Computer Architecture and System Programming Laboratory

TA Session 3

EFLAGS

SHIFT

Little Endian

Sections (.bss, .data, .rodata, .text, Heap, Stack)

Stack (push, pop)

C calling convention

gnu debugger – basics

EFLAGS – Flags / Status Register

- **flag** is a single independent bit of (status) information
- each flag has a **two-letter symbol name**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Status Flags

CF = Carry flag

PF = Parity flag

AF = Auxiliary carry flag

ZF = Zero flag

SF = Sign flag

OF = Overflow flag

Control flag

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

CF - Carry Flag

CF gets ‘1’ if **result is larger than "capacity" of target operand**

$11111111_b + 00000001_b = 1\underbrace{00000000_b}_{} = 0$ → CF = 1

CF gets ‘1’ if **subtract larger number from smaller** (borrow out of “capacity” of target operand)

$00000000_b - 00000001_b \equiv$
$100000000_b - 00000001_b \equiv 11111111_b$

$00000000_b - 00000001_b = 11111111_b$ → CF = 1

```
char x=0b11111111;  // x=-1;
unsigned char x=0b11111111; // x=255;
```

unsigned arithmetic → $11111111_b = 255$ (decimal) → got wrong answer
signed arithmetic → $11111111_b = -1$ (decimal) → ignore CF flag

otherwise CF gets ‘0’

- **In unsigned arithmetic, watch CF flag to detect errors.**
- In signed arithmetic ignore CF flag.

OF – Overflow Flag

addition of two positive numbers results negative → OF gets ‘1’

$\mathbf{0}1000000_b + \mathbf{0}1000000_b = 10000000_b$ → OF = 1

addition two negative numbers results positive → OF gets ‘1’

$\mathbf{1}0000000_b + \mathbf{1}0000000_b = 100000000_b = 0$ → OF = 1

otherwise OF gets ‘0’

$01000000_b + 00010000_b = 01010000_b$ → OF = 0
$01100000_b + 10010000_b = 11110000_b$ → OF = 0
$10000000_b + 00010000_b = 10010000_b$ → OF = 0
$11000000_b + 11000000_b = 10000000_b$ → OF = 0

- **In signed arithmetic, watch OF flag to detect errors.**
- In unsigned arithmetic ignore OF flag.

ZF, SF, PF, and AF Flags

ZF – **Zero Flag -** set if a result is zero; cleared otherwise

mov eax, 0
add eax, 0 → ZF = 1

SF – **Sign Flag – set if a result is negative** (i.e. MSB of the result is 1) ; cleared otherwise

$00000000_b - 00000001_b = 11111111_b \rightarrow SF = 1$

PF – **Parity Flag -** set if low-order eight bits of result contain even number of 1-bits; cleared otherwise

$1111111111010000_b + 0000000000001000_b = 1111111111011000_b \rightarrow$ 4 bits are ‘1’ → PF = 1)
$1111000011000000_b + 0000000000001000_b = 1111000011001000_b \rightarrow$ 3 bits are ‘1’ → PF = 0)

AF – **Auxiliary Carry Flag** (or **Adjust Flag**) is set if there is a carry from low nibble (4 bits) to high nibble or a borrow from a high nibble to low nibble

$0000\underbrace{1111}_{\text{nibble}}{}_b + 0000\underbrace{0001}_{\text{nibble}}{}_b = 00010000_b \rightarrow AF = 1$

$000\underbrace{10000}_{\text{nibble}}{}_b - 000\underbrace{0001}_{\text{nibble}}{}_b = = 00001111_b \rightarrow AF = 1$

Instructions seen till now – affecting flags

MOV NOT JMP

does not affect flags

AND OR

OF and CF flags are cleared; SF, ZF, and PF flags are set according to the result.
The state of AF flag is undefined.

DEC INC

OF, SF, ZF, AF, and PF flags are set according to the result. CF flag is not affected.

ADD ADC SUB SBB NEG CMP

OF, SF, ZF, AF, PF, and CF flags are set according to the result.

full set of assembly instructions may be found here

ADC - add integers with carry

Example:

adc AX, BX ;(AX = AX+BX+CF)

SBB - subtract with borrow

Example:

sbb AX, BX ;(AX = AX-BX-CF)

Instructions seen till now – affecting flags

MOV NOT JMP

does not affect flags

AND OR

OF and CF flags are cleared; SF, ZF, and PF flags are set according to the result. The state of AF flag is undefined.

DEC INC

OF, SF, ZF, AF, and PF flags are set according to the result. CF flag is not affected.

ADD ADC SUB SBB NEG CMP

OF, SF, ZF, AF, PF, and CF flags are set according to the result.

full set of assembly instructions may be found here

ADC - add integers with carry

Example:

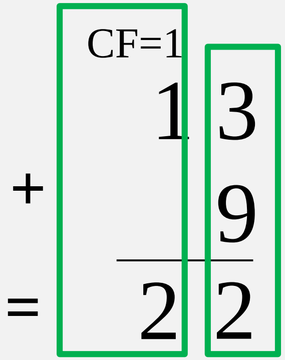
adc AX, BX ;(AX = AX+BX+CF)

SB… rrow

Exa…

sbb … X-CF)

ADC illustration example:



<instruction> r/m8(16,32) 1/CL/imm8

SHL, SHR – Bitwise Logical Shifts on first operand

- number of bits to shift is given by second operand
- vacated bits are filled with zero
- (last) shifted bit enters to CF flag

shift indeed performs **fast** division / multiplication by 2

Example:

mov al ,10110111_b ; AL = 10110111_b

shr al, 1 ; shift right 1 bit → AL = 01011011_b, CF = 1

shl al, 4 ; shift left 4 bits → AL = 10110000_b, CF = 1

SAL, SAR – Bitwise Arithmetic Shift on first operand

- vacated bits are filled with zero for SAL
- vacated bits are filled with copies of the **original high bit** of the source operand for **SAR**
- (last) shifted bit enters to CF flag

Example:

mov al ,10110111_b ; AL = 10110111_b

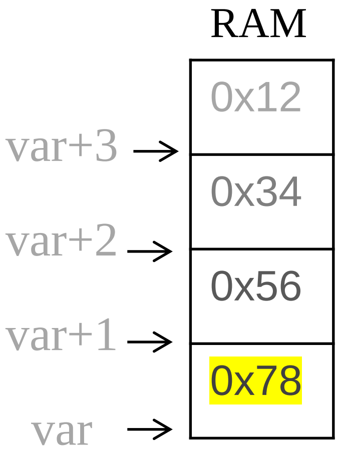
sar al, 3 ; shift arithmetical right 3 bits → AL = 11110110_b, CF = 1

sal al, 2 ; shift (arithmetical) left 2 bits → AL = 11011000_b, CF = 1

Little Endian - store least significant byte in lowest address

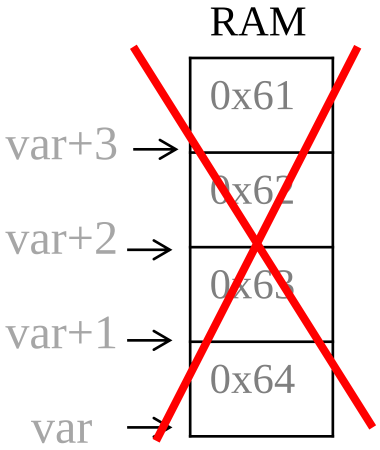
var: dd 0x12345678

	3	2	1	0
EAX	0x12	0x34	0x56	0x78



var: dd 'abcd'

