Stack Operations

Stack

A LIFO (Last In First Out) data structure

□ New value is added to the top of stack

□ Existing values are removed from the top of stack

□ An essential part of calling from and returning to the procedures

□ Real life example

□ A stack of plates

Runtime Stack (1/3)

□ A memory array managed by CPU using ESP/SP (Extended Stack Pointer) register and SS (Stack Segment)

□ ESP/SP always points to the last value pushed on the top of stack and holds offset of that value in Stack Segment

□ ESP/SP cannot be manipulated directly instead it can be modified indirectly by instructions such as PUSH, POP, CALL, RET

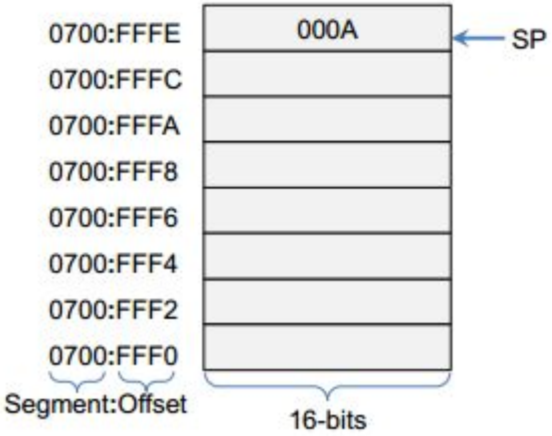
Runtime Stack (2/3)

- In protected mode i.e. 32-bit mode, size of each stack location is 32-bits
- In real-address mode i.e. 16-bit mode, size of each stack location is 16-bits
- emu8086 uses real-address mode
- Runtime Stack is different from Stack Abstract Data Type which is typically written in a HLL

Runtime Stack (3/3)

□ SS contains the base address of Stack Segment

□ SP contains the offset of value at the top of stack



Push Operation (1/3)

□ A push operation in stack puts the value on the top location available in stack and decrements the stack pointer by size of stack element

□ Size of each stack element is 32 bits in protected address mode

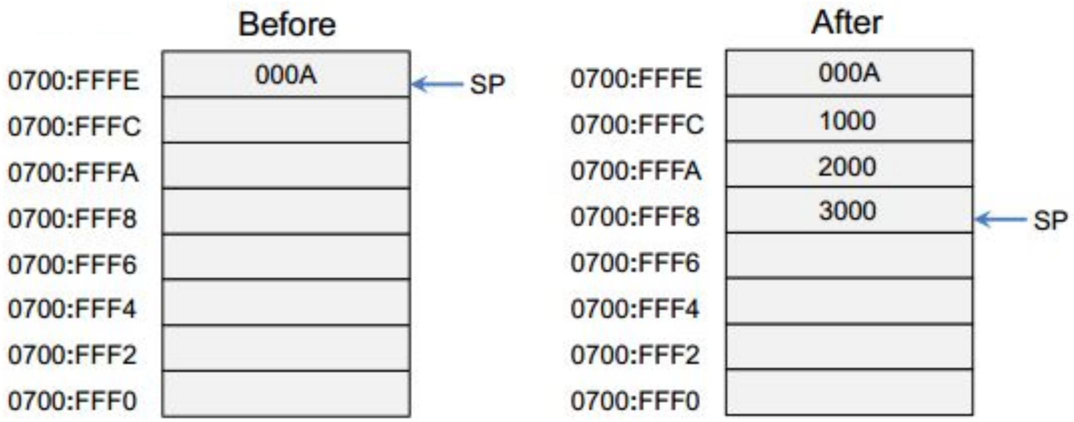
□ Size of each stack element is 16 bits in real address mode

Push Operation (2/3)

- SP is decremented by 2 with each push operation

- These values are pushed on stack

- 1000h
- 2000h
- 3000h



Before

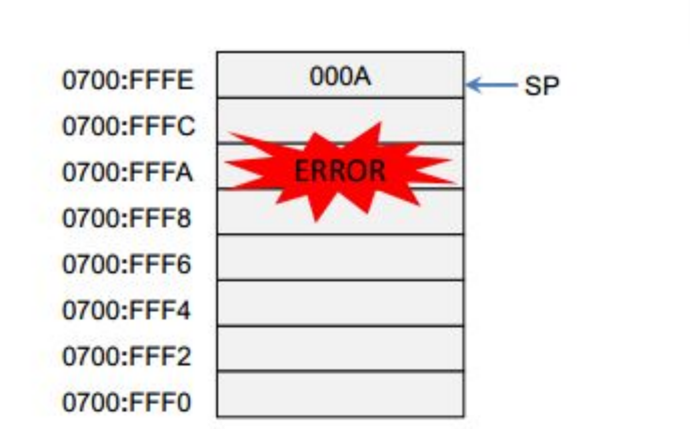
0700:FFFE	000A	← SP
0700:FFFC		
0700:FFFA		
0700:FFF8		
0700:FFF6		
0700:FFF4		
0700:FFF2		
0700:FFF0		

