Any Questions so Far?

Multiplication

- Why is multiplication more tricky than addition/subtraction?
- It can produce double length results.
- Signed and Unsigned behaviour is different!

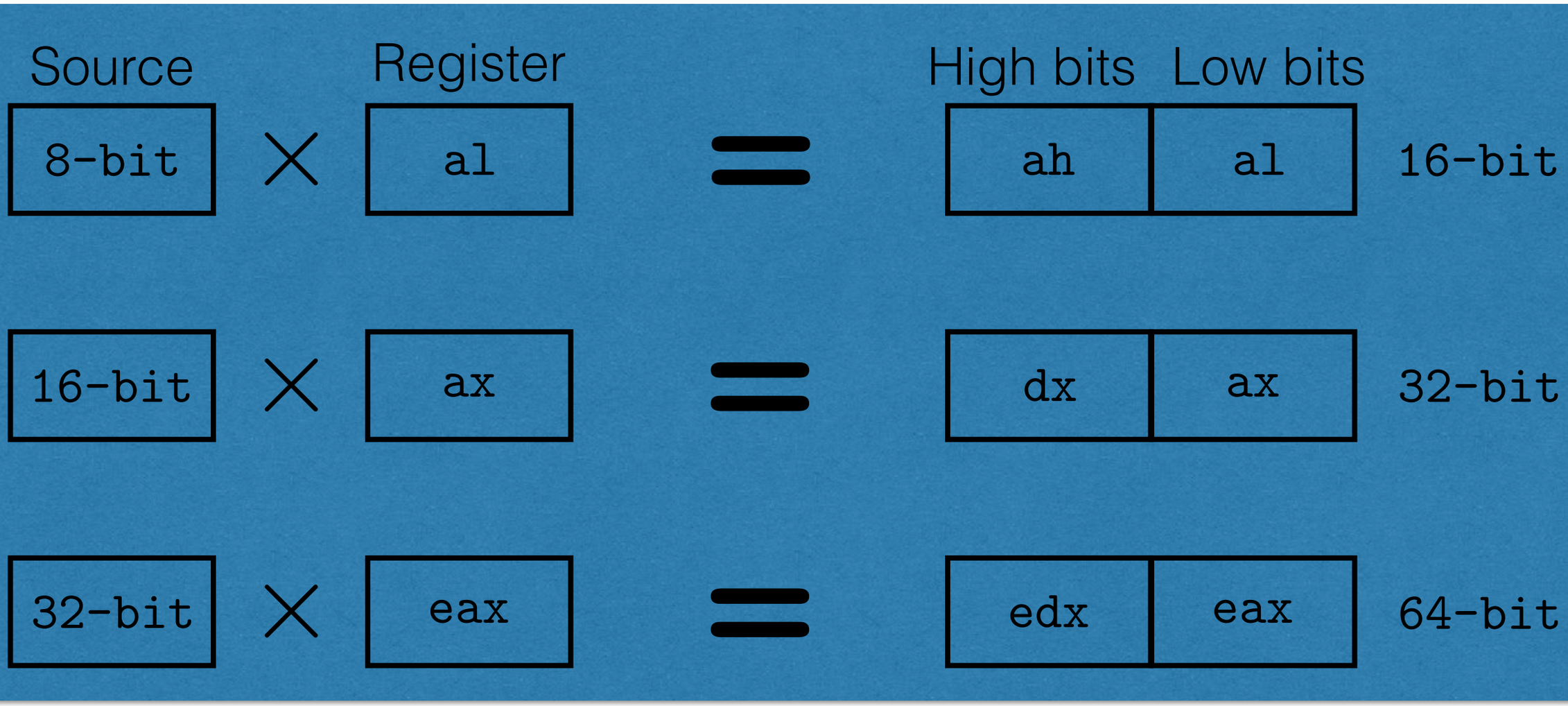
Multiplication instruction format. Destination is in (part of) EAX register:

```
mul     source  ;(unsigned multiplication) source can be a
                ;register or memory (immediate not allowed)

imul    source  ;source can be a register or
                ;memory (immediate not allowed)
```

Multiplication

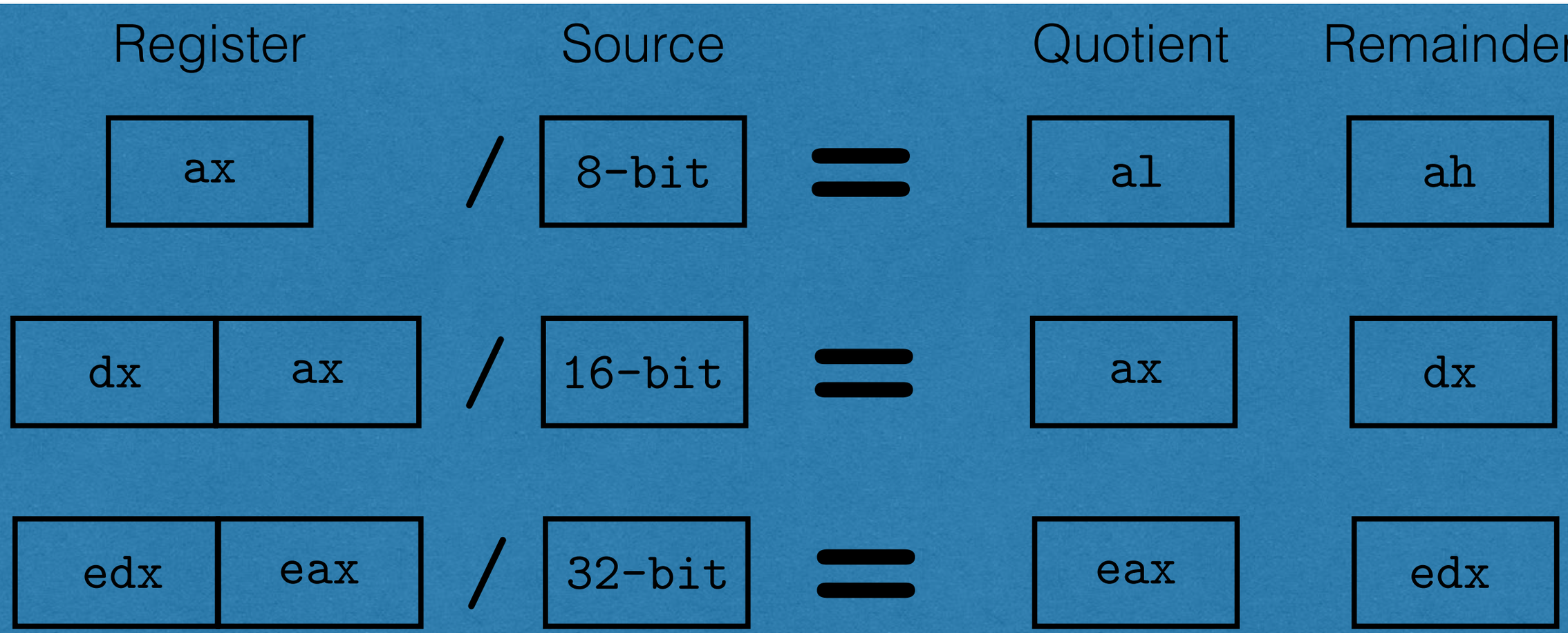
- It supports 8-bit, 16-bit and 32-bit source.



