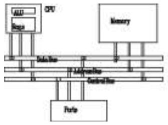
Procedure

Computer Organization and Assembly Languages

Yung-Yu Chuang

with slides by Kip Irvine

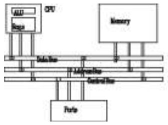
Overview



- Stack Operations
- Defining and Using Procedures
- Stack frames, parameters and local variables
- Recursion
- Related directives

Stack operations

Stacks

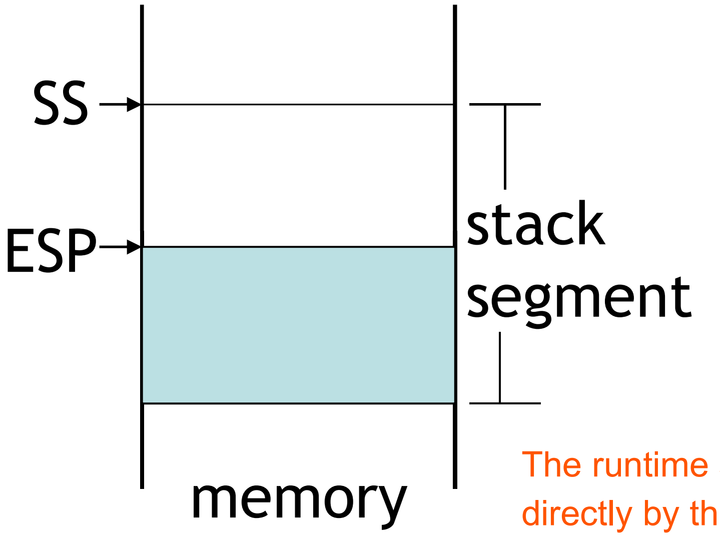


- LIFO (Last-In, First-Out) data structure.
- push/pop operations
- You probably have had experiences on implementing it in high-level languages.
- Here, we concentrate on *runtime stack*, directly supported by hardware in the CPU. It is essential for calling and returning from procedures.

The runtime stack stores information about the active subroutines of a computer program.

Runtime stack

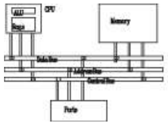
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) * : point to the top of the stack usually modified by **`CALL`**, **`RET`**, **`PUSH`** and **`POP`**



The runtime stack is a memory array managed directly by the CPU, using the ESP register, known as the stack pointer register.

