What is JMP (Jump)?

The **JMP (Jump)** instruction is used to **change the flow of a program**.
Normally, instructions are executed **line by line**, one after another.
But when we use a **JMP**, the processor can **skip some lines** and **go to another part** of the program.

You can think of it like: "Go from here → directly to there."

Why use JMP?

- To **skip** some part of a program.
- To **repeat** or **loop** a part of code.
- To **branch** to another section depending on a condition.

Types of JMP Instructions

There are **two main categories** of JMP:
1️⃣ **Unconditional Jump** — always jumps.
2️⃣ **Conditional Jump** — jumps **only if** a certain condition is true (based on flag bits).

Unconditional JMP

This jump **always happens**, no matter what.
The processor just **goes to the new address** and continues from there.

There are **3 types** of unconditional JMP based on how far they can jump 👇

Type	Size	Range	Name	Meaning
1️⃣	2 bytes	+127 to -128 bytes	**Short Jump**	Jump to a nearby location
2️⃣	3 bytes	±32K bytes (in real mode)	**Near Jump**	Jump anywhere within current code segment
3️⃣	5 bytes	Anywhere in memory	**Far Jump**	Jump to another segment (different area in memory)

Short Jump

- Also called a **Relative Jump**.
- Used for **small jumps** within the same code area (up to 127 bytes forward or 128 bytes backward).
- The jump **distance** (not the full address) is stored.
- The CPU adds this distance to the **Instruction Pointer (IP)** to find where to go next.

👉 **Simple example:**

```
JMP SHORT NEXT
...
NEXT: MOV AX, BX
```

Here, the processor skips some lines and starts executing from the label `NEXT`.

💡 **Trick to remember:**
Short jump = small steps nearby

Near Jump

- Used for **larger jumps** within the **same code segment**.
- Can move up to **±32K bytes** in real mode.
- In **protected mode** (80386 and above), it can jump within **±2G bytes** (because 32-bit displacement).
- Adds a **16-bit (or 32-bit)** signed distance to IP.

👉 **Example:**

```
JMP NEAR TARGET
```

CPU adds displacement (like +0002H) to IP → goes to that location.

💡 **Trick to remember:**
Near jump = longer jump, same area

Far Jump

- Used to jump **outside the current code segment** (to another segment).
- Contains both **segment address (CS)** and **offset (IP)**.
- This allows jump to **anywhere in memory**.

💡 **Trick to remember:**
Far jump = jump to a faraway segment

Label in JMP

A **label** is just a **name for a memory address**.
When we write:

```
JMP START
```

The processor jumps to the instruction **marked with the label `START:`**

Example:

```
START: MOV AX, 1
JMP NEXT
NEXT:  MOV BX, AX
A colon (:) is always added after a label name when it is used
with JMP or CALL instructions.
```

Special Syntax

JMP $+2

It means "jump forward 2 bytes from the current instruction."
Here `$` means "current address".

Near Jump — (Relocatable Relative Jump)

A **Near Jump** is similar to the **Short Jump**, but it can go **further** — up to **±32K bytes** in real mode.

It's called **relocatable** because the **jump distance** (not the actual address) is stored.
So even if the **code segment moves** to a new memory location,
the distance between the jump instruction and the target instruction **remains the same**.
That's why the program **still works without any change!**

What does "Relocatable" mean?

Think of a relative jump like giving directions from your current position.Instead of saying "Go to house number 5234", you say "Go 20 steps forward".If your code moves somewhere else in memory, the "20 steps" is still valid, because it's relative to where you are, not an absolute address.

Why does it still work if the code moves?

- Because the jump doesn't depend on **where the code is in memory**, only **how far the target is from the jump instruction**.
    - Original location: jump instruction at `1000h`, target at `1050h`.Distance = `+50h`,Code moves to new location: jump at `2000h`, target at `2050h`.Distance = still `+50h` → CPU jumps correctly.

Why It's Important:

This feature makes Intel processors great for general computers —
because the same program can run from **any location in memory** (relocatable software).

⚙️ Example (Example 6–2):

```
0000 33DB        XOR BX,BX
0002 B8 0001     START: MOV AX,1
0005 03 C3       ADD AX,BX
0007 E9 0200 R   JMP NEXT
; (some memory gap)
0200 8B D8       NEXT: MOV BX,AX
0202 E9 0002 R   JMP START
```

Explanation:

- `JMP NEXT` jumps to the instruction labeled `NEXT:` which is **200H bytes ahead**.
- The **assembler** marks this jump as "**relocatable**" **(R)**.
- That means — if we move this code block to another part of memory, the jump still works correctly because the **distance (0200H)** stays the same.

Near Jump = relative, relocatable, longer step within same segment.

Far Jump — (Intersegment Jump)

A **Far Jump** is used to jump to a **completely different code segment**.
That means it changes **both:**

- **CS (Code Segment)**
- **IP (Instruction Pointer)**

So, it can go **anywhere in memory** — even outside the current segment.

Structure (5 bytes total):

Bytes	Meaning
1	Opcode (EAH)
2–3	New Offset (IP value)
4–5	New Segment (CS value)

Example (Example 6–3):

```
EXTRN UP:FAR

0000 33 DB           XOR BX,BX
0002 B8 0001         START: ADD AX,1
0005 E9 0200 R       JMP NEXT
; (skipped memory)
0200 8B D8           NEXT: MOV BX,AX
0202 EA 0002 ---- R JMP FAR PTR START
0207 EA 0000 ---- E JMP UP
```

- `JMP FAR PTR START` → performs a **far jump** inside same file (relocatable).

- `JMP UP` → jumps to a label **UP**, which is **external** (from another file).
    - `EXTRN UP:FAR` means the label is **external** and **far away** (different segment).
- The **linker** later fills the correct segment and offset addresses.
  Far Jump = Jump to another segment (new CS + IP).

Jump with Register Operand — (Indirect Jump)

Here, the **jump address** is not written directly.
Instead, it's stored inside a **register** (like AX, BX, or EAX).
So, the CPU **jumps to the address stored in that register.**

This is called an **Indirect Jump**.

How It Works:

- `JMP AX` → means: "Jump to the address that's inside AX."
- It directly puts the value of AX into **IP** (Instruction Pointer).
- It does **not add** to IP like relative jumps do.
- Used often with **jump tables** (for menu or key-based programs).

