Important AL Techniques

CO 2103 Assembly Language

Topics

- Indirect Addressing
- Logic Control Structures
- STACK
- Subroutine / Procedure

Indirect Addressing

Indirect Addressing

- Indirect Addressing is a commonly used technique to handle memory operands
  - Indirect Operands
  - Indexed Operands
  - Pointers
  - Examples:
    - Array Sum (1)
    - Array Sum (2)
    - Displaying a String

Indirect Operands - 1

- An indirect operand holds the address of a variable, usually an array or string. It can be *dereferenced* (just like a pointer)

```
.data
num db 21,22,23,24,25

.code
main:  …

    mov si,offset num    ;SI contains address of num
    mov al,[si]          ;dereference SI (AL=21)

    inc si
    mov al,[si]          ;dereference SI (AL=22)

    …
```

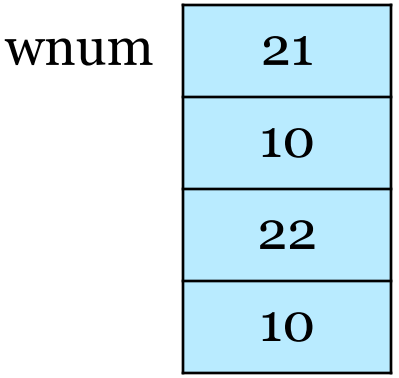
OFFSET returns the distance in bytes, of a *label* from the beginning of its enclosing segment

Indirect Operands - 2

- When dealing with memory operands, use PTR to specify the size to avoid ambiguity

```
.data
wnum dw 1021h,1022h
.code
main:  …

    mov si,offset wnum
    mov al,byte ptr [si];AL=21h
    mov ax,word ptr [si];AX=1021h
    inc word ptr [si]    ;[wnum]=1022h
    inc si
    inc byte ptr [si]    ;[wnum]=1122h
    mov al,byte ptr [si];AL=11h
    …
```



PTR Overrides the default type of a label (variable). Provides the flexibility to access part of a variable. Note little endian ordering of data in memory.

Indexed Operands
(Relative Indirect)

- An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

[label + reg] label[reg]

```
.data
warray dw 1000h,2000h,3000h
.code
…
mov si,0
mov ax,[warray + si]    ; AX = 1000h
mov ax,warray[si]       ; alternative format
add si,2
add ax,[warray + si]
add si,2
add ax,[warray + si]    ; AX = sum of the array
…
```

Pointers

- A variable can be declared, as a pointer, to contain the offset of another variable

```
.data
warray dw 1000h,2000h,3000h
wptr dd warray   ;wptr has offset of warray

.code
…
mov si,wptr   ;si has value of wptr (offset of warray)
mov ax,[si]   ;AX = 1000h
…
```

Example 1: Array Sum (1)

- Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
warray dw 1000h,2000h,3000h
.code
…
mov si,offset warray
mov ax,[si]
add si,2        ; or: add si,type warray
add ax,[si]
add si,2
add ax,[si]     ; AX = sum of the array
…
```

The TYPE operator returns the size, in bytes, of a single element of a data declaration.

Example 2: Array Sum (2)

```
.data
warray dw 0100h,0200h,0300h,0400h
COUNT = LENGTHOF warray

.code
mov ax,0                    ;zero accumulator
mov di,offset warray        ;address of array
mov cx,COUNT                ;loop counter
L1:
add ax,[di]                 ;add an element
add di,type warray          ;point to next element
    loop L1                     ;dec CX, repeat until CX=0
```

Alternative:
COUNT = ($-*warray*) / (type *warray*)

The LENGTHOF operator counts the number of elements in a single data declaration.

Example 3: Displaying a String

```
.data
string dw "This is a string."
COUNT = LENGTHOF string

.code
mov si,offset string    ;address of string
mov cx,COUNT            ;loop counter
L1:
    mov ah,2                ;DOS function: display char
    mov dl,[si]             ;get character from array
    int 21h                 ;display it
    inc si                  ;point to next char
    loop L1                 ;dec CX, repeat until CX=0
```

Alternative:
Use function INT 21h with AH=9 to display '$' terminated string.

Logic Control Structures

Program Control @ Low Level

- Much of the intelligence in any program relies on its decision making capabilities or its control/logic structures
- In high-level languages, common control structures include:
  - IF Statement
  - Compound IF Using OR
  - Compound IF Using AND
  - Loops
    - DO-WHILE Loop
    - REPEAT-UNTIL Loop
  - CASE Structure

Simple IF Statement - 1

Pseudocode:

```
if (AX = op2) then
   <do something>
end if
```

Implementation:

```
    cmp   ax,op2
    je    True
    jmp   EndIf
True:
    <do something>
EndIf:
    …
```

Simple IF Statement - 2

- Reversing the condition shorten the implementation by one instruction

Pseudocode:

```
if (AX = op2) then
   <do something>
end if
```

Implementation:

```
    cmp  ax,op2
    jne  EndIf
    <do something>
EndIf:
    …
```

ELSE statements can be implemented under the Endif

Compound IF Statement Using OR

Pseudocode:

```
if (AL > op1)
   or (AL >= op2)
   or  (AL = op3)
   <do something>
end if
```

Implementation:

```
    cmp   al,op1
    jg    L1
    cmp   al,op2
    jge   L1
    cmp   al,op3
    je    L1
    jmp   L2
L1:
    <do something>
L2:
```

Compound IF Statement Using AND

Pseudocode:

```
if (AL > op1)
   and (AL >= op2)
   and (AL = op3)
   <do something>
end if
```

Implementation:

```
    cmp  al,op1
    jng  L1
    cmp  al,op2
    jnge L1
    cmp  al,op3
    jne  L1
    <do something>
L1:
```

Combining AND and OR

Pseudocode:

```
if (AL > op1)
   and (AL >= op2)
   or (AL = BL)
   <do something>
end if
```

Implementation:

```
        cmp  al,op1
        jng  L1
        cmp  al,op2
        jnge L1
        jmp  L2
L1:     cmp  al,bl
        jne  L3
L2:     <do something>
L3:
```

Other combination of IF-THEN-ELSE logic structures can be implemented through correct arrangement of CMP, conditional JMP instructions and "doing" statements

DO-WHILE Loop - 1

Pseudocode:

```
do while (op1 < op2)
   <do something>
enddo
```

Implementation:

```
While:
    cmp   op1,op2
    jnl   Enddo
    <do something>
    jmp While
Enddo:
```

DO-WHILE Loop – 2
(combined with IF-ELSE)

Pseudocode:

```
do while (op1 < op2)
   <statement1>
   if op2 = op3 then
      <statement2>
      <statement3>
   else
      <statement4>
   endif
enddo
```

Implementation:

```
While:
        cmp   op1,op2
        jnl   Enddo
        <statement1>
        cmp op2,op3
        jne L1
        <statement2>
        <statement3>
        jmp L2
L1:     <statement4>
L2:     jmp While
Enddo:
```

REPEAT-UNTIL structure

Pseudocode:

```
repeat
   <statement1>
   <statement2>
   <statement3>
until (op1 = op2)
   or (op1 > op3)
```

Implementation:

