

UNIT 2

Introduction to Assembly Language Programming

(10 Hours) — Microprocessor and Computer Architecture (CACS155)

1. Assembly Language Programming Basics

Assembly language is a low-level, symbolic programming language in which each instruction corresponds directly to one machine-code instruction of the processor. Instead of writing raw binary/hex opcodes, the programmer uses mnemonics (short English-like abbreviations such as MOV, ADD, JMP) that are easier to read, write and debug than pure machine code, while still giving the programmer complete control over the hardware.

1.1 Statement Format

Every assembly language statement (except comments) is generally written using four fields, though not all of them are compulsory in every line:

Example: `START: MVI A, 32H ; load 32H into accumulator`

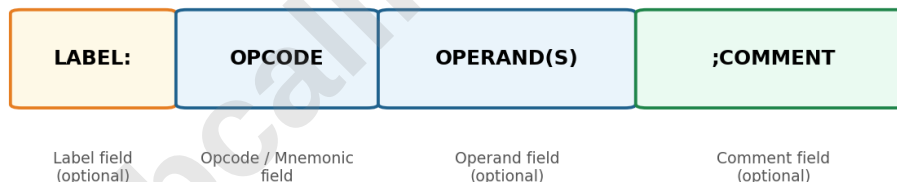


Figure 2.1 — Format of a typical assembly language statement

Field	Purpose
Label	An optional symbolic name given to a memory location/instruction so that it can be referenced (e.g., by a jump or call) elsewhere in the program.
Opcode (Mnemonic)	The symbolic operation code that specifies what operation is to be performed (e.g., MOV, ADD, JMP).
Operand(s)	The data or register(s)/address(es) on which the operation acts. Some instructions need none, one, or two operands.
Comment	Optional explanatory text preceded by a semicolon (;); ignored by the assembler but improves program readability.

1.2 Assembler Directives (Pseudo-Instructions)

Unlike actual instructions, assembler directives (also called pseudo-opcodes) do not generate machine code directly — they instruct the assembler itself on how to process the program.

Directive	Purpose
ORG (Origin)	Specifies the starting memory address from which the following code/data is to be stored.
END	Marks the physical end of the source program.
EQU (Equate)	Assigns a fixed numerical value to a symbolic label, used for constants.
DB (Define Byte)	Reserves and initializes one or more bytes of memory with given 8-bit data.
DW (Define Word)	Reserves and initializes memory with 16-bit (word) data.

Note: Assembler directives are instructions to the assembler program itself, not to the microprocessor — they generate no executable object code.

bcalink.com

2. Classification of Instructions and Addressing Modes

The 8085 instruction set (74 opcodes / 246 total instruction bytes counting variants) can be grouped functionally into five broad categories:

Category	Function	Examples
Data Transfer	Copies data between registers, or between a register and memory, without altering flags (except a few special cases).	MOV, MVI, LXI, LDA, STA, LHLD, SHLD
Arithmetic	Performs addition, subtraction, increment and decrement operations.	ADD, SUB, ADC, SBB, INR, DCR, INX, DCX
Logical	Performs bitwise AND/OR/XOR, comparison, rotate and complement operations.	ANA, ORA, XRA, CMP, RAL, RAR, CMA
Branching	Alters the normal sequential flow of program execution, conditionally or unconditionally.	JMP, JZ, JNZ, CALL, RET, RST
Machine Control	Controls processor operations such as halting or enabling interrupts; affects no data.	HLT, NOP, EI, DI, SIM, RIM

As introduced in Unit 1, every operand within these instructions is accessed using one of five addressing modes — recapped here with the emphasis on how each affects instruction length and execution speed:

Addressing Mode	How Operand is Located	Relative Speed
Immediate	Operand value is embedded in the instruction itself	Fast — no extra memory access needed
Register	Operand is inside a CPU register	Fastest — no memory access at all
Direct	Full 16-bit memory address given in the instruction	Slower — needs 3-byte instruction, one memory read/write
Indirect (Register Indirect)	Register pair (usually H-L) holds the address of the operand	Moderate — one extra memory access via HL
Implied	Operand is fixed by the opcode (usually the Accumulator)	Fastest — 1-byte instruction

3. 8085 Instruction Sets

This section summarises the most frequently used instructions from each functional group, along with their operation and typical byte/T-state cost.

