Virtual University	**CS-401** **ASSEMBLY LANGUAGE PROGRAMMING** **SHORT NOTES FOR MID-TERM** **Chapter 1 to 6** **PREPARED BY ALMAS AFZAL**	 **Learn with ALMAS AFZAL**
0349-7927349	**Contact For LMS Handling & Paid Projects** almasafzal88@gmail.com	**15 August 2025**

CS401 – Preface

PREPARED BY ALMAS AFZAL

1. What is Assembly Language?

- A programming language that talks directly to the computer.
- In high-level languages (like C, C++, Java), there is a **"gap"** between the **programmer and the machine.**
- This gap is filled by **compilers or interpreters**, but in assembly, you bypass most of them.
- It works **close to hardware**, giving you more control.

2. Is it difficult?

- People think it's hard, but it's actually simple.
- It looks hard because we are used to thinking computers are complicated.
- Assembly is just **direct and clear** — what you write is exactly what the computer does.

3. Course focus

- Examples will use IBM PC with Intel processors.
- Concepts will be platform-independent (can work on any system).

Basic knowledge needed:

1. Digital logic (AND, OR, NOT)
2. Binary numbers and arithmetic

4. Why learn Assembly?

Optimization: Makes code faster by removing unnecessary instructions.

Real-time systems: Needed when operations must finish in a fixed time.
Fine control: You control each instruction — no hidden steps like in "for" loops in high-level languages.

5. When to use it?

- ❖ In performance-critical parts of a program (e.g., image processing).
- ❖ To make programs run faster by saving CPU cycles.
- ❖ To meet exact timing requirements in hardware control systems.

6. How to learn it?

- ❖ You can't just read; you must practice.
- ❖ Try every example, run code, test, and experiment.
- ❖ Only by doing this will you see how powerful and simple it is.

Chapter 1
Introduction to Assembly Language

PREPARED BY ALMAS AFZAL

1.1. BASIC COMPUTER ARCHITECTURE

1. Computer main parts:

1. **Processor (CPU)** – Does the operations.
2. **Memory (RAM)** – Stores data.
3. **I/O devices** – Connects computer to outside world (keyboard, display, etc.).

For now, we focus only on Processor + Memory.

2. Communication between CPU & Memory

To work, CPU must:

- ❖ Know which data to **read/write**.
- ❖ Tell memory **whether** to read or write.
- ❖ Actually **transfer the data**.

This is done via 3 types of buses:

Bus	Purpose	Direction
Address Bus	Sends memory address (location) to access.	**Unidirectional** (CPU → Memory)
Data Bus	Transfers actual data between CPU & Memory.	**Bidirectional**

Bus	Purpose	Direction
Control Bus	Sends control signals (read/write, status info).	**Bidirectional**

3. How each bus works

Address Bus

- Tells memory exactly which cell to use (like giving an address to a postman).
- Always **one-way** from CPU to Memory.
- Example: "Fifth book from top" = specific memory location.
- **Memory cell size** is called **cell width** (often 8-bit = 1 byte).

Data Bus

- Transfers actual data.
- Can go **both** directions:
  1. CPU → Memory (write)
  2. Memory → CPU (read)

Control Bus

Sends read/write commands and special status signals.
Also lets peripherals send **"status"** back to CPU (e.g., "door locked" message).
Ensures synchronization — CPU reads when memory is ready, writes when memory can accept.

4. Why synchronization is important

- If CPU wants to read but memory is trying to write → No communication happens.
- Control bus ensures CPU and memory act in correct order (one speaks, one listens).

PREPARED BY ALMAS AFZAL

1.2 Registers

What are Registers?

- Tiny, super-fast storage areas inside the CPU.
- Used to hold data (operands) temporarily while processing.
- Faster than main memory.
- Very few in number → precious resource.

Why do we need Registers?

- CPU can get only one data item at a time from memory via the data bus.

❖ But most operations (like addition) need **two operands**.

Solution:

❖ Store one operand in a register.
❖ Bring the second operand from memory.
❖ Perform operation quickly.
❖ Sometimes both operands are in registers for **maximum speed**.

Registers vs Memory

Registers	Memory
Very fast	Slower
Few in number	Large size
Inside CPU	Outside CPU
Expensive	Cheaper

Types of Registers

1. Accumulator

➢ Main register for **math & logic** operations.
➢ Width of accumulator = CPU word size
➢ (e.g., 32-bit CPU → 32-bit accumulator).

2. Pointer / Index / Base Register

➢ Holds **address of data** (not the data itself).
➢ Useful in **loops & arrays** (e.g., accessing each element in order).
➢ Changes automatically in repetitive tasks.

3. Flags Register (Program Status Word)

➢ Stores **status bits** after operations.
➢ Each bit has its **own meaning** (not combined).
➢ Example: Carry Flag stores extra bit after addition overflow.

4. Program Counter / Instruction Pointer

➢ Holds **address of next instruction** to execute.
➢ Moves **step-by-step** through the program.
➢ CPU understands numbers **(opcodes)**, not words like "ADD".

PREPARED BY ALMAS AFZAL

- Mnemonics (e.g., ADD A, B) are symbolic forms → converted by Assembler into opcodes.

1.3 Instruction Groups

What are Instruction Groups?

Every processor has a set of opcodes for:

- Moving data
- Doing math & logic
- Controlling program flow
- Special tasks

Mnemonics (human-readable names for opcodes) vary by manufacturer:
One CPU may use MOV, another may use LOAD or STORE.
Basic idea is same → only names differ.
Grouping instructions makes it easy to learn a new processor.

Main Instruction Groups

1. Data Movement Instructions

Purpose: Move data from one place to another.

Sources/Destinations can be:

1. Registers
2. Memory
3. Peripheral devices

Examples:

mov ax, bx → Copy data from bx to ax.
lad 1234 → Load data from address 1234 into a register.

Key point: Only moves data, doesn't change it.

2. Arithmetic and Logic Instructions

Arithmetic → Add, subtract, multiply, divide.
Logic → AND, OR, XOR, NOT (complement).

Examples:

PREPARED BY ALMAS AFZAL

and ax, 1234 → Bitwise AND with 1234.
add bx, 0534 → Add 0534 to bx.
add bx, [1200] → Add data from memory address 1200 to bx.

[1200] means get data from memory at address 1200.

3. Program Control Instructions

Normally CPU executes instructions in order using the Instruction Pointer.

Sometimes we want to:

- Jump to another part of program if a condition is met.
- Call another block of code, then return.
- These instructions change the normal flow of execution.

Examples:

cmp ax, 0 → Compare ax with 0.
jne 1234 → Jump to address 1234 if Not Equal.

Used for: decision making, loops, and branching.

4. Special Instructions

- Rare but important commands.
- Change processor's behavior directly.

Examples:

cli → Clear Interrupt Flag → Ignore outside signals (focus on current task).
sti → Set Interrupt Flag → Resume listening to outside signals.
Like special commandos → used in critical tasks

1.4 Intel iAPX88 Architecture

We will use a specific processor architecture to make our study practical.
The one chosen here is IBM PC with Intel architecture because:

- It is easily available.
- Free assemblers and debuggers exist for it.
- It is widely used in different fields.

Although we'll focus on this architecture, the concepts apply to other processors too — **only the mnemonics** (instruction names) change.

PREPARED BY ALMAS AFZAL

What is iAPX88?

- **Full form:** Intel Advanced Processor Extensions 88.
- Commonly known as **8088**.
- Used in **first IBM PCs**.
- Very successful processor in its time.

Our Course Focus:

First half → Study 8088 processor (16-bit).
Second half → Study iAPX386 processor (32-bit).

1.5 History

Early Intel processors:

Intel first made **4-bit** processors, but they were not very powerful.

8080 Processor:

- First important processor from Intel.
- **8-bit processor.**
- Became popular because it was simple and had a versatile architecture (could be used in many ways).

8085 Processor:

- **Improved version of 8080.**
- Again became popular, especially among engineers, because it was logical and easy to use.