- The carry and overflow flags are set if the HB are non-zero and are cleared otherwise.

Division

- Why is division more tricky than multiplication?
- We have two results… and overflow…



Flags are not set!

Signed Division

- If signed division, we need to sign extend the dividend in the registers:

E.g. 16-bit source case: -251 = FFFF FF14 $\Longrightarrow$ ax=FF14, dx=FFFF

- Functions `cbw, cwd, cdq` exist precisely for this purpose:

```
; Perform -91/14
mov   al, -91     ; al=-91
cbw               ; sign-extending al into ah
mov   cl, 14      ; load source into cl
idiv  cl          ; al=-6, ah=-7
```

PTR Directive

- Sometimes the assembler can't detect automatically the size of an operation and we have to provide it explicitly

```
mov eax,    4                ; eax = 4
mov ebx,    OFFSET array     ; move start address of array to ebx
mov [ebx], eax               ; array[0] = 4
inc [ebx]                    ; ERROR, operand must have size! You want to increase the
                             ; value pointed by ebx, but is this 8-bit, 16-bit or 32-bit?
inc DWORD PTR [ebx]          ; ebx = 5
```

The PTR directive tells specifically what size is the operand to be written in the memory location: BYTE, WORD, DWORD, etc…

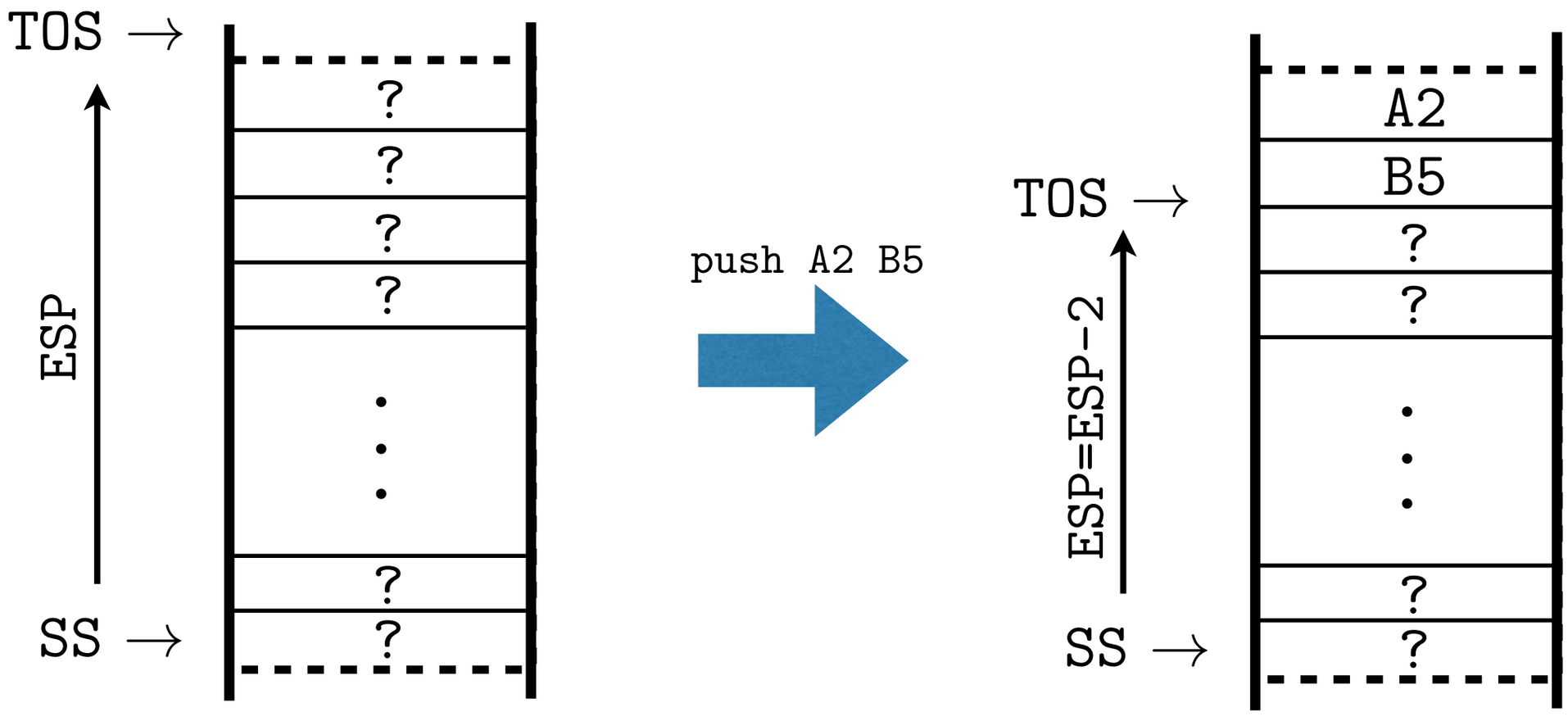
```
mov [ebx], 12
```

The assembler can't figure out the size of the operand.
But most importantly you can't have two operands from memory!
(In this case memory/immediate!)

