Flow Control

- Flow Control Instructions
- Stack Implementation
- Procedures
- Parameter Passing
- Recursion

Flow-Control Instructions

- CPUs generally execute a linear sequence of instructions
  i.e. first instruction followed by the next one in memory (etc)

- The PC holds the address of the next instruction to be fetched

- Flow Control Instructions change the PC contents

- Three basic types of control transfer instructions:

    - HALT
    - JUMP
    - CALL

HALT

- Places microprocessor in an idle state. (Intel x86 HLT)
- Requires an interrupt or reset to continue.
- Can be used for power saving
  - Intel ‘C1’ state. Internal CPU clock is stopped.
  - Only units to receive a clock are the bus interface unit and the Programmable Interrupt Controller (PIC).

Jump

- A jump is a straightforward change of the PC contents:
- **Unconditional**: This type of jump is always executed.
  e.g. JMP M
  transfer execution to memory address M
- **Conditional**: This jump is only executed if the condition is attached to the jump is satisfied. (usually tests flag bits in the status register)
  - JZ M
    Jump if the Zero flag is set in the status register
  - JGE M
    Jump if greater than or equal to zero

Calls

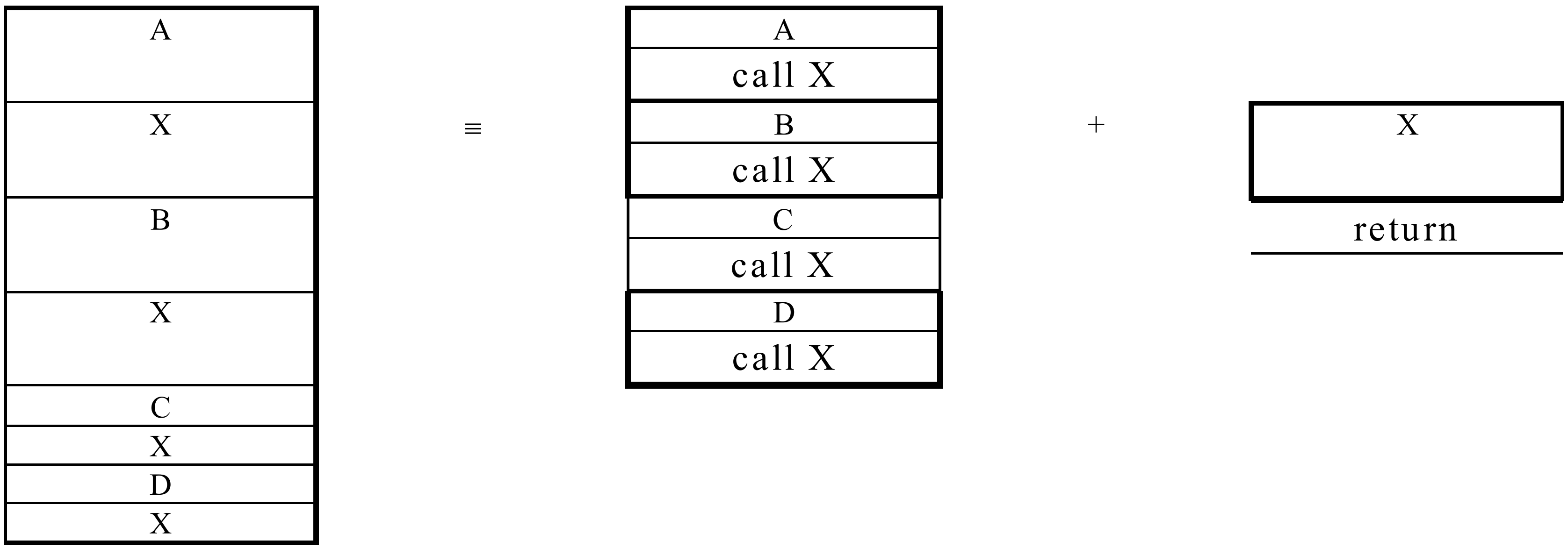
- A call differs from a jump in that its action is not completed with the change of flow.
- A call instruction acts like a jump but remembers the address of the next instruction following the call instruction itself.
- At some later point, the program can RETURN to this remembered point and continue execution.
- This is the mechanism for SUBROUTINES
- Code Economy , Code Modularity