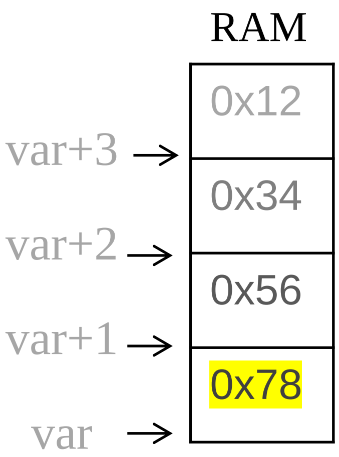
'abcd' = 0x61626364



Little Endian - store least significant byte in lowest address

var: dd 0x12345678

	3	2	1	0
EAX	0x12	0x34	0x56	0x78


RAM

var+3 →

var+2 →

var+1 →

var →

0x12

0x34

0x56

0x78

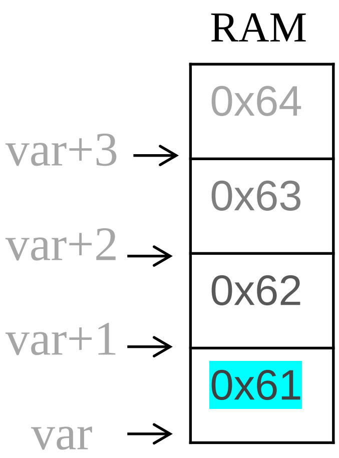
String is translated by NASM into numeric value in reversed order.
Thus, when (numeric value of)string inserted into memory, we get it in source order.

var: dd 'abcd'

⇩

var: dd 0x64636261

	3	2	1	0
EAX	0x64	0x63	0x62	0x61


RAM

var+3 →

var+2 →

var+1 →

var →

0x64

0x63

0x62

0x61

Sections

Every process consists of sections that are accessible to the process during its execution.

Data

.bss - holds uninitialized data

.data - holds initialized data

.rodata - holds read-only data

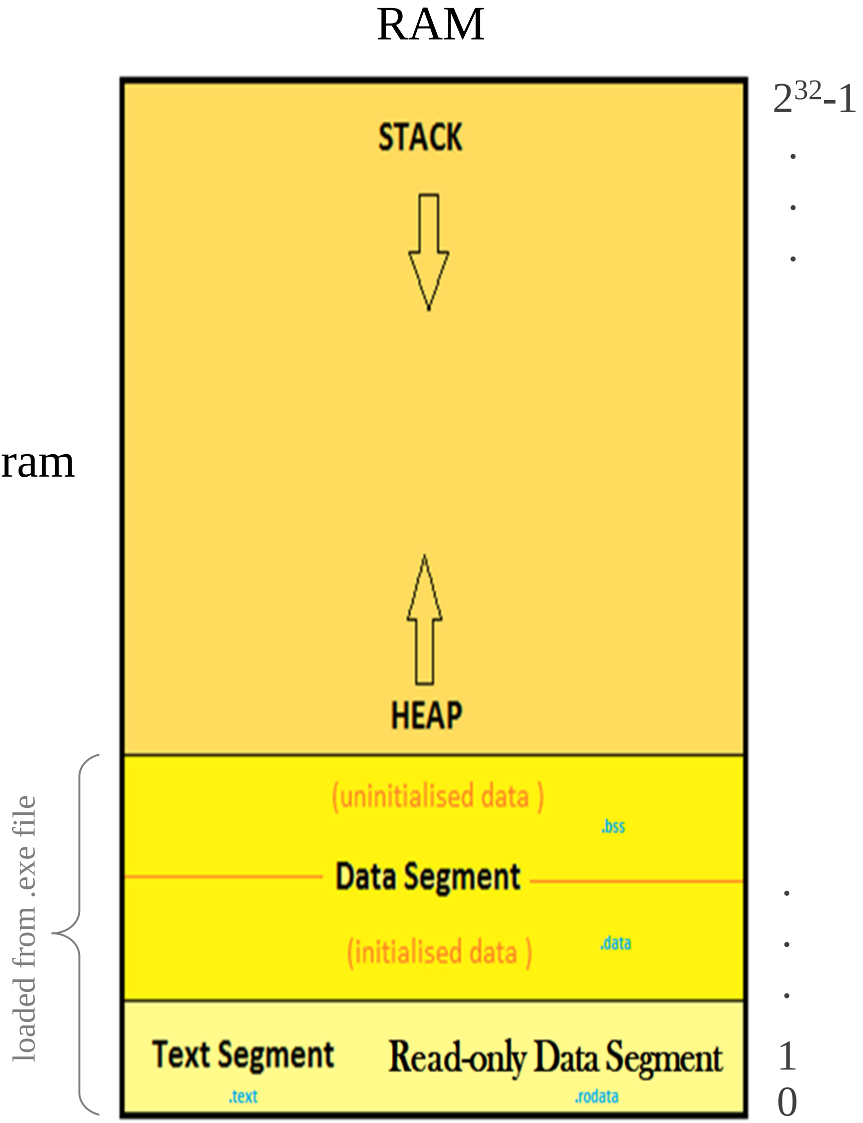
Text

.text – holds executable instructions of a program

Stack* – holds functions activation frame

Heap – holds dynamically allocated memory

* Stack pointer is normally initialized to point to the top of stack and proceed downward. In fact, we can point to any location we like and manage any area of memory as stack.



Sections

Every process consists of sections that are accessible to the process during its execution.

Data

.bss - holds uninitialized data

.data - holds initialized data

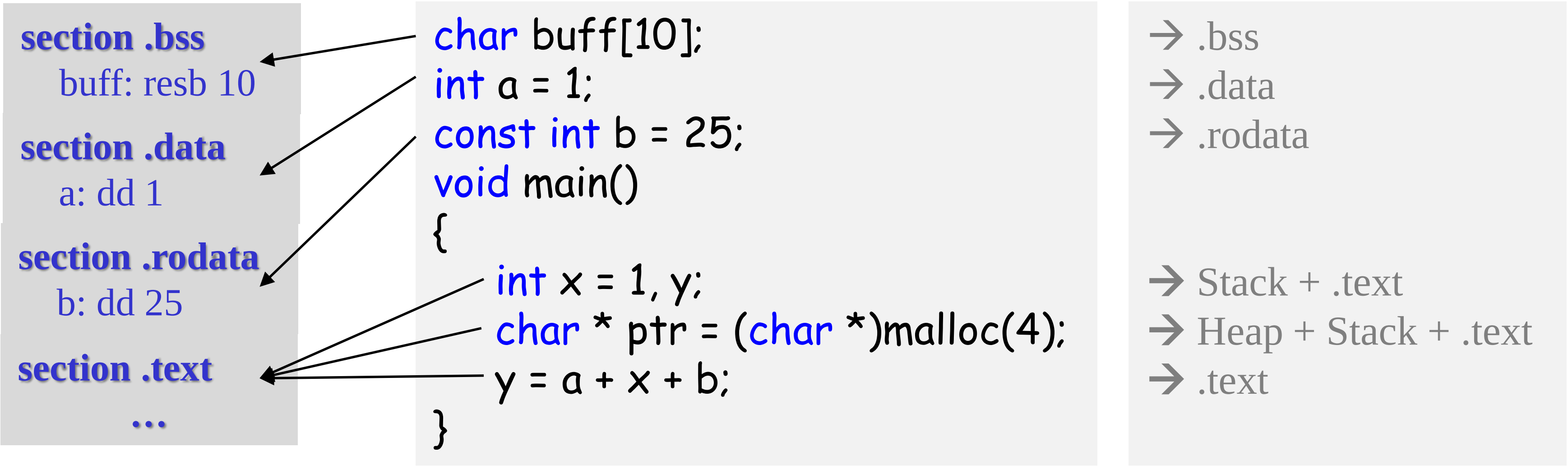
.rodata - holds read-only data

Text

.text – holds executable instructions of a program

Stack* – holds functions activation frame

Heap – holds dynamically allocated memory



```
section .bss
    buff: resb 10
section .data
    a: dd 1
section .rodata
    b: dd 25
section .text
    …
```

```
char buff[10];
int a = 1;
const int b = 25;
void main()
{
    int x = 1, y;
    char * ptr = (char *)malloc(4);
    y = a + x + b;
}
```

