Содержание

1 Часть 1.

1.1 Область применения языков программирования низкого уровня. Поколения ПК IBM PC.

1. Область применения языков программирования низкого уровня.

Низкоуровневые языки программирования используются везде, где необходима максимальная производительность:

1. Там, где требуется максимальная скорость выполнения: ядра ОС и программы, включаемые в них, основные компоненты компьютерных игр;
2. То, что непосредственно взаимодействует с внешними устройствами: драйвера для внешних устройств;
3. Всё, что должно максимально использовать возможности процессора: ядра многозадачных ОС, программы перевода в защищённый режим;
4. Всё что использует возможности ОС: вирусы, антивирусы, программы защиты и взлома защит;
5. Программы, предназначенные для обработки больших объёмов информации и требующие максимальной эффективности, например, программы, управляющие БД.

2. Поколения ПК IBM PC:

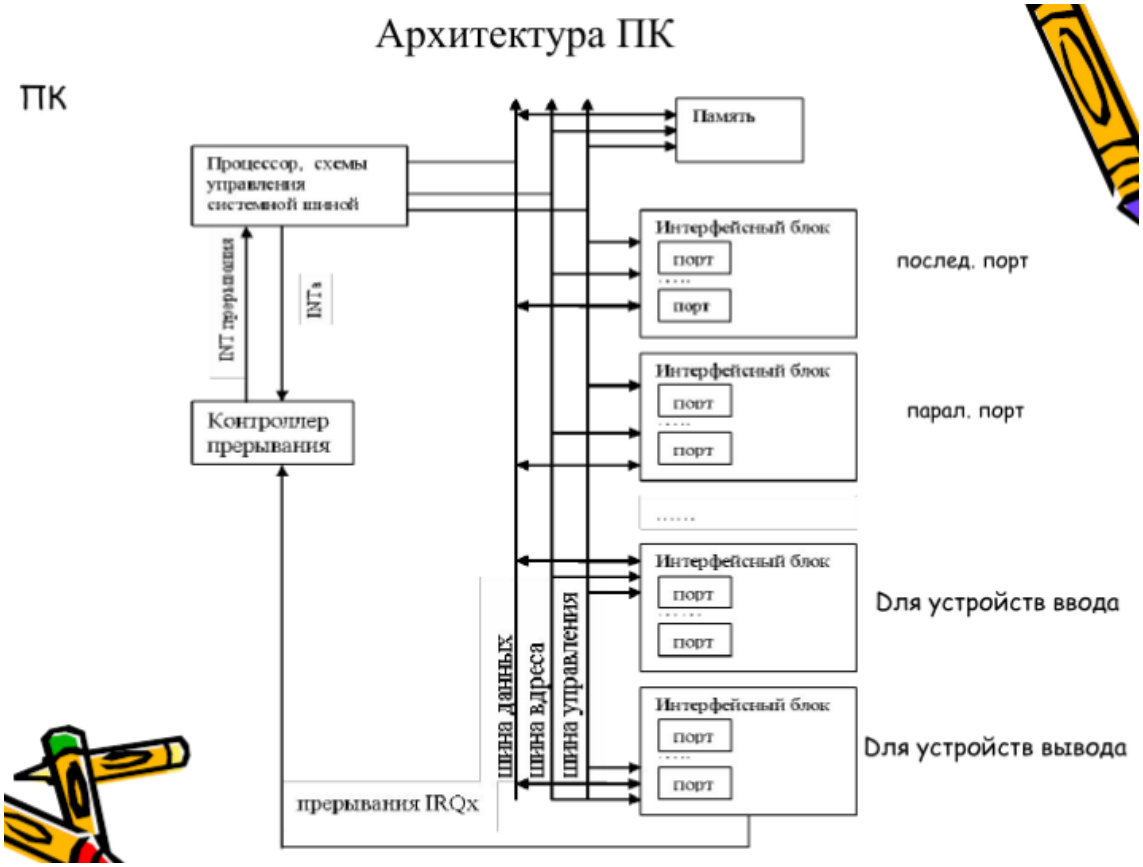
1. 1981 г. — IBM PC;
2. 1984 г. — IBM PC AT (Advenced Technology);
3. 1987 г. — 32-разрядный i386;
4. 1990 г. — i486: 1,5 млн транзисторов, 1Мкм технология, 5-ти стадийный конвейер для выполнения команд, кэш-память на кристалле процессора 8Кбайт;
5. 1993 г. — 64-разрядный МП "Pentium": 3,1 млн транзисторов, 0,8 Мкм технология;
6. Pentium Pro, Pentium 2, Pentium 3 с тактовой частотой 300-600 МГц;
7. "Willmate"800-1200 МГц, кэш до 1 Мбайта 2000 г. С 2002 г. — 0,13 Мкм, 146 мм^2 55 млн транзисторов.

1.2 Базовая архитектура ПК IBM PC.

В современных ПК реализован магистрально-модульный принцип построения. Все устройства (модули) подключены к центральной магистрали — системной шине, которая включает в себя адресную шину, шину данных и шину управления.

Шина — это набор линий связи, по которым передаётся информация от одного из источников к одному или нескольким приёмникам. Адресная шина однонаправленная, адреса передаются от процессора. Шина данных двунаправленная, данные передаются как от процессора, так и к процессору. В шину управления входят линии связи и однонаправленные и двунаправленные.

Внешние устройства работают значительно медленее процессора, поэтому для организации параллельной работы процессора и внешних устройств в архитектуру компьютера входит система прямого доступа к памяти (ДМА) и интерфейсные блоки, включающие в себя устройсва управления внешними устройствами (контроллеры, адаптеры) ...



1.3 Организация памяти в реальном режиме работы. Сегментые регистры. Понятие исполняемого и физического адреса.

1. Организация памяти в реальном режиме работы:

Процессор ix86 после включения питания устанавливается в реальный режим адрессации памяти и работы процессора.

В этом режиме процессор может работать с ОП как с непрерывным массивом байтов (модель памяти flat), так и с разделённой на много массиво — сегментов (в этом случае адрес байта состоит из 2 частей: адрес начала сегмента и смещение внутри сегмента). В реальном режиме размер сегмента фиксирован и составляет 64 Кбайта. Адрес сегмента кратен 16 и в 16-ричной СС может быть записан в виде $XXXX0_{16}$ и четыре старшие цифры адреса сегмента содержатся в сегментном регистре.

2. Сегментные регистры:
DS, ES, FS, GS, CS, SS

DS, ES, FS, GS — 16-разрядные сегментные регистры, используемые для определения начала сегментов данных.

CS — сегментный регистр кодового сегмента.

SS — сегментый регистр стека. Он устанавливается автоматически ОС и в нём хранится адрес начала сегмента стека, а указатель на вершину стека хранится в SP. Стек растёт от максимального адреса к минимальному при добавлении элементов в него. В модели памяти flat стек размещается в старших адресах, а программа в младших.

