5.1.4 Accumulator

A **Processor Register** (AC) is required for doing operations on data. This register holds data on which addition, subtraction, shift and logical operations are to be carried out. The result of arithmetic and logical operations is automatically stored in the **Accumulator**.

Thus, it is used for storing result and for doing operations (arithmetic as well as logical) on its content.

Program Counter : It deals with the order of the execution of instructions. It holds the address of the next instruction to be executed. Thus, it acts as a pointer which points to the memory location where the next instruction is stored.

Temporary Register : A register used for holding temporary data generated during processing. It is basically made for **processor.**

Instruction Register : A register used for storing instruction is called **Instruction Register**. The instruction read from the memory is to be places in some register known as Instruction Register.

Data Register : Register used to hold the data (operand) read from **memory**.

Address Register : Register used to hold the address of memory word.

INPR: Input register will hold, receives data from an input device.

OUTR: It holds data that need to be sent to output devices.

Table below describe the brief function and the number of bits that different registers contains.

List of Registers

Register Symbol	No. of bits	Register Name	Function of Register
GPR	8	**General Purpose Register**	Holds data for processing, execution
AC	16	**Accumulator**	Processor Register
PC	12	**Program Counter**	Holds address of next Instruction
TR	16	**Temporary Register**	Holds Temporary Data
IR	16	**Instruction Register**	Holds Instruction Code
DS	16	**Data Register**	Holds Memory Operand
AR	12	**Address Register**	Holds address of Memory
INPR	8	**Input Register**	Holds Input Character
OUTR	8	**Output Register**	Holds output Character

Table 5.1

5.1.5 Flags

There are number of indicators known as **flags** that show the processor's status. Most of these flags represent the results of the last operations. For example, the addition of two numbers might produce a negative sign, an overflow, a carry, or a value of zero.

These flags are represented by a single bit such as if the result of an addition is negative, the sign flag would set to 1. If the result was not a negative number (zero or greater than zero), the sign **flag** would equal to 0.

5.1.6 Stacks

During the execution of operation, there are number of times when the Processor needs to use a **temporary memory** to store different data values so that they can be used again when required. Every processor has a finite number of registers. But if an application needs more registers than available, the register value that is not needed immediately by **Processor** can be stored in the temporary memory. Also, when a processor needs to jump to a **subroutine** or **function**, it needs to remember the instruction from where it jumped so that it can return back to the same place when the subroutine is completed. Hence, the return address must be stored and this return address is generally stored in this temporary memory. This temporary memory is known as **Stack.**

The **Stack** is a block of memory locations reserved to functions as temporary memory. When a processor puts a piece of data, on the top of stack, the data below it cannot be removed until the data above it is removed. This type of **memory location** is referred as LAST-IN-FIRST-OUT or **LIFO**.

The two main operations that the **processor** can perform on the stack are **PUSH** and **POP**. It can either store the value of a register to the top of the stack or can remove the top element from the stack. Storing the data to the stack is referred to as pushing and removing the top data from stack is referred as poping.

5.1.7 I/O Ports

Input/Output ports, referred as **I/O ports**, are any connection hat exist between the processor and its external devices. For example, a **USB printer** can be connected to the computer system through an I/O (USB) port. Using this port the computer can issue commands and send to be printed.

5.2 GENERAL REGISTER ORGANIZATION

A bus organization for seven CPU registers as shown in below figure. The output of each registers is connected to two **multiplexers(MUX)** to form the two buses A & B. the selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common **ALU**. The operation selected in the ALU determines the arithmetic or logic **microinstructions** to be performed. The result of the **micro operation** is available for the output data and also goes into the inputs of seven registers. the register that receives the information from the output bus is selected by the **decoder.** The decoder activates one of the register load inputs and thus providing a transfer path between the output data bus and the inputs of the selected destination register.

Let the operation be **R1 ← R2 + R3**

To perform this operation, the control must provide

SELA >> Place the contents of R2 into bus A.