Introduction of 8088

Intel then launched its **first 16-bit processor**, called **8088**.
At that time, personal computers were just starting to be developed.

Memory jump:

8085 → maximum memory 64 KB.
8088 → could handle 1 MB memory.

IBM used 8088 in its personal computers (PC).
First IBM PCs ran at 4.43 MHz — considered very fast for that time.

Why was it so successful?

- Right technology at the right time.

PREPARED BY ALMAS AFZAL

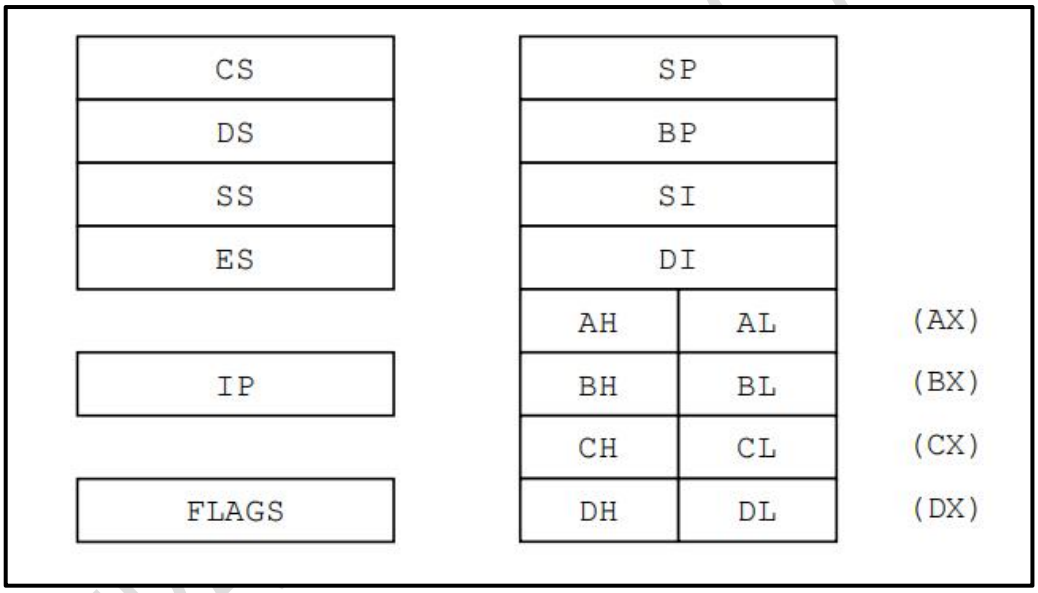
- ❖ IBM PC XT became very popular because:
- ❖ Open architecture → easy for other companies to make compatible products.
- ❖ Easily available technical information.

Unexpected Long Life

- ➢ Even Intel didn't expect it to last so long.
- ➢ The timer tick in the processor was made for **only 5 years** of operation.
- ➢ But even **25 years later**, the architecture was still being used.
- ➢ Because of the old tick design, special handling was needed from time to time.

1.6 Register Architecture

The iAPX88 processor has 14 registers. These are small storage areas inside the CPU **used for storing data, addresses, and controlling program flow.**



1. General Purpose Registers (AX, BX, CX, DX)

These can be used for many purposes (storing numbers, temporary results, etc.).
Each is **16-bit** but can be **split into two 8-bit registers**:

Example: AX → AH (high 8 bits) + AL (low 8 bits).
Changing AH or AL also changes AX.

Names have special meanings:

AX → Accumulator (used in some special instructions).
BX → Base (used in memory addressing).
CX → Counter (used in loops).
DX → Destination (used in input/output operations).

2. Index Registers (SI, DI)

- SI = **Source Index**, DI = **Destination Index**.
- Hold **addresses** in memory.
- Commonly used in **string operations**, but can also be used in normal calculations.
- **Always 16-bit** (cannot be split like AX, BX, etc.).

3. Instruction Pointer (IP)

- Stores the **address of the next instruction** to run.
- We **cannot directly change** it like normal registers (only program control instructions can change it).

4. Stack Pointer (SP)

Points to the **top of the stack** (special memory area used for storing temporary data).

5. Base Pointer (BP)

Also points **into the stack memory** but usually for accessing function parameters.

6. Flags Register (FLAGS)

Contains status bits that store results of operations or control processor behavior.

Important flags:

1. **C (Carry)** → Stores extra bit when result is too big.
2. **P (Parity)** → Checks if number of 1-bits is even or odd (used in communication).
3. **A (Auxiliary Carry)** → Carry between lower nibble (4 bits) and upper nibble (used in BCD/hex operations).
4. **Z (Zero)** → Set if the result is 0.
5. S (Sign) → Shows if result is positive or negative.
6. **T (Trap)** → Used for debugging (single-step mode).
7. **I (Interrupt)** → Allows or blocks external interrupts.
8. **D (Direction)** → Controls string operation direction (forward/backward).
9. **O (Overflow)** → Set if signed number result is too big to fit.

7. Segment Registers (CS, DS, SS, ES)

Used in segmented memory model:

- CS = Code Segment (where program instructions are stored).
- DS = Data Segment (where program data is stored).
- SS = Stack Segment (where stack memory is located).

PREPARED BY ALMAS AFZAL

- ES = Extra Segment (extra data storage).

1.7. OUR FIRST PROGRAM

"Program is an ordered set of instructions for the processor."

Program Logic (Plain English)

```
move 5 to ax
move 10 to bx
add bx to ax
move 15 to bx
add bx to ax
```

Assembly Version

```
mov ax, 5    ; AX = 5
mov bx, 10   ; BX = 10
add ax, bx   ; AX = AX + BX
mov bx, 15   ; BX = 15
add ax, bx   ; AX = AX + BX
mov ax, 0x4c00  ; exit program
int 0x21
```

Important Terms

- mov → copy data from source to destination.
- add → destination = destination + source.
- ; → comment (ignored by computer).
- org 0x0100 → tells assembler program will start at memory offset 0100h (for .COM files).
- mov ax, 0x4c00 + int 0x21 → ends the program in DOS.

Assembler, Linker, and Debugger

- To run an assembly program, we need to convert it into executable machine code.
- This is done using an **assembler**. In our course, we use **NASM (Netwide Assembler)** – it's free and open source.
- We also use a debugger to check and test the program step-by-step. We will use **AFD (A Fullscreen Debugger)**.

Assembling a Program

If our program file name is **EX01.ASM**, we type:

PREPARED BY ALMAS AFZAL

"nasm ex01.asm –o ex01.com –l ex01.lst"

EX01.COM → executable file (ready to run)
EX01.LST → listing file (shows program code, opcodes, and memory offsets)

Little-endian vs Big-endian

Big-endian: stores most significant byte first (used by Motorola).
Little-endian: stores least significant byte first (used by Intel).

Important Points

- COM files always start at offset 0100h (256 in decimal) – that's why we use org 0x0100 at the start of the program.
- Debugger can read the COM file and show the program even without the source code.
- When running in the debugger:
- AX and BX will change as we add numbers.
- IP (Instruction Pointer) changes after each instruction to point to the next one.
- Since this program does not use memory yet, only the registers change.

1.8. SEGMENTED MEMORY MODEL

Why it was needed (Rationale)

- Old processors like 8080 and 8085 used a linear memory model – the whole memory looked like one continuous array.
- They could only access 64 KB of memory because their address bus had 16 lines.
- When Intel designed 8088, they wanted:
- Compatibility with 8080/8085 programs.
- More memory than 64 KB (since it was too small).

Solution: Segmented Memory Model – divides memory into separate parts (Code, Data, Stack) instead of one continuous block.

How it works (Mechanism)?

- Think of memory as a building and segments as windows to look into different parts of it.
- 8088 can access 1 MB memory using 20-bit addresses.
- Each segment (window) is 64 KB big and can be moved anywhere in the 1 MB space.

