

# Stack

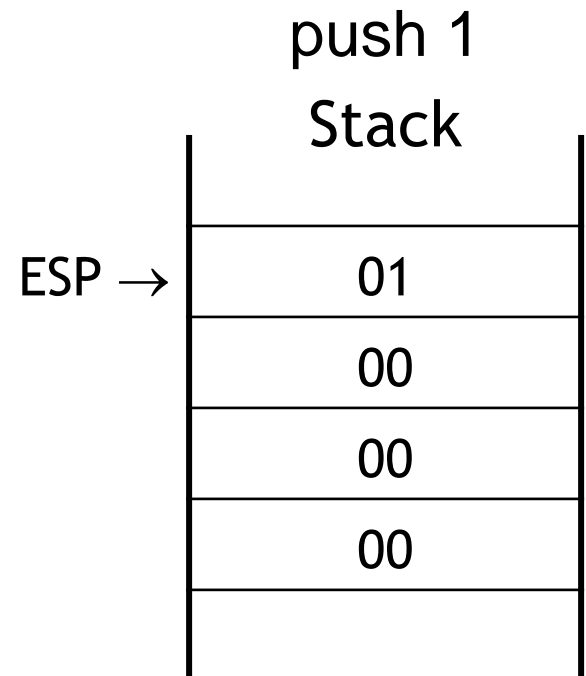
- is a part of memory (stack segment)
- stack segment is defined using the `.stack` directive  
In the included file `SmallWin.inc` in the sample project:  
`.STACK 4096`
- it is managed by the processor using the ESP register

## Stack instructions

push register/memory/number

- stores the operand on the top of the stack.
- The operand must be of type word or dword.
- A number occupies always 32 bits in the 32-bit mode.

$ESP = ESP - 2(4)$ ,  $[ESP] = \text{operand}$



## pop register/memory

- retrieves the operand from the stack.
- The operand must be of type word or dword.

operand = [ESP], ESP = ESP + 2(4)

## pushad

- stores all general-purpose registers on the stack in the order: EAX, ECX, EDX, EBX, ESP (original contents), EBP, ESI, EDI

## popad

- retrieves all general-purpose registers from the stack in the order EDI, ESI, EBP, the next dword is ignored, EBX, EDX, ECX, EAX

pushf

- stores the lower half of the EFLAGS register on the stack

pushfd

- stores the EFLAGS register on the stack

popf

- retrieves the lower half of the EFLAGS register from the stack

popfd

- retrieves the EFLAGS register from the stack

Instructions popf and popfd change flags, the other stack instructions do not.

# Procedures

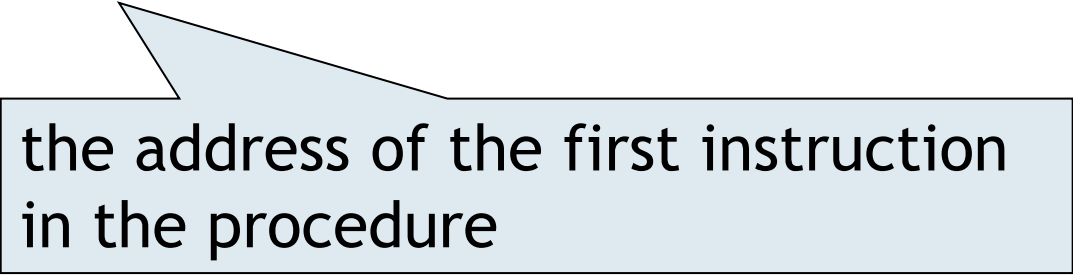
Declaration:

```
name PROC [language] [USES registers] [,parameters]
```

```
    the body of the procedure
```

```
    ret; return to the calling program
```

```
name ENDP
```



the address of the first instruction  
in the procedure

A label is visible only in the procedure in which it is defined.

## Procedure call

Direct: `call name of the procedure`

Indirect: `call register/memory`

- stores the return address on the stack and jumps to the first instruction of the procedure.

The return address is the current contents of the EIP register, i.e. the offset of the instruction following the `call` instruction.

`call` is compiled in the same way as `jmp`.

