Unit-III

Part-1: MICROPROGRAMMED CONTROL

Contents:

- ✓ Control memory
- ✓ Address Sequencing
- ✓ Microprogram Example
- ✓ Design of Control Unit

Introduction:

- The function of the control unit in a digital computer is to initiate sequence of microoperations.
- Control unit can be implemented in two ways
  - Hardwired control
  - Microprogrammed control

Hardwired Control:

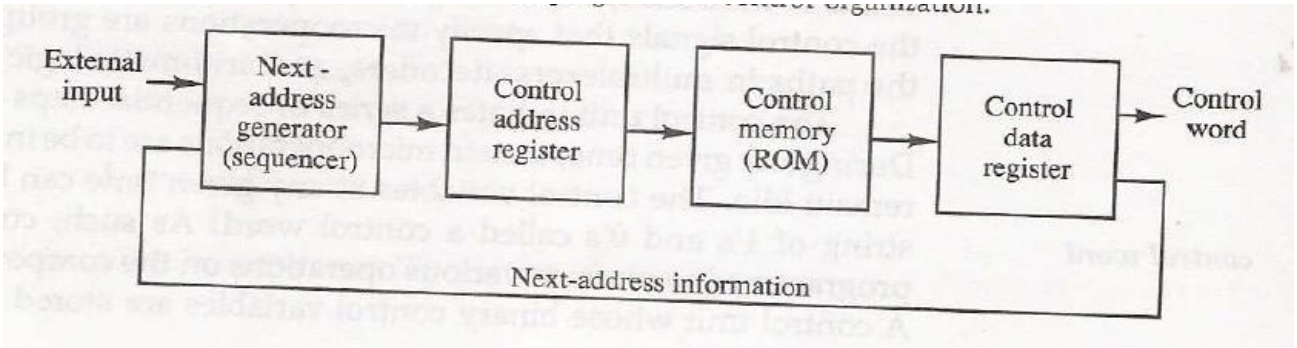
- ✓ When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be *hardwired.*
- ✓ The key characteristics are
  - High speed of operation
  - Expensive
  - Relatively complex
  - No flexibility of adding new instructions
- ✓ Examples of CPU with hardwired control unit are Intel 8085, Motorola 6802, Zilog 80, and any RISC CPUs.

Microprogrammed Control:

- ✓ Control information is stored in control memory.
- ✓ Control memory is programmed to initiate the required sequence of micro-operations.
- ✓ The key characteristics are
  - Speed of operation is low when compared with hardwired
  - Less complex
  - Less expensive
  - Flexibility to add new instructions
- ✓ Examples of CPU with microprogrammed control unit are Intel 8080, Motorola 68000 and any CISC CPUs.

1. Control Memory:

- The control function that specifies a microoperation is called as ***control variable.***
- When control variable is in one binary state, the corresponding microoperation is executed. For the other binary state the state of registers does not change.
- The active state of a control variable may be either 1 state or the 0 state, depending on the application.
- For bus-organized systems the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.
- ***Control Word:*** The control variables at any given time can be represented by a string of 1's and 0's called a control word.
- All control words can be programmed to perform various operations on the components of the system.
- ***Microprogram control unit:*** A control unit whose binary control variables are stored in memory is called a microprogram control unit.
- The control word in control memory contains within it a *microinstruction.*
- The microinstruction specifies one or more micro-operations for the system.
- A sequence of microinstructions constitutes a *microprogram*.
- The control unit consists of control memory used to store the microprogram.
- Control memory is a permanent i.e., read only memory (ROM).
- The general configuration of a micro-programmed control unit organization is shown as block diagram below.



- The control memory is ROM so all control information is permanently stored.
- The control memory address register (CAR) specifies the address of the microinstruction and the control data register (CDR) holds the microinstruction read from memory.
- The next address generator is sometimes called a microprogram sequencer. It is used to generate the next micro instruction address.
- The location of the next microinstruction may be the one next in sequence or it may be located somewhere else in the control memory.
- So it is necessary to use some bits of the present microinstruction to control the generation of the address of the microinstruction.
- Sometimes the next address may also be a function of external input conditions.
- The control data register holds the present microinstruction while next address is computed and read from memory. The data register is times called a *pipeline register.*

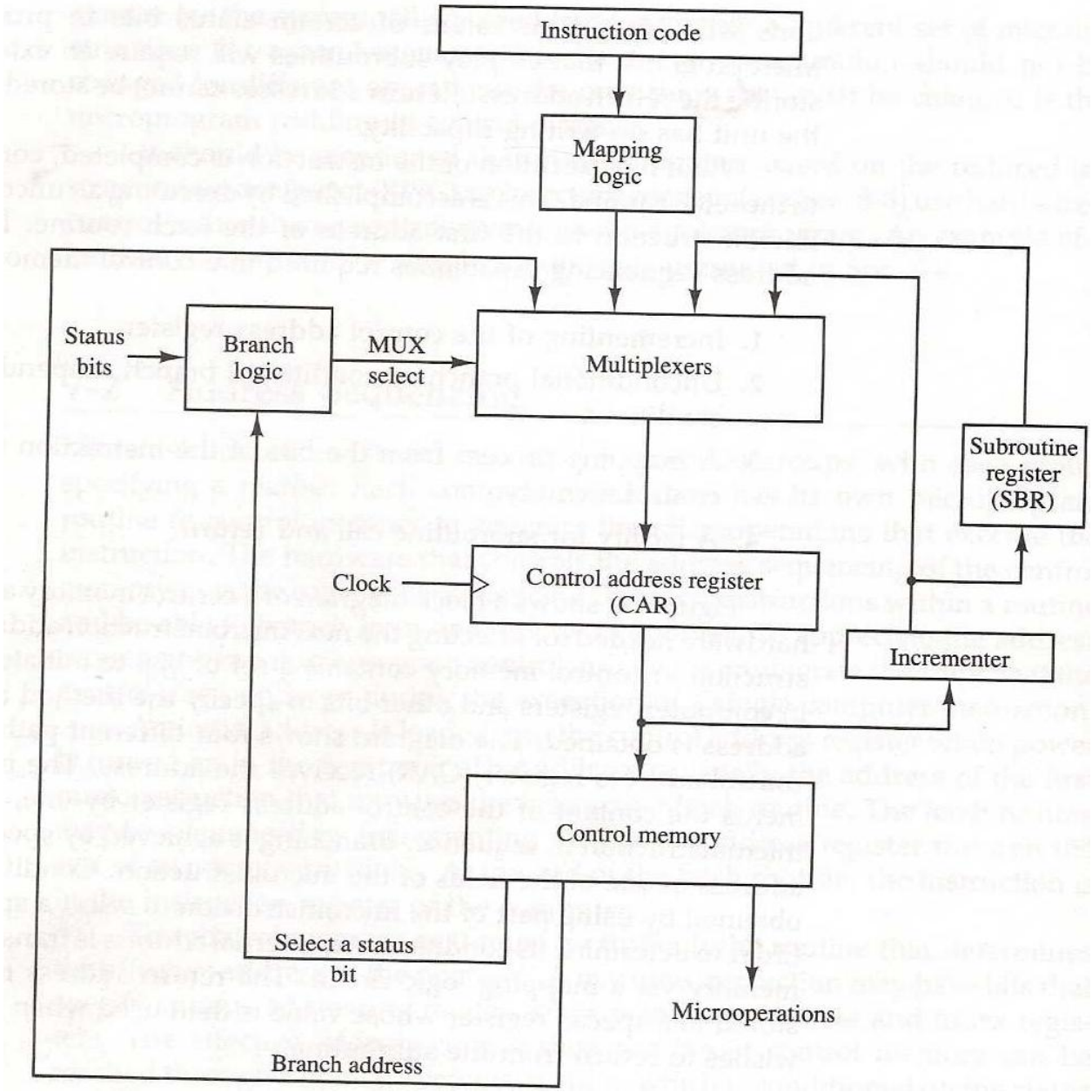
- A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory
- The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations
- These microinstructions generate the microoperations to:
    - fetch the instruction from main memory
    - evaluate the effective address
    - execute the operation
    - return control to the fetch phase for the next instruction