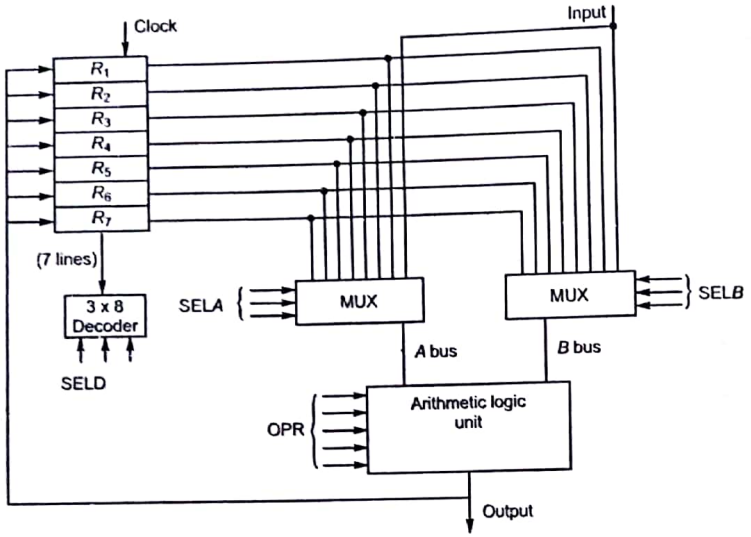
SELB >> Place the contents of R3 into bus B

ALU >> Operation selector

OPR >> Provide the arithmetic addition A+B

SELD >> Transfer the contents of the output bus into R1.

At the beginning of the **clock cycle**, the four control selection variables generated R2 and R3 must be available in the **control unit**. Two source registers propagate through **multiplexers** and



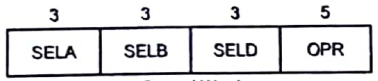
General Register Organization
Fig. 5.2

the ALU, to the outputs bus, and to the input of destination **Register**, during the **clock cycle** interval. At the next clock transition, the information from output bus is transferred to the destination register R1.

5.2.1 Control Word

The group of binary assigned to perform a specified operation is known as control word.

There are 14 binary selection inputs in the units, and their combined value specified a control word. It contains of four field as shown in Fig. 5.3.



Control Word
Fig 5.3

Three fields contain three bits each, one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a fegister for the B input of the ALU. The three bits of SELD select a destination register using the **decoder** and its seven load outputs. The five bits of **OPR** select onr of the operations in the **ALU**.

The 14-bit **control word** when applied to the selection inputs specify a particular **Microoperations**. The encoding of register selections is specified in following Table 5.2.

Binary code	SELA	SELB	SELD
000	input	input	none
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Encoding of Register Selection Fields
Table 5.2

When the 3-bit binary code for SELA or SELB is 000, the respective **Multiplexer** selects the external input data as shown in Figure, when the 3-bit binary code for SELD = 000, no destination register is selected and the content of output bus is for external output.

The OPR field has five bits. The encoding for five bit OPR field is specified in the Table 5.3.

OPR	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Addition	ADD
00101	Subtract	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

Encoding of ALU Operation
Table 5.3

Let the microoperation given by the statement is

R1 ← R4 ∧ R5

This statement specifies R4 for the A input of ALU, R5 for the B input of ALU, and R1 as the destination **Register**. The **microoperation** to be performed is **AND operation** between R4 and R5. The **control word** for the above statement according to the Table 5.2 and Table 5.4 is as follow:

SELA	SELB	SELD	OPR
R1	R4	R5	AND
001	100	101	01000

Table 5.4

Thus, the **control word** is **001 100 101 01000.**

5.3 STACK ORGANIZATION

The **Stack** is also known as last-in first-out list. The stack can be consider as a storage method in which the item that stored last is the first item to be removed. The most common example of the stack phenomenon, is a pile of trays in a cafeteria. A tray which is placed last on the top of pile is the first to be taken off.

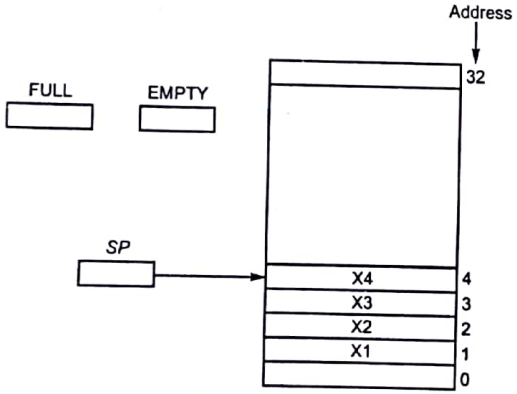
The stack in a digital computer is a part of memory unit. Also, with the stack an **address register** is associated that holds the address of the last element stored in the stack. This address register is known as **Stack Pointer** (SP). Thus, the stack pointer always points to the top most element of the stack.

