

## Содержание

1	Лекция 1	2
2	Лекция 2	7
3	Лекция 3	13
4	Лекция 5	20
5	Лекция 6	27
6	Лекция 7	31
7	Лекция 8	34
8	Лекция 9	37
9	Лекция 10	40
10	Лекция 11	41
11	Лекция 12	44

# 1 Лекция 1

**При классификации программ** принято деление на прикладные или проблемные пользовательские и программы обеспечивающие функционирование работы комплекса систем в автономном режиме.

**Прикладные программы** — наборы долговременных библиотек программ, используемых для решения задач из конкретной области применения техники.

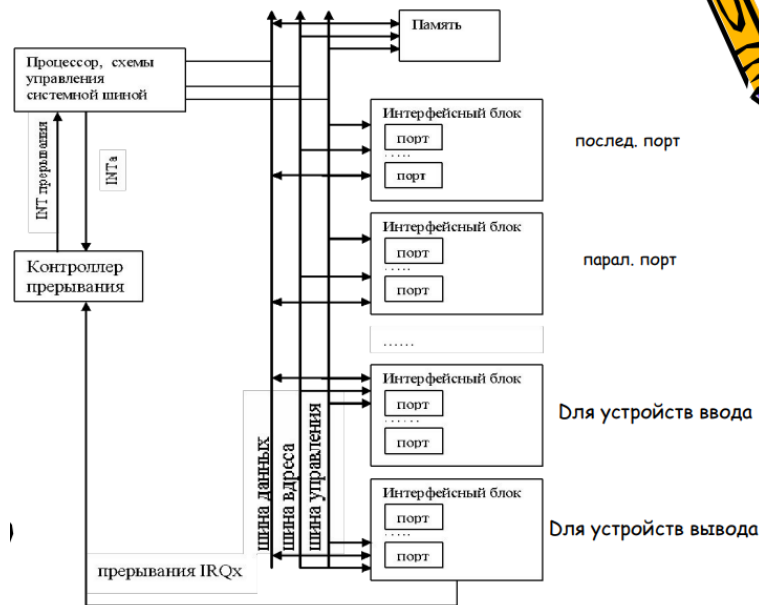
**Системные программы** используются для разработки новых программ, модификации уже существующих программ и выполнения программ автономным способом.

**Управляющие системные программы**, которые обеспечивают стабильность работы вычислительных систем, управляют процессами обработки, входят в ядро операционной системы, постоянно находятся в оперативной памяти, называются резидентными. А управляющие системные программы, которые загружаются в оперативную память перед их выполнением, называются транзитивными.

**Системные обрабатывающие программы** выполняются как специальное приложение и используются для разработки новых модификаций существующих программ.

**Архитектура ПК** включает в себя структурную организацию, то есть набор блоков и устройств, объединённых в вычислительную систему, и функциональную организацию, обеспечивающую работу этих блоков.

## Архитектура ПК



**Архитектура ПК** с точки зрения программиста — набор доступных блоков устройств.

**Современные ПК** имеют магистрально-модульный принцип построения, то есть к единой магистрали — системной шине подключены различные устройства.

**Шина** — набор линий, по которым информация передаётся от одного из источников к одному или нескольким приёмникам.

**Существует три типа шин:** адресная, данных и шина управления. По адресной шине информация передаётся от процессора, шина данных двунаправлена, то есть данные передаются от и к процессору, а шина управления включает в себя односторонние и двусторонние каналы связи.

**Процессор** работает существенно быстрее внешних устройств, поэтому для организации параллельной работы процессора и внешних устройств в архитектуру ПК включены: канал прямого доступа к памяти и информационные блоки (устройство управления внешними устройствами).

**Чтобы синхронизировать работу внешних устройств и процессора** используется система прерывания. Если некоторому устройству требуется работа процессора, то это устройство посылает сигнал прерывания,

он проходит через контролер прерывания, если условия выполнены, то контроллер посылает процессору прерывание, процессор обрабатывает его и возвращает устройству выполнение внешним устройством.

**Существуют различные типы и классификации** прерываний, они бывают внешние и внутренние, маскируемые и немаскируемые.

**Процессор с точки зрения** программиста — набор программно доступных средств.

**X86 процессор** при выключённом питании устанавливается в реальный режим работы оперативной памяти и процессора, но реальная система переводит его в защищённый режим, обеспечивающий многозадачность и ресурсы для этих задач. Начиная с 386 процессора доступны 16 основных регистров, 11 регистров для работы с мультимедиа и сопроцессором и некоторые управляющие регистры.

**Регистры общего назначения** могут использоваться для хранения адресов, данных и команд. При работе с 16-ти разрядными данными их имена: AX, BX, CX, DX.

**Для процессора** минимальной единицей информации является байт, он может работать с регистрами: AL, AH, BL, BH, DL, DH. Эти регистры имеют их собственные имена, отражающие их назначение.

**AX** — аккумулятор, в него записывается результат.

**BX** — базовый регистр, используется при адресации операндов по базе.

**CX** — счётчик, автоматически используется для организации циклов, работы со стеком.

**DX** — регистр данных.

## Регистры указателей и индексов

**Регистры индексов** используется для сложной адресации операндов, а регистры указателей SP, BP используются для работы со стеком.

**Сегментные регистры.** Рассматриваемый процессор может работать с оперативной памятью, как с 1 непрерывным массивом данных (flat), и памятью, разделённой на сегменты, в этом случае адрес байта состоит из 2-х частей: адрес начала сегмента и адрес внутри сегмента. И для получения адреса начала сегмента используются сегментные регистры DS, ES, FS, GS, CS, SS (16-ти разрядные).

**Операционная система** может размещать сегменты в любом месте оперативной памяти и даже временно на жёстком диске.

**Сегментных регистров 6**, но это не значит, что программа может использовать только 6 сегментов, программист может изменить содержимое сегментного регистра и попасть на другой адрес.

**Сегментный регистр называют** селектором, а с каждым селектором связан программно недоступный регистр — дескриптор и в защищённом режиме именно в дескрипторе находится адрес начала сегмента, его размер и дополнительная информация.

**В защищённом режиме** размер сегмента  $\leq 4\text{Гб}$ , а в реальном режиме размер сегмента фиксирован и равен 64Кб и в сегментном регистре находятся старшие цифры 16-тиричного адреса начала сегмента. Адрес сегмента всегда кратен 16 и поэтому младшие цифры равны 0.

**4 сегментных регистра** DS, ES, FS, GS используются для хранения сегмента данных. CS содержит адрес кодового сегмента, SS — стекового сегмента.

**Сегмент стека** реализуется особым образом, адрес начала сегмента стека определяется автоматически ОПС и записывается в SS, а при добавлении элемента в стек, указатель на вершину стека SP уменьшается.

**При работе с памятью** в режиме flat программы хранятся в младших адресах, а стек в старших.

**Стек используется** для работы с подпрограммами, в него записываются фактические параметры, а если программист хочет хранить локальные параметры, тогда после загрузки в стек фактических параметров, содержимое регистра Sp записывается в BP и тогда обращение к фактическим параметрам реализуется по формуле  $BP+k$ , а к локальным  $BP-n$ , где  $n$  и  $k$  вычисляет сам программист, исходя из количества параметров и их размера.

**IP** — счётчик команд указатель команд, в нём содержится смещение для следующей исполняемой команды.

**Регистр флагов FLAGS** (32-х разрядный), определяет состояние программы и процессора в каждый текущий момент времени.

31	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	AC	VM	RF		NF	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF		

- CF - перенос
- PF - четность
- AF - полуперенос
- ZF - флаг нуля
- SF - флаг знака
- TF - флаг трассировки
- IF - флаг прерывания
- DF - флаг направления
- OF - флаг переполнения

А в защищённом режиме добавляется ещё 5 флагов:

- AC - флаг выравнивания операндов
- VM - флаг виртуальных машин
- RF - флаг маскирования прерывания
- NT - флаг вложенной задачи
- IOPL - уровень привилегий ввода/вывода.

**Регистры** 1, 5, 15, 19-31 не используются.

**CF** — флаг переноса, устанавливается в 1, если в результате выполнения операций произошёл перенос из старшего разряда.

**PF** — флаг чётности, устанавливается в 1, если в младшем байте результата чётное число единиц.

**AF** — флаг полупереноса, устанавливается в 1, если при сложении произошёл перенос из третьего разряда в 4, а при вычитании из 4 в 3.

**ZF** — флаг нуля, устанавливается в 1, если результат равен 0.

**SF** — флаг знака, всегда равен знаку операнда ( $+$   $\rightarrow$  1, 0 или  $-$   $\rightarrow$  0).

**TF** — флаг трассировки, установленный в 1, переводит процессор в режим отладки.

**IF** — флаг прерывания, установленный в 1, может маскировать какое-то прерывание.

**DF** — флаг направления, определяет режим работы со строками, установленный в 1 приводит к обработке строк от старшего адреса к младшему,

автоматически уменьшая содержимое регистров индексов на размер операндов, при утановлении флага в 0, всё происходит наоборот.

