IT Fundamentals

Dr/Mai Ramadan Ibraheem

Lecturer at Information Technology Dept.,

Faculty of Computers and Information – KFS University

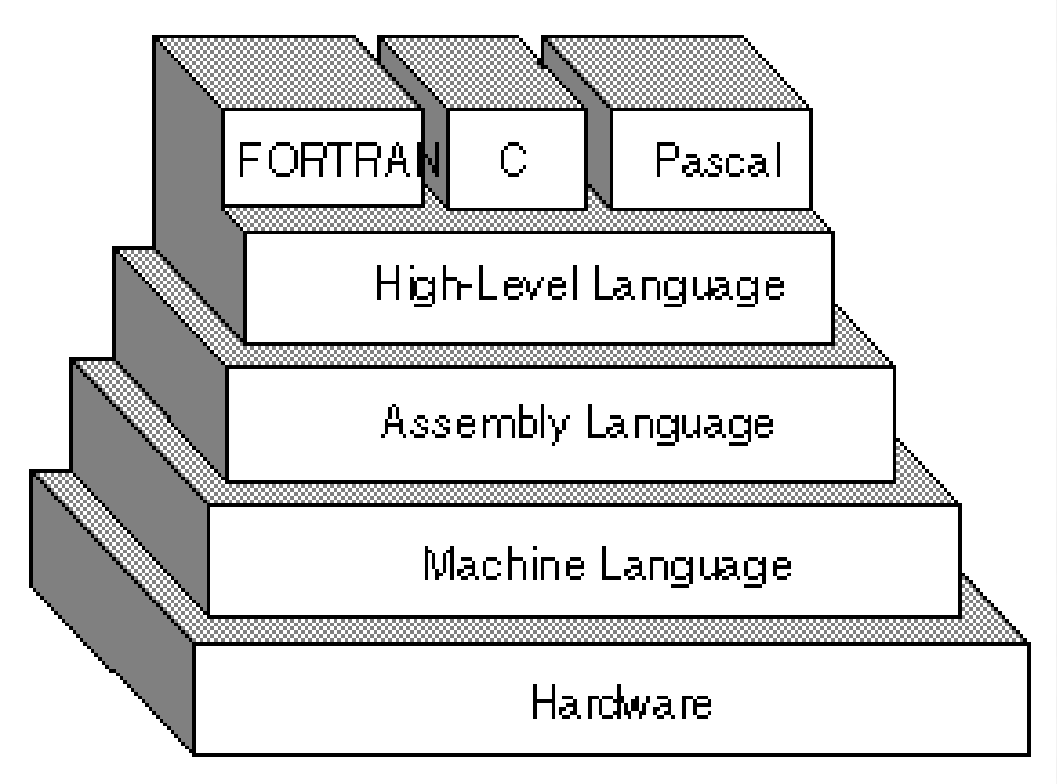
Outline

- Introduction.
- Machine level representation of data.
- Digital logic.
- Assembly level machine organization
- Hardware realizations of algorithms.
- Operating systems and virtual machines.
- Computing applications.
- Introduction to net-centric computing.

Outline

- Assembly Language.
- Von Neumann Model.

Assembly Programming page 85



Assembly language

- Is a low-level programming language for specific to a particular computer architecture
- In contrast to most high-level programming languages, which are generally portable across multiple systems.
- Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