## Procedure return

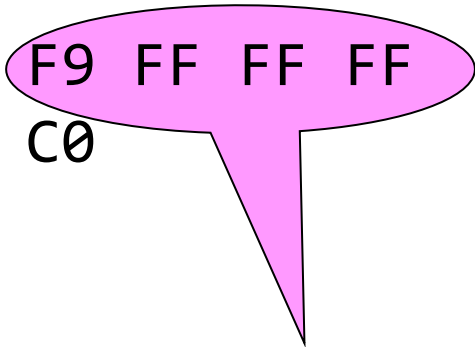
```
ret [number]
```

- retrieves the return address from the stack and stores it to EIP.
- If the instruction has a direct operand (number), it is added to the ESP register after the return address is retrieved from the stack.

Machine code:

004033F0 90  
004033F1 C3

004033F2 E8 F9 FF FF FF  
004033F7 33 C0

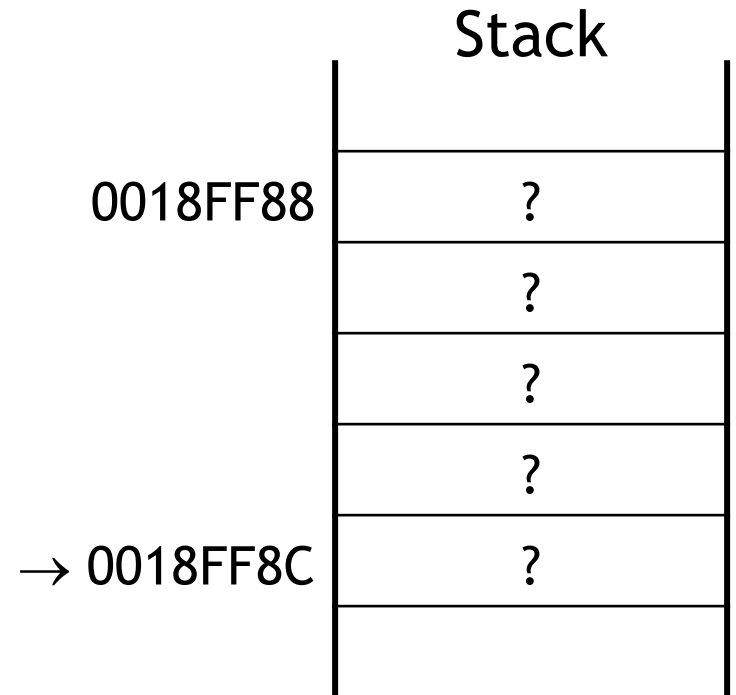
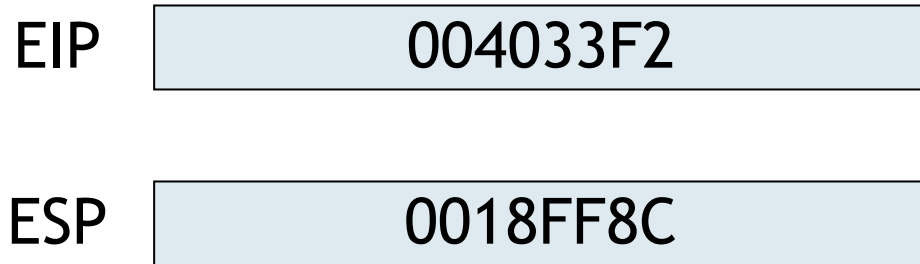