After

0700:FFFE	000A	
0700:FFFC	1000	
0700:FFFA	2000	
0700:FFF8	3000	← SP
0700:FFF6		
0700:FFF4		
0700:FFF2		
0700:FFF0		

Push Operation (3/3)

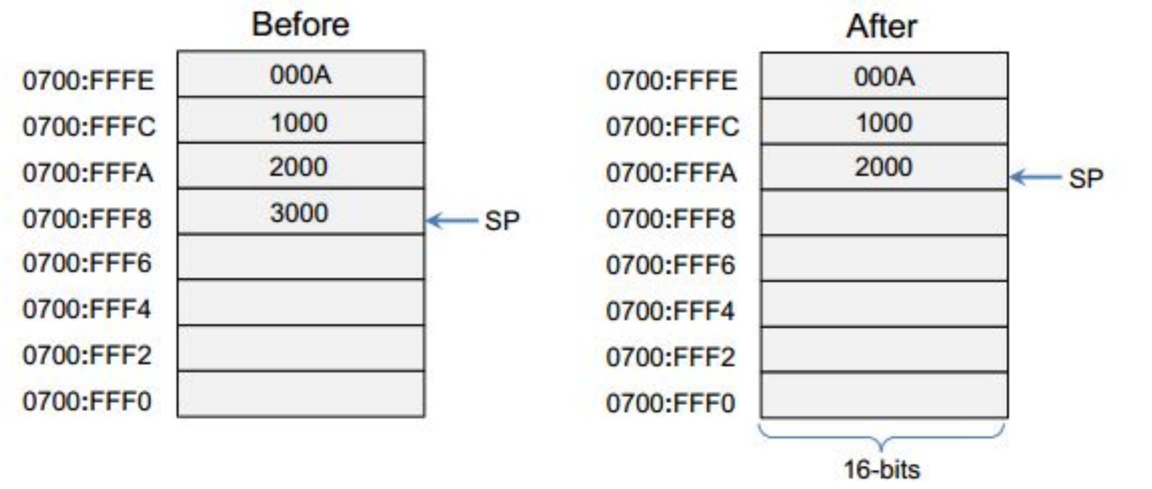
□ This value is pushed on stack
1 000 0000h



Pop Operation

□ Removes value from top of stack

□ SP is incremented by stack element size with each pop operation



Before

0700:FFFE	000A	
0700:FFFC	1000	
0700:FFFA	2000	
0700:FFF8	3000	← SP
0700:FFF6		
0700:FFF4		
0700:FFF2		
0700:FFF0		

After

0700:FFFE	000A	
0700:FFFC	1000	
0700:FFFA	2000	← SP
0700:FFF8		
0700:FFF6		
0700:FFF4		
0700:FFF2		
0700:FFF0		

PUSH Instruction

□ PUSH instruction is executed in two steps

□ First decrements SP by the size of stack element

□ Then copies the source operand on top of stack

□ PUSH instruction formats

□ PUSH reg/mem16 →contents of 16-bit register or 16-bit memory location is pushed on stack

□ PUSH imm16 →16-bit immediate value is pushed on stack

□ Examples are

□ PUSH AX

□ PUSH 10h

POP Instruction

- POP instruction is executed in two steps

- First the contents of stack element pointed to by SP are copied into destination operand

- Then SP is incremented by the size of stack element

- Only one POP instruction formats

- POP reg/mem16 → copies the value pointed to by SP into 16-bit register or 16-bit memory location

- Examples are
- POP AX
- POP var ;where var is a 16-bit memory location

PUSHF and POPF Instructions

□ PUSHF is used to push EFLAGS register on the stack

□ POPF pops the stack into EFLAGS register

□ When using these instructions, make sure program’s execution path does not skip over POPF instruction

□ Syntax is
□ PUSHF
□ POPF

PUSHA and POPA Instructions

□ PUSHA instruction pushes all 16-bit general purpose register on the stack in given order

□ AX, CX, DX, BX, SP, BP, SI, DI

□ POPA instruction pops the same registers in the reverse order

□ Useful when modifying many general purpose registers inside a procedure

Stack Applications

- Registers can be saved temporarily when used for more than one purpose
- When CALL instruction executed, return address is saved on the stack
- Arguments are passed to a subroutine by pushing them on the stack
- Stack can be used as temporary storage for local variables inside a subroutine

LECTURE 2

Defining and Using Procedures

- □ Creating a Procedure
- □ CALL and RET instructions
- □ Nested Procedure Calls
- □ Local and Global Labels
- □ Procedure Parameters
- □ USES Operator