→ .bss
→ .data
→ .rodata

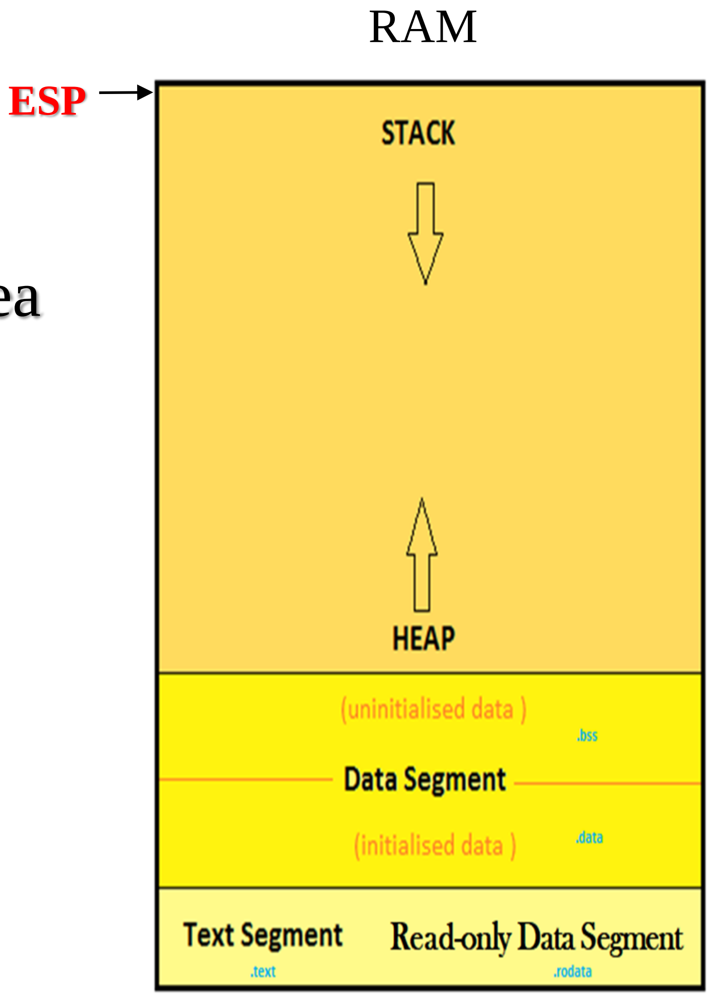
→ Stack + .text
→ Heap + Stack + .text
→ .text

Stack

- Stack is temporary storage memory area

- **ESP** points to the top of Stack
  (by default, it is highest RAM address)

- **Stack addresses go from high to low**



Stack Operations

PUSH - push data on Stack

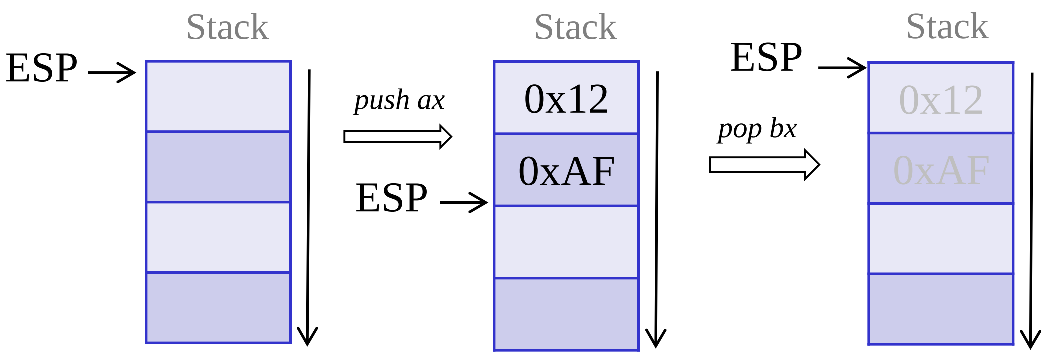
- decrements ESP by 2/4 bytes (according to the operand size)
- stores the operand value at ESP address on Stack (in Little Endian manner)

POP - load a value from Stack

- reads the operand value at ESP address on Stack (in Little Endian manner)
- increment ESP by 2/4 bytes (according to the operand size)

```
mov ax, 0x12AF
push ax
pop bx
```

In 32-bit mode, you can push 32 or (with operand-size prefix) 16 bits.



https://stackoverflow.com/questions/45127993/how-many-bytes-does-the-push-instruction-push-onto-the-stack-when-i-dont-specif

Stack Operations

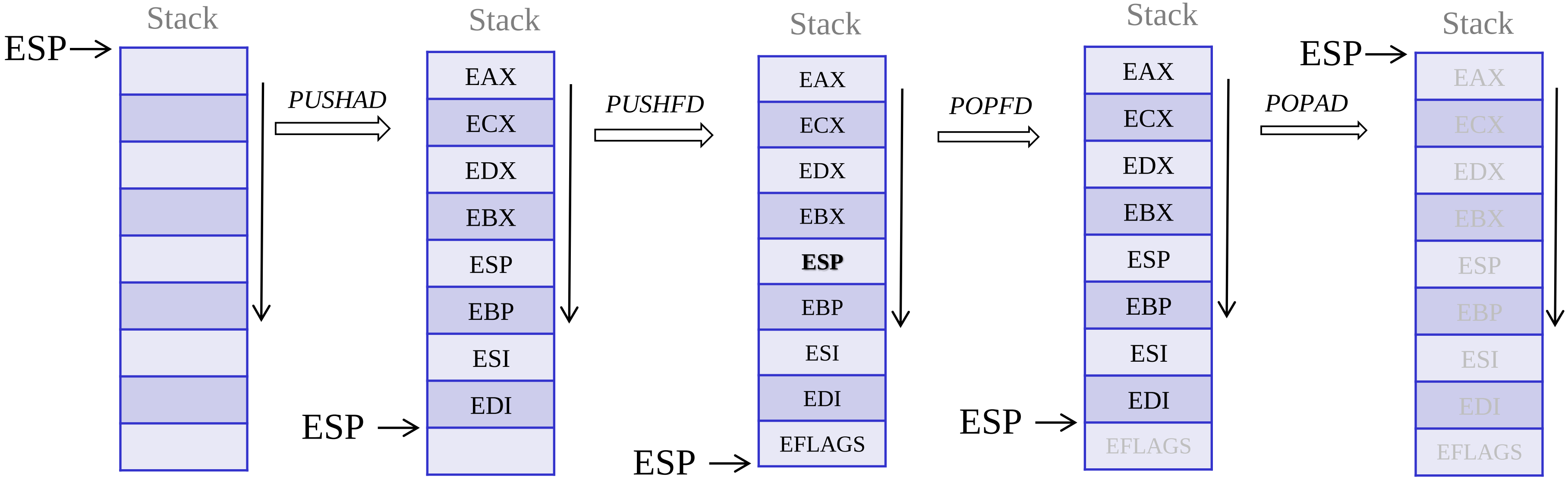
PUSHAD (push all double) - pushes values of EAX, ECX, EDX, EBX, **ESP**, EBP, ESI, and EDI onto Stack

original ESP value before PUSHAD

PUSHFD (push flags double) - push value of EFLAGS onto Stack

POPAD (pop all double) - pop a dword from Stack into each one of (successively) EDI, ESI, EBP, nothing (placeholder for ESP), EBX, EDX, ECX, and EAX

POPFD (pop flags double) - pop a dword from Stack into EFLAGS



C Calling Convention

```
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

C Calling Convention

```
section .bss
  result: resd 1
section .text
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

