

2017

Глава 3 *Модульное программирование*

МГТУ им. Н.Э. Баумана

Факультет Информатика и системы управления

Кафедра Компьютерные системы и сети

Лектор: д.т.н., проф.

Иванова Галина Сергеевна

3.1 Организация передачи управления в процедуру и обратно

Процедура в ассемблере – это относительно самостоятельный фрагмент, к которому возможно обращение из разных мест программы.

- На языках высокого уровня такие фрагменты оформляют соответствующим образом и называют подпрограммами: *функциями* или *процедурами* в зависимости от способа возврата результата.
- Поддержка модульного принципа для ассемблера означает, что в языке существуют специальные **машинные** команды вызова подпрограммы и обратной передачи управления.
- Кроме машинных команд в языке существует набор макрокоманд и директив, упрощающий работу с процедурами.

Команды вызова процедуры и возврата управления

1. Команда вызова процедуры:

CALL rel32/r32/m32 ; вызов внутрисегментной
; процедуры (*near* - ближний)
CALL sreg:r32/m48 ; вызов межсегментной процедуры
; (*far* - дальний)

2. Команда возврата управления:

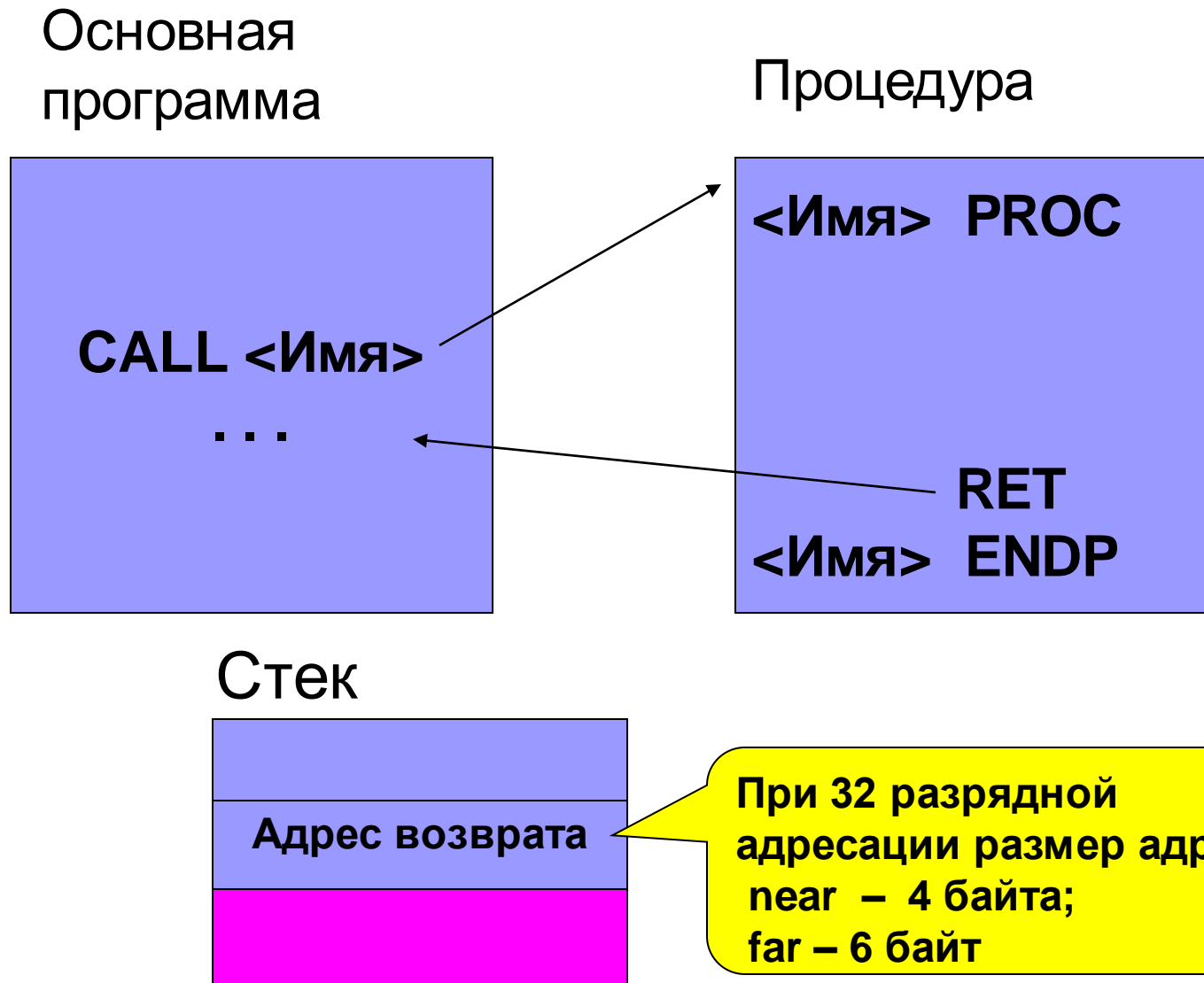
RET [<Целое>]

где <Целое> – количество байт, извлекаемых из стека при возврате управления – используется для удаления из стека параметров процедуры (см. далее).

При выполнении команды вызова процедуры автоматически в стек заносится адрес команды, следующей за командой вызова процедуры, – адрес возврата.

Команда возврата управления выбирает этот адрес из стека и осуществляет переход по нему.

Организация передачи управления в процедуру



Пример 3.1 Процедура MaxDword ()

.CONST

MsgExit DB "Press Enter to Exit",0AH,0DH,0

.DATA

A DWORD 56

B DWORD 34

.DATA?

D DWORD ?

inbuf DB 100 DUP (?)

.CODE

Start:

call MaxDword ; *вызов процедуры*

XOR EAX,EAX

Invoke StdOut,ADDR MsgExit

Invoke StdIn,ADDR inbuf,LengthOf inbuf

Invoke ExitProcess,0

END Start

Текст
процедуры

Текст процедуры

MaxDword **PROC**

push EAX ; сохранить регистр

push EBX ; сохранить регистр

lea EBX,D ; загрузить адрес результата

mov EAX,A ; загрузить первое число в регистр

cmp EAX,B ; сравнить числа

jg con ; если первое больше, то на запись

mov EAX,B ; загрузить второе число в регистр

con: mov [EBX],EAX ; записать результат

pop EBX ; восстановить регистр

pop EAX ; восстановить регистр

ret ; вернуть управление

MaxDword **ENDP**

или **mov** **D, EAX**

3.2 Передача данных в подпрограмму

Данные могут быть переданы в подпрограмму:

- **через регистры** — перед вызовом процедуры параметры или их адреса загружаются в регистры, также в регистрах возвращаются результаты;
- **напрямую** — с использованием механизма глобальных переменных:
 - ☐ при совместной трансляции,
 - ☐ при раздельной трансляции;
- **через таблицу адресов** — в программе создается таблица, содержащая адреса параметров, и адрес этой таблице передается в процедуру через регистр;
- **через стек** — перед вызовом процедуры параметры или их адреса заносятся в стек, после завершения процедуры они из стека удаляются.

3.2.1 Передача параметров в регистрах

Пример 3.2 а. Определение суммы двух целых чисел

.DATA

A DWORD 56

B DWORD 34

.DATA?

D DWORD ?

inbuf DB 100 DUP (?)

.CODE

Start:

; Занесение параметров в регистры

lea EDX,D ; адрес результата

mov EAX,A ; первое число

mov EBX,B ; второе число

call SumDword ; вызов процедуры

Invoke StdOut,ADDR MsgExit

Invoke StdIn,ADDR inbuf,LengthOf inbuf

Invoke ExitProcess,0

Процедура, получающая параметры в регистрах

```
SumDword PROC
    add     EAX, EBX
    mov     [EDX], EAX
    ret
SumDword ENDP
```

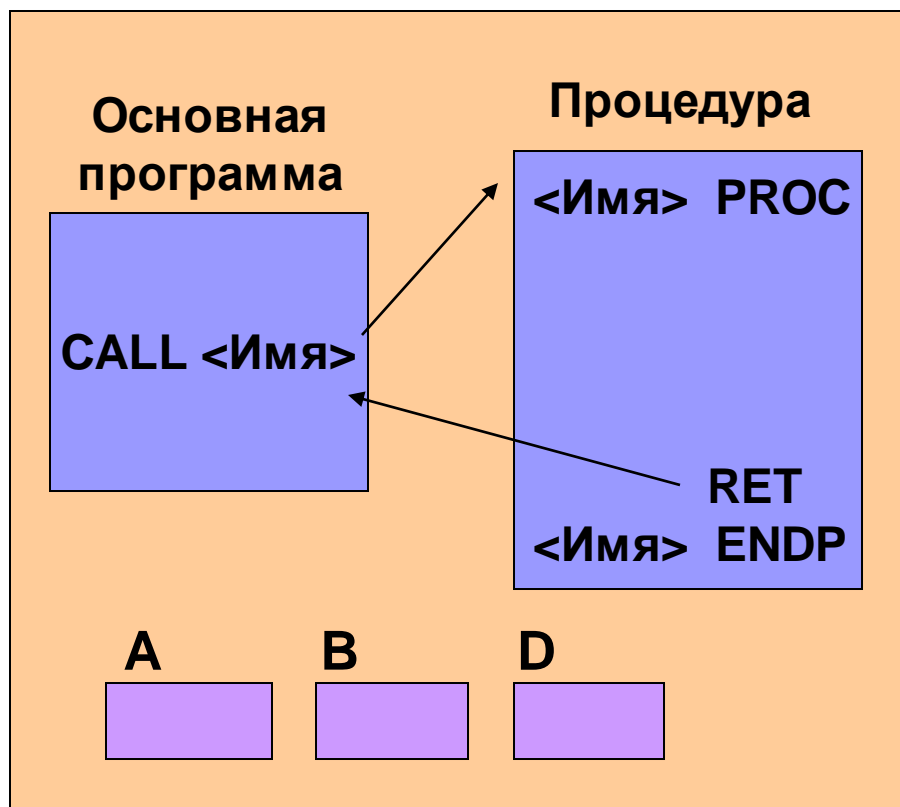
; завершение модуля

```
End      Start
```

Процедуры, получающие параметры в регистрах, используется, если количество параметров невелико, и в программе на ассемблере можно найти соответствующее количество незанятых регистров.