3.1 Data Transfer Instructions

Mnemonic	Operation	Bytes
MOV Rd, Rs	Copy contents of source register Rs into destination register Rd	1
MVI R, data	Move an 8-bit immediate value into register R	2
LXI Rp, data16	Load a 16-bit immediate value into a register pair	3
LDA addr	Load the accumulator directly from a 16-bit memory address	3
STA addr	Store the accumulator's contents directly into a 16-bit memory address	3
LHLD addr / SHLD addr	Load/Store the H-L pair from/to memory	3
XCHG	Exchange the contents of H-L and D-E register pairs	1

3.2 Arithmetic Instructions

Mnemonic	Operation	Flags Affected
ADD R / ADD M	Accumulator = Accumulator + register/memory content	S, Z, AC, P, CY
ADI data	Accumulator = Accumulator + 8-bit immediate data	S, Z, AC, P, CY
SUB R / SUI data	Accumulator = Accumulator – register/immediate data	S, Z, AC, P, CY
INR R / DCR R	Increment / decrement register or memory content by 1	S, Z, AC, P (not CY)
INX Rp / DCX Rp	Increment / decrement a 16-bit register pair by 1	None
DAA	Decimal Adjust Accumulator — corrects result of BCD addition	S, Z, AC, P, CY

3.3 Logical and Branching Instructions

Mnemonic	Operation
ANA R / ANI data	Logical AND of Accumulator with register/immediate data
ORA R / XRA R	Logical OR / Exclusive-OR of Accumulator with register
CMP R / CPI data	Compare Accumulator with register/immediate data (sets flags only)
RAL / RAR	Rotate Accumulator left/right through Carry
JMP addr	Unconditional jump to the specified 16-bit address

Mnemonic	Operation
JZ / JNZ / JC / JNC addr	Conditional jump — taken if Zero/Not-Zero/Carry/No-Carry flag condition is met
CALL addr / RET	Call a subroutine (pushes return address on stack) / return from subroutine

Example 2.1

Write and analyze a short instruction sequence to add the contents of registers B and C, and store the result in memory location 3000H.

Solution

MOV A, B ; copy B into accumulator ($A = B$)

ADD C ; $A = A + C$

STA 3000H ; store result of A into memory address 3000H

HLT ; stop program execution

4. Assembling, Executing and Debugging the Programs

Converting a symbolic assembly language program into machine-executable binary code, and then verifying its correctness, follows a systematic sequence of steps:

Stage	Description
1. Writing Source Code	The programmer writes the mnemonic instructions in a text (.asm/.src) file, following the correct statement format.
2. Assembling	An Assembler program translates the mnemonics into machine code (object code), resolving labels/addresses and flagging syntax errors. Two-pass assemblers scan the source twice: once to build a symbol table of all labels, and again to generate the final object code.
3. Linking/Loading	The object code is loaded into memory starting at the address specified by the ORG directive, ready for execution.
4. Executing	The processor runs the program starting from the entry point, using registers and memory as defined by the instructions.
5. Debugging	Errors (syntax, logical, or runtime) are located and corrected using debugging tools/techniques such as single-stepping, breakpoints, and register/memory examination.

4.1 Types of Assemblers

- One-pass assembler — scans the source code only once; forward references (labels used before being defined) are harder to resolve.
- Two-pass assembler — Pass 1 builds a symbol table of all labels and their addresses; Pass 2 generates the actual object code using this table, correctly resolving both forward and backward references.