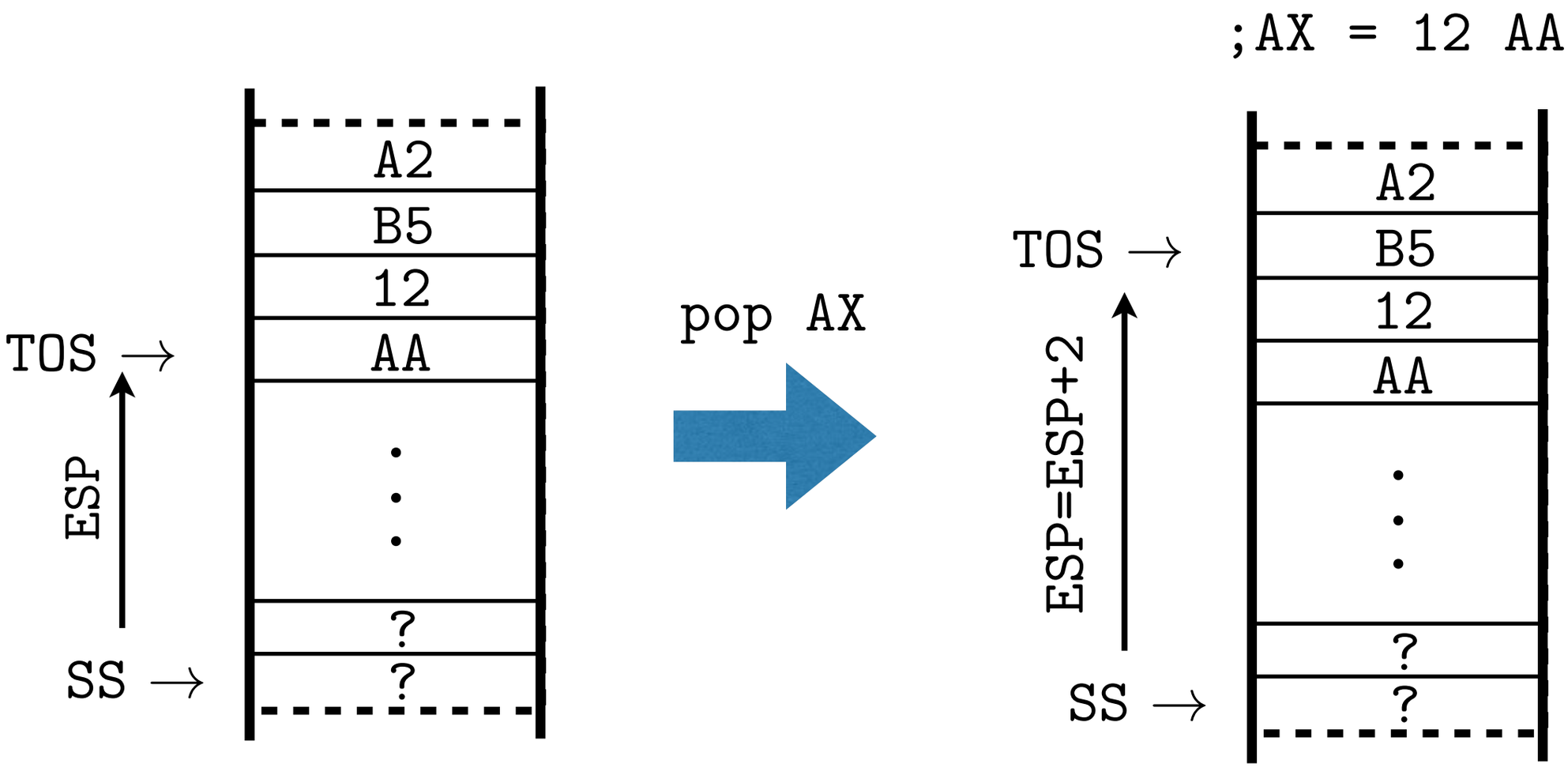
```
    mov [ebx], 12           ; NOT ALLOWED!
    mov eax  , 12           ; eax=12
    mov [ebx], eax          ; array[0]=12
    inc DWORD PTR [ebx] ; array[0]=13
```

The Stack

- Memory is accessed by the assembler as a stack structure (LIFO).
- SS points to the start of the stack segment in memory. Memory access in the stack are relative offsets to the SS.
- ESP points to the top of the stack. The Stack grows **down** in space!



The Stack: pop



- In IA-32 you can write only words and double words into stack.
- Stack is used for temporary storage of variables.

The Stack: Example

- Exchange two words stored in memory:

```
value1 DW 12
value2 DW 15
```

```
xchg value1, value2    ;memory-memory, illegal!
```

The Stack: Example

- Exchange two words stored in memory:

```
value1 DW 12
value2 DW 15
```

```
push  value1
push  value2
pop   value1
pop   value2
```

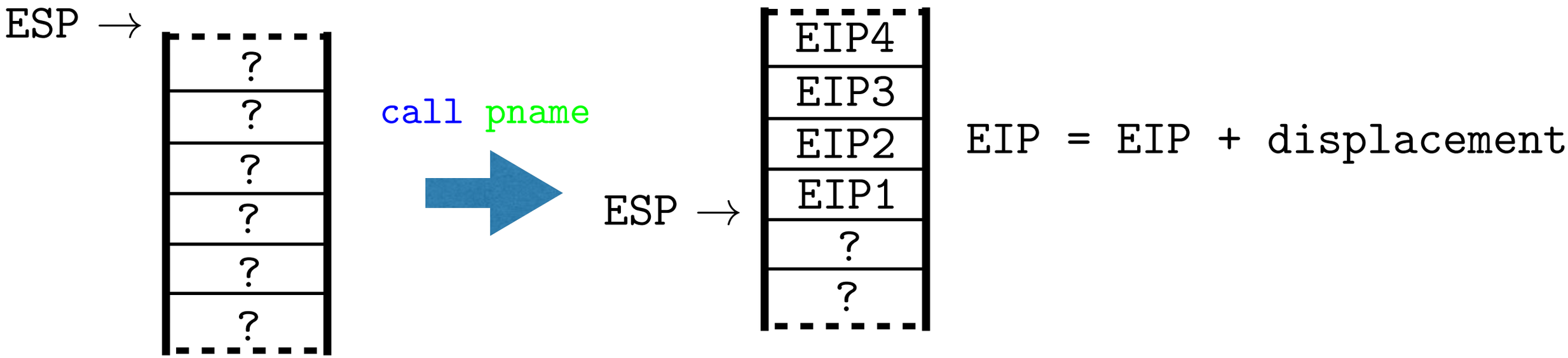
Procedure Calls

```
proc_name PROC
    ;here goes the procedure body
    ret 0
proc_name ENDP
```

- Procedure are called with: `call proc_name`
- What is the behaviour of call/ret?
- How are parameters passed?
- What happens to registers?
- How are local variables managed?

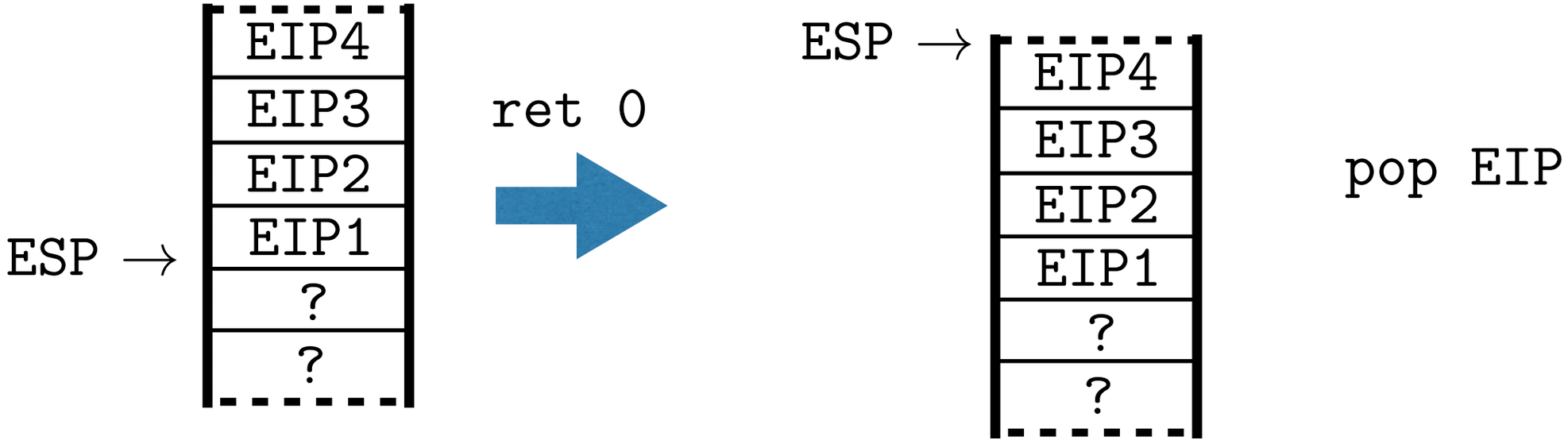
How is the Call handled?

