Unit-2

Instruction:-The tasks carried out by a computer program consists of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. Basic computer instructions are the elementary operations that a computer system can perform. These instructions are typically divided into three categories: data transfer instructions, arithmetic and logic instructions, and control instructions.

- Data transfer instructions are used to move data between different parts of the computer system. These instructions include load and store instructions, which move data between memory and the CPU, and input/output (I/O) instructions, which move data between the CPU and external devices.

- Arithmetic and logic instructions are used to perform mathematical operations and logical operations on data stored in the system. These instructions include add, subtract, multiply, and divide instructions, as well as logic instructions such as AND, OR, and NOT.

- Control instructions are used to control the flow of instructions within the computer system. These instructions include branch instructions, which transfer control to different parts of the program based on specified conditions, and jump instructions, which transfer control to a specified memory location.

Register Transfer Notation:- Register transfer notation used to describe how data is passed between CPU registers during the execution of instructions. It is written in human readable format as close to assembly language as possible.

- ➢ Transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem.

- ➢ Most of the time, we identify a location by a symbolic name standing for its hardware binary address. Example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2;

- ➢ processor registers names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on.

- ➢ The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression R1<= [LOC] Means that the contents of memory location LOC are transferred into processor register R1.

- ➢ As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as R3<= [R1] [R2]. This type of notation is known as Register Transfer Notation (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left- hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

Assembly Language Notation:- Another type of notation to represent machine instructions and programs. For this, we use an assembly language format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement Move LOC, R1

- The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1are overwritten.
- The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

  Add R1, R2, R3

Basic Instruction Formats:-

The instruction formats are a sequence of bits (0 and 1). These bits, when grouped, are known as fields. Each field of the machine provides specific information to the CPU related to the operation and location of the data.

The instruction format also defines the layout of the bits for an instruction. It can be of variable lengths with multiple numbers of addresses. These address fields in the instruction format vary as per the organization of the registers in the CPU. The formats supported by the CPU depend upon the Instructions Set Architecture implemented by the processor.

Depending on the multiple address fields, the instruction is categorized as follows:

1. Three address instruction
2. Two address instruction
3. One address instruction
4. Zero address instruction

The operations specified by a computer instruction are executed on data stored in memory or processor registers. The operands residing in processor registers are specified with an address. The registered address is a binary number of k bits that defines one of the 2^k registers in the CPU. Thus, a CPU with 16 processors registers R0 through R15 and will have a four-bit register address field.

Example: The binary number 0011 will designate register R3.

A computer can have instructions of different lengths containing varying numbers of addresses. The number of address fields of a computer depends on the internal design of its registers. Most of the computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. Single Accumulator Organization

All the operations on a system are performed with an implied accumulator register. The instruction format in this type of computer uses **one address field**.

For example, the instruction for arithmetic addition is defined by an assembly language instruction 'ADD.'

Where X is the operand's address, the ADD instruction results in the operation.

$AC \leftarrow AC + M[X]$.

AC is the accumulator register, M[X] symbolizes the memory word located at address X.

2. General Register Organization

The general register type computers employ **two or three address fields** in their instruction format. Each address field specifies a processor register or a memory. An instruction symbolized by ADD R1, X specifies the operation $R1 \leftarrow R + M[X]$.

This instruction has two address fields: register R1 and memory address X.

3. Stack Organization

A computer with a stack organization has PUSH and POP instructions that require an address field. Hence, the instruction PUSH X pushes the word at address X to the top of the stack. The stack pointer updates automatically. In stack-organized computers, the operation type instructions don't require an address field as the operation is performed on the two items on the top of the stack.

Types of Instruction Formats

1. Zero Address Instruction

This instruction does not have an operand field, and the location of operands is implicitly represented. The stack-organized computer system supports these instructions. To evaluate the arithmetic expression, it is required to convert it into reverse polish notation.

Example: Consider the below operations, which shows how X = (A + B) ∗ (C + D) expression will be written for a stack-organized computer.

TOS: Top of the 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

2. One Address Instruction

This instruction uses an implied accumulator for data manipulation operations. An accumulator is a register used by the CPU to perform logical operations. In one address instruction, the accumulator is implied, and hence, it does not require an explicit reference. For multiplication and division, there is a need for a second register. However, here we will neglect the second register and assume that the accumulator contains the result of all the operations.

Example: The program to evaluate X = (A + B) ∗ (C + D) is as follows:

LOAD	A	AC ← M [A]
ADD	B	AC ← A [C] + 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 accumulator(AC) register and a memory operand.
M[ ] is any memory location. M[T] addresses a temporary memory location for storing the intermediate result.
This instruction format has only one operand field. This address field uses two special instructions to perform data transfer, namely:

- LOAD: This is used to transfer the data to the accumulator.
- STORE: This is used to move the data from the accumulator to the memory.