**ОФ** — флаг переполнения, устанавливается в 1, если результат не уместился.

## 2 Лекция 2

**Регистр** — набор из  $n$  устройств, способный хранить  $n$ -разрядное двоичное число.

**Оперативная память** состоит из байтов, байт состоит из 8 информационных битов-разрядов, разряды с 0 по 3 называются цифровой частью байта, а с 4 по 7 зонной частью байта.

**Оперативная память** 32-х разрядного процессора может достигать 4Гб с адресами от 0 до  $2^{32}-1$ , что в 16-ричной системе счисления записывается от 00000000 до FFFFFFFF.

**Байты** могут объединяться в поля фиксированной и переменной длины. Адресом поля является адрес младшего байта, входящего в поле. Длина поля — количество байтов, входящих в поле. Поля фиксированной длины имеют собственные имена. Слово состоит из 2-х байтов, двойное слово из 4-х байтов. Поля переменной длины могут начинаться с любого байта.

**Процессор** может работать с непрерывной и сегментированной памятью. Если память сегментированна, то физический адрес байта состоит из 2-х частей: <сегмент>:<смещение>. И получается он по формуле: адрес начала сегмента + исполняемый адрес.

**Смещение** — исполняемый адрес, который формируется в зависимости от способа адресации операндов. В защищённом режиме может быть определено до 16083 сегментов, размером до 4Гб, процессор может обработать 64Тб виртуальной памяти.

**В реальном режиме** старшие 4 цифры адреса начала сегмента хранятся в сегментном регистре, адрес сегмента кратен 16, поэтому физический адрес получается смещением содержимого сегментного регистра на 4 разряда (двоичных) влево и прибавлением к нему исполняемого адреса.

**Физический адрес** следующей исполняемой команды получается: содержимое регистра CS, смещённое на 4 разряда + IP. **Пример:**

$$\begin{aligned}\text{ФА} &= (\text{CS}) + (\text{IP}) \\ (\text{CS}) &= 7A15_{16} = 0111\ 1010\ 0001\ 0101\ 0000_2 \\ (\text{IP}) &= C7D9_{16} = 1100\ 0111\ 1101\ 1001_2 \\ \text{ФА} &= 86929_{16} = 1000\ 0110\ 1001\ 0010\ 1001_2\end{aligned}$$

## Форматы данных

**Рассматриваемый нами процессор** может обрабатывать целые числа без знака, целые числа со знаком, действительные числа с плавающей точкой, двоично-десятичные числа, символы, строки и указатели.

**Целое число без знака** может занимать байт, слово или двойное слово и изменяться от 0 до 255(байт), от 0 до 65535(слово) и от 0 до 4294967295(двойное слово).

**Целое число со знаком** может занимать байт, слово или двойное слово при этом старший разряд (7(байт), 15(слово) или 31(двойное слово)) отводится под знак числа (0 положительной, 1 отрицательное), остальные разряды под цифры числа.

**Цифры со знаком** хранятся в дополнительном коде, дополнительный код положительного числа равен числу, а дополнительный код отрицательного числа может быть вычислен по формуле:  $10^n - |x|$ .

**Например**, представим в слове отрицательное число -AC7 ( $10^4 - AC7 = F539$ ).

**Дополнительный код** двоичного числа можно получить инверсией разрядов и прибавлением 1 к младшему разряду.

**Например**, -12 (в байте) ( $12 = 11110100$ )

**Вычитание в машине:** дополнительный код уменьшаемого прибавляется к вычитаемому, чтобы получить  $65 - 42 = 23$

1)  $65 = 0100\ 0001$

2)  $-42 = 1101\ 0110$

3)  $65 + (-42) = 0100\ 0001 + 1101\ 0110$

**Число с плавающей точкой** может занимать 32 разряда, 64 разряда или 80 разрядов, и называется оно короткое вещественное, длинное вещественное и рабочее вещественное.

**Число с плавающей точкой** состоит из 3-х частей: знак (1 разряд), машинный порядок (8(если в общем число занимает 32 разряда) или 11(64 разряда), или 15(80 разрядов) разрядов), мантисса (23(32 разряда) или 52(64 разряда), или 64(80 разрядов) разряда).

**Машинный порядок** не явным образом содержит в себе знак порядка, он изменяется от всех 0 до всех 1, в поле, которое ему определено, а с истинным порядком машинный порядок связан формулой:

$$P_M = P_i + 127_{10} (1023_{10}, 16383_{10})$$



Пример,  $3060_{10}$  представить в виде числа с плавающей точкой, занимающего 4 байта.

$$1) 3060_{10} = BF4_{16}$$

$\frac{1}{\text{ОСНОВАНИЕ СИСТ. СЧИСЛЕНИЯ}}$

2) нормализуем число  $0. BF4 \cdot 10^3_{16}$

3) получим машинный порядок  $Пм = 3_{16} + 7F_{16} = 82_{16}$

4) запишем в разрядную сетку в 2-ичной системе счисления:

0 1000 0010 011 1111 0100 0000 0000 0000<sub>2</sub>

Или в 16-ричном виде:  $413F4000_{16}$

0100 0001 0 011 1111 0100 0000 0000 0000<sub>2</sub>

- 1) переведем в 16-ричное
- 2) нормализуем число
- 3) получаем машинный порядок
- 4) записываем в разрядную сетку, но хитрость заключается в том, что для экономии памяти старшая цифра мантиисы нормализованного числа не записывается в разрядную сетку (т.к. она равна 1(вроде всегда))

**Двоично-десятичные числ** могут обрабатываться процессором как 8-ми разрядные в упакованном или неупакованном формате, а сопроцессором могут обрабатываться 80-ти разрядные в упакованном формате.

**Упакованный формат** предполагает хранение 2-х цифр в байте, а неупакованный только 1 цифры в цифровой части байта.

**Символьные данные** хранятся в ASCII коде, каждому символу отводится 1 байт памяти.

**Строковые данные** — это последовательности байтов, слов или двойных слов и указатели. Существует два типа указателей: длинный (48 разрядов селектор(16) + смещение(32)) и короткий (32 разряда).

## Форматы команд

**В машинном формате команда** — последовательность двоичных цифр, состоящая из двух частей, определяющих код операции и адресную часть, то есть где хранятся данные и куда можно записать результат.

**Рассматриваемый процессор может работать** с безадресными командами, с одноадресными командами, двухадресными и трёхадресными командами. В памяти команда может занимать от 1 до 15 байтов в зависимости от кода операции количество операндов и места их расположения. А располагаться операнды могут непосредственно в команде, в регистрах или в оперативной памяти.

**Наибольшее количество команд** двухадресных и тогда формат называют: R-R, M-M, R-M, M-R, R-D, M-D.

Существуют различные способы адресации операторов, а данные могут занимать байт, слово или двойное слово. И исполняемый адрес операнда зависит от способа адресации и может состоять из трёх частей (база, индекс и смещение). Например, [BX][SI]M

#### Способы адресации (в реальном режиме):

- 1) регистровая,
- 2) непосредственная,
- 3) прямая,
- 4) косвенно-регистровая,
- 5) по базе со смещением,
- 6) прямая с индексированием,
- 7) по базе с индексированием.

Машинный формат двухадресной команды, для которой один операнд находится всегда в регистре, а второй – в регистре или памяти можно представить следующим образом:

байты	1	2	3	4
биты	7 2 1 0	7 6 5 4 3 2 1 0	7 0	7 0
поля	код операции d w	MOD reg r/m	disp H	disp L

“disp H/disp L” – “старшая / младшая часть смещения.

Поля “код операции” и иногда “reg” определяют выполняемую операцию.

Поле “d” определяет место хранения первого операнда.

Поле “w” определяет с какими данными работают: с байтами, или словами.

Если w = 0, команда работает с байтами, w = 1 - со словами.

reg” - определяет один операнд, хранимый в регистре.

Поля “mod”, “disp H” и “disp L” определяют второй операнд, который может храниться в регистре или в памяти.

Если mod = 11, то второй операнд находится в регистре, он определяется полем “r/m”, а “disp H/disp L” – отсутствует, команда будет занимать 2 байта в памяти, если mod <> 11, то второй операнд находится в памяти.

Машинный формат двухадресной команды


Значение поля “mod” определяет как используется смещение:

mod  $\begin{cases} 0, \text{ disp} - \text{отсутствует} \\ 1, \text{ disp} = \text{disp L} - \text{с распространением знака до 16} \\ 10, \text{ смещение состоит из disp H и disp L.} \end{cases}$

Поля “reg” и “r/m” определяют регистры:

reg / r/m	000	001	010	011	100	101	110	111
w = 0	AL	CL	DL	BL	AH	CH	DH	BH
w = 1	AX	CX	DX	BX	SP	BP	SI	DI

Физический адрес определяется так:



r/m	ИА	ФА
000	(BX) + (SI) + disp	+ (DS)
001	(BX) + (DI) + disp	+ (DS)
010	(BP) + (SI) + disp	+ (SS)

Машинный формат двухадресной команды.

r/m	ИА	ФА
011	(BP) + (DI) + disp	+ (SS)
100	(SI) + disp	+ (DS)
101	(DI) + disp	+ (DS)
110	(BP) + disp	+ (SS)
111	(BX) + disp	+ (DS)

В ассемблере результат всегда посылается по адресу первого операнда.

