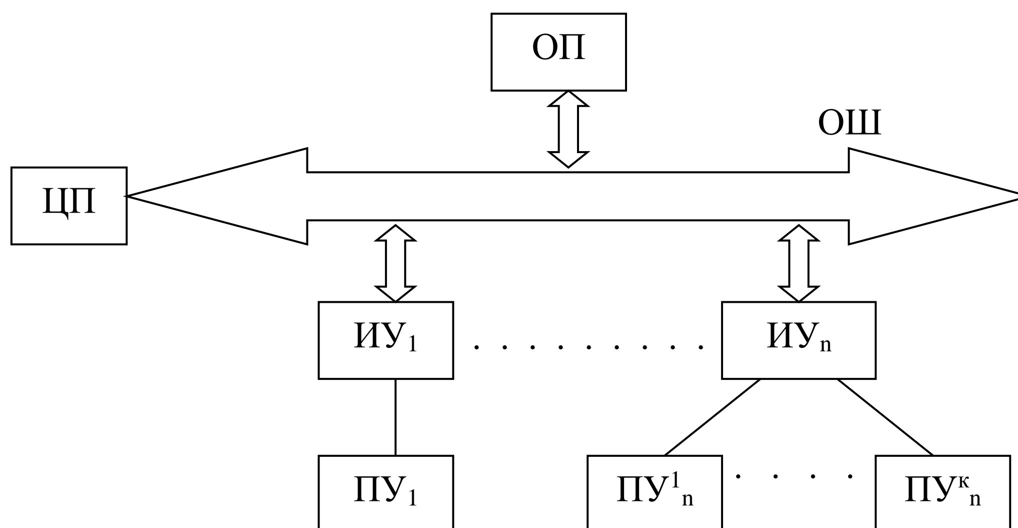



Рис. 8. Структура ЭВМ с общей шиной

***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. На рис. 9 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.

Рис.9. 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. 10). 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.

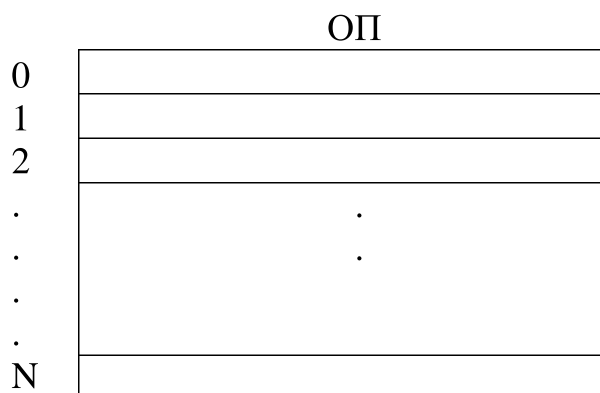



Рис. 10. Структура ОП

***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.

## 3.2. Работа центрального процессора

На рис. 11 приведена типичная архитектура (структура) ЦП. ***Устройство управления*** выполняет дешифрирование и исполнение инструкций. Набор рабочих регистров предназначен для адресации и выполнения вычислительных операций. ***Регистр*** – ячейка памяти, скорость обмена с которой намного выше, чем с другими видами памяти – с ОП и, тем более, с ВП. ***Арифметико-логическое устройство*** предназначено для выполнения арифметических и логических инструкций.

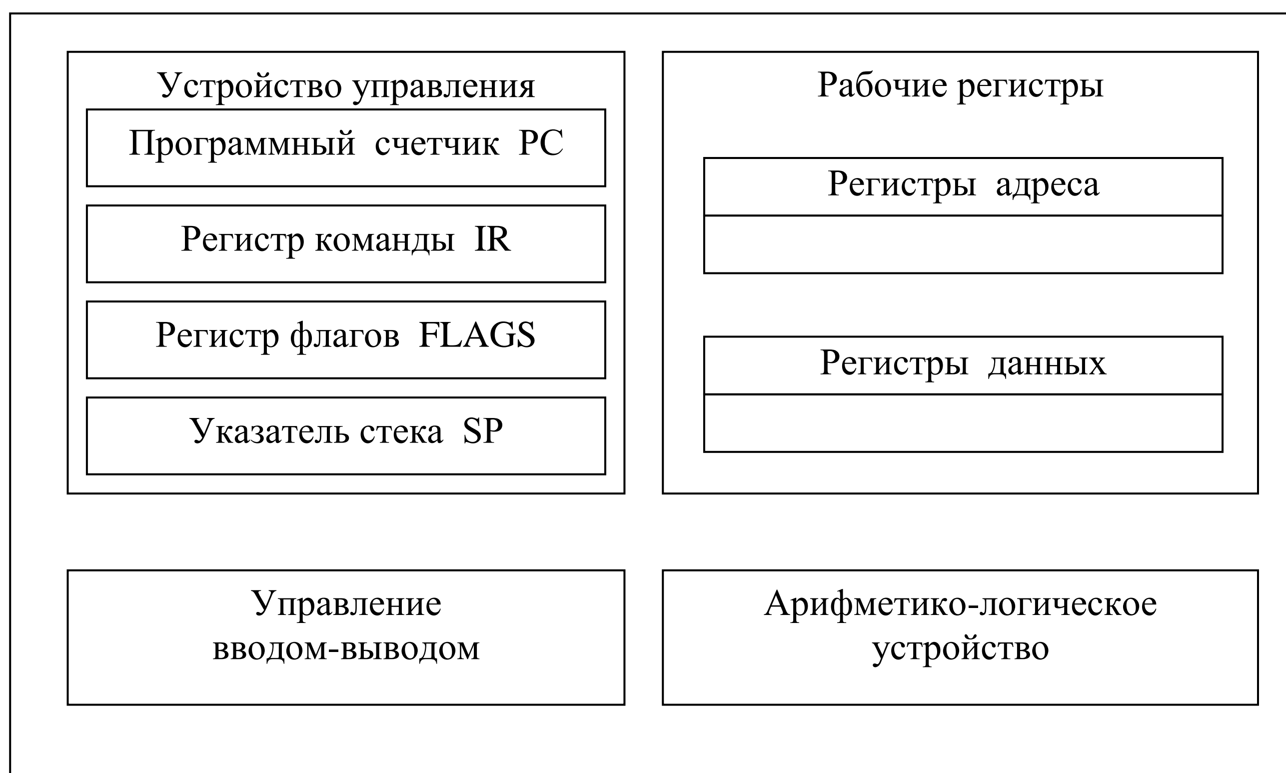



Рис. 11. Структура ЦП

Программа на машинном языке выполняется последовательно (инструкция за инструкцией) до тех пор, пока не встретится ***инструкция перехода***. ***Регистр команды*** **IR** содержит текущую команду на время ее дешифрации (распознавания) и выполнения. А ***программный счетчик*** **PC** предназначен для хранения адреса следующей команды.

***Регистр флагов*** **FLAGS** отражает текущее состояние ЦП. В этом регистре есть биты (флаги), отражающие результат предыдущей арифметической или логической инструкции. Например, флаг нуля **ZF** устанавливается в единицу при получении в предыдущей инструкции нулевого результата. Кроме того, FLAGS содержит управляющие флаги, устанавливаемые (сбрасываемые) специальными инструкциями ЦП. Значение такого флага сохраняется до выполнения на ЦП следующей инструкции, специально предназначенной для работы с этим флагом.

На рис. 12 приведен алгоритм выполнения инструкций в ЦП. Когда текущая инструкция завершена, адрес из PC выдается на шину адреса, ОП помещает следующую инструкцию на шину данных и ЦП вводит команду в IR. Пока дешифрируется эта инструкция, определяется ее длина в байтах и производится увеличение содержимого PC на эту длину, в результате чего PC адресует следующую инструкцию. Когда выполнение данной инструкции заканчивается, содержимое PC помещается на шину адреса и цикл повторяется.

***Инструкция безусловного перехода*** позволяет изменить естественный порядок следования инструкций. Это делается путем замещения содержимого PC (т.е. адреса следующей по порядку инструкции) адресом, задаваемым самой инструкцией перехода.

***Инструкции условных переходов*** замещают или не замещают содержимое PC в зависимости от результатов предыдущих инструкций, отраженных в FLAGS. Например, если после инструкции «вычитание» в программе находится инструкция «переход по нулю», то переход осуществляется, если FLAGS показывает получение при вычитании нулевого результата. Если же FLAGS фиксирует ненулевой результат вычитания, переход не производится.

Когда реализован переход, начинается новая последовательность инструкций с адреса, к которому осуществлен переход.

## 3.3. Архитектура микропроцессора Intel 80x86

Этот 16-битный микропроцессор используется в качестве ЦП в ПЭВМ IBM PC/XT. Его структура приведена на рис. 13.

Функцию регистра команды IR выполняет ***очередь команд***, в которую для повышения быстродействия заранее считываются из ОП следующие по порядку в программе инструкции. Регистрами адреса являются сегментные регистры.

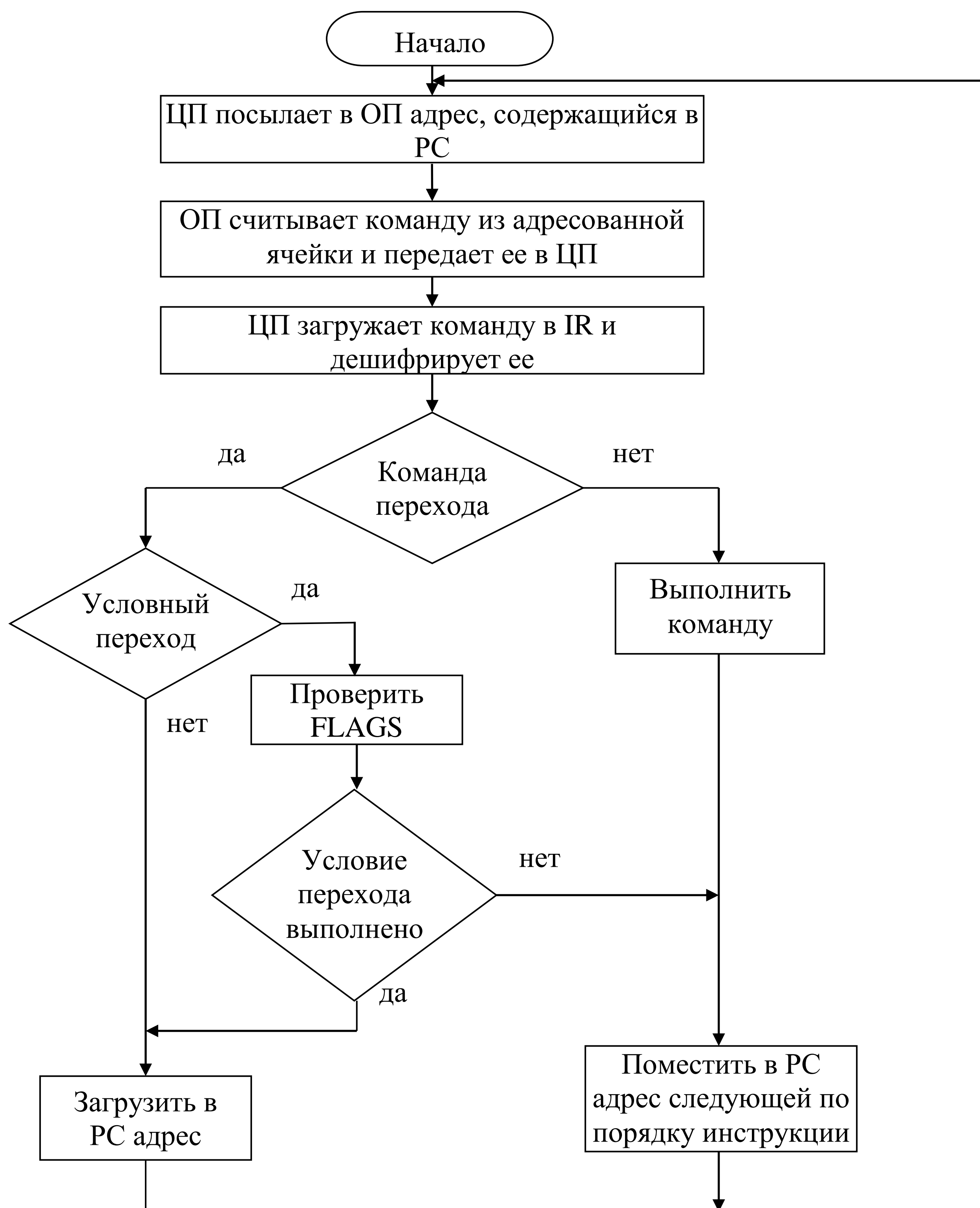



Рис. 12. Алгоритм работы ЦП

К регистрам данных относятся регистры **AX**, **BX**, **CX** и **DX**. Они предназначены для хранения операндов и результатов операций и 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 арифметических функций имеют и специальные назначения.

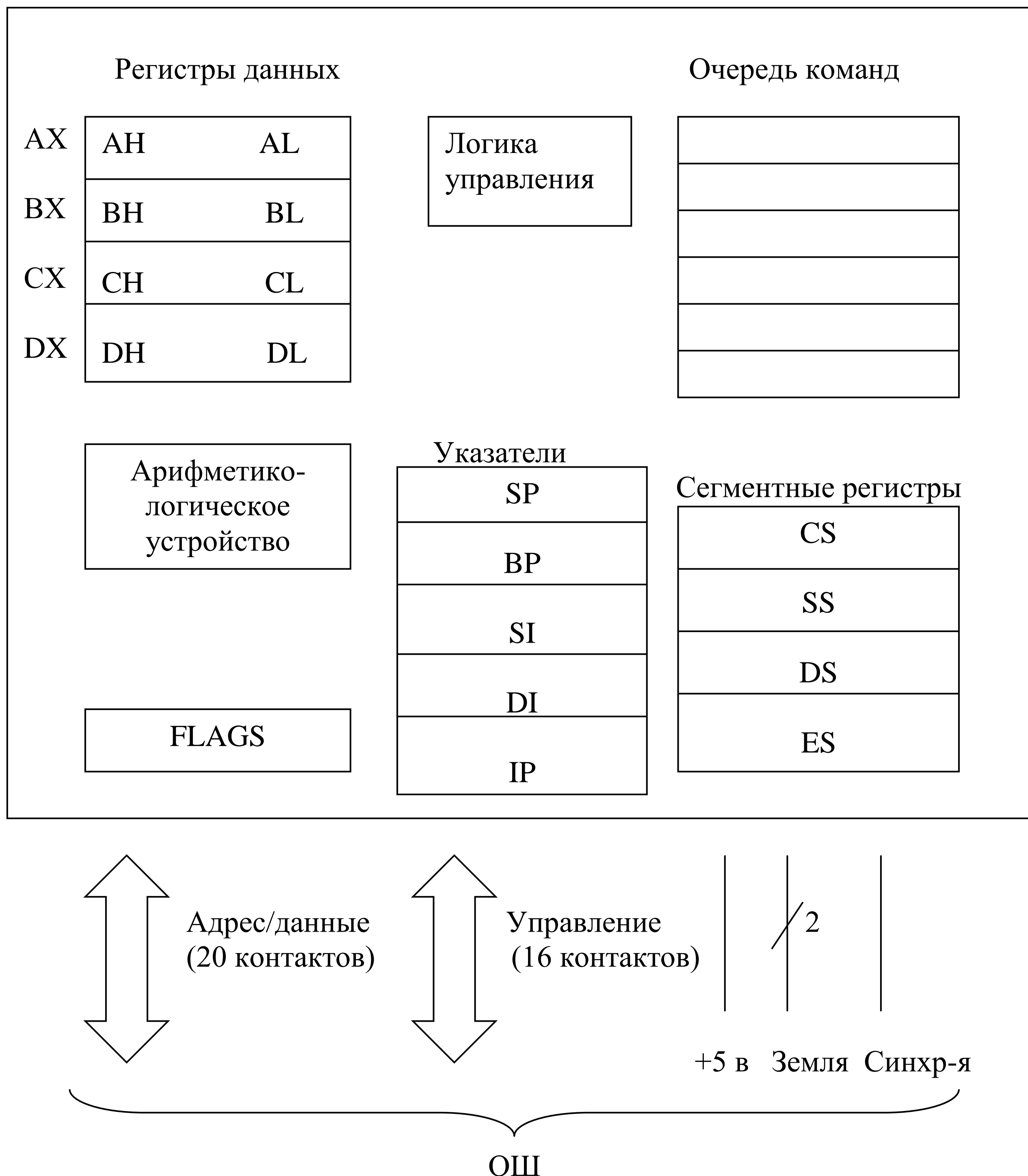



Рис. 13. Структура микропроцессора i8086

В группу регистров ***указатели*** входят указатель команды **IP** и регистр **SP**, которые фактически являются соответственно программным счетчиком и указателем стека, но полные адреса инструкции и стека образуются

суммированием содержимого этих регистров и содержимого сегментных регистров CS и SS, которые рассматриваются далее.

Регистр **BP** является базовым при обращении к стеку. Регистры **SI** и **DI** являются индексными.

Теперь рассмотрим применение сегментных регистров. 20-и 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.

Мысленно разобъем ОП на участки по 16 байт, называемые ***параграфами*** (рис. 14). Регистр сегмента кода **CS** содержит номер параграфа, с которого начинается выделенный нашей программе сегмент. Например, это может быть число 0002h. Для получения реального адреса начальной ячейки параграфа необходимо его номер (0002h) умножить на число 16: 0002h x 10h = 00020h.

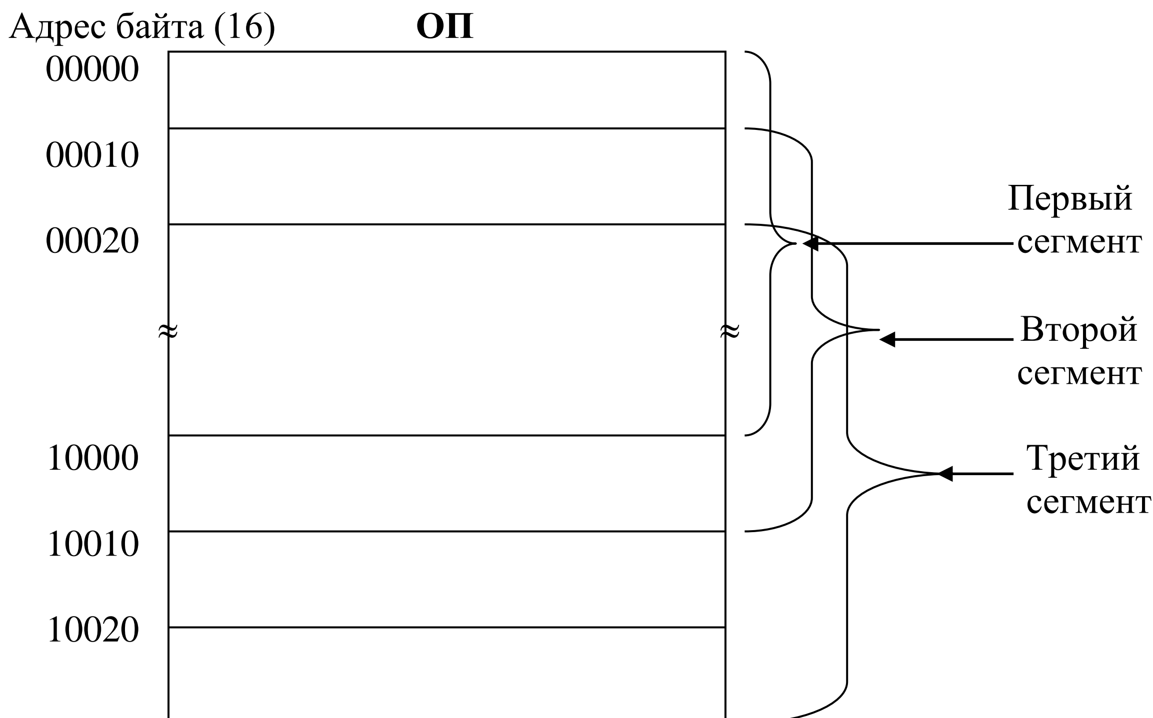



Рис. 14. Разбиение ОП на параграфы

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) x 10h + (IP).