```
repeat:
   <statement1>
   <statement2>
   <statement3>
test1:
   cmp  op1,op2
   je   endif
test2:
   cmp  op1,op3
   jng   repeat
endif:
```

DO-WHILE tests the condition(s) before the statements
REPEAT-UNTIL tests the condition(s) after the statements

CASE Structure

- The CASE structure allows a multiway branch by comparing a single value to a list of values

```
Pseudocode:

case input of
   'A'   :Process_A;
   'B'   :Process_B;
   'C'   :Process_C;
   'D'   :Process_D;
end
```

CASE Structure LL Implementation

- The most direct implementation involves CMP and conditional jump instructions

```
mov   al,input     ; case al (input) of
cmp   al,'A'       ; 'A':
je    Process_A    ;   goto Process_A
cmp   al,'B'       ; 'B':
je    Process_B    ;   goto Process_B
cmp   al,'C'       ; 'C':
je    Process_C    ;   goto Process_C
cmp   al,'D'       ; 'D':
je    Process_D    ;   goto Process_D
```

- Alternatively, procedure calls can be used

Case Structure Using Procedure Calls

```
        mov al,input      ; case al of
        cmp al,'A'     ; 'A':
        jne L1
        call   Process_A ; call Process_A
        jmp L4            ; jump to L4
L1: cmp al,'B'     ; 'B':
        jne L2
        call   Process_B ; call Process_B
        jmp L4            ; jump to L4
L2: cmp al,'C'     ; 'C':
        jne L3
        call   Process_C ; call Process_C
        jmp L4            ; jump to L4
L3: cmp al,'D'     ; 'D':
        jne L4
        call   Process_D ; call Process_D
L4:                       ;  end
```

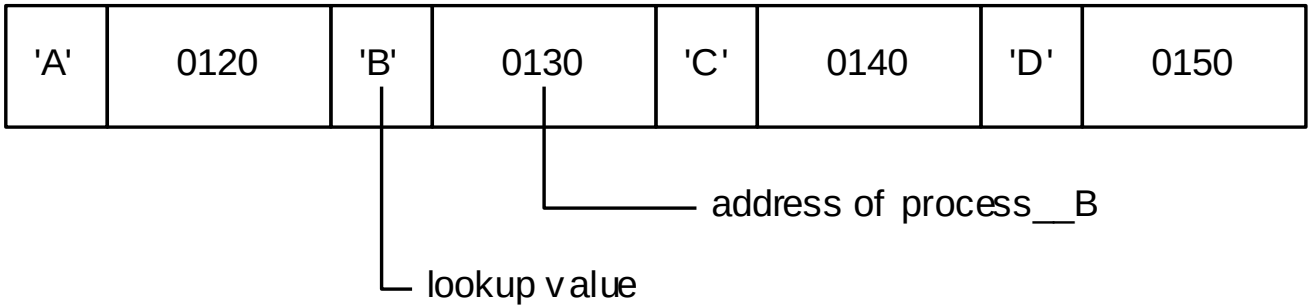
CASE Structure using Lookup Table

- A neat implement of CASE structure can be done using a lookup table
  - A table may contain lookup values (input values) and corresponding procedure offsets
  - Unlike previous implementations of case structure, this table can be modified at runtime.

```
CaseTable  db   'A'          ; lookup value
           dw   process_A    ; address of procedure
           db   'B'
           dw   process_B
           db   'C'
           dw   Process_C
           db   'D'
           dw   Process_D
```

Lookup Table in Memory

- Assume that four different procedures are located at offsets 0120, 0130, 0140, and 0150, respectively. This is how the table would look in memory:



- When AL matches a character in the table, it will call the corresponding procedure whose address is located in the table immediately following the matching character.

CASE Implementation Using Lookup Table

```
NumberOfEntries = 4
input db ?
.code
      mov   al,input              ; value to be found
      mov   bx,offset CaseTable   ; point BX to the table
      mov   cx,NumberOfEntries    ; loop counter
  L1: cmp   al,[bx]               ; match found?
      jne   L2                    ; no: continue
      call  word ptr [bx+1]       ; yes: call the procedure
      jmp   L3                    ; exit the search
  L2: add   bx,3                  ; point to next entry
      loop  L1                    ; repeat until CX = 0
  L3:
```

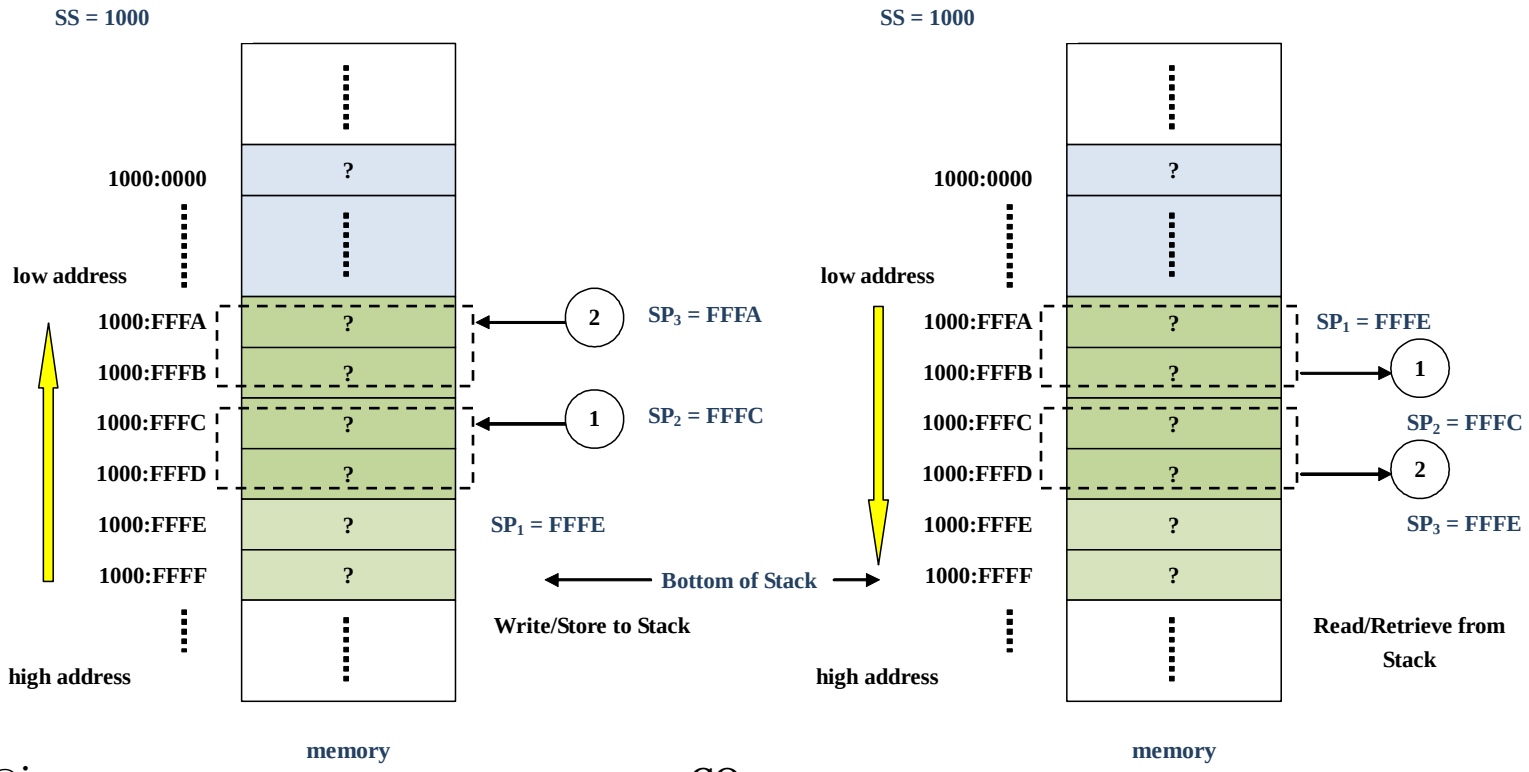
STACK

Stack – intro

- Last In First Out (LIFO) or First In Last Out data structure (automated)
- Temporary data storage in the memory (mandatory for CPU operation)
- Usage:
  - Save the return address when a CALL instruction is executed
  - Save and restore registers (PUSH and POP instructions)
  - Pass parameters or return results to/from subroutine
  - Create local variables inside a procedure
- A procedure's stack frame includes passed parameters/results, the return address, and local variables
- Stack operations are always 16-bit (2 bytes), i.e. accessed on word boundaries only

Stack - structure

- Stack is in the Stack Segment and SS register stores its start address
- SP (Stack Pointer) indicates current top of stack (with valid content)
- Usually grows from high memory to low memory (always 16-bit)



Stack related Instructions

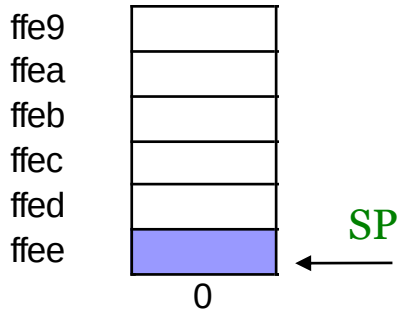
- PUSH and POP
  - PUSH instruction pushes (stores) the content of a register (16-bit), memory locations (2-byte) or a word (16-bit) on the top of the stack
  - POP instruction pops (retrieves) into the content of a register (16-bit) or memory locations (2-byte) from the top of the stack
- CALL and RET
  - CALL instruction direct the flow of execution to a subroutine by modifying the IP, however, the CPU automatically pushes the IP (pointer to next instruction) to the Stack before modifying it
  - RET instruction direct the flow of execution from subroutine back to the calling program by popping the return address from the Stack into IP
- Variants of above instructions: IRET, PUSHF, PUSHA*, POPA*, etc

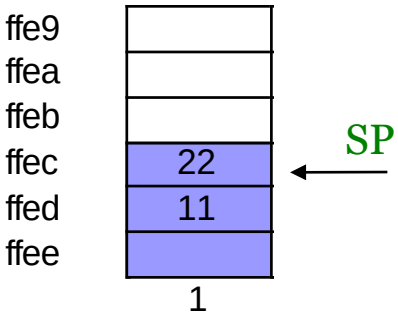
*on higher x86

Stack in Action: PUSH & POP instructions

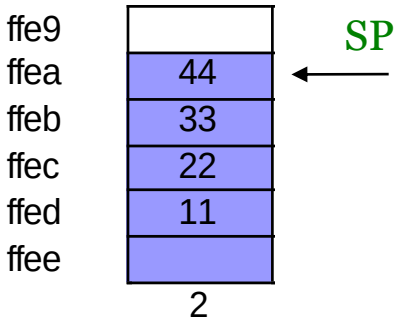
- Example: AX=1122, BX=3344, SP=FFEE initially