Addressing

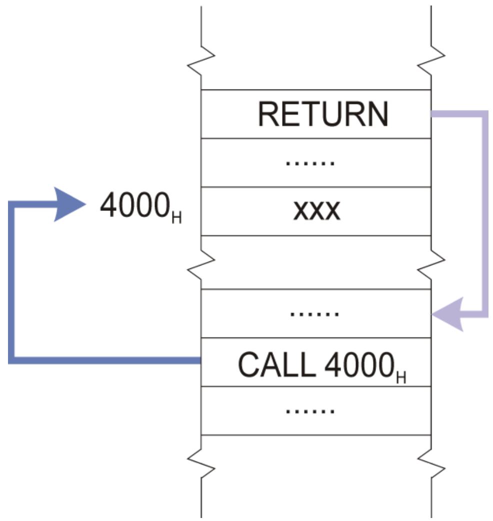
In both jumps and calls, the target address must be specified. There are 3 major modes:

- **Direct**
  The operand of the operation is the target address.
- **Indirect**
  The operand is the address where the value of the target address is held.
- **PC Relative**
  The operand is an offset which is added to the value in the PC to form the target address. (displacement may be limited)

Code economy and modularity



A Subroutine Call



Stacks

Q How does the call instruction remember the correct return address?

Top of Stack (ToS)

A It is done using a Stack.

- A Last-In-First-Out (LIFO) stack is used for subroutine calling.
- Allows recursion

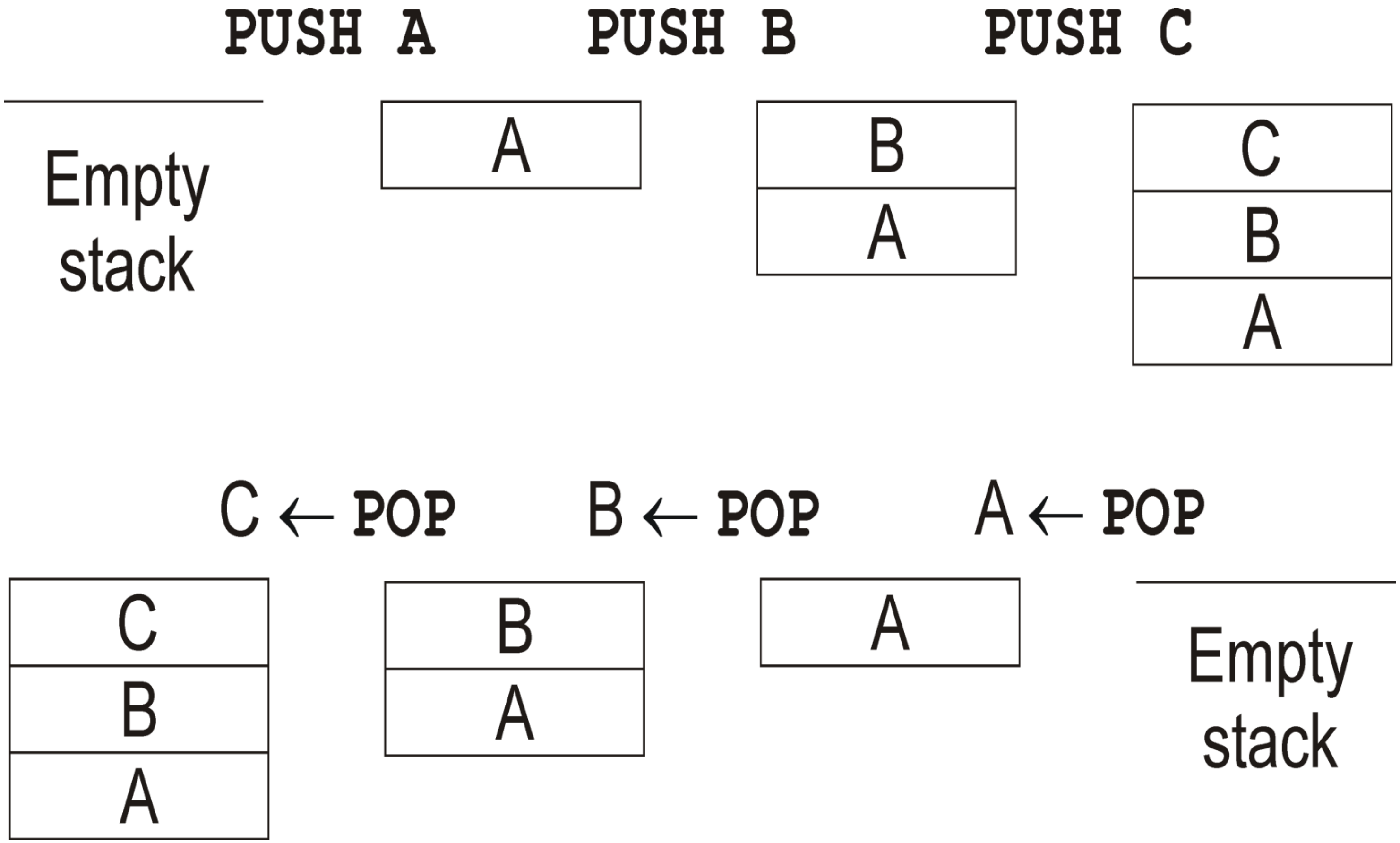
A LIFO stack consists of:

- a block of normal memory
- two operations used to put data on to the stack (store it in the memory), and remove data from the stack.

PUSH	Puts a value on to the stack.
POP	Retrieves the last value that was pushed onto the stack

The data is retrieved from the LIFO stack in the reverse order to the order in which it is placed on to the stack - hence the name.

Consider an empty stack, and the following operations.
(by convention stacks grow down))



Note …..

- A PUSH/POP pair will leave the stack as it was originally.
- POP applied to an empty stack is an error,
- PUSH applied to a full stack (depends on the implementation) is an error.

The CPU maintains the address of the current top of the stack (TOS) in a special **Stack Pointer** register, **SP**.

How is a LIFO Stack Implemented?

- Usually, memory is provided from normal system memory.
- The address holding the ToS is identified by a Stack Pointer register (SP).
- When PUSH or POP is executed, data in the stack is not moved, only the value of SP is changed. For example.

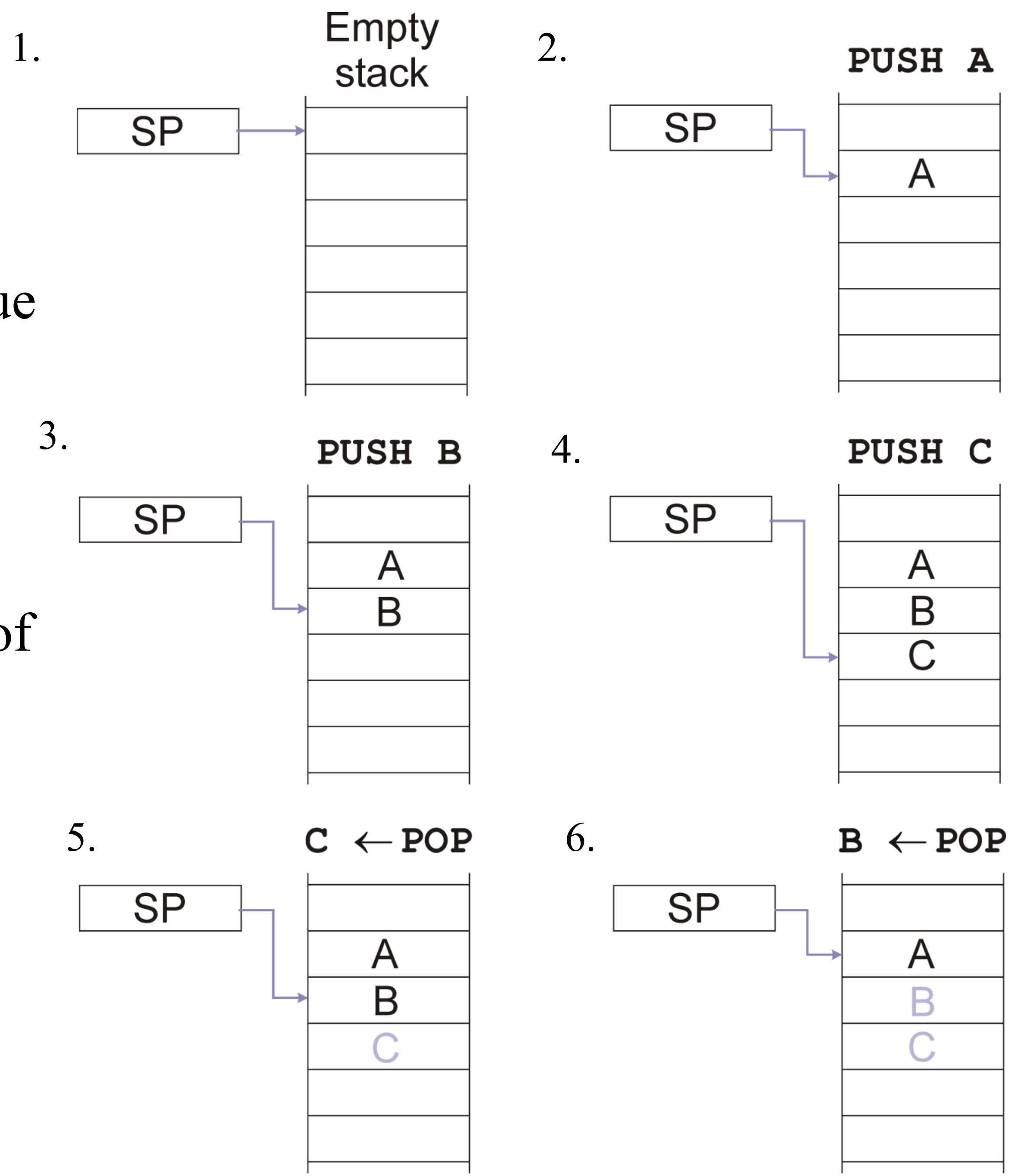
```
PUSH x  ≡      DEC  SP
               MOV (SP), x

POP x   ≡      MOV x, (SP)
               INC  SP
```