Например, пусть (CS) = 0002h, а (IP) = 0100h, тогда R = 2h x 10h + 100h = 120h (рис. 15).

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.

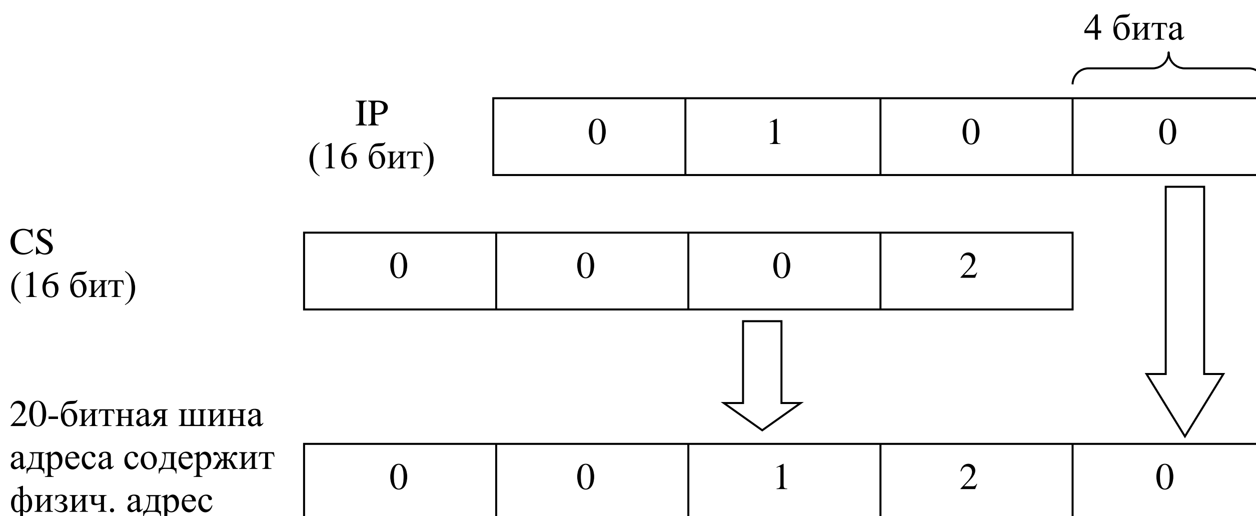



Рис. 15. Получение физического адреса

В принципе, программа может обрабатывать любые ячейки ОП. Для этого достаточно в регистр CS записать новое значение. В результате происходит смена текущего сегмента.

# 4. СПИСКИ

## 4.1. Основные понятия

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. 16). 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.

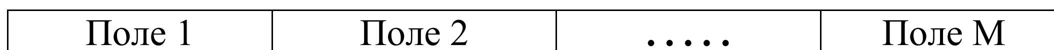



Рис. 16. Структура записи

В списке записи расположены не как попало, а 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. 17). Пример списка, который не является линейным, приведен на рис.18.

Рис. 17. Линейный список

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. 18. Нелинейный список

## 4.2. Несвязанные списки

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.

Пользуясь этими переменными, нетрудно входить в список как с начала, так и с конца (рис.19).

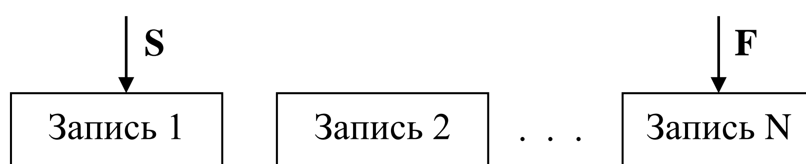



Рис. 19. S и F – указатели на начало и конец линейного списка

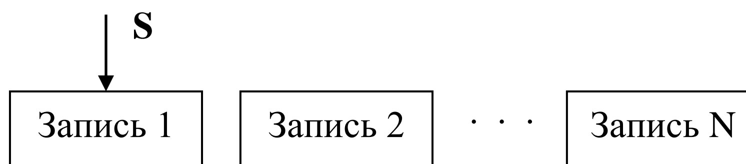
***Стек*** – линейный список, над которым допустимы только две операции:

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

Т.к. включения и исключения из стека производятся только в его начале, то нужна только одна переменная S.

На рис. 20 и 21 приведены примеры включения и исключения записи.

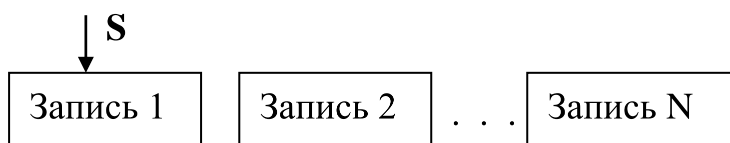
Исходный стек:




Результат:

Рис. 20. Включение записи k в стек

Исходный стек:




Результат:

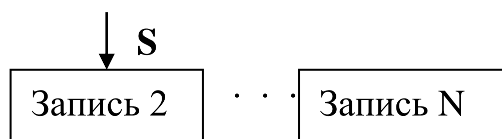



Рис. 21. Исключение записи из стека

***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. 22 и 23 приведены примеры включения и исключения записи.

Исходная очередь:

Результат:

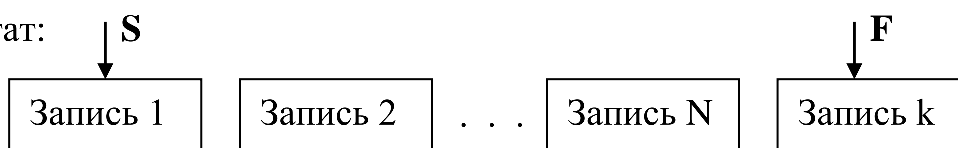
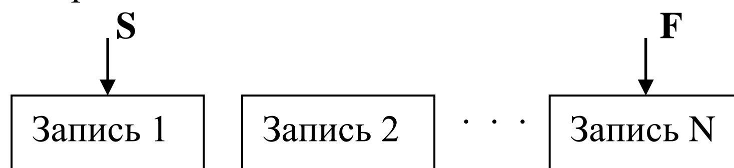



Рис. 22. Включение записи k в несвязанную очередь

Исходная очередь:




Результат:

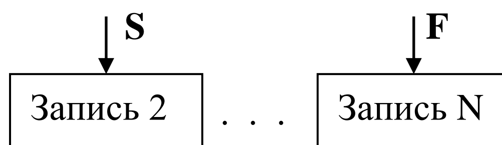
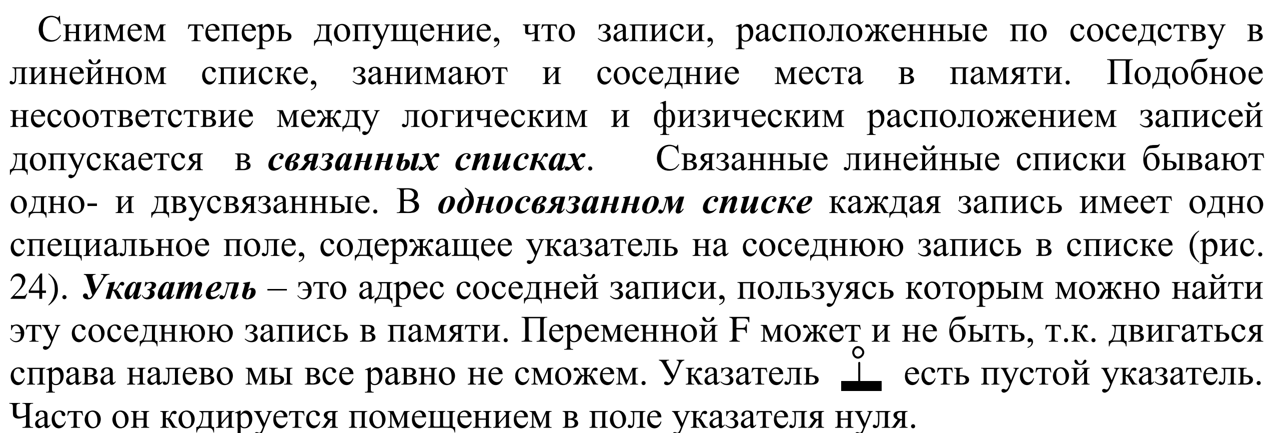



Рис. 23. Исключение записи из несвязанной очереди

## 4.3. 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 двусвязанные. В ***односвязанном списке*** каждая запись имеет одно специальное поле, содержащее указатель на соседнюю запись в списке (рис. 24). ***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. Указатель ⊥ есть пустой указатель. Часто он кодируется помещением в поле указателя нуля.

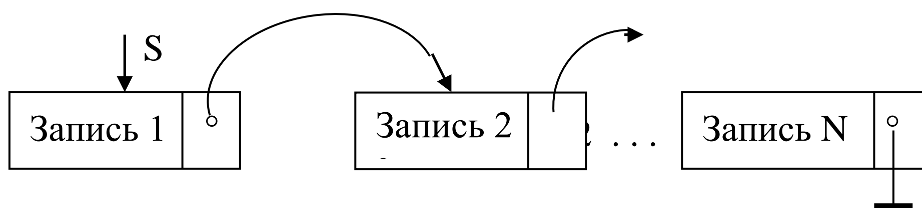



Рис. 24. Пример линейного односвязанного списка

В ***двусвязанном списке*** каждая запись имеет два поля указателей на обе соседние записи (рис. 25).

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 одно, а очередь как одно- так и двусвязанной. Операции над этими списками аналогичны операциям над соответствующими несвязанными списками.

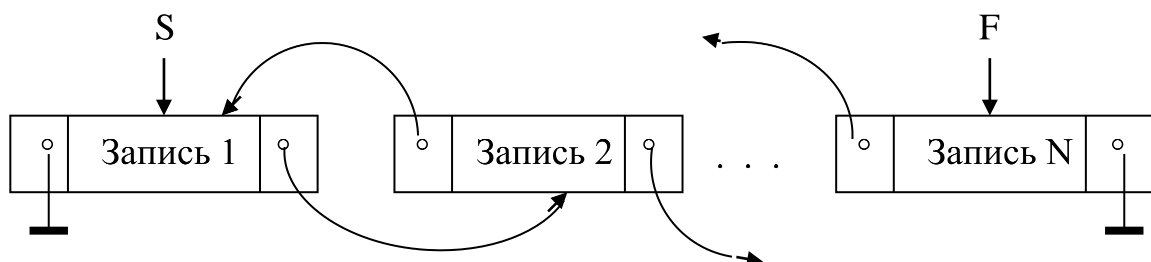



Рис. 25. Двусвязанный линейный список

В отличие от несвязанных, связанные списки могут быть нелинейными. Важнейшим видом нелинейных списков является ***дерево*** (рис. 26).

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.

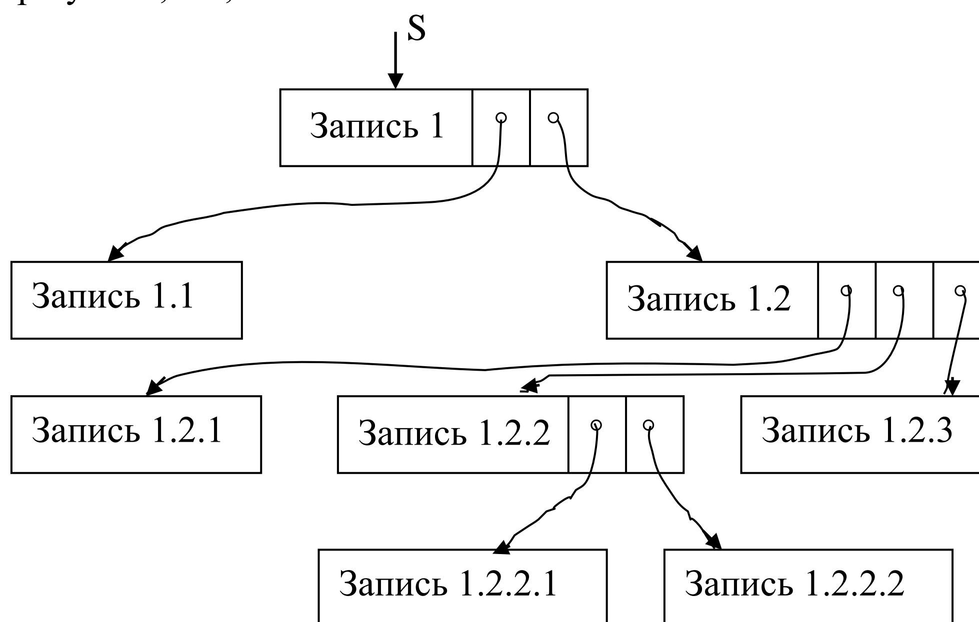



Рис.26. Пример дерева

# 5. АППАРАТНАЯ ПОДДЕРЖКА ВЗАИМОДЕЙСТВИЯ МЕЖДУ ПРОГРАММНЫМИ МОДУЛЯМИ

***Модуль*** – относительно независимая часть проектируемой системы. При разработке всякой сколько-нибудь сложной программы ее приходится разбивать на модули, называемые ***подпрограммами***. К подпрограммам относятся процедуры, а также подпрограммы, вызываемые через прерывания.

Любой ЦП имеет машинные инструкции, обеспечивающие передачу управления между модулями, а также инструкции для работы со стеком, который широко используется для передачи данных между подпрограммами.

## 5.1. Работа со стеком

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

Допустим, что для нашей программы ОС выделила область (сегмент) ОП, начиная с параграфа N 20h (рис. 27). Регистр CS содержит число 20h 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. Специальный регистр **SS**, называемый ***регистром сегмента стека***, является указателем на начало этой области. В простых программах для размещения стека ОС выделяет ту же область, что и для инструкций программы, поэтому первоначальное содержимое SS совпадает с содержимым CS.

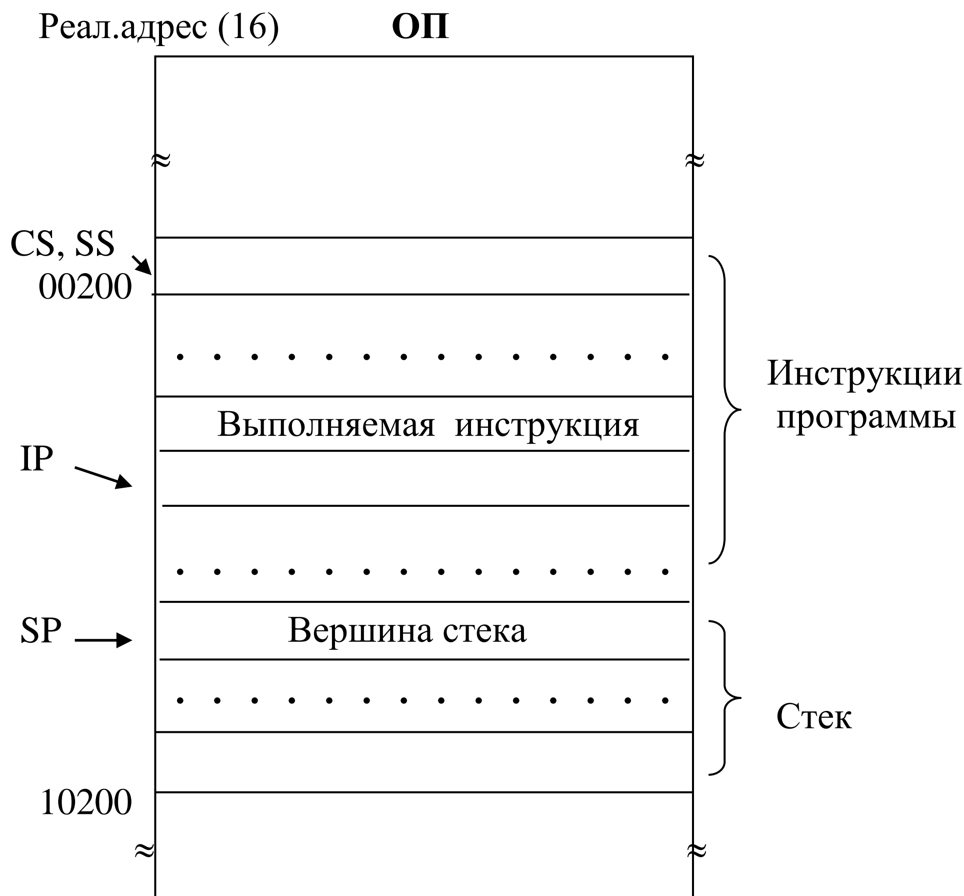



Рис. 27. Пример размещения программы в ОП

Т.к. инструкции программы записываются в сегмент, начиная с его начала, то данные стека должны находиться в конце сегмента. Для лучшего использования памяти стек “растет” в сторону меньших адресов. Смещение вершины относительно начала сегмента хранится в регистре **SP**, который называется ***указателем стека***. Первоначально операционная система помещает в SP максимально возможное число без знака, т.е. FFFF или чуть меньшее число. В результате SP указывает на "дно" стека. При добавлении слова данных в стек содержимое SP уменьшается на 2, а при исключении слова из стека – увеличивается на 2.

Включение слова данных в стек выполняет машинная инструкция **PUSH** α, где α - адрес ячейки памяти (ОП или регистровой памяти), содержимое которой следует включить в стек. Исключение слова данных из стека выполняет инструкция **POP** α, где α - адрес ячейки памяти, в которую следует считать слово данных из стека.

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.

## 5.2. Процедуры

***Процедура*** – это список инструкций, который можно вызывать из различных мест программы. Переход к процедуре называется ***вызовом***, а соответствующий переход назад называется ***возвратом***. Вызов процедуры выполняет инструкция **CALL** α , где α - адрес, по которому находится первая инструкция процедуры. Возврат осуществляет инструкция ***RET*** (рис. 28). Возврат после каждого вызова осуществляется к инструкции, которая находится в памяти сразу за инструкцией CALL .

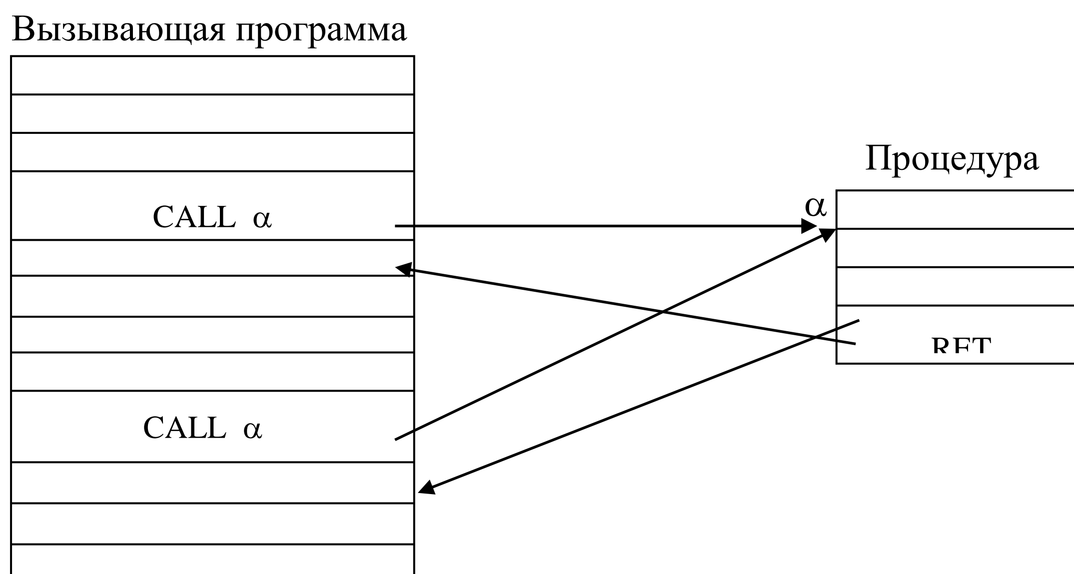



Рис. 28. Вызов процедуры и возврат из нее

При вызове процедуры необходимо выполнять следующие три требования:

1) в отличие от инструкций переходов при выполнении инструкции CALL необходимо запомнить адрес следующей инструкции, чтобы можно было 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 инструкции CALL и RET. CALL помещает адрес следующей за ней инструкции (хранится в IP) в программный стек, а RET извлекает этот адрес из стека и помещает его в указатель команды IP.

