Subroutines

Thorne : Chapter 8

(Irvine Edition IV : Section 5.5)

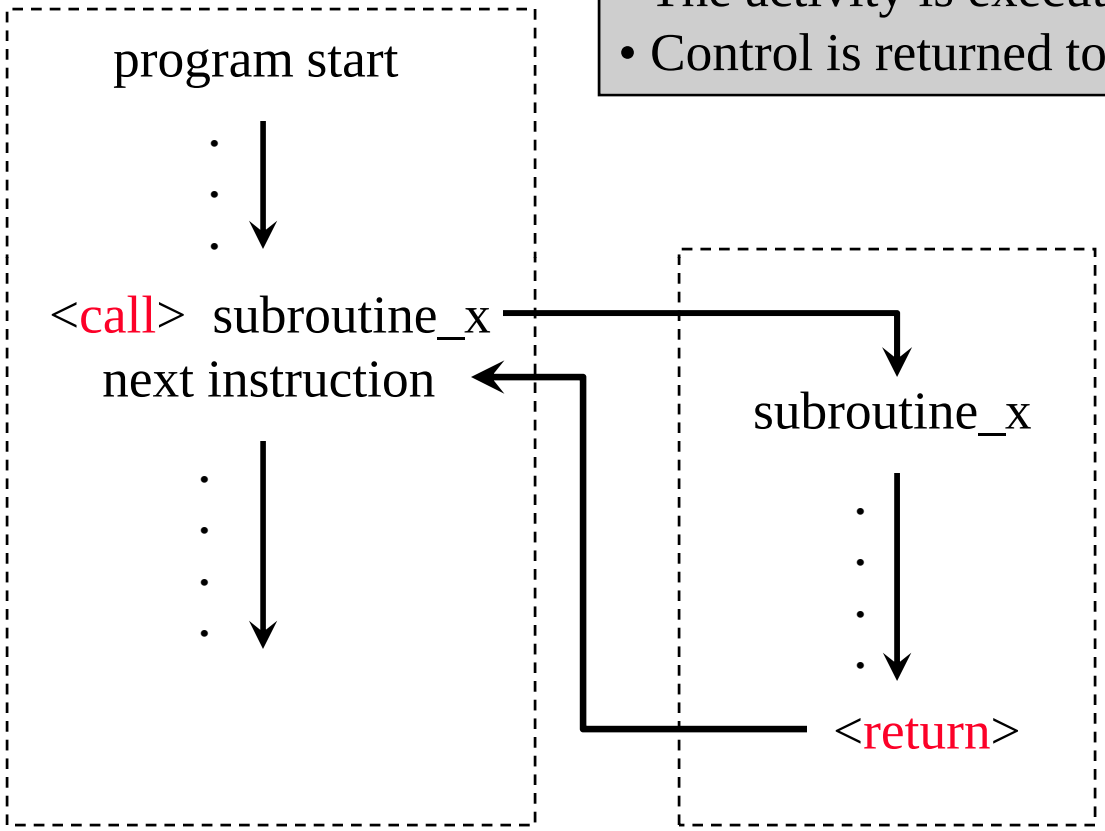
Subroutines

- A sequence of instructions that can be called from various places in the program
- Allows the same operation to be performed with different parameters
- Simplifies the design of a complex program by using the divide-and-conquer approach
- Simplifies testing and maintenance: separation of concerns
- Data structures handled by different subroutines: information hiding

-In a high-level language, they are called : **function**, procedure, method
-In assembly languages, they are called : **subroutine**

