## BCA 3RD Semester

**Course Code: BCA-S3-01**

**Course Title: Computer Organization And Architecture**

### Unit-I

**Data Representation:** Since the early days of human civilization, people have been using their fingers, sticks, etc. for counting things. The need for counting probably originated when man started to use animals for domestic purposes and practice animal breeding for fulfilling his needs and requirements. As daily activities become more complex, numbers became more important in trade, time, distance, and in all spheres of human life. Ever since people discovered that it was necessary to count objects, they have been looking for easier ways of counting. To count large numbers, man soon started to count in groups, and various number systems were formed. The number system can be categorized into two broad categories.

- **Non-Positional Number System:** In ancient times, people used to count with their fingers. When fingers became insufficient for counting, stones and pebbles were used to indicate the values. This method of counting is called non-positional number system. It was very difficult to perform arithmetic operations with such a number system, as it had no symbol for zero.
- **Positional Number System:** Apositional number system is any system that requires a finite number of symbols/digits of the system to represent arbitrarily large numbers. When using these systems the execution of numerical calculations becomes simplified, because of finite set of digits are used. The value of each digit in a number is defined not only by the symbol but also by the symbol's position.

**Number System:**Number system is used to represent information in quantitative form. Some of the common number systems are binary, octal, decimal and hexadecimal. A number system of base (also called radix) r is a system, which has r distinct symbols for r digits. A string of these symbolic digits represents a number. To determine the value that a number represents, we multiply the number by its place value that is an integer power of r depending on the place it is located and then find the sum of weighted digits.

1. **Decimal Numbers:** Decimal number system has ten digits represented by 0,1,2,3,4,5,6,7,8 and 9. Any decimal number can be represented as a string of these digits and since there are ten decimal digits, therefore, the base or radix of this system is 10. Thus, a string of number 234.5 can be represented as:

   $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$

2. **Binary Numbers:** In binary numbers we have two digits 0 and 1 and they can also be represented, as a string of these two-digits called bits. The base of binary number system is 2.

   For example, 101010 is a valid binary number.

   Decimal equivalent of a binary number:

   For converting the value of binary numbers to decimal equivalent we have to find its value, which is found by multiplying a digit by its place value.

   For example, binary number 101010 is equivalent to:

   $1\times2^5+0\times2^4+1\times2^3+ 0\times2^2+1\times2^1+0\times2^0$

   $= 1\times32 + 0\times16 + 1\times8 + 0\times4 + 1\times2 + 0\times1$

   $= 32 + 8 + 2 = 42$ in decimal.

3. **Octal Numbers:** An octal system has eight digits represented as 0,1,2,3,4,5,6,7. For finding equivalent decimal number of an octal number one has to find the quantity of the octal number which is again calculated as:

   Octal number $(23.4)_8$.

   (Please note the subscript 8 indicates it is an octal number, similarly, a subscript 2 will indicate binary, 10 will indicate decimal and H will indicate Hexadecimal number, in case no subscript is specified then number should be treated as decimal number or else whatever number system is specified before it.)

   Decimal equivalent of Octal Number:

| |
|---|
| $(23.4)_8$ |
| $= 2\times8^1 +3\times8^0 +4\times8^{-1}$ |
| $= 2\times8+3\times1+4\times 1/8$ |
| $=16+3+0.5$ |
| $= (19.5)_{10}$ |

4. **Hexadecimal Numbers:** The hexadecimal system has 16 digits, which are represented as 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. A number (F2)H is equivalent to

| |
|---|
| $F\times16^1 + 2\times16^0$ |
| = (15×16) + 2 // (As F is equivalent to 15 for decimal) |
| = 240 + 2 |
| = $(242)_{10}$ |

**Fixed Point Representation:** The fixed-point numbers in binary uses a sign bit. A positive number has a sign bit 0, while the negative number has a sign bit 1. In the fixed-point numbers we assume that the position of the binary point is at the end, that is, after the least significant bit. It implies that all the represented numbers will be integers. A negative number can be represented in one of the following ways:

- Signed magnitude representation
- Signed 1's complement representation, or
- Signed 2's complement representation.

(Assumption: size of register = 8 bits including the sign bit)

**Signed Magnitude Representation**

| **Decimal Number** | **Representation (8 bits)** | |
|---|---|---|
| | **Sign Bit** | **Magnitude (7 Bits)** |
| **+6** | **0** | **000 0110** |
| **-6** | **0** | **000 0110** |
| **No change in the Magnitude, only sign bit changes** | | |

**Signed 1's Complement Representation**

| **Decimal Number** | **Representation (8 bits)** | |
|---|---|---|
| | **Sign Bit** | **Magnitude/ 1's complement for negative number (7 Bits)** |
| **+6** | **0** | **000 0110** |
| **-6** | **0** | **000 0110** |
| **For negative number take 1's complement of all the bits (including sign bit) of the positive number** | | |

**Signed 2's Complement Representation**

| Decimal Number | Representation (8 bits) | |
|---|---|---|
| | Sign Bit | Magnitude/ 1's complement for negative number (7 Bits) |
| +6 | 0 | 000 0110 |
| -6 | 0 | 000 0110 |
| For negative number take 2's complement of all the bits (including sign bit) of the positive number | | |

**Floating Point Representation:**Floating-point number representation consists of two parts. The first part of the number is a signed fixed-point number, which is termed as mantissa, and the second part specifies the decimal or binary point position and is termed as an Exponent. The mantissa can be an integer or a fraction. Please note that the position of decimal or binary point is assumed and it is not a physical point, therefore, wherever we are representing a point it is only the assumed position.

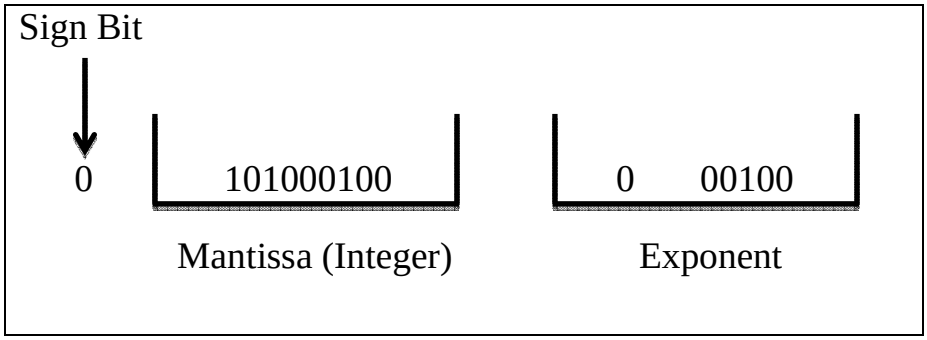
**Example:** A decimal + 12.34 in a typical floating point notation can be represented in any of the following two forms:

This number in any of the above forms (if represented in BCD) requires 17 bits for mantissa (1 for sign and 4 each decimal digit as BCD) and 9 bits for exponent (1 for sign and 4 for each decimal digit as BCD). Please note that the exponent indicates the correct decimal location. In the first case where exponent is +2, indicates that actual position of the decimal point is two places to the right of the assumed position, while exponent– 2 indicates that the assumed position of the point is two places towards the left of assumed position. The assumption of the position of point is normally the same in a computer resulting in a consistent computational environment.

