Массивы, структуры

Тема сегодняшней лекции – массивы и структуры.
Мы уже разобрали, как работать с различными директивами определения данных и с ассемблерными операторами, которые эти определения осуществляют, сегодня же разберем то, как работать с массивами и разнообразными структурами данных.
Дадим формальное определение:
массив *- структурированный тип данных, состоящий из некоторого числа элементов одного типа.*
Для того чтобы разобраться в возможностях и особенностях обработки массивов в программах на ассемблере, нужно ответить на следующие вопросы:
Как описать массив в программе?
Как инициализировать массив, то есть как задать начальные значения его элементов?
Как организовать доступ к элементам массива?
Как организовать массивы с размерностью более одной?
Как организовать выполнение типовых операций с массивами?
Специальных средств описания массивов в программах ассемблера, конечно, нет. Мы используем директивы определения данных.
Сначала поговорим о способах адресации.
Большинство команд процессора вызываются с аргументами, которые в ассемблере принято называть операндами. Например: команда сложения содержимого регистра с числом требует задания двух операндов — содержимого регистра и числа.
Регистровая адресация
Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. Для этого в тексте программы указывается название соответствующего регистра, например: команда, копирующая в регистр AX содержимое регистра BX, записывается как
mov ax, bx
Непосредственная адресация
Некоторые команды (все арифметические, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы. Например: команда
mov ax,2
помещает в регистр AX число 2.
Прямая адресация
Если у операнда, располагающегося в памяти, известен адрес, то его можно использовать. Если операнд - слово, находящееся в сегменте, на который указывает ES, со смещением от начала сегмента 0001, то команда
mov ax,es:0001
поместит это слово в регистр AX. В реальных программах для задания статических переменных обычно используют уже рассмотренные нами директивы определения данных, которые позволяют ссылаться на статические переменные не по адресу, а по имени. Тогда, если в сегменте, указанном в ES, была описана переменная word_var размером в слово, можно записать ту же команду как
mov ax,es:word_var
В таком случае ассемблер сам заменит слово word_var на соответствующий адрес. Если селектор сегмента данных находится в DS, то имя сегментного регистра при прямой адресации можно не указывать, DS используется по умолчанию. Прямая адресация иногда называется адресацией по смещению.
Адресация отличается для реального и защищенного режимов. В реальном (так же как и в режиме V86) смещение всегда 16-битное. Это значит, что ни непосредственно указанное смещение, ни результат сложения содержимого разных регистров в более сложных методах адресации не могут превышать границ слова.
При работе в Windows, DOS4G, PMODE и в других ситуациях, когда программа будет запускаться в защищенном режиме, смещение не должно превышать границ двойного слова.
Косвенная адресация
По аналогии с регистровыми и непосредственными операндами адрес операнда в памяти также можно не указывать, а хранить в любом регистре. До процессора 80386 для этого можно было

использовать только BX, SI, DI и BP. Например, следующая команда помещает в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение - в BX:
mov ax,[bx]
Как и в случае с прямой адресацией, DS используется по умолчанию, но не всегда: если смещение берут из регистра BP, то в качестве сегментного регистра применяется SS. В реальном режиме можно свободно работать со всеми 32-битными регистрами, надо только следить, чтобы их содержимое не превышало границ 16-битного слова.

```
MOV SI, OFFSET X

MOV AL, BYTE PTR [SI]
…….
X DB 10 DUP(?)       ;[0; 9]
```