- In this case, SP must be initialised at the top end of memory reserved for the stack.

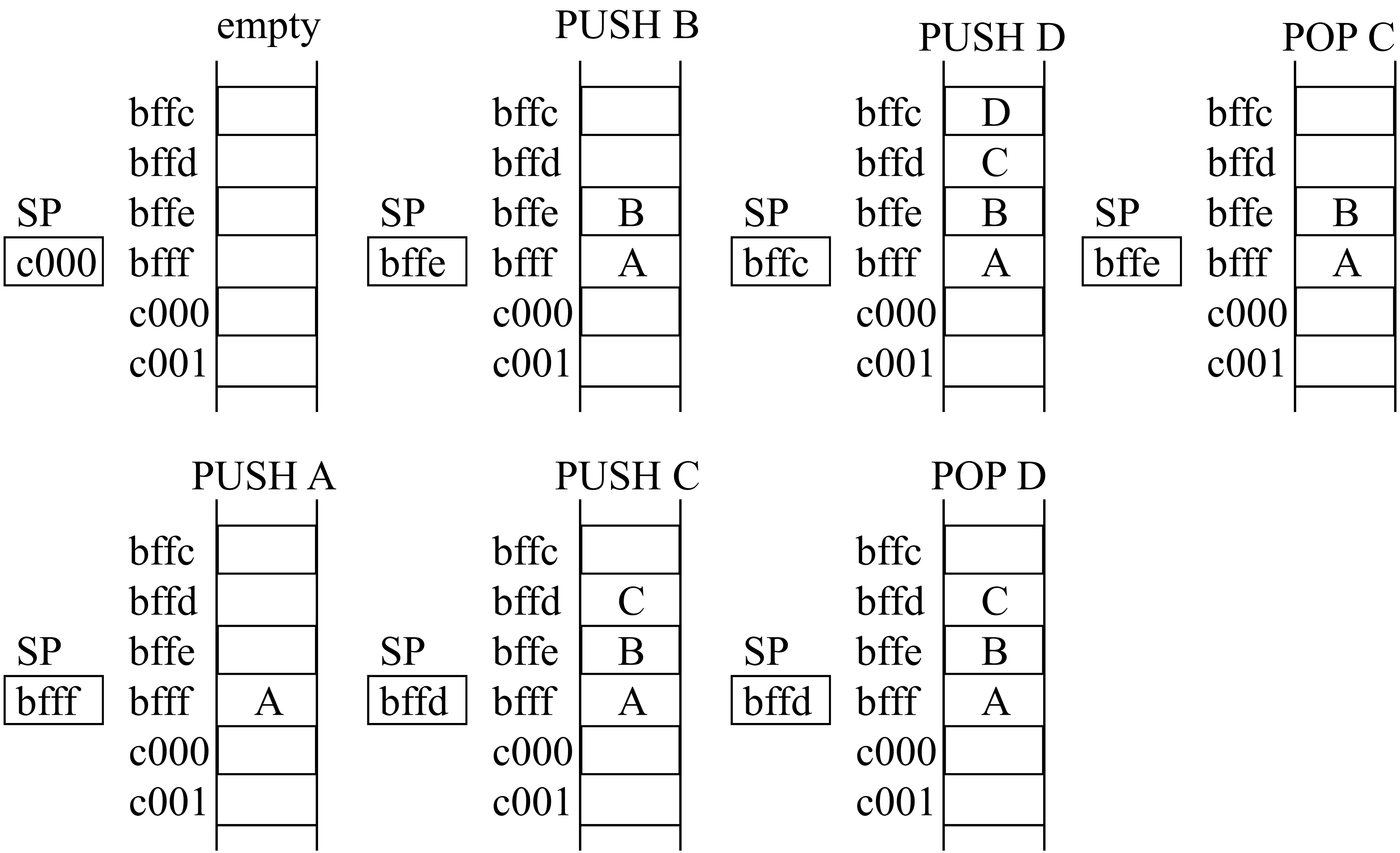
Note …..
The first location of the stack is unused due to the way we define PUSH and POP