5.3.1 Push and Pop Operation

Insertion and deletion of items are the operations related with the stack. The process of inserting an item into the stack is known as push operation. The process of deleting an item from the stack is known a pop operation. These operation are done by incrementing or decrementing the **Stack Pointer** (SP).

5.3.2 Register Stack

A stack can be organized by a finite number of registers or a stack can be a finite number of **Memory Words**. The stack pointer contains the address of the word that is currently on the top of the stack. A **32-Word Register Stack** is shown in Fig. The **Stack Pointer** contains a binary value. Currently, there are four items XI, X2, X3 and X4 are placed in the stack with X4 at the top of stack so the content of stack pointer is 4. The items are removed from the stack by using **POP instruction.** When we remove the top item X4 from the stack, X3 is now on top of stack and the content of **SP** is decrement so that the SP holds the address 3. To insert a new item, first the SP will incremented and then the item is inserted so that SP points to the top of the stack.



Register Stacl
Fig. 5.4

In a **32-Word Register Stack**, the address of each location will be of five bits since 25 = 32. *Thus; stack pointer will be of five bits and cannot exceed the value* 11111. Thus, when the SP pointer

content is 11111, the one-bit register FULL is set to 1, indicating that the stack is full and there is no location empty for any more item. Similarly when the content of SP = 00000 another one-bit register EMPTY is set to 1 indicating that the stack is empty and there is no element in the stack that can be deleted from the stack. The data register DR holds the items that is to written into the stack or read out of the stack.

Initially, the SP is cleared to 0 so the **stack pointer** points to the word at address 0. Also, the one-bit register FULL is cleared to 0, indicating that the stack is not full and the register EMPTY is set to 1. A new item is inserted into the stack by push operation. The **PUSH** operation will be the set of following **microoperations** :

	SP ← SP +1	Increment stack poiner
	M[SP] ← DR	Add item on the top of stack
If (SP+0) then	(FULL ← 1)	Check if stack is full
	EMPTY ← 0	Mark the stack not empty

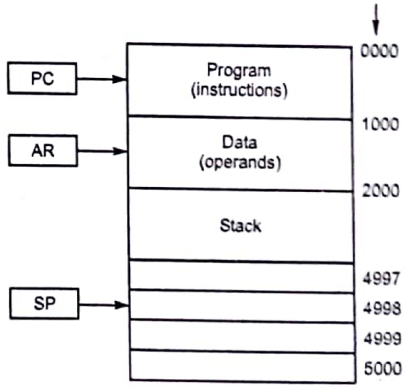
If the stack is not empty, an item can be deleted from the stack using the POP operation. The POP operation is implemented by the following set of microoperations.

	DR ← M [SP]	Read item from the top of stack
	SP ← SP – 1	Decrement stack pointer
If (SP=0) then	(EMPTY ← 1)	Check if stack is empty
	FULL ← 0	Mark the stack not full.

The top item is read from the stack into DR, then the SP is decremented by 1 so that it points to top of stack. The SP is checked whether it is zero on not. If zero, EMPTY sets to 1 indicating that the stack is empty.

5.3.3 Memory Stack

A stack can also be implemented using **Random-Access Memory** attached to the **CPU**. This can be implemented by assigning a portion of memory for stack operation using the processor register as a **Stack Pointer**. The computer **memory** is partitioned into three parts as program,



Memory Stack
Fig. 5.5

data, and stack as shown in Fig. PC points to the address of the next instruction stored in memory. The stack pointer (SP) points to the top of the stack.

The initial value of SP is 5000 and the first item stored in stack is at address 4999, the second item at address 4998 and so on. The last address that can be used for stack is 2000 i.e. the final value of stack is 2000. The stack grows in reverse order with decreasing addresses. A new item into the stack is inserted using PUSH operations as:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

The **stack pointer** is decremented first so that it points to the next address of the stack and then the item from the data register is inserted into the top of the stack. An item can be deleted from the stack using POP operation as

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

The top item of the stack is read into the **data register DR** and then the stack pointer is incremented by 1 so that it points to the top item of the **stack.**

Most of the computer does not provide any method to check the stack overflow or underflow to check whether the stack is full or empty. One possible method is to used two processor registers holding the addresses 2000 (upper limit) and 5000 (lower limit) respectively. Then, stack pointer is compared every time the **push operation** takes place with the upper-limit register and its with the lower-limit register, after the pop operation takes place.

