Introduction to Microprocessors and Assembly Language

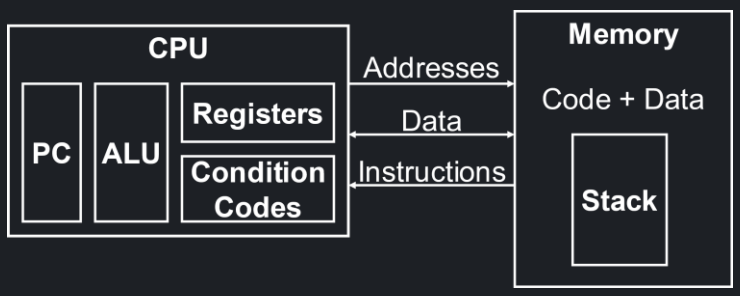
Table of Contents

A **microprocessor** (abbreviated as μP or uP) is a Silicon Chip that contains an electronic **central processing unit** (CPU). The words 'microprocessor' and CPU can be used interchangeably. It is made from miniaturized **transistors** and other circuit elements on a single semiconductor **integrated circuit** (IC). These ICs in turn, are only possible due to VLSI technology. They are thus called VLSI chips.

The first ever commercial microprocessor was built by Intel, the **Intel 4004**. It was a single-core, 4-bit microprocessor, meaning its word size was 4-bits. Nowadays, we use multicore, 64-bit microprocessors.

We will be taking a journey from the first microprocessor to modern day ones, but it is not actually possible to discuss every single microprocessor. Instead, we will be focusing on the **Intel 8086** microprocessor. This is a 16-bit microprocessor that is generally taught at the basic level since it is easy to understand how it works. We will also be studying the 80x86 or the x86 family of microprocessors.

Concept of a Computer



In the above diagram, we can see the major components of a standard personal computer. It essentially has two parts, the microprocessor or **CPU**, and **peripherals**, both of which are integrated circuits. In this diagram, we are considering memory to be the peripheral device, but it can be input and output devices as well.

The CPU is a **stand-alone** system, and it requires **data** from **peripheral devices** to work with. Otherwise, in and of itself, it cannot really do anything. The core components of a CPU are:

- **ALU** – The CPU uses this to perform arithmetic and logical operations on data.
- **Registers** – These are temporary storages where data can be stored after being brought in from the peripherals. The CPU will take this data, operate on it and store the results back onto the registers. Peripherals will then receive the data from the registers.
- **PC** – This is the Program Counter. Technically, this is also a register, but it has a special purpose. It points to the memory location where the next instruction is located. This is the instruction which must be loaded onto the registers next.

To connect the CPU with the peripherals, we require a **BUS**, which is a collection of **wires** through which **signals** are sent. For example, an 8-bit BUS has 8 wires and can exchange 8-bits of data at a time. The BUS can be divided into several parts, such as the **Address BUS**, the **Data BUS** and the **Control BUS**. The Address BUS is **unidirectional**, since only the CPU can send the required address of the data to be sent. The Data BUS is **bi-directional**, since data can come from either direction.

Computer Program Languages

A CPU executes computer programs. Computer programs can be written in three types of languages

1. Low-Level Language
2. Mid-Level Language
3. High-Level Language

High-level languages are the ones we use all the time. These make use of built-in functions that do the core work of **interacting** with the CPU for us, thus making our lives easier. For example, if we want to print something to the console, we can call some form of the print function.

Low-level languages are ones where we have to do all the work by ourselves. This gives us a huge amount of control, since we can literally specify things like which memory location to access, but it is also a lot of work for very little gain. If we try to build large programs using a low-level language, it can quickly become frustrating.

Neither of these can be understood by the computer though. The computer only understands **Machine Language**, which is a series of 1s and 0s. The **compiler** must convert whatever high-level language computer program we give it into machine language, after which the computer will be able to process it. For low-level languages, this conversion is done by the **assembler**. In fact, **Assembly Language**, used by the assembler, is itself a low-level language.

Assembly Language

It is very difficult to program a microprocessor in its native hexadecimal machine language, which is why we use **Assembly Language** instead. Assembly Language is a **low-level language** that gets passed through an **Assembler**, which uses some pre-defined logic to convert each command in the assembly Language code into **Machine Language**.

8085 Assembly Language

Address	Mnemonics	Machine (Hex) Code	Description
202A	MVI A, 32h	3E 32	Copies 32h into accumulator
202C	MVI B, 48h	06 48	Copies 48h into register B
202E	ADD B	80	Adds contents of register B with contents of accumulator and stores the results in accumulator
202F	STA [41 FF]	D3 01	Stores contents of accumulator in memory location 41 FF
2031	HLT	76	Stops the program

The table shows a simple program written in Assembly Language that performs some arithmetic operations.