Example (Example 6–4):

```
.DATA
TABLE: DW ONE
       DW TWO
       DW THREE

.CODE
TOP:    MOV AH,1
        INT 21H
        SUB AL,31H       ; convert key '1','2','3' to 0,1,2
        JB TOP           ; if less than 1, go back
        CMP AL,2
        JA TOP           ; if more than 3, go back
        MOV AH,0
        ADD AX,AX        ; double (0→0,1→2,2→4)
        MOV SI,OFFSET TABLE
        ADD SI,AX
        MOV AX,[SI]      ; get address from TABLE
        JMP AX           ; jump to ONE/TWO/THREE

ONE:    MOV DL,'1'
        JMP BOT
```

```
TWO:    MOV DL,'2'
        JMP BOT
THREE:  MOV DL,'3'
BOT:    MOV AH,2
        INT 21H
```

Explanation (in easy terms):

- Program reads a key ('1', '2', or '3').
- According to the key pressed, a different address (ONE, TWO, or THREE) is chosen from the **jump table**.
- Then `JMP AX` goes to that address directly.
  indirect Jump = jump to address stored inside a register.

Type	Jump T...	Target Range	What Ch...	Example	Mnemonic Tip
Short Jump	Unconditional	±127 bytes	IP only	`JMP SHORT NEXT`	Small jump nearby
Near Jump	Unconditional	±32K bytes	IP only	`JMP NEXT`	Long jump, same segment
Far Jump	Unconditional	Anywhere	CS + IP	`JMP FAR PTR LABEL`	Jump to another segment
Register Jump	Indirect	Anywhere in same segment	IP = register value	`JMP AX`	Jump using register
Conditional Jump	Conditional	±127 bytes	IP if condition	`JZ LABEL`	Jump only if flag true

Indirect Jumps Using an Index

An **indirect jump** means — instead of jumping directly to a fixed address,

👉 the processor **reads an address from memory (or register)** and then jumps there.

So, the CPU doesn't know *where* it will jump until it looks inside memory/register.
That's why it's called *indirect* (not direct).

`JMP AX`

➡️ means jump to the address stored inside AX.
If AX = 2050H → CPU will jump to address 2050H.

Now, what's Indirect Jump Using an Index?

This means we'll use **a table of addresses** (called a **jump table**) and an **index (like SI)** to select which address to jump to.

So basically:

- You press 1, 2, or 3 from the keyboard.

- Each number corresponds to a **different address** inside the jump table.
- The CPU looks at the correct address using **SI (index register)** and jumps there.

Example 6–5 (in easy explanation)

```
TABLE: DW ONE
       DW TWO
       DW THREE
```

👉 This is the **jump table**.
It stores **addresses of labels** (ONE, TWO, THREE).

1️⃣ Read a key from the keyboard → store it in AL.
2️⃣ Subtract 31H (ASCII of '1') → so key '1' becomes 0, '2' becomes 1, etc.
3️⃣ If key < 1 or key > 3 → go back and read again.
4️⃣ Multiply the result by 2 → because each address in the table is 2 bytes long.
5️⃣ Move that result to SI (index register).
6️⃣ Then this line:

```
JMP TABLE[SI]
```

➡️ means "jump to the address inside the table pointed by SI".

So, if you pressed **1**, SI = 0 → jumps to `TABLE[0]` → label `ONE`
If you pressed **2**, SI = 2 → jumps to `TABLE[2]` → label `TWO`
If you pressed **3**, SI = 4 → jumps to `TABLE[4]` → label `THREE`

Result:

It jumps to ONE, TWO, or THREE → displays 1, 2, or 3 on screen.

Conditional Jumps and Conditional Sets

A **conditional jump** happens **only if a certain condition is true**.
If not true → program just continues normally.

Example:

```
JC LABEL
```

➡️ Jump to LABEL **if Carry Flag (C) = 1**.

Flags used in Conditional Jumps:

Flag	Meaning
Z (Zero)	Result = 0

C (Carry)	Carry out occurred
S (Sign)	Result is negative
O (Overflow)	Overflow occurred
P (Parity)	Even number of 1s in result

```
CMP AL, BL    ; compare AL and BL
JE EQUAL      ; jump if AL = BL (Z=1)
```

➡️ If both are equal → jump to EQUAL.
Else → continue next instruction.

Jump Range:

- In 8086 → only **short jump** (within -128 to +127 bytes).
- In 80386 and later → can jump **farther** (up to ±32K or ±2G in 64-bit).

Common Conditional Jumps (Simple meanings)

Instruction	Meaning	Condition
JE / JZ	Jump if equal / zero	Z = 1
JNE / JNZ	Jump if not equal	Z = 0
JC	Jump if carry	C = 1
JNC	Jump if no carry	C = 0
JS	Jump if sign (negative)	S = 1
JNS	Jump if no sign (positive)	S = 0
JO	Jump if overflow	O = 1
JNO	Jump if no overflow	O = 0
JP / JPE	Jump if parity even	P = 1
JNP / JPO	Jump if parity odd	P = 0
JCXZ	Jump if CX = 0	CX = 0

For Number Comparison

Type	Use these jumps
Signed numbers	JG, JL, JGE, JLE
Unsigned numbers	JA, JB, JAE, JBE

🧩 “Above / Below” → for **unsigned**
🧩 “Greater / Less” → for **signed**

Example 6–6 (Simple Meaning)

Program searches through a table of 100 bytes to find the value **0AH**.

1️⃣ CX = 100 → number of bytes to check.

2️⃣ AL = 0AH → value we are searching.

3️⃣ `REPNE SCASB` → searches byte by byte.

4️⃣ `JCXZ NOT_FOUND` → if CX = 0 → means searched all 100 but didn't find → jump to NOT_FOUND.

Conditional Set Instructions (SETcc)

These are like **conditional jumps**,
but instead of *jumping*, they **set a byte** (memory or register) to **01H or 00H** depending on the flag condition.

If the condition is true → **set = 01H**
If the condition is false → **set = 00H**

So they "store" the result of a condition into a variable —
which can be **checked later** in the program.

`SETNC MEM`

👉 If Carry flag = 0 (no carry) → MEM = 01H

👉 If Carry flag = 1 (carry happened) → MEM = 00H

This is useful when you want to **remember** a condition result for later use.