3. Понятие исполняемого и физического адреса:

Физический адрес представляет собой двадцатибитное беззнаковое целое, которое идентифицирует расположение байта в пространстве памяти.

ФА = АС + ИА

АС — адрес начала сегмента (значение сегментного регистра).

ИА — исполняемый адрес (смещение, в байтах, относительно начала сегмента).

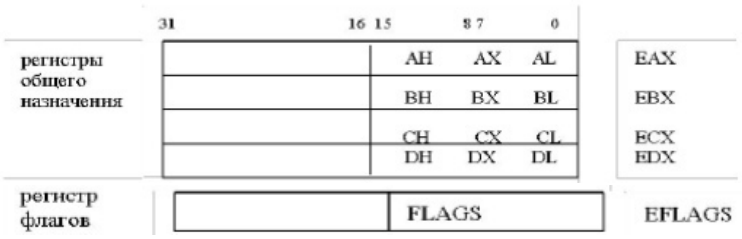
1.4 Процессор с точки зрения программиста. Регистры общего назначения. Регистры флагов.

1. Процессор с точки зрения программиста:

Процессор с точки зрения программиста — совокупность программнодоступных средств процессора.

2. Регистры общего назначения:

Регистр – это набор из n устройств, способных хранить n-разрядное двоичное число.



AX — аккумулятор.

DX — регистр данных.

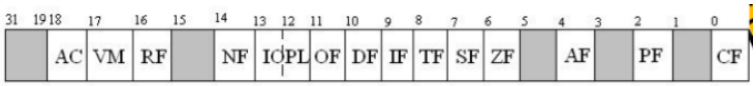
BX — регистр базы используется для организации специальной адресации операндов по базе.

CX — счётчик используется автоматически для организации циклов и при работе со строками.

Вышеперечисленные регистры могут использоваться для временного хранения адресов и данных.

3. Регистры флагов:

Регистр флагов FLAGS или EFLAGS определяет состояние процессора и программы в каждый текущий момент времени.



Биты 1, 3, 5, 15, 19 - 31 - не используются, зарезервированы.

В реальном режиме используют 9 флагов, из них 6 реагируют на результат выполнения команды, 3 определяют режим работы процессора.

В защищенном режиме используются 5 дополнительных флагов, определяющих режим работы процессора.

CF устанавливается в 1, если при выполнении команды сложения осуществляется перенос за разрядную сетку, а при вычитании требуется заем. 0FFFFh + 1 = 0000h и CF = 1 при работе со словами

PF = 1, если в младшем байте результата содержится четное количество единиц.

AF = 1, если в результате выполнения команды сложения (вычитания) осуществлялся перенос (заем) из 3-го разряда байта в 4-й (из 4-го в 3-й).

ZF = 1, если результатом выполнения операции является 0 во всех разрядах результата.

SF всегда равен знаковому разряду результата.

TF = 1 прерывает работу процессора после каждой выполненной команды.

DF определяет направление обработки строк данных, если DF= 0 – обработка строк идет в сторону увеличения адресов, 1 - в сторону уменьшения, (автоматическое увеличение или уменьшение содержимого регистров индексов SI и DI).

OF = 1, если результат команды превышает максимально допустимый для данной разрядной сетки.

IOPL = 1, если уровень привилегии текущей программы меньше значения этого флажка, то выполнение команды ввод/вывод для этой программы запрещен.

NT - определяет режим работы вложенных задач.

RF позволяет маскировать некоторые прерывания процессора.

VM - позволяет перейти из защищенного режима в режим виртуальных машин.

AC =1 приведет к сообщению об ошибке, если адреса операндов длиной в слово или двойное слово не будут кратны двум и четырем соответственно.

1.5 Понятие команды и директивы в Ассемблере. Формат команды и директивы.

1. Понятие команды и её формат:

```
[<метка>[:]] <код операции> [<операнды>, ...] [; комментарии]
```

Команда — это цифровой двоичный код, состоящий из двух подпоследовательностей двоичных цифр, одна из которых определяет код операции (сложить, умножить, переслать), вторая — определяет операнды, участвующие в операции и место хранения результата.

Команда может быть одно-, двух- и трёхадресной. Команда в памяти может занимать от 1 до 15 байт и длина команды зависит от кода операции, количества и места расположения операндов.

Операнды могут располагаться в регистрах, в памяти и непосредственно в команде и размер операндов может быть — байт, слово или двойное слово.

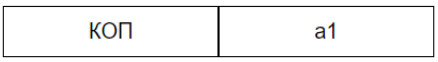
Трёхадресная команда: a1, a2 числа, участвующие в операции, al — адрес, по которому будет помещён результат.

КОП	al	a2	a3

Двухадресная команда: a1 — первый операнд, куда в дальнейшем будет помещён результат команды, a2 — второй операнд.

КОП	a1	a2

Одноадресная команда: a1 — либо число, участвующие в операции, либо адрес, по которому будет помещён результат.



Понятие директивы и формат директивы:

Директива — псевдооперация — информирует Ассемблер о чём-либо, например, какой объём памяти выделить под переменную или как следует объединять инструкции для формирования программного модуля, но сама директива в собранной программе никак не проявляется.

```
[<симв. имя>] <код псевдооперации> <операнды> [; комментарии]
```

Операндов может быть любое количество.

1.6 Структура программы на Ассемблере с исполением стандартных директив сегментации.

Исходный модуль на Ассемблере — последовательность строк, команд, директив и комментариев.

Исходный модуль просматривается Ассемблером, пока не встретится директива end. Обычно программа на Ассемблере состоит из 3 семгментов: стека, данных и кода.

Каждый сегмент начинается диретктивой начала сегмента — Segment и заканчивается диретктивой конца сегмента — ends, в операдах директивы Segment содержится информация о назначении сегмента.

Кодовый сегмент представляет собой программу решения поставленной задачи.

В сегменте стека выделяется место под стек.

В сегменте данных описываются данные, используемые в программе, выделяется место под промежуточные и окончательные результаты.

```
;сегмент стека
Sseg Segment stack 'stack'
    db 256 dup (ü)
    ;...
Sseg ends

;сегмент данных
Dseg Segment 'data'
    ;...
Dseg ends

;сегмент кода
Cseg Segment 'code'
    assume CS:Cseg, DS:Dseg, SS:Sseg    ;связь
        сегментных
```

```
                                        ;регистров
                                            и
                                        сегментов

                                        ;это
                                        позволяет

                                        Асссемблеру

                                        ;правильно

                                        добавить

                                        ;префиксы
                                        при
                                        обращении

                                        ;к
                                        регистрам.

                                        Например
                                        ,
                                        ;SP
                                        эквивалентен

                                        ;SS:SP, а
                                        SS =
                                        Sseg

    Start proc far                      ;главная
        процедура
        ;...
        ret
    Start endp
Cseg ends
end start
```

1.7 Основные элементы языка Ассемблера: имена, константы, переменные, выражения.