3. Two Address Instructions

This instruction is most commonly used in commercial computers. This address instruction format has three operand fields. The two address fields can either be memory addresses or registers.

MODE	OPCODE	OPERAND 1	OPERAND 2

Example: 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 transfers the operands to the memory from the processor registers. R1, R2 registers.

4. Three Address Instruction

The format of a three address instruction requires three operand fields. These three fields can be either memory addresses or registers.

MODE	OPCODE	OPERAND 1	OPERAND 2	OPERAND 3

Example: The program in assembly language X = (A + B) ∗ (C + D) Consider the instructions given below that explain each instruction's register transfer operation.

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

Two processor registers, R1 and R2.

The symbol M [A] denotes the operand at memory address symbolized by A. The operand1 and operand2 contain the data or address that the CPU will operate. Operand 3 contains the result’s address.

Stack Organisation:

Register 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 stack in digital computers is essentially a memory unit with an address register that can count only. The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack. The physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

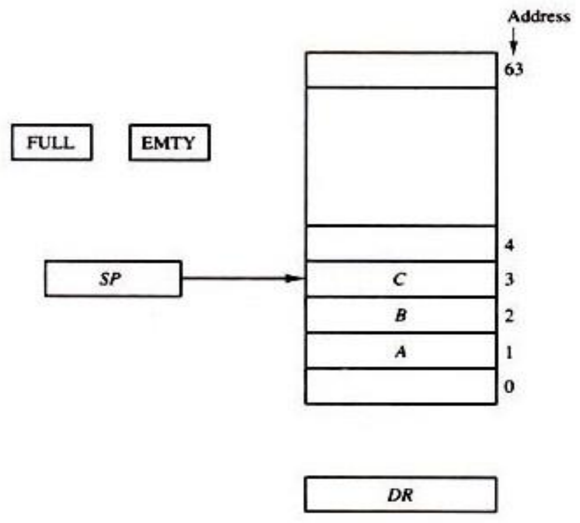


Fig 3.2 Register Stack Organisation

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. Figure shows the organization of a 64-word register stack.

The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address2. To insert a 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 are incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 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:

If the stack is not full (FULL =0), a new item is inserted with a push operation. The push operation consists of the following sequences of micro operations:

SP ← SP+ 1	Increment stack pointer
M [SP]←DR	WRITE ITEM ON TOP OF THESTACK
IF (SP = 0) then (FULL←1)	Check is stack is full
EMTY←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 into the top of the stack. SP holds the address of the top of the stack and that M[SP] denotes the memory word specified by the address presently available in SP. 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 set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so 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 sequences of micro operations:

DR ←M[SP]	Read item on top of the stack
SP ← SP - 1	Decrement stack pointer
IF (SP = 0) then (EMTY ← 1)	Check if stack is empty
FULL←0	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 to1.This condition is reached if the item read was in location1. Once this item 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 SP is decremented, SP is changes to 111111, which is equivalent to decimal63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL = 1 or popped when EMTY =1.

Memory Stack:

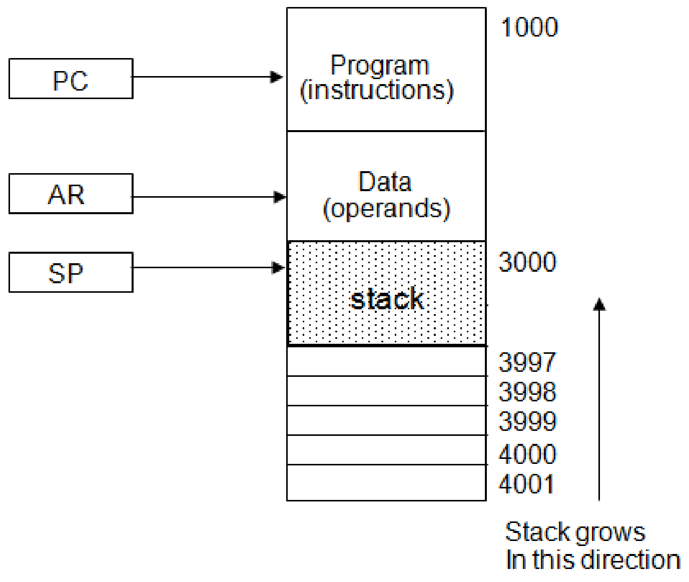


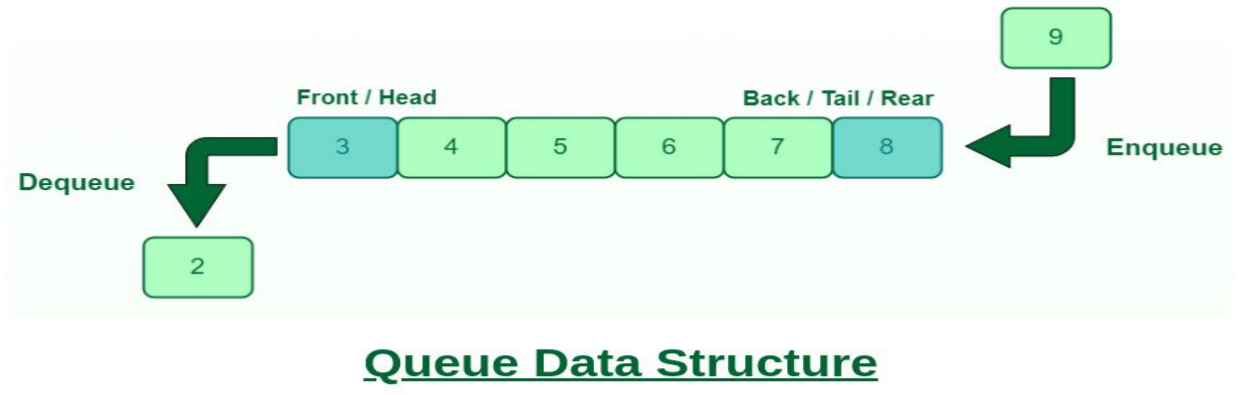
Fig 3.3 Memory Stack Organisation

The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program which is used during the fetch phase to read an instruction. The address registers AR points at an array of data which is used during the execute phase to read an operand. The stack pointer SP points at the top of the stack which is used to push or pop items into or from the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. 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 is3000. We assume that the items in the stack communicate with a data register DR.

Queue :-

- Another useful data structure that is similar to the stack is called a queue.
  - Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis.
  - Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.
- There are two important differences between how a stack and a queue are implemented.
  - One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time.
  - On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

- We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



Fifo Property in Queue

Queue Representation:

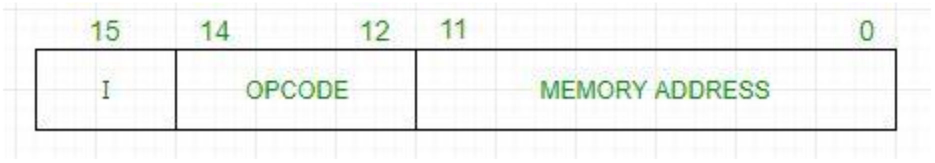
Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

- **Queue:** the name of the array storing queue elements.
- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Basic Instruction Types :-

The basic computer has 16-bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

1. **Memory Reference –** These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12-bit address, 3-bit opcode (other than 111) and 1-bit addressing mode for direct (0) and indirect (1) addressing.



Example – IR register contains = 0001XXXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

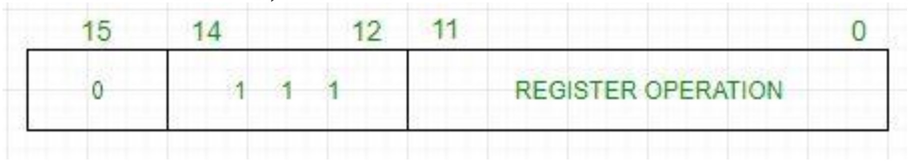
Hence, DR ← M [AR]

AC ← AC + DR, SC ← 0

Symbol	Hexadecimal Code		Description
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	Add memory word to AC

Symbol	**Hexadecimal Code**		**Description**
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store AC content in memory
BUN	4xxx	Cxxx	Branch Unconditionally
BSA	5xxx	Dxxx	Branch and Save Return Address
ISZ	6xxx	Exxx	Increment and skip if 0

2. **Register Reference –** These instructions perform operations on registers rather than memory addresses. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.

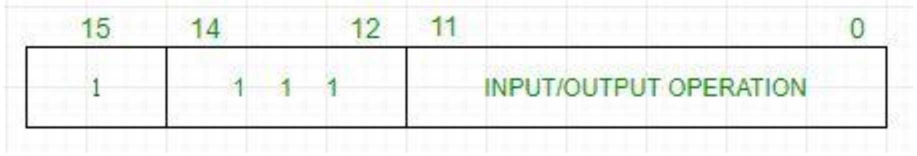


Example – IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.
Hence, AC ← ~AC

Symbol	Hexadecimal Code	Description
CLA	7800	Clear AC
CLE	7400	Clear E(overflow bit)
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right AC and E
CIL	7040	Circulate left AC and E
INC	7020	Increment AC
SPA	7010	Skip next instruction if AC > 0

SNA	7008	Skip next instruction if AC < 0
SZA	7004	Skip next instruction if AC = 0
SZE	7002	Skip next instruction if E = 0
HLT	7001	Halt computer

3. **Input/ Output –** These instructions are for communication between computer and outside environment. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.