Вы помните оператор [], который обозначает индексное обращение – также напоминаю, что использовать его можно только с индексными регистрами (SI и DI, source index и destination index) и ещё двумя модификаторами, о которых позднее. В данном примере показан один из вариантов использования данных регистра и оператора для обращения к элементу массива X. В данном случае si – не индекс массива, а адрес элемента в общей структуре памяти. Такая адресация называется косвенной базовой (или регистровой).
Также к регистру SI в ней можно добавить смещение: mov al, byte ptr [si+1], но это уже следующий метод адресации:
Адресация по базе со сдвигом
Теперь скомбинируем два предыдущих метода адресации. Следующая команда
mov ax,[bx+2]
помещает в регистр AX слово, которое есть в сегменте, указанном в DS, со смещением на два больше, чем число из BX. Так как слово занимает ровно 2 байта, эта команда поместила в AX слово, непосредственно следующее за тем, которое было в предыдущем примере. Такая форма адресации используется в тех случаях, когда в регистре находится адрес начала структуры данных, а доступ надо осуществить к какому-нибудь ее элементу. Еще один вариант применения адресации по базе со сдвигом — доступ из подпрограммы к параметрам, переданным в стеке, используя регистр BP в качестве базы и номер параметра в качестве смещения, что детально рассмотрим позднее.
Другие допустимые формы записи этого способа адресации:
mov ax,[bp]+2
mov ax,2[bp]
До процессора 80386 в качестве базового регистра разрешалось использовать только BX, BP, SI или DI и сдвиг мог быть только байтом или словом (со знаком). С помощью этого метода разрешается организовывать доступ к одномерным массивам байтов: смещение соответствует адресу начала массива, а число в регистре - индексу элемента массива, который надо считать. Очевидно, что, если массив состоит не из байтов, а из слов, придется умножать базовый регистр на два, а если из двойных слов - на четыре.

```
XOR SI, SI

MOV AL, BYTE PTR X[SI]
…….
X DB 10 DUP(?)       ;[0; 9]
```

Другой пример. Здесь X берется за базовый адрес, от которого уже отсчитывается смещение в виде регистра SI. Такая индексация называется косвенной индексной. Её тоже можно использовать со смещением: mov al, byte ptr x[si+1].

```
адрес(X[i]) = X+(type X)*i
```

При использовании косвенной индексной адресации индексный регистр автоматически умножается на тип X (напоминаю, тип – размер элемента). Регистр SI в данном примере называется модификатором адреса, а полный адрес – исполнительным.

XOR SI, SI

MOV AL, BYTE PTR X[BX][SI]
…….
A DB N DUP(M DUP(?))
Регистр BX – модификатор
Регистр SI – модификатор
Полный адрес – исполнительный

$$Аисп = (A + [BX] + [SI]) \bmod 2^{16}$$

Если массив многомерный (объявлять такие можно благодаря вложенности конструкции DUP, которую мы разбирали на прошлых лекциях), то обращение к его элементам производится так же, как в С. Все необходимые модификации адреса (умножение значения регистра на размерность типа) будут осуществлены самим ассемблером. Здесь точно так же индексные регистры называются модификаторами адреса, а полный адрес – исполнительным.

MOV AX, A[BX]

Команда будет работать не с адресом A, а с исполнительным адресом (эффективным)
Замена адреса на исполнительный – модификация адреса. Регистр, участвующий при модификации - модификатор

$$Аисп = (A + [BX]) \bmod 2^{16}$$

При использовании индексного оператора в косвенной индексной адресации команда работает не с базовым адресом, а с исполнительным (или эффективным). В процессе трансляции ассемблер преобразовывает адрес в исполнительный, используя модификатор.

BX, BP	**SI, DI**
A[BX][SI]	A[BP][SI]
A[BX][DI]	A[BP][DI]

- ✓ Регистры модификаторы: BX, BP, SI, DI
- ✓ При модификации адреса по двум регистрам, регистры-модификаторы должны быть из разных групп!
- ✓ При модификации адреса по одному регистру, регистром- модификатором может быть любой регистр из четырёх

ADD A[SI], 5 ;si = 100

$$Аисп=A+[SI]=A+100.$$

Косвенные ссылки

MOV [BX], 5 => MOV 0[BX], 5
При отсутствии базового адреса (то есть когда используется косвенная базовая/регистровая адресация) он считается равным нулю. В таком случае при отсутствии оператора уточнения типа будут возникнуть ошибки неопределенного типа данных.
MOV BYTE PTR [BX], 5

Вторая строчка будет верно обработана компилятором.

Заключать в квадратные скобки регистры – не модификаторы запрещено!