3.2.2 Процедуры с глобальными переменными (совместная трансляция)

Исходный модуль



При совместной трансляции, когда основная программа и процедура объединены в один исходный модуль, ассемблер строит общую таблицу символьных имен. Следовательно, и основная программа и процедура могут обращаться к символьным именам, объявленным в том же модуле.

Способ не технологичен:

- процедуры не универсальны;
- большое количество ошибок.

Процедура, работающая с глобальными переменными при совместной трансляции

Пример 3.2 b. Определение суммы двух чисел.

.DATA

A DWORD 56 ; первое число

B DWORD 34 ; второе число

.DATA?

D DWORD ? ; место для результата

.CODE

Start: call SumDword

 . . .

SumDword PROC

mov EAX, A ; поместили в регистр 1-е число

add EAX, B ; сложили со вторым

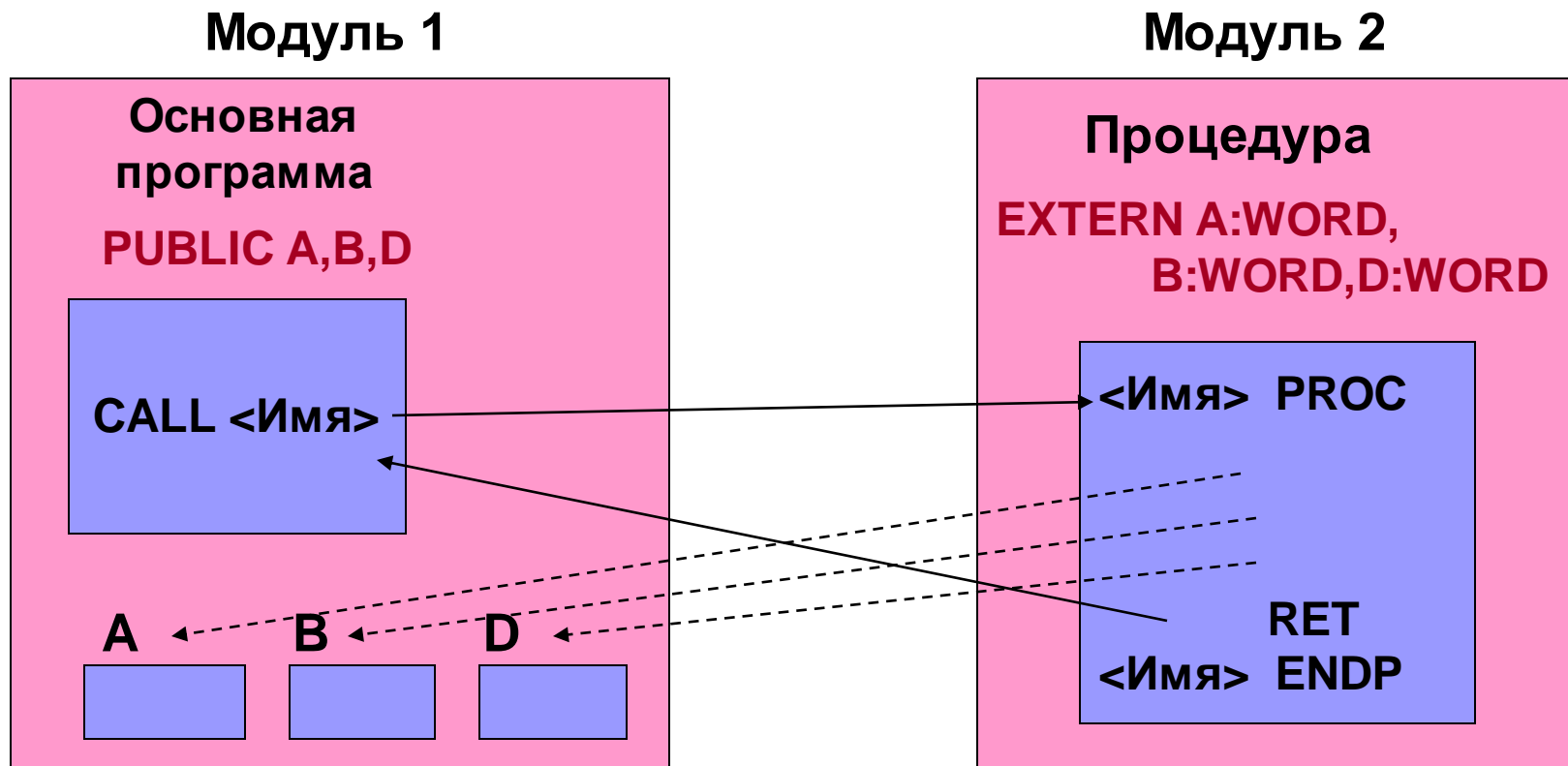
mov D, EAX ; результат отправили на место

ret

SumDword ENDP

End Start

3.2.3 Многомодульные программы



Объединение модулей осуществляется во время компоновки программ. Программа и процедуры, размещенные в разных исходных модулях, на этапе ассемблирования «не видят» символических имен друг друга. Чтобы сделать имена видимыми за пределами модуля, их объявляют «внешними». Для этого используют директивы **PUBLIC**, **EXTERN** или **EXTERND**.

Директивы описания глобальных переменных

Директива описания внутренних имен, к которым возможно обращение извне:

PUBLIC [<Язык>] <Имя> [, <Язык>] <Имя>...

где <Язык> – описатель, определяющий правила формирования внутренних имен (см. далее);

<Имя> – символическое имя, которое должно быть доступно (видимо) в других модулях.

Директива описания внешних имен, к которым есть обращение в этом модуле:

EXTERN [<Язык>] <Имя> [(<Псевдоним>)]:<Тип>

[, [<Язык>] <Имя> [(<Псевдоним>)]:<Тип> ...

где <Тип> - NEAR, FAR, BYTE, WORD, DWORD и т.д.

Универсальная директива описания имен обоих типов – может использоваться вместо PUBLIC и EXTERN:

EXTERNDEF [<Язык>] <Имя>:<Тип> [[<Язык>] <Имя>:<Тип>]...

Основная программа при раздельной трансляции

Пример 3.2 с. Сложение двух чисел.

.DATA

A DWORD 56

B DWORD 34

.DATA?

D DWORD ?

PUBLIC A,B,D ; объявление внутренних имен

EXTERN SumDword:near ; объявление внеш. имен

.CODE

Start: call SumDword ; вызов подпрограммы

· · ·

Процедура при раздельной трансляции

.586

.MODEL flat, stdcall

OPTION CASEMAP:NONE

EXTERN A:DWORD, B:DWORD, D:DWORD

.CODE

```
SumDword PROC    c
    push    EAX
    mov     EAX, A
    add     EAX, B
    mov     D, EAX
    pop     EAX
    ret
SumDword ENDP
END
```

3.2.4 Передача параметров через таблицу адресов

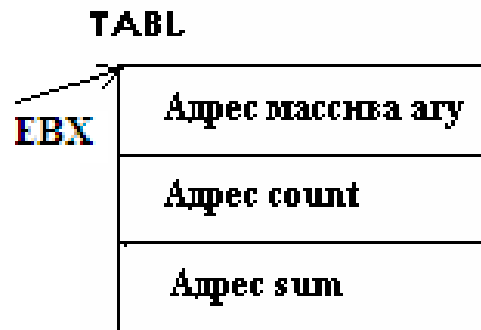
Пример 3.2 d. Сумма элементов массива

```
.DATA
ary      SWORD  5,6,1,7,3,4 ; массив
count    DWORD  6           ; размер массива
sum       SWORD  ?           ; сумма элементов
tabl     DWORD  3 dup(?)     ; таблица адресов параметров
EXTERN   masculc:near
.CODE
```

Start:

; формирование таблицы адресов параметров

```
mov      tabl,offset ary
mov      tabl+4,offset count
mov      tabl+8,offset sum
mov      EBX,offset tabl
call     masculc
. . .
```