Example – IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputing character. Hence, INPUT character from peripheral device.

Symbol	Hexadecimal Code	Description
INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt On
IOF	F040	Interrupt Off

Arithmetic Instructions:- These are the instructions which perform basic arithmetic operations such as addition, subtraction and a few more. In 8085 microprocessor, the destination operand is generally the accumulator. Following is the table showing the list of arithmetic instructions:

Opcode	**Operand**	**Explanation**	**Example**
ADD	R	A = A + R	ADD B

ADD	M	A = A + Mc	ADD 2050
ADI	8-bit data	A = A + 8-bit data	ADI 50
ADC	R	A = A + R + prev. carry	ADC B
ADC	M	A = A + Mc + prev. carry	ADC 2050
ACI	8-bit data	A = A + 8-bit data + prev. carry	ACI 50
SUB	R	A = A – R	SUB B
SUB	M	A = A – Mc	SUB 2050
SUI	8-bit data	A = A – 8-bit data	SUI 50
SBB	R	A = A – R – prev. carry	SBB B
SBB	M	A = A – Mc -prev. carry	SBB 2050
SBI	8-bit data	A = A – 8-bit data – prev. carry	SBI 50
INR	R	R = R + 1	INR B
INR	M	M = Mc + 1	INR 2050
DCR	R	R = R – 1	DCR B
DCR	M	M = Mc – 1	DCR 2050

1 Add) – The content of operand are added to the content of the accumulator and the result is stored in accumulator .
with the addition instruction , the following 3 operations can be done.

1) any 8 bit number can be added to the contents of the accumulator and the result is stored in the accumulator.

2) The contents of a register can be added to the contents of the accumulator and result is stored in the accumulator .

3) The contents of a memory location can be added to the contents of the accumulator and result is stored in accumulator.

this 1 byte instruction.

example – add b c it adds the content of accumulator to the content of the register b

2) SUB Any 8 bit data or the contents of a register or contents of a memory location can be subtracted from the contents of the accumulator.
with the subtraction instruction the following 3 operator can be done .

1) any 8 bit number can be subtracted from the contents of the accumulator . The result is stored in the accumulator.

In the table, R stands for register M stands for memory Mc stands for memory contents r.p. stands for register pair

Logical instructions:-These are a set of instructions that perform logical operations on data in registers and memory. Logical operations are operations that manipulate the bits of data without affecting their numerical value. These operations include AND, OR, XOR, and NOT, the destination operand is always the accumulator. Here logical operation works on a bitwise level.

The logical instructions in the 8085 microprocessor include:

1. ANA – Logical AND: This instruction performs a logical AND operation between the accumulator and a specified register or memory location, and stores the result in the accumulator. For example, the instruction “ANA B” performs a logical AND operation between the contents of the accumulator and the contents of the B register.
2. ORA – Logical OR: This instruction performs a logical OR operation between the accumulator and a specified register or memory location, and stores the result in the accumulator. For example, the instruction “ORA C” performs a logical OR operation between the contents of the accumulator and the contents of the C register.
3. XRA – Logical XOR: This instruction performs a logical XOR operation between the accumulator and a specified register or memory location, and stores the result in the accumulator. For example, the instruction “XRA M” performs a logical XOR operation between the contents of the accumulator and the contents of the memory location pointed to by the HL register.
4. CPL – Logical Complement: This instruction performs a logical complement operation on the contents of the accumulator. This operation flips all the bits of the accumulator, effectively reversing its value.
5. CMA – Complement Accumulator: This instruction performs a bitwise complement operation on the contents of the accumulator. This operation flips all the bits of the accumulator, effectively reversing its value.

Following is the table showing the list of logical instructions:

OPCODE	**OPERAND**	**DESTINATION**	**EXAMPLE**
ANA	R	A = A AND R	ANA B
ANA	M	A = A AND Mc	ANA 2050
ANI	8-bit data	A = A AND 8-bit data	ANI 50
ORA	R	A = A OR R	ORA B
ORA	M	A = A OR Mc	ORA 2050
ORI	8-bit data	A = A OR 8-bit data	ORI 50

