## UNIT 2:

**Microprogrammed Control:** Control memory, Address sequencing, micro program example, design of control unit.

**Central Processing Unit:** General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

Micro Programmed Control:

Control Memory

 The control unit in a digital computer initiates sequences of microoperations

 The complexity of the digital system is derived form the number of sequences that are performed

 When the control signals are generated by hardware, it is hardwired

 In a bus-oriented system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and ALUs.

 The control unit initiates a series of sequential steps of microoperations

 The control variables can be represented by a string of 1"s and 0"s called a control word

 A microprogrammed control unit is a control unit whose binary control variables are stored in memory

 A sequence of microinstructions constitutes a microprogram

 The control memory can be a read-only memory

 Dynamic microprogramming permits a microprogram to be loaded and uses a writable control memory

 A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory

 The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations

 These microinstructions generate the microoperations to: o fetch the instruction from main memory o evaluate the effective address o execute the operation o return control to the fetch phase for the next instruction

 The control memory address register specifies the address of the microinstruction

 The control data register holds the microinstruction read from memory

 The microinstruction contains a control word that specifies one or more microoperations for the data processor

 The location for the next microinstruction may, or may not be the next in sequence

 Some bits of the present microinstruction control the generation of the address of the next microinstruction

 The next address may also be a function of external input conditions

 While the microoperations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next microinstructions

 Typical functions of a sequencer are: o incrementing the CAR by one o loading into the CAR and address from control memory o transferring an external address o loading an initial address to start the control operations

 A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory

 The address value is the input for the ROM and the control work is the output

 No read signal is required for the ROM as in a RAM

 The main advantage of the microprogrammed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes

 To establish a different control sequence, specify a different set of microinstructions for control memory

**Address Sequencing**

 Microinstructions are stored in control memory in groups, with each group specifying a routine

 Each computer instruction has its own microprogram routine to generate the microoperations

 The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another

 Steps the control must undergo during the execution of a single computer instruction: o Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction

o The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address o The next step is to generate the microoperations that

execute the instruction by considering the opcode and applying a mapping o After execution, control must return to the fetch routine by executing an unconditional branch

 The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained

 Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition

 The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions

 The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic

 The branch logic tests the condition, if met then branches, otherwise, increments the CAR

 If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer

 For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR

 A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located

 The status bits for this type of branch are the bits in the opcode

 Assume an opcode of four bits and a control memory of 128 locations

 The mapping process converts the 4-bit opcode to a 7-bit address for control memory

 This provides for each computer instruction a microprogram routine with a capacity of four microinstructions

 Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body ofthe microprogram

 Frequently many microprograms contain identical section of code

 Microinstructions can be saved by employing subroutines that use common sections of microcode

 Microprograms that use subroutines must have a provisions for storing the return address during a subroutine call and restoring the address during a subroutine return

 A subroutine register is used as the source and destination for the addresses

**Microprogram Example**

• In the block diagram four registers and ALU areassociated with the processor unit. –DR, AR, PC, AC and–ALU•DR can receive information from AC, PC or memory (selected byMUX)•AR can receive information from PC or DR (selected by MUX)•PC can receive information only from AR.

➢ The process of code generation for the control memory is called microprogramming. ➢Transfer of information among registers in the processor is through MUXs rather than a bus.

## Design of Control Unit

The Control Unit is classified into two major categories:

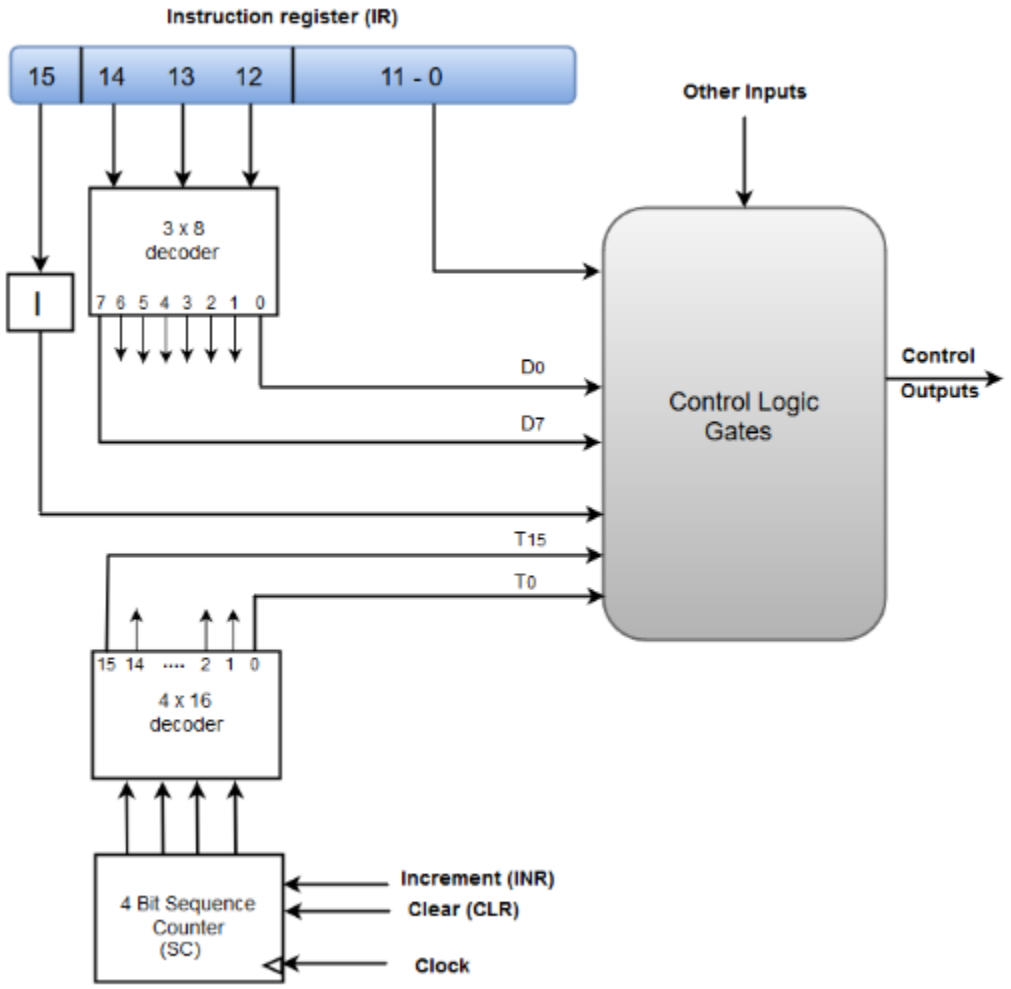
Hardwired Control

Microprogrammed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits.