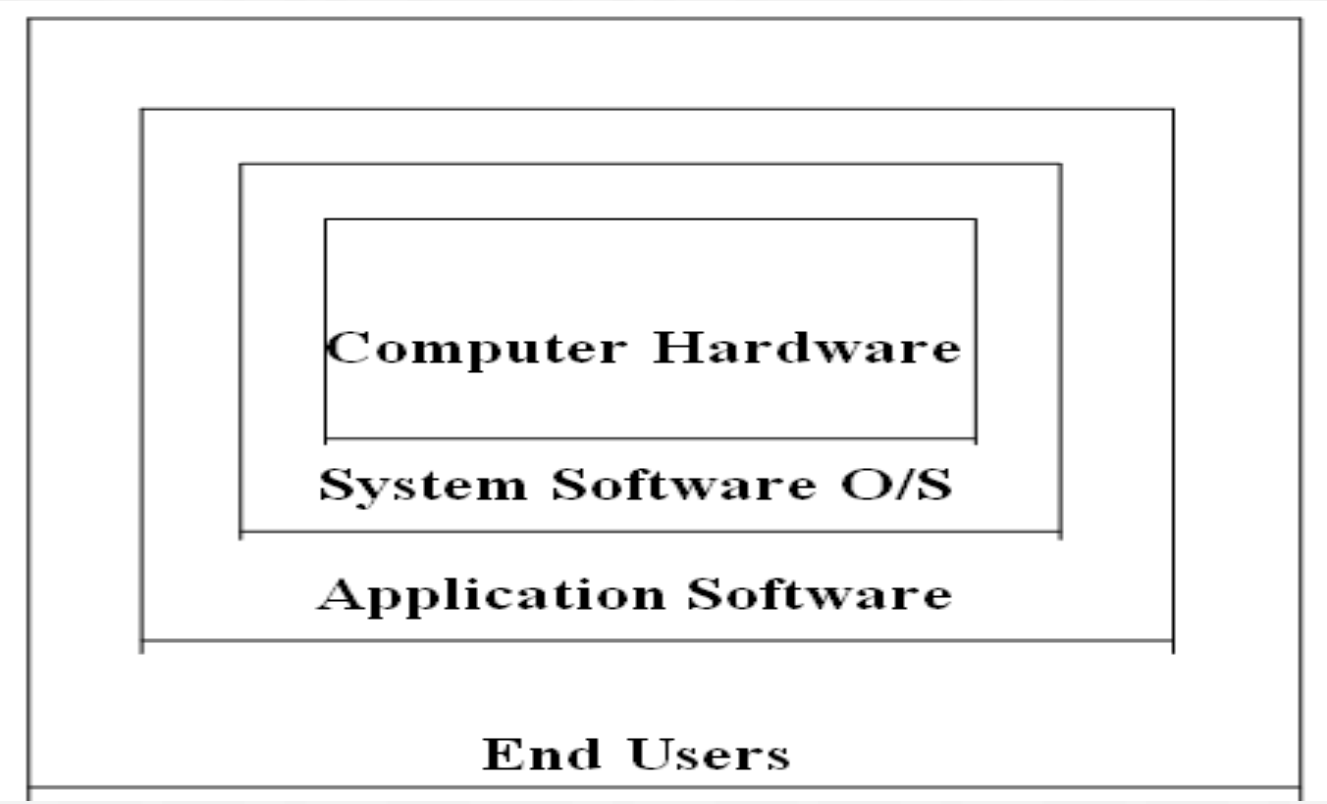
What is Assembly Language?

- Each PC has a microprocessor that manages arithmetical, logical, and control activities.
- Each family of processors has its own set of instructions for handling various operations
- such as getting input from keyboard, displaying on screen and performing various jobs.
- These set of instructions are called 'machine language instructions'.

What is Assembly Language?

- A processor understands only machine language instructions, which are strings of 1's and 0's.
- However, machine language is too obscure and complex for using in software development.
- So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Multilevel Machine



Multilevel Machine

- The most important **system software** is the **operating system**,
- O/S is an integrated system of programs that manages the system resources, and provides various support services
- such as executing the application programs of users.

Programming Languages Categories

- **Machine Language (First-generation Language)**
- Use binary coded instruction.
- **Assembly Language (Second-generation Language)**
- Use symbols to represent operation codes and storage locations.

Programming Languages Categories

- **High-Level Languages (Third-generation Languages)**
- Use statements that closely resemble human language or standard notation of mathematics.
- **Fourth-generation Languages**
- Use nonprocedural programming languages.