OPCODE	OPERAND	DESTINATION	EXAMPLE
XRA	R	A = A XOR R	XRA B
XRA	M	A = A XOR Mc	XRA 2050
XRI	8-bit data	A = A XOR 8-bit data	XRI 50
CMA	none	A = 1's complement of A	CMA
CMP	R	Compares R with A and triggers the flag register	CMP B
CMP	M	Compares Mc with A and triggers the flag register	CMP 2050
CPI	8-bit data	Compares 8-bit data with A and triggers the flag register	CPI 50
RRC	none	Rotate accumulator right without carry	RRC
RLC	none	Rotate accumulator left without carry	RLC
RAR	none	Rotate accumulator right with carry	RAR
RAL	none	Rotate accumulator left with carry	RAR
CMC	none	Compliments the carry flag	CMC
STC	none	Sets the carry flag	STC

In the table,
R stands for register
M stands for memory
Mc stands for memory contents

Shift micro-operations are those micro-operations that are used for the serial transfer of information. These are also used in conjunction with arithmetic micro-operation, logic micro-operation, and other data-processing operations. There are three types of shift micro-operations:

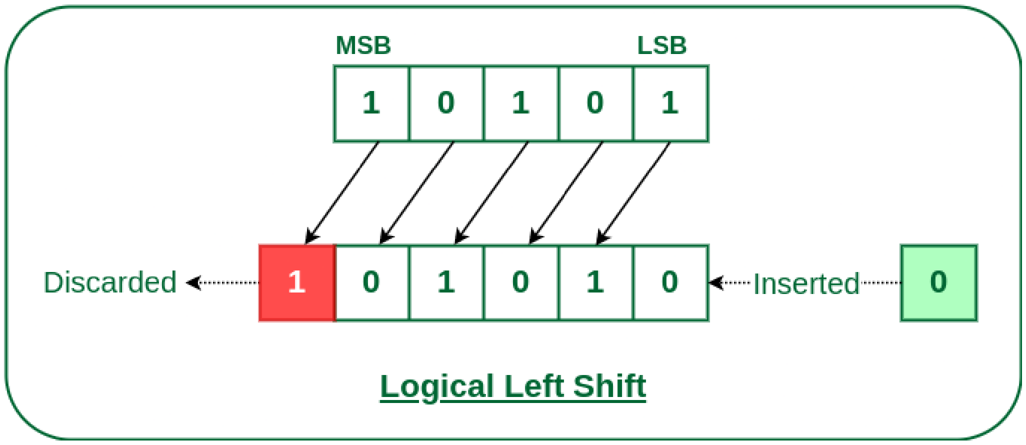
<u>1. Logical Shift:</u>

It transfers the 0 zero through the serial input. We use the symbols '<<' for the logical left shift and '>>' for the logical right shift.

- **Logical Left Shift:**

In this shift, one position moves each bit to the left one by one. The Empty least significant bit (LSB) is filled with zero (i.e, the serial input), and the most significant bit (MSB) is rejected.

The left shift operator is denoted by the double left arrow key (<<). The general syntax for the left shift is shift-expression << k.



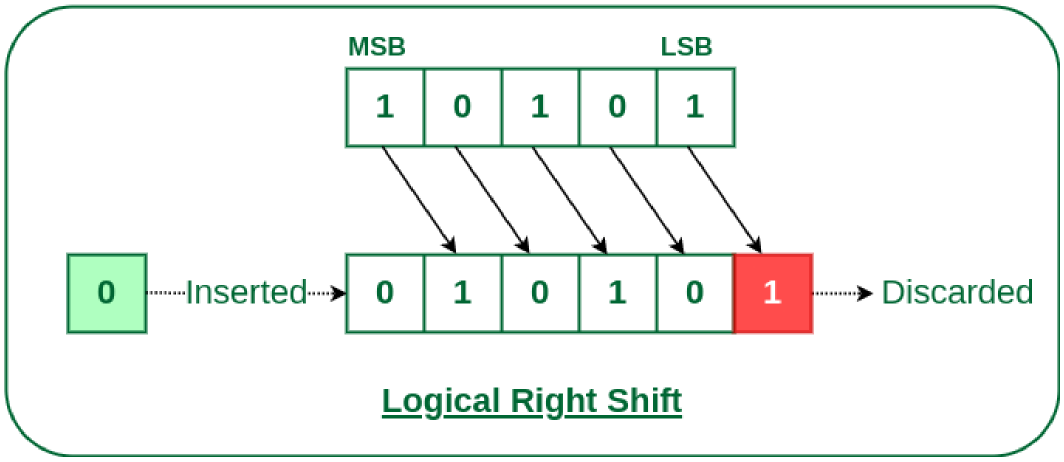
Logical Left Shift

Note: Every time we shift a number towards the left by 1 bit it multiplies that number by 2.

- **Logical Right Shift**

In this shift, each bit moves to the right one by one and the least significant bit(LSB) is rejected and the empty MSB is filled with zero.

The right shift operator is denoted by the double right arrow key (>>). The general syntax for the right shift is "shift-expression >> k".



Logical Right Shift

Note: Every time we shift a number towards the right by 1 bit it divides that number by 2.