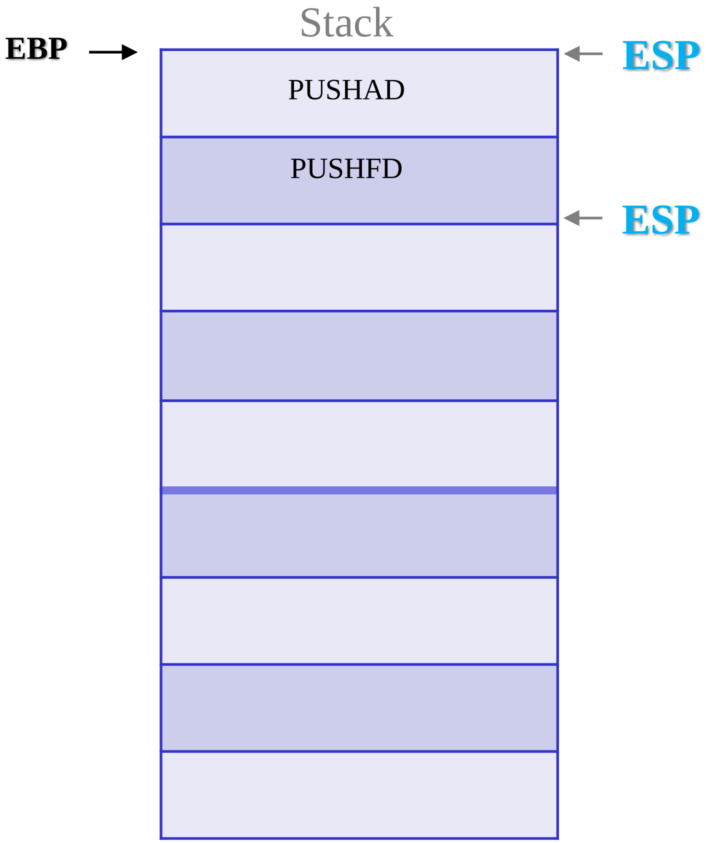
C Calling Convention

```
section .bss
  result: resd 1
section .text
main:          ; caller code
pushad                     ; backup registers
pushfd                     ; backup EFLAGS
```

```
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

According to C calling convention, the called function is allowed to mess up values of EAX, ECX and EDX registers. EBX, ESI, and EDI registers' values should be preserved and restored by the called function.

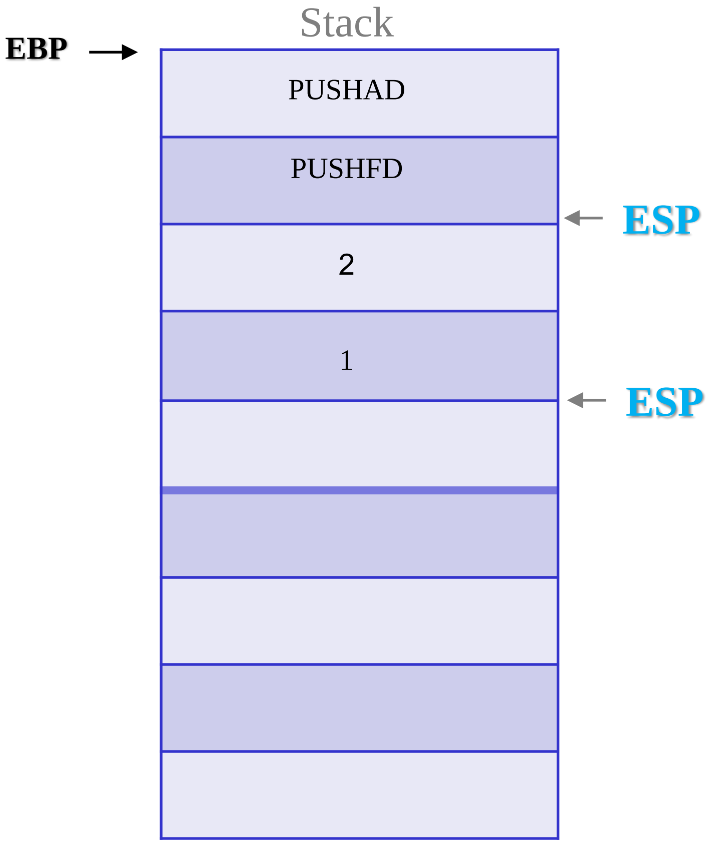
But in fact it is not 100% sure that all C code obeys this (may be compiler implementation dependent), so it is a good idea to backup and restore all registers.



C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                      ; backup registers
pushfd                      ; backup EFLAGS
push dword 2                ; push second argument
push dword 1                ; push first argument
```

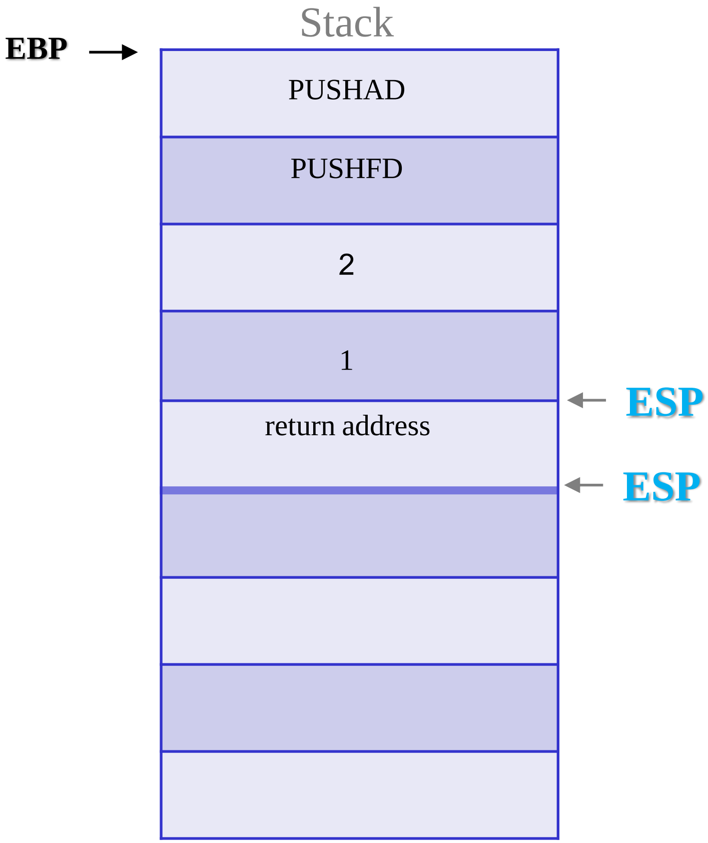
```
int Func(int x, int y) {
    int sum;
    sum = x + y;
    return sum;
}
int result;
void main() {
    result = Func(1, 2);
}
```