The Stack Pointer SP contains the address of the Top of the Stack ToS

SP always points to the last item pushed onto the stack.



Sometimes memory maps may be drawn upside down to illustrate the Top of Stack notion.



CALL and RETURN

- Provide a basic mechanism for subroutine calling.
- Implemented using the stack.

The **CALL** instruction will implement the following:

PUSH the contents of the PC onto the stack
(this is the return address)
Load the start address of the subroutine into the PC
Implement other desired operations

The **RETURN** instruction will implement the following:

POP the return address from the stack to the PC
(returns to point in program at which the subroutine was called)
Implement other desired operations.

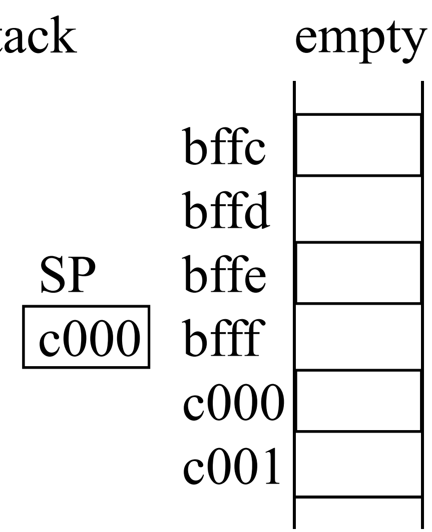
Passing Parameters

If a subroutine has parameters, these must be passed from the calling program to the procedure.

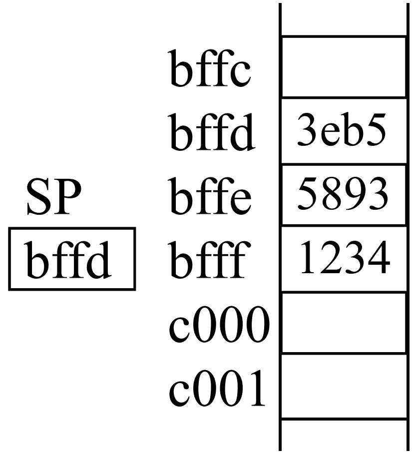
Passed in registers – fast but limited, registers are needed for many things

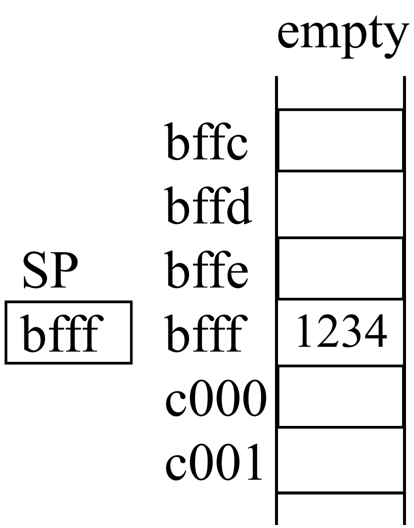
Pushed onto the stack – Before the CALL, push the parameters onto the stack.