Conditional set = **Test a flag → Save result as 1 or 0 → Use it later**

Instruction	Condition	Meaning
SETA	Z=0 and C=0	Set if Above (unsigned >)
SETAE	C=0	Set if Above or Equal
SETB	C=1	Set if Below
SETBE	Z=1 or C=1	Set if Below or Equal
SETC	C=1	Set if Carry
SETE / SETZ	Z=1	Set if Equal / Zero
SETNE / SETNZ	Z=0	Set if Not Equal / Not Zero
SETG	Z=0 and S=O	Set if Greater (signed >)
SETGE	S=O	Set if Greater or Equal
SETL	S≠O	Set if Less (signed <)
SETLE	Z=1 or S≠O	Set if Less or Equal

SETNC	C=0	Set if No Carry
SETO	O=1	Set if Overflow
SETNO	O=0	Set if No Overflow
SETS	S=1	Set if Sign (negative)
SETNS	S=0	Set if No Sign (positive)
SETP / SETPE	P=1	Set if Parity Even
SETNP / SETPO	P=0	Set if Parity Odd

Imagine a program checks if addition created a carry.

- If carry = 0 → memory mark it as **"No Carry" (01H)**
- Later, you can check this memory value to know that carry was clear at that point.

So, **SET instructions record condition outcomes** for later use.

LOOP Instruction

`LOOP` = **Repeat a block of code** a specific number of times.
It automatically uses **CX** (or ECX/RCX) as the counter.

1️⃣ Decrease CX by 1
2️⃣ If CX ≠ 0 → jump to label
3️⃣ If CX = 0 → exit loop

So, it's like:

```
CX = 100
L1:
   ; code
   LOOP L1
```

➡️ This runs the block **100 times**

example(from book):

Adds data from two memory blocks and stores results back.

```
MOV CX, 100         ; loop 100 times
MOV SI, OFFSET BLOCK1
MOV DI, OFFSET BLOCK2
L1: LODSW           ; load data from BLOCK1
    ADD AX, ES:[DI]; add data from BLOCK2
    STOSW           ; store result
    LOOP L1         ; repeat until CX=0
```

Each loop → adds one pair of numbers
After 100 loops → done.

Mode	Counter Used
16-bit	CX
32-bit	ECX
64-bit	RCX

Conditional LOOPs

These are special forms of LOOP that depend on both **CX** and a **flag condition**.

LOOPE (Loop While Equal)

Also called **LOOPZ**

Keep looping if CX ≠ 0 **AND** Zero Flag = 1 (means equal condition is true)

Exit loop when ZF=0 or CX=0.

LOOPNE (Loop While Not Equal)

Also called **LOOPNZ**

Keep looping if CX ≠ 0 **AND** Zero Flag = 0 (means not equal)

Exit loop when ZF=1 or CX=0.

Instruction	Loops while	Stops when
LOOP	CX ≠ 0	CX = 0
LOOPE / LOOPZ	CX ≠ 0 **and** ZF = 1	CX = 0 or ZF = 0
LOOPNE / LOOPNZ	CX ≠ 0 **and** ZF = 0	CX = 0 or ZF = 1

Imagine checking a list until you find a mismatch:

```
MOV CX, 10
COMPARE_LOOP:
    CMP AL, BL
    LOOPE COMPARE_LOOP
```

➡️ It will **keep looping while they are equal** (ZF=1).

➡️ It **stops** as soon as a difference is found (ZF=0).

LOOPE / LOOPZ (Loop While Equal)

Imagine you are checking candies in a box. You want to pick all the candies that are red.

- You pick one candy at a time.

- **If it's red (equal to what you want) → keep picking**
- **Stop when it's not red (not equal) or box is empty**

This is exactly like `LOOPE` — loop **while equal**.

LOOPNE / LOOPNZ (Loop While Not Equal)

Imagine you are walking through a queue looking for your friend.

- You ask each person: "Are you my friend?"
- **If they are NOT your friend → keep moving**
- **Stop when you find your friend or the queue ends**

This is exactly like `LOOPNE` — loop **while not equal**.

- **LOOPE = loop while "condition is met / equal"**
- **LOOPNE = loop while "condition not met / not equal"**

What are Program Control Instructions?

These are instructions that **control the flow** of the program —
👉 deciding **which part of the code will run** depending on some condition.
If `AL` contains a lowercase letter → convert to uppercase
Otherwise → do something else.

Two ways to control flow

Method	Description
(a) Using `.IF`, `.ELSE`, `.ENDIF`	Easy, readable, like C language (works only in **MASM 6.xx**)
(b) Using `CMP` + conditional jump (like `JE`, `JB`, `JA`, etc.)	Harder, longer, but works in all assemblers

The Dot Commands (MASM 6.xx Only)

Command	Meaning
`.IF`	Starts a condition check
`.ELSEIF`	Another condition if first one is false
`.ELSE`	Runs when none of the above is true
`.ENDIF`	Ends the if-block
`.WHILE ... .ENDW`	Loop that runs while condition is true
`.REPEAT ... .UNTIL`	Loop that repeats until condition is true

These commands **start with a dot (.)**, and only work in **MASM 6.xx**,
not in **old MASM 5.10** or **Visual C++ inline assembler**.

Example 6–8(a) — Using `.IF` (Easy way)

```
.IF AL >= 'A' && AL <= 'F'
    SUB AL,7
.ENDIF
SUB AL,30H
```

- If the value in `AL` is between `'A'` and `'F'`,
  → subtract 7 from AL

- Then subtract `30H` (to get hexadecimal number)

```
if (AL >= 'A' && AL <= 'F')
    AL = AL - 7;
AL = AL - 0x30;
```

Example 6–8(b) — Without `.IF` (Harder way)

```
cmp al,41h   ; compare AL with 'A'
jb Later     ; jump if below
cmp al,46h   ; compare AL with 'F'
ja Later     ; jump if above
sub al,7
Later:
sub al,30h
```

Same logic as (a), but using **CMP** and **conditional jumps** (`JB`, `JA`).

So `.IF` just makes it **shorter and easier** to read.

