1. What is the ADD Instruction?

👉 The ADD instruction is used to add two numbers.
It can add:

- **Two registers**
- **Register and memory**
- **Register and immediate value (a fixed number).But you cannot add:Memory + Memory.Segment registers (like CS, DS, ES)**

2. Sizes of Addition

The microprocessor can perform addition for:

- **8-bit numbers** (like AL, BL)
- **16-bit numbers** (like AX, BX)
- **32-bit numbers** (like EAX, EBX)
- **64-bit numbers (like RAX, RBX in newer processors)**

3. Types of Addition Instructions

Instruction	Meaning
`ADD AL, BL`	AL = AL + BL (adds BL to AL)
`ADD CX, DI`	CX = CX + DI
`ADD CL, 44H`	CL = CL + 44H (adds a constant number)
`ADD [BX], AL`	Adds AL to memory location pointed by BX
`ADD AL, [EBX]`	Adds the value stored in memory (EBX) to AL
`ADD BX, [SI+2]`	Adds the memory value (address = SI+2) to BX
`ADD RAX, RBX`	Adds RBX to RAX (used in 64-bit mode)

So basically, ADD destination, source
→ The result is stored in destination.

4. Register Addition Example

```
ADD AX, BX

ADD AX, CX

ADD AX, DX
```

🧾 **Explanation:**

- **First line: AX = AX + BX**
- **Second line: AX = (AX + BX) + CX**
- **Third line: AX = (AX + BX + CX) + DX**

So finally, AX holds the sum of AX + BX + CX + DX.

5. Flags That Change After Addition

Whenever the processor adds numbers, some flag bits change automatically (to tell about the result):

Flag	Meaning
Z (Zero)	1 if result = 0
C (Carry)	1 if carry occurs from MSB
S (Sign)	1 if result is negative
P (Parity)	1 if number of 1 bits is even
A (Auxiliary Carry)	For BCD correction
O (Overflow)	1 if result is too large for the size

These flags help in later decisions (like checking overflow, carry, etc.).6. Immediate Addition Example

This is when you add a fixed value (constant) to a register.

```
MOV DL, 12H     ; put 12H into DL

ADD DL, 33H     ; add 33H to DL
```

🧾 **Step-by-step:**

- **DL = 12H**
- **Then DL = 12H + 33H = 45H**

✅ **After addition:**

- **Z = 0 (result not zero)**
- **C = 0 (no carry)**
- **A = 0 (no half-carry)**
- **S = 0 (positive)**

- **P = 0 (odd parity)**
- **O = 0 (no overflow)**

7. Memory-to-Register Addition Example

Suppose you have data stored in memory named NUMB and NUMB+1, and you want to add them to AL.

```
MOV DI, OFFSET NUMB  ; DI points to NUMB

MOV AL, 0            ; clear AL (make it 0)

ADD AL, [DI]         ; add NUMB to AL

ADD AL, [DI+1]       ; add NUMB+1 to AL
```

🧾 **Explanation:**

- **`OFFSET NUMB` gives the memory address of NUMB.**
- **`[DI]` means "data at memory location stored in DI".**
- **So we are adding both NUMB and NUMB+1 to AL.**
- **The final result is stored in AL**

Suppose we have an array called ARRAY with 10 bytes.
We want to add element numbers 3, 5, and 7.

You can do something like this:

```
MOV AL, ARRAY[3]

ADD AL, ARRAY[5]

ADD AL, ARRAY[7]
```

So, AL will hold the **sum of ARRAY[3] + ARRAY[5] + ARRAY[7]**.

Type	Example	Meaning
Register + Register	`ADD AX, BX`	AX = AX + BX
Register + Immediate	`ADD DL, 33H`	DL = DL + 33H
Memory + Register	`ADD AL, [DI]`	AL = AL + memory[DI]
Register + Memory	`ADD BX, [SI+2]`	BX = BX + memory[SI+2]

⚠️ **Not allowed: Memory + Memory, Segment registers.**

Rule	What it means
Memory + Memory	CPU cannot read/write two memory locations at once. Must go via a
Segment Reg + Segment Reg	Segment registers cannot be used together in arithmetic. Must use a general-purpose register for calculations.

MOV [1000h], [2000h] ; ❌ Cannot move directly from memory to memory

MOV AX, [2000h] ; First, load memory into a register

MOV [1000h], AX ; Then, store register into memory

Segment registers example

ADD DS, ES ; ❌ You cannot add two segment registers

MOV AX, DS ; Load DS into AX

ADD AX, BX ; Arithmetic using general-purpose register

MOV DS, AX ; Store result back into DS if needed

Example 5–4: Array Addition (8-bit numbers)

```
MOV AL, 0          ; clear AL to store total sum
MOV SI, 3          ; start from element 3 in array
ADD AL, ARRAY[SI]      ; add array element 3
ADD AL, ARRAY[SI+2]    ; add array element 5
ADD AL, ARRAY[SI+4]    ; add array element 7
```

💡 Explanation:

1. `MOV AL, 0` → Clear AL (we'll store the total sum here).
2. `MOV SI, 3` → Start at array element number 3 (remember arrays are like a list of values).
3. `ADD AL, ARRAY[SI]` → Add value of element 3 to AL.
4. `ADD AL, ARRAY[SI+2]` → Add element 5 (3 + 2 = 5).
5. `ADD AL, ARRAY[SI+4]` → Add element 7 (3 + 4 = 7).

✅ **Final Result: AL = ARRAY[3] + ARRAY[5] + ARRAY[7]**

Example 5–5: Array Addition (16-bit numbers using scaling)

```
MOV EBX, OFFSET ARRAY   ; EBX = address of ARRAY
MOV ECX, 3              ; ECX = element number 3
MOV AX, [EBX + 2*ECX]   ; get element 3
MOV ECX, 5
ADD AX, [EBX + 2*ECX]   ; add element 5
MOV ECX, 7
ADD AX, [EBX + 2*ECX]   ; add element 7
```

💡 **Explanation:**

- **In 16-bit arrays, each number takes 2 bytes (1 word = 2 bytes).**
- **So, to move from one element to another, the address must be multiplied by 2.**
- **That's why we write 2 × ECX — this is called a scaling factor.**

✅ **After the last line, AX contains:**
ARRAY[3] + ARRAY[5] + ARRAY[7]

Increment Instruction (INC)

INC means *Increment* → it simply adds 1 to a register or memory location.

```
INC destination
```

