ТАНЯ 1-20
МАРГО 21-40
кря тутуу 41 - 60
Настя 61-80

1. Архитектура МП 8088 и 80386. Образование физического адреса.
2. Характеристики регистров
3. Флаги.
4. Сегментные регистры по умолчанию.
5. Структура одномодульной программы MS DOS. Повторные описания сегментов.
6. Возможные структуры кодового сегмента.
7. Возможные способы начала выполнения и завершщения программы MS DOS.
8. Структура программы из нескольких исходных модулей MS DOS
9. Стандартные директивы описания сегментов.
10. Переменные, метки, символические имена и их атрибуты.
11. Виды предложений языка Ассемблер.
12. Директивы (псевдооператоры): назначение и формы записи.
13. Возможные комбинации сегментов и умолчания.
14. Директивы ASSUME, ORG, END.
15. Структура процедур.
16. Внешние имена.
17. Типы данных и задание начальных значений.
18. Способы описания меток, типы меток.
19. Команды условных переходов.
20. Команды организации циклов.
21. Способы адресации.
22. Организация рекурсивных подпрограмм.
23. Арифметические команды
24. Связывание подпрограмм.
25. Команды CALL и RET.
26. Способы передачи параметров подпрограмм.
27. Способы сохранения и восстановления состояния вызывающей программы.
28. Конвенции языков высокого уровня.
29. Команды сдвига.
30. Команды логических операций.
31. Команды обработки строк и префиксы повторения.
32. Команды пересылки строк.
33. Команды сравнения строк.
34. Команды сканирования строк.
35. Команды загрузки строк.
36. Команды сохранения строк.
37. Листинг программы.
38. Макросредства.
39. Макроопределеня (макрофункций и макропроцедур) и макрокоманды.
40. Директива INCLUDE и LOCAL.
41. Рекурсия в макроопределениях.
42. Параметры в макросах.
43. Директивы условного ассемблирования и связанные с ними конструкции.
44. Директивы IFB и IFNB в макроопределениях.
45. Директивы IFIDN и IFDIF в макроопределениях.
46. Операции ;; % & < > ! в макроопределениях.
47. Блоки повторения REPT, IRP/FOR, IRPC / FORC, WHILE.
48. Директива EQU и = в MASM.

49. Директива TEXTEQU в MASM32.
50. Типы макроданных text и number.
51. Именованные макроконстанты MASM32
52. Макроимена, числовые и текстовые макроконстанты.
53. Директивы echo и %echo
54. Способы вывода значений макропеременных и макроконстант с пояснениями
55. Операции в выражениях, вычисляемых препроцессором MASM:
56. Подготовка ассемблерных объектных модулей средствами командной строки для использования в средах разработки консольных приложений на ЯВУ.
57. Добавление ассемблерных модулей в проект консольного приложения C.
58. Добавление ассемблерных модулей в проект консольного приложения PASCAL.
59. Использование ассемблерных вставок в модулях на ЯВУ.
60. Вызов подпрограммы C из ассемблерной в VS C++.
61. Передача глобальных данных, определённых в консольной программе VS C++, в ассемблерный модуль.
62. Передача глобальных данных, определённых в ассемблерном модуле в консольнй модуль C.
63. Средства отладки в CodeView. Примеры.
64. Средства отладки в средах разработки консольных приложений на ЯВУ.
65. Способы адресации. Термины и смысл.
66. Связывание подпрограмм. Конвенции C, PASCAL, STDCALL, РЕГИСТРОВАЯ.
67. Многострочные макроопределения. Формальные и фактические параметры.
68. Многострочные макроопределения и директивы условного ассемблирования
69. Многострочные макроопределения и директивы генерации ошибок
70. Многострочные макроопределения и операции в них.
71. Макропроцедуры. Определения и вызовы
72. Рекурсивные макропроцедуры
73. Макрофункции. Определения и вызовы
74. Рекурсивные макроопределения
75. Числовые макроконстанты. Определение и примеры использования
76. Числовые макропеременные. Определение и примеры использования
77. Константные выражения. Операции, вычисляемые препроцессором в выражениях
78. Текстовые макропеременные. Способы определения
79. Блоки повторения REPT, FOR, FORC, WHILE.
80. Стандартные макрофункции обработки строк @CATSTR и @SUBSTR.
81. Инструменты отладки макросредств. Окно командной строки, листинг, сообщения препроцессора
82. Применение макросредств при определении переменных
83. Применение макросредств при создании фрагментов кода

1) Архитектура МП 8088 и 80386

8086 — 16 разрядный микропроцессор. Имеет 14 16 разрядных регистров, 16-разрядную шину данных, 20разрядную шину адреса (поддерживает адресацию до 1 Мб памяти), поддерживает 98 инструкций. 8088 16разрядный микропроцессор. Имеет 14 16-разрядных регистров. у него 8-разрядная шина данных.

80386 — расширение архитектуры 8086 до 32-разрядной. Появился защищенный режим, позволяющий 32-битную адресацию; все регистры, кроме сегментных, расширились до 32 бит (приставка E, например EAX); добавлены два дополнительных 16- разрядных сегментных регистра FS и GS, а также несколько специальных 32битных регистров (CRx, TRx, DRx). В защищенном режиме используется другая модель памяти.

Образование физического адреса.

Память представляет собой линейную последовательность байтов, поделенную на параграфы. Параграфы - последовательности из 16 идущих подряд байт, у первого из которых адрес кратен 16. Параграф является минимальным возможным в x86 сегментом. Полный физический адрес составляется из номера сегмента и смещения относительно начала этого сегмента (“байт 5 сегмента 3”).

<полный адрес> = <номер сегмента> * <размер сегмента> + <смещение>

2) Характеристики регистров.

Регистр процессора — сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора (указатель команды).

Регистры общего назначения	Аккумулятор	EAX AX AH AL
	База	EBX BX BH BL
	Счётчик	ECX CX CH CL
	Данные	EDX DX DH DL
Указательные регистры	Указатель базы	EBP BP
	Указатель стека	ESP SP
Указатель команды	Указатель команды	EIP IP
Индексные регистры	Индекс источника	ESI SI
	Индекс получателя	EDI DI
Регистр состояния	Флаги	EFLAGS 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 Резерв для INTEL VM RF 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 - NT IOPL OF DF IF TF SF ZF - AF - PF - CF
Сегментные регистры	Сегмент кода	CS (16 бит)
	Сегмент данных	DS (16 бит)
	Дополнительный сегмент	ES (16 бит)
	F сегмент	FS (16 бит)
	G сегмент	GS (16 бит)
	Сегмент стека	SS (16 бит)
Специальные регистры	Управляющий регистр машины	CR0
	Резервный регистр для INTEL	CR1
	Линейный адрес прерывания из-за отсутствия страницы	CR2
	Базовый регистр таблицы страниц	CR3
	Регистр сегмента состояния задачи	TSSR
	Регистр таблицы глоб. дескриптора	GDTR
	Регистр таблицы дескр. прерывания	IDTR
	Регистр таблицы лок. дескриптора	LDTR
	и т.д.	...

Всего архитектуры 8086 и 8088 содержат 14 16ти битных регистров При этом, например, AL нижние 8 бит регистра AX, AH верхние (это не отдельные

регистры!). В архитектуре 80386 все регистры, кроме сегментных, были расширены до 32 бит (приставка E), при этом, например, AX стал нижней половиной EAX и т.д., а так же были добавлены два новых 16 битных сегментных регистра FS и GS и несколько специальных регистров (CRx, DRx, TRx).

3) Флаги.

Каждый бит в регистре FLAGS является флагом. Флаг – это один или несколько битов памяти, которые могут принимать двоичные значения (или комбинации значений) и характеризуют состояние какого-либо объекта. Обычно флаг может принимать одно из двух логических значений. Поскольку в нашем случае речь идёт о бите, то каждый флаг в регистре может принимать либо значение 0, либо значение 1. Флаги устанавливаются в 1 при определённых условиях, или установка флага в 1 изменяет поведение процессора. На рис. 2.4 показано, какие флаги находятся в разрядах регистра FLAGS.

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

Carry Flag (CF) бит: 0 - **Флаг переноса**. Устанавливается в 1, если результат предыдущей операции не уместился в приёмнике и произошёл перенос из старшего бита или если требуется заём (при вычитании). Иначе установлен в 0. Например, этот флаг будет установлен при переполнении.

Parity Flag (PF) бит: 2 - **Флаг чётности**. Устанавливается в 1, если младший байт результата предыдущей команды содержит чётное количество битов, равных 1. Если количество единиц в младшем байте нечётное, то этот флаг равен 0.

Auxiliary Carry Flag AF 4 - **Вспомогательный флаг переноса** (или **флаг полупереноса**). Устанавливается в 1, если в результате предыдущей операции произошёл перенос (или заём) из третьего бита в четвёртый. Этот флаг используется автоматически командами двоично-десятичной коррекции

Zera Flag ZF 6 **Флаг нуля**. Устанавливается 1, если результат предыдущей команды равен 0

Sign Flag SF 7 **Флаг знака**. Этот флаг всегда равен старшему биту результата
Trap Flag TF 8 **Флаг трассировки** (или **флаг ловушки**). Он был предусмотрен для работы отладчиков в пошаговом выполнении,

которые не используют защищённый режим. Если этот флаг установить в 1, то после выполнения каждой программной команды управление временно передаётся отладчику (вызывается прерывание 1)

Interrupt Enable Flag IF 9 **Флаг разрешения прерываний**. Если сбросить этот флаг в 0, то процессор перестанет обрабатывать прерывания от внешних устройств. Обычно его сбрасывают на короткое время для выполнения критических участков программы

Direction Flag DF 10 **Флаг направления**. Контролирует поведение команд обработки строк. Если установлен в 1, то строки обрабатываются в сторону уменьшения адресов, если сброшен в 0, то наоборот

Overflow Flag OF 11 **Флаг переполнения**. Устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное. И наоборот

```
  01101110
+
  01101110
=
  11011100
```

Или в десятичной системе 110 + 110 = 220. Но мы то работаем с числами со знаком. В этом случае максимальное значение для байта равно 127. А полученное нами двоичное число будет на самом деле отрицательным числом –36. В случае переполнения устанавливается флаг OF в регистре флагов

I/O Privilege Level IOPL 12 13 Уровень приоритета ввода/вывода

Nested Task NT 14 Фраг вложенности задач

Зарезервированные биты 1, 3, 5, 15

4) Сегментные регистры по умолчанию.

В качестве сегментных регистров используются не любые регистры, а сегментные регистры CS, DS, SS, ES.

С учетом таких соглашений, позволяющих явно не указывать сегментные регистры, а подразумевать по умолчанию, машинные программы обычно строятся так: все команды программы размещаются в од-ном сегменте памяти, начало которого заносится в регистр CS; все данные размещаются в другом сегменте, начало которого заносится в ре-гистр DS; если нужен стек, то под него отводится третий сегмент памя-ти, начало которого записывается в регистр SS. После этого практически во всех командах можно указывать не полные адресные пары, а лишь сме-щения, т.к. сегментные регистры в этих парах будут восстанавливаться автоматически. Соглашение о сегментных регистрах позволяют уменьшить размер программы.

Сегментные регистры (CS, DS, ES, SS) нужны для хранения номера сегмента. Они используются вместе с адресными регистрами (SI, DI, IP, SP), которые указывают на смещение относительно начала сегмента, для получения полного адреса.

CS (регистр сегмента кода) используется с IP для адресации инструкций программы (IP указывает на следующую к выполнению команду), при этом по умолчанию при выполнении команд передачи управления (например JMP) их операнд воспринимается как смещение относительно текущего сегмента (ближний адрес). Можно передавать управление и в другой сегмент (дальний адрес: номер сегмента и смещение).

DS (регистр сегмента данных) используется по умолчанию для адресации данных (например MOV A, AX преобразуется в MOV DS:A, AX).

ES (регистр доп. сегмента) используется для тех же целей, если требуется доп. сегмент данных или связь приемник-передатчик (DS:SI → ES:DI, например). Требуется некоторыми командами (например MOVS).

SS (регистр сегмента стека) вместе с SP (указателем стека) используется для организации стека. SP при запуске программы содержит размер стека, если он не равен 0 или 64, и 0 в противном случае.

SS и CS устанавливаются автоматически при запуске программы, DS и ES надо устанавливать вручную

5) Структура программы одномодульной MS DOS. Повторные описания сегментов.

Пример одномодульной программы:

```
SEG_STACK SEGMENT PARA STACK ‘STACK’
          DB 64 DUP(‘STACK-’)
SEG_STACK ENDS

SEG_DATA SEGMENT PARA ‘DATA’
          S DW 42
SEG_DATA ENDS

SEG_CODE SEGMENT PARA ‘CODE’
         ASSUME CS:SEG_CODE, DS:SEG_DATA, SS:SEG_STACK

PROGSTART:
        MOV AX, SEG_DATA
        MOV DS, AX
        ; ... DO Smth
        MOV AH, 4Ch
        INT 21h
SEG_CODE ENDS
END  PROGSTART
```

Повторные описания сегментов обычно используются, когда один и тот же сегмент описывается в нескольких модулях, или чтобы расположить данные рядом с операциями над ними. При этом в случае разных модулей указываются видимые с наружи(в других модулях) элементы (PUBLIC name) , а в модулях , где эти элементы надо увидеть, в том же сегменте указываются их имена и тип (EXTRN name: type)

PUBLIC ИМЯ[, ИМЯ…] – объявление в любом месте данного модуля описанных в нём имён, которые могут использоваться в других модулях.

EXTRN ИМЯ:ТИП[, ИМЯ:ТИП…] – объявление для использования в данном модуле имён, описанных в других модулях, и их типов

ПРАВИРО: директиву EXTRN объявления внешних данных следует размещать внутри описания того же сегмента (если он есть в данном модуле), в котором они были описаны в другом модуле, иначе – вне описаний сегментов

```
; МОДУЛЬ 1
SEG_DATA SEGMENT PARA PUBLIC ‘DATA’
       PUBLIC ASS
       EXTRN S: BYTE
      A DW 666
SEG_DATA ENDS

; МОДУЛЬ 2
SEG_DATA SEGMENT PARA PUBLIC ‘DATA’
       PUBLIC S
```

```
      EXTRN A: WORD
     S DB 42
SEG_DATA ENDS
```

6) Возможные структуры кодового сегмента.

```
ИМЯ_СЕГМЕНТА_КОДА SEGMENT [<вид_выравнивания>] [‘CODE’]
   ASSUME CS:ИМЯ_СЕГМЕНТА_КОДА, DS:ИМЯ_СЕГМЕНТА_ДАННЫХ, SS:ИМЯ_СЕГ_СТЕКА

   ПРОЦ1 PROC [FAR|NEAR]
          ...
          CALL ПРОЦ2
          ...
          CALL ПРОЦ3
          ...
          RET|RETF
   ПРОЦ1 ENDP

   ПРОЦ2 PROC
         ...
         RET
   ПРОЦ2 ENDP

   ПРОЦ3 PROC FAR
         ...
         RET    ; ЭТОТ RET АВТОМАТИЧЕСКИ ЗАМЕНИТСЯ НА RETF
   ПРОЦ3 ENDP

   ...

МЕТКА_ТОЧКИ_ВХОДА:
     ; ...
ИМЯ_СЕГМЕНТА_КОДА ENDS
END МЕТКА_ТОЧКИ_ВХОДА
```

С сегментацией связаны понятия ближнего и дальнего адреса.

Естественно, что дальние обращения выполняются медленнее.

- **SHORT**(короткий переход) - процедура достижима из диапазона от (-128) до (+127) байтов от текущей точки программы. В этом случае для представления дистанции в команде перехода достаточно одного байта.
- **NEAR**(ближний переход) - процедура достижима из диапазона от (32768) до (+32767) байтов. В этом случае для представления дистанции необходимы два байта, и управление может передаваться в пределах всего сегмента. При таком переходе

модифицируется только указатель команд IP. По умолчанию любая процедура имеет тип *NEAR*.

- **FAR**(дальний переход) - процедура достижима из другого сегмента программного кода. В этом случае в команде перехода адрес точки назначения задается ее полным логическим адресом в формате segment:offset, а не дистанцией. При таком переходе модифицируются регистры CS и IP.

FAR — подпрограмма дальнего вызова (можно вызывать из других сегментов),
NEAR — ближнего (вызов только из этого сегмента).
RET — возврат из подпрограммы ближнего,
RETF— дальнего вызова. В отличие от обычного RET'а, который вытаскивает из стека только значение IP, RETF вытаскивает из стека значения CS и IP.

По умолчанию стоит NEAR. <вид_выравнивания> может быть PARA, BYTE, WORD, PAGE. Указывает, чему должны быть кратны адрес начала и конца сегмента. По умолчанию PARA. Еще вместо метки точки входа можно использовать FAR-процедуру. Тогда можно заканчивать программу точно так же по INT 21H / AX: 4Ch, а можно просто делать в конце RET, перед этим запушив в стек 0, а в начале программы перед выставлением регистра DS запушить его старое значение. Остальное не отличается.

7) Возможные способы начала выполнения и завершения программы типа .EXE

Способы входа:
END <метка в кодовом сегменте>,
END <процедура (far)>
Ставится в коде программы после сегмента кода (только ОДИН РАЗ ВО ВСЕЙ ПРОГРАММЕ). См. выше. Способы выхода: при помощи INT 21h / AX = 4C00h и при помощи RET'а, если программа началась со входа в подпрограмму. При этом если выход с помощью RET, то на верхушке стека должен лежать 0, а за ним DS, положенный туда в начале программы, а AX должен быть 0. Почему: при старте программы DOS кладет в DS адрес своей системной структуры, называемой Program Segment Prefix (PSP). Эта структура содержит аргументы командной строки, адрес возврата в DOS и прочую информацию. Первые два байта ее ВСЕГДА равны машинному коду команды INT 20h. Поэтому, когда мы кладем в стек DS, а затем 0 прямо в самом начале программы, по RET(F) в CS положится адрес PSP, а в IP - 0. А нулевая инструкция в этом нашем новом CS и есть INT 20h, которая является одним из прерываний, завершающих программу в DOS, и она будет выполнена процессором следующей после RET. Это прерывание берет из AX код ошибки (0 = нет ошибки).
Пример:

```
; МОДУЛЬ 1 ГЛАВНЫЙ
StkSeg  SEGMENT PARA STACK 'STACK'
```

```
        DB      200h DUP (?)
StkSeg  ENDS
;
DataS   SEGMENT WORD 'DATA'
HelloMessage    DB   13                ;курсор поместить в нач.
строки
                DB   10                ;перевести курсор на нов.
строку
                DB   'Hello, world !'  ;текст сообщения
                DB   '$'               ;ограничитель для функции DOS
DataS   ENDS
;
Code    SEGMENT WORD 'CODE'
        ASSUME  CS:Code, DS:DataS
DispMsg:
        mov   AX,DataS                 ;загрузка в AX адреса
сегмента данных
        mov   DS,AX                    ;установка DS
        mov   DX,OFFSET HelloMessage   ;DS:DX - адрес строки
        mov   AH,9                     ;AH=09h выдать на дисплей
строку
        int   21h                      ;вызов  функции DOS
        mov   AH,7                     ;AH=07h ввести символ без эха
        INT   21h                      ;вызов  функции DOS
        mov   AH,4Ch                   ;AH=4Ch завершить процесс
        int   21h                      ;вызов  функции DOS
Code    ENDS
        END   DispMsg
```

8) Структура программы из нескольких исходных модулей MS DOS.

```
; МОДУЛЬ 1 ГЛАВНЫЙ
PUBLIC SHIT1

DSEG SEGMENT PARA PUBLIC ‘DATA’
     ASS DB 22
DSEG ENDS

SSEG SEGMENT PARA STACK ‘STACK’
     DB 64 DUP (‘ASS--’)
SSEG ENDS

CSEG SEGMENT PARA PUBLIC ‘CODE’
     ASSUME CS:CSEG, DS:DSEG, SS:SSEG
     EXTRN SHIT2:NEAR

     SHIT1 PROC FAR
           CALL SHIT2
           MOV AH, 4Ch
           INT 21h
     SHIT1 ENDP

CSEG ENDS
END SHIT1 ;Это определяет точку входа
          ; во всю программу

; МОДУЛЬ 2
DSEG2 SEGMENT PARA PUBLIC ‘DATA’
      FUCK DB “HELLO AVGN!$”
DSEG2 ENDS
EXTRN ASS:BYTE

CSEG SEGMENT PARA PUBLIC ‘CODE’
     ASSUME CS:CSEG, DS:DSEG2
     PUBLIC SHIT2

     SHIT2 PROC NEAR
          MOV AX, SEG ASS
          MOV ES, AX
          MOV AH, 09h
          MOV DX, FUCK
          INT 21h
          INC ES:ASS
          RET
     SHIT2 ENDP
CSEG ENDS
END    ;Здесь нихрена нет,поэтому модуль
       ; неглавный
```

Сначала создаются объектники отдельных модулей с помощью компилятора (masm.exe), а затем всёкомпонуется в экзешник линковщиком (link.exe).