Table of Relational Operators in `.IF`

Operator	Meaning	Example
	Equal	`.IF AL == 'A'`
!=	Not equal	`.IF AL != 'Z'`
>	Greater than	`.IF AL > 10`
<	Less than	`.IF AL < 10`
>=	Greater or equal	`.IF AL >= 10`
<=	Less or equal	`.IF AL <= 10`
&&	Logical AND	`.IF AL >= 'A' && AL <= 'Z'`
||	Logical OR	
!	Logical NOT	`.IF !condition`

Example 6–9 — Convert lowercase to uppercase

```
.IF AL >= 'a' && AL <= 'z'
    SUB AL,20H
.ENDIF
```

- If AL contains a lowercase letter (between `a` to `z`)
  → Subtract 20H to make it uppercase (because ASCII difference between lowercase and uppercase is 20H)
  'a' = 61H → `61H - 20H = 41H` ('A')

Example 6–10(a) — Convert key to hexadecimal

```
.IF AL >= 'a' && AL <= 'f'
    SUB AL,57H
.ELSEIF AL >= 'A' && AL <= 'F'
    SUB AL,37H
.ELSE
    SUB AL,30H
.ENDIF
```

- If key is `'a'–'f'` → subtract 57H
- Else if `'A'–'F'` → subtract 37H
- Otherwise → subtract 30H

This converts an ASCII letter or digit into its **hexadecimal value**.

Same thing in C++ (Example 6–10(b))

```
if (temp >= 'a' && temp <= 'f')
    temp -= 0x57;
else if (temp >= 'A' && temp <= 'F')
    temp -= 0x37;
else
    temp -= 0x30;
```

You can see that `.IF`, `.ELSEIF`, `.ENDIF` in assembly work **just like** `if`, `else if`, `else` in C++.

- **Lowercase `'a'..'f'`**:
  - `'a'` = 0x61 → `0x61 - 0x57 = 0x0A` (decimal 10)
  - `'b'..'f'` → `0x0B..0x0F` → numeric 11–15
- **Uppercase `'A'..'F'`**:
  - `'A'` = 0x41 → `0x41 - 0x37 = 0x0A` (decimal 10)

- `'B'..'F'` → `0x0B..0x0F` → numeric 11–15

Both convert ASCII hex letters to numeric values 10–15 using simple hex subtraction.

The `.WHILE` Loop

```
.WHILE condition
    (instructions)
.ENDW
```

```
while (condition) {
    instructions;
}
```

Example 6–11:

```
.WHILE AL != 0DH     ; loop until Enter key is pressed (0DH =
Enter)
    MOV AH,1          ; read key from keyboard
    INT 21H
    STOSB             ; store the key into memory (buffer)
.ENDW
```

1. The program keeps reading keys from the keyboard one by one.
2. Each key (letter or symbol) is stored in an array called **BUF**.
3. When you press **Enter (ASCII code 0DH)** → the loop stops.
4. After that, a `$` sign is added at the end of the string (for DOS display).
5. Finally, it shows the string you typed.

- `.WHILE` starts the loop.
- `.ENDW` ends the loop.
- It checks the condition **before** running the loop — same as C's `while`.
- If you want to **stop early**, you can use `.BREAK`.
- To **skip to next iteration**, use `.CONTINUE`.

```
.WHILE 1
    MOV AH,1
    INT 21H
    .BREAK .IF AL == 0DH   ; stop if Enter key
    .CONTINUE .IF AL == 15 ; skip rest if AL = 15
.ENDW
```

```
while (1) {
    read_key();
    if (AL == 0x0D) break;
    if (AL == 15) continue;
}
```

The `.REPEAT — .UNTIL` Loop

```
.REPEAT
    (instructions)
.UNTIL condition
```

➡️ Works like C's:

```
do {
    instructions;
} while (!condition);
```

It runs **at least once**, then checks the condition at the end.

Example 6–12:

```
.REPEAT
    MOV AH,1
    INT 21H
    STOSB
.UNTIL AL == 0DH
```

1. Reads a character from keyboard.
2. Stores it in memory (BUF).
3. Keeps repeating until Enter (0DH) is pressed.
4. Then appends `$` and displays the full typed line.

Difference from `.WHILE`:

- `.WHILE` checks **before** loop starts.
- `.REPEAT—.UNTIL` checks **after** loop ends once.

So `.REPEAT` is easier when you want the loop to **run at least one time**.

The `.UNTILCXZ` Loop

This is a **special repeat loop** that runs **a fixed number of times** using the **CX register** as a counter.

```
MOV CX, count
.REPEAT
    (instructions)
```

`.UNTILCXZ`
It keeps looping until `CX = 0`.
(Each loop automatically decreases CX by 1.)

Example 6–13:

```
MOV CX,100              ; loop 100 times
MOV DI,OFFSET THREE     ; target array
MOV SI,OFFSET ONE       ; source array 1
MOV BX,OFFSET TWO       ; source array 2

.REPEAT
    LODSB               ; load byte from ONE → AL
    ADD AL,[BX]         ; add byte from TWO
    STOSB               ; store result in THREE
    INC BX              ; move to next element in TWO
.UNTILCXZ
```

- There are three arrays: ONE, TWO, THREE.
- Each has 100 bytes.
- The program adds ONE[i] + TWO[i] and stores it in THREE[i].
- `.UNTILCXZ` repeats the loop 100 times until CX = 0.

It works like:

```
for (i = 0; i < 100; i++) {
    THREE[i] = ONE[i] + TWO[i];
}
```

Quick Comparison Table

Type	Checks Condition	Runs At Least Once?	Uses CX Counter?	Similar to (C/C++)
`.WHILE ... .END`	Before loop	❌	❌	`while()`
`.REPEAT ... .UNTIL`	After loop	✅	❌	`do...while()`
`.UNTILCXZ`	Uses CX	✅ (if CX>0)	✅	`for` loop with counter

What is a Procedure?

A **procedure** (also called *subroutine*, *method*, or *function*) is a **small block of code** that performs **one specific task**.
You can **call** it whenever needed — instead of writing the same code again and again.So it saves memory and makes your program neat and reusable.

How Procedures Work

When a program uses a procedure:

1. The **main program** jumps to the **procedure** using a **CALL** instruction.
2. The **procedure runs** and performs its task.
3. When it's done, the **RET (Return)** instruction sends control **back** to where it was called from.

CALL instruction

- Used to **jump to** a procedure.
- Before jumping, it **stores the return address** (the address of the next instruction after CALL) on the **stack**.
- This is how the CPU knows where to come back after finishing the procedure.

