Subroutines

Subroutines

- Sequence of instructions that can be called from various places in a program
- Same operation to be performed with different parameters
- Simplifies design of complex program
- Simplifies testing and maintenance: separation of concerns
- Data structures handled by different subroutines: information hiding

- In a high-level language, called: **function**, **procedure**, **method**

- In assembly languages, 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

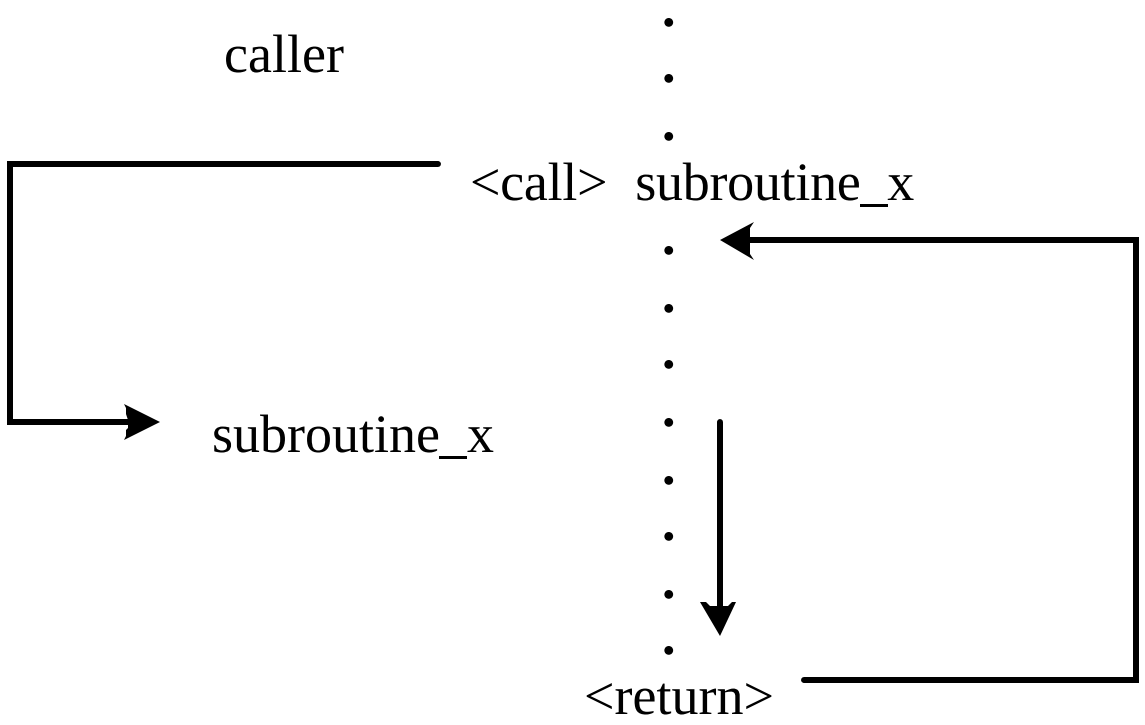
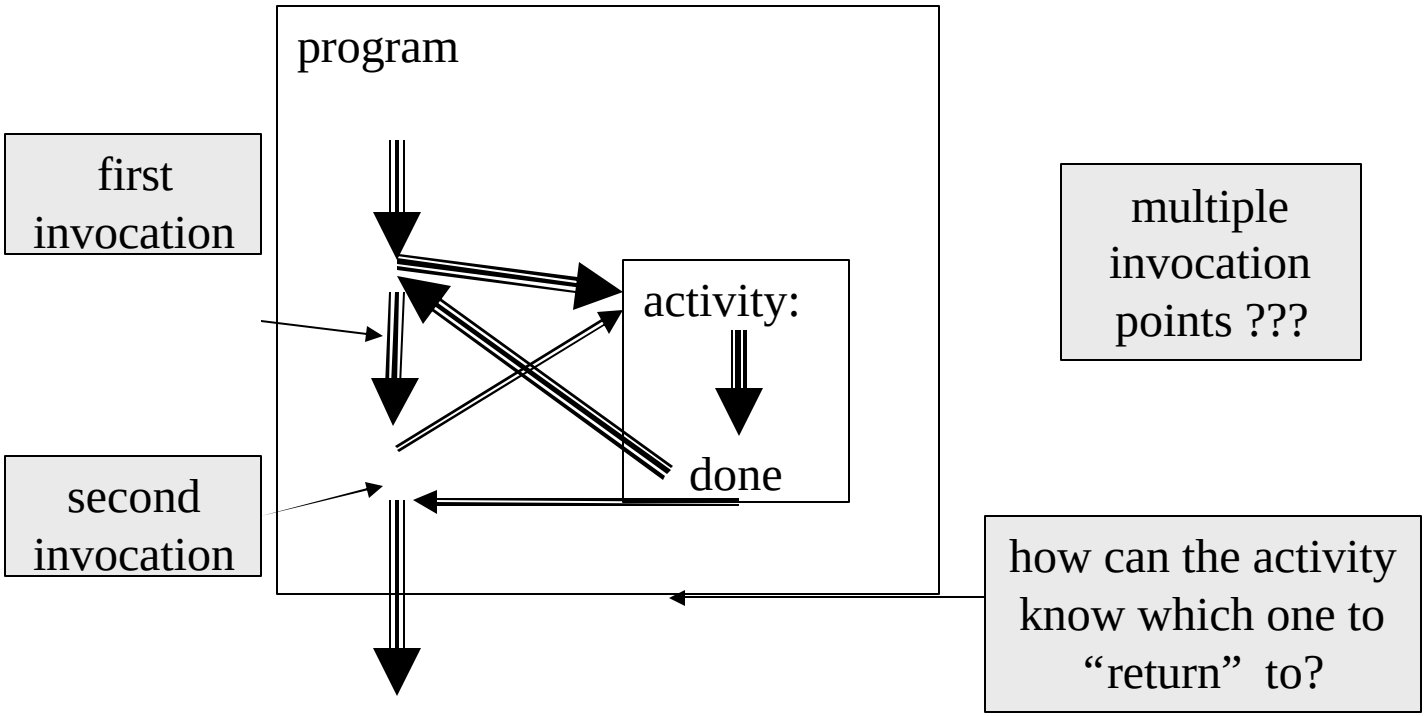