Floating-point numbers are often represented in normalized forms. A floating point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in normalized form. For example, a BCD mantissa + 370 which is 0 0011 0111 0000 is in normalized form because these leading zero's are not part of a zero digit. On the other hand a binary number 0 01100 is not in a normalized form. The normalized form of this number is:

| | | |
|---|---|---|
| 0 | 1100 | 0100 |
| Sign | Normalized Mantissa | Exponent (assuming fractional Mantissa |

A floating binary number +1010.001 in a 16-bit register can be represented in normalized form (assuming 10 bits for mantissa and 6 bits for exponent).




A zero cannot be normalized as all the digits in mantissa in this case have to be zero. Arithmetic operations involved with floating point numbers are more complex in nature, take longer time for execution and require complex hardware. Yet the floatingpoint representation is a must as it is useful in scientific calculations. Real numbers are normally represented as floating point numbers.

The following figure shows a format of a 32-bit floating-point number.

| 0 | 1 ... 8 | 9 ... 31 |
|---|---|---|
| Sign | Biased Exponent = 8 bits | Significant = 23 bits |

**Floating Point Number Representation**

The characteristics of a typical floating-point representation of 32 bits in the above figure are:

- Left-most bit is the sign bit of the number;
- Mantissa or significant and should be in normalized form;
- The base of the number is 2, and
- A value of 128 is added to the exponent. (Why?) This is called a bias.

A normal exponent of 8 bits normally can represent exponent values as 0 to 255. However, as we are adding 128 for getting the biased exponent from the actual exponent, the actual exponent values represented in the range will be – 128 to 127.

## Error Detection and Correction Codes

Computer is an electronic media; therefore, there is a possibility of errors during data transmission. Such errors may result from disturbances in transmission media or external environment. But what is an error in binary bit? An error bit changes from 0 to 1 or 1 to 0. One of the simplest error detection codes is called parity bit.

**Parity Bit:** A parity bit is an error detection bit added to binary data such that it makes the total number of 1’s in the data either odd or even. For example, in a seven bit data 0110101 an 8th bit, which is a parity bit may be added. If the added parity bit is even parity bit then the value of this parity bit should be zero, as already four 1’s exists in the 7-bit number. If we are adding an odd parity bit then it will be 1, since we already have four 1 bits in the number and on

adding 8th bit (which is a parity bit) as 1 we are making total number of 1's in the number (which now includes parity bit also) as 5, an odd number.

Similarly in data 0010101 Parity bit for even parity is 1
Parity bit for odd parity is 0

The error detection mechanism can be defined as follows:

Figure

**Error detection and correction**

**The Objective:** Data should be transmitted between a source data pair reliably, indicating error, or even correcting it, if possible.

**The Process:**

- An error detection function is applied on the data available at the source end an error detection code is generated.
- The data and error detection or correction code are stored together at source.
- On receiving the data transmission request, the stored data along with stored error detection or correction code are transmitted to the unit requesting data (Destination).
- On receiving the data and error detection/correction code from source, the destination once again applies same error detection/correction function as has been applied at source on the data received (but not on error detection/ correction code received from source) and generates destination error detection/correction code.
- Source and destination error codes are compared to flag or correct an error as the case may be.

The parity bit is only an error detection code. The concept of error detection and correction code has been developed using more than one parity bits. One such code is Hamming error correcting code.

**Hamming Error-Correcting Code:** Richard Hamming at Bell Laboratories devised this code. We will just introduce this code with the help of an example for 4 bit data. Let us assume a four bit number b4, b3, b2, b1. In order to build a simple error detection code that detects error in one bit only, we may just add an odd parity bit. However, if we want to find which bit is in error then we may have to use parity bits for various combinations of these 4 bits such that a bit error can be identified uniquely.

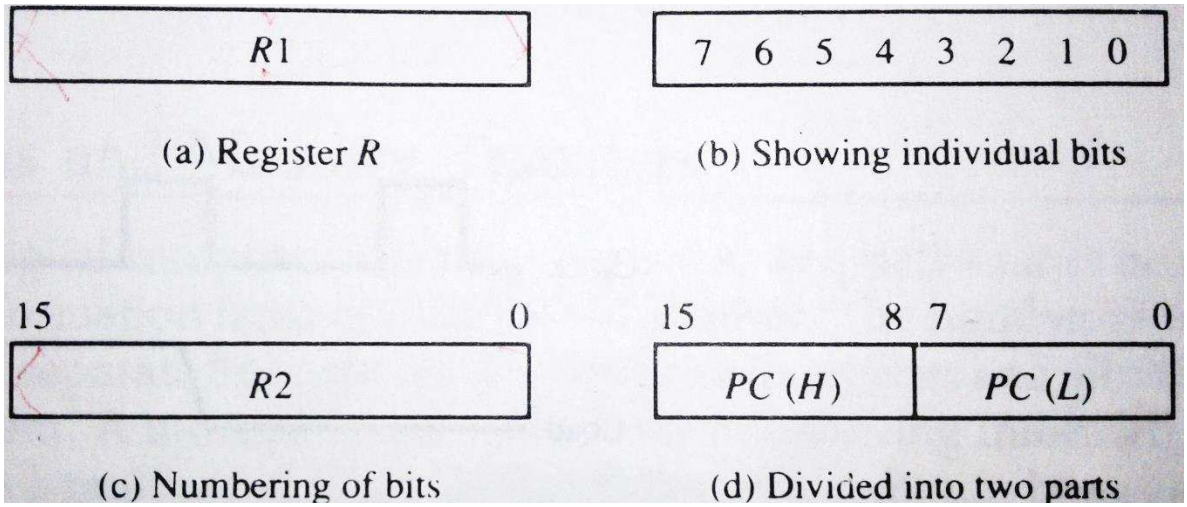
**Register Transfer Language:** The symbolic notation used to describe the micro-operation transfers among registers is called register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated micro-operation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. A register transfer language is a system for expressing in symbolic form the micro-operation

sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

**Register Transfer:** Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register.For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter), IR (for instruction register), and R1 (for processor register). The individual flip flops in an n-bit register are numbered in sequence from 0 through n – 1, starting from 0 in the rightmost position and increasing the numbers towards left. The figure given below shows the representation of registers in block diagram form. Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