2. 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:
    - 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
    - The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
    - The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a *mapping process.*
        - *The transformation of the instruction code bits to an address in control memory where the routine of instruction located is referred to as mapping process.*
    - After execution, control must return to the fetch routine by executing an unconditional branch
- In brief the address sequencing capabilities required in a control memory are:
    - Incrementing of the control address register.
    - Unconditional branch or conditional branch, depending on status bit conditions.
    - A mapping process from the bits of the instruction to an address for control memory.

- A facility for subroutine call and return.

- The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.



- 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.
- In the figure four different paths form which the control address register (CAR) receives the address.
  - The incrementer increments the content of the control register address register by one, to select the next microinstruction in sequence.
  - Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
  - Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
  - An external address is transferred into control memory via a mapping logic circuit.
  - The return address for a subroutine is stored in a special register, that value is used when the micoprogram wishes to return from the subroutine.

Conditional Branching:

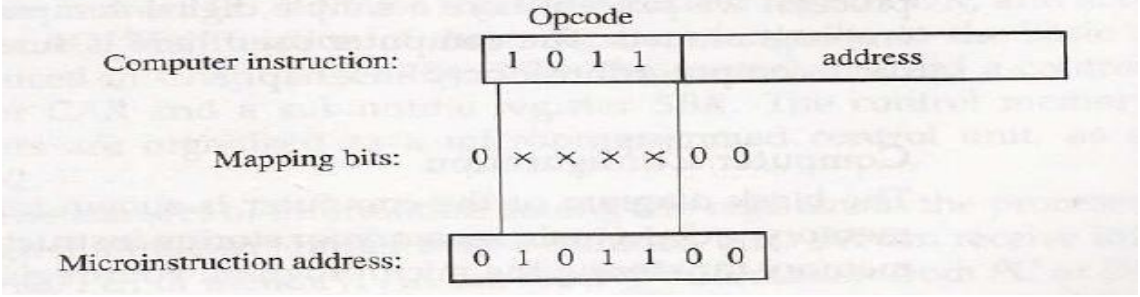
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode 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.

Mapping of Instruction:

- 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 shown in below figure.

Figure 7-3 Mapping from instruction code to microinstruction address.



- Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

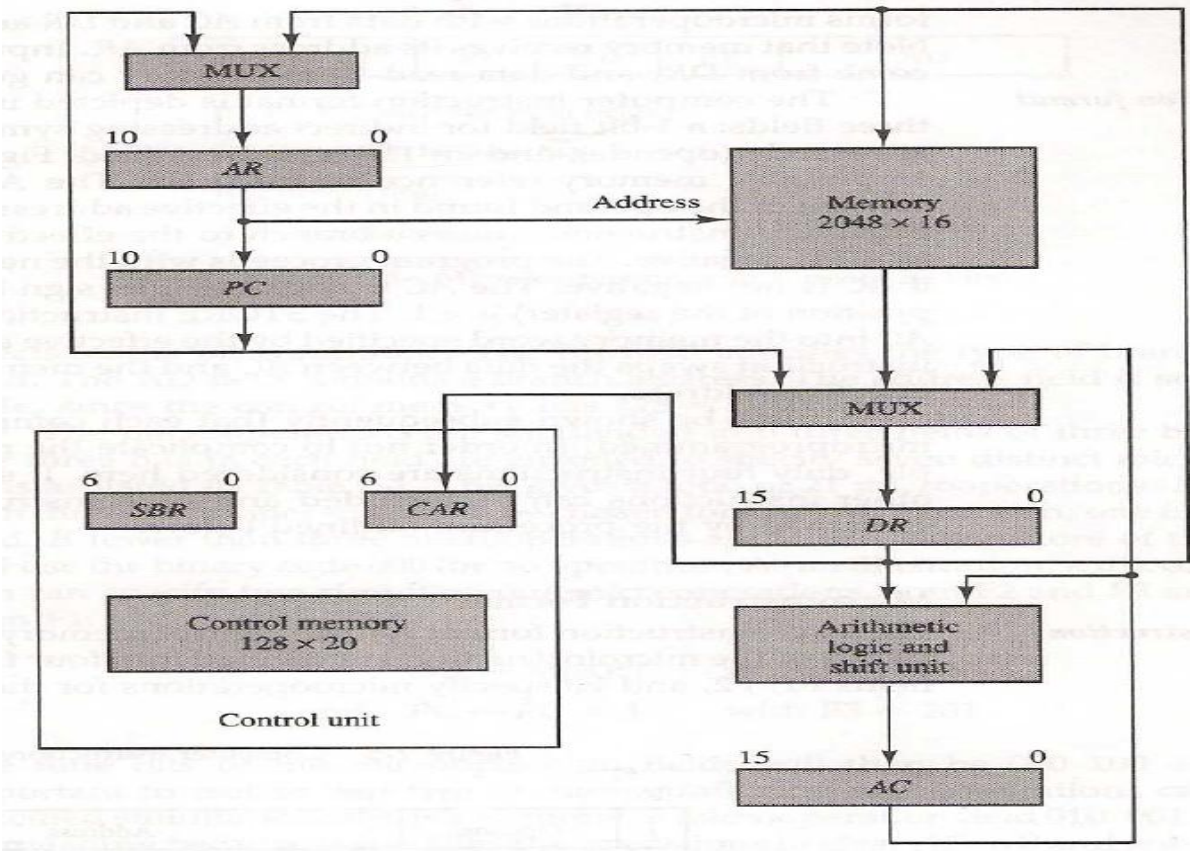
Subroutines:

- 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 of the 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 provision 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

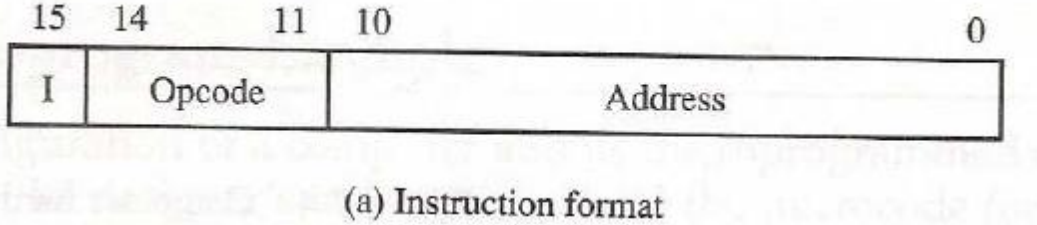
3. Microprogram Example:

- The process of code generation for the control memory is called *microprogramming.*
- The block diagram of the computer configuration is shown in below figure.
- Two memory units:
  - Main memory – stores instructions and data
  - Control memory – stores microprogram
- Four processor registers
  - Program counter – PC
  - Address register – AR
  - Data register – DR
  - Accumulator register - AC
- Two control unit registers
  - Control address register – CAR
  - Subroutine register – SBR

- Transfer of information among registers in the processor is through MUXs rather than a bus.



- The computer instruction format is shown in below figure.



(a) Instruction format

- Three fields for an instruction:
  - 1-bit field for indirect addressing
  - 4-bit opcode
  - 11-bit address field

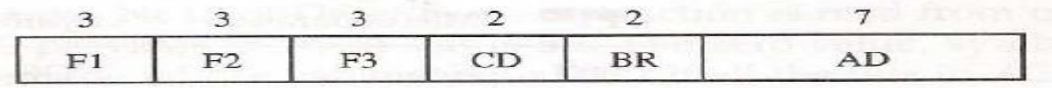
- The example will only consider the following 4 of the possible 16 memory instructions

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M\,[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M\,[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA],\ M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

- The microinstruction format for the control memory is shown in below figure.



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

- The microinstruction format is composed of 20 bits with four parts to it
  - Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]
  - The CD field selects status bit conditions [2 bits]
  - The BR field specifies the type of branch to be used [2 bits]
  - The AD field contains a branch address [7 bits]
- Each of the three microoperation fields can specify one of seven possibilities.
- No more than three microoperations can be chosen for a microinstruction.
- If fewer than three are needed, the code 000 = NOP.
- The three bits in each field are encoded to specify seven distinct microoperations listed in below table.

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

- Five letters to specify a transfer-type microoperation
  - First two designate the source register
  - Third is a 'T'
  - Last two designate the destination register

  AC ← DR F1 = 100 = DRTAC
- The condition field (CD) is two bits to specify four status bit conditions shown below

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

- The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction.

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

- Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five parts
    1. The label field may be empty or it may specify a symbolic address. Terminate with a colon (:).
    2. The microoperations field consists of 1-3 symbols, separated by commas. Only one symbol from each field. If NOP, then translated to 9 zeros
    3. The condition field specifies one of the four conditions
    4. The branch field has one of the four branch symbols
    5. The address field has three formats
        a. A symbolic address – must also be a label
        b. The symbol NEXT to designate the next address in sequence
        c. Empty if the branch field is RET or MAP and is converted to 7 zeros
- The symbol ORG defines the first address of a microprogram routine.
- ORG 64 – places first microinstruction at control memory 1000000.

Fetch Routine:

- The control memory has 128 locations, each one is 20 bits.
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63.
- Can start the fetch routine at address 64.
- The fetch routine requires the following three microinstructions (locations 64-66).
- The microinstructions needed for fetch routine are:

$AR \leftarrow PC$

$DR \leftarrow M[AR]$, $PC \leftarrow PC + 1$

$AR \leftarrow DR(0\text{–}10)$, $CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

- It's Symbolic microprogram:

```
        ORG 64
FETCH:  PCTAR           U    JMP    NEXT
        READ, INCPC     U    JMP    NEXT
        DRTAR           U    MAP
```

- It's Binary microprogram:

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

4. Design of control Unit:

- The control memory out of each subfield must be decoded to provide the distinct microoperations.
- The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- The below figure shows the three decoders and some of the connections that must be made from their outputs.

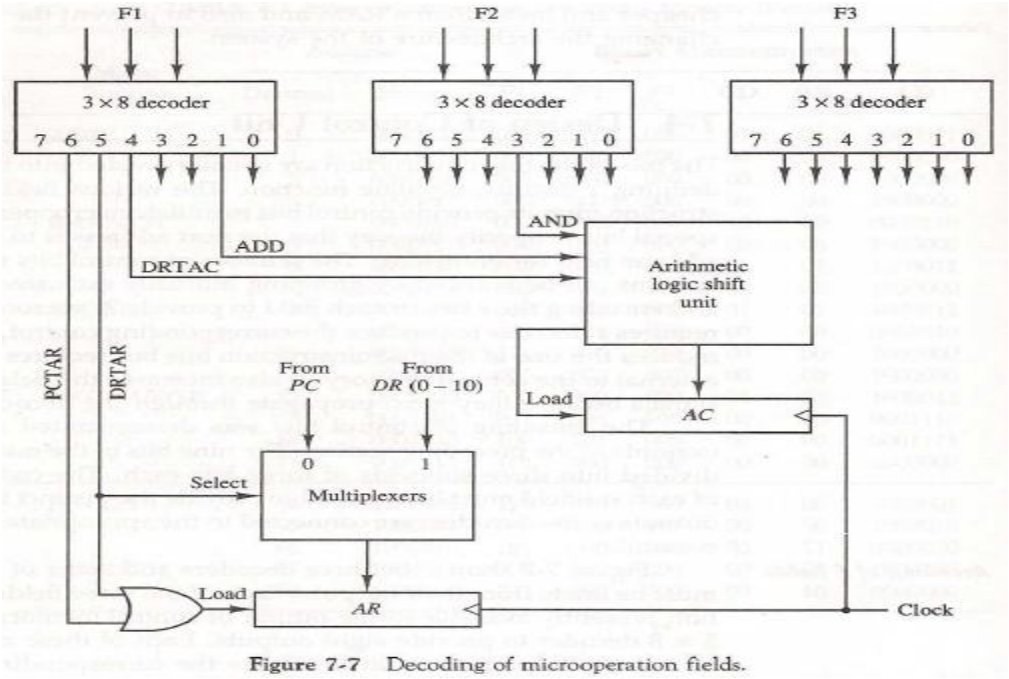


Figure 7-7 Decoding of microoperation fields.