Процедура, получающая параметры через таблицу адресов

.586

.MODEL flat, stdcall

OPTION CASEMAP:NONE

.CODE

masculc proc c

push AX ; сохранение регистров

push ECX

push EDI

push ESI

; использование таблицы адресов параметров

mov ESI, [EBX] ; адрес массива

mov EDI, [EBX+4] ; адрес размера

mov ECX, [EDI] ; размер массива

mov EDI, [EBX+8] ; адрес результата

TABL

EBX

Адрес массива агу
Адрес count
Адрес sum

Процедура, получающая параметры через таблицу адресов

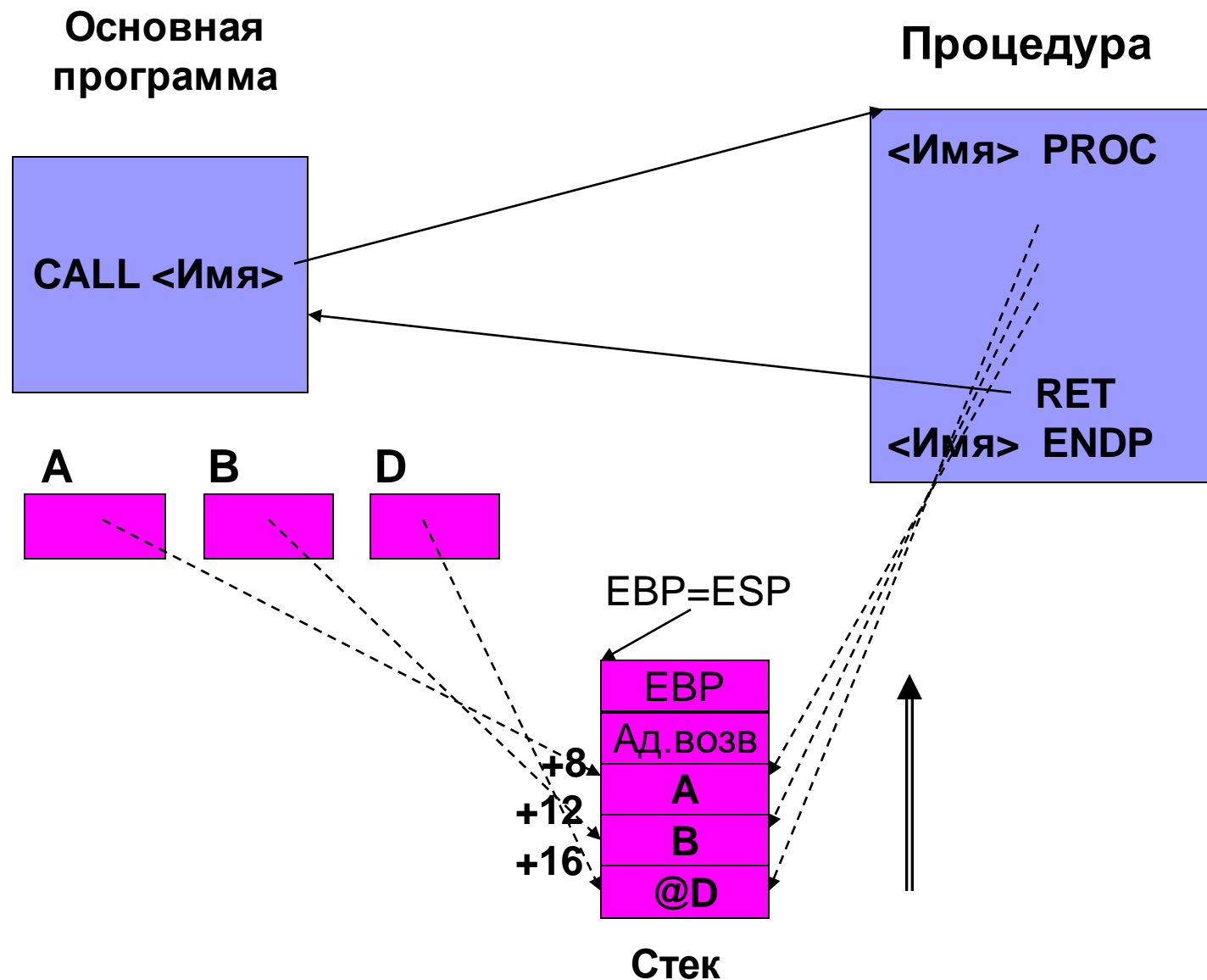
; суммирование элементов массива

```
        xor     AX,AX  
cyc1:   add     AX,[ESI]  
        add     ESI,2  
        loop    cyc1
```

; формирование результатов

```
        mov     [EDI],AX  
        pop     ESI      ; восстановление регистров  
        pop     EDI  
        pop     ECX  
        pop     AX  
        ret  
masculc endp  
END
```

3.2.5 Передача параметров через стек



Пример 3.2 е. Максимальное из двух чисел.

.DATA

A	DWORD	56
B	DWORD	34

.DATA?

D	DWORD	?
---	-------	---

.CODE

Start:

```
lea    EBX,D ; получение адреса результата
push   EBX   ; загрузка в стек адреса результата
push   B     ; загрузка в стек второго числа
push   A     ; загрузка в стек первого числа

call   MaxDword

. . .
```

Получение
управления
процедурой

Стек

Адрес возв

A

B

Адрес D

Исходное
состояние
стека



Процедура, получающая параметры через стек

MaxDword PROC

```
push    EBP  
mov     EBP, ESP
```

Пролог

```
push    EAX  
push    EBX
```

```
mov     EBX, [EBP+16] ; адрес D
```

```
mov     EAX, [EBP+8] ; A
```

```
cmp     EAX, [EBP+12] ; B
```

```
jg      con
```

```
mov     EAX, [EBP+12]
```

```
con:    mov     [EBX], EAX
```

```
pop     EBX
```

```
pop     EAX
```

```
mov     ESP, EBP
```

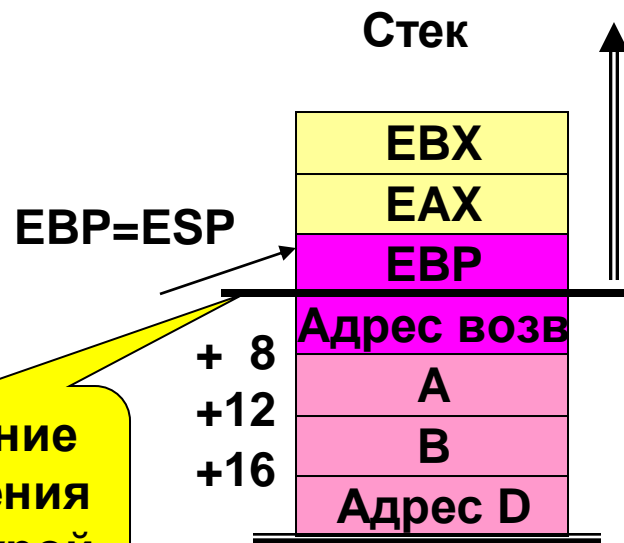
```
pop     EBP
```

```
ret     12
```

Эпилог

MaxDword ENDP

Удаление из стека
области параметров



Исходное
состояние
стека

3.3 Директивы описания процедур

1. Директива заголовка процедуры:

**<Имя процедуры> PROC [<Тип вызова>] [<Конвенция о связи>]
[<Доступность>]
[USES <Список используемых регистров>]
[,<Параметр> [:<Тип>]]...**

Тип вызова:

far – межсегментный;

near – внутрисегментный (используется по умолчанию).

Конвенция о связи (по умолчанию используется указанная в **.MODEL**):

STDCALL – стандартные Windows;

C – принятые в языке C,

PASCAL – принятые в языке Pascal и др.

Доступность – видимость процедуры из других модулей:

public – общедоступная (используется по умолчанию);

private – внутренняя;

export – межсегментная и общедоступная.

Директивы описания процедур (2)

Список используемых регистров – содержит регистры, используемые в процедуре, для их автоматического сохранения и восстановления.

Параметр – имя параметра процедуры.

Тип – тип параметра или VARARG. Если тип не указан, то по умолчанию для 32-х разрядной адресации берется DWORD. Если указано VARARG, то вместо одного аргумента разрешается использовать список аргументов через запятую.

Пример:

```
ABC      PROC NEAR STDCALL PUBLIC USES EAX,  
          X:DWORD, Y:BYTE, H:DWORD PTR
```

Директивы описания процедур (3)

2. Директива описания локальных переменных:

```
LOCAL <Имя> [ [<Количество>] ] [ :<Тип> ]  
          [ , <Имя> [ [<Количество>] ] [ :<Тип> ] ] ...
```

Описывает переменные, размещаемые в стеке, т.е. локальные.
Используется только в процедурах. Помещается сразу после PROC.

Пример:

```
ABC    PROC    USES EAX, X:VARARG  
        LOCAL  ARRAY[20]:BYTE  
        . . .
```


Директивы описания процедур (3)

3. Директива объявления прототипа:

**<Имя процедуры> PROTO [<Тип вызова>] [<Соглашения о связи>]
[<Доступность>]
[,<Параметр> [:<Тип>]]...**

Значения параметров совпадают со значениями параметров директивы PROC. Используется для указания списка и типов параметров для директивы INVOKE.

Пример:

MaxDword **PROTO** NEAR STDCALL PUBLIC

X:DWORD, Y:DWORD, ptrZ:PTR DWORD

или с учетом умолчаний:

MaxDword **PROTO** X:DWORD, Y:DWORD, ptrZ:PTR DWORD

Директивы описания процедур (4)

4. Директива вызова процедуры:

INVOKE <Имя процедуры или ее адрес> [, <Список аргументов>]

Аргументы должны совпадать с параметрами по порядку и типу.

Типы аргументов директивы INVOKE:

- **целое значение**, например:
27h, -128;
- **выражение целого типа, использующее операторы получения атрибутов полей данных**:
TYPE mas, SYZEOF mas+2, OFFSET AR, (10*20);
- **регистр**, например:
EAX, BH;
- **адрес переменной**, например:
Ada1, var2_2;
- **адресное выражение**, например:
4[EDI+EBX], Ada+24, ADDR AR.

Операторы получения атрибутов полей данных

ADDR <Имя поля данных> – возвращает ближний или дальний адрес переменной в зависимости от модели памяти – для Flat – ближний;

OFFSET <Имя поля данных> – возвращает смещение переменной относительно начала сегмента – для Flat совпадает с ADDR;

TYPE <Имя поля данных> – возвращает размер в байтах элемента описанных данных, например:

```
A    BYTE 34 dup (?); // размер = 1
```

LENGTHOF <Имя поля данных> – возвращает количество элементов, заданных при определении данных, например

```
A    BYTE 34 dup (?); // 34 элемента
```

SIZEOF <Имя поля данных> – возвращает размер поля данных в байтах;

<Тип> PTR <Имя поля данных> – изменяет тип поля данных на время выполнения команды.

Пример 3.3 Использование PROC, PROTO и INVOKE

MaxDword **PROTO** X:DWORD, Y:DWORD, ptrZ:PTR DWORD

.DATA

A DWORD 56

B DWORD 34

.DATA?

D DWORD ?