- The `eip` register is used to keep track of the memory location of the following instruction to be fetched.
- Upon fetching a procedure call, `eip` is updated to point to the first instruction in the procedure call.
- The old value of `eip` is pushed onto the stack by the assembler to remember where to pick up computations from upon return.



Return

- Upon return the old value of `eip` is popped from the stack and ESP is updated accordingly



Parameter Passing using Registers

```
main:
    mov eax, 12
    mov ebx, 15
    call sum
    push eax
    print "The result of the sum is:"
    pop eax
    print str$(eax),13,10
end main

sum PROC
    add eax, ebx
sum ENDP
```

- Intuitive method but limited by number of available registers…

Parameter Passing using the Stack

- Alternatively procedure's parameters can be pushed onto the stack and accessed by the procedure from there.

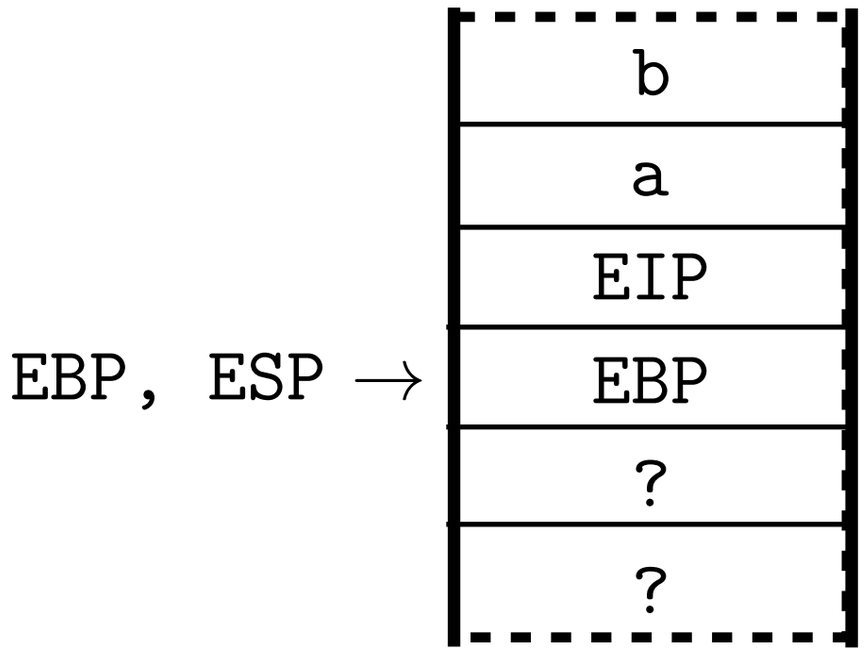
- Function parameters are pushed from right to left. [Why?]

```
int my_sum(int a, int b)
```

```
push b
push a
call sum
```

- ebp is used to access the parameters from the stack in the function.

```
push ebp
mov  ebp, esp
mov  eax, [ebp+8] ;eax:=a
```

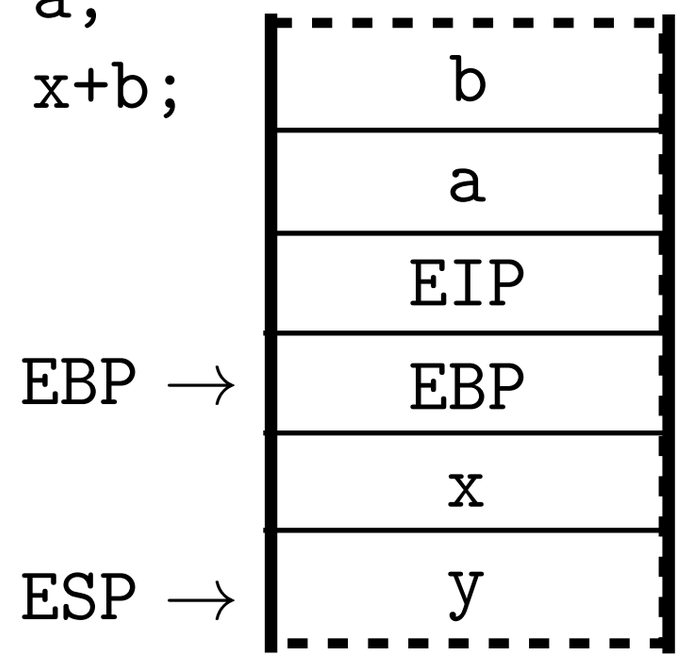


Local Variables

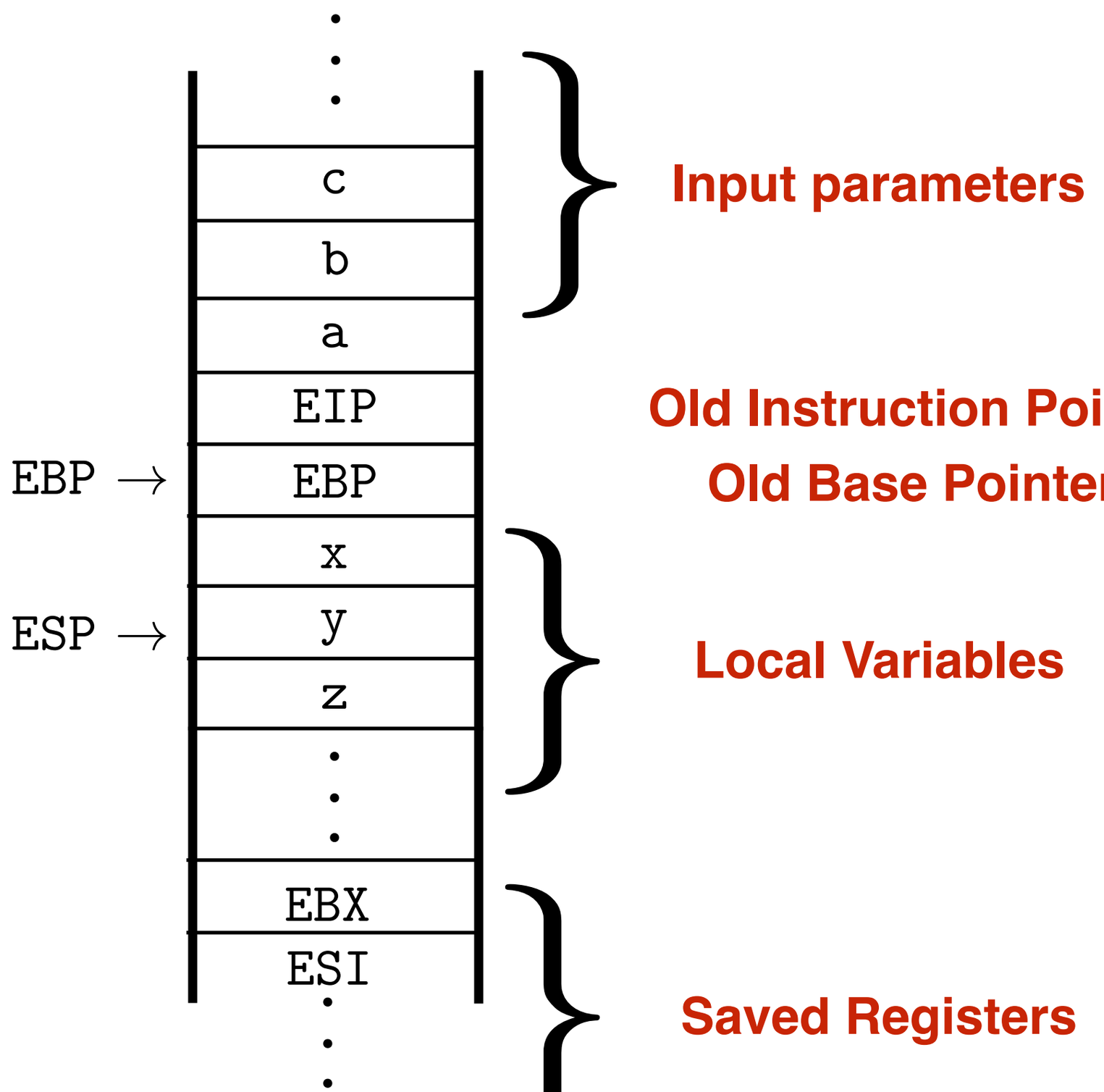
- The stack can be used also to create and access local variables.

```
int my_sum(int a, int b){
int x;
int y;
x = a;
y = x+b;
...
}
```

```
sub esp, 8          ;allocate space
mov eax, [ebp+8]    ;eax=a
mov [ebp-4], eax    ;x=a
mov eax, [ebp+12]   ;eax=b
mov [ebp-8], eax    ;y=b
...
```