ПРАВИРО: директиву EXTRN объявления внешних данных следует размещать внутри описания того же сегмента (если он есть в данном модуле), в котором они были описаны в другом модуле, иначе – вне описаний сегментов.

9) Стандартные директивы описания сегментов

Стандартно сегменты на языке Assembler описываются с помощью директивы SEGMENT. Синтаксическое описание сегмента представляет собой следующую конструкцию

<имя сегмента> SEGMENT [выравнивание] [тип комбинирования][класс]
<тело сегмента>
<имя сегмента> ENDS

```
StkSeg  SEGMENT PARA STACK 'STACK'
        DB      200h DUP (?)
StkSeg  ENDS

DataS   SEGMENT WORD 'DATA'
DataS   ENDS

Code SEGMENT WORD 'CODE'
        ASSUME  CS:Code, DS:DataS
Code    ENDS
```

<ВЫРАВНИВАНИЕ>:

- BYTE без - 1 байт
- WORD по границе слова - 2 байта
- PARA по границе параграфа - 16 байт (по умолчанию)
- PAGE по 256 байт (странице)

<ТИП КОМБИНИРОВАНИЯ>:

- **PUBLIC** – конкатенация частей сегментов с одним именем и классом при компоновке с учётом выравнивания (склеиваются) (регистр DS следует загружать самим, а регистр будет загружаться автоматически)
- **STACK** сегмент является частью стека (то же, что и PUBLIC, но для сегмента стека (регистр SS будет загружен автоматически))
- **COMMON** расположение на одном адресе с другими COMMON-сегментами (накладывается) (сегменты разных модулей, имеющие одно имя и класс, будут начинаться с одного адреса памяти, а объём выделенной для них памяти будет равен длине самого длинного из этих сегментов)

- **AT** nnnnh – началом сегмента будем номер параграфа, указанного после слова AT
- **НИЧЕГО(PRIVATE)**(по умолчанию) - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля. (сегмент сам по себе, те не участвует в каких-либо комбинациях с сегментами, описанными в других модулях, даже если имеет с ними одно и то же имя и класс. В файле распределения памяти .MAP, который строится компоновщиком, он будет представлен отдельной строкой)

```
PUBLIC output_X
EXTRN X: byte

DS2 SEGMENT AT 0b800h
     CA LABEL byte
     ORG 80 * 2 * 2 + 2 * 2
     SYMB LABEL word
DS2 ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
     assume CS:CSEG, ES:DS2
output_X proc near
     mov ax, DS2
     mov es, ax
     mov ah, 10
     mov al, X
     mov symb, ax
     ret
output_X endp
CSEG ENDS
```

<КЛАСС>:
идентификатор (‘STACK’, ‘CODE’, ‘DATA’, . . .)
НИЧЕГО(по умолчанию == отсутствие параметра – компоновщик группирует сегменты разных модулей по именам их классов. Отсутствие имени класса == класс с пустым именем.

С помощью директивы ASSUME можно сообщить транслятору, какой сегмент, к какому сегментному регистру «привязан», или, говоря более точно, в каком сегментном регистре хранится адрес сегмента..
Формат директивы ASSUME:

ASSUME <сегментный регистр>:<имя сегмента>

Директивы **SEGMENT и ASSUME - стандартные директивы сегментации**.

Для простых программ, которые содержат по одному сегменту кода, данных и стека легче использовать упрощенные описания соответствующих сегментов.

Трансляторы MASM и TASM предоставляют возможность использования упрощенных директив сегментации (вместо SEGMENT).

Замечание. Стандартные и упрощенные директивы сегментации не исключают друг друга.

Совет 1. Стандартные директивы используются, когда программист хочет получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.
Совет 2. Упрощенные директивы целесообразно использовать
1) для простых программ
2) программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня (это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления).

Упрощенные директивы определения сегмента
(для режима MASM)

.CODE [имя] Определение начала или продолжения сегмента кода. Возможно определение нескольких сегментов данного типа.
.DATA определение начала или продолжения сегмента **инициализированных** данных. Также используется для определения данных типа near.
.STACK [размер] определение начала или продолжения сегмента стека. Параметр [размер] задает размер стека. По дефолту — 1024б
.CONST определение начала или продолжения сегмента констант.
.DATA? определение начала или продолжения сегмента **неинициализированных** данных. Также для определения данных типа near.
.FARDATA [имя] определение начала или продолжения сегмента **инициализированных** данных типа far. Возможно определение нескольких сегментов данного типа.
.FARDATA? [имя]
Директива предназначена для определения начала или продолжения сегмента **неинициализированных** данных типа far. Возможно определение нескольких сегментов данного типа.

Директива MODEL позволяет вам задать для программы несколько стандартных моделей сегментации. Вы можете также использовать ее для задания языка для процедур программы.

Директива MODEL имеет следующий синтаксис:
MODEL <модель_адр.> [<модиф_адр.>] [,<язык>] [,<модиф.языка>]:
<модель_адр> : модель адресации (FLAT 32битная стандартная)
<модиф_адр>:тип адресации: use16|use32|dos
<язык>, <модиф.языка>: определяют особенности соглашений о вызовах

10) Переменные, метки, символические имена и их атрибуты.

Конструкции ассемблера формируются из идентификаторов и ограничителей.
Идентификатор — набор символов (буквы, цифры, “_”, “.”, “?”, “$”, “@”).

- Символ “.” только в начале идентификатора.
- Идентификатор не может начинаться с цифры,
- может размещаться только на одной строке и содержит только от 1 до 31 символа.

```
Alpha           ; Допустимые имена
String_Count
@DELAY
1Array          ; Недопустимое имя
```

С помощью идентификаторов можно представить переменные, метки и имена.
Переменные идентифицируют хранящиеся в памяти данные и имеют три **атрибута**:

- **Сегмент** базовый адрес логического сегмента, в котором описана переменная, и может принимать значение из диапазона 0...FFFFh. Значение можно получить с помощью директивы SEG.
- **Смещение** расстояние в байтах от начала логического сегмента до переменной и может иметь значение в диапазоне 0...FFFFh. Значение можно получить с помощью директивы OFFSET.
- **Тип** характеризует длину переменной в байтах (DB(1 байт), DW(2), DD(4), DF/DP(6), DQ(8), DT(10))
-

Если сегмент Data, начинающийся с физического адреса 05000h, имеет вид:

```
Data        SEGMENT
        Alpha           DB    ?
        Beta            DD    ?
        Gamma           DW    100 DUP (?)
Data        ENDS,
```

то рассмотренные директивы вернут следующие значения:

SEG Alpha = *SEG* Beta = *SEG* Gamma = 500h;
OFFSET Alpha = 0, *OFFSET* Beta = 1, *OFFSET* Gamma = 5;
TYPE Alpha = 1, *TYPE* Beta = 4, *TYPE* Gamma = 2;

***LENGTH* Gamma = 100;**
***SIZE* Gamma = 200.**

Метка — частный случай переменной. это определенное программистом имя, используемое для пометки данной команды. Значением метки является адрес помеченной команды. Метки всегда заканчиваются двоеточием и используются в качестве операндов в командах передачи управления. (На неё можно ссылаться посредством переходов и вызовов.)
Имеет два атрибута:

- **Сегмент** (-||-)
- **Смещение** (-||-)
- **Дистанция** - характеризует расстояние в байтах, в пределах которого метка достижима для команды передачи управления. Этот атрибут аналогичен атрибуту "Тип" переменной.
  Значение - директива TYPE, для меток типа *SHORT* и *NEAR*
  Возвращается значение (-1), а для меток типа *FAR*(-2).
  - **SHORT**(короткий переход) - метка достижима из диапазона от (-128) до (+127) байтов от текущей точки программы. В этом случае для представления дистанции в команде перехода достаточно одного байта.
  - **NEAR**(ближний переход) - метка достижима из диапазона от (32768) до (+32767) байтов. В этом случае для представления дистанции необходимы два байта, и управление может передаваться в пределах всего сегмента. При таком переходе модифицируется только указатель команд IP. По умолчанию любая метка имеет тип *NEAR*.
  - **FAR**(дальний переход) - метка достижима из другого сегмента программного кода. В этом случае в команде перехода адрес точки назначения задается ее полным логическим адресом в формате segment:offset, а не дистанцией. При таком переходе модифицируются регистры CS и IP.

RET — возврат из подпрограммы ближнего,
RETF— дальнего вызова. В отличие от обычного RET'а, который вытаскивает из стека только значение IP, RETF вытаскивает из стека значения CS и IP.

По умолчанию стоит NEAR. Еще вместо метки точки входа можно использовать FAR-процедуру. Тогда можно заканчивать программу точно так же по INT 21H / AX: 4Ch, а можно просто делать в конце RET, перед этим запушив в стек 0, а в начале программы перед выставлением регистра DS запушить его старое значение. Остальное не отличается.

Добавлено примечание ([1]): Есть ли это вообще у меток????

Добавлено примечание ([2R1]): а ты это пробовала?

```
.386
.model flat, c
```

```
public StrLength
.code

StrLength:    ;метка
      ; doing smth
      ret
end
```

Символические имена — используются для идентификации данных и меток в ассемблерных программах. Символы, определённые директивой EQU, имеющие значение типа “символ” или “число”. (константы короче)?

```
DB ?              ; Просто выделена память в сегменте под 1 байт
DB 64 DUB(B,C,D)  ; 64 байта, заполненные поочерёдно B,C и D
A DB 123          ; Переменная A
B EQU 456         ; Константа B
C = 902           ; Константа C, которую можно переобъявлять
```

Директивы *EQU* и = аналогичны по назначению, но имеют следующие различия:

1) директива *EQU* может использоваться для присвоения имени как числового, так и символьного значения;

2) имена, определенные директивой =, можно переопределять, а директивой *EQU* - нельзя.

11) Виды предложений языка Ассемблер.

Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Предложения бывают четырех типов:

- **Команды** (и их операнды) — символические аналоги машинных инструкций (мнемоники). В процессе трансляции они преобразуются в машинный код ассемблером (например MOV AX, 0)
- **Макрокоманды** - оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями.
- **Директивы** — указания ассемблеру на выполнение определенных действий; не имеют аналога в машинном коде (например A EQU 9, SEG).

- **Комментарии** — строка или часть строки из любых символов, начинающаяся с символа ‘;’. Комментарии игнорируются транслятором.

Для того чтобы транслятор ассемблера мог распознать предложение, они должны формироваться по определенным синтаксическим правилам.

Предложения языка ассемблера состоят из следующих компонент:

- ***имя метки*** — идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает. это точка программы, на которую передается управление;
- ***имя*** — идентификатор, отличающий данную директиву от других одноименных директив. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики;
- ***код операции и директива*****(мнемоника)**— это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора;
- ***операнды*** — части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Метка и имя отличаются синтаксически. В masm после метки ставится : а после имени нет. Физический смысл одинаков – это адрес памяти. Во всех случаях, когда ассемблер встречает в программе имя или метку, он заменяет её на адрес той ячейки памяти, которую метка или имя именует.

Имя – это имя переменной программы, то есть ячейки памяти. Следовательно, имя стоит в предложении, содержащем псевдокоманду.

Операнды – операндами команд являются регистры (имена), непосредственные операнды, адреса памяти, задаваемые в виде констант, литералов, символических имен или сложных выражений, включающих в себя специальный синтаксис.

Добавлено примечание ([3]): Что оставить? То, что выше или это?

Добавлено примечание ([4R3]): и то и то

Добавлено примечание ([5R3]): ток примеры еще давай я примеров накидаю

12) Директивы (псевдооператоры): назначение и формы записи

Директивы — указания ассемблеру на выполнение определенных действий; не имеют аналога в машинном коде (чем отличаются от команд, которые на этапе трансляции преобразуются в машинный код).

Директивы определения набора инструкций: например, строка .386 в начале файла с кодом говорит ассемблеру использовать только наборы команд из архитектуры 80386 и ниже

Упрощенные директивы определения сегмента (для режима MASM)
.CODE [имя] Определение начала или продолжения сегмента кода. Возможно определение нескольких сегментов данного типа.
.DATA определение начала или продолжения сегмента инициализированных данных. Также используется для определения данных типа near.
.STACK [размер] определение начала или продолжения сегмента стека. Параметр [размер] задает размер стека. По дефолту — 1024б
.CONST определение начала или продолжения сегмента констант.
.DATA? определение начала или продолжения сегмента неинициализированных данных. Также для определения данных типа near.
.FARDATA [имя] определение начала или продолжения сегмента инициализированных данных типа far. Возможно определение нескольких сегментов данного типа.
.FARDATA? [имя]
Директива предназначена для определения начала или продолжения сегмента неинициализированных данных типа far. Возможно определение нескольких сегментов данного типа.

Директива MODEL позволяет вам задать для программы несколько стандартных моделей сегментации. Вы можете также использовать ее для задания языка для процедур программы.
Директива MODEL имеет следующий синтаксис:
MODEL <модель_адр.> [<модиф_адр.>] [,<язык>] [,<модиф.языка>]:
<модель_адр> : модель адресации (FLAT 32битная стандартная)
<модиф_адр>:тип адресации: use16|use32|dos
<язык>, <модиф.языка>: определяют особенности соглашений о вызовах
.model flat, c

Также **директивы объявления данных** DB, DW, DD, ...

```
DataByte   DB    10, 4, ?, 10h        ; Описание переменных
DataWord   DW    100, 100h, -5, ?     ; с инициализацией на
DataDWord  DD    3*20, 12345678h      ; числовые значения
```

Псевдооператоры:
<ИМЯ_ИДЕНТИФИКАТОРА> EQU <ВЫРАЖЕНИЕ>: используется для задания констант. Например, ASS EQU 10 вызовет последующую замену каждого слова ASS в коде на 10 транслятором. Замена “тупая”, то есть во всем тексте программы каждое вхождение того что слева будет заменено на то что справа.

Выражения могут быть и текстовые, и числовые, и даже куски текста программы.
<ИМЯ_ИДЕНТИФИКАТОРА> = <ЧИСЛОВОЕ_ВЫРАЖЕНИЕ>: то же, что и EQU, но константы, заданные =, можно переопределять позднее в коде и работает он только для числовых выражений. При этом числовое выражение может включать что угодно, что может в числовом виде вычислить компилятор, например: 3+2 , ASS+4 (если ASS — заданная выше числовая константа).

13) Возможные комбинации сегментов и умолчания.
Если комбинация не указана, то по умолчанию сегменты считаются разными, даже если у них одно имя и класс. Комбинировать объявления сегментов (сегментные директивы) можно с помощью указания ТИПА сегмента.

<ИМЯ_СЕГМЕНТА> SEGMENT [<ВЫРАВНИВАНИЕ>] [<ТИП>] [<КЛАСС>]
. . .
<ИМЯ_СЕГМЕНТА> ENDS

<ВЫРАВНИВАНИЕ>:
- BYTE без - 1 байт
- WORD по границе слова - 2 байта
- PARA по границе параграфа - 16 байт (по умолчанию)
- PAGE по 256 байт (странице)

<ТИП КОМБИНИРОВАНИЯ>:
- **PUBLIC** – конкатенация частей сегментов с одним именем и классом при компоновке с учётом выравнивания (склеиваются) (регистр DS следует загружать самим, а регистр будет загружаться автоматически)
- **STACK** сегмент является частью стека (то же, что и PUBLIC, но для сегмента стека (регистр SS будет загружен автоматически))
- **COMMON** расположение на одном адресе с другими COMMON-сегментами (накладывается) (сегменты разных модулей, имеющие одно имя и класс, будут начинаться с одного адреса памяти, а объём выделенной для них памяти будет равен длине самого длинного из этих сегментов)
- **AT** nnnnh – началом сегмента будем номер параграфа, указанного после слова AT
- **НИЧЕГО(PRIVATE)**(по умолчанию) - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля. (сегмент сам по себе, те не участвует в каких-либо комбинациях с сегментами, описанными в других модулях, даже если имеет с ними одно и то же имя и класс. В файле распределения памяти .MAP, который строится компоновщиком, он будет представлен отдельной строкой)

<КЛАСС>:

идентификатор (‘STACK’, ‘CODE’, ‘DATA’, . . .)
НИЧЕГО(по умолчанию == отсутствие параметра – компоновщик группирует сегменты разных модулей по именам их классов. Отсутствие имени класса == класс с пустым именем.

Пример:

```
; МОДУЛЬ 1 ГЛАВНЫЙ

STK SEGMENT para STACK 'STACK'
     db 100 dup(0)
STK ENDS

SD1 SEGMENT para common 'DATA'
     W dw 3444h
SD1 ENDS
END
; МОДУЛЬ 2

SD1 SEGMENT para common 'DATA'
     C1 LABEL byte
     ORG 1h
     C2 LABEL byte
SD1 ENDS

CSEG SEGMENT para 'CODE'
     ASSUME CS:CSEG, DS:SD1
main:
     mov ax, SD1
     mov ds, ax
     mov ah, 2
     mov dl, C1
     int 21h
     mov dl, C2
     int 21h
     mov ax, 4c00h
     int 21h
CSEG ENDS
END main
```

14) Директивы ASSUME, ORG, END.

ASSUME <СЕГ_РЕГ1>:<ИМЯ_СЕГ1>[, <СЕГ_РЕГ2>:<ИМЯ_СЕГ2>[, . . .]]
Пример: ASSUME DS:DATA_SEG, ES:EXTRA_SEG

Директива специализации сегментов. Обычно пишется в сегменте кода второй строчкой. Пример сообщает ассемблеру, что для сегментирования адресов из сегмента DATA_SEG выбирается регистр DS, из EXTRA_SEG — регистр ES. Теперь префиксы DS: , ES: при использовании переменных из этих сегментов можно опускать, и ассемблер их будет ставить самостоятельно. (как using namespace std)

Директива END

```
CSEG SEGMENT PARA PUBLIC ‘CODE’
       ; ... assume smth
     START:
       ; ... do smth
CSEG ENDS
END START
```

Здесь директива **END** у казывает на то, что точка входа в программу находится на метке START, а так же на окончание файла с исходным кодом. Строки после нее игнорируются. Если меток в коде нет, можно после END ничего не указывать. Если опустить **END**, MASM выдаст ошибку.

Директива **ORG**
Вы можете использовать директиву ORG, чтобы установить счетчик адреса в значение текущего сегмента. Директива ORG имеет следующий синтаксис:

ORG выражение ;задание текущей позиции в сегменте

где "выражение" не может содержать никаких имен идентификаторов с опережающими ссылками. Оно может быть константой или смещением от идентификатора в текущем сегменте, либо смещением относительно текущего счетчика адреса.

Необходимо отметить, что при использовании этой директивы возможна ситуация, когда программист приказывает транслятору разместить новый код программы по уже написанному месту, поэтому использование этой директивы допустимо только в крайних случаях. Обычно это использование векторов прерываний.

На понимание ORG: Допустим первые 100H байт перед запущенным кодом резервируется под разного рода управляющие структуры. Потому вся адресация внутри кода должна начинаться с смещением 100H байт. Директива org 100H как раз и говорит компилятору что всю адресацию внутри кода нужно сместить именно на эти 100H байт.
т.е. например у тебя в коде есть область где ты собираешься что то хранить которую ты объявляешь например

```
data1 DB 30h
```
которую внутри кода ты адресуешь как
```
mov ax, OFFSET Data1
```
при компиляции структура OFFSET Data1 преобразовывается в конкретный адрес к которому прибавиться значение из org.

```
PUBLIC output_X
EXTRN X: byte

DS2 SEGMENT AT 0b800h
      CA LABEL byte
      ORG 80 * 2 * 2 + 2 * 2
      SYMB LABEL word
DS2 ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
      assume CS:CSEG, ES:DS2
output_X proc near
      mov ax, DS2
      mov es, ax
      mov ah, 10
      mov al, X
      mov symb, ax
      ret
output_X endp
CSEG ENDS
END
```

Задача:

Составить программу из двух модулей. Модуль lr05-1-1.asm содержит описание стека, сегмента данных с переменной X db 'R' и сегмент кода с точкой входа. lr05-1-1.asm управление передаётся в lr05-1-2.asm (в программе должен быть один сегмент кода), где выполняется вывод в видеопамять значения из X в 3-ю строку и 2-й столбец и выход из ПОДпрограммы по RET. Видеопамять должен представлять сегмент DS2 второго модуля, используя директивы ORG и LABEL поместите в его начало переменную CA типа BYTE со значением '0', а во второе знакоместо в третьей строке – переменную Z типа WORD. Для заполнения Z использовать AX, поместив в AL значение X, а в AH - число 10 (формат вывода).
(Из методы: “ORG *выраж* – задание текущей позиции в сегменте.”)