**Адресация регистровая:**

MOV AX, BX ; (BX) → AX

Машинный формат: 1001 0011 1100 0011

“код операции” 100100

“d” = 1

“w” = 1

“mod” = 11

“reg” = 000

“r/m” = 011

**Непосредственная адресация:**

MOV AX,25 ; 25 → AX

В ассемблере как и в языках высокого уровня есть именованные константы, они определяются с помощью EQU

**Конструкция:** <имя> <EQU> <значение>

**Пример:** (CONST EQU 34h)

**Прямая адресация** (адрес операнда прямо в команде записывается):

**MOV AX, ES : 0001 ;**

ES – регистр сегмента данных, 0001 – смещение внутри сегмента.

Содержимое двух байтов, начиная с адреса (ES) + 0001 пересылаются в AX - ((ES) + 0001) → AX.

Прямая адресация может быть записана с помощью символического имени, которому предварительно был присвоен некоторый адрес оперативной памяти, а присвоен может быть с помощью директивы.

например: DB – байт,

DW – слово,

DD – двойное слово.

Если в сегменте ES содержится директива Var\_p DW, тогда по команде

**MOV AX, ES : Var\_p ; ((ES) + Var\_p) → AX.**

Например, если команда имеет вид:



**MOV AX, Var\_p; ((DS) + Var\_p) → AX.**

**Косвенно-регистровая операция** (в регистре содержится адрес операнда), в записи косвенно-регистровая от регистровой отличается записью регистра в квадратных скобках.

**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.

**Адресация 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.

**BP** работает с сегментом стека.

**Прямая с индексированием адресация:**

**MOV AX, MAS[SI]** ; ((DS) + (SI) + MAS) → AX

**MAS** – адрес в области памяти.

С помощью этой адресации работают с одномерными массивами или с полями структур, символическое имя определяет адрес начала массива или структуры, а содержимое **SI** используется для перехода от одного элемента массива к другому или от одного поля структуры к другому.

С **двумерными массивами** используется адресация по базе с индексированием.

**MOV AX, Arr[BX][DI]** ; ((DS) + (BX) + (DI) + Arr) → AX.

Символическое имя определяет адрес начала массива, с помощью индексного регистра реализуется переход от одного элемента к другому (в строке), а с помощью базового регистра от одной строке к другой.

### 3 Лекция 3

#### Особенности использования команд пересылки

**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.

**MOV DX, 100h**  
**MOV DS, DX**

- 4) Нельзя изменять командой **MOV** содержимое регистра **DS**;
- 5) Данные в памяти хранятся в перевёрнутом виде, а в регистрах в нормальном виде, и команда пересылки учитывает это, например, **R DW 1234h**

В байте с адресом **R** будет **34h**, в байте с адресом **R+1** будет **12h**.

**MOV AX, R**; 12h → AH, 34h → AL.

6) Размер передаваемых данных определяется типом операндов в команде;

**X DB ?** ; X - адрес одного байта в памяти.  
**Y DW ?** ; Y определяет поле в 2 байта в памяти.  
**MOV X, 0** ; очищение одного байта в памяти.  
**MOV Y, 0** ; очищение двух байтов в памяти.  
**MOV AX, 0** ; очищение двух байтов регистра  
**MOV [SI], 0** ; сообщение об ошибке.



В последнем случае необходимо использовать специальный оператор PTR.

**<тип> PTR <выражение>**

Выражение может быть константным или адресным, а тип это:

BYTE, WORD, DWORD и т.д.

**byte PTR 0** ; 0 воспринимается как байт

**word PTR 0** ; 0 воспринимается как слово

**byte PTR op1** ; один байт в памяти начиная с этого адреса

**MOV byte PTR [SI], 0;**



≡ **MOV [SI], byte PTR 0;**

= **MOV [SI], word PTR 0** ; 0 → ((DS) + (SI))

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

**MOV AH, 500** ; сообщение об ошибке.

**MOV AX, X** ; ошибка, X – 1 байт, AX – 2 байта.

**MOV AL, R** ; ошибка

**MOV AL, byte PTR R** ; (AL) = 34h

**MOV AL, byte PTR R+1** ; (AL) = 12h

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

К командам пересылки относят:

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]**

## Структура программы на Ассемблере

**Ассемблер** — это низкоуровневый язык программирования и программа, написанная на Ассемблере, должна пройти 3 этапа обработки, как и программа, написанная на любом другом языке программирования.

1-й этап — преобразование исходного модуля в объектный — ассемблирование. Исходных модулей может быть один или несколько.

2-й этап — с помощью программы редактора связей объектные модули объединяются в загрузочный, исполняемый модуль.

3-й этап — выполнение программы.

**Существует два типа исполняемых модулей:** exe-файл и com-файл. В результате выполнения второго этапа получается исполняемый exe-файл, чтобы получить com-файл, необходимо выполнить ещё один этап обработки — преобразование exe-файла в com-файл.

**Исходный файл на Ассемблере состоит из команд и директив.** Команды преобразуются в машинные коды, реализующие алгоритм решения задачи. Директивы описывают, каким образом необходимо выполнять ассемблирование и объединение модулей. Они описывают форматы данных, выделяемые области памяти и т. д.

## Команды и директивы на Ассемблере

**Команда на Ассемблере состоит из четырёх полей:**

**[<имя>[:]] <код операции> [<операнды>] [:комментарии]**

**Поля отделяют друг от друга хотя бы 1 пробелом.** Все поля, кроме <код операции>, могут отсутствовать. <Имя> - символическое имя Ассемблера. Имя используется в качестве метки обращения к команде, передачи управления на данную команду. [:] после имени означает, что метка является внутренней. Код операции определяет какое действие должно выполнить процессор. Поле <операнды> содержит адреса данных, или данные, участвующие в операции, а также место расположения результатов операции. Операндов может быть от 1 до 3, они отделяются друг от друга запятой.

Комментарии отделяются кроме пробела ещё и ";" и могут занимать всю строку или часть строки.

Например:

**JMP M1**

; команда безусловной передачи управления на команду с меткой **M1**.

-----/-----  
 **M1: MOV AX, BX**

; пересылка содержимого регистра **BX** в регистр **AX**.

В комментарии будем записывать в виде (BX) \_\_\_\_AX

Директива, как и команда, состоит из 4 полей:

[<имя>] <код псевдооперации> <операнды> [<комментарии>]

Здесь <имя> — символическое имя Ассемблера, <код псевдооперации> — определяет назначение директивы. Операндов может быть различное количество и для одной директивы.


Например:

**M1 DB 1, 0, 1, 0, 1** ; директива **DB** определяет 5 байтов памяти и заполняет их 0 или 1 соответственно, адрес первого байта — M1.

**M2 DB ?,?,?** ; директива **DB** определяет три байта памяти ничем их не заполняя, адрес первого — M2.

**Proc** ; директива начала процедуры,

**endp** ; директива конца процедуры,

 **Segment** ; директива начала сегмента,

**ends** ; директива конца сегмента.

**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: сегмента стека, сегмента данных, сегмента кода.

```
; сегмент стека
Sseg Segment...
-----/-----
Sseg ends
; сегмент данных
Dseg Segment...
-----/-----
Dseg ends
; сегмент кода
Cseg Segment...
-----/-----
Cseg ends
end start
```



Каждый сегмент начинается директивой начала сегмента `-Segment` и называется директивой конца сегмента `-ends`, в операндах директивы `Segment` содержится информация о назначении сегмента.

## Назначение сегментов

В кодовом сегменте специальная директива . . .

**ASSUME SS:SSeg, DS:DSeg, CS:CSeg, ES:DSeg;**

на Dseg ссылаются и DS, и ES.

**Кодовый сегмент оформляется как процедура**, это может быть одна процедура или несколько последовательных процедур, или вложенных процедур.

Структура кодового сегмента с использованием двух вложенных процедур выглядит следующим образом:

```
Cseg Segment...
ASSUME SS:SSeg, DS:DSeg, CS:CSeg
pr1 Proc
-----/-----
      pr2 Proc
-----/-----
      pr2 endp
-----/-----
pr1 endp
Cseg ends
```

В сегменте стека выделяется место под стек.

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

Кодовый сегмент содержит программу решения поставленной задачи.

```
... ; Prim1.ASM
; сегмент стека
      Sseg Segment...
              DB 256 DUP(?)
      Sseg ends
; сегмент данных
      Dseg Segment...
              X DB 'A'
              Y DB 'B'
              Z DB 'C'
      Dseg ends
;
```

```

Cseg Segment...
    ASSUME SS:SSeg, DS:DSeg, CS:Cseg
    Start Proc FAR
        Push DS
        Push AX
        MOV DX, DSeg
        MOV DS, DX
        CALL Main
        Ret
    Start endp
    Main Proc NEAR
        ADD AL, X
        MOV AX, Y
        -----/-----
        Ret
    Main endp
Cseg ends
    and Start

```

## Структура программы

- 1) строки 1, 5, 11 — это комментарии.
- 2) кодовый сегмент содержит две последовательные процедуры. Первая процедура — внешняя, об этом говорит параметр FAR.
- 3) Строки 15-18 — реализуют связь с операционной системой и определяют адрес начала сегмента данных.
- 4) Строка 19 — это обращение к внутренней процедуре Main, строка 20, команда Ret — возврат в ОС.
- 5) Main — внутренняя процедура, о чём говорит параметр NEAR в директиве начала Proc.
- 6) Директива end имеет параметр Start, определяющий точку входа в программу, то есть, с которой должно начинаться выполнение программы.

