ICS 51 Discussion:
Function Calls in Assembly

Aniket Shivam

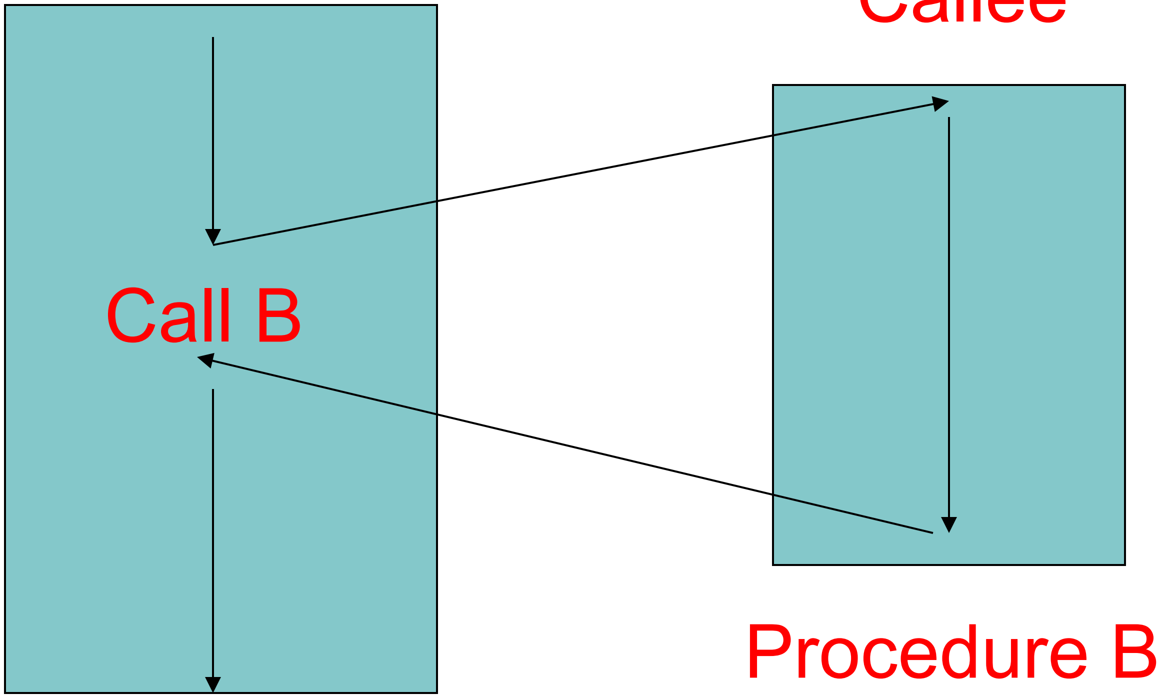
02/21/18

Some slides courtesy of Jelena Trajkovic

Outline

- How to call a procedure/function?
  - Call/Return
  - Parameter passing
  - Return value
  - Save/Restore registers
  - Local variable allocation
- Call Parameters
- Two’s Complement

Procedure call



Caller

Callee

Call B

Procedure B

Procedure A

Procedure call

- Several issues need to be addressed:
  - How to call and return from a procedure?
  - How to pass parameters to callee?
  - How to pass the return value?
  - How to save/restore registers to avoid register usage conflict in caller and callee?
  - Where to allocate local variables?

- Solution: Call stack

What is a stack?

- Stack is a Last In, First Out (LIFO) data structure.
- Items added/pushed to the stack goes on “top”.
- Items removed/popped from the stack are fetched from the “top”.

http://www.cs.armstrong.edu/liang/animation/web/Stack.html

Call Stack

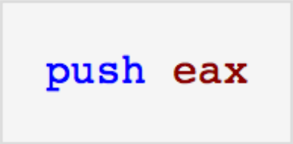
- Call Stack stores information about the active subroutines of a program.
- Although maintenance of the call stack is important for the proper functioning of most software, the details are normally hidden in high-level programming languages.
  - Usually, compiler maintains it.

Call Stack Implementation

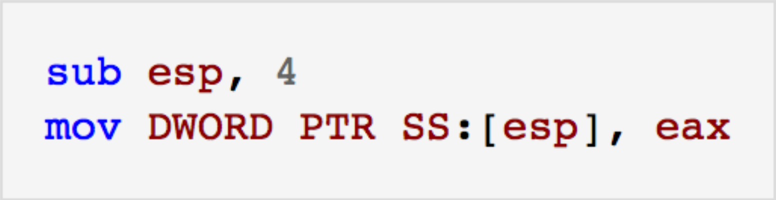
- Stack is implemented as an array in a region of the memory.
- Two important registers:
  - ***ESP*** points to top of the stack
    - Managed by the hardware
  - ***EBP*** is a ***user*** register to keep track of data on the stack

Call Stack Implementation

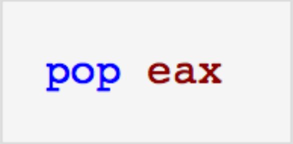
- Push and Pop Instructions



```
push eax
```



```
sub esp, 4
mov DWORD PTR SS:[esp], eax
```



```
pop eax
```

```
mov eax, DWORD PTR SS:[esp]
add esp, 4
```

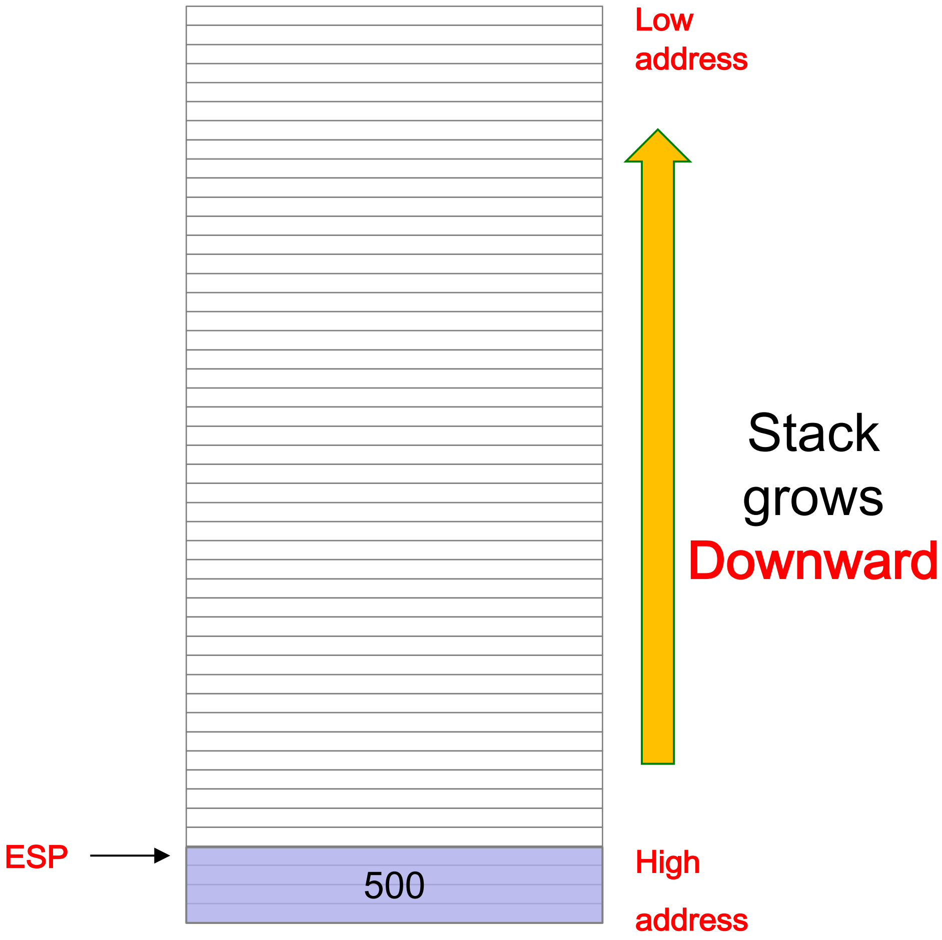
Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

EAX ?

EBX ?