- The three fields of the microinstruction in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.
- Each of the output must be connected to proper circuit to initiate the corresponding microoperation as specified in previous topic.
- When F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.
- Similarly, when F1= 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig, outputs 5 and 6 of decoder *F1* are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.
- The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.
- The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.
- For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively.

Microprogram Sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.
- The block diagram of the microprogram sequencer is shown in below figure.
- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
    - The first multiplexer selects an address from one of four sources and routes it into control address register *CAR.*
    - The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.

- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR.*
- The other three inputs to multiplexer come from
  - The address field of the present microinstruction
  - From the out of SBR
  - From an external source that maps the instruction
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T variable is equal to 1; otherwise, it is equal to 0.
- The *T* value together with two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.

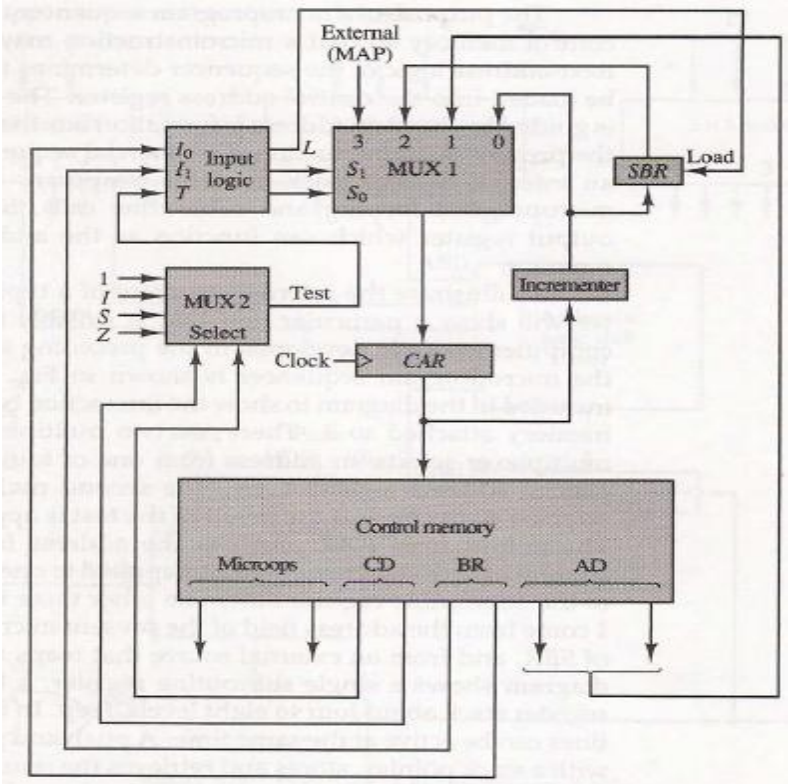


Figure 7-8 Microprogram sequencer for a control memory.

- The input logic circuit in above figure has three inputs I_0, I_1, and T, and three outputs, S_0, S_1, and L.
- Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR.
- The binary values of the selection variables determine the path in the multiplexer.
- For example, with S_1S_0 = 10, multiplexer input number 2 is selected and establishes transfer path from SBR to CAR.
- The truth table for the input logic circuit is shown in Table below.

BR Field	Input			MUX 1		Load *SBR*
	I_1	I_0	T	S_1	S_0	L
0 0	0	0	0	0	0	0
0 0	0	0	1	0	1	0
0 1	0	1	0	0	0	0
0 1	0	1	1	0	1	1
1 0	1	0	×	1	0	0
1 1	1	1	×	1	1	0

- Inputs I_1 and I_0 are identical to the bit values in the BR field.
- The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1).
- The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

$$S_1 = I_1$$
$$S_0 = I_1 I_0 + I'_1 T$$
$$L = I'_1 T I_0$$

UNIT-II

Part-2: CENTRAL PROCESSING UNIT

- ✓ Stack Organization
- ✓ Instruction Formats
- ✓ Addressing Modes
- ✓ Data Transfer And Manipulation
- ✓ Program Control
- ✓ Reduced Instruction Set Computer (RISC)

Introduction:

- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Fig. 8-1.

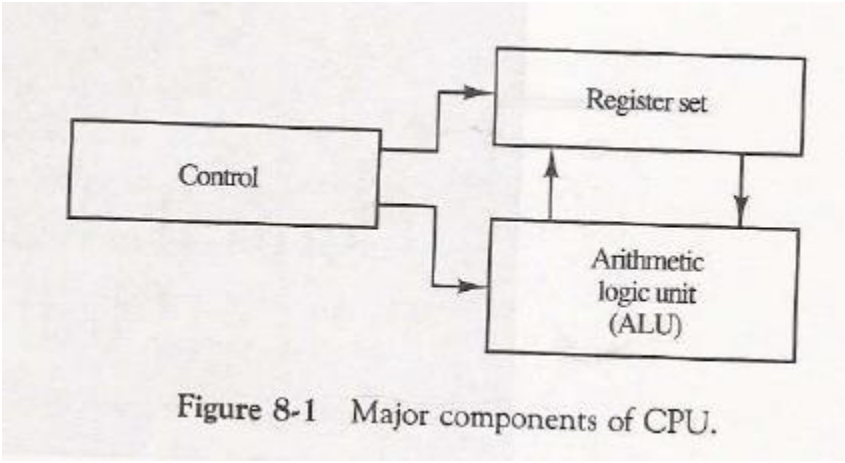


Figure 8-1 Major components of CPU.

- The register set stores intermediate data used during the execution of the instructions.
- The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

1. Stack Organization:

- A stack or last-in first-out (LIFO) is useful feature that is included in the CPU of most computers.
- Stack:
  - 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.
- In the computer stack is a memory unit with an address register that can count the address only.
- The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack.
- The two operations that are performed on stack are the insertion and deletion.
- The operation of insertion is called *PUSH.*
- The operation of deletion is called *POP.*
- These operations are simulated by incrementing and decrementing the stack pointer register (SP).

Register Stack:

- A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.
- The below figure shows the organization of a 64-word register stack.

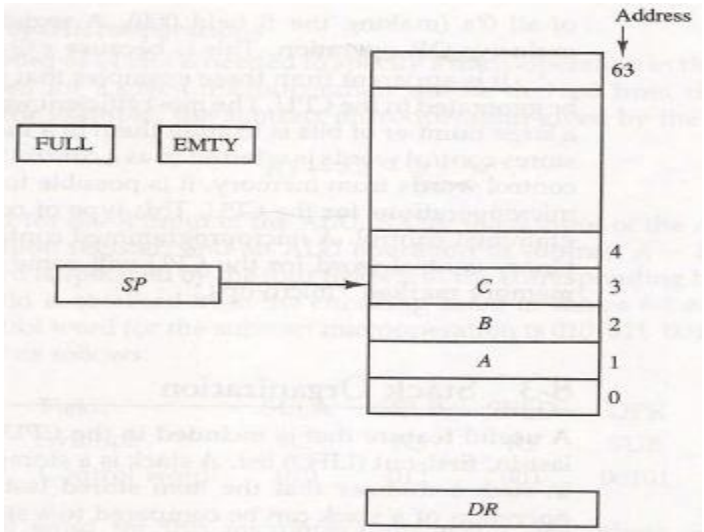


Figure 8-3 Block diagram of a 64-word stack.