4 segment registers are used:

PREPARED BY ALMAS AFZAL

CS – Code Segment (where program instructions are).
DS – Data Segment (where data is stored).
SS – Stack Segment (for function calls and temporary storage).
ES – Extra Segment (extra data area, used for strings).

Base + Offset Concept

A memory address = Segment Base + Offset.

Example:

CS (segment base) tells where the code starts.
IP (offset) tells the position of the next instruction inside that code segment.
IP is only 16-bit → can see only 64 KB at a time.
To see another part of memory → change the segment base (CS, DS, SS, ES).

Paragraph Boundaries

Segments always start at addresses divisible by 16 bytes (called **paragraph boundaries**).
First segment = 0000 (address 00000)
Next segment = 0001 (address 00010 in hex = 16 decimal)

Logical vs Physical Address

- **Logical Address** = Segment:Offset (e.g., 1DDD:0100)
- **Physical Address** = Actual 20-bit location in RAM (e.g., 1DED0)

Chapter 2
Addressing Modes

2.1. DATA DECLARATION

1. Operands in Assembly

Example: mov ax, 5

MOV → opcode (instruction)
AX → destination operand (where value will be stored)
5 → source operand (value to move)

If value is directly written inside instruction (like 5), it is called **Immediate Operand**.

PREPARED BY ALMAS AFZAL

2. Why We Need Data in Memory

- Using only constants (immediate values) is not practical.
- Constants **cannot** be changed → they cannot be destination operands.
- Real programs need **variables** stored in memory.

3. Data Declaration Directives

To store data in memory, we use **assembler directives**:
db (define byte) → reserves 1 byte in memory.
Example:
x db 5 ; store value 5 in one byte

dw (define word) → reserves **2 bytes (16 bits)** in memory.
Example:
y dw 1234h ; store word (2 bytes) in memory

Similar directives exist for double word and quad word, but usually we use **db and dw**.

4. Labels (Variable Names)

We give a **name (label)** to memory location for easy reference.
Example:
num db 10

Here **num** is a label → assembler will replace it with the actual memory address.
Just like variables in high-level languages (C, Java).

2.2. DIRECT ADDRESSING

1. What is Direct Addressing?

In direct addressing, data is stored in **memory variables** instead of constants.
Instruction format:
mov ax, [num1]

→ This means load value from memory location num1 into AX register.

The **brackets []** mean that we are accessing memory.

2. Example Program (Adding 3 Numbers)

PREPARED BY ALMAS AFZAL

```
[org 0x0100]
mov ax, [num1]    ; load first number
mov bx, [num2]    ; load second number
add ax, bx        ; sum
mov bx, [num3]    ; load third number
add ax, bx        ; add to sum
mov [num4], ax    ; store result in memory
mov ax, 0x4c00    ; terminate
int 0x21
```

num1: dw 5
num2: dw 10
num3: dw 15
num4: dw 0

3. Key Points

- Labels (like num1, num2) represent **memory addresses**.
- Assembler replaces labels with actual addresses during program execution.
- Data stored with dw (word = 2 bytes).

4. Optimized Example – Using One Label

num1: dw 5, 10, 15, 0

We can access numbers using offsets:

num1 → first value
num1+2 → second value
num1+4 → third value
num1+6 → result location

This saves space (no need for separate labels like num2, num3, num4).

5. Storing Multiple Values

We can declare multiple words in one line:
num1: dw 5, 10, 15, 0
This stores values in consecutive memory locations.

6. Adding Directly into Memory

We can also add numbers directly into memory:

PREPARED BY ALMAS AFZAL

```
mov ax, [num1]      ; first number
mov [num1+6], ax    ; store in result
mov ax, [num1+2]    ; second number
add [num1+6], ax    ; add to result
mov ax, [num1+4]    ; third number
add [num1+6], ax    ; add to result
```

Here, [num1+6] is a memory variable that stores the sum.
Important: Destination can be register or memory, but not both memory at the same time.

7. Rule of Operands in Assembly

✔ Allowed:

Register → Register
Register → Memory
Memory → Register
Constant → Register
Constant → Memory

✖ Not Allowed:

Memory → Memory
Constant → Constant
Memory → Constant

Only one operand can be in brackets ([]) per instruction.

2.3. SIZE MISMATCH ERRORS

1. What is a Size Mismatch?

If we declare data with **dw (word)** but access it as a **byte (or vice versa)**, the program may still run but results will be wrong.

Example mistake:
If data is declared with db but accessed using a word register (AX), the CPU may pick two bytes together, giving wrong results.

2. Example (Byte vs Word)

PREPARED BY ALMAS AFZAL

```
num1: db 5, 10, 15, 0   ; each number = 1 byte
mov al, [num1]          ; read 1st number into AL
mov bl, [num1+1]        ; read 2nd number into BL
add al, bl              ; add them
mov bl, [num1+2]        ; read 3rd number
add al, bl              ; add it
mov [num1+3], al        ; store result
```

Correct because:

- Data is defined as bytes (db)
- Registers used (al, bl) are also byte-sized.
- Offsets increase by 1 for each byte.

3. Programmer's Responsibility

- The assembler does not check logical mistakes.
- It only stops illegal instructions (syntax errors).
- Matching data size (db vs dw) with the register size (al vs ax) is the programmer's duty.

4. Register Size Rule

Example:

mov ax, [num1] → word move (because AX = 16 bits).
mov al, [num1] → byte move (because AL = 8 bits).
If sizes don't match (like mov ax, bl) → assembler gives error.

5. Ambiguous Instructions

Some instructions are unclear (assembler can't guess size).

Example:

mov [num1], 5 ; ✕ ambiguous

Is 5 a **byte (05h) or a word** (0005h)?

Assembler doesn't know, so we must specify:

```
mov byte [num1], 5   ; store 5 as byte
mov word [num1], 5   ; store 5 as word
```

2.4. REGISTER INDIRECT ADDRESSING

1. Why Register Indirect Addressing?

PREPARED BY ALMAS AFZAL

If we have many numbers (e.g., 100), direct addressing becomes too long (hundreds of instructions).
Solution → use a register to hold memory address → so we can move to next location easily.

2. Registers That Can Hold Addresses

In 8088 (iAPX88) processor, only these 4 registers can hold addresses:
BX, BP, SI, DI
These are used for memory access with square brackets [ ].

3. Example (Adding 3 Numbers)

```
mov bx, num1    ; BX = address of num1
mov ax, [bx]    ; load first number from memory
add bx, 2       ; move to next word (2 bytes)
add ax, [bx]    ; add second number
add bx, 2       ; move to next word
add ax, [bx]    ; add third number
add bx, 2       ; move to result location
mov [bx], ax    ; store sum
```

Explanation:
mov bx, num1 → loads address of num1 into BX (not value).
[bx] → means **"use the address inside BX to access memory."**
add bx, 2 → next word (since word = 2 bytes).

This way, BX moves from one number to the next → no fixed addresses.

4. Looping Example (Adding 10 Numbers)

```
mov bx, num1     ; BX = address of first number
mov cx, 10       ; CX = counter (10 numbers)
mov ax, 0        ; AX = sum (initialize)
l1: add ax, [bx] ; add number to AX
   add bx, 2     ; move to next number
   sub cx, 1     ; decrease counter
   jnz l1        ; repeat until CX = 0
mov [total], ax  ; save result
```

Explanation:
AX = holds sum.
BX = holds changing memory address.
CX = counter (controls how many times loop runs).
jnz = jump if not zero (repeats loop until CX = 0).

This forms a loop → very powerful in Assembly.

5. Important Points

Square brackets matter:
mov bx, num1 → loads address.
mov ax, [bx] → loads value from that address.

Register roles:

BX/BP → based addressing.
SI/DI → indexed addressing.

Both methods are similar, just different names.

2.5. REGISTER + OFFSET ADDRESSING

1. What is Register + Offset Addressing?

- A mix of direct addressing + register indirect addressing.
- Instead of storing the full memory address in register, we keep only the index/offset in a register (like BX).
- At runtime, the base address of array + register offset gives the actual memory location.