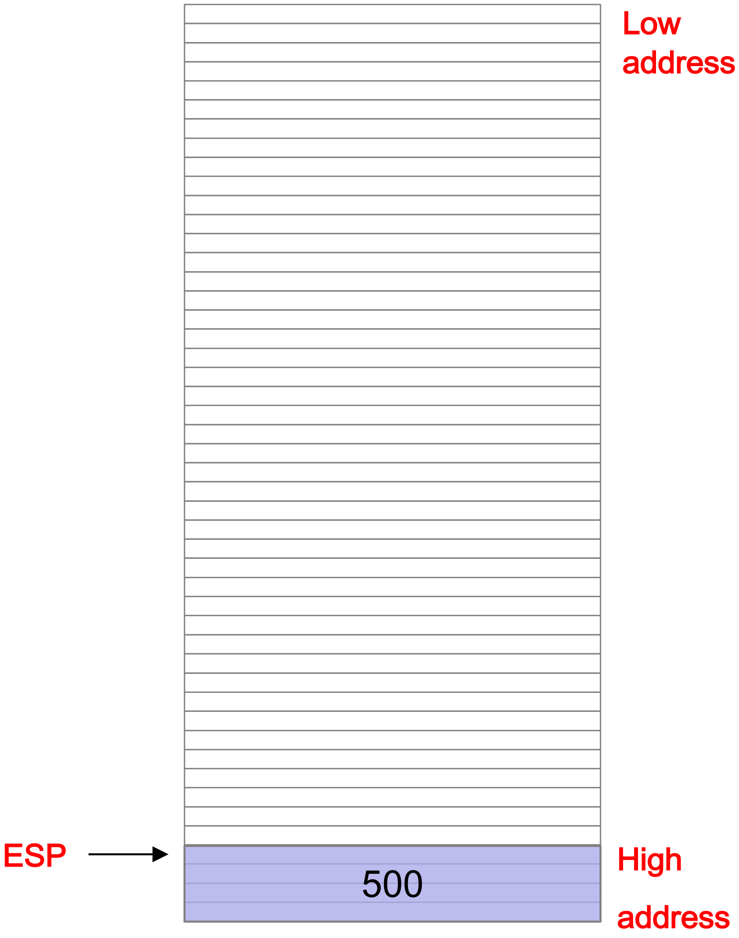
ESP 0x0040ff24



Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

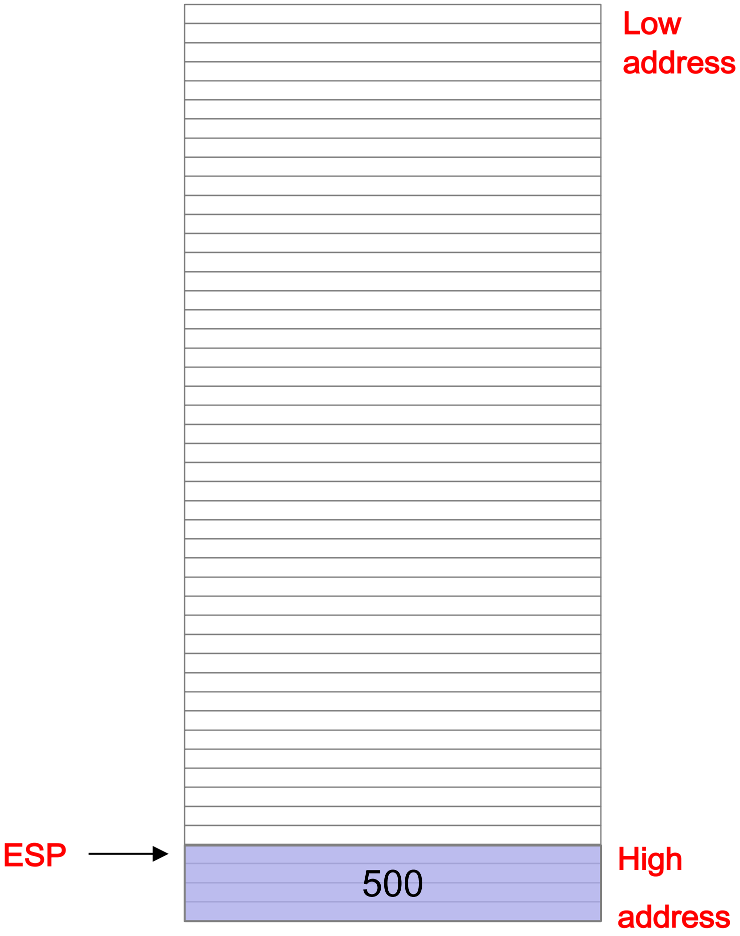
EAX	1000
EBX	?
ESP	0x0040ff24



Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

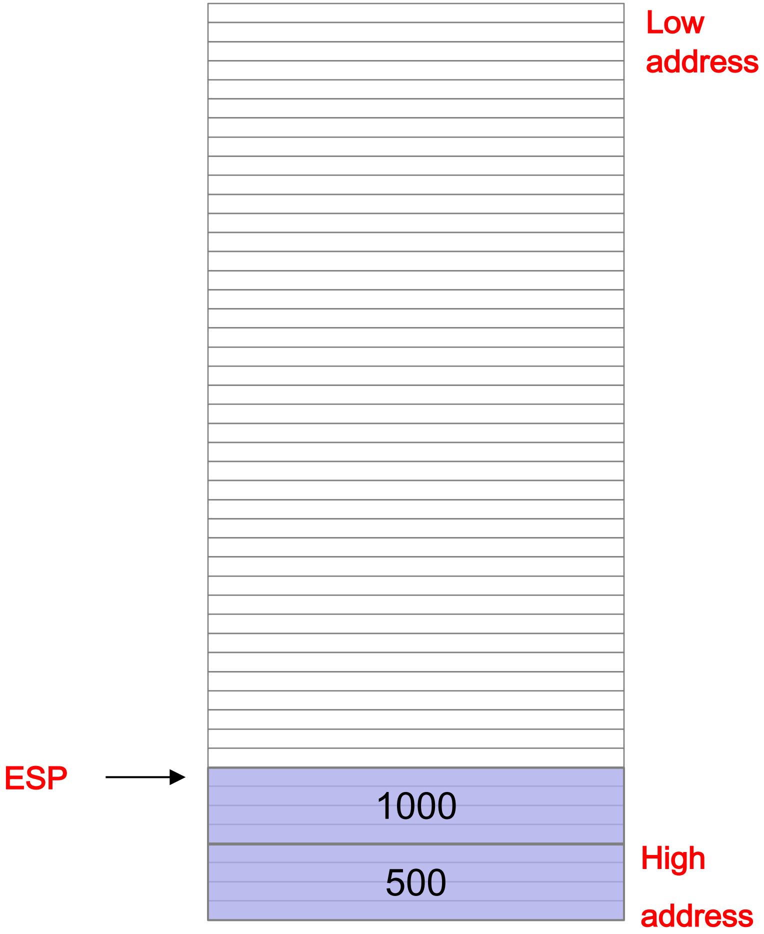
EAX	1000
EBX	2000
ESP	0x0040ff24



Stack: Example

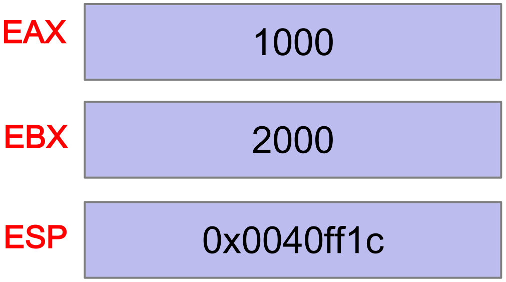
```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

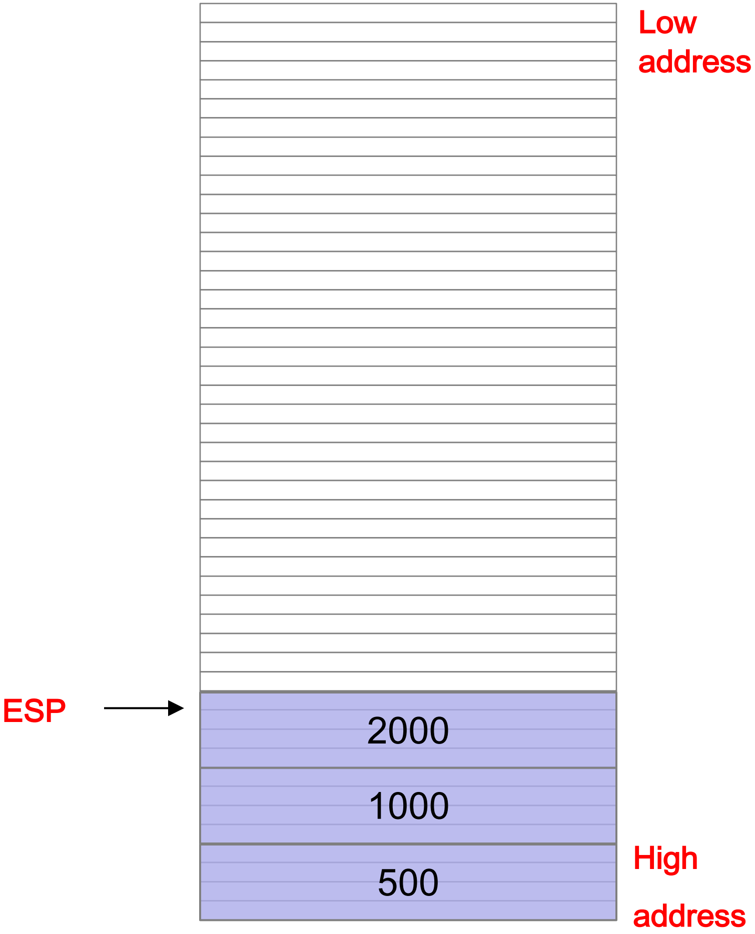
EAX	1000
EBX	2000
ESP	0x0040ff20



Stack: Example

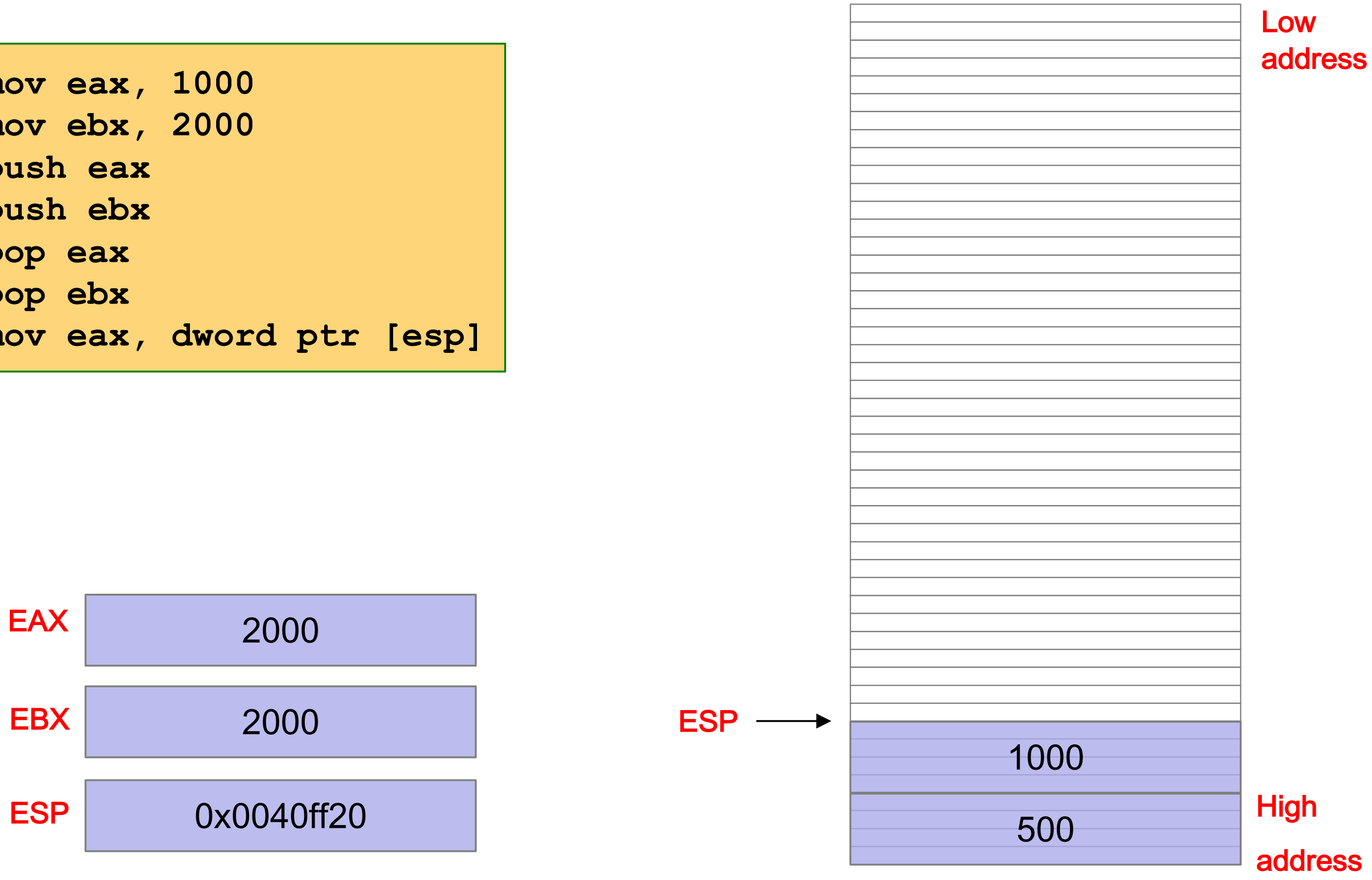
```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```





Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```



EAX 2000

EBX 2000

ESP 0x0040ff20

Low address

ESP

1000

500

High address

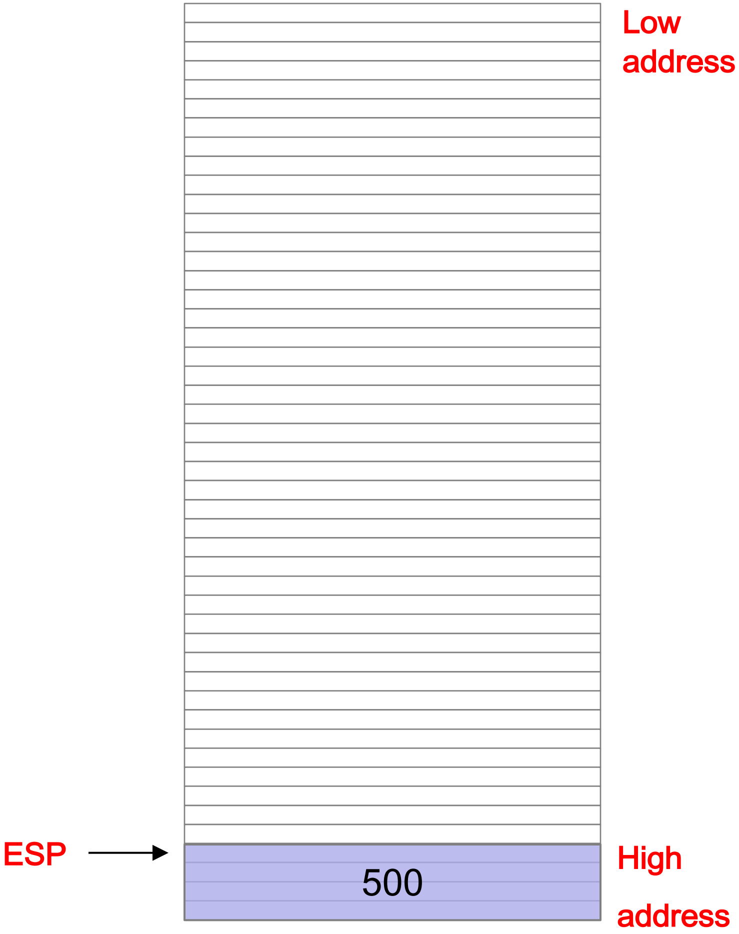
Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

EAX 2000

EBX 1000

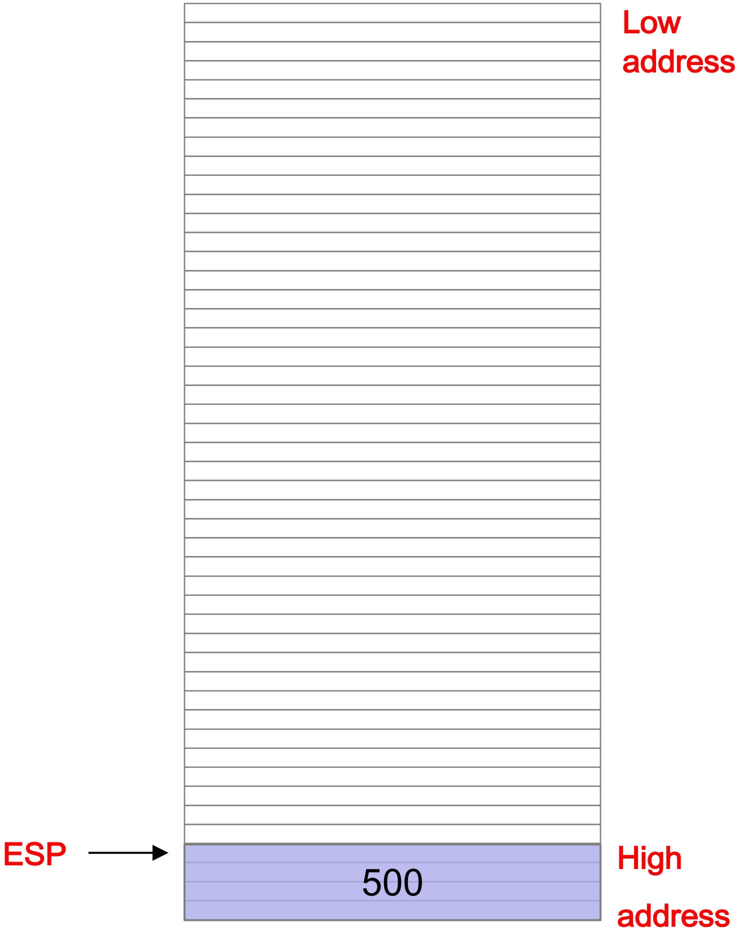
ESP 0x0040ff24



Stack: Example

```
mov eax, 1000
mov ebx, 2000
push eax
push ebx
pop eax
pop ebx
mov eax, dword ptr [esp]
```

EAX	500
EBX	1000
ESP	0x0040ff24



Calling Convention

- Several issues need to be addressed:
  - How to call and return from a procedure?
  - How to pass parameters to callee?
  - How to pass the return value?
  - How to save/restore registers to avoid register usage conflict in caller and callee?
  - Where to allocate local variables?

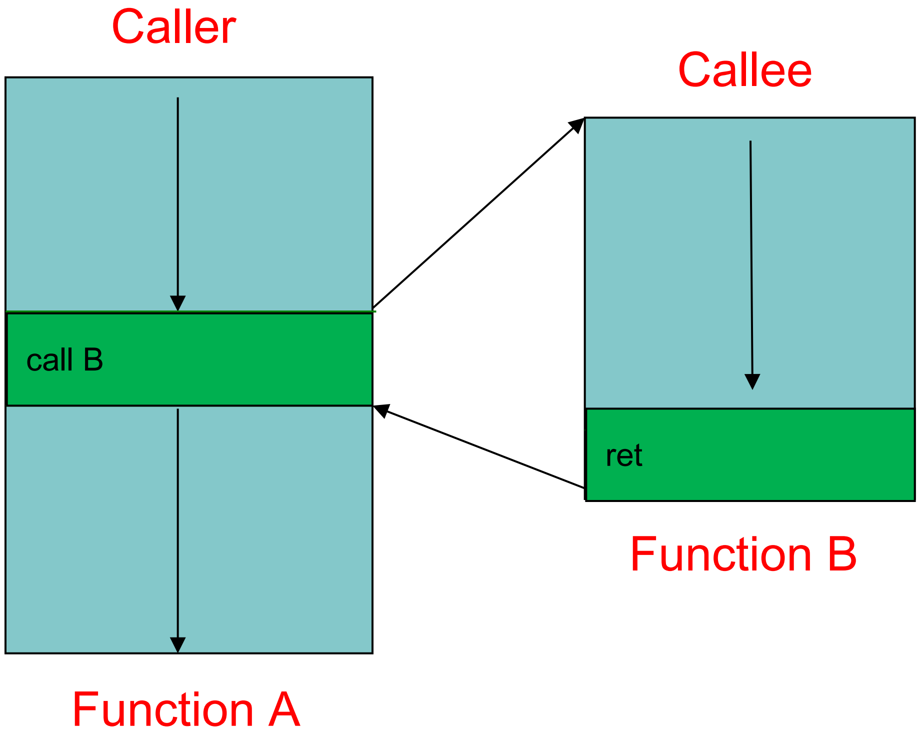
- Calling conventions describe how programmers should implement these steps for a given compiler/OS.