15) Структура процедур.

Вызов процедуры производится при помощи инструкции CALL. Если процедура NEAR, то в стек заносится только значение IP следующей за CALL’ ом

инструкции. Если процедура FAR, то дополнительно в стек заносится CS, так как FAR процедура обычно лежит в другом кодовом сегменте.

```
...
CALL A ; После выполнения этой строчки в стеке будет лежать адрес
CS:[MOV AX, BX] MOV AX, BX
...

; Общий вид процедуры
ИМЯ_ПРОЦ PROC [NEAR|FAR]
      ...
      RET ; Если PROC FAR, то ассемблер заменит RET на RETF
ИМЯ_ПРОЦ ENDP
```

Пример:
Возможное начало и завершение процедуры, которой передается управление при старте программы.

Возврат командой RET

Процедура должна быть дальней

```
ИМЯ_ПРОЦ PROC FAR ;дальняя процедура
      PUSH DS     ; помещаем в стек
      MOV AX, 0    ;сегментную часть PSP
      PUSH AX     ; и число 0

      MOV AX, ИмяСегментаДанных  ; настройка DS на
      MOV DS,AX                 ;сегмент данных
      ;ASSUME DS:ИмяСегментаДанных     ;помогаем ассемблеру
разобраться с сегм--ами
      ...
      RET ;тоже дальняя команда
ИМЯ_ПРОЦ ENDP
```

16) Внешние имена

PUBLIC ИМЯ[, ИМЯ…] – объявление в любом месте данного модуля описанных в нём имён, которые могут использоваться в других модулях.

EXTRN ИМЯ:ТИП[, ИМЯ:ТИП…] – объявление для использования в данном модуле имён, описанных в других модулях, и их типов

Работает на метках, именах процедур и переменных/константах.

ПРАВИРО: директиву EXTRN объявления внешних данных следует размещать внутри описания того же сегмента (если он есть в данном модуле), в котором они были описаны в другом модуле, иначе – вне описаний сегментов

МОДУЛЬ 1

```
EXTRN output_X: near

STK SEGMENT PARA STACK 'STACK'
      db 100 dup(0)
STK ENDS

DSEG SEGMENT PARA PUBLIC 'DATA'
      X db 'R'
DSEG ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
      assume CS:CSEG, DS:DSEG, SS:STK
main:
      mov ax, DSEG
      mov ds, ax
      call output_X
      mov ax, 4c00h
      int 21h
CSEG ENDS
PUBLIC X
END main
```

МОДУЛЬ 2

```
PUBLIC output_X
EXTRN X: byte

DS2 SEGMENT AT 0b800h
      CA LABEL byte
      ORG 80 * 2 * 2 + 2 * 2
      SYMB LABEL word
DS2 ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
      assume CS:CSEG, ES:DS2
output_X proc near
      mov ax, DS2
      mov es, ax
      mov ah, 10
      mov al, X
      mov symb, ax
      ret
output_X endp
CSEG ENDS
```

```
END
```

17) . Типы данных и задание начальных значений.

- Целые числа:
  - [+|-]XXX - обычно десятичное

```
W dw 3444
```

  - [+|-]XXXb - (BIN) в двоичной системе

```
W dw 3444b
```

  - [+|-]XXXq - (OCT) в восьмеричной системе (q, чтобы не спутать с 0)
  - [+|-]XXXo - (OCT) в восьмеричной системе
  - [+|-]XXXd - (DEC) в десятичной системе
  - [+|-]XXXh - (HEX) в шестнадцатеричной системе

```
W dw 3444h
```

- Символы и строки (символ — один чар, строка — 2 и более):
  - ‘Символы’
  - “Символы” (Строки в DOS’е $-терминированные)

```
ENT         DB    '>> $'
NLINE       DB    10, 13, '$'
```

- Структуры: (от 16/06 Евтих сказал убрать структуры, Надеюсь нам тоже)

Начальное значение может быть неопределенным. В таком случае вместо начального значения ставится “?”.

18) Способы описания меток, типы меток.

Метка в языке ассемблера может содержать следующие символы:
Буквы: от A до Z и от a до z
Цифры: от 0 до 9
Спецсимволы:
знак вопроса (?)
точка (.) (только первый символ)
знак "коммерческое эт" (@)
подчеркивание (_)
доллар ($)
Первым символом в метке должна быть буква или спецсимвол. Цифра не может быть первым символом метки, а символы $ и ? иногда имеют специальные значения и обычно не рекомендуются к использованию
Если метка располагается перед командой процессора, сразу после нее всегда ставится символ «:» (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

```
some_loop:
        lodsw ; считать слово из строки,
        cmp ax,7 ; если это 7 - выйти из цикла
        loopne some_loop
```

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится: в

```
codesg segment
        lodsw ; считать слово из строки,
        cmp ax,7 ; если это 7 - выйти из цикла
codesg ends
```

19) Команды условных переходов при работе с ЦБЗ и ЦСЗ.

Имеется 19 команд, имеющих 30 мнемонических кодов операций. (по правде, команд и мнемоник дохрена) Команды проверяют состояние отдельных флагов или их комбинаций, и при выполнении условия передают управление по адресу, находящемуся в операнде команды, иначе следующей команде.

ЦБЗ = целые без знака

ЦСЗ = целые со знаком

Команда	Мнемоника	С чем работает	Прыжок, когда
JE | JZ	Equal (Zero)	все	`ZF == 1`
JNE | JNZ	Not Equal (Not Zero)	все	`ZF == 0`
JG | JNLE	Greather	ЦСЗ	`(ZF == 0) && (SF == OF)`
JGE | JNL	Greather or Equal	ЦСЗ	`SF == OF`
JL | JNGE	Less	ЦСЗ	`SF != OF`
JLE | JNG	Less or Equal	ЦСЗ	`(ZF == 1) || (SF != OF)`
JA | JNBE	Above	ЦБЗ	`(CF == 0) && (ZF == 0)`
JAE | JNB	Above or Equal	ЦБЗ	`CF == 0`
JB | JNAE	Below	ЦБЗ	`CF == 1`
JBE | JNA	Below or Equal	ЦБЗ	`(CF == 1) || (ZF == 1)`
JO	Overflow	все	`OF == 1`
JNO	Not overflow	все	`OF == 0`
JC	Carry	все	`CF == 1`
JNC	Not Carry	все	`CF == 0`
JS	Sign (< 0)	ЦСЗ	`SF == 1`
JNS	Not Sign (>= 0)	ЦСЗ	`SF == 0`
JP	Parity	все	`PF == 1`
JNP	Not Parity	все	`PF == 0`
JCXZ	CX is Zero	все	`CX == 0`

Обработка двоичного числа со знаком:

```
SB          PROC NEAR ; all the same in all the funcs
            PUSH BP
            MOV  BP, SP
            MOV  CX, [BP + 4] ; SI
            MOV  BX, [BP + 6] ; X
            CMP  CL, 0 ; CL - flag minus
            JE   SB_SKIP_NEG ; if CL == 0
            MOV  AH, 2 ; print minus before the num
            MOV  DL, '-'
            INT  21H
            NEG  BX ; make negative
SB_SKIP_NEG: ; if the number is positive
            PUSH BX
```

```
            PUSH CX
            CALL UB ; call unsigned bin
            POP BP
            RET 4 ; delete x, si
SB          ENDP
```

20) Команды организация циклов

LOOP, метка(параметр) ;где метка(параметр) — определяет адрес перехода от -128 до +127 от команды LOOP.

Мнемоника	Описание
LOOP	--CX, если (cx != 0) - повсторяем цикл(переходим к метке,указанной в команде)
LOOPE | LOOPZ	--CX, если (CX != 0) && (ZF == 1) повторяем цикл
LOOPNE | LOOPNZ	--CX, если (CX != 0) && (ZF == 0) - повторяем цикл

```
Примеры использования:
Обычный цикл:
MOV CX, 5
SUMMATOR:
      ADD AX, 10
LOOP SUMMATOR

Аналог на C (просто для топ-прогеров)
for (int CX = 5; CX > 0; CX--)
AX += 10;

Вложенный цикл:
MOV CX, 5
SUMM:
      PUSH CX
      ADD AX, 10
      MOV CX, 7
      SUMBX:
            ADD BX, 10

      LOOP SUMBX
      POP CX
LOOP SUMM
```

```
Аналог на C (просто для топ-прогеров):
for (int CX = 5; CX > 0; CX--)
{
      AX += 10;

      // вместо PUSH использую другую переменную
      for (int CX2 = 7; CX2 > 0; CX2--)
            BX += 10;
}
```

21) Способы адресации

В инструкциях программ операнды строятся из объектов, представляемых собой имена переменных и меток, представленных своим сегментом смещения (регистры, числа, символические имена с численным значением). Запись этих объектов с использованием символов: [ ], +, - , . в различных комбинациях называют способами адресации.

1.Непосредственная адресация (значение): операнд задается непосредственно в инструкции и после трансляции входит в команду, как составная часть 1 и 2 байта.

MOV AL, 2 (1 BYTE)

MOV BX, OFFFFH (2 BYTES)

2. Регистровая адресация: значение операнда находится в регистре, который указывается в качестве аргумента инструкции.

MOV AX, BX

3. Прямая адресация: в инструкции используется имя переменной, значение которой является операндом.

MOV AX, X (в AX значение X)

MOV AX, CS:Y (в AX CS по смещению Y)

4. Косвенная регистровая адресация: в регистре, указанном в инструкции, хранится смещение (в сегменте) переменной, значение которой и является операндом. Имя регистра записывается в [ ].

MOV BX, OFFSET Z

…

MOV BYTE PTR[BX], ‘ASS’ ; в байт, адресованный регистром BX, запомнить ‘ASS’ в сегменте DS

MOV AX, [BX] ; в AX запомнить слово из сегмента, адрес BX со смещением ДS

MOV BX, CS:[SI] ; записать данные в BX, находящиеся в CS по смещению SI

5. Косвенная базово-индексная адресация (адресация с индексацией и базированием): смещение ячейки памяти, где хранятся значения

операндов, вычисляется, как сумма значений регистров, записанных в качестве параметра в инструкции. [R1+R2],где R1∈{BX,BP} , R2∈{SI,DI}

```
MOV AX, [BX +  DI]
MOV AX, [BP + SI]
```

Для BX по умолчанию сегменты из регистра DS. Для BP по умолчанию сегменты из регистра SS.

6. [Косвенная] базово-индексная адресация со смещением : (прямая с базированием и индексированием), отличается от п.5 тем, что ещё прибавляется смещение, которое может быть представлено переменной или ЦБЗ. (целое без знака число)

```
; [R1 + R2 +- ЦБЗ]
; [R1] [R2 +- ЦБЗ]
; имя [R1 + R2 +- АБС. ВЫРАЖЕНИЕ]
; имя [R1] [R2 +- АБС. ВЫРАЖЕНИЕ]

MOV AX, ARR[EBX][ECX * 2]

;ARR + (EBX) + (ECX) * 2
[имя] [база] [индекс [* масштаб]] [Абс. выражение]
Смещение операнда = смещение имени + смещение базы + значение индекса * масштаб + значение а
```

22) Организация рекурсивных подпрограмм

Рекурсивные алгоритмы предполагают реализацию в виде процедуры, которая сама себя вызывает. При этом необходимо обеспечить, чтобы каждый последовательный вызов процедуры не разрушал данных, полученных в результате предыдущего вызова. Поэтому необходимо сохранять используемые в процедуре регистры (например, EBP) и прочие промежуточные данные в стеке. Если процедура работает по соглашениям вызова Turbo C или Turbo Pascal, за этим также надо проследить (ADD SP, передача параметров через стек и прочие операции).

Пример рекурсивной процедуры по соглашениям Turbo C типа void recur(int):

```
; вызов
PUSH CX
MOV DX, 5
PUSH DX
CALL RECUR
ADD SP, 2
POP CX

; процедура (SI, DI, DS не сохраняем, потому что они не меняются)
RECUR PROC FAR
        PUSH BP
        MOV BP, SP
```

```
        MOV CX, [BP+6] ; получаем аргумент
        CMP CX, 0 ; если он 0
        JE RECEND ; then exit
        ; do smth with it
        PUSH CX ; вызываем эту же процедуру
        CALL SUMM
        ADD SP, 2 ; do not forget about the args
RECEND:
        POP BP
        RETF
RECUR ENDP
```

Еще один пример (необязательно но на всякий) N!
Перед первым вызовом в EAX хранится единица, результат работы функции будет в EAX.

```
factorial proc near
        cmp ECX, 0h ; в ECX хранится N
        je done
        push EDX ; сохранение EDX перед умножением
        mul ECX
        pop EDX ; восстановление EDX
        dec ECX
        call factorial
done:
        ret
factorial endp
```

23) Арифметические команды

ADD Приемник, Источник
Сложение байтов или слов
Функция: Приемник := Приемник + Источник
изменяет флаги:OF, SF, ZF, AF, PF, CF.
ADC Приемник, Источник
Сложение байтов или слов с переносом
Функция:
Приемник := Приемник + Источник+CF
изменяет флаги:OF, SF, ZF,AF,PF,CF.

```
  CX 0000 F000 AX
   +       +
  DX 0000 8000 BX
   +CF
   ----------
  CX 0001 7000 AX
```

```
MOV AX,0F000H
MOV BX,8000H
MOV CX,0
MOV DX,0
ADD AX,BX
ADC CX,DX
```

INC Приемник Увеличение байта или слова на 1
Функция: Приемник:=Приемник+1
изменяет флаги:OF,SF,ZF,AF,PF.
! не влияет на CF !
AAA - коррекция в AL неупакованного кода при сложении

Например, если при сложении чисел 06 и 09 в регистре AX окажется число 000Fh, то команда AAA исправит его, и в регистре AX будет число 0**1**0**5** (неупакованное десятичное число **15**):

```
MOV AX, 15   ; AH = 00, AL = 0Fh = 15
AAA          ; AH = 01, AL = 05
```

DAA - коррекция в AL упакованного кода после сложения (иногда как перевод из 16 в 10)? оно же даж называется decimal adjust after some shit ПАЕБАТЬ+

Пример 1

```
mov AL,87h ;Упакованное BCD 87
add AL,04h ;После сложения AL=8Bh
daa ;AL=91h, т.е. упакованное BCD 91
```

Пример 2

```
mov AL,87h ;Упакованное BCD 87
add AL,11h ;После сложения AL=97h
daa ;AL=97h, т.е. упакованное
;BCD 97 (в данном случае
;команда daa ничего не делает)
```

SUB Приемник, Источник Вычитание байтов или слов
Функция: Приемник := Приемник-Источник
изменяет флаги:OF,SF,ZF,AF,PF,CF.

SBB Приемник,Источник
Вычитание байтов или слов с заемом
Функция: Приемник:=Приемник-Источник-CF
изменяет флаги:OF,SF,ZF,AF,PF,CF.

```
 CX 0020 7000 AX
      -       -
  DX 0000 8000 BX
```

-CF

CX 001F F000 AX

```
MOV AX,7000H
MOV BX,8000H
MOV CX,020H
MOV DX,0
SUB AX,BX
SBB CX,DX
```

DEC Приемник Уменьшение байта или слова на 1
Функция: Приемник:=Приемник-1
изменяет флаги:OF,SF,ZF,AF,PF.
! не влияет на CF !

NEG Приемник
Изменение знака байта или слова
Функция: Приемник:=-Приемник
изменяет флаги:OF,SF,ZF,AF,PF.
! Если Приемник <> 0, то CF:=1, иначе CF:=0 и Приемник сохранит значение 0!
! Попытка изменить знак для -128 или -32768 не изменит операнд, но установит OF:=1!

УМНОЖЕНИЕ

без знака: MUL Источник(байт|слово) {Рег|ОП}
Функция MUL для байтов: AX:=AL*Источник,
AH - старшая часть произведения,
AL - младшая часть произведения.
CF:=OF:= 0, если AH=0; 1, если AH<>0.=
флаги SF,ZF,AF,PF после операции не определены .
Пример.

```
MOV Al,255
MOV DL,4
MUL DL
```

Функция MUL для слов: (DX,AX):=AX*Источник,
DX - старшая часть произведения,
AX - младшая часть произведения.
CF:=OF:=0, если DX=0; 1, если DX<>0.
флаги SF,ZF,AF,PF после операции не определены .
Пример.

```
MOV AX,0FFFFH
MOV DX,4
MUL DX
```

УМНОЖЕНИЕ

со знаком: IMUL Источник(байт|слово) {Рег|ОП}

Функция IMUL для байтов: AX:=AL*Источник,
AL - младшая часть произведения,
AH - старшая часть произведения.
CF:=OF:=0, если AH=знаковое расширение AL (т.е. результат уместился в AL); 1, если иначе.
флаги SF,ZF,AF,PF после операции не определены .

Пример.

```
MOV Al,-16
MOV DL,4
IMUL DL
```

Функция IMUL для слов: (DX,AX):=AX*Источник,
AX - младшая часть произведения,
DX - старшая часть произведения.
CF:=OF:= 0, если DX=знаковое расширение AX (т.е. результат уместился в AX); 1, если иначе.
флаги SF,ZF,AF,PF после операции не определены .

Пример.

```
MOV AX,0FFH
MOV DX,4
IMUL DX
```

3. AAM - коррекция в AH и AL неупакованных чисел после умножения

ДЕЛЕНИЕ без знака:

DIV Источник(байт|слово) {Рег|ОП}

Функция DIV для байтов:
AL:= частное от деления AX:Источник,
AH:= остаток от деления AX:Источник.
флаги OF,SF,ZF,AF,PF,CF после операции не определены .

! Если частное не м.б. размещено в AL, т.е. если оно >0FFH, то происходит прерывание типа 0 ("деление на 0").

Пример.

```
MOV AH,0
MOV AL,240
MOV DL,4
DIV DL  ;"деление на 0" при AH>=4
```

Функция DIV для слов:

AX:= частное от деления (DX,AX):Источник,
DX:= остаток от деления (DX,AX):Источник.
флаги OF,SF,ZF,AF,PF,CF после операции не определены .
! Если частное не м.б. размещено в AX, т.е. если оно >0FFFFH,
то происходит прерывание типа 0 ("деление на 0").
Пример.

```
XDIV    DW  4
MDIV:   MOV AX,8000H
        MOV DX,3   ;"деление на 0" при DX>=4
        DIV XDIV
```

2. Деление байтов или слов со знаком: IDIV Источник(байт|слово) {Рег|ОП}
Функция IDIV для байтов:
AL:= частное от деления AX:Источник,
AH:= остаток от деления AX:Источник.
флаги OF,SF,ZF,AF,PF,CF после операции не определены .
! Если частное не м.б. размещено в AL, т.е. если оно выход-
ит за диапазон -127..+127 (81H..7FH), то происходит пре-
рывание типа 0 ("деление на 0").
Пример.

```
        MOV AH,0
        MOV AL,240
        MOV DL,4
        IDIV DL  ;"деление на 0" при AH>=2
```

;! Результат уменьшается до целого числа отбрасыванием дробной части частного. Остаток имеет тот же знак, что и делимое.

```
        MOV AX,3
        MOV BX,2
        IDIV BL
        MOV AX,-3
        IDIV BL
```

Функция IDIV для слов:
AX:= частное от деления (DX,AX):Источник,
DX:= остаток от деления (DX,AX):Источник.
флаги OF,SF,ZF,AF,PF,CF после операции не определены .
! Если частное не м.б. размещено в AL, т.е. если оно выход-
ит за диапазон -32767..+32767 (8001H..7FFFH), то про-
исходит прерывание
типа 0 ("деление на 0").
Пример.

```
XIDIV   DW  4
MIDIV:  MOV AX,8000H
        MOV DX,1   ;"деление на 0" при DX>=2
        IDIV XDIV
```

3. AAD - готовит в AL неупакованное десятичное для последующего

деления командой DIV.

24) Связывание подпрограммы.

Связывание подпрограмм — совокупность действий по:
- передаче управления подпрограмме
- передаче параметров через регистры, общую область памяти, области параметров
- возврату управления из подпрограмм
- запоминанию и восстановлению состояния вызывающей программы

Все это — связывание подпрограмм в исполняемом модуле.
В более широком плане, когда речь идет о **динамическом вызове подпрограмм** или программ, хранящиеся на диске в виде отдельных модулей, к этому перечню добавляется:
1. загрузка модуля подпрограмм в ОЗУ и разрешение внешних ссылок (по передаче управления и по передаче данных).
2. освобождение памяти, после завершения работы вызываемого модуля и возврат управления из вызываемого модуля.