Typical Procedure Stack



Calling Conventions

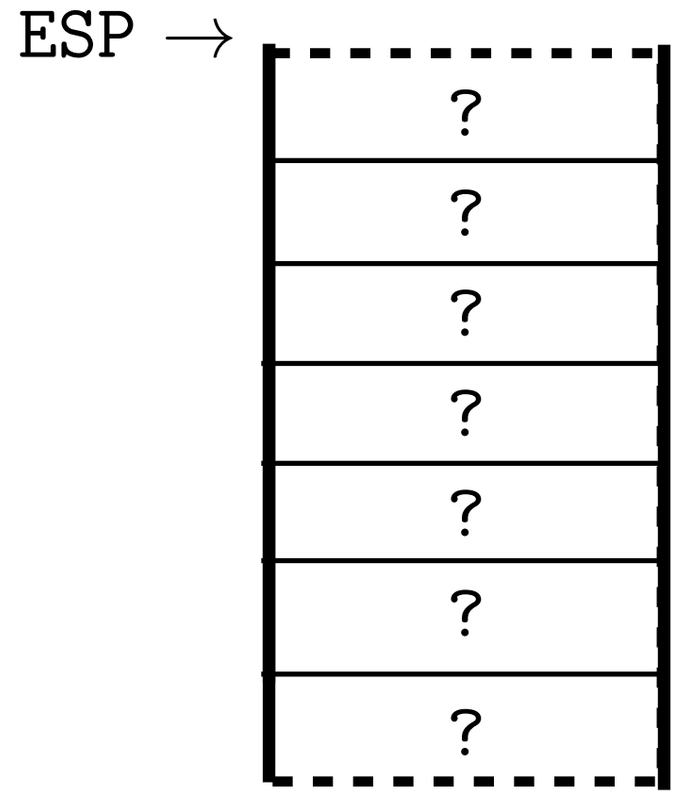
- Who cleans the stack, the caller or the callee? Which registers are preserved?

- Various calling convention have been developed and made standard for various applications. They function as “contract” between the caller and the callee, it is the programmer responsibility to follow them properly!

Calling Conventions

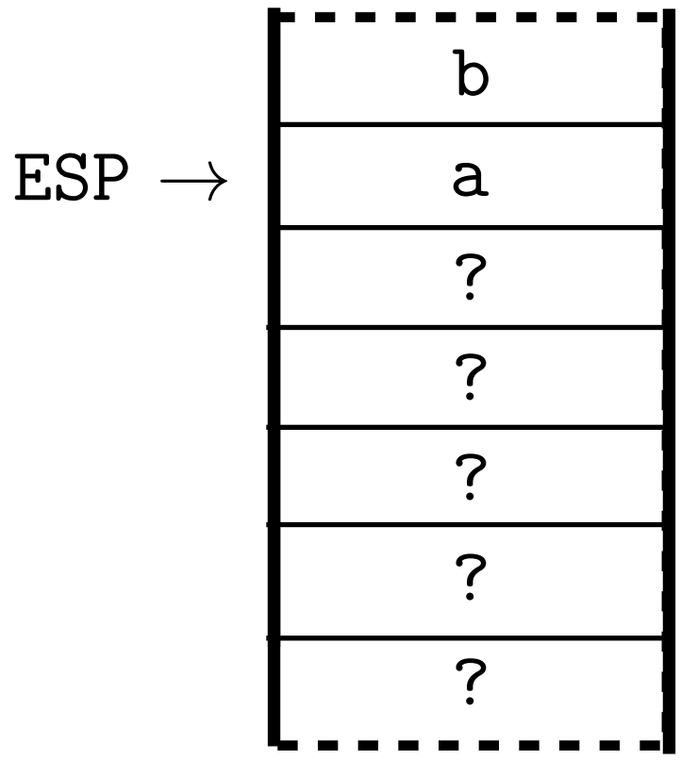
- stdcall [used in Win32]:
    - EDI, ESI, EBP and EBX registers **preserved** across the call.
    - EDX:EAX used for return values.
    - **Callee** responsible for cleaning the stack.
- _cdecl [used for C programs]:
    - EDI, ESI, EBP and EBX registers **preserved** across the call.
    - EDX:EAX used for return values.
    - **Caller** responsible for cleaning the stack.

Function Calling Example _cdecl (not optimised example)



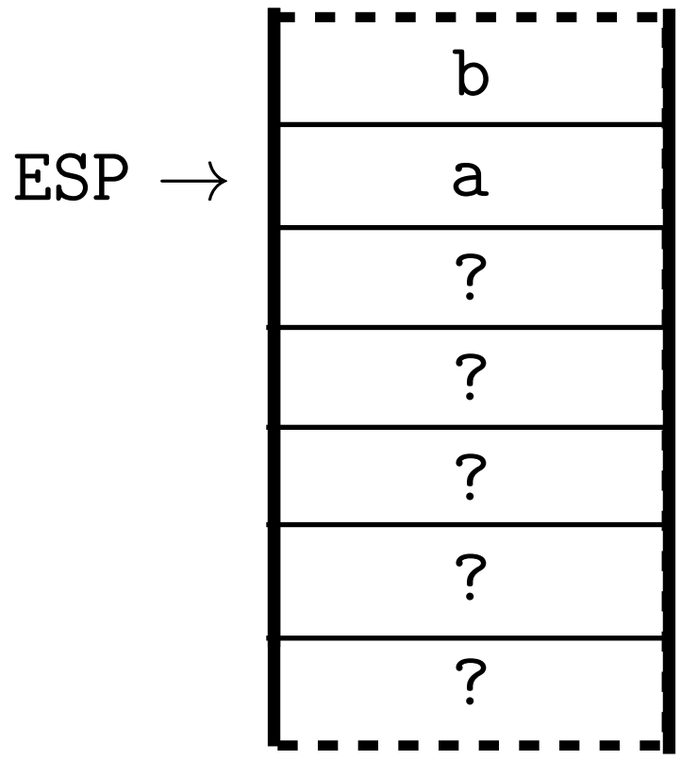
Function Calling Example _cdecl (not optimised example)

```
push b
push a
```