A :	Аисп=A	
E[M] :	Аисп=(E+[M]) mod 2^{16}	(M: BX,BP,SI,DI)
E[M1][M2] :	Аисп=(E+[M1]+[M2]) mod 2^{16}	(M1: BX,BP; M2: SI,DI)

В операторе индексации могут участвовать только BX, BP, SI и DI. Любой другой регистр в них вызовет ошибку транслятора.
Load Effective Address
MOV BX, OFFSET array
……..
Array db N dup(?)
LEA BX, array
……..
Array db N dup(?)

```
LEA r16,m
LEA r32,m
```

Инструкция LEA – Load Effective Address – загружает исполняемый (эффективный) адрес участка памяти в регистр без необходимости OFFSET-а. Участок памяти может быть задан не только в виде начального адреса массива, но и с помощью адресации. С помощью команды LEA можно вычислить адрес переменной, которая описана сложным способом адресации (например, по базе с индексированием, что часто используется при работе с массивами и строками). Оператор OFFSET позволяет определить смещение только при компиляции, и в отличие от него команда LEA может сделать это во время выполнения программы. Хотя в остальных случаях обычно вместо LEA используют **MOV** и OFFSET, то есть
LEA ПРИЁМНИК, ИСТОЧНИК
это то же самое, что и
MOV ПРИЁМНИК, offset ИСТОЧНИК
С помощью LEA можно вычислить адрес переменной, которая описана сложным методом адресации, например по базе с индексированием. Если адрес - 32-битный, а регистр-приемник - 16-битный, старшая половина вычисленного адреса теряется, если наоборот, приемник - 32-битный, а адресация- 16-битная, то вычисленное смещение дополняется нулями.

XLAT/XLATB
LEA BX, Letter
Mov al, 5
XLATB ;al = ‘F’
Letter db ‘ABCDEF’

LEA BX, Letter

```
Mov al, 11
XLATB                    ;al = 'A'
Letter db '0123456789ABCDEF'
```

BX – адрес таблицы
AL – номер элемента таблицы

Команда XLATB применяется для перекодирования байта по таблице. Эта команда удобна когда нужно преобразовать один байт в другой байт и есть точная таблица соответствия. Для того чтобы воспользоваться командой XLATB нужно сначала поместить в регистр (BX) адрес таблицы соответствия, в регистр AL - индекс в этой таблице, а в регистр ES - адрес сегмента, в котором таблица находится. После этого можно вызвать команду XLATB. После выполнения команды в регистре AL будет содержаться значение из таблицы, на которую указывает BX, по индексу, который в AL хранился.
Работа команды XLAT может быть записана в виде следующей формулы:
AL = [BX][AL]
В качестве аргумента для XLAT можно указывать имя таблицы, но эта информация процессором не используется и служит только в качестве комментария; XLATB короче, если комментарий не нужен, и не требует аргумента.
Например, можно написать следующий вариант преобразования шестнадцатеричного числа в ASCII-код соответствующего ему символа:

```
mov al, 0Ch
mov bx, offset htable
xlatb
```

если в сегменте данных, на который указывает регистр ES, было записано

```
htable db "0123456789ABCDEF"
```

```
<имя процедуры> PROC <параметр>
                <тело процедуры>
<имя процедуры> ENDP
```

Процедура (подпрограмма) — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого приоритета и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Процедуру можно определить и как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Функция – процедура, способная возвращать некоторое значение.
Процедуры ценны тем, что могут быть активизированы в любом месте программы. Процедурам могут быть переданы некоторые аргументы, что позволяет, имея одну копию кода в памяти и изменять ее для каждого конкретного случая использования, подставляя требуемые значения аргументов.
Процедурой в ассемблере является все то, что в других языках называют подпрограммами, функциями, процедурами и т. д. Ассемблер не накладывает на процедуры никаких ограничений - на любой адрес программы можно передать управление командой CALL, и оно вернется к вызвавшей процедуре, как только встретится команда RET. Такая свобода выражения легко может приводить к трудночитаемым программам.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP.
Последвательность вызова процедуры проста: передать параметры, передать управление на начало процедуры и по её окончанию вернуться в место её вызова и продолжить выполнение программы.
Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);
- в конце (после команды, возвращающей управление операционной системе);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы;
- в другом модуле.

Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив `PROC` и `ENDP`, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы `END`, обозначающей конец программы:

```
…
.code
myproc proc near
ret
myproc endp
start proc
call myproc
…
start endp
end start
```

В этом фрагменте после загрузки программы в память управление будет передано первой команде процедуры с именем `start`.
Объявление имени процедуры в программе равнозначно объявлению метки, поэтому директиву `PROC` в частном случае можно рассматривать как форму определения метки в программе.
Размещение процедуры в конце программы предполагает, что последовательность команд, ограниченная директивами `PROC` и `ENDP`, будет размещена после команды, возвращающей управление операционной системе.

```
…
.code
start proc
call myproc
…
start endp
myproc proc near
ret
```

```
myproc endp
end start
```
Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае требуется предусмотреть обход тела процедуры, ограниченного директивами `PROC` и `ENDP`, с помощью команды безусловного перехода `jmp`:

```
…
.code
start proc
jmp ml
myproc proc near
ret
myproc endp
ml:
…
start endp
end start
```
Последний вариант расположения описаний процедур — в отдельном модуле — предполагает, что часто используемые процедуры выносятся в отдельный файл.

Поскольку имя процедуры обладает теми же атрибутами, что и метка в команде перехода, то обратиться к процедуре можно с помощью любой команды условного или безусловного перехода. Но благодаря специальному механизму вызова процедур можно сохранить информацию о контексте программы в точке вызова процедуры. Под контекстом понимается информация о состоянии программы в точке вызова процедуры. В системе команд микропроцессора есть две команды, осуществляющие работу с контекстом. Это команды `call` и `ret`:

- call **ИмяПроцедуры** — вызов процедуры (подпрограммы).
- ret **число** — возврат управления вызывающей программе.

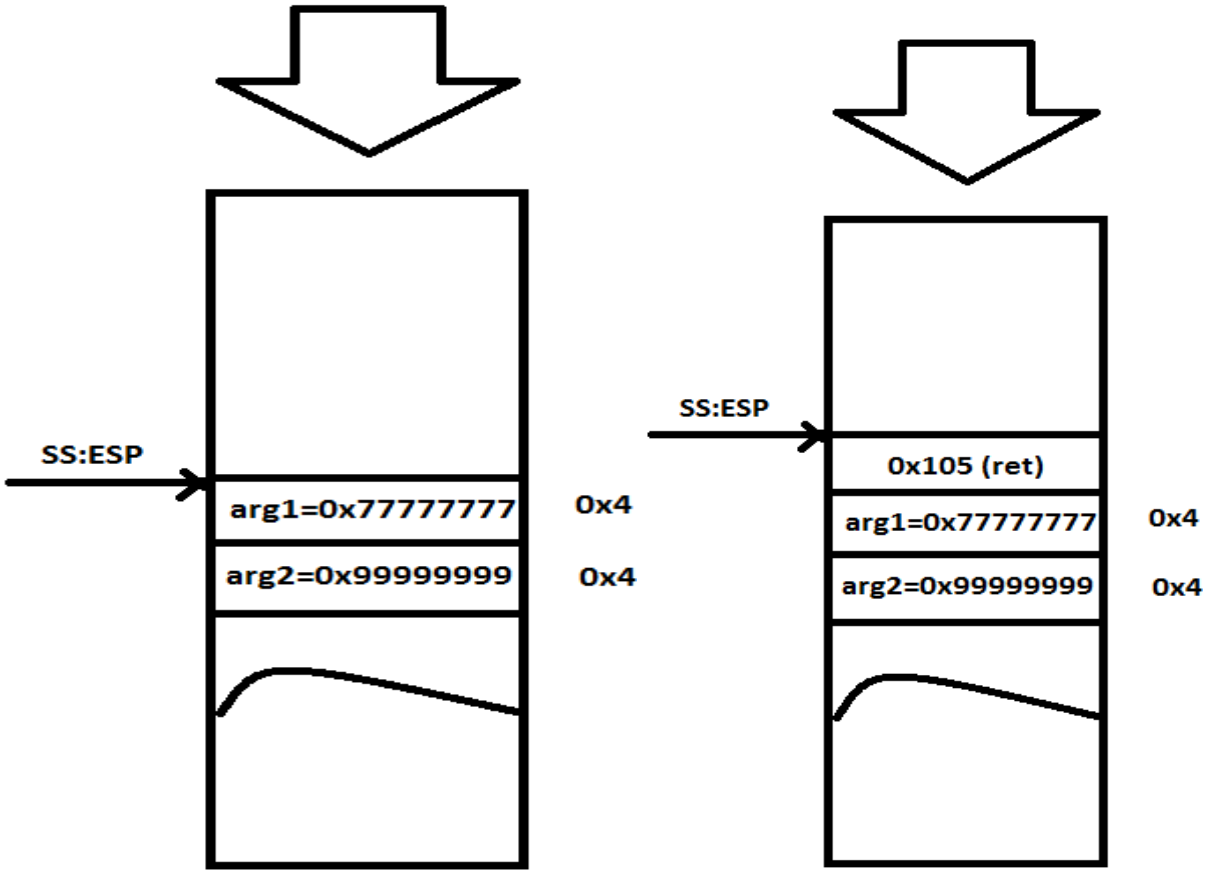
число — необязательный параметр, обозначающий количество байт, удаляемых из стека при возврате из процедуры.