RET instruction

- Used to **return** from a procedure.
- It **takes the saved address** from the stack and **goes back** to the instruction that follows CALL.

Why Use the Stack?

Because the stack **remembers** where the program came from.
Each time you CALL a procedure:

- The **return address** is **pushed** (saved) onto the stack.
- RET later **pops** (removes) that address to go back.

A procedure always starts and ends like this:

```
SUMS PROC NEAR
   ADD AX, BX
   ADD AX, CX
   ADD AX, DX
   RET
SUMS ENDP
```

PROC → starts procedure
ENDP → ends procedure

Types of Procedures

There are **two types** of procedures depending on memory location:

Near Procedure (Local)

- Used **within the same code segment.**

- **CALL** and **RET** work with **16-bit** address (IP only).
- Saves **less data** on the stack → **faster.**
- Example opcode: **C3H** (RET)

Think of it as: *calling your friend in the same building*.

Far Procedure (Global)

- Used when the procedure is in **another segment** (anywhere in memory).
- CALL saves **both CS (Code Segment)** and **IP (Instruction Pointer)** on the stack.
- RET takes both back from the stack.
- Example opcode: **CBH** (RET)

Think of it as: *calling your friend in another building*.

Type	What gets stored on Stack	RET Opcode	Can call from anywhere?
Near CALL	Only IP (16-bit)	C3H	❌ Same segment only
Far CALL	IP + CS (32-bit)	CBH	✅ Any segment

Step-by-Step (Near CALL Example)

Suppose your program is running at address **1003H**, and it CALLs a procedure.

1. CALL saves **return address (1005H) → Stack**
2. Then program jumps to **procedure address**
3. Procedure executes
4. RET pops **1005H** from stack and goes back

Stack before CALL

```
SP = FFFF
```

Stack after CALL

```
SP = FFFD  (because return address 1005H pushed)
```

Then RET will remove (pop) it and return to 1005H.

SP মানে Stack Pointer,
এটা এমন একটা রেজিস্টার যেটা স্ট্যাকের টপ (সর্বশেষ ব্যবহার হওয়া জায়গা) নির্দেশ করে।

স্ট্যাক নিচের দিকে বাড়ে — মানে নতুন কিছু **push** করলে SP এর মান **কমে যায়**।**SP = FFFFh ; স্ট্যাক একদম খালি**

এবং CPU নিচের ইনস্ট্রাকশনটা চালাতে যাচ্ছে:

`CALL 1005h`
এর মানে হলো — **একটা সাবরুটিনে (procedure)** চলে যাও,
কিন্তু **ফিরে আসার ঠিকানা (1005h)** টা মনে রেখে দাও।

`CALL` ইনস্ট্রাকশন **return address (1005h)** টা **স্ট্যাকে push করে দেয়**। যেহেতু ১টা word = ২ byte, তাই,push করলে SP ২ কমে যায়।

`SP = FFFFh → FFFD h`

Memory Address	Data Stored
FFFD	05h (low byte)
FFFE	10h (high byte)

অর্থাৎ 1005h (return address) এখন স্ট্যাকে সংরক্ষিত।

সাবরুটিন শেষ হলে যখন CPU `RET` চালাবে, তখন:

1️⃣ স্ট্যাক থেকে ২ বাইট (05h + 10h) **pop** করবে।

2️⃣ সেগুলো একত্রে করে **1005h** বানাবে।

3️⃣ এই 1005h টাকে **IP (Instruction Pointer)** এ লোড করবে। SP আবার ২ বাড়বে → **SP = FFFD + 2 = FFFFh**।

ধাপ	কাজ	SP মান	
CALL এর আগে	SP = FFFFh	স্ট্যাক খালি	
CALL করলে	SP = FFFD h	1005h স্ট্যাকে রাখা হয়	
RET করলে	SP = FFFFh	1005h পড়ে ফিরে আসে	

1005h হলো **ফিরে আসার ঠিকানা (Return Address)**
FFFD হলো **স্ট্যাক পয়েন্টারের নতুন মান** (যখন সেটা স্ট্যাকে সংরক্ষণ করা হয়)**1005h স্ট্যাকে রাখা হয়**, তাই SP নিচের দিকে **কমে গিয়ে FFFD হয়**।

In 64-bit mode

- A far CALL/RET saves or retrieves **8 bytes** (because addresses are larger).

“CALL goes → RET comes back.”
CALL saves the return address → RET brings you back.
Near = same segment, Far = different segment.

CALL with Register Operand

Sometimes, instead of giving a **fixed address** in the CALL instruction,
we can give the **address stored in a register.**

Example:CALL BX

1. The **CALL instruction** first **pushes** the current **IP (Instruction Pointer)** onto the **stack**. (This saves the return address.)
2. Then it **jumps** to the address stored inside the **BX register**.
3. The program starts running from that new address (the procedure).

- Only **offset (16-bit address)** is stored in the register.
- You can use any 16-bit register (like BX, DX, SI, DI, etc.)
- But **not segment registers** (CS, DS, etc.)

Why we can't use segment registers (CS, DS, etc.) in `CALL BX`:

1. Segment registers don't hold *offset addresses* .The **segment registers (CS, DS, SS, ES)** hold **segment base addresses**, not the **offset (actual instruction position)**.

- Example:`CS` = Code Segment (e.g., 2000H) ,`IP` = Instruction Pointer (e.g., 0100H) ,Actual address = CS:IP = 2000:0100

So, `CS` alone doesn't point to an exact instruction — it just marks a *block (segment)*.

2. CALL needs an offset, not a segment

- The `CALL BX` form is an **intra-segment call** → it only jumps within the same code segment.
- That means it only needs the **offset address** (like what BX, DX, SI, or DI can hold).
- Segment registers don't store such offsets — they store the *base* of a segment.

3. Changing segments requires a far CALL

- If you want to call a procedure in another segment, you must use a **FAR CALL**, which specifies **both segment and offset** (`CALL FAR PTR address`), not just a segment register like CS.
  You can't use `CALL CS` or `CALL DS` because segment registers don't contain the actual instruction location (offset); only general-purpose registers can hold that.

```
MOV BX, OFFSET DISP   ; store procedure address in BX
CALL BX               ; call procedure at address in BX
```

This means → the program will go to the procedure named **DISP** (its address is in BX).