Advantages of Assembly Language

Makes one aware of :

- How to programs OS interface,
- How data is represented in memory and other external devices;
- How the processor accesses and executes instruction;
- How instructions access and process data;
- How a program accesses external devices.

Advantages of Assembly Language

Other advantages:

- Requires less memory and execution time;
- Allows hardware-specific complex jobs in an easier way;
- Suitable for time-critical jobs;
- Assembly uses **mnemonic sequences** instead of numeric operation codes and can use **symbolic labels** instead of manually calculating offsets.

Outline

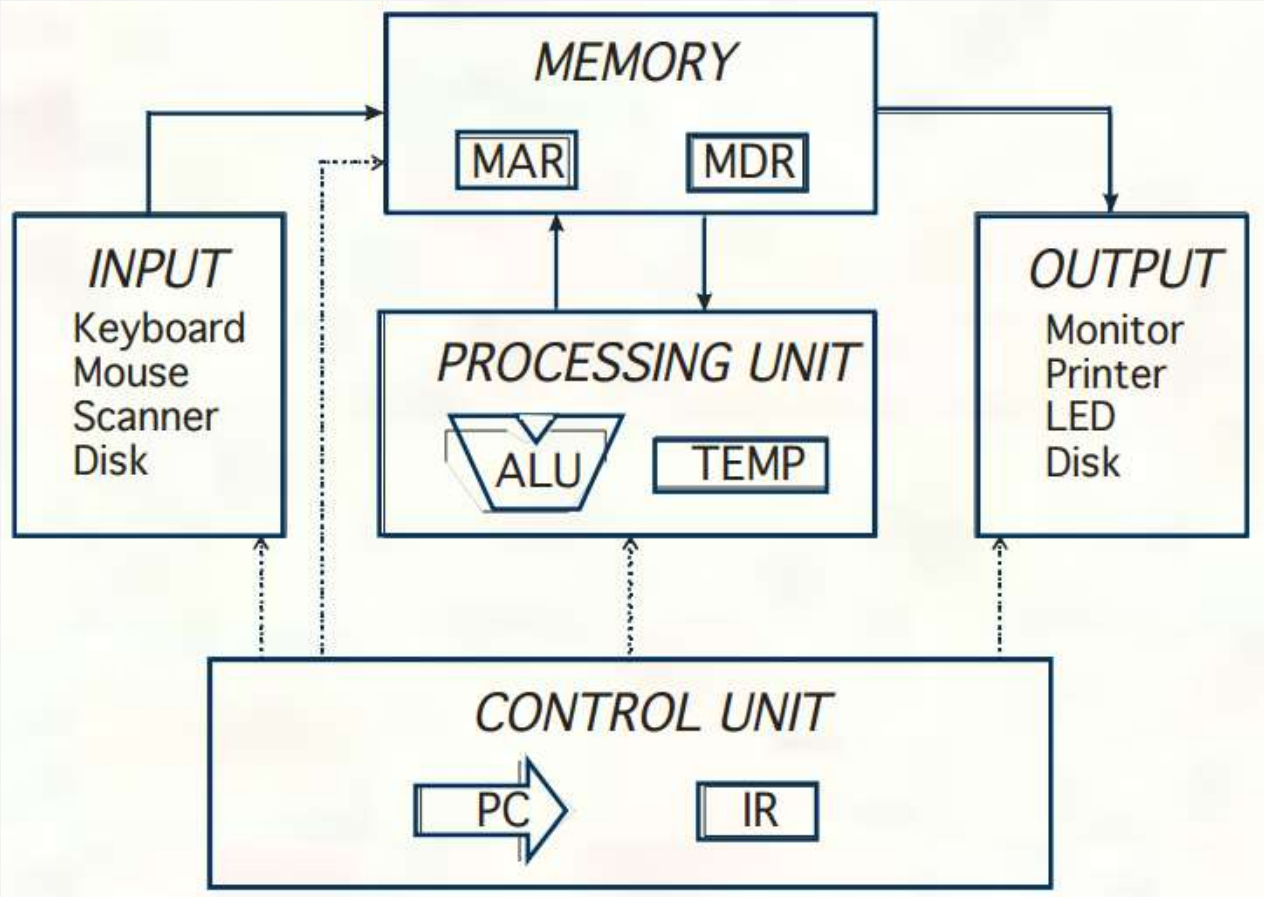
- Assembly Language.
- Von Neumann Model.