**Внутренняя процедура** — это процедура, к которой можно обратиться только из того сегмента, в котором она содержится. К внешней процедуре можно обратиться из любого сегмента. По умолчанию (если в директиве начала процедуры параметр отсутствует) процедура является внутренней.

## Слова, константы, выражения, переменные

**Символические имена в Ассемблере могут состоять** из строчных и прописных букв латинского алфавита, цифр от 0 до 9 и некоторых символов '!', '?', ...

**В программе на Ассемблере могут использоваться** константы 5 типов: целые двоичные, десятичные, шестнадцатеричные, действительные с плавающей точкой, символьные.

**Целые двоичные** — это последовательности 0 и 1 со следующими за ними символом 'b', например 10101010b или 11000011b.

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

**Целые шестнадцатиричные 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, включенные 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 непосредственно 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** = 6 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 ;

## 4 Лекция 5

### Сложение и вычитание в Ассемблере

Арифм-ие 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.

Примеры:

X = 1234AB12h, Y = 5678CD34h, X + Y =

**MOV AX, 1234h**

**MOV BX, 0AB12h**

**MOV CX, 5678h**

**MOV DX, 0CD34h**

**ADD BX, DX**

**ADC AX, CX**

Умножение и деление:

Умножение беззнаковых чисел.  
**MUL OP2** ; (OP2)\*(AL)  $\vee$  (AX)  $\vee$  (EAX)  $\rightarrow$  AX  $\vee$  DX:AX  $\vee$  EDX:EAX  
 Умножение знаковых чисел.  
**IMUL OP2**; аналогично MUL  
**IMUL OP1, OP2** ; i386 и > **IMUL op1, op2, op3** ; i186 и >  
 OP1 всегда регистр, OP2 – непосредственный операнд, регистр или память.  
 При умножении результат имеет удвоенный формат по отношению к сомножителям. Иногда мы точно знаем, что результат может уместиться в формат сомножителей, тогда мы извлекаем его из AL, AX, EAX.  
 Размер результата можно выяснить с помощью флагов OF и CF.  
 Если OF = CF = 1, то результат занимает двойной формат, и OF = CF = 0, результат укладывается в формат сомножителей.  
 Остальные флаги не изменяются.

Деление беззнаковых чисел: Деление знаковых чисел.  
**DIV OP2** ; OP2 = r  $\vee$  m **IDIV OP2** ; OP2 = r  $\vee$  m  
 (AX)  $\vee$  (DX:AX)  $\vee$  (EDX:EAX) делится на указанный операнд и результат помещается в AL  $\vee$  AX  $\vee$  EAX,  
 остаток помещается в AH  $\vee$  DX  $\vee$  EDX.

(OP2 - регистр или память)

Содержимое ааккумулятора AX или AX:DX или EAX:EDX делится на указанный операнд и результат записывается в AL или AX или EAX, в зависимости от типа результата. Остаток помещается в AH, DX или EDX.

Значение флагов при делении не меняется, но могут быть ошибки деления на 0 или переполнения.

**MOV AX, 600**  
**MOV BH, 2**  
**DIV BH** ; 600 div 2 = 300 - не уместается в AL.

При выполнении команд умножения и деления необходимо следить за размером операндов и при необходимости за значениями флажков сдвига и переполнения.

Посмотрим фрагмент программы, в котором цифры целого беззнакового байтового числа N записывают в байты памяти, начиная с индекса D как символы.

c = N mod 10  
 b = (N div 10) mod 10  
 a = (N div 10) div 10  
 Перевод в символы: код(i) = код ('0') + i

-----  
**D** N DB ?  
 D DB 3 Dup (?)

```

MOV BL, 10 ; делитель
MOV AL, N ; делимое
MOV AH, 0 ; расширяем делимое до слова
; или CBW AH конвертируем до слова
DIV BL ; AL = ab, AH = c
ADD AH, '0'
MOV D+2, AH
MOV AH, 0
DIV BL ; AL = a, AH = b
ADD AL, '0'
MOV D, AL
ADD AH, '0'
MOV D+1, AH

```



## Директивы внешних ссылок

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

Пример: `Public <имя>,[<имя>]`

Эта директива определяет указанные имена как глобальные величины к которым можно обращаться из другого модуля. Именем может быть метка или переменная.

**Если некоторое имя** определено в модуле А как глобальное, а к нему нужно обращаться из других модулей, например В и С, то в этих модулях В и С, должна быть директива

`EXTRN <имя>:<тип>` (можно несколько)

Имя одно и тоже, что написано в директиве `Public`, а тип зависит от значения имени, если имя это имя переменной, то на месте слова тип может стоять одно из ключевых слов (`BYTE`, `WORD`, `DWORD`, `FWORD`, `QWORD`, `TWORD`), если имя это метка, то типом может быть `NEAR` или `FAR`.

**Директива EXTRN** говорит, что эти имена являются внешними для данного модуля.

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**

**Директива INCLUDE** позволяет подключить на этапе ассемблирования файлы, расположенные на диске, например: INCLUDE <имя файла>

**INCLUDE C:\WORK\Prim.ASM**

На этапе ассемблирования содержимое этого файла запишется на место этой директивы.

## Команды управления

**Команды управления** управляют кодом вычислительного процесса. К ним относятся команды условной передачи управления, безусловной передачи управления и команды организации управления.

**Команды безусловной передачи управления имеют вид:**

**JMP** <имя>

**Имя - метка команды**, которая будет выполняться следующей за JMP, команда, на которую передаём управление, может располагаться в том же кодовом сегменте, что и JMP, а может и в другом кодовом сегменте.

**JMP M1** ; по умолчанию M1 имеет тип NEAR

**Если метка** содержится в другом кодовом сегменте, то в том сегменте, куда передаём управление, должна быть директива Public M1, а в сегменте с JMP должна быть директива ETRN M1: FAR

**Передачи бывают прямыми или косвенными**, можно использовать прямую (JMP M1) и косвенную (JMP [BX]) адресацию.

**Команда безусловной передачи управления** на ближнюю метку занимает в памяти 3 байта, передача на дальнюю метку занимает 5 байтов памяти. Если мы знаем, что передаём управление не далее чем на -128 или 127 байтов, то можно использовать команду занимающую 1 байт памяти.

**За командой JMP** должна следовать команда с меткой (обязательно, чтобы можно было вернуться к команде, следующей за JMP)

**К командам безусловной передачи управления** относятся команды обращения к подпрограммам и возврата из подпрограммы.

**Процедура обязательно имеет** тип NEAR или FAR, последний нужно указывать обязательно.

**NEAR может быть вызвана** только из того модуля в котором содержится. Основная или головная программа всегда имеет тип FAR, поскольку к ней обращается из отладчика. Если подпрограмм немного, то их размещают в том же сегменте, что и основная программа, а если их много, то для них выделяют отдельный кодовый сегмент.

## Процедуры Near и Far

```

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

```



## Процедуры Near и Far

<pre> 2) extrn p2: far       cseg  segment.....           assume .....       p1  proc far       -----           call p2       -----           ret       p1  endp       cseg  ends </pre>	<pre>       public p2       cseg1 segment.....           assume .....       p2  proc far       -----           ret       p2  endp       cseg1 ends </pre>
--	---

**Команда CALL <имя>**, которая может использовать как прямую адресацию, так и косвенную. При обращении к подпрограмме в стеке сохраняется адрес возврата, то есть адрес команды, следующей за командой CALL, но если мы обращаемся к подпрограмме ближнего типа, то в стеке сохраняется только смещение, а если к внешней процедуре, к подпрограмме,



кодирующей в другом кодовом сегменте, то в стек записывается полный адрес (начало сегмента и смещение относительно него).

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.



Команда **RET** может реализовать выход из подпрограммы как NEAR, так и FAR

RETN - из подпрограммы ближнего типа вызова

RETF - из подпрограммы дальнего типа вызова

Параметр n необязательный, он говорит сколько надо байтов в стеке нужно очистить

### Примеры прямого и косвенного перехода



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) -----

jmp z ; ошибка

-----



Z DW L

3) jmp word ptr z

z DW L

### Команды условной передачи управления

Команды условной передачи управления делят на 4 типа:

- 1) команды используемые после команд сравнения
- 2) команды используемые после команд отличных от команд сравнения
- 3) команды сравнения, но реагирующие на значения флагов
- 4) команды, реагирующие на значение регистра CX

**Общий вид объявления Jx <метка>** (J - всегда первая, а затем следует несколько букв), метка в этой команде имеет право отстоять не более чем на 127 байтов.

