

4. Processor Fundamentals

4.1 Central Processing Unit (CPU) Architecture

- **Von Neumann Architecture:** computer architecture which introduced the concept of the stored program → uses a single storage structure to hold both data and instructions
 - Early computers are fed data while running.
- **Basic features:**
 1. A central processing unit (a CPU or a process)
 2. A process able to access memory directly
 3. Computer memories that could store programs as well as data
 4. Stored programs made up of instructions that could be executed in a sequential order

Main Components

- **Arithmetic Logic Unit (ALU):** Performs all the mathematical and logical operations required when processing data and instructions
- **Control Unit (CU):** Directs the operation of the processor and manages the flow of data within the CPU and between the CPU and other components
 1. *Instruction Fetching:* Retrieves instructions from memory
 2. *Instruction Decoding:* Deciphers what operation is to be performed
 3. *Execution:* Directs the ALU, registers, and other components to execute the instruction
 4. *Control of Data Flow:* Manages the flow of data within the CPU and to/from memory
 5. *Timing and Control:* Ensures that all parts of the CPU and peripherals are synchronized and operating correctly
- **System Clock:** Provides regular, constant time signals on the control bus that help synchronize all the computer's operations
- **Immediate Access Store (IAS):** Provides fast, direct access storage for data and instructions that are actively being used
 - The CPU takes data and programs held in the backing store and puts them into the IAS temporarily because the IAS's read/write is faster than that of the backing store
 - Another name for primary memory (e.g., **RAM**)

Registers

- Temporary component in the processor which can be general or specific in its use that holds data or instructions as part of the fetch-execute cycle.
 - **General purpose:** versatile and can be used for a variety of functions
 - they can be used for different purposes at different times
 - **Special purpose:** used to perform specific, pre-defined functions
 - e.g., program counter

Special Purpose Registers

- **Program Counter (PC):** Stores the memory address of the next instruction to be executed by the CPU
- **Current Instruction Register (CIR):** Stores the current instruction being decoded and executed by the CPU
- **Index Register (IX):** Stores an offset value, which is added to the base address of an array or memory location to calculate the effective address of the data being accessed or manipulated

- **Memory Address Register (MAR):** Stores the address in memory from where data is to be fetched or to where data is to be stored
- **Memory Data Register (MDR):** Temporarily stores the data being read from or written to memory
- **Status Register (SR):** Holds flags that indicate the current state of the processor and the outcomes of various operations, guiding subsequent instructions
- **Accumulator (ACC):** Stores the result of any interim calculations of the ALU

Buses

- Used in computers as a parallel transmission component; each wire in the bus transmits one bit of data at a time
- **Data Bus:** Transmission of data and instructions, bidirectional
- **Address Bus:** Transmission of addresses, unidirectional
- **Control Bus:** Transmission of control signals, bidirectional

CPU Performance

- **Bus Width:** the larger the bus width, the more data can be transferred at the same time throughout the CPU (e.g., 64-bit v.s. 32-bit)
- **Number of Cores:** the more cores a CPU has, the more FDE cycles it can carry out at once (e.g., dual-core and quad-core)
 - doubling the core \neq doubling the performance, as the CPU needs to communicate with both cores, causing performance degradation
- **Clock Speed:** the faster the clock speed, the more number of cycles the CPU can perform per second
 - **overclocking** might happen: the clock speed is higher than the computer was designed for \rightarrow leading async operations (as a new instruction is sent when the prior one is not completed) & overheating
- **Cache Memory:** the larger the cache memory, the more data can be stored in the **cache** to be accessed by the CPU directly
 - **cache:** a small, fast memory built into the CPU that stores the most frequently accessed data and instructions, reducing the time it takes to fetch data from the slower main memory
 - A large cache size might **decrease** performance as it takes a long time to search over the data inside the large cache

Computer Port

- Input and output devices are connected to a computer via **ports**
- **USB:** serialized data transfer; data connection controlled by a host (e.g., a computer) that communicates with connected devices
 - the host recognizes the device automatically when it is connected and establishes a communication channel
- **HDMI:** sends audio and video data signals together, using a single cable for high-definition transmission
 - when connected, HDMI encodes digital signals and source and decode them at the display
- **VGA:** an older, analog method for transmitting video signals, primarily used for simple display connections

F-E Cycle

- Be familiar with the memory register notation, e.g., $MAR \leftarrow PC$

1. Fetch Stage

1. $MAR \leftarrow [PC]$: The content of the PC is transferred to the MAR; this is the address of the next instruction to be fetched
2. $PC \leftarrow [PC] + 1$: The PC is incremented by 1 to point to the next instruction
3. $MDR \leftarrow [[MAR]]$: The address that is stored by the MAR is fetched from the memory and placed in MDR
4. $CIR \leftarrow [MDR]$: The instruction in the MDR is transferred to the CIR

2. Decode Stage

- The CU uses an **instruction set** to decode the instructions into machine codes for the CPU to execute