5.3.4 Reverse Polish Notation

Let us consider an expression x + y. The plus operator is placed in between the two operands x and y. Such a notation is known as **infix notation.** If the operator is placed before the two operands as +xy, the notation is said to be **prefix notation**, also known as polish notation. If the operator is placed after the two operands as xy+, the notation is said to be postfix notation, also known as **Reverse Polish Notation**. Thus, the three notation are

x + y **Infix Notation**
+ xy **Prefix or Polish Notation**
xy + **Postfix or Reverse Polish Notation.**

For Stack manipulation the reverse polish notation is best suited. The **reverse polish notation** for the expression A * B + C * D is AB * CD * +.

5.3.5 Conversion to Reverse Polish Notation

The conversion of an expression from **infix notation** to the reverse polish notation must be done according to the operational hierarchy that follows for infix notation. First perform all arithmetic operations inside the inner parentheses, then inside outer parantheses, then do multiplication and division operations and lastly the addition and subtraction operations.

Example: Convert the infix expression (Y + y) * [z * (w+ v) + s] into reverse polish notation.

Solution. The two sub-expression (x + y) and (w + v) will solved first. Thus the Postfix expression of these subexpression will be xy+ and wv+ respectively.

*Now, in the square bracket z will be multiplied by (w + v). Thus, the postfix of this multiplication is zwv + *.*

This multiplication result is then added to s will result in zwv + *s +

Finally, xy+ and zwv + *s + will be multiplied together to get

xy + zwv + *s + *

\The procedure is shown again as

$$(x + y) * [z * (w + v) + s] = xy + * [z * wv + + s]$$
$$= xy + * [zwv + * + s]$$
$$= xy + * zwv + *s +$$
$$= xy + zwv + *s + *$$

Example: Convert the infix notation A*B + A*{B*D+C*E) into RPN.

Solution.

$$A*B+A*(B*D + C*E)$$
$$= AB* + A* (BD* + CE*)$$
$$= AB* + A * BD * CE * +$$
$$= AB * + ABD *CE* + *$$
$$= AB* ABD *CE* + * +$$

5.3.6 Evaluation of Arithmetic Expression

Consider an expression A * B + C * D in **infix notation**. Its reverse polish notation is AB * CD * +. This postfix expression will be evaluated as follows : scan the expression from left to right. Whenever an operator is found, perform the operation with the two operands on the left side of operator. Remove the operator and the two operands and replaced them by the result obtained by performing that operation. Continue in the same manner and repeat the procedure for every operator found until there are no more operators.

Thus, for the **Reverse Polish Notation** AB * CD* + first we find the operator * and the two operands to the left of * are A and B. Thus, we perform A * B and replace A, B and * by the product we get

(A * B) CD* +

The next **operator** is * and the two operands to the left of * are C and D. Thus, we perform C * D and replace C, D and * by the product, we get

(A *B)(C*D) +

The next operator is + and the two operands to the left + are the two products (A * B) and (C * D), hence the result obtained is

A*B + C*D.

Any arithmetic expression can be evaluated using **stack** as follows :

(i) Convert the given infix expression into its equivalent reverse polish notation.

(ii) Scan the expression from left to right.

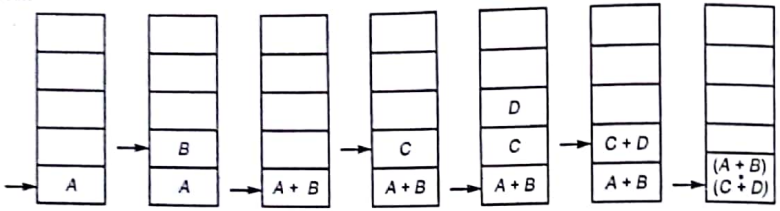
(iii) While scanning when operands are found, pushed them into the stack as they appears.

(iv) When operators are found, pop two top most operands from the stack, perform the operation involving the operator and then pushed back the result into the stack.