Instruction	Meaning
`INC BL`	BL = BL + 1
`INC SP`	SP = SP + 1
`INC EAX`	EAX = EAX + 1
`INC BYTE PTR [BX]`	Adds 1 to the byte stored at memory[BX]
`INC WORD PTR [SI]`	Adds 1 to word at memory[SI]
`INC DWORD PTR [ECX]`	Adds 1 to doubleword at memory[ECX]
`INC DATA1`	Adds 1 to the variable DATA1
`INC RCX`	Adds 1 to RCX (used in 64-bit mode)

Example 5–6 (Using INC for Memory Addressing)

```
MOV DI, OFFSET NUMB    ; DI = address of NUMB
MOV AL, 0              ; clear AL
ADD AL, [DI]           ; add NUMB
INC DI                 ; increase DI → points to NUMB+1
ADD AL, [DI]           ; add NUMB+1
```

💡 Explanation:

- `DI` first points to NUMB.
- `ADD AL, [DI]` adds NUMB to AL.
- `INC DI` increases DI by 1 → now points to NUMB+1.
- `ADD AL, [DI]` adds NUMB+1 to AL.
  ✅ So final AL = NUMB + (NUMB+1)

- ### Important Notes:

- INC changes all flags except the Carry flag (CF).
  → This means Carry bit remains the same after INC.
- Use INC DI to move to next byte address.
- If your array has word-sized (2 bytes) data, use:
    - `ADD DI, 2` → moves pointer 2 bytes ahead.
    - For doubleword (4 bytes): `ADD DI,`
    - ### `Addition with Carry (ADC)`

ADC means *Add with Carry*.
It adds two numbers plus the carry flag (CF).

```
ADC destination, source
```

Instruction	Meaning
`ADC AL, AH`	AL = AL + AH + carry
`ADC CX, BX`	CX = CX + BX + carry
`ADC EBX, EDX`	EBX = EBX + EDX + carry
`ADC RBX, 0`	RBX = RBX + 0 + carry

`ADC DH, [BX]`	DH = DH + memory[BX] + carry

💡 Why Use ADC?

When you want to add large numbers (more than 16-bit or 32-bit), you must use ADC to include the carry from the lower part of addition.

Example 5–7: 32-bit Addition (using two 16-bit registers)

```
ADD AX, CX      ; add lower 16 bits

ADC BX, DX      ; add higher 16 bits + carry
```

💡 Explanation:

- **Step 1: Add low parts (AX + CX).**
  If the result creates a carry → CF = 1.
- **Step 2: Add high parts (BX + DX + CF).**
  So the carry from the first addition is included automatically.

Result: BX–AX = DX–CX + BX–AX (full 32-bit addition)

- **Add lower 16 bits:AX = AX + CX**
- **If this addition produces a carry, the carry flag CF = 1.**
- **Step 2b: Add higher 16 bits plus the carry:BX = BX + DX + CF ; CF comes from lower 16-bit addition**

✅ **Now `BX:AX` contains the full 32-bit result.**

Suppose we have two 32-bit numbers:

```
Number1 = BX:AX = 0x0001:0xFFFF   ; BX = 0x0001, AX = 0xFFFF

Number2 = DX:CX = 0x0000:0x0001   ; DX = 0x0000, CX = 0x0001
```

Visual Diagram

```
Step 1: ADD AX, CX

   AX = FFFF

 + CX = 0001
```

```
 ------------
   AX = 0000   (CF = 1)


Step 2: ADC BX, DX
   BX = 0001
 + DX = 0000
 + CF = 1
 ------------
   BX = 0002


Result: BX:AX = 0002:0000
```

Example 5–8: 64-bit Addition (using 32-bit registers)

```
ADD EAX, ECX    ; lower 32 bits
ADC EBX, EDX    ; upper 32 bits + carry
```

In 64-bit processors, you can just use:ADD RAX, RBXbecause RAX and RBX already store 64-bit values.

Exchange and Add (XADD)

XADD = eXchange and ADD.
It adds the source to the destination and then swaps (exchanges) their values.

```
BL = 12H
DL = 02H
XADD BL, DL
```

1. **Add DL to BL → BL = 12H + 02H = 14H**
2. **Then exchange values → DL = old value of BL (12H)**

- **BL = 14H**
- **DL = 12H**

- **XADD works with all register sizes and memory operands.**
- It's available in **80486 and later processors**.

SUB Instruction (Normal Subtraction)

👉 **It can subtract:**

- **register from register**
- **memory from register**
- **immediate (constant) value from register or memory**

But it CANNOT subtract memory from memory or **segment registers**.

Example (Register Subtraction)

```
SUB BX, CX    ; BX = BX - CX
SUB BX, DX    ; BX = BX - DX
```

Explanation:
The value of `CX` is subtracted from `BX`, then `DX` is subtracted from the new value of `BX`. After every subtraction, the flag register is updated (like carry, zero, sign, overflow etc.).

Example (Immediate Subtraction)

```
MOV CH, 22H    ; load 22H into CH
SUB CH, 44H    ; CH = 22H - 44H
```

Result:
22H – 44H = DEH (because it borrows).

Flags after this operation:

Flag	Meaning	Status
Z	Zero flag	0 → result not zero
C	Carry flag	1 → borrow occurred
A	Auxiliary carry	1 → half-borrow
S	Sign flag	1 → result is negative
P	Parity flag	1 → even number of 1's
O	Overflow flag	0 → no overflow

- **Carry Flag (C) shows borrow instead of carry.**

- **Overflow happens only if result > +127 or < -128 (for 8-bit signed numbers).**

2. DEC Instruction (Decrement)

`DEC` means decrease by 1.
It is a special form of subtraction that subtracts 1 from a register or a memory location.

✅ **It does not use a second operand — just decreases the value itself.**

```
DEC BH        ; BH = BH - 1

DEC CX        ; CX = CX - 1

DEC BYTE PTR [DI]   ; memory at [DI] = [DI] - 1
```

Why "PTR" is needed:

When using memory, the assembler must know the size of data (byte, word, double word).
So we write:

- **`DEC BYTE PTR [DI]` → subtract 1 from a byte**
- **`DEC WORD PTR [BP]` → subtract 1 from a word**
- **`DEC DWORD PTR [EBX]` → subtract 1 from a double word**

3. SBB Instruction (Subtract with Borrow)

`SBB` = Subtract with Borrow
This is used when we do subtraction on large numbers (more than 16 or 32 bits).

It works just like `SUB`, but it also subtracts the carry flag (C) if there was a borrow from the previous subtraction.

```
SBB AX, BX    ; AX = AX - BX - carry

SBB CL, 2     ; CL = CL - 2 - carry
```

When we subtract **multi-word (large)** numbers.