* SP in Real-address mode

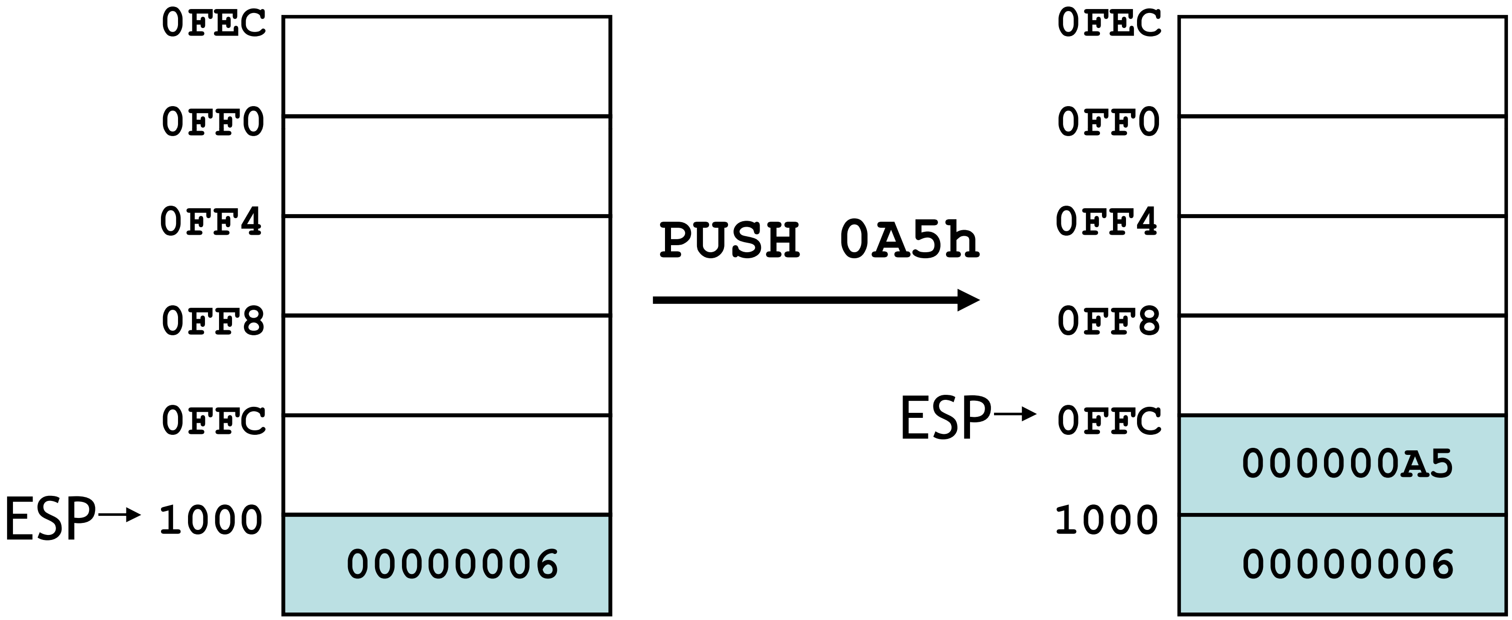
PUSH and **POP** instructions



- **PUSH** syntax:
  - **PUSH *r/m16***
  - **PUSH *r/m32***
  - **PUSH *imm32***
- **POP** syntax:
  - **POP *r/m16***
  - **POP *r/m32***

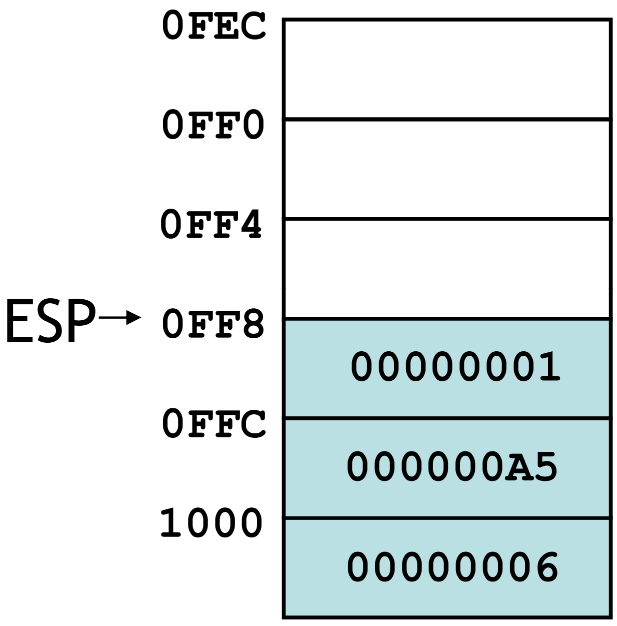
PUSH operation (1 of 2)

- A **push** operation decrements the stack pointer by 2 or 4 (depending on operands) and copies a value into the location pointed to by the stack pointer.



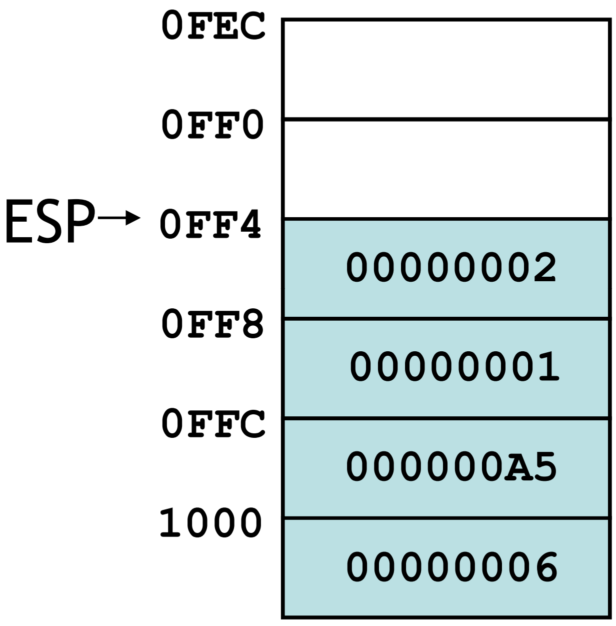
PUSH operation (2 of 2)