The column for **Mnemonics** has the actual **commands** for Assembly Language, and the column for **Machine (Hex) Code** is the result of the corresponding commands

being converted into **Machine Language** by the Assembler. The actual system would use binary instead of hex of course.

The **Address** column stores the address of each corresponding Assembly Language instruction. Each instruction requires a **different number of bytes** to store, and the addresses assigned reflect this. For example, the first instruction is converted to `2` bytes of Machine Language code and the starting address for the instruction is `202A`. Thus, the next instruction starts at `202C`. The third instruction on the other hand, is converted to `1` byte of Machine Language code and starts at `202E`. Thus, the next instruction starts at `202F`.

1. `MVI A, 32h` simply means 'Move the immediate Next Value into the Accumulator', the **immediate next value** being the one given immediately after the instruction. In this case, the immediate next value is `32h`. The `MVI` instruction takes **two parameters**, the first being the **destination**, and the second being the **source**.

   `32h` indicates `32` in hexadecimal. If we wanted `32` in binary, we would say `00110010b`, and for decimal, we would just say `32`.

   We will be covering the details of how `MVI A` was converted into `3E`, i.e. how a particular Assembly Language instruction is converted into Machine Language, in a later lecture.

2. We similarly move the hexadecimal value `48h`, this time into the **register** `B`.

3. **Add** the value stored in register `B`, which we know to be `48h`, to the **accumulator**. The `ADD` instruction takes just **one parameter**, the value to add. It automatically adds this to the accumulator only. The **results** of the operation are stored in the **accumulator**.

4. **Store** the contents of the **accumulator** in the **memory location** `41 FF`. Again, the fact that the content comes from the accumulator is implied. Anything specified within **square-brackets** indicates a memory location.

5. **Stop** the program. This instruction is implied at the end of every program.

Address	Mnemonics	Description
2020	`MVI B, 24`	Copies 24 into accumulator
2022	`INR B`	Increments content of register B by 1
2023	`MOV A, B`	Copies content of register B into accumulator
2024	`SUB B`	Subtracts content of register B from the accumulator and stores results in accumulator
2025	`OUT 01h`	Display contents of accumulator on port 01h
2028	`HLT`	Stops the program

The table above shows another program.

1. **Copy** the value `24` (in decimal) into register `B`.

2. The value stored in register `B` is **incremented**.

3. The third instruction is like the `MVI` instruction we saw earlier. The `MOV` instruction takes a **location** as the second parameter instead of a value.

4. The fourth instruction acts like the `ADD` instruction. `SUB` **subtracts** the value of `B` from the accumulator.

5. Every I/O device must be connected to the processor via some **Interface ICs**. The fifth instruction sends the content of the **accumulator** to whatever interface IC is connected to the port `01h`.

6. Stop the program.

8086 Assembly Language

The Assembly Language commands for 8086 are generally the same as those for 8085, other than a few differences. It is more **structured**.

Address	Hex Object Code		Mnemonics		Description
			Op-Code	Operand	
0100	E4	27	IN	AL, 27h	AL = Input from port 27h
0102	88	C3	MOV	BL, AL	Copy AL to BL
0104	E4	27	IN	AL, 27h	AL = Input from port 27h
0106	00	D8	ADD	AL, BL	AL = AL + BL
0107	E6	30	OUT	30h, AL	Output AL to port 30h
0109	F4		HLT		Stop the program

Notice that the Mnemonics are divided into **op-codes** and **operands**. Each op-code has a binary expression. Each operand is basically the parameter.

1. The `IN` op-code is used to take **input** into the **accumulator** from whatever interface is connected to the port `27h`. If the device is a keyboard, the command will wait for an input.

   Notice how it is **not implied** that the content should go to the accumulator. We had to stated that specifically. Also notice that the accumulator is denoted as `AL` instead of just `A`. 8086 has an accumulator register that is labelled `AX`. It is a 16-bit register that can be further divided into **two 8-bit registers**, `AH` (accumulator high) and `AL` (accumulator low). This is because not all operations require 16-bits, and storing 8-bit instructions in a 16-bit register would be a **waste of space**.

2. The `MOV` op-code is used to **copy** the contents of the accumulator into the register `BL`. Again, notice how `BL` is used to denoted a register instead of just `B`. 8086 has a base register, `BX`, which is also a 16-bit register and can be divided into **two 8-bit parts**, `BH` and `BL`.

3. We take another input into the accumulator in the same way that we did in the first step.

4. Next, we add the contents of the register `BL` to the contents of the accumulator using the `ADD` op-code. Notice how `ADD` takes **two** parameters now. The **results** are stored in the `AL` register.

5. The `OUT` op-code is used to send the contents of the **accumulator** to the interface IC connected to the specified port. Again, we needed to specify that the output needs to come from the accumulator. It was **not implied**.

6. Finally, the `HLT` op-code **stops** the program.