Рассмотрим псевдопрограмму

```
void test(DWORD arg1,DWORD arg2)
{
DWORD var1 = 0x11111111;
DWORD var2 = 0x22222222;
        Eax = var1;
        Ebx = var2;
        Push data
        Pop ecx
}
//main
Test(0x77777777, 0x99999999);

push arg2;      //arg2 = 0x99999999;
push arg1;      //arg1 = 0x77777777;
100: call test
105: xor eax,eax
```

Стадии стека



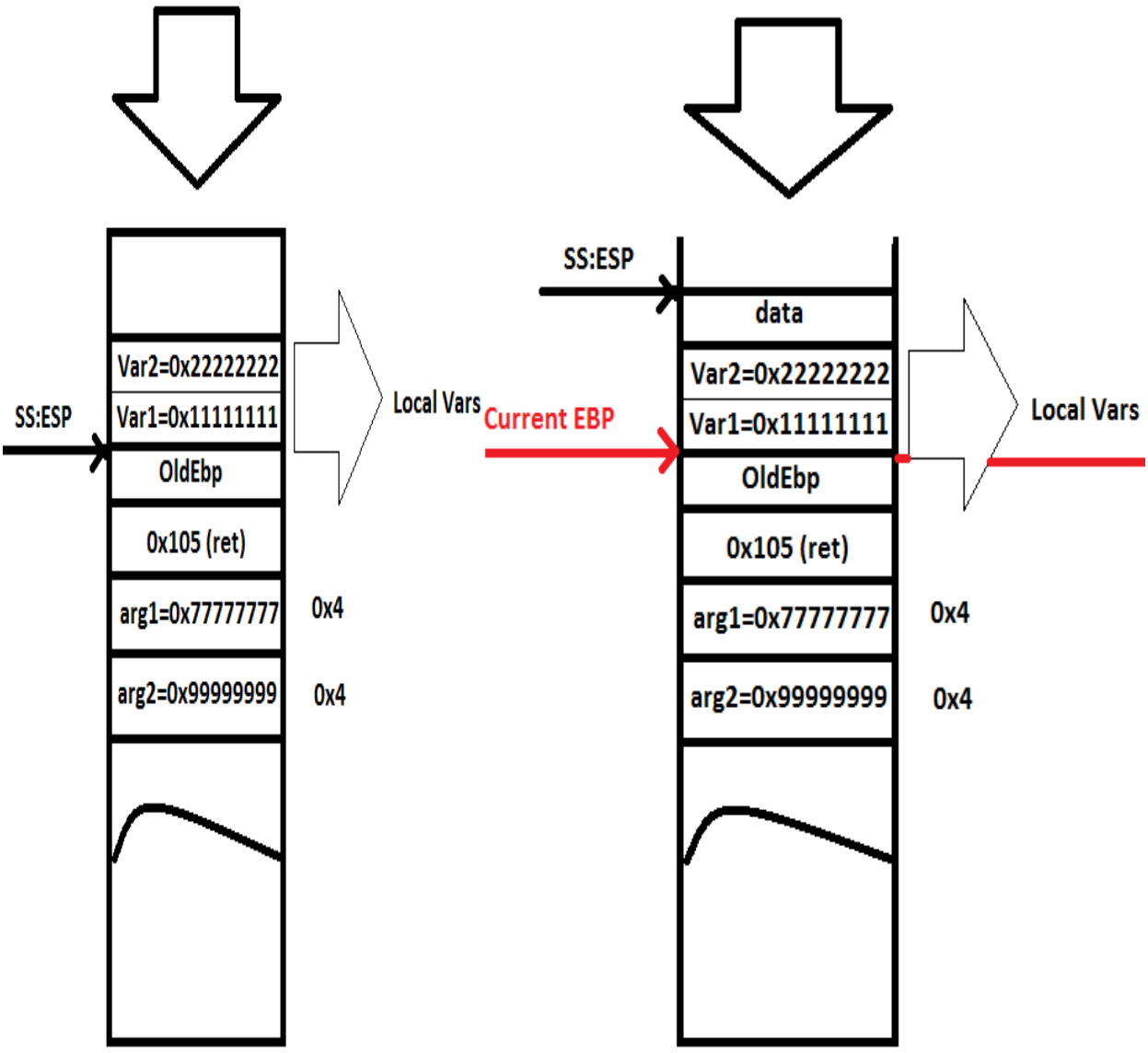
Пролог функции стандартного вызова

```
push ebp;          //сохраняю содержимое ebp
Mov ebp, esp       //запоминаю вершину стека
Sub esp, 8         //резервирую место в стеке
                   // для лок-х переменных
                   //размером 8 байт
```