- The same stack after pushing two more integers:



0FEC
0FF0
0FF4
ESP→ 0FF8
00000001
0FFC
000000A5
1000
00000006

PUSH 01h

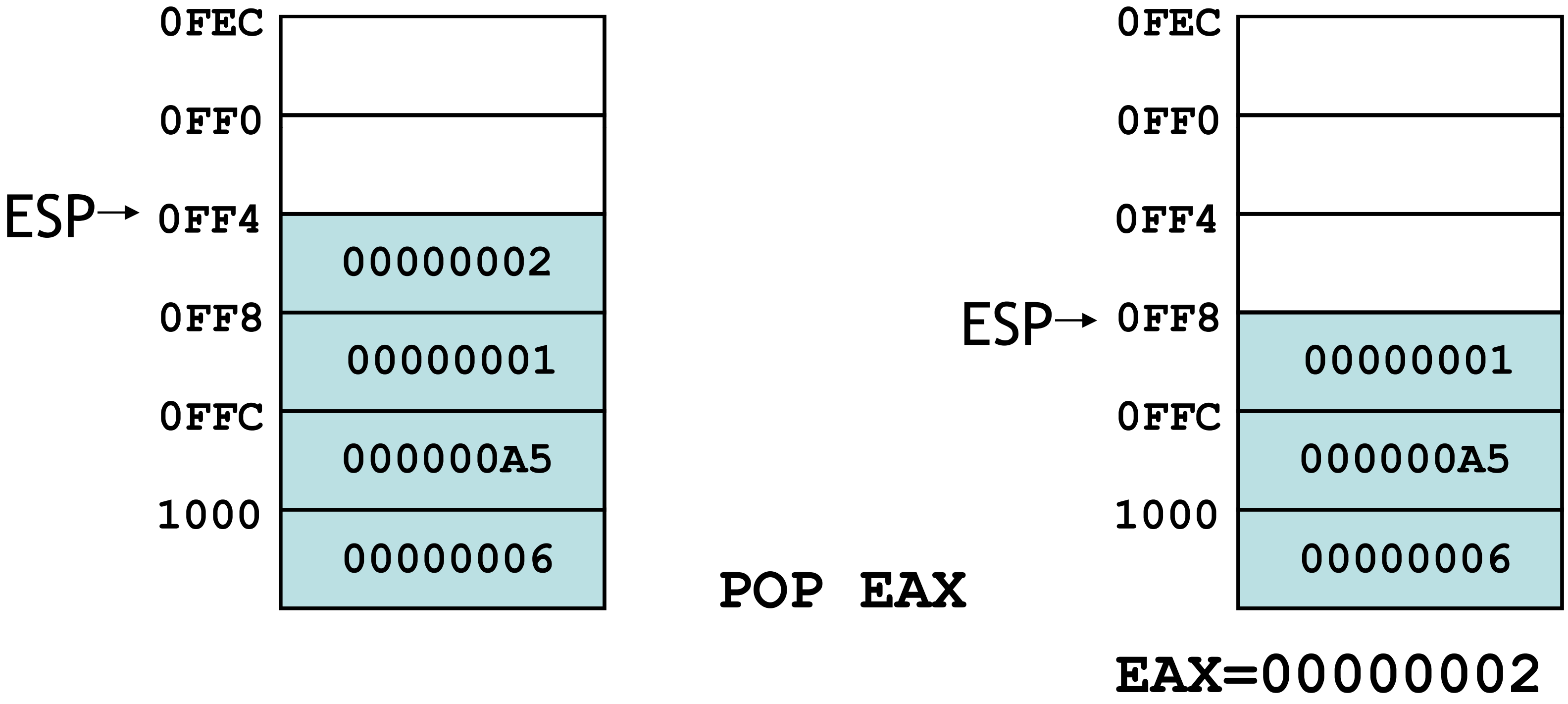


0FEC
0FF0
ESP→ 0FF4
00000002
0FF8
00000001
0FFC
000000A5
1000
00000006

PUSH 02h

POP operation

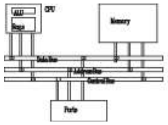
- Copies value at stack[ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4, depending on the attribute of the operand receiving the data