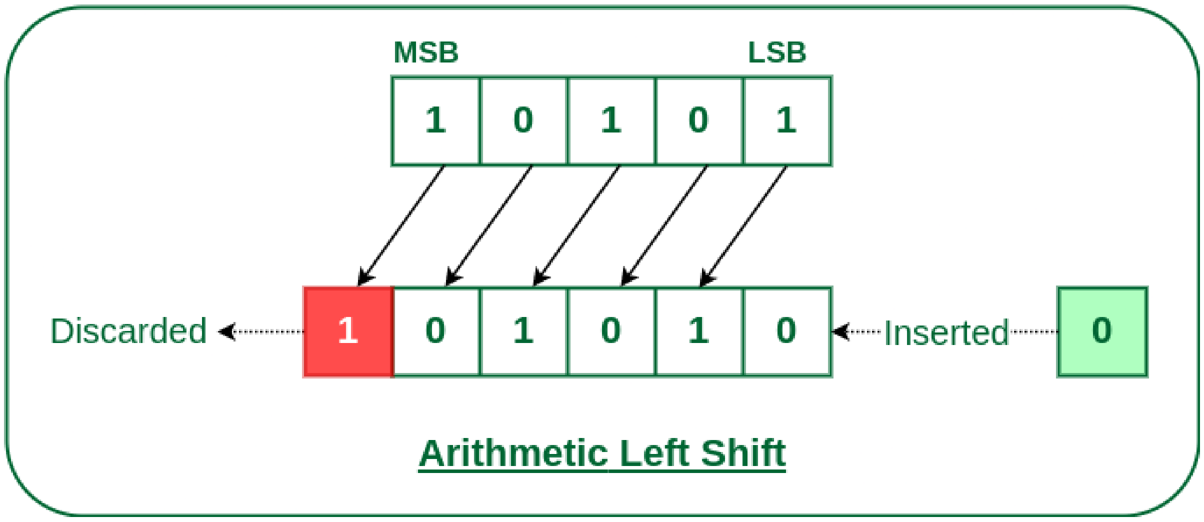
<u>2. Arithmetic Shift:</u>

The arithmetic shift micro-operation moves the signed binary number either to the left or to the right position. Following are the two ways to perform the arithmetic shift.

1. Arithmetic Left Shift
2. Arithmetic Right Shift

- **Arithmetic Left Shift:**

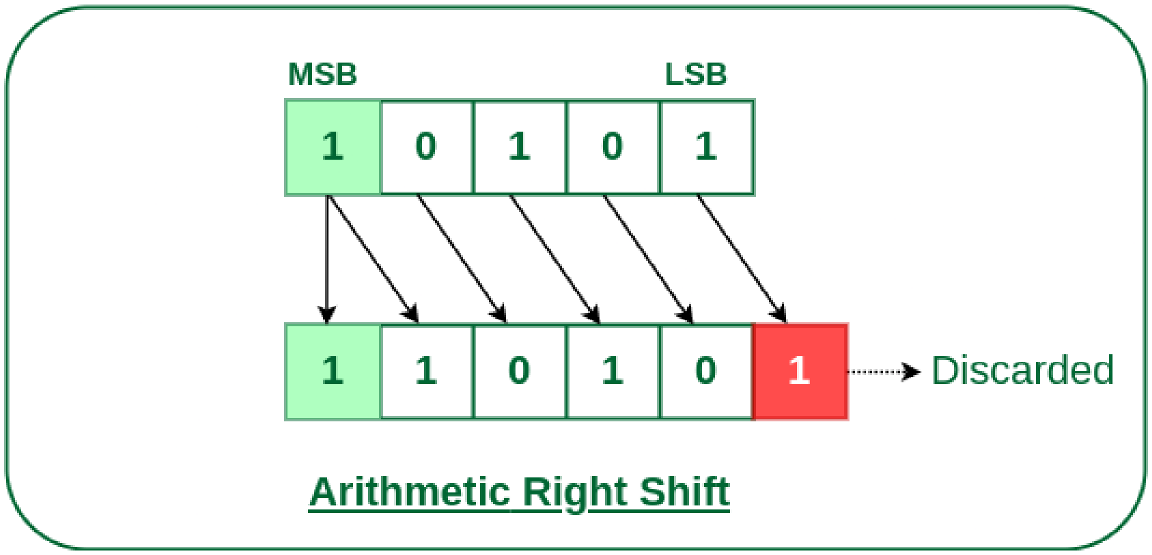
In this shift, each bit is moved to the left one by one. The empty least significant bit (LSB) is filled with zero and the most significant bit (MSB) is rejected. Same as the Left Logical Shift.