C Calling Convention

```
section .bss
  result: resd 1
section .text
main:        ; caller code
pushad                    ; backup registers
pushfd                    ; backup EFLAGS
push dword 2              ; push second argument
push dword 1              ; push first argument
call Func                 ; call the function → push return address
                          ; into Stack and jump to function code
```

```
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

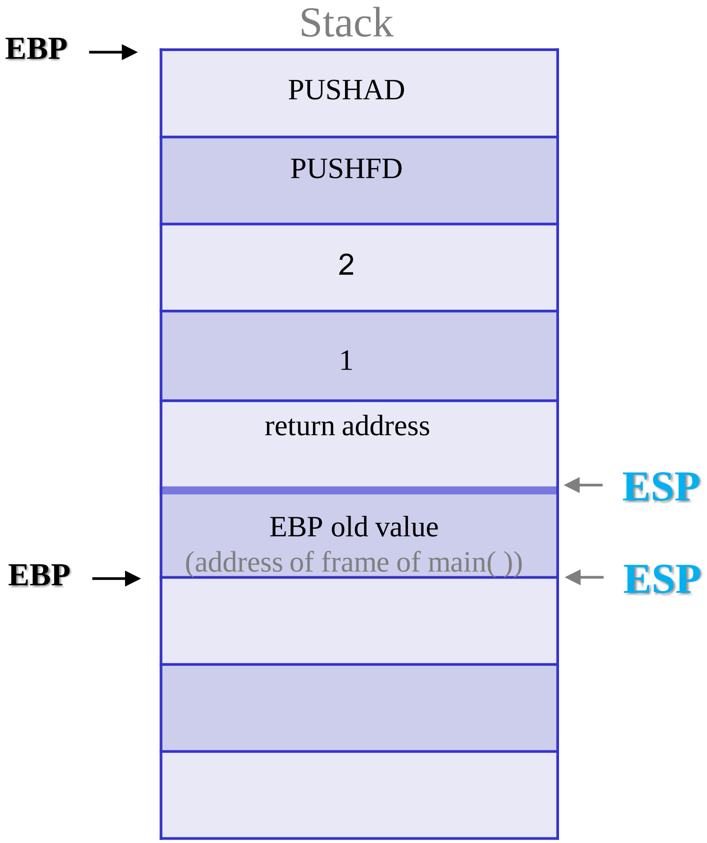


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                      ; backup registers
pushfd                      ; backup EFLAGS
push dword 2                ; push second argument
push dword 1                ; push first argument
call Func                   ; call the function → push return address
                            ; into Stack and jump to function code

Func:          ; callee code
   push ebp                    ; backup EBP
   mov ebp, esp                ; reset EBP to the current ESP
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

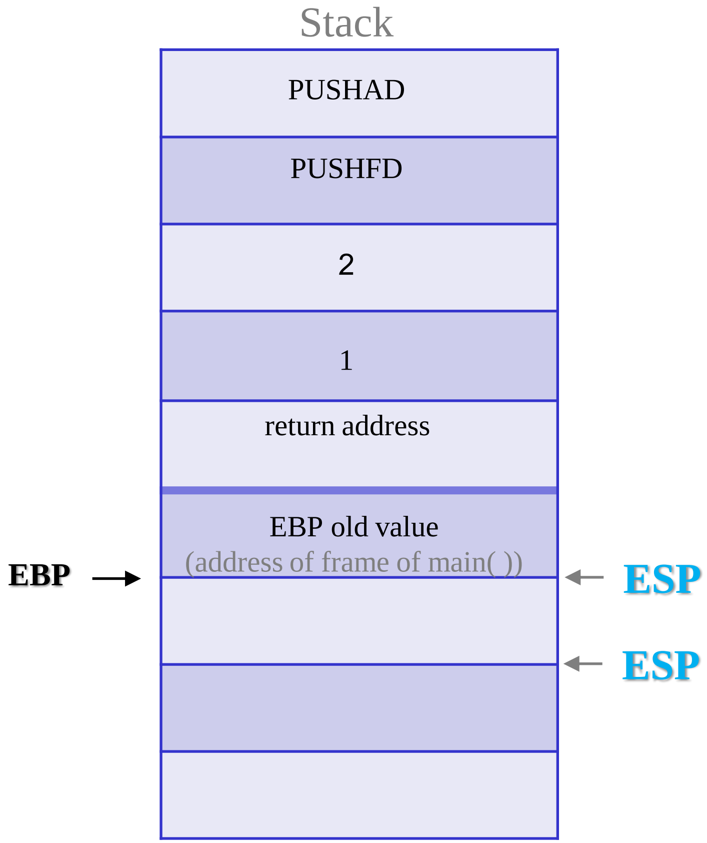


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                  ; backup registers
pushfd                  ; backup EFLAGS
push dword 2            ; push second argument
push dword 1            ; push first argument
call Func               ; call the function → push return address
                        ; into Stack and jump to function code
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

```
Func:          ; callee code
   push ebp                  ; backup EBP
   mov ebp, esp              ; reset EBP to the current ESP
   sub esp, 4                ; allocate space for local variable sum
```

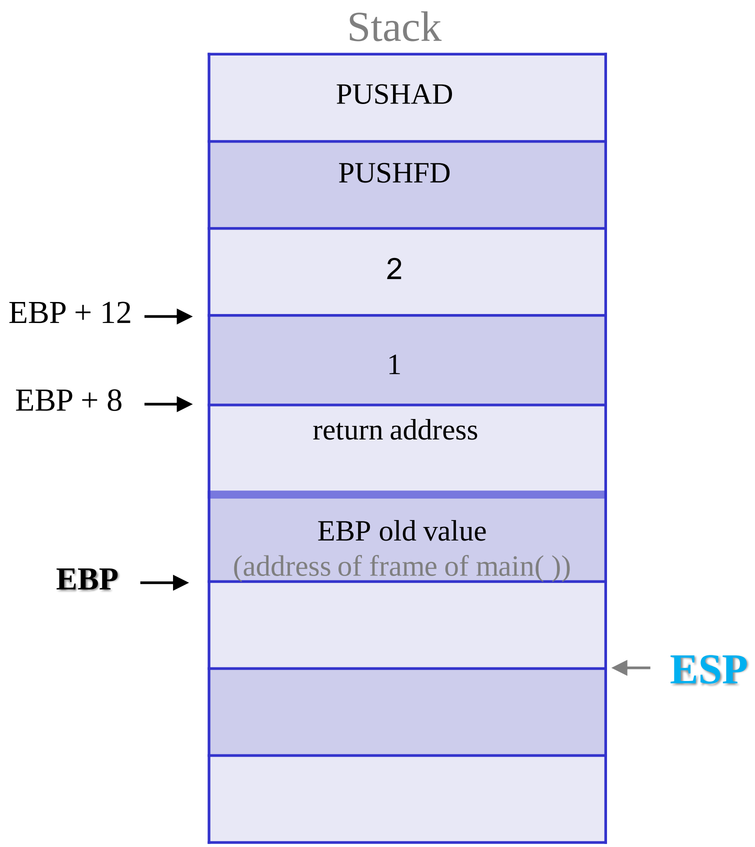


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                        ; backup registers
pushfd                        ; backup EFLAGS
push dword 2                  ; push second argument
push dword 1                  ; push first argument
call Func                     ; call the function → push return address
                              ; into Stack and jump to function code
```

```
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