Calling Convention

Högertrafikomläggningen, the day where traffic in Sweden switched from the left to the right side of the road 1967

Call and return from functions

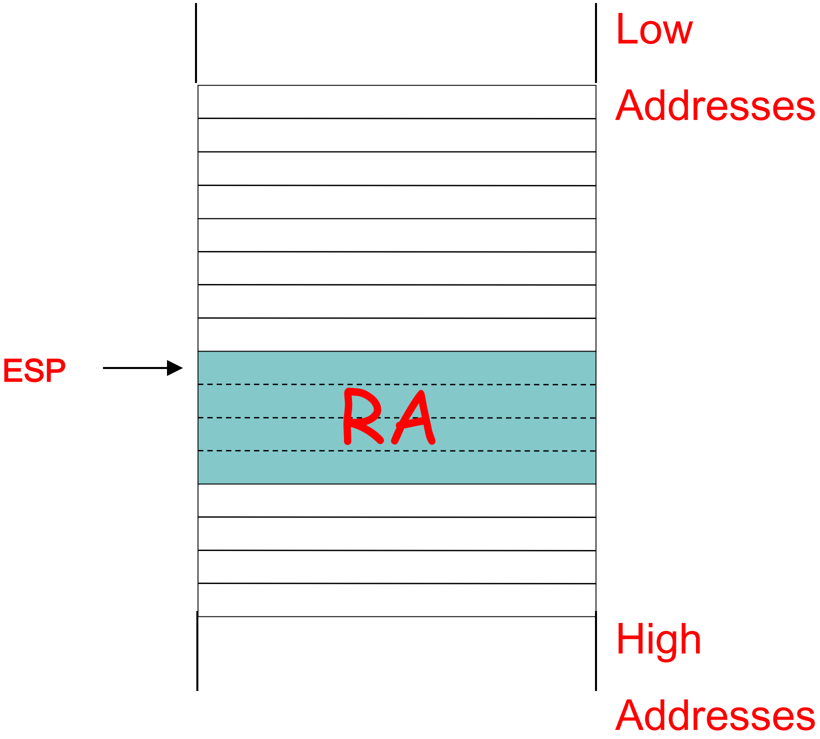
Two more assembly instructions (CALL & RET)



Call/return

Caller-side:

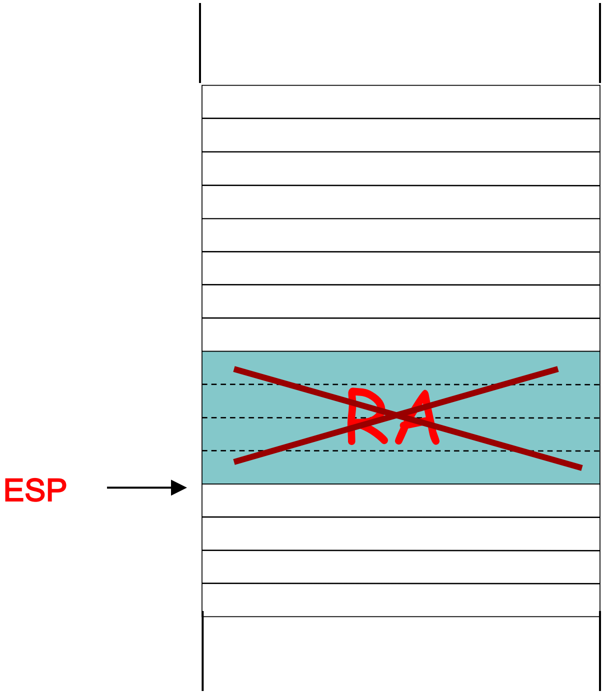
- CALL procedure_name
  - Pushes the *Return Address (RA)* (the code location right after *CALL* instruction) onto the call stack.
  - Jumps to the code location of *proc_name*.



Call/return

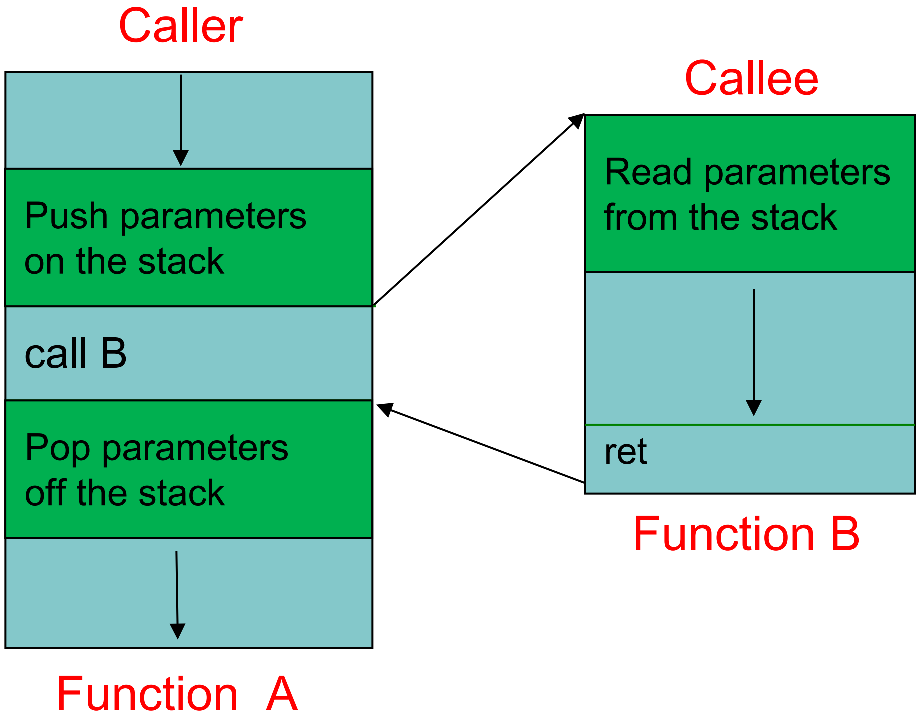
Callee-side:

- RET
  - Pops the *return address* from the call stack.
  - Jumps to the *return address* (the code right after the *CALL*).



Parameter passing

- Need to store parameters somewhere accessible by the function
  - Not too many registers
  - Use Stack


Caller
Push parameters on the stack
call B
Pop parameters off the stack
Function A
Callee
Read parameters from the stack
ret
Function B

Parameter passing

- Parameters are passed via stack.
- The caller prepare the parameters by pushing the parameters onto the stack.
  - The order is from right to left, i.e. push the last parameter first, and push the first parameter last.
- The callee reads the parameters by regular memory accessing (NOT pop!!). Addressing uses EBP as the base pointer.
- Example:
  - call proc_name(param$_1$, param$_2$, …, param$_n$)

Parameter passing (caller side)

At the caller:

- Before *call* instr., prepare parameters:

  push param_n

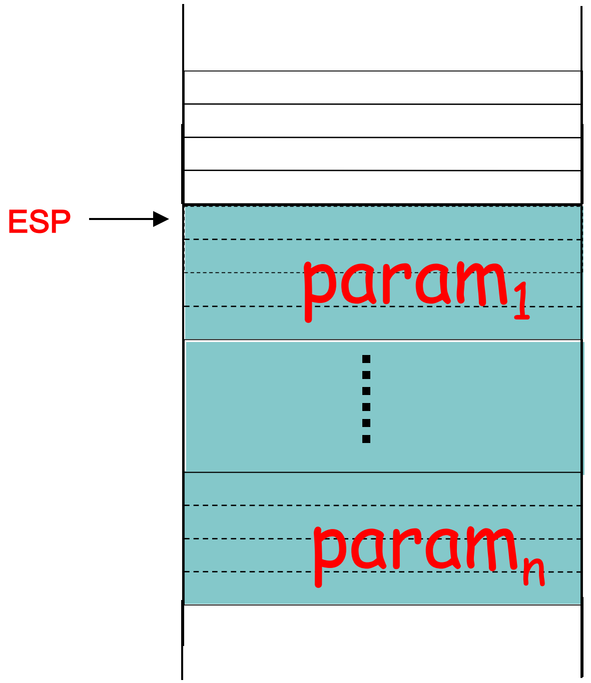
  ……..

  push param_1

IMPORTANT:

The order of PUSHs

is from right to left!!