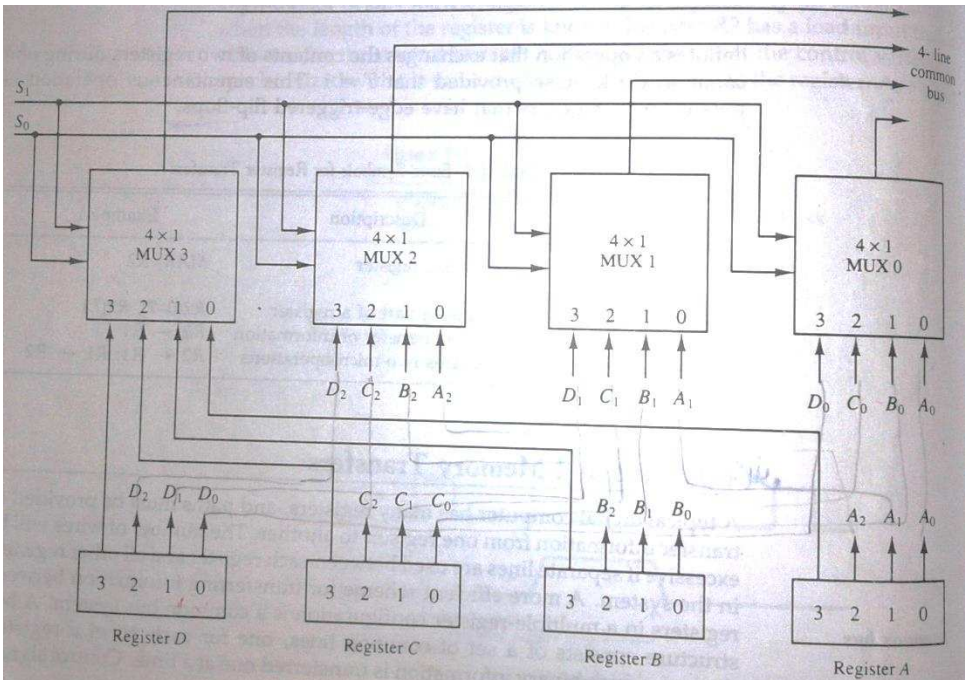
$$R2 \leftarrow R1$$

denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.




**Bus and Memory Transfer:** A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred at one time. Control signals determine which register is selected by the bus during each particular register transfer. One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in figure given below. Each register has four bits,

numbered 0 through 3. The bus consists of for 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S1 and S0.




**Memory Transfer:** The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into memory is called a write operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M. Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read: DR ← M [AR]

This cause a transfer of information into DR from the memory word M selected by the address in AR. The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR. The write operation can be stated symbolically as follows:

Write: M[AR] ← R1

This causes a transfer of information from R1 into the memory word M selected by the address in AR.

**Micro Operation:** A micro-operation is an elementary operation performed normally during one clock pulse. On the information stored in one or more registers. The result of the operation may replace the previous content of a register or is transferred to a new register or a memory location. A digital system performs a sequence of micro-operations on data stored in registers

or memory. The specific sequence of micro-operations performed is predetermined for an instruction. Thus, an instruction is a binary code specifying a definite sequence of micro-operations to perform a specific function.

For example, a C program instruction sum = sum + 7, will first be converted to equivalent assembly program:

- Move data from memory location "sum" to register R1 (LOAD R1, sum)
- Add an immediate operand to register (R1) and store the results in R1 (ADD R1, 7)
- Store data from register R1 to memory location "sum" (STORE sum, R1).

Thus, several machine instructions may be needed (this will vary from machine to machine) to execute a simple C statement. But, how will each of these machine statements be executed with the help of micro-operations? Let us try to elaborate the execution steps:

- Fetch the instructions.
  - Pass the address of Program Counter (PC) to Memory Address Register (MAR).
  - Issue the memory read operation to fetch instruction in the Buffer Register for data, such as M(BR).
  - Increment Program Counter to refer to next instruction in sequence and bring instruction to Instruction Register (IR).
- Execute the instruction
  - Decode the instruction to ascertain operation.
  - As one of the operands is already available in R1 register and the second operand is an immediate operand so fetch operand step is not required. The immediate operand is available in the address part of the instruction.
  - Perform the ALU based addition with R1 and buffer register, store the result in R1.

Thus, we may have to execute the instruction in several steps. For the subsequent discussion, for simplicity, let us assume that each micro-operation can be completed in one clock period, although some micro-operations require memory read/write that may take more time. Let us first discuss the type of micro-operations. The most common micro-operations performed in a digital computer can be classified into four categories:

1. Register transfer micro-operations: simply transfer binary information from one register to another.
2. Arithmetic micro-operations: perform simple arithmetic operations on numeric data stored in registers.
3. Logic micro-operations: perform bit manipulation (logic) operations on nonnumeric data stored in registers.
4. Shift micro-operations registers: perform shift operations on data stored in registers.

1. **Register Transfer Micro-operations:** These micro-operations, as the name suggests transfer information from one register to another. The information does not change during these micro-operations. A register transfer micro-operation may be designed as:

R1← R2. The ← symbol implies that the contents of register R2 are transferred to register R1. R2 here is a source register while R1 is a destination register. We will use this notation throughout this section. Please note the following important points about register transfer micro-operations.

- For a register transfer micro-operation there must be a path for data transfer from the output of the source register to the input of destination register.
- In addition, the destination register should have a parallel load capability, as we expect the register transfer to occur in a predetermined control condition. We will discuss more about the control unit in Unit 4 of this block.
- A common path for connecting various registers is through a common internal data bus of the processor. In general the size of this data bus should be equal to the number of bits in a general register.

2. **Arithmetic Micro-operations:** These micro-operations perform simple arithmetic operations on numeric data **stored in registers**. The basic arithmetic micro-operations are addition, subtraction, increment, decrement, and shift.