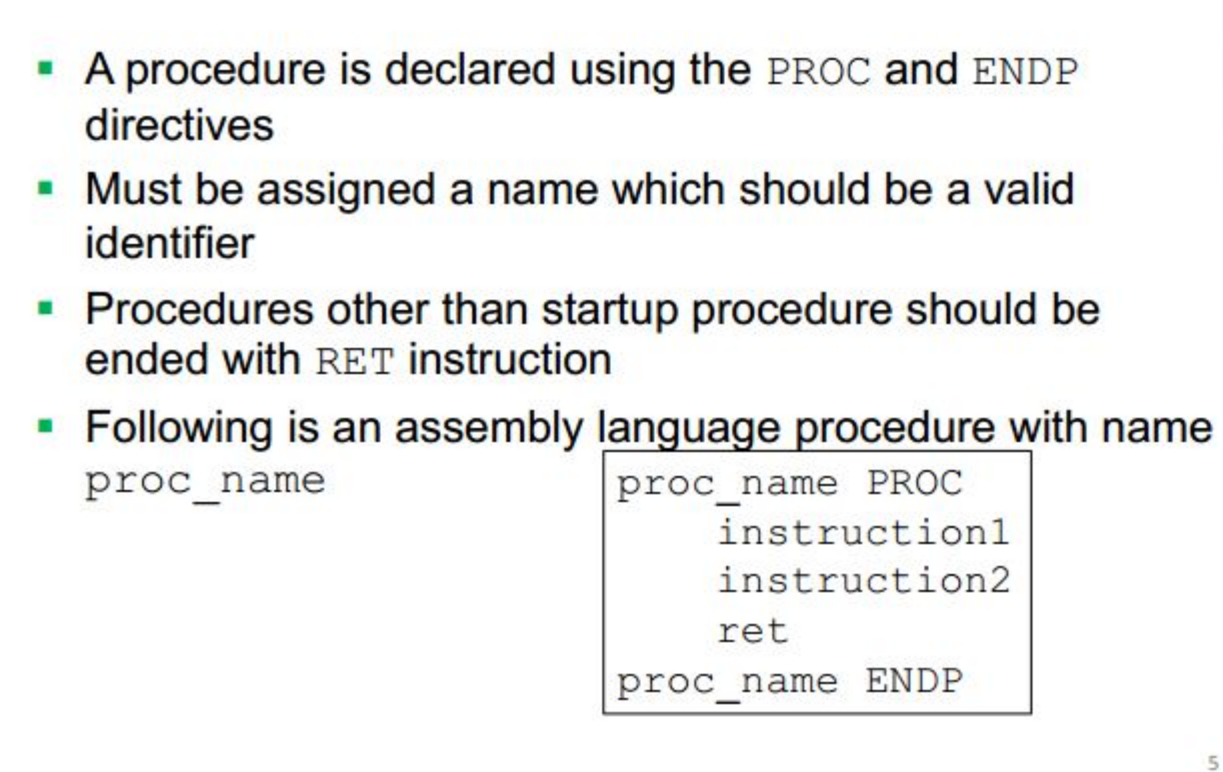
Procedure

- A complex code can be divided in different independent elements

- Such elements are called functions in C++ and Procedures in assembly language

- Procedure is a named block of statements that ends with a return statement

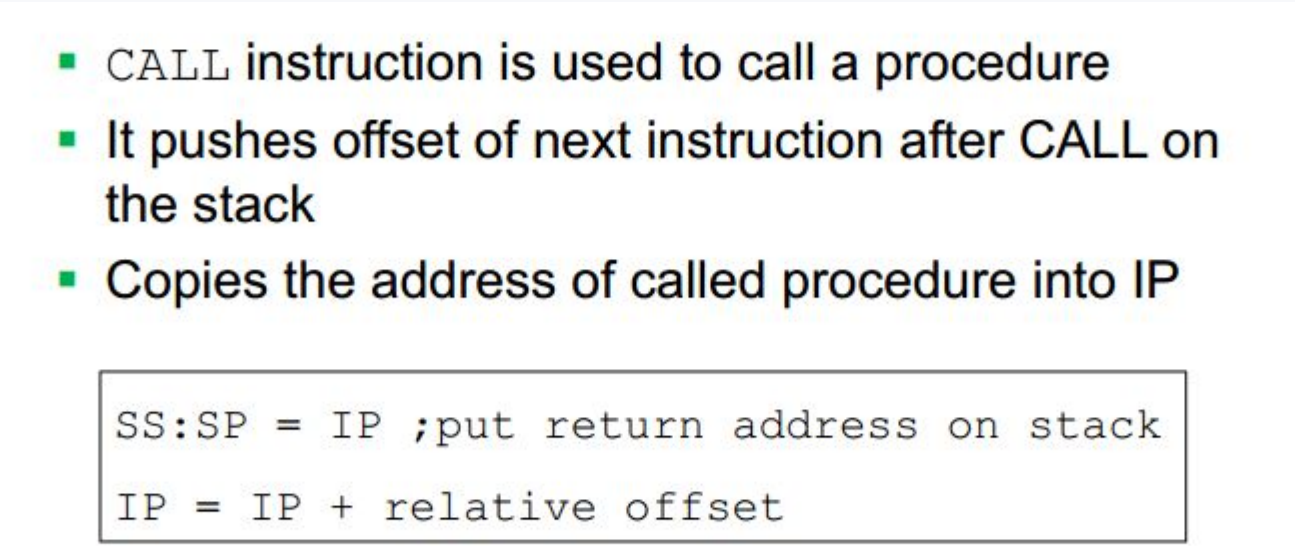
Creating a Procedure



- A procedure is declared using the `PROC` and `ENDP` directives
- Must be assigned a name which should be a valid identifier
- Procedures other than startup procedure should be ended with `RET` instruction