displacement: -7

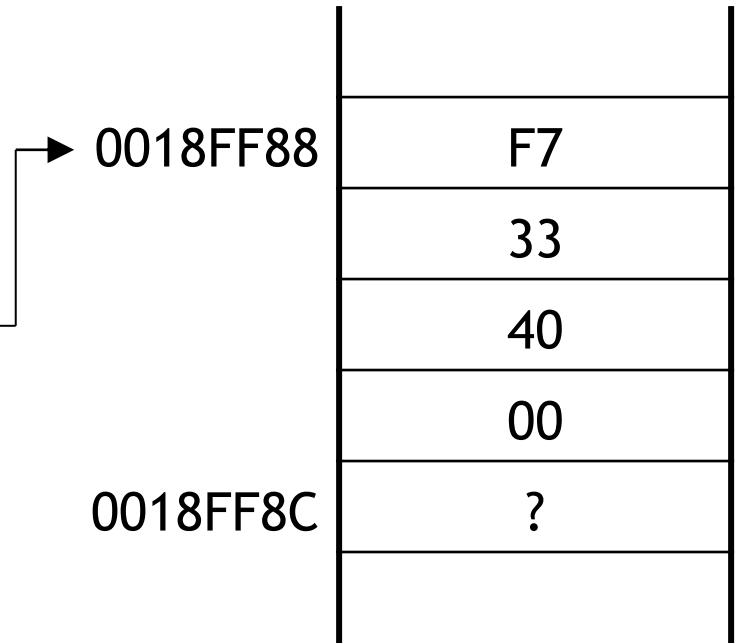
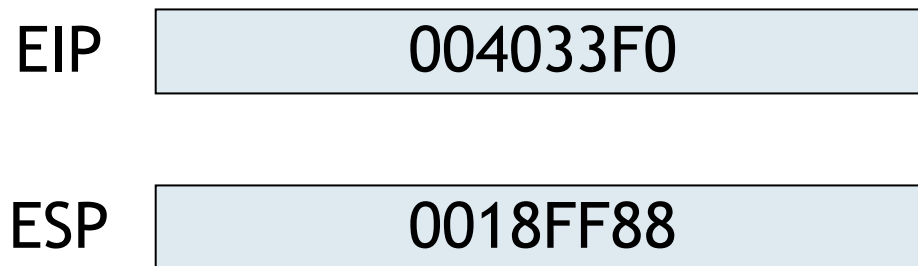
```
.code  
Nothing PROC  
    nop  
    ret  
Nothing ENDP  
  
main PROC  
    call Nothing  
    xor eax,eax
```



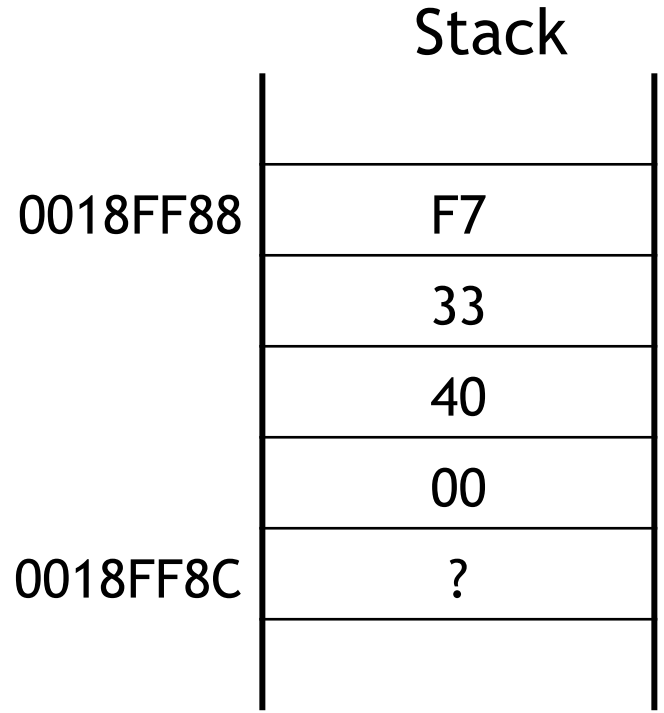
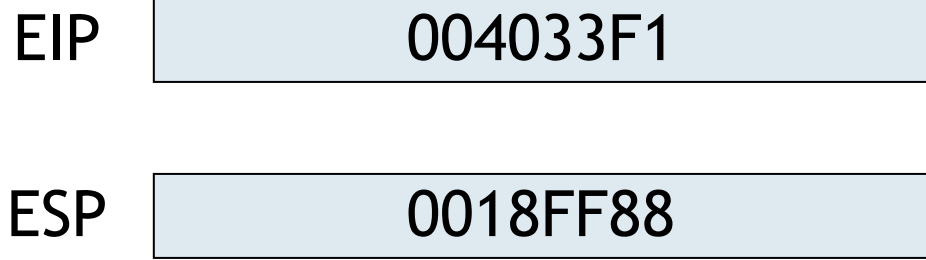
At the beginning of the main procedure:



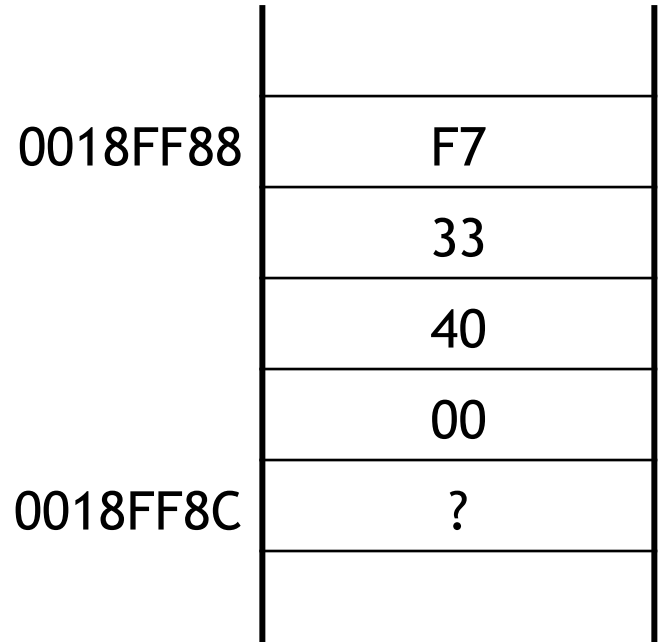
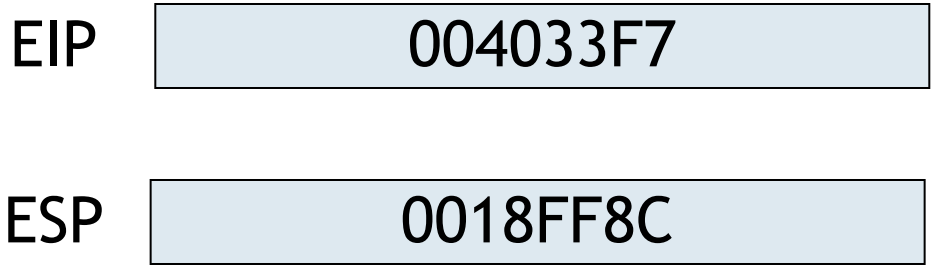
After call Nothing:



After nop:



After ret:



## Procedures with parameters

Parameters may be passed:

- in general-purpose registers
  - in pure assembly language programs
  - fast
  - disadvantage: the original contents of the registers must be saved before the parameters are loaded into them
- in the stack

## Passing parameters in the stack

Before the procedure call, the parameters are pushed on the stack. In the procedure, the parameters are accessed using the indirect addressing mode with the base register EBP.

Two ways to pass parameters:

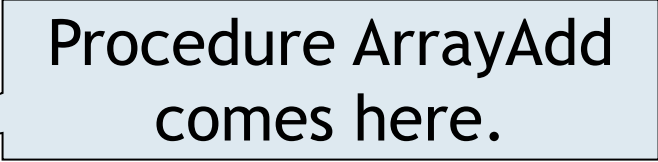
- **pass by value** - the caller passes the value of the parameter; input-only parameters.
- **pass by reference** - the caller passes the address of a variable; is useful when the procedure modifies the actual parameter or when you pass large data structures between procedures.

- Write a procedure that adds a given number to each element of an array of type byte. Parameters of the procedure include:
  - the address of the array
  - the length of the array (in bytes)
  - the number to be added

The address is passed by reference, the other parameters are passed by value.

```
.data
Array DB 0,1,2,3,4
ArrayLength DD lengthof Array
Number EQU 1
```

```
.code
```



Procedure ArrayAdd  
comes here.

```
main PROC
    push offset Array
    push ArrayLength
    push Number
    call ArrayAdd
    exit
main ENDP
```

Stack after call ArrayAdd:

ESP	return address
ESP + 4	Number
ESP + 8	ArrayLength
ESP + 12	offset Array
ESP + 16	?

## Procedure:

## Stack after call ArrayAdd:

```
ArrayAdd PROC
```

```
mov ebp,esp
```

```
mov ebx,[ebp+12]; offset
```

```
mov ecx,[ebp+8]; length
```

```
mov al,[ebp+4]; number
```

```
Next:
```

```
add [ebx],al
```

```
inc ebx
```

```
loop Next
```

```
ret 12 ; retrieves the return address and adds 12  
       ; to ESP to discard the parameters  
       ; from the stack
```

```
ArrayAdd ENDP
```

ESP	return address
ESP + 4	Number
ESP + 8	ArrayLength
ESP + 12	offset Array
ESP + 16	?