**The following image shows the block diagram of a Hardwired Control organization**.



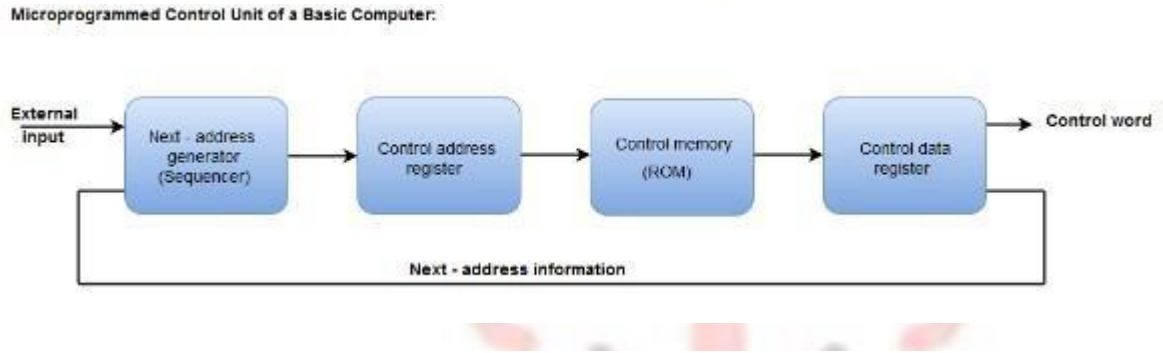

- A Hard-wired Control consists of two decoders, a sequence counter, and a number of logic gates.
- An instruction fetched from the memory unit is placed in the instruction register (IR).
- The component of an instruction register includes; I bit, the operation code, and bits 0 through 11.
- The operation code in bits 12 through 14 are coded with a 3 x 8 decoder.
- The outputs of the decoder are designated by the symbols D0 through D7.
- The operation code at bit 15 is transferred to a flip-flop designated by the symbol I.
- The operation codes from Bits 0 through 11 are applied to the control logic gates.

- The Sequence counter (SC) can count in binary from 0 through 15.

The Microprogrammed Control organization is implemented by using the programming approach.

In Microprogrammed Control, the micro-operations are performed by executing a program consisting of micro-instructions.

The following image shows the block diagram of a Microprogrammed Control organization.




- The Control memory address register specifies the address of the micro-instruction.
- The Control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control register holds the microinstruction fetched from the memory.
- The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.
- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- The next address generator is often referred to as a micro-program sequencer, as it determines the address sequence that is read from control memory.

**Central Processing Unit:**

The operation or task that must perform by CPU is:

• Fetch Instruction: The CPU reads an instruction from memory.

• Interpret Instruction: The instruction is decoded to determine what action is required.

• Fetch Data: The execution of an instruction may require reading data from memory or I/O module.

• Process data: The execution of an instruction may require performing some arithmetic or logical operation on data.

• Write data: The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the CPU needs a small internal memory. These storage locations are generally referred as registers. The major components of the CPU are an arithmetic and logic unit (ALU) and a control unit (CU). The ALU does the actual computation or processing of data. The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU. The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components: Data Bus: Data bus is used to transfer the data between main memory and CPU. Address Bus: Address bus is used to access a particular memory location by putting the address of the memory location. Control Bus: Control bus is used to provide the different control signal generated by CPU to different part of the system. As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

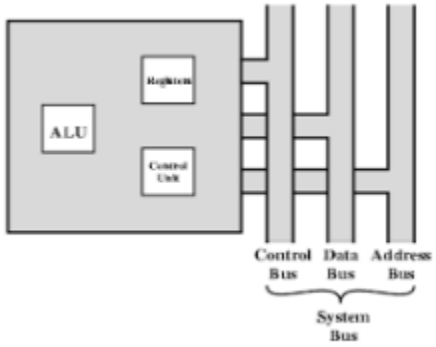
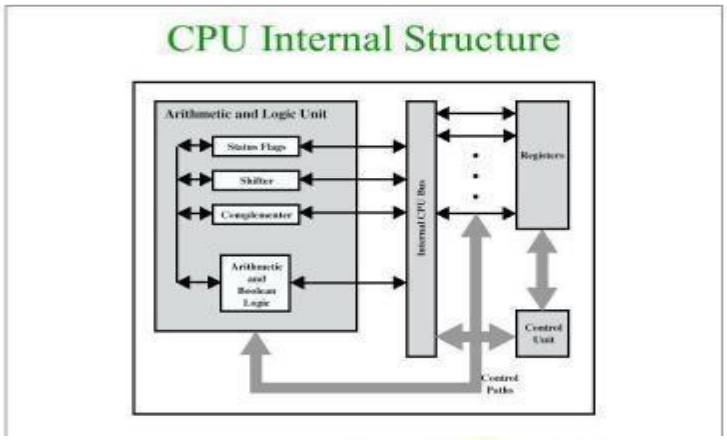



Figure 1: CPU with the system bus.