1. Имена:

Символические имена на ассемблере могут состоять из строчных и прописных букв латинского алфавита, цифр от 0 до 9 и некоторых символов '_', '.', '?', ...

2. Константы:

В программе на Ассемблере могут использоваться константы пяти типов: целые двоичные, десятичные, шестнадцатеричные, действительные с плавающей точкой, символьные.

Целые двоичные – это последовательности 0 и 1 со следующим за ними символом **‘b’, например, 10101010b или 11000011b.**

Целые десятичные - это обычные десятичные числа, возможно заканчивающиеся буквой d, например, **– 125 или 78d.**

Целые шестнадцатеричные числа – должны начинаться с цифры и заканчиваются всегда ‘h’, если первый символ – **‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’,** то перед ним необходимо поставить 0, иначе они будут восприниматься как символические имена.

Числа действительные с плавающей точкой представляются в виде мантиссы и порядка, например, **– 34.751e+02 – это 3475.1 или 0.547e-2 – это 0.00547.**

Строковые данные – это последовательности символов, заключенные в апострофы или двойные кавычки, например,

'abcd', 'a1b2c3', ‘567'.

Также, как и в языках высокого уровня, в Ассемблере могут использоваться именованные константы. Для этого существует специальная директива **EQU**. Например,

M EQU 27 ; директива EQU присваивает имени M значение 27.

Переменные в Ассемблере определяются с помощью директив определения данных и памяти, например,

v1 DB ?

v2 DW 34

или с помощью директивы ‘ = ’

v3 = 100

v3 = v3+1

Константы в основном используются в директивах определения или как непосредственные операнды в командах.

Выражения в Ассемблере строятся из операндов, операторов и скобок.

Операнды – это константы или переменные.

Операторы – это знаки операций (арифметических, логических, отношений и некоторых специальных)

Арифметические операции: **‘+’, ‘-‘, ‘*’, ‘/’, mod**.
Логические операции: **NOT, AND, OR, XOR**.
Операции отношений: **LT(<), LE(≤), EQ(=), NE(≠), GT(>), GE(≥).**
Операции сдвига: сдвиг влево **(SHL),** сдвиг вправо **(SHR)**
Специальные операции: **offset и PTR**
offset <имя> - ее значением является смещение операнда, а операндом может быть метка ли переменная;
PTR – определяет тип операнда:
BYTE = 1 байт,
WORD = 2 байт,
DWORD = 4 байт,
FWORD = 6 байт,
QWORD = 8 байт,
TWORD = 10 байт;

или тип вызова: **NEAR** – ближний, **FAR** – дальний.
Примеры выражений: **1) 10010101b + 37d 2) OP1 LT OP2**
3) (OP3 GE OP4) AND (OP5 LT OP6) 4) 27 SHL 3 ;

1.8 Команды пересылки, особенности их использования. Команды пересылки безусловной и условной, команды загрузки адреса.

1. Команды перессылки, особенности их использования:

Команды перессылки позволяют копировать данные из регистра в регистр, из памяти в регистр и из регистра в память.

Особенности использования команд перессылки:

1. Нельзя пересылать информацию из одной области памяти в другую;
2. Нельзя пересылать информацию из одного сегментного регистра в другой;
3. Нельзя пересылать непосредственный операнд в сегментный регистр, но если такая необходимость возникает, то нужно использовать в качестве промежуточного один из регистров общего назначения.

   MOV DX, 100h

   MOV DS, DX;
4. Нельзя изменять командой MOV содержимое регистра CS;
5. Данные в памяти хранятся в перевёрнутом виде, а в регистрах в нормальном виде, и команда перессылки это учитывает.

   R DW 1234h В байте с адресом R будет 34h, а в R + 1 — 12h

   MOV AX, R; 12h → AH, 34h → AL;

6. Размер передаваемых данных определяется типом операндов в команде.

   X DB ? ;один байт

   Y DW ?

   MOV Y,0 ;0 воспринимается как слово

   MOV byte PTR [SI], 0

   <тип> PTR <выражение>, где выражение — константа или адрес, а тип это размер данных;

7. Если тип обоих операндов определяется, то эти типы должны соответствовать друг другу.

   MOv AH, 500 ;ошибка

   MOV AH, 240 ;всё верно.

2. Команды пересылки безусловной и условной, команды загрузки адреса:

К командам пересылки относят команду обмена значений операндов.

XCHG OP1, OP2 ; r↔r ∨ r↔m
MOV AX, 10h ;
MOV BX, 20h ;
XCHG AX, BX ; (AX) = 20h, (BX) = 10h

Для перестановки значений байтов внутри регистра используют **BSWOP**.

(EAX) = 12345678h
BSWOP EAX ; (EAX) = 78563412h

Команды конвертирования:

CBW ; безадресная команда, (AL) → AX.
CWD ;(AX) → DX:AX
CW**E** ; (AX) → EAX (для i386 и выше)
CDF ; (EAX) → EDX:EAX (для i386 и выше)

Команды условной пересылки CMOVxx

CMOVL AL, BL ; если (AL) < (BL), то (BL) → (AL)

Загрузка адреса.

LEA OP1, OP2 ; вычисляет адрес OP2 и пересылает первому операнду, который может быть только регистром.

LEA BX, M[DX][DI]

1.9 Сегмент стека, команды для работы со стеком, команды прерывания.

1. Сегмент стека, команды для работы со стеком:

Стек определяется с помощью регистров SS и SP(ESP). Сегментный регистр SS содержит адрес начала сегмента стека. ОС сама выбирает этот адрес и пересылает его в регистр SS. Регистр SP указывает на вершину стека и при добавлении элемента стека содержимое этого регистра уменьшается на длину операнда.

Добавить элемент в стек можно с помощью команды:
PUSH <операнд>,
где операндом может быть как регистр так и переменная.
Удалить элемент с вершины стека можно с помощью операции:
POP <операнд>.

Для i186 и > PUSHA/POPA позволяют положить/удалить в стек содержимое всех регистров общего назначения в последовательности AX, BX, CX, DX, SP, BP, SI, DI.

Для i386 и > PUSHAD/POPAD позволяют сделать тоже самое, но уже с 32-разрядными регистрами

К любому элементу стека можно обратиться следующим образом:
MOV BP,SP ;(SP) → BP
MOV AX,[BP+6] ;(SS:(BP+6)) → AX.

2. Команды прерывания:

`int` — **команда прерывания**. Ее выполнение приводит к передаче управления DOS или BIOS, а после завершения какой-то системной обрабатывающей программы управление передается следующей команде.

Действия, происходящие после выполнения `int`, будут зависеть от значения операнда и от значений, хранящихся в регистрах.