Figure4.8 Program flow during a subroutine call

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 **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
```

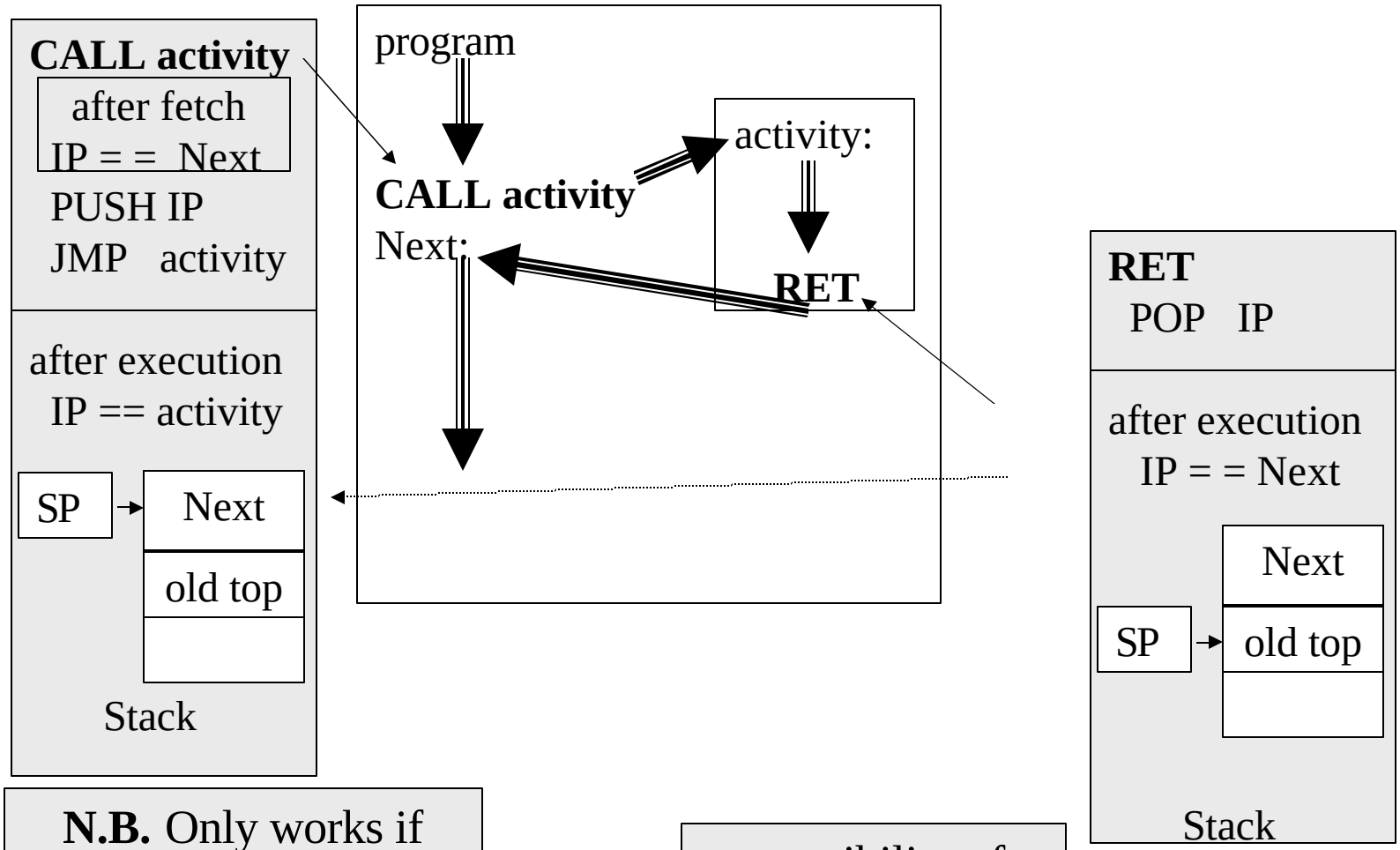
```
RET               ; return from subroutine
```

Execution Semantics:

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