- The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, *B, C, in* that order.
- In above figure C is on top of the stack so that the content of *SP* is 3.
- For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- Now the top of the stack is B, so that the content of SP is 2.
- Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.
- In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.
- Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significant bits.
- Then the one-bit register FULL is set to 1, when the stack is full.
- Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 1 when the stack is empty of items.
- DR is the data register that holds the binary data to be written into or read out of the stack.

PUSH:

- Initially, *SP* is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full.
- If the stack is not full (if FULL = 0), a new item is inserted with a push operation.
- The push operation is implemented with the following sequence of microoperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

- The stack pointer is incremented so that it points to the address of next-higher word.
- A memory write operation inserts the word from DR the top of the stack.

- The first item stored in the stack is at address 1.
- The last item is stored at address 0.
- If *SP* reaches 0, the stack is full of items, so FULL is to 1.
- This condition is reached if the top item prior to the last push way location 63 and, after incrementing *SP,* the last item is stored in location 0.
- Once an item is stored in location 0, there are no more empty registers in the stack, so the EMTY is cleared to 0.

POP:

- A new item is deleted from the stack if the stack is not empty (if EMTY = 0).
- The pop operation consists of the following sequence of min operations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

- The top item is read from the stack into DR.
- The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set 1.
- This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value of SP.
- If a pop operation reads the item from location 0 and then is decremented, *SP* changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in the stack.

Memory Stack:

- In the above discussion a stack can exist as a stand-alone unit. But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stack pointer.
- The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.

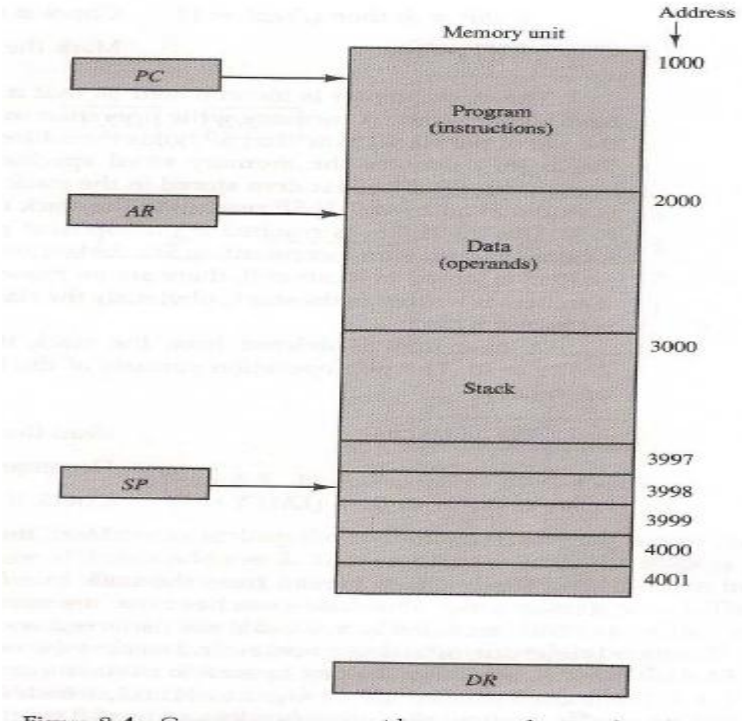


Figure 8-4 Computer memory with program, data, and stack segments.

- The program counter *PC* points at the address of the next instruction in program.
- The address register AR points at an array of data.
- The stack pointer SP points at the top of the stack.

- The three registers are connected to a common address bus, and either one can provide an address for memory.
    - PC is used during the fetch phase to read an instruction.
    - AR is used during the exec phase to read an operand.
    - *SP* is used to push or pop items into or from stack.
- As shown in Fig. 8-4, the initial value of SP is 4001 and the stack grows with decreasing addresses.
- Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.
- No provisions are available for stack limit checks.
- The items in the stack communicate with a data register *DR.* A new item is inserted with the push operation as follows:

    SP← SP-1

    M [SP] ← DR

- The stack pointer is decremented so that it points at the address of the next word.
- A memory write operation inserts the word from DR into the top of stack. A new item is deleted with a pop operation as follows:

    DR ← M [SP]

    SP← SP+1

- The top item is read from the stack into DR. The stack pointer is then decremented to point at the next item in the stack.
- Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).
- The stack limits can be checked by using processor registers:
    - one to hold the upper limit (3000 in this case)
    - Other to hold the lower limit (4001 in this case).
- After a push operation, *SP* compared with the upper-limit register and after a pop operation, *SP* is a compared with the lower-limit register.
- The two microoperations needed for either the push or pop are

    (1) An access to memory through SP

    (2) Updating SP.

- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation:

- A stack organization is very effective for evaluating arithmetic expressions.
- The common arithmetic expressions are written in *infix notation,* with each operator written *between* the operands.
- Consider the simple arithmetic expression.

    A*B+C*D

- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- The Polish mathematician Lukasiewicz showed that arithmetic expression can be represented in *prefix notation.*
- This representation, often referred to as *Polish notation,* places the operator before the operands. So it is also called as *prefix notation*.
- The *Postfix notation,* referred to as *reverse Polish notation (RPN),* places the operator after the operands.
- The following examples demonstrate the three representations

    Eg: A+B -----> Infix notation

+AB ------> Prefix or Polish notation

AB+ -------> Post or reverse Polish notation

- The reverse Polish notation is in a form suitable for stack manipulation. The expression

  A*B+C*D

  Is written in reverse polish notation as

  AB* CD* +

  And it is evaluated as follows

  - ✓ Scan the expression from left to right.
  - ✓ When operator is reached, perform the operation with the two operands found on the left side of the operator.
  - ✓ Remove the two operands and the operator and replace them by the number obtained from the result of the operation.
  - ✓ Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.
- For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.
- The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.
- The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.
- The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.
- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Evaluation of Arithmetic Expressions:

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.
- This procedure is employed in some electronic calculators and also in some computer.
- The following numerical example may clarify this procedure. Consider the arithmetic expression

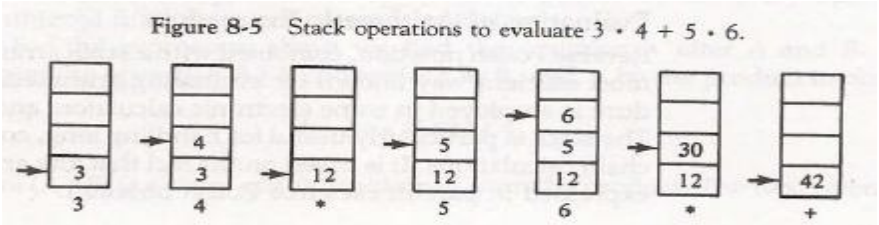
  (3*4) + (5*6)

  In reverse polish notation, it is expressed as

  34 * 56* +

- Now consider the stack operations shown in Fig. 8-5.