Пример связывания:

файл №1

```
PUBLIC X
EXTRN exit: far

SSTK SEGMENT para STACK 'STACK'
        db 100 dup(0)
SSTK ENDS

SD1 SEGMENT para public 'DATA'
        X db 'X'
SD1 ENDS

SC1 SEGMENT para public 'CODE'
        assume CS:SC1, DS:SD1
main:
        jmp exit
SC1 ENDS
END main
```

файл №2

```
SD2 SEGMENT para 'DATA'
        Y db 'Y'
```

```
SD2 ENDS

SC2 SEGMENT para public 'CODE'
        assume CS:SC2, DS:SD2
exit:
        ; do smth
        mov ax, 4c00h
        int 21h
SC2 ENDS
END
```

25) Команды CALL и RET.

CALL передает управление в другую строку программы, при этом сохраняя в стеке адрес возврата для команды RET (он указывает на следующую после CALL инструкцию). Если переход в другой сегмент, также сохраняется текущий сегмент кода (CS), в который нужно будет вернуться. Передача управления подпрограмм с помощью команды CALL:
CALL имя_процедуры или имя_метки в подпрограмме(прямая адресация)
CALL [переменная] (косвенная адресация)

Инструкция RET в Ассемблере выполняет возврат из ближней процедуры. У этой команды обычно нет операндов, хотя в качестве операнда может быть чётное число (только непосредственное значение).
Если не заморачиваться о таких вещах как ближний/дальний вызов, то можно сказать, что эта команда выполняет выход из программы или процедуры.
Алгоритм работы команды RET:

1. Получить из стека IP (адрес возврата в программу из процедуры).
2. Если имеется операнд, то SP = SP + операнд. Операнд необязателен, но если он есть, то после того, как будет считан адрес возврата, из стека будет удалено столько байтов, сколько указано в операнде. Это требуется, если при вызове процедуры ей передавались параметры через стек.

Никакие **флаги** при выполнении команды не изменяются.
В зависимости от того, как была описана процедура, которую завершает команда RET, ассемблер может заменить команду RET на команду RETN или RETF.

```
.model tiny
.code
ORG 100H
```

```
start:
        CALL MYPROC                     ; вызвать процедуру
        ADD AX, 1
        RET                             ; вернуться в ос
MYPROC PROC                             ; объявление процедуры
        MOV AX, 1
        RET                             ; вернуться в программу
MYPROC ENDP

END START
```

26) Способы передачи параметров подпрограмм

1. Через регистры:

плюсы: быстро, просто

минусы: мало регистров, нужно постоянно следить за значениями в регистрах, регистры маленькие

```
mulSI  PROC
                mov AX, SI
                mul n
                mov SI, AX ; AX = AL * n = SI * 5
                ret
mulSI  ENDP

print_elem PROC
                push SI
                call mulSI
                mov DL, B[SI][BX]
                pop SI
                call print
                ret
print_elem ENDP
```

2. Через общую область памяти, которая организуется при помощи COMMON сегментов:

плюсы: удобно возвращать большое кол-во данных

минусы: можно испортить данные извне + данные перекрываются в памяти, что не есть хорошо

```
PARSEG SEGMENT COMMON ; потом грузим этот сегмент в DS или ES
        px DW ? ; Параметр 1 ; вызывающей и вызываемой стороны и пишем
        py DW ? ; Параметр 2 ; читаем параметры
        re DW ? ; Результат
PARSEG ENDS
```

3. Через общий стек (используется чаще всего):

плюсы: можно передавать большое количество параметров

минусы: количество параметров ограничено размерами стека и разрядностью BP + требуется наличие стека

```
PUSH EBP ; current place in the stack, EBP N NF
MOV  EBP, ESP  ; go to current position in stack
MOV  ECX, [EBP + 8] ; get N
MOV  EBX, [EBP + 12] ; get address of NF
```

4. Через области параметров, адреса которых передаются через регистры:

(по сути то же, что и 1, но передаем адреса начала списков параметров и может быть их количество)

плюсы: можно передавать большое количество параметров или даже переменное число параметров

минусы: должна быть заранее подготовленная под параметры и заполненная область памяти, надо иметь в виду, что вызываемый код может работать с другой областью памяти.

27) Способы сохранения и восстановления состояния вызывающей программы

Запоминание и восстановление состояния вызывающей программной единицы (т.е. регистров).

Вариант запоминания регистров определяется следующим:

- кто (вызывающая / вызываемая программная единица) является владельцем области сохранения
- кто из них фактически выполняет сохранение регистров

Вариант 0:

Добавлено примечание ([6]): Добавить пример?

Хранилище — область (стек) вызываемой программы.
Сохранитель (восстановитель) — вызывающая программа
“+” минимум затраты времени и памяти на сохранение, т.к. вызываемая программа знает, какие нужно сохранить регистры
“-” если много вызовов подпрограммы, то команды, выполняющие сохранение регистров, будут занимать значительную часть кода

Вариант 1:

Хранилище — область (стек) вызываемой программы.
Сохранитель (восстановитель) — вызываемая подпрограмма
“+” сокращение кода основной подпрограммы
“-” подпрограмма не знает, какие регистры сохранять, сохранение регистров описывается в специальном соглашении.

```
mulSI  PROC
                mov AX, SI
```

```
                mul n
                mov SI, AX ; AX = AL * n = SI * 5
                ret
mulSI  ENDP

print_elem PROC
                push SI
                call mulSI
                mov DL, B[SI][BX]
                pop SI
                call print
                ret
print_elem ENDP
```

28) Конвенции языков высокого уровня

Соглашения о связях в Turbo Pascal

Передача параметров через стек в порядке загрузки слева направо. Возврат результата:

- скалярные результаты по 2, 1 байт через AX (AL), указатели (4 байта) через DX:AX (строка указатель)
- Real (6байтовые паскальные) в DX:BX:AX (это не адресация, просто возврат через эти регистры делится на куски по два байта в таком порядке, сверху такой же смысл)

Сохранение результатов (SI , DI , BP) в подпрограмме .
За уничтожение параметров в стеке отвечает вызываемая программа, например используя RET N (N — количество байт, которые надо очистить)

Соглашения о связях в Turbo C

Передача параметров через общий стек. Помещаются параметры из списка параметров функции в стек в порядке справа налево. Если результат возвращается через имя функции, то он помещается, как правило, в регистр AX или DX:AX.
Сохранение регистров внутри подпрограммы (BP, SP, CS, DS, SS, SI, DI) , т.е. их поместить в стек в начале подпрограммы, если их значения могут изменяться в подпрограмме.
За уничтожение параметров в стеке отвечает вызывающая сторона.
Освобождение стека выполняет вызывающая сторона командой ADD SP,N (N — количество байт, которые надо очистить).

Соглашение	Параметры	Очистка стека	Регистры

PasKAL	Слева направо	Процедура	--
C	Справа налево	Вызывающая программа	--
Fastcall (быстрый или регистровый вызов)	Слева направо	Процедура	EAX, EDX, ECX, далее стек
STDcall (стандартный вызов)	Справа налево	Процедура	--

Конвенция Pascal заключается в том, что параметры из программы на языке высокого уровня передаются в стеке и возвращаются в регистре AX/EAX, — это способ, принятый в языке PASCAL, — просто поместить параметры в стек в естественном порядке.

а) **Конвенция pascal**. Структура стека показана на рисунке 3.11.

```
. 386 .
model flat
.code
public ADD1
 ADD1 proc
        push EBP
          push EBP, ESP
          mov EAX, [EBP+16]
          add EAX, [EBP+12]
          mov EDX, [EBP+8]
          mov [EDX], EAX
          pop EBP
          ret 12 ; стек освобождает процедура
ADD1 endp
end
```

б) **Конвенция cdecl**. Структура стека показана на рисунке 3.12.

```
 . 386
 . model flat
 . code
 public ADD1
ADD1 proc
          push EBP
          push EBP, ESP
          mov EAX, [EBP+8]
          add EAX, [EBP+12]
          mov EDX, [EBP+16]
```

```
        mov [EDX], EAX
        pop EBP
        ret ;стек освобождает
        ;вызывающая программа
ADD1 endp
 end
```

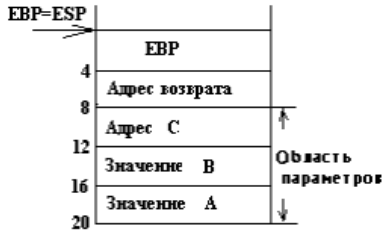


Рисунок 3.11 – Структура стека для конвенции pascal

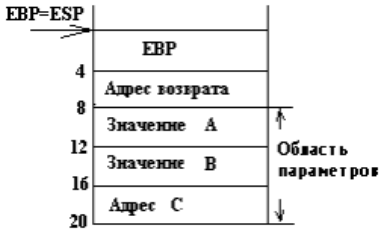


Рисунок 3.12– Структура стека для конвенции cdecl

в) **Конвенция register**

```
 .386 Размещение параметров:
 .model flat                    ;первый параметр A в регистре EAX
 .code                                  ;второй параметр B в регистре EDX
 public ADD1                    ;третий параметр адрес C в регистре ECX
ADD1 proc
        add EDX,EAX
        mov [ECX],EDX
        ret ; стек освобождает вызывающая программа
ADD1 endp
 end
```

г) Конвенция stdcall. Структура стека показана на рисунке 3.13.

```
 .386
 .model flat
 .code
 public ADD1
ADD1 proc
         push EBP
        push EBP, ESP
         mov EAX, [EBP+8]
        add EAX, [EBP+12]
         mov EDX, [EBP+16]
         mov [EDX], EAX
         pop EBP
```

```
        ret 12 ; стек освобождает процедура
ADD1 endp
 end
```

Рисунок 3.13 – Структура стека для конвенции stdcall

29) Команды сдвига

SHL операнд, количество_сдвигов

SHR операнд, количество_сдвигов

SHL и **SHR** сдвигают биты операнда (регистр/память) влево или вправо соответственно на один разряд и изменяют флаг переноса cf. При логическом сдвиге все биты равноправны, а освободившиеся биты заполняются нулями. Указанное действие повторяется количество раз, равное значению второго операнда.

SAL операнд, количество_сдвигов

SAR операнд, количество_сдвигов

Команда **SAR** — сдвигает биты операнда (регистр/память) вправо на один разряд, значение последнего вытолкнутого бита попадает в флаг переноса, а освободившиеся биты заполняются знаковым битом.

Команда **SAL** — сдвигает биты операнда (регистр/память) влево на один разряд, значение последнего вытолкнутого бита попадает в флаг переноса, а освободившиеся биты заполняются нулями, при этом знаковый бит не двигается

rol операнд, количество_сдвигов

ror операнд, количество_сдвигов

rcl операнд, количество_сдвигов

rcr операнд, количество_сдвигов

ROL и **ROR** сдвигают все биты операнда влево(для ROL) или вправо(для ROR) на один разряд, при этом старший(для ROL) или младший(для ROR) бит операнда вдвигается в операнд справа(для ROL) или слева(для ROR) и становится значением младшего(для ROL) или старшего(для ROR) бита операнда; одновременно выдвигаемый бит становится значением флага переноса cf. Указанные действия повторяются количество раз, равное значению второго операнда.

RCL и **RCR** сдвигают все биты операнда влево (для RCL) или вправо (для RCR) на один разряд, при этом старший(для RCL) или младший(для RCR) бит становится значением флага переноса cf; одновременно старое значение флага переноса cf вдвигается в операнд справа(для RCL) или слева(для RCR) и становится значением младшего(для RCL) или старшего(для RCR) бита операнда. Указанные действия повторяются количество раз, равное значению второго операнда.

```
MOV DX, [BP + 8] ; DX = X
        MOV SI, 16 ; maximum digit
MOV CL, 4 ; length of bitwise shift

SHR DX, CL ; DX = X, CL = 4, delete last num
```

30) Команды логических операций.

1. Логическое отрицание для байтов или слов:

NOT Приемник {Регистр|Оперативная память}

Функция: Инвертирование значений разрядов Приемника.

Не изменяет флаги.

Пример. NOT(10101001) -> 01010110 *

```
MOV AH,10101001B
NOT AH
```

Пример. NOT(0000h) -> 1111h *

```
MOV AX,0
NOT AX
NOP ;Флаги не изменились!
```

2. Логическое умножение (И, конъюнкция) для байтов или слов:

AND Приемник,Источник

Приемник - {Рег|ОП}

Источник - {Рег|ОП|Знач}

Функция: Приемник:=Приемник & Источник

при поразрядном выполнении &:

X1 = 0 0 1 1

X2 = 0 1 0 1

X1 & X2 = 0 0 0 1

Флаги PF,ZF,SF устанавливаются в соответствии с результатом,

CF:=OF:=0,

AF - не определен.

Пример.

```
MOV AH, 00001111B; &
MOV AL, 11000001B
AND AH,AL ;AH= 00000001B
NOP
```

3. Логическое сложение (ИЛИ, дизъюнкция) для байтов или слов:

OR Приемник,Источник

Приемник - {Рег|ОП}

Источник - {Рег|ОП|Знач}

Функция: Приемник:=Приемник V Источник

при поразрядном выполнении V:

X1 = 0 0 1 1

X2 = 0 1 0 1

X1 V X2 = 0 1 1 1

Флаги PF,ZF,SF устанавливаются в соответствии с результатом,

CF:=OF:=0,

AF - не определен.

Пример.

```
MOV AH, 00001111B; V
MOV AL, 11000001B
OR AH,AL ;AH= 11001111B
NOP
```

4. Неравнозначность (Исключающее ИЛИ, сумма по модулю 2) для байтов или слов:

XOR Приемник,Источник

Приемник - {Рег|ОП}

Источник - {Рег|ОП|Знач}

Функция: Приемник:=Приемник (+) Источник

при поразрядном выполнении (+):

X1 = 0 0 1 1

X2 = 0 1 0 1

X1(+)X2 = 0 1 1 0

Флаги PF,ZF,SF устанавливаются в соответствии с результатом,

CF:=OF:=0,

AF - не определен.

Пример.

```
MOV AH, 00001111B; (+)
MOV AL, 11000001B
XOR AH,AL ;AH= 11001110B
NOP
```

5. Тестирование для байтов или слов:

TEST Приемник,Источник

Приемник - {Рег|ОП}

Источник - {Рег|ОП|Знач}

Функция: Вычисляет: Приемник & Источник (не меняя операндов) для последующей установки флагов.

Флаги PF,ZF,SF устанавливаются в соответствии с результатом,

CF:=OF:=0,

AF - не определен.

Пример.

```
MOV AH,00001111B
TEST AH,11110000B
JNZ M5
NOP
M5: NOP
```

6. Проверка битов

BT Источник, индекс

Назначение: извлечение значения заданного бита в флаг cf

Алгоритм работы:

- получить бит по указанному номеру позиции в операнде источник;
- установить флаг cf согласно значению этого бита.

Команду bt используют для определения значения конкретного бита в операнде источник. Номер проверяемого бита задается содержимым второго операнда (значение числом из диапазона 0...31). После выполнения команды, флаг cf устанавливается в соответствии со значением проверяемого бита.

```
mov ebx, 01001100h
bt ebx, 8 ; проверка состояния бита 8 и установка cf = 1
jc m1 ; перейти на м1, если проверяемый бит равен 1
```

7. Проверка бита с инверсией (дополнением)

BTC Источник, индекс

Назначение: извлечение значения заданного бита в флаг cf и изменение его значения в операнде на обратное.

Алгоритм работа:

- получить значение бита с номером позиции индекс в операнде источник;
- инвертировать значение выбранного бита в операнде источник;
- установить флаг cf исходным значением бита.

Команда btc используется для определения и инвертирования значения конкретного бита в операнде источник. Номер проверяемого бита задается содержимым второго операнда индекс (значение из диапазона 0...31). После выполнения команды флаг cf устанавливается в соответствии с исходным значением бита, то есть тем, которое было до выполнения команды.

```
mov ebx, 01001100h
; проверка состояния бита 8 и его обращение:
btc ebx, 8 ; cf = 1 и ebx = 01001000h
```

8. Проверка бита с его сбросом в ноль

BTR источник, индекс

Назначение: извлечение значения заданного бита в флаг cf и изменение его значения на нулевое

Алгоритм работы:

- получить значение бита с указанным номером позиции в операнде источник;
- установить флаг cf значением выбранного бита;
- установить значение исходного бита в операнде в 0.

Команда btr используется для определения значения конкретного бита в операнде источник и его сброса в 0. Номер проверяемого бита задается содержимым второго операнда индекс (значение из диапазона 0...31). В результате выполнения команды флаг cf устанавливается в соответствии со значением исходного бита, то есть тем, что было до выполнения операции.

```
mov ebx, 01001100h

; проверка состояния бита 8 и его сброс в 0

btr ebx, 8 ; cf  = 1 и ebx = 01001000h
```

9. Проверка бита с его установкой в 1

BTS источник, индекс

Назначение: извлечение значения заданного бита операнда в флаг cf и установка этого бита в единицу

Алгоритм работы:

- получить значение бита с указанным номером позиции в операнде источник;
- установить флаг cf значением выбранного бита;
- установить значение исходного бита в операнде источник в 1.

Команда bts используется для определения значения конкретного бита в операнде источник и установки проверяемого бита в 1. Номер проверяемого бита задается содержимым второго операнда индекс (значение из диапазона 0...31). После выполнения команды флаг cf устанавливается в соответствии со значением исходного бита, то есть тем, что было до выполнения операции.

```
mov ebx, 01001100h

; проверка состояния бита 0 и его установка в 1

bts ebx, 0 ; cf  = 0 и ebx = 01001001h
```

31) Команды обработки строк и префиксы повторения.

Этими командами могут обрабатываться строки (последовательности байтов или слов). Максимальная длина такой строки - 64 кб (так как регистры 16-битные). С ними часто используются префиксы повторения.
Существуют пять основных операций или примитивов, выполняющих обработку одного элемента - слова (байта) за один приём.
Команды-примитивы (текущий элемент = на него сейчас указывают SI или DI):

- сравнение (CMPS): сравнивает поэлементно источник и приемник;

- сканирование (SCAS): ищет значение AX(AL) в приемнике (сравнивает текущий элемент с AX(AL));
- пересылка (MOVS): копирует поэлементно из источника в приемник
- загрузка (LODS): копирует текущий элемент из источника в AX(AL);
- сохранение в ОЗУ (STOS): заменяет текущий элемент в приемнике на AX(AL).

Каждая команда имеет 4 разновидности (на примере MOVS):
- без суффикса размера элементов, но с явным указанием приёмника и источника: MOVS <ПРИ>, <ИСТ>
- ключевое слово с суффиксом B и без параметров (команда побайтовой обработки): MOVSB
- ключевое слово с суффиксом W и без параметров (команда обработки слов): MOVSW
- ключевое слово с суффиксом D и без параметров (команда обработки двойных слов): MOVSD

В версиях этих команд без параметров считается, что источник находится в сегменте DS по смещению SI (DS:SI), приемник — в ES:DI. После выполнения команды автоматически изменяется значение в SI и DI, причём SI и DI возрастают на 1 (если в конце B), 2 (если в конце W) или 4 (если в конце D) при флаге DF = 0, уменьшаются на 1/2/4 при DF = 1.
Состоянием DF можно управлять командами CLD (DF=0) и STD (DF=1). В версии с параметрами 1 или 2 выбирается автоматически в зависимости от разрядности аргумента.

Префиксы повторения:
- REP <команда> - выполнять команду CX раз.
- REPE <команда> - выполнять команду пока CX<>0 перед и (REPZ <команда>) FZ=1 после выполнения команды.
- REPNE <команда> - выполнять команду пока CX<>0 перед и (REPNZ <команда>) FZ=0 после выполнения команды.

```
CLD                 ; установка флага ДФ в ноль
REP MOVSB           ; в ЕДИ кладется ЕСИ, оба регистра увеличиваются на 1,
                ;это делается CX раз,строка ЕСИ полностью копируется в ЕДИ
```

32) Команды пересылки строк

Команды пересылки строк MOVSB, MOVSW, MOVS, MOVSD.
с суффиксом B - для обработки байтов,
с суффиксом W - для обработки слов,
с суффиксом D - для обработки двойных слов,
без суффикса - обработка байтов или слов определяется типом операндов.

Пример:

```
...
std ; df := 1
```

```
        add edi, ecx ; приемник
        dec edi
        add esi, ecx ; источник
        dec esi ; df - справа налево
norev:
        rep movsb; из esi в edi записывает байты ecx - сколько последовательно
если df- 0 слева направо если df - 1 справа налево
```

33) Команды сравнения строк.

Команды сравнения строк CMPSB, CMPSW, CMPS, CMPSD.

- с суффиксом B - для обработки байтов,
- с суффиксом W - для обработки слов,
- с суффиксом D - для обработки двойных слов,
- без суффикса - обработка байтов или слов определяется типом операндов.