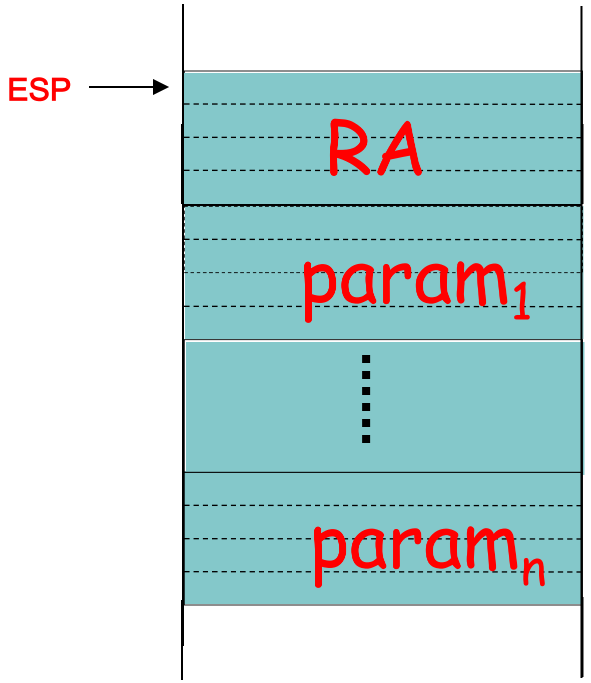


Parameter passing (caller side)

At the caller:

- *call* instr.:

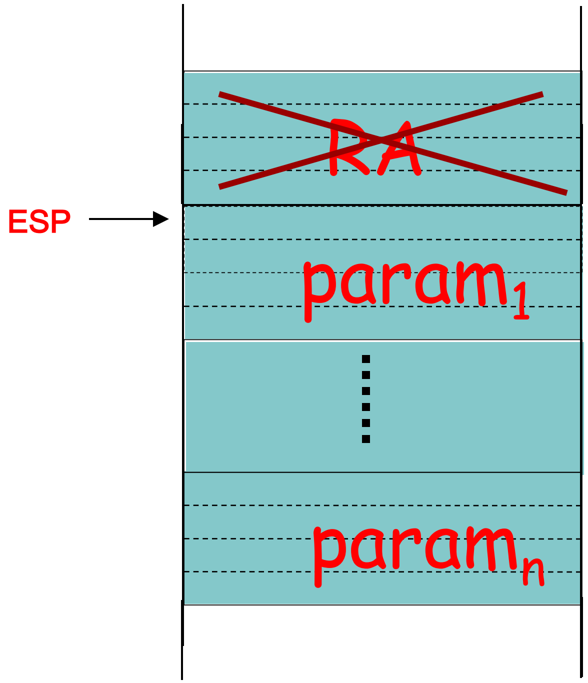
  call proc_name



Parameter passing (caller side)

At the caller:

- After the *call* returns:

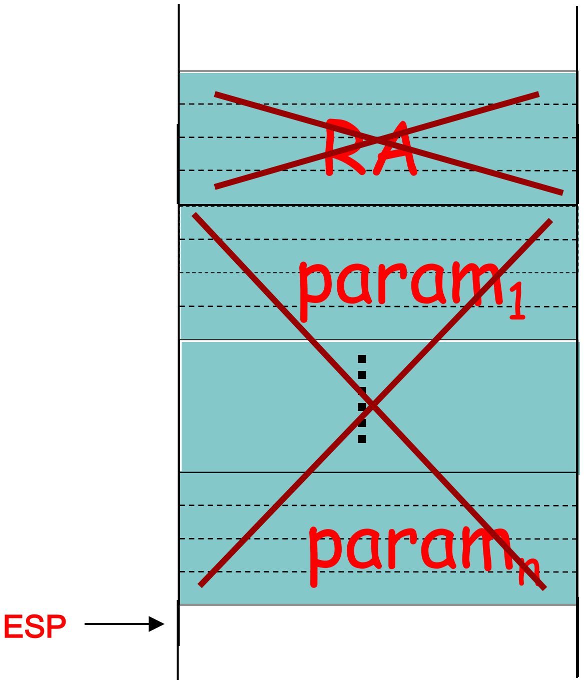


Parameter passing (caller side)

At the caller:

- After the *call* returns, restore ESP by adding # of bytes the parameters have occupied on the stack (discard parameters):

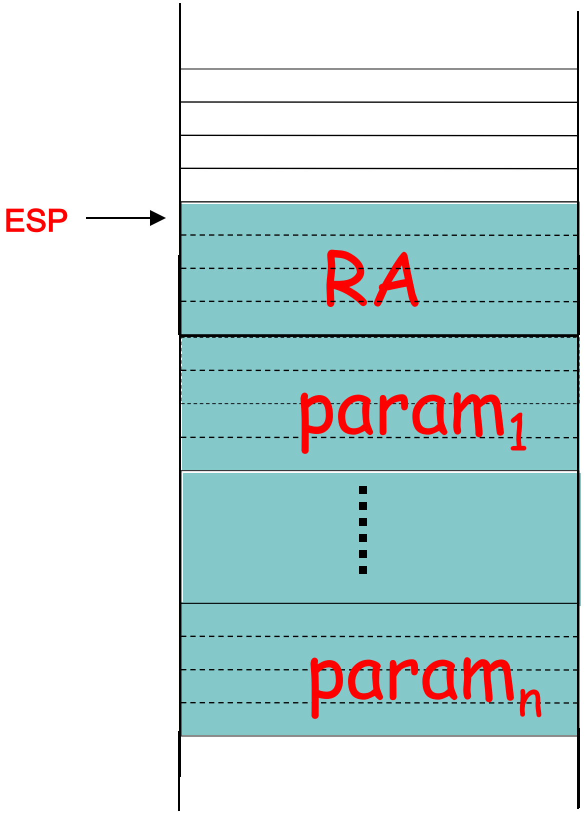
  add ESP,
      size_of_params



Parameter passing (callee side)

At the callee:

- At the very beginning of callee, i.e., at the entry point of the procedure:

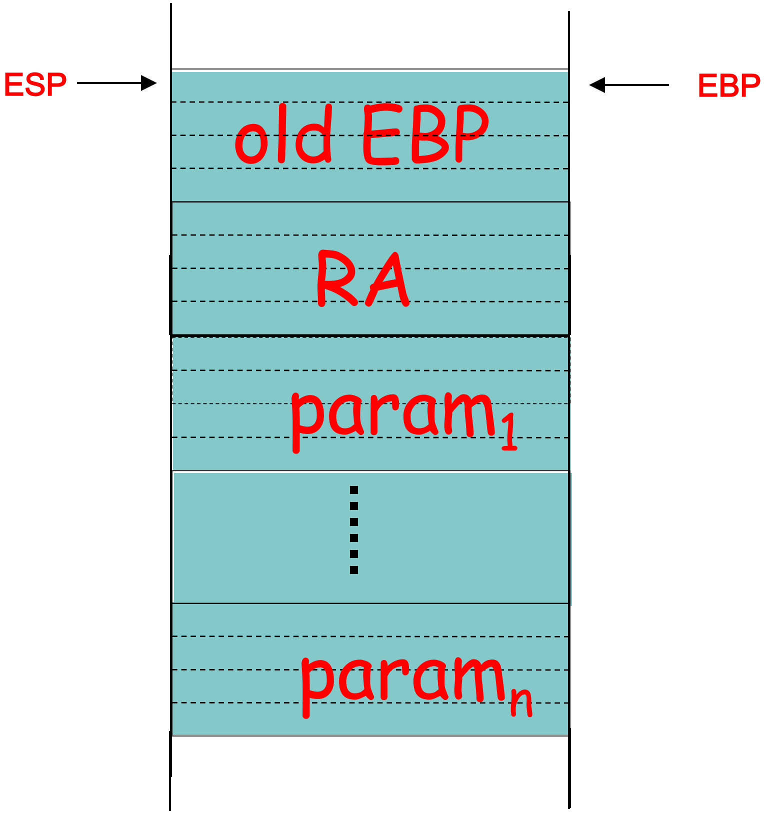


Parameter passing (callee side)

At the callee:

- Use EBP as base register to address the parameters.
- EBP is the base pointer to the current stack frame of the callee.
- The prologue to prepare EBP:

```
push EBP
move EBP, ESP
```



Parameter passing (callee side)

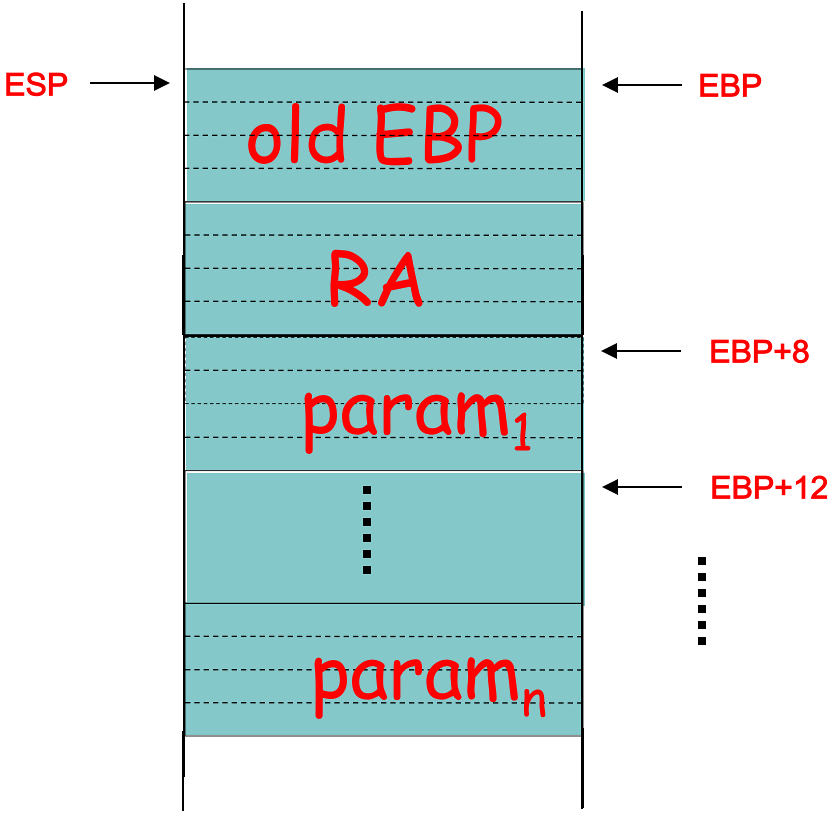
At the callee:

- Now use MOV to retrieve parameter values from the stack memory.

```
mov ebx, [EBP + 8]
mov ecx, [EBP + 12]
………
```

IMPORTANT:

Do NOT use POP to get the parameter values!!