2. Why Use It?

- Same register can work as index for arrays.
- Easier to move through arrays by changing index only.
- Can be reused for different arrays without storing their full addresses.

3. Example (Adding 10 Numbers)

PREPARED BY ALMAS AFZAL

```
[org 0x0100]
mov bx, 0       ; BX = array index = 0
mov cx, 10      ; CX = counter = 10 numbers
mov ax, 0       ; AX = sum = 0

l1: add ax, [num1+bx] ; add array element to AX
   add bx, 2       ; move to next element (word = 2 bytes)
   sub cx, 1       ; decrease counter
   jnz l1          ; repeat until CX = 0

mov [total], ax   ; save result in memory
mov ax, 0x4c00    ; exit program
int 0x21

num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
total: dw 0
```

4. Step-by-Step Understanding

mov bx, 0 → BX holds index (0), not the address.
add ax, [num1+bx] → actual address = base address of num1 + BX.
add bx, 2 → BX moves in steps of 2 (because array elements are words = 2 bytes each).
sub cx, 1 + jnz → creates loop until all 10 numbers are added.

5. Types of Register + Offset

- If BX or BP used → called Base + Offset.
- If SI or DI used → called Index + Offset.

2.6. SEGMENT ASSOCIATION

1. Effective Address

- In addressing modes (like base + offset), the final result after addition is called **Effective Address (EA)**.
- EA = Offset inside a segment (not the full memory address).
- To access real memory, EA must combine with a Segment.
- **Physical Address = Segment × 16 + Effective Address**

2. Segments and Their Role

Memory in iAPX88 is divided into segments.
Important segment registers:
CS → Code Segment (for instructions)
DS → Data Segment (for data)
SS → Stack Segment (for stack)

PREPARED BY ALMAS AFZAL

ES → Extra Segment (for extra data transfer)

3. Default Segment Associations

Each register has a default segment:
CS ↔ IP (locked, cannot be changed)
SS ↔ SP / BP (locked, cannot be changed for SP, default for BP)
DS ↔ BX, SI, DI (can be changed)

4. Segment Override

Normally, registers use their default segment.
If we want to use a different segment temporarily, we use Segment Override Prefix.
Example:
mov ax, [cs:bx] ; BX will use CS instead of DS

After this instruction, default association comes back automatically.
Prefixes are special bytes (e.g., CS = 2Eh, ES = 26h).

5. COM Files and Overlapping Segments

- In COM files (DOS programs), all segments (CS, DS, SS, ES) have the same value.
- This means code, data, and stack are all inside one segment.
- That's why in examples, programs work fine without explicitly mentioning segments.

2.7. ADDRESS WRAPAROUND

What is Wraparound?

Wraparound happens when memory addressing goes beyond the limit and then starts again from the beginning, just like a circle.

There are two types:

1. Segment Wraparound

- Happens inside a segment during effective address calculation.
- If the addition of offset produces a carry, that carry is dropped.
- This makes the address restart from the beginning of the segment.

2. Physical Memory Wraparound

- Happens when the final physical address is too large for the address bus.
- 8088/8086 have 20-bit address bus → maximum address = 1 MB.
- If calculation produces 21 bits, the extra bit (carry) is dropped.
- This makes the address wrap around to the start of memory (00000h).

PREPARED BY AIMAS AFZAL

2.8. ADDRESSING MODES SUMMARY

Addressing Modes (iAPX88)

Operands in Assembly can be: **Immediate, Register, or Memory.**
For memory operands, 7 addressing modes are used.

General form:
Base + Index + Offset

- ❖ One, two, or none of them can be skipped.
- ❖ Must use 16-bit registers (e.g., BX, BP, SI, DI).
- ❖ Cannot use 8-bit registers (like BL, BH).
- ❖ Cannot add two bases or two indexes (BX+BP ✕, SI+DI ✕).

1. Direct Addressing

Address is given directly.
Example:
mov [1234], ax → store AX at address 1234.
mov [1234], al → store AL at address 1234.

2. Base Register Indirect

Address is inside a base register (BX or BP).
Example:
mov [bx], ax → store AX at address in BX (Data Segment).
mov [bp], al → store AL at address in BP (Stack Segment).

3. Index Register Indirect

Address is inside an index register (SI or DI).
Example:
mov [si], ax → store AX at address in SI (Data Segment).
mov [di], ax → store AX at address in DI (Data Segment).

4. Base + Offset

Base register + constant offset.
Example:
mov [bx+300], ax → BX + 300 = address.
mov [bp+300], ax → BP + 300 = address (Stack Segment).

5. Index + Offset

Index register + constant offset.
Example:
mov [si+300], ax → SI + 300 = address.

PREPARED BY ALMAS AFZAL

mov [di+300], al → DI + 300 = address.

6. Base + Index
One base register + one index register.
Example:
mov [bx+si], ax → BX + SI = address.
mov [bp+di], al → BP + DI = address (Stack Segment).
Useful for 2D arrays (base = row, index = column).

7. Base + Index + Offset
Base + Index + constant offset.
Example:
mov [bx+si+300], ax → BX + SI + 300 = address.
Used for complex array access (2D with offset).

Chapter 3
Branching

3.1. COMPARISON AND CONDITIONS

1. CMP Instruction
- CMP = Compare (subtracts SRC from DEST, but does not change operands).
- Only updates FLAGS (ZF, CF, SF, OF).
- Always used before a conditional jump.

2. How CMP Works
- If DEST - SRC = 0 → ZF = 1 (equal).
- If borrow needed (unsigned smaller) → CF = 1.
- If signed result negative → SF set.
- If signed overflow occurs → OF set.

3. Signed vs Unsigned Comparisons
Unsigned numbers: Range → 0 … 65535.
Use JA, JAE, JB, JBE (Above/Below).
Signed numbers: Range → -32768 … 32767.
Use JG, JGE, JL, JLE (Greater/Less).

Same binary data can mean different results depending on signed/unsigned interpretation.

4. Flag-Based Results (Important Cases)

PREPARED BY ALMAS AFZAL

Equal (DEST = SRC): ZF = 1.

Unsigned:
DEST < SRC → CF = 1.
DEST ≤ SRC → ZF = 1 OR CF = 1.
DEST ≥ SRC → CF = 0.
DEST > SRC → ZF = 0 AND CF = 0.

Signed:
DEST < SRC → SF ≠ OF.
DEST ≤ SRC → ZF = 1 OR SF ≠ OF.
DEST ≥ SRC → SF = OF.
DEST > SRC → ZF = 0 AND SF = OF.

5. Key Idea

- ✧ CMP only subtracts → meaning depends on which conditional jump you use next.
- ✧ JA/JB → unsigned checks.
- ✧ JG/JL → signed checks.
- ✧ Wrong jump instruction = wrong result.

3.2. CONDITIONAL JUMPS

- ❖ After a CMP instruction, processor sets flags (ZF, CF, SF, OF, PF).
- ❖ Conditional jumps check these flags to decide whether to jump or not.
- ❖ They are used for decision making (if, else, loops) in Assembly.
- ❖ Many jumps have different names but same meaning/opcode (renaming is done by Intel for readability).