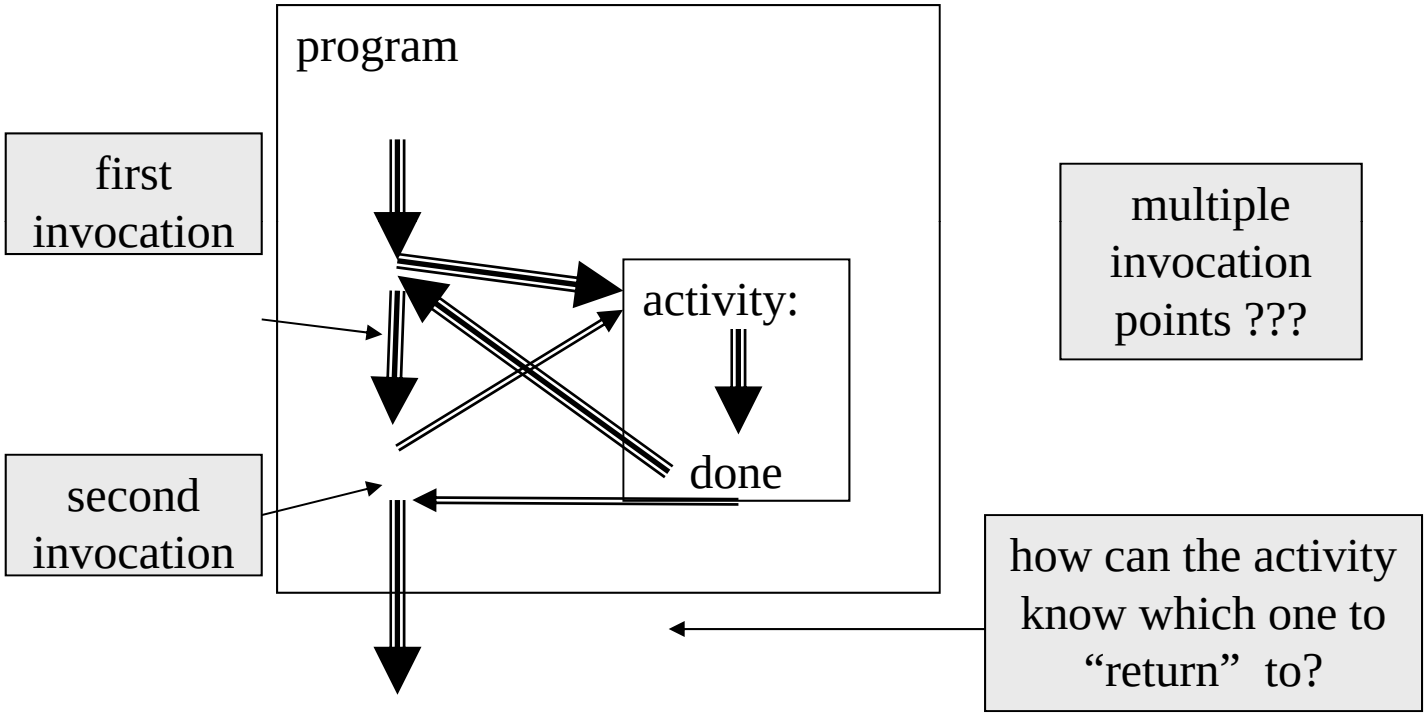
Subroutine Processing

Subroutines are a form of control flow
- Control is passed to the activity
- The activity is executed
- Control is returned to the invocation point



Program flow during a subroutine call

Multiple Subroutine Calls



During invocation, the invocation point must be saved.
During return, the invocation point must be restored.

Machine Level Implementation of Subroutines

CALL target ; invoke target subroutine

Execution Semantics:

1. Save the **return address** (address of next instruction) on run-time stack

   PUSH IP

IP value **AFTER fetching** CALL instruction!