POP EAX

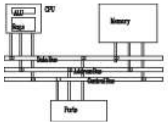
EAX=00000002

When to use stacks



- Temporary save area for registers
- To save return address for CALL
- To pass arguments
- Local variables
- Applications which have LIFO nature, such as reversing a string

Example of using stacks



Save and restore registers when they contain important values. Note that the **PUSH** and **POP** instructions are in the opposite order:

```
push esi            ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal  ; starting OFFSET
mov ecx,LENGTHOF dwordVal; number of units
mov ebx,TYPE dwordVal ;size of a doubleword
call DumpMem        ; display memory

pop ebx             ; opposite order
pop ecx
pop esi
```

Example: Nested Loop

When creating a nested loop, push the outer loop counter before entering the inner loop:

```
   mov ecx,100      ; set outer loop count
L1:                 ; begin the outer loop
   push ecx         ; save outer loop count

   mov ecx,20       ; set inner loop count
L2:                 ; begin the inner loop
   ;
   ;
   loop L2          ; repeat the inner loop

   pop ecx          ; restore outer loop count
   loop L1          ; repeat the outer loop
```

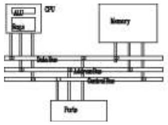
Example: reversing a string

```
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) – 1

.code
main PROC
; Push the name on the stack.
  mov ecx,nameSize
  mov esi,0
L1:
  movzx eax,aName[esi]    ; get character
  push eax                ; push on stack
  inc esi
  Loop L1
```

Example: reversing a string

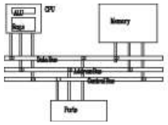
```
; Pop the name from the stack, in reverse,
; and store in the aName array.
  mov ecx,nameSize
  mov esi,0
L2:
  pop eax            ; get character
  mov aName[esi],al  ; store in string
  inc esi
  Loop L2

  exit
main ENDP
END main
```

Related instructions



- **PUSHFD** and **POPFD**
  - push and pop the EFLAGS register
  - **LAHF**, **SAHF** are other ways to save flags
- **PUSHAD** pushes the 32-bit general-purpose registers on the stack in the following order
  - **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**
- **POPAD** pops the same registers off the stack in reverse order
  - **PUSHA** and **POPA** do the same for 16-bit registers

Example

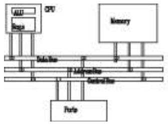


```
MySub PROC
     pushad
     ...
     ; modify some register
     ...
     popad
     ret
MySub ENDP
```

Do not use this if your procedure uses registers for return values

Defining and using procedures

Creating Procedures



- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function
- Following is an assembly language procedure named sample:

```
sample PROC
   .
   .
   ret
sample ENDP
```

A named block of statements that ends with a return.

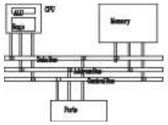
Documenting procedures

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- Receives: A list of input parameters; state their usage and requirements.
- Returns: A description of values returned by the procedure.
- Requires: Optional list of requirements called preconditions that must be satisfied before the procedure is called.

For example, a procedure of drawing lines could assume that display adapter is already in graphics mode.

Example: SumOf procedure



```
;-----------------------------------------------
SumOf PROC
;
; Calculates and returns the sum of three 32-bit
;   integers.
; Receives: EAX, EBX, ECX, the three integers.
;           May be signed or unsigned.
; Returns: EAX = sum, and the status flags
;          (Carry, Overflow, etc.) are changed.
; Requires: nothing
;-----------------------------------------------
   add eax,ebx
   add eax,ecx
   ret
SumOf ENDP
```

CALL and **RET** instructions

- The **CALL** instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into **EIP**
- The **RET** instruction returns from a procedure
  - pops top of stack into **EIP**
- We used **jl** and **jr** in our toy computer for **CALL** and **RET**, **BL** and **MOV PC, LR** in ARM.