4.2 Common Debugging Techniques

- Single-stepping — executing the program one instruction at a time while observing register and flag contents after each step.
- Breakpoints — inserting a temporary stop instruction at a chosen address so execution pauses there for inspection.
- Memory/register examination — reading back memory locations or register contents to verify intermediate results against expected values.
- Trace/simulation — using a simulator to trace the flow of execution without needing actual hardware.

Note: Common sources of bugs include incorrect flag assumptions after arithmetic/logical instructions, off-by-one errors in loop counters, and forgetting to save/restore registers around subroutine calls.

5. Developing Counters and Time Delay Routines

Because the 8085 executes instructions extremely fast, software time-delay routines are commonly used to introduce a deliberate, calculable pause — for example, to control the blink rate of an LED or the scan rate of a display. The technique relies on loading a register (or register pair) with a count value, and then repeatedly decrementing it in a loop until it reaches zero.

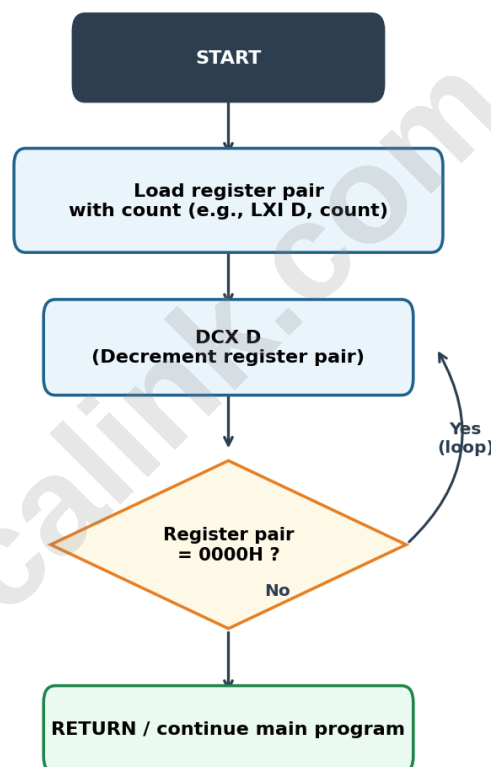


Figure 2.2 — Flowchart of a basic time-delay loop using a register pair

A typical delay subroutine using register pair D-E might be written as follows:

Example 2.2 — Time Delay Subroutine

DELAY: LXI D, 0FFFFH ; load D-E with the count (max 16-bit value)

LOOP: DCX D ; decrement D-E pair

MOV A, D ; copy D into accumulator

ORA E ; OR with E to test if D-E = 0000H

JNZ LOOP ; repeat loop if not yet zero

RET ; return once D-E = 0000H

The total delay produced can be estimated by counting the T-states consumed inside the loop and multiplying by the clock period, then multiplying by the number of iterations:

Example 2.3

If the loop body (DCX D, MOV A,D, ORA E, JNZ LOOP) takes approximately 16 T-states per iteration, and the 8085 runs at 3 MHz, estimate the total delay for an initial count of 1000 (decimal).

Solution

Clock period $T = 1 / 3 \text{ MHz} \approx 0.333 \mu\text{s}$

Time per iteration = $16 \times 0.333 \mu\text{s} \approx 5.33 \mu\text{s}$

Total delay $\approx 1000 \times 5.33 \mu\text{s} \approx 5.33 \text{ ms}$

(For longer delays, nested loops — an outer register-pair loop repeating an inner loop — are used to multiply the achievable delay.)

Counters — programs that count events or repetitions — use exactly the same increment/decrement and conditional-jump structure as delay loops, but the count itself (rather than the elapsed time) is the quantity of interest, e.g., counting the number of times a switch is pressed or the number of items processed.