3. Execute Stage

- The CU directs the components to execute the instruction being decoded. Different processes are taken for different types of commands

Interrupts

- a signal sent from a device or from software to the processor, causing the processor to temporarily stop what it is doing and service the interrupt
- **Interrupt Service Routine (ISR)**: low-level functions designed to handle interrupts; ISRs are specific to each interrupt and contain the code to handle the interrupt

Causes of interrupts

1. A timing signal
2. A hardware fault (e.g., a paper jam in the computer)
3. A software error (e.g., division by zero)
4. User interaction (e.g., the user presses a key to stop the current process)

Applications of interrupts

1. Multitasking
2. Response to real-time events, like user input or sensor signals
3. System control and coordination, managing various hardware and software functions simultaneously

Handling of interrupts

1. **Interrupt Acknowledgement**: the CPU recognizes the interrupt and determines the **interrupt priority** to see whether to serve the interrupt
2. **Save State**: the CPU saves its current state (like register contents) to return later
3. **Execute ISR**: the CPU executes the appropriate ISR
4. **Restore State and Resume**: after ISR execution, the CPU restores its previous state and resumes its interrupted task

4.2 Assembly Language

- **Assembly Language**: A low-level programming language that uses mnemonic to represent machine-level instructions. It is more readable than machine code and specific to a computer's architecture.
- **Machine Code**: Binary codes that are directly executed by a CPU

Stages of Assembly

1. First Pass: Analysis

- **Symbol Table Creation:** Identifies and stores all labels (symbols) used in the program to the **symbol table**, like variable names and jump labels, along with their addresses
- **Syntax Checking:** Checks for syntax errors in the code, ensuring that all the instructions and operands are valid

2. Second Pass: Synthesis

- **Opcode Resolution:** Converts mnemonics into their corresponding opcodes (operation codes) – the machine code instructions.
- **Address Resolution:** Resolves addresses for labels and variables. The addresses from the symbol table created in the first pass are used to replace the labels in the instructions.

Application of Two-Pass Assembler Process

```
START:  MOV A, 5      // Move 5 into register A
        ADD B         // Add the contents of register B to A
        JMP START     // Jump to the start
```

First Pass:

- **Symbol Table:**
 - `START` : Address of the first instruction
 - e.g., `START 100`

- **Syntax Checking:**
 - All instructions are valid

Second Pass:

- **Opcode Resolution:**
 - `MOV A, 5` is converted to its binary equivalent opcode
 - `ADD B` is converted into its corresponding machine code
- **Address Resolution:**
 - `JMP START` is resolved with the address of `START` from the symbol table

Grouping of Assembly Codes

- In assembly language, instructions are typically grouped based on their functionality → helps in understanding and organizing the code
- **Data Movement:** `LDM, LDD, LDI, LDX, LDR, MOV, STO`
- **Input and Output of Data:** `IN, OUT`
- **Arithmetic Operations:** `ADD, SUB, INC, DEC`
- **Conditional and Unconditional Instructions:** `JMP, JPE, JPN`
- **Compare Instructions:** `CMP, CMI`

Addressing Modes

- **Immediate:** the value of the operand only is used
 - `LDD #200` stores the value of 200 in the ACC
- **Direct:** the content of the memory location in the operand is used
 - `LDD 200` stores the content in **address 200** in the ACC

- **Indirect:** the content of the content of the memory location in the operand is used
 - `LDI 200` stores the content in the address denoted by the content in **address 200**
- **Indexed:** the content of the memory location found by adding the content of the index register (IX)
 - `LDX 200` stores the content in the **address 200 + value of IX**
- **Relative:** the memory address used is the current memory address added to the operand
 - `JMR #5` would jump to the instruction **5 locations after the current one**

4.3 Bit Manipulation

Binary Shifts

- **Logical Shift:** bits shifted out of the register are replaced with zeros
- **Arithmetic Shift:** the sign of the number is preserved
- **Cyclic Shift:** no bits are lost during a shift; bits shifted out of one end of the register are introduced at the other end of the register
- Shifts are always performed on the ACC

Bit Manipulation in Monitoring & Control

- each bit in a register/memory location can be used as a **flag** and would need to be tested, set, or cleared separately
- **Operations**
 - `AND <mask>` is used to **check if the bit has been set**
 - `OR <mask>` is used to **set the bit**
 - `XOR <mask>` is used to **clear the bit**
 - The results of logical bit manipulation are always stored in the ACC

Example

- Code to test sensor 3
- B denotes a binary number, # denotes a denary number, & denotes a hexadecimal number

```
LDD sensors // Load content of sensors into ACC
AND #B100   // Select bit 3 only
CMP #B100   // Check if bit 3 is set
JPN process // Jump to process routine if bit not set
LDD sensors // Load content of sensors into ACC
XOR #B100   // Clear bit 3 as sensor 3 has been processed
```