Assume stack pointer
SP = c000

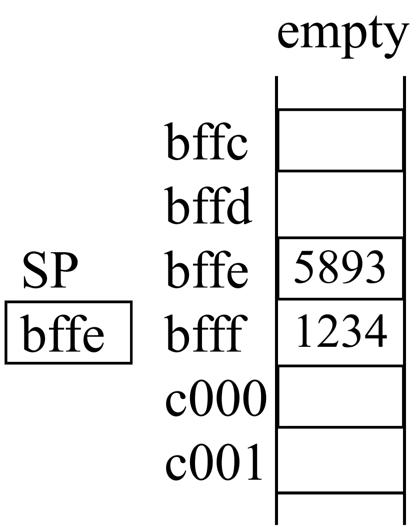



```
MOV  AX  1234h
PUSH AX
PUSH 5893h
PUSH 3eb5h
```

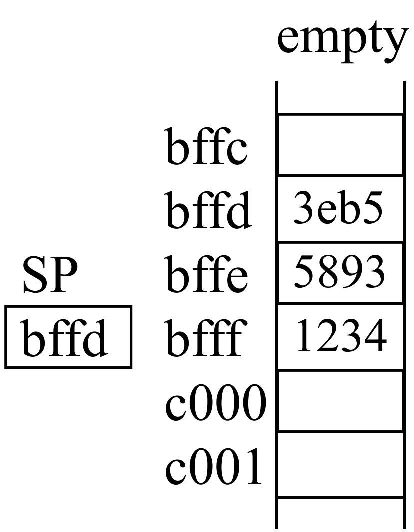




MOV AX 1234h
PUSH AX
PUSH 5893h
PUSH 3eb5h



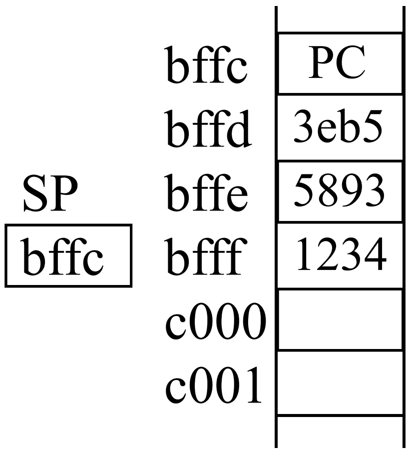
MOV AX 1234h
PUSH AX
PUSH 5893h
PUSH 3eb5h




```
MOV  AX  1234h
PUSH AX
PUSH 5893h
PUSH 3eb5h
```

Problem: We can't just POP the parameters off the stack for the subroutine to use.

Why? When you execute the CALL, it pushes the return address (PC) onto the stack.
If the called routine executes a POP it won't get a parameter, it will get the return address.

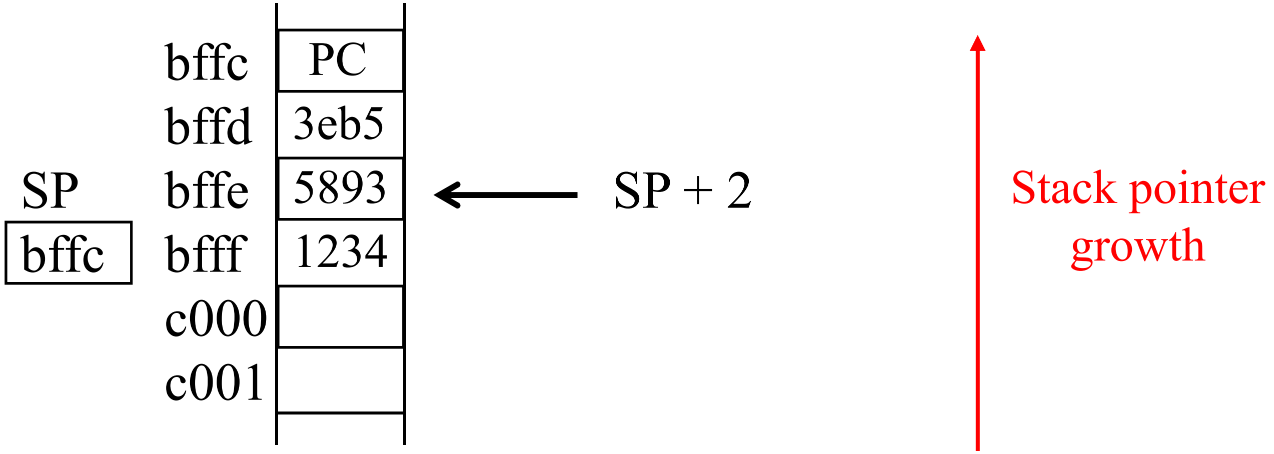


You could pop the PC off the stack and then get your data. However, if you forget to put PC back on top, you will never be able to return to the calling program!