- Following is an assembly language procedure with name `proc_name`

```
proc_name PROC
    instruction1
    instruction2
    ret
proc_name ENDP
```

CALL Instruction

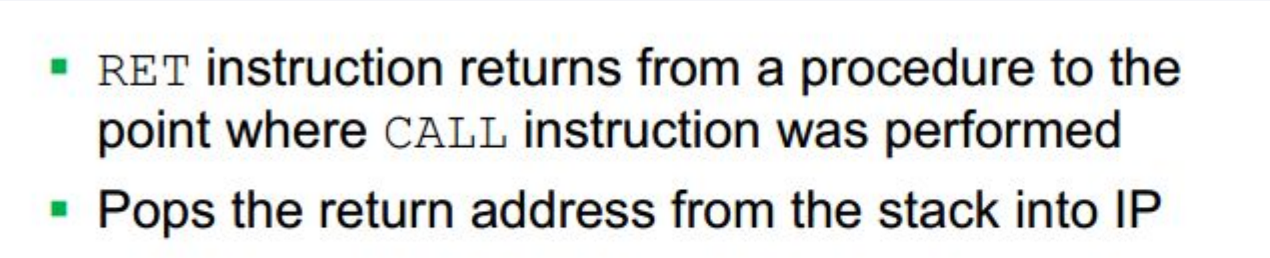


- `CALL` instruction is used to call a procedure
- It pushes offset of next instruction after CALL on the stack
- Copies the address of called procedure into IP

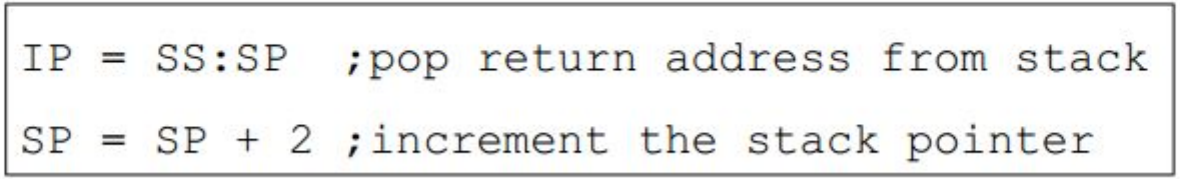
```
SS:SP = IP ;put return address on stack

IP = IP + relative offset
```