The **DISP procedure** shows a character on screen:

```
DISP PROC NEAR
   MOV AH, 2     ; DOS function to display a character
   INT 21H
```

```
   RET
DISP ENDP
CALL BX = “Go to the address that BX is pointing to.”
```

CALL with Indirect Memory Address

Sometimes we want to **choose** which procedure to call **based on a value** (like 0, 1, or 2).
So, we can **store procedure addresses in a table** (called a *lookup table*) and CALL one based on that value.

```
TABLE DW ZERO      ; address of procedure ZERO
      DW ONE       ; address of procedure ONE
      DW TWO       ; address of procedure TWO

CALL TABLE[2*EBX]
```

If `EBX = 1`,
then `CALL TABLE[2*EBX]` = `CALL TABLE[2]` → calls the **ONE** procedure.
If `EBX = 2`, → calls the **TWO** procedure.

Each address is 2 bytes long, so we multiply EBX by 2.

First, understand what `TABLE` really is:

```
TABLE DW ZERO      ; 0th entry → address of ZERO
      DW ONE       ; 1st entry → address of ONE
      DW TWO       ; 2nd entry → address of TWO
```

Each **DW (Define Word)** stores **2 bytes** (the *address* of each procedure).
So memory looks like this:

Address (Table Index)	Contains (Procedure Address)
TABLE + 0	ZERO
TABLE + 2	ONE
TABLE + 4	TWO

Now when we write:CALL TABLE[2*EBX]

It means:“Go to the address stored in the table entry at offset `2*EBX` from TABLE.”

EBX	2*EBX	Memory position	Called procedure
0	0	TABLE + 0	ZERO
1	2	TABLE + 2	ONE
2	4	TABLE + 4	TWO

So, when EBX = 2,
`CALL TABLE[2*EBX] = CALL TABLE[4]`

→ **It calls the procedure TWO**, not "call 4" —
because **TABLE[4]** means *the address stored 4 bytes after TABLE*,
and that address points to the **TWO** procedure.
You're not calling number 4 — you're calling the address stored at position 4 (which corresponds to `TWO`).

Far Pointer Version

If the table has **far addresses (CS + IP)**, then each entry is **4 bytes** long.

We write:

```
CALL FAR PTR [4*EBX]
Indirect CALL lets the program pick which procedure to run — like
a menu selection or function table.
```

RET Instruction (Return)

After a procedure finishes, the **RET instruction** sends control **back** to where it was called from.

- **Near RET (C3H)** → pops **IP (16-bit)** from stack.
- **Far RET (CBH)** → pops **IP + CS (32-bit)** from stack.

That's how the CPU "remembers" where to go back.So, after CALL pushes the return address,
RET pops it back and goes there.

Protected Mode (80386+):

- **Far RET** removes **6 bytes** → (4 for EIP + 2 for CS)
- **Near RET** removes **4 bytes** → (for EIP only)

RET with Immediate Operand

Sometimes, a **RET instruction has a number**, like:

```
RET 4
```

What this means:

1. It **returns** from the procedure (normal RET).
2. Then it **adds 4 to SP**, which **removes 4 bytes** from the stack (like clearing extra data).

In some high-level languages (like **C/C++** or **PASCAL**),
parameters (values) are **pushed on the stack** before a procedure call.
RET 4 removes those parameters automatically.

Example

```
PUSH AX       ; push parameter 1 (2 bytes)
```

```
PUSH BX      ; push parameter 2 (2 bytes)
CALL ADDM    ; call procedure ADDM
```
Inside the procedure:

```
ADDM PROC NEAR
   PUSH BP
   MOV BP, SP
   MOV AX, [BP+4]  ; get parameter 1
   ADD AX, [BP+6]  ; add parameter 2
   POP BP
   RET 4           ; return and remove both parameters (4
bytes)
ADDM ENDP
```
So after returning, stack is clean — no leftover parameters!

Code Recap:

```
PUSH AX      ; pushes 2 bytes (1st parameter)
PUSH BX      ; pushes 2 bytes (2nd parameter)
CALL MyProc  ; pushes 2 more bytes (return address)
```
So before entering the procedure, the **stack** (top to bottom) looks like this:

Stack Content (top ↓)	Size	Description
Return Address	2 bytes	From CALL
BX	2 bytes	2nd parameter
AX	2 bytes	1st parameter

Total pushed = 6 bytes.Inside procedure:

```
MyProc PROC
   ; do something
   RET 6       ; return + remove 6 bytes
MyProc ENDP
```
Now here's what happens:

`RET` **pops** the return address → program control goes back to after the `CALL`.
Then it **adds 6 to SP** → removes **6 extra bytes** from the stack.

Before `RET 6`,
stack has:

- Return address (2 bytes)
- Parameters (4 bytes)
  Total = 6 bytes (which matches the RET 6 operand).

- So `RET 6` = **perfectly balanced** — it cleans up **all** (return + parameters).

- 🚨 **But wait!**

In your comment, you said:

"RET 6 — return + remove 4 bytes (the 2 parameters)"

That's **not correct here**
`RET 6` actually removes **6 bytes after** popping the return address.
So total removed = 8 bytes (2 for return + 6 more).

Instruction	Removes Return Addr	Adds to SP	Total Removed	Meaning
`RET`	✅ Yes	0	2 bytes	Just return
`RET 4`	✅ Yes	4	6 bytes	Return + 4 bytes (2 params)
`RET 6`	✅ Yes	6	8 bytes	Return + 6 bytes (3 params)

Since only **two parameters (4 bytes)** were pushed,
you should use → `RET 4`, not `RET 6`.If you use `RET 6`,
SP will move 2 bytes too far — stack gets unbalanced (program may crash or behave wrongly).Use
`RET n` where n = total bytes of parameters pushed before the call.

What is an Interrupt?

An **interrupt** means **stopping the main program for a short time** to do something important, and then coming **back to continue** where it left off.

It's like when you're doing homework and your phone rings you pause your work, answer the call, then return to studying.

Types of Interrupts

Type	Who causes it	Meaning
Hardware Interrupt	Comes from **outside** (a device, e.g., keyboard, timer)	Like a **hardware-generated CALL**
Software	Comes from **inside** (a program instruction)	Like a **special CALL**

Sometimes, **internal interrupts** are also called **exceptions**.
All interrupts call a small program called an **Interrupt Service Procedure (ISP)** or **Interrupt Handler**.
This small program handles the interrupt, then returns to the main program.