Сравнение байтов CMPSB

Функция: сравнение байта по адресу DS:SI с байтом по адресу ES:DI

а) DF = 0,

```
        CMP [DS:SI], [ES:DI] ; (т.е. вычитаем из источника приемник и ставим
флаги)
        SI = SI + 1
        DI = DI + 1
```

б) DF = 1

```
        CMP [DS:SI], [ES:DI]
        SI = SI - 1
        DI = DI - 1
```

Сравнение слов CMPSW

Функция: сравнение слова по адресу DS:SI со словом по адресу ES:DI То же самое, но сравниваются слова и

а) SI = SI + 2 DI = DI + 2

б) SI = SI - 2 DI = DI - 2

Аналогично с двойными словами (4 байта) CMPSD

Общая команда CMPS <Приемник>, <Источник>

Функция: то же, что и команды выше, но для адресов начала источника и приемника, указанных в операндах:

а) ~CMPSB, если <Приемник> и <Источник> - BYTE

б) ~CMPSW, если <Приемник> и <Источник> - WORD

в) ~CMPSD, если <Приемник> и <Источник> - DWORD

Примечание: CMPS выполняет CMP <ИСТОЧ>, <ПРИЕМ>, а не <ПРИЕМ>, <ИСТОЧ>! То есть вычитается из источника приемник, а не наоборот.

Пример. Сравнить строки Str1 и Str2 из 8 символов и установить:

```
        N=-1 при Str1 < Str2;
```

```
        N=0 при Str1=Str2;
        N=1 при Str1>Str2.
DSEG SEGMENT PARA PUBLIC 'DATA'
Str1 DB '....._i_'
N DB ?
DSEG ENDS

ESEG SEGMENT PARA PUBLIC 'DATA'
Str2 DB '....._I_'
ESEG ENDS
M2:
        MOV SI,OFFSET Str1
        MOV DI,OFFSET Str2
        CLD
        MOV CX,8
        MOV N,-1
        REPE CMPS Str1,Str2 ;Str2 - Str1
        JB M202 ;Str2 < Str1
        JE M201 ;Str2 = Str1
        INC N
M201: INC N
M202: JMP M2; $ + 2
```

34) Команды сканирования строк.

1) Сканирования байтов SCASB
Функция:

а) При DF=0 (исп. CLD)	б) При DF=1 (исп. STD)
- Установка флагов CF, PF, AF, ZF, SF, OF по значению разности: AL-Байт[ES:DI] (AL - Приемник)	- Установка флагов CF, PF, AF, ZF, SF, OF по значению разности: AL-Байт[ES:DI] (AL - Приемник)
- DI:=DI+1	- DI:=DI-1

2) Сканирования слов SCASW
Функция:

а) При DF=0 (исп. CLD)	б) При DF=1 (исп. STD)
- Установка флагов CF, PF, AF, ZF, SF, OF по значению разности: AX-Слово[ES:DI] (AX - Приемник)	- Установка флагов CF, PF, AF, ZF, SF, OF по значению разности: AX-Слово[ES:DI] (AX - Приемник)
- DI:=DI+2	- DI:=DI-2

3) Сканирования байтов или слов SCAS Приемн
Функция:
а) Та же, что у SCASB, если тип Приемн = BYTE
а) Та же, что у SCASW, если тип Приемн = WORD
Пример. Найти длину строки Str3, заканчивающейся кодом 0,и число пробелов в конце строки *

```
 DSEG SEGMENT PARA PUBLIC 'DATA'
SX      DB   'SS       ',0
LSX     DW   0
XSX     DW   0
DSEG ENDS
 M3:    MOV DI,OFFSET SX
        PUSH ES
        PUSH DS
        POP ES
        ASSUME ES:DSEG  ; ! ДИРЕКТИВА НЕОБХОДИМА
        MOV AL,0
        CLD
        MOV CX,-1  ; если задать CX:=0, то
 REPNE SCAS SX   ; эта команда будет пропущена
        SUB DI,2
        NEG CX
        SUB CX,2
        MOV LSX,CX
        STD
        MOV AL,' '
  REPE SCASB
        INC CX
        NEG CX
        ADD CX,LSX
        MOV XSX,CX
        JMP M5
```

35) Команды загрузки строк.

1) Загрузки байтов LODSB
Функция:
а) При DF=0 (исп. CLD)
б) При DF=1 (исп. STD)
- AL:=Байт[DS:SI] - AL:=Байт[DS:SI]
; в регистр al закидываем то, что в DS:SI, движемся к след
- SI:=SI+1 - SI:=SI-1

```
INIT_PORT:
```

```
        DB      '$CMD0000'              ;The string we want to send
        .
        .
        CLD                             ;Move forward through string at INIT_PORT
        LEA     SI, INIT_PORT           ;SI gets starting address of string (LEA - получение
адреса)
        MOV     CX, 8                   ;CX is counter for LOOP instruction
 AGAIN:
         LODSB                          ;Load a byte into AL...
        OUT     250,AL                  ;   ...and output it to the port.
        LOOP    AGAIN
```

2) Загрузки слов LODSW
Функция:
а) При DF=0 (исп. CLD)
б) При DF=1 (исп. STD)
- AX:=Слово[DS:SI] - AX:=Слово[DS:SI]
- SI:=SI+2 - SI:=SI-2
3). Загрузка байтов или слов SCAS Источн
Функция:
а) Та же, что у LODSB, если тип Источн = BYTE
б) Та же, что у LODSW, если тип Источн = WORD

36) Команды сохранения строк.
1) Сохранения байтов STOSB
Функция:
а) При DF=0 (исп. CLD)
б) При DF=1 (исп. STD)
- Байт[ES:DI]:=AL - Байт[ES:DI]:=AL
- DI:=DI+1 - DI:=DI-1

2) Сохранения слов STOSW
Функция:
а) При DF=0 (исп. CLD)
б) При DF=1 (исп. STD)
- AX:=Слово[ES:DI] - AX:=Слово[ES:DI]
- DI:=DI+2 - DI:=DI-2

3) Сохранения байтов или слов STOS Приемн
Функция:
а) Та же, что у STOSB, если тип Приемн = BYTE
б) Та же, что у STOSW, если тип Приемн = WORD

Пример.

Если команду записи в строку использовать в сочетании с префиксом REP, то такая команда будет полезна для инициализации блока памяти; следующий пример иллюстрирует инициализацию сто-байтового блока памяти, расположенного по адресу BUFFER, в 0 :

```
        MOV  AL, 0      ;значение,которое присваиваем
                        ;при инициализации
        LEA  DI, BUFFER ;загружаем стартовый адрес блока памяти
        MOV  CX, 100    ;размер блока памяти
        CLD             ;будем двигаться в прямом направлении
    REP  STOSB          ;сравните эту строку с ПРИМЕРом для STOS
```

37) Листинг программы.

Файл с исходным кодом программы, в котором развернуты все макросы, метки заменены на адреса, константы — на их значения. Генерируется ассемблером в процессе подготовки к трансляции. Для получения у MASM можно указать флаги `/Zi /L`, у ML — флаг `/Fl`.

Управление листингом.

1) Директива .LALL: далее включать в листинг программы полные макрорасширения (кроме комментариев после ;;).
2) Директива .XALL (.LISTMACROALL): далее включать в листинг программы только те предложения макрорасширений, которые генерируют коды и данные.
3) Директива .SALL: далее не выводить тексты макрорасширений в листинг.
4) Директива .NOLIST: далее вообще прекратить вывод листинга до появления иной директивы.

Пример.

```
.LALL
A3_1 MACRO
NOP ; MO A3_1
NOP ;; MO A3_1
ENDM
A3_1
.SALL
A3_2 MACRO
NOP
ENDM
A3_2
```

38) Макросредства.

При написании любой программы на ассемблере, возникают некоторые трудности: повторяемость некоторых идентичных или незначительно

отличающихся участков программы; необходимость включения в каждую программу участков кода, которые уже были использованы в других программах; ограниченность набора команд.
Для решения этих проблем в языке ассемблер существуют макросредства.
Отличие макросредств от подпрограмм:

- подпрограммы требуют подготовки для вызова (это бывает дольше, чем сам код)
- если в программе много макровыводов, то увеличивается размер кода
- параметры при записи макрокоманды должны быть известны уже на стадии ассемблирования
- в макрокомандах можно пропускать параметры

Пример:
Описание макроса:

```
PRINTSTR MACRO STR
  MOV  DX, STR
  MOV  AH, 9
  INT  21H
ENDM
```

Вывод строки по смещению R.
Пример вызова:

```
LEA      BX, SOME_STR
PRINTSTR BX ; макрокоманда
```

39) Макроопределения (макрофункций и макропроцедур) и макрокоманды.

Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.
Макрорасширения (результат макроподстановки)— последовательность предложений языка, порождающаяся макрогенератором при обработке макрокоманды. /* под управлением макроопределения.*/
Макрокоманда (ссылка на макрос) — предложение в исходном тексте программы, которое воспринимается макрорасширением, как приказ построить макрорасширение и вставить на её место (“вызов” макроса).
Описание макроопределения:

```
ИМЯ MACRO [форм.пар.1[,форм.пар.N]]
  <предложения языка Ассемблер (тело)>
ENDM
```

Формат макрокоманды:

```
<имя макроса> <фактические параметры через запятую и/или пробел>
```

Пример:
Описание макроса:

```
PRINTSTR MACRO STR
  MOV  DX, STR
  MOV  AH, 9
  INT  21H
ENDM
```
Вывод строки по смещению R.
Пример вызова:
```
LEA      BX, SOME_STR
PRINTSTR BX ; макрокоманда
```

Замечания по исполнению макроопределений:

- длина формальных параметров ограничена длиной строки
- нет ограничений, кроме физических
- макроопределения рекомендуется размещать в начале программы
- макрос можно определять внутри другого макроса, но тогда вложенный макрос будет доступен только после первого вызова внешнего макроса
- определение макроса с именем, которое ранее уже было использовано (переопределение макроса) затирает предыдущее определение
- макрос можно уничтожить с помощью PURGE <имя_макроса>{, <имя макроса>}
- макроопределения можно хранить в отдельном txt файле и включать в программу с помощью `INCLUDE`

40) Директива INCLUDE и LOCAL.

INCLUDE <имя файла без кавычек>

Вставляет содержимое файла в код (прям как в сях) <имяфайлабезкавычек> — ссылка на файл.

Пример:

```
.386
.model flat, c
.listall
include macros.inc
```

Директива Local

LOCAL v1, ..., vk (k >= 1)
Указывается, какие имена меток следует рассматривать как локальные.
В макрокомандах:
LOCAL идентификатор[, идентификатор].
В процедурах:
LOCAL элемент[, элемент].[= идентификатор]

```
M MACRO
        ...
```

```
        L:
                ...
ENDM
```

Если макрос будет вызван несколько раз, то в коде появятся несколько меток L, что недопустимо. LOCAL L заставляет макрогенератор заменять метки L на имена вида ??xxxx, xxxx 4-значное hex число. [??0000 - ??FFFF]. Макрогенератор запоминает номер, который он использовал последний раз при подстановке (??n) и в следующий раз подставит n+1 (??(n+1))
ДАННЫЙ ПРИМЕР ВЫВОДИТ НА ЭКРАН МЕСАГЕ БОКС

```
.MODEL FLAT, STDCALL
PrintName macro Name
local STR1, STR2, METKA
jmp METKA
STR1    DB «Программа»,0
STR2 DB «Меня зовут: &Name «,0
METKA:
PUSH  0
PUSH  OFFSET STR1
PUSH  OFFSET STR2
PUSH  0
CALL  MessageBoxA@16
endm
Init macro
EXTERN MessageBoxA@16:NEAR
endm
Init
.CODE
START:
PrintName <Лена>
PrintName <Таня>
RET
END START
```

41) Рекурсия в макроопределениях.
В теле макроопределения может содержаться вызов другого макроопределения или даже того же самого.

```
RECUR MACRO P
 MOV AX, P
 IF P
        RECUR %P-1
 ENDIF
```

Добавлено примечание ([7]): этого хватит?

Добавлено примечание ([8R7]): по моему мало но я хз

```
ENDM
```

Такой макрос будет вызывать сам себя, пока P не станет равно 0. Например, RECUR 3 развернется в

```
MOV AX, 3
MOV AX, 2
...
```

---

42) Параметры в макросах.

```
ИМЯ MACRO [форм.пар.1[,форм.пар.N]]
        <предложения языка Ассемблер (тело)>
 ENDM
```

Список Формальных Параметров - имена через запятую, используемые в предложениях тела макроопределения.
Пример вызова макроса с фактическими параметрами 1, 2:

```
MACRONAME 1,2
```

Фактические параметры перечисляются через запятую, список параметров не обязательно должен присутствовать. Запятые писать обязательно, даже в том случае, если 1 из параметров отсутствует. Также в качестве параметров можно использовать выражения.

```
;Пример макроса с параметром:
PRINT_STR MACRO STR ;STR параметр
        MOV AH,9
        MOV DX,STR
        INT 21H
ENDM

;Пример макроса без параметров
EXIT_APP MACRO
MOV AH,4CH
INT 21H
ENDM
```

Число фактических параметров может отличаться от формальных параметров в макрокоманде. Если фактических больше, чем формальных, то лишние фактические параметры игнорируются. Если фактических меньше, чем формальных, то формальные параметры, которые не соответствуют фактическим, заменяются на пустую строку.

Полный синтаксис формального аргумента следующий:

имя_формального_аргумента[:тип]
где тип может принимать значения:
REQ, которое говорит о том, что требуется обязательное явное задание фактического аргумента при вызове макрокоманды;
=<любая_строка> — если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении будет вставлено значение по умолчанию, соответствующее значению любая_строка.
Будьте внимательны: символы, входящие в любая_строка, должны быть заключены в угловые скобки.

43) Директивы условного ассемблирования и связанные с ними конструкции.

IFB <par> - условие истинно, если фактический параметр par не был задан в МКоманде (скобки <> обязательны)
IFNB <par> - условие истинно, если фактический параметр par был задан в МКоманде (скобки <> обязательны)
IFIDN <s1>,<s2> - условие истинно, если строки s1 и s2 совпадают (скобки <> обязательны)
IFDIF <s1>,<s2> - условие истинно, если строки s1 и s2 различаются (скобки <> обязательны)
IF www - условие истинно, если значение www<>0
IFE www - условие истинно, если значение www=0
IFDEF name - условие истинно, если имя name было описано выше
Также можно использовать с ELSE, как и обычный IF

```
IFDEF A
        ADD AX,A
ELSE
        EXITM
ENDIF
```

IFNDEF name - условие истинно, если имя name не было описано выше
IF1 - условие истинно 1-м шаге ассемблирования
IF2 - условие истинно 2-м шаге ассемблирования

44) Директивы IFB и IFNB в макроопределениях.

IFB <par>
Функция: Условие истинно, если фактический параметр par не был задан в МКоманде (макрокоманде) (скобки <> обязательны).
Цель - проверка наличия фактических параметров

IFNB <par>
Функция: Условие истинно, если фактический параметр par

был задан в МКоманде (скобки <> обязательны)
Цель - проверка наличия фактических параметров
Пример. МО вычисления суммы N элементов массива, адрес которого передается через BX. Результат сохраняется в R (по умолчанию - в регистре AX).
Сохранить BX и AX, если AX не представляет результат.

```
SM      MACRO    N,R
  LOCAL M
  IFNB <R>
      PUSH AX
  ENDIF
      PUSH BX
      XOR AX,AX
      MOV CX,N
M:    ADD AL,[BX]
      INC BX
      LOOP M
      POP BX
  IFNB <R>
      MOV R,AX
      POP AX
  ENDIF
      ENDM
;в сегменте данных
REZ     DW  0
ARRW    LABEL WORD
ARR     DB  1,2,3,4,5,6
;в сегменте кода
MIFB:   LEA BX,ARR
      SM  6,DX ; см листинг(81 вопрос)
      SM  6    ; см листинг
      SM       ; см листинг
```

Листинг

Для отладки можно посмотреть листинг. Директивы управления листингом (.LALL, .XALL .SALL, .NOLIST в MASM16 и те же или подобные в MASM32: .LISTALL, .LISTMACROALL) и составом макроопределений (INCLUDE, PUHGE).
Директива .LALL: далее включать в листинг программы полные макрорасширения (кроме комментариев после ;;).
Директива .XALL (.LISTMACROALL): далее включать в листинг программы только те предложения макрорасширений, которые генерируют коды и данные
Директива .SALL: далее не выводить тексты макрорасширений. в листинг.
Директива .NOLIST: далее вообще прекратить вывод листинга до появления

иной директивы.
Пример.

```
.LALL
 A3_1  MACRO
      NOP   ;  MO A3_1
      NOP   ;; MO A3_1
      ENDM
```

Для того, чтобы получить Имя.lst, окно командной строки следует настроить на папку проекта и выполнять трансляцию командой
ml.exe /c /Fl Имя.asm
где ml.exe – программа, выполняющая препроцессорную обработку и затем – только компиляцию (ключ /c) файла Имя.asm в объектный модуль(Имя.obj) и построение файла листинга (Имя.lst).
Также для отладки можно использовать средства вывода значений макропеременных и макроконстант с пояснениями (%ECHO).
Объединить вывод по %ECHO можно с исходным текстом модуля, но не с листингом, используя ключ /EP. А если ещё и перенаправить в файл name.EP:
ml.exe /c /Fl /EP name.asm> name.EP
то этот файл можно будет просматривать на одной из вкладок VS C++.
Сообщения препроцессора (? он сказал, что это про %ECHO)
(Об ошибках в макроопределениях и макрокомандах может быть сообщено в листинге)Для отладки можно посмотреть листинг. Директивы управления листингом (.LALL, .XALL .SALL, .NOLIST в MASM16 и те же или подобные в MASM32: .LISTALL, .LISTMACROALL) и составом макроопределений (INCLUDE, PUHGE).
Директива .LALL: далее включать в листинг программы полные макрорасширения (кроме комментариев после ;;).
Директива .XALL (.LISTMACROALL): далее включать в листинг программы только те предложения макрорасширений, которые генерируют коды и данные
Директива .SALL: далее не выводить тексты макрорасширений. в листинг.
Директива .NOLIST: далее вообще прекратить вывод листинга до появления иной директивы.
Пример.

```
.LALL
 A3_1  MACRO
      NOP   ;  MO A3_1
      NOP   ;; MO A3_1
      ENDM
```

Для того, чтобы получить Имя.lst, окно командной строки следует настроить на папку проекта и выполнять трансляцию командой
ml.exe /c /Fl Имя.asm
где ml.exe – программа, выполняющая препроцессорную обработку и затем –

только компиляцию (ключ /c) файла Имя.asm в объектный модуль(Имя.obj) и построение файла листинга (Имя.lst).
Также для отладки можно использовать средства вывода значений макропеременных и макроконстант с пояснениями (%ECHO).
Объединить вывод по %ECHO можно с исходным текстом модуля, но не с листингом, используя ключ /EP. А если ещё и перенаправить в файл name.EP:
ml.exe /c /Fl /EP name.asm> name.EP
то этот файл можно будет просматривать на одной из вкладок VS C++.
Сообщения препроцессора (? он сказал, что это про %ECHO)
(Об ошибках в макроопределениях и макрокомандах может быть сообщено в листинге)

45) Директивы IFIDN и IFDIF в макроопределениях.
IFIDN, — истинно,если строки(любойтекст) S1 и S2совпадают. (ifIDeNtical)
IFDIF , — истинно, если строки S1 и S2 различаются. (if DIFferent)
<,> не метасимволы, они обязательны.

```
SOMETHING MACRO
IFIDN <A>, <B>
        DISPLAY 'ЭТОНИКОГДАНЕВЫПОЛНИТСЯ'
        EXITM
ENDIF
…
 ENDM
```

46) Операции ;; % & < > ! в макроопределениях.
1. ;; — комментарий.
формат: ;; text
2. % — если требуется вычисление в строке некоторого константного выражения или передача его по значению в макрос.
формат: %ВЫРАЖЕНИЕ, например: K EQU 5, %K+2 ; =>7
3. ! — символ, идущий после данного знака будет распознан, как символ, а не как операция или директива.
формат: !c, например, если нам надо задать строку &A&: “ !&A!&”
4. &— склейка текста (склеить к параметру). Используется для задания модифицируемых идентификаторов и кодов операций.
формат: &par | &par& | par&, например "Something &A&: &B& something" в этой строке &A& и &B& будут заменены на фактические значения A и B. Алсо, это интерполяция строк.
5. <> — содержит часть текста программы (в макрорасширении заменяется ровно на то что в скобках). Внутри скобок последовательность любых символов.
формат: <text>, например <5+2>

```
DEBUGMSG MACRO WHERE, WHAT
        DBGMSG&WHERE DB “&WHAT& occured at &WHERE&$”
ENDM
; Вызов DEBUGMSG 135,<A huge error> развернется в DBGMSG135 DB “A huge error occured at 135$”
```

47) Блоки повторения REPT, IRP/FOR, IRPC/FORC, WHILE.