Common Conditional Jumps

1. Equality Jumps

JE / JZ → Jump if Equal / Zero (ZF = 1)
JNE / JNZ → Jump if Not Equal / Not Zero (ZF = 0)

2. Unsigned Comparisons

JA / JNBE → Jump if Above (ZF = 0 AND CF = 0)
JAE / JNB → Jump if Above or Equal (CF = 0)
JB / JNAE → Jump if Below (CF = 1)
JBE / JNA → Jump if Below or Equal (ZF = 1 OR CF = 1)

3. Signed Comparisons

JG / JNLE → Jump if Greater (ZF = 0 AND SF = OF)
JGE / JNL → Jump if Greater or Equal (SF = OF)
JL / JNGE → Jump if Less (SF ≠ OF)
JLE / JNG → Jump if Less or Equal (ZF = 1 OR SF ≠ OF)

PREPARED BY ALMAS AFZAL

4. Flag Specific Jumps

JC → Jump if Carry (CF = 1)
JNC → Jump if Not Carry (CF = 0)
JO → Jump if Overflow (OF = 1)
JNO → Jump if Not Overflow (OF = 0)
JS → Jump if Sign (SF = 1 → negative)
JNS → Jump if Not Sign (SF = 0 → positive)
JP / JPE → Jump if Parity Even (PF = 1)
JNP / JPO → Jump if Parity Odd (PF = 0)

5. Special Jump

JCXZ → Jump if CX = 0 (used with loops, independent of Zero Flag).

3.3. UNCONDITIONAL JUMP

Problem: If we put data at the start of code (offset 0100), the debugger tries to execute that data as instructions → gives wrong results.

Solution: Use JMP (Unconditional Jump) at the beginning.

- It always jumps (ignores all flags).
- We use it to skip over data and directly start code execution.

Syntax

jmp label

- ❖ CPU jumps directly to label.
- ❖ Flags are not checked.

Key Points

- ❖ Unconditional Jump (JMP) = always executes.
- ❖ Used at the beginning of a program if data is placed at offset 0100h.
- ❖ Makes sure first instruction is valid, not data.

So,

Conditional jumps = decision making (based on flags).
Unconditional jump = always jumps, mostly used to skip data.

3.4. RELATIVE ADDRESSING

When we write:
jmp start

- ❖ In debugger, it may show like JMP 0119 (jumping to address 0119).
- ❖ But in machine code, operand is not 0119 (absolute address).

PREPARED BY ALMAS AFZAL

- Instead, it stores how far to jump (forward/backward) from the current IP (Instruction Pointer).

How it works.?

Example: Opcode → F21600

- F2 = opcode for jump
- 1600 = offset (in little-endian, means 0016)
- At 0100, after fetching the first instruction, IP = 0103.
- Add 0016 → 0119.
- So CPU jumps to 0119.

Key Idea

- Relative addressing = instruction gives a distance (relative offset), not a fixed address.
- Absolute addressing = gives the exact address.
- In assembly, we just use labels (jmp start).
- Assembler + CPU handle all offset math automatically.

Why important?

Makes code relocatable → program can run correctly even if loaded at a different memory address.
We don't calculate offsets manually, assembler does it.

3.5. TYPES OF JUMP

There are 3 types of jumps in Assembly:

1. Short Jump
2. Near Jump
3. Far Jump

1. Near Jump

- Uses a 16-bit offset (relative address).
- Can jump anywhere within the same segment.
- If the offset is large, it wraps around inside the segment.

Example instruction forms:

```
EB disp       ; short form
E9 disp_low disp_high  ; near form
```

2. Short Jump

- Uses an 8-bit signed offset (1 byte only).
- Range: -128 to +127 bytes from current IP.

PREPARED BY ALMAS AFZAL

- If offset is negative → CPU moves backward.

Example: 75 F2
75 = opcode
F2 = offset (signed byte).
Conditional jumps can only be short.
Unconditional jumps can be short, near, or far.

3. Far Jump

- Not relative → it is absolute addressing.
- Needs segment + offset (2 bytes each = 4 bytes).
- Used to jump outside the current segment (to another code segment).
- Updates both CS (Code Segment) and IP (Instruction Pointer).
- Only JMP, CALL, RET instructions support far form.

Difference b/w these Jumps:

Jump Type	Size of Instruction	Range	Works Across Segments?
Short	2 bytes	-128 to +127 bytes	No
Near	3 bytes	Full segment (64 KB)	No
Far	5 bytes	Whole memory	Yes

3.6. SORTING EXAMPLE

What is Sorting?

1. **Ascending order:** Smallest → Largest
2. **Descending order:** Largest → Smallest

Bubble Sort Algorithm

- Compare two consecutive elements.
- If not in order → swap them.
- Repeat for whole list = 1 pass.
- Need up to N passes for N elements.
- If no swap occurs in a pass → stop (list is sorted).

Assembly Code (Important Points)

```
data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
swap: db 0       ; flag to check if any swap happened
```

BX → index for array.
AX → first number, compared with next.
DX → used for swapping values.
swap flag → if no swap = sorted.

PREPARED BY ALI G AFZAL

JBE (Jump if Below or Equal) → used for comparison in ascending sort.
If changed to JAE → descending sort.

Difference Between Signed & Unsigned Sorting

Instruction	Interpretation	Result with -40, -35
JBE	Unsigned	-40, -35 treated as very large numbers (go to end).
JLE	Signed	-40, -35 treated as negative (come at start).

Chapter 4
Bit Manipulations

PREPARED BY ALMAS AFZAL

4.1. MULTIPLICATION ALGORITHM

Multiplication is repeated addition using binary rules.

In binary:
If multiplier digit = 1 → write multiplicand.
If multiplier digit = 0 → write 0000.
Each next step → shift multiplicand left (like adding crosses in decimal multiplication).
Finally → add all partial results to get the product.

Example (13 × 5)

- Multiplicand = 1101 (13)
- Multiplier = 0101 (5)
- Expected Result = 01000001 (65)

Important Observations

Binary advantage:
- If digit = 0 → result is 0.
- If digit = 1 → result is multiplicand itself.
- No real multiplication required.

Shifting Rule:
- 1st bit → no shift.
- 2nd bit → shift once.
- 3rd bit → shift twice.
- 4th bit → shift thrice.

Answer size:
If both inputs are 4 bits → result can be up to 8 bits.

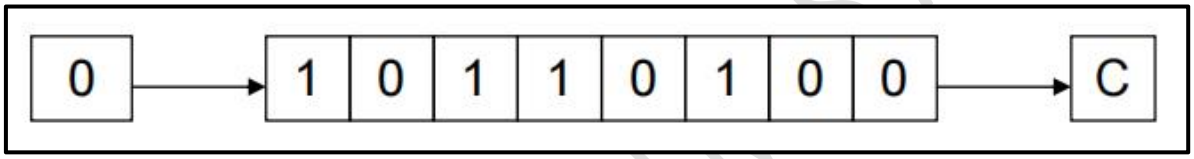
Requirements for Assembly Algorithm

- ❖ Need bit checking → to read multiplier bits one by one.
- ❖ Need shift instructions → to shift multiplicand left for each step.
- ❖ Need addition → to add partial products into final result.

4.2. SHIFTING AND ROTATIONS

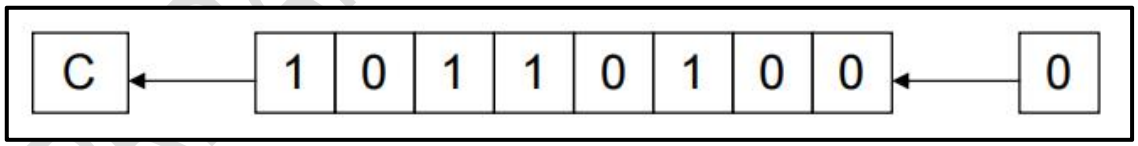
1. SHR (Shift Logical Right)

- ➢ Inserts 0 from left, moves all bits to the right.
- ➢ Rightmost bit → goes to Carry Flag (CF).
- ➢ Example use: divide unsigned numbers by 2.