Figure 8-5 Stack operations to evaluate 3 · 4 + 5 · 6.



- Each box represents one stack operation and the arrow always points to the top of the stack.
- Scanning the expression from left to right, we encounter two operands.
- First the number 3 is pushed into the stack, then the number 4.
- The next symbol is the multiplication operator *.
- This causes a multiplication of the two top most items the stack.
- The stack is then popped and the product is placed on top of the stack, replacing the two original operands.
- Next we encounter the two operands 5 and 6, so they are pushed into the stack.
- The stack operation results from the next * replaces these two numbers by their product.
- The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

2. Instruction Formats:

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
    1. An operation code field that specifies the operation to be perform
    2. An address field that designates a memory address or a processor register.
    3. A mode field that specifies the way the operand or the effective address is determined.
- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruct format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPU organizations:
    1. Single accumulator organization.
    2. General register organization.
    3. Stack organization.

Single Accumulator Organization:

- ✓ In an accumulator type organization all the operations are performed with an implied accumulator register.
- ✓ The instruction format in this type of computer uses one address field.
- ✓ For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as
    - **ADD X**
- ✓ Where X is the address of the operand. The ADD instruction in this case results in the operation *AC* ←*AC +M[X]. AC* is the accumulator register and *M[X]* symbolizes the memory word located at address X.

General register organization:

- ✓ The instruction format in this type of computer needs three register address fields.
- ✓ Thus the instruction for an arithmetic addition may be written in an assembly language as

  ADD R1, R2, R3

  to denote the operation R1←R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- ✓ Thus the instruction **ADD R1, R2** would denote the operation R1 ← *R1 +* R2. Only register addresses for R1 and R2 need be specified in this instruction.
- ✓ General register-type computers employ two or three address fields in their instruction format.
- ✓ Each address field may specify a processor register or a memory word.
- ✓ An instruction symbolized by **ADD R1, X** would specify the operation R1← R1 + M[X].
- ✓ It has two address fields, one for register R1 and the other for the memory address X**.**

Stack organization:

- ✓ The stack-organized CPU has PUSH and POP instructions which require an address field.
- ✓ Thus the instruction **PUSH X** will push the word at address X to the top of the stack.
- ✓ The stack pointer is updated automatically.
- ✓ Operation-type instructions do not need an address field in stack-organized computers.
- ✓ This is because the operation is performed on the two items that are on top of the stack.
- ✓ The instruction **ADD** in a stack computer consists of an operation code only with no address field.
- ✓ This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- ✓ There is no need to specify operands with an address field since all operands are implied to be in the stack.

- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.

- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

  X= (A+B) * (C+D)

- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

Three Address Instructions:

- ✓ Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- ✓ The program assembly language that evaluates **X = (A+B) * *(C+*D**) is shown below, together with comments that explain the register transfer operation of each instruction.

```
ADD   R1, A, B    R1←M[A] + M[B]
ADD   R2, C, D    R2←M[C] + M[D]
MUL   X, R1, R2   M[X]←R1*R2
```

- ✓ The symbol M [A] denotes the operand at memory address symbolized by A.
- ✓ The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- ✓ The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instructions:

- ✓ Two-address instructions formats use each address field can specify either a processor register or memory word.
- ✓ The program to evaluate X = (A+B) * *(C+*D) is as follows

```
MOV   R1, A     R1←M[A]
ADD   R1, B     R1←R1 + M[B]
MOV   R2, C     R2←M[C]
ADD   R2, D     R2←R2 + M[D]
MUL   R1,R2     R1←R1*R2
MOV   X, R1     M[X]←R1
```

- ✓ The MOV instruction moves or transfers the operands to and from memory and processor registers.
- ✓ The first symbol listed in an instruction is assumed be both a source and the destination where the result of the operation transferred.

One Address Instructions:

- ✓ One-address instructions use an implied accumulator *(AC)* register for all data manipulation.
- ✓ For multiplication and division there is a need for a second register. But for the basic discussion we will neglect the second register and assume that the *AC* contains the result of all operations.
- ✓ The program to evaluate X=(A+B) * *(C+*D) is

```
LOAD    A    AC←M[A]
ADD     B    AC←AC + M[B]
STORE   T    M[T]←AC
LOAD    C    AC←M[C]
ADD     D    AC←AC + M[D]
MUL     T    AC←AC*M[T]
STORE   X    M[X]←AC
```

- ✓ All operations are done between the AC register and a memory operand.
- ✓ T is the address of a temporary memory location required for storing the intermediate result.

Zero Address Instructions:

- ✓ A stack-organized computer does not use an address field for the instructions ADD and MUL.
- ✓ The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- ✓ The following program shows how X = (A+B) * *(C+*D) will be written for a stack-organized computer. *(TOS* stands for top of stack).

```
PUSH    A    TOS ← A
PUSH    B    TOS ← B
ADD          TOS ← (A + B)
PUSH    C    TOS ← C
PUSH    D    TOS ← D
ADD          TOS ← (C + D)
MUL          TOS ← (C + D) * (A + B)
POP     X    M[X] ← TOS
```

- ✓ To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- ✓ The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

RISC Instructions:

- ✓ The instruction set of a typical RISC processor is use only load and store instructions for communicating between memory and CPU.
- ✓ All other instructions are executed within the registers of CPU without referring to memory.
- ✓ LOAD and STORE instructions that have one memory and one register address, and computational type instructions that have three addresses with all three specifying processor registers.
- ✓ The following is a program to evaluate X=(A+B)*(C+D)

```
LOAD    R1, A          R1 ← M[A]
LOAD    R2, B          R2 ← M[B]
LOAD    R3, C          R3 ← M[C]
LOAD    R4, D          R4 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
ADD     R3, R3, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1*R3
STORE   X, R1          M[X] ← R1
```

- ✓ The load instructions transfer the operands from memory to CPU register.
- ✓ The add and multiply operations are executed with data in the register without accessing memory.
- ✓ The result of the computations is then stored memory with a store in instruction.

3. Addressing Modes

- ➢ The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- ➢ Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  - o To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  - o To reduce the number of bits in the addressing field of the instruction
- ➢ Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are *implied* and *immediate* modes.

Implied Mode:

- ✓ In this mode the operands are specified implicitly in the definition of the instruction.
- ✓ For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- ✓ All register reference instructions that use an accumulator are implied mode instructions.
- ✓ Zero address in a stack organization computer is implied mode instructions.