2. Transfer control to activity

   JMP target

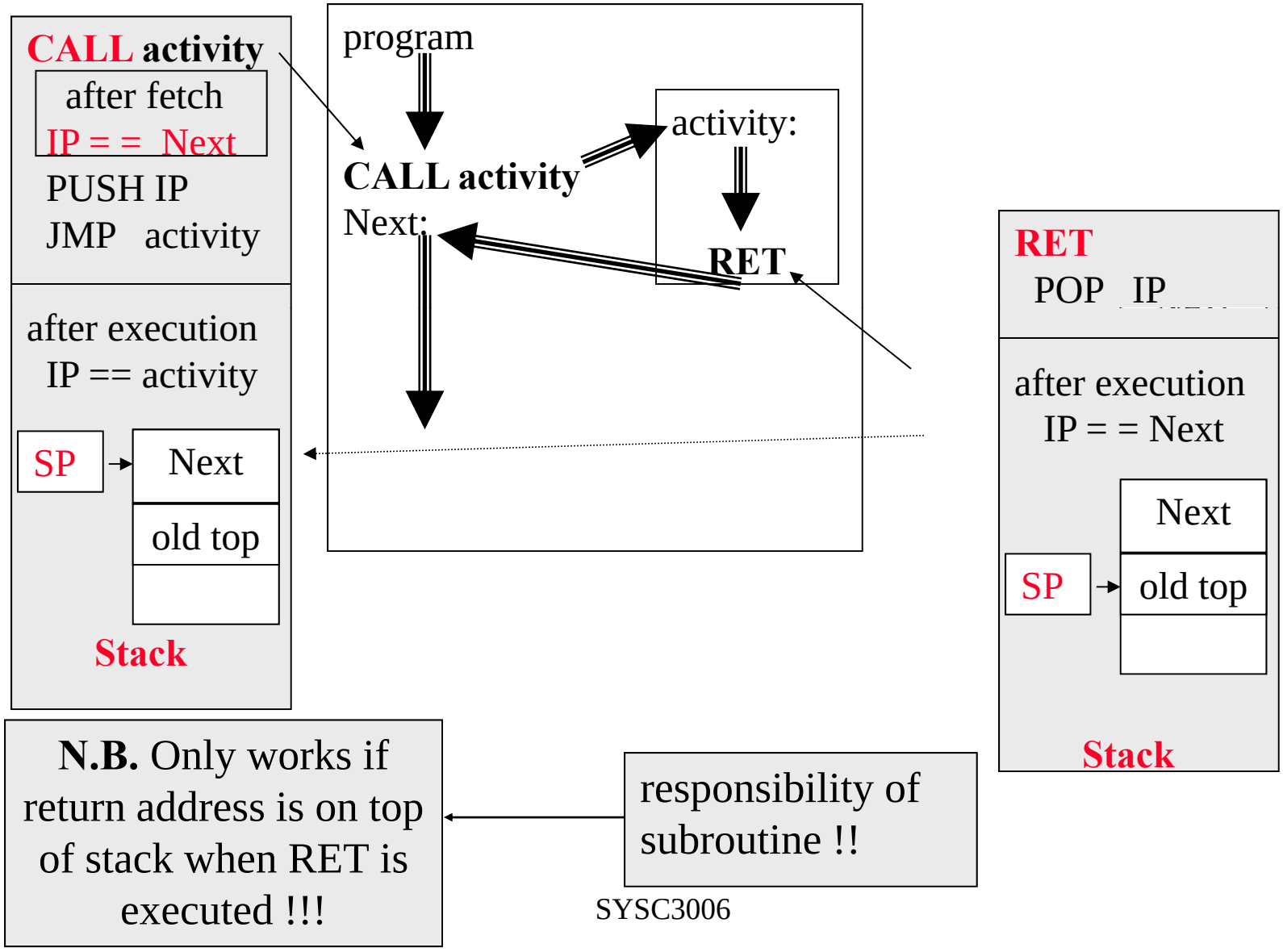
RET ; return from subroutine

Execution Semantics:

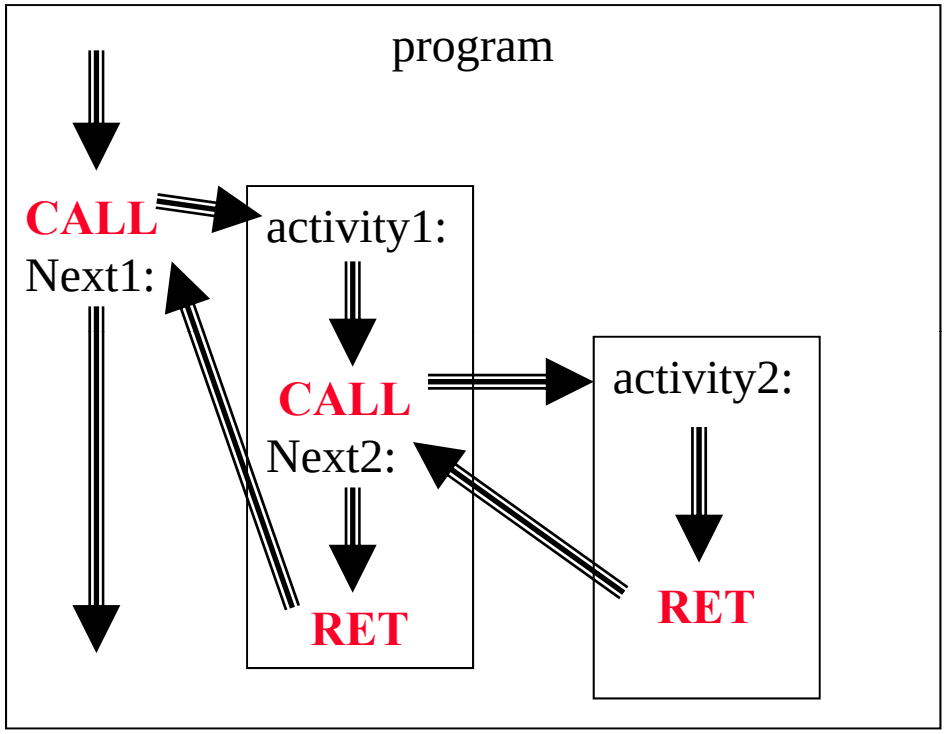
1. Return control to address currently saved on top of stack

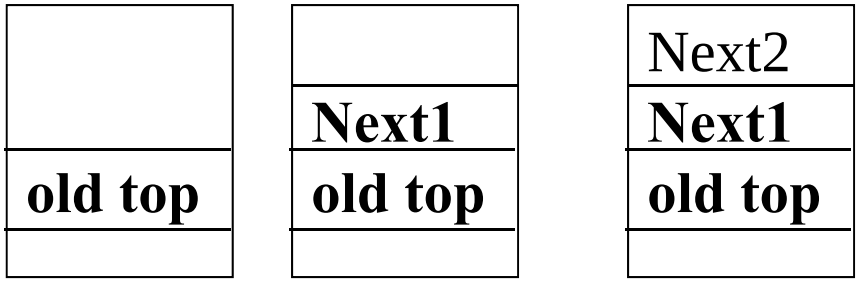
   POP IP

Subroutine Processing



Nested Subroutine Calls





Runtime Stack

Assembly Support : PROC Directive

- Informally, a subroutine is any named sequence of instructions that end in a return statement
- Intel Assembly has additional directives that provide more structure for encapsulation of the subroutine

```
.code
EXTRN subr:PROC
 main PROC
    MOV AX, @data
    MOV DS, AX
    …
    CALL subr
    …
    MOV AX, 4C00
    INT 21h
 main ENDP
 END main
```

Subroutine

```
 .code
 subr PROC NEAR
 PUBLIC subr
        PUSH BP
        MOV  BP, SP
        …
        …
        POP BP
        RET
 subr ENDP
 END
```