RET Instruction

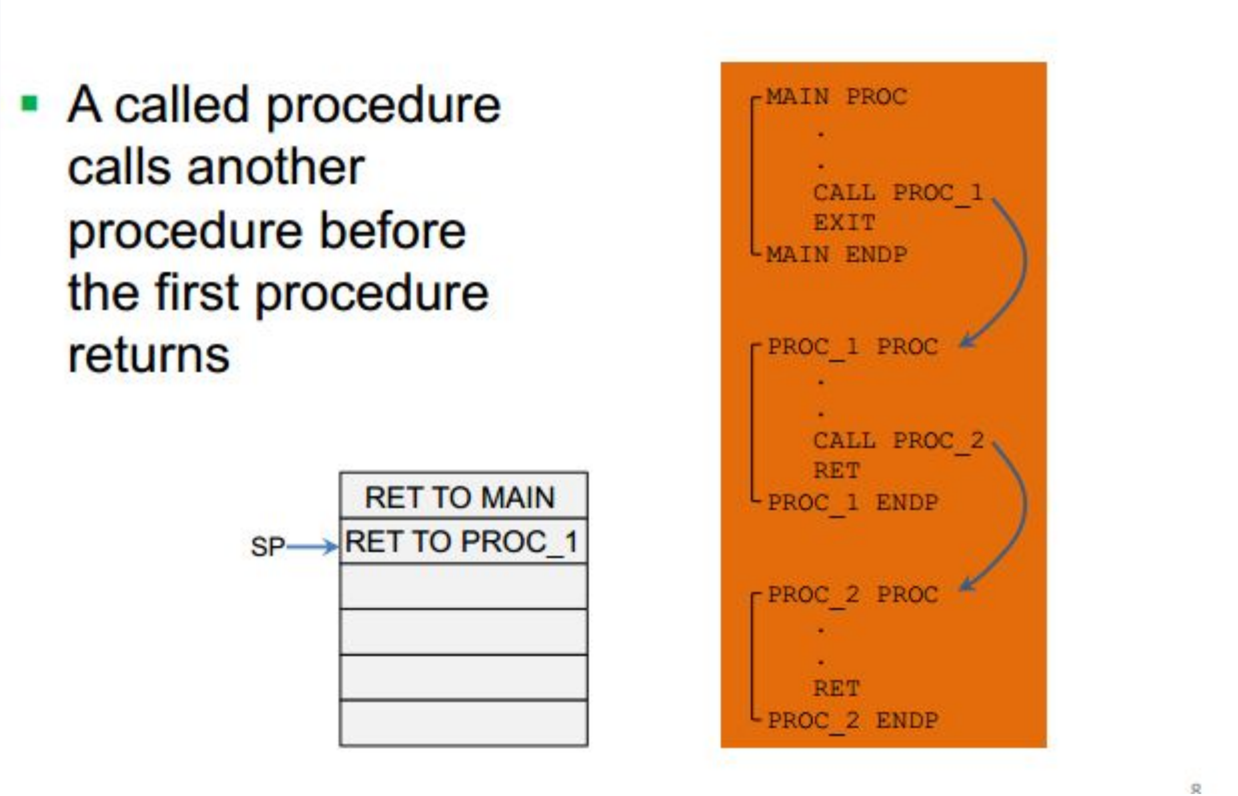


- `RET` instruction returns from a procedure to the point where `CALL` instruction was performed
- Pops the return address from the stack into IP



```
IP = SS:SP  ;pop return address from stack
SP = SP + 2 ;increment the stack pointer
```

Nested Procedure Call



- A called procedure calls another procedure before the first procedure returns

RET TO MAIN
RET TO PROC_1

```
MAIN PROC
    .
    .
    CALL PROC_1
    EXIT
MAIN ENDP

PROC_1 PROC
    .
    .
    CALL PROC_2
    RET
PROC_1 ENDP

PROC_2 PROC
    .
    .
    RET
PROC_2 ENDP
```