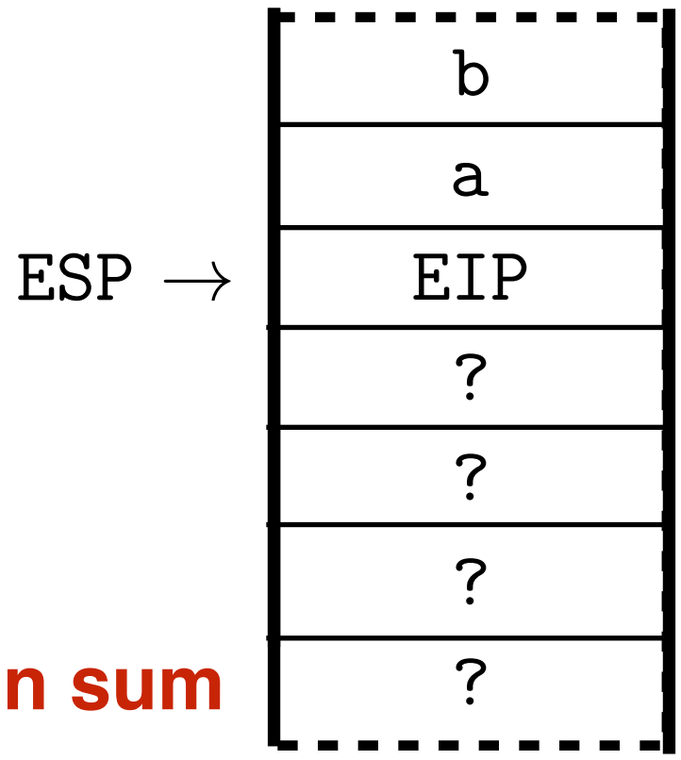
Function Calling Example _cdecl (not optimised example)

```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
push b
push a
call sum
```

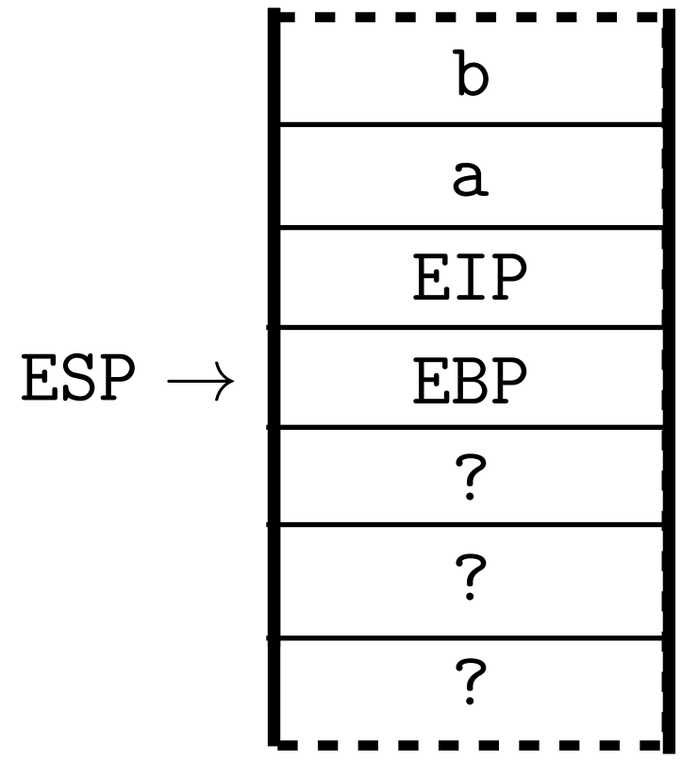


EIP gets updated to point to the instruction in sum

Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
```

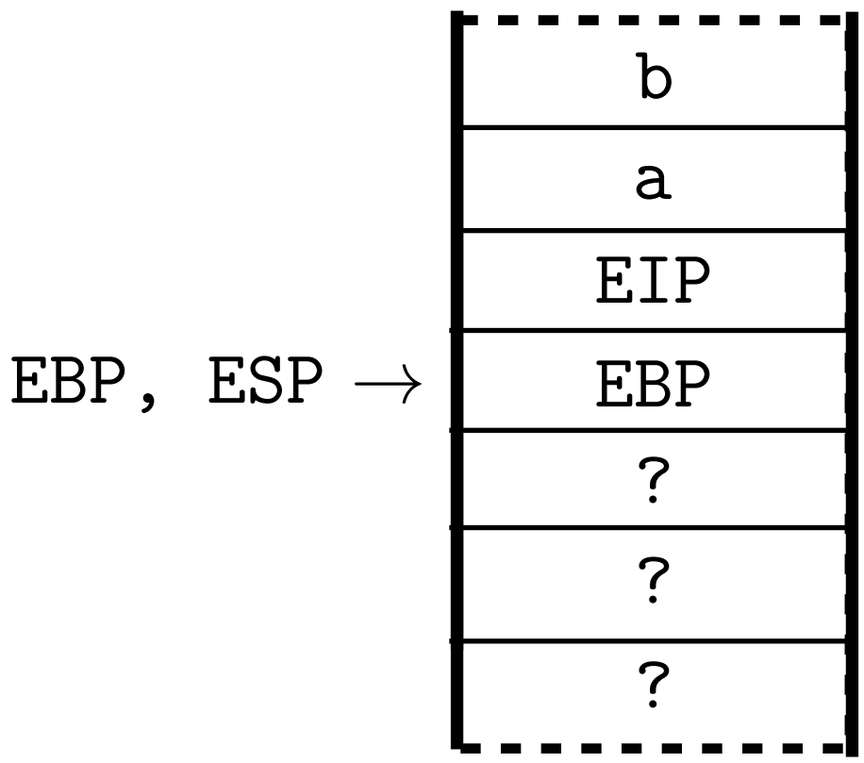
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
```