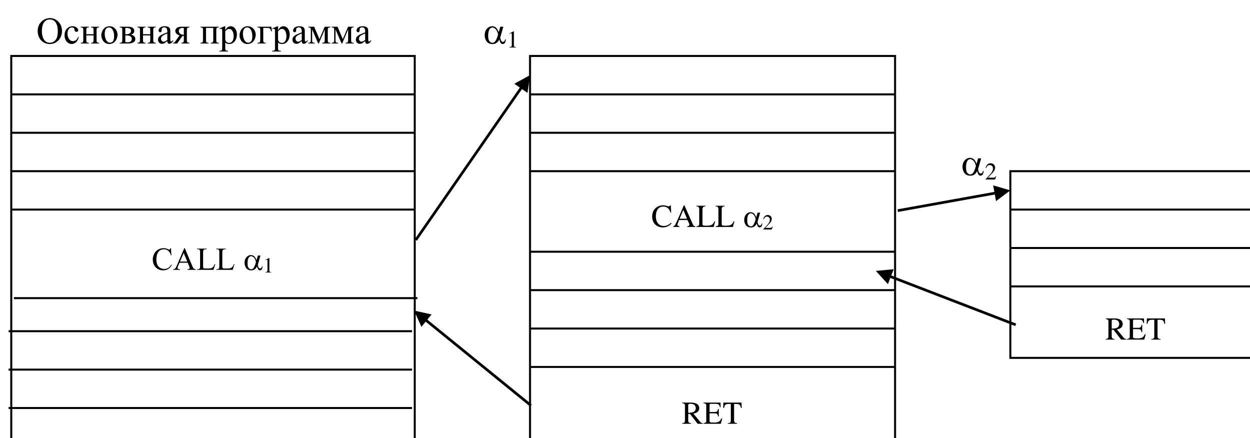



Рис. 29. Вложенный вызов процедур

Второе требование также выполняется с помощью стека. В начале процедуры инструкции PUSH помещают в стек содержимое запоминаемых регистров, а в конце ее инструкции POP извлекают из стека запомненные слова и помещают их в регистры. Порядок регистров при восстановлении обратен их порядку при запоминании.

Третье требование обеспечивается путем использования регистров общего назначения и с помощью областей ОП. Если передаваемых данных немного, то достаточно регистров, если много – то эти данные помещаются в область ОП, а начальный адрес этой области записывается в регистр. Данные, передаваемые процедуре при ее вызове называются ее ***входными параметрами***. А данные, которые передаются процедурой в вызывающую ее программу, называются ***выходными параметрами*** процедуры.

## 5.3. 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. 30). Каждый ***вектор прерываний*** занимает два слова (4 байта) ОП и соответствует своему типу прерывания. Содержимым вектора является адрес в ОП первой ячейки обработчика прерываний.

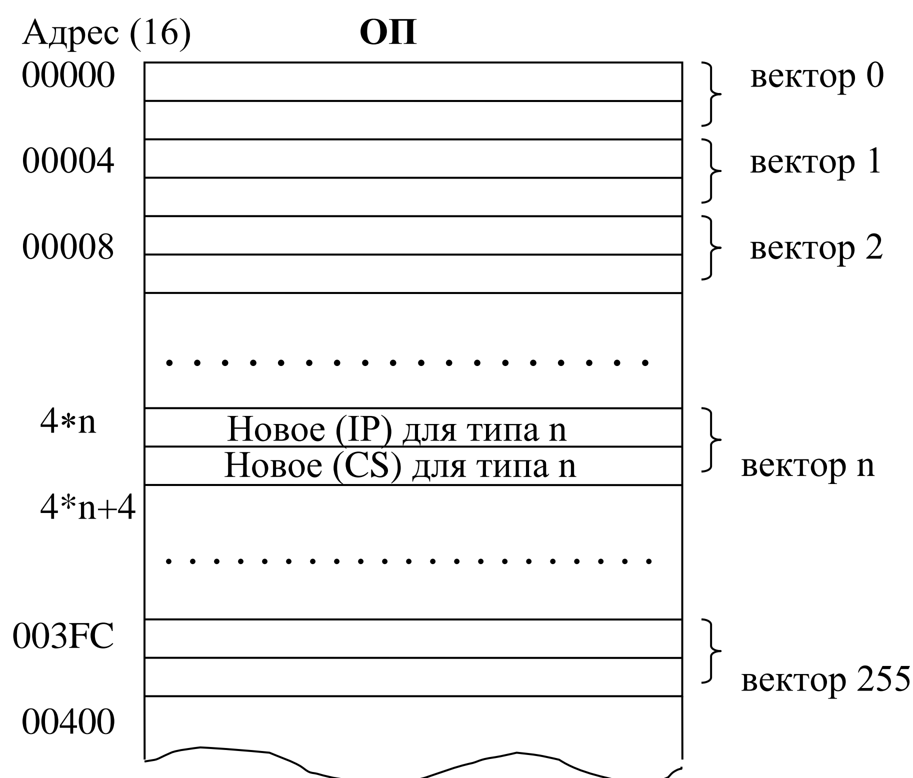



Рис. 30. Область векторов прерываний

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.

# 6. ОСНОВНЫЕ ОПЕРАТОРЫ АССЕМБЛЕРА

## 6.1. Типы операторов

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

***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.

## 6.2. Операторы обработки данных

Операторы обработки данных делятся на:
1) 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.

### 6.2.1. Арифметические операторы

На рис. 31 приведена классификация арифметических инструкций процессора i8086. Они 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.

Сложение:
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, находящемся 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
```

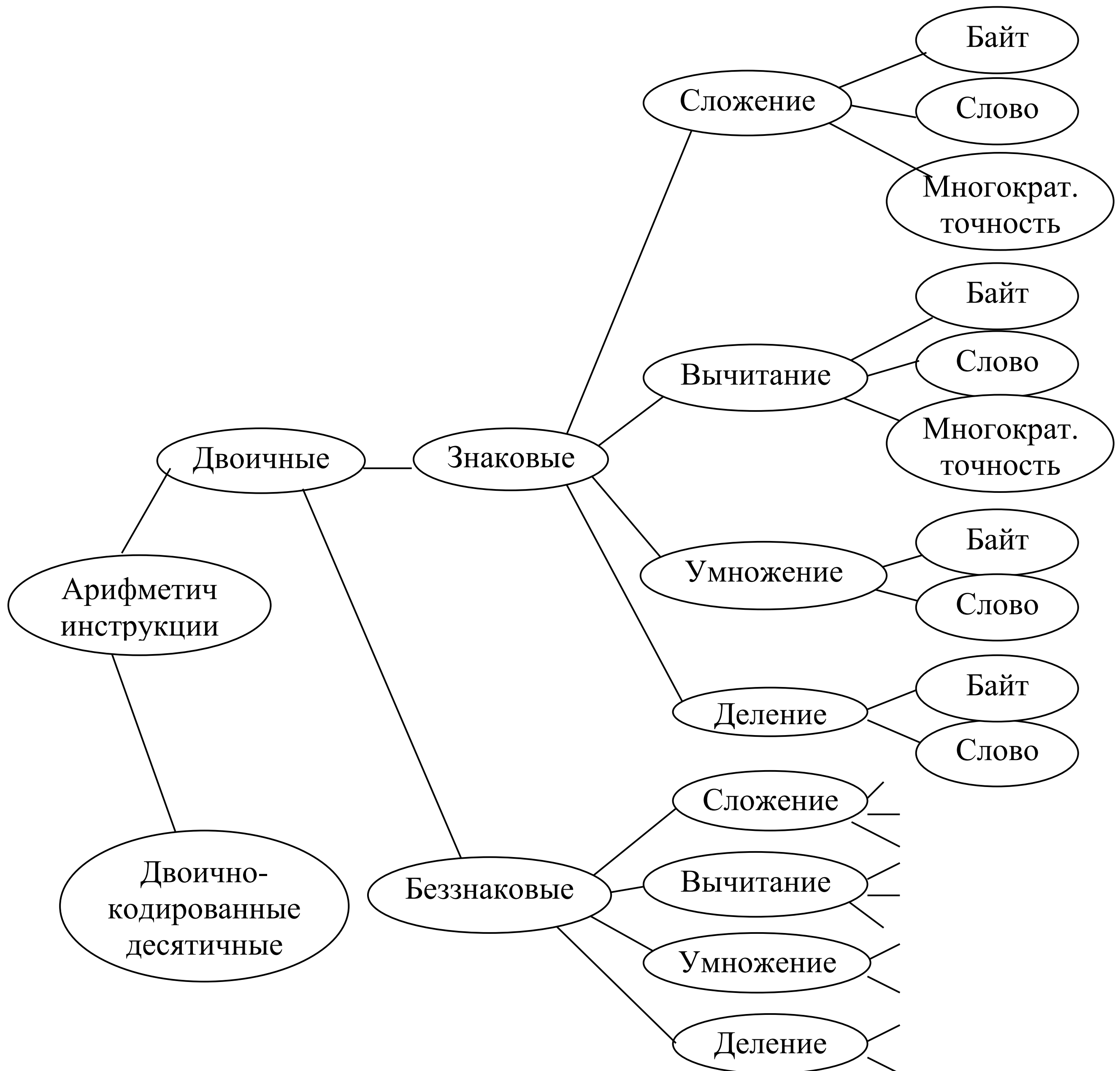



Рис. 31. Классификация арифметических инструкций i8086

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. Если 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.

### 6.2.2. 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   ; Обну-е 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.

### 6.2.3. 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 п.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
```

### 6.2.4. 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. 32).

| 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.32. 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. Т.к. 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.

Пусть, например, команда ADD выполнила следующее сложение:

```
  0010 0011 0100 0101
+ 0011 0010 0001 1001
---------------------
  0101 0101 0101 1110
```

тогда после ее выполнения получаются состояния флажков:

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

Если ADD выполнила сложение:

```
  0101 0100 0011 1001
+ 0100 0101 0110 1010
---------------------
  1001 1001 1010 0011
```

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.

### 6.2.5. 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. 33). При каждом сдвиге в освободившийся нулевой бит заносится 0. Например, пусть (AL) = 10110100b, CF = 0, тогда:

SHL AL, 1 ; 01101000 → AL, 1 → CF

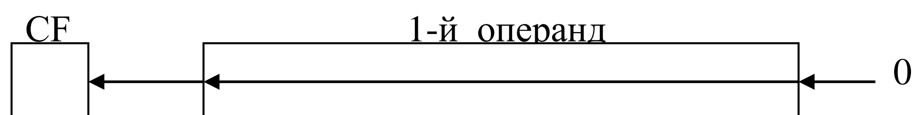



Рис. 33. Логический сдвиг влево SHL

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) = 10110100b, 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. 34). Например, пусть (AL) = 10110100b, CF = 1, тогда:

SHR AL, 1 ; 01011010 → AL, 0 → CF

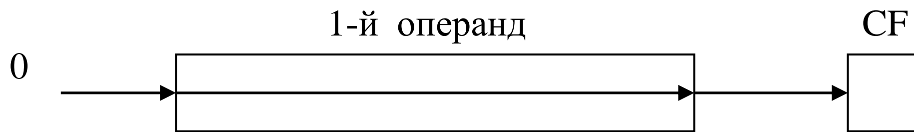



Рис. 34. Логический сдвиг вправо SHR

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

**SAR** – ***арифметический сдвиг вправо***. Данный оператор сдвигает число со знаком. При сдвиге в старшие освобождающиеся биты дублируется знак операнда (рис. 35). Например, пусть (AL) = 10110100b, CF = 1, тогда:

SAR AL, 1 ; 11011010 → AL, 0 → CF

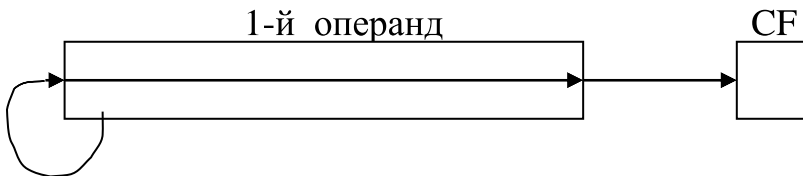



Рис. 35. Арифметический сдвиг вправо SAR

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. 36). Например, пусть (AL) = 10110100b, CF = 0, тогда:

ROL AL, 1 ; 01101001 → AL, 1 → CF

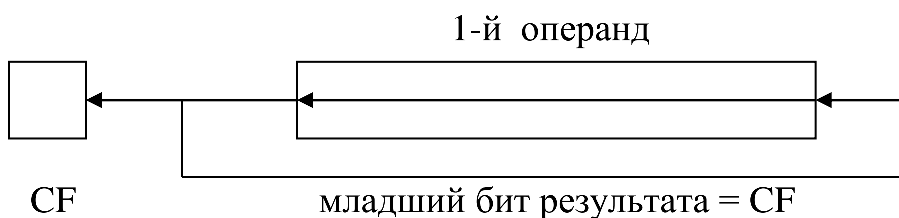



Рис. 36. Циклический сдвиг влево ROL

**ROR** – ***циклический сдвиг вправо*** (рис. 37). Например, пусть (AL) = 10110100b, CF = 1, тогда:

ROR AL, 1 ; 01011010 → AL, 0 → CF

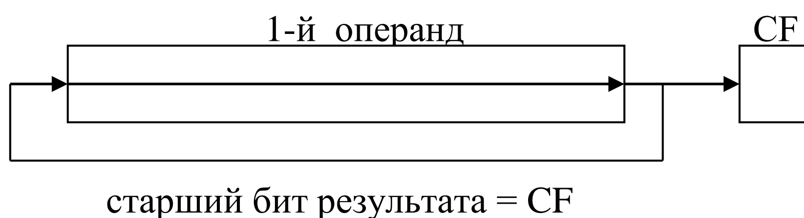



Рис. 37. Циклический сдвиг вправо ROR

**RCL** – ***циклический сдвиг влево через перенос*** (рис. 38). Например, пусть (AL) = 10110100b, CF = 1, тогда:

RCL AL, 1 ; 01101001 → AL, 1 → CF

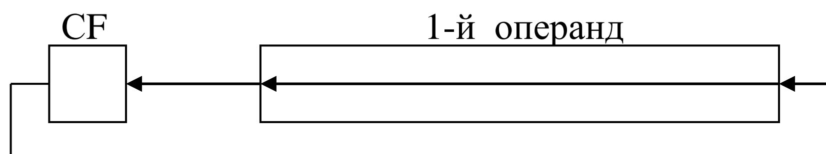



Рис. 38. Циклический сдвиг влево через перенос RCL

**RCR** – ***циклический сдвиг вправо через перенос*** (рис. 39). Например, пусть (AL) = 10110100b, CF = 1, тогда:

RCR AL, 1 ; 11011010 → AL, 0 → CF

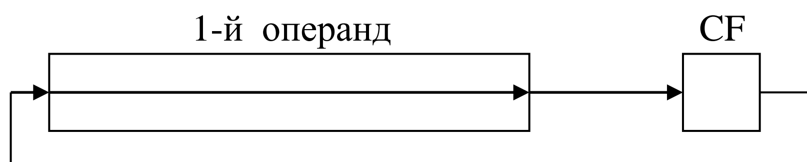



Рис. 39. Циклический сдвиг вправо через перенос RCR

### 6.2.6. 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.

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
```

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
```

Т.к. 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.

## 6.3. 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. 40). 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) x 16 + L.

Рис. 40. Расположение сегмента данных в ОП

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 операторе 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.

***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) .

## 6.4. 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.

### 6.4.1. Метки

***Метка*** – символьное имя, которое может быть присвоено элементу данных, исполнительному оператору или модулю программы. Максимальная длина метки – 31 символ. Это могут быть следующие символы:

1) 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. Ассемблер не делает разницы между заглавными и строчными буквами. Примеры меток: COUNT, Page25, $E10. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

### 6.4.2. 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. 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
```

### 6.4.3. 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).

### 6.4.4. 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
```

### 6.4.5. 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
```

## 6.5. 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.

### 6.5.1. Операторы условных переходов

Формат оператора условного перехода:

$$\text{КОП} \quad \alpha ,$$

где $\alpha$ - метка, или метка + (-) выражение, вычисление которого дает константу. Операнд $\alpha$ 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:

3) **JA** (или **JNBE**) – перейти, если больше. Условие: (CF = 0) & (ZF = 0);
4) **JAE** (или **JNB**) – перейти, если больше или равно. Условие: CF = 0;
5) **JB** (или **JNAE**) – перейти, если меньше. Условие: CF = 1;
6) **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:

7) **JG** (или **JNLE**) – перейти, если больше. Условие: (SF = OF)&(ZF = 0);
8) **JGE** (или **JNL**) – перейти, если больше или равно. Условие: SF = OF;
9) **JL** (или **JNGE**) – перейти, если меньше. Условие: SF ≠ OF;
10) **JLE**(или **JNG**) – перейти, если меньше или равно. Условие: (SF≠ OF) V (ZF=1).

Каждому условному оператору перехода соответствует противоположный по смыслу оператор. Например, приведенные на рис. 41 два фрагмента программ эквивалентны.

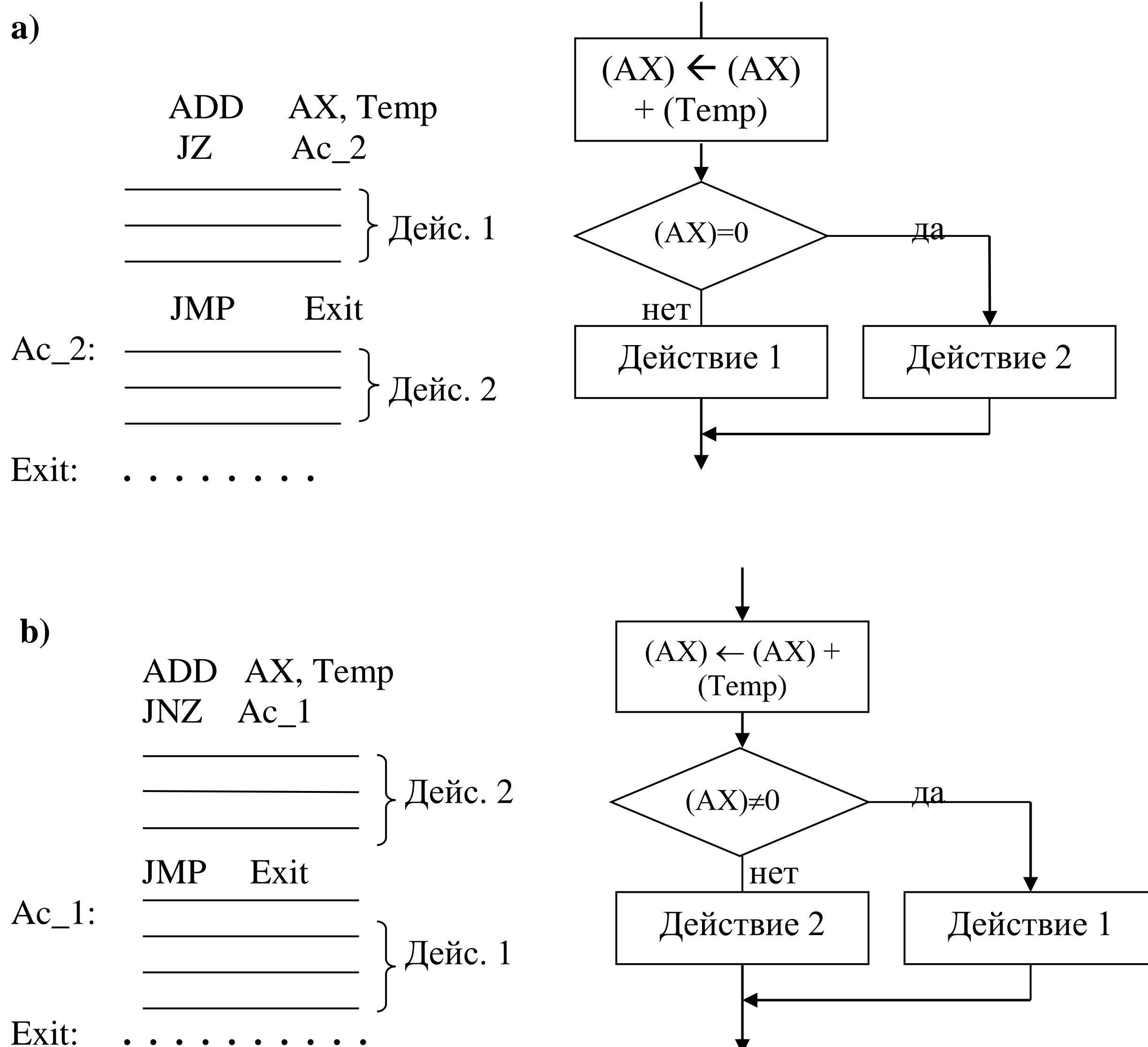



Рис.41. Фрагменты программ a и b эквивалентны

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.

### 6.5.2. 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. Т.к. 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)x16 + 1ABh.

### 6.5.3. Операторы циклов

***Цикл*** – многократное повторение группы операторов, называемых ***телом цикла***, до тех пор, пока выполняется некоторое условие (рис. 42).

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
```