Parameter Passing in Procedures

□ Parameter passing is different and complicated in assembly than in HLL

□ In assembly language

□ First place all required parameters in a mutually accessible storage area

□ Then call the procedure

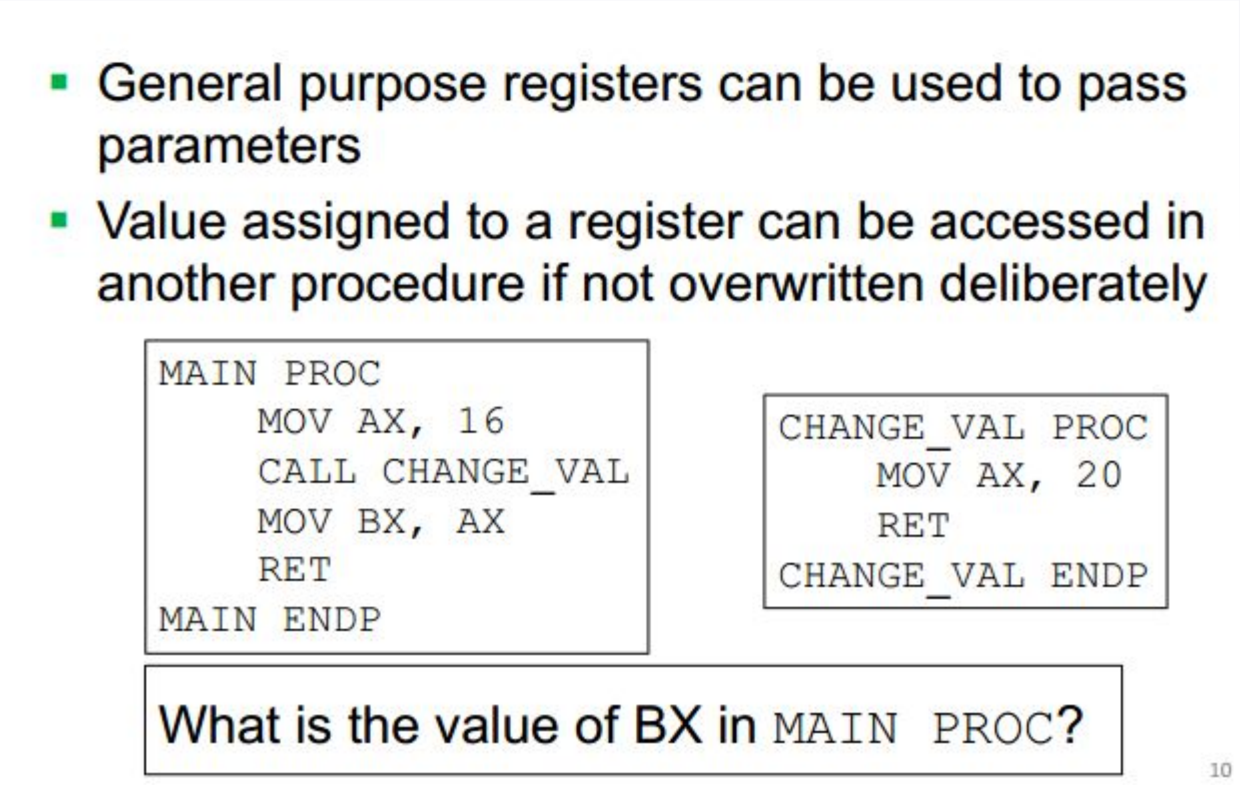
□ Types of storage area are

□ Registers (general purpose registers are used)

□ Memory (Stack is used)

□ Two common methods for parameter passing
□ Register Method
□ Stack Method

Parameter passing through registers



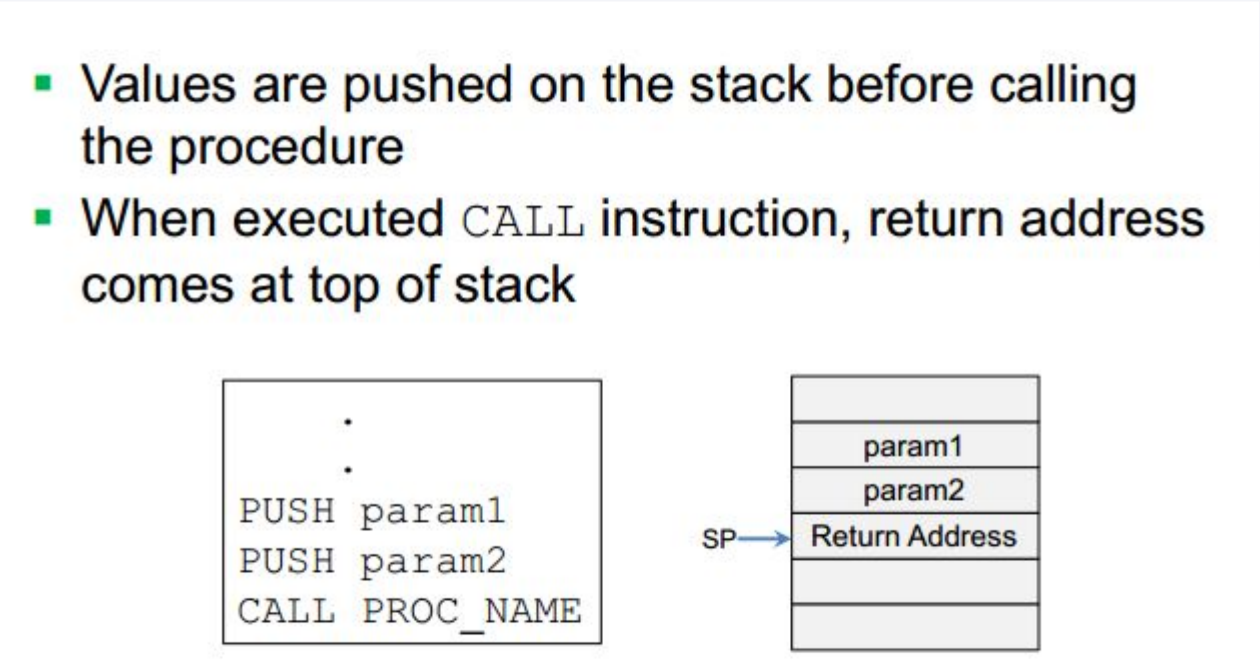
- General purpose registers can be used to pass parameters
- Value assigned to a register can be accessed in another procedure if not overwritten deliberately

```
MAIN PROC
    MOV AX, 16
    CALL CHANGE_VAL
    MOV BX, AX
    RET
MAIN ENDP
```

```
CHANGE_VAL PROC
    MOV AX, 20
    RET
CHANGE_VAL ENDP
```