```
push ax     ;1. push AX to stack, SP=SP-2
push bx     ;2. push BX to stack, SP=SP-2
pop ax      ;3. pop AX from stack, SP=SP+2
pop bx      ;4. pop BX from stack, SP=SP+2
;note SP automatically updated and Little Endian system
```

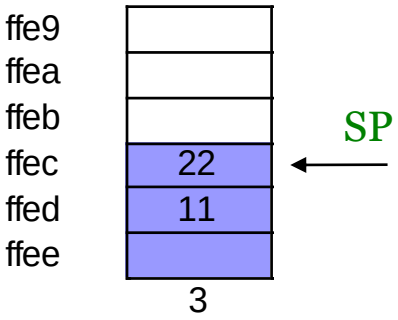




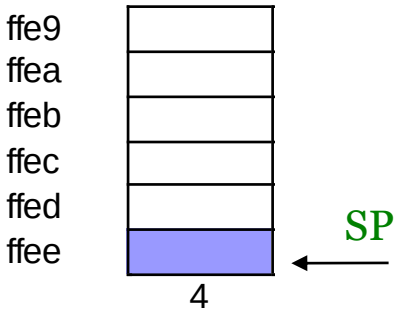
AX=1122, BX=3344



AX=1122, BX=3344



AX=3344, BX=3344

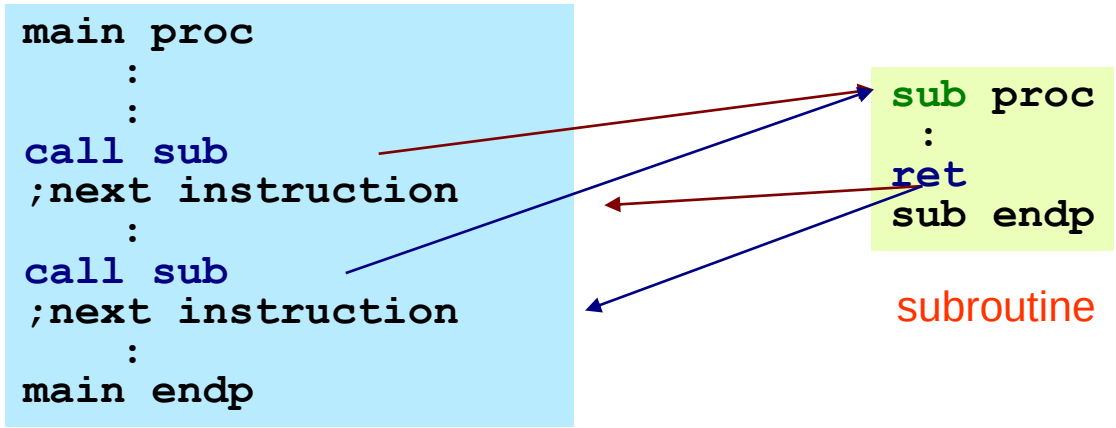


AX=3344, BX=1122

Subroutine / Procedure

Subroutine (Procedure)

- Subroutine is a subprogram, ending with a return instruction, that is usually written to perform specific task and can be called (executed) from different places of the main program as well as other subprogram; a function, method or procedure is a subroutine
- Subroutines are re-useable code modules: can be called again and again. A subroutine provides an easy way to encapsulate specific procedure which can then be used without worrying how it works



Procedure

- Basic mechanism for declaring a procedure:

```
procname    proc   {NEAR or FAR}
<statements>
procname    endp
```

- Consider the following two procedures:

```
NProc   proc    near
        mov     ax, 0
        ret
NProc   endp

FProc   proc    far
        mov     ax, 0FFFFH
        ret
FProc   endp
```

and:

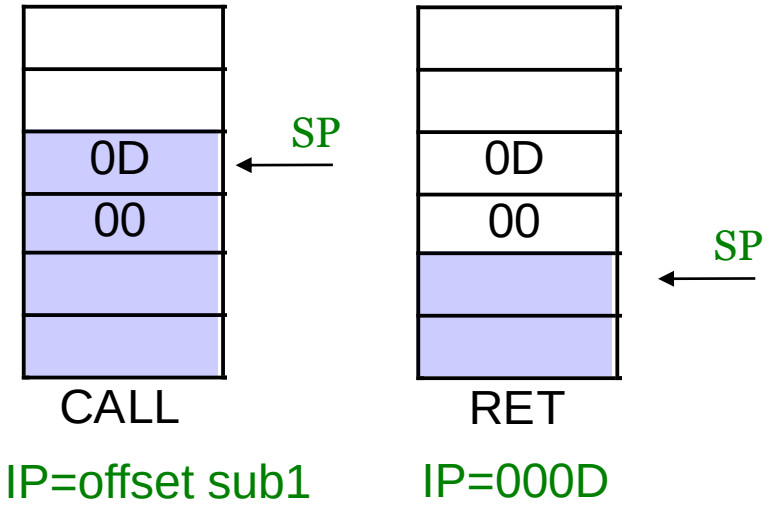
```
        call    NPROC
        call    FPROC
```

- Assembler automatically generates a three-byte (near) call for the first call instruction. It will generate a five-byte (far) call instruction for the second call.

Calling Near Procedure

- A near procedure is a procedure stored within the same code segment
  - IP pushed to the Stack upon CALL
  - offset of the procedure is used to modify IP upon CALL
  - return IP popped from the Stack upon RET

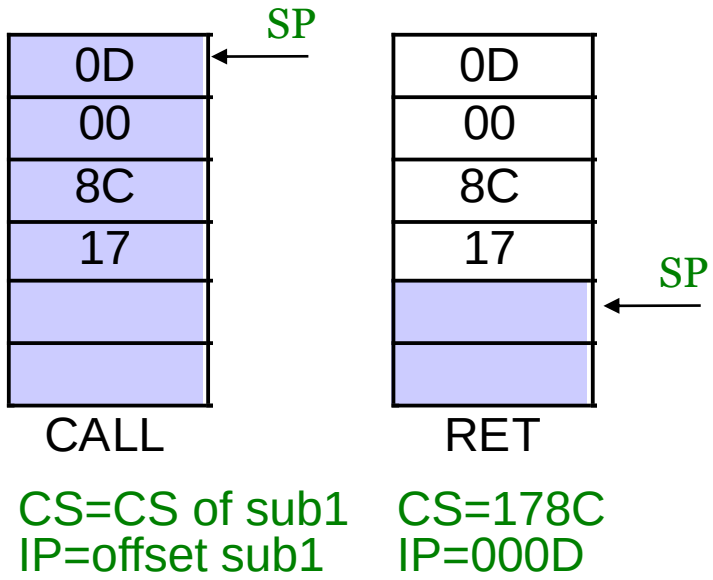
```
main proc
          :
000A   call sub1
000D   ;next instruction
          :
main endp
```



Calling Far Procedure

- A far procedure is a procedure stored in different code segment from that of the calling code
  - CS and IP pushed to the Stack upon CALL
  - CS and offset of the procedure is used to modify CS and IP upon CALL
  - return CS and IP popped from the Stack upon RET

```
main proc
                :
178C:000A call far ptr sub1
178C:000D ;next instruction
                :
main endp
```



Preserving Registers - 1

- It is common practice to save and restore any registers that a procedure plans to modify
  - PUSH affected registers at the beginning of the procedure
  - POP the registers, in reverse order, at the end of the procedure

```
writeint proc
push cx    ;save registers that will change
push bx
push si
    :
    :
pop  si    ;restore the same registers
pop  bx    ;(in reverse order)
pop  cx
ret        ;return to calling program
writeint endp
```

Preserving Registers - 2

- What would happen to the following program if writeint did not preserve CX, BX, and SI?

```
;This program print a list of integers
main proc
    :
mov  cx,LIST_COUNT
mov  bx,DECIMAL_RADIX
mov  si,offset aList
    :
L1: mov  ax,[si]
call writeint     ;print the integer
add  si,2         ;point to next
loop L1           ;loop CX times
    :
main endp
```

Nested Procedure Calls - 1

```
main proc
          :
000A   call sub1 ;(1)
000D   ;next instruction
          :
main endp
```

```
sub1 proc
          :
       call sub2 ;(2)
005B   ;next instruction
          :
       ret
sub1 endp
```

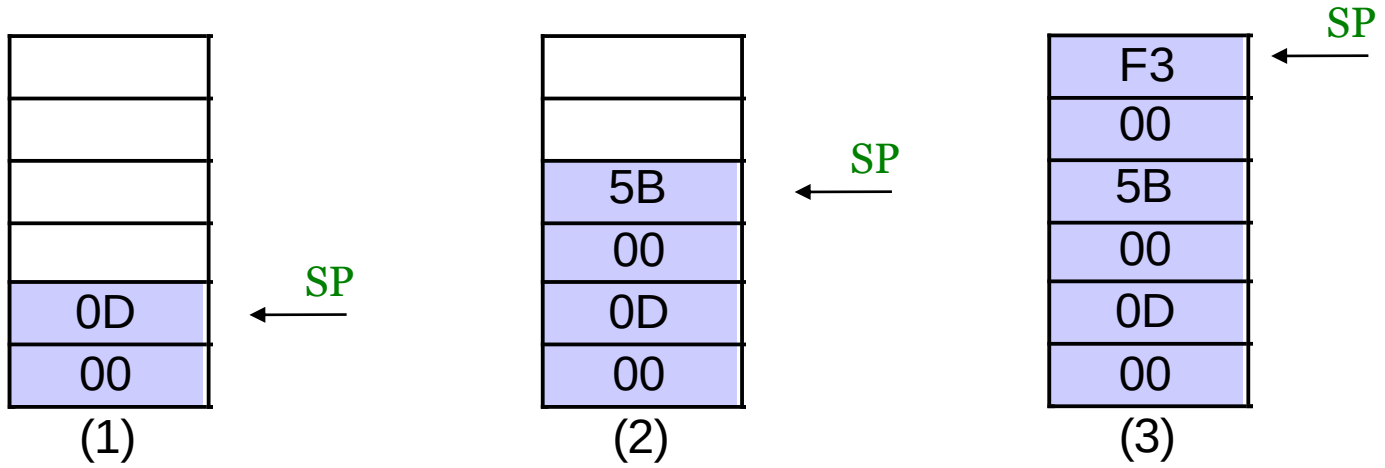
```
sub2 proc
          :
       call sub3 ;(3)
00F3   ;next instruction
          :
       ret
sub2 endp
```

```
sub3 proc
          :
       ret
sub3 endp
```

Next slide: status of Stack at execution point (1), (2) and (3)

Nested Procedure Calls - 2

- Stack in action (refer previous slide):
  (assuming no other Stack operations)



- Ensure procedures are properly organized, e.g. avoid overlapping procedures

Recursion

- Recursion – a procedure calling itself
- Happens under the following circumstances:
  - a procedure directly calls itself
  - Procedure A calls one or more other procedures, and somewhere in the execution of these, one of them calls Procedure A (indirect recursion)
- There must be a way to stop the recursion, or it will run out of control
  - conditional jump is usually used to accomplish this

Recursion Example: Sum of Integers - 1