Например, прерывания `int 21h` вызовут одну из DOS-функций, прерывания `int 10h` — функции видео драйвера, `int 16h` — функции драйвера клавиатуры.

1.10 Модели памяти, организация программы с помощью точечных директив.

В программе на Ассемблере могут использоваться упрощенные (точечные) директивы.

`.model <модель>` — определяет модель памяти, выделяемой для программы.
`<модель>` — одна из:

- `tiny` — под всю программу один сегмент памяти;
- `small` — по одному сегменту на данные и программу;

- `medium` — под данные один сегмент, под программу несколько;
- `compact` — под программу один сегмент, под данные несколько;
- `large` — под данные и под программу выделяется по нескольку сегментов;
- `huge` — позволяет использовать сегментов больше, чем потенциально может поместиться в оперативной памяти.

Пример:

```
1  .model small
2
3  .stack 100h
4
5  .data
6          str1 db 'Line1$'
7
8  .code
9  begin:
10         ; инициализировать DS
11         mov AX, @data
12         mov DS, AX
13
14         ; вывести строку
15         mov AH, 09h
16         mov DX, offset str1
17         int 21h
18
19         ; выйти из программы
20         mov AX, 4C00h
21         int 21h
22 end begin
```

2 Часть 2.

2.1 Команды безусловной передачи управления. Обращение к подпрограммам и возврат из них.

1. Команды безусловной передачи управления:

Команды безусловной передачи управления имеют вид

JMP <имя>,

где имя определяет метку команды, которая будет выполняться следующей за этой командой. Эта команда может располагаться в том же кодовом сегменте, что и команда **JMP** или в другом сегменте.

JMP M1; по умолчанию M1 имеет тип NEAR

Если метка содержится в другом сегменте, то в том сегменте, в который передаётся управление, должно быть **Public M1**, а из которого — **EXTRN M1: FAR**.

Кроме того, передачу можно осуществлять с использованием косвенной адресации (**JMP [BX]**).

Команда, осуществляющая близкую передачу, занимает 3 байта, а дальняя — 5 байтов. А если передача осуществляется не далее как на -128 или 127 байтов, то можно использовать команду безусловной передачи данных, занимающую 1 байт (**JMP Short M1**).

Примеры

```
1) ------------
   a  dw L  ; значением  a  является смещение для переменной L
  jmp L ; прямой переход по адресу L
        jmp a ; косвенный переход -  goto (a) = goto L
  --------------
2) ----------------------
   mov DX, a ; значение  a  пересылается в DX
   jmp DX  ; косвенный переход - goto (DX) = goto L
   ---------------------
   3) ----------------      3)  jmp  word ptr z
      jmp z ; ошибка          ------------------------------
       ----------------            z  DW  L
   Z  DW L                  ----------------------------
```

```
2) extrn p2: far                    public p2
   cseg     segment……                 cseg1    segment…..
            assume ……                 assume ……
   p1       proc far       p2         proc far
   ------------------------            ----------------------
            call p2                    ret
------------------------            p2  endp
            ret             cseg1      ends
   p1       endp
   cseg     ends
```

2. Обращение к подпрограммам и аозврат из них:

К командам безусловной передачи управления относятся команды обращения к подпрограммам, процедурам и возврат из них.

Процедура обязательно имеет тип дальности (по умолчанию NEAR, FAR необходимо указывать.)

Процедура типа NEAR может быть вызвана только из того же кодового сегмента, в котором содержится её описание. Процедура типа FAR может быть вызвана из любого мегмента.

(Главная программа всегда имеет тип FAR, так как к ней обращаются из ОС).

Команда вызова процедуры:

CALL <имя> ;

Адресация может быть использована как прямая, так и косвенная.

При обращении к процедуре типа NEAR в стеке сохраняется адрес возврата, адрес команды, следующей за CALL содержится в IP или EIP.

При обращении к процедуре типа FAR в стеке сохраняется полный адрес возврата CS:EIP.

Возврат из процедуры реализуется с помощью команды **RET.**

Она может иметь один из следующих видов:

RET [n] ; возврат из процедуры типа NEAR, и из процедуры типа FAR

RETN [n] ; возврат только из процедуры типа NEAR

RETF [n] ; возврат только из процедуры типа FAR

Параметр n является необязательным, он определяет какое количество байтов удаляется из стека после возврата из процедуры.

Пример:

```
1)    Cseg   segment….
             assume …..
         p1   proc far
         -------------------------
             call p2
         m:  mov AX, BX
         --------------------------
             ret
         p1   endp
           p2      proc near
             m1: mov CX, DX
             ---------------------------
             ret
         p2   endp
         Cseg ends
```

2.2 Исполняемые COM-файлы и их отличия от EXE-файлов + примеры.

После обработки компилятором и редактором связей мы на выходе получаем исполняемый exe-файл, размером не менее 512 байт (объём обязательного файла загрузки), но из него (не всегда) можно получить com-файл.

Отличие COM-файлов от EXE-файлов:

1. В com-файлах отсутствует блок начальной загрузки и следовательно он занимает меньше места, чем exe-файл;
2. exe-файл может занимать произвольный объём ОП, а com-файл — только один сегмент памяти;
3. В com-файле данные располагаются там же, где и программа.

   Т.к. вся программа содержится в одном сегменте, перед выполнением программы все сегментые регистры содержат в качестве значения адрес префикса программного сегмента — PSP. Поэтому нужно осуществлять обход префикса программного сегмента с помощью директивы **org 100h**.

Пример создания com-файла:
1:

```
TITLE Prog_Com_файл
Page 60,85
CSeg Segment Para 'Code'
ASSUME SS:CSeg, DS:CSeg, CS:CSeg
Org 100h

Start: JMP Main
    St1 DB 'String1',13,10,'$'
    St2 DB 'String2','$'

    Main Proc
        MOV AH,9
        LEA DX,St1
        Int 21h
        LEA DX,St2
        Int 21h
        MOV AH,4CH
        Int 21h
    Main endp
    CSeg ends
end Start
```

2:

```
.Model tiny
.Code
    JMP Met
    St1 DB 'String1','$'
    Met:
        MOV AH,09h
        LEA DX,St1
        Int 21h
        MOV AH,4CH
        Int 21h
    end Met
```

3:

```
Beg Proc
    MOV AH,09h
    LEA DX,St1
    int 21h
    MOV AH,4CH
    int 21h
Beg endp
St1 DB 'String1','$'
end beg
```

2.3 Способы адресации операндов + примеры команд.

В командах на Ассемблере результат всегда пересылается по адресу первого операнда.