**Stack Organization:**

A useful feature that is included in the CPU of most computers is a stack or last in, first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off. The stack in digital computers is essentially a memory unit with an address register that can only( after an initial value is loaded in to it).The register that hold the address for the stack is called a stack pointer (SP) because its value always points at the top item in stack. Contrary to a stack of trays where the tray it self may be taken out or inserted, the physical registers of a stack are always available for reading or writing. The two operation of stack are the insertion and deletion of items. The operation of insertion is called PUSH because it can be thought of as the result of pushing a new item on top. The operation of deletion is called POP because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

**INSTRUCTION FORMATS:**

We know that a machine instruction has an opcode and zero or more operands. Encoding an instruction set can be done in a variety of ways. Architectures are differentiated from one another by the number of bits allowed per instruction (16, 32, and 64 are the most common), by the number of operands allowed per instruction, and by the types of instructions and data each can process. More specifically, instruction sets are differentiated by the following features: 1. Operand storage in the CPU (data can be stored in a stack structure or in registers) 2. Number of explicit operands per instruction (zero, one, two, and three being the most common) 3. Operand location (instructions can be classified as register-to-register, register-tomemory or memory-to-memory, which simply refer to

the combinations of operands allowed per instruction) 4. Operations (including not only types of operations but also which instructions can access memory and which cannot) 5. Type and size of operands (operands can be addresses, numbers, or even characters) Number of Addresses: One of the characteristics of the ISA(Industrial Standard Architecture) that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Three-Address Machines:

In three-address machines, instructions carry all three addresses explicitly. The RISC processors use three addresses. Table X1 gives some sample instructions of a threeaddress machine.

In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

mult T,C,D ; T = C*D

add T,T,B ; T = B + C*D

sub T,T,E ; T = B + C*D - E

add T,T,F ; T = B + C*D - E + F

add A,T,A ; A = B + C*D - E + F + A

Table :T1 Sample three-address machine instructions

| Instruction | Semantics |
|---|---|
| add dest,src1,src2 | Adds the two values at src1 and src2 and stores the result in dest<br>M(dest) = [src1] + [src2] |
| sub dest,src1,src2 | Subtracts the second source operand at src2 from the first at src1 and stores the result in dest<br>M(dest) = [src1] - [src2] |
| mult dest,src1,src2 | Multiplies the two values at src1 and src2 and stores the result in dest<br>M(dest) = [src1] * [src2] |

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table.

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary T and in the last one, it is A. This is the motivation for using two addresses, as we show next.

Two-Address Machines : In two-address machines, one address doubles as a source and destination. Usually, we use dest to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. Sample instructions of a two-address machine On these machines, the C statement A = B + C * D - E + F + A is converted to the following code: load T,C ; T = C mult T,D ; T = C*D add T,B ; T = B + C*D sub T,E ; T = B + C*D - E add T,F ; T = B + C*D - E + F add A,T ; A = B + C*D - E + F + A Table :T2 Sample Two-address machine instructions:

| Instruction | Semantics |
|---|---|
| load dest,src | Copies the value at src to dest<br>M(dest) = [src] |
| add dest,src | Adds the two values at src and dest and stores the result in dest<br>M(dest) = [dest] + [src] |
| sub dest,src | Subtracts the second source operand at src from the first at dest and stores the result in dest<br>M(dest) = [dest] - [src] |
| mult dest,src | Multiplies the two values at src and dest and stores the result in dest<br>M(dest) = [dest] * [src] |

One-Address Machines : In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the accumulators. In most machines, there is just a single accumulator register. This kind of design, called accumulator machines, makes sense if memory is expensive. In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to

store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table X3. In these machines, the C statement A = B + C * D - E + F + A is converted to the following code: load C ; load C into the accumulator mult D ; accumulator = C*D add B ; accumulator = C*D+B sub E ; accumulator = C*D+B-E add F ; accumulator = C*D+B-E+F.

add A ; accumulator = C*D+B-E+F+A store A ; store the accumulator contents in A

Table :T3 Sample ONE-address machine instructions

| Instruction | | Semantics |
|---|---|---|
| load | addr | Copies the value at address addr into the accumulator accumulator = [addr] |
| store | addr | Stores the value in the accumulator at the memory address addr M(addr) = accumulator |
| add | addr | Adds the contents of the accumulator and value at address addr accumulator = accumulator + [addr] |
| sub | addr | Subtracts the value at memory address addr from the contents of the accumulator accumulator = accumulator - [addr] |
| mult | addr | Multiplies the contents of the accumulator and value at address addr accumulator = accumulator * [addr] |

Zero-Address Machines : In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table X4 gives some sample instructions for the stack machines.