Addition micro-operation is specified as:

R3 ←R1 +R2

It means that the contents of register R1 are added to the contents of register R2 and the sum is transferred to register R3. This operation requires three registers to hold data along with the Binary Adder circuit in the ALU. Binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any lengths and is constructed with full-adder circuits connected in cascade. An n-bit binary adder requires n full-adders. Add micro-operation, in accumulator machine, can be performed as:

AC ←AC + DR

Subtraction is most often implemented in machines through complement and adds operations. It is specified as:

R3 ←R1 − R2

R3 ←R1 + (2’s complement of R2)

R3 ←R1 + (1’s complement of R2 + 1)

R3 ←R1 + R2 + 1 (The bar on top of R2 implies 1’s complement of R2 which is bitwise complement)

Adding 1 to the 1’s complement produces the 2’s complement. Adding the contents of R1 to the 2’s complement of R2 is equivalent to subtracting the contents of R2 from R1 and storing the result in R3. We will describe the basic circuit required for these micro-operations in the next unit. The increment micro-operation adds one to a number in a register. This operation is designated as:

R1 ← R1 + 1

This can be implemented in hardware by using a binary-up counter. The decrement micro-operation subtracts one from a number in a register. This operation is designated as:

$$R1 \leftarrow R1 - 1$$

3. **Logic Micro-operations:** Logic operations are basically binary operations, which are performed on the string of bits stored in the registers. For a logic micro-operation each bit of a register is treated as a variable. A logic micro-operation: R1 ← R1. R2 specifies AND operation to be performed on the contents of R1 and R2 and store the results in R1. For example, if R1 and R2 are 8 bits registers and:

   R1 contains 10010011 and
   R2 contains 01010101

   Then R1 will contain 00010001 after AND operation.

   Some of the common logic micro-operations are AND, OR, NOT or Complement, Exclusive OR, NOR, and NAND. In many computers only four: AND, OR, XOR (exclusive OR) and complement micro-operations are implemented.

4. **Shift Micro-operations:** Shift is a useful operation, which can be used for serial transfer of data. Shift operations can also be used along with other (arithmetic, logic, etc.) operations. For example, for implementing a multiply operation arithmetic micro-operation (addition) can be used along with shift operation. The shift operation may result in shifting the contents of a register to the left or right. In a shift left operation a bit of data is input at the right most flip-flop while in shift right a bit of data is input at the left most flipflop. In both the cases a bit of data enters the shift register. Depending on what bit enters the register and where the shift out bit goes, the shifts are classified in three types. These are:
   - Logical
   - Arithmetic and
   - Circular.

   In logical shift the data entering by serial input to left most or right most flip-flop (depending on right or left shift operations respectively) is a 0. If we connect the serial output of a shift register to its serial input then we encounter a circular shift. In circular shift left or circular shift right information is not lost, but is circulated.

   In arithmetic shift a signed binary number is shifted to the left or to the right. Thus, an arithmetic shift-left causes a number to be multiplied by 2, on the other hand a shift-right causes a division by 2. But as in division or multiplication by 2 the sign of a number should not be changed, therefore, arithmetic shift must leave the sign bit unchanged.

**Bus and Memory Transfers:** A digital computer has many registers, and rather than connecting wires between all registers to transfer information between them, a common bus is

used. Bus is a path Execution (consists of a group of wires) one for each bit of a register, over which information is transferred, from any of several sources to any of several destinations.

From a register to Bus: BUS ←R. The implementation of bus is explained in Unit 3 of this block.

The transfer from bus to register can be expressed symbolically as:

$$R1 \leftarrow BUS$$

The content of the selected register is placed on the BUS and the content of the bus is loaded into register R1 by activating its load control input.

**Memory Transfer:** The transfer of information from memory to outside world i.e., I/O Interface is called a read operation. The transfer of new information to be stored in memory is called a write operation. These kinds of transfers are achieved via a system bus. It is necessary to supply the address of the memory location for memory transfer operations.

**Von Neumann Architecture:** The key features of von Neumann Architecture are:

- The most basic function performed by a computer is the execution of a program, which involves:
  - the execution of an instruction, which supplies the information about an operation, and
  - the data on which the operation is to be performed.

  The control unit (CU) interprets each of these instructions and generates respective control signals.
- The Arithmetic Logic Unit (ALU) performs the arithmetic and logical operations in special storage areas called registers as per the instructions of control unit. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing.
- An Input/ Output system involving I/O devices allows data input and reporting of the results in proper form and format. For transfer of information a computer system internally needs the system interconnections. One such interconnection structure is BUS interconnection.
- Main Memory is needed in a computer to store instructions and the data at the time of Program execution. Memory to CPU is an important data transfer path. The amount of information, which can be transferred between CPU and memory, depends on the size of BUS connecting the two.
- It was pointed out by von-Neumann that the same memory can be used for Storing data and instructions. In such a case the data can be treated as data on which processing can be performed, while instructions can be treated as data, which can be used for the generation of control signals.
- The von Neumann machine uses **stored program concept,** i.e., the program and data are stored in the same memory unit for execution. The computers prior to this idea used to store programs and data on separate memories. Entering and modifying these

programs was very difficult as they were entered manually by setting switches, plugging, and unplugging.

- Execution of instructions in von Neumann machine is carried out in a sequential fashion (unless explicitly altered by the program itself) from one instruction to the next.