For example, if we have a 32-bit number divided into two 16-bit parts:

```
Higher part → BX, Lower part → AX

Subtract from SI (high), DI (low)
```

Then:

```
SUB AX, DI    ; subtract low 16 bits

SBB BX, SI    ; subtract high 16 bits with borrow
```

➡️ **The carry flag (C) passes the borrow from the first subtraction to the next — ensuring correct wide subtraction.**

Instruction	Meaning	Example
SUB	Subtract normally	`SUB AX, BX` → AX = AX - BX
DEC	Subtract 1	`DEC CX` → CX = CX - 1
SBB	Subtract with borrow	`SBB BX, SI` → BX = BX - SI - carry

Addition (ADC):

Step1: ADD AX, DI ; lower word

Step2: ADC BX, SI ; higher word + carry

`Result: BX:AX = full sum`

Subtraction (SBB):

```
Step1: SUB AX, DI    ; lower word

Step2: SBB BX, SI    ; higher word - borrow

Result: BX:AX = full difference
```

✅ So, **SBB is essentially the subtraction version of ADC**.

1. CMP (Compare) Instruction

`CMP` means **compare two values by subtracting** one from another,
but ❗ it **does not store** the result —
it **only changes the flag bits** (like Zero, Carry, Sign, etc).

So, the destination register stays the same.
Flags tell whether the first value is equal, greater, or smaller than the second.

```
CMP destination, source
```

It performs internally:
`destination - source`
(but result is not saved, only flags change)

✅ **Register–Register**
✅ **Register–Memory**
✅ **Register–Immediate**

❌ **Memory–Memory**
❌ **Segment Register compares not allowed**

```
CMP AL, 10H     ; compare AL with 10H
JAE SUBER       ; jump if AL >= 10H
```

- **`CMP AL,10H` subtracts 10H from AL (only affects flags).**
- **`JAE` (Jump if Above or Equal) checks the flags.**
- **If AL ≥ 10H → program jumps to label `SUBER`.**
- **If not, it continues normally.**

Common Conditional Jump Instructions:

Instruction	Meaning	Condition
JA	Jump Above	AL > 10H
JB	Jump Below	AL < 10H
JAE	Jump Above or Equal	AL ≥ 10H
JBE	Jump Below or Equal	AL ≤ 10H
JE / JZ	Jump if Equal or Zero	AL = 10H
JNE / JNZ	Jump if Not Equal	AL ≠ 10H

CMP + Conditional Jump → used for decision making in assembly (like `if` condition in C).

Example Table (How CMP Works)

Instruction	What Happens
`CMP CL, BL`	CL - BL (flags set)
`CMP AX, SP`	AX - SP (flags set)
`CMP RDI, RSI`	RDI - RSI (64-bit mode)
`CMP [DI], CH`	(memory at [DI]) - CH
`CMP AX, 2000H`	AX - 2000H

2. CMPXCHG (Compare and Exchange)

`CMPXCHG` = Compare and Exchange
→ It compares the destination operand with accumulator (AX, EAX, or RAX).

→ If they are equal, it copies the source value into the destination.
→ If they are not equal, it copies the destination value into the accumulator.

CMPXCHG destination, source

CMPXCHG CX, DX

- **Compares CX with AX**
- **If CX = AX → DX → CX**
- **If CX ≠ AX → CX → AX**

✅ **Works with 8-, 16-, and 32-bit data (80486–Core2 processors).**

- **`CMPXCHG8B` → compares two 64-bit values (in EDX:EAX).**
- **`CMPXCHG16B` → for 128-bit comparison (used in 64-bit mode).**
- **Z flag (Zero Flag)** = 1 → if equal
- **Z flag = 0 → if not equal**

🪲 **(Note: Older Intel CPUs had a small bug with this instruction.)**

3. Multiplication Instructions (MUL & IMUL)

Used to multiply two numbers.

There are two types:

Type	Name	Type of Number
MUL	Unsigned Multiply	Positive numbers only
IMUL	Signed Multiply	Can be positive or negative

Rules:

- **The multiplicand is always in AL, AX, or EAX.**
- **The multiplier can be any register or memory operand.**
- **The result (product) is double in size compared to operands.**

Operand size	Result stored in
8-bit × 8-bit	AX (16-bit)
16-bit × 16-bit	DX:AX (32-bit)

32-bit × 32-bit	EDX:EAX (64-bit)
64-bit × 64-bit (in 64-bit mode)	RDX:RAX (128-bit)

- **AX = lower part → useable immediately DX = higher part → overflow part. পরে যদি full 32-bit result প্রয়োজন → DX:AX হিসেবে use করতে পারি।**
- **Example (Unsigned)**

```
MOV BL, 5

MOV CL, 10

MOV AL, CL

MUL BL          ; AL * BL → AX

MOV DX, AX      ; move result to DX
```

Output:
5 × 10 = 50 → stored in AX → moved to DX.

Example (Signed)

```
MOV AL, -5

MOV BL, 10

IMUL BL         ; signed multiplication
```

Result: -50 → stored in AX (two's complement form)

Flags Affected:

Flag	Meaning
C (Carry)	Set if upper half of product ≠ 0
O (Overflow)	Same as Carry
Others (Z, S, P, A)	Undefined or unpredictable

✅ **If higher half = 0 → no overflow (C=O=0)**
❌ **If higher half ≠ 0 → overflow (C=O=1)**

Example MUL Table

Instruction	Operation

`MUL CL`	AL × CL → AX
`IMUL DH`	AL × DH (signed) → AX
`MUL TEMP`	AL × [TEMP] → AX
`IMUL BYTE PTR [BX]`	AL × [BX] (signed) → AX

Immediate Multiplication (IMUL with 3 operands)

✅ **Only works for signed numbers**
✅ **Introduced from 80186 and above**

Format:IMUL destination, source, immediate_data

IMUL CX, DX, 12H ; CX = DX * 12H (signed)

IMUL BX, NUMBER, 1000H ; BX = NUMBER * 1000H

Both `destination` and `source` must be 16-bit registers.Result → stored in destination.

Division

Two types again:

- **Unsigned division → `DIV`**
- **Signed division → `IDIV`**

Type	Dividend	Divisor	Quotient	Remainder
8-bit	AX	8-bit	AL	AH
16-bit	DX:AX	16-bit	AX	DX
32-bit	EDX:EAX	32-bit	EAX	EDX
64-bit	RDX:RAX	64-bit	RAX	RDX

👉 **Dividend is always double the size of divisor.**
👉 **There is no immediate division instruction.**

Two Common Division Errors

1. **Divide by zero** → not allowed

2. **Divide overflow → result is too large to fit in the destination**

If any occurs → **CPU generates an interrupt (error message)**

(a) 8-bit Division

```
MOV AX, 0010H   ; AX = 16

MOV BL, 02H     ; BL = 2

DIV BL
```

✅ **Quotient → AL = 8**
✅ **Remainder → AH = 0**

For **signed division**, use `IDIV`.