(v) Continue, scanning the expression until there are no more operators.

(vi) Finally, the result of the expression will remain on the top of the stack.

To illustrate this, consider the expression (A + B) * (C + D). In reverse polish notation this expression is AB + CD + *. The stack operation is shown in Fig. 4.6. The arrow ( ) points to the top of the **stack**.



Stack Operation to Evaluate (A+B) and (C+D)

Fig. 5.6

5.4 INSTRUCTION FORMATS

The most common fields found in the instruction are ;

(i) An operation code field that specifies the operation to be performed. It is known as opcode field.

(ii) An address field that designates the register address and/or a memory address.

(iii) A mode field that specifies the way the **operands** or the effective address is determined.

For example,

ADD Rl, RO. ADD is the opcode and R1, R0 are the address field.

Operations specified by computer instructions are executed on some data stored in **Memory** or some **Registers**. Operands residing on memory are specified by memory address and operands residing on **Processor Registers** are specified by register address. A register address is a binary number of K-bits that defines one of 2K registers in the **CPU**. Thus, if a CPU has processor registers R0 to R15, then address of each registers will be of four bits. For example, the binary information 0101, is the address of register R5.

The instruction may be of several different lengths containing different number of addresses. The number of address fields in the instruction format of a computer system depends on the internal architecture/organization of registers. The different types of CPU organization are :

(i) **Single Accumulator Organization**

(ii) **General Register Organization**

(iii) **Stack Organization**

5.4.1 Accumulator-type Organization

All operations are performed with an implied accumulator register. The instruction format uses one **address field**, i.e. only one operand address is specified in the **instruction**. The other operand is in the accumulator. The result is placed in the **accumulator**.

For example,

ADD X, AC ← AC + M[X]

The ADD X instruction means add the content at memory location X, symbolizes as M[X], with

the content of accumulator. Thus, the previous value of **accumulator** will be lost and the accumulator contain the result of above instruction.

5.4.2 General-Register Organization

The **instruction format** in this type of computer needs two or three addresses. The number of addresses in the instruction can be reduced to two from three if the destination register is same as one of the source registers.

In two address instructions both operand address are specified. The result is placed in one of the specified addresses. In **three-address instruction** two addresses are specified for the two operands and one address of the result. Thus, general-register-type computers employ two or three address fields in the instruction format. Each address field may specify a **processor register** or a **memory location.**

Examples :

ADD Rl, R2, R3 Rl ← R2 + R3

The above instruction contains three register addresses. The operation performed is the add operation between the content of processor register R2 and R3 and result is to be placed into the destination register Rl.

ADD Rl, R2 Rl ← Rl + R2

The above instruction consists of only two register addresses. Rl and R2 are source registers where Rl also serves the purpose of destination register. The instruction specifies the add operation between the contents of Rl and R2 and result to be stored into Rl.

MOV Rl, R2 Rl ←R2

Mnemonic MOV is used for transfer instruction. The instruction contains only two register address Rl and R2 where R2 is the source register and Rl is the destination. Thus, in transfer-type instruction only two addresses are required. The instruction specifies move the content of R2 into register Rl.

Add Rl, X Rl ← Rl + M[X]

This instruction has two address field, Rl the register address and X a memory address.

5.4.3 Stack Organization

Stack-oriented machines do-not contain any accumulator or **general-purpose registers**. Computers with **stack organization** have **PUSH** and **POP** instructions which requires an address field. Thus the instruction

PUSH X TOP ← M[X]

will push the word /data at address X to the top of the stack. The SP is automatically updated. The operation instruction does not contain any address field because the operation is performed on two top most operands of the stack.

For example,

ADD

The instruction **ADD** consist of only operation code with no address field. This instruction pops the top two operands from the stack, add the numbers and then **PUSH** the result into the stack.

5.4.4 Address Instruction Set

To show how the number of address affects a computer program, we will evaluate the arithmetic statement

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

using three, two, one or zero **address instruction.**

ADD, SUB, DIV and MUL are used for arithmetic operations, MOV for the transfer-operation. LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C and D and the result must be stored in memory address X. RI, R2 are the register and T is the address of temporary memory location used to store intermediate result.