Immediate Mode:

- ✓ In this mode the operand is specified in the instruction itself.
- ✓ In other words an immediate-mode instruction has an operand rather than an address field.
- ✓ Immediate-mode instructions are useful for initializing registers to a constant value.
- ➢ The address field of an instruction may specify either a memory word or a processor register.
- ➢ When the address specifies a processor register, the instruction is said to be in the register mode.

Register Mode:

- ✓ In this mode the operands are in registers that reside within the CPU.
- ✓ The particular register is selected from a register field in the instruction.

Register Indirect Mode:

- ✓ In this mode the instruction specifies a register in CPU whose contents give the address of the operand in memory.
- ✓ In other words, the selected register contains the address of the operand rather than the operand itself.
- ✓ The advantage of a register indirect mode instruction is that the address field of the instruction uses few bits to select a register than would have been required to specify a memory address directly.

Auto-increment or Auto-Decrement Mode:

- ✓ This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

- ➢ The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- ➢ Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- ➢ The basic two mode of addressing used in CPU are *direct* and *indirect* address mode.

Direct Address Mode:

- ✓ In this mode the effective address is equal to the address part of the instruction.
- ✓ The operand resides in memory and its address is given directly by the address field of the instruction.
- ✓ In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode:

- ✓ In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- ✓ Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

- ➢ A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- ➢ The effective address in these modes is obtained from the following computation:

Effective address =address part of instruction + content of CPU register

- ➢ The CPU register used in the computation may be the program counter, an index register, or a base register.
- ➢ We have a different addressing mode which is used for a different application.

Relative Address Mode:

- ✓ In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Indexed Addressing Mode:

- ✓ In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- ✓ An index register is a special CPU register that contains an index value.

Base Register Addressing Mode:

- ✓ In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- ✓ This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

Numerical Example:

- To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7.

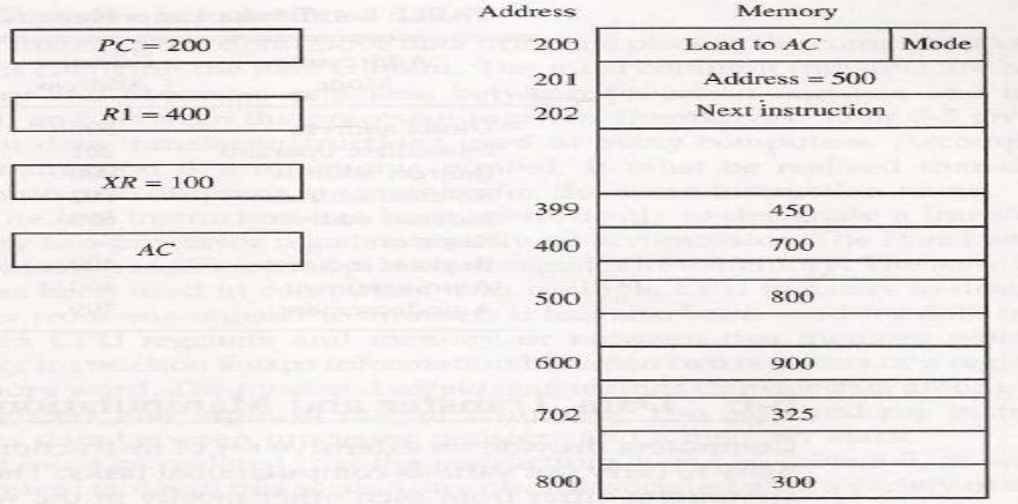


Figure 8-7 Numerical example for addressing modes.

- The two-word instruction at address 200 and 201 is a "load to *AC*" instruction with an address field equal to 500.
- The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.
- *PC* has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is 100.
- AC receives the operand after the instruction is executed.
- In the **direct address mode** the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is 500.
- In the **immediate mode** the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC.
- In the **indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the **relative mode** the effective address is 500 + 202 =702 and the operand is 325. (the value in PC after the fetch phase and during the execute phase is 202.)
- In the **index mode** the effective address is XR+ 500 = 100 + 500 = 600 and the operand is 900.
- In the **register mode** the operand is in R1 and 400 is loaded into *AC.*
- In the **register indirect mode** the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The **auto-increment mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The **auto-decrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.
- Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

4. Data Transfer and Manipulation:

- Most computer instructions can be classified into three categories:

1. *Data transfer instructions*
2. *Data manipulation instructions*
3. *Program control instructions*

Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- Table 8-5 gives a list of eight data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- The ***load*** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The ***store*** instruction designates a transfer from a processor register into memory.
- The ***move*** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- The ***exchange*** instruction swaps information between two registers or a register and a memory word.
- The ***input*** and ***output*** instructions transfer data among processor registers and input or output terminals.
- The ***push*** and ***pop*** instructions transfer data between processor registers and a memory stack.
- Different computers use different mnemonics symbols for differentiate the addressing modes.
- As an example, consider the *load to accumulator* instruction when used with eight different addressing modes.
- Table 8-6 shows the recommended assembly language convention and actual transfer accomplished in each case

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

- *ADR* stands for an address.
- *NBA* a number or operand.
- X is an index register.
- The @ character symbolizes an indirect addressing.
- The $ character before an address makes the address relative to the program counter *PC.*
- The # character precedes the operand in an immediate-mode instruction.
- R1 is a processor register.
- AC is the accumulator register.

- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolic address.
- The register mode is symbolized by giving the name of a processor register.
- In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses.
- The auto-increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The auto-decrement mode would use a minus instead.

Data Manipulation Instructions:

- Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.
- The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

1. Arithmetic instructions

- ✓ The four basic arithmetic operations are addition, subtraction, multiplication and division.
- ✓ Most computers provide instructions for all four operations.
- ✓ Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- ✓ A list of typical arithmetic instructions is given in Table 8-7.

TABLE 8-7 Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

- ✓ The increment instruction adds 1 to the value stored in a register or memory word.
- ✓ A number with all 1's, when incremented, produces a number with all 0's.
- ✓ The decrement instruction subtracts 1 from a value stored in a register or memory word.
- ✓ A number with all 0's, when decremented, produces number with all 1's.
- ✓ The add, subtract, multiply, and divide instructions may be use different types of data.
- ✓ The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code.
- ✓ An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.
- ✓ The mnemonics for three add instructions that specify different data types are shown below.

  ADDI Add two binary integer numbers
  ADDF Add two floating-point numbers
  ADDD Add two decimal numbers in BCD

- ✓ A special carry flip-flop is used to store the carry from an operation.
- ✓ The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- ✓ Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation.
- ✓ The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

2. Logical and bit manipulation instructions

- ✓ Logical instructions perform binary operations on strings of bits store, registers.
- ✓ They are useful for manipulating individual bits or a group of that represent binary-coded information.
- ✓ The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- ✓ By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- ✓ Some typical logical and bit manipulation instructions are listed in Table 8-8.