Example

MOV AX, 0010H

MOV BL, 0FDH ; -3

```
IDIV BL
```

→ AX = +16 / -3 = quotient -5, remainder +1

(b) 16-bit Division

- **Dividend = DX:AX (32 bits)**
- **Divisor = 16-bit register or memory**
- **Quotient = AX**
- **Remainder = DX**

```
MOV AX, -100

MOV CX, 9

CWD          ; sign-extend AX into DX

IDIV CX
```

→ Quotient = -11 (AX)
→ Remainder = -1 (DX)

(c) 32-bit Division

Used in **80386 and above**

- **Dividend = EDX:EAX (64 bits)**
- **Divisor = 32-bit number**
- **Quotient = EAX**
- **Remainder = EDX**

```
CDQ          ; sign-extend EAX into EDX

IDIV ECX
```

Converting Before Division

Before division, data must be made double-width (sign or zero extended):

Type	Unsigned Instruction	Signed Instruction
8 → 16 bit	MOVZX	CBW
16 → 32 bit	MOVZX	CWD
32 → 64 bit	MOVZX	CDQ

```
MOV AL, NUMB

MOV AH, 0        ; zero-extend

DIV NUMB1

MOV ANSQ, AL     ; quotient

MOV ANSR, AH     ; remainder
```

Size	MUL Result	DIV Input	Quotient	Remainder
8-bit	AX	AX ÷ reg/mem8	AL	AH
16-bit	DX:AX	DX:AX ÷ reg/mem16	AX	DX
32-bit	EDX:EAX	EDX:EAX ÷ reg/mem32	EAX	EDX
64-bit	RDX:RAX	RDX:RAX ÷ reg/mem64	RAX	RDX

Type	Unsigned Instruction	Signed Instruction
8 → 16 bit	Zero extend (MOVZX)	CBW (Convert Byte to Word)
16 → 32 bit	Zero extend (MOVZX)	CWD (Convert Word to Doubleword)
32 → 64 bit	Zero extend (MOVZX)	CDQ (Convert Doubleword to Quadword)

Explanation:

- **Unsigned → ছোট সংখ্যা বড় করতে শুধু 0 যোগ করা (Zero-extend)**
- **Signed → ছোট সংখ্যা বড় করতে sign bit copy করা (Sign-extend)**

When you divide two numbers in microprocessor instructions, you usually get two results:

- **Quotient (main answer)**
- **Remainder (what's left after division)**

There are a few options:

Method 1: Round Quotient after Division

Goal: Divide two numbers and round the quotient to nearest integer

Steps:

1. **`DIV` → divide AX by BL—Quotient → AL,Remainder → AH**
2. **Double the remainder → `ADD AH, AH`**
3. **Compare with divisor → `CMP AH, BL`**
4. **If doubled remainder ≥ divisor → increment quotient → `INC AL`**
5. **Else → leave quotient as is**

Example:13 ÷ 2 → quotient = 6, remainder = 1

`Double remainder = 2 < 2? no → leave quotient = 6`

✅ Result: **Rounded quotient**

Method 2: Get Fractional Part using Remainder

Goal: Find fractional part after integer division (0–255 scale)

Steps:

1. **`DIV` → get quotient and remainder—Quotient → save AL.Remainder → AH**
2. **Clear AL → `MOV AL,0`**
3. **Make AX = remainder × 256 (AH:AL)**
4. **`DIV BL` → divide by divisor to get fractional part**
5. **Save AL → fractional remainder**

Example:13 ÷ 2 → quotient = 6, remainder = 1

`Make AX = 1*256 = 256`

`Divide by 2 → 256 ÷ 2 = 128`

`AL = 128 → fractional part ≈ 0.5`

✅ Result: **Fractional part in 8-bit fixed-point**

Method	Purpose	Result Stored
1	Round integer quotient	AL → rounded quotient
2	Find fractional remainder	AL → fractional part (0–255 scale)

64-Bit Division (in Pentium 4 and Core2)

Modern 64-bit processors can divide very large numbers.

🔷 Dividend → stored in RDX:RAX

🔷 Quotient → goes to RAX

🔷 Remainder → goes to RDX

Examples:

Instruction	Operation
`DIV RCX`	RDX:RAX ÷ RCX → quotient in RAX, remainder in RDX
`IDIV DATA4`	RDX:RAX ÷ memory (signed division)
`DIV QWORD PTR[RDI]`	Divide RDX:RAX by 64-bit memory number

BCD numbers store **two decimal digits per byte**.

- **Example: 1234 → stored as 12 34 in BCD (each nibble = 1 decimal digit).**
- **Addition/subtraction must adjust the result so it's still a valid BCD.**

🔷 Key Instructions:

Instruction	Use
DAA	Decimal Adjust after Addition
DAS	Decimal Adjust after Subtraction

Rules:

- **Only works with AL register.**
- **Adjusts results of ADD/ADC or SUB/SBB to produce valid BCD.**

Example 5–18 — BCD Addition

Add 1234 + 3099 (packed BCD):

```
MOV DX,1234H        ; DX = 1234 BCD

MOV BX,3099H        ; BX = 3099 BCD


; Add lower byte (DL + BL)

MOV AL, BL

ADD AL, DL

DAA                  ; Adjust sum to BCD

MOV CL, AL           ; Save result


; Add higher byte (DH + BH) + carry

MOV AL, BH

ADC AL, DH

DAA
```

```
MOV CH, AL          ; Save result
```

Result: CX = 4333 (BCD sum)

Example 5–19 — BCD Subtraction

Subtract 1234 - 3099:

```
MOV DX,1234H

MOV BX,3099H


; Subtract lower byte

MOV AL, BL

SUB AL, DL

DAS

MOV CL, AL


; Subtract higher byte + borrow

MOV AL, BH

SBB AL, DH

DAS

MOV CH, AL
```

Result: CX = correct BCD subtraction result

ASCII Arithmetic

ASCII-coded numbers are **30H–39H** for digits 0–9.