Solution: Use indirect addressing based on the stack pointer register (SP). Parameters can then be used without removing them from the stack.

```
mov ax, [SP + 2]  ; take the address in SP
                  ; add 2
                  ; load ax with the contents
                  ; of the word at that address
```

We add because the stack grows down !



Consider the following example:

procedure

```
procedure ProcA(a,b,c: integer);
begin
end;
```

calling program

```
begin
      ProcA(3,4,5);
end;
```

```
ProcA:
      (SP+1) accesses c
      (SP+2) accesses b
      (SP+3) accesses a
      RET
main:
      PUSH  3
      PUSH  4
      PUSH  5
      CALL  ProcA
```

Parameters can be passed by value or reference

By Value: The actual value of the parameter is pushed onto the stack. The procedure can alter the value and it is discarded when the procedure returns.

By Reference: The address of the value is pushed onto the stack. Any modification of the value by the procedure persists after the procedure returns.

Return Values: Stack space must be allocated for any values to be returned. Values must be returned before the stack space is de-allocated.

Local Variables: Variables local to the subroutine must be allocatd space on the stack.

STACK FRAMES

For a general procedure call, the stack holds:

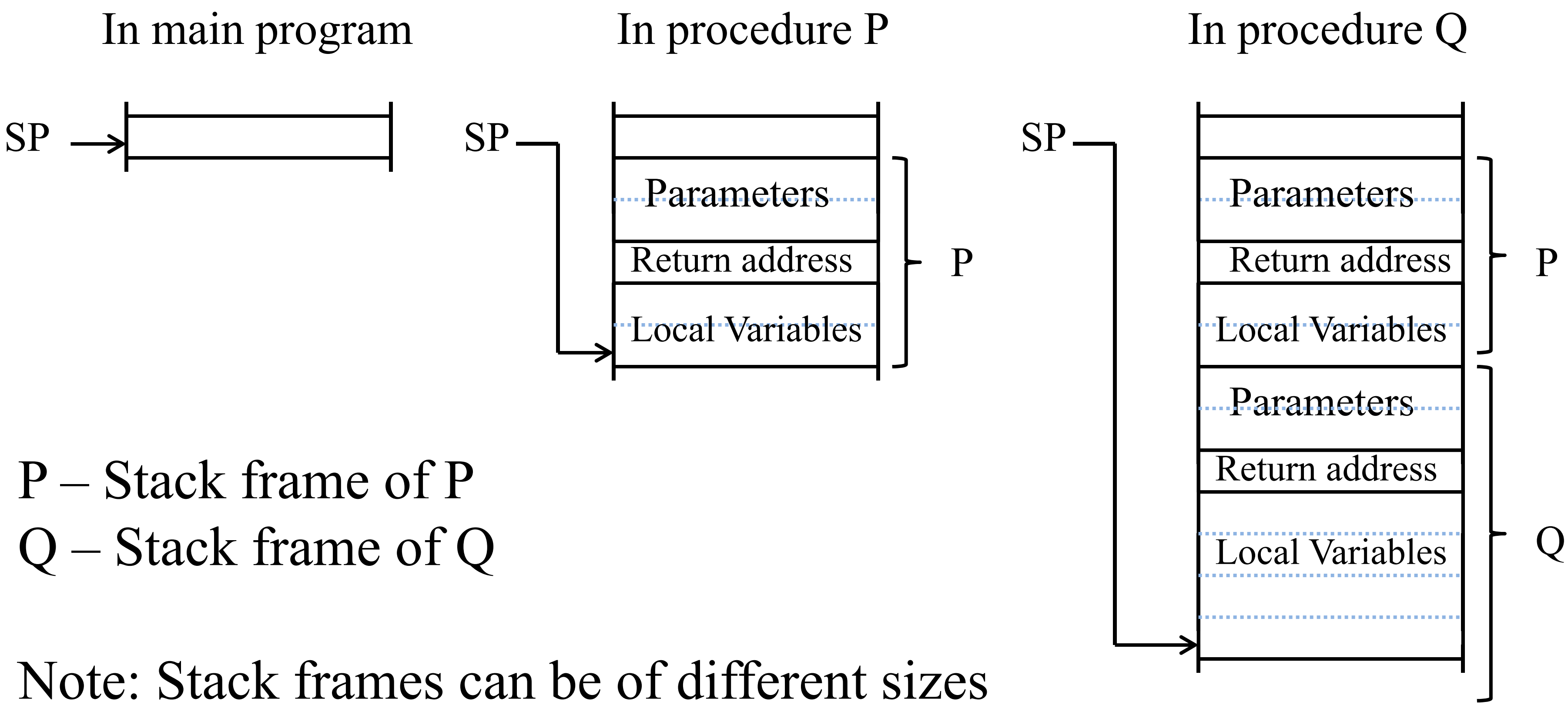
- The return address
- Passed parameters
- Local Variables
- Returned values