REPT константное_выражение

Предназначена для повторения некоторого блока операторов заданное количество раз. Отличие от WHILE – REPT автоматически каждую итерацию уменьшает константное выражение на 1, а в WHILE его нужно изменять вручную, но не обязательно на 1.

Пример. Целочисленное деление содержимого BX на 8:

```
        MOV BX,16
        REPT 3
          SHR BX,1  ;сдвиг вправо - деление на 2
        ENDM
```

IRP формальный_аргумент, <строка1, строка2…>

Повторяет блок операторов столько раз, сколько в списке в угловых скобках строк. На каждой итерации формальный аргумент принимает значение одной из строк, что дает возможность через форм. арг. к ней обратиться, пока список не исчерпается.

Пример. Определение переменных A0,A1,A2,A3 с начальными значениями 0,1,2,3 соответственно.

```
    IRP X,<0,1,2,3> ;параметры - числа
      A&X DB X
    ENDM
```

Пример из лабы:

```
IRP CUR_ITEM, <REG_LIST>
                IFIDN <CUR_ITEM>, <F>
                        PUSHF
                ELSE
                        PUSH CUR_ITEM
                ENDIF
ENDM
```

IRPC формальный_аргумент, <строка>

Принцип работы аналогичен IRP, но вместо списка строк передается одна строка и каждую итерацию формальному аргументу присваивается значение одного ее символа.

Пример:

```
IRPC X, ABC
        X DB ‘X’
```

ENDM
После развертки (макрорасширение):
A DB ‘A’
B DB ‘B’
C DB ‘C’ …

WHILE константное_выражение
Позволяет повторить некоторый блок операторов в зависимости от значения указанного в ней логического выражения. При выполнении этих директив макрогенератор будет вставлять в макрорасширение указанное количество строк, пока константное выражение не станет равно 0.
Пример.

```
j = 1
sum = 0
WHILE j LE 5
        sum = sum + j
        j = j + 1
ENDM
```

48) Директива EQU и = в MASM.
Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

метка equ выражение

```
truth equ 1
message1 equ 'Try again$'
var2 equ 4[si]
       cmp ax,truth ; cmp ax,1
       db message1 ; db 'Try again$'
       mov ax,var2 ; mov ax, 4[si]
```

Директива EQU чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде #define препроцессора языка C.

Директива = эквивалентна EQU, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена. Каждый ассемблер предлагает целый набор специальных предопределенных меток — это может быть текущая дата (@date или ??date), тип процессора (@cpu) или имя того или иного сегмента программы, но единственная предопределенная метка, поддерживаемая всеми рассматриваемыми нами ассемблерами, — $. Она всегда соответствует текущему адресу. Например, команда jmp $ выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

49) Директива TEXTEQU в MASM32.

Существует три формата директивы textequ:
имя TEXTEQU <текст>
имя TEXTEQU текстовый__макрос
имя TEXTEQU %константное_выражение

В первом случае символу присваивается указанная в угловых скобках <. . . > текстовая строка.
Во втором случае — значение заранее определенного текстового макроса.
В третьем случае — символической константе присваивается значение целочисленного выражения.

В приведенном ниже примере переменной prompt1 присваивается значение текстового макроса continueMsg:

```
continueMsg TEXTEQU <"Хотите продолжить (Y/N)?">
. data
prompt1 BYTE continueMsg
```

Символ, определенный с помощью директивы TEXTEQU, можно переопределить в
программе в любой момент.

50. Типы макроданных text и number.

Если при подстановке в команду или в директиву определения переменной макрос любого типа (text или number) всегда заменяется своим значением, то по %ECHO это справедливо только для типа text, а для типа number(числовые макропеременные, определённые директивой =, имеют тип number) необходимо выполнить преобразование к типу text. Проще всего это сделать стандартной функцией @CatStr(S) непосредственно в директиве %ECHO(S - макропеременная), сопроводив поясняющим текстом, например, так: %ECHO '@CatStr(%S)' = @CatStr(%S)
Пример макропроцедуры, вычисляющей S=P!:

```
MP_REC MACRO P,S
S=P
MOV EAX,P
IFE P EQ 1
S=1
ELSE
MP_REC P-1
S=S*P
ENDIF
ENDM
```

```
.CODE
MP_REC 3,S
MOV EAX,S
%ECHO '@CatStr(%S)' = @CatStr(%S)
```

51. Именованные макроконстанты MASM32

- **<ИМЯ_ИДЕНТИФИКАТОРА> EQU <ВЫРАЖЕНИЕ>:** используется для задания констант.

Например, **ASS EQU 10** вызовет последующую замену каждого слова ASS в коде на 10 транслятором. Замена “тупая”, то есть во всем тексте программы каждое вхождение того что слева будет заменено на то что справа. Выражения могут быть и текстовые, и числовые, и даже куски текста программы.

```
MacroConstant    EQU  123          ;; numeric equates

MacroVar         =  123            ;; numeric equates

MacroText        EQU  <string>     ;; text macro

MacroText        TEXTEQU <string>   ;; text macro
```

52. Макроимена, числовые и текстовые макроконстанты.

Макроимена или макроподстановка - символьное имя, заменяющее что-либо. Используется для простейшей замены: во всех местах, где встречается имя, вместо него будет помещен замещающий текст.

Символьная константа – идентификатор, которому поставлено в соответствие числовое выражение или строка.

Константы можно определить с помощью

Имя EQU <выражение> — присвоение некоторому выражению символического имени или идентификатора

Имя = <выражение> — присвоение некоторому выражению символического имени или идентификатора

Имя TEXTEQU <текст> — присвоение некоторому тексту символического имени или идентификатора

Имя TEXTEQU текстовый макрос — присвоение некоторому значению текстового макроса символического имени или идентификатора

Имя TEXTEQU %константное выражение — присвоение некоторому символьному эквиваленту значения константного выражения символического имени или идентификатора

Несмотря на внешнее и функциональное сходство, псевдооператоры EQU, =, TEXTEQU различаются следующим:

с помощью псевдооператора EQU идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки; псевдооператор

= может использоваться только с числовыми выражениями; TEXTEQU – только строки идентификаторы, определенные с помощью псевдооператоров = и TEXTEQU , можно переопределять в исходном тексте программы, определенные с использованием псевдооператора EQU — нельзя.

```
continueMsg TEXTEQU <"Хотите продолжить (Y/N)?">
. data
prompt1 BYTE continueMsg
```

53. Директивы echo и %echo

Директива echo выводит следующую после неѐ и пробела часть строки на экран Echo Hello World.I %echo вычисляет свой параметр, т.е. значение переменной T, и выводит его %echo @CatStr(A,+,B) OUT: A+B %echo вычисляет свой параметр, т.е. вызывает макрофункцию @CatStr, которая подставляет вместо себя результат своей работы, т.е. строку A+B, а %echo выводит эту строку.

54. Способы вывода значений макропеременных и макроконстант с пояснениями

Если при подстановке в команду или в директиву определения переменной макрос любого типа (text или number) всегда заменяется своим значением, то по %ECHO это справедливо только для типа text, а для типа number(числовые макропеременные, определённые директивой =, имеют тип number) необходимо выполнить преобразование к типу text. Проще всего это сделать стандартной функцией @CatStr(S) непосредственно в директиве %ECHO(S - макропеременная), сопроводив поясняющим текстом, например, так: %ECHO '@CatStr(%S)' = @CatStr(%S)
Пример макропроцедуры, вычисляющей S=P!:

```
MP_REC MACRO P,S
S=P
MOV EAX,P
IFE P EQ 1
S=1
ELSE
MP_REC P-1
S=S*P
ENDIF
ENDM
.CODE
MP_REC 3,S
MOV EAX,S
%ECHO '@CatStr(%S)' = @CatStr(%S)
```

55. Операции в выражениях, вычисляемых препроцессором MASM
ЕСЛИ У ВАС ЭТОТ ВОПРОС, БИЛЕТ МОЖНО ЗАМЕНИТЬ

1) Операция %

Формат: % www

Функция: Вычислить выражение перед представлением числа в символьной форме.

Пример переопределения макропроцедуры MP_REC добавлением оператора % для вычисления изменённого выражения при переходе к очередному шагу рекурсии

```
MP_REC MACRO P
MOV AX,P
IF P
MP_REC %(P-1) ;;перед записью в MPасш-ние вычислить P-1
ENDIF
ENDM
MP_REC 3 ; ! см. листинг(81 вопрос)
MOV EAX,S
%ECHO '@CatStr(%S)' = @CatStr(%S)
```

2) Операция <>

Формат : <txt>

Функция: Текст txt может содержать любые символы и рассматривается как одно целое. Если <txt> используется в качестве фактического параметра, то txt выносится в макрорасширение без изменений.

Пример. Использование макрокоманды в качестве фактического параметра макрокоманды

```
WZ MACRO REG,TELO
LOCAL M1,M2,M3
MOV CX,0
M1: CMP REG,0
LOOPNE M2
JMP M3
M2: TELO
JMP M1
M3:
ENDM
BLOK MACRO N
MUL N ;;DX:AX:=N*AX
DEC N ;;N:=N-1
ENDM
MOV BX,3 ; ВЫЧИСЛИТЬ ФАКТОРИАЛ ДЛЯ 3
MOV AX,1 ; ЗДЕСЬ БУДЕТ РЕЗУЛЬТАТ 6 = 3!
WZ ebx,<BLOK ebx ; - это МакроКоманда>
```

3) Операция &

Формат : &par | par& | &par&

Функция: Приклеить текст параметра par к тому, что стоит слева | справа | справа и слева.
Пример
```
PRIMER1 MACRO I,J,K
MOV A&I,J&L
INC E&K&X
ENDM
PRIMER1 H,B,C; ! см. Листинг ниже:(81 вопрос)
00000000 8A E3 1 MOV AH,BL
00000002 41 1 INC ECX
```
4) Операция !
Формат : !C
Функция: Считать C символом, а не знаком операции.
Пример. A!&B - это текст A&B, а не склейка B с A!.
5) Операция ;;
Формат : ;; txt
Функция: Текст txt не выносится в макрорасширение.
Арифметические операции +-*/
Логические операции OR, XOR, AND, NOT
Операции сдвига SHL, SHR
Операция смещения []

56. Подготовка ассемблерных объектных модулей средствами командной строки для использования в средах разработки консольных приложений на ЯВУ.

Несмотря на огромные возможности языков высокого уровня, иногда возникает необходимость применения Ассемблера в программах на ЯВУ (языки высокого уровня). Наиболее распространены два подхода:

1) ассемблерные вставки на встроенном ассемблере;
   Пример в VS.
   ```
   int _tmain(int argc, _TCHAR* argv[])
   {
           // что-то на C
           _asm
           {
                   ; что-то на asm
           }
   }
   ```
2) подключаются внешние ассемблерные модули: файл с процедурами пишется на внешнем ассемблере, компилируется в объектный файл OBJ, который подключается к проекту на ЯВУ.

Ассемблерные вставки применяются:
- для повышения быстродействия программы.
- для вызова команд, не используемых компилятором ЯВУ.

Masm.exe [flags] prog.asm,,,; Команда получения объектного модуля для последующего использования. Если их несколько можно написать bat-файл.

57. Добавление ассемблерных модулей в проект консольного приложения C.

Чтобы подключение ассемблерного модуля к программе на ЯВУ было корректным, он должен удовлетворять правилам, в соответствии с которыми создает программу компилятор ЯВУ. Так, должны совпадать имена сегментов, конвенции вызова и т.д

В вызывающем модуле cpp должен быть прописан прототип функции и указано соглашение по которым происходит вызов

Пример *Extern “C” void ppstart();*

Опция *extern* указывает компилятору, что тело функции следует искать в объектных файлах или библиотеках. Квалификатор "C" нужен, чтобы компилятор C++ не искажал имена функций

– Каркас ассемблерного модуля для подключения к программам Win32.

```
.486        ; 32-разрядные приложения
.model flat      ; в программах Win32 используется линейная модель
памяти (flat)
.bss ; в этом сегменте описываются неинициализированные данные
.const ; в этом сегменте описываются типизированные константы
.data ; в этом сегменте описываются переменные с начальными
значениями
 .code
PUBLIC Имя_процедуры  ; чтобы процедуру можно было вызывать из
     ; программы на ЯВУ, её нужно объявить экспортируемой
    ; реализация процедуры
Имя_процедуры proc near     ; все процедуры – ближние
     push  ebp   ; если процедура с параметрами, то в начале
процедуры
                                 ; нужно сохранить ebp в стеке,
либо скопировать его в другой
                                 ; регистр
     mov   ebp, esp   ; затем установить ebp = esp для обращения к
параметрам
     …           ; здесь тело процедуры
     pop   ebp   ; в конце восстанавливаем ebp
     ret   N     ; если процедура освобождает стек из-под параметров
сама, то  N – число байтов, которое занимает стековый кадр, N всегда
кратно 4.
Имя_процедуры    endp
end         ; конец модуля
```

58. Добавление ассемблерных модулей в проект консольного приложения PASCAL.
КАВО КАВО
см 57
(для Delphi)
Для подключения объектного файла в начало программы (между строками **Program** … или **Unit** … и строкой **Uses** …) добавляется директива:
{$LINK имя_файла.obj}
или {$L имя_файла.obj}
В секции описания процедур и функций помещается заголовок ассемблерной подпрограммы, а вместо тела пишется зарезервированное слово **external**. Так компилятор понимает, что тело подпрограммы нужно искать во внешнем OBJ-файле. Например
Procedure MyProc(X,Y:integer); stdcall; **external**;
Function MyFunc(X,Y:integer):integer; cdecl; **external**;
Если предполагается, что внешняя подпрограмма имеет переменное количество аргументов, то записывается директива **varargs**. Эту директиву можно использоваться только совместно с конвенцией *cdecl.*
59. Использование ассемблерных вставок в модулях на ЯВУ.

Для того, чтобы сделать ассемблерную вставку нужно написать __asm { текст вставки }.
Внутри ассемблерного блока можно обращаться по именам к переменным, функциям, процедурам и меткам. Переменные, объявленные внутри блока директивами DB, DW и т.п. будут размещены в сегменте кода, а не данных. Это нужно учесть, чтобы компилятор не стал исполнять их значения как машинные коды – это может привести к ошибке исполнения программы.
Если внутри ассемблерного блока нужны переходы по меткам, их делают локальными – имена таких меток следует начинать с символа @. Область действия локальной метки ограничена ассемблерным блоком.
Для вызова ассемблерной процедуры или метки необходимо написать ее прототип перед main используя соглашение ЯВУ. Пример: Extern “C” void ppstart().

60. Вызов подпрограммы C из ассемблерной в VS C++.
Надо объявить функцию C или C++ в директиве EXTRN и воспользоваться инструкцией call: CODESEG

EXTRN _cfuntion:proc

.....

call _cfunction

Здесь предполагается, что функция, названная _cfunction, существует в программе, подлежащей компоновке вместе с ассемблерным модулем. Если

функции требуются параметры, то простые параметры, такие как символы и целые числа часто передаются непосредственно в стек. Сложные переменные, такие как строки, структуры и множества, передаются посредством ссылок, т.е. по адресу. Пример:

```
void showscore( int thescore)
{
printf("\nThe score is: %d\n, thescore);
}
```

Чтобы вызвать функцию showscore из ассемблерного модуля, передавая значение переменной типа слова в качестве thescore, можно написать:

```
CODESEG
EXTRN showscore: proc
mov ax, 76 ; Присвоение score регистру
push ax ; Передача параметра в стек
call _showscore ; Вызов функции C
```

61) Передача глобальных данных, определённых в консольной программе VS C++, в ассемблерный модуль.

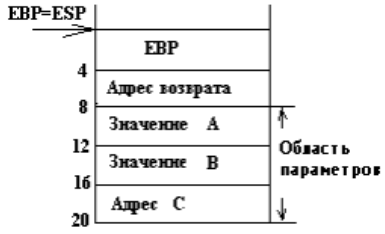


Рисунок 3.12– Структура стека для конвенции cdecl

Передача глобальных данных осуществляется с помощью стека. Ассемблерная подпрограмма, содержащая процедуру, может получать параметры, передаваемые в качестве параметров в функции при использовании стека. Первый параметр располагается на смещении SP + 8, каждый следующий параметр может быть получен путем извлечения из стека со смещением в 4, т.е. SP + 12, где SP есть указатель на вершину стека.

при вызове функции с прототипом:

```
void pro(int a, int b, int * c);
```

в стек сначала будет занесено смещение параметра с длиной 4 байта, затем b и a по четыре байта каждый, а затем, как обычно, ближний адрес возврата.

Адрес возврата - адресс метки, следующей за командой call.

Си позволяет ассемблеру объявлять новые глобальные переменные, доступные для всех модулей. Это достигается за счет размещения этих переменных в сегменте данных, отведенном для глобальных переменных, и описания его внутренней директивой PUBLIC

Пример смотрите в след вопросе, начинается с кода на с++

62) Передача глобальных данных, определённых в ассемблерном модуле в консольный модуль С.

Передача параметров из ассемблерного модуля в консольный модуль осуществляется с помощью регистра EAX. В нем может содержаться как значение, возвращаемое функцией, так и адрес данных.
Возвращаемые значения должны быть записаны в регистры:
char, short, enum, – в регистр AX;
int, указатель near – в регистр EAX;
float, double – в регистры TOS и ST(0) сопроцессора;
struct – записывается в память, а в регистр записывается указатель на нее. В качестве
исключения структуры длиной в 1 и 2 байта возвращаются в AX, а 4 байта – в EAX.
Си позволяет ассемблеру объявлять новые глобальные переменные, доступные для всех модулей. Это достигается за счет размещения этих переменных в сегменте данных, отведенном для глобальных переменных, и описания его внутренней директивой PUBLIC

```
extern "C" void __cdecl ADD1(int a,int b);
extern int d;
int main()
{
        int a,b;
        printf("Input a and b:\n");
         scanf("%d %d",&a,&b);
         ADD1(a,b);
        printf("d=%d.",d);
         getch();
        return 0;
        };
```

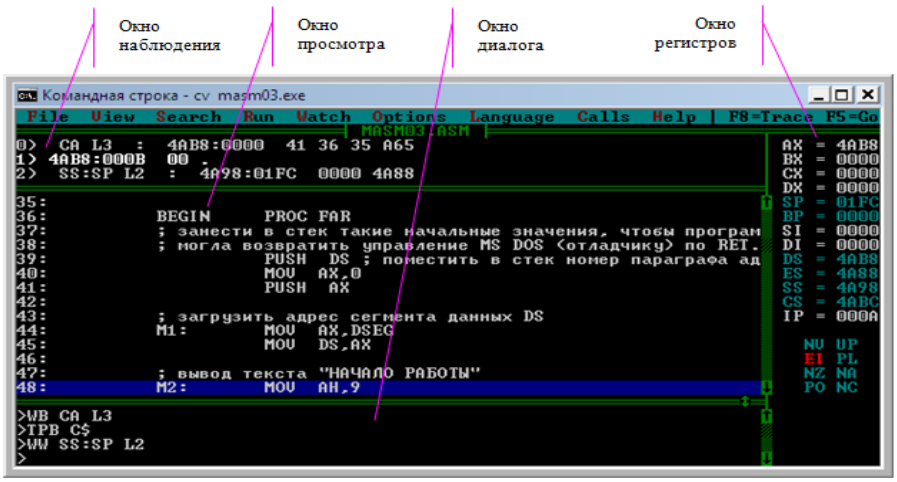
Текст программы на ассемблере:

```
 .586
.model flat
 .data
 public ?d@@3HA
```

```
?d@@3HA DD ?
.code
public ADD1
_ADD1 proc
        push EBP
        mov EBP,ESP
        mov EAX,[EBP+8]
        add EAX,[EBP+12]
        mov ?d@@3HA,EAX
        pop EBP
        ret
_ADD1 endp
end
```

63) Средства отладки в CodeView. Примеры.

Окно отладчика CV



Клавиша Alt – переход к работе с меню, далее ←, ↑, →, ↓; esc - выход из меню.

Управлять процессом отладки можно тремя способами: с помощью меню, путем использования функциональных и управляющих клавиш, а также с помощью команд отладчика. Часто одно и то же действие можно реализовать несколькими путями.
В отладчике возможно проведение пошаговой трассировки(f10,P) и трассировки с заходом в подпрограмму (f8, T).
Возможна установка точек условного и безусловного останова. Чтобы выполнить программу до конца или до точки останова, следует нажать клавишу f5(G).
Точки безусловного останова. (остановка происходит всегда или для определенного количества проходов)

BP [Адрес] [КолПрох] [”командыCV”]] – установить точку безусловного останова на команде по заданному параметром Адрес адресу с остановкой перед выполнением команды перед последним из заданного КолПрох числа проходов через неё и выполнить команды, перечисленные через ; в списке ”командыCV”.