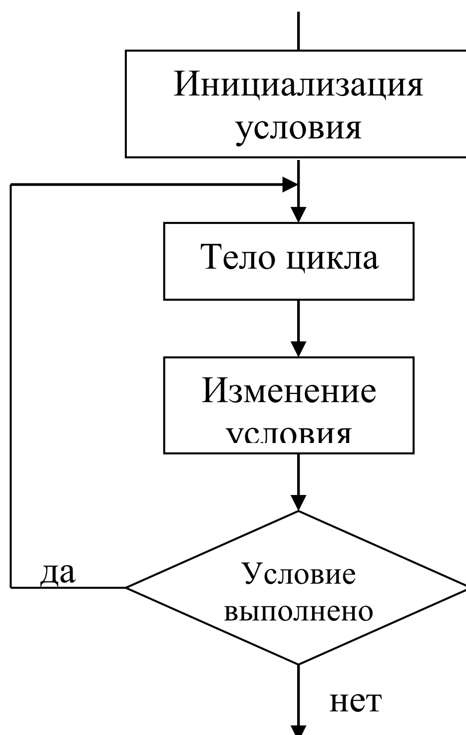



Рис. 42. Блок-схема цикла

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:

```
        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 . Т.е. 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.

### 6.5.4. 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**.

### 6.5.5. 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 .

Оператор останова **HLT** останавливает ЦП. Такие операторы часто встречаются в диагностических программах.

Оператор холостого хода **NOP** не действует ни на флаги, ни на регистры, ни на ячейки ОП. Единственное – он увеличивает содержимое IP на 1. Данный оператор имеет много применений для упрощения отладки программы. Два из них:

1) кодом оператора (90h) можно «забить» объектный код в том случае, если вам надо удалить машинную инструкцию, не транслируя программу заново;
2) можно сделать оператор NOP последним в тестируемой программе и тем самым получить удобное место для остановки трассировки.

### 6.6. Вспомогательные псевдооператоры

***Вспомогательные псевдооператоры*** – дополняют основные операторы (исполнительные и псевдооператоры) с целью уточнения их операндов. Примером такого оператора является рассмотренный в п. 6.5.2 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
```

## 6.7. 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.

# 7. ПРОЕКТИРОВАНИЕ ПРОГРАММЫ

## 7.1. Введение

Разработка любой программы включает три этапа:

1) проектирование;
2) кодирование;
3) отладка.

Целью этапа ***проектирование*** является получение алгоритма решения задачи по переработке информации. Целью этапа ***кодирование*** является запись полученного алгоритма на выбранном языке (языках) программирования. ***Отладка*** программы включает ***тестирование*** (проверку правильности ее работы) и ***исправление ошибок***, найденных в результате тестирования.

Только для очень небольших программ этапы разработки программы выполняются строго последовательно во времени (рис. 43). На практике эти этапы выполняются ***параллельно*** (одновременно). Т.е. одновременно какая-то часть программы проектируется, другая ее часть кодируется, а третья – отлаживается. Естественно, что проектирование при этом несколько опережает этап кодирования, а кодирование – отладку (рис. 44).

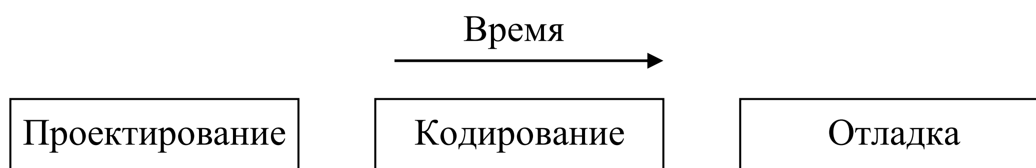



Рис. 43. Последовательное выполнение этапов разработки программы

Существующие технологии программирования различаются между собой реализацией указанных трех этапов. Главным из них является этап проектирования. При правильном его выполнении этапы кодирования и отладки выполняются достаточно просто. И наоборот, недостаточно хорошо выполненное проектирование программы всегда предшествует черезмерно продолжительным этапам кодирования и отладки.

Рис. 44. Параллельное выполнение этапов разработки программы

После завершения разработки программы начинается ее ***сопровождение***. Во время сопровождения решаются две основные задачи:

1) исправление выявляющихся в ходе эксплуатации ошибок;
2) корректировка программы, вызванная изменениями в решаемой ею задаче по переработке информации.

## 7.2. Результаты проектирования

В процессе проектирования программы обычно получают три вида документов: 1) системную схему; 2) дерево подпрограмм; 3) алгоритмы подпрограмм.

***Системная схема*** – небольшой документ, показывающий взаимодействие между программой – с одной стороны, и устройствами ввода-вывода и файлами во внешней памяти – с другой. Иными словами, системная схема описывает кратко интерфейс программы. На рис. 45 приведен пример системной схемы для программы, выполняющей запросы своих пользователей по предоставлению сведений о сотрудниках организации.

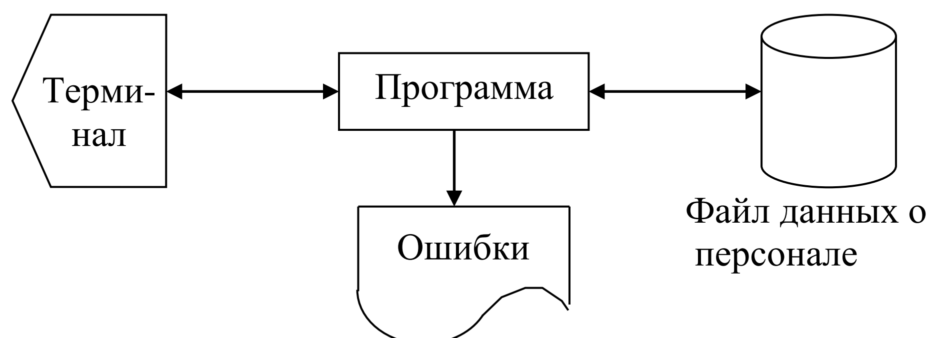



Рис. 45. Пример системной схемы

Краткое описание интерфейсов программы, предоставляемое системной схемой, необходимо дополнить их подробным словесным описанием. Для этого необходимо описать структуру входных и выходных сообщений программы. ***Входные сообщения*** – сообщения, вводимые в программу пользователем с помощью клавиатуры терминала. ***Выходные сообщения*** – сообщения, выводимые программой пользователю (на экран его терминала). Кроме того, необходимо привести описание файлов, с которыми работает программа.

Два интерфейса не показываются на системной схеме: 1) между программой и ЦП; 2) между программой и ОП. Для описания этих интерфейсов используются ***параметры эффективности*** программы: 1) затраты времени ЦП на выполнение программы; 2) объем ОП, требуемый программе. В начале проектирования необходимо задать предельные значения этих двух параметров.

***Дерево подпрограмм*** показывает – из каких подпрограмм состоит программа и как эти подпрограммы связаны по управлению. ***Связь по управлению*** – вызов (инициирование) одной подпрограммы другой. В состав подпрограмм входят процедуры, а также подпрограммы, вызываемые через программные прерывания. Причем системные подпрограммы, вызываемые через прерывания, в дерево подпрограмм обычно не включают.

Корнем дерева подпрограмм является процедура, называемая ***главной*** (***основной***) ***подпрограммой***. На следующем уровне дерева находятся подпрограммы, которые вызываются из основной подпрограммы и т.д. (рис. 46).

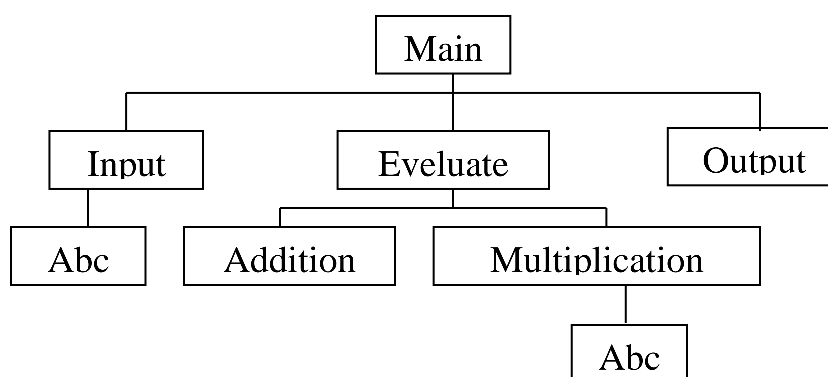



Рис. 46. Пример дерева подпрограмм

Как видно из рис. 46, одна и та же подпрограмма (Abc) может быть многократно изображена на одном и том же дереве подпрограмм. Реально в памяти находится лишь одно тело подпрограммы, а ее повторения в дереве делаются для того, чтобы не затемнять рисунок пересечениями. (Дерево без пересечений воспринимается намного лучше.)

Проектирование любой подпрограммы включает два этапа: 1) определение интерфейсов; 2) получение алгоритма. На этапе ***определения интерфейсов*** подпрограмма рассматривается как “черный ящик” и для нее определяются границы (интерфейсы) с внешним миром. Основные подсистемы внешнего мира, с которыми взаимодействует подпрограмма:

1) подпрограмма (подпрограммы), вызывающая данную подпрограмму;
2) структуры данных (переменные, файлы), с которыми работает данная подпрограмма, но которые определены вне ее;
3) пользователь разрабатываемой программы.

Из перечисленных трех типов внешних подсистем только вызывающая подпрограмма является обязательной. Остальные подсистемы (пользователь и внешние структуры данных) при разработке конкретной подпрограммы могут отсутствовать. В этом случае интерфейс подпрограммы (процедуры) – перечень данных, которыми она обменивается с вызывающей ее программой. Сюда относятся входные и выходные параметры процедуры, а также области данных, на которые эти параметры указывают. Чем этот перечень меньше, тем лучше. (В идеальном случае подпрограмма вообще не имеет информационного объмена с вызывающими ее программами, но подобные подпрограммы могут выполнять лишь ограниченный набор функций.)

Результаты этапа выявления интерфейсов подпрограммы могут быть зафиксированы в системной схеме подобно тому, как это делалось для всей разрабатываемой программы. Но обычно это не делают, а ограничиваются описанием интерфейсов в виде вводных комментариев подпрограммы (п. 8.6). В любом случае точное описание интерфейсов подпрограммы должно предшествовать разработке ее алгоритма. Иными словами, не выполнив постановку задачи, нельзя приступать к ее решению.

Одним из приемов, упрощающих восприятие логики алгоритма, является последовательное применение регистров для информационного обмена между подпрограммами. Рекомендуется следующее использование регистров:

DL, DX – для передачи байта или слова в подпрограмму;
AL, AX – для возвращения байта или слова из подпрограммы;
BX:AX – для возвращения двойного слова;
DS:DX – передача и возвращение адресов;
CX - счетчик повторения и другие счетчики;
флаг CF – устанавливается при ошибке, при этом код ошибки возвращается в одном из регистров, например, в AL или в AX.

На этапе ***разработки алгоритма*** выявляется совокупность шагов, обеспечивающих перевод входных данных подпрограммы в ее выходные данные. Для представления алгоритма может быть использован один из

специально предназначенных для этого языков, например, ***язык блок-схем***. На рис. 47 приведены основные символы (блоки) этого языка.

Основным требованием, предъявляемым к алгоритму, является его простота. Т.к. чем проще алгоритм программы, тем проще ее кодирование, отладка и сопровождение (эксплуатация). Для того, чтобы обеспечить простоту алгоритма, необходимо, чтобы он удовлетворял требованиям структурного программирования.

## 7.3. Структурное программирование

***Структурное программирование*** – построение программ, алгоритмы которых включают только ***допустимые управляющие структуры***. Перечислим эти структуры:

1) ***цепочка*** – простая последовательность блоков типа “действие” и “вывод” (рис. 48 а);

2) ***двухальтернативный выбор*** (структура **IF-THEN-ELSE**) – в зависимости от того, выполняется ли условие или нет, осуществляется действие 1 или действие 2 (рис. 48 б);

3) ***многоальтернативный выбор*** (структура **CASE**) – в зависимости от того, какое из взаимоисключающих условий 1...N выполняется, осуществляется действие 1...N. Если ни одно из этих условий не выполняется, осуществляется действие (N+1) (рис. 48 в);

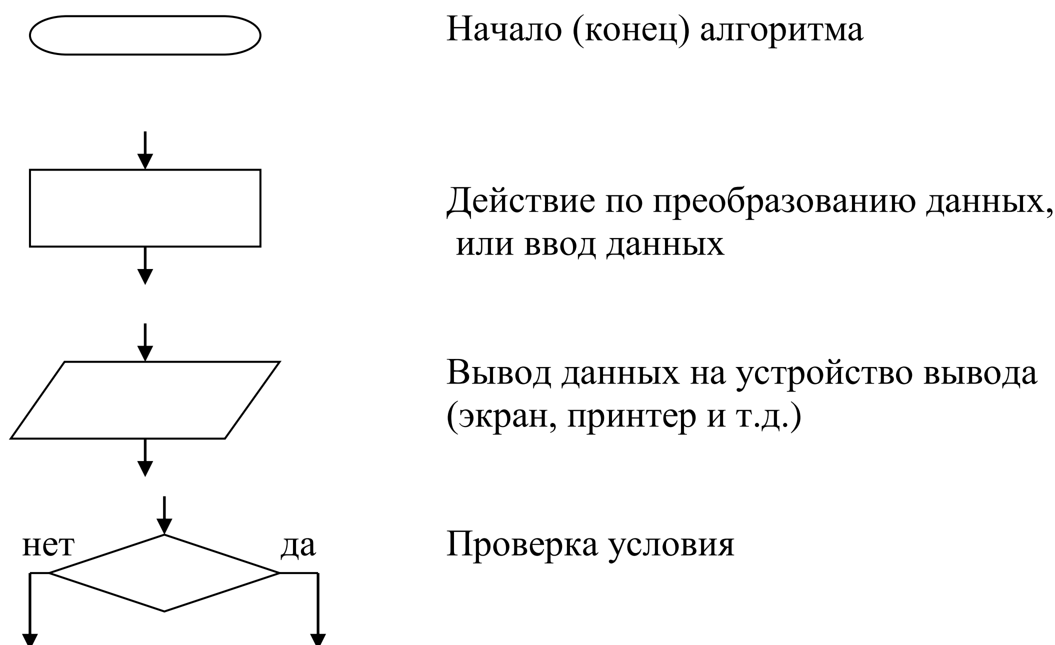



Рис. 47. Основные элементы блок-схем

4) ***цикл*** **ПОКА** (структура **DO-WHILE**) – повторение действия, образующего тело цикла, до тех пор, пока выполняется заданное условие (рис. 49 а). Для того, чтобы цикл был конечным, тело цикла должно влиять на выполнение условия;

5) ***индексный цикл*** (частный случай цикла ПОКА) – повторение действия, образующего тело цикла, заданное число N раз (рис. 49 б);

6) ***цикл*** **ПОКА** с выходом – повторение действия, образующего тело цикла, до тех пор, пока выполняется заданное условие 1. Если после очередного выполнения тела цикла будет выполнено дополнительное условие 2, осуществляется “досрочный” выход из цикла (рис. 49 в);

7) ***цикл*** **ДО** (структура **DO-UNTIL**) – повторение действия, образующего тело цикла, до тех пор, пока не выполнится заданное условие (рис. 49 г). В отличие от цикла ПОКА, тело цикла ДО выполняется, по крайней мере, один раз.

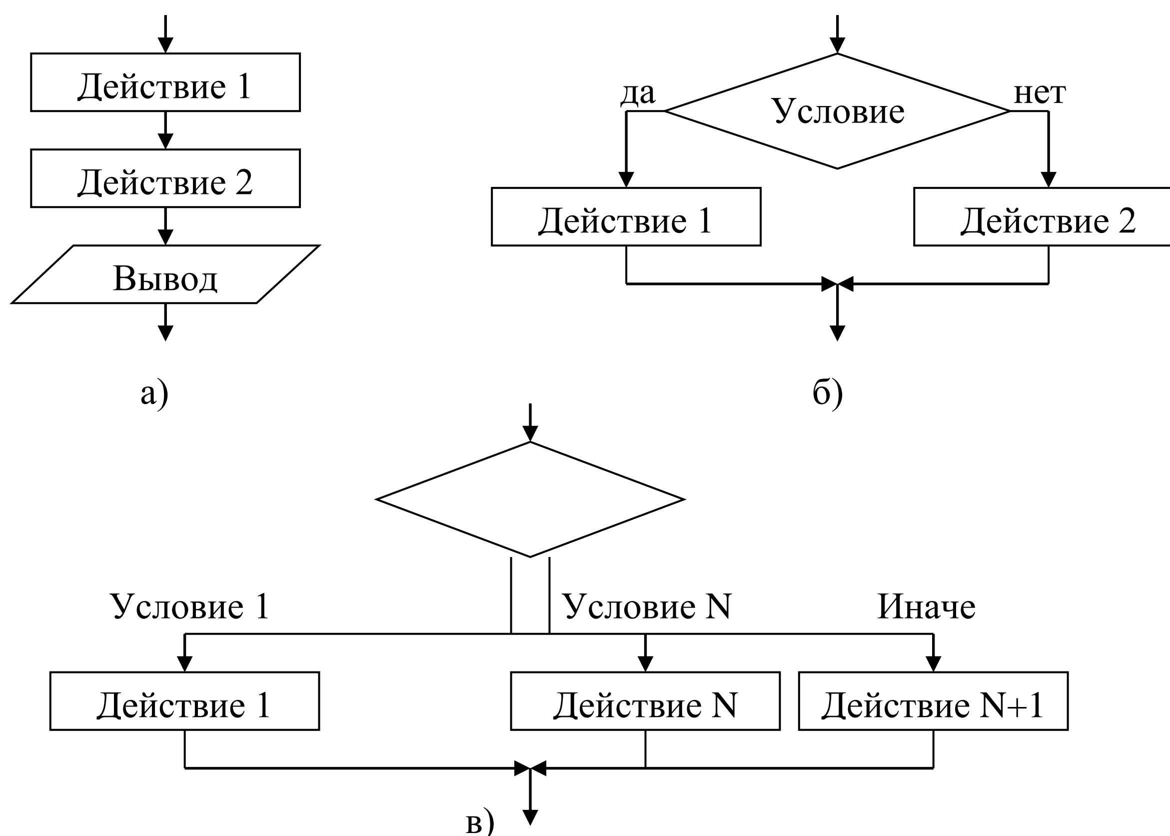



Рис. 48. Ациклические управляющие структуры: а) цепочка; б) двухальтернативный выбор; в) многоальтернативный выбор

Важной особенностью всех допустимых управляющих структур является то, что каждая из них имеет один вход и один выход.