Issues in Subroutine Calls

We shall define subroutines using C-like prototypes

Return TYPE name Argument list : type name

```
void display (word number, byte base)
// Display the given number
// base = 0 for binary, =1 for HEX

byte absoluteValue (word number)
// Return the absolute value of the given
   number

boolean getSwitches ( byte &settings )
// Return the current settings of the
   switches and true if the switches
   bounced.
```

Issues in Subroutine Calls : Scope and Arguments

```
unsigned int displayAddress;

int main ()
{
    int number = 5, number2 = 6;
    display ( number2, 0);
    …
}
void display (word number, byte base)
{
    int divisor, digit;
    if (base == 0) divisor = 2 ;display in bin
    else divisor = 16; display in hex
    digit = number / divisor;
    …
    displayAddress++;
 }
```

Global variable

number2 is a PARAMETER

number is an ARGUMENT

Local variable

Issues in Subroutine Calls : Value versus Reference

```
int main ()
{
   int number = 5, number2 = 6;
   display1 ( number2, 0);
   display2 ( &number, 0);
}
void display1 (word number, byte base)
{
   …
   number = number / divisor;
}
void display2 (word &number, byte base)
{
   …
   number = number / divisor;
}
```

By Value

By Reference

Implementing Parameter Passing

- Parameters can be passed in various ways :
  1. Global Variables
  2. Registers
  3. On the stack.

- Global Variables
  - The parameter is a shared (static) memory variable (**in DS!**)
  - Parameters is passed when
    - **Caller** puts the value in the variable
    - **Callee** reads the value from the variable.

Parameter Passing using Global Variables

```
.data
 Value  DW  ?
```

C prototype: void activity(word aValue)

```
Caller          MOV     Value, 245
                CALL    activity
                . . .
Callee      activity PROC
                MOV     AX,  Value
                . . .
                RET
            activity ENDP
```

Passing parameters via global variables is **NOT** widely used in practice

- Consider nested subroutines (eg. a subroutine that calls itself)
- Consider large programs with many subroutines, each with many parameters;
- However, sometimes it is the only way (eg. interrupts), later!

Parameter Passing using Registers

- Parameters can *alternatively* be passed in registers
  - Each parameter is assigned to a particular register
  - **Caller** would load the registers with appropriate values
  - **Callee** would read registers to get the value.
- Register Parameters are used in DOS function call

```
MOV     AH, 9                   ; AH = OS Function (9=Print)
MOV     DX, OFFSET message      ; DX = Address of msg
INT         21h                 ; “Call” DOS function
```

- Advantage : Little overhead since values are in registers
- Disadvantage : There is a finite number of registers
  - What to do if more parameters than registers?

POLICY for SYSC-3006

Parameter Passing using the Runtime Stack

- Parameters can *alternatively* be passed on the runtime stack
  - **Caller** pushes the parameters onto the stack
  - **Callee** indexes into stack to access arguments

C prototype: void activity(word aValue)

```
Caller      MOV   DX, 245
            PUSH   DX
            CALL    activity
                  . . .
Callee      activity PROC
            MOV BP, SP
            MOV AX, [BP + ? ]
            . . .
            RET
      activity ENDP
```

Indirect addressing!

Why does SYSC-3006 use Parameter Passing via Stack ?

- Let's look at nested subroutine calls again …

```
void main ()
{
   …
   sub1(245)
}
```

```
void sub1(word value)
{
   …
   sub2(value*2)
}
```

```
void sub2(word x)
{
   …
   sub1(3);
}
```

Stack frame

Return to main
245

Return to sub1
490
Return to main
245

Return to sub2
3
Return to sub1
490
Return to main
245

SYSC-3006 Subroutine Policies – Register Save/Restore

- **Problem** : Subroutines need to use registers. What if the registers contain values that are needed by the caller upon return?
- **Solutions**: Assign responsibility to either caller or callee
  1. Caller has the responsibility to save all useful values before calling the subroutine.
     - The callee is then free to use any register
     - Upon return, the caller restores the useful values
  2. Callee (subroutine) must save any register before it uses it and restore it to its original value before returning.
     - Caller is guaranteed that its registers are the same before and after the subroutine call.
     - More efficient because subroutine knows what registers it uses.
- **3006 Policy** : Solution 2 with one exception : The register(s) used to pass out return TYPE cannot be preserved

SYSC-3006 Subroutine Policies – Local Variables

- **Problem** : Subroutines often have local variables that exist only for the duration of the subroutine.
  - Example

```
double average (double array[], int number)
{
    double total = 0;
    for (int i=0; i< number; i++ )
    {
        total += array[I];
    }
    double result = total / number;
    return result;
}
```

Local Variables

- **SYSC-3006 Policy** : Local variables are maintained as register variables or by using the stack as a temporary storage buffer.

SYSC-3006 Subroutine Policies – Parameter Passing

- **3006 Policy** : Parameters shall be passed on the stack.
  - The **caller** must push the parameters on the stack before calling
    - With multiple parameters, parameters are pushed from right-to-left
  - The **caller** must remove the parameters from the stack upon return.

base is in AL

Example: `void display (word number, byte base)`

```
Caller:   MOV   AL, 0 ; binary
          PUSH  AX
          PUSH  [BX+SI]
          CALL  display
          ADD   SP, 4
```

Byte parameters are passed in LSB of a word

Parameters can be cleared by POPing or by simply adjusting the SP. Why ADD ? Why 4 ?

number is at address (BX+SI)

SYSC-3006 Subroutine Policies – Parameter Passing

– The **callee** must index into the stack to access the parameter values, using a stack frame ….