.CODE

Start: **INVOKE** MaxDword, A, B, **ADDR** D

. . .

MaxDword **PROC** **USES** EAX EBX,
X:DWORD, Y:DWORD, ptrZ:PTR DWORD

mov EBX, ptrZ

mov EAX, X

cmp EAX, Y

jg con

mov EAX, Y

con: mov [EBX], EAX

ret

MaxDword **ENDP**

Определяет аргумент
как указатель типа
DWORD

3.4 Функции ввода-вывода консольного режима (MASM32.lib)

Библиотека MASM32.lib содержит специальные функции ввода вывода консольного режима:

1. Процедура ввода в консольном режиме:

StdIn PROC IpszBuffer:DWORD, ; буфер ввода
bLen:DWORD ; размер буфера ввода до 128 байт

2. Процедура удаления символов конца строки при вводе:

StripLF PROC string:DWORD ; буфер ввода

3. Процедура вывода завершающейся нулем строки в окно консоли:

StdOut PROC IpszText:DWORD ; буфер вывода, зав. нулем

4. Функция позиционирования курсора:

locate PROC x:DWORD, y:DWORD ; местоположение курсора,
(0,0) – левый верхний угол

5. Процедура очистки окна консоли:

ClearScreen PROC

Пример 3.4 Программа извлечения корня квадратного

$$1 = 1^2$$

$$1+3 = 4 = 2^2$$

$$1+3+5 = 9 = 3^2$$

.DATA

zap	DB	'Input value <65024: ',13,10,0
string	DB	10 dup ('0')
otw	DB	13,10,'Root ='
rez	DB	' ',13,10,0

Программа извлечения корня квадратного (2)

.CODE

Start:

; ВВОД

vvod:

Invoke StdOut, ADDR zap *; ВЫВОД запроса*

Invoke StdIn, ADDR string, LengthOf string *; ВВОД*

Invoke StripLF, ADDR string *; преобразование*

конца

; строки в ноль

String

3 2 0 2 4 ↵

33	32	30	32	34	0D	0A	?	?	?
----	----	----	----	----	----	----	---	---	---

String

3 2 0 2 4

33	32	30	32	34	00	0A	?	?	?
----	----	----	----	----	----	----	---	---	---

Программа извлечения корня квадратного (3)

; Преобразование

String 3 2 0 2 4

33	32	30	32	34	00	0A	?	?	?
----	----	----	----	----	----	----	---	---	---

ESI ↗

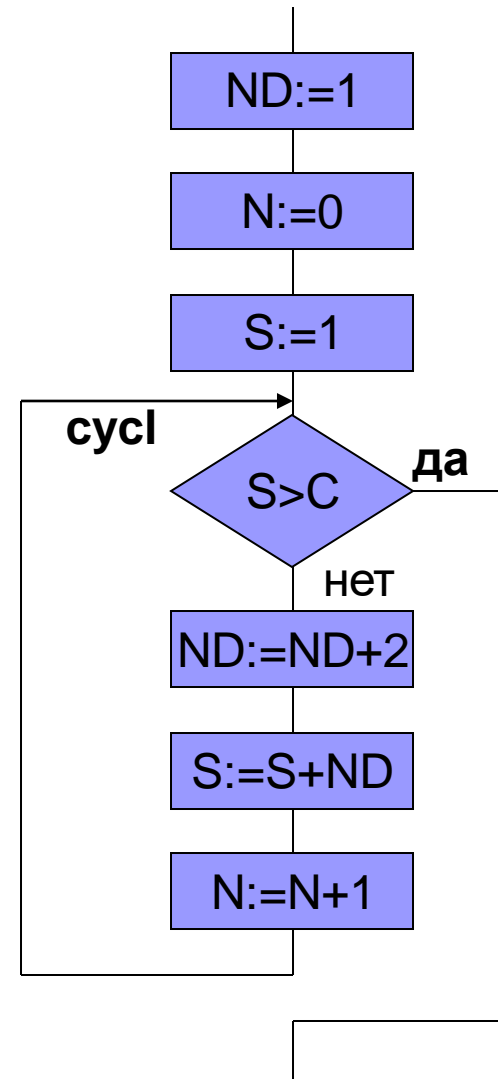
```

mov     BH, '9'
mov     BL, '0'
lea     ESI, string
cld
xor     DI, DI
cycle:  lodsb                ; загружаем символ
        cmp     AL, 0        ; если 0, то на вычисление
        je      calc
        cmp     AL, BL      ; это цифра ?
        jnb     vvod
        cmp     AL, BH
        ja      vvod
        sub     AL, 30h     ; получаем цифру из символа
        cbw                ; расширяем
        push    AX          ; сохраняем
        mov     AX, 10      ; заносим 10
        mul     DI          ; умножаем, результат в DX:AX
        pop     DI          ; в DI - цифра
        add     AX, DI
        mov     DI, AX      ; в DI - число
        jmp     cycle
    
```


Программа извлечения корня квадратного (4)

;Вычисление sqrt(dx#ax)

```
calc:      mov     BX,1
           mov     CX,0
           mov     AX,1 ; сумма
cycle:     cmp     AX,DI
           ja      preobr
           add     BX,2
           add     AX,BX
           jc      vvod
           inc     CX
           jmp     cycle
```



Программа извлечения корня квадратного (5)

; Преобразование

```
preobr:  mov     AX,CX
          mov     EDI,2
          mov     BX,10
again:   cwd
          div     BX
          add     DL,30h
          mov     rez[EDI],DL
          dec     EDI
          cmp     AX,0
          jne     again
          Invoke  StdOut,ADDR otw
```

Функции преобразования данных

1. *Функция преобразования завершающейся нулем строки в число:*

atoi proc lpSrc:DWORD ; результат – в EAX

2. *Функция преобразования строки (зав. нулем) в беззнаковое число:*

ustr2dw proc pszString:DWORD ; результат – в EAX

3. *Функция преобразования строки в число:*

atodw proc uses edi esi, String:PTR BYTE ; результат – в EAX

4. *Процедура преобразования числа в строку (16 байт):*

ltoa proc lValue:DWORD, lpBuffer:DWORD

5. *Процедура преобразования числа в строку:*

dwtoa proc public uses esi edi, dwValue:DWORD, lpBuffer:PTR BYTE

6. *Процедура преобразования беззнакового числа в строку:*

udw2str proc dwNumber:DWORD, pszString:DWORD

Пример 3.5 Преобразование ввода

.CODE

Start:

; Ввод

```
vvod:      Invoke StdOut,ADDR zap
           Invoke StdIn,ADDR string,LengthOf string
           Invoke StripLF,ADDR string
```

; Преобразование

```
           Invoke atol,ADDR string ;результат в EAX
mov        DI,AX
```

Пример 3.5 Преобразование вывода

; Преобразование

```
preobr:  mov     word ptr root,CX
```

```
        Invoke dwtoa,root,ADDR rez
```

; Вывод

```
        Invoke StdOut,ADDR otw
```

```
        . . .
```

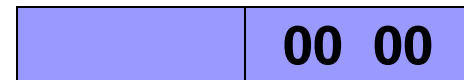
.DATA

```
root     DWORD 0
```

```
otw       DB     13,10,'Root ='
```

```
rez       DB     16 dup (?)
```

root



CX



otw

rez



3.5 Создание рекурсивных процедур

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. Для этого, каждый вызов должен запоминать свой набор параметров, регистры и все промежуточные результаты.

Для сохранения данных очередного вызова и передачи параметров следующей активации процедуры лучше использовать стек. А удобную организацию стека позволяет организовать структуры.

Структура

Структура представляет собой шаблон с описаниями форматов данных, который можно накладывать на различные участки памяти, чтобы затем обращаться к полям этих участков памяти с помощью имен, определенных в описании структуры.

Формат описания структуры:

<Имя структуры> **STRUCT**

 <Описание полей>

<Имя структуры> **ENDS**

где <Описание полей> – любой набор псевдокоманд определения переменных или вложенных структур.

Пример:

```
Student  struct
```

```
    Family      db 20 dup ( ' ' )      ; Фамилия студента
```

```
    Name        db 15 dup ( ' ' )      ; Имя
```

```
    Birthdata   db ' / / '            ; Дата рождения
```

```
Student  ends
```

Последовательность директив описывает, но не размещает в памяти структуру данных!!!

Структура (2)

Кроме того, структуры используются, когда в программе многократно повторяются сложные коллекции данных с единым строением, но с различными значениями полей. В этом случае, для создания такой структуры в памяти достаточно использовать имя структуры как псевдокоманды по шаблону:

<Имя переменной> <Имя структуры>

<<Значение поля 1>, ..., <Значение поля n>>

Примеры:

stud1 Student <'Иванов' , 'Петр' , '23/12/72' >

stud2 Student <'Сидоров' , 'Павел' , '12/05/84' >

Для чтения или записи в элемент структуры применяется точечная нотация:

<Имя структуры>. <Имя поля>.

Пример:

stud1 .Name [EBX]

Пример Ex03_06. Вычисление факториала

$$N! = \begin{cases} N*(N-1)!, & \text{при } N \neq 0 \text{ – рекурсивное утверждение;} \\ 1 & , \text{ при } N=0 \text{ – базисное утверждение.} \end{cases}$$

Фрейм активации включает:

- значение регистра EBP – 4 байта,
- адрес возврата для случая ближнего вызова – 4 байта,
- число N на данном уровне рекурсии – 2 байта,
- адрес результата – 4 байта.

Для работы с фреймом опишем структуру, перечисляя поля в том порядке, в котором они будут размещаться в стеке:

FRAME STRUCT

SAVE_EBP DD ?

SAVE_EIP DD ?

N DW ?

RESULT_ADDR DD ?