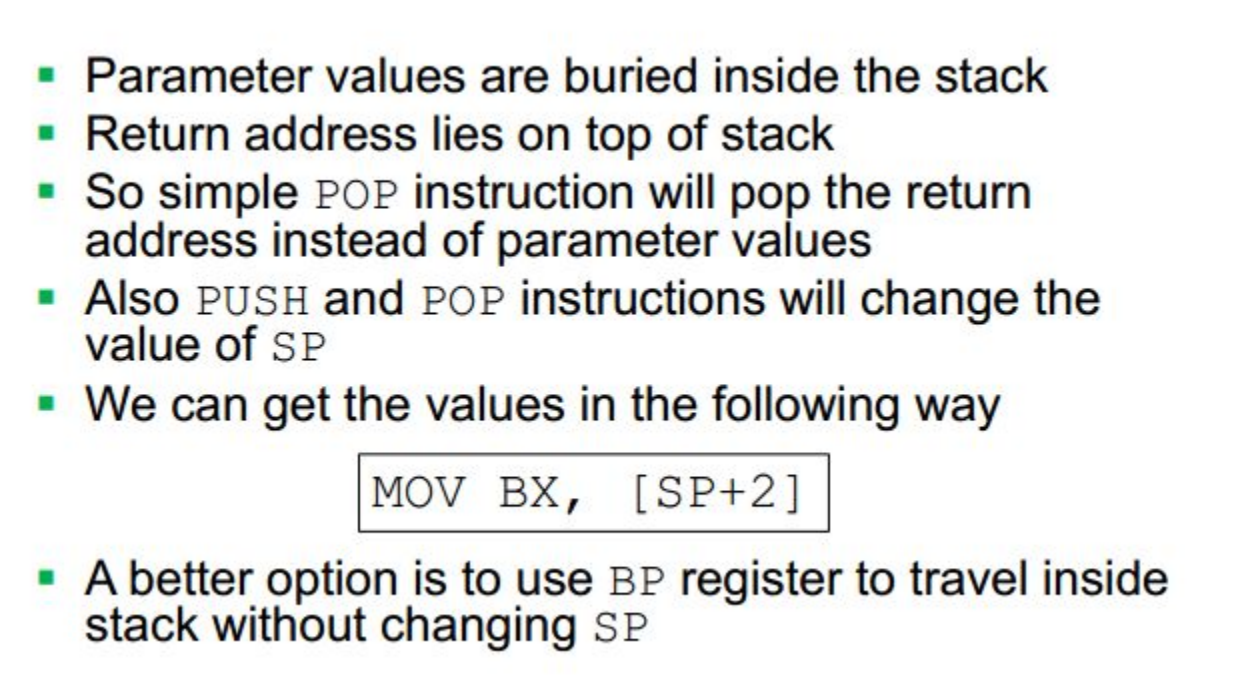
What is the value of BX in `MAIN PROC`?

Parameters passing using STACK



- Values are pushed on the stack before calling the procedure
- When executed `CALL` instruction, return address comes at top of stack

```
     .
     .
PUSH param1
PUSH param2
CALL PROC_NAME
```