```
Func:          ; callee code
   push ebp                      ; backup EBP
   mov ebp, esp                  ; set EBP to Func activation frame
   sub esp, 4                    ; allocate space for local variable sum
   mov ebx, [ebp+8]              ; get first argument
   mov ecx, [ebp+12]             ; get second argument
```

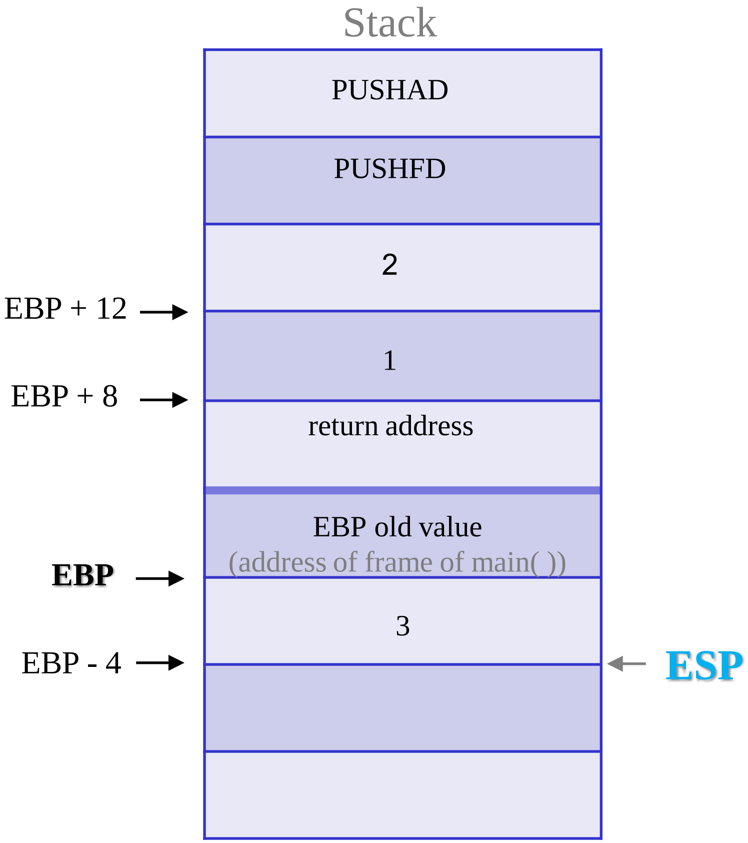


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                   ; backup registers
pushfd                   ; backup EFLAGS
push dword 2             ; push second argument
push dword 1             ; push first argument
call Func                ; call the function → push return address
                         ; into Stack and jump to function code
```

```
int Func(int x, int y) {
    int sum;
    sum = x + y;
    return sum;
}
int result;
void main() {
    result = Func(1, 2);
}
```

```
Func:          ; callee code
   push ebp                ; backup EBP
   mov ebp, esp            ; set EBP to Func activation frame
   sub esp, 4              ; allocate space for local variable sum
   mov ebx, [ebp+8]        ; get first argument
   mov ecx, [ebp+12]       ; get second argument
   add ebx, ecx            ; calculate x+y
   mov [ebp-4], ebx        ; set sum to be x+y
```

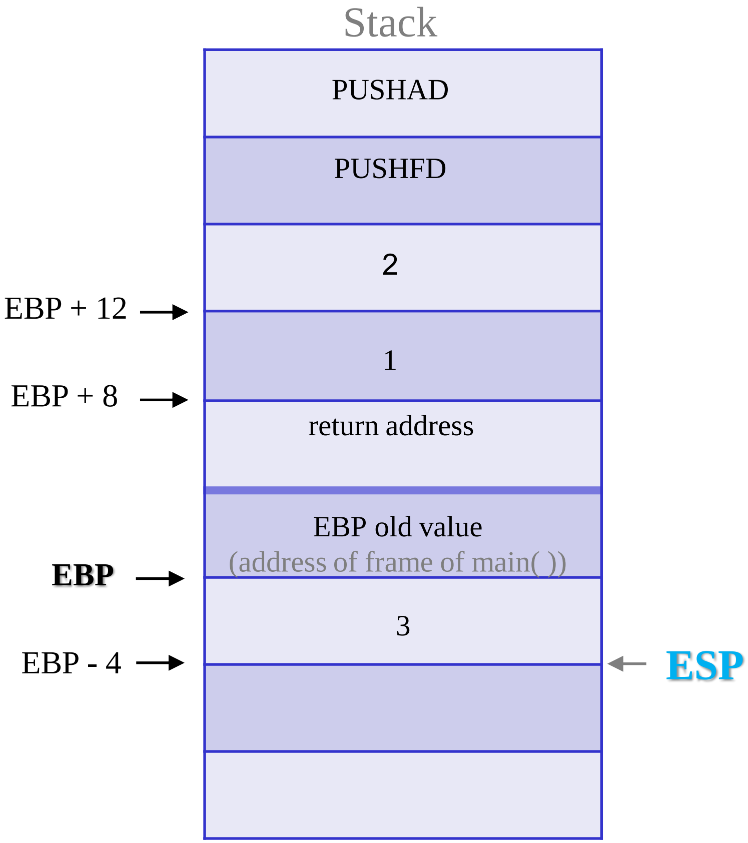


C Calling Convention

```
section .bss
  result: resd 1
section .text
main:          ; caller code
pushad                   ; backup registers
pushfd                   ; backup EFLAGS
push dword 2             ; push second argument
push dword 1             ; push first argument
call Func                ; call the function → push return address
                         ; into Stack and jump to function code
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

```
Func:          ; callee code
   push ebp                  ; backup EBP
   mov ebp, esp              ; set EBP to Func activation frame
   sub esp, 4                ; allocate space for local variable sum
   mov ebx, [ebp+8]          ; get first argument
   mov ecx, [ebp+12]         ; get second argument
   add ebx, ecx              ; calculate x+y
   mov [ebp-4], ebx          ; set sum to be x+y
   mov eax, [ebp-4]          ; put return value into EAX
```

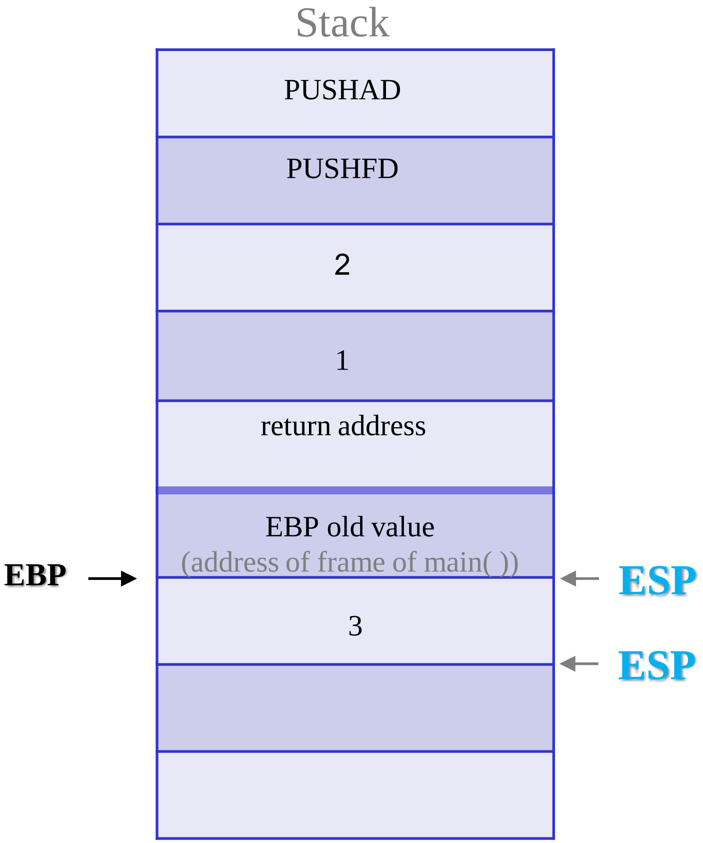


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                  ; backup registers
pushfd                  ; backup EFLAGS
push dword 2            ; push second argument
push dword 1            ; push first argument
call Func               ; call the function → push return address
                        ; into Stack and jump to function code
```

```
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

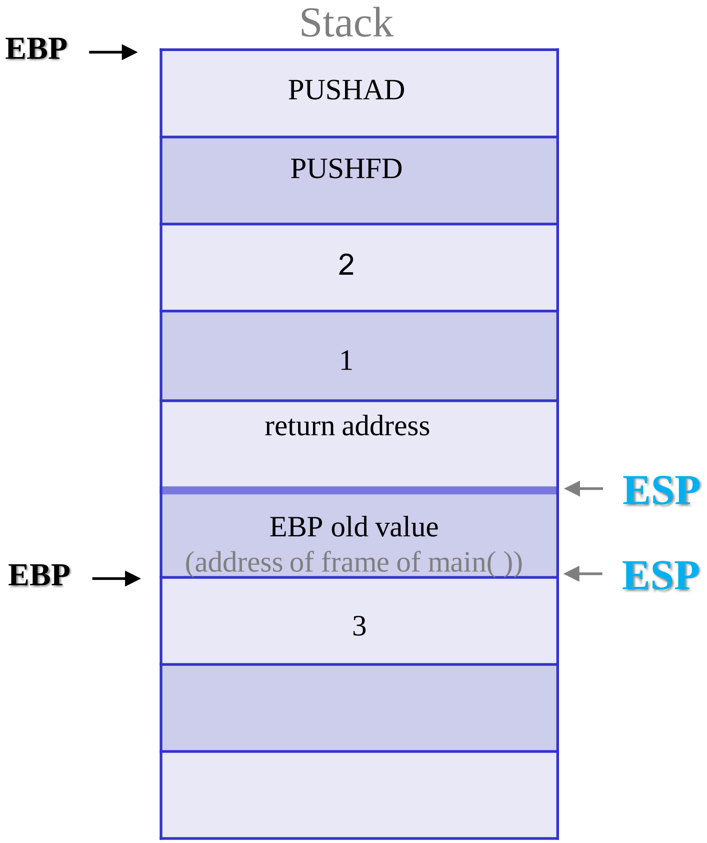
```
Func:          ; callee code
   push ebp                ; backup EBP
   mov ebp, esp            ; set EBP to Func activation frame
   sub esp, 4              ; allocate space for local variable sum
   mov ebx, [ebp+8]        ; get first argument
   mov ecx, [ebp+12]       ; get second argument
   add ebx, ecx            ; calculate x+y
   mov [ebp-4], ebx        ; set sum to be x+y
   mov eax, [ebp-4]        ; put return value into EAX
   mov esp, ebp             ; "free" function activation frame
```



C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                        ; backup registers
pushfd                        ; backup EFLAGS
push dword 2                  ; push second argument
push dword 1                  ; push first argument
call Func                     ; call the function → push return address
                              ; into Stack and jump to function code
```

```
int Func(int x, int y) {
    int sum;
    sum = x + y;
    return sum;
}
int result;
void main() {
    result = Func(1, 2);
}
```