```
POP   IP
```

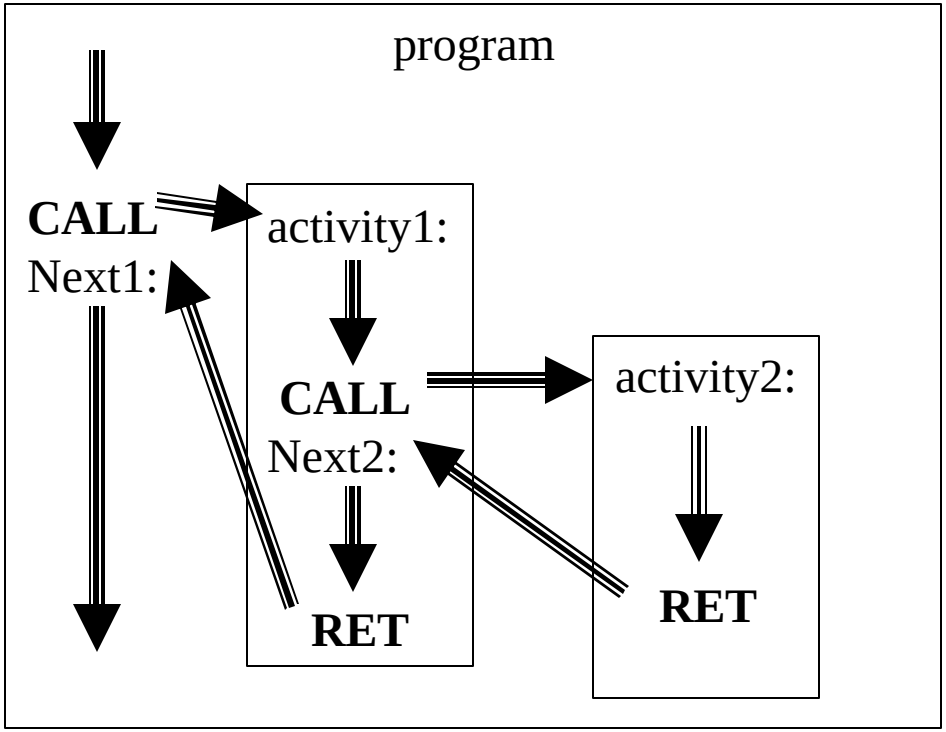
Subroutine Processing

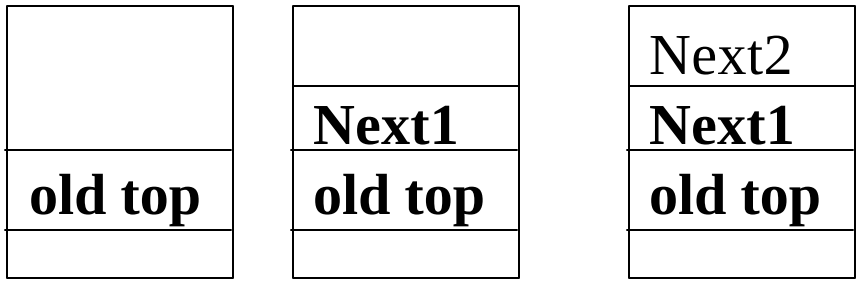


N.B. Only works if return address is on top of stack when RET is executed !!!

responsibility of subroutine !!

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: additional directives that provide more structure for encapsulation of the subroutine

```
main    PROC                    subr PROC
   ...                             ...
   MOV   AX, 4C00h                 RET
   INT   21h                    subr ENDP
main ENDP
```

Issues in Subroutine Calls

We shall define subroutines using C-like prototypes. Include as comments in Assembly programs.


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

; byte absoluteValue (word number)
; Return absolute value of given number

; boolean getSwitches ( byte &settings )
; Return current settings of 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
    else divisor = 16;
    digit = number / divisor;
    …
    displayAddress++;
 }
```

Global variable

Local variable

number2 is a PARAMETER

number is an ARGUMENT

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
  - Parameters is passed when
    - **Caller** puts the value in the variable
    - **Callee** reads the value from the variable.

Parameter Passing using Global Variables

C prototype: void activity(word Value)

```
                Value  DW
Caller          MOV    Value, 245
                CALL   activity
                       . . .

Callee          activity PROC
                       MOV  AX, Value
                       . . .
                       RET
                activity ENDP
```

Passing parameters via global variables: not widely used in practice

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

Parameter Passing using Registers

- Parameters *alternatively* be passed in registers
  - Each parameter assigned to a particular register
  - **Caller** load registers with appropriate values
  - **Callee** read registers to get values.

- Register Parameters used in DOS

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

- Advantage: little overhead (values in registers)
- Disadvantage: a finite number of registers
  - What to do if more parameters than registers?