2. SHL / SAL (Shift Logical Left / Shift Arithmetic Left)

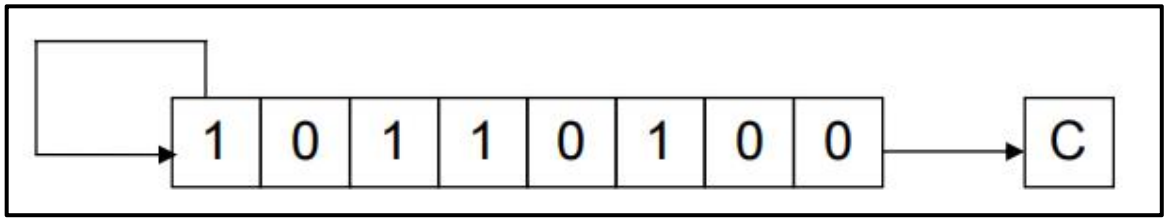
- ➢ Inserts 0 from right, moves all bits to the left.
- ➢ Leftmost bit → goes to CF.
- ➢ SHL and SAL are same.
- ➢ Equivalent to multiplying by 2.
- ➢ Overflow can happen if important bit is lost (check with JO).



3. SAR (Shift Arithmetic Right)

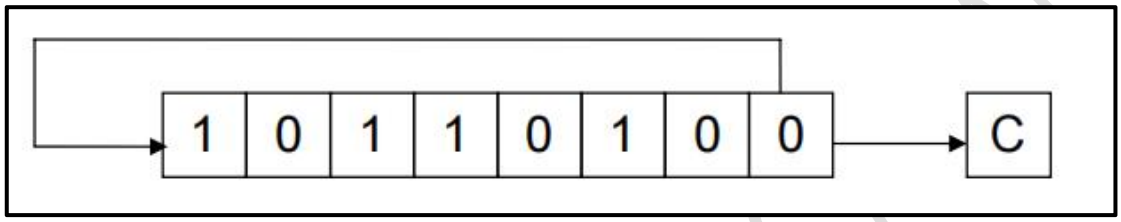
- ➢ For signed numbers (sign bit must not change).
- ➢ Copies MSB (sign bit) while shifting right.
- ➢ Rightmost bit → goes to CF.
- ➢ Equivalent to dividing signed numbers by 2.
- ➢ Correct for negatives (unlike SHR).

PREPARED BY ALI S AFZAL



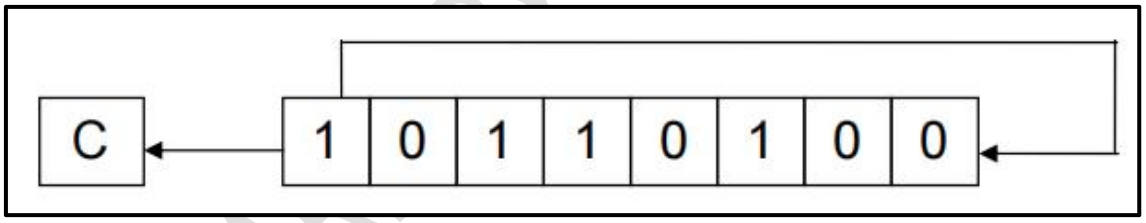
4. ROR (Rotate Right)

- All bits move right, last bit comes back to left side.
- That last bit is also copied to CF.
- Like a circular pipe, bits keep rotating.



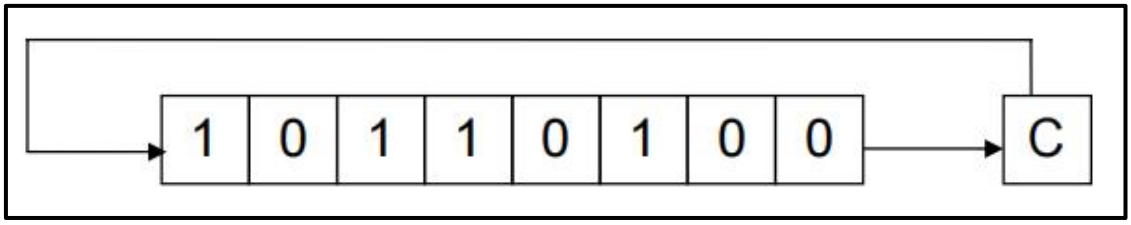
5. ROL (Rotate Left)

- Opposite of ROR.
- All bits move left, MSB comes back to right side.
- MSB also copied to CF.



6. RCR (Rotate through Carry Right)

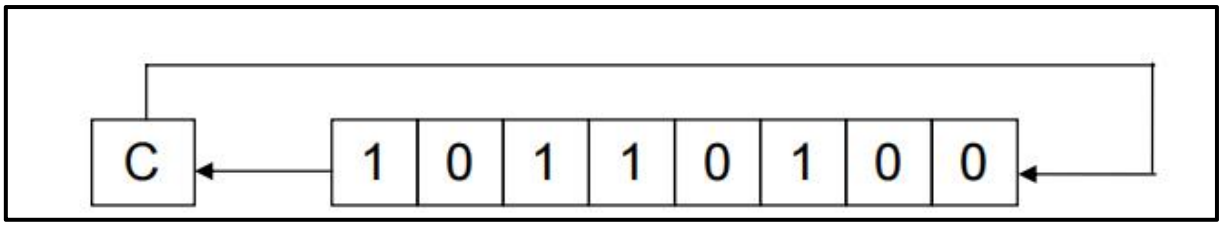
- Similar to ROR but includes Carry Flag as part of rotation.
- Carry enters from left, rightmost bit goes into CF.
- Makes a 9-bit (8+CF) or 17-bit (16+CF) rotation.



7. RCL (Rotate through Carry Left)

- Opposite of RCR.
- Carry enters from right, MSB goes into CF.

- Also 9-bit or 17-bit rotation including carry.



Key Points

- SHL/SAL = Multiply by 2
- SAR = Divide signed numbers by 2
- SHR = Divide unsigned numbers by 2
- ROL/ROR = Normal rotation
- RCL/RCR = Rotation with carry included

4.3. MULTIPLICATION IN ASSEMBLY LANGUAGE

Multiplication in binary works like normal multiplication, but instead of writing all intermediate results, we add them step by step.

Algorithm:

- Set result = 0.
- Check rightmost bit of multiplier:
- If 1, add multiplicand to result.
- If 0, skip addition.
- Left shift multiplicand (×2).
- Right shift multiplier (/2).
- Repeat steps according to number of bits (4, 8, 16, etc.).

Key points:

- SHR (Shift Right): divides by 2, rightmost bit → CF.
- SHL (Shift Left): multiplies by 2.
- CF (Carry Flag): tells us if multiplier's bit = 1.
- JC (Jump if Carry): used to test multiplier bit.
- DEC (Decrement): reduces counter by 1.
- JNZ (Jump if Not Zero): repeats loop until counter = 0.

Example (13 × 5):

PREPARED BY ALMAS AFZAL

```
   1101 = 13        Accumulated Result
   0101 = 5
   -----                          0 (Initial Value)
   1101 = 13        0 + 13  = 13
  0000x = 0         13 + 0  = 13
 1101xx = 52        13 + 52 = 65
0000xxx = 0         65 + 0  = 65 (Answer)
```

4.4. EXTENDED OPERATIONS

1. Why Extension is Needed?

- In 4-bit multiplication, we could safely use 8-bit registers.
- But for 16-bit × 16-bit = 32-bit result, we need a way to:
- Shift 32-bit numbers (since the multiplicand grows with left shifts).
- Add 32-bit numbers (because the accumulated result can exceed 16 bits).
- If we only used 16-bit registers, important bits would be lost during shifting or addition.

2. Extended Shifting

- CPU instructions like SHL/SHR only work on 8-bit or 16-bit registers.
- To shift a 32-bit number, we split it into two 16-bit words:
- Lower word (least significant)
- Upper word (most significant)

Algorithm for Left Shift (example):

```
shl word [num1], 1     ; shift lower 16 bits
rcl word [num1+2], 1   ; rotate carry into upper 16 bits
```

The dropped MSB from the lower half is passed into the upper half via carry flag.

Algorithm for Right Shift:

```
shr word [num1+2], 1   ; shift upper 16 bits right
rcr word [num1], 1     ; rotate carry into lower 16 bits
```

This lets us shift numbers of any size (32, 64, 1024 bits) by repeating the logic.