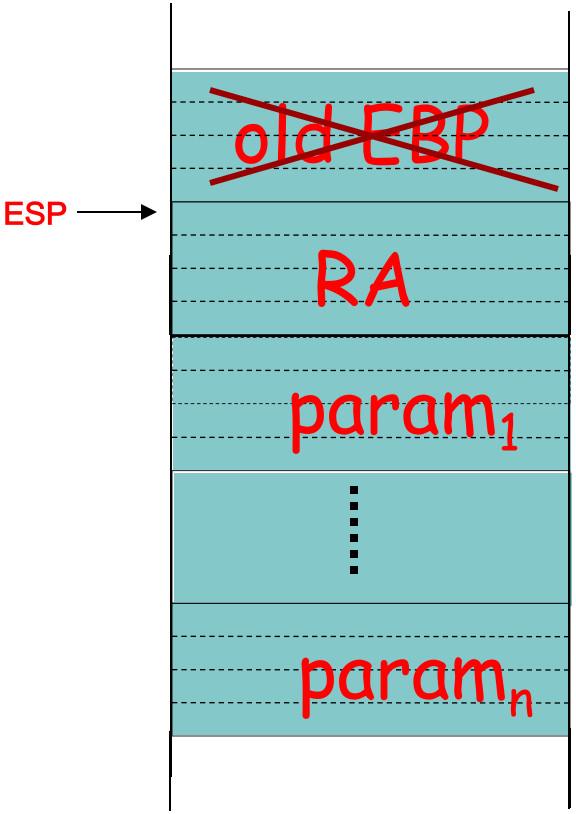


Parameter passing (callee side)

At the callee:

- The epilogue to restore the old EBP:

  pop EBP

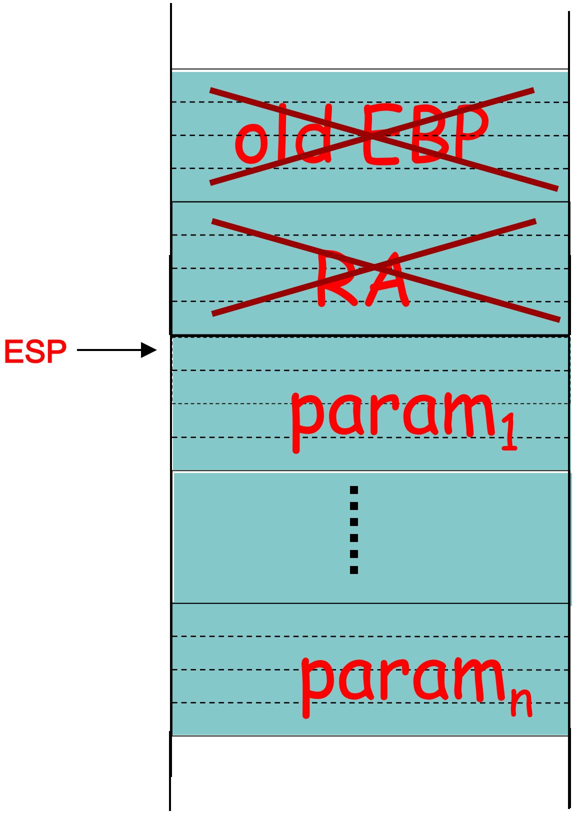


Parameter passing (callee side)

At the callee:

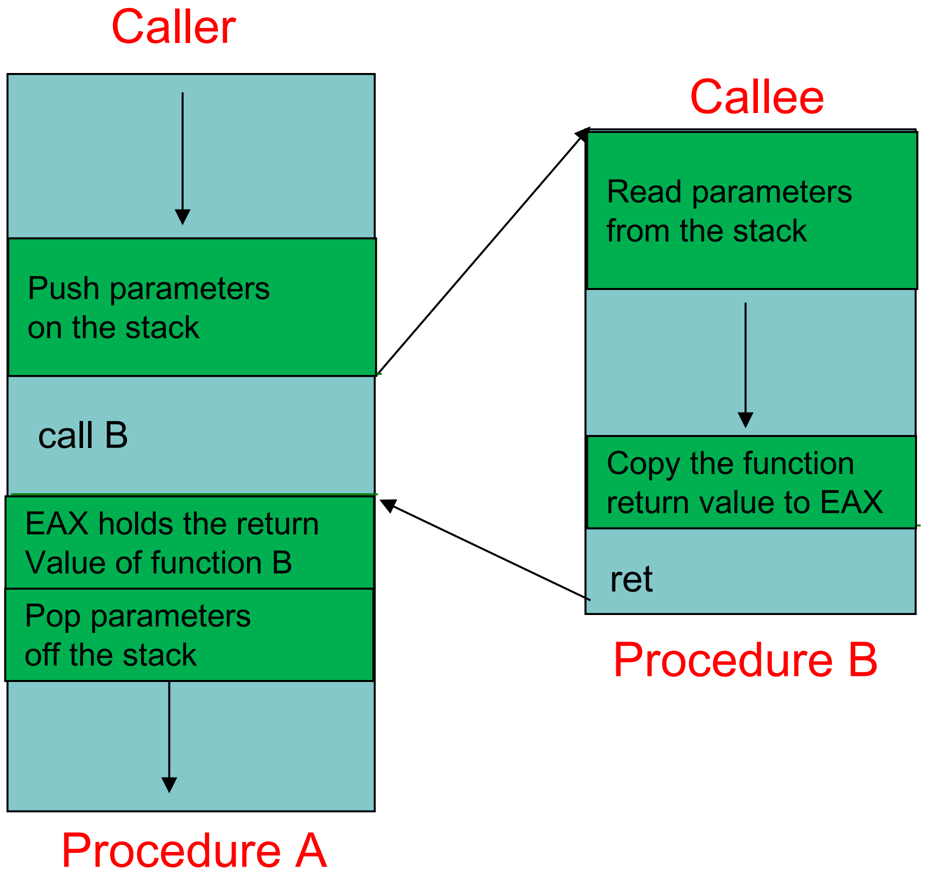
- Finally:

  RET

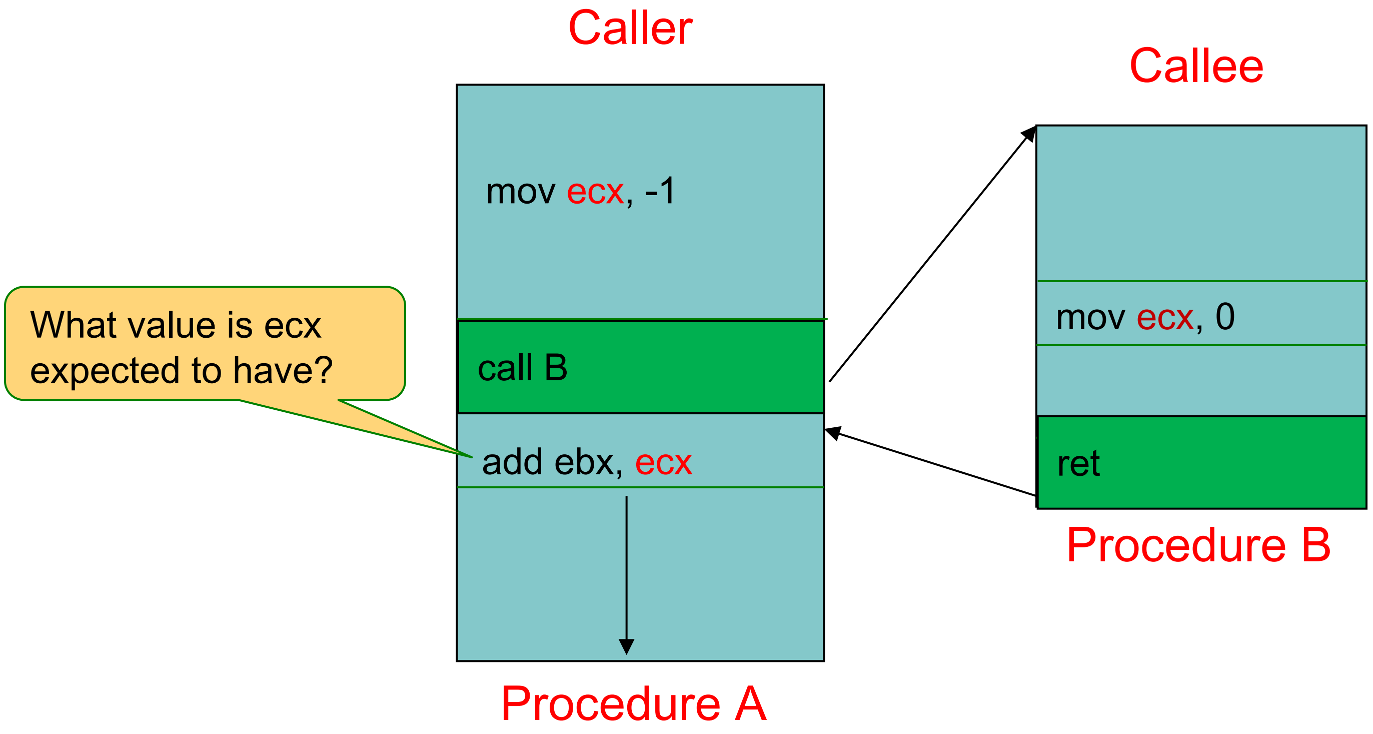


Return value

- EAX is used to store the return value.
- In the callee, save the return value to EAX before RET.
- In the caller, read the return value from EAX.