CALL-RET example (1 of 2)

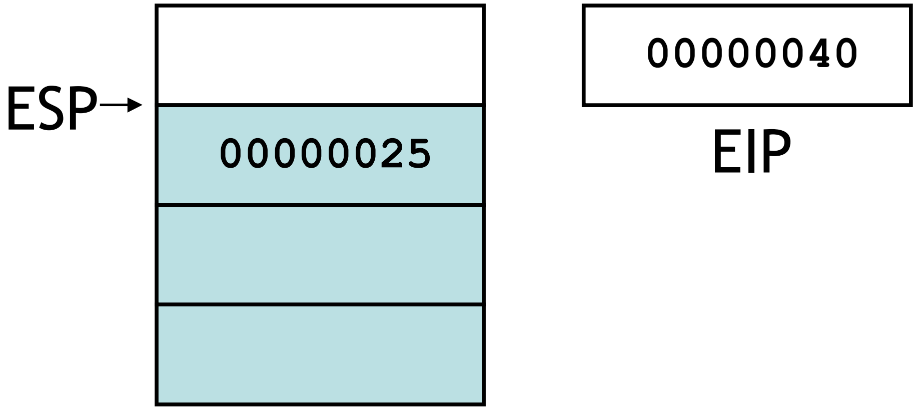
```
main PROC
   00000020 call MySub
   00000025 mov eax,ebx
   .
   .
main ENDP

MySub PROC
   00000040 mov eax,edx
   .
   .
   ret
MySub ENDP
```

0000025 is the offset of the instruction immediately following the CALL instruction

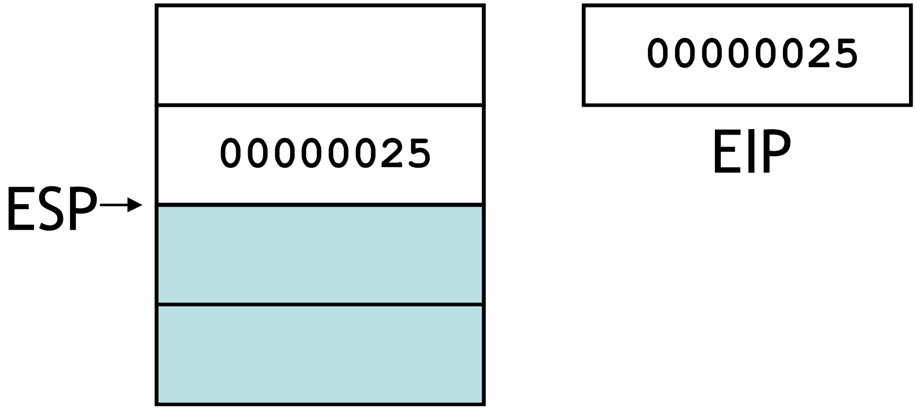
00000040 is the offset of the first instruction inside MySub

CALL-RET example (2 of 2)