Виды адресаций операндов:

1. Регистровая:

```
        MOV AX,BX ;(BX)->AX
```

;

2. Непосредственная:

```
        MOV AX,25 ;25->AX
        CONST EQU 34h ;именованая константа CONST
        MOV AX,CONST ;34h->AX
```

;

3. Прямая:

   Если известен адрес памяти, начиная с которого размещается операнд, то в команде можно непосредственно указать этот адрес.

```
        MOV AX,ES:0001
```

ES — регистр данных, 0001 — смещение внутри сегмента

Прямая адресация может быть записана с помощью символического имени, которое предварительно поставлено в соответствие некоторому адресу памяти.

Например, в сегменте ES содержится директива Var_p DW

```
MOV AX,ES:Var_p ;((ES)+Var_p)->AX
MOV AX,Var_p ;((DS)+Var_p)->AX
```

;

4. Косвенно регистровая:

   Как в регистровой адресации, только в регистре содержится не сам операнд, а адрес, в котором он содержится

   ```
   MOV AX,[SI]
   ```

   Могут использоваться регистры:

   SI,DI,BX,BP,EAX,EBX,ECX,EDX,EBP,ESI,EDI

   Не могут использоваться:

   AX,CX,DX,SP,ESP;

5. По базе со смещением:

   ```
   MOV AX,[BX]+2 ;((DS)+(BX)+2)->AX.
   == MOV AX,[BX + 2]
   == MOV AX,2[BX]
   MOV AX,[BP+4] ;((SS)+(BP)+4)->AX.
   ```

   ;

6. Прямая с индексированием:

   ```
   MOV AX,MAS[SI] ;((DS)+(SI)+MAS)->AX
   ```

   MAS — адрес в области памяти.

   Символическое имя определяет начало массива, а переход от одного элемента к другому осуществляется с помощью содержимого индексного регистра;

7. По базе с индексированием:

   ```
   MOV AX,Arr[BX][DI] ;((DS)+(BX)+(DI)+Arr)->
       AX
   ```

   Эта адресация используется для работы с двумерными массивами. Символическое имя определяет начало массива, с помощью базового регистра осуществляется переход от одной строки матрицы к другой, а с помощью индексного регистра — переход внутри строки.

2.4 Директивы определения данных и памяти.

Общий **вид директивы определения данных**:

```
[<симв. имя>] dX <операнд>[, ...] [; комментарии]
```

где X — это `B`, `W`, `D`, `F`, `Q` или `T`.

Операндов может быть один или несколько, каждый операнд должен быть константым выражением (числом), строковым литералом или символом `?`. Директива гарантирует выделение указанного объема оперативной памяти под операнд, инициализацию области начальным значением (если не используется `?`).

```
.data
        R1 db 0, ?, 0   ; выделено 3 байта, нулевой и второй
                        ; инициализированы нулями, первый
                        ; не инициализируется
```

Другой вариант директивы `dX`, упрощающий выделение памяти под массивы:

```
[<симв. имя>] dX <конст. выр.> dup(<операнд>) [; комментарии]
```

Она выделит память под `<конст. выр.>` значений и инициализирует каждую ячейку значением `<операнд>`.

Замечание: директивой `db` нельзя определить строковую константу длиной более 255 символов, а вместе с `dw` нельзя использовать строковые литералы длиной более двух символов.

2.5 Директивы внешних ссылок, организация межмодульных связей.

Директивы внешних ссылок позволяют организовать связь между различными модулями и файлами, расположенными на диске

Public <имя> [, <имя>,…,<имя>] –

определяет указанные имена как глобальные величины, к которым можно обратиться из других модулей. <имя> – имена меток и переменных, определенных с помощью директивы **‘=’ и EQU**.
Если некоторое имя определено в модуле А как глобальное и к нему нужно обратиться из других модулей В и С, то в этих модулях должна быть директива вида

EXTRN <имя>:<тип> [,<имя>:<тип>…]

Здесь имя то же, что и в **Public**, а тип определяется следующим образом:
если <имя> – это имя переменной, то типом может быть:

BYTE, WORD, DWORD, FWORD, QWORD, TWORD;

если <имя> – это имя метки, то типом может быть

NEAR, FAR.

Директива **EXTRN** говорит о том, перечисленные имена являются внешними для данного модуля.

Пример:
В модуле А содержится:

```
Public TOT
-------/-------
TOT DW 0  ;
```

чтобы обратиться из В и С к имени TOT, в них должна быть директива

EXTRN TOT:WORD

В Ассемблере есть возможность подключения на этапе ассемблирования модулей, расположенных в файлах на диске

INCLUDE <имя файла>

Пример:
INCLUDE C:\WORK\Prim.ASM
Prim.ASM, расположенный в указанной директории, на этапе ассемблирования записывается на место этой директивы.

2.6 Команды двоичной арифметики: сложение, вычитание, умножение, деление.

1. Сложение и вычитание:

Сложение (вычитание) беззнаковых чисел выполняется по правилам аналогичным сложению (вычитанию) по модулю 2^k принятым в математике….В информатике, если в результате более k разрядов, то к+1-й пересылается в CF.

$X + Y = (X + Y) \bmod 2^k = X + Y$ и CF = 0, если $X + Y < 2^k$
$X + Y = (X + Y) \bmod 2^k = X + Y - 2^k$ и CF = 1, если $X + Y >= 2^k$

Пример, работая с байтами, получим:
$250 + 10 = (250 + 10) \bmod 2^8 = 260 \bmod 256 = 4$
$260 = 1\ 0000\ 0100_2$, CF = 1, результат $- 0000\ 0100_2 = 4$

$X - Y = (X - Y) \bmod 2^k = X - Y$ и CF = 0, если $X >= Y$
$X - Y = (X - Y) \bmod 2^k = X + 2^k - Y$ и CF = 1, если $X < Y$
Пример: в байте
$1 - 2 = 2^8 + 1 - 2 = 257 - 2 = 255$, CF = 1

Сложение (вычитание) знаковых чисел сводится к сложению (вычитанию) с использованием дополнительного кода.

$$X = 10^n - |X|$$

В байте: $-1 = 256 - 1 = 255 = 11111111_2$
$-3 = 256 - 3 = 253 = 11111101_2$
$3 + (-1) = (3 + (-1)) \bmod 256 = (3+255) \bmod 256 = 2$
$1 + (-3) = (1 + (-3)) \bmod 256 = 254 = 1$ 1111110_2
Ответ получили в дополнительном коде, следовательно результат получаем в байте по формуле **$X = 10^n - |X|$, т.е.**
$x = 256 - 254 = |2|$ и знак минус. Ответ -2.

Переполнение происходит, если есть перенос из старшего цифрового в знаковый, а из знакового нет и наоборот, тогда
OF = 1. Программист сам решает какой флажок анализировать
OF или CF, зная с какими данными он работает.

Арифметические операции изменяют значение флажков OF, CF, SF, ZF, AF, PF.