Примеры:

JE M1 ; передача управления на команду с меткой M1, если ZF = 1  
 JNE M2 ; передача управления на команду с меткой M2, если ZF = 0  
 JC M3 ; передача управления на команду с меткой M3, если CF = 1  
 JNC M4 ; передача управления на команду с меткой M4, если CF = 0

**ADD AX, BX**

**JC M**

если в результате сложения CF = 1, то управление передается на команду с меткой M, иначе – на команду, следующую за JC

**SUB AX, BX**

**JZ Met**

если результатом вычитания будет 0, то ZF = 1 и управление передается на команду с меткой Met.

Часто команды передачи управления используются после команд

сравнения **<метка> CMP OP1, OP2**

По этой команде выполняется (OP1) – (OP2) и результат всегда не посылается, формируются только флаги.



### Команды условной и безусловной передачи управления

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



Если нужно реализовать условную передачу управления больше чем на 127 байт, то можно изменить условие передачи управления

**if AX = BX goto m** следует заменить на:

**if AX < > BX goto L**

**Goto m ; m – дальняя метка**

-----

**L: ----- ; L – близкая метка**


На Ассемблере это будет так:

cmp AX, BX

jne L

jmp m

мет: -----  
L: -----



С помощью команд jx и jmp можно реализовать цикл с предусловием:

1) while x > 0 do S;

beg: cmp x, byte ptr 0

jle fin

S

jmp beg

fin: -----

и с постусловием:

2) do S while x > 0;

beg:

S

cmp x, byte ptr 0

jg beg

fin: -----



## 5 Лекция 6

Заголовок Команды для организации циклов  
(слайд с loop метками)

В форме 1 из содержимого CX вычитается единица, если окажется, что CX != 0, то управление передаётся на указанную метку (CX содержит количество итераций)

Во второй форме из CX вычитается 1, если CX != 0 и ZF == 1, то управление передаётся на указанную метку. А это значит, что цикл завершается, происходит выход из цикла (передаётся управление следующее за loop), или CX = 0, или ZF = 0, или это произойдёт одновременно.

В 3 варианте уменьшается содержимое CX если CX != 0 и одновременно ZF = 0, то управление передаётся на указанную метку, если условие нарушено, то происходит выход из цикла.

Примеры циклов (слайд с loop метками)

Если CX используется для других целей, тогда можно поступить следующим образом

mov SI, 0 (далее с того же слайда)

Дана матрица целых байтовых величин размером 4 на 5, нужно подсчитать количество 0 в каждой строке матрицы, заменить 0 на константы, например 0FF, будем решать задачу с помощью директив стандартной сегментации, выделив под стек 256 байтов, а кодовый сегмент оформим как 2 последовательные процедуры. Внешняя реализует связь с операционной системой и обращается к внутренней процедуре, решающей поставленную задачу

(пример с слайда) + дописать комментарии

```
title prim.asm
page , 132
Sseg segment para stack 'stack'
db 256 dup (?)
Sseg ends
Dseg segment para public 'data'
Dan db 0,2,5,0,91
db 4,0,0,15,47
db 24,15,0,9,55
db 1,7,12,0,4
Dseg ends
Cseg segment para public 'code'
Assume cs: cseg, ds:dseg, ss:sseg
start proc far
PUSH DS
PUSH AX
mov BX,Dseg
mov DS,BX
call main
ret
start endp
main proc near
mov BX, offset Dan
mov CX, 4
nz1: PUSH CX
mov DI, 0
mov SI, 0
```

```

mov CX, 5
nz2: PUSH CX
cmp byte ptr[BX+SI], 0
jne mz
mov byte ptr[BX+SI], 0FFh
inc DL
mz: inc SI
POP CX
kz2: loop nz2
add DL, 0
mov AH, 6
int 21h
add BX, 5
pop CX
kz:1 loop nz1
ret
main endp
Cseg ends
end start

```

Заголовок организация циклов в ассемблере

Массивы описываются, определяются с помощью директив определения данных и памяти, возможно с помощью конструкции повторения `dup`

Например:

`x DW 30 dup (?)` - выделила в памяти место под одномерный массив `x` (массив слов), состоящий из 30 элементов, но в этом описании неясно как мы будем нумеровать элементы массива, мы можем пронумеровать их от 0 до 29, может от 1 до 30, а может от `k` до `29+k`, в зависимости от постановки задачи, если это не обговорено, то удобнее нумеровать в ассемблере с 0, потому что адрес любого элемента массива будет записываться наиболее происходит

Можем записать

адрес  $(x[i]) = x + (\text{type } x) * i$

Для двумерного массива - `A[0..n-1, 0..m-1]`

адрес  $(i,j)$  можно вычислить так

адрес  $(A[i,j]) = A + m * (\text{type } A) * i + (\text{type } A) * j$

Например для нашего одномерного массива это было

$x + 2*i = x + \text{type}(x) * i$

Адрес состоит из двух частей. Из постоянной части `x` и переменной  $2 * i$ , поэтому логично использовать для адресации элементов одномерного массива прямую индексацию со смещением, т.е. `x` - смещение, а  $2*i$  - в регистре `SI` или `DI`

место под двумерный массив можно выделить следующим образом:

`A DD n DUP (m Dup (?))` - двумерный массив слов размером `n` на `m`

Если мы будем нумеровать от 0 элементы строк и столбцов, то адрес  $(A[i,j]) = A + m * 4 * i + 4 * j$

A - постоянная, переменные  $m * 4 * i$  и  $4 * j$ , то есть адресация по базе с индексированием.

Фрагмент программы, в которой в регистр AL записывается количество строк матрицы байтов размерность  $10 * 20$ , тип элементов - byte, имя массива - x. Посчитаем количество строк матрицы, в которых 1-ый элемент повторяется не менее 1 раза.

(Программа на слайде)

Заголовок Команды побитовой обработки данных

К ним относятся:

- 1) логические команды,
- 2) команды сдвига
- 3) Установки
- 4) сброса
- 5) инверсии битов

Логические (с слайда)

Второй операнд в этих командах называют маской, а основным назначением команды and - установка в 0 с помощью маски соответствующих разрядов первого операнда, потому что нулевые разряды маски обнуляют соответствующие разряды первого операнда, а единичные оставляют без изменения.

Маску можно задавать как константу в команде, можно хранить в регистре или в памяти. Соответственно можно использовать любой способ адресации

Пример (соответствующий слайд)

Команда `and OP1, OP2` - ложь только если оба разряда ложь, эта команда используется для установки в 1 некоторых разрядов первого операнда в соответствии с маской - OP2

Пример (соответствующий слайд)

В команде могут использоваться различные способы адресации:

пример (соответствующий слайд)

Команда `xor OP1, OP2`, если операнды одинаковые, то результат 0, если различные, то результат 1

Пример (соответствующий слайд)

Команда `not` выполняет инверсию операнда. Значение при этом не изменяется.

Команды `xor` обнуляет регистр AX быстрее, чем `mov` и `sub`

(слайд после этой строки)

Определить количество задолжников в группе из 20 студентов, информация содержится в массиве байтов X DB 20 DUP (?), причём (соответствующие слайды (2))

Команды сдвига

Арифметические логические команды сдвига : первая буква s, вторая определяет тип сдвига логический или арифметический, 3 определяет влево или вправо, причём первый оператор может быть регистром или памятью, а второй это константа или регистр CL, в котором используется 5 младших разрядов.

Пример (соответствующий слайд) + следующий слайд про сдвиги больше чем на 1

Циклические сдвиги (соответствующий слайд)

+ ещё следующий слайд

Циклические сдвиги с переносом содержимого флажка CF (соот)

для всех команд сдвига флажки ZF, SF, PF, устанавливаются в соответствии с результатом, AF тоже неопределён. OF - не определён при сдвигах на несколько разрядов, а при сдвигах на 1 разряд в зависимости от команды (слайд)

Для самостоятельного изучения (соответствующий слайд)

## 6 Лекция 7

Заголовок структуры в ассемблере

Структуры - комбинированный тип, содержащий данные различного типа, занимающие последовательные поля памяти. (109 тип)

Чтобы использовать переменную структурного типа, нужно описать структуры (шаблон структуры), на основании которого место в памяти при ассемблировании не выделяется, а для транслятора только информация о формате структуры. Это описание структуры выглядит следующим образом:

```
<имя типа> struct  
описание полей  
<имя типа> ends
```

Для описания полей используется директивы определения данных и памяти (byte, word, DW), а имена указанные в этих директивах и будут именами полей.

Имена полей в директивах по стандарту считаются уникальными в рамках программы, и ещё по стандарту не допускаются вложенные структуры.

Например (слайд 110):

? - означает, что значения по умолчанию нет.

Имена полей - имена в директивах, а значения называются значениями, принятыми по умолчанию.

Расположение данных (слайд 110)

(слайд 111) Описание переменных:

уголки - не метасимволы, а реальные символы языка, внутри которых через запятую указываются начальные значения полей.