Parameter Passing using the Runtime Stack

POLICY for SYSC-3006

- Parameters *alternatively* passed on the runtime stack
  - **Caller** pushes 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 AX, [SP + ? ]
            . . .
            RET
       activity ENDP
```

Indirect addressing!

SYSC-3006 Subroutine Policies – Register Save/Restore

- **Problem**: Subroutines use registers. What if registers contain values needed by the caller upon return?

- **Solutions**:
  1. Caller save useful values before calling the subroutine.
     - Upon return, caller restores useful values

  2. Callee (subroutine) save any register before it uses it. Restores original value before returning.
     - Caller guaranteed that its registers are the same before and after subroutine call.
     - More efficient: subroutine knows what registers it uses.

- **3006 Policy**: Solution 2 with one exception: 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 maintained as register variables or by using the stack as a temporary storage buffer.

SYSC-3006 Subroutine Policies – Parameter Passing

- **3006 Policy**: Parameters passed on the stack.
  - **Caller** push parameters on the stack before calling
    - multiple parameters: parameters are pushed right-to-left
  - **Caller** remove parameters from the stack upon return.

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

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

Byte parameters passed in LSB of a word

Parameters can cleared by POPping or simply adjusting SP. Why ADD? Why 4?

SYSC-3006 Subroutine Policies – Parameter Passing

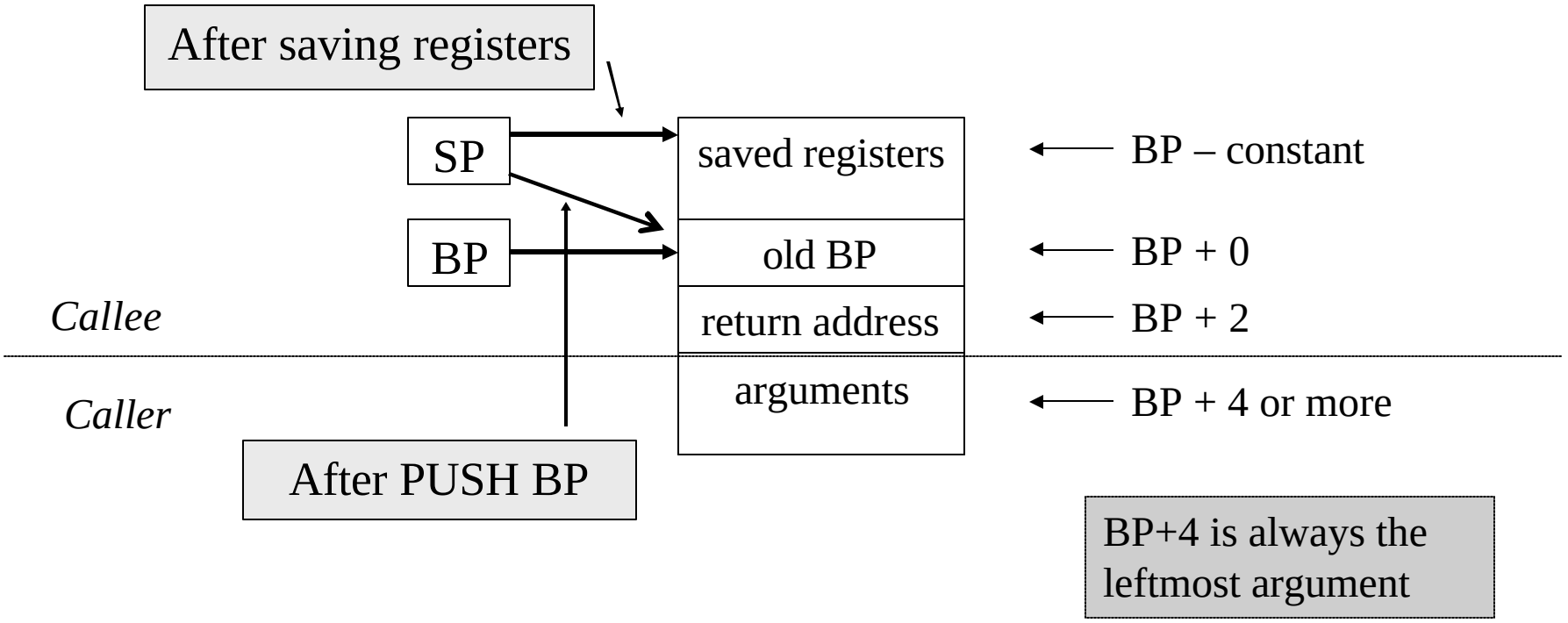
- **Callee** must index into the stack to access parameter values, using a stack frame.

- Stack frame: consistent view of the stack upon beginning the core code of the subroutine.
  - Provides uniform method for accessing parameters passed on the stack.
  - Uses BP based indirect addressing regardless of number of arguments and/or number of registers saved/restored by subroutine

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

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

Standard Entry Code

Standard Exit Code

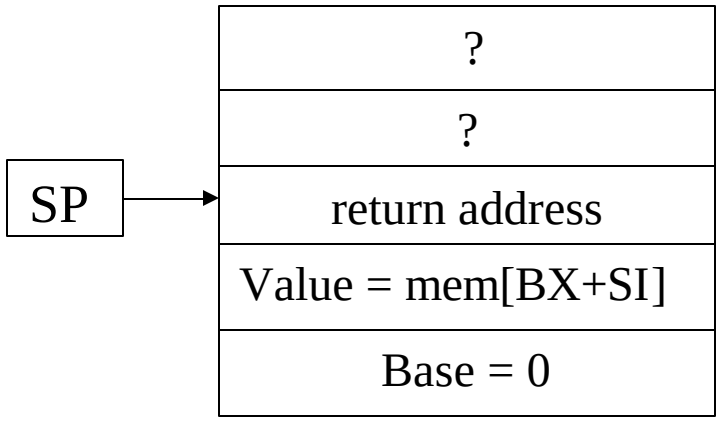
SYSC-3006 Subroutine Policies – Stack Frame

- Example : Recall our previous example