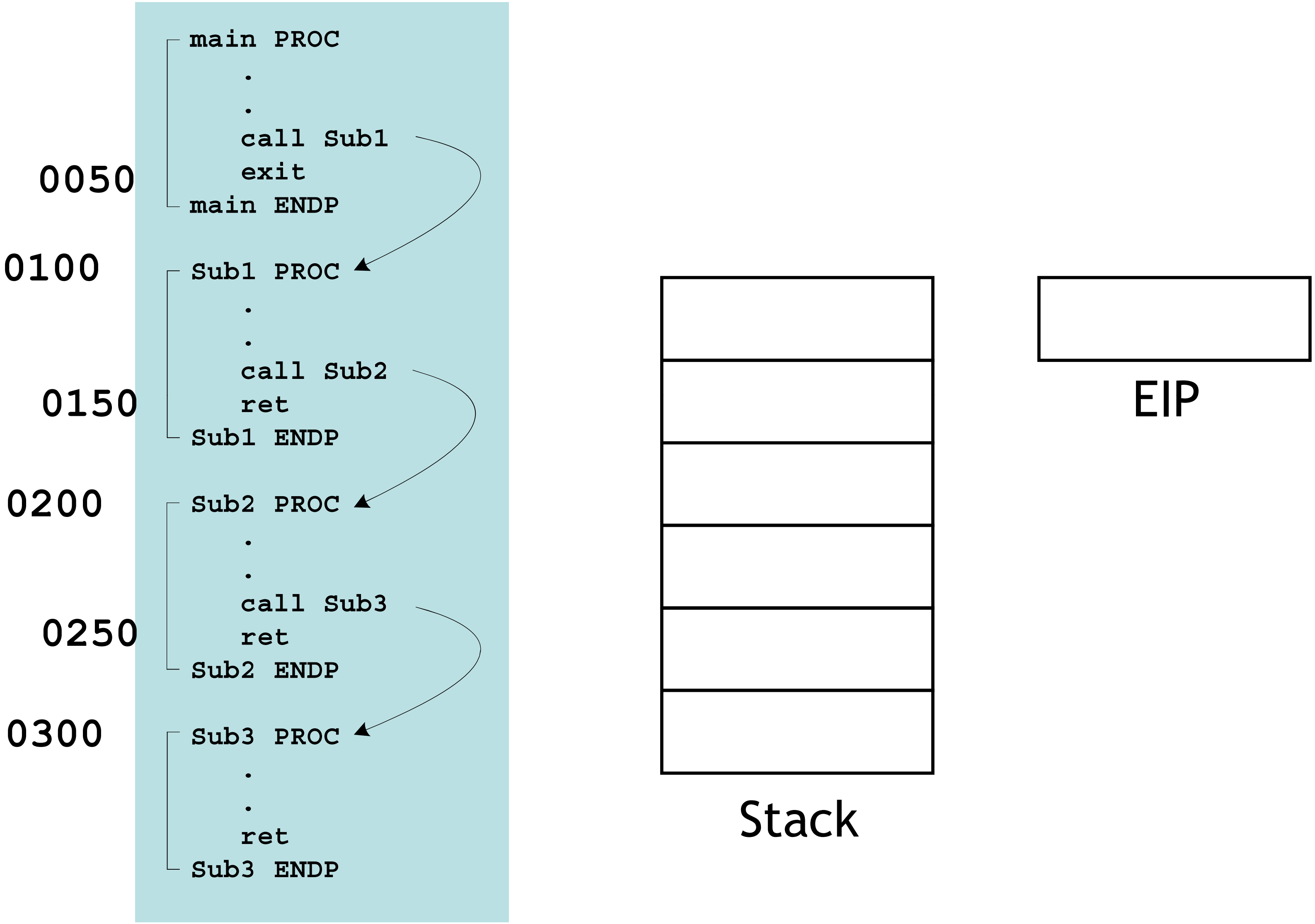
The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP



The RET instruction pops 00000025 from the stack into EIP



Nested procedure calls



```
      main PROC
          .
          .
          call Sub1
0050      exit
      main ENDP

0100  Sub1 PROC
          .
          .
          call Sub2
0150      ret
      Sub1 ENDP

0200  Sub2 PROC
          .
          .
          call Sub3
0250      ret
      Sub2 ENDP

0300  Sub3 PROC
          .
          .
          ret
      Sub3 ENDP
```

Local and global labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main PROC
   jmp L2                   ; error!
L1::                        ; global label
   exit
main ENDP

sub2 PROC
L2:                         ; local label
   jmp L1                   ; ok
   ret
sub2 ENDP
```

Procedure parameters (1 of 3)

- A good procedure might be usable in many different programs
- Parameters help to make procedures flexible because parameter values can change at runtime
- General registers can be used to pass parameters

Procedure parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
   mov esi,0            ; array index
   mov eax,0            ; set the sum to zero

L1:
   add eax,myArray[esi] ; add each integer to sum
   add esi,4            ; point to next integer
   loop L1              ; repeat for array size

   mov theSum,eax       ; store the sum
   ret
ArraySum ENDP
```

Procedure parameters (3 of 3)

This version returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Recevies: ESI points to an array of doublewords,
;           ECX = number of array elements.
; Returns:  EAX = sum
;------------------------------------------------
    push esi
    push ecx
    mov eax,0               ; set the sum to zero
L1: add eax,[esi]           ; add each integer to sum
    add esi,4               ; point to next integer
    loop L1                 ; repeat for array size
    pop ecx
    pop esi
    ret
ArraySum ENDP
```

Calling `ArraySum`

```
.data
array DWORD 10000h, 20000h, 30000h, 40000h
theSum DWORD ?
.code
main PROC
  mov     esi, OFFSET array
  mov     ecx, LENGTHOF array
  call    ArraySum
  mov     theSum, eax
```

USES operator

- Lists the registers that will be saved (to avoid side effects) (return register shouldn't be saved)

```
ArraySum PROC USES esi ecx
   mov eax,0  ; set the sum to zero
   ...
```

MASM generates the following code:

```
ArraySum PROC
   push esi
   push ecx
   .
   .
   pop ecx
   pop esi
   ret
ArraySum ENDP
```