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

- ✓ The clear instruction causes the specified operand to be replaced by 0's.
- ✓ The complement instruction produces the 1's complement by inverting all bits of the operand.
- ✓ The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- ✓ The logical instructions can also be used to performing bit manipulation operations.
- ✓ There are three bit manipulation operations possible: a selected bit can cleared to 0, or can be set to 1, or can be complemented.
    - The AND instruction is used to clear a bit or a selected group of bits of an operand.
    - The OR instruction is used to set a bit or a selected group of bits of an operand.
    - Similarly, the XOR instruction is used to selectively complement bits of an operand.
- ✓ Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, or complemented.
- ✓ Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

3. Shift Instructions:

- ✓ Shifts are operations in which the bits of a word are moved to the left or right.
- ✓ The bit shifted in at the end of the word determines the type of shift used.
- ✓ Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations.
- ✓ In either case the shift may be to the right or to the left.
- ✓ Table 8-9 lists four types of shift instructions.

TABLE 8-9 Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

- ✓ The logical shift inset to the end bit position.
- ✓ The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- ✓ Arithmetic shifts usually conform to the rules for signed-2's complement numbers.
- ✓ The arithmetic shift-right instruction must preserve the sign bit in the leftmost position.
- ✓ The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- ✓ This is a shift-right operation with the end bit remaining the same.
- ✓ The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.

- ✓ The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.
- ✓ The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- ✓ Thus a rotate-left through *carry* instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shift the entire register to the left.

5. Program Control:

- Program control instructions specify conditions for altering the content of the program counter.
- The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
- This instruction provides control over the flow of program execution and a capability for branching to different program segments.
- Some typical program control instructions are listed in Table 8.10.

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

- Branch and jump instructions may be conditional or unconditional.
- An unconditional branch instruction causes a branch to the specified address without any conditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- The skip instruction does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing program counter.
- The call and return instructions are used in conjunction with subroutines.
- The compare instruction forms a subtraction between two operands, but the result of the operation not retained. However, certain status bit conditions are set as a result of operation.
- Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.

Status Bit Conditions:

- The ALU circuit in the CPU have status register for storing the status bit conditions.
- Status bits are also called *condition-code* bits or *flag* bits.
- Figure 8-8 shows block diagram of an 8-bit ALU with a 4-bit status register.

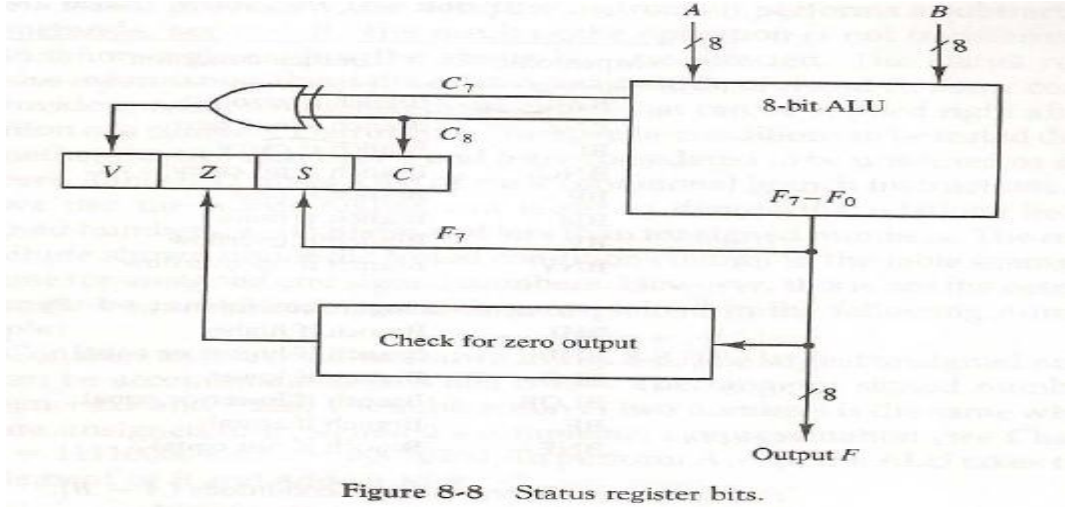


Figure 8-8 Status register bits.

- The four status bits are symbolized by C, S, *Z,* and V. The bits are set or cleared as a result of an operation performed in the ALU.
    - Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
    - S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
    - Bit *Z* (zero) is set to 1 if the output of the ALU contains all 0's. It is clear to 0 otherwise. In other words, *Z* = 1 if the output is zero and *Z =0 if the output is not zero.*
    - Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries equal to 1, and cleared to 0 otherwise.
- The above status bits are used in conditional jump and branch instructions.

Subroutine Call and Return:

- A subroutine is self contained sequence of instructions that performs a given computational task.
- The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save return address.
- A subroutine is executed by performing two operations
  (1) The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return
  (2) Control is transferred to the beginning of the subroutine.
- The last instruction of every subroutine, commonly called *return from subroutine,* transfers the return address from the temporary location in the program counter.
- Different computers use a different temporary location for storing the return address.
- The most efficient way is to store the return address in a memory stack.
- The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack.
- A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

- The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

Program Interrupt:

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request.
- The interrupt procedure is similar to a subroutine call except for three variations:
    - The interrupt is initiated by an internal or external signal.
    - Address of the interrupt service program is determined by the hardware.
    - An interrupt procedure usually stores all the information rather than storing only PC content.

Types of interrupts:

- ✓ There are three major types of interrupts that cause a break in the normal execution of a program.
- ✓ They can be classified as
    - **External interrupts:**
        - These come from input—output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

- Ex: I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure.

- **Internal interrupts:**
  - These arise from illegal or erroneous use of an instruction or data.
  - Internal interrupts are also called *traps.*
  - Ex: interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

✓ Internal and external interrupts are initiated form signals that occur in hardware of CPU.

- **Software interrupts**
  - A software interrupt is initiated by executing an instruction.
  - Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.

6. Reduced Instruction Set Computer:

➢ A computer with large number instructions is classified as a *complex instruction set computer,* abbreviated as CISC.

➢ The computer which having the fewer instructions is classified as a *reduced instruction set computer,* abbreviated as RISC.

CISC Characteristics:

✓ A large number of instructions--typically from 100 to 250 instructions.
✓ Some instructions that perform specialized tasks and are used infrequently.
✓ A large variety of addressing modes—typically from 5 to 20 differ modes.
✓ Variable-length instruction formats
✓ Instructions that manipulate operands in memory

RISC Characteristics:

✓ Relatively few instructions
✓ Relatively few addressing modes
✓ Memory access limited to load and store instructions
✓ All operations done within the registers of the CPU
✓ Fixed-length, easily decoded instruction format
✓ Single-cycle instruction execution
✓ Hardwired rather than microprogrammed control
✓ A relatively large number of registers in the processor unit
✓ Efficient instruction pipeline