3. Extended Addition (ADC)

- Normal addition (ADD) can't handle the carry beyond 16 bits.
- Extended addition uses ADD + ADC:
- ADD lower halves.
- ADC upper halves (includes carry from lower addition).

PREPARED BY ALMAS AFZAL

Example (32-bit addition):

```
mov ax, [src]        ; lower word
add [dest], ax       ; add lower halves
mov ax, [src+2]       ; upper word
adc [dest+2], ax      ; add upper halves with carry
```

Similarly, SBB (subtract with borrow) is used for extended subtraction.

4. Extended Multiplication

- The multiplier (16 bits) is shifted right to test each bit.
- The multiplicand (16 bits) is stored in 32-bit space so left shifting doesn't lose bits.
- On every '1' bit of multiplier: add multiplicand to result using extended addition.
- Shift multiplicand left using extended shifting.

Important Observations:

- Result of 1300 × 500 = 650000 (0x09EB10).
- Multiplicand after 16 left shifts → 05140000 (as expected).
- Extended shifting + extended addition make large multiplications possible.

So in short:

- Extended shifting = SHL/SHR + RCL/RCR to handle >16 bits.
- Extended addition = ADD + ADC to handle >16 bits.
- Extended multiplication = combination of above two.

4.5. BITWISE LOGICAL OPERATIONS

Bitwise Logical Operations

The 8088 processor provides bit-wise operations.
This means the operation works on each bit separately of the operands (byte or word).

The four main operations are:

1. AND
2. OR
3. XOR
4. NOT

1. AND Operation

Rule: Result bit = 1 only if both bits are 1. Otherwise 0.

PREPARED BY ALMAS AFZAL

Truth Table:

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

Example: AND AX, BX → Each bit of AX and BX is ANDed.
Use: Often used for masking (keeping some bits and clearing others).

2. OR Operation

Rule: Result bit = 1 if any one or both bits are 1.

Truth Table:

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

Example: OR AX, BX
Use: Used to set bits to 1 without affecting other bits.

3. XOR Operation (Exclusive OR)

Rule: Result bit = 1 if bits are different (one is 1, other is 0).

Truth Table:

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

Example: XOR AX, BX
Use: Useful in encryption, cryptography, image processing.
Property: Reversible (If A XOR B = C, then C XOR B = A).

4. NOT Operation

Rule: Inverts each bit (forms one's complement).

0 → 1
1 → 0

Example: NOT AX
Use: Used to flip all bits.

4.6. MASKING OPERATIONS

1. Selective Bit Clearing (AND Operation)

Purpose: To make some bits zero while keeping others unchanged.
Method: Use a mask with 1 where bits should remain same and 0 where bits should be cleared.

2. Selective Bit Setting (OR Operation)

Purpose: To make some bits 1 while keeping others unchanged.
Method: Use a mask with 0 where values remain same and 1 where bits should be set.

3. Selective Bit Inversion (XOR Operation)

Purpose: To invert (toggle) specific bits.
Method: Mask with 1 at positions to be inverted, 0 where unchanged.
Difference from NOT: NOT inverts all bits, but XOR inverts only selected bits.

4. Selective Bit Testing (AND / TEST Instruction)

Purpose: To check if a specific bit is set (1) or clear (0).

Method 1: Use AND with a mask (but it destroys destination value).
Method 2 (better): Use TEST instruction (non-destructive AND).
TEST does not change values, only sets flags.

Key Instruction Notes:

- AND → clear bits.
- OR → set bits.
- XOR → invert bits.
- TEST → check bits (non-destructive).

PREPARED BY ALMAS AFZAL

Chapter 5
Subroutines

5.1. PROGRAM FLOW

Problem of Code Reuse
- In assembly, we may need the same algorithm (like multiplication or sorting) many times in a program.
- Writing the same code again and again is not practical (e.g., 100 times).
- We need a way to write code once and use it many times.

Jumps (JMP) Problem
- Normal JMP instruction only makes a permanent jump.
- Once we jump, the processor doesn't know where to return.
- Example: If we jump to bubble sort from 100 different places, how will it know where to go back?
- So, JMP cannot be used for temporary diversions.

Need for Temporary Diversion
- We need a way to go to some code, execute it, and return back to the exact place from where we came.
- Like a "roundabout" on the road → you return back to where you started.

CALL and RET Instructions
CALL: Used to temporarily jump to a subroutine (function).
RET: Used to return back to the instruction after CALL.
Together, CALL and RET make code reusable.
Example: We can write bubble sort once, then CALL it whenever needed.

Technical Notes
- CALL changes IP (Instruction Pointer) and SP (Stack Pointer).
- RET uses the saved address to return back correctly.
- CALL and RET are usually used together, but they can also be used alone in special cases.

Parameters
Problem: If we CALL bubble sort, it will always sort the same array.
In real programs, we need flexibility (different arrays, ascending/descending, signed/unsigned).
The values passed from the caller to the subroutine are called **parameters**.
Parameters can be passed using **registers**.

PREPARED BY ALMAS AFZAL

In short:

JMP = permanent diversion (no return).
CALL + RET = temporary diversion (with return).
Parameters = extra information given to subroutine (via registers, etc.).

5.2. OUR FIRST SUBROUTINE

What is a Subroutine?

A subroutine is a reusable piece of code (like a function in C).
We use CALL to go to it and RET to come back.
Example: Bubble Sort written as a subroutine.

Problem (Register Destruction)

- Subroutine uses AX, CX, DX, SI for sorting.
- After returning, these registers no longer hold original values.
- Caller cannot rely on register values after a CALL.
- In large programs, this becomes a big issue.

Solution (coming next): Use the system stack to save and restore registers.

5.3. STACK

What is Stack?

Stack = special memory area used in LIFO (Last In First Out) way.
Only the top element can be added or removed.
Push → put (save) value on stack.
Pop → take (restore) value from stack.

Example: Like balls in a test tube → last ball in, comes out first.

Stack in 8088 Processor

Works with words (2 bytes), not single bytes.
Decrementing stack → new items stored at lower memory addresses.

Controlled by:

1. SS (Stack Segment) = tells memory segment of stack.
2. SP (Stack Pointer) = shows top of stack.

After PUSH, SP = SP – 2.
After POP, SP = SP + 2.

Main Uses of Stack

- Saving registers before subroutine → so values are not lost.

PREPARED BY ALMAS AFZAL

- CALL instruction → saves IP (next instruction address) on stack.
- RET instruction → takes back IP from stack → program continues after CALL.
- RET n → returns and also clears extra parameters (n bytes).

Key Points to Remember

- Stack works like temporary storage.
- CALL and RET automatically use stack.
- PUSH and POP let us manually save/restore registers.
- Order of PUSH/POP is very important.
- Helps in solving the problem of register destruction in subroutines.

5.4.SAVING AND RESTORING REGISTERS

- Subroutines often change registers (AX, CX, SI etc.), which may disturb the calling code.
- To avoid this, registers must be saved before use and restored after use.

How to Save & Restore

- Use PUSH to save register value on the stack.
- Use POP to restore register value from the stack.
- Order matters: If AX, CX, SI are PUSHed, they must be POPped in reverse order (SI → CX → AX).

Example

In Bubble Sort subroutine:

- AX, CX, SI are saved with PUSH.
- They are restored with POP before returning.
- Caller sees registers unchanged.

Instructions

PUSH

- Decreases SP (Stack Pointer) by 2.
- Copies source register → stack.
- Used for saving registers or passing parameters.

POP

- Copies data from stack → destination register.
- Increases SP by 2.
- Used to restore registers.

CALL

- Transfers control to a subroutine (procedure).
- Saves return address (IP, CS) on the stack.

PREPARED BY ALMAS AFZAL

Two types:

1. Near Call (Intra-segment) → within same segment.
2. Far Call (Inter-segment) → different segment.

RET (Return)

- ❖ Transfers control back to caller after subroutine.
- ❖ Pops saved return address (IP, CS) from stack.
- ❖ RETF used for far return (inter-segment).