Interrupt Vector Table (Real Mode)

In **real mode**, the first **1024 bytes of memory (00000H–003FFH)** are used as the **interrupt vector table**.
Each entry (called a **vector**) takes **4 bytes**. **Each interrupt vector =**
→ 2 bytes for IP (Instruction Pointer)
→ 2 bytes for CS (Code Segment)

- 🔷 **Vector 0 → address 0000H–0003H**
- 🔷 **Vector 1 → address 0004H–0007H**
- 🔷 **and so on...**

There are **256 interrupt vectors** in total (0–255 or 00H–FFH).

In Protected Mode

In **protected mode**, the old vector table is replaced with an **Interrupt Descriptor Table (IDT)**.
Each entry here is **8 bytes long** and gives more detailed information about each interrupt.

Reserved and User Interrupts

Range	Used For	Who uses it
0–31 (00H–1FH)	System & CPU errors	Intel (reserved)
32–255 (20H–FFH)	Free to use	**User programs**

Interrupt No.	Function
0	Divide Error
1	Single-step
2	NMI (Non-Maskable Interrupt)
3	Breakpoint
4	Overflow
6	Invalid Opcode
10 (0x0A)	Floating-point error
20–255	User-defined interrupts

Software Interrupt Instructions

The microprocessor gives you **3 main software interrupt instructions**:

Instruction	Meaning	Use
INT n	Software interrupt with number *n*	General interrupt (most used)

INT 3	Special 1-byte interrupt	Used for debugging/ breakpoint
INTO	Interrupt on overflow	Used when overflow flag = 1

How INT Works (Step-by-step)

`INT 10H`

1️⃣ **CPU pushes FLAGS onto the stack →**
It saves the current program status (like carry, zero, overflow, etc.) so it can restore them later after handling the interrupt.

2️⃣ **CPU clears T and I flags →**
It turns off single-step mode (T) and disables other interrupts (I) to prevent disturbance while the current interrupt is being handled.

3️⃣ **CPU pushes CS onto the stack →**
It saves the current Code Segment value — so the CPU knows where to return after finishing the interrupt.

4️⃣ **CPU loads new CS from vector table →**
It looks up the interrupt vector table (at memory address `4 * interrupt number`) and loads the new Code Segment for the interrupt routine.

5️⃣ **CPU pushes IP onto the stack →**
It stores the address of the next instruction (where the program should continue later) onto the stack.

6️⃣ **CPU loads new IP from vector table →**
It gets the new Instruction Pointer from the same vector table — this is the starting address of the interrupt handler.

7️⃣ **CPU jumps to the new address (interrupt handler) →**
Now control is transferred to the interrupt service routine (ISR), which will execute the special task for this interrupt.

`INT = PUSHF + FAR CALL`

`INT vs CALL`

Feature	INT Instruction	CALL Instruction
What it does	Calls an interrupt service routine	Calls a subroutine
What it pushes	**FLAGS + CS + IP**	**CS + IP**
Used for	System functions, errors	User procedures

If we write:INT 10H

Then vector number = 10H
Memory address = 10H × 4 = 40H

So CPU will read the 4 bytes from **40H–43H** for the **IP and CS** of the service routine.

- The **Interrupt Vector Table** is at memory **0000:0000 – 0000:03FF** in real mode.
- It stores addresses for **256 possible interrupts** (00H–FFH).
- Each entry = **4 bytes**:2 bytes for **IP (offset),**2 bytes for **CS (segment)**

Why multiply by 4?

- Each interrupt number points to an **entry in the table**.
- Since **each entry is 4 bytes**, the **address of the entry** = `interrupt number × 4`.
- `INT 10H` → interrupt number = 10H = 16 decimal
- Each entry = 4 bytes → entry starts at:
  `16 × 4 = 64 decimal = 40H`
- So CPU reads **4 bytes from 0040H–0043H**:
  - 0040H–0041H → IP
  - 0042H–0043H → CS
- **INT n** → 2 bytes long
- **INT 3** → 1 byte long (used for breakpoints in debugging)

Software interrupts are used to call **system services** like:

- Display on screen (video)
- Print
- Disk read/write

You don't need to remember their addresses — the **INT instruction** knows them through the vector table!

IRET / IRETD Instruction

When an interrupt service routine finishes, we need to **go back** to where the program stopped. That's what **IRET (Interrupt Return)** does.

How IRET Works:

1️⃣ Pops **IP** from stack
2️⃣ Pops **CS** from stack
3️⃣ Pops **FLAGS** from stack

`IRET = POPF + FAR RET`
This means the program returns to the main program exactly as it was — even restoring the **I (Interrupt Enable)** and **T (Trap)** flags!

Difference between IRET and IRETD

Mode	Instruction	What it pops
Real Mode	**IRET**	16-bit IP + CS + FLAGS
Protected Mode	**IRETD**	32-bit EIP + CS + FLAGS

INT 3 – Breakpoint Interrupt

- **Purpose:** Special software interrupt for **debugging**.
- **Size:** 1 byte only (other INT instructions are 2 bytes).
- **Use:** Insert INT 3 in the code to **pause execution** — this is a **breakpoint**.
- **Why useful:** Easier to set breakpoints in debugging because it takes **less memory**.

```
INT 3   ; Breakpoint inserted here
```

INTO – Interrupt on Overflow

- **Purpose:** Conditional interrupt.
- **Condition:** Checks **Overflow flag (O)**:
  - If `O = 0` → do nothing
  - If `O = 1` → interrupt occurs using **vector 4**
- **Use case:** For **signed arithmetic operations** (addition/subtraction).
- **Alternative:** You can also detect overflow using the **JO instruction**.

```
ADD AX, BX
INTO   ; Will trigger interrupt if overflow occurs
```

Interrupt Service Procedure (ISP)

- **Definition:** A small routine that executes when an interrupt occurs.
- **Special:** Ends with **IRET** instead of RET.
- **Flags:** Flags register is **saved on the stack** during execution.
- **Registers:** Save all registers that are changed using `USES`.