```
void  display(word Value, byte Base);
```

Call set up: (By the caller)

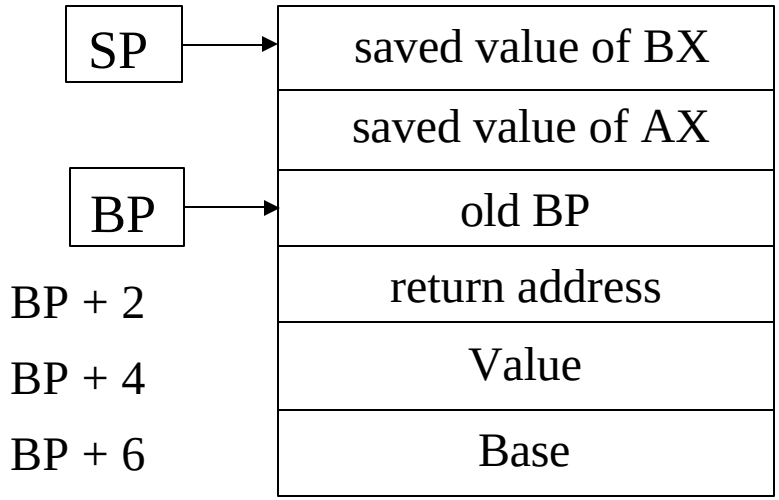
```
MOV    AL, 0          ; Base = binary
PUSH   AX
PUSH  [BX + SI]       ; Value to display
CALL   Display16
```



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
```



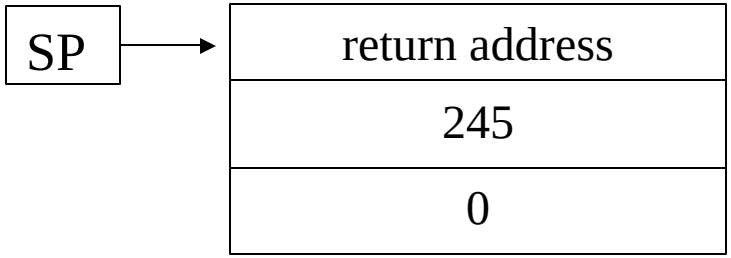
Issue : pass by value vs. pass by reference