Von Neumann model page 79

The basic structure of "von Neumann machine" (or model):

- **A memory**, containing instructions and data.
- **A processing unit**, for performing arithmetic and logical operations.
- **A control unit**, for interpreting instructions.

Von Neumann model



Von Neumann model

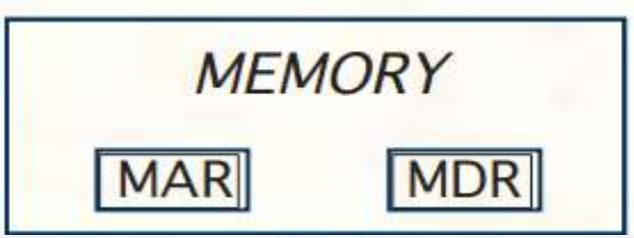
Memory: 2^k×m array of stored bits.

Address: unique (k-bit) identifier of *location*.

Contents: m-bit *value* stored in location.

Basic Operations: **LOAD**, read a value from a memory location and **STORE**, write a value to a memory location.

How processing unit get data to/from memory?



MAR: Memory Address Register

MDR: Memory Data Register

To LOAD a location (A)

1. Write the **address** (A) into the MAR.
2. Send a "**read**" signal to the memory.
3. Read the **data** from MDR.

To STORE a value (X) to a location (A)

1. Write the **data** (X) to the MDR.
2. Write the **address** (A) into the MAR.
3. Send a "**write**" signal to the memory.

Processing Unit

- Functional Units
- ALU = Arithmetic and Logic Unit could have many functional units.
- Special-purpose (multiply, square root, …)
- Performs ADD, AND, NOT

Input and Output

- Input and Output Devices *for getting data* into and out of computer memory.
- Each device has its *own interface*, usually a set of *registers* supports keyboard (input) and monitor (output)
- keyboard: data register (KBDR) and status register (KBSR)
- Monitor (Display): Data Register (DDR) and Status Register (DSR).
- Some devices *provide both* input and output disk, network Program that *controls access to a device* is usually called a driver.

Control Unit

- *Orchestrates execution* of the program.
- Instruction Register (IR) contains the current instruction. Program Counter (PC) contains the address of the next instruction to be executed.
- Control unit: **reads** an instruction from memory the instruction's address is in the PC interprets the instruction, **generating** signals that tell the other components what to do, an **instruction** may take *many machine cycles* to complete.