Key Point:

Always save registers with PUSH before using them in subroutines, and restore them with POP before returning, so caller's registers remain safe.

5.5. PARAMETER PASSING THROUGH STACK

Why use Stack for Parameters?

- ➢ Registers are limited (only 7 general registers).
- ➢ Subroutines may need more parameters → registers are not enough.
- ➢ Stack is expandable and can hold many parameters.
- ➢ Parameters stay safe in stack even across function calls.

How Parameters are Passed?

- ➢ Caller pushes parameters on stack.
- ➢ Then subroutine is called → return address is also pushed on top.
- ➢ So, parameters are below the return address.

Accessing Parameters

- ❖ We cannot simply POP parameters (because first POP gives return address).
- ❖ Solution: use Base Pointer (BP) to "peek" inside stack.
- ❖ BP is linked with Stack Segment (SS).

Steps:

Save old BP → push bp
Copy SP into BP → mov bp, sp

Now:

[bp] = old BP
[bp+2] = return address
[bp+4] = 2nd parameter
[bp+6] = 1st parameter

This makes accessing parameters easy without disturbing SP.

Stack Clearing

Parameters must be removed after subroutine returns (to avoid stack overflow).

PREPARED BY ALMAS AFZAL

Two ways:
1. Caller clears stack → needs extra instructions every time.
2. Callee clears stack (common way) → uses RET n to remove parameters automatically.

Advantage: avoids repetition, cleaner code.

5.6. LOCAL VARIABLES

Local Variables = Variables used only inside a subroutine.
They do not need permanent memory like global variables.
Created when subroutine runs, deleted after it ends → memory is reused.

Why use Stack for Local Variables?
Stack is the best place for temporary/local variables.
A gap in the stack is made for them by decreasing SP.

Example:

```
push bp      ; save old BP
mov bp, sp   ; set BP as reference
sub sp, 2    ; create 2-byte gap (for 1 word local variable)
```

This gap acts like local variable storage.

Accessing Variables
Parameters → accessed with positive offsets (e.g., [bp+4], [bp+6]).
Local variables → accessed with negative offsets (e.g., [bp-2]).

Stack Cleanup
After subroutine ends → remove gap.
Done by restoring stack pointer:

```
mov sp, bp   ; restore old SP (preferred)
pop bp
```

Alternative: add sp, 2 (but not recommended, since we may forget size).

Key Points
- Local variables exist only during subroutine execution.
- They save memory compared to permanent (global) variables.

Access:
1. Parameters = [bp+4], [bp+6] (positive)

PREPARED BY ALMAS AFZAL

2. Locals = [bp-2], [bp-4]… (negative)

Example: Swap flag in Bubble Sort is stored as local variable.

Chapter 6
Display Memory

6.1. ASCII CODES

What is ASCII?

- **ASCII = American Standard Code for Information Interchange.**
- Standard numeric representation of characters → ensures 'A' on one computer is 'A' on every computer.
- Used for global information exchange (Internet, files, etc.).

How it Works.?

- Keyboard sends number → Computer interprets it → VGA card shows shape → Monitor glows pixels.
- Example: ASCII code of 'A' = 65 (decimal) or 41h (hex).
- Display, printers, other computers all understand same code.

Types

1. Standard ASCII

- 128 characters (0–127).
- Includes alphabets, numbers, symbols, and control characters.

2. Extended ASCII (IBM PC)

- Adds 128 more characters (128–255).
- Used for line-drawing, corners, and special characters (since early PCs had no graphics mode).
- Not an official standard, but became widely accepted.

In short:

ASCII is a universal numeric code for characters. Standard ASCII has 128 symbols, Extended ASCII has 256. Characters are arranged contiguously (A–Z, a–z, 0–9), which makes operations like case conversion easy.

6.2. DISPLAY MEMORY FORMATION

PREPARED BY ALMAS AFZAL

The computer shows characters on the screen using **ASCII codes.** For example, if we send 0x41 (which is ASCII for 'A') to the VGA card, it will light up pixels to display the letter A.

The computer treats the **video memory (VGA memory)** just like normal system memory. This means the CPU can read/write it directly. Whatever ASCII code is placed in video memory, its matching symbol appears on the screen.

The screen has 80 columns and 25 rows (80×25 = 2000 characters). Each screen character takes **2 bytes** in memory:

1. First byte → ASCII code of the character (like 'A').
2. Second byte → Attribute (color, brightness, blinking, etc.).

So:
First 80 words = first row of screen.
Next 80 words = second row, and so on.

This way, video memory (linear) is mapped to screen (2D).

Display Memory Base Address

To keep things standard, video memory is always mapped at address 0xB8000.

- The first byte here = top-left character of the screen.
- **Example**: If we put ASCII for 'A' at this location, 'A' will appear in the top-left corner.

Attribute Byte (Color Control)

The second byte (attribute) decides text color and effects:

Bit	Meaning
7	Blink (1 = blink)
6	Background Red
5	Background Green
4	Background Blue
3	Foreground Intensity
2	Foreground Red
1	Foreground Green
0	Foreground Blue

This allows **16 foreground** colors and **8 background** colors.

6.3. HELLO WORLD IN ASSEMBLY LANGUAGE

Declaring Characters / Strings

PREPARED BY ALMAS AFZAL

In assembly, characters are stored as their ASCII codes.
We don't need to remember ASCII codes because assembler allows simple syntax:

```
db 0x61, 0x62, 0x63   ; ASCII codes of a, b, c
db 'a', 'b', 'c'      ; same as above
db 'abc'              ; string form
```

So 'he**llo world'** is stored as ASCII codes automatically.

Program Explanation

First Assembly program prints **"hello world"**.

Steps:

String declared → message db **'hello world'**.
Length stored → dw 11.
Subroutine (clrscr) → clears the screen before printing.
Subroutine (printstr) → prints the string on screen.

Important Points (MCQs)

- Video memory address = 0xb800 (text mode).
- ES register points to video memory.
- DI (Destination Index) → points to screen location (top left).
- SI (Source Index) → points to string in memory.
- CX → stores string length (used for loop).
- AH → stores character attribute (color/brightness).
- AL → stores ASCII character.
- Together AX = [attribute + character].
- Loop instruction (loop nextchar) = repeat until CX = 0.
- After printing, program ends with int 21h.

6.4. NUMBER PRINTING IN ASSEMBLY

Problem

- In assembly, numbers are stored in binary form (e.g., 10 is stored as 00001010 in binary).
- But on the screen, we need to show numbers as characters (ASCII codes of digits '1' and '0').

- If we directly put 10 in video memory, it won't display "10", it will display a weird symbol (ASCII code of 10 = newline).

So we need a way to convert numbers into their ASCII representation before printing.

PREPARED BY ALMAS AFZAL

DIV Instruction (8088 CPU)

- DIV BL → divides 16-bit AX by 8-bit BL → result in AL, remainder in AH.
- DIV BX → divides 32-bit DX:AX by 16-bit BX → result in AX, remainder in DX.
- If result too big → divide overflow error occurs.

MCQs

- If mov bx, 10 → prints in decimal.
- If mov bx, 2 → prints in binary.
- If mov bx, 8 → prints in octal.
- Hexadecimal is trickier because digits A–F need extra handling.

6.5. SCREEN LOCATION CALCULATION

MUL Instruction (Multiply)

- MUL is used for unsigned multiplication.
- If operand is byte, multiply with AL → result in AX (AH:AL).
- If operand is word, multiply with AX → result in DX:AX.
- **Example:** used here to multiply row number by 80.

Parameters of a Program:

1. X position (column)
2. Y position (row)
3. Attribute (color/style)
4. Address of string
5. Length of string

*We calculate memory location from row & column, use MUL to handle multiplication, and store string + attribute in video memory. This allows printing text **anywhere** on the screen with **desired color**.*

BEST OF LUCK!

PREPARED BY ALMAS AFZAL