(слайд 111) значения полей + пример

Приоритетными значениями являются начальные значения. Если при описании переменной значением поля является ?, или какое-то другое значение, то значение полей по умолчанию игнорируется. Пример (слайд 112)

Если при описании переменной используется значение пусто, то в качестве начального значения принимается значение по умолчанию.

Значение по умолчанию устанавливается для тех полей, которые являются одинаковыми для нескольких переменных, например, для массива структур.

Если отсутствует начальное значение, нескольких полей при описании переменных и эти поля являются последними в описании, то запятые при описании можно не ставить.

пример (слайд 113)

Если нет совсем начальных значений у переменной, то нужно оставлять пустые угловые скобки

пример (слайд 113)

Можно писать за 1 раз несколько переменных, а можно писать массив структур, используя конструкцию DUP

пример (слайд 113)

Описан одномерный массив структур типа TData, в котором первый элемент массива имеет начальное значение 2000, 4, 1, а остальные элементы массива в качестве начальных получают значения, принятые по умолчанию.

Имя и адрес первой структуры dst, второй - dst+4, третьей dst+8,..., это начало первого элемента структуры.

Обращаться к полям можно также как и в языках высокого уровня:

пример + шаблон(слайд 114)

Ассемблер имени типа структуры и имени переменной структурного типа присваивает тип, определяющий размер структуры (количество байт, занимаемых структурой)

пример (слайд 114)

И тогда эту информацию можно использовать при программировании, то есть скопировать значение полей из одной структуры в другую

пример (слайд 114)

Из памяти в память пересылать нельзя.

пример (слайд 114)

Точка, при обращении к полю структуры, это оператор, который позволяет вычислить адрес поля по формуле

формула (слайд 115)

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

(слайд 115)

Смещение поля в структуре - это имя поля, а адресное выражение может быть любой сложности

пример + замечания(слайд)

Если при описании типа структуры, при описании некоторого поля, используется несколько параметров или конструкция повторения DUP, то при описании переменной этого типа, у данного поля не должно быть начального значения, может быть только пустым, но есть одно исключение: если по умолчанию значением поля является строка, то у этого поля по этой переменной, то при описании переменной у этого поля может быть начальное значение, но размером не больше, чем строка по умолчанию, если окажется меньше, то справа это поле дополнится пробелами.

пример (слайд)



пример (используем прямое обращение к полям)(следующий слайд + следующий слайд)

пример (обращение к полям структуры в цикле)(слайд)

пример (обращение к полям структур цикл в цикле)(слайд + следующий слайд + следующий слайд)

Заголовок Записи в ассемблере

Записи в ассемблере это упакованные данные. Это данные, которые занимают нецелые байты или слова, а части байтов или слов, т.е. запись в ассемблере может занимать 1 байт или 2 байта, то есть слово, а поля записи занимают последовательные биты в байте или слове.

Поля должны быть прижаты к правой границе записи, прижаты к правой границе поля и между ними не должно быть неопределённых разрядов, размер записи может быть только байтом или словом, размер поля записи может быть любым, но так, чтобы в сумме размеры полей не превышали байта или слова. Если сумма размеров полей, меньше байта или слова, то старшие разряды этого байта или слова заполняются нулями, но никакого отношения к записи не имеют.

Поля, так же как и у структуры, имеют собственные имена, но обращаться к ним напрямую нельзя, так как это несколько разрядов, а наименьший адресуемый элемент для процессора это байт. Так же как и для структуры, должно быть описание типа записи (шаблон), на основании которого не выделяется место, это описание может располагаться в любом месте программы, но до первого описания переменной такого типа.

пример (слайд)

Поле по определению есть

(слайд)

Размер и выражение - это константные выражения. Размер определяет размер поля в битах (сколько разрядов занято полем), а выражение, если оно есть, определяет значение поля по умолчанию, но в отличие от структуры, знак вопроса не допускается

пример (слайд)

Используя идентификатора типа опишем переменные следующего типа  
пример (слайд)

Угловые значения - это символы языка, внутри которых через запятую записываются начальные значения, а начальным значением может быть константное выражение, знак вопроса или пусто, в отличие от структуры знак вопроса определяет нулевое начальное значения, а начальное значение пусто приведёт к тому, что у поля будет значение, принятое по умолчанию

пример (слайд)

С запятыми можно поступить так же как и у структуры: если они последние, то можно их не писать, если значений нет, то можно тоже не писать.

пример (слайд)

массив записей (слайд)

Все 100 записей будут иметь значение, принятые по умолчанию.

пример (слайд)

Можно присвоить поля записи одной записи другой, тогда мы пользуемся конструкцией

пример (слайд)

Для работы с полями есть специальные операторы `width` и `mask`

(слайд)

оператор `width` определяет размер в битах указанного операнда, это будет размер поля или размер всей записи.

оператор `mask` определяет маску

(слайд)

результатом будет маска, в котором разряды оператора заполняются 1, а все остальные 0

пример (слайд)

пример (выявить родившихся первого числа)(слайд)

## 7 Лекция 8

заголовок системное программирование

Основная программа оформляется как внешняя проце

подпрограмма - вспомогательный алгоритм, к которому возможно многократное обращение. В ассемблере представлена только в виде процедуры, которая оформляется так же как и основная программа с помощью директив `proc` и `endp`

шаблон (слайд)

Описание процедуры может размещаться в любом месте программы, но так, чтобы управление на неё попадало с помощью команды `call`, но можно и с помощью передачи управления (так делать не стоит), поэтому подпрограммы обычно размещают после основной программы или перед основной программой, а если подпрограмм много, то можно их собрать в отдельный кодовый сегмент, но тогда для обращения к ним нужно будет использовать директивы `public` и `extern`

пример определения процедур(слайд)

После имени в директиве `proc` ":" не ставится, но используется это имя как метка первой команды подпрограммы.

Метки в подпрограммах могут быть локализованными или не локализованными, в зависимости от типа компилятора (в `tasm` они не локализованы, в `masm` и `rasn` локализованы), но основная проблема и задача при использовании подпрограмм - это передача параметров. Передавать их можно по значению, по ссылке, по возвращаемому значению, по результату и отложенным вычислением.

Параметры можно передавать через регистры, можно передавать через глобальные переменные, через стек, через потоки кода и через блоки данных.

Использование регистров для фактических параметров - наиболее простой способ. Вызывающая программа, перед обращением к подпрограмме

загружает фактический параметр в некоторые регистры, а подпрограмма извлекает их оттуда и использует, такой способ реализуется в некоторых функциях `os` и `bios`, но регистров мало, и, если их недостаточно, то тогда можно пометить параметр в глобальную переменную, у которой подпрограмма будет обращаться. Такой способ не эффективен, так как в этом случае нельзя реализовать рекурсию и иногда повторное обращение к программе.

Передача параметров через стек заключается в том, что перед обращением к подпрограмме, фактические параметры загружаются в стек, подпрограмма извлекает их и использует. Именно такой способ используется в языках высокого уровня.

Передача параметров в потоке кода заключается в том, что параметры располагаются сразу за командой `call` и подпрограмма, чтобы их использовать должна обратиться к адресу возврата, который записывается в стек автоматически командой `call`, но теперь, чтобы возвратиться из подпрограммы, перед выходом из подпрограммы необходимо изменить адрес возврата, на адрес команды, следующий за данными. Этот способ сложнее, чем передача через регистры, глобальную переменную или стек, но примерно так же как и в блоке параметров.

В блоке данных фактические параметры размещаются в некоторой области памяти, обычно это сегмент данных. Адрес начала этого блока передаётся процедуре любым способом: через регистры, глобальную переменную, стек, потоки кода или даже в другом блоке параметров, и такой способ часто используется в функциях операционной системы или `bios`, например, поиск данных DTA или загрузка и исполнение программы EPV.

При передаче параметров по значению подпрограмма передаётся значение фактического параметра, это значение копируется в подпрограмме и используется, следовательно, изменить это значение подпрограмма не может. Такой способ используется для передачи параметров небольшого размера.

Пример (слайд)

Все числа знаковые размером с слово, используем простейший способ передачи параметров через регистры, то есть загружает параметры в `Ax` и `Bx` и результат записываем в `Ax`.

процедура (слайд)

фрагмент вызывающей программы (слайд)

Передача параметров по адресу

Оформим как процедуру вычисления  $x = x \div 16$

Параметр 1, он входной и выходной и должен содержаться в какой-то ячейке памяти, а если мы хотя несколько раз обратиться к этой процедуре, то нужно иметь несколько адресов параметров. Можно использовать регистры (чаще всего `BX`, `BP`, `SI`, `DI`). Пусть мы передаём через регистр `BX`, запишем в начале фрагмент вызывающей программы

пример (слайд)

Процедура может выглядеть так (слайд)

Мы использовали команду сдвига вместо деления, которая выполнится значительно быстрее (сдвиг на 4 разряда вправо эквивалентен делению

на 16). Процедуру начали с сохранения CX, так как мы используем часть регистра в подпрограмме, а возможно, что CX используется в вызывающей подпрограмме, поэтому сначала сохранили его, а после вывели. Поскольку регистров мало + их использует и вызывающая программа и подпрограмма, то это значит, что при входе в процедуру, сохранить содержимое регистров в стеке, а перед выходом их восстановить. Но, конечно, если результат хранится в регистре его не надо сохранять и восстанавливать. Именно для этого и существуют pusha и popa.