На основе любой из допустимых структур может быть получена составная управляющая структура. Для этого достаточно блок “действие”, имеющийся в этой структуре, заменить на требуемую структуру. Нетрудно заметить, что полученная составная структура также имеет только один вход и только один выход и, следовательно, также отвечает требованиям структурного программирования. Используя подобную вложенность, можно записать алгоритм любого размера. На практике это реализуется путем вложенности алгоритмов подпрограмм.

Что касается размера алгоритма подпрограммы, то желательно, чтобы он был невелик (примерно 10 блоков). В любом случае данный алгоритм должен умещаться на одной странице, т.к. иначе восприятие логики алгоритма будет затруднено. Данное ограничение нетрудно обеспечить: любую допустимую управляющую структуру в алгоритме можно заменить одним действием, поручив ее выполнение “дочерней” подпрограмме.

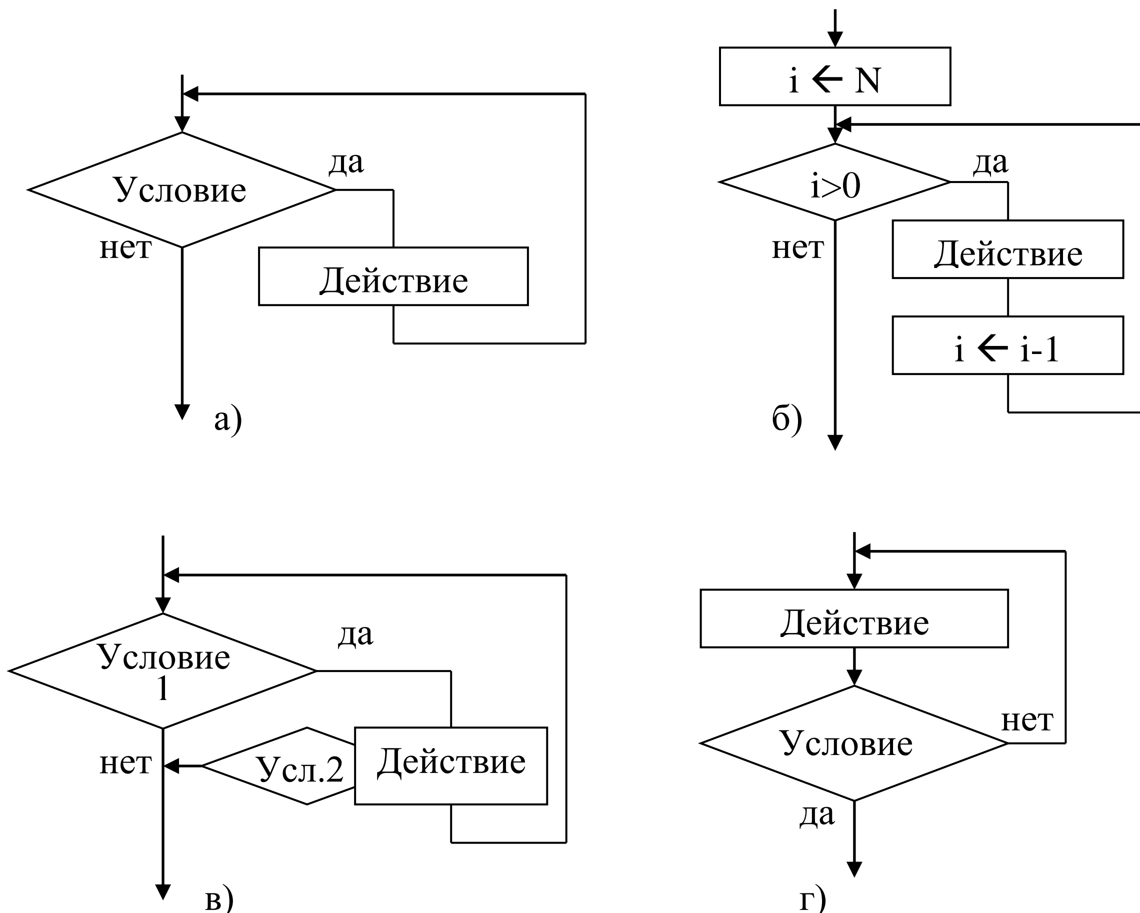



Рис. 49. Циклические управляющие структуры: а) цикл ПОКА; б) индексный цикл; в) цикл ПОКА с выходом; г) цикл ДО

В качестве примера рассмотрим алгоритм решения следующей простой задачи. Пусть программа должна ввести с клавиатуры два числа x и y, а затем

вывести на экран сообщение о том, какое число больше. После этого программа должна попрощаться. Алгоритм программы приведен на рис. 50.

В этом алгоритме управляющей структурой верхнего уровня является цепочка, состоящая из трех элементов, средний из которых представляет собой вложенную управляющую структуру – двухальтернативный выбор. Одна из ветвей этого выбора также представляет собой вложенную структуру (двухальтернативный выбор).

В качестве второго примера рассмотрим блок-схему процедуры ввода шестнадцатеричной цифры (рис. 51). Данная процедура выполняет ввод символа с клавиатуры. Если этот символ является шестнадцатеричной цифрой, процедура получает на основе кода ASCII значение цифры, помещает ее в один из регистров (для передачи в качестве выходного параметра), а также выводит цифру на экран. Если символ не является шестнадцатеричной цифрой, цикл повторяется.

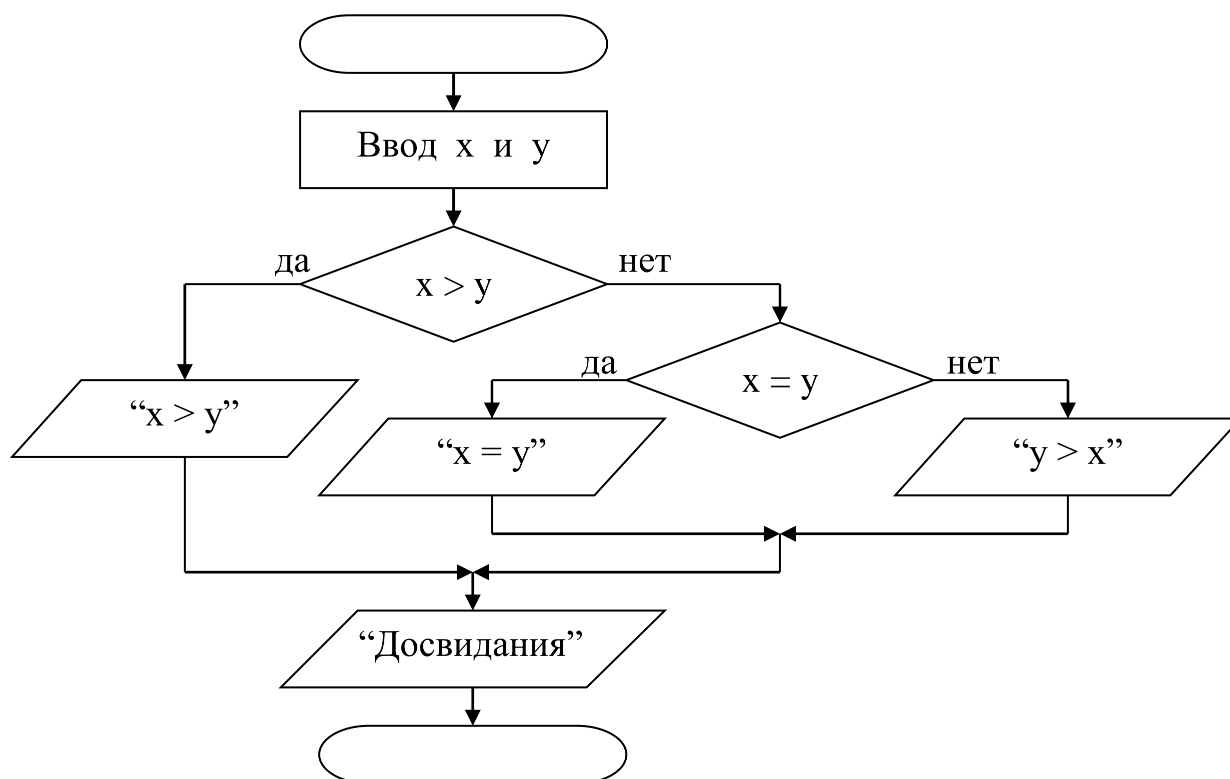
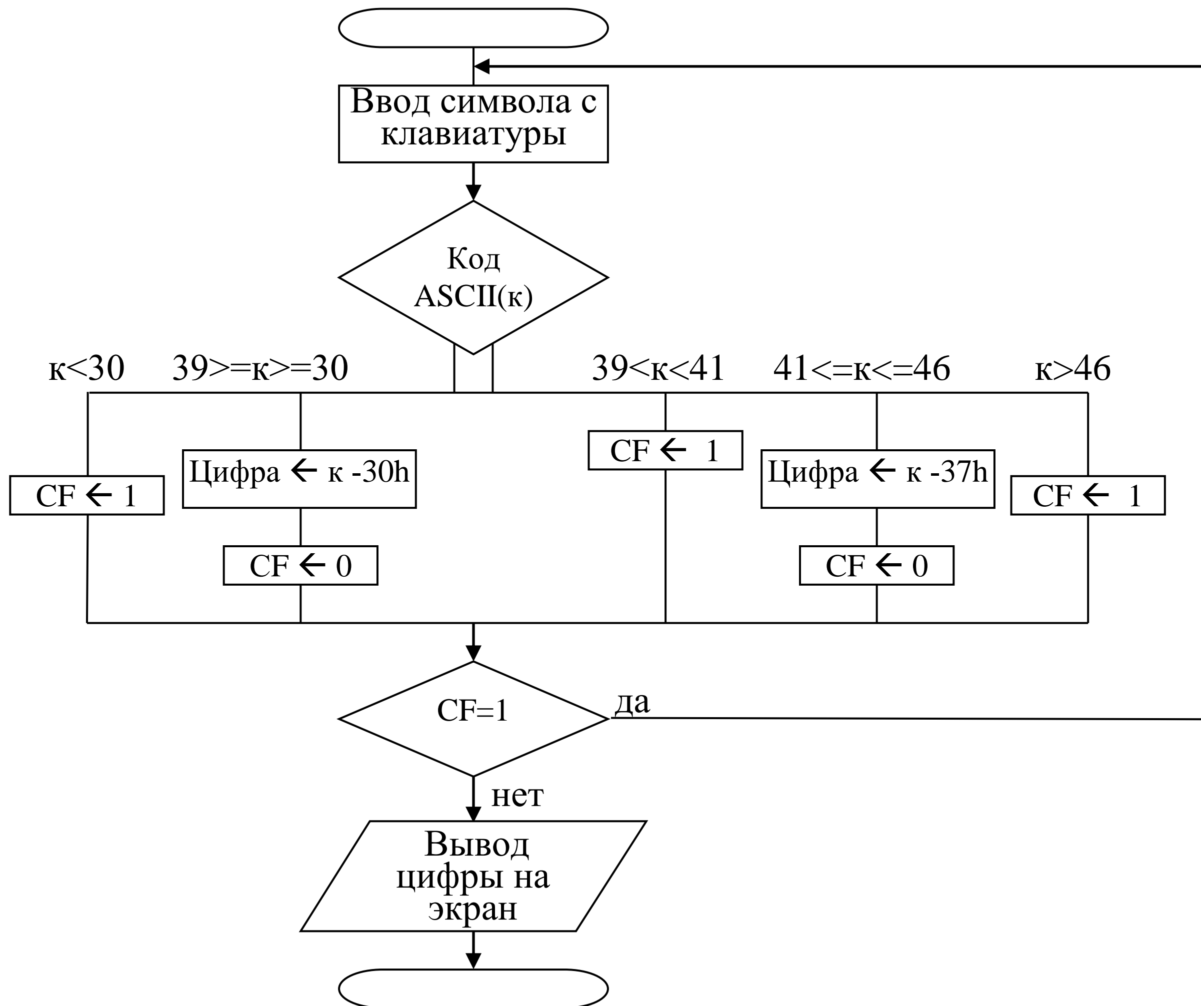



Рис. 50. Алгоритм программы сравнения двух чисел

В этом алгоритме 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 (в качестве флага ошибки обычно используют флаг переноса CF). Телом цикла является цепочка из этапа “Ввод символа с клавиатуры” и конструкции многоальтернативного выбора. В каждой ветви этого выбора устанавливается значение флага ошибки (CF=0 – введена цифра, CF=1 – цифры нет). Подобное

использование флагов широко распространено на практике для обеспечения структурности алгоритмов программ.




CF - флаг ошибки (0 - ошибки нет, 1 - ошибка есть)

Рис. 51. Алгоритм ввода одной шестнадцатеричной цифры

## 7.4. Методы проектирования программ

Применение любого метода проектирования начинается с получения системной схемы, т.к. не уточнив задачу, которую должна решать программа, дальнейшее ее проектирование не имеет смысла. Другие результаты проектирования (дерево подпрограмм и алгоритмы подпрограмм) могут быть получены различным образом во времени, в зависимости от используемого метода проектирования. Все эти методы можно разбить на три группы, различающиеся выбором объектов (подпрограмм), подлежащих проектированию:

1) проектирование “сверху-вниз”;
2) проектирование “снизу-вверх”;

3) проектирование “из центра”.

В методе проектирования **сверху вниз** построение дерева подпрограмм начинается с корня дерева, т.е. с главной подпрограммы. Для этой подпрограммы разрабатывается алгоритм, отвечающий требованиям структурного программирования. После этого принимается решение о способе реализации каждого этапа алгоритма. При этом относительно простые этапы алгоритма должны кодироваться в рамках данной подпрограммы, а более сложные этапы должны быть реализованы путем вызова соответствующих подпрограмм. Имена этих подпрограмм заносятся в дерево подпрограмм. Далее выбирается любая нерассмотренная подпрограмма и для нее выполняется разработка алгоритма. Подобный рост дерева подпрограмм продолжается до тех пор, пока не останется подпрограмм, требующих алгоритмизации.

В методе проектирования **снизу вверх** первоначально проектируются подпрограммы, выполняющие сравнительно простые функции. (Разработка многих из этих подпрограмм сводится к выбору готовых подпрограмм, представленных в соответствующих библиотеках.) Далее разрабатываются подпрограммы, выполняющие более сложные функции и пользующиеся услугами ранее разработанных подпрограмм. Этот процесс продолжается до тех пор, пока очередная проектируемая подпрограмма не будет решать исходную задачу по переработке информации.

В методе проектирования **из центра** первоначально разрабатываются подпрограммы, предназначенные для решения задач по переработке информации, которые будут полезны (предположительно) для решения основной задачи. Но в отличие от метода “снизу-вверх” эти подпрограммы пользуются услугами таких подпрограмм, по крайней мере часть из которых на данный момент времени еще не разработана. Далее выполняется проектирование этих подпрограмм (движение “вниз”), а также проектирование вышестоящих подпрограмм (движение “вверх”). Движение “вверх” выполняется и заканчивается аналогично методу “снизу-вверх”, а движение “вниз” – аналогично методу “сверху-вниз”.

# 8. КОДИРОВАНИЕ И ОТЛАДКА

## 8.1. Кодирование управляющих структур

На этапах кодирования программы и ее отладки реализуются те преимущества, которые предоставляет хорошо проведенное ее проектирование. Наличие в программе лишь управляющих структур, допускаемых структурным программированием, в значительной степени сводит процесс кодирования к “механическому” отображению этих структур на выбранном языке программирования. Ниже рассматривается кодирование основных управляющих структур на языке ассемблера.

Кодирование цепочки не требует применения каких-либо управляющих операторов. Оно выполняется простым объединением операторов, кодирующих действия, образующие цепочку.

Кодирование двухальтернативного выбора основано на использовании операторов условного перехода. Пример такого кодирования приведен на рис.52. Другой пример рассматривался ранее в п.6.5.1 (см. рис.41).

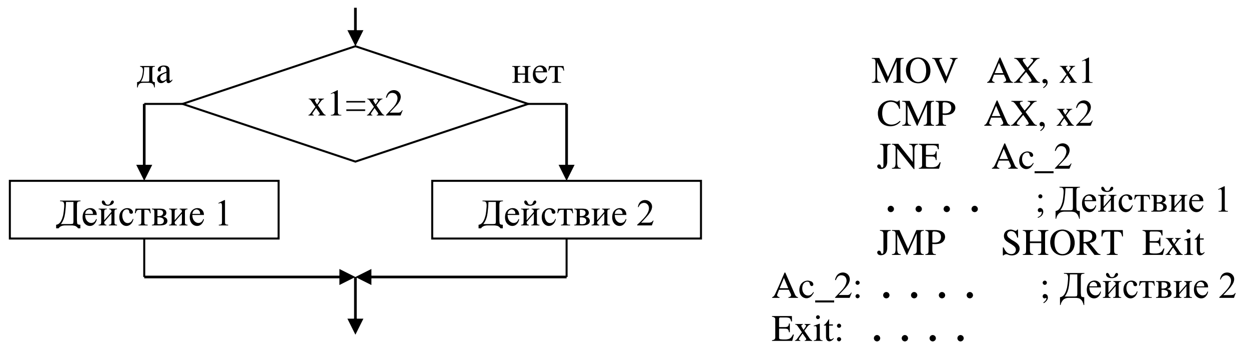



Рис. 52. Пример кодирования двухальтернативного выбора

Кодирование многоальтернативного выбора также основано на использовании операторов условного перехода. Соответствующий пример приведен на рис. 53.

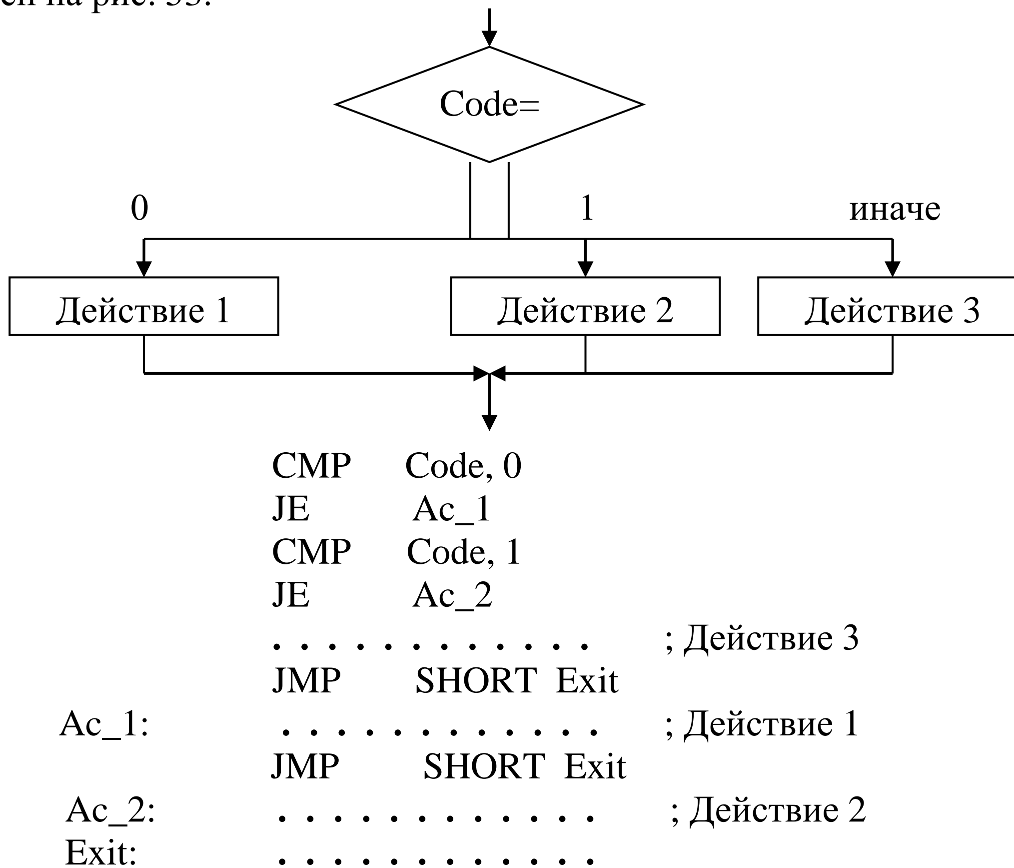



Рис. 53. Пример кодирования многоальтернативного выбора

Пример кодирования цикла ПОКА приведен на рис. 54. Кодирование частного случая цикла ПОКА - индексного цикла выполняется с помощью оператора LOOP. Оно было рассмотрено ранее в п. 6.5. Для кодирования индексного цикла с выходом могут быть использованы операторы LOOPNZ и LOOPZ. Соответствующий пример приведен в п. 6.5.