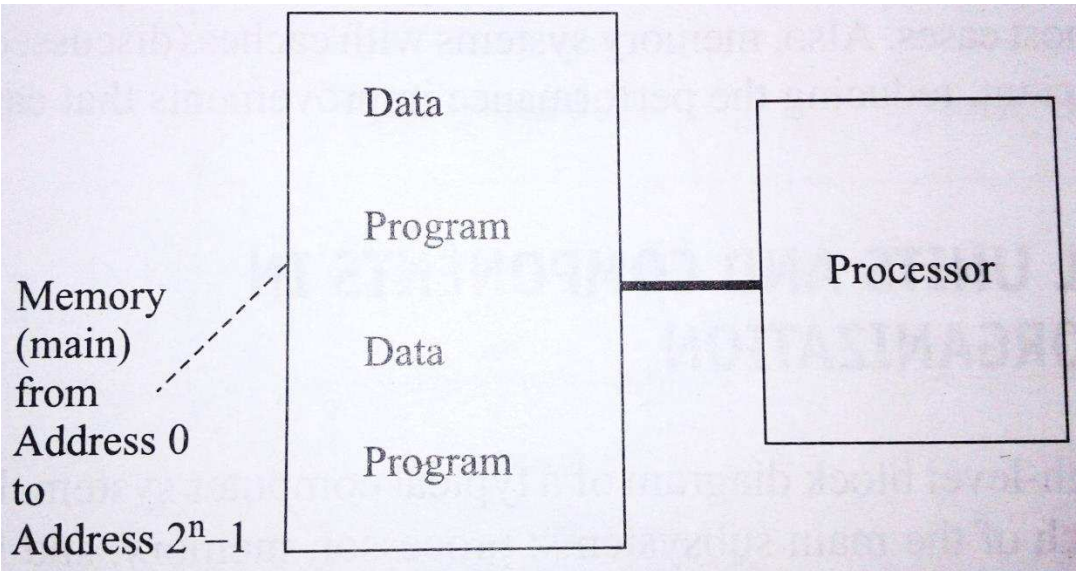
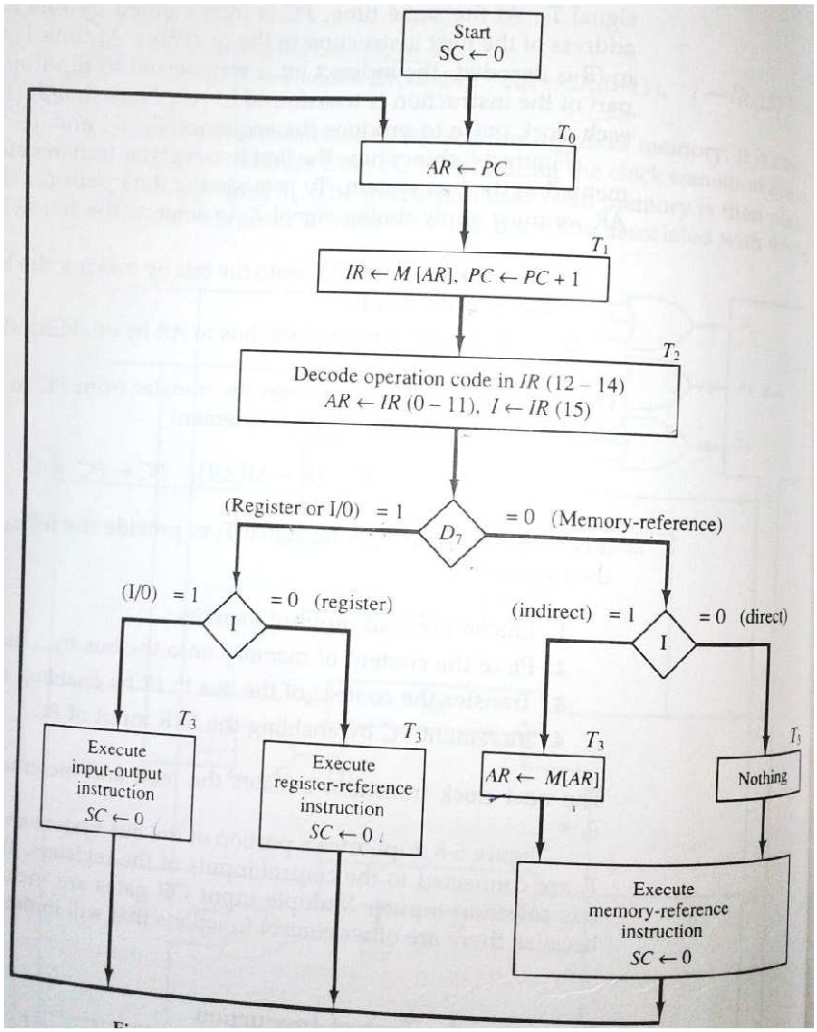



Figure shows the basic structure of a conventional von Neumann machine

**Instruction Cycle:** There are various types of operations that may be required by computer for execution of instruction. The following are the possible steps:

| S. No. | **Step to be performed** | **How is it done** | **Who does it** |
|---|---|---|---|
| 1 | Calculate the address of next instruction to be executed. | The Program Counter (PC register stores the address of next instruction. | Control Unit (CU). |
| 2 | Get the instruction in the CPU register | The memory is accessed and the desired instruction is brought to register (IR) in CPU | Memory Read operation is done. Size of instruction is important. In addition, PC is incremented to point to next instruction in sequence. |
| 3 | Decode the instruction | The control Unit issues necessary control signals | Control Unit. |
| 4 | Evaluate the operand address | CPU evaluates the address based on the addressing mode specified. | CPU under the control of CU. |
| 5 | Fetch the operand | The memory is accessed and the desired operands brought into the CPU Registers. | Memory Read |
| | **Repeat steps 4 and 5 if instruction has more than one operands.** | | |
| 6 | Perform the operation as decoded in steps3. | The ALU does evaluation of arithmetic or logic, instruction or the transfer of control operations. | ALU/CU |

| | | | |
|---|---|---|---|
| 7 | Store the results in memory | The value is written to desired memory location. | Memory write |



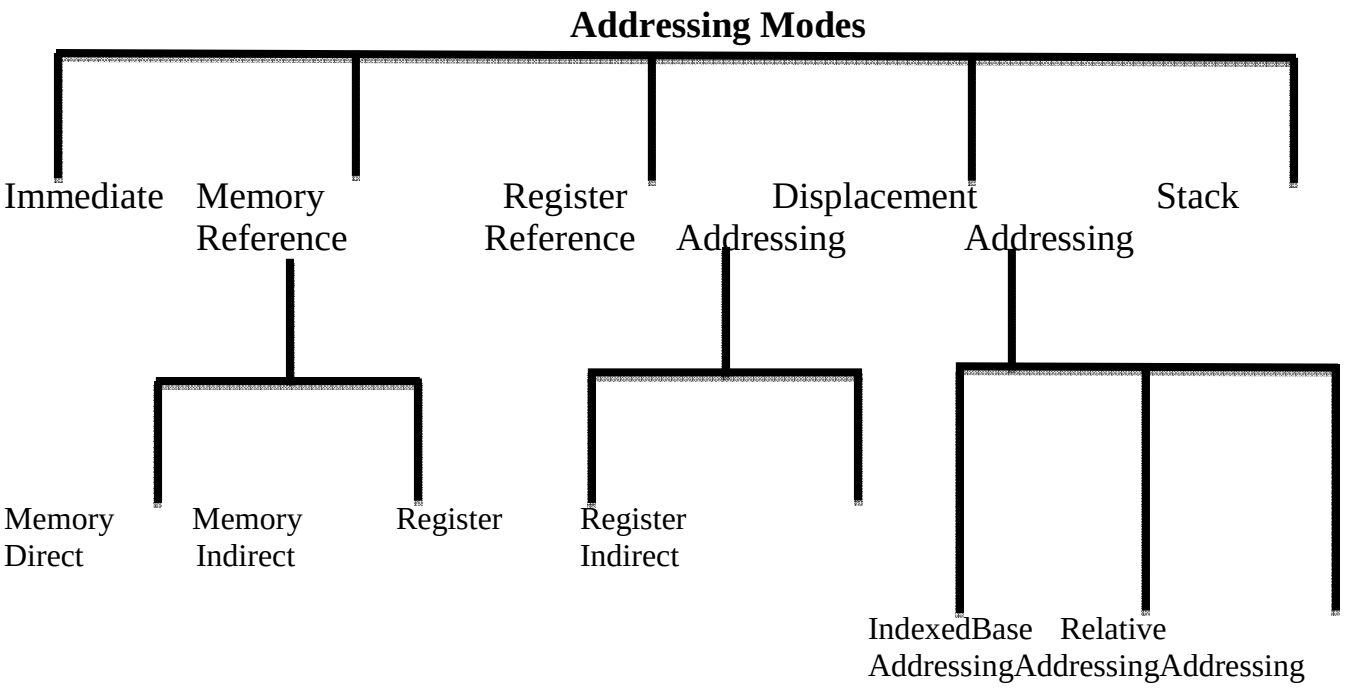

Thus, in general, the execution cycle for a particular instruction may involve more than one stage and memory references. In addition, an instruction may ask for an I/O operation. Considering the steps above, let us work out a more detailed view of instruction cycle. The Figure given above gives a diagram of an instruction cycle. The instruction cycle shown in figure consists of following states/stages:

- First the address of the next instruction is calculated, based on the size of instruction and memory organization. For example, if in a computer an instruction is of 16 bits and if memory is organized as 16-bits words, then the address of the next instruction is evaluated by adding one in the address of the current instruction. In case, the memory is organized as bytes, which can be addressed individually, then we need to add two in the current instruction address to get the address of the next instruction to be executed in sequence.
- Now, the next instruction is fetched from a memory location to the CPU registers such as Instruction register.
- The next state decodes the instruction to determine the type of operation desired and the operands to be used.

- In case the operands need to be fetched from memory or via Input devices, then the address of the memory location or Input device is calculated.
- Next, the operand is fetched (or operands are fetched one by one) from the memory or read from the Input devices.
- Now, the operation, asked by the instruction is performed.
- Finally, the results are written back to memory or Output devices, wherever desired by first calculating the address of the operand and then transferring the values to desired destination.

**Addressing Modes:**The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other.

All computers employ more than one addressing schemes to give programming flexibility to the user by providing facilities such as pointers to memory, loop control, indexing of data, program relocation and to reduce the number of bits in the operand field of the instruction. Offering a variety of addressing modes can help reduce instruction counts but having more modes also increases the complexity of the machine and in turn may increase the average Cycles per Instruction (CPI).Most of the machines employ a set of addressing modes. The following tree shows the common addressing modes:



## Common Addressing Modes

In general not all of the above modes are used for all applications. However, some of the common areas where compilers of high-level languages use them are:

| **Addressing Mode** | **Possible Use** |
|---|---|
| Immediate | For moving constants and initialization of variables |
| Direct | Used for global variables and less often for local variables |
| Register | Frequently used for storing local variables of procedures |
| Register Indirect | For holding pointers to structure in programming languages C |
| Index | To access members of an array |
| Auto-index mode | For pushing or popping the parameters of procedures |
| Base Register | Employed to relocate the programs in memory specially in multi-programming systems |
| Index | Accessing iterative local variables such as arrays |
| Stack | Used for local variables |

1. **Immediate Addressing:** The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

   Operand = A

   This mode can be used to define and use constants or set initial values ofvariables. Typically, the number will be stored in twos complement form; the leftmostbit of the operand field is used as a sign bit. When the operand is loadedinto a data register, the sign bit is extended to the left to the full data word size.In some cases, the immediate binary value is interpreted as an unsigned nonnegativeinteger.

   The advantage of immediate addressing is that no memory reference otherthan the instruction fetch is required to obtain the operand, thus saving one memoryor cache cycle in the instruction cycle.The disadvantage is that the size of the numberis restricted to the size of the address field, which, in most instruction sets, issmall compared with the word length.

Salient points about the addressing mode are:

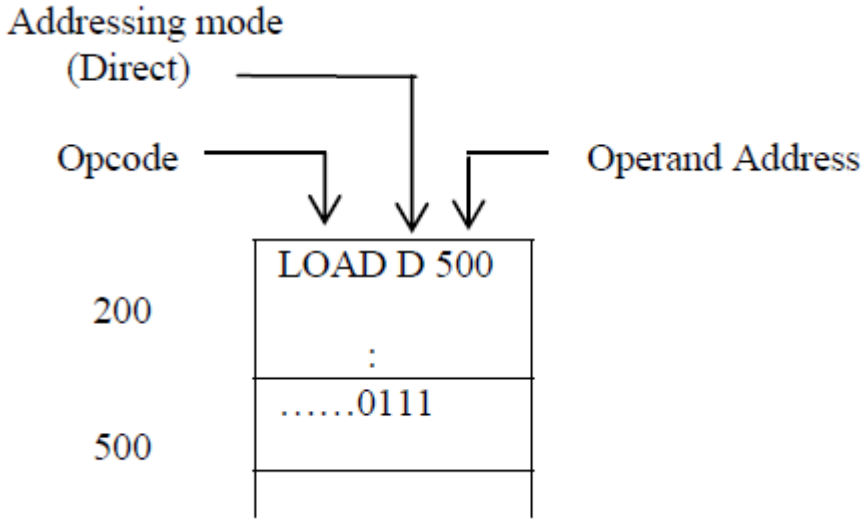
- This addressing mode is used to initialize the value of a variable.
- The advantage of this mode is that no additional memory accesses are required for executing the instruction.
- The size of instruction and operand field is limited. Therefore, the type of data specified under this addressing scheme is also restricted. For example, if an instruction of 16 bits uses 6 bits for opcode and 2 bits for addressing mode, then 10 bits can be used to specify an operand. Thus, $2^{10}$ possible values only can be assigned.

2. **Direct Addressing:** A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

   EA =A

   In this scheme the operand field of the instruction specifies the **direct address** of the intended operand, e.g., if the instruction LOAD 500 uses direct addressing, then it will result in loading the contents of memory cell 500 into the CPU register. In this mode the

intended operand is the address of the data in operation. For example, if memory cell 500 contains 7, as in the diagram below, then the value 7 will be loaded to CPU register.




Some salient points about this scheme are:

- This scheme provides a limited address space because if the address field has n bits then memory space would contain $2^n$ memory words or locations.
- The effective address in this scheme is defined as the address of the operand, that is,

  EA ← A and (EA in the above example will be 500)

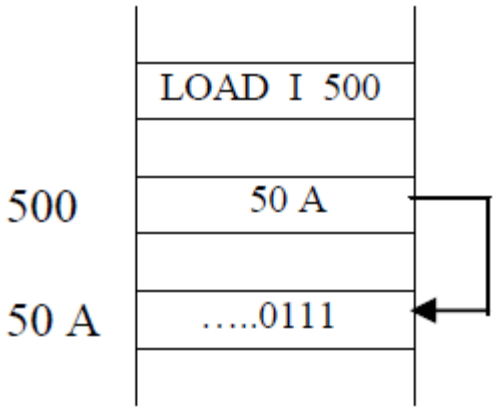
  D = (EA) (D in the above example will be 7)

  The second statement implies that the data is stored in the memory location specified by effective address.
- In this addressing scheme only one memory reference is required to fetch the operand.