В Ассемблере команда ‘+’
ADD OP1, OP2 ; (OP1) + (OP2) → OP1
ADC OP1, OP2 ; (OP1) + (OP2) + (CF) → OP1
XADD OP1, OP2; i486 и >
(OP1) ↔ (OP2) (меняет местами), (OP1) + (OP2) → OP1
INC OP1 ; (OP1) + 1 → OP1
В Ассемблере команда ‘-‘
SUB OP1, OP2 ; (OP1) – (OP2) → OP1
SBB OP1, OP2 ; (OP1) – (OP2) – (CF) → OP1
DEC OP1 ; (OP1) – 1 → OP1.

В командах сложения и вычитания можно использовать любые способы адресации:

2. Умножение и деление:

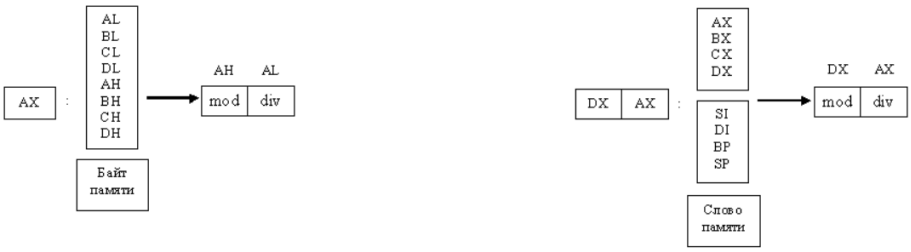
Существуют две формы команды деления — одна для двоичных чисел без знака DIV, а другая для чисел со знаком IDIV.

Форма:

DIV OP2

IDIV OP2

Один из операндов (делимое) всегда в два раза длиннее операнда делителя.



Из флагов после деления не определяется ни один, так как даже при переполнении будет вызвано программное прерывание.

Форма операции деления:

MUL OP2

IMUL OP2

Результат всегда в два раза длиннее сомножителей.

Если OF=CF = 1, то результат занимает двойной формат, и OF = CF = 0, результат умещается в формат сомножителей. Остальные флаги не изменяются.

Для умножения и деления первые операнды — это всегда регистры, а второй может быть как регистром или памятью, так и непосредственным (константой).

2.7 Команды побитовой обработки данных: логические операции, операции сдвига.

К командам побитовой обработки данных относятся логические команды, коанды сдвига, установки, сброса и инверсии битов.

1. Логические команды:

Это команды and,or,not,xor.

Для всех логических команд, кроме not, операнды одновременно не могут находиться в памяти, OF = CF = 0, AF — не определён, SF, ZF, PF определяются результатом команды.

and выполняет логическое усножение
and OP1, OP2 ;(op1)*(op2)->op1

второй операнд назывется маской, так как с помощью него можно устанавливать в 0 биты первого операнда. Второй операнд может быть извлечён из регистра, из памяти или непосредственно.

Пример: (AL) = 1011 0011, (DL) = 0000 1111,
and AL, DL ; (AL) = 0000 0011

or выаолняет логическое сложение
or OP1, OP2 ;(op1)+(op2)->op1

По сути эта команда устанавливает в единици биты первого операнда с помощью второго.

Если во всех битах будет 0, то ZF = 1.

(AL) = 1011 0011, (DL) = 0000 1111
or AL, DL ; (AL) = 1011 1111 .....

xor является симметрической разностью
xor op1,op2

Также важно помнить, что xor является методом самого быстрого обнуления регистров.

Например: (AL) = 1011 0011, маска = 000 01111
xor AL, 0Fh ; (AL) = 1011 1100

not выполняет инверсию битов
not op

Если (AL) = 0000 0000, **not AL** ; (AL) = 1111 1111

Общий формат команд арифметического и логического **сдвига**:

```
sXY  <оп. 1>, <оп. 2> [; комментарий]
```

где

- *X* — `h` (логический), `a` (арифметический),
- *Y* — `l` (влево), `r` (вправо),
- `<оп. 1>` — регистр или память,
- `<оп. 2>` — константа или младшие 5 бит регистра `CL`.

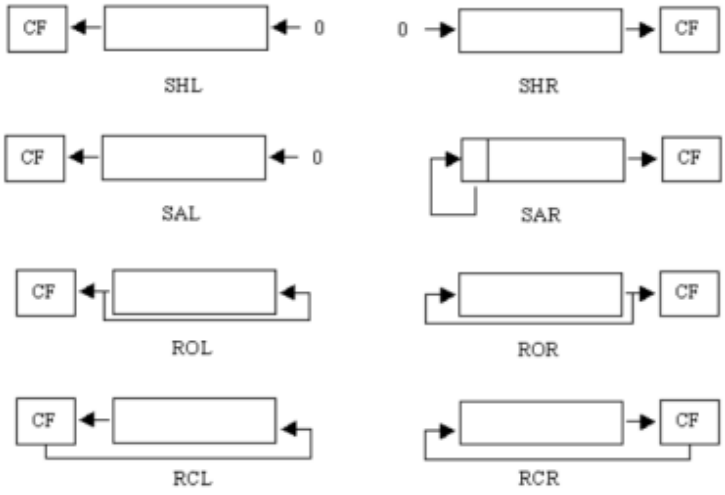


Рис. 2: Различные сдвиги

Логический сдвиг можно также считать *беззнаковым сдвигом*. То есть при сдвиге вправо образовавшиеся пустые места заполняются 0. При арифметическом сдвиге вправо, пустые места заполняются копиями старшего (знакового) бита. Поэтому его называют *знаковым сдвигом*.

При любом типе сдвига последний «ушедший» из числа бит помещается в CF.
Замечание: shl и sal синонимичны.
Расширенные инструкции сдвигов (i186 и новее):

```
shrd R/M, R, I8/CL
shld R/M, R, I8/CL
```

Содержимое R/M сдвигается на количество разрядов I8/CL, аналогично shr и shl, но пустые биты заполняются битами из операнда R:

- для сдвига вправо — начиная с самого старшего бита R/M, младшие биты R «задвигаются» в пустые ячейки;
- для сдвига влево — начиная с самого младшего бита R/M, старшие биты R «задвигаются» в пустые ячейки.

(да, это довольно сложно описать. Может быть поэтому А. Г. не пыталась в своей презентации.)

Циклические сдвиги:

- rol, ror — выполняют циклические сдвиги, причем последний вышедший из разрядной сетки бит оказывается в CF.
- rcl, rcr — выполняют циклические сдвиги, причем значение флага CF тоже используется и в первую очередь оказывается в числе (см. картинку 2).

2.8 Работы с массивами в ассемблере.

Адресация в одномерных массивах:

```
адрес(Arr[i]) = Arr + i * размер(Arr[i])
```

Адресация в двумерных массивах (двумерный массив $n \times m$):

```
адрес(Arr[i][j]) = Arr + i * m * размер(Arr[i][j]) + j *
                   размер(Arr[i][j])
```

`размер()` — число, соответствующее размеру в байтах одного элемента массива. Например, если массив был задан при помощи `dd`, то размер каждого элемента будет 4.