Инициализация локальных переменных

.....

Стадии стека



Доступ к переменным через bp

ax = Var1: mov ax, WORD ptr[bp-2];

bx = Var2: mov bx, WORD ptr[bp-4];

ax = arg1: mov ax, DWORD ptr[bp+4]

bx = arg2: mov bx, WORD ptr[bp+6].

Программные сегменты

- Повторное использование кода
- Логическое разграничение кода
- Параметризация кода

Физически ***сегмент*** представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Каждая программа содержит 3 типа сегментов:

- Сегмент кодов – содержит машинные команды для выполнения. Обычно первая выполняемая команда находится в начале этого сегмента, и операционная система передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента кодов (CS) адресует данный сегмент.
- Сегмент данных – содержит данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.
- Сегмент стека — содержит адреса возврата как для программы (для возврата в операционную систему), так и для вызовов подпрограмм (для возврата в главную программу), а также используется для передачи параметров в процедуры. Регистр сегмента стека (SS) адресует данный сегмент. Адрес текущей вершины стека задается регистрами SS:SP.

Разумеется, ассемблер позволяет изменять устройство программы как угодно - помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Упрощенные директивы сегментации

Для задания сегментов в тексте программы можно пользоваться упрощенными директивами:

- `.CODE` — для указания начала сегмента кода;
- `.DATA` — для указания начала сегмента данных;
- `.STACK` — для указания начала сегмента стека.

Однако использование упрощенных директив сегментации не позволяет создать более трех сегментов для одной программы.

Стандартные директивы сегментации

Наряду с упрощенными директивами сегментации может также использоваться стандартная директива `SEGMENT`, которая определяет начало любого сегмента. Синтаксис:

```
ИмяСегмента SEGMENT Параметры
…
ИмяСегмента ENDS
```
Директива `ENDS` определяет конец сегмента.
Параметры - readonly, выравнивание, тип, разряд, класс. Все пять являются необязательными.

Если в программе есть несколько сегментов с одним именем, то они соединяются в один

```
A SEGMENT
        A1 db 400h DUP(?)
        A2 db 8
A ENDS
B SEGMENT
        B1 dw ?
        b2 dd ?
B ENDS
C SEGMENT
        ASSUME ES: A, DS:B,  CS: C
L:      mov ax, A
        mov bx, B
C ENDS
```

Параметры	**Значения**
Атрибут выравнивания сегмента	BYTE , WORD, DWORD, PARA, PAGE, MEMPAGE
Атрибут комбинирования сегментов	PRIVATE, PUBLIC, COMMON, AT xxxx, STACK
Атрибут класса сегмента (тип класса)	‘class’, ‘code’, и тд.
Атрибут размера сегмента	USE16, USE32

Readonly: если операнд присутствует, MASM выдаст сообщение об ошибке на все команды, выполняющие запись в данный сегмент. Другие ассемблеры этот операнд игнорируют.
Атрибут выравнивания сегмента (тип выравнивания) `align` сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном

выравнивании доступ к данным в процессорах, совместимых с базовым i8086, выполняется быстрее. Допустимые значения этого атрибута следующие:

- BYTE — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
- WORD — сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
- DWORD — сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание по границе двойного слова);
- PARA — сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание по границе параграфа) - используется по умолчанию;
- PAGE — сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание по границе страницы размером 256 байт);
- MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей страницы памяти размером 4 Кбайт);

По умолчанию тип выравнивания имеет значение PARA.

