

Содержание

1	Часть 1.	2
1.1	Область применения языков программирования низкого уровня. Поколения ПК IBM PC.	2
1.2	Базовая архитектура ПК IBM PC.	3
1.3	Организация памяти в реальном режиме работы. Сегментные регистры. Понятие исполняемого и физического адреса. . . .	4
1.4	Процессор с точки зрения программиста. Регистры общего назначения. Регистры флагов.	4
1.5	Понятие команды и директивы в Ассемблере. Формат команды и директивы.	6
1.6	Структура программы на Ассемблере с исполнением стандартных директив сегментации.	7
1.7	Основные элементы языка Ассемблера: имена, константы, переменные, выражения.	8
1.8	Команды пересылки, особенности их использования. Команды пересылки безусловной и условной, команды загрузки адреса.	10
1.9	Сегмент стека, команды для работы со стеком, команды прерывания.	11
1.10	Модели памяти, организация программы с помощью точечных директив.	12
2	Часть 2.	13
2.1	Команды безусловной передачи управления. Обращение к подпрограммам и возврат из них.	13
2.2	Исполняемые СОМ-файлы и их отличия от EXE-файлов + примеры.	16
2.3	Способы адресации операндов + примеры команд.	17
2.4	Директивы определения данных и памяти.	19
2.5	Директивы внешних ссылок, организация межмодульных связей.	20
2.6	Команды двоичной арифметики: сложение, вычитание, умножение, деление.	20

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:

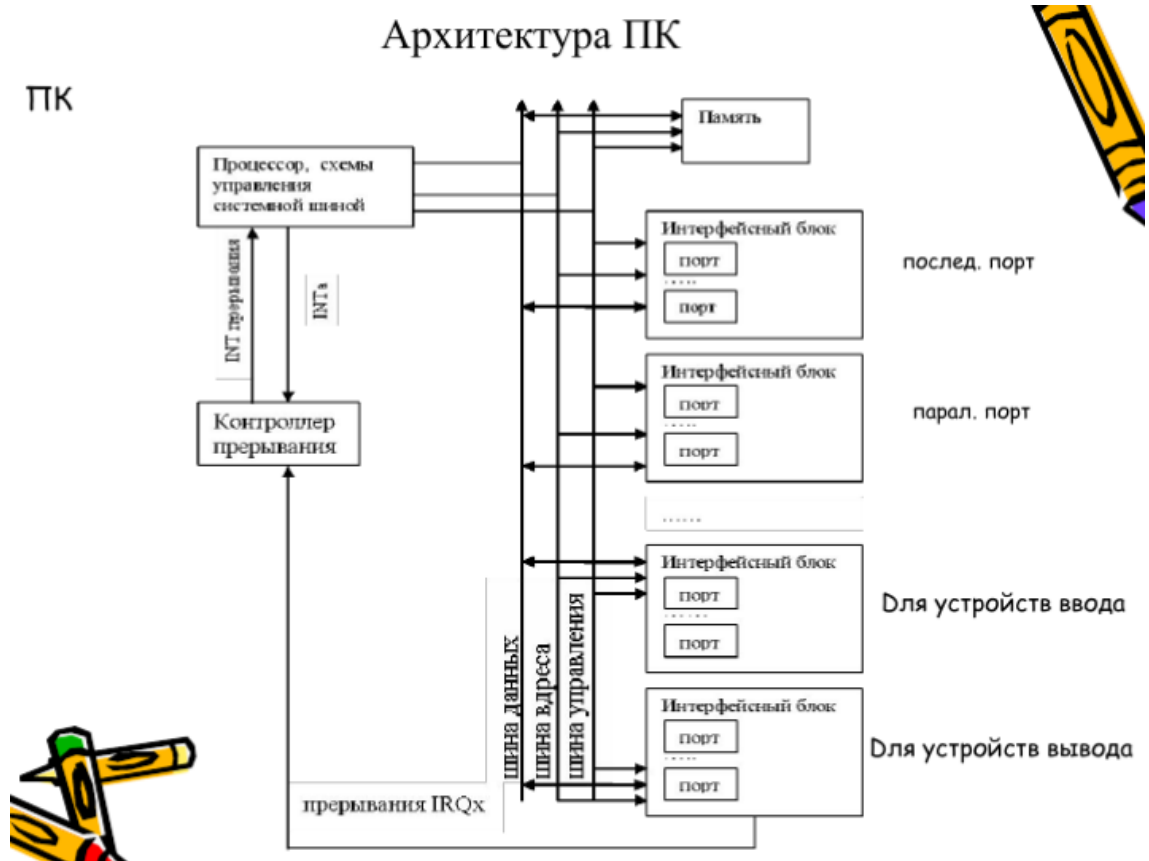
1. 1981 г. — IBM PC;
2. 1984 г. — IBM PC AT (Advanced Technology);
3. 1987 г. — 32-разрядный i386;
4. 1990 г. — i486: 1,5 млн транзисторов, 1Мкм технология, 5-ти стадийный конвейер для выполнения команд, кэш-память на кристалле процессора 8Кбайт;
5. 1993 г. — 64-разрядный МП "Pentium": 3,1 млн транзисторов, 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 мм² 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 медленнее 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 устройства управления внешними устройствами (контроллеры, адаптеры) ...