Caller
Push parameters on the stack
call B
EAX holds the return Value of function B
Pop parameters off the stack
Procedure A
Callee
Read parameters from the stack
Copy the function return value to EAX
ret
Procedure B

Why saving/restoring registers?



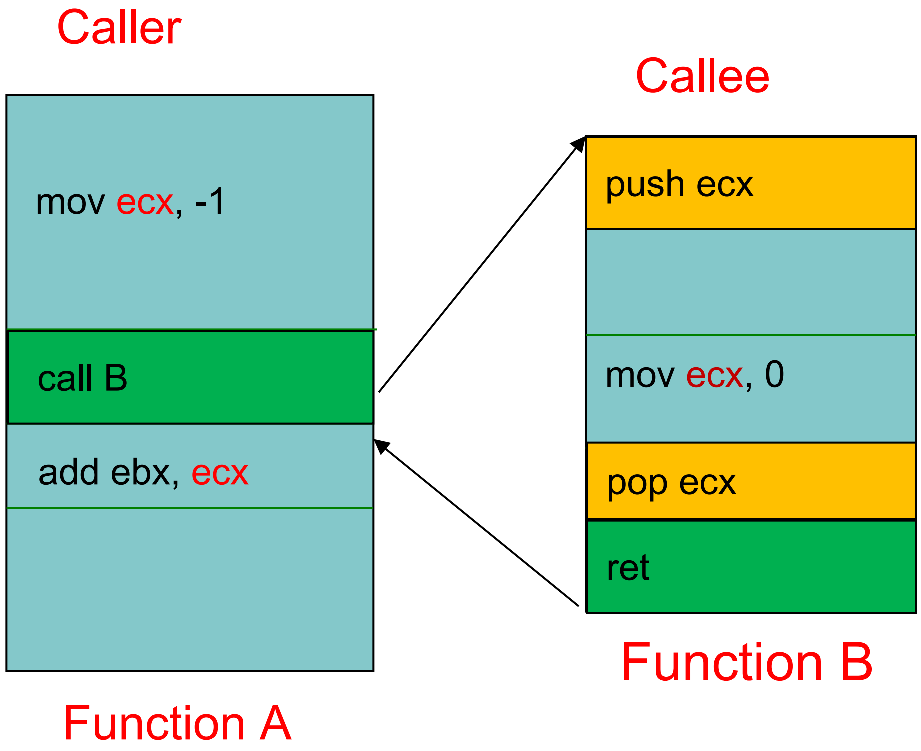
- Value of ECX should be saved across function call

Save/restore registers

- Sometimes, the caller and the callee need to use the same registers. How to solve the conflict?
  - Use stack
- Registers that may cause conflicts should be saved (pushed) onto the stack before executing callee procedure body, and should be restored (popped) after the call is done.
- Two general ways to save/restore registers
  - Callee-saved
  - Caller-saved

Callee-saved

- Save/restore is done in callee: register save/restore code as part of the callee prologue/epilogue.



Callee-saved

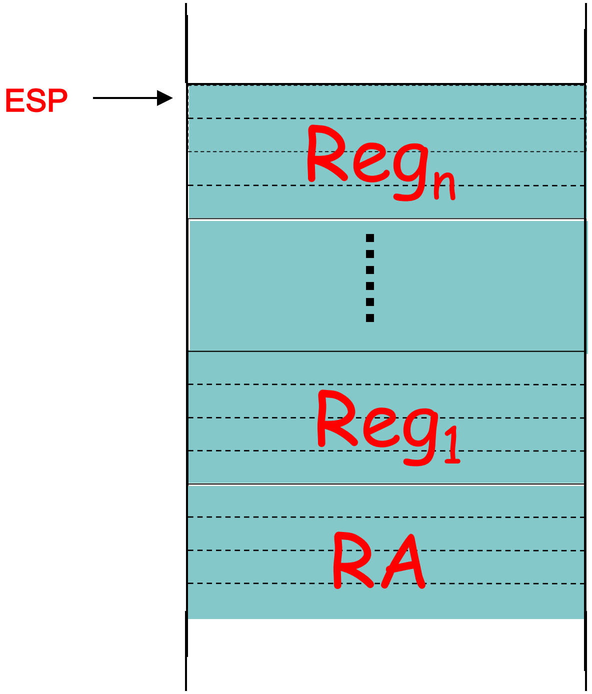
At the callee:

- In prologue:

  push reg_1

  ……..

  push reg_n



Callee-saved

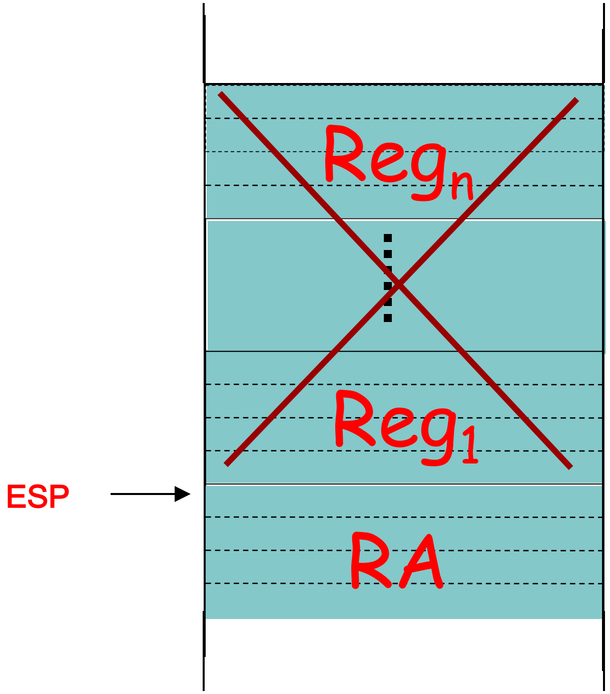
At the callee:

- In epilogue:

  pop reg_n

  ....

  pop reg_1

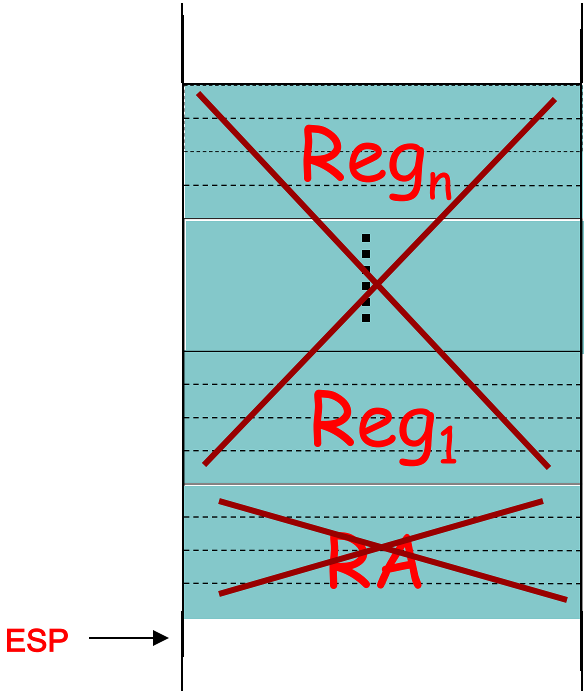


Callee-saved

At the callee:

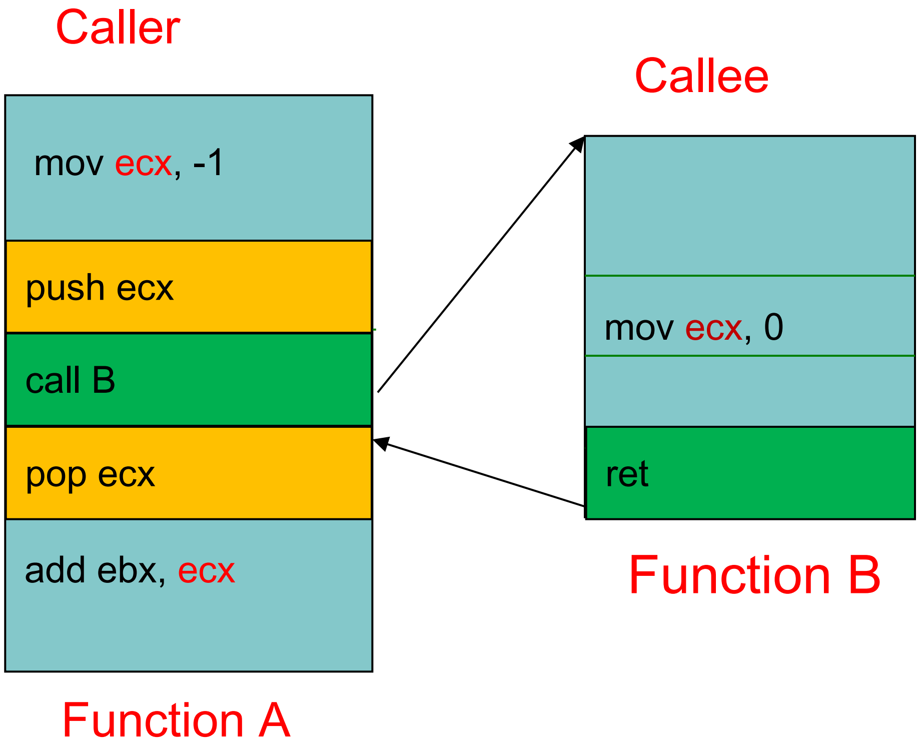
- Finally:

  RET



Caller-saved

- Save/restore is done in caller: push registers before the *call* instruction, and pop the registers after the *call* instruction is done.