Problem: Direct addition/subtraction gives wrong result (because ASCII codes are not numeric values).Key Instructions:

Instruction	Use
AAA	ASCII Adjust After Addition
AAD	ASCII Adjust Before Division

AAM	ASCII Adjust After Multiplication
AAS	ASCII Adjust After Subtraction

Example 5–20 — ASCII Addition

Add ASCII 1 + ASCII 9:

```
MOV AX, 31H        ; AL=31H (ASCII '1'), AH=0

ADD AL, 39H        ; Add ASCII '9'

AAA                ; Adjust sum to ASCII

ADD AX, 3030H      ; Convert to proper ASCII result
```

Result: AX = 3130H → ASCII '10'

Tip: AAA clears AH and adjusts AL so that result is correct ASCII.

AAD Instruction

- Used **before division**.
- Converts **two-digit unpacked BCD in AX** into **AL quotient + AH remainder**.
- **AX should contain BCD number before using AAD.**

Last line: `ADD AX, 3030H`

- **AL and AH contain binary numbers (numeric digits) after AAA.**
- **ASCII code for '0' = 30H.**
- Adding 3030H converts **binary numbers to displayable ASCII**.
- **AL = lower digit, AH = higher digit → AX now has two ASCII digits ready for display.**

This line converts the binary result into ASCII digits so it can be shown correctly on screen or output device.

Context: ASCII digits

- **ASCII '0'–'9' = 30H–39H**
- **উদাহরণ:**

```
ASCII '1' = 31H → Binary: 0011 0001
```

```
ASCII '5' = 35H → Binary: 0011 0101
```

- লক্ষ্য: **ASCII থেকে শুধু actual digit বের করা** → BCD 0–9

Masking with AND

```
AND BX, 0F0FH
```

- **Mask = 0F0FH → Binary: 0000 1111 0000 1111**
  - **Left byte (high) = 0F → keep lower nibble, high nibble clear**
  - **Right byte (low) = 0F → same**

```
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
```

BX = 3135H → ASCII '1' '5'

Byte	Binary	Mask	Result	Hex
31H	0011 0001	0000 1111	0000 0001	01H
35H	0011 0101	0000 1111	0000 0101	05H

- **High nibble cleared** → 0011 → 0000
- **Low nibble kept** → actual digit

✅ **Result: BX = 0105H → BCD digits 1 and 5**

- **ASCII digit: `0011 xxxx` → high nibble = 3 (for ASCII code)**
- **Mask = 0000 1111 → keeps low nibble**
- Low nibble = **actual number 0–9** → BCD
- **ASCII → BCD = remove high nibble → keep low nibble**
- **Instruction: `AND reg,0F` (or for 16-bit: `AND BX,0F0F`)**

Masking with OR

```
AL = 0011 0100  ; Binary = 34H

Mask = 0000 1111 ; Binary = 0FH
```

- Goal: **Set lower 4 bits to 1**

```
0 OR 0 = 0

0 OR 1 = 1

1 OR 0 = 1

1 OR 1 = 1
```

OR operation → output = 1 যদি কোনো এক input = 1

Step-by-step Calculation

```
AL = 0011 0100

Mask = 0000 1111

OR   = 0011 1111
```

✅ **Result: AL = 3FH → Lower 4 bits set to 1, high 4 bits remain same**

- OR → **একটি 1 থাকলেই output 1**
- **Mask = 0000 1111 → ensures lower 4 bits = 1**
- **High 4 bits remain unchanged → preserves original data**

Masking with XOR

```
AL = 0011 0101 ; 35H

Mask = 0000 1111 ; 0FH
```

- Goal: **Invert lower 4 bits**

```
0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0
```

XOR → output = 1 যদি inputs আলাদা হয়

AL = 0011 0101

`Mask = 0000 1111`

`XOR  = 0011 1010`

✅ **Result: AL = 3AH → Lower 4 bits flipped, high 4 bits unchanged**

XOR → **different bits → 1, same bits → 0**

- **Mask = 0000 1111 → flips only lower 4 bits**
- **High 4 bits remain same → preserves original data**

Instruction	Effect on bits	Example calculation	Result
AND	Clears bits (0)	0011 0101 AND 0000 1111	0000 0101
OR	Sets bits (1)	0011 0100 OR 0000 1111	0011 1111
XOR	Flips/inverts bits	0011 0101 XOR 0000 1111	0011 1010

OR (Inclusive OR)

- Output = 1 if any input = 1

- Used to SET bits

- Example: OR AL, 0Fh → lower 4 bits = 1

XOR (Exclusive OR)

- Output = 1 only if inputs are different

- Used to FLIP bits, CLEAR register, compare

- Example: XOR AX, AX → clears AX

AND (for comparison)

- Output = 1 only if both inputs = 1

- Used to CLEAR bits

- Example: AND BX, 0F0Fh → mask high nibble

AND → Always No (0)

OR → One Required (1)

XOR → Opposite / Flip

TEST Instruction

The **TEST instruction** performs the **AND operation**,
but unlike the AND instruction, it **does not change** the destination value.

👉 It only **updates the flag register** (especially the **Zero Flag, ZF**) based on the result.

Difference between AND and TEST:

Instruction	Effect on Operand	Purpose
AND	Changes the destination value	Performs real logic AND
TEST	Does *not* change the destination value	Only checks bits and sets flags

How it works:

It checks (tests) whether specific **bits** are 1 or 0 in a register.

- If the tested bit = 0 → **Zero flag (ZF) = 1**
- If the tested bit = 1 → **Zero flag (ZF) = 0**

Commonly used with:

`JZ` (Jump if Zero)
`JNZ` (Jump if Not Zero)

👉 These are used to make **decisions** depending on whether a bit is set or not.

Example:

```
TEST AL, 1        ; Test the rightmost bit of AL
JNZ RIGHT         ; Jump to RIGHT if bit = 1
TEST AL, 128      ; Test the leftmost bit of AL (128 = 80H)
JNZ LEFT          ; Jump to LEFT if bit = 1
```

💡 Meaning:

- `TEST AL, 1` checks the **least significant bit (LSB)**.
- `TEST AL, 128` checks the **most significant bit (MSB)**.

If any of those bits are 1, the `JNZ` instruction jumps to the specified label.

Example TEST Instructions (Table 5–19)

Instruction	Meaning
`TEST DL, DH`	DL AND DH → result affects flags only
`TEST CX, BX`	CX AND BX → updates flags
`TEST EAX, 256`	Tests if bit 8 of EAX is set
`TEST AH, 4`	Tests if bit 2 of AH is set

Bit Test Instructions (BT, BTS, BTR, BTC)

These are **special TEST-type instructions** available in **80386 and later processors**.

They **test specific bit positions** and store the result in the **Carry Flag (CF)**.