## Formal parameters

Directive PROC allows you to assign formal names to the parameters. Then the procedure accesses the parameters using their names instead of the indirect addresses with the EBP register.

Advantages:

- more readable source code in the procedure
- you need not remember the offset of the parameter relative to the top of the stack.

You may define the type for each formal parameter. If the type is not specified, dword is supposed.



# Calling procedures according to a high-level language conventions

Language	Parameters are pushed on the stack	Who discards the parameters
Pascal Basic Fortran	from left to right	procedure (using <code>ret n</code> )
C Prolog	from right to left	calling program ( <code>add esp, n</code> )
stdcall (in 32-bit applications, when Windows services are called)	from right to left	procedure

## Procedure with formal parameters and stdcall specification

In Irvine32.inc: INCLUDE SmallWin.inc

In SmallWin.inc: `.MODEL flat, stdcall`

ArrayAdd PROC paOffset, paLength, paNumber:byte

```
    mov ebx,paOffset
```

```
    mov ecx,paLength
```

```
    mov al,paNumber
```

Next:

```
    add [ebx],al
```

```
    inc ebx
```

```
    loop Next
```

```
    ret
```

ArrayAdd ENDP

The compiler automatically includes at the beginning of the procedure:

```
push ebp
```

```
mov ebp,esp
```

and at the end of the procedure:

```
leave
```

```
ret 0Ch
```

```
mov esp,ebp
```

```
pop ebp
```

## Calling procedure with actual parameters

```
main PROC  
    INVOKE ArrayAdd, offset Array, ArrayLength, Number  
    exit  
main ENDP
```

```
INVOKE is translated into:  
push 1  
push dword ptr ds:[406005h]  
push 406000h  
call ArrayAdd
```

### *Comment*

The procedure must be declared before INVOKE.

## Preserving affected registers in the procedure

```
ArrayAdd PROC USES eax ebx ecx paOffset, paLength,  
paNumber:byte  
    mov ebx,paOffset  
    mov ecx,paLength  
    mov al,paNumber  
Next:  
    add [ebx],al  
    inc ebx  
    loop Next  
    ret  
ArrayAdd ENDP
```

The compiler automatically includes  
at the beginning of the procedure:

```
push ebp  
mov ebp,esp  
push eax  
push ebx  
push ecx
```

and at the end of the procedure:

```
pop ecx  
pop ebx  
pop eax  
leave  
ret 0Ch
```

## Local variables

Local variables exist only during the procedure execution; they disappear before the procedure returns.

Local variables are allocated in the stack above the return address and the stored EBP from the main program.

Syntax:

```
LOCAL variable1 [, variable2] ...
```

Example:

```
LOCAL Sum:byte, String[8]:byte
```

```
lea edx,String; store offset of String into edx
```

- Modify the procedure ArrayAdd so that to sum all elements of the Array. The sum will be in the local variable Sum.

```
ArrayAdd PROC USES eax ebx ecx paOffset, paLength,  
paNumber:byte
```

```
    LOCAL Sum:byte
```

```
    mov ebx,paOffset
```

```
    mov ecx,paLength
```

```
    mov al,paNumber
```

```
    mov Sum,0
```

```
Next:
```

```
    add [ebx],al
```

```
    mov ah,[ebx]
```

```
    add Sum,ah
```

```
    inc ebx
```

```
    loop Next
```

```
    ret
```

```
ArrayAdd ENDP
```

The compiler automatically includes at the beginning of the procedure:

```
push ebp
```

```
mov ebp,esp
```

```
add esp,0FFFFFFCh; -4
```

```
push eax
```

```
push ebx
```

```
push ecx
```

and at the end of the procedure:

```
pop ecx
```

```
pop ebx
```

```
pop eax
```

```
leave
```

```
ret 0Ch
```

Stack after add esp, -4:

locations that can be used in the procedure	
ESP	Sum
EBP	caller's EBP
EBP + 4	return address
EBP + 8	offset Array
EBP + 12	ArrayLength
EBP + 16	Number
EBP + 20	?