PROGRAM EXECUTION AND TRANSFER INSTRUCTIONS

These instructions cause a branch in the program sequence.
There are 2 main types of branching:

i. Near branch
ii. Far Branch

i. Near Branch

This is an **Intra-Segment Branch** i.e. the branch is to a new location within the current segment only.
Thus, **only** the value of **IP needs to be changed**.
If the Near Branch is in the **range of −128 to 127**, then it is called as a **Short Branch**.

ii. Far Branch

This is an **Inter-Segment Branch** i.e. the branch is to a new location in a different segment.
Thus, the values of **CS and IP need to be changed**.

JMP (Unconditional Jump)

INTRA-Segment (NEAR) JUMP

The Jump address is specified in two ways:

1) INTRA-Segment Direct Jump

The new Branch location is specified directly in the instruction
The new address is calculated by **adding** the 8 or16-bit **displacement** to the IP.
The CS does not change.
A +ve displacement means that the Jump is ahead (forward) in the program.
A -ve displacement means that the Jump is behind (backward) in the program.
It is also called as *Relative Jump*.

```
Eg: JMP Prev        ; IP ← offset address of "Prev".
    JMP Next        ; IP ← offset address of "Next".
```

2) INTRA-Segment Indirect Jump

The New Branch address is specified indirectly through a **register** or a **memory location**.
The value in the IP is **replaced** with the new value.
The CS does not change.

```
Eg: JMP WORD PTR [BX]        ; IP ← {DS:[BX], DS: [BX+1]}
```

INTER-Segment (FAR) JUMP

The Jump address is specified in two ways:

3) INTER-Segment Direct Jump

The new Branch location is **specified directly** in the instruction
Both **CS and IP get new values**, as this is an inter-segment jump.

```
Eg: Assume NextSeg is a label pointing to an instruction in a different segment.
    JMP NextSeg      ; CS and IP get the value from the label NextSeg.
```

4) INTER-Segment Indirect Jump

The new Branch location is **specified indirectly** through a **register** or a **memory location.**
Both **CS and IP get new values**, as this is an inter-segment jump.

```
Eg:JMP DWORD PTR [BX]        ; IP ← {DS:[BX], DS: [BX+1]},
                             ; CS ← {DS:[BX+2], DS:[BX+3]}
```

JCondition (Conditional Jump)

This is a conditional branch instruction.
If condition is **TRUE**, then it is **similar to** an **INTRA-Segment Direct Jump.**
If condition is FALSE, then branch does not take place and the next sequential instruction is executed**.**
The destination must be in the range of –128 to 127 from the address of the instruction (i.e. **ONLY SHORT Jump**).
Eg: **JNC Next** ; *Jump to Next If Carry Flag is not set (CF = 0).*
The various conditional jump instructions are as follows:

Mnemonic	Description	Jump Condition
Common Operations		
JC	Carry	**CF = 1**
JNC	Not Carry	CF = 0
JE/JZ	Equal or Zero	**ZF = 1**
JNE/JNZ	Not Equal or Not Zero	ZF = 0
JP/JPE	Parity or Parity Even	**PF = 1**
JNP/JPO	Not Parity or Parity Odd	PF = 0
Signed Operations		
JO	Overflow	**OF = 1**
JNO	Not Overflow	OF = 0
JS	Sign	**SF = 1**
JNS	Not Sign	SF = 0
JL/JNGE	Less	**(SF Ex-Or OF) = 1**
JGE/JNL	Greater or Equal	(SF Ex-Or OF) = 0
JLE/JNG	Less or Equal	**((SF Ex-Or OF) + ZF) = 1**
JG/JNLE	Greater	((SF Ex-Or OF) + ZF) = 0
Unsigned Operations		
JB/JNAE	Below	**CF = 1**
JAE/JNB	Above or Equal	CF = 0
JBE/JNA	Below or Equal	**(CF Ex-Or ZF) = 1**
JA/JNBE	Above	(CF Ex-Or ZF) = 0

CALL (Unconditional CALL)

CALL is an instruction that transfers the program control to a sub-routine, with the intention of coming back to the main program.
Thus, in CALL 8086 **saves the address of the next instruction into the stack** before branching to the sub-routine.
At the end of the subroutine, control transfers back to the main program using the return address from the stack.
There are two types of CALL: Near CALL and Far CALL.

INTRA-Segment (NEAR) CALL

The **new subroutine** called must be **in the same segment** (hence intra-segment).
The CALL **address** can be **specified directly** in the instruction **OR indirectly** through Registers or Memory Locations.
The following sequence is executed for a NEAR CALL:

i. 8086 will **PUSH Current IP** into the Stack.
ii. **Decrement SP by 2**.
iii. **New value** loaded **into IP**.

iv. **Control transferred** to a subroutine within the same segment.