Для одномерных массивов удобнее всего будет использовать адресацию *прямую с индекисрованием*:

```
Arr[i].
```

Для двумерных массивов — *по базе с индексированием*:

```
Arr dd 64 dup(128 dup(?))
; ...
Arr[BX][DI]          ; Две переменные величины:
                     ; BX = i * 128 * размер(Arr[i][j])
                     ; DI = j * размер(Arr[i][j])
```

2.9 Команды условной передачи управления, организация циклов в ассемблере с помощью команд передачи управления.

Общий вид **команды условной передачи управления**:

```
jX  <метка> [; комментарии]
```

где X соответствует одному из условий для передачи управления. Если условие выполняется, то будет совершен прыжок, иначе выполнение продолжится следующей идущей командой.

`<метка>` должна отстоять от `j`X не дальше, чем на -128 или 127 байт.

Передача управления осуществляется на основе значений флагов, которые установила предыдущая команда. Например, если после вычитания `ZF = 1`, то это значит, что результат предыдущей был нулем, и прыжок `jz` выполнится.

Можно также напрямую «сравнить» два числа командой `cmp`. Она выполнит обычное вычитание второго операнда из первого, но результат никуда записан не будет. Это установит некоторые флаги в значения, пригодные для сравнения одного числа относительно другого.

Существует также команда `test`, которая выполнит побитовое `and` над операндами и установит флаги в нужные значения (более предпочтительна при проверке чисел на равенство).

условие	Для беззнаковых чисел	Для знаковых чисел
$>$	JA	JG
$=$	JE	JE
$<$	JB	JL
$>=$	JAE	JGE
$<=$	JBE	JLE
$<>$	JNE	JNE

В принципе, по этой теме это все, что есть в презентации. Если кому-то захочется, можно еще попытаться разобраться в этом.

Для краткости предположим, что непосредственно перед оператором условной передачи управления была выполнена инструкция `cmp OP1, OP2`. Тогда для перехода с условием

- Равно:
  - Для беззнаковых и для знаковых (`je`). `OP1` равен `OP2`, если `ZF = 1`.
- Больше или равно:
  - Для беззнаковых (`jae`, `jnb`). `OP1` больше или равен `OP2`, если нам не понадобился перенос разряда, то есть `CF = 0`.
  - Для знаковых (`jge`, `jnl`). Если `SF = 0` и не было знакового переполнения или если `SF = 1` и было знаковое переполнение, то `OP1` больше или равен `OP2`. Это условие можно упростить до `SF = OF`.
- Больше (`ja`, `jnbe` | `jg`, `jnle`): точно такие же условия, что для «больше или равно», но конъюнктивно добавляется условие `ZF = 0`, чтобы исключить равенство `OP1` и `OP2`.
- Меньше:
  - Для беззнаковых (`jb`, `jnae`). `OP1` меньше `OP2`, если нам понадобился перенос разряда, то есть `CF = 1`.
  - Для знаковых (`jl`, `jnge`). Если `SF = 0` и было знаковое переполнение, или если `SF = 1` и не было знакового переполнения, то `OP1` меньше `OP2`. Или кратко, $\texttt{SF} \neq \texttt{OF}$.
- Меньше или равно (`jbe`, `jna` | `jle`, `jng`): точно такие же условия, что для «меньше», но дизъюнктивно добавляется условие `ZF = 1`.

Если нужно совершить переход на метку, которая дальше -128 или 127 байтов, то следует инвертировать условие перехода, и сразу после инвертированного условного перехода добавить инструкцию безусловного перехода на нужную метку:

```
          ; cmp AX, BX
          ; je M      ; не можем, далеко!

          cmp AX, BX
          jne Cont
          jmp M
    Cont:
```

```
        ; ...
```

Оранизация **цикла с предусловием**:

```
l1: ; while x > 0 do S;
        cmp x, byte ptr 0
        jle e1
        ; S
        jmp l1
e1:
        ; ...
```

Организация **цикла с постусловием**:

```
l1: ; do S while x > 0;
        ; S
        cmp x, byte ptr 0
        jg l1
e1:
        ; ...
```

2.10 Команды для организации циклов в Ассемблере.

- `loop <метка>` — `CX = CX - 1`, и если `CX` $\neq 0$, то перейти на `<метка>`.
- `loope <метка> | loopz <метка>` — `CX = CX - 1`. Если `CX = 0`, то выйти из цикла. Иначе проверить флаг `ZF`:
  - если `ZF = 0`, то выйти из цикла;
  - если `ZF = 1`, то совершить переход на `<метка>`;
- `loopne <метка> | loopnz <метка>` — `CX = CX - 1`. Если `CX = 0`, то выйти из цикла. Иначе проверить флаг `ZF`:
  - если `ZF = 0`, то совершить переход на `<метка>`;
  - если `ZF = 1`, то выйти из цикла.

Пример использования:

```
        mov CX, 100
l1:
        ; ...
        loop l1
l2:
        ; ...
```

Если `CX` нужен внутри цикла:

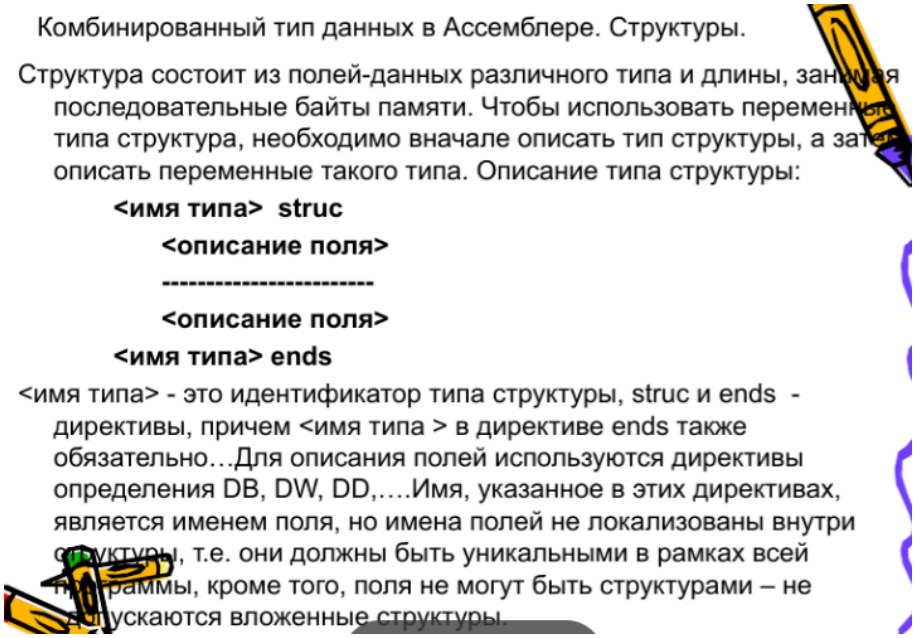
```
        mov CX, 100
l1:
        push CX
        ; ...
        pop CX
        loop l1
l2:
        ; ...
```

3 Часть 3.