Types and their functions:

Instruction	Meaning	After Testing...
BT	Tests a bit	Only tests (no change)
BTS	Tests and Sets a bit	Makes the bit = 1
BTR	Tests and Resets a bit	Makes the bit = 0
BTC	Tests and Complements a bit	Flips the bit (1→0, 0→1)

Example:

```
BT  AX,4     ; Tests bit 4 of AX → result stored in Carry Flag (CF)
BTS CX,9     ; Test and set bit 9 of CX
BTR CX,0     ; Test and clear bit 0 of CX
BTC CX,12    ; Test and complement bit 12 of CX
```

💡 After these operations:

- **BTS** ensures the bit becomes 1
- **BTR** ensures the bit becomes 0
- **BTC** flips the bit

So you can **control individual bits** easily.

Example (from book):

```
BTS CX,9      ; set bit 9
BTS CX,10     ; set bit 10
BTR CX,0      ; clear bit 0
BTR CX,1      ; clear bit 1
```

```
BTC CX,12    ; flip bit 12
```

NOT and NEG Instructions

Both **NOT** and **NEG** deal with inversion (changing values).

NOT (Logical Inversion)

- Performs **one's complement** (flips every bit).
- 1 becomes 0, 0 becomes 1.
- Used for **logical** operations.

Example:

```
NOT CH      ; Invert all bits of CH
NOT EBX     ; Invert all bits of EBX
```

If CH = `11001001`, after NOT → `00110110`

NEG (Arithmetic Negation)

- Performs **two's complement**, i.e., changes the **sign** of a number.
- Converts **positive** ↔ **negative**.

```
NEG AX      ; AX = -AX
```

If AX = `0005H`, after NEG → `FFFBH` (which means -5 in signed form).

Summary Table

Instruction	Meaning
`NOT CH`	Logical (bitwise) NOT
`NEG CH`	Arithmetic negation (2's complement)
`NOT TEMP`	Invert all bits in memory variable TEMP

NOT → bitwise opposite (1↔0)
NEG → arithmetic opposite (+ ↔ −)

SHIFT Instructions (Introduction)

Shift instructions **move bits** to the left or right inside a register or memory.

They are used for:

- Bit manipulation
- Multiplication/division by powers of 2

- Data alignment

Types of Shift Instructions:

Type	Direction	Description
SHL / SAL	Left	Logical and arithmetic left shift (multiply by 2^n)
SHR	Right	Logical right shift (divide by 2^n, fill with 0)
SAR	Right	Arithmetic right shift (divide by 2^n, keeps sign bit same)

Example:

```
MOV AL, 00001111b
SHL AL, 1      ; AL = 00011110b (×2)
SHR AL, 1      ; AL = 00001111b (÷2)
```

🧠 **SHL (Left Shift)** → multiply by 2

🧠 **SHR (Right Shift)** → divide by 2

Logical vs Arithmetic Right Shift

Type	Leftmost Bit Filled With
Logical Right Shift (SHR)	0
Arithmetic Right Shift (SAR)	Copy of Sign Bit (for signed numbers)

Types of Shift:

Type	Full Form	Used For	Works With
SHL	Shift Logical Left	Multiply unsigned number by 2	Unsigned numbers
SHR	Shift Logical Right	Divide unsigned number by 2	Unsigned numbers
SAL	Shift Arithmetic Left	Multiply signed number by 2	Signed numbers
SAR	Shift Arithmetic Right	Divide signed number by 2	Signed numbers

🧩 How shifting affects numbers:

- **Shift Left (SHL or SAL)** → Multiplies the number by **2** for each bit shifted.
- **Shift Right (SHR or SAR)** → Divides the number by **2** for each bit shifted.

👉 Example:
If you shift left by 2 → number × 2 × 2 = number × 4
If you shift right by 2 → number ÷ 4

Example (From Table 5–22)

Instruction	Meaning
`SHL AX,1`	AX is shifted left 1 bit (×2)
`SHR BX,12`	BX is shifted right 12 bits (÷2^{12})
`SAL DATA1,CL`	Shift DATA1 left by number stored in CL
`SAR SI,2`	Shift SI right 2 bits (for signed numbers)

Important Note:

- If **CL** register is used → It stores the shift count (number of bit positions).
- The value in **CL doesn't change** after shifting.
- Shift count follows **modulo rule**:
  - 32-bit → modulo 32 (e.g., shift by 33 → same as shift by 1)
  - 64-bit → modulo 64

যখন তুমি কোনো register (যেমন AX, BX, EAX ইত্যাদি) কে shift করো,
তখন **shift করার bit সংখ্যাটা CL register-এ** রাখা যায়। যেমন:MOV CL, 33 SHL EAX, CL

CPU-এর shift instruction গুলো একটা "**modulo rule**" মেনে চলে —মানে, **তুমি যত bits shift করতে চাও, সেটা register size অনুযায়ী ভাগশেষ (remainder)** হিসেবে নেয়া হয়। Modulo মানে "ভাগশেষ"।
যেমন33 ÷ 32 = 1 ভাগশেষ 1,তাহলে 33 mod 32 = 1
তুমি যদি 32-bit register (যেমন EAX) shift করো,তাহলে shift count হিসেব হবে (count mod 32)।

তুমি যা লিখবে	আসলে যত bit shift হবে
32	0 bit (32 mod 32 = 0)
33	1 bit (33 mod 32 = 1)
40	8 bit (40 mod 32 = 8)

🔶 তাই `SHL EAX, 33` → আসলে `SHL EAX, 1` এর মতোই কাজ করবে।

একই নিয়ম, শুধু এখানে divisor 64।

তুমি যা লিখবে	আসলে যত bit shift হবে

64	0 bit (64 mod 64 = 0)
65	1 bit (65 mod 64 = 1)
70	6 bit (70 mod 64 = 6)

🔶 তাই `SHR RAX, 65` → আসলে `SHR RAX, 1` এর মতোই কাজ করবে।

কারণ CPU register-এর size নির্দিষ্ট — তুমি register-এর চেয়ে বেশি bit shift করতে পারো না।
তাই CPU স্বয়ংক্রিয়ভাবে সেই বড় সংখ্যাটা "ভাগশেষে" (modulo) রূপান্তর করে।

Register size	Rule	Example
8-bit	modulo 8	shift by 9 → same as shift by 1
16-bit	modulo 16	shift by 17 → same as shift by 1
32-bit	modulo 32	shift by 33 → same as shift by 1
64-bit	modulo 64	shift by 65 → same as shift by 1

Example 5–30 — Two ways to shift DX left by 14 bits:

```
SHL DX,14          ; Method 1 — immediate shift count
MOV CL,14
SHL DX,CL          ; Method 2 — use CL register
```

Both do the same thing: shift DX left by 14 bits.