Передача параметров по ссылке в блоке параметров

Массив - много параметров. Адрес начала массива передаём через регистр и посмотрим такой пример. Пусть для двух байтовых массивов X и Y, один на 100, другой на 50 элементов и нужно вычислить сумму максимумов и загрузить в регистр

Пример (слайд)

Напишем процедуру поиска максимального в одномерном массиве, если элементы массива нумеруются с нуля. (слайд)

Передача параметров через стек - это универсальный способ, можно использовать при любом количестве параметров, но способ сложнее чем через регистры.

Пусть у нас есть подпрограмма у которой есть k параметров  $PP(a_1, \dots, a_k)$  размером в слово и сохраняем их в стеке последовательно слева направо.

Обращение к процедуре (слайд)

Содержимое стека (слайд)

Чтобы обратиться к параметра  $a_1, \dots, a_k$  мы можем использовать регистр BP, но вначале ему нужно присвоить значение SP, но вполне вероятно BP используется в вызывающей программе, поэтому его нужно предварительно сохранить в процедуре.

То есть входные действия будут такими:

пример(слайд)

стек будет выглядеть так (слайд)

Чтобы обратиться к k-тому параметру можно использовать выражение (слайд)

Затем реализуем программы для вспомогательного алгоритма.

Выходные действия в процедуре должны заключаться в восстановлении регистра BP и очистка стека от параметров.

образец (слайд)

Команда `ret n` - это количество освобождаемых байтов в стеке Команда `ret` вначале считывает адрес возврата, а затем удаляет из стека n байтов.

Очистку стека от параметров можно выполнять в процедуре (подпрограмме), а можно выполнять после выхода из подпрограммы, после выхода из подпрограммы можно это сделать так

`add SP, 2*k`

Но если очистку делать в процедуре, то исходный текст основной программы будет меньше, а если параметров много, то для простого использования фактических параметров можно использовать при входе в проце-

дуру использовать директиву `eqi`, чтобы присвоить символические имена параметрам вместо выражений

пример (слайд)

Пример передачи параметров через стек

Пусть наша процедура (очищает) заполняет нулями массив, а вызывающая программа обращается к этой процедуре для очистки двух массивов `X` и `Y` по 100 и 50 элементов соответственно. Параметры будем передавать через стек, размер можно передавать по значению, а имя массива по адресу (по ссылке), потому что этот параметр входной и выходной параметр.

Процедура (слайд)

восстановление регистров и выходные действия (слайд)

Передача по значению пример (слайд)

передача по ссылке пример (слайд)

Передача параметров по возвращаемому значению объединяет передачу по ссылке и по значению: процедуре передаётся адрес переменной, она делает локальную копию этого параметра, работает с этой копией, а в конце записывает эту копию

(слайд)

Передача параметра по имени макроопределения (слайд)

Обращение к ПП может быть таким (слайд)

Передача параметров отложенным вычислением. В этом случае наша процедура получает адрес подпрограммы, вычисляющая значения фактических параметров для неё. Такой механизм используется чаще в ИИ и ОС.

Заголовок Использование локальных параметров.

## 8 Лекция 9

Заголовок локальные параметры подпрограмм

Если параметров немного, то их можно хранить в регистрах, если много, то возможны различные варианты, однако, если мы будем хранить их в сегменте данных, то большую часть времени они не будут использоваться, то есть, лучше хранить локальные параметры в стеке.

Например, если параметров локальных немного, то стек может выглядеть так (слайд)

Сразу после входных действий нужно выделить память под локальные параметры изменив значение регистра `SP`, тогда шаблон подпрограммы, в которой и локальные и фактические параметры передаются через стек и локальные содержатся в стеке, можно представить следующим образом (слайд 22)

Также нужно сохранить в стеке все регистры, которые будем использовать в подпрограмме.

Посчитаем количество различных символов в заданной строке. Строка - массив символов, адрес начала которой передадим через регистр `BX`, длину строки - `CX`, а результат будем хранить в регистре `AX`. Для этого создадим локальный массив `l` размером в 256 байтов и `k`-тому элементу этого

массива будем присваивать 1, если символ, цифровой код которого равен k, содержится в этой строке. Вначале массив обнулим, а в конце посчитаем количество единиц. К первому элементу массива можно обратиться по формуле  $l1 = [BP - 265]$ , а к lk  $lk = [BP - 256 + k]$

программа (слайд + слайд + слайд)

Заголовок рекурсия в ассемблере

Заикливание не произойдёт, если в программе есть 2 ветви: рекурсия и нерекурсивная и, при выполнении некоторого условия, вычислительный процесс идёт не по рекурсивной ветви.

Если параметры будут храниться в сегменте данных, то рекурсии не получится. Если есть такая необходимость, то при входе в программу их нужно записать в стек, а потом использовать, то есть в рекурсии параметры нужно передавать через стек.

Примеры: (слайд)

Заголовок работа со строками

Строка это последовательность байтов, слов, или двойных слов. Строка источник находится по адресу DS:SI или EDS:ESI, а строка приёмник по адресу ES:DI, или ES:EDI.

Каждая команда работает только с 1 элементом строки (байтом, словом или двойным словом. Это зависит от типа команды или от типа Операнда), чтобы выполнить действие над всей строкой нужно использовать специальные префиксы, которые действуют только на строковые команды.

Префиксы:

1) гер - повторять

2) гере - повторять пока равно

3) герз - повторять пока ноль

4) герне - повторять пока не равно

4) герпз - повторять пока не ноль

Префикс гер заставляет повторять выполнение строкой команды n раз, n должно содержаться в регистре CX, а если CX будет = 0, то команда не выполнится ни разу.

гер и герз выполняет указанную команду пока ZF = 1, но не более n раз. представить работу команды можно в виде (слайд)

герне герпз указанная строковая команда будет выполняться до тех пор пока ZF = 0, но не более n раз. Псевдокод (слайд)

гер работает с командами: movs, lods, stos, ins, outs

герне, герз, герне, герпз с командами: cmps, scas

Команды для копирования строк (слайд)

Если используется вариант movs, то ассемблер по типу операндов сам определяет сколько байтов нужно переслать (1, 2 или 4). В этой команде DS можно изменить на регистры ES,GS,FS,CS,SS, но ES изменять нельзя.

Для всех строковых команд, после выполнения команды, содержимое регистров SI DI автоматически изменяется в зависимости от флага DF. Содержимое индексных регистров увеличивается на 1 или 2, или 4 (зависит от вида команды), если DF = 0, иначе содержимое регистров уменьшается на то же значение.

команды сравнение строк (слайд)

1-ый вариант команды в зависимости от типа операндов ассемблер сравнивает содержимое байтов, слов или двойных слов (зависит от типа операндов).

Эти команды используются с префиксами гере, герз, герне, герпз

scas op1

виды команд (слайд)

При работе 1-ого варианта команды количество сравниваемых байтов зависит от разрядности операндов.

Команды scmpb и scasd устанавливают флаги аналогично команде scmpb.

Команды считывания строки из памяти и загрузки в аккумулятор.

lods op2

варианты (слайд)

Команды записи из регистра AL, AX или EAX в память в строку, расположенную по адресу ES:DI или ES:EDI.

lods op2 работает как lodsb, lodsw, lodsq, в зависимости от типа операнда и здесь DS можно заменить на ES, FS, GS SS.

stos op1

варианты (слайд)

При использовании этих команд с префиксом гер строка длиной в (слайд) команды считывания из порта ввода/вывода

ins op1, DX

Любой из этих команд можно считать из порта ввода вывода номер который содержится в регистре DX (байт, слово, двойное слова) и записать в память ES:DI или ES:EDI. Работа 1 зависит от типа операнда.

При использовании с повторителем гер можно прочитать блок данных и переслать их в некоторую область памяти.

Запись в порт содержимое ячейки памяти, размером в байт, слово или двойное слово, находящейся по адресу DS:SI или DS:ESI

outs DX,op2

(слайд)

с повторителем может вывести много данных

Номер порта должен быть всегда в DX

Команды управления флагами.

В зависимости от флага DF мы можем просматривать строку в сторону увеличения адресов или уменьшения адресов, но этот флаг должен устанавливать сам программист с помощью команд (слайд):

Кроме флага DF программист может изменить следующие флаги: (слайд + слайд)

Загрузка сегментных регистров (слайд)

Для всех команд op2 - переменная в ОП размеров в 32 или 48 бит в зависимости от разряда операндов. Первые 16 бит этой переменной загружаются в соответствующий сегмент DS, ES и т.д., а следующие 16 или 32 - в регистр общего назначения, указанный в качестве первого операнда.

