Defining and Using Procedures

Muhammad Afzaal
m.afzaal@nu.edu.pk

Book Chapter

- “Assembly Language for x86 Processors”
- Author “Kip R. Irvine”
- 6th Edition
- Chapter 5
  - Section 5.5

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
```

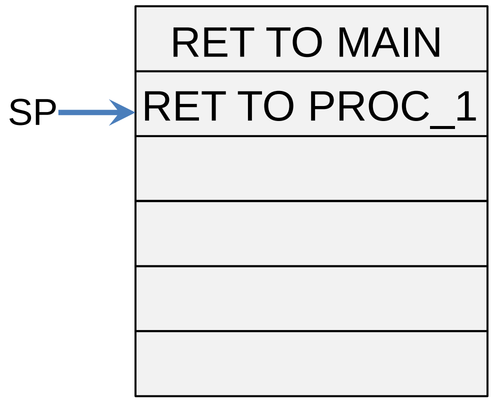
RET Instruction

- `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



```
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 using 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
```

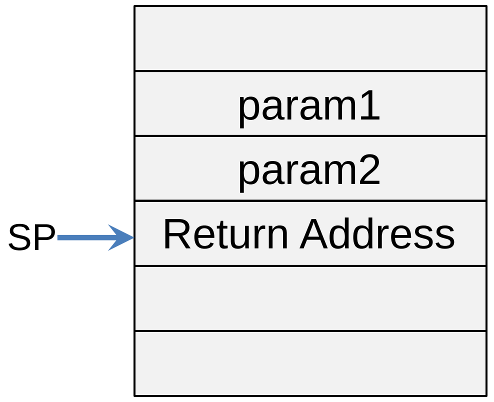
```
CHANGE_VAL PROC
    MOV AX, 20
    RET
CHANGE_VAL ENDP
```

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

Parameter Passing using Stack (1/2)

- 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 Passing using Stack (2/2)

- 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

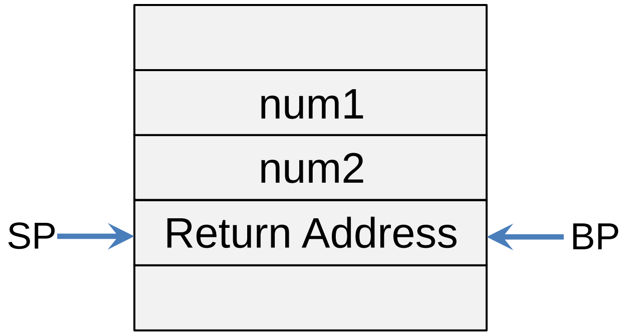
```
MOV BX, [SP+2]
```

- 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]
```



- `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 (1/2)

- 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

USES Operator (2/2)

```
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
```