5.4.4.1 Three-Address Instruction

ADD RI, A, B	RI	←	M[A] + M[B]
ADD R2, C, D	R2	←	M[C] + M[D]
MUL X, RI, R2	X	←	RI * 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.

5.4.4.2 Two Address Instruction

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 * R*
MOV X, R1	M[X]	←	R1

5.4.4.3 One-Address Instruction

One address instruction use an accumulator (AC) register for all data manipulation.

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 * ACT.
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 intermediate result.

5.4.4.4 Zero-Address Instruction

To evaluate arithmetic expression for zero-address machine, the expression must be in reverse-polish notation. Also, the instructions like **ADD, MUL** does not requires an operand field. It simply pop-up the two top most operands from the stack, perform the operation and places the result on the top of the stack. However, PUSH and POP instructions requires an address field to specify the

operand that communicates with the stack, TOS stands for top of stack. The **reverse polish notation** of expression.

X = (A + B) * (C + D) is evaluated as
= (AB+) * (CD+)
= AB + CD + *

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	←	(A+B) * (C+D)
POP X	M[X]	←	TOS

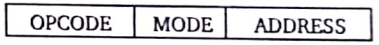
5.5 ADDRESSING MODES

Each instruction needs data on which it has to perform the specified operation. The operand (data) may be in **accumulator**, general purpose register or at some specified memory location. Thus, there are various ways of specifying the address of the data, known as **addressing modes**.

5.5.1 Instruction cycle

1. Fetch the instruction from the memory.
2. Decode the instruction
3. Execute the instruction.

PC *i.e.* **program counter** keeps track of the instructions in the program stored in the memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding determines the operation to be performed, the addressing mode of the instruction and the location of the operands. The computer then executes the **instruction** and returns to the step 1 to fetch the next instruction in sequence.



Instruction Format
Fig. 5.7

There are **two addressing modes** that need no address field at all. They are implied and immediate. **Zero-Address instructions** in a stack-organized computer are implied-mode instructions since the **operands** are implied to be on top of the stack.

5.5.2 Implied Addressing Mode

Also known as implicit or inherent **addressing mode.** The operands are specified implicitly in the definition of the instruction itself.

For example,

"Complement Accumulator" is an **implied mode** instruction because the operand in the accumulator register is implied in the definition of the instruction.

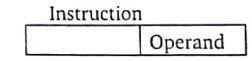
CMA : Take complement of the content of AC,

RLC : Rotate the contents of the Accumulator.

All reference instruction that use an accumulator are **implied-mode instructions**.

5.5.3 Immediate Addressing Mode

In this mode the operands is specified in the instruction itself, i.e. in **immediate addressing mode**, instruction has an operand field rather than address field. The operand field contains the actual operand. This mode are useful for initializing registers to a constant value.



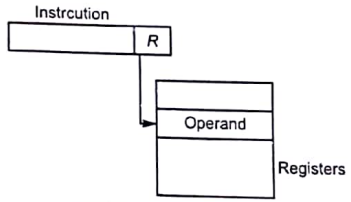
Immediate Addressing Mode
Fig. 5.8

For Exmaple,

MVI 06	Move 06 to the accumulator
ADD 05	Add 05 to the content of the AC.

5.5.4 Register Addressing Mode

In **Register addressing mode** the operands are in registers that resides within the CPU, the contents of the register is the operand itself.



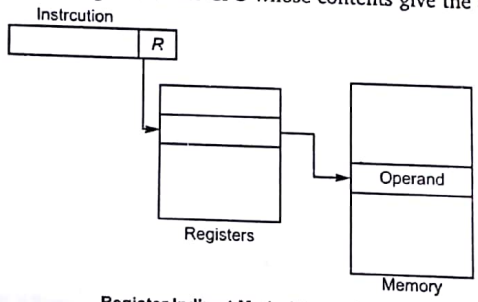
Register Addressing Mode
Fig. 5.9

Example,

MOV R1, R2, R1 ← R2	Transfer the content of register R2 to that of register R1
ADD R1 AC ← AC +R1	Add the content of register R1 to that of accumulator
LD R1 AC ← R1 LOAD	the content of register R1 to the accumulator

5.5.5 Register Indirect Mode Addressing

The instruction specifies a register in the CPU whose contents give the address of the memory



Register Indirect Mode Addressing
Fig. 5.10

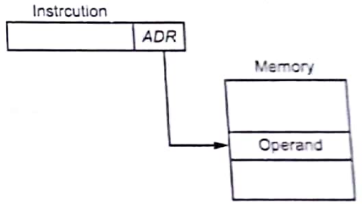
location where the operand is stored, i.e. the selected register contains the address of the operand rather than the operand itself.