Оператор наблюдения(дает возможность наблюдать за значениями переменных по ходу выполнения трассировки)
W[Тип] ДиапазонАдресов – установить в окне наблюдения оператор наблюдения - строку, отображающую текущие значения в ячейках указанного диапазона как данных заданного тпа.(W? - для выражений WP? - при истинности выражения)

ПРИМЕР

Установить наблюдение за переменными CA и строкой KA (см. рисунок)

>WB CA L3

0) CA L3 : 4AB8:0000 41 36 35 A65
Точки условного останова(остановка при изменении какого-либо условия)
TP[Тип] ДиапазонАдресов – установить в окне наблюдения точку трассировки -строку, отображающую условие останова: при изменении значения в заданном диапазоне(TP? для выражений)

ПРИМЕР

Установить условный останов при смене значения переменными C$ (см. рисунок)

>TPB C$

1) 4AB8:000B 00 .
По ходу выполнения строк программы в правое поле выводится текущее состояние регистров процессора и его флагов.

Не выходя из отладчика можно увидеть результат работы программы, если она выводит что-либо на экран(f4)

64) Средства отладки в средах разработки консольных приложений на ЯВУ.

В консольных приложениях в среде VS возможна отладка программы одновременно состоящей из ассемблерного модуля и консольного модуля на c++. Для пошаговой трассировки по ассемблерному модулю необходимо в запущенном режиме отладки и заранее поставленной точке останова, находящейся на вызове процедуры, исполняемой ассемблером, перейти к дизассемблерному коду, внутри которого можно пройтись построчно с помощью клавиши f8. Также возможны горячие клавиши для просмотра значений регистров и флагов и отслеживания адресов памяти. (Кнопки “Память” и “Регистры”).

65) Способы адресации. Термины и смысл.

В инструкциях программ операнды строятся из объектов, представляемых собой имена переменных и меток, представленных своим сегментом смещения (регистры, числа, символические имена с численным значением). Запись этих

объектов с использованием символов: [ ], +, - , . в различных комбинациях называют способами адресации.

1.Непосредственная адресация (значение): операнд задается непосредственно в инструкции и после трансляции входит в команду, как составная часть 1 и 2 байта.

MOV AL, 2 (1 BYTE)
MOV BX, OFFFFH (2 BYTES)

2. Регистровая адресация: значение операнда находится в регистре, который указывается в качестве аргумента инструкции.

MOV AX, BX

3. Прямая адресация: в инструкции используется имя переменной, значение которой является операндом.

MOV AX, X (в AX значение X)
MOV AX, CS:Y (в AX CS по смещению Y)

4. Косвенная регистровая адресация: в регистре, указанном в инструкции, хранится смещение (в сегменте) переменной, значение которой и является операндом. Имя регистра записывается в [ ].

MOV BX, OFFSET Z
…
MOV BYTE PTR[BX], ‘ASS’ ; в байт, адресованный регистром BX, запомнить ‘ASS’ в сегменте DS

MOV AX, [BX] ; в AX запомнить слово из сегмента, адрес BX со смещением ДS
MOV BX, CS:[SI] ; записать данные в BX, находящиеся в CS по смещению SI

5. Косвенная базово-индексная адресация (адресация с индексацией и базированием): смещение ячейки памяти, где хранятся значения операндов, вычисляется, как сумма значений регистров, записанных в качестве параметра в инструкции. [R1+R2],где R1∈{BX,BP} , R2∈{SI,DI}

MOV AX, [BX+DI]
MOV AX, [BP+SI]

Для BX по умолчанию сегменты из регистра DS. Для BP по умолчанию сегменты из регистра SS.

6. [Косвенная] базово-индексная адресация со смещением : (прямая с базированием и индексированием), отличается от п.5 тем, что ещё прибавляется смещение, которое может быть представлено переменной или ЦБЗ.

; [R1 + R2 +- ЦБЗ]
; [R1] [R2 +- ЦБЗ]
; имя [R1 + R2 +- АБС. ВЫРАЖЕНИЕ]
; имя [R1] [R2 +- АБС. ВЫРАЖЕНИЕ]

MOV AX, ARR[EBX][ECX * 2]

;ARR + (EBX) + (ECX) * 2
[имя] [база] [индекс [* масштаб]] [Абс. выражение]
Смещение операнда = смещение имени + смещение базы + значение индекса * масштаб + значение a

66) Связывание подпрограмм. Конвенции C, PASCAL, STDCALL, РЕГИСТРОВАЯ.

Соглашение	Параметры	Очистка стека	Регистры
Paskal	Слева направо	Процедура	--
C	Справа налево	Вызывающая программа	--
Fastcall (быстрый или регистровый вызов)	Слева направо	Процедура	EAX, EDX, ECX, далее стек
STDcall (стандартный вызов)	Справа налево	Процедура	--

Конвенция Pascal заключается в том, что параметры из программы на языке высокого уровня передаются в стеке и возвращаются в регистре AX/EAX, — это способ, принятый в языке PASCAL, — просто поместить параметры в стек в естественном порядке.
а) **Конвенция pascal**. Структура стека показана на рисунке 3.11.

```
. 386 .
model flat
.code
public ADD1
 ADD1 proc
        push EBP
        push EBP, ESP
        mov EAX, [EBP+16]
        add EAX, [EBP+12]
        mov EDX, [EBP+8]
        mov [EDX], EAX
        pop EBP
        ret 12 ; стек освобождает процедура
ADD1 endp
end
```

б) **Конвенция cdecl**. Структура стека показана на рисунке 3.12.

```
 . 386
 . model flat
 . code
 public ADD1
ADD1 proc
        push EBP
        push EBP, ESP
        mov EAX, [EBP+8]
        add EAX, [EBP+12]
        mov EDX, [EBP+16]
        mov [EDX], EAX
        pop EBP
        ret ;стек освобождает
        ;вызывающая программа
ADD1 endp
 end
```

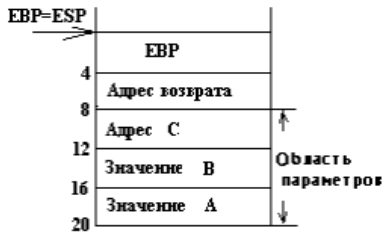


Рисунок 3.11 – Структура стека для конвенции pascal

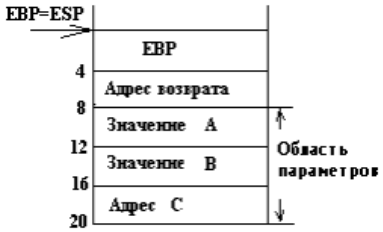


Рисунок 3.12– Структура стека для конвенции cdecl

в) **Конвенция register**

```
 .386 Размещение параметров:
 .model flat                    ;первый параметр A в регистре EAX
 .code                                  ;второй параметр B в регистре EDX
 public ADD1                    ;третий параметр адрес C в регистре ECX
ADD1 proc
        add EDX,EAX
        mov [ECX],EDX
        ret ; стек освобождает вызывающая программа
ADD1 endp
 end
```

г) **Конвенция stdcall**. Структура стека показана на рисунке 3.13.

```
.386
.model flat
.code
public ADD1
ADD1 proc
        push EBP
        push EBP, ESP
        mov EAX, [EBP+8]
        add EAX, [EBP+12]
        mov EDX, [EBP+16]
        mov [EDX], EAX
        pop EBP
        ret 12 ; стек освобождает процедура
ADD1 endp
end
```

Рисунок 3.13 – Структура стека для конвенции stdcall

67) Многострочные макроопределения. Формальные и фактические параметры.

ИМЯ MACRO [форм.пар.1[,форм.пар.N]]
        <предложения языка Ассемблер (тело)>
ENDM

СписОК Формальных Параметров - имена через запятую, используемые в предложениях тела макроопределения.

Пример вызова макроса с фактическими параметрами 1, 2:

MACRONAME 1,2

Фактические параметры перечисляются через запятую, список параметров не обязательно должен присутствовать. Запятые писать обязательно, даже в том случае, если 1 из параметров отсутствует. Также в качестве параметров можно использовать выражения.

```
;Пример макроса с параметром:
PRINT_STR MACRO STR ;STR параметр
        MOV AH,9
        MOV DX,STR
```

```
        INT 21H
ENDM

;Пример макроса без параметров
EXIT_APP MACRO
MOV AH,4CH
INT 21H
ENDM
```

Число фактических параметров может отличаться от формальных параметров в макрокоманде. Если фактических больше, чем формальных, то лишние фактические параметры игнорируются. Если фактических меньше, чем формальных, то формальные параметры, которые не соответствуют фактическим, заменяются на пустую строку.

Полный синтаксис формального аргумента следующий:
имя_формального_аргумента[:тип]
где тип может принимать значения:
REQ, которое говорит о том, что требуется обязательное явное задание фактического аргумента при вызове макрокоманды;
=<любая_строка> — если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении будет вставлено значение по умолчанию, соответствующее значению любая_строка.
Будьте внимательны: символы, входящие в любая_строка, должны быть заключены в угловые скобки.
Многострочным макроопределением является все, что со словом macro и больше чем с 1 предложением языка ассемблер
Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.

68) Многострочные макроопределения и директивы условного ассемблирования

Многострочным макроопределением является все, что со словом macro и больше чем с 1 предложением языка ассемблер
Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.

Структуры условного ассемблирования IF W, где W, W1, W2,… вычисляемые препроцессором выражения:

```
        1) IF.W. Директива IF условного ассемблирования
        . . . . . \ Блок предложений ассемблера, выполняемых
```

. . . . . / при истинности условия
ENDIF Директива условного ассемблирования
2) IF.W1. Директива IF условного ассемблирования
. . . . . \ Блок1 предложений ассемблера, выполняемых
. . . . . / при истинности условия
ELSEIF W2 Директива условного ассемблирования
. . . . . \ Блок2 предложений ассемблера, выполняемых
. . . . . / , если условие ложно
ELSE Директива условного ассемблирования
. . . . . \ Блок3 предложений ассемблера, выполняемых
. . . . . / , если условие ложно
ENDIF Директива условного ассемблирования
Блоков ELSEIF может быть несколько или не быть совсем.

Другие директивы IF условного ассемблирования:
IFB <par> - условие истинно, если фактический параметр par не был задан в МКоманде (скобки <> обязательны)
IFNB <par> - условие истинно, если фактический параметр par был задан в МКоманде (скобки <> обязательны)
IFIDN <s1>,<s2> - условие истинно, если строки s1 и s2 совпадают (скобки <> обязательны)
IFDIF <s1>,<s2> - условие истинно, если строки s1 и s2 различаются (скобки <> обязательны)
IF www - условие истинно, если значение www<>0
IFE www - условие истинно, если значение www=0
IFDEF nam - условие истинно, если имя nam было описано выше
IFNDEF nam - условие истинно, если имя nam не было описано выше
IF1 - условие истинно 1-м шаге ассемблирования
IF2 - условие истинно 2-м шаге ассемблирования

Директива IFB.
IFB <par>
Функция: Условие истинно, если фактический параметр par
не был задан в МКоманде (макрокоманде) (скобки <> обязательны).
Цель - проверка наличия фактических параметров

Директива IFNB.
IFNB <par>
Функция: Условие истинно, если фактический параметр par
был задан в МКоманде (скобки <> обязательны)
Цель - проверка наличия фактических параметров
Пример. МО вычисления суммы N элементов массива, адрес
которого передается через BX. Результат сохраняется
в R (по умолчанию - в регистре AX).

Сохранить BX и AX, если AX не представляет результат.
МО можно располагать и внутри, и вне сегментов

```
SM MACRO N,R
        LOCAL M
        IFNB <R>
                PUSH AX
        ENDIF
                PUSH BX
                XOR AX,AX
                MOV CX,N
        M:      ADD AL,[BX]
                INC BX
                LOOP M
                POP BX
        IFNB <R>
                MOV R,AX
                POP AX
        ENDIF
                ENDM
                ;в сегменте данных
REZ DW 0
ARRW LABEL WORD
ARR DB 1,2,3,4,5,6
;в сегменте кода
MIFB: LEA BX,ARR
        SM 6,DX ; см листинг(81 вопрос)
        SM 6 ; см листинг
        SM ; см листинг
```

Директива IFIDN.
IFIDN <s1>,<s2>
Функция: Условие истинно, если строки s1 и s2 совпадают
Цель - проверка фактических параметров

```
MM MACRO R1,R2 T
LOCAL L
IFDIF <R1>,<R2> ;;R1 и R2 – разные регистры
CMP R1,R2
IFIDN<T>,<MAX> ;; T=MAX ?
JGE L ;;да – поместить JGE L в макрорасширение
ELSE
JLE L ;; нет поместить JLE L
ENDIF
```

```
    MOV R1,R2
    L:
    ENDIF
    ENDM
```

Директива IFDIF.

IFDIF <s1>,<s2>

Функция: Условие истинно, если строки s1 и s2 не совпадают

Цель - проверка фактических параметров

Пример. МО вычисления суммы N элементов массива, адрес которого передается через BX, а результат сохраняется в R (по умолчанию - в регистре AX). Если первый параметр будет задан символом 0, то результат также должен быть равен 0 *

```
SUM MACRO N,R
        IFIDN <0>,<N>
                IFB <R>
                        MOV AX,0
                ELSE
                        MOV R,0
                ENDIF
        ELSE
                SM N,R
        ENDIF
ENDM
```

SUM 0;результат IFIDN <0>,<N> положительный

d EQU 0 ;определение макроконстанты со значением 0

SUM O ;результат IFIDN <0>,<N> отрицательный

Директива IF.

IF W

Функция: Условие истинно, если значение выражения W<>0.

Директива IFE W

IFE W

Функция: Условие ложно, если значение выражения www=0.

! Директивы IF и IFE могут использоваться для анализа выражений вне и внутри макроопределений, в том числе для анализа параметров.

Пример. МО вычисления суммы N элементов массива, адрес которого передается через BX, а результат сохраняется в R (по умолчанию - в регистре AX). Если первый параметр будет иметь значение <= 0, то результат должен быть равен 0

```
SUMM MACRO N,R
```

```
        IF N LE 0
                IFB <R>
                        MOV AX,0
                ELSE
                        MOV R,0
                ENDIF
        ELSE
                SM N,R
        ENDIF
ENDM
```

Директива IFDEF.

IFDEF nam

Функция: Условие истинно, если имя nam было определено ранее.

```
        ifndef sw ;если sw не определено, то выйти из макроса
        EXITM
        else ;иначе — на вычисление
        mov cl,n
        ife sw
        sal x,cl ;умножение на степень 2 сдвигом влево
        else
        sar x,cl ;деление на степень 2 сдвигом вправо
        endif
        endif
```

Директива IFNDEF

IFNDEF nam

Функция: Условие ложно, если имя nam не было описано выше.

! Директивы IFDEF и IFNDEF могут использоваться для анализа выражений вне и внутри макроопределений, в том числе для анализа параметров.

Пример. МО вычисления суммы N элементов массива, адрес которого передается через BX, а результат сохраняется в R (по умолчанию - в регистре AX). Если первый параметр будет иметь значение <= 0, то результат должен быть равен 0

```
NMIN = 3
        IFNDEF NMAX
                IFDEF NMIN
                        SUMM NMIN,REZ
                ELSE
                        SUMM 6,REZ
                ENDIF
        ELSE
                SUMM NMAX,REZ
```

```
ENDIF
```

69) Многострочные макроопределения и директивы генерации ошибок

Многострочным макроопределением является все, что со словом macro и больше чем с 1 предложением языка ассемблер

Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.

Директива	№ ошибки	Текст сообщения на экране
.ERRB <s>	94	Forced error - пустая строка
.ERRNB <s>	95	Forced error - непустая строка
.ERRIDN <s1>,<s2>	96	Forced error - строки одинаковые
.ERRDIF <s1>,<s2>	97	Forced error - строки разные
.ERRE www	90	Forced error - выражение = 0
.ERRNZ www	91	Forced error - выражение <> 0
.ERRNDEF nam	92	Forced error - имя не определено
.ERRDEF nam	93	Forced error - имя определено
.ERR1	87	Forced error - pass 1
.ERR2	88	Forced error - pass 2
.ERR	89	Forced error

Пример использования.

```
 SUMMA    MACRO N,R,M
    .ERRB <M> ;; не задан массив
    IF TYPE M NE 1
      .ERR ;тип массива не BYTE
      IF1
        %OUT ОШИБКА: тип массива не BYTE
      ENDIF
      EXITM
    ENDIF
    LEA BX,M
    SM  N,R ; SM-макроопределение непосредственно суммирования элементов корректного массива
    ENDM
```

70) Многострочные макроопределения и операции в них.

Многострочным макроопределением является все, что со словом macro и больше чем с 1 предложением языка ассемблер

Макроопределения — специальным образом оформленная последовательность предложений языка ASM. Код управления которой ASM (точнее его часть порождает — макрогенератор) макрорасширение макрокоманд.

Операция %
Формат: % www
Функция: Вычислить выражение перед представлением числа в символьной форме.
Пример переопределения макропроцедуры MP_REC (см. выше вопрос 55 мб) добавлением оператора % для вычисления изменённого выражения при переходе к очередному шагу рекурсии

```
MP_REC MACRO P
        MOV AX,P
    IF P
        MP_REC %(P-1) ;;перед записью в MPасш-ние вычислить P-1
    ENDIF
ENDM
MP_REC 3 ; ! см. листинг(81 вопрос)
MOV EAX,S
```

%ECHO '@CatStr(%S)' = @CatStr(%S)
Результат остался прежним (S равно 6), как видно из исходного текста, объединённого с текстом вывода по %ECHO '@CatStr(%S)' = @CatStr(%S) в файле P1.EP:

```
.CODE
P1:
    S =1
    S =1
    S =1
    MPR %1-1,S
    S = S*1
    S = S*2
    S = S*3
MOV EAX,S
ECHO '@CatStr(%S)' = 6
```

Правда, исчезли детали алгоритма макроопределения, остались (пере)определения макропеременных (но без вычислений), макрокоманды, вывод по %ECHO и то, с чем в конечном счете должен работать компилятор.
Ещё пример на использование оператор % при вычислении N! но с использованием макрофункции:

```
MFREC MACRO N
%ECHO @CATSTR(N)
    IF N GT 1
        EXITM %MFREC(%(N-1))*(N)
    ELSE
        EXITM %1
    ENDIF
```

```
ENDM
.CODE
M TEXTEQU %3
R TEXTEQU %MFREC(M) ; R TEXTEQU %MFREC(3)
MOV ECX, R ; MOV ECX, MFREC(3)
```

файле P1.EP мы увидим:

```
M TEXTEQU %3
ECHO 3
ECHO 2
ECHO 1
EXITM %MFR(%(1-1))*(1)
R TEXTEQU %6
MOV ECX, 6
```

В файле P1.lst мы увидим:

```
= 3 M TEXTEQU %3
1 %ECHO @CATSTR(M)
1 IF M GT 1
2 %ECHO @CATSTR(2)
2 IF 2 GT 1
3 %ECHO @CATSTR(1)
3 IF 1 GT 1
3 EXITM %MFR(%(1-1))*(1)
3 ELSE
3 EXITM %1
2 EXITM %MFR(%(2-1))*(2)
1 EXITM %MFR(%(M-1))*(M)
= 6 R TEXTEQU %MFR(M)
00000005 B9 00000006 MOV ECX, R
```

Операция <>

Формат : <txt>

Функция: Текст txt может содержать любые символы и рассматривается как одно целое. Если <txt> используется в качестве фактического параметра, то txt выносится в макрорасширение без изменений.

Пример. Использование макрокоманды в качестве фактического параметра макрокоманды

```
WZ MACRO REG,TELO
        LOCAL M1,M2,M3
        MOV CX,0
M1: CMP REG,0
        LOOPNE M2
        JMP M3
```

```
M2: TELO
        JMP M1
M3:
        ENDM
BLOK MACRO N
        MUL N ;;DX:AX:=N*AX
        DEC N ;;N:=N-1
        ENDM

        MOV BX,3 ; ВЫЧИСЛИТЬ ФАКТОРИАЛ ДЛЯ 3
        MOV AX,1 ; ЗДЕСЬ БУДЕТ РЕЗУЛЬТАТ 6 = 3!
WZ ebx,<BLOK ebx ; - это МакроКоманда>
```

Операция &
Формат : &par | par& | &par&
Функция: Приклеить текст параметра par к тому, что стоит слева | справа | справа и слева.
Пример

```
PRIMER1 MACRO I,J,K
        MOV A&I,J&L
        INC E&K&X
        ENDM
        PRIMER1 H,B,C; ! см. Листинг ниже:
00000000 8A E3 1 MOV AH,BL
00000002 41 1 INC ECX
```

Операция !
Формат : !C
Функция: Считать С символом, а не знаком операции.
Пример. A!&B - это текст A&B, а не склейка B с A!.
Операция ;;
Формат : ;; txt
Функция: Текст txt не выносится в макрорасширение.