3.1 Работа с файлами. Дескриптор файла. Создать файл, открыть, закрыть, удалить файл.

4 Часть 4.

4.1 Структуры в Ассемблере и их использование.



Например,

```
TData  struc          ; TData – идентификатор типа
    y  DW  2000       ; y, m, d – имена полей. Значения, указанные
    m  DB  ?          ; в поле операндов директив DW и DB ,        d  DB
28  ; называются значениями полей, принятыми
TData ends            ; по умолчанию.
```

? – означает, что значения по умолчанию нет.

На основании описания типа в программу ничего не записывается и память не выделяется. Описание типа может располагаться в любом месте программы, но только до описания переменных данного типа. На основании описания переменных Ассемблером выделяется память в соответствии с описанием типа в последовательных ячейках, так что в нашем случае размещение полей можно представить так:

	2б	1б	1б	размеры полей в байтах
TData	y	m	d	
+0	+2	+3		смещение относительно начала структуры.

Описание переменных типа структуры осуществляется с помощью директивы вида:

имя переменной имя типа <нач. значения>

Здесь уголки не метасимволы, а реальные символы языка, внутри которых через запятую указываются начальные значения полей. Нач-ым значением может быть: 1) ? 2) выражение 3) строка 4) пусто.

Например:

```
                    y    m   d
dt1  TData  <?, 6, 4>          ?    6   4
dt2  TData  <1999, , >      1999    ?  28
dt3  TData  < , , >         2000    ?  28
```

Идентификатор типа TData используется как директива для описания переменных также, как используются стандартные директивы DB, DW и т.д. Если начальные значения не будут умещаться в отведенное ему при описании типа поле, то будет фиксироваться ошибка. Правила использования начальных значений и значений по умолчанию:

- приоритетными являются начальные значения полей, при описании переменных, т.е. если при описании переменной для поля указан ? или какое-либо значение, то значения этих полей по умолчанию игнорируются.

Правила использования начальных значений и значений по умолчанию:

1) если в поле переменной указан знак **?**, то это поле не имеет начального значения, даже если это поле имеет значение по умолчанию (поле **y** переменной **dt1**);
2) Если в поле переменной указано выражение или строка, то значение этого выражения или сама строка становится начальным значением этого поля (поля **m** и **d** переменной **dt1** и поле **y** переменной **dt2**);
3) Если начальное значение поля переменной «**пусто**» - ничего не указано при описании переменной, то в качестве начального устанавливается значение по умолчанию – значение, указанное при описании типа, если же в этом поле при описании типа стоит знак **?**, то данное поле не имеет никакого начального значения (поля **m** переменных **dt2** и **dt3**).

Значения по умолчанию устанавливаются для тех полей, которые являются одинаковыми для нескольких переменных одного типа, например, год поступления на факультет одинаков для группы студентов. Любая переменная может изменять свое значение в процессе выполнения программы и поэтому структура может не иметь как значений по умолчанию, так и начальных значений.

Отсутствие начального значения отмечается запятой.

Если отсутствуют начальные значения нескольких последних полей, то запятые можно не ставить. Если отсутствуют значения первого поля или полей, расположенных в середине списка полей, то запятые опускать нельзя. Например:

dt4 TData <1980, ,> можно dt4 TData <1980>

dt5 TData <, , 5> нельзя заменить на dt5 TData < 5 >.

Если отсутствуют все начальные значения, опускаются все запятые, но угловые скобки сохраняются:

dt6 TData < >

При описании переменных, каждая переменная описывается отдельной переменной, но можно описать массив структур, для этого в директиве описания переменной указывается несколько операндов и (или) конструкция повторения DUP. Например:

dst TData <, 4, 1>, 25 DUP (< >)

Описан массив из 26 элементов типа TData, и первый элемент (первая структура) будет иметь начальные значения 2000, 4, 1, а все остальные 25 в качестве начальных будут иметь значения, принятые по умолчанию: 2000, ?, 28.

Имя первой структуры dst, второй – dst+4, третьей – dst+8 и т.д.

Работать с полями структуры можно также, как с полями переменной комбинированного типа в языках высокого уровня:

<имя переменной > . < имя поля>

Например, dt1.y, dt2.m, dt3.d

Ассемблер приписывает имени типа и имени переменной размер (тип), равный количеству байтов, занимаемых структурой

type TData = type dt1 = 4

И это можно использовать при программировании, например, так:

```
; выполнить побайтовую пересылку dt1 в dt2
        mov CX, type TData  ; количество повторений в CX
        mov     SI, 0       ; i = 0
   m:   mov AL, byte ptr dt1[SI] ; побайтовая пересылка
        mov byte ptr dt2[SI], AL ; dt1 в dt2
        inc SI              ; i = i+1
        loop m              ; использование byte ptr обязательно….
```

Точка, указанная при обращении к полю, это оператор Ассемблера, который вычисляет адрес по формуле:

<адресное выражение> + <смещение поля в структуре>

Тип полученного адреса совпадает с типом поля, т.е.

type (dt1.m) = type m = byte

Адресное выражение может быть любой сложности, например:

1) mov AX, (dts+8).y
2) mov SI, 8
   inc (dts[SI]).m ; Аисп = (dts + [SI]). m = (dts + 8).m
3) lea BX, dt1
   mov [BX].d, 10 ; Аисп = [BX] + d = dt1.d

Замечания:

1) type (dts[SI]).m = type (dts[SI].m) = 1, но
   type dts[SI].m = type dts = 4
2) Если при описании типа структуры в директиве, описывающей некоторое поле, содержится несколько операндов или конструкция повторения, то при описании переменной этого типа данное поле не может иметь начального значения и не может быть определено знаком ?, это поле должно быть пустым.

Одно исключение: если поле описано как строка, то оно может иметь начальным значением строку той же длины или меньшей; в последнем случае строка дополняется справа пробелами.

Например:
```
student struc
    f  DB  10 DUP (?)      ; фамилия
    i  DB  " ******* "     ; имя
    gr  DW  ?              ; группа
    oz  DB   5, 5, 5       ; оценки
student ends
```

Описание переменных:

```
st1  student <"Petrov", >; нельзя, т.к. поле f не строка
st2  student  < , "Petr", 112, > ; можно, f – не имеет начального
          ; значения
st3  student  < , "Aleksandra" >
     ; нельзя, в i 10 символов, а допустимо не больше 7.
```

4.2 Записи в Ассемблере, их описание и использование.