Example LD (R1) AC ← M[R1]

5.5.6 Direct Addressing Mode

Also known as **absolute addressing Mode**. In this mode the address of data (i.e. operand) is specified in the instruction itself, i.e. the operand resides in the memory and its address are given directly by the address field of the instruction.

Example LD ADR AC ← M[ADR]

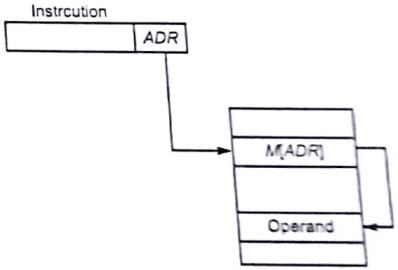


Direct Addressing Mode
Fig. 5.11

5.5.7 Indirect Addressing Mode

In this mode address field of the instruction gives the address where the operand is stored in the memory.

LD ADR AC ← M[M] [ADR]



Indirect Addressing Mode
Fig. 5.12

5.5.8 Displacement Addressing Mode

Displacement addressing mode are of three types :

(i) **Relative Addressing Mode.**

(ii) **Indexed Addressing Mode**

(iii) **Base Register Addressing Mode**

These addressing modes require that the **address field** of the **instruction** be added to the content of a specific register in the CPU to get the effective address. The effective address is calculated as

effective address = address part of instruction + content of the CPU register

The CPU register used in the computation of effective address may be program counter, an index register or a base register.

5.5.9 Relative Addressing Mode

The content of the program counter is added to the address part of the instruction in order to obtain the **effective address**.

For example, let the program counter contains 825 and the address part of the instruction contains the number 24. The instruction at memory location 825 is read from memory during fetch phase and the program counter is incremented by one to 826. Hence, the **effective address** computation for the relative address mode is 826 + 24 = 850.

5.5.10 Index Address Mode

The content of the **index register** is added to the address part of the instruction in order to obtain the effective address.

5.5.11 Base Register Addressing Mode

The content of the base register is added to the address part of the instruction in order to obtain the effective address.

5.6 DATA TRANSFER AND MANIPULATION

Computer instructions are broadly classified into three different categories
(i) Data transfer instructions
(ii) Data manipulation instructions
(iii) Program control instructions

5.6.1 Data Transfer Instructions

Data transfer instructions are those instructions that transfers the data from one location to another without changing the data content. These transfers can be between the two processor registers or between the memory location and processor registers or between the processor registers and input or output. Different **data transfer instructions** (with their **mnemonic**) used in many computers are listed in Table 5.5.

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

Data Transfer Instructions
Table 5.5

Different computer can use different **mnemonic** for the same instruction. The load instruction is used to transfer a data content from memory location to processor register, called an accumulator. The store instruction is used to transfer a data content from processor register to some memory location. Move instruction is used to transfer the data content from one processor register to another. Whenever it is required to swap information between two registers or a register and a memory location, the exchange instruction is used. To transfer data content from processor registers and input or output terminals, respectively the input and output instructions are used. Push and pop instructions are used to transfer data content between processor registers and a memory stack-Different **addressing mode** for the load instruction is shown in Table 5.6, where ADR is the memory address, NBR is the number or operand, X is index register, Rl is the **processor register** and AC is **accumulator**.

Addressing Mode for Load Instruction

Addressing Mode	Instruction	Register Transfer
Direct	LD ADR	AC ← M[ADR]
Indirect	LD @ ADR	AC ← M [M[ADR]]
Register	LD R1	AC ← R1
Register Indirect	LD (R1)	AC ← M[R1]
Relative	LDS ADR	AC ← M[PC + ADR]
Index	LD ADR (X)	AC ← M [ADR + XR]
Immediate	LD # NBR	AC ← NBR

Table 5.6

The **character** @ before memory address indicates indirect address. In case of **register indirect mode**, the register that holds the **memory address** is enclosed in parentheses.' The **character $** before memory address makes the address relative to the program counter PC. The **character #** before the operand indicates **immediate mode instruction**.