```
Main proc
mov  cx,5          ;counter (number of integers)
mov  ax,0          ;holds the sum, start with 0
call Sum           ;find sum of 5+4+3+2+1
L1: mov  ax,4C00h     ;return to DOS
int  21h
Main endp

Sum proc
;Sum up cx+(cx-1)+...+1
or   cx,cx         ;check counter value
jz   L2            ;quit if zero
add  ax,cx         ;otherwise, add to sum
dec  cx            ;decrement counter
call Sum           ;recursive call
L2: ret               ;return to calling program
Sum endp
```

Recursion Example: Sum of Integers - 2

- Stack and registers' content

Pushed on Stack	CX	AX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Passing Parameters

- In a "high-level" language, procedures can be declared which
  - take no parameters and return no result
  - take one or more parameters and return no result
  - take no parameters but still return a result
  - take one or more parameters and return a result
- Similarly with AL programs
- Three ways to pass parameter(s) (and return result(s)):
  - through registers
  - through memory
  - through the stack
- Example programs (following slides) to print horizontal line with n (parameter) '*'

Parameters Passing in HLL

```
In main program:
...
z = sum(1,2);     `return z=3
...

The procedure:

int sum(int x,y);
    int w = x+y;
ret w;
```

Parameter thru Register - 1

- Easy to implement and is fast
  - move the parameters into registers before calling the procedure

```
prnstr proc near ; to preserve registers
    mov cl,0   ; count = 0
prn:   inc cl     ; count = count + 1
    cmp cl,al ; if count > n [parameter in AL]
    jg done    ; then done
    mov ah,02h; else, write character to screen
    mov dl,'*'; character to write
    int 21h    ; print('*')
    jmp prn    ; repeat
done:  ret     ; return to main program
prnstr endp
```

Parameter thru Register - 2

- Calling the procedure in previous slide:

```
main proc
    mov ax,@data  ;point DS to data segment
    mov ds,ax
    mov al,10     ;passing parameter in AL
    call prnstr
    mov ax,4c00h  ; function: exit to dos
    int 21h
main endp
```

Parameter thru Memory - 1

- Easy to implement (registers released for other use) but is slow
  - move the parameters into memory before calling the

```
prnstr proc near     ; to preserve registers
    mov cl,0         ; count = 0
prn:   inc cl          ; count = count + 1
    cmp cl,[n]       ; if count > n [parameter in n]
    jg done          ; then done
    mov ah,02h       ; else, write character to screen
    mov dl,'*'       ; character to write
    int 21h          ; print('*')
    jmp prn          ; repeat
done:  ret           ; return to main program
prnstr endp
```

Parameter thru Memory - 2

- Calling the procedure in previous slide:

```
main proc
    mov ax,@data  ;point DS to data segment
    mov ds,ax
    mov al,10
    mov [n],al;passing parameter in memory at n
    call prnstr
    mov ax,4c00h  ;function: exit to dos
    int 21h
main endp
```

- Remember to declare the memory location

```
.data
n   db ?
```

Parameter thru Stack - 1

- Most powerful and flexible, but complicated to implement

```
prnstr proc near     ; to preserve registers
    pop ax          ; retrieve parameter from stack
    mov cx,0        ; count = 0 [stack ops are 16-bit]
prn:   inc cx         ; count = count + 1
    cmp cx,ax       ; if count > n [parameter in AX]
    jg done         ; then done
    mov ah,02h      ; else, write character to screen
    mov dl,'*'      ; character to write
    int 21h         ; print('*')
    jmp prn         ; repeat
done:  ret          ; return to main program
prnstr endp
```

Will this work?

Parameter thru Stack - 2

- Calling the procedure in previous slide:

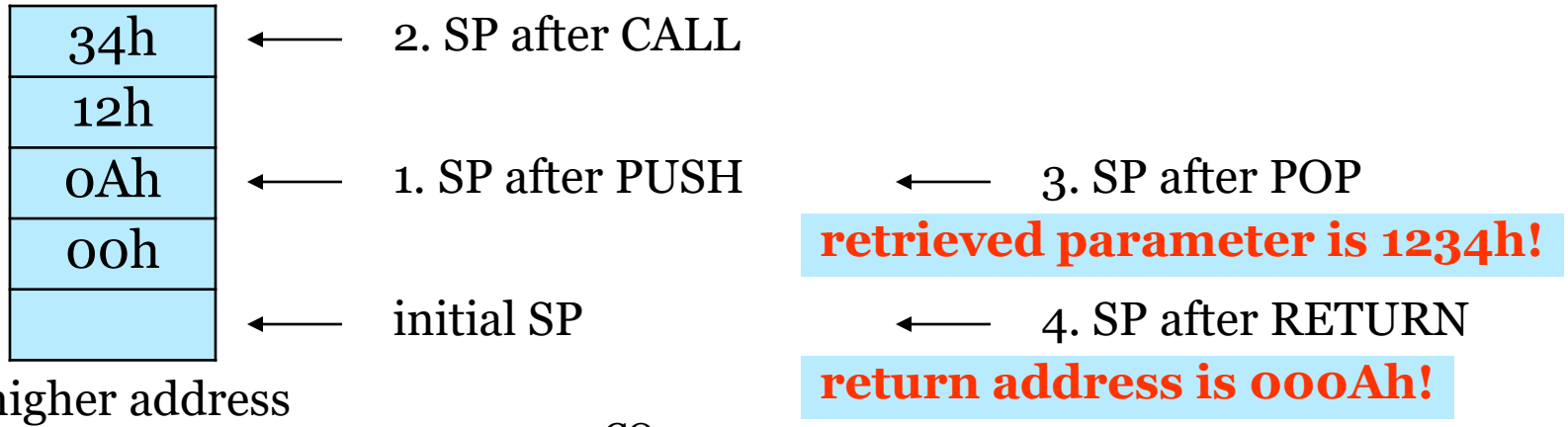
```
main proc
    mov ax,@data  ;point DS to data segment
    mov ds,ax
    push 10       ;passing parameter in stack
    call prnstr
    mov ax,4c00h  ;function: exit to dos
    int 21h
main endp
```

Will this work?

- Let's investigate the stack …

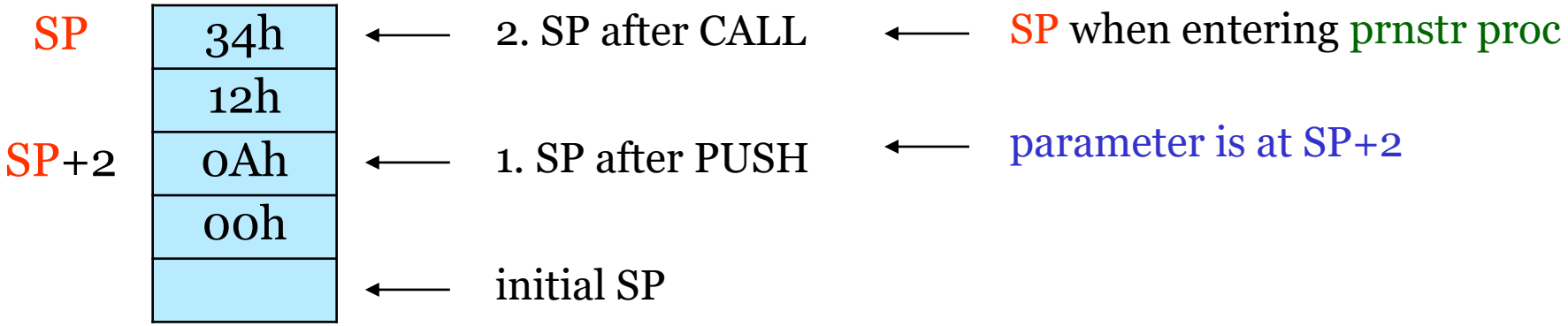
Parameter thru Stack - 3

- Sequence of Stack operations (16-bit):
  - 1. push in main proc
  - 2. call in main proc
  - 3. pop in prnstr proc
  - 4. return in prnstr proc
- Assume the instruction after call prnstr is at CS:1234h



Parameter thru Stack - 4

- Observed problems (from previous slide):
  - return address on top of stack (SP should not be moved)
  - parameter "below" the return address in the stack, but we need it before return (ret)
- Solution:
  - access the parameter(s) using another pointer (without affecting SP), i.e. not using POP



Parameter thru Stack - 5

- Modified procedure:

```
prnstr proc near     ; to preserve registers
    mov cx,0       ; count = 0
prn:   inc cx          ; count = count + 1
    cmp cx,[sp+2] ; if count > n [parameter in SP+2]
    jg done        ; then done
    mov ah,02h     ; else, write character to screen
    mov dl,'*'     ; character to write
    int 21h        ; print('*')
    jmp prn        ; repeat
done:  ret         ; return to main program
prnstr endp
```

- No change to main proc

Parameter thru Stack - 6

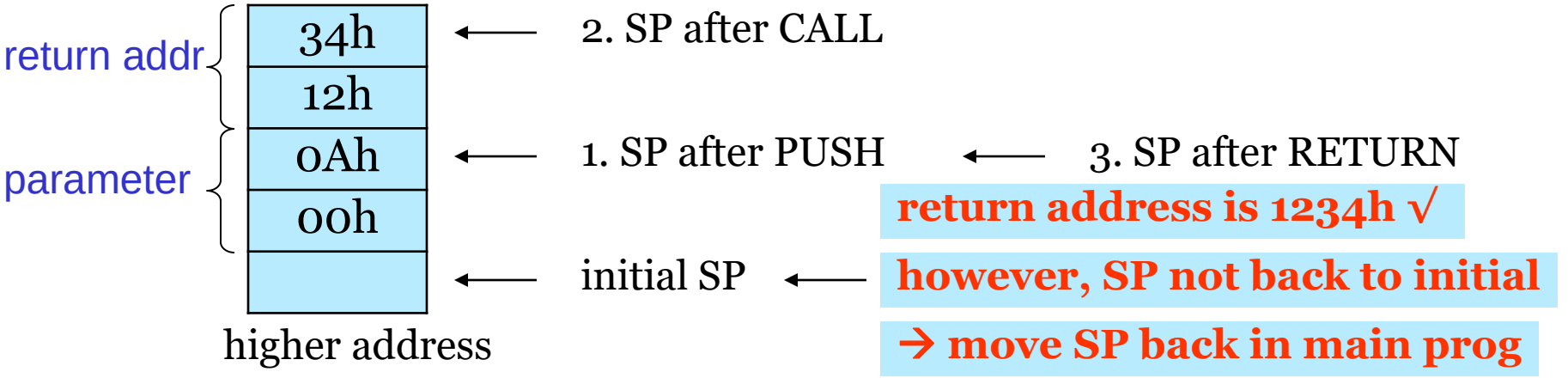
- May not be able to use SP directly

```
prnstr proc near     ; to preserve registers
    mov bp,sp      ; copy value of sp to bp
    mov cx,0       ; count = 0
prn:   inc cx          ; count = count + 1
    cmp cx,[bp+2] ; if count > n [parameter in BP+2]
    jg done        ; then done
    mov ah,02h     ; else, write character to screen
    mov dl,'*'     ; character to write
    int 21h        ; print('*')
    jmp prn        ; repeat
done:  ret         ; return to main program
prnstr endp
```

Will this work?

Parameter thru Stack – 7

- Sequence of Stack operations (16-bit):
  - 1. push in main proc
  - 2. call in main proc
  - 3. return in prnstr proc
- Assume the instruction after call prnstr is at CS:1234h



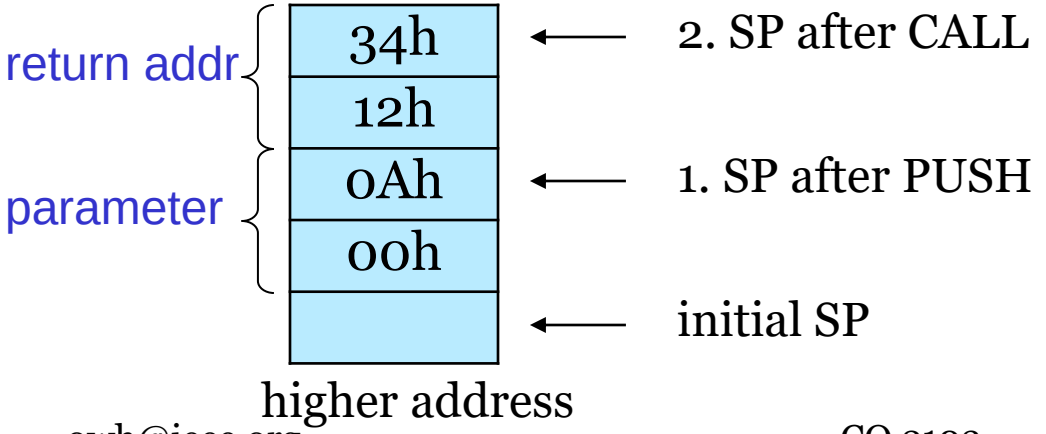
Parameter thru Stack - 8

- Calling the procedure in previous slide:

```
main proc
    mov ax,@data  ;point DS to data segment
    mov ds,ax
    push 10       ;passing parameter in stack
    call prnstr
    add sp,2      ;remove parameter from stack
    mov ax,4c00h  ;function: exit to dos
    int 21h
main endp
```

Parameter thru Stack – 9

- Sequence of Stack operations (16-bit):
  - 1. push in main proc
  - 2. call in main proc
  - 3. return in prnstr proc
  - 4. add in main proc
- Assume the instruction after call prnstr is at CS:1234h



3. SP after RETURN

return address is 1234h √

4. SP after ADD

SP back to initial position√

Parameter thru Stack – 10

- 80x86 ret instruction can take an optional pop-value, representing the number of bytes to remove from the stack on returning from the subroutine.
  - ret 2 ; return and remove 2 bytes from stack
  - using the above instruction, we do not need to adjust SP in main proc after the call, i.e. add sp,2 instruction will not be required
- Summary of steps:
  - in main proc: push parameter(s) to stack
  - in subroutine: provides fixed pointer (BP) to stack frame, use the pointer to access parameters, return and remove the parameters from stack

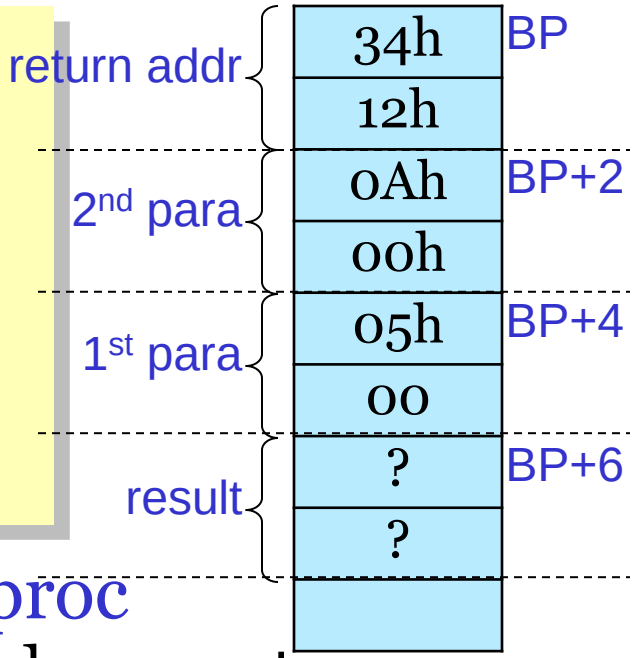
Returning Result - 1

- A procedure may return result through stack
- Add two steps in main proc: reserve space in stack for result, retrieve after call
  - in main proc: reserve space in stack, push parameter(s) to stack
  - in subroutine: provides fixed pointer (BP) to stack frame, use the pointer to access parameters, return and remove the parameters from stack
  - in main proc: retrieve result from the stack

Returning Result - 2

- Consider a procedure to add two values (passed in as parameters) and return the sum as result

```
sum proc near
  xor ax,ax       ;clear AX
  mov bp,sp       ;set up Stack Frame
  mov ax,[bp+4]   ;1st para from stack
  add ax,[bp+2]   ;2nd para from stack
  mov [bp+6],ax   ;result to stack
  ret 4    ;return & remove parameters
sum endp
```



- Next slide shows setup in main proc
  - how the result space is reserved and parameters passed into stack

Returning Result - 3

```
main proc
    ...
    sub sp,2   ;make space on stack for integer result
    push w[i]  ;push integer parameters
    push w[j]  ;              onto stack
    call sum   ;sum(i,j)
    pop w[k]   ;store integer result in k
    ...
main endp

.data
i   dw 5       ;1st parameter
j   dw 10      ;2nd parameter
k   dw ?       ;result
```

Getting Organized - 1

- In main program:

```
...
sub sp,2        ;result space, if required
push par1       ;pushes any
"     "         ;parameters
push parn       ;onto the stack
call sbrtn      ;call the procedure
pop result      ;retrieve result, if required
...
```

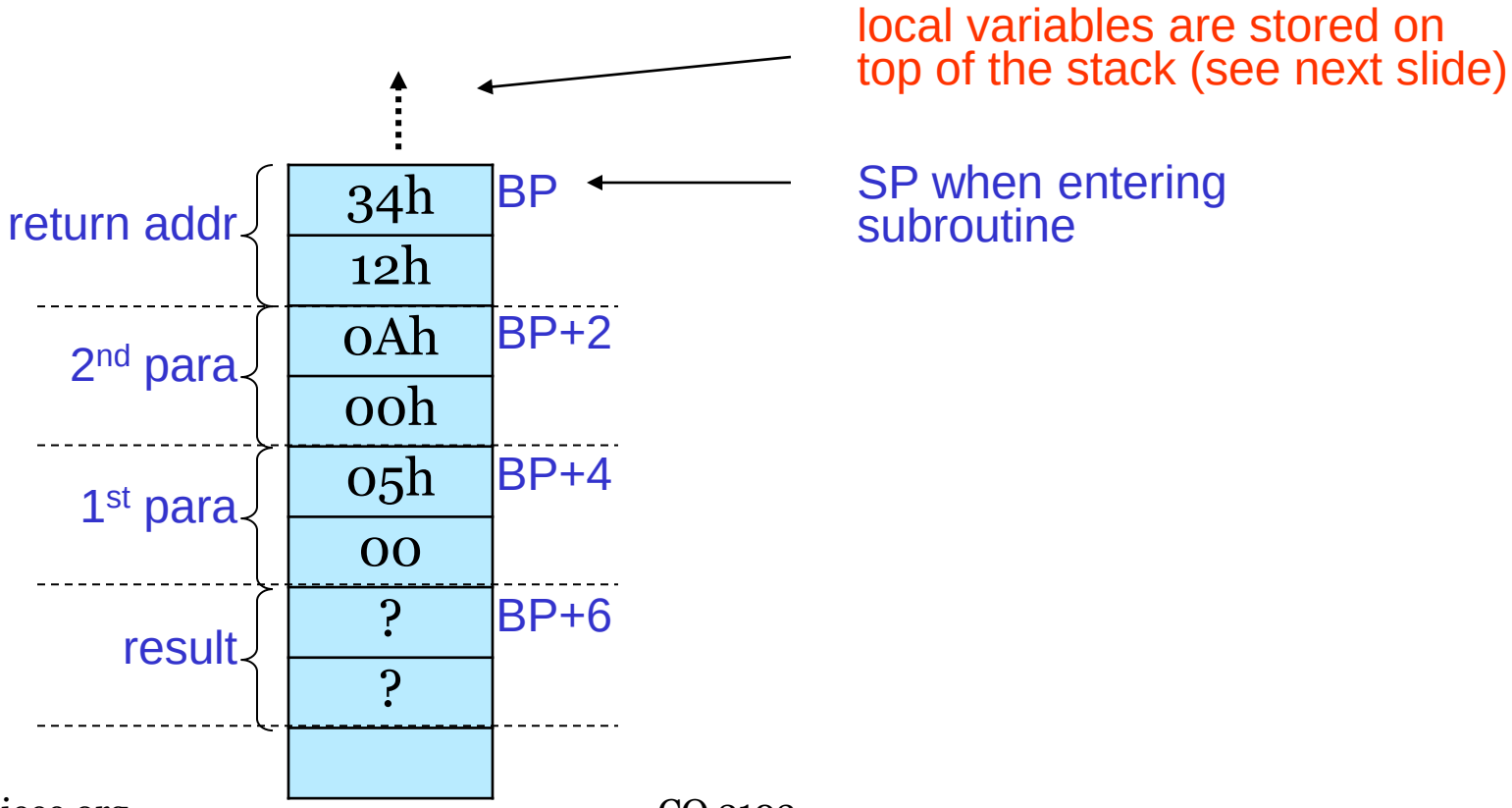
Getting Organized - 2

- In subroutine:

