Assembler Directives for Procedures

- Assembler provides two directives to define procedures: PROC and ENDP
- To define a NEAR procedure, use

```
proc-name    PROC    NEAR
```

  - In a NEAR procedure, both calling and called procedures are in the same code segment
- A FAR procedure can be defined by

```
proc-name    PROC    FAR
```

  - Called and calling procedures are in two different segments in a FAR procedure

Assembler Directives for Procedures (cont'd)

- If FAR or NEAR is not specified, NEAR is assumed (i.e., NEAR is the default)
- We focus on NEAR procedures
- A typical NAER procedure definition

```
proc-name    PROC
    . . . . .
<procedure body>
    . . . . .
proc-name    ENDP
```

proc-name should match in PROC and ENDP

Pentium Instructions for Procedures

- Pentium provides two instructions: **`call`** and **`ret`**
- **`call`** instruction is used to invoke a procedure
- The format is

  `call proc-name`

  `proc-name` is the procedure name
- Actions taken during a near procedure call:

```
SP := SP - 2            ; push return address
(SS:SP) := IP           ;       onto the stack
IP := IP + relative displacement    ; update IP
                        ; to point to the procedure
```


Pentium Instructions for Procedures (cont'd)

- **`ret`** instruction is used to transfer control back to the calling procedure
- How will the processor know where to return?
  - Uses the return address pushed onto the stack as part of executing the **`call`** instruction
  - Important that TOS points to this return address when **`ret`** instruction is executed
- Actions taken during the execution of **`ret`** are:

```
IP := (SS:SP)           ; pop return address
SP := SP + 2            ;       from the stack
```

Pentium Instructions for Procedures (cont'd)

- We can specify an optional integer in the **`ret`** instruction
  - The format is

```
ret    optional-integer
```

  - Example:

```
ret 6
```

- Actions taken on **`ret`** with optional-integer are:

> IP := (SS:SP)
> SP := SP + 2 + optional-integer

How Is Program Control Transferred?

```
Offset(hex)  machine code(hex)
                          main    PROC
                          . . . . . .
cs:000A      E8000C       call    sum
cs:000D      8BD8         mov     BX,AX
                          . . . . . .
                          main    ENDP
-------------------------------------------
                          sum     PROC
cs:0019      55           push    BP
                          . . . . . .
                          sum     ENDP
-------------------------------------------
                          avg     PROC
                          . . . . . .
cs:0028      E8FFEE       call    sum
cs:002B      8BD0         mov     DX,AX
                          . . . . . .
                          avg     ENDP
```

Parameter Passing

- Parameter passing is different and complicated than in a high-level language
- In assembly language
  - » You should first place all required parameters in a mutually accessible storage area
  - » Then call the procedure
- Type of storage area used
  - » Registers (general-purpose registers are used)
  - » Memory (stack is used)
- Two common methods of parameter passing:
  - » Register method
  - » Stack method

Parameter Passing: Register Method

- Calling procedure places the necessary parameters in the general-purpose registers before invoking the procedure through the **`call`** instruction
- Examples:
  - * **`PROCEX1.ASM`**
    - » call-by-value using the register method
    - » a simple sum procedure
  - * **`PROCEX2.ASM`**
    - » call-by-reference using the register method
    - » string length procedure

Pros and Cons of the Register Method

- Advantages
  - Convenient and easier
  - Faster
- Disadvantages
  - Only a few parameters can be passed using the register method
    - Only a small number of registers are available
  - Often these registers are not free
    - freeing them by pushing their values onto the stack negates the second advantage

Parameter Passing: Stack Method

- All parameter values are pushed onto the stack before calling the procedure
- Example:

```
push    number1
push    number2
call    sum
```