Table :T4 Sample Zero-address machine instructions

| Instruction | Semantics |
|---|---|
| push addr | Places the value at address addr on top of the stack push([addr]) |
| pop addr | Stores the top value on the stack at memory address addr M(addr) = pop |
| add | Adds the top two values on the stack and pushes the result onto the stack push(pop + pop) |
| sub | Subtracts the second top value from the top value of the stack and pushes the result onto the stack push(pop – pop) |
| mult | Multiplies the top two values in the stack and pushes the result onto the stack push(pop * pop) |

two are special instructions that use a single address and are used to move data between memory and stack.

All other instructions use the zero-address format. Let‟s see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

A = B + C * D - E + F + A

is converted to the following code:

push E ; <E>

push C ; <C, E>

push D ; <D, C, E>

mult ; <C*D, E>

push B ; <B, C*D, E>

add ; <B+C*D, E>

sub ; <B+C*D-E>

push F ; <F, B+D*C-E>

add ; <F+B+D*C-E>

push A ; <A, F+B+D*C-E>

add ; <A+F+B+D*C-E>

pop A ; < >

On the right, we show the state of the stack after executing each instruction.

The top element of the stack is shown on the left. Notice that we pushed E early because we need to subtract it from (B+C*D).

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the stack depth. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth.

**Addressing Modes**

We have examined the types of operands and operations that may be specified by machine instructions. Now we have to see how is the address of an operand specified, and how are the bits of an instruction organized to define the operand addresses and operation of that instruction

Addressing Modes: The most common addressing techniques are

• Immediate

• Direct

• Indirect

• Register

• Register Indirect

• Displacement

• Stack

All computer architectures provide more than one of these addressing modes.

The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of effective address. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer. To explain the addressing modes, we use the following notation:

| | | |
|---|---|---|
| **A** | = | contents of an address field in the instruction that refers to a memory |
| **R** | = | contents of an address field in the instruction that refers to a register |
| **EA** | = | actual (effective) address of the location containing the referenced operand |
| **(X)** | = | contents of location X |

**Immediate Addressing:**

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction: OPERAND = A This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the

number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

**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

It requires only one memory reference and no special calculation.

**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.

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

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.

The exact register location of the operand in case of Register Addressing Mode is shown in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.

**Diaplacement Addressing**: A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing: EA = A + (R) Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are

added to A to produce the effective address. The general format of Displacement Addressing is shown in the Figure 4.6. Three of the most common use of displacement addressing are: • Relative addressing • Base-register addressing • Indexing

Relative Addressing: For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction. Base-Register Addressing: The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. Indexing: The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

## UNIT 3:

**Data Representation:** Data types, Complements, Fixed Point Representation, Floating Point Representation.

**Computer Arithmetic:** Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating–point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

### BASIC COMPUTER DATA TYPES:

• Binary information in digital computers is stored in memory or processor registers.

• The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form.

• A number system of base or radix r is a system of that uses distinct symbols for r digits

• The decimal number system in everyday use employs radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6,7, 8, and 9; highest number being r-1

The binary number system uses the radix 2. The two digit symbols used are 0 and 1.

• The string of digits 101101 is interpreted to represent 1 x 25 + 0 x 24 + 1 x 23 + 1 x 22 + 0 x 21 + 1 x 20 = 45

• Besides the decimal and binary number systems,

• Octal (radix 8)- 0,1,2,3,4,5,6,7 and