– A stack frame is a consistent view of the stack upon beginning the core code of the subroutine.

- It provides a uniform method for accessing parameters passed on the stack using BP based indirect addressing regardless of the number of arguments and/or the number of registers saved/restored by the subroutine

SYSC-3006 Subroutine Policies – Stack Frame

```
anySub proc
    PUSH BP
    MOV BP, SP
    ; PUSH any registers used
    ; Core code of the subroutine where the work
       is done
    ; POP all registers that were saved
    ;  (in reverse order!)
    POP BP
    RET
anySub endp
```

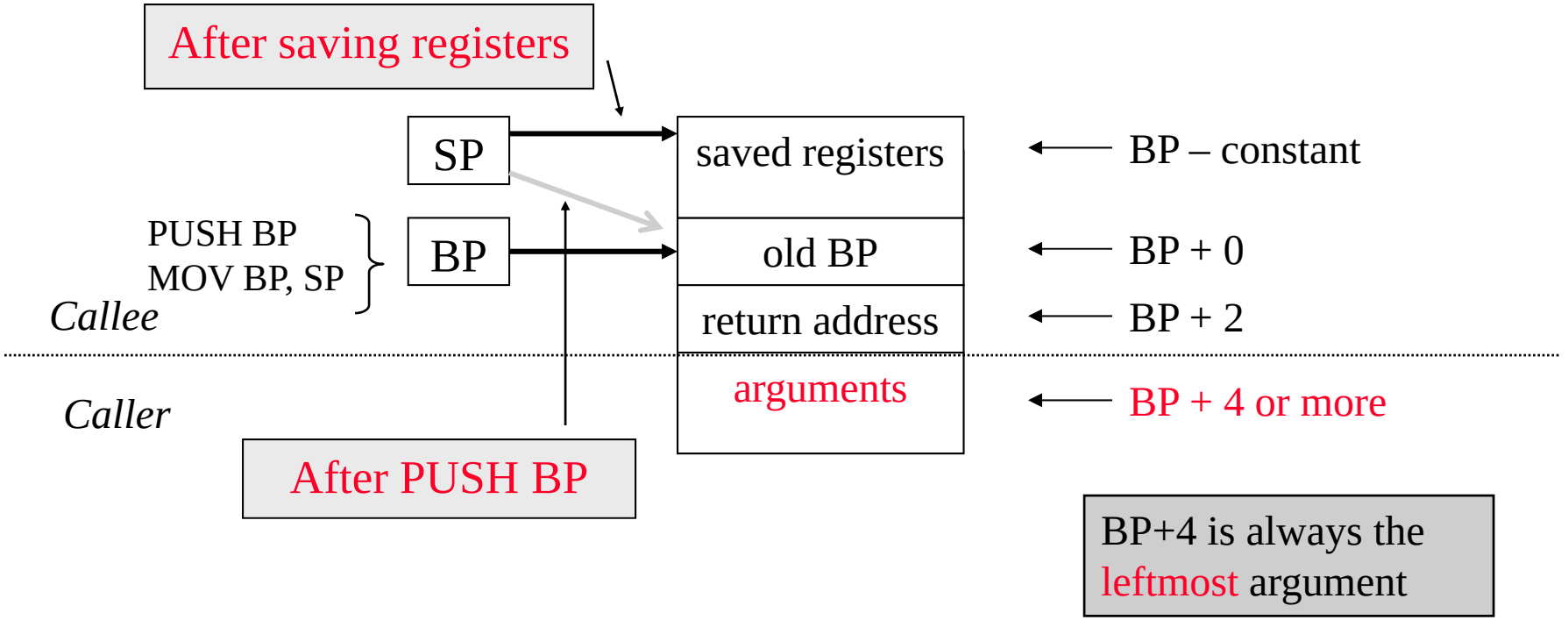
Standard Entry Code

Standard Exit Code

SYSC-3006 Subroutine Policies – Stack Frame

- The stack frame associated with the subroutine skeleton

Stack Frame is another policy



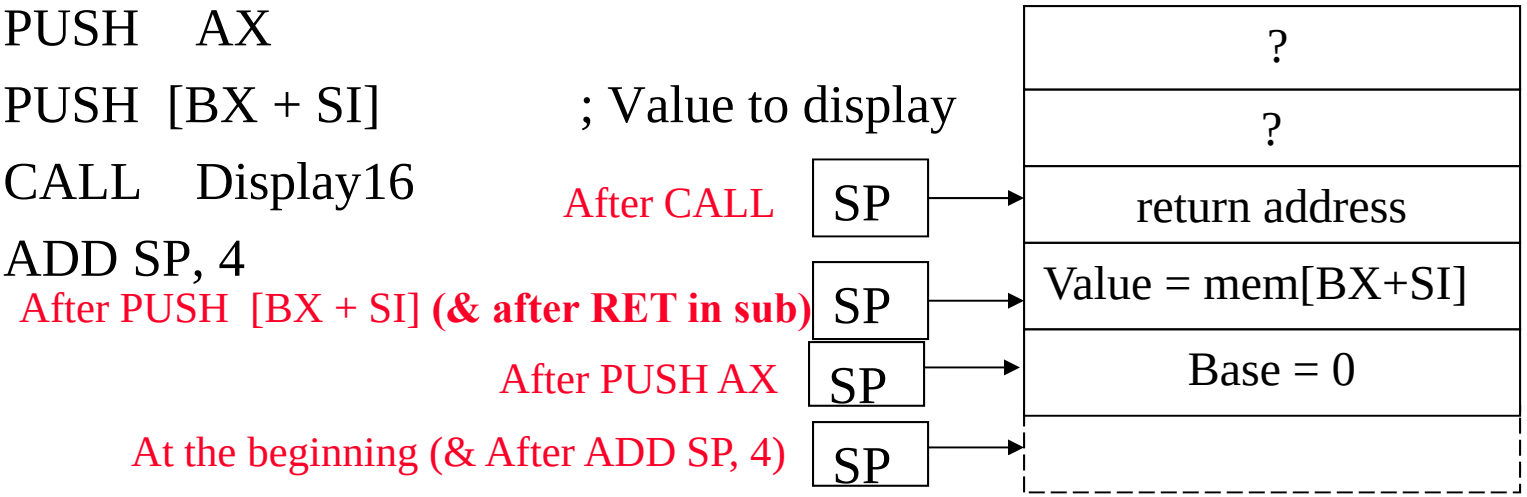
SYSC-3006 Subroutine Policies – Stack Frame

- Example : Recall our previous example

  void display(word Value, byte Base);

Call set up: (By the caller)

