

# Jump Instruction

- A vital part of any programming language is to be able to “jump” to another part of the program and continue executing from there.

**JMP ADDRESS**

- This 8086 instruction causes the IP Register to be loaded with “ADDRESS”, effectively causing the program to jump to there.
- We have seen the use of the jump instruction to cause an infinite loop on the computer

**0100:0106 JMP 0106**

- 0106 becomes the new point of execution of the program each time this instruction is encountered.

# More Jumps

- This type of jump is called an “unconditional jump”. This means that irrespective of the outcome of previous instructions or the state of the flags “**jmp always**” to the new point of execution.
- Complimentary instructions “**jmp sometimes**” are provided where some condition has occurred which we want to affect the control of the “jump”. These instructions are known as “**conditional jumps**”.

```
address:  ..  
          ..  
          dec ax  
          jne address  
          mov ax, cx
```

The first instruction simply subtracts 1 from **ax**. The jump then takes place if the ZF flag is 0. Otherwise the next instruction is fetched and executed.

# More Jumps

```
cmp ax,bx
```

```
ja address
```

- The `cmp` instruction subtracts `bx` from `ax`, and throws away the result(!)
- However it does **set the flags** according to the result.
  - `ja` is short for ‘**jump if above**’
  - `jg` is short for ‘**jump if greater**’

## What's the difference?

- `ja` will cause a jump if the **unsigned** number in `ax` is greater than the **unsigned** number in `bx`. It will cause a jump if **ZF** and **CF** are 0.
- `jg` does the same for **signed** numbers. It will jump if **ZF** is 0, and **SF=OF**
  - *The programmer rarely has to think about the state of the flags.*

# Relative Jumps

- All conditional jumps must have a destination 127 bytes forward, or 128 bytes backwards, from the address of the instruction following the jump.
- The address given in the jump is actually stored in machine code as a single 2's complement byte.

| <u>Address</u> | <u>Code</u> | <u>Assembly Language</u> |          |
|----------------|-------------|--------------------------|----------|
| 100            | 48          | fred:                    | dec ax   |
| 101 102        | 75 FD       |                          | jne fred |

**FD** is the 2's complement for **-3**

By coding the jumps as relative displacements in this way, a program can be made *relocatable*, that is the same program can be placed in any part of the memory map, and still run correctly.

To be *relocatable* a program must never refer to an actual physical program address.

# Loop

- To make longer jumps, use the unconditional **jmp** instruction, which can have a 16-bit displacement.
- A particularly common use of a conditional jump is

```
dec cx  
jne labelx
```

which decrements `cx`, and jumps if it is not equal to 0, to the given address.

- A single instruction **loop** is provided to do the same thing.

```
loop labelx
```

- The `loop` instruction does not, however, affect the flags in any way.

# Subroutines

A common programming requirement is the **subroutine**, a piece of code which will be used many times as the program executes. Rather than writing the same piece of code repeatedly, we simply create a subroutine that is coded once but can be called over and over again. (Java method)

To **jump to a subroutine** use the **call** instruction.

To **return from a subroutine**, use the **ret** instruction.


Consider the following subroutine, which causes a **delay** by “counting-down” **cx** to zero.

# Subroutine

## Machine Code

```
.          - (some code)
E8 00 0F          call delay
8B C3          addr1: mov ax,bx
.          - (some code)
.          - (some code)
E8 00 04          call delay
8B CA          addr2: mov cx,dx
.          - (some code)
stop: jmp stop
B9 FF FF          delay: mov cx,0FFFFh
E2 FE          again: loop again
C3              ret
```

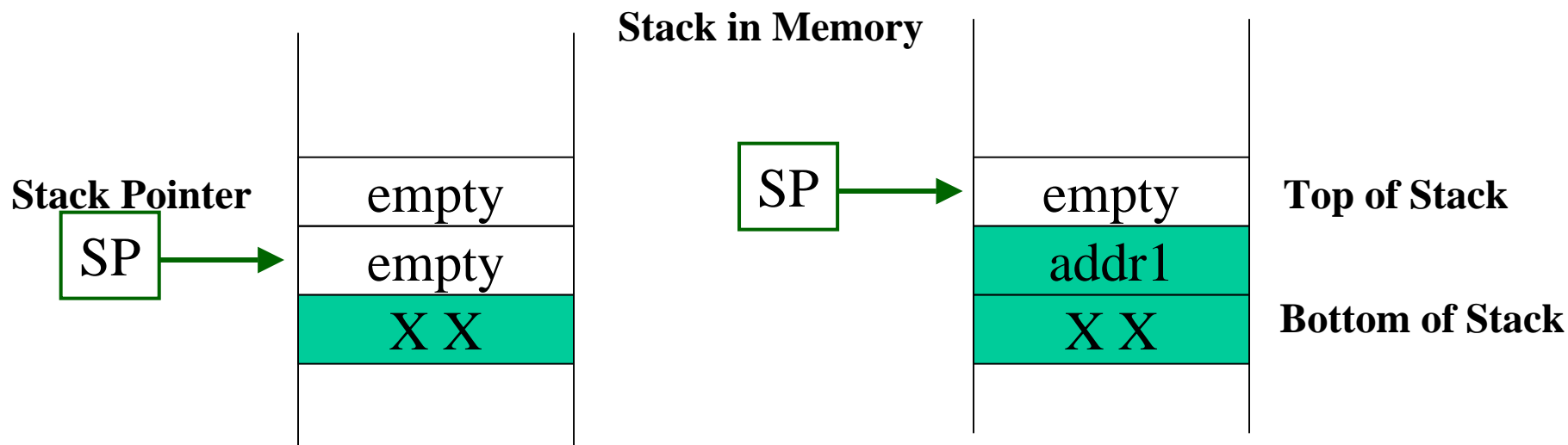
When the **call** instruction is executed, the address of **delay** is placed into the IP. The subroutine is then executed. When the **ret** instruction is executed the address of the next instruction after the **call** is placed in the IP, and execution continues from there.