Кодирование цикла ДО выполняется с помощью оператора условного перехода. Пример такого кодирования приведен на рис. 55.

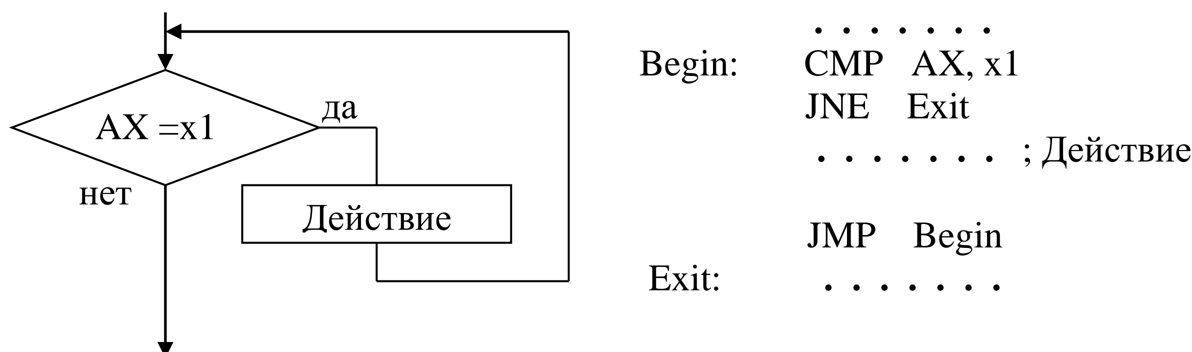



Рис. 54. Пример кодирования цикла ПОКА

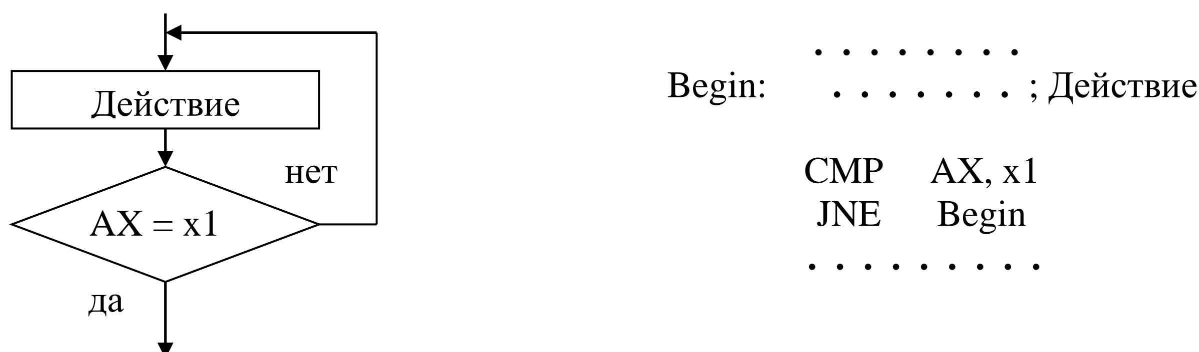



Рис. 55. Пример кодирования цикла ДО

## 8.2. Этапы преобразования программы

На рис. 56 приведены этапы получения и выполнения машинной программы. На первом из этих этапов программист вводит в память машины текст программы, записанный на языке программирования, например, на ассемблере. Подобный текст называется ***исходной программой***.

Для удобства программиста исходная программа может быть записана в виде не одного, а нескольких файлов, называемых ***исходными модулями***. Перечень исходных модулей программы, а также перечень процедур внутри каждого из этих модулей, образуют ***файловую структуру программы***.

Исходные модули – наиболее крупные модули ассемблерной программы. В принципе любую такую программу можно поместить в единственный файл с

расширением .asm . Но на практике это очень неудобно, т.к. трудно ориентироваться в большом количестве подпрограмм, выполняющих разнородные функции. Намного проще, если в каждом конкретном файле находятся лишь процедуры, выполняющие “родственные” функции, например, операции обмена с конкретным типом периферийного устройства. Программист сам определяет – как разбить свою программу на модули. Более того, он может программировать свои исходные модули на разных языках.

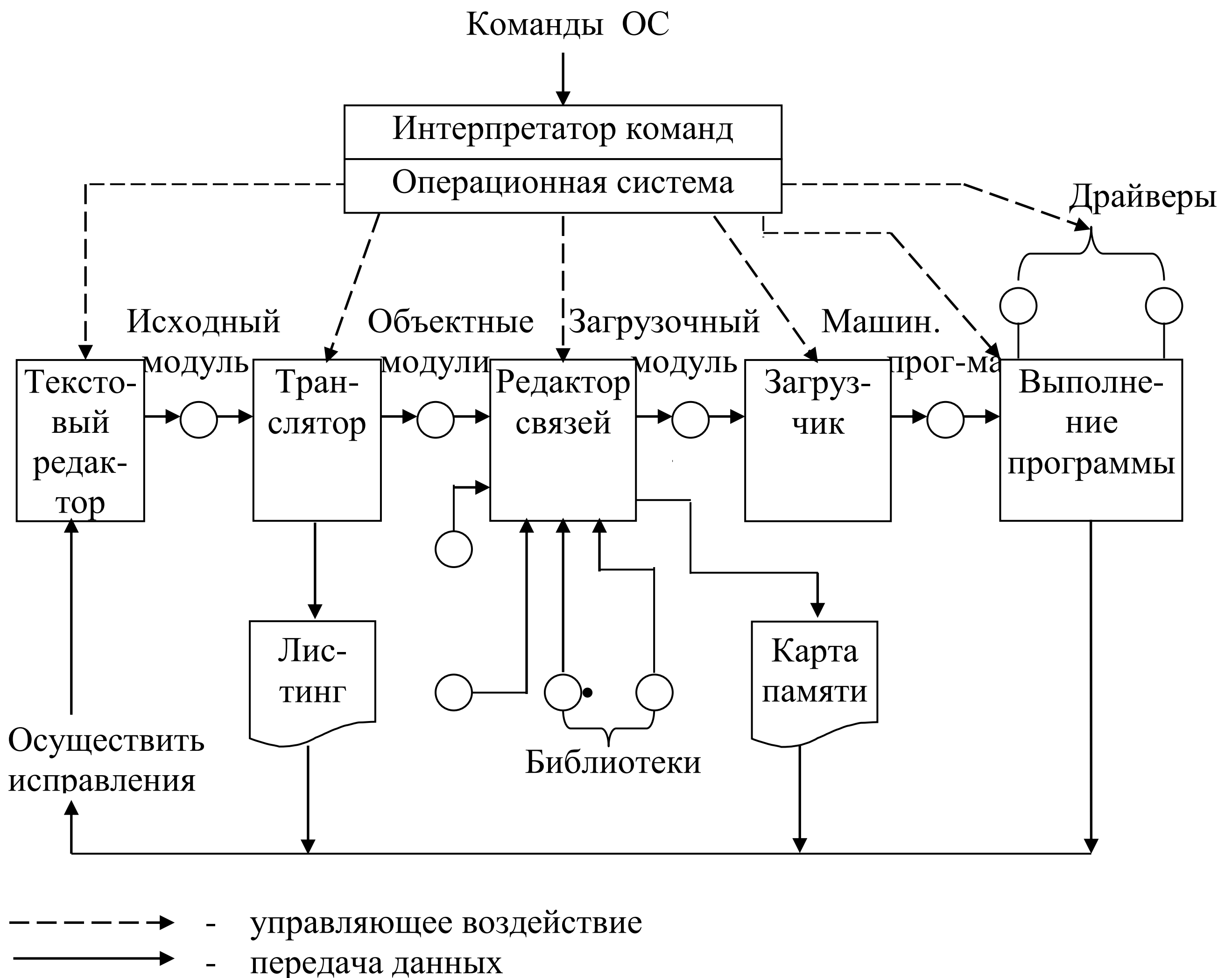



Рис.56. Общая схема создания и выполнения программы

Не существует специальных псевдооператоров, обозначающих начало исходного модуля на ассемблере. А в качестве последнего оператора любого исходного модуля всегда записывается псевдооператор **END**. Если исходный файл является главным (содержит главную подпрограмму), то он завершается псевдооператором END с операндом, который задает точку входа в программу. При этом если выполнение программы должно начинаться с первой инструкции главной подпрограммы, то в качестве операнда END задается имя этой подпрограммы.

Для создания своего исходного модуля программист использует ***текстовый редактор***. Эта системная программа записывает текст программы, набираемый на клавиатуре в коде ASCII, в обычный текстовый файл.

Исходный модуль является входными данными для системной программы, называемой ***транслятором***. Трансляторы делятся на ***ассемблеры*** и ***компиляторы***. Исходный модуль для транслятора-ассемблера пишется на языке-ассемблере. А исходный модуль для компилятора пишется на языке программирования высокого уровня.

В результате одного выполнения транслятора исходный программный модуль преобразуется в ***объектный программный модуль***. При этом частично решается задача получения последовательности машинных инструкций, отображающих алгоритм модуля. Транслятор окончательно записывает коды операций машинных инструкций, 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 – инструкция 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 инструкция процедуры находится в ОП).

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.

В том случае, если исходный модуль записан на ассемблере, для описания внешних выходных меток используется псевдооператор **PUBLIC**. Его структура:

PUBLIC <имя> [ , . . . ] ,

где наличие квадратных скобок означает, что одним псевдооператором PUBLIC можно задать несколько внешних выходных меток. В программе псевдооператоры PUBLIC располагаются в любом месте, но до определения соответствующего объекта (процедуры или данного).

Для описания внешних входных меток используется псевдооператор **EXTRN**. Его структура:

EXTRN <имя>:<тип> [ , . . . ] ,

где <имя> - метка, определенная в другом исходном модуле, а <тип> задается следующим образом:

1) для переменной это одно из значений: BYTE, WORD или DWORD;
2) для процедуры это NEAR или FAR;
3) для константы, определенной с помощью псевдооператора EQU или = это ABS.

При этом псевдооператоры EXTRN обычно записываются где-то в начале исходного модуля. Например, пусть в главном исходном модуле Disp.asm используются процедура Sort и массив байтов Vector, определенные в других исходных модулях. А процедура Del2 и массив Buff, наоборот, определены в файле Disp.asm, а используются не только в нем, но и в других файлах. Тогда интересующие нас фрагменты Disp.asm могут иметь вид:

```
          . . . . . . . . . . . . . . .
        EXTRN    Sort:NEAR,  Vector:BYTE
          . . . . . . . . . . . . . . .
Disp      PROC   NEAR                   ;  Главная процедура
          . . . . . . . . . . . . . . .
            CALL     Sort               ;  Сортировка массива
          . . . . . . . . . . . . . . .
            MOV     AL, Vector[BX]      ;  Пересылка байта]
          . . . . . . . . . . . . . . .
Disp      ENDP
          . . . . . . . . . . . . . . .
        PUBLIC    Del2
Del2      PROC   NEAR                   ;  Процедура удаления
          . . . . . . . . . . . . . . .
Del2      ENDP
          . . . . . . . . . . . . . . .
        PUBLIC     Buff
Buff       DB    16  DUP (?)            ;  Массив из 16 байтов
          . . . . . . . . . . . . . . .
      END     Disp
```

При получении объектного модуля вместо внутренних меток и внешних выходных меток транслятор проставляет в те машинные инструкции, которые

используют соответствующие адреса, или смещение относительно текущего содержимого счетчика команд PC 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 инструкции на ЦП (см. п.3.3).

Что касается внешних входных меток, то обработать их транслятор не может. Он ничего не знает о размещении соответствующих программных объектов в памяти, т.к. 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 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 некоторых операционных системах (ОС), например в 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.

Для того, чтобы выполнить прикладную или какую-то системную программу (текстовый редактор, транслятор или редактор связей), программист сообщает имя загрузочного модуля требуемой программы подпрограмме ОС, называемой ***интерпретатором команд***. При работе в среде MS-DOS это имя набирается пользователем в командной строке в ответ на приглашение системы. А при работе в среде WINDOWS это имя задается путем установки курсора мыши на требуемое имя и последующим нажатием ее клавиши. В любом случае 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 инструкций. ***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.

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, в не 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. Например, в первых двух байтах PSP находится код машинной инструкции INT 20h. Данная инструкция программного прерывания выполняет возврат в ОС при завершении программы. Сама же эта инструкция 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 стоит инструкция 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 инструкцией 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 инструкции INT 21h (функция 4Ch). Этот способ позволяет возвращать из программы в ОС код возврата (в регистре AL), позволяющий информировать операционную систему о причине завершения программы.

В процессе своего выполнения машинная программа обращается за помощью к системным программам – ***драйверам ввода-вывода***. Обращение производится путем выполнения инструкции программного прерывания INT.

## 8.3. Кодирование виртуальных сегментов

На рис.57 представлена наиболее полная модульная структура ассемблерной программы. Наличие модулей обусловлено желанием упростить процесс получения программы на языке ассемблера, а также обусловлено желанием улучшить использование памяти для размещения соответствующей машинной программы.

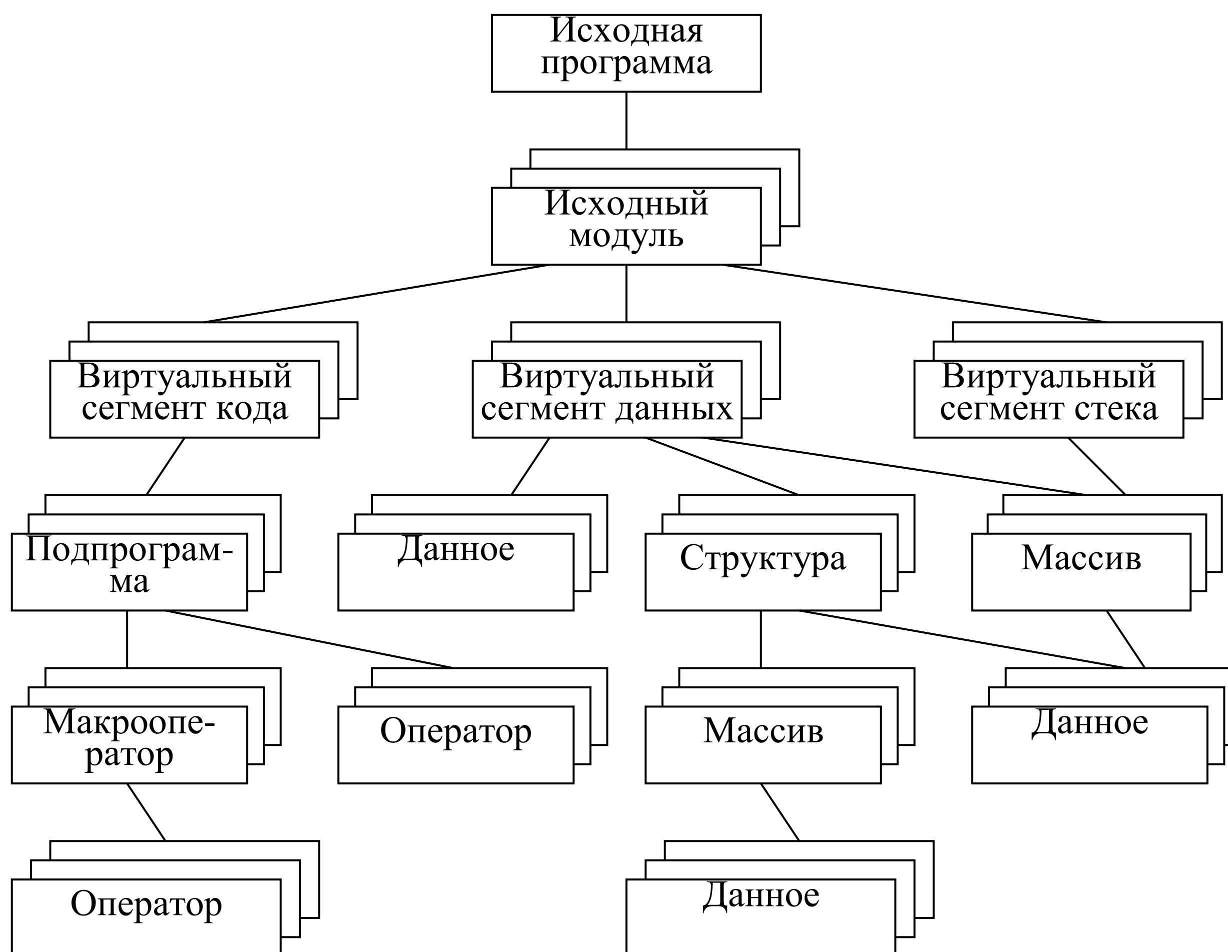



Рис. 57. Модульная структура программы на ассемблере

“Листьями” изображенной на рис. 57 модульной структуры являются ***элементарные модули*** – модули, которые не могут быть представлены в виде совокупности более мелких модулей. Элементарными модулями являются исполнительные операторы ассемблера и отдельные данные (переменные), заданные в исходной программе с помощью псевдооператоров DB, DW, DD или DQ.

Совокупность операторов может быть представлена в программе в виде более крупного модуля - макрооператора, а совокупность отдельных данных – в виде массива. Еще более крупный исполнительный модуль – подпрограмма. В общем случае она состоит из исполнительных операторов и макрооператоров. А более крупный модуль данных – структура, которая в общем случае включает массивы и отдельные данные. Кодирование всех перечисленных модулей было рассмотрено ранее в п.6.

Еще более крупными модулями исходной программы являются виртуальные сегменты. ***Виртуальный сегмент*** – участок исходной программы, выделенный программистом по функциональному признаку. Это могут быть: виртуальные сегменты кода, виртуальные сегменты данных и виртуальные сегменты стека. Выделение сегментов в исходной программе

обусловлено не удобством программирования (программировать без сегментов было бы удобнее), а стремлением добиться желательного размещения машинной программы в ОП.

Каждый виртуальный сегмент задается в программе с помощью псевдооператоров **SEGMENT** и **ENDS**:

```
<имя>    SEGMENT  [<атрибуты>]
         . . . . . . . . . .              ; Тело сегмента
<имя>    ENDS
```

Обратите внимание, что имя сегмента используется дважды – в начале и в конце сегмента. ***Атрибуты сегмента*** – свойства сегмента, задаваемые явно или по умолчанию. Они предназначены для того, чтобы задать размещение в ОП отрезков машинной программы, соответствующих виртуальным сегментам. В ЭВМ, основанным на процессоре i8086, память выделяется программе в виде одной или нескольких непрерывных областей размером до 64K, называемых ***физическими сегментами***. Поэтому можно сказать, что атрибуты сегментов задают соответствие между виртуальными и физическими сегментами.

***Атрибуты выравнивания*** – определяют выравнивание сегмента в памяти: **BYTE** – сегмент начинается со следующего байта; **WORD** – начало сегмента выравнивается по границе слова; **DWORD** – начало сегмента выравнивается по границе двойного слова; **PARA** – начало сегмента выравнивается по границе параграфа (задается по умолчанию); **PAGE** – начало сегмента выравнивается по границе страницы (256 байт).