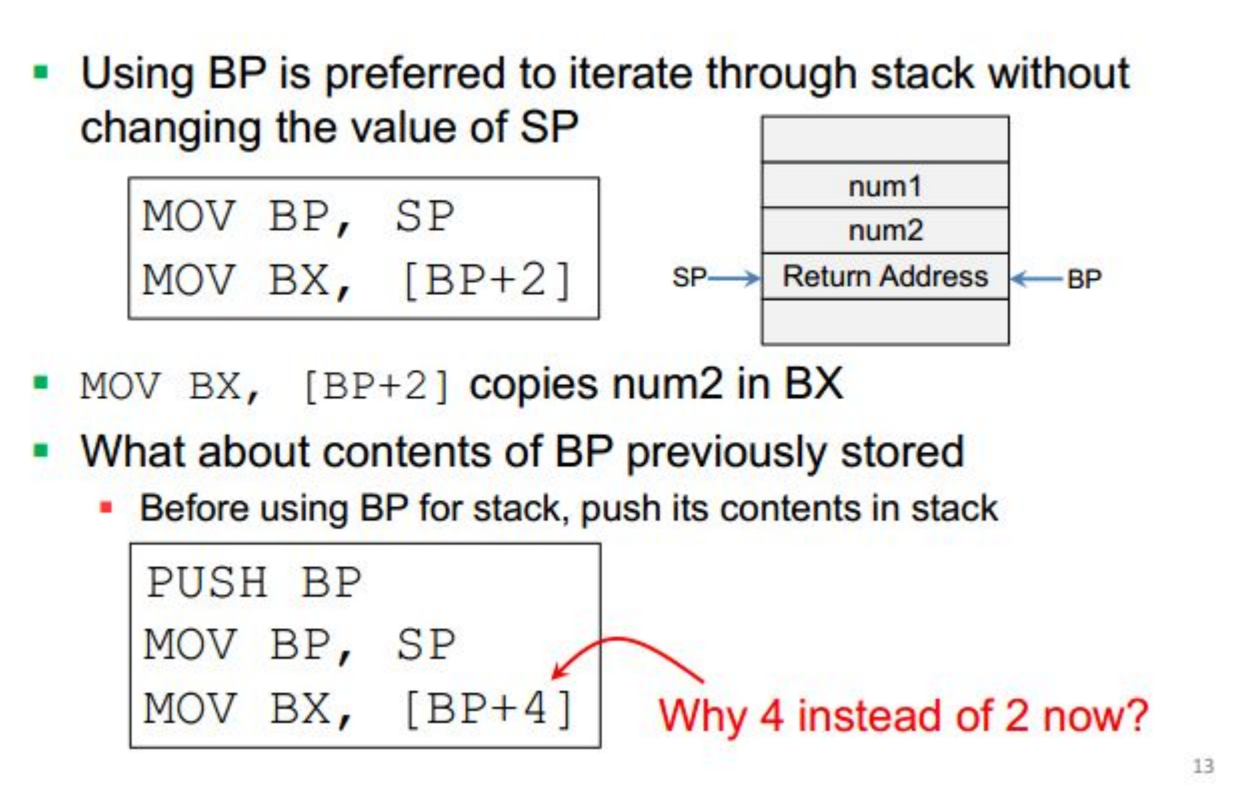


- Parameter values are buried inside the stack
- Return address lies on top of stack
- So simple `POP` instruction will pop the return address instead of parameter values
- Also `PUSH` and `POP` instructions will change the value of `SP`
- We can get the values in the following way

```
MOV BX, [SP+2]
```

- A better option is to use `BP` register to travel inside stack without changing `SP`

Using BP to travel inside STACK



- Using BP is preferred to iterate through stack without changing the value of SP

```
MOV BP, SP
MOV BX, [BP+2]
```

num1

num2

SP → Return Address ← BP

- `MOV BX, [BP+2]` copies num2 in BX
- What about contents of BP previously stored
  - Before using BP for stack, push its contents in stack

```
PUSH BP
MOV BP, SP
MOV BX, [BP+4]
```

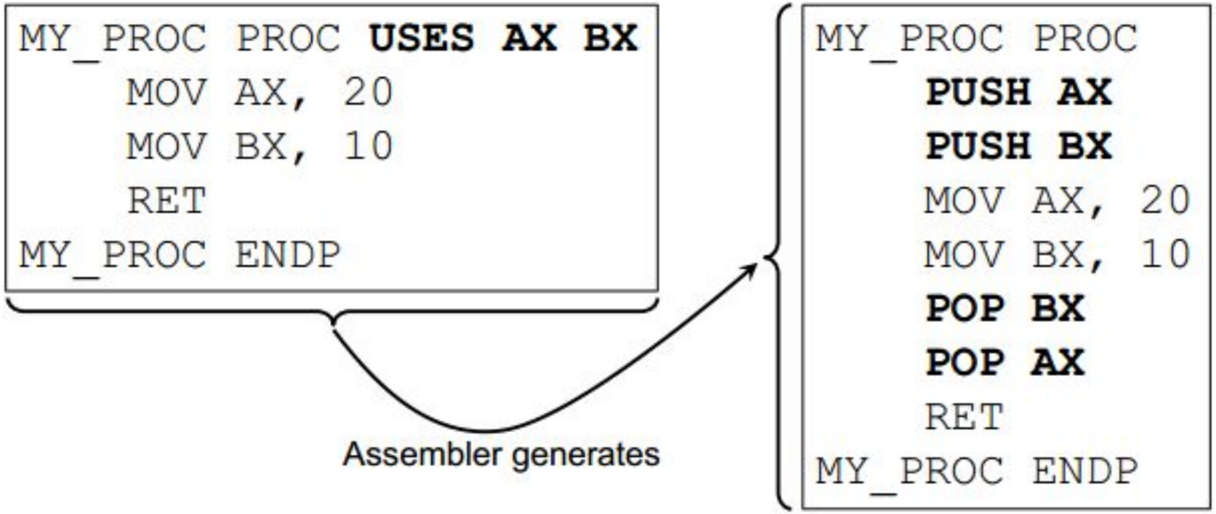
Why 4 instead of 2 now?

USES Operator

□ All registers modified in a procedure should be saved on stack and restored before return

□ USES operator facilitates the saving and restoring of registers in an easy way

□ USES operator is used right after PROC directive and lists names of all registers modified inside procedure



```
MY_PROC PROC USES AX BX
    MOV AX, 20
    MOV BX, 10
    RET
MY_PROC ENDP
```

Assembler generates

```
MY_PROC PROC
    PUSH AX
    PUSH BX
    MOV AX, 20
    MOV BX, 10
    POP BX
    POP AX
    RET
MY_PROC ENDP
```

THANKS!

Any questions?

You can find me at:

- A.qadeer@nu.edu.pk
- Office #213, Visiting Hours Only