Атрибут комбинирования сегментов (комбинаторный тип) combine сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. По умолчанию атрибут комбинирования принимает значение PRIVATE. Значениями атрибута комбинирования сегмента могут быть:

- PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
- PUBLIC — заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
- COMMON — располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;

AT xxxx — располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением xxxx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий

сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса. Результат выражения, использующегося в качестве операнда для AT, равен этому адресу, деленному на 16.
Например: segment at 40h - сегмент, начинающийся по абсолютному адресу
0400h. Такие сегменты обычно содержат только метки, указывающие на об-
ласти памяти, которые могут потребоваться программе;

- `STACK` — определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра `SS`. Комбинированный тип `STACK` (стек) аналогичен комбинированному типу `PUBLIC`, за исключением того, что регистр `SS` является стандартным сегментным регистром для сегментов стека. Регистр `SP` устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип `STACK` не используется, программист должен явно загрузить в регистр `SS` адрес сегмента (подобно тому, как это делается для регистра `DS`).

Атрибут размера сегмента `dim`. Для процессоров i80386 и выше сегменты могут быть 16- или 32-разрядными. Это влияет прежде всего на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

- `USE16` — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных. Используется по умолчанию.
- `USE32` — сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных. В модели памяти `FLAT` используется по умолчанию именно это значение атрибута размера сегмента

Атрибут класса сегмента (тип класса) `‘class’` — это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при сборке программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется класс `‘code’`). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.

Все сегменты сами по себе равноправны, так как директивы `SEGMENT` и `ENDS` не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву `ASSUME`. Эта директива сообщает транслятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах.

Директива	Эквивалент
.CODE	_TEXT segment word public 'CODE'
.stack размер	STACK segment para public 'stack'
.data	_DATA segment word public 'DATA'
.data?	_BSS segment word public 'BSS'
.const	CONST segment word public 'CONST'
.FARDATA	имя_сегмента segment para private 'FAR_DATA'
.fardata?	имя_сегмента segment para private 'FAR_BSS'

Размер стека не является обязательным параметром, по умолчанию он равен 1 Кб.

Сегмент .data? обычно не включается в программу, а располагается за концом памяти, так что все описанные в нем переменные на момент загрузки программы имеют неопределенные значения; т.е. сегмент описывает неинициализированные данные.

.const описывает сегмент неизменяемых данных.

Имя идентификатора	Значение переменной
@code	Физический адрес сегмента кода
@data	Физический адрес сегмента данных типа near
@fardata	Физический адрес сегмента данных типа far

@fardata?	Физический адрес сегмента неинициализированных данных типа far
@stack	Физический адрес сегмента стека

Модели памяти задаются директивой .MODEL

.model модель

где модель - одно из следующих слов:

TINY - код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;

SMALL - код размещается в одном сегменте, а данные и стек - в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;

COMPACT - код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);

MEDIUM - код размещается в нескольких сегментах, а все данные - в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;

LARGE и HUGE — и код, и данные могут занимать несколько сегментов;

FLAT - то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, - 4 Мб.

```
model   small              ;модель памяти
.data                      ;сегмент данных
message db      'Введите две шестнадцатеричные цифры,$'
.stack                     ;сегмент стека
db      256     dup ('?')       ;сегмент стека
.code                      ;сегмент кода
main    proc               ;начало процедуры main
mov     ax,@data           ;заносим адрес сегмента
   ;данных в регистр ax
mov     ds,ax   ;ax в ds
........
.........
mov     ax,4c00h           ;пересылка 4c00h в регистр ax
int     21h                ;вызов прерывания с номером 21h
main    endp               ;конец процедуры main
end     main               ;конец программы с точкой входа main
```