***Атрибуты сочетания*** – определяют, как будут объединяться друг с другом виртуальные сегменты, имеющие одно и то же имя, но расположенные в разных исходных файлах. **PRIVATE** – сегмент не будет объединяться с другими одноименными сегментами и поэтому для него будет выделен отдельный физический сегмент (значение по умолчанию). **PUBLIC** – сегмент будет объединяться в памяти с другими одноименными сегментами по принципу последовательного соединения. **STACK** – то же, что и PUBLIC, но с учетом того, что объединяются стеки. (Вспомним, что стек “растет” в отличие от других данных в сторону меньших, а не больших адресов.) **COMMON** – объединяются одноименные сегменты так, что они начинаются с одного и того же адреса в памяти. Это позволяет более удобно использовать в нескольких исходных файлах одни и те же данные.

***Атрибуты класса*** – задают имя класса, к которому относится сегмент. Имя класса выполняет роль дополнительного имени сегмента, которое используется редактором связей для определения тех разноименных виртуальных сегментов, которые должны отображаться в один и тот же физический сегмент. Иными словами атрибут класса выполняет для разноименных (но однотипных) виртуальных сегментов ту же роль, что и атрибут сочетания PUBLIC для одноименных виртуальных сегментов.

Атрибут класса задается в виде символьной строки, заключенной в кавычки. В принципе можно задать любую строку (редактор связей реагирует только на равенство строк), но рекомендуются следующие значения: ‘CODE’ (сегмент кода), ‘DATA’ (сегмент данных), ‘STACK’ (сегмент стека).

Пример виртуального сегмента кода:

```
Cseg    SEGMENT   PARA  PUBLIC  ‘CODE’
        ASSUME     CS:Cseg, DS:Dseg
  Caller   PROC    NEAR
              . . . . . . . . . . .
              CALL   Callee      ;  Вызов процедуры
              . . . . . . . . . . .
              RET
  Caller   ENDP
  Callee   PROC    NEAR    ;  Вызываемая процедура
              . . . . . . . . . . .
              RET
  Callee    ENDP
Cseg      ENDS
```

Данный виртуальный сегмент состоит из двух процедур, одна из которых вызывает другую. Отрезок машинной программы, соответствующий этому сегменту, будет загружен в ОП, начиная с начала параграфа (PARA), будет подсоединен к другим одноименным сегментам, также имеющим атрибут PUBLIC, а также будет находиться в том же физическом сегменте (если суммарный объем не превысит 64К), что и другие сегменты класса ‘CODE’.

Пример виртуального сегмента данных:

```
Dseg     SEGMENT   PARA  PUBLIC  ‘DATA’
   a          DB    ?                              ;  Один байт
   b          DB    ?                              ;      --//--
   Squ        DB     1, 4, 9, 16, 25, 36, 49, 64   ;  Массив байтов
Dseg      ENDS
```

Пример сегмента стека:

```
_Stack    SEGMENT   PARA   STACK  ‘STACK’
            DB          64   DUP ( ‘STACK   ‘)            ;  Область стека
_Stack    ENDS
```

В данном примере для размещения стека зарезервирована область памяти длиной 64 x 8 = 512 байт, что достаточно для размещения 256 слов (работа со стеком производится только пословно). Заполнение области стека осмысленным текстом выполнено в примере для того, чтобы при анализе с помощью отладчика (например, DEBUG) области памяти, занимаемой машинной программой, нетрудно было бы найти область стека. (По мере

записи в стек новых слов данные, записанные в область стека при инициализации, будут уничтожаться.)

Приведенные выше атрибуты псевдооператора SEGMENT позволяют загрузить в один и тот же физический сегмент однотипные сегменты, имеющие или одинаковые основные или дополнительные имена. Для того, чтобы потребовать загрузку в один и тот же физический сегмент разнотипных виртуальных сегментов (например, сегмента данных и сегмента стека) необходимо в начале исходного модуля, содержащего объединяемые виртуальные сегменты, поместить псевдооператор **GROUP**, имеющий структуру:

<имя> GROUP <имя1>, <имя2> [, . . .]

Здесь сегменты с именами "имя1", "имя2" и т.д. объединяются в один сегмент с именем "имя". Обычно это делается для всех сегментов данных и сегмента стека с целью совместного использования переменных в этих сегментах.

## 8.4. Влияние на программу типа загрузочного модуля

Сколько физических сегментов требуется программе? Минимальное число таких сегментов определяется размером программы. Например, для размещения машинной программы длиной 100K требуется не менее двух сегментов ОП. В зависимости от содержания программы число запрашиваемых сегментов может превышать минимально необходимое их число. Возможные варианты выбора числа физических сегментов называются ***моделями памяти***. Из них наиболее распространены следующие:

1) ***минимальная модель*** – программа загружается в единственный физический сегмент, поэтому длина программы не должна превышать 64K;

2) ***малая модель*** - программа загружается в два сегмента ОП. В один сегмент загружаются все инструкции (код) программы, а во второй – все данные и программный стек. Все виртуальные сегменты данных и стека должны быть сгруппированы (с помощью псевдооператора GROUP);

3) **средняя модель** – программа загружается в несколько (более двух) сегментов ОП. Код программы занимает все сегменты памяти, кроме одного, в который загружаются все данные и программный стек;

4) ***компактная модель*** – программа загружается в несколько (более двух) сегментов ОП. Код программы занимает один физический сегмент, а данные и стек занимают все остальные сегменты;

5) ***большая модель*** – наиболее общий случай. Код программы занимает несколько сегментов памяти. Данные и стек также загружаются в несколько физических сегментов.

Минимальная модель памяти реализуется, в частности, в программах, имеющих загрузочные модули типа .com. Исходная программа,

ориентированная на получение .com-файла, имеет следующие отличия от исходной программы, ориентированной на получение .exe-файла:

1) задает размещение машинной программы в физическом сегменте со смещением 256 (100h) байтов относительно начала сегмента, то есть сразу за PSP. Для этого в начале виртуального сегмента кода в главном исходном файле помещается псевдооператор **ORG 100h** ;
2) задает в качестве точки входа в программу первый исполнительный оператор программы. Причем соответствующая машинная инструкция должна находиться в сегменте ОП со смещением 100h относительно начала сегмента. (Для .exe-файла точка входа может быть любой.) Вспомним, что точка входа задается операндом псевдооператора END в конце главного исходного модуля;
3) 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;
4) 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, а настройка .com-файла не производится.) Например, недопустимы следующие операторы:

```
MOV     AX,  _Data              ;  _Data – сегмент данных
MOV     SI, SEG  Buff           ;  Сегментная часть адреса
                                ;    переменной Buff → SI
```

Пример исходной программы, ориентированной на получение загрузочного модуля типа .com:

```
;
;         Программа выводит сообщение “Hello” на экран
;         ---------------------------------------------------------------
;
cr    EQU     0Dh              ;  Код ASCII возврата каретки
lf    EQU     0Ah              ;  Код ASCII перевода строки

_Text    SEGMENT   PUBLIC  ‘CODE’
         ORG      100h
         ASSUME   CS:_Text
  Print   PROC      NEAR
             MOV     DX, OFFSET Msg        ;  DX ← адрес сообщения
             MOV     AH, 09h               ;  Вывод строки
             INT     21h                   ;    на экран
             MOV     AX, 4C00h             ;  Возврат в DOS с
```

```
            INT     21h                         ;   кодом завершения 0
  Print   ENDP
   Msg     DB    'Hello', cr, lf, '$'           ;  Выводимая строка
_Text    ENDS
   END      Print
```

Эта исходная программа состоит из единственного исходного модуля, который, в свою очередь, 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 инструкций, что неизбежно привело бы к сбою в работе процессора. Что касается размещения соответствующей машинной программы внутри сегмента ОП, то здесь принципиально важным является взаимное расположение инструкций (кода) программы и ее стека (см. п.5.1) .

Пример исходного модуля типа .exe , выполняющего то же самое:

```
;
;          Программа выводит сообщение "Hello" на экран
;          ----------------------------------------------------------------
;
cr    EQU      0Dh               ;  Код ASCII возврата каретки
lf     EQU      0Ah               ;  Код ASCII перевода строки

 _Text     SEGMENT   PUBLIC    'CODE'              ;  Сегмент кода
          ASSUME    CS:_Text, DS:_Data, SS:_Stack
   Print    PROC      NEAR
               MOV    AX, _Data                     ;  Сделаем сегмент
               MOV    DS, AX                        ;    данных адресуемым
               MOV    DX, OFFSET Msg                ;  DX ← адрес сообщ-я
               MOV    AH, 09h                       ;  Вывод строки
               INT      21h                         ;    на экран
               MOV    AX, 4C00h                     ;  Возврат в DOS с
               INT      21h                         ;    кодом завершения 0
   Print     ENDP
 _Text     ENDS

 _Data     SEGMENT    PUBLIC   'DATA'               ;  Сегмент данных
   Msg     DB   'Hello', cr, lf, '$'                ;  Выводимая строка
 _Data     ENDS

 _Stack    SEGMENT    STACK   'STACK'               ;  Сегмент стека
           DB    128  DUP (?)
 _Stack    ENDS
    END    Print
```

В данном исходном модуле три виртуальных сегмента – сегмент кода _Text, сегмент данных _Data и сегмент стека _Stack.

В обеих приведенных исходных программах в начале виртуального сегмента кода находится псевдооператор **ASSUME**:

ASSUME [CS:<имя1> ,] [DS:<имя2> ,] [ES:<имя3> ,] [SS:<имя4> ]

где <имя1> - имя виртуального сегмента кода; <имя2> - имя виртуального сегмента данных; <имя3> - имя дополнительного сегмента данных; <имя4> - имя виртуального сегмента стека.

Данный псевдооператор информирует транслятор о том, что для адресации элементов машинной программы (инструкций и элементов данных), соответствующих указанному виртуальному сегменту, используется указанный сегментный регистр (CS, DS, ES или SS). Иными словами, псевдооператор ASSUME информирует транслятор о типе каждого виртуального сегмента. Это может показаться странным, но без использования ASSUME транслятор "не знает", для чего предназначен тот или иной виртуальный сегмент. Например, в приведенном выше примере виртуального сегмента стека, несмотря на трехкратное использование слова "steck" при описании сегмента, транслятор "не понимает", что речь идет именно о стеке, т.к. использует каждое из трех слов для выполнения частных, не связанных между собой операций.

Необходимо отметить, что ASSUME только информирует транслятор о использовании сегментных регистров, но не требует от него записи в эти регистры каких-то значений. Запись в регистры сегментов выполняет сама машинная программа на этапе выполнения.

## 8.5. Замена логических сегментов памяти

Благодаря наличию сегментных регистров машинная программа имеет дело с ***логическими сегментами* ОП**. В любой момент времени любая машинная программа имеет в своем распоряжении четыре логических сегмента ОП размером 64K каждый:

1) ***сегмент кода*** – сегмент ОП, номер начального параграфа которого находится в регистре CS;
2) ***сегмент данных*** – сегмент ОП, номер начального параграфа которого в регистре DS;
3) ***дополнительный сегмент данных*** – сегмент ОП, номер начального параграфа которого в регистре ES;
4) ***сегмент стека*** – сегмент ОП, номер начального параграфа которого в регистре SS.

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

1) для сегмента кода – IP;

2) для сегмента данных – BX, SI, DI;
3) для дополнительного сегмента данных – BX, DI;
4) для сегмента стека – SP.

Таким образом, логический адрес любой ячейки ОП представляет собой пару: (<регистр сегмента> , <регистр смещения>) . Преобразование логического адреса в физический адрес происходит при попадании соответствующей машинной инструкции на ЦП (см. п.3.3).

Каждый из четырех логических сегментов ОП играет строго определенную роль при выполнении инструкций машинной программы:

1) следующая инструкция программы выбирается только из сегмента кода;
2) инструкции, выполняющие операции со стеком (PUSH, POP, CALL, INT, RET, IRET) имеют дело с тем стеком, на который “указывает” SS;
3) инструкции, у которых ячейки памяти задаются только одним операндом, имеют дело с сегментом данных;
4) инструкции, у которых ячейки памяти задаются обоими операндами, используют и сегмент данных и дополнительный сегмент данных.

Для того, чтобы инструкция обрабатывала данное не из “привычного”, а из другого логического сегмента, необходимо использовать вспомогательный псевдооператор замены сегмента – **DS:**, **ES:**, **SS:** или **CS:**. В операторе ассемблера этот оператор записывается в поле уточняемого операнда. Например, следующий оператор

```
MOV     ES:[SI], BX
```

выполняет пересылку содержимого регистра BX в слово дополнительного сегмента данных, смещение которого относительно начала сегмента содержится в регистре SI.

Соответствие между логическими и реальными сегментами ОП зависит от используемой модели памяти. Например, если машинная программа получена в результате загрузки .com-файла (используется минимальная модель памяти), то все четыре логических сегмента программы отображаются на единственный реальный сегмент памяти. При этом во всех четырех сегментных регистрах содержится номер начального параграфа физического сегмента. (Запись в сегментные регистры выполняет загрузчик перед тем, как передать управление загруженной программе.)

Во время выполнения .com-программы содержимое сегментных регистров может изменяться только в результате выполнения инструкции INT. Выполнение данной инструкции обязательно приводит к смене сегмента кода, т.к. системные и прикладные программы размещаются в разных сегментах ОП. Обработчик программного прерывания может выполнить замену сегмента данных и сегмента стека. Но обычно сегмент стека остается без изменения, что позволяет использовать его для хранения данных, которыми обмениваются прикладная программа и обработчик прерывания.

Во время загрузки .exe-файла загрузчик помещает в сегментные регистры номер начального параграфа главного физического сегмента, в котором

находятся PSP и главная подпрограмма. В отличие от .com-файла, замену содержимого сегментных регистров могут выполнять не только инструкции INT и IRET, но и некоторые другие инструкции. Подобная замена не требуется, если программа использует минимальную или малую модели памяти.

Если программа использует среднюю или большую модель памяти (эти модели требуют загрузки инструкций программы в несколько физических сегментов), то замену содержимого регистра CS (смену сегмента кода) выполняют инструкции CALL и JMP при условии, что метка перехода (имя процедуры) имеет в исходной программе тип FAR (дальний).

Если программа использует компактную или большую модель памяти (эти модели требуют загрузки данных программы в несколько сегментов ОП), то замену содержимого регистров DS и ES выполняют инструкции программы MOV (см. п. 6.2.3, п. 6.6). Кроме того, в начале своего выполнения .exe-программа, обычно, перезагружает регистры DS и ES так, чтобы они указывали не на начало PSP, а на начало участков машинной программы, соответствующих виртуальным сегментам данных (см. пример .exe-файла). Очевидно, что в этом случае логический сегмент данных перекрывает реальный сегмент ОП, выделенный программе, но не совпадает с ним.

## 8.6. Комментирование программы

Несмотря на то, что комментарии в программе на ассемблере или каком-то другом языке программирования никак не отображаются в текст машинной программы, их роль весьма велика по следующим причинам. Во-первых, отладка программы может быть достаточно продолжительной во времени и поэтому детали программы забываются ее автором. Во-вторых, отладка программы, как правило, не заканчивается при сдаче программы в эксплуатацию, т.к. любая сколько-нибудь сложная программа имеет ошибки. Причем поиск и исправление ошибок в программе во время ее эксплуатации обычно выполняет не автор программы, а другие лица. В-третьих, весьма вероятно, что в процессе эксплуатации программы появится потребность в ее модификации. В любом случае отладка и модификация программы с плохими комментариями может превратиться в настоящий кошмар.

Так как любая программа представляет собой совокупность подпрограмм, то и комментарии программы можно рассматривать как совокупность комментариев подпрограмм. В общем случае комментарии подпрограммы включают: 1) вводные комментарии; 2) текущие комментарии.

***Вводные комментарии*** подпрограммы являются главными. Они представляют собой достаточно точное описание ее интерфейсов и позволяют очень быстро понять назначение процедуры и ее взаимосвязи с другими подпрограммами. Состав вводных комментариев:

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.

Пример вводного комментария:

```
;
;        Процедура дампирует 16 байт памяти в одну строку
;              шестнадцатеричных чисел
;      ------------------------------------------------------------------------
;  Входы :           DX – номер первого байта строки в  Sector
;  Вызовы :          Write_char,   Write_hex
;  Читается :        Sector
;
Disp_line            PROC          NEAR
. . . . . . . . . . . . . . . . . . . . .
 Disp_line           ENDP
```

Что касается ***текущих комментариев***, поясняющих назначение отдельных операторов подпрограммы 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 инструкций и поэтому заслуживает пояснения.

## 8.7. Отладка программы

Существуют два противоположных способа отладки: “сверху-вниз” и “снизу-вверх”. А также комбинированный способ – “из центра”. Обычно, хотя и не всегда, выбор метода отладки определяется выбором метода проектирования (см. п.7.4).

В методе отладки ***снизу-вверх*** отладке подпрограммы предшествует отладка тех подпрограмм, которые в ней вызываются. Так как при отладке нижестоящей подпрограммы вышестоящая отсутствует, вместо нее используется ***процедура-имитатор***. Имитатор задает входные параметры тестируемой процедуры и, возможно, выводит на экран ее выходные параметры.

В методе отладки ***сверху-вниз*** сначала отлаживается корневая вершина в дереве подпрограмм. Т.к. на момент отладки вызываемые процедуры отсутствуют, то они заменяются ***процедурами-заглушками***. Заглушка имеет тот же интерфейс, что и настоящая процедура (имя, входные и выходные

параметры). Но тело процедуры почти отсутствует. Оно содержит лишь операторы, выполняющие присваивание выходным параметрам процедуры правдоподобных значений.

В процессе отладки процедуры проверяется правильность ее логики, а также тестируется информационный обмен с вызываемыми процедурами (заглушками). После того, как отладка данной процедуры завершена, одна из заглушек заменяется настоящей процедурой и процесс тестирования повторяется для этой новой процедуры.

Т.к. при отладке процедуры вызывающая ее подпрограмма была отлажена раньше, то не требуется тратить усилия на создание тестовой среды. Сам процесс тестирования при этом для программиста намного приятнее. Несмотря на это, на практике данный метод используется сравнительно редко, т.к. требует привычки.

На практике часто используется метод ***из центра***. Т.е. выбирается процедура из центра дерева подпрограмм и отлаживается. Постепенно отлаженная часть программы «растет» и вверх и вниз.

# 9. ОРГАНИЗАЦИЯ ИНФОРМАЦИИ ВО ВНЕШНЕЙ ПАМЯТИ

## 9.1. Файлы

Важным частным случаем линейных несвязанных списков являются ***файлы***.

На любом носителе информация хранится в виде длинной битовой строки, т.е. в виде последовательности нулей и единиц. Но в отличие от ОП, к ячейкам которой ЦП обращается по их номеру (реальному адресу), для работы с информацией, находящейся на носителе ВП, эта информация разбивается на части, называемые файлами. Более подробное определение: ***файл*** – информация, расположенная в непрерывной или разрывной области носителя ВП, имеющая имя, уникальное для данной ВС.

Возможность размещения несвязанного линейного списка, каковым является файл, в разрывной области памяти обусловлена тем, что для поиска нужной записи файла используется вспомогательный (связанный) список. Этот список будет рассмотрен в п.9.2.

Что касается ***имени файла***, которое ему может дать программист, то оно включает ***собственно имя файла*** (обязательная часть) и ***расширение имени*** (необязательная часть). Эти части имени разделяются точкой. Применение расширений позволяет давать “родственным” файлам “родственные” имена, что весьма удобно для программиста. 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).

Имя файла должно удовлетворять требованиям используемой операционной системы. При программировании в среде MS-DOS собственно имя файла содержит от 1 до 8 символов, а расширение от 0 до 3 символов. В состав символов могут входить строчные и прописные латинские буквы, цифры, а также некоторые служебные символы: - ; _ ; $ ; # ; & ; @ ; ! ; % ; ~ ; ^ ; ( ; ) ; { ; } . Если в состав имени входят строчные (малые) буквы, то MS-DOS всегда воспринимает их как прописные (большие).