MULTIPLICATION USING SHIFTS

You can multiply numbers **without using MUL**, just by shifting and adding.

👉 Example 5–31 — Multiply AX by 10 (decimal)

10 in binary = `1010` → (8 + 2)

```
SHL AX,1        ; AX × 2
MOV BX,AX
SHL AX,2        ; AX × 8
ADD AX,BX       ; (8×AX + 2×AX) = 10×AX
```

Similarly:

Multiply by	Binary	Explanation
18	10010	(16×AX + 2×AX)
5	101	(4×AX + 1×AX)

This method is **faster** than the MUL instruction in older processors.

SHL AX,1,মানে: AX কে ১ bit left shift করা, Left shift by 1 = ×2
অর্থাৎ,নতুন AX = পুরনো AX × 2
MOV BX,AX, মানে: AX-এর মান BX-এ কপি করা
তাই এখন BX = (পুরনো AX × 2)
SHL AX,2মানে: এখনকার AX কে ২ bit left shift করো,Left shift by 2 = ×4
তবে খেয়াল করো — এই সময় AX আগেই ×2 হয়েছিল।
তাহলে:নতুন AX = (পুরনো AX × 2) × 4 = পুরনো AX × 8

ADD AX,BX
AX = AX + BX

এখন আমরা জানি —AX = পুরনো AX × 8 , BX = পুরনো AX × 2

AX = (পুরনো AX × 8) + (পুরনো AX × 2)
AX = পুরনো AX × (8 + 2)
AX = পুরনো AX × 10
AX = 10 × (initial AX)
SHL = গুণ (×). SHR = ভাগ (÷)

প্রতিটি left shift মানে আগের সংখ্যার ২ গুণ:

SHL by 1 → ×2

SHL by 2 → ×4

SHL by 3 → ×8

```
AX = 5
SHL AX,1 → AX = 10
MOV BX,AX → BX = 10
SHL AX,2 → AX = 10×4 = 40
ADD AX,BX → AX = 40 + 10 = 50
```

DOUBLE-PRECISION SHIFTS (80386 and above)

These work with **two registers** together (like a big number).

Instruction	Meaning
SHLD	Shift Left Double
SHRD	Shift Right Double

They use **three operands**:

```
SHLD destination, source, count
SHRD destination, source, count
```

Example:

```
SHRD AX,BX,12
```

→ AX shifts **right** by 12 bits,
→ Rightmost 12 bits of **BX** fill into left of AX.

```
SHLD EBX,ECX,16
```

→ EBX shifts **left**, and
→ Leftmost 16 bits of **ECX** fill into right side of EBX.

SHRD (Shift Right Double) Destination এর কোন bit shift হয়: ডানদিকে shift হয়। Source এর কোন bit ঢোকে: Source-এর lower (ডান) bits।কোন দিক দিয়ে ঢোকে: ডান দিক থেকে ঢোকে।

SHLD (Shift Left Double) Destination এর কোন bit shift হয়: বামদিকে shift হয়।Source এর কোন bit ঢোকে: Source-এর higher (বাম) bits।কোন দিক দিয়ে ঢোকে: বাম দিক থেকে ঢোকে।

AX = 0001 1100 0000 0000b

BX = 1111 0000 1111 0000b

SHLD AX, BX, 4

AFTER SHIFT AX = 1100 0000 0000 0000

After fill up by BX AX = 1100 0000 0000 1111

4. ROTATE INSTRUCTIONS — (Bits go around)

👉 What is “Rotate”?

Rotate moves bits **in a circle** — bits that go out from one end come back from the other.

Think of it like rotating a wheel:

- Left rotate → bits move left, overflow bit comes to right.
- Right rotate → bits move right, overflow bit comes to left.

🔶 Types of Rotate:

Instruction	Meaning
ROL	Rotate Left
ROR	Rotate Right
RCL	Rotate Left through Carry
RCR	Rotate Right through Carry

📘 Example (From Table 5–23)

Instruction	Meaning
`ROL SI,14`	Rotate SI left 14 times
`RCL BL,6`	Rotate BL left through Carry 6 times
`RCR AH,CL`	Rotate AH right through Carry by value in CL
`ROR WORD PTR [BP],2`	Rotate value in memory right 2 times

💡 Example 5–32 — Shift a 48-bit number left

Registers: DX (high), BX (mid), AX (low)

```
SHL AX,1        ; Shift AX left → bit moves to Carry
RCL BX,1        ; Carry moves into BX
RCL DX,1        ; Carry moves into DX
```

✅ This way, all three registers act like one long 48-bit number shifted left once.

```
DX = 0001H
BX = 0002H
AX = 0004H
```

প্রতিটাই **16-bit register**

তাহলে মোট 👉 48-bit number:DX : BX : AX

```
0001 : 0002 : 0004
```

এটা একসাথে ধরলে একটা **48-bit বড় সংখ্যা**।

`SHL AX,1`, `RCL BX,1`, `RCL DX,1`

`SHL AX,1`

```
AX = 0004H → Binary = 0000 0000 0000 0100
```

Left shift by 1 → সব bit এক ধাপ করে বামে যাবে।

```
Before: 0000 0000 0000 0100
After : 0000 0000 0000 1000
```

- বাম দিকের bit (MSB) = 0, তাই Carry Flag (CF) = 0, ডান দিকে 0 ঢুকবে।

```
AX = 0008H
CF = 0
```

RCL BX,1

```
BX = 0002H → Binary = 0000 0000 0000 0010
CF = 0 (আগের step থেকে)
```

RCL মানে "Rotate through Carry Left"
→ সব bit এক ধাপ করে বামে যাবে,Carry Flag ঢুকবে ডান দিক দিয়ে।

```
Before: CF | 0000 0000 0000 0010
             ↓
After : 0000 0000 0000 0100 | CF
```

এখন বাম দিকের bit (MSB) = 0 → তাই CF = 0

```
BX = 0004H
CF = 0
```

RCL DX,1

```
DX = 0001H → Binary = 0000 0000 0000 0001 ,CF = 0
```

Rotate left through carry again:

```
Before: CF | 0000 0000 0000 0001
After : 0000 0000 0000 0010 | CF
DX = 0002H
CF = 0
```

Register	Before	After	Meaning
DX	0001H	0002H	Shifted + carried
BX	0002H	0004H	Got AX's carry
AX	0004H	0008H	Shifted left once
CF	—	0	No overflow carry

BEFORE AND AFTER SHIFT

```
DX : BX : AX = 0001 0002 0004
DX : BX : AX = 0002 0004 0008
```

পুরো 48-bit সংখ্যা দ্বিগুণ হয়েছে!