```
sbrtn proc near
    push bp     ;preserve BP (this was omitted in previous
                ;examples for simplicity; it is necessary to
                ;preserve BP before changing it)
    mov bp,sp   ;provides a fixed pointer into Stack Frame
    push …      ;preserve other registers, if necessary
    ...         ;your code
    ...         ;goes here, use BP to access parameters and result
    pop …       ;restores registers
    pop bp      ;restore BP
    ret n       ;returns to main program
                ;removing n parameter bytes from stack
sbrtn endp
```

Declaring Local Variables

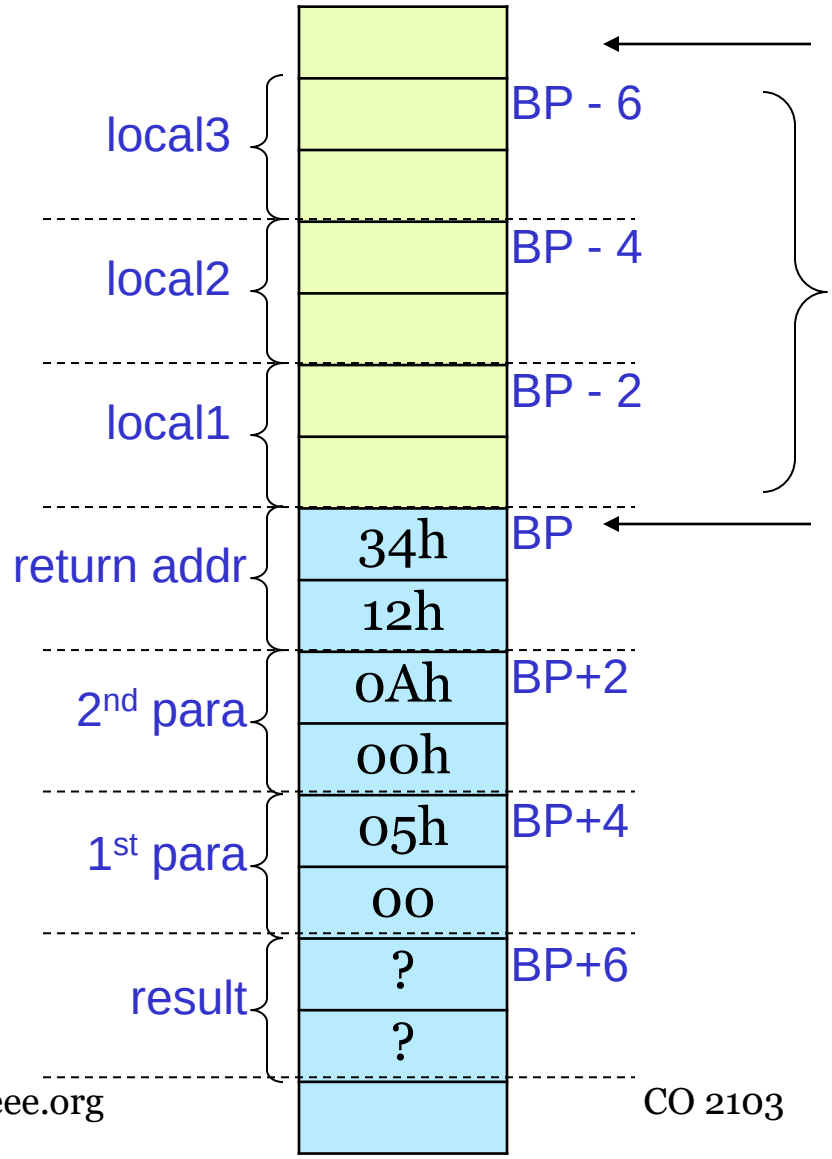
- Local variables will be on top of the stack, when it enters the subroutine



Local variables are declared in subroutine

```
sbrtn proc near
    local1  equ  [bp-2]          ; offsets to local
    local2  equ  [bp-4]          ; variables within
    local3  equ  [bp-6]          ; Stack Frame
    push bp    ;preserve BP
    mov bp,sp  ;provides a fixed pointer into Stack Frame
    sub sp,6   ; local storage: 6 bytes (3 words)
    push …     ;preserve other registers, if necessary
    ...        ;your code goes here, use BP to access parameters
    ...        ;and result; use labels (e.g. local1) or BP to
               ;access local variables)
    pop …      ;restores registers
    mov sp,bp  ;release Stack Frame (alternative: add sp,6)
    pop bp     ;restore BP
    ret n      ;returns to main program
               ;removing n parameter bytes from stack
sbrtn endp
```

Stack with Local Variables



"**`sub sp,6`**" move SP here so that Stack Frame for this subroutine should start here; i.e. will not interfere with local variables

local variables

SP when entering subroutine; "**`mov sp,bp`**" restores SP to this point when returning from the subroutine

Getting Organized - 1

- In main program:

```
...
sub sp,2        ;result space, if required
push par1       ;pushes any
"     "         ;parameters
push parn       ;onto the stack
call sbrtn      ;call the procedure
pop result      ;retrieve result, if required
...
```

Getting Organized - 2

```
sbrtn proc near                    ;in subroutine
    local1  equ  [bp-2]            ; offsets to local variables
    ...
    push bp    ;preserve BP
    mov bp,sp  ;provides a fixed pointer into Stack Frame
    sub sp,m   ;local variables storage: m bytes
    push …     ;preserve other registers, if necessary
    ...        ;your code goes here, use BP to access parameters
    ...        ;and result; use labels (e.g. local1) or BP to
               ;access local variables)
    pop …      ;restores registers
    mov sp,bp  ;release Stack Frame (alternative: add sp,m)
    pop bp     ;restore BP
    ret n      ;returns to main program
               ;removing n parameter bytes from stack
sbrtn endp
```

Time Delay Subroutine - 1

- Most commonly used subroutine
- Basically doing nothing for the required time
- Computation of delay time requires knowledge of processor speed (clock period) and instruction clocks

```
;short time delay
delay:    mov cx,n   ;assume this takes N1 clocks to complete
nothing:  nop        ;assume this takes N2 clocks to complete
          loop nothing  ;assume this takes N3 clocks to complete
```

Clock period `Tc = 1/fc` where **`fc`** is processor clock speed in *Hz*

Delay time `Td = [ N1 + n*(N2+N3) ] * Tc` *seconds*

`N1, N2, N3` are calculated from datasheet of the processor

Time Delay Subroutine - 2

- Maximum value of **n** is **65,535** (**CX** is 16-bit)
- For a given hardware, **fc** is fixed, maximum time delay will be:
  - **Td = [N1 + 65,535*(N2+N3)] * Tc** seconds
- Extend time delay using nested loops

```
;long time delay (use more nests for longer delay)
delay:    mov cx,m   ;N1 clocks
nest:     mov bx,n   ;N1 clocks
nothing:  dec bx     ;N4 clocks
          jnz nothing   ;N5 when true, N6 when false
          loop nest ;N3 clocks
```

Delay time **Td = [N1 + m*(N1+ n*N4+(n-1)*N5+N6 +N3)] * Tc** *sec*

Summary

- Indirect Addressing – useful in dealing with data structure, e.g. array
- Implementation of Program Control using CMP, conditional JMP and order of "doing" statements
- Importance of Stack – crucial usage
  - LIFO – growing from high to low memory
  - Instructions affecting Stack – PUSH, POP, CALL, RET and more
- Use of Stack in Procedure calls
  - save return address
  - preservation of registers
  - parameters & results passing
  - local variables