# Stack

Q. How does the CPU know this address to return to?

A. The **call** instruction, before it jumps to the subroutine address, **pushes** the address of the next instruction onto **the stack**, from where the **ret** instruction can **retrieve** it.

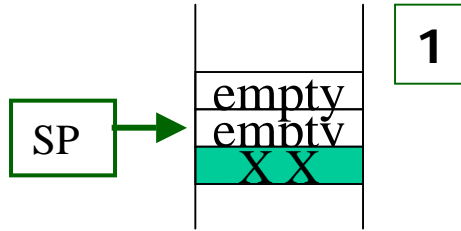


This use of the stack is very elegant. It works also for nested subroutines:

# Nested Subroutines on the Stack

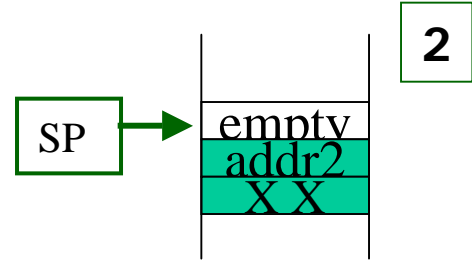
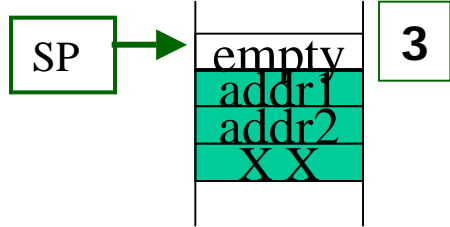
```

sub1: <do>
      <sub1>
      ret 4
  
```



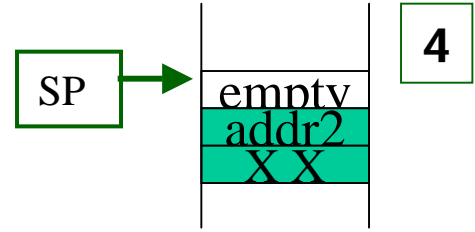
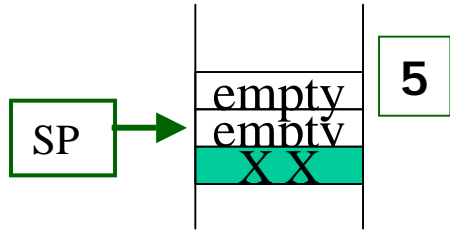
```

..
sub2: <do>
      <sub2>
      call sub1 3
addr1:ret 5
  
```



```

..
;main program
.. 1
.. 2
call sub2
addr2:mov ax,bx
..
  
```



# Other Stack Uses

The stack can also be used neatly to:

- (a) Pass parameters into a subroutine
- (b) Preserve register values

```
mov cx,0FFFFh
push cx
call delay
mov ax,bx

delay: pop cx
here:  loop here
      ret
```

Passing Parameter CX to Delay

```
mov ax,0FFFFh
push ax
call copybx
pop ax

copybx: mov ax,bx
        dec ax
        ret
```

Preserving AX which get destroyed in copybx

# The INT instruction

An instruction similar to call is `int` (software interrupt).

When an `int n` instruction is executed:

- The **Flags register**, **CS** and **IP** are pushed onto the stack.
- **IP** and **CS** are loaded with the contents of physical memory locations  $n*4$  and  $n*4+2$ .

A “**jump table**” must be set up at the bottom of the memory map, from 0 - 3FF. The `int` instruction is used to access special operating system subroutines.

# Some Useful Instructions

```
test ax,bx
```

This instruction logically AND's its two operands (**ax** and **bx** in this case), and sets the flags according to the outcome, and discards the results.

i.e. `test ax,8000` will **set the zero flag**  
( **ZF** ) **if** the number in **ax** is positive.

```
neg bx
```

This **negates** the value in **bx**. (i.e complements it and adds 1).

```
mov [bx],3
```

 What does this instruction do?

In cases of ambiguity, use *byte ptr* or *word ptr* in the instruction.

```
mov byte ptr [bx],3 OR mov word ptr [bx],3
```