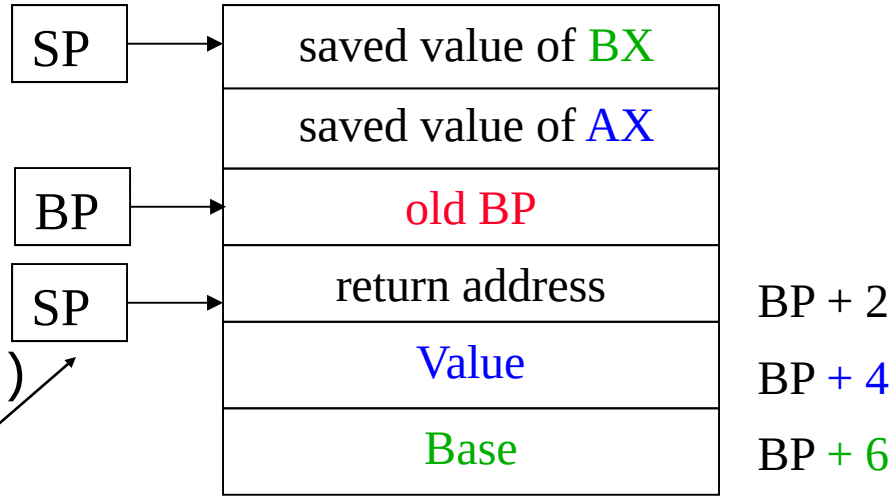
```
MOV     AL, 0              ; Base = binary
PUSH    AX
PUSH  [BX + SI]            ; Value to display
CALL    Display16
ADD SP, 4
```



SYSC-3006 Subroutine Policies – Stack Frame

Subroutine Implementation (In body of Display **)**

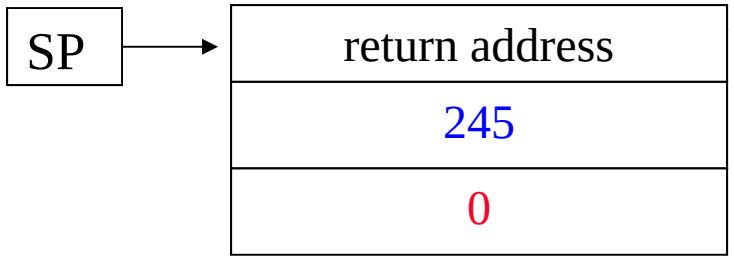
```
display PROC
    PUSH BP
    MOV BP, SP
    PUSH AX
    PUSH BX
    ; Get value
    MOV   AX, [BP + 4]
    ; if (base == binary)
    MOV  BL, [BP+6]
    CMP  BL, 0
    …
    POP BX
    POP AX
    POP BP
    RET
display ENDP
```



Issue : pass by value vs. pass by reference

- **Definition** : Pass by value
  - The argument is a copy of the value of interest
  - In high-level languages like C++, pass-by-value is the default way to pass simple variables (primitive types like int, char, float)

- **Example** : Pass-by-Value




```
int myValue;
myValue = 245;
display( myValue, 0 );
```

```
myValue dw 245
MOV AL,0
PUSH AX
PUSH myValue
CALL display
ADD SP, 4
```

Content of myValue is PUSHed but not the address!