3. **Indirect Addressing:**With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as *indirect addressing:*

$$EA = (A)$$

In this scheme the operand field of the instruction specifies the **address** of the **address of** intended operand, e.g., if the instruction LOAD I 500 uses indirect addressing scheme, and contains a value 50A, and memory location 50A contains 7, then the value 7 will get loaded in the CPU register.




Some salient points about this scheme are:

- In this addressing scheme the effective address EA and the contents of the operand field are related as:

  EA = (A) and (Content of location 500 that is 50A above)

  D = (EA) (Contents of location 50A that is 7)
- The drawback of this scheme is that it requires two memory references to fetch the actual operand. The first memory reference is to fetch the actual address of the operand from the memory and the second to fetch the actual operand using that address.
- In this scheme the word length determines the size of addressable space, as the actual address is stored in a Word. For example, the memory having a word size of 32 bits can have $2^{32}$ indirect addresses.

4. **Register Addressing**: Register addressing is similar to direct addressing. The only difference is that the
   address field refers to a register rather than a main memory address:

   EA = R

   OR When operands are taken from register(s), implicitly or explicitly, it is called register addressing. These operands are called register operands. If operands are from memory locations, they are called memory operands. In this scheme, a register address is specified in the instruction. That register contains the operand. It is conceptually similar to direct addressing scheme except that the register name or number is substituted for memory address. Sometimes the address of register may be assumed implicitly, for example, the Accumulator register in old machines.

   In this addressing scheme the effective address is calculated as:

   EA = R

   D = (EA)

   The major advantages of register addressing are:

   Register access is faster than memory access and hence register addressing results in faster instruction execution. However, register obtains operands only from memory; therefore, the operands that should be kept in registers are selected carefully and efficiently.

The size of register address is smaller than the memory address. It reduces the instruction size. For example, for a machine having 32 general purpose registers only 5 bits are needed to address a register.

5. **Register Indirect Addressing:** Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address.

   EA + (R)

   In this addressing scheme, the operand is data in the memory pointed to by a register. In other words, the operand field specifies a register that contains the address of the operand that is stored in memory. This is almost same as indirect addressing scheme except it is much faster than indirect addressing that requires two memory accesses.
   The effective address of the operand in this scheme is calculated as:

   EA= (R) and D = (EA)

6. **Indexed Addressing Scheme:** In this scheme the operand field of the instruction contains an address and an index register, which contains an offset. This addressing scheme is generally used to address the consecutive locations of memory (which may store the elements of an array). The index register is a special CPU register that contains an index value. The contents of the operand field A are taken to be the address of the initial or the reference location (or the first element of array). The index register specifies the distance between the starting address and the address of the operand.
   Thus, the effective address in this scheme is calculated as:

   EA = A +(R)
   D = (EA)
   (DA is Direct address)

7. **Base Register Addressing:** An addressing scheme in which the content of an instruction specifies base register is added to the displacement field or address field of the instruction. This is similar to indexed addressing scheme except that the role of Address field and Register is reversed. In indexing Address field of instruction is fixed and index register value is changed, whereas in Base Register addressing, the Base Register is common and Address field of the instruction in various instructions is changed. In this case:

   EA = A+ (B)
   D = (EA)
   (B) Refers to the contents of a base register B.

8. **Relative Addressing Scheme:** In this addressing scheme, the register R is the program counter (PC) containing the address of the current instruction being executed. The operand field A contains the displacement (positive or negative) of an instruction or data with respect to the current instruction. This addressing scheme has advantages if the memory references are nearer to the current instruction being executed.
9. **Stack Addressing:** In this addressing scheme, the operand is implied as top of stack. It is not explicit, but implied. It uses a CPU Register called Stack Pointer (SP). The SP points to the top of the stack i.e. to the memory location where the last value was pushed. A stack provides a sort-of indirect addressing and indexed addressing. This is

not a very common addressing scheme. The operand is found on the top of a stack. In some machines the top two elements of stack and top of stack pointer is kept in the CPU registers, while the rest of the elements may reside in the memory.

## Unit-II

**Control Unit:** The two basic components of a CPU are the control unit and the arithmetic and logic unit. The control unit of the CPU selects and interprets program instructions and then sees that they are executed. The basic responsibilities of the control unit are to control:

**a)** Data exchange of CPU with the memory or I/O modules.

**b)** Internal operations in the CPU such as:

- moving data between registers (register transfer operations)
- making ALU to perform a particular operation on the data
- regulating other internal operations.

To define the functions of a control unit, one must know what resources and means it has at its disposal. A control unit must know about the:

**a)** Basic components of the CPU

**b)** Micro-operation this CPU performs.

The CPU of a computer consists of the following basic functional components:

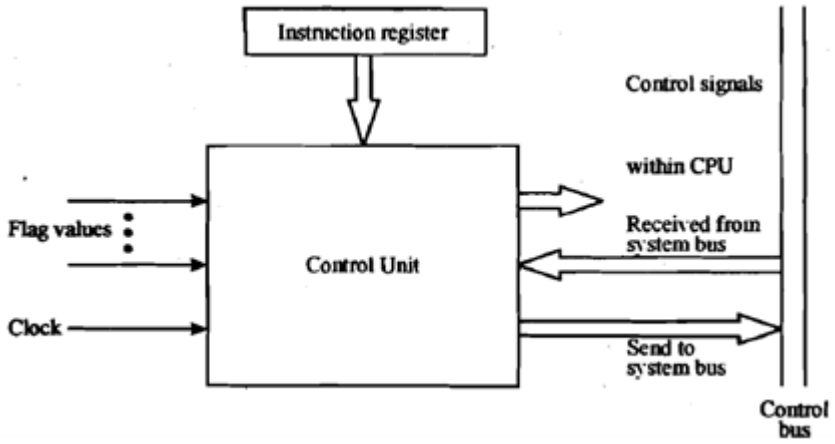
- **The Arithmetic Logic Unit (ALU),** which performs the basic arithmetic and logical operations.
- **Registers** which are used for information storage within the CPU.
- **Internal Data Paths:** These paths are useful for moving the data between two registers or between a register and ALU.
- **External Data Paths:** The roles of these data paths are normally to link the CPU registers with the memory or I/O interfaces. This role is normally fulfilled by the system bus.
- **The Control Unit:** This causes all the operations to happen in the CPU.