Arithmetic Left Shift

➢ **Arithmetic Right Shift:**

In this shift, each bit is moved to the right one by one and the least significant(LSB) bit is rejected and the empty most significant bit(MSB) is filled with the value of the previous MSB.



Arithmetic Right Shift

<u>3. Circular Shift:</u>

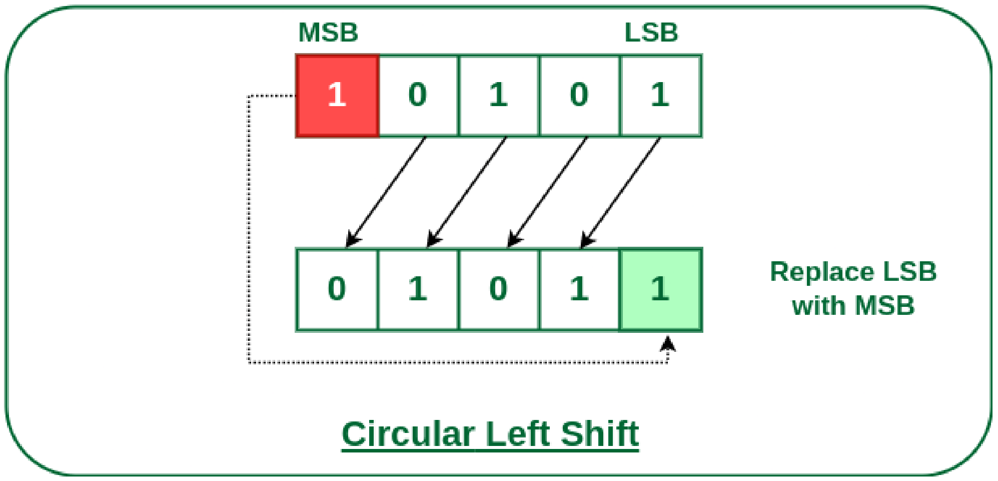
The circular shift circulates the bits in the sequence of the register around both ends without any loss of information.

Following are the two ways to perform the circular shift.

1. Circular Shift Left
2. Circular Shift Right

➢ **Circular Left Shift:**

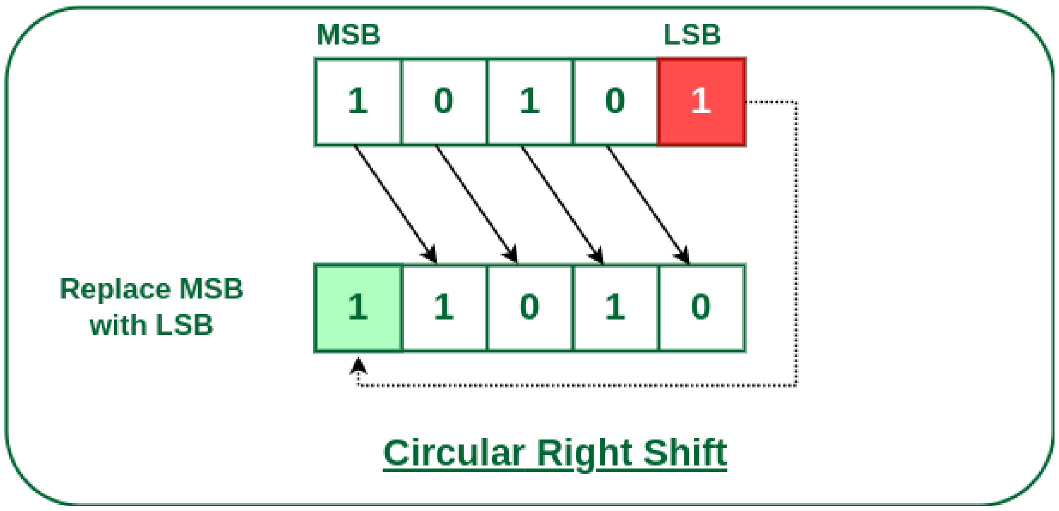
In this micro shift operation each bit in the register is shifted to the left one by one. After shifting, the LSB becomes empty, so the value of the MSB is filled in there.



Circular Left Shift

- **Circular Right Shift:**

In this micro shift operation each bit in the register is shifted to the right one by one. After shifting, the MSB becomes empty, so the value of the LSB is filled in there.



Circular Right Shift

Branching instructions refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction.

The three types of branching instructions are:

1. Jump (unconditional and conditional)
2. Call (unconditional and conditional)
3. Return (unconditional and conditional)

1. **Jump Instructions –** The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag. Jump instructions are 2 types: Unconditional Jump Instructions and Conditional Jump Instructions.