6. Interfacing Concepts

Interfacing refers to the hardware and software arrangement that allows the microprocessor to communicate with external devices such as memory chips, keyboards, displays and sensors, through the address, data and control buses. Two schemes are used to interface I/O devices with the 8085:

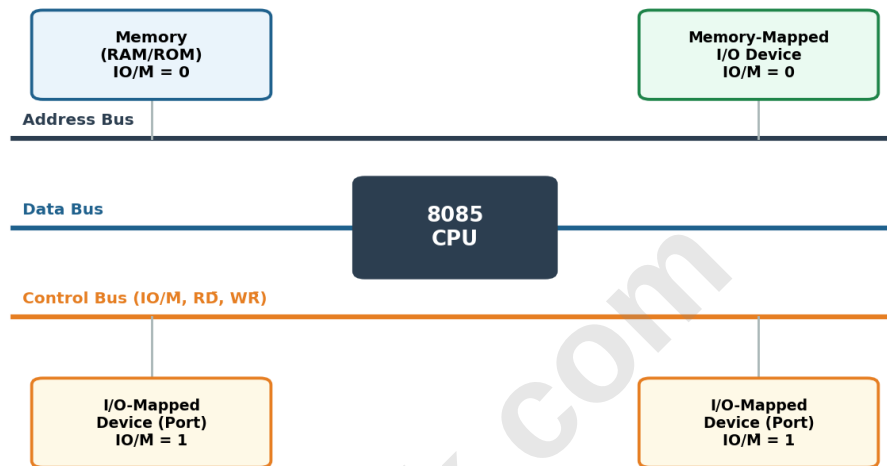


Figure 2.3 — Memory-mapped I/O versus I/O-mapped (peripheral-mapped) I/O

Scheme	Address Space Used	Instructions Used	Key Trade-off
Memory-Mapped I/O	Shares the same 16-bit address space as memory	Any memory-reference instruction (MOV, LDA, STA, etc.)	Larger, more flexible instruction set for I/O, but reduces the address space available for actual memory.
I/O-Mapped (Peripheral) I/O	Separate 8-bit I/O address space (256 ports)	Only IN and OUT instructions	Keeps full memory address space available; but only the Accumulator can be used to transfer I/O data.

6.1 IN and OUT Instructions

Mnemonic	Operation	Bytes
IN port	Reads data from the specified 8-bit input port address into the Accumulator	2

Mnemonic	Operation	Bytes
OUT port	Writes the contents of the Accumulator to the specified 8-bit output port address	2

During any I/O-mapped operation, the IO/M control signal is driven high ($\text{IO}/\text{M} = 1$) to indicate that the address bus carries an I/O port address rather than a memory address; the 8085 duplicates this 8-bit port address on both the upper and lower halves of the 16-bit address bus. For memory-mapped operations, IO/M is driven low ($\text{IO}/\text{M} = 0$).

Note: Interfacing also requires attention to timing: peripherals slower than the CPU may need to insert wait states via the READY line so that data is not read/written before it is valid.

Key Takeaways

- ✓ An assembly statement has up to four fields: Label, Opcode, Operand(s), and Comment.
- ✓ Assembler directives (ORG, END, EQU, DB, DW) instruct the assembler, not the CPU, and generate no machine code.
- ✓ 8085 instructions fall into five categories: Data Transfer, Arithmetic, Logical, Branching, and Machine Control.
- ✓ A two-pass assembler resolves both forward and backward label references by building a symbol table in Pass 1.
- ✓ Time-delay routines use register-pair decrement loops (DCX + conditional jump); delay time = iterations $\times$ T-states per loop $\times$ clock period.
- ✓ Interfacing uses either Memory-Mapped I/O (shares memory address space, more instructions available) or I/O-Mapped I/O (separate 256-port space, uses only IN/OUT).
- ✓ The IO/M control signal distinguishes a memory access ($\text{IO}/\text{M} = 0$) from an I/O access ($\text{IO}/\text{M} = 1$).

— End of Unit 2 —