71) Макропроцедуры. Определения и вызовы.

Макропроцедура (МП) - многострочное макроопределение.
Структура:

```
ИмяМО MACRO СписФормПар     ; заголовок МО
      ; . . . . . .
      ; предложения языка ассемблера ; тело  МО
      ; . . . . . .
          ENDM              ; конец текста МО
```

ИмяМО - имя макроопределения (МП),

СписФормПар - имена через запятую, используемые в предложениях тела макроопределения. СписФорПар может отсутствовать.
Завершение работы МП –при достижении директивы ENDM, или по директиве EXITM без параметра.
Формат макрокоманды макропроцедуры (КМП)
ИмяМП СписФактПар
ИмяМО - имя МП, к которой происходит обращение,
СписФактПар - список фактических параметров через запятую, заменяющих при построении макрорасширения соответствующие формальные параметры. Элементами списка могут быть имена, строки, выражения.
Пример. МП для вывода на экран символа, заданного константой, или в регистре, или в ОП (в оперативной памяти)

```
PutCh   MACRO Ch    ;; Заголовок МО
    MOV DL,Ch ;; Ch содержит символ, который
    MOV AH,2  ;; будет выведен на экран
    INT 21H
ENDM
```

Вызов:

```
X1    DB    'A'
```

PutCh X1 ; mov DL, X1 - будет выведен символ A

72) Рекурсивные макропроцедуры.

Пример макропроцедуры, вычисляющей S=P!.

```
MP_REC  MACRO P,S
        MOV EAX,P
        IFE P EQ 1
                S=1
        ELSE
                MP_REC %(P-1)
                S=S*P
        ENDIF
ENDM
```

73) Макрофункции. Определения и вызовы.

Макрофункции (структура как и у макропроцедуры)
Макропроцедура (МП) - многострочное макроопределение.
Так же как и при обработке макропроцедуры, при обработке макрофункции препроцессор подставляет вместо ее имени набор ассемблерных команд, единственное отличие состоит в том, что она всегда возвращает константу(целую или строковую) в вызывающею ее программу с помощью директивы EXITM, которая немедленно прекращает дальнейшую обработку макроопределения. При вызове макрофункции список ее аргументов следует заключить в круглые скобки.
Формат макрокоманды макрофункции (КМФ)
ИмяМФ(СписФактПар)
ИмяМО - имя МП, к которой происходит обращение,

СписФактПар - список фактических параметров через запятую, заменяющих при построении макрорасширения соответствующие формальные параметры. Элементами списка могут быть имена, строки, выражения.
Пример определения и вызова макрофункции:

```
; определение
IS_DEFINE MACRO symbol
        IFDEF symbol
                EXITM 1
        ELSE EXITM 0
        ENDIF
ENDM
; вызов
IF IS_DEFINE (RealMode)
        MOV ax, dseg
        MOV ds, ax
ENDIF
```

74) Рекурсивные макроопределения.

В теле макроопределения может содержаться вызов другого макроопределения или даже того же самого.
Рекурсивная макрофункция:

```
MFREC MACRO N
                IF N GT 1
                        EXITM %MFREC(%(N-1))*(N)
                ELSE
                        EXITM %1
                ENDIF
        ENDM
```

Рекурсивная макропроцедура:

```
MP_REC  MACRO P, S
        MOV EAX, P
        IFE P EQ 1
                S=1
        ELSE
                MP_REC %(P-1)
                S=S*P
        ENDIF
ENDM
```

75) Числовые макроконстанты. Определение и примеры использования.

Задаются при помощи EQU

Директива EQU присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

метка equ выражение

```
truth equ 1
message1 equ 'Try again$'
var2 equ 4[si]
        cmp ax,truth ; cmp ax,1
        db message1 ; db 'Try again$'
        mov ax,var2 ; mov ax, 4[si]
```

Макроконстанта определяется с помощью ключевого слова EQU(от слова equivalence – равносильно). Например

```
One equ 2*3
Username equ 1024/8
```

Понимать это нужно так: макрогенератор (препроцессору) строку «One» везде заменит на результат вычисления выражения 2*3, а строку «Username» - на результат вычисления выражения 1024/8. Единожды определив такие числовые макроконстанты, переопределить их потом нельзя. Замены имён макроконстант на строки-значения производятся в исходном тексте в строках, расположенных ниже определения. Поэтому макроконстанты нужно определять в самом начале asm-файлов.

76) Числовые макропеременные. Определение и примеры использования.

Макропеременные могут быть только целочисленными. Они создаются на период компиляции с помощью операции = (знак равенства). Значение в процессе компиляции можно переопределять сколько угодно раз. Например

```
MyVar = 10
REPT 4
        MyVar = MyVar + 2
ENDM
```

Еще пример из 12 лабы.

```
Ca = 8 ;Ca - числовая переменная
IFNB <FUNCTION_PARAMS>
        IRP cur_p, <FUNCTION_PARAMS>
                cur_p EQU [EBP + Ca]
                Ca = Ca + 4
        ENDM
ENDIF
```

77) Константные выражения. Операции, вычисляемые препроцессором в выражениях.

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры equ и " = " (равно).

Эти псевдооператоры предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, макроассемблер подставит вместо него соответствующее выражение.

Примеры:

Предположим, что в сегменте данных закодирована следующая директива EQU:

```
TIMES EQU 10
```

Имя, в данном случае TIMES, может быть представлено любым допустимым в Ассемблере именем. Теперь, в какой бы команде или директиве не использовалось слово TIMES Ассемблер подставит значение 10.

```
.equ DDRB  = 0x17  ;присвоение имени DDRB значения 0x17
.equ PORTB = DDRB + 1 ;присвоение имени PORTB значения 0x18
```

В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.
Операции:
1) Операция %
Формат: % www
Функция: Вычислить выражение перед представлением числа в символьной форме.
Пример переопределения макропроцедуры MP_REC добавлением оператора % для вычисления изменённого выражения при переходе к очередному шагу рекурсии

```
MP_REC MACRO P
MOV AX,P
IF P
MP_REC %(P-1) ;;перед записью в MPасш-ние вычислить P-1
ENDIF
ENDM
MP_REC 3 ; ! см. листинг (81 вопрос)
MOV EAX,S
%ECHO '@CatStr(%S)' = @CatStr(%S)
```

2) Операция <>
Формат : <txt>
Функция: Текст txt может содержать любые символы и рассматривается как одно целое. Если <txt> используется в качестве фактического параметра, то txt выносится в макрорасширение без изменений.
Пример. Использование макрокоманды в качестве фактического параметра макрокоманды

```
WZ MACRO REG,TELO
LOCAL M1,M2,M3
MOV CX,0
M1: CMP REG,0
LOOPNE M2
```

```
JMP M3
M2: TELO
JMP M1
M3:
ENDM
BLOK MACRO N
MUL N ;;DX:AX:=N*AX
DEC N ;;N:=N-1
ENDM
MOV BX,3 ; ВЫЧИСЛИТЬ ФАКТОРИАЛ ДЛЯ 3
MOV AX,1 ; ЗДЕСЬ БУДЕТ РЕЗУЛЬТАТ 6 = 3!
WZ ebx,<BLOK ebx ; - это МакроКоманда>
```

3) Операция &
Формат : &par | par& | &par&
Функция: Приклеить текст параметра par к тому, что стоит слева | справа | справа и слева.
Пример

```
PRIMER1 MACRO I,J,K
MOV A&I,J&L
INC E&K&X
ENDM
PRIMER1 H,B,C; ! см. Листинг ниже(81 вопрос):
00000000 8A E3 1 MOV AH,BL
00000002 41 1 INC ECX
```

4) Операция !
Формат : !C
Функция: Считать C символом, а не знаком операции.
Пример. A!&B - это текст A&B, а не склейка B с A!.
5) Операция ;;
Формат : ;; txt
Функция: Текст txt не выносится в макрорасширение.
Арифметические операции +-*/
Логические операции OR, XOR, AND, NOT
Операции сдвига SHL, SHR
Операция смещения []

78) Текстовые макропеременные. Способы определения.

Директива equ задаèт или числовые макроконстанты (их далее уже нельзя менять), или текстовые макропеременные (их далее можно менять), например:

```
V equ 100
V equ 200; ОШИБКА
V equ abc; ОШИБКА
W equ abc
W equ def; НЕТ ОШИБКИ
W equ 300; НЕТ ОШИБКИ
```

W equ ijk; ОШИБКА

Текстовой макро может быть любой строкой не более 255 символов. Поскольку он имеет статус переменной, его значение может быть изменено.

```
;; Текстовой макро (Макропеременная строкового типа)
WASM EQU <One Wonderful Wonderful ASM>
;; Текстовой макро (Макропеременная строкового типа)
WASM_RU TEXTEQU <>
```

Существует три формата директивы textequ:
имя TEXTEQU <текст>
имя TEXTEQU текстовый__макрос
имя TEXTEQU %константное_выражение

В первом случае символу присваивается указанная в угловых скобках <. . . > текстовая строка.
Во втором случае — значение заранее определенного текстового макроса.
В третьем случае — символической константе присваивается значение целочисленного выражения.

- **<ИМЯ_ИДЕНТИФИКАТОРА> EQU <ВЫРАЖЕНИЕ>:** используется для задания констант. Например, **ASS EQU 10** вызовет последующую замену каждого слова ASS в коде на 10 транслятором. Замена “тупая”, то есть во всем тексте программы каждое вхождение того что слева будет заменено на то что справа. Выражения могут быть и текстовые, и числовые, и даже куски текста программы.

79) Блоки повторения REPT, FOR, FORC, WHILE.

REPT константное_выражение
Предназначена для повторения некоторого блока операторов заданное количество раз. Отличие от WHILE – REPT автоматически каждую итерацию уменьшает константное выражение на 1, а в WHILE его нужно изменять вручную, но не обязательно на 1.
Пример. Целочисленное деление содержимого BX на 8:

```
MOV BX,16
REPT 3
  SHR BX,1  ;сдвиг вправо - деление на 2
ENDM
```

IRP формальный_аргумент, <строка1, строка2…> (тоже самое что и **for**)
Повторяет блок операторов столько раз, сколько в списке в угловых скобках строк. На каждой итерации формальный аргумент принимает значение одной

из строк, что дает возможность через формальные аргументы к ней обратиться, пока список не исчерпается.
Пример. Определение переменных A0,A1,A2,A3 с начальными значениями 0,1,2,3 соответственно.

```
    IRP X,<0,1,2,3> ;параметры - числа
      A&X DB X
    ENDM
```

Пример из лабы:

```
IRP CUR_ITEM, <REG_LIST>
                IFIDN <CUR_ITEM>, <F>
                        PUSHF
                ELSE
                        PUSH CUR_ITEM
                ENDIF
ENDM
```

IRPC формальный_аргумент, <строка> (тоже самое что и **forc**)
Принцип работы аналогичен IRP, но вместо списка строк передается одна строка и каждую итерацию формальному аргументу присваивается значение одного ее символа.
Пример:

```
IRPC X, ABC
        X DB 'X'
ENDM
```

После развертки (макрорасширение):

```
A DB 'A'
B DB 'B'
C DB 'C' …
```

WHILE константное_выражение
Позволяет повторить некоторый блок операторов в зависимости от значения указанного в ней логического выражения. При выполнении этих директив макрогенератор будет вставлять в макрорасширение указанное количество строк, пока константное выражение не станет равно 0.
Пример.

```
j = 1
sum = 0
WHILE j LE 5
        sum = sum + j
        j = j + 1
ENDM
```

80) Стандартные макрофункции обработки строк @CATSTR и @SUBSTR.

@CatStr(string1 [[, string2...]]), макрофункция

Возвращает строку, созданную объединением строк параметров функции.

Пример:

```
%echo @CatStr(<my>,var)
```

Вывод:Myvar

@SubStr(string, position [[, length]]) макрофункция

Возвращает подстроку строки string, начиная с позиции, указанной в параметре position (отсчёт начинается с 1). Если необязательный параметр length задан, он ограничивает размер возвращаемой строки. Параметр length не может быть меньше нуля, и не может быть строкой.

Пример:

```
%echo @SubStr(1234567890,2)
%echo @SubStr(1234567890,1,5)
```

Вывод:

234567890

12345

Чтобы вывести значение целочисленной макропеременной необходимо воспользоваться макрофункцией @CatStr(), и перед аргументом указать оператор %.

%echo @CatStr(%T)

OUT: 6

%T вычисляет (извлекает) целое значение из макропеременной T, а макрофункция @CatStr преобразует это число в строку символов для вывода, поэтому выводится 6, а не 06.

Стандартная макрофункция @SubStr и директива SubStr могут порождать множество подстрок типа text с числовыми и нечисловыми значениями, причём при одних и тех же значениях параметров директива SubStr определит (переопределит) макропеременную типа text, а макрофункция @SubStr вернёт значение, совпадающее со значением макропеременной. Следующий вложенный цикл позволяет перебрать и вывести значения подмножеств строки 1234

```
j=1
while j LE 4
        i=1
        WHILE i le 5-j
                names SubStr <ABCD>,i,j
                %ECHO 'names SubStr <ABCD>,i,j' out: names , i, j
                %ECHO '@SubStr (ABCD,i,j)' out: @SubStr (ABCD,i,j)
                i=i+1
```

```
        endM
        j=j+1
endm
```

Этот код представляет вывод всех подстрок строки ABCD (A, B, C,D,AB, BC, CD, ABC, BCD, ABCD), последовательно присваиваемых директивой макропеременной и возвращаемых макрофункциеей при изменяющихся значениях второго параметра (i) – начала подстроки и третьего параметра (j) – длина подстроки. Следующие строки представляют результаты вывода по %ECHO:

```
'names SubStr <ABCD>,i,j' out: A , 1,1
'@SubStr (ABCD,i,j)' out: A
'names SubStr <ABCD>,i,j' out: B , 2,1
'@SubStr (ABCD,i,j)' out: B
'names SubStr <ABCD>,i,j' out: C , 3,1
'@SubStr (ABCD,i,j)' out: C
'names SubStr <ABCD>,i,j' out: D , 4,1
'@SubStr (ABCD,i,j)' out: D
'names SubStr <ABCD>,i,j' out: AB , 1,2
'@SubStr (ABCD,i,j)' out: AB
'names SubStr <ABCD>,i,j' out: BC , 2,2
'@SubStr (ABCD,i,j)' out: BC
'names SubStr <ABCD>,i,j' out: CD , 3,2
'@SubStr (ABCD,i,j)' out: CD
'names SubStr <ABCD>,i,j' out: ABC , 1,3
'@SubStr (ABCD,i,j)' out: ABC
'names SubStr <ABCD>,i,j' out: BCD , 2,3
'@SubStr (ABCD,i,j)' out: BCD
'names SubStr <ABCD>,i,j' out: ABCD , 1,4
'@SubStr (ABCD,i,j)' out: ABCD
```

81) Инструменты отладки макросредств. Окно командной строки, листинг, сообщения препроцессора.

(Еще к инструментам относятся средства отладки в среде VS - вопрос 64)

Для отладки можно посмотреть листинг. Директивы управления листингом (.LALL, .XALL .SALL, .NOLIST в MASM16 и те же или подобные в MASM32: .LISTALL, .LISTMACROALL) и составом макроопределений (INCLUDE, PUHGE).

Директива .LALL: далее включать в листинг программы полные макрорасширения (кроме комментариев после ;;).

Директива .XALL (.LISTMACROALL): далее включать в листинг программы только те предложения макрорасширений, которые генерируют коды и данные

Директива .SALL: далее не выводить тексты макрорасширений. в листинг.

Директива .NOLIST: далее вообще прекратить вывод листинга до появления иной директивы.

Пример.

```
.LALL
 A3_1  MACRO
     NOP   ;  MO A3_1
     NOP   ;; MO A3_1
     ENDM
```

Для того, чтобы получить Имя.lst, окно командной строки следует настроить на папку проекта и выполнять трансляцию командой
ml.exe /c /Fl Имя.asm
где ml.exe – программа, выполняющая препроцессорную обработку и затем – только компиляцию (ключ /c) файла Имя.asm в объектный модуль(Имя.obj) и построение файла листинга (Имя.lst).
Также для отладки можно использовать средства вывода значений макропеременных и макроконстант с пояснениями (%ECHO).
Объединить вывод по %ECHO можно с исходным текстом модуля, но не с листингом, используя ключ /EP. А если ещё и перенаправить в файл name.EP:
ml.exe /c /Fl /EP name.asm> name.EP
то этот файл можно будет просматривать на одной из вкладок VS C++.
Сообщения препроцессора (? он сказал, что это про %ECHO)
(Об ошибках в макроопределениях и макрокомандах может быть сообщено в листинге)

82) Применение макросредств при определении переменных.

Макропеременные могут быть только целочисленными. Они создаются на период компиляции с помощью операции = (знак равенства). Значение в процессе компиляции можно переопределять сколько угодно раз. Например

```
MyVar = 10
REPT 4
     MyVar = MyVar + 2
ENDM
Еще пример из 12 лабы.
Ca = 8 ;Ca - числовая переменная
IFNB <FUNCTION_PARAMS>
     IRP cur_p, <FUNCTION_PARAMS>
          cur_p EQU [EBP + Ca]
          Ca = Ca + 4
     ENDM
ENDIF
```

Директива equ задаèт или числовые макроконстанты (их далее уже нельзя менять), или текстовые макропеременные (их далее можно менять), например:

```
V equ 100
V equ 200; ОШИБКА
V equ abc; ОШИБКА
```

```
W equ abc
W equ def; НЕТ ОШИБКИ
W equ 300; НЕТ ОШИБКИ
W equ ijk; ОШИБКА
```

Текстовой макро может быть любой строкой не более 255 символов. Поскольку он имеет статус переменной, его значение может быть изменено.

```
;; Текстовой макро (Макропеременная строкового типа)
WASM EQU <One Wonderful Wonderful ASM>
;; Текстовой макро (Макропеременная строкового типа)
WASM_RU TEXTEQU <>
```

Существует три формата директивы textequ:
имя TEXTEQU <текст>
имя TEXTEQU текстовый__макрос
имя TEXTEQU %константное_выражение

В первом случае символу присваивается указанная в угловых скобках <. . . > текстовая строка.
Во втором случае — значение заранее определенного текстового макроса.
В третьем случае — символической константе присваивается значение целочисленного выражения.

- **<ИМЯ_ИДЕНТИФИКАТОРА> EQU <ВЫРАЖЕНИЕ>:** используется для задания констант. Например, **ASS EQU 10** вызовет последующую замену каждого слова ASS в коде на 10 транслятором. Замена “тупая”, то есть во всем тексте программы каждое вхождение того что слева будет заменено на то что справа. Выражения могут быть и текстовые, и числовые, и даже куски текста программы.

83) Применение макросредств при создании фрагментов кода.

При написании любой программы на ассемблере, возникают некоторые трудности: повторяемость некоторых идентичных или незначительно отличающихся участков программы; необходимость включения в каждую программу участков кода, которые уже были использованы в других программах; ограниченность набора команд.
Для решения этих проблем в языке ассемблер существуют макросредства.
Отличие макросредств от подпрограмм:

- подпрограммы требуют подготовки для вызова (это бывает дольше, чем сам код)

- если в программе много макровыводов, то увеличивается размер кода
- параметры при записи макрокоманды должны быть известны уже на стадии ассемблирования
- в макрокомандах можно пропускать параметры

Пример:
Описание макроса:

```
PRINTSTR MACRO STR
  MOV  DX, STR
  MOV  AH, 9
  INT  21H
ENDM
```

Вывод строки по смещению R.
Пример вызова:

```
LEA      BX, SOME_STR
PRINTSTR BX ; макрокоманда
```

Применение макросредств экономит время выполнения программы. Поэтому в программах критических по времени следует применять макросредства, а если необходимо экономить память следует применять процедуры.
Если в повторяющемся участке кода много команд (т.е. большой фрагмент) лучше описать его как процедуру. Если же небольшую группу команд описать процедурой, то число вспомогательных команд по ее вызову и передаче параметров станет сравнимым с числом команд самой процедуры, ее время выполнения станет на много больше.
Большие участки кода рекомендуется описывать как процедуры, а маленькие - как макроопределения.

Еще одно отличие использования макросредств и процедур заключается в том, что параметрами процедур могут быть только операнды команд, а параметрами макрокоманд могут быть любые последовательности символов, в том числе и сами команды.
(Ну можно сюда пример закинуть)
Пример. Вывод строки на экран

```
outstr   macro  str       ; форм. параметр - имя строки
         push ax
         mov ah,09h
         lea dx, str
         int 21h
         pop ax
endm
```