Issue : pass by value vs. pass by reference

When passing-by-value : Inside the subroutine, arguments passed in the stack can be treated like local variables

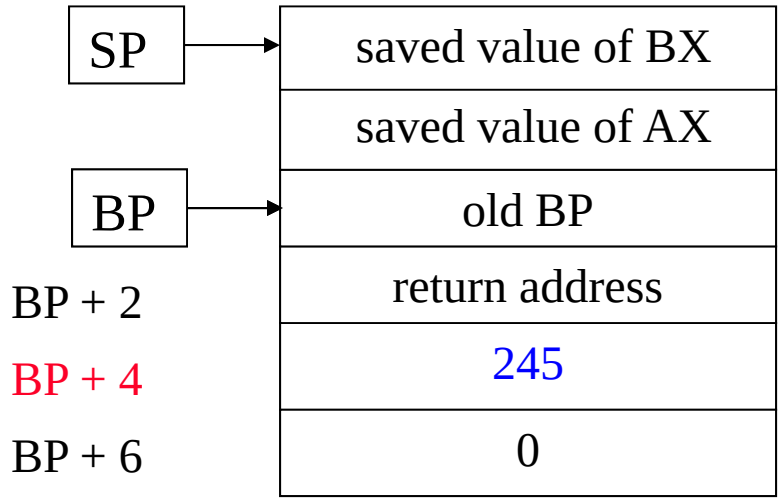
- The contents of the stack can be read … and modified
- The variable is local and exists ONLY during the subroutine execution
  - Why ?
  - Consequence : Any modifications to the arguments on the stack are not persistent and cannot be seen by the caller

In Previous DisplayExample

- The subroutine can change the copy of MyValue

```
MOV     [BP + 4], AX
```

- The change will be made to the copy on the stack, and not to the original variable.



Issue : pass by value vs. pass by reference

- **Definition** : Pass by reference
  - The argument is the address of a memory variable
  - Used when you need access to the caller's variables either :
    - The purpose of subroutine is to modify caller's variables
    - To pass large composite structures that would require too much time/space on the stack if passed-by-value.

- In high-level languages,
  - Default : Pass-by-value

```
int value;
```

  - Pass-by-reference requires additional syntax : & operator.

```
int & value;
```

Issue : pass by value vs. pass by reference

- **Example** : Pass by reference

void SortArray (int & SortMe[ ], int Size);

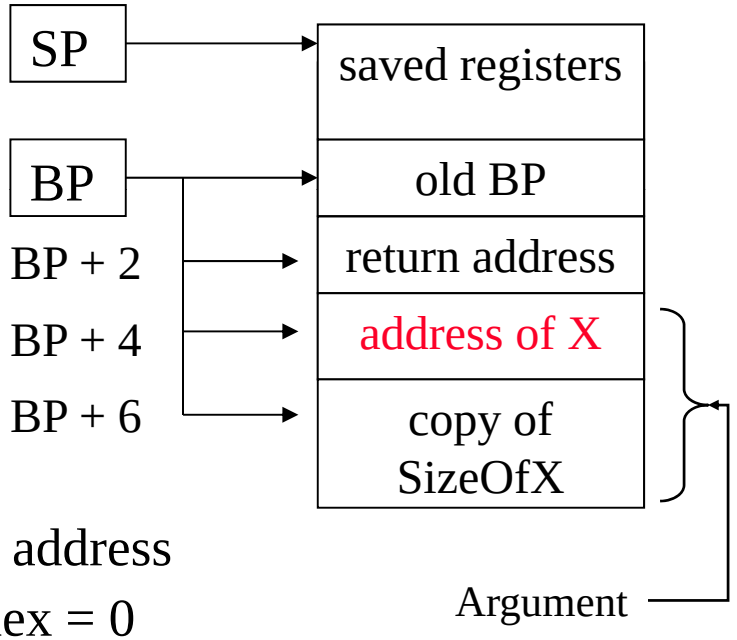
```
; array declaration
X           DW
            DW
    . . .
SizeOfX     DW

Caller :
    PUSH        SizeOfX
    MOV         AX, OFFSET X
    PUSH        AX
    CALL        SortArray
    ADD         SP, 4
```

pass **offset** of array X

why not: PUSH X????

Issue : pass by value vs. pass by reference

- **Example** : Pass by reference



Callee : Inside the subroutine SortArray:

```
MOV     BX, [ BP + 4 ]    ; get array address
MOV     SI, 0             ; array index = 0
. . .
MOV     AX, [ BX + SI ]   ; get array element
```

SYSC-3006 Subroutine Policies – Return Types

Subroutines can return information to the caller in two ways

1. Return (a) value(s) in (a) variable(s) that is (are) passed-by-reference
2. Return a value via the subroutine's return type

   Example :

   ```
   boolean  AbsValue( int & X, int Y );
   ```

   where **boolean** is usually a byte, with 0 = false,

   non-zero = true

SYSC-3006 Subroutine Policies – Return Types

- Passing the return type back from the subroutine to the caller could be done in any of the three ways used to pass parameters in .
  - Global variables (same troubles as before)
  - On the stack
    - For example, after passing any parameters, the caller could allocate an extra word in stack before call
      - SUB SP, 2
    - callee could return value there
  - Via registers (There is only one return type, need only one register)

SYSC-3006 Subroutine Policies – Return Types

- Return-Value POLICY in SYSC-3006 (same as most High Level Languages)
  - return 8-bit value in AL
  - return 16-bit value in AX
  - return 32-bit value in DX:AX (as with 32-bit values for DIV instruction)

- Implications of Return-Value Policy
  - do not save/restore register(s) used for return-value
  - the purpose of the subroutine is to return a value in the register(s)
  - if 8-bit value (returned in AL) – subroutine is not responsible for persistence of AH value

Are the SYSC-3006 Subroutine Policies Practical ?