FRAME ENDS

Факториал. Основная программа

.DATA

n DW 5 ; *исходное число*

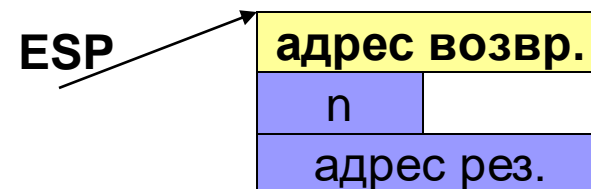
.DATA?

result DW ? ; *память под результат*

.CODE

Start:

```
push    offset result ; запись адреса результата
push    n            ;    запись исходного числа
call    fact
... 
```



Факториал. Рекурсивная процедура

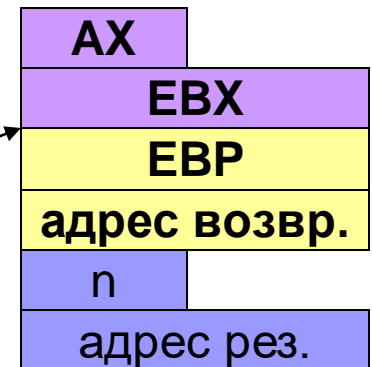
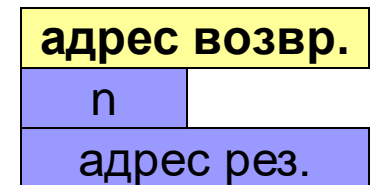
```

fact      PROC
          push    EBP
          mov     EBP, ESP
          push    EBX
          push    AX

; извлечение из стека адреса результата
          mov     EBX, FRAME.result_addr[EBP]

; извлечение из стека текущего N      EBP=ESP
          mov     AX, FRAME.n[EBP]
          cmp     AX, 0
          je      done      ; выход из рекурсии
          push    EBX      ; сохранение в стеке адреса рез.
          dec     AX      ; N=N-1
          push    AX      ; сохранение в стеке очередного N
          call    fact      ; рекурсивный вызов

```



Факториал. Рекурсивная процедура (2)

; извлечение из стека адреса результата

```
mov     EBX,FRAME.result_addr[EBP]
```

; вычисление результата очередной активации

```
mov     AX,[EBX]
```

```
mul     FRAME.n[EBP]
```

```
jmp     short return
```

```
done:   mov     EAX,1
```

; запоминаем результат активации

```
return: mov     [EBX],AX
```

```
pop     AX
```

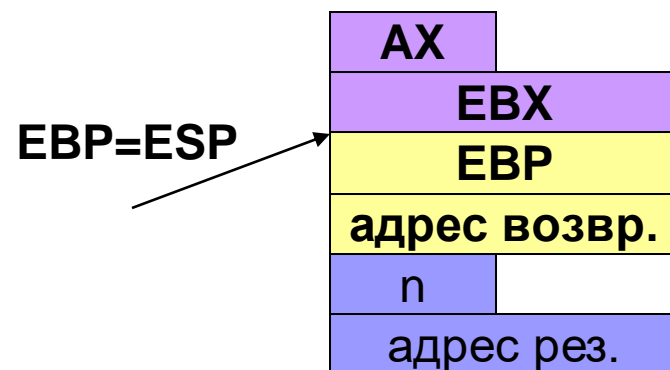
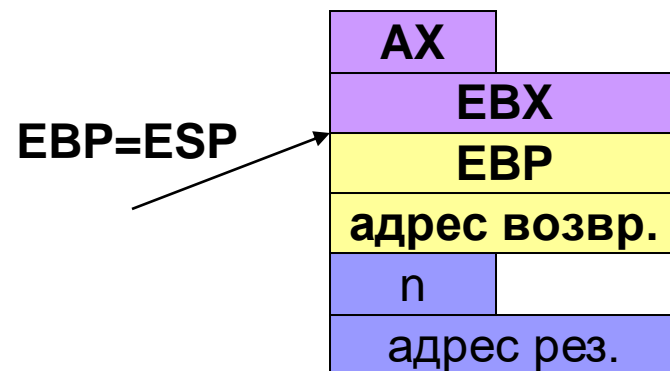
```
pop     EBX
```

```
pop     EBP
```

```
ret     6
```

```
fact    ENDP
```

```
End     Start
```



3.6 Связь разноязыковых модулей

Основные проблемы связи разноязыковых модулей:

- осуществление совместной компоновки модулей;
- организация передачи и возврата управления;
- передача параметров:
 - с использованием глобальных переменных,
 - с использованием стека (по значению и по ссылке),
- обеспечение возврата результата функции;
- обеспечение корректного использования регистров процессора.

Конвенции о связях WINDOW's

Конвенции о связи определяют правила передачи параметров.

№	Название в MASM32	Delphi Pascal	C++Builder	Visual C++	Порядок записи пар-ров в стек	Удале- ние пар-ров из стека	Исполь- зование регист- ров
1	PASCAL	pascal	__ pascal	-	прямой	проце- дура	-
2	C	cdecl	__ cdecl	__ cdecl	обрат- ный	осн. прогр.	-
3	STDCALL	stdcall	__ stdcall	__ stdcall	обрат- ный	проце- дура	-
4	-	register	__ fastcall	__ fastcall	обрат- ный	проце- дура	до 3-х (VC – до 2-х)
5	-	safecall	-	-	обрат- ный	проце- дура	

Конвенции о связях WINDOW's (2)

- тип вызова: **NEAR**;
- модель памяти: **FLAT**;
- пролог и эпилог – стандартные, текст зависит от конвенции и наличия локальных переменных:

□ пролог:

```
push    EBP
mov     EBP, ESP
[ sub   ESP, <Размер памяти локальных переменных>]
```

□ эпилог:

```
mov     ESP, EBP
pop     EBP
ret     [<Размер области параметров>]
```

Конвенции о связях WINDOW's (3)

- особенности компиляции и компоновки:

Delphi	C++ Builder	Visual C++
Преобразует все строчные буквы имен в прописные	Различает прописные и строчные буквы в именах	Различает прописные и строчные буквы в именах
Не изменяет внешних имен	Помещает «_» перед внешними именами	Помещает «_» перед внешними именами
Внутреннее имя совпадает с внешним	@<имя>\$q<описание параметров>	@<имя> @<количество параметров * 4>

- можно не сохранять регистры: **EAX, EDX, ECX.**
- необходимо сохранять регистры: **EBX, EBP, ESI, EDI.**

3.6.1 *Delphi PASCAL – MASM32*

- в модуле на Delphi Pascal процедуры и функции, реализованные на ассемблере, должны быть объявлены и описаны как внешние **external** с указанием конвенции связи, например:

procedure ADD1 (A,B:integer; Var C:integer); *pascal;external*;

- модуль ассемблера предварительно ассемблируется и подключается с использованием директивы **обычно** – в секции реализации модуля Delphi Pascal:

{\$I <Имя объектного модуля>}

Delphi PASCAL – MASM32

- СОВМЕСТИМОСТЬ ЧАСТО ИСПОЛЬЗУЕМЫХ ДАННЫХ:

Word – 2 байта,

Byte, Char, Boolean – 1 байт,

Integer, Pointer – 4 байта,

массив – располагается в памяти по строкам,

строка (**shortstring**) – содержит байт длины и далее символы;

- параметры передаются через стек:

- ☐ по значению – в стеке копия значения,

- ☐ по ссылке – в стеке указатель на параметр;

- результаты функций возвращаются через регистры:

- ☐ байт, слово – в **AX**,

- ☐ двойное слово, указатель – в **EAX**,

- ☐ строка – через указатель, помещенный в стек после параметров.

Пример 3.7 Delphi PASCAL – MASM32

Описание в Delphi:

Implementation

```
{ $I <Конвенция>.obj } // Имя файла совпадает с конвенцией  
procedure ADD1 (A,B:integer; Var C:integer); <Конвенция>;external;
```

Вызов процедуры: **ADD1(A,B,C);**

Указание

Для ассемблирования установить в настройках проекта RadASM:

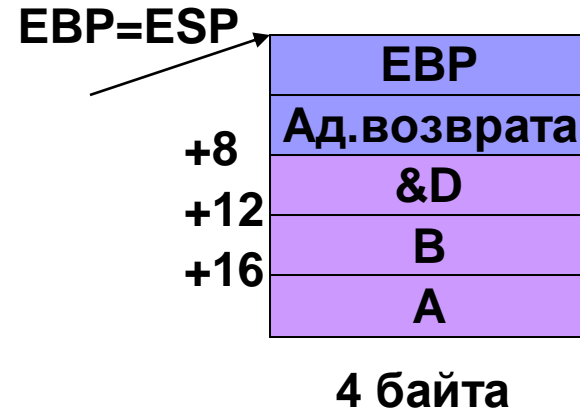
3,O,\$B\ML.EXE /c,2 или

добавить в Turbo Delphi инструмент (меню **Tools/Configure tools/Add**),
назначив в качестве инструмента программу-ассемблер ml.exe:

Title:	Masm32	- название;
Program:	C:\masm32\bin\ml.exe	- путь и ассемблер;
Working Dir:		- пусто (текущий каталог);
Parameters:	/c /FI \$EDNAME	- текущий файл редактора среды, ассемблирование и листинг

Пример 3.7 Конвенция PASCAL

```
.586
.model flat
.code
public ADD1
ADD1 proc
    push    EBP
    mov     EBP, ESP
    mov     EAX, [EBP+16]
    add     EAX, [EBP+12]
    mov     EDX, [EBP+8]
    mov     [EDX], EAX
    pop     EBP
    ret     12
ADD1 endp
end
```