(a) Unconditional Jump Instructions: Transfers the program sequence to the described memory address.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
JMP	address	Jumps to the address	JMP 2050

(b) Conditional Jump Instructions: Transfers the program sequence to the described memory address only if the condition in satisfied.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
JC	address	Jumps to the address if carry flag is 1	JC 2050
JNC	address	Jumps to the address if carry flag is 0	JNC 2050
JZ	address	Jumps to the address if zero flag is 1	JZ 2050
JNZ	address	Jumps to the address if zero flag is 0	JNZ 2050
JPE	address	Jumps to the address if parity flag is 1	JPE 2050
JPO	address	Jumps to the address if parity flag is 0	JPO 2050
JM	address	Jumps to the address if sign flag is 1	JM 2050
JP	address	Jumps to the address if sign flag 0	JP 2050

2. Call Instructions – The call instruction transfers the program sequence to the memory address given in the operand. Before transferring, the address of the next instruction after CALL is pushed onto the stack. Call instructions are 2 types: Unconditional Call Instructions and Conditional Call Instructions.
(a) Unconditional Call Instructions: It transfers the program sequence to the memory address given in the operand.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
CALL	address	Unconditionally calls	CALL 2050

(b) Conditional Call Instructions: Only if the condition is satisfied, the instructions executes.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
CC	address	Call if carry flag is 1	CC 2050
CNC	address	Call if carry flag is 0	CNC 2050
CZ	address	Calls if zero flag is 1	CZ 2050

OPCODE	OPERAND	EXPLANATION	EXAMPLE
CNZ	address	Calls if zero flag is 0	CNZ 2050
CPE	address	Calls if parity flag is 1	CPE 2050
CPO	address	Calls if parity flag is 0	CPO 2050
CM	address	Calls if sign flag is 1	CM 2050
CP	address	Calls if sign flag is 0	CP 2050

3. Return Instructions – The return instruction transfers the program sequence from the subroutine to the calling program. Return instructions are 2 types: Unconditional Jump Instructions and Conditional Jump Instructions.
(a) Unconditional Return Instruction: The program sequence is transferred unconditionally from the subroutine to the calling program.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
RET	none	Return from the subroutine unconditionally	RET

(b) Conditional Return Instruction: The program sequence is transferred unconditionally from the subroutine to the calling program only is the condition is satisfied.

OPCODE	OPERAND	EXPLANATION	EXAMPLE
RC	none	Return from the subroutine if carry flag is 1	RC
RNC	none	Return from the subroutine if carry flag is 0	RNC
RZ	none	Return from the subroutine if zero flag is 1	RZ
RNZ	none	Return from the subroutine if zero flag is 0	RNZ
RPE	none	Return from the subroutine if parity flag is 1	RPE
RPO	none	Return from the subroutine if parity flag is 0	RPO
RM	none	Returns from the subroutine if sign flag is 1	RM
RP	none	Returns from the subroutine if sign flag is 0	RP

Basic Input/output operations:-

- The data on which the instructions operate are not necessarily already stored in memory. Data
- need to be transferred between processor and outside world (disk, keyboard, etc.) Input/ Output
- (I/O) operations are essential, and the way they are performed can have asignificant effect on the performance of the computer.
- Let's consider an example of read in a character input from a keyboard and produce character output on a display screen.
- A simple way of performing such I/O tasks is to use method known as program-controlled I/O
- Rate of data transfer of different units (keyboard, display, processor) participating in this may vary.
- Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.

- A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on.
- Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.
- The keyboard and the display are separate devices as shown in Figure below.
- The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen.
- One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

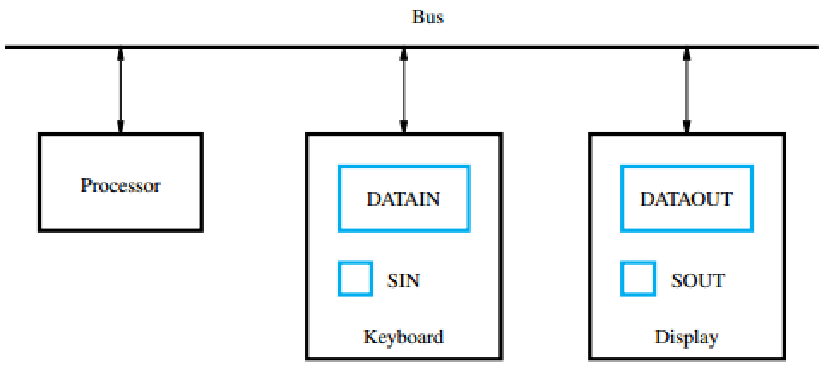


Figure Bus connection for processor, keyboard, and display.

- Consider the problem of moving a character code from the keyboard to the processor. Striking a key store the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN, as shown in the figure. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.
- An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1.
- The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device interface.