- Is it worth the effort to understand the 3006 Subroutine policies ?
- The policies follow industry practices for compiler-writing
  - Proof of the pudding : Additional Intel Directives for Subroutines

INVOKE

INVOKE display, 256, 0 generates

```
PUSH 0
PUSH 256
CALL display
```

ADDR

INVOKE display, ADDR myValue, 0 generates

```
PUSH 0
PUSH offset myValue
CALL display
```

An alternate form of RET

RET immediate

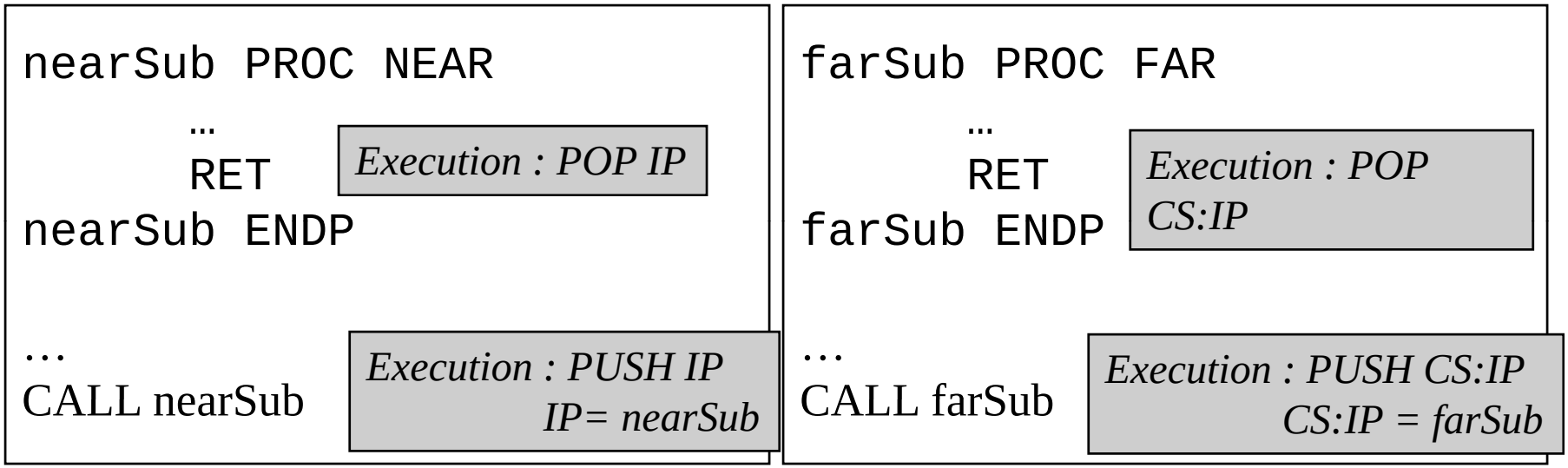
Add the immediate value to SP after popping the return address (Why?)

Intra- versus Inter-Segment Subroutine Calls

- All of the examples of subroutines so far use intra-segment control flow
  - Only the IP is saved/changed/restored.
  - **Terminology** : These subroutines are **NEAR**
- In large programs and/or software libraries, subroutines can be located in different code segment
  - Require inter-segment control flow where both the CS and the IP are saved/changed/restored.
  - **Terminology** : These subroutines are **FAR**
- The PROC directive uses an optional modifier to denote the type of control flow
  - PROC NEAR or PROC FAR
  - By default, without any modifier, a subroutine is NEAR.

Example : NEAR versus FAR subroutines

- The PROC modifier influences how a subroutine is called



1. How does the program know if it is NEAR or FAR ?
2. Don’t we need different RET statements ?
3. Draw a picture of a FAR stack frame?Which is pushed first:CS or IP?
4. Can a subroutine be both NEAR and FAR ?
5. Can a subroutine call a FAR subroutine that is in the same segment ?

NEAR versus FAR subroutines

```
0000                    .code
0000                    main PROC
0000  E8 0004            CALL  nearsub
0003  0E E8 0001          CALL  farsub
0007                    main ENDP

                        ; NEAR subroutine
0007                    nearsub PROC NEAR
0007  C3                 RET
0008                    nearsub ENDP

                        ; FAR subroutine
0008                    farsub PROC FAR
0008  CB                 RET
0009                    farsub ENDP
                        END main
```

E8 → CALL

0E → PUSH CS

C3 → RET (NEAR)

Intra-segment

CB→ RET (FAR)

Inter-segment