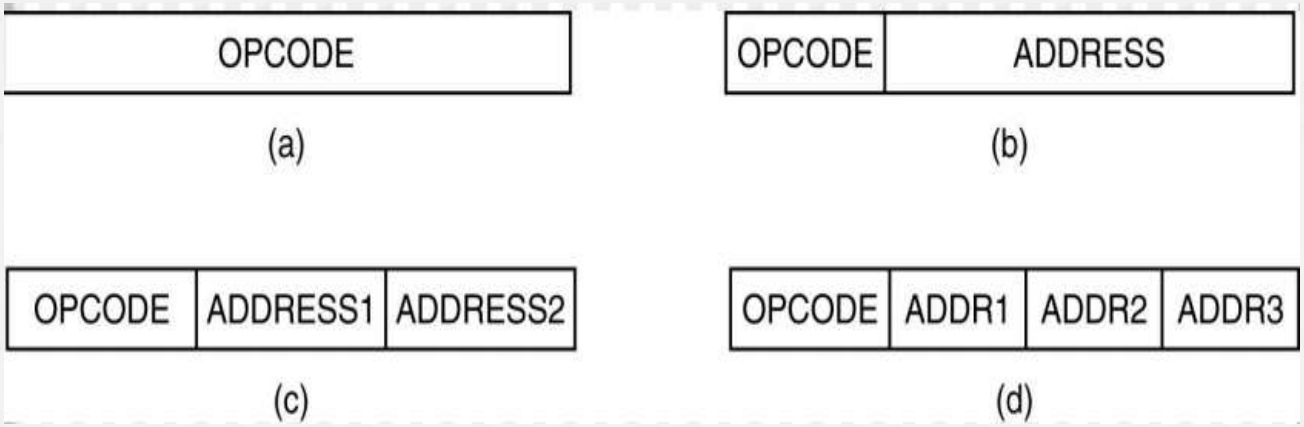
Instruction Processing

Instruction Processing

- The instruction is the *fundamental unit of work*.
- Specifies two things:
- **Opcode or "field"**: (operation code) operation to be performed
- **Operands or "constant value"** : data/locations to be used for operation, provide supplemental information required for the operation.
- An instruction is encoded as *a sequence of bits*.

Instruction Format page 93

- Four common instruction formats:



- (a) Zero-address instruction. (b) One-address instruction
- (c) Two-address instruction. (d) Three-address instruction

Instruction Processing

- Fetch
- Decode
- Execute

Fetch

- The first step, fetch, involves **retrieving** an instruction (which is represented by a number or sequence of numbers) from program memory.
- The instruction's location (address) in program memory is determined by a *program counter* (PC), which stores a number that identifies the address of the *next instruction* to be fetched.

Fetch

- After an instruction is fetched, the PC is **incremented** by the length of the instruction so that it will *contain the address* of the next instruction in the sequence.

Summaries Fetch

- Load next instruction (at address stored in PC) from *memory* into *Instruction Register* (IR).
- Copy contents of PC into *MAR*. Send "read" signal to memory.
- Copy contents of *MDR* into IR.
- Then *increment* PC, to the next instruction in sequence. PC becomes PC+1.

Instruction Processing

- Fetch
- Decode
- Execute

Decode

- This step, performed by the **instruction decoder**,
- The instruction is **converted into signals** that control other parts of the CPU.
- The way in which the instruction is interpreted is defined by the CPU's **Instruction Set Architecture** (ISA).

Decode

- Often, one group of bits (that is, a "field") within the instruction, called the **opcode**, indicates *which operation* is to be performed,
- While the remaining fields usually *provide supplemental information* required for the operation, such as the **operands**.

Summaries Decode

- First identify the opcode.
- The first no. of bits of instruction.
- A decoder asserts a control line corresponding to the desired opcode.
- Depending on opcode, identify other operands from the remaining bits.

Instruction Processing

- Fetch
- Decode
- Execute

Execute

- Depending on the CPU architecture, this may consist of a **single action** or a **sequence of actions**.
- During each action, various parts of the CPU are *electrically connected* so they can *perform* all or part of the desired operation and then the action is *completed*, typically *in response* to a clock pulse.

Execute

- Very often the *results* are written to an *internal CPU register* for quick access by subsequent instructions.
- The ALU is configured to perform an *addition* operation so that the *sum of its operand* inputs will appear at its *output*, and the ALU output is *connected to storage* (e.g., a register or memory) that will *receive* the sum.