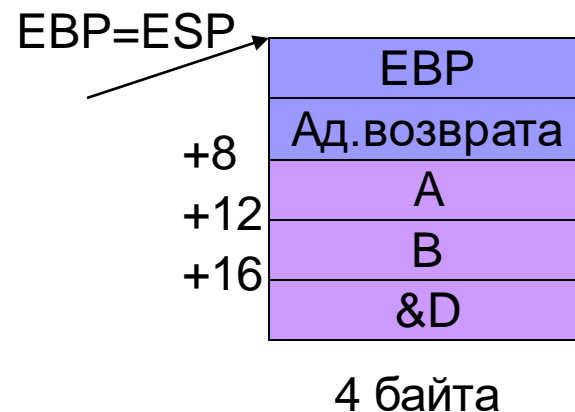


Пример 3.7 Конвенция cdecl

```
.586
.model flat
.code

public ADD1

ADD1 proc
    push    EBP
    mov     EBP, ESP
    mov     EAX, [EBP+8]
    add     EAX, [EBP+12]
    mov     EDX, [EBP+16]
    mov     [EDX], EAX
    pop     EBP
    ret
ADD1 endp
end
```



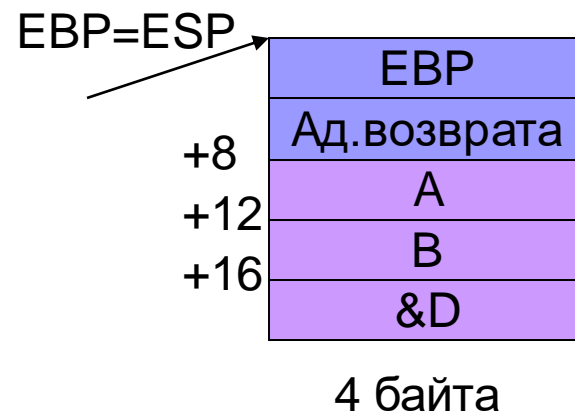
Пример 3.7 Конвенция stdcall (safecall = stdcall + исключение при ошибке)

```
.586
.model flat
.code

public ADD1

ADD1 proc
    push    EBP
    mov     EBP, ESP
    mov     EAX, [EBP+8]
    add     EAX, [EBP+12]
    mov     EDX, [EBP+16]
    mov     [EDX], EAX
    pop     EBP
    ret     12

ADD1 endp
end
```



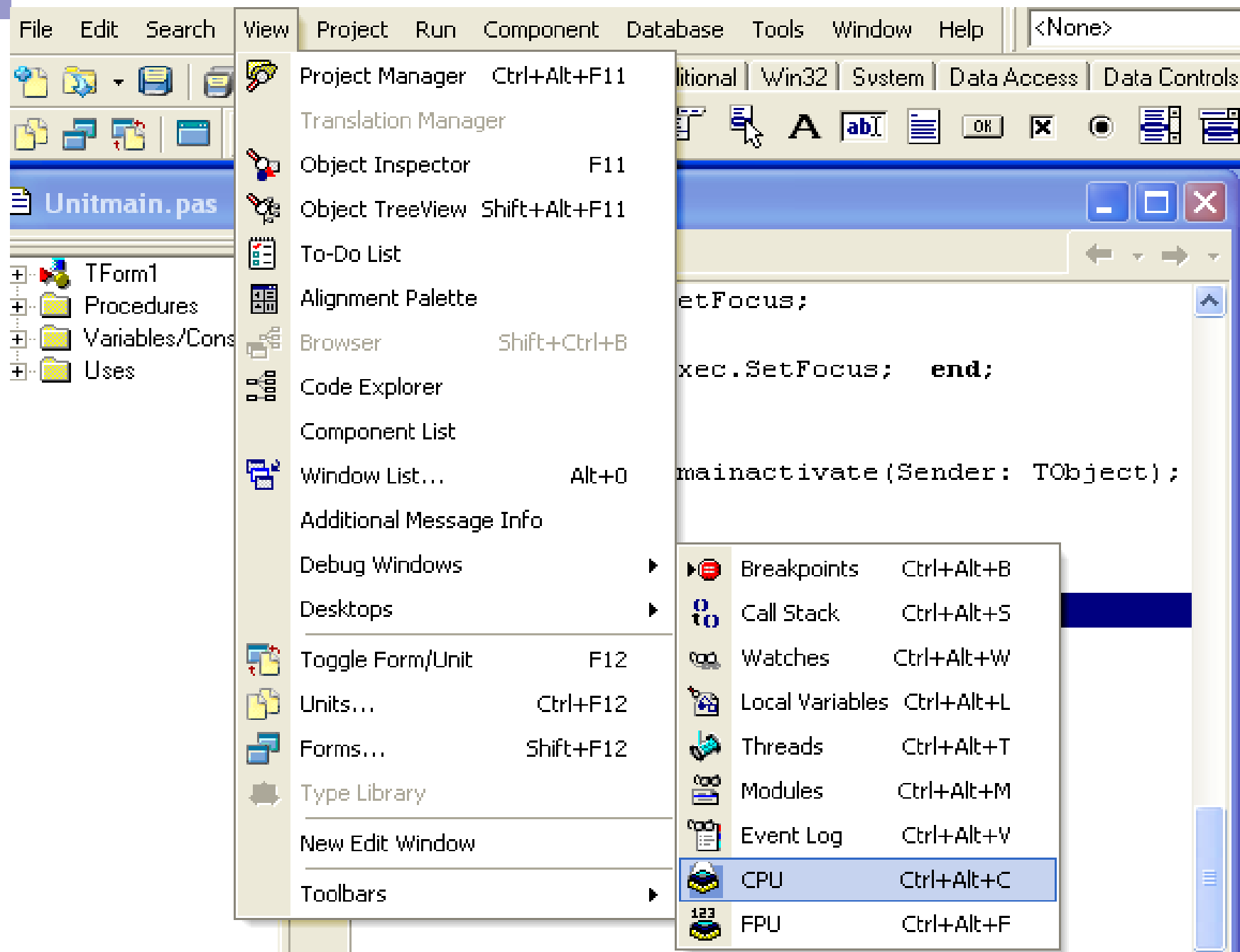
Пример 3.7 Конвенция register

```

      .586
      .model    flat
      .code
      public    ADD1
ADD1   proc
      add       EDX, EAX
      mov       [ECX], EDX
      ret
ADD1   endp
      end
```

1-й параметр А в EAX;
2-й параметр В в EDX;
3-й параметр &С в ECX
остальные параметры
в обратном порядке в
стеке

Ад.возврата



Окно CPU

Project1 - Turbo Delphi - Unit1 [Stopped]

File Edit Search View Refactor Project Run Component Tools Window Help

Debug Layout

Unit1 CPU

Thread #7812

ADD1:

0046360C	55	push ebp
0046360D	8BEC	mov ebp,esp
0046360F	8B4510	mov eax,[ebp+\$10]
00463612	03450C	add eax,[ebp+\$0c]
00463615	8B5508	mov edx,[ebp+\$08]
00463618	8902	mov [edx],eax
0046361A	5D	pop ebp
0046361B	C20C00	ret \$000c
0046361E	8BC0	mov eax,eax

Array_add:

00463620	B8A4A54600	mov eax,\$0046a5a4
00463625	B905000000	mov ecx,\$00000005
0046362A	800005	add byte ptr [eax],ecx

EAX	0018F50C	CF	0
EBX	01DB7170	PF	0
ECX	01DF5058	AF	1
EDX	0018F4D4	ZF	0
ESI	00000005	SF	0
EDI	0018F6B0	TF	0
EBP	0018F510	IF	1
ESP	0018F4DC	DF	0
EIP	0046360C	OF	0
EFL	00000212	IO	0
CS	0023	NF	0
DS	002B	RF	0
SS	002B	VM	0

0018F52C	0043A3D6	.
0018F528	01DD81A0	.

Пример 3.7 Процедура без параметров

Увеличение каждого элемента массива А на 5

```
procedure Array_add;pascal;external;
```

```
.586
```

```
.MODEL flat
```

```
.DATA
```

```
EXTERNDEF A:SBYTE; описание внешнего имени
```

```
.CODE
```

```
PUBLIC Array_add
```

```
Array_add proc
```

```
mov    eax,offset A    ; обращение к массиву А
```

```
mov    ecx,5
```

```
cycl:  add    byte ptr 0[eax],5
```

```
inc    eax
```

```
loop   cycl
```

```
ret
```

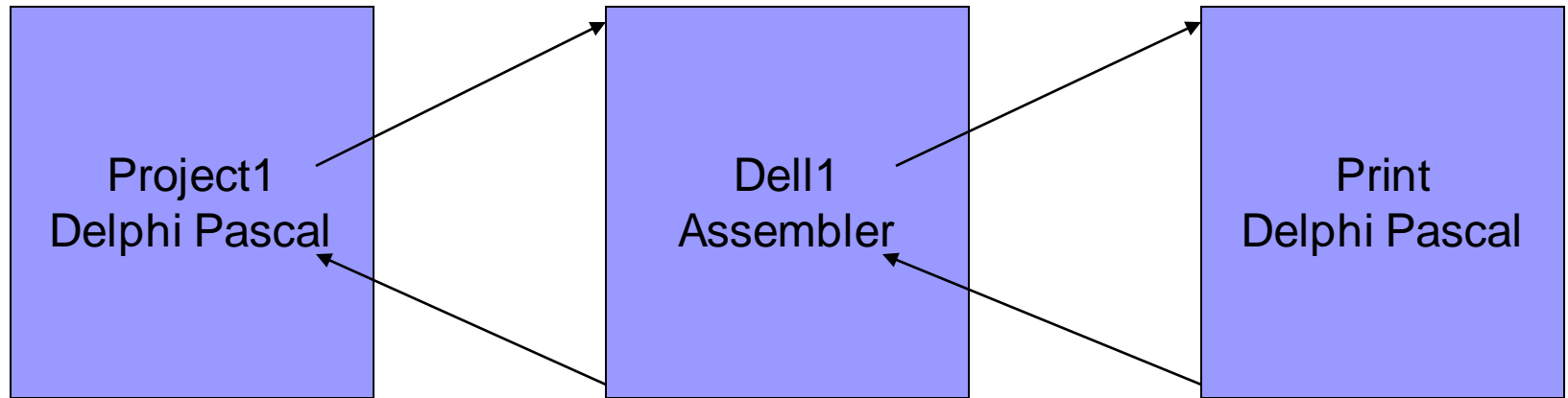
```
Array_add endp
```

```
end
```

ESP

Адрес возв.