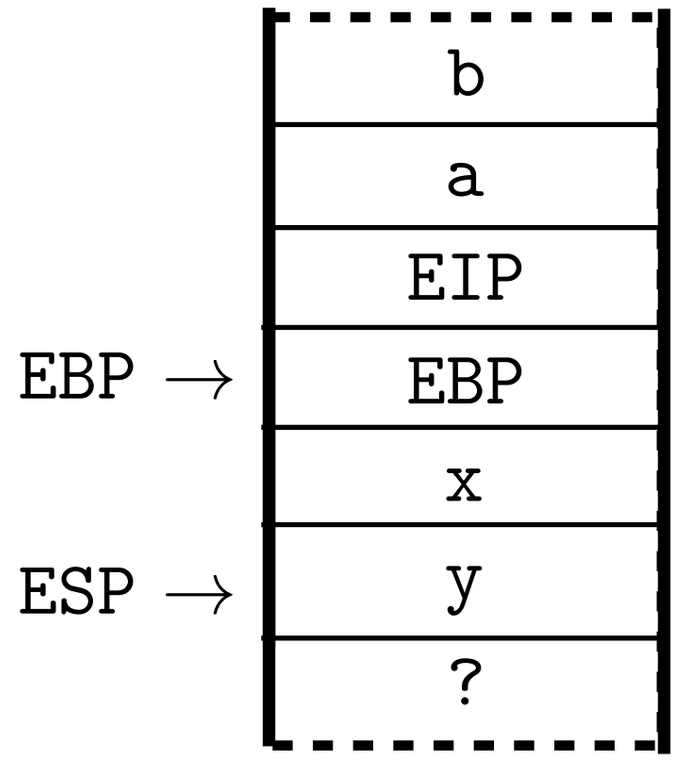
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
```

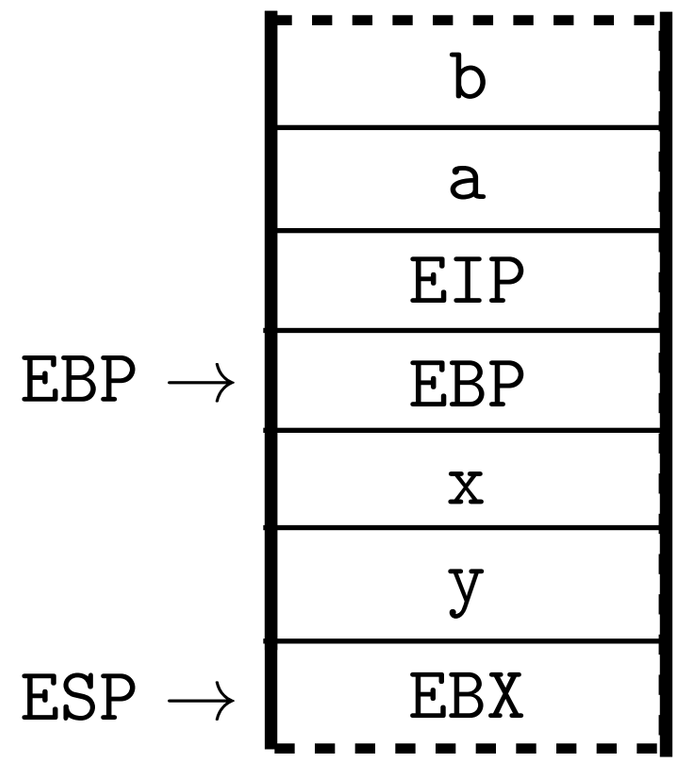
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
```

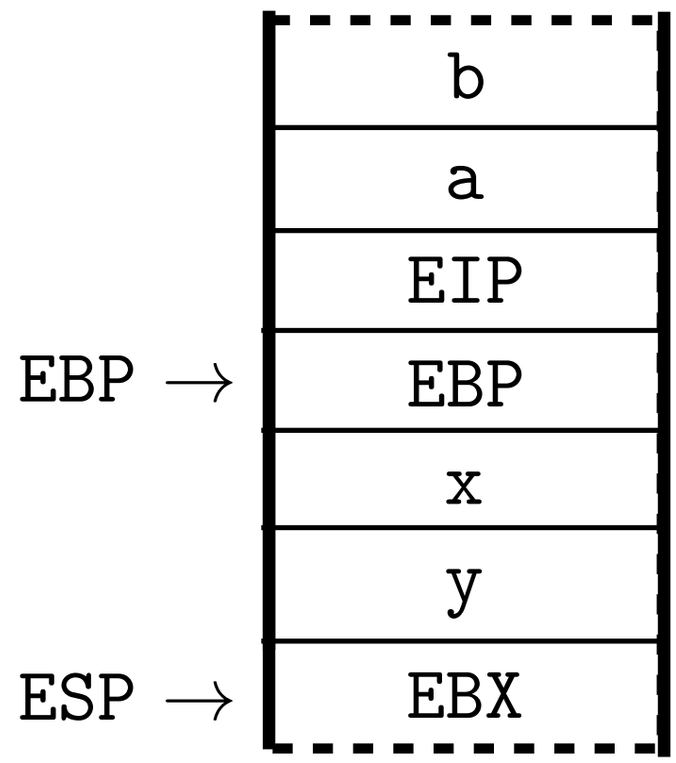
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]   ; ebx=a
mov [ebp-4], ebx   ; x=ebx=a
mov eax, [ebp+12]  ; eax=b
add eax, [ebp-4]   ; eax=eax+x=b+x
mov [ebp-8], eax   ; y=eax=x+b
```

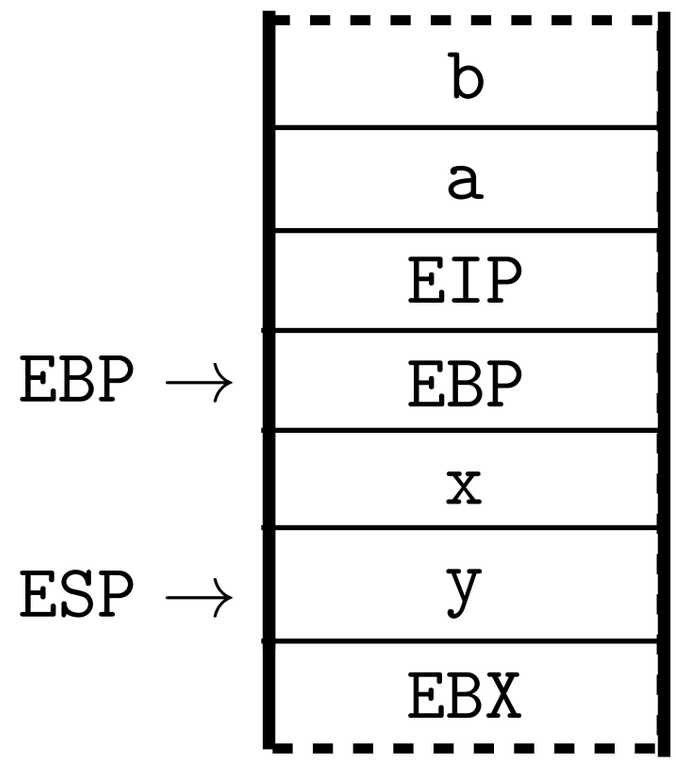
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]   ; ebx=a
mov [ebp-4], ebx   ; x=ebx=a
mov eax, [ebp+12]  ; eax=b
add eax, [ebp-4]   ; eax=eax+x=b+x
mov [ebp-8], eax   ; y=eax=x+b
pop ebx             ; restoring ebx
; here I'd restore any other register
; that needs preserving
```