The block of space associated with a procedure is called the STACK FRAME.

The stack frame exists only as long as the procedure is being executed. When the procedure returns, the stack frame ceases to exist.

Procedure Calls within Other Procedure Calls

It is possible for a procedure to be called from within another procedure. Consider the stack that results from calling procedure Q from procedure P which is called by the main program.



P – Stack frame of P
Q – Stack frame of Q

Note: Stack frames can be of different sizes

Recursion

Recursion involves a procedure calling itself, e.g. factorial, n!

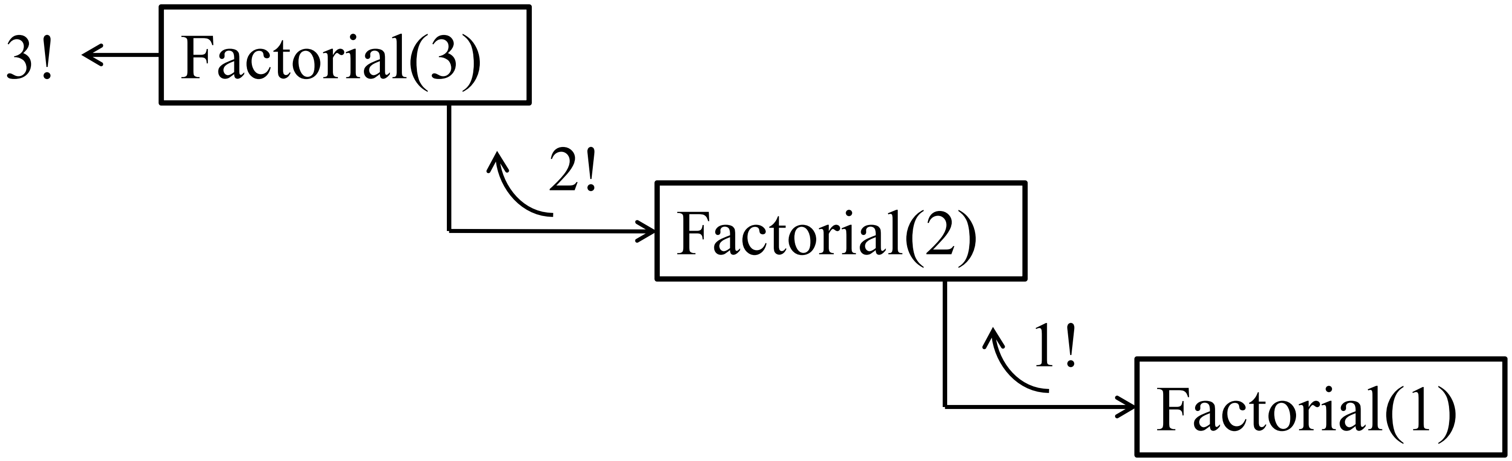
If n = 3,   3! = 3 * 2 * 1 = 6

```
int factorial(int n)
{
if (n = 1)
    return 1;
else
   return n * factorial (n-1);
}
```

Termination point for the recursive call – the test if n is unity.

The programmer must ensure that there is an 'exit point' from the chain of recursive calls after which the sequence of called procedures returns. If there is no termination point, a stack overflow error will occur when all of the available stack space has been used up. This can also happen if a sufficiently large stack is not allocated.

Call sequence for evaluating 3!



Recursive algorithms make extensive use of the stack – which is invariably implemented in external memory, thereby incurring a performance penalty. Unrolling recursive algorithms into simple loops typically reduces the number of accesses to external memory, thus improving execution speed.