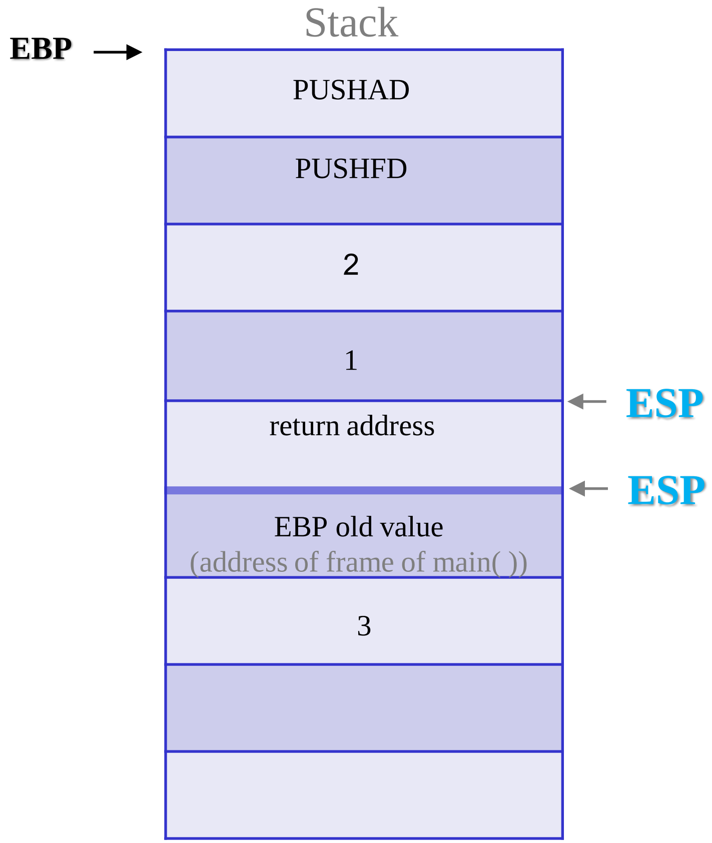


```
Func:          ; callee code
   push ebp                      ; backup EBP
   mov ebp, esp                  ; set EBP to Func activation frame
   sub esp, 4                    ; allocate space for local variable sum
   mov ebx, [ebp+8]              ; get first argument
   mov ecx, [ebp+12]             ; get second argument
   add ebx, ecx                  ; calculate x+y
   mov [ebp-4], ebx              ; set sum to be x+y
   mov eax, [ebp-4]              ; put return value into EAX
   mov esp, ebp                  ; "free" function activation frame
   pop ebp                       ; restore activation frame of main()
```

C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                     ; backup registers
pushfd                     ; backup EFLAGS
push dword 2               ; push second argument
push dword 1               ; push first argument
call Func                  ; call the function → push return address
                           ; into Stack and jump to function code
```

```
int Func(int x, int y) {
    int sum;
    sum = x + y;
    return sum;
}
int result;
void main() {
    result = Func(1, 2);
}
```