The micro-operations performed by the CPU can be classified as:

- Micro-operations for data transfer from register-register, register-memory, I/O register etc.
- Micro- operations for performing arithmetic, logic and shift operations. These micro-operations involve use of registers for input and output.

The basic responsibility of the control unit lies in the fact that the control unit must be able to guide the various components of CPU to perform a specific sequence of micro-operations to achieve the execution of an instruction.

**Structure of Control Unit** A control unit has a set of input values on the basis of which it produces an output control signal, which in turn performs micro-operations. These output signals control the execution of a program. A general model of control unit is shown in Figure given below.



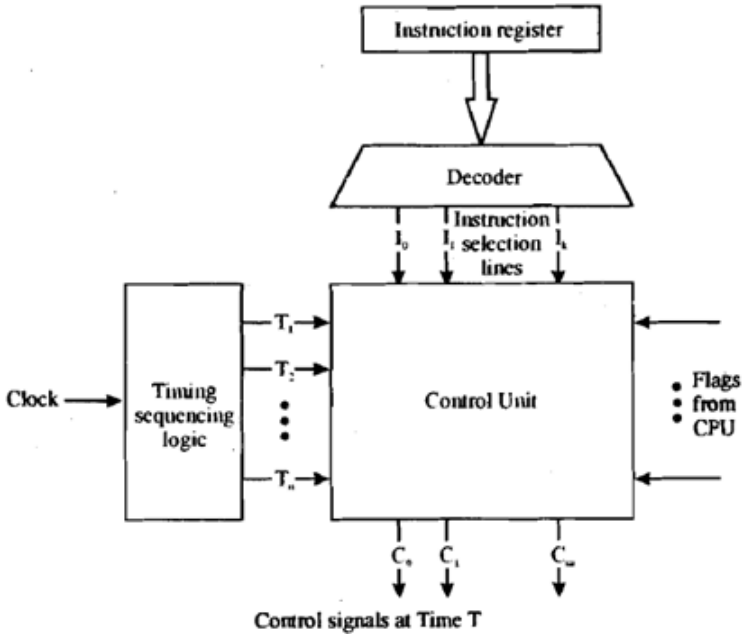

A variety of techniques have been used to organize a control unit. Most of them fall into two major categories:

1. Hardwired control organization
2. Micro programmed control organization

1. **Hardwired control organization:** In the hardwired organization, the control unit is designed as a combinational circuit. That is, the control unit is implemented by gates, flip-flops, decoder and other digital circuits. Hardwired control units can be optimized for fast operations. The block diagram of control unit is shown in Figure 3. The major inputs to the circuit are instruction register, the clock, and the flags. The control unit uses the opcode of instruction stored in the IR register to perform different actions for different instructions. The control unit logic has a unique logic input for each opcode. This simplifies the control logic. This control line selection can be performed by a decoder.

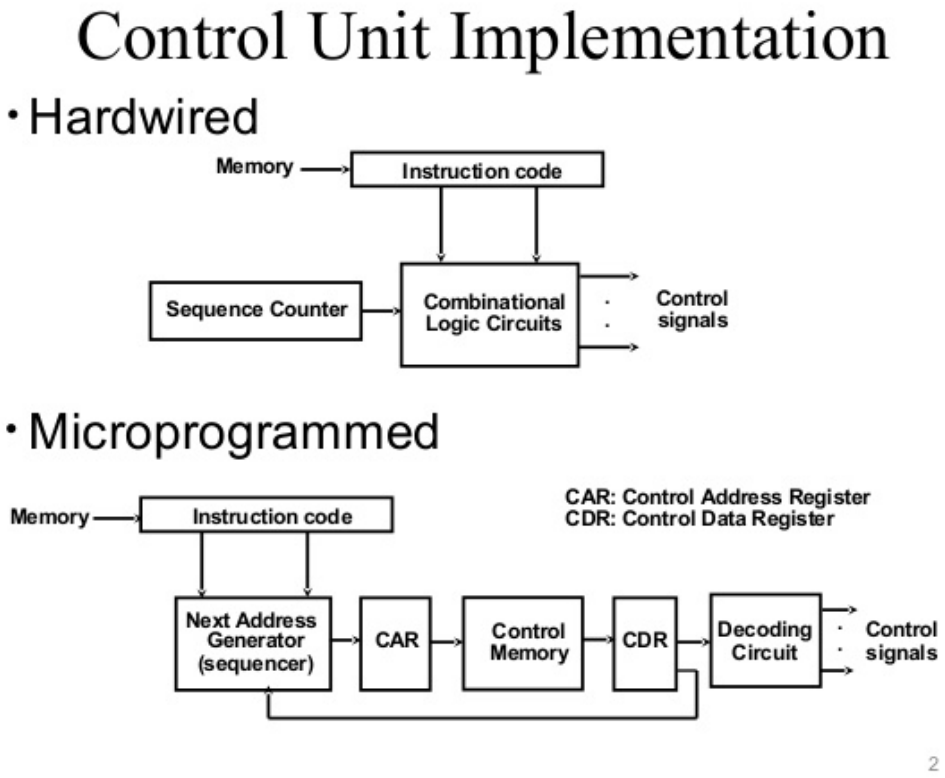
   A decoder will have n binary inputs and $2^n$ binary outputs. Each of these $2^n$ different input patterns will activate a single unique output line. The clock portion of the control unit issues a repetitive sequence of pulses for the SS duration of micro-operation(s). These timing signals control the sequence of execution of instruction and determine what control signal needs to applied at what time for instruction execution.

**Block Diagram of Control Unit Operation**



2. **Micro programmed control organization:** An alternative to a hardwired control unit is a micro-programmed control unit, in which the logic of the control unit is specified by a micro-program. A micro-program is also called firmware (midway between the hardware and the software). It consists of:
    - One or more micro-operations to be executed; and
    - The information about the micro-instruction to be executed next.

The general configuration of a micro-programmed control unit is demonstrated in Figure given below:




Figure(**Operation of Micro-Programmed Control Unit**)

The micro-instructions are stored in the control memory. The address register for the control memory contains the address of the next instruction that is to be read. The control memory Buffer Register receives the micro-instruction that has been read. A micro-instruction execution primarily involves the generation of desired control signals and signals used to determine the next micro-instruction to be executed. The sequencing logic section loads the control memory address register. It also issues a read command to control memory. The following functions are performed by the micro-programmed control unit:

- The sequence logic unit specifies the address of the control memory word that is to be read, in the Address Register of the Control Memory. It also issues the READ signal.
- The desired control memory word is read into control memory Buffer Register.

- The content of the control memory buffer register is decoded to create control signals and next-address information for the sequencing logic unit.
- The sequencing logic unit finds the address of the next control word on the basis of the next-address information from the decoder and the ALU flags.