Eg: **CALL subAdd** *; {SS:[SP-1], SS:[SP-2]} ← IP, SP ← SP – 2,*
; IP ← New Offset Address of subAdd.

INTER-Segment (FAR) CALL

The **new subroutine** called is in **another segment** (hence inter-segment).
Here CS and IP both get new values.
The CALL address can be specified directly OR through Registers or Memory Locations.
The following sequence is executed for a Far CALL:

i. **PUSH CS** into the Stack.
ii. **Decrement SP** by 2.
iii. **PUSH IP** into the Stack.
iv. **Decrement SP** by 2.
v. **Load CS** with new segment address.
vi. **Load IP** with new offset address.
vii. **Control transferred** to a subroutine in the new segment.

Eg: **CALL subAdd** *; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,*
; {SS:[SP-1], SS:[SP-2]} ← CS, SP ← SP – 2,
; CS ← New Segment Address of subAdd,
; IP ← New Offset Address of subAdd.

There is **NO PROVISION for Conditional CALL**.

RET --- Return instruction

RET instruction causes the control to return to the main program from the subroutine.

Intrasegment-RET

Eg: RET ; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2
RET n ; IP ← SS:[SP], SS:[SP+1]
; SP ← SP + 2 + n

Intersegment-RET

Eg: RET ; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4
RET n ; IP ← SS:[SP], SS:[SP+1]
; CS ← SS:[SP+2], SS:[SP+3]
; SP ← SP + 4 + n

Please Note: The programmer writes the intra-seg and Inter-seg RET instructions in the same way. It is the assembler, which distinguishes between the two and puts the right opcode.
#Please refer Bharat Sir's Lecture Notes for this ...

Differentiate between

	JMP INSTRUCTION	CALL INSTRUCTION
1	JMP instruction is used to **jump to a new location** in the program and continue	Call instruction is used to **invoke a subroutine, execute it and then return** to the main program.
2	A jump simply **puts the branch address into IP**.	A call **first stores the return address into the stack** and then loads the branch address into IP.
3	In 8086 Jumps can be either **unconditional or conditional**.	In 8086, Calls are only **unconditional**.
4	Does **not use the stack**	**Uses the stack**
5	Does **not need a RET** instruction.	**Needs a RET** instruction to return back to main program.

Differentiate between

	PROCEDURE (FUNCTION)	MACRO
1	A procedure (Subroutine/ Function) is a set of instruction needed repeatedly by the program. It is **stored as a subroutine and invoked from several places by the main program.**	A Macro is similar to a procedure but is not invoked by the main program. Instead, the **Macro code is pasted into the main program wherever the macro name is written in the main program.**
2	A subroutine is **invoked by a CALL** instruction and control returns by a RET instruction.	A Macro is simply accessed by **writing its name**. The entire macro code is pasted at the location by the assembler.
3	**Reduces the size** of the program	**Increases the size** of the program
4	**Executes slower** as time is wasted to push and pop the return address in the stack.	**Executes faster** as return address is not needed to be stored into the stack, hence push and pop is not needed.
5	**Depends on the stack**	**Does not depend on the stack**

Type 1) **Iteration Control Instructions**

These instructions **cause** a series of **instructions to be executed repeatedly**.
The **number of iterations** is loaded **in CX** register.
CX is **decremented by 1**, after every iteration. Iterations occur **until CX = 0**.
The **maximum difference between** the **address** of the instruction and the address of the Jump **can be 127**.

1) LOOP Label
Jump to specified label if CX not equal to 0; and decrement CX.

```
Eg:      MOV CX, 40H
   BACK: MOV AL, BL
         ADD AL, BL
           .
           .
           .
         MOV BL, AL
         LOOP BACK      ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0.
```

2) LOOPE/LOOPZ Label (Loop on Equal / Loop on Zero)
Same as above except that looping occurs ONLY if Zero Flag is set (i.e. ZF = 1)

```
Eg:      MOV CX, 40H
   BACK: MOV AL, BL
         ADD AL, BL
           .
           .
           .
         MOV BL, AL
         LOOPZ BACK     ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 1.
```

3) LOOPNE/LOOPNZ Label (Loop on NOT Equal / Loop on NO Zero)
Same as above except that looping occurs ONLY if Zero Flag is reset (i.e. ZF = 0)

```
Eg:      MOV CX, 40H
   BACK: MOV AL, BL
         ADD AL, BL
           .
           .
           .
         MOV BL, AL
         LOOPZ BACK     ; Do CX ← CX – 1.
                        ; Go to BACK if CX not equal to 0 and ZF = 0.
```