```
Func:          ; callee code
   push ebp                    ; backup EBP
   mov ebp, esp                ; set EBP to Func activation frame
   sub esp, 4                  ; allocate space for local variable sum
   mov ebx, [ebp+8]            ; get first argument
   mov ecx, [ebp+12]           ; get second argument
   add ebx, ecx                ; calculate x+y
   mov [ebp-4], ebx            ; set sum to be x+y
   mov eax, [ebp-4]            ; put return value into EAX
   mov esp, ebp                ; "free" function activation frame
   pop ebp                     ; restore activation frame of main()
   RET                         ; return from the function
```

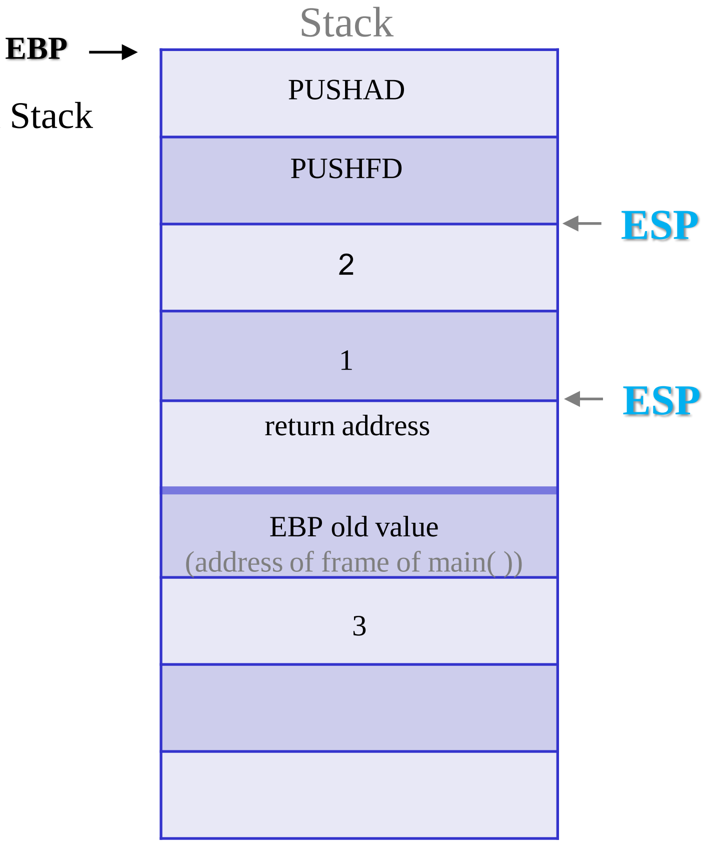
C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                    ; backup registers
pushfd                    ; backup EFLAGS
push dword 2              ; push second argument
push dword 1              ; push first argument
call Func                 ; call the function → push return address
                          ; into Stack and jump to function code

mov [result], eax         ; retrieve return value from EAX
add esp, 8                ; “free" space allocated for function arguments in Stack


Func:          ; callee code
   push ebp                 ; backup EBP
   mov ebp, esp             ; set EBP to Func activation frame
   sub esp, 4               ; allocate space for local variable sum
   mov ebx, [ebp+8]         ; get first argument
   mov ecx, [ebp+12]        ; get second argument
   add ebx, ecx             ; calculate x+y
   mov [ebp-4], ebx         ; set sum to be x+y
   mov eax, [ebp-4]         ; put return value into EAX
   mov esp, ebp             ; “free” function activation frame
   pop ebp                  ; restore activation frame of main()
   RET                      ; return from the function
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```



Stack

EBP →

PUSHAD

PUSHFD

← ESP

2

1

← ESP

return address

EBP old value
(address of frame of main( ))

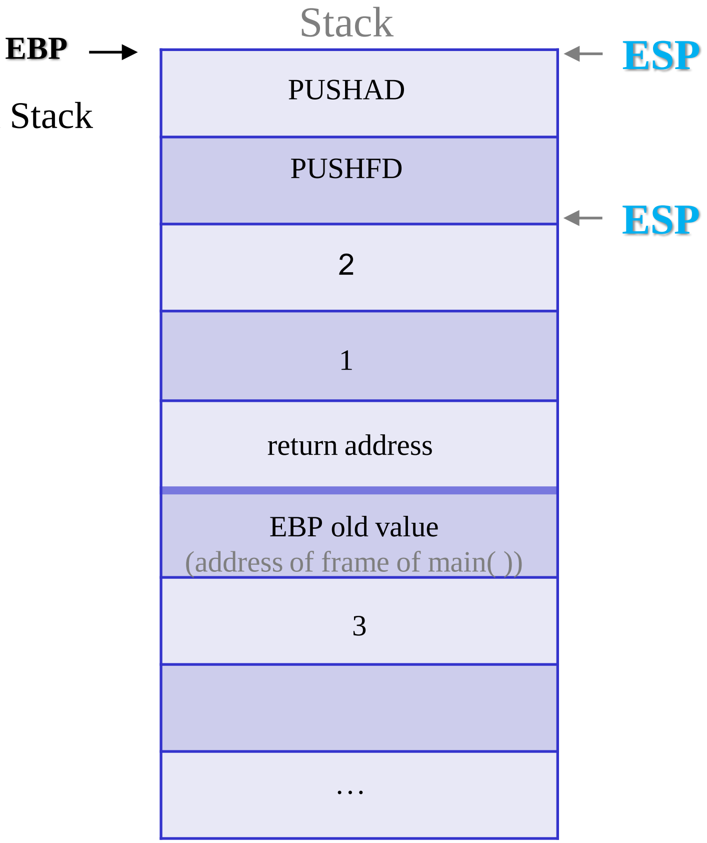
3

C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                      ; backup registers
pushfd                      ; backup EFLAGS
push dword 2                ; push second argument
push dword 1                ; push first argument
call Func                   ; call the function → push return address
                            ; into Stack and jump to function code

mov [result], eax           ; retrieve return value from EAX
add esp, 8                  ; "free" space allocated for function arguments in Stack
popfd                       ; restore flags register
popad                       ; restore registers
…
Func:          ; callee code
   push ebp                   ; backup EBP
   mov ebp, esp               ; set EBP to Func activation frame
   sub esp, 4                 ; allocate space for local variable sum
   mov ebx, [ebp+8]           ; get first argument
   mov ecx, [ebp+12]          ; get second argument
   add ebx, ecx               ; calculate x+y
   mov [ebp-4], ebx           ; set sum to be x+y
   mov eax, [ebp-4]           ; put return value into EAX
   mov esp, ebp               ; "free" function activation frame
   pop ebp                    ; restore activation frame of main()
   RET                        ; return from the function
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```

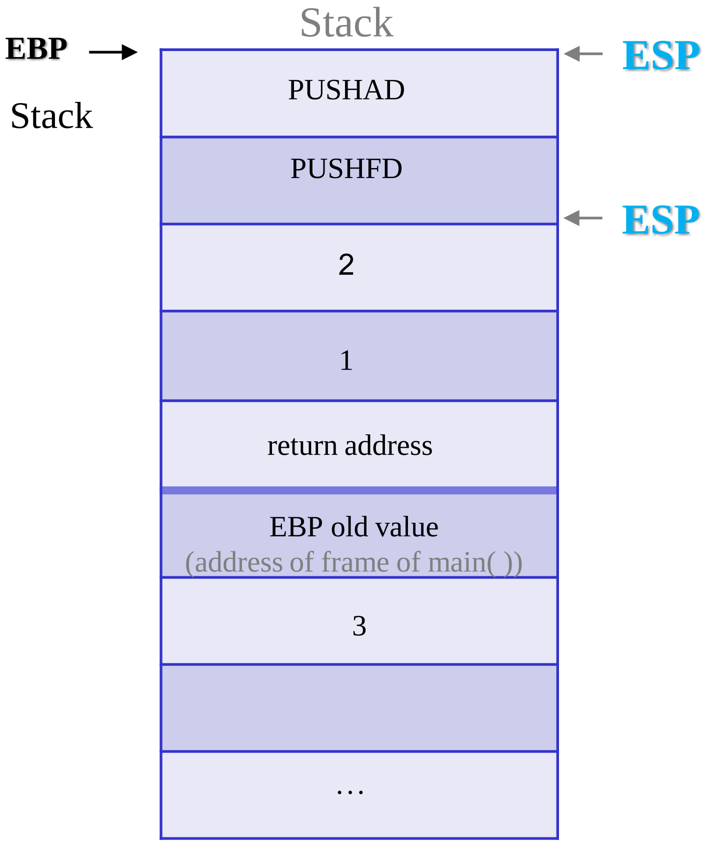


C Calling Convention

```
section .bss
   result: resd 1
section .text
main:          ; caller code
pushad                      ; backup registers
pushfd                      ; backup EFLAGS
push dword 2                ; push second argument
push dword 1                ; push first argument
call Func                   ; call the function → push return address
                            ; into Stack and jump to function code

mov [result], eax           ; retrieve return value from EAX
add esp, 8                  ; “free" space allocated for function arguments in Stack
popfd                       ; restore flags register
popad                       ; restore registers
…
Func:          ; callee code
   push ebp                 ; backup EBP
   mov ebp, esp             ; set EBP to Func activation frame
   sub esp, 4               ; allocate space for local variable sum
   mov ebx, [ebp+8]         ; get first argument
   mov ecx, [ebp+12]        ; get second argument
   add ebx, ecx             ; calculate x+y
   mov [ebp-4], ebx         ; set sum to be x+y
   mov eax, [ebp-4]         ; put return value into EAX
   mov esp, ebp             ; “free” function activation frame
   pop ebp                  ; restore activation frame of main()
   RET                      ; return from the function
```

```c
int Func(int x, int y) {
   int sum;
   sum = x + y;
   return sum;
}
int result;
void main() {
   result = Func(1, 2);
}
```



gdb debugger – very basic usage

- ❑ run Gdb from the console by typing:

  gdb executableFileName

- ❑ add breaking points by typing:

  break label

- ❑ start debugging by typing:

  run parameters (argv)

(gdb) set disassembly-flavor intel — change presentation of assembly-language instructions from the default Motorala conventions, that are used by gcc, to the Intel conventions that are used by nasm, that is, from opcode source, dest to opcode dest, src

(gdb) layout asm — this will display the assembly language

(gdb) layout regs – this will display registers

- **s/si** – one step forward
- **c** – continue to run the code until the next break point.
- **q** – quit gdb
- **p/x $eax** – prints the value in eax
- **x $esp**– prints esp value (address) and value (dword) that is stored in this address. It is possible to use label instead of esp.

  Type **x** again will print the next dword in memory.