5.6.2 Data Manipulation Instructions

Data manipulation instructions are those instructions that perform arithmetic, shift or logic operations to manipulate the data. Thus, **data manipulation instructions** are broadly divided into three basic categories :

- **Arithmetic instructions**
- **Shift instructions**
- **Logic instructions**

5.6.2.1 Arithmetic Instruction

Addition, subtraction, multiplication and division are the four basic **arithmetic operations**. Most of the computers provide the instructions to perform these operations. Increment (or decrement) instructions adds 1 (or subtracts 1) to the value stored in a register or some **memory word**. A list of standard arithmetic instructions is shown in Table 5.7.

Arithmetic Instructions

Name of Instruction	Mnemonic
Add	ADD
Substract	SUB
Multiply	MUL
Divide	DIV
Increment	INC
Decrement	DEC
Negate (2's complement)	NEC
ADD with carry	ADD C
Substract with Borrow	SUB B

Table 5.7

5.6.2.2 Shift Instruction

Shift operations can be a circular or **arithmetic shift** or can be simple **logical shift** in which the bits of a word moved to the left or to the right. For both the cases, logical shift-left or logical shift right, 0 is inserted at the end bit position. **Rotate instructions** produces **circular shift**. It circulates the bits around the two end without loss of information. The instructions like rotate through carry treats a carry bit as an extention of the **Register** whose word is being rotated. Thus, rotate-left through carry instructions transfers the carry bit into the rightmost bit position and transfers the leftmost bit into the carry and at the same time shifts the entire register to left. The arithmetic shift-left instruction inserts 0 at the rightmost bit position. The arithmetic shift right instructions leaves the sign bit unchanged and shifts the bit (including the sign bit) to the right, and the rightmost bit is lost. Basic shift instructions are listed in Table 5.8.

Name of Instruction	Mnemonic
Logical shift left	SHL
Logical shift right	SHR
Rotate left	ROL
Rotate right	ROR
Arithmetic shift left	SHLA
Arithmetic shift right	SHR A
Rotate left through Carry	ROLC
Rotate right through Carry	RORC

Table 5.8

5.6.2.3 Logical Instruction

Different **logical instructions** are listed in the Table 5.9. **AND, OR** and **XOR** instructions provides the corresponding logical operations. The complement instruction produces the 1's complement of the operand. Clear instructions replaces all bits of the operand by O's. Clear carry, set carry and complement carry are instructions that are performed on the individual bits. If the instruction is clear carry the **carry bit** is cleared to 0. If it is set carry the carry bit sets to 1. Similarly, if the instruction is complement carry the carry bit complements and the bit changes from 0 to 1 or from 1 to 0.

Logical Instructions

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

Table 5.9

5.7 PROGRAM CONTROL INSTRUCTIONS

Program Control Instructions are those instructions that can alter the flow of control. Generally, the programs executed in a straight line, with one instruction sequentially following the another. Most programs consist of a number of loops in which a series of instructions repeats until a specific requirement achieved. Also, a program can consist of various test to determine which of several actions to take. Thus, a transfer of control to the address of an instruction that does not immediately follow the one currently executing is required. This transfer of control may be forward, to execute a new series of instructions or backward, to re-execute the same instructions. The address of the next instruction to be executed is contained in the program counter and is automatically incremented each time after the instruction is fetched from the memory, so that it contains the address of the next instruction in sequence. Hence, it is the program control type of instruction that can change address of program counter and causes the flow of control of program to be altered.

Different **program control instructions** are listed in the Table 5.10. Branch and Jump instructions can be conditional or unconditional. The conditional branch instructions are those instructions that contains some condition and the **program counter** is loaded with the branch address when the specified condition on the instruction is fulfilled, and the next **instruction** is fetch from that address. If the condition is false, the program counter is not changed and the next instructions is fetched from the next location in sequence. Both branch and Jump instructions can used interchangeably and is usually a one-address instruction, as **BR ADR** where **ADR** is a symbolic name given to an address. When the branch instruction is executed, it causes the value ADR to transfer into the program counter and the next instructions will fetched from this location **ADR**.

Program Control Instructions

Name of Instruction	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare	CMP
Test	TST

Table 5.10