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

1. Организация памяти в реальном режиме работы:

Процессор ix86 после включения питания устанавливается в реальный режим адресации 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 массивов — 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 CS может быть записан в виде XXXX0_{16} и четыре старшие цифры адреса сегмента содержатся в сегментном регистре.

2. Сегментные регистры: $\text{DS}, \text{ES}, \text{FS}, \text{GS}, \text{CS}, \text{SS}$

$\text{DS}, \text{ES}, \text{FS}, \text{GS}$ — 16-разрядные сегментные регистры, используемые для определения начала сегментов данных.

CS — сегментный регистр кодового сегмента.

SS — сегментный 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.

$$\text{ФА} = \text{АС} + \text{ИА}$$

АС — адрес начала сегмента (значение сегментного регистра).

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

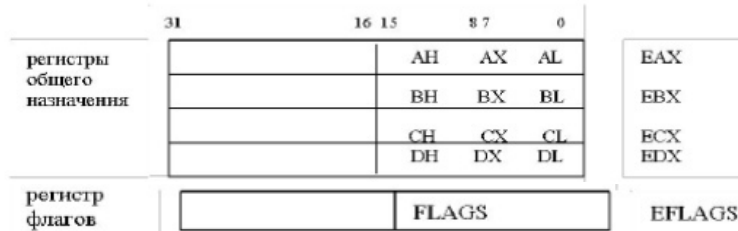
1.4 Процессор с точки зрения программиста. Регистры общего назначения. Регистры флагов.

1. Процессор с точки зрения программиста:

Процессор с точки зрения программиста — совокупность программно-доступных средств процессора.

2. Регистры общего назначения:

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



AX — аккумулятор.

DX — регистр данных.

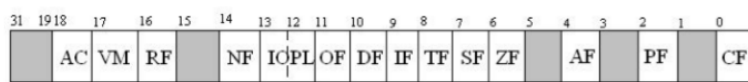
BX — регистр базы используется для организации специальной адресации операндов по базе.

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>, ...] [; комментарий]

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, a1 — адрес, по которому будет помещён результат.

КОП	a1	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.

КОП	a1
-----	----

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> [; комментарий]

Операндов может быть любое количество.

1.6 Структура программы на Ассемблере с исполнением 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 сегментов: стека, данных и кода.

Каждый сегмент начинается директивой начала сегмента — `Segment` и заканчивается директивой конца сегмента — `ends`, в операторах 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 (0)
    ;...
Sseg ends

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

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

```

	<i>; регистров</i>
	<i>и</i>
	<i>сегментов</i>
	<i>; это</i>
	<i>позволяет</i>
	<i>Ассемблеру</i>
	<i>; правильно</i>
	<i>добавить</i>
	<i>; префиксы</i>
	<i>при</i>
	<i>обращении</i>
	<i>; к</i>
	<i>регистрам.</i>
	<i>Например</i>
	<i>,</i>
	<i>; SP</i>
	<i>эквивалентен</i>
	<i>; SS:SP, а</i>
	<i>SS =</i>
	<i>Sseg</i>
	<i>; главная</i>
<code>Start proc far</code>	
<code> <i>процедура</i></code>	
<code> <i>;...</i></code>	
<code> ret</code>	
<code>Start endp</code>	
<code>Cseg ends</code>	
<code>end start</code>	

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**.

Символьные 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,

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 пересылки, особенности их использования:

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

Особенности использования команд пересылки:

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.
4. Нельзя изменять командой MOV содержимое регистра CS;
5. Данные в памяти хранятся в перевёрнутом виде, а в регистрах в нормальном виде, и команда пересылки 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 ; $r \leftrightarrow r \vee r \leftrightarrow m$

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

CQE ; (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 > PUSH/POP позволяют положить/удалить в стек содержимое всех регистров общего назначения в последовательности AX, BX, CX, DX, SP, BP, SI, DI.

Для i386 и > PUSH/POP 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 несколько 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 СОМ-файлы и их отличия от ЕХЕ-файлов + примеры.

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

Отличие СОМ-файлов от ЕХЕ-файлов:

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

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

Пример создания сом-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 ; именованная 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 константным 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.