Загрузка сегментных регистров пример (слайд)

пример использования команд работы со строками (слайд)

пример 2 нужно в строке S состоящей из 500 символов заменить первое вхождение "\*" на ".". (слайд)

## 9 Лекция 10

Заголовок строки переменной длины

Со строками в ассемблере можно работать как в c++ (недостаток, если неэффективно сравнивать строки большой длины), когда строка заканчивается признаком конца строки и при увеличении или уменьшении строки этот признак конца строки нужно переносить, а можно работать как в языке Паскаль. Вначале указывается длина строки, а затем символы строки, сколько места нужно отводить под длину строки завист от количества символов в строке. Если в строке может быть не более 255 символов, то на строку достаточно 1 байта, и тогда, если имя строки, например S, то по адресу S содержится длина строки S+i - i-ый символ строки.

Например, посмотрим пример программы, удалить из строки S первое вхождение символа \*. (слайд 41)

Заголовок представление и работа со списками

Односвязный линейный список можно представить следующим образом: (слайд)

В каждом элементе списка есть информационая часть (info) и связывающая часть (link), стандартных процедур для работы со списками в ассемблере нет. Динамические переменные располагаются в специальной области оперативной памяти, которую называют кучей (heap). Размер кучи зависит от количества динамических переменных, то есть от количества и длины списков в программе. Допустим, что для кучи достаточно 64кб и пусть начало кучи определяет регистр ES, если внутри динамической памяти элемент имеет адрес A (смещение относительно начала кучи), то полный физический адрес кучи определяется ES:A, но поскольку ES для всех переменных ES один и тот же, то адрес элемента - A, пусть информационное поле занимает 2 байта, тогда элемент списка можно предствать как следующую структуру (слайд)

Элемент можно описать как A node<>. Доступ к полям этой переменной можно записать

ES: A.elem

ES: A.next

Пустая ссылка. Определи константу NULL EQU 0 и будем пользоваться этим символическим именет так же как в c++. Ссылки на первые элементы списков обычно хранятся в статической памяти, например в DS, как переменные размером в слово (слайд)

Чтобы работать со списком, мы просматриваем элементы списка 1 за други и для этого нужно знать адрес текущего элемента списка. Используем для хранения этого адреса регистр BX, в котором будет храниться только смещение текущего элемента, смещение текущего элемента - адрес относительно начала списка. Для обращения:



ES:[BX]

Просто нельзя писать поскольку по умолчанию мы работаем с регистром BS, тогда обращение к полям текущего списка это есть

ES:[BX].elem и ES:[BX].next

- 1) Анализ информационного поля (слайд)
- 2) переход к следующему элементу списка (слайд)
- 3) Проверка на конец списка (слайд)
- 4) поиск элемента с заданным значением информационного поля: Если nsp - начало списка, x - искомая величина, в AL записываем 1, если элемент есть, иначе 0. Тогда (слайд)
- 5) Вставка нового элемента в список:

В ассемблере нет процедуры new, которая выделяет место для нового элемента в куче, её нужно создавать самим, но пусть для нас такая процедура есть и она выделяет в куче адрес байта, начиная с которого, можно разместить следующий элемент списка и этот адрес процедура new(её нужно написать самому) передаёт через регистр DI, тогда вставить элемент в начало списка. (слайд)

- 6) удаление элемента из списка:

Пусть для адреса 1-ого элемента списка используем BX, для 2-ого DI и предполагаем, что у нас уже есть процедура dispose DI, которая удаляет элемент с адресом DI, освобождая место в динамической памяти.

Реализация (слайд)

## 10 Лекция 11

При выполнении программы свободные и занятые ячейки динамической памяти располагаются не последовательно, а произвольно, потому что элементы списков в программе создаются произвольно и произвольным образом удаляются.

Чтобы создать элемент списка место выделяется в куче, а при удалении элемента из списка рабочей программы это место добавляется в динамическую память, поэтому для простоты работы с динамической памятью, её можно объединить в список. Адрес начала списка хранится в некоторой фиксированной ячейке, а список называют списком свободной памяти (ССП), и тогда работать с ним можно как с обычным списком.

Назовём указатель на начало СПП heapptr, работаем с ним по-обычному.

Указатель heapptr должен храниться в начале кучи, в ячейке с относительным адресом 0.

Описание сегмента кучи, в котором может разместиться n элементов размером в двойное слово: (слайд 49)

Адрес начала кучи должен храниться в ES и программист загружает его туда сам, а байты этого сегмента нужно объявить в список. heapptr указывает на начало списка, тогда: (слайд 49)

Инициализация кучи и загрузка её начала в регистр ES: (слайд 50)

Заголовок процедуры new и dispose

Процедура создания может выглядеть следующим образом:

(слайд 51)

Пример процедуры dispose: Процедуре dispose передаётся адрес удаляемого элемента в регистре DI, которую нужно присоединить к ССП.

(слайд 52)

Заголовок макросредства в языке ассемблер

Макросредства - самое сильное средство программирования (а может и нет).

К макросредствам относятся:

- 1) блоки повторений.
  - 2) макросы.
  - 3) директивы условной генерации.
- заголовок блоки повторений

Блоки повторений позволяют включать в исходный текст программы последовательности команд и количество включений такой последовательности зависит от заголовка блока. Кроме того, от заголовка блока зависит будет ли эта последовательность команд повторяться n раз в неизменном виде или в каком-то модифицированном.

Макросы больше похожи на подпрограммы потому что существует описание или определение макроса, его называют макроопределением. И существует обращение к макросу, которое называют макрокомандой.

Выполнение макрокоманды называется макроподстановкой, в результате которой в исходном тексте программы, на месте макрокоманды, появляется макрорасширение. Сколько мы огратимся к макросы, сколько будет макрокоманд в программе, столько макрорасширений появится в исходном тексте программы.

Макрос отличается от подпрограммы:

- 1) при обращении к подпрограмме мы передаём управление на участок памяти, где хранится определение подпрограммы, а при обращении к макросу мы получаем в исходном тексте тело макроса.
- 2) при использовании макросов мы экономим время на обращение к подпрограмме и возврат из неё, но проигрываем в памяти.
- 3) если повторяется небольшой фрагмент программы, то рекомендуют использовать макросы, ну а если вспомогательный алгоритм большой, то используем подпрограмму.

Программа, содержащая макросредства, транслируется и ассемблируется в несколько этапов, на 1 этапе программа преобразуется в числый код без макросредств (препроцессорная обработка или макрогенерация), а на втором этапе происходит преобразование в машинный код.

Нужно сказать, что как блоки повторений, так и определение макросов могут располагаться в самой программе, могут соежаться в файлах на диске и подключаться с помощью директивы include <имя файла>

Заголовок Блоки повторений

Общий вид блока повторений: (слайд)

тело - любое количество любых операторов, предложений, в том числе и блоков повторений.

endm определяет конец тела блока. Количество повторений телл и способы модификации тела блока зависят от заголовка.

Возможны следующие заголовки:

(слайд 55)

Оно может быть вычислено на этапе макрогенерации, в результате которого n копий тела блока записывается в данном месте программы.

Пример (слайд 55)

Можно создать массив, состоящий из ASCII кодов прописных русских букв: (слайд 56)

Пример (задержка работы процессора на время выполнения команд jmp)

(слайд 56)

Второй вид заголовка (слайд 57)

P - формальный параметр

Vk - фактические параметры

<> - не мета символы, а символы языка (они обязательны)

И при макрогенерации тело блока будет повторяться k раз, но не в неизменном виде, а так, чтобы в i-той копии формальный параметр был заменён фактическим параметром Vi.

P - символическое имя, если оно случайно совпадёт с служебным именем, то ничего страшного не произойдёт.

Список фактических параметров, если он пуст, то блок просто игнорируется.

Например: (слайд 57)

Например: (слайд 58)

Фактическим параметром может быть как объект программы, так и часть команды, часть предложения, главное, чтобы после подстановки получилась правильная команда ассемблера.

Например: (слайд 58)

Вид заголовка (3) (слайд 59)

P - формальный параметр

Si - символы, любые, кроме ; и пробела.

И при выполнении макрогенерации на месте этого блока будет k копий тела, так что в i-той копии формальный параметр P заменён на Si.

Чтобы пробел и ; были фактическими параметрами нужно заключить всю последовательность в угловые скобки.

Пример (слайд 59)

Заголовок Макрооператоры

Макрооператоры могут использоваться в блоках повторений и в макроопределениях, для записи формальных и фактических параметров.

Таких макрооператоров 5:

амперсант - указывает границу формального параметра, чтобы выделить этот параметр из общего текста, после макрогенерации этот символ пропадает.

Например(слайд 60)

Если несколько знаков амперсанта стоят рядом, то макрогенератор за 1 проход открывает 1 знак амперсанта.

Пример (слайд 61)

Макрооператор <> - угловые скобки, действует так: весь текст, заключённый в скобки действует как строка. Этот макрооператор используется часто для передачи параметров (слайд 61-62).

Макрооператор ! - действует так же как угловые скобки, но на 1 символ.

Макрооператор процент - говорит макрогенератору, что следующий за ним текст нужно вычислить как значение выражения и использовать как параметр.

Например (слайд 62)

Макрооператор ;; - определяет макрокомментарий. Текст макрокомментария не включается в макрорасширения и в листинг программы.

## 11    Лекция 12