Example (sum of DI, SI, BP, BX → AX):

```
INTS PROC FAR USES AX
ADD AX, BX
ADD AX, BP
ADD AX, DI
ADD AX, SI
IRET
INTS ENDP
```

- Notice **IRET** at the end, not RET.
- Saves AX on stack automatically because of `USES AX`.

Interrupt Control Instructions

- **STI → Set Interrupt Flag** → I = 1 → enables **INTR pin** (hardware interrupts).
- **CLI → Clear Interrupt Flag** → I = 0 → disables **INTR pin**.
- **Why enable interrupts early in ISP?**
- Many I/O devices rely on interrupts.
- Delaying enables can cause **system problems**.

Interrupts in Personal Computers

- Early PCs were 8086/8088 based → only had interrupts **0–4**.
- Later PCs keep these **for compatibility**.
- Access to **protected mode interrupts** (Windows) is via **kernel functions**.
- Protected mode uses **Interrupt Descriptor Table (IDT)**.

Interrupt Descriptor Table (IDT) is used in **Protected Mode** instead of the old **Interrupt Vector Table (IVT)** from Real Mode. It stores important information about all interrupts, traps, and exceptions in the system. Each entry in the IDT tells the CPU **where to jump (address of the interrupt routine)** and **what privilege level** is needed to execute it.

Unlike IVT, which is always fixed at memory address **0000:0000**, the IDT can be placed **anywhere in memory**. Its location and size are stored in a special register called **IDTR (Interrupt Descriptor Table Register)**. The IDTR holds two things — the **base address** (where the IDT starts) and the **limit** (how big the table is).

In short, the IDT is a **flexible and secure table** that helps the CPU handle interrupts safely in protected mode.

64-bit Mode Interrupts

- Uses **IRETQ** instruction.
- Pops **8-byte return address** from stack → for 64-bit addresses.
- Pops **32-bit EFLAG** and places into **RFLAG**.
- Otherwise, **behavior same as 32-bit mode**.

Mode	Return Instruction	Stack Content Popped
Real Mode	IRET	IP, CS, FLAGS
Protected Mode	IRETD	EIP, CS, FLAGS
64-bit Mode	IRETQ	RIP (8 bytes), RFLAGS (32 bits used)

In modern CPUs, different operating modes (Real, Protected, and 64-bit) use different return instructions after handling an interrupt. In **Real Mode**, the `IRET` instruction is used, which pops the **Instruction Pointer (IP)**, **Code Segment (CS)**, and **FLAGS** from the stack to return to the exact point where the interrupt occurred. In **Protected Mode**, the `IRETD` instruction is used instead, which pops **EIP**, **CS**, and **EFLAGS** — allowing return to a 32-bit address. In **64-bit Mode**, the `IRETQ` instruction performs the same task but with **64-bit addresses**; it pops **RIP (8 bytes)** and **RFLAGS** from the stack. Although the address width increases to 64 bits, only the **lower 32 bits of RFLAGS** are used. In simple terms, `IRET`, `IRETD`, and `IRETQ` are used to restore the program's previous state and resume execution after an interrupt in 16-bit, 32-bit, and 64-bit environments respectively.

Carry Flag Control

The **carry flag (C)** is used in multi-word addition/subtraction and error reporting.

Instruction	Function
STC	Set carry flag (C = 1)
CLC	Clear carry flag (C = 0)
CMC	Complement carry flag (C = 1 → 0, 0 → 1)

- DOS/BIOS procedures often set C = 1 to indicate **error** after a procedure.
- Example: Disk read error → C = 1, Success → C = 0.

WAIT / TEST Pin

- **WAIT:** Monitors **BUSY/TEST pin**.
    - **80286/80386:** checks BUSY pin

    - **8086/8088:** checks TEST pin
- **Logic:**
    - If pin = 1 → continue execution
    - If pin = 0 → wait until pin becomes 1
- **Use:** Synchronize with **coprocessor** (8087–80387)

HLT – Halt Execution

- Stops the CPU until:
    - Interrupt occurs
    - Hardware reset
    - DMA operation
- **Use:** Wait for interrupt or synchronize hardware.
- **Note:** Under DOS/Windows, HLT **won't completely stop** PC because OS uses interrupts extensively.

NOP – No Operation

- Does nothing, executes **short delay**.
    - Pad software for future instructions
    - Waste time (timing) — not very accurate on modern CPUs
- **Tip:** Often place **10 NOPs every 50 bytes** in old ML programs.

LOCK Prefix

- Ensures **exclusive access** to memory in multi-CPU systems.
- Sets **LOCK pin = 0** temporarily.
- Used for **atomic instructions**, e.g.:

```
LOCK MOV AL,[SI]
```

- **Effect:** Prevents other bus masters from interfering.

The **LOCK prefix** makes an instruction **atomic**, meaning it cannot be interrupted or interfered with by any other CPU or device until it finishes. During a locked instruction, **interrupts are temporarily delayed**, ensuring that memory access is completely safe and consistent. In a normal instruction, only the current CPU's operation is protected, but other CPUs can still access the same memory, which may cause data conflicts. The LOCK prefix prevents this by blocking other CPUs from accessing the memory until the instruction is done. This ensures **safe, synchronized updates**

to shared memory in multi-CPU systems — like correctly updating a shared bank balance without data loss.

ESC – Coprocessor Instructions

- Sends instructions to **floating-point coprocessor** (8087–Core2).
- Microprocessor handles **memory address**, otherwise acts as NOP.
- **Replaced now** by instructions like **FLD, FST, FMUL, etc.**

BOUND – Boundary Check

- Compares a **register** to **upper & lower memory limits**.
- If **register < lower** or **> upper** → **interrupt type 5** occurs.
- Otherwise, program continues.
- BOUND SI, DATA
- DATA → lower bound, DATA+2 → upper bound
- **Return address:** Points to BOUND instruction itself if interrupt occurs

ENTER / LEAVE – Stack Frame Instructions

- Used to **create & destroy stack frames** for procedures (C++ style).
- **ENTER ,** → Reserve `<size>` bytes for local variables, store BP.
- **LEAVE** → Restore SP and BP to previous values.
- **Example:**

```
ENTER 8,0   ; reserve 8 bytes for stack frame
```

- Stack Frame Layout:

Memory Location	Content
Old SP	BP
New SP+0…+7	Local Variables (8 bytes)

Use: Access local variables through BP.