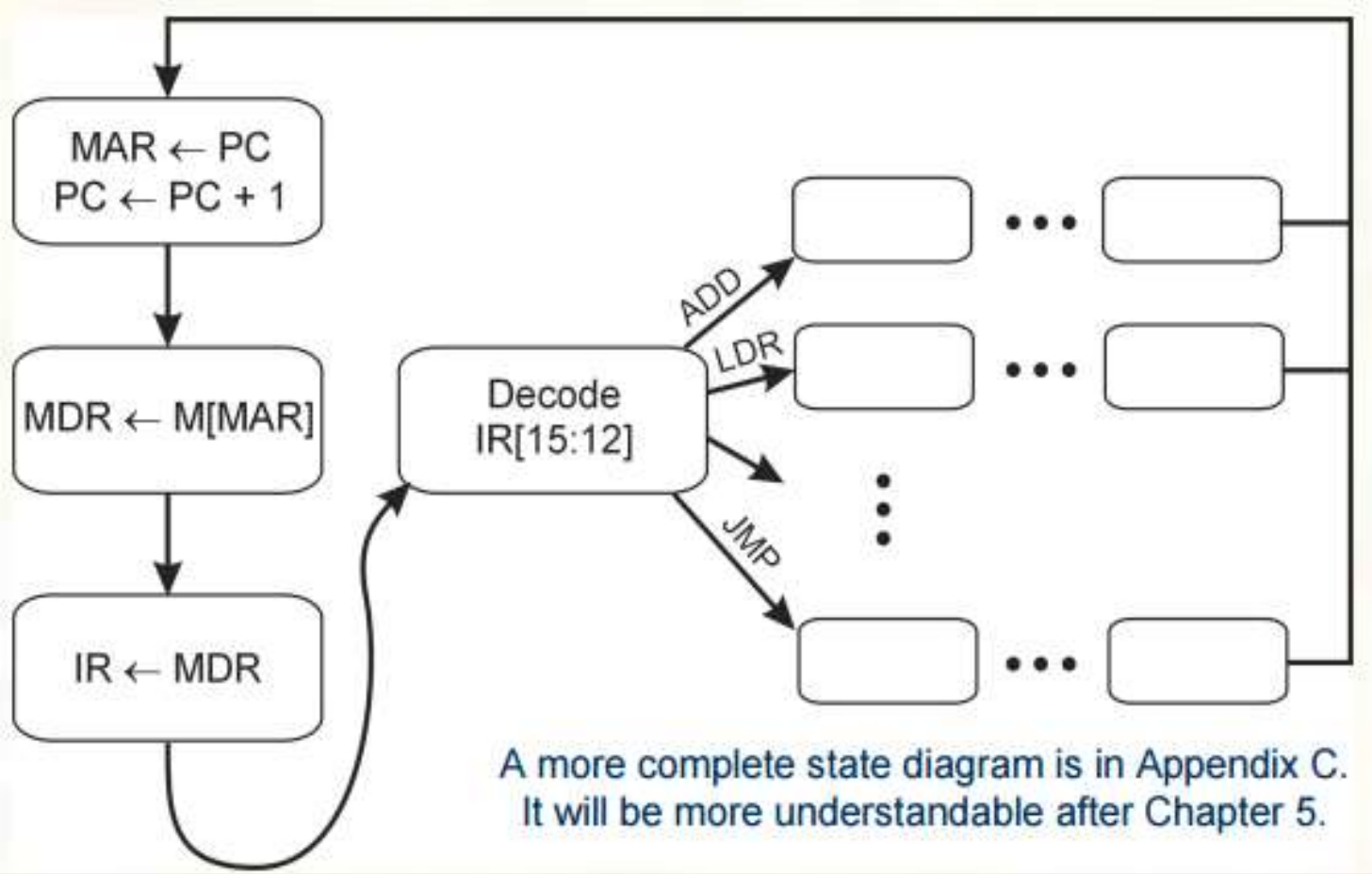
Summaries Execute

- When the clock pulse occurs, the sum will be transferred to storage.
- CP, the Clock pulse is used to *synchronize the timing* of hardware components. The speed of the computer's processor, measured in MHz or GHz, refers to its number of clock pulse cycles per second.

STORE RESULT

- Write results to destination. (register or memory)
- Examples: result of ADD is placed in destination register result of memory load is placed in destination register for store instruction,
- data is stored to memory write *address* to **MAR**, *data* to **MDR** assert WRITE signal to memory.

Control Unit State Diagram




End


Thank You