Caller-saved

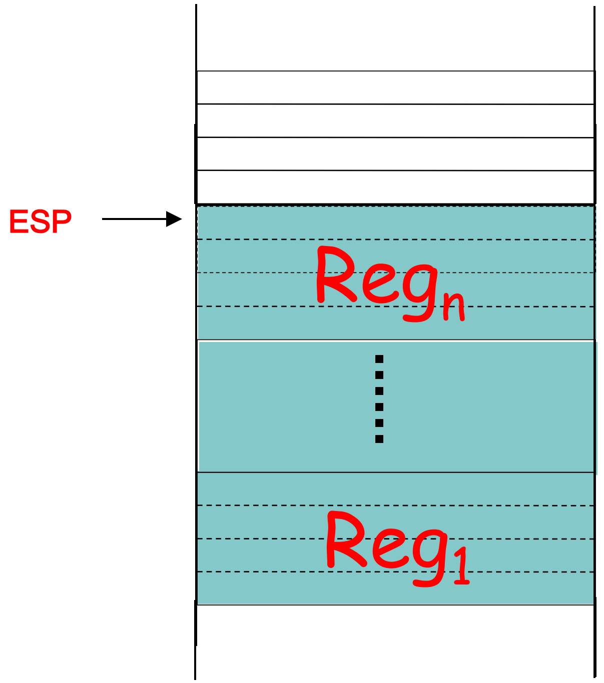
At the caller:

- Before *call* instr.:

  push reg_1

  ....

  push reg_n

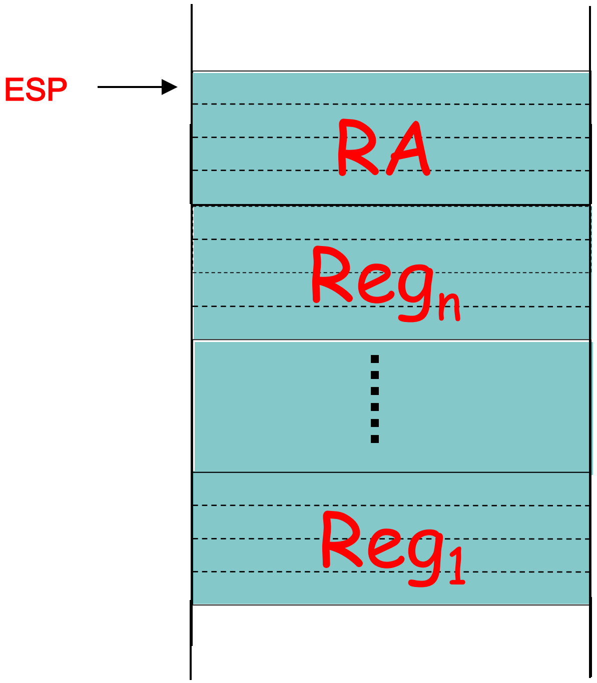


Caller-saved

At the caller:

- *call* instr.:

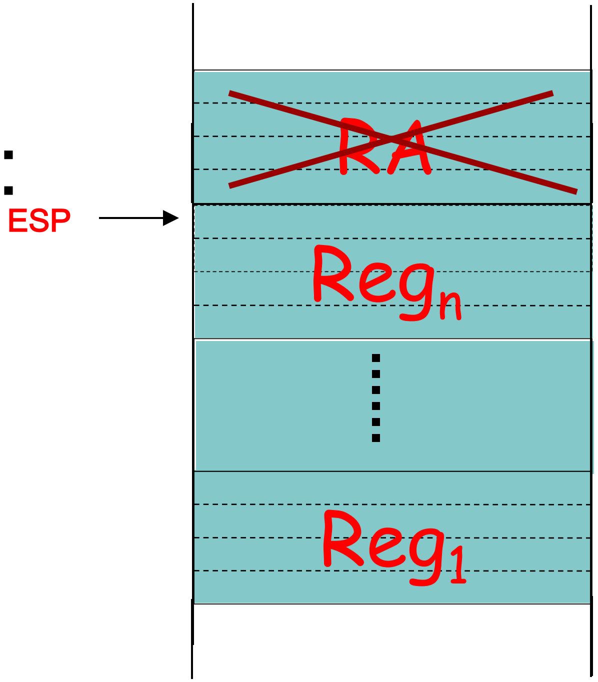
  call proc_name



Caller-saved

At the caller:

- After the *call* returns:



Caller-saved

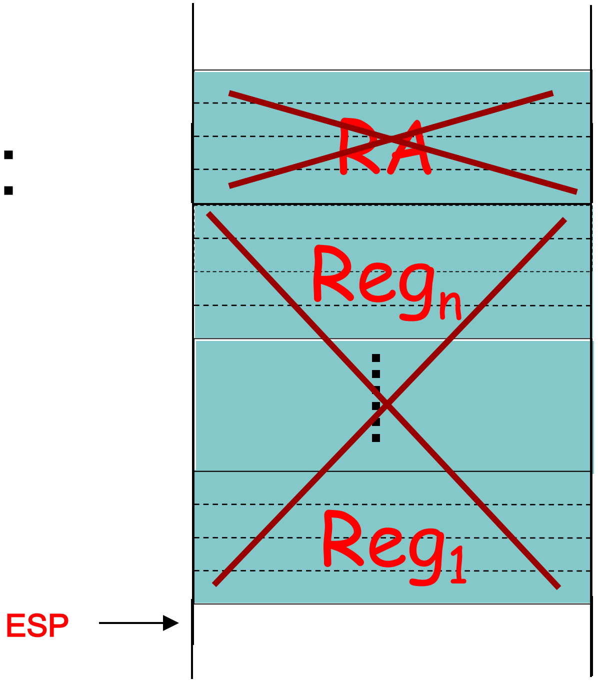
At the caller:

- After the *call* returns:

  pop reg_n

  ………

  pop reg_1



Convention

- **EAX, ECX, EDX** must be saved by the caller
- **EBX, EDI, ESI** must be saved by the callee

Save/restore registers

- IMPORTANT:

  Registers should be popped off the stack in the reverse order: the last register pushed is the first popped.

Allocating local variables (in callee)

```
void MyFunction() {
        int a, b, c;
        ...
        return
}
```

```
push ebp          ; save the value of ebp
mov ebp, esp      ; ebp now points to the top of the stack
sub esp, 12       ; space allocated on the stack for the local variables
```

```
a = 10;
b = 5;
c = 2;
```

```
mov [ebp - 4], 10  ; location of variable a
mov [ebp - 8], 5   ; location of b
mov [ebp - 12], 2  ; location of c
```

Deallocating local variables

```
void MyFunction() {
                int a, b, c;
                ...
                return
}
```

```
push ebp            ; save the value of ebp
mov ebp, esp        ; ebp now points to the top of the stack
sub esp, 12         ; space allocated on the stack for the local variables
…
…
mov esp, ebp
pop ebp
ret
```

Call-by-value vs. Call-by-reference

- Call-by-value parameter passing:
  - In the caller, the *value* of the actual parameter is passed to the callee (pushed to the stack).

    ```
    push param
    ```

  - In the callee, the *value* of the parameter is retrieved from the stack and used afterwards.

    ```
    mov ebx, [ebp + 8]
    ```

Call-by-value vs. Call-by-reference

- Call-by-reference parameter passing:
  - In the caller, the memory *address* of the actual parameter is passed to the callee (pushed to the stack).

    ```
    lea edx, param
    push edx
    ```

  - LEA (Load Effective Address)

    Usage: LEA reg, mem

  - In the callee, the *address* of the parameter is retrieved from the stack and used afterwards.

    ```
    mov ebx, [ebp + 8]
    ```

    Afterwards: [ebx] should be used to access this call-by-reference parameter.

Two’s Complement

- Used for signed representation of numbers.
- Complement w.r.t. 2^N
  - 5 + Two’s complement of 5

    = 0101 + 1011 =10000 (2^4)
- For 8-bit register:
  - Unsigned Integer range: 0 to 255
  - Signed Integer range : -128 to 127
- Most significant bit signifies the sign
  - 0 for positive, 1 for negative
- Same arithmetic calculation for signed numbers as unsigned numbers

Two’s Complement

- Calculation:
  - First, invert the binary representation.
  - Second, add 1 to it.
- Example:
  - Integer = -12
  - 12 = 0000 1100
  - -12 = 1111 0011 + 1 = 1111 0100
- Calculate 44 – 12 = 44 + (-12)

  0010 1100

  + 1111 0100

  0010 0000 i.e. 32