- **Definition**: pass by value
  - Argument: a copy of the value of interest
  - High-level languages (C++), pass-by-value is default to pass simple variables (primitive types like int, char, float)

- **Example**:

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

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



Issue: pass by value vs. pass by reference

Passing-by-value: Inside subroutine, arguments in the stack treated like local variables

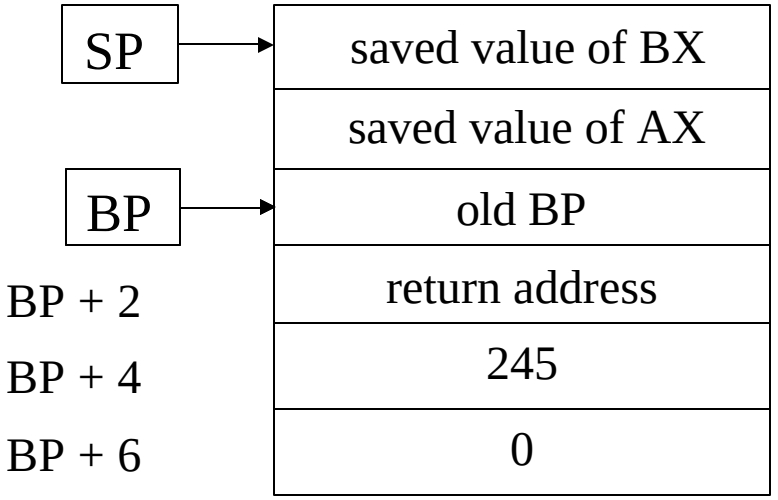
- Contents of stack can be read and modified
- Variable: local and exists ONLY during the subroutine execution
  - Why ?
  - Consequence: any modifications to arguments on stack are non-persistent; cannot be seen by the caller

Previous Display Example

- Subroutine can change copy of MyValue

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

- Change made to copy on the stack, and not to the original variable.



Issue : pass by value vs. pass by reference

- **Definition**: pass by reference
  - Argument: **address** of a memory variable
  - Used to access caller's variables if:
    - Purpose of subroutine: modify caller's variables
    - Pass large composite structures (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

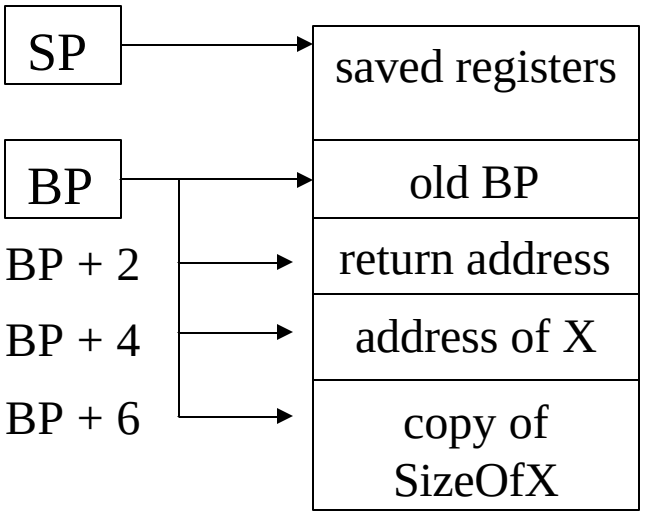
```
; void SortArray (int & SortMe[ ], int Size);

; array declaration
X:          DW
            DW
    . . .
SizeOfX:    DW

Caller :            ; SortArray (X[], SizeOfX);
    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 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 values in a variables 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 return type back from subroutine to caller 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 parameters, caller could allocate extra word in stack before call

      ```
      SUB  SP, 2
      ```
    - callee could return value there
  - Via registers (only one return type, need only one register)

SYSC-3006 Subroutine Policies – Return Types

- High Level Languages: registers to pass return type of a subroutine.
- So will SYSC-3006

- Return-Value POLICY in this course
  - 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)

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