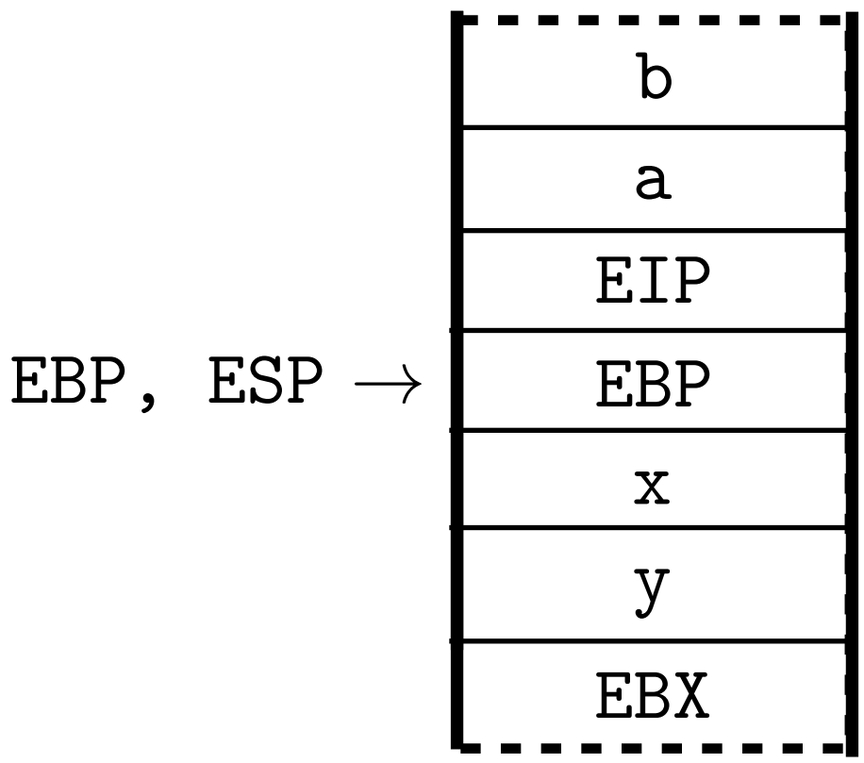
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]    ; ebx=a
mov [ebp-4], ebx    ; x=ebx=a
mov eax, [ebp+12]   ; eax=b
add eax, [ebp-4]    ; eax=eax+x=b+x
mov [ebp-8], eax    ; y=eax=x+b
pop ebx             ; restoring ebx
; here I'd restore any other register
; that needs preserving
add esp, 8          ; clearing local variables
```

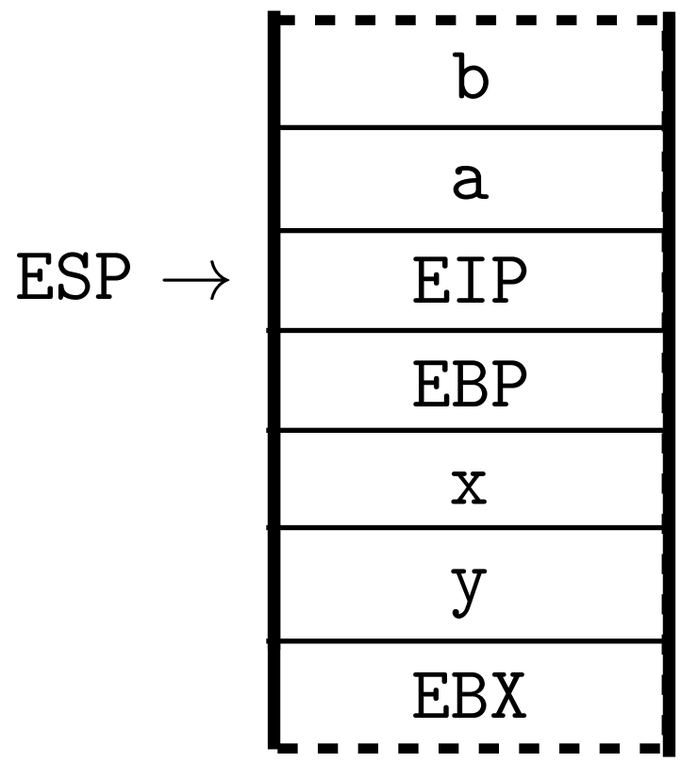
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]    ; ebx=a
mov [ebp-4], ebx    ; x=ebx=a
mov eax, [ebp+12]   ; eax=b
add eax, [ebp-4]    ; eax=eax+x=b+x
mov [ebp-8], eax    ; y=eax=x+b
pop ebx             ; restoring ebx
; here I'd restore any other register
; that needs preserving
add esp, 8          ; clearing local variables
pop ebp             ;restoring ebp
```

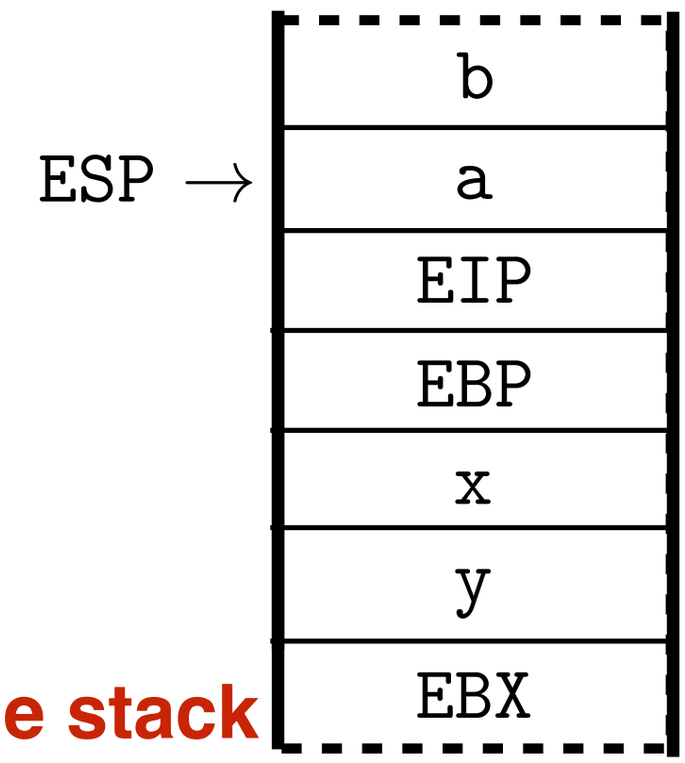
```
push b
push a
call sum
```



Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]    ; ebx=a
mov [ebp-4], ebx    ; x=ebx=a
mov eax, [ebp+12]   ; eax=b
add eax, [ebp-4]    ; eax=eax+x=b+x
mov [ebp-8], eax    ; y=eax=x+b
pop ebx             ; restoring ebx
; here I'd restore any other register
; that needs preserving
add esp, 8          ; clearing local variables
pop ebp             ;restoring ebp
ret 0
sum ENDP
```

```
push b
push a
call sum
```



EIP gets restored from the stack

Function Calling Example _cdecl (not optimised example)

```
sum PROC
push ebp
mov ebp, esp
sub esp, 8          ; local variable space
push ebx            ; NB: not really needed
;Push here any other registers
;that need preserving
mov ebx, [ebp+8]    ; ebx=a
mov [ebp-4], ebx    ; x=ebx=a
mov eax, [ebp+12]   ; eax=b
add eax, [ebp-4]    ; eax=eax+x=b+x
mov [ebp-8], eax    ; y=eax=x+b
pop ebx             ; restoring ebx
; here I'd restore any other register
; that needs preserving
add esp, 8          ; clearing local variables
pop ebp             ;restoring ebp
ret 0
sum ENDP
```

```
push b
push a
call sum
;clearing the stack
add  esp, 8
;function output is in eax
```