যেমনটা expected ছিল, কারণ left shift মানে multiply by 2।

5. BIT SCAN INSTRUCTIONS (80386 and later)

These **find the first '1-bit'** in a number.

Instruction	Full Form	Scans From
BSF	Bit Scan Forward	Left → Right
BSR	Bit Scan Reverse	Right → Left

👉 How it works:

- If a `1-bit` is found →
  → its **bit position** is stored in destination register
  → **Zero Flag (ZF) = 0**
- If no `1-bit` is found (all zeros) →
  → **ZF = 1**

🧩 Example:

```
EAX = 60000000H
BSF EBX,EAX
```

→ First 1-bit found at **bit 30**
→ EBX = 30, ZF = 0

```
BSR EBX,EAX
```

→ Scans from right → first 1-bit at **bit 29**
→ EBX = 29, ZF = 0

Bit Scan Instructions — কী করে?

👉 মানে ধরো কোনো binary সংখ্যায় অনেকগুলো bit আছে (0 আর 1)।
ওরা খুঁজে বের করবে **প্রথম যে জায়গায় (position) 1 আছে।**

Instruction	Full Form	কোন দিক থেকে খোঁজে
BSF	Bit Scan Forward	বাম → ডান (Left → Right)
BSR	Bit Scan Reverse	ডান → বাম (Right → Left)

🎯 কী হয় যদি "1" পাওয়া যায়?

- যেই জায়গায় (position) প্রথম 1 পাওয়া যায় →সেই bit-এর **position number** (0 থেকে গোনা হয়) → destination register এ জমে। আর **ZF = 0** হয়।

যদি কোনো 1 না পাওয়া যায় (মানে সব 0 থাকে)**

→ তাহলে **ZF = 1** হবে
→ এবং destination register **অপরিবর্তিত থাকবে।**

```
EAX = 60000000H
EAX = 0110 0000 0000 0000 0000 0000 0000 0000₂
bit no: 31 ............. 0
binary : 0 1 1 0 0000 .... 0000
```

`BSF EBX, EAX`BSF = Bit Scan Forward (বাম → ডান)

মানে 👉 bit 0 থেকে শুরু করে ডান থেকে বাম দিকে খুঁজবে প্রথম "1-bit"। Binary number দেখো:0110 0000 0000 0000

সবচেয়ে **ডান দিক থেকে গুনলে**
প্রথম 1 পাওয়া যাবে **bit 29 এ** না, **bit 29 এর বাম দিকেরটা bit 30**।

- bit 29 = 1,bit 30 = 1
  (কিন্তু BSF ডান দিক থেকে খোঁজে, তাই প্রথম 1 পায় **bit 29 এ**।)EBX = 29,ZF = 0

`BSR EBX, EAX`

BSR = Bit Scan Reverse (বাম → ডান নয়, বরং ডান → বাম)

মানে 👉 bit 31 থেকে শুরু করে বাম দিক থেকে ডান দিকে খুঁজবে। Binary number এ সবচেয়ে বামদিকের 1-bit হলো **bit 30**। EBX = 30,ZF = 0

Instruction	Direction	পাওয়া 1-bit এর Position	Result Register	ZF
BSF EBX,EAX	ডান দিক থেকে (bit 0 → 31)	bit **29**	EBX = 29	0
BSR EBX,EAX	বাম দিক থেকে (bit 31 → 0)	bit **30**	EBX = 30	0

String instructions are used to **manipulate or compare blocks of memory**, like searching or skipping certain bytes.

SCAS — String Scan

Purpose: Compare a register (AL, AX, or EAX) with **memory byte/word/doubleword**.

Opcode	Compare Type	Register Used	Memory Operand

SCASB	Byte	AL	[ES:DI]
SCAS W	Word	AX	[ES:DI]
SCASD	Doubleword	EAX	[ES:DI]

How it works:

- Compares **AL/AX/EAX** with memory at **[ES:DI]**.
- Does **not change AL/AX/EAX or memory**.
- Affects **flags** (ZF, etc.).

Direction: Controlled by **Direction Flag (DF)**:

  - `CLD` → auto-increment DI
  - `STD` → auto-decrement DI

Prefix	Meaning
REPNE	Repeat while **not equal**
REPE	Repeat while **equal**

Example 5–33 — Find 00H in memory_SIR HAS DONE IT

```
MOV DI, OFFSET BLOCK ; point DI to memory
CLD                   ; auto-increment
MOV CX, 100           ; 100 bytes to scan
XOR AL, AL            ; AL = 00H
REPNE SCASB           ; repeat until AL=memory or CX=0
```

Example 5–34 — Skip spaces (20H)

```
CLD
MOV CX, 256           ; length of string
MOV AL, 20H           ; ASCII space
REPE SCASB            ; skip spaces while AL = memory
```

CMPS — Compare StringsP

Opcode	Compare Type	Source	Destination
CMPSB	Byte	[DS:SI]	[ES:DI]
CMPS W	Word	[DS:SI]	[ES:DI]
CMPSD	Doubleword	[DS:SI]	[ES:DI]
CMPSQ	Quadword (64-bit)	[DS:SI]	[ES:DI]

How it works:

- Compares **memory[SI]** with **memory[DI]**.
- **SI and DI auto-increment/decrement** depending on DF.
- Flags are set after comparison.
- **Prefixes (like SCAS):**

Prefix	Meaning
REPE / REPZ	Repeat while **equal**
REPNE / REPNZ	Repeat while **not equal**

Example 5–35 — Compare two strings

```
MOV SI, OFFSET LINE      ; source string
MOV DI, OFFSET TABLE     ; destination string
CLD                       ; auto-increment
MOV CX, 10                ; number of bytes to compare
REPE CMPSB                ; compare bytes while equal
```

Interpretation:

- Stops when **CX=0** or **memory differs**.
- After execution:
  - `CX=0` → all bytes matched
  - ZF=0 → strings not equal
  - ZF=1 → strings match (depending on last comparison)

Quick Tips for Exams:

1. **SCAS** → Compare **register vs memory**
2. **CMPS** → Compare **memory vs memory**
3. **Prefixes:REPE / REPZ → Repeat if equal**
   -]**REPNE / REPNZ** → Repeat if not equal
4. **Direction Flag:**
   - **CLD** → DI/SI increments (forward),,,,**STD** → DI/SI decrements (backward)
5. **AL/AX/EAX** → used for SCAS
6. **SI & DI** → used for CMPS