Что касается уникальности имени файла в пределах ВС, то имя, данное файлу программистом, таким свойством не обладает. Получение уникального имени файла будет рассмотрено в п. 9.3.

Одна и та же информация (битовая строка), содержащаяся в файле, может быть различным образом разбита на записи, в зависимости от того, какая программа работает с файлом. Прикладная программа имеет дело с ***логическими записями файла***, а некоторые системные программы – с ***физическими записями файла***. В отличие от физических записей, которые отражают лишь фактическое размещение информации на носителе, логические записи выделяются по смысловому признаку, не учитывая физические особенности носителя.

Например, пусть файл содержит сведения о служащих организации. Тогда в качестве логической записи файла целесообразно рассматривать битовую строку, содержащую сведения об одном сотруднике организации. Если этот файл находится на магнитном диске, то в качестве физической записи часто используют ***сектор диска*** длиной 512 байт.

В качестве второго примера файла рассмотрим каталог. ***Каталог*** – служебный файл, содержащий сведения о других файлах (в том числе, возможно, и о других каталогах). Файл-каталог состоит из 32-байтовых логических 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. Две записи каталога всегда используются особо. В одной из них содержится адрес размещения на диске самого каталога, а во второй – адрес "родительского" каталога.

## 9.2. Таблица размещения файлов

Так как файлы хранятся на носителях ВП (например, на дисках), то пространство этих носителей должно распределяться между файлами. Такое распределение выполняется фиксированными объемами памяти, называемыми ***кластерами***. Кластер – непрерывная область ВП, размер которой зависит от типа носителя и равняется размеру сектора диска, умноженному на степень двойки: 512 x N, где N = 1, 2, 4, . . . . Весь объем пространства носителя, за исключением небольшой его части,

зарезервированной для размещения корневого каталога и FAT-таблицы, поделено на кластеры.

Таблица размещения файлов (FAT) имеет столько 12- или 16-битных элементов, сколько кластеров носителя могут распределяться между файлами. Иными словами, FAT представляет собой уменьшенную модель распределяемой части носителя. Ее наличие позволяет размещать файл в разрывной области ВП (вспомним, что файл – несвязанный линейный список). Для этого каждому файлу ставится в соответствие вспомогательный линейный связанный список, построенный из элементов таблицы FAT.

Вспомним, что одно из полей записи каталога, описывающей файл, содержит его начальный адрес. Этот адрес представляет собой номер первого кластера файла и, следовательно, номер соответствующего элемента в таблице FAT. Содержимым этого элемента является номер следующего элемента связанного списка, который совпадает с номером следующего кластера файла. Если элемент FAT-таблицы соответствует последнему кластеру файла, то он содержит специальное число (FFFh).

Если программа запрашивает у ОС какую-то логическую запись файла, то поиск этой записи системой выполняется в два этапа. На первом из них при помощи связанного списка из элементов FAT-таблицы ищется искомая физическая запись (кластер) файла. А на втором этапе внутри кластера находится логическая запись.

## 9.3. Файловая структура системы

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 диск целиком или его часть, имеющий имя, уникальное для данной ВС. 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. 58. Корнем дерева является ***корневой каталог***. Этот каталог расположен в фиксированной области диска и состоит из 32-байтовых записей, структура которых аналогична структуре записей обычного каталога. Корневой каталог имеет имя “\” .

На следующем уровне дерева находятся те файлы и каталоги, которым соответствуют записи в корневом каталоге. Поле начального адреса в такой записи выполняет роль указателя на соответствующий файл (каталог). Аналогично, каталоги первого уровня дерева “порождают” файлы и каталоги второго уровня и т.д.

Т.к. ВС имеет в общем случае несколько логических дисков, то имя каждого такого диска может рассматриваться в качестве переменной S (см. п.4.2),

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 MS-DOS для этого достаточно набрать имя требуемого логического диска.

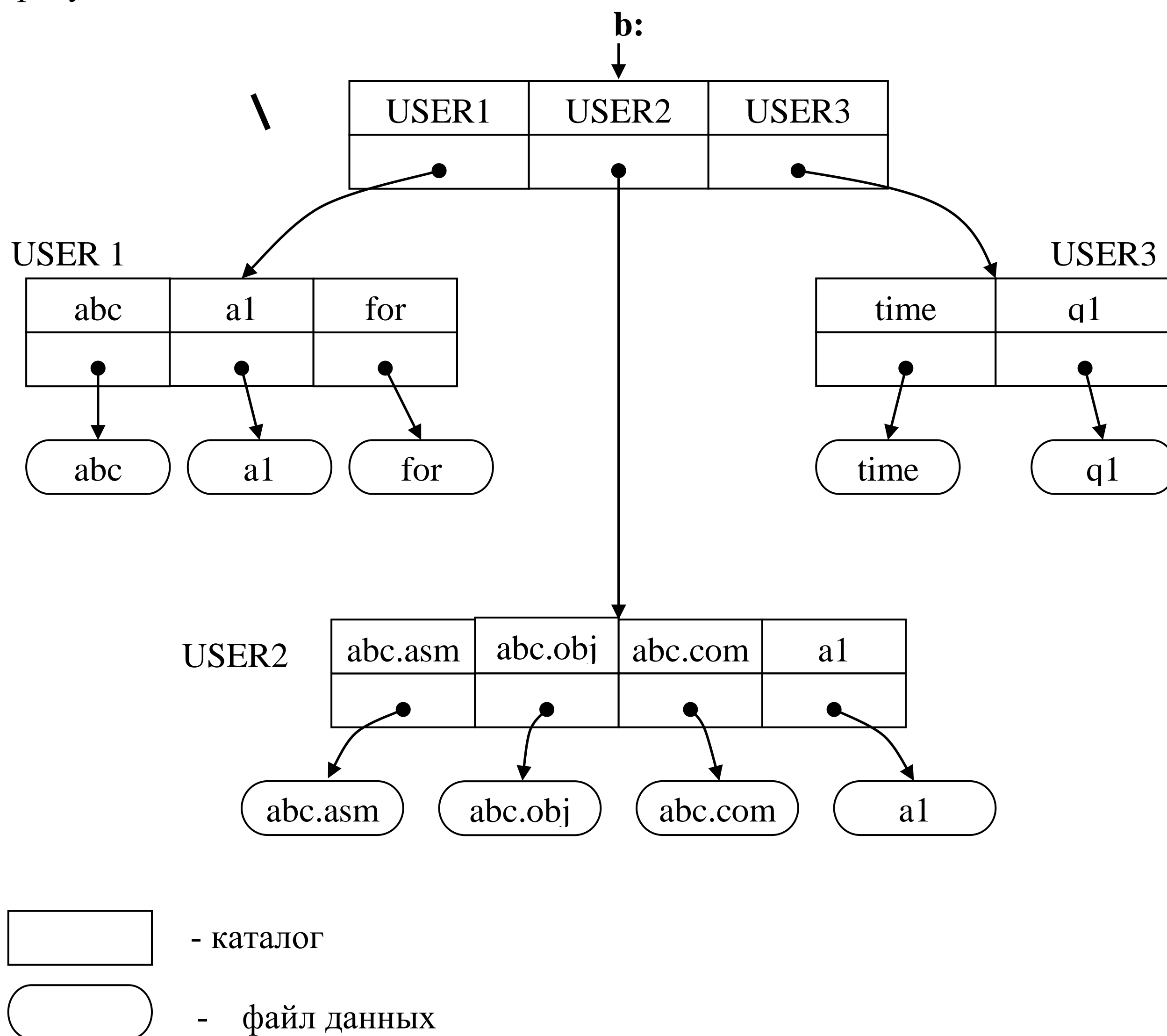



Рис. 58. Пример файловой структуры логического диска

Одним из достоинств древовидной файловой структуры является то, что она обеспечивает уникальность имен файлов внутри ВС. Дело в том, что ОС работает не с тем именем файла, которое ему дал программист, а использует имя-путь. ***Имя-путь файла*** получается последовательным соединением всех имен в дереве, начиная от корня и кончая именем данного файла. При этом имя каждого каталога завершается символом “\”. Пример имени-пути: \USER2\abc.asm . В примере на рис. 58 два файла имеют одно и то же имя a1, но ОС использует для работы с каждым из этих файлов свое имя-путь.

Единственное ограничение: все файлы, связанные с одним каталогом, должны иметь разные имена.

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 связан с текущим каталогом, то его можно задать для ОС не с помощью имени-пути, а просто по имени. ОС сама получит имя-путь файла, соединив имя-путь каталога с именем файла. С помощью команды ОС пользователь может сменить текущий каталог.

Применение специальной утилиты, например Norton Commander, значительно упрощает задание имен файлов.

# 10. СИСТЕМНЫЕ ПРОГРАММЫ

## 10.1. Классификация системных программ

При подготовке прикладной программы к выполнению, а также во время самого выполнения нельзя обойтись без использования системных программ. На рис. 59 приведена классификация таких программ.

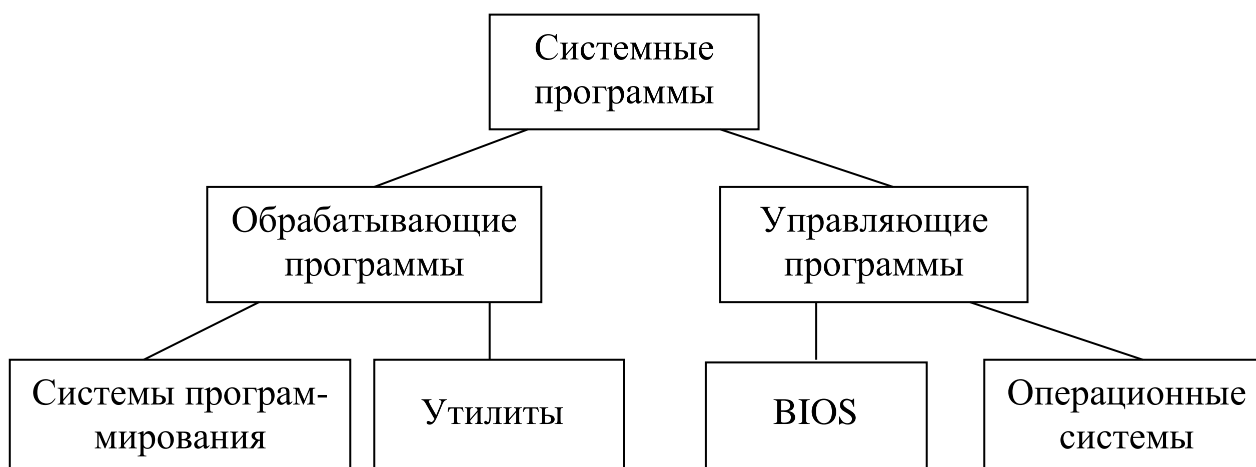



Рис. 58. Классификация системных программ

***Обрабатывающие 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 и 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*** – преобразование информации.

Запуск обрабатывающей программы аналогичен запуску прикладной программы (см. п. 8.2).

Основные функции ***управляющих программ***:

1) оказание помощи прикладным и системным обрабатывающим программам в использовании ими ресурсов ВС. При этом различают информационные, программные и аппаратные ресурсы;
2) обеспечение одновременного выполнения нескольких прикладных и (или) системных обрабатывающих программ.

Вторая из этих функций реализуется лишь в ***мультипрограммных*** **ОС**, например, в WINDOWS-95. В ***однопрограммных*** **ОС**, к которым относится MS-DOS, эта функция отсутствует.

Что касается первой функции, то она реализуется во всех ВС. Соответствующие системные подпрограммы делятся на две группы: подпрограммы BIOS и подпрограммы ОС. **BIOS** – базовая система ввода-вывода. Сюда относятся системные подпрограммы, находящиеся в ПЗУ (постоянное запоминающее устройство). Эти подпрограммы выполняют многие функции обмена с периферийными устройствами.

Инициирование управляющих подпрограмм, относящихся как к ОС так и к BIOS, происходит при попадании на ЦП инструкции прикладной программы INT (см. п. 5.3).

## 10.2. Утилиты

***Утилита*** – системная программа, выполняющая перенос информации и, возможно, некоторое ее преобразование. Перенос может быть многократным – с одного устройства на второе, со второго на третье и т.д. Перенос может быть циклическим – с первого устройства на второе, а затем обратно на первое.

Примером простой утилиты является программа копирования файла – COPY. Она переносит битовую строку, образующую файл, с одного носителя информации (диска) на другой и, возможно, присваивает полученному файлу новое имя.

Другая простая утилита – DIR. Она выводит (переносит) на экран содержимое каталога, указанного в качестве операнда команды DIR. Если операнд опущен, на экран выводится содержимое текущего каталога.

Утилитой является и любой текстовый редактор. Он выполняет переносы информации между диском, ОП, экраном и клавиатурой. Например, при создании нового файла информация в коде ASCII вводится с клавиатуры в область ОП, одновременно отображаясь на экране. Затем эта информация переносится из ОП на диск.

Довольно сложной утилитой является Norton Commander. 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 системой MS-DOS за счет того, что она переносит имя загрузочного модуля из позиции экрана, отмеченной пользователем с помощью псевдокурсора (псевдокурсор – светящийся прямоугольник) в то место ОП, откуда это имя может взять интерпретатор команд ОС.

Программа-утилита может быть реализована в виде отдельного загрузочного модуля с расширением .com или .exe, или может быть “встроена” в ОС. Во втором случае утилита не является логической частью ОС, а входит в нее “механически” с целью сократить затраты времени на загрузку утилиты с диска в ОП.

## 10.3. Лингвистические процессоры

***Лингвистический процессор*** – системная программа, выполняющая перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его представления, ориентированная на программиста, преобразуется в форму, ориентированную на процессор.

Лингвистические процессоры делятся на трансляторы и интерпретаторы. В результате работы ***транслятора*** алгоритм, записанный на некотором входном языке, преобразуется в алгоритм, записанный на другом – выходном языке. Наряду с трансляторами-ассемблерами и компиляторами к этому классу лингвистических процессоров относятся редакторы связей и загрузчики. Например, загрузчик выполняет перевод алгоритма программы с языка загрузочного модуля (для .exe-файла этот язык отличается от машинного) в язык машинных инструкций.

***Интерпретатор***, в отличие от транслятора, не получает новое представление алгоритма. Выполнив перевод очередного оператора входного алгоритма в соответствующую совокупность машинных инструкций, интерпретатор обеспечивает их выполнение. Затем преобразуется тот входной оператор, который должен выполняться следующим по логике алгоритма и т.д.

Примером интерпретатора является интерпретатор команд ОС. Входные операторы (команды) этого лингвистического процессора представляют собой имена загрузочных модулей, подлежащих выполнению. Эти команды поступают на вход интерпретатора или с клавиатуры терминала, или от утилиты типа Norton commander, или из командного файла. Командный файл имеет расширение .bat и содержит программу, операторами которой являются команды ОС (основные операторы), а также вспомогательные операторы, позволяющие задавать порядок выполнения основных операторов, отличный от последовательного.

Выполнение (интерпретация) очередной команды ОС сводится к загрузке и инициированию требуемой утилиты, лингвистического процессора или прикладной программы. Интересно, что с точки зрения самой ОС интерпретатор ее команд (как и любой другой лингвистический процессор или утилита) представляет собой обычную прикладную программу. Так как интерпретатор команд ОС может обращаться к ней, как и прикладная программа, только с помощью инструкций INT.

Например, загрузочный модуль интерпретатора команд MS-DOS хранится на диске в виде файла Command.com. Если мы заменим содержимое этого файла, сохранив его имя, то MS-DOS будет работать с другим интерпретатором команд, “не заметив” этого.

## 10.4. Подпрограммы управления аппаратурой

Среди управляющих программ, оказывающих помощь прикладным программам в использовании ресурсов ВС, особо важное значение имеют программы управления аппаратурой, называемые ***драйверами***. Каждый драйвер обслуживает однотипные периферийные или другие устройства. В ВС обязательно имеются драйвер экрана, драйвер клавиатуры, драйвер дисковода (для каждого типа дисководов свой драйвер) и некоторые другие драйверы.

Кроме драйверов реальных устройств многие ОС имеют драйверы виртуальных устройств. Например, в состав таких “устройств” могут входить некоторые области ОП и ВП. Поэтому встречаются: драйвер таблицы FAT, драйвер файловой структуры логического диска и т.п. При наличии обширной системы виртуальных устройств практически любая помощь прикладным программам в использовании ресурсов ВС оказывается драйверами. При этом драйверы виртуальных устройств обычно пользуются услугами других драйверов, управляющих виртуальными или реальными устройствами.

Драйверы некоторых реальных устройств входят в состав как ОС так и BIOS. Причем ОС может иметь несколько драйверов для устройств данного типа. Например в п. 8.4 были рассмотрены два примера программ (.com и .exe), выполняющие вывод строки символов на экран путем обращения (INT

21h, функция 09h) к драйверу экрана, входящему в состав ОС. В следующем фрагменте программы вывод на экран этой же строки выполняется путем обращения (INT 21h, функция 40h) к другому драйверу MS-DOS, выполняющему работу с файлами. При этом экран считается стандартным файлом, имеющим дескриптор файла, равный 1. (Дескриптор файла – число, заменяющее имя файла в некоторых операциях над ним.)

```
        . . . . . . . . . . . . . . . . . . . .
        MOV    CX, Msgl              ;  CX ← Длина сообщения
        MOV    DX, SEG  Msg          ;  DS:DX ← Адрес сообщения
        MOV    DS, DX                ;              --//--
        MOV    DX, OFFSET Msg        ;              --//--
        MOV    AH, 40h               ;  40h – функция записи в файл
        MOV    BX, 1                 ;  1 – дескриптор файла (экрана)
        INT    21h                   ;  Вывод строки на экран
      . . . . . . . . . . . . . . . . . . . . .
Msg     DB   ‘Hello’, cr, lf         ;  Сообщение
Msgl    EQU  $-Msg                   ;  Длина сообщения
      . . . . . . . . . . . . . . . . . . . . .
```

В следующем фрагменте программы вывод этой же строки символов на экран производится путем обращения (INT 10h, функция 0Eh) к драйверу BIOS, выполняющему работу с экраном. Так как данный драйвер выводит строку не целиком, а посимвольно, то в программе предусмотрен индексный цикл.

```
        . . . . . . . . . . . . . . . . . . . .
        MOV    CX, Msgl              ;  CX ← Длина сообщения
        MOV    AH, 0Eh               ;  0Eh – функция вывода символа
        MOV    BH, 0                 ;
        XOR    SI, SI                ;  SI ← 0
Next:   MOV    AL, Msg[SI]           ;  Вывод следующего
        INT    10h                   ;    символа
        INC    SI
        LOOP   Next                  ;  Повторить для след. символа
      . . . . . . . . . . . . . . . . . . . . .
Msg     DB   ‘Hello’, cr, lf         ;  Сообщение
Msgl    EQU  $-Msg                   ;  Длина сообщения
      . . . . . . . . . . . . . . . . . . . . .
```

Что касается внутренней структуры драйвера, то она представляет собой совокупность подпрограмм, связанных между собой не по управлению (подпрограммы драйвера не вызывают друг друга), а через общие структуры данных. Обычно одной из подпрограмм драйвера реального устройства является обработчик прерываний данного устройства. Другая подпрограмма (процедура) выполняет инициализацию (подготовку к работе) самого драйвера и устройства, выполняясь в начале работы системы. Остальные

процедуры обслуживают запросы прикладных и системных программ по работе с данным устройством.

Многие прикладные программы работают с ПУ, в особенности с экраном и клавиатурой, не с помощью системных драйверов, а напрямую, то есть, фактически, имеют свои собственные драйверы. Причиной этого, во-первых, является неэффективность системных драйверов (большое время выполнения), а во-вторых, желание учесть индивидуальные потребности прикладной программы по работе с конкретным устройством.  К сожалению, программирование драйверов выходит за рамки данного вводного курса.