Пример 3.7 Pascal – Assembler - Pascal

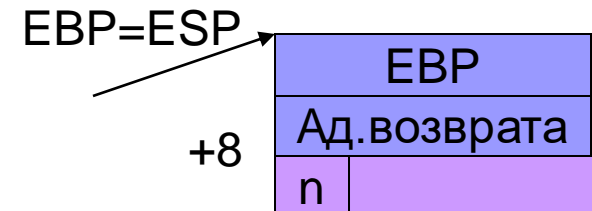
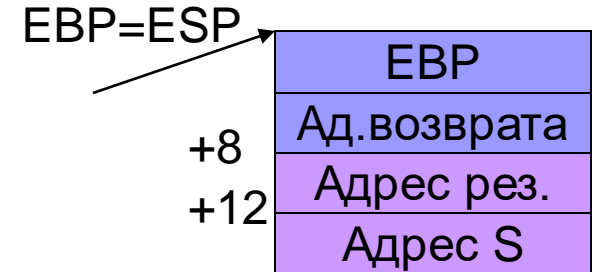


implementation

```
{ $L string.obj }
```

```
function Dell1(S: ShortString) :  
    ShortString; pascal; external;
```

```
procedure Print(n: byte); pascal;  
begin  
    Form1.Edit3.text := inttostr(n);  
end;
```



Пример 3.7 Pascal – Assembler – Pascal (2)

```
.586
.MODEL flat
.CODE
PUBLIC Del11
EXTERNDDEF Print:near

Del11 PROC
    push    EBP
    mov     EBP, ESP
    push    ESI
    push    EDI
    push    EBX
    mov     ESI, [EBP+12] ; адрес исходной строки
    mov     EDI, [EBP+8]  ; адрес строки-результата
    xor     ECX, ECX
    mov     CL, [ESI]     ; загрузка длины строки
    inc     ESI
    inc     EDI
```

Diagram illustrating the stack frame structure for the `Del11` procedure:

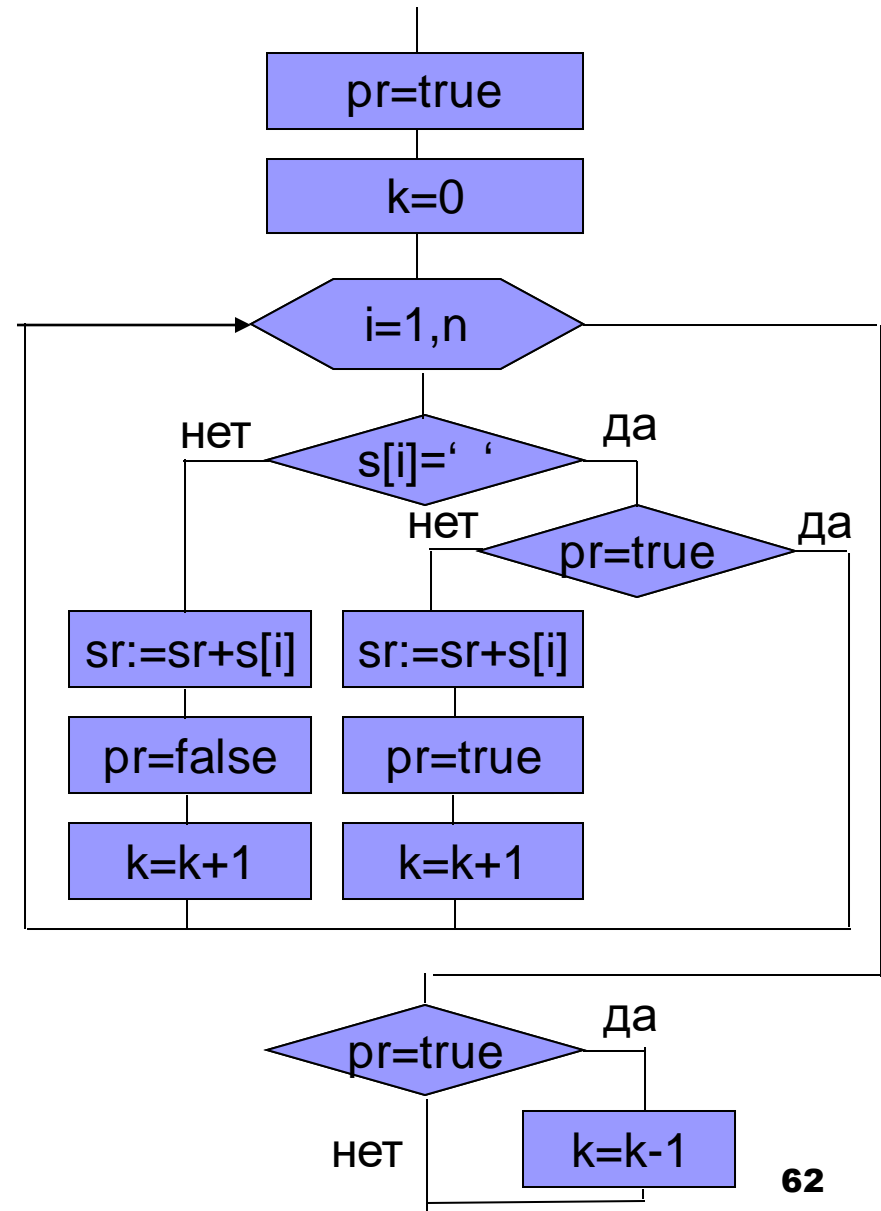
- ESP points to the top of the stack (EBX).
- EBP points to the base of the stack frame (EBP).
- The stack contains: EBX, EDI, ESI, EBP, a return address (Ад.возврата) at offset +8, a result address (Адрес рез.) at offset +12, and a source address (Адрес S).
- The result address (Адрес рез.) points to DS:EDI.
- The source address (Адрес S) points to ES:ESI.

Пример 3.7 Pascal – Assembler – Pascal (3)

```

                                mov     DL, 0
                                jcxz    prod3
                                mov     BX, 1
                                cld
cyc11:                         lodsb
                                cmp     AL, ' '
                                je       prod1
                                mov     BX, 0
                                inc     DL
                                stosb
                                jmp     prod2
prod1:                         cmp     BX, 1
                                je       prod2
                                mov     BX, 1
                                inc     DL
prod2:                         loop    cyc11

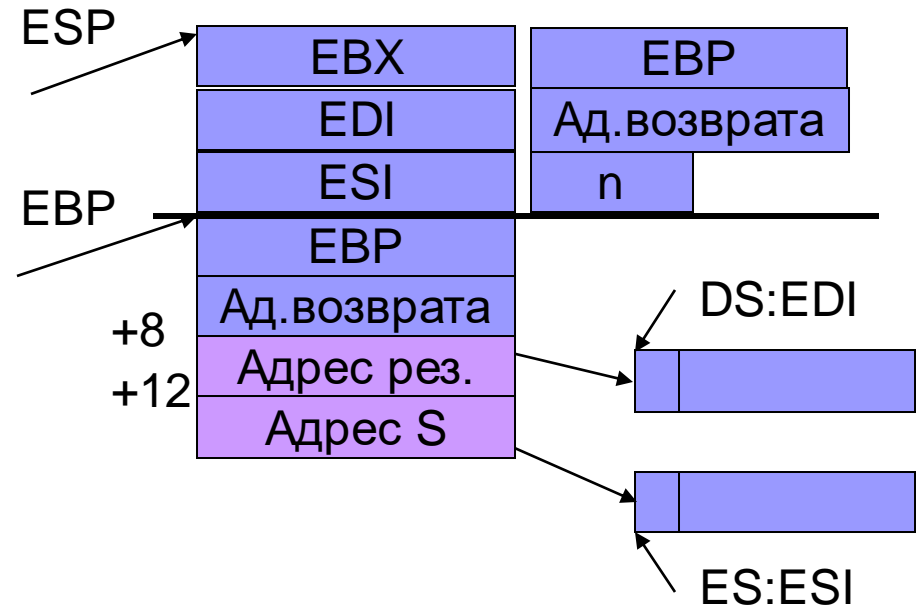
```



Пример 3.7 Pascal – Assembler – Pascal (4)

```

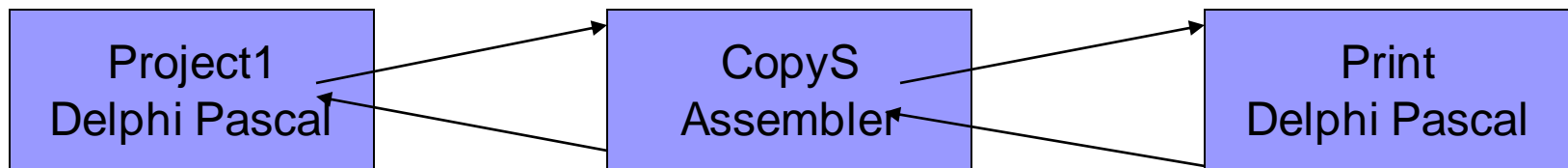
        cmp     DL, 0
        je      prod3
        cmp     BX, 1
        jne     prod3
        dec     DL
prod3:   mov     AL, DL
        mov     EDI, [EBP+8]
        stosb
        pop     EBX
        pop     EDI
        pop     ESI
        push    AX
        call    Print
        mov     ESP, EBP
        pop     EBP
        ret     8
Del11   endp
        end
    
```



Локальные данные подпрограмм

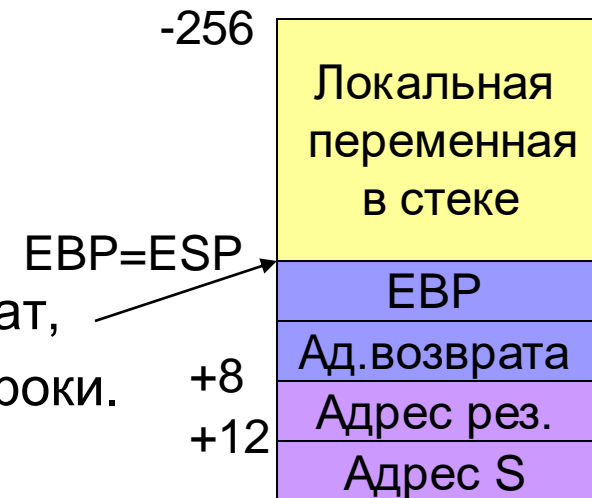
Паскаль не позволяет создавать в подпрограммах глобальные переменные, поэтому в подпрограммах работают с локальными данными, размещаемыми в стеке.

Пример 3.13. Организация локальных переменных без использования директив ассемблера



Подпрограмма на ассемблере:

- получает строку,
- копирует в локальную память,
- затем копирует из лок. памяти в результат,
- вызывает Паскаль для вывода длины строки.



Пример 3.13. Организация локальных переменных

```
implementation
```

```
{ $L Copy }
```

```
{ $R *.dfm }
```

```
function
```

```
    CopyS (St: ShortString) : ShortString; pascal; external;
```

```
procedure Print (n: integer) ; pascal;
```

```
begin Form1.Edit3.Text:=inttostr(n); end;
```

```
procedure TForm1.Button1Click (Sender: TObject) ;
```

```
Var S, St: ShortString;
```

```
begin St:=Edit1.Text;
```

```
    S:=CopyS (St) ;
```

```
    Edit2.Text:=S;
```

```
end;
```


Пример 3.13а. Без использования директив

```
.586
.MODEL flat

A   STRUCT                ; объявляем структуру
S   BYTE 256 DUP (?)      ; лок. переменная
A   ENDS                  ; завершение структуры

.CODE

CopyS public CopyS
      externdef Print:near

proc CopyS
push   EBP                ; сохранение EBP
mov    EBP,ESP             ; загрузка нового EBP
sub    ESP,260             ; место под лок. переменные
push   ESI                 ; сохранение регистров
push   EDI
mov     ESI,[EBP+12]        ; адрес параметра
lea     EDI,A.S[EBP-256]    ; обращение к лок.п.
xor     EAX,EAX
lodsb                     ; загрузка длины строки
stosb                      ; сохранение длины строки
```

-256

EBP →

Локальная переменная в стеке

EBP

Пример 3.13а. Без использования директив

```
mov     ECX,EAX      ; загрузка счетчика
cld
rep movsb             ; копирование строки
lea     ESI,A.S[EBP-256] ; загрузка адр. копии
mov     EDI,[EBP+8]   ; загрузка адр. рез-та
lodsb              ; загрузка длины строки
stosb              ; сохранение длины строки
mov     ECX,EAX      ; загрузка счетчика
rep movsb            ; копирование строки
pop     EDI           ; восстановление регистров
pop     ESI
push    EAX           ; сохранение длины строки
call    Print         ; вывод длины строки
mov     ESP,EBP       ; удаление лок. переменных
pop     EBP           ; восстан. старого EBP
ret     8             ; выход и удал. параметров
CopyS   endp
end
```

Пример 3.136. С помощью директив

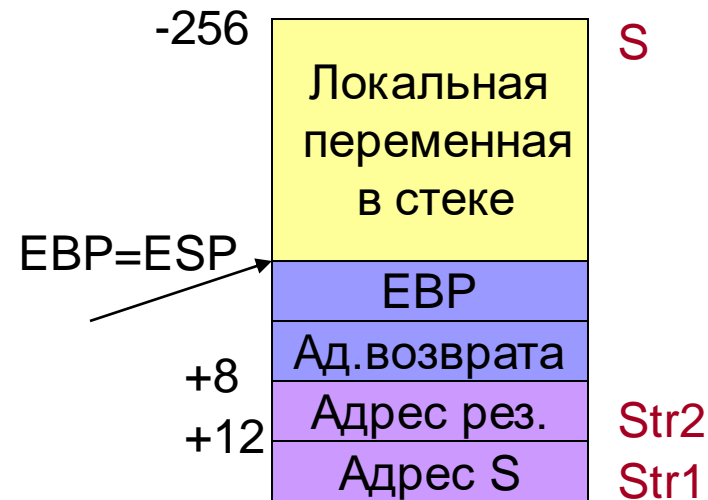
```
.CODE
public    CopyS
externdef    Print:near

CopyS    PROC        NEAR PASCAL PUBLIC USES ESI EDI,
                Str1:PTR DWORD,Str2:PTR DWORD

LOCAL    S[256]:byte

; Копируем строку в локальную память

mov      ESI,Str1
lea      EDI,S
xor      EAX,EAX
lodsb
stosb
mov      ECX,EAX
cld
rep movsb
```



Пример 3.13б. С помощью директив

; Копируем строку в результат

```
    lea     ESI, S
    mov     EDI, Str2
    lodsb
    stosb
    mov     ECX, EAX
    rep movsb
```

; Выводим длину строки

```
    push    EAX
    call    Print
```

```
    ret
CopyS    endp
end
```

3.6.3 Visual C++ – MASM32

- в модуле на Visual C++ подключаемые процедуры и функции должны быть объявлены как внешние **extern** с указанием конвенции связи, например:

```
extern void __pascal add1(int a,int b,int *c);
```

- при ассемблировании должны быть использованы опции:

- ☐ для Masm32: `ml /coff /c add1.asm`

- ☐ для Tasm 5.0: `tasm32 /ml add1.asm`

если ассемблирование выполняется в Visual Studio, то необходимо добавить внешний инструмент **Tools\External Tools...\Add:**

Title:	Masm32
Command:	C:\masm32\bin\ml.exe
Arguments:	/coff /c /F1 \$(ItemPath)
Initial directory:	\$(ItemDir)

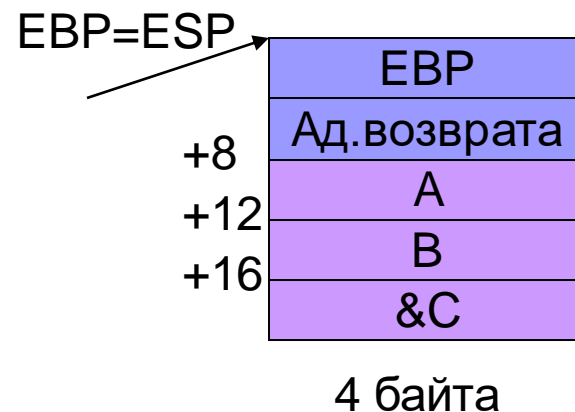
- файл с расширением `.obj` необходимо подключить к проекту посредством пункта меню **Project/Add existing item...**
- вызов процедуры должен оформляться по правилам C++, например:

```
add1(a,b,&c);
```

Пример 3.9 Конвенция `__cdecl`

```
extern "C" void _cdecl add1(int a,int b,int *c);
```

```
.586  
.model flat  
.code  
public _add1  
_add1 proc  
    push    EBP  
    mov     EBP,ESP  
    mov     EAX,[EBP+8]  
    add     EAX,[EBP+12]  
    mov     EDX,[EBP+16]  
    mov     [EDX],EAX  
    pop     EBP  
    ret  
_add1 endp  
end
```



Пример 3.10 Объявление внешних переменных

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>

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;
};
```

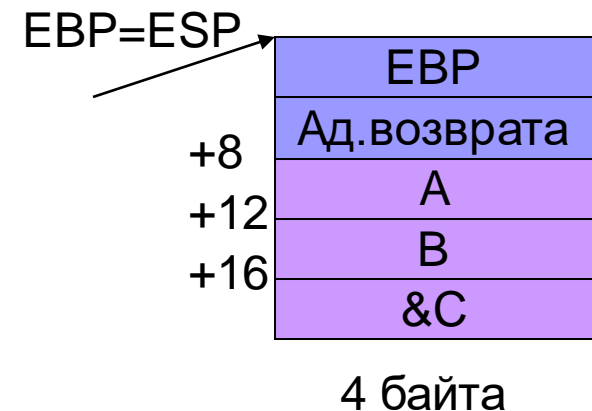
Объявление внешних переменных в процедуре на ассемблере

```
.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
```


Пример 3.11 Конвенция __stdcall

```
extern "C" void __stdcall ADD1(int a,int b,int *c);
```

```
.386
.model flat
.code
public ?ADD1@@YGXHHPAH@Z
?ADD1@@YGXHHPAH@Z proc
    push EBP
    mov EBP,ESP
    mov ECX,[EBP+8]
    add ECX,[EBP+12]
    mov EAX,[EBP+16]
    mov [EAX],ECX
    pop EBP
    ret 12
?ADD1@@YGXHHPAH@Z endp
end
```



Пример 3.12 Конвенция __fastcall

```
extern "C" void __fastcall add1(int a,int b,int *c);
```

```
.386
```

```
.model flat
```

```
.code
```

```
public @ADD1@12
```

```
@ADD1@12 proc
```

```
push EBP
```

```
mov EBP,ESP
```

```
add ECX,EDX
```

```
mov EDX,[EBP+8]
```

```
mov [EDX],ECX
```

```
pop EBP
```

```
ret 4 ; стек освобождает процедура
```

```
@ADD1@12 endp
```

```
end
```

Только два параметра в регистрах ECX и EDX, третий и далее в обратном порядке в стеке

