Computer Organization

ASR-CO-2.1

UNIT-2

JNTUK
R16
2-2 CSE

Syllabus : Machine Instruction and Programs

- Instructions and Instruction Sequencing
  - Register Transfer Notation
  - Assembly language Notation
  - Basic Instruction types
- Addressing Modes
- Basic Input Output Operations
- The Role of Stacks and Queues in Computer Programming Equation
- Component of Instructions
  - Logic Instructions
  - Shift and Rotate Instructions.

Text Book :

1. Computer Organization, Carl Hamacher, Zvonks Vranesic, Safea Zaky, 5th Edition, McGraw Hill.

(1) Instructions and Instruction Sequencing:

→ A computer must have instructions capable of performing four types of operations.

- Data transfers between the memory and the processor registers.
- Arithmetic and Logic operations on data
- Program Sequencing and Control.
- I/O Transfers.

(i) Register Transfer Notation:

→ We need to describe the 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.

→ Names for the addresses of memory locations may be LOC, PLACE, A, VAR2 processor register names may be R_0, R_5 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.

Example: $R1 \leftarrow [LOC]$

- means that the contents of memory location LOC are transferred into processor register R1.

Example: $R3 \leftarrow [R1] + [R2]$

- means the operation that adds the contents of registers R1 and R2 and then places their sum into register R3.

→ This type of notation is known as Register Transfer Notation (RTN)

→ The right-hand side of RTN is always denotes a value and the left-hand side is the name of a location where the value is to be placed.

(ii) Assembly Language Notation:

→ Assembly language format is a notation to represent machine instructions and programs.

Example: MOVE LOC, R1

- Here data transferred from memory location LOC to processor register R1
- The contents of LOC are ~~not~~ unchanged by the execution of this instruction, but the old contents of Register R1 are overwritten.

Example: Add R1, R2, R3

- Here, 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,~~

(iii) Basic Instruction Types:

→ There are 4 types of Instructions available

a) Three-address Instructions
b) Two-address Instructions
c) One-address Instructions
d) Zero-address Instructions

a) Three-address Instructions:

→ An instruction having three operands

Syntax: Opcode Source1, Source2, Destination

Example: Adding two numbers

ADD A,B,C (∵ C ← [A] + [B])

- where A,B are called Source Operands, C is called destination operand.
- Opcode means Operation Code.

b) Two-address Instructions:

→ An instruction having two operands

Syntax: opcode Source, Destination

Example: MOVE B,C (∵ C ← [B])
ADD A,C (∵ C ← [A] + [C])

c) One-address Instructions:

→ An instruction having only one operand
→ When a second operand is needed, as in the case of an ADD instruction, a processor register, called Accumulator is used.

Example: LOAD A (∵ AC ← [A])
ADD B (∵ AC ← [AC] + [B])
STORE C (∵ C ← [AC])

d) Zero-address Instructions:

→ An instruction having zero operands.
→ It uses stack operations PUSH and POP to perform operations.

Example:

```
PUSH A    (∵ TOS ← [A])
PUSH B    (∵ TOS ← [B])
ADD       (∵ TOS ← [A] + [B])
POP C     (∵ C ← [TOS])
```

Another Example: Evaluate X = (A+B) * (C+D)

Three Address:

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

Two Address:

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

One-Address:

```
LOAD A     (∵ AC ← [A])
ADD B      (∵ AC ← [AC] + [B])
STORE T1   (∵ T1 ← [AC])
LOAD C     (∵ AC ← [C])
ADD D      (∵ AC ← [AC] + [D])
MUL T1     (∵ AC ← [AC] * [T1])
STORE X    (∵ X ← [AC])
```

Zero-Address:

```
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     (∵ X ← [TOS])
```

② Addressing Modes:

→ The different ways in which the location of an operand is specified in an instruction are referred to as Addressing Modes.

→ The different Generic Addressing Modes is given as

S.NO	Name	Assembler Syntax	Addressing function
1	Immediate	#Value	Operand = Value
2	Register	R_i	EA = R_i
3	Absolute (Direct)	LOC	EA = LOC
4	Indirect	(R_i) (LOC)	EA = $[R_i]$ EA = [LOC]
5	Index	$X(R_i)$	EA = $[R_i]$ + X
6	Base with Index	(R_i, R_j)	EA = $[R_i]$ + $[R_j]$
7	Base with index and Offset	$X(R_i, R_j)$	EA = $[R_i]$ + $[R_j]$ + X
8	Relative	X(PC)	EA = [PC] + X
9	Auto Increment	(R_i) +	EA = $[R_i]$; Increment R_i
10	Auto Decrement	$-(R_i)$	Decrement R_i EA = $[R_i]$

(i) Implementation of Variables and Constants:

→ In Assembly language, a variable is represented by allocating a register (or) a memory location to hold its value.

→ There are two addressing modes to access variables

a) Register mode

b) Absolute mode.

*) Register Mode:

→ The operand is the contents of a processor Register

Example: MOVE R1, R2

- Here R_1, R_2 are registers.

*) Absolute (Direct) Mode:

→ The operand is in a memory location

Example: ADD A,B

→ Address and Data constants are represented in Assembly language using the Immediate mode.

*) Immediate Mode:

→ The operand is given explicitely in the instruction.

Example: MOVE #200, R0

→ places the value 200 in Register R0

→ # sign indicates that this value is to be used as an immediate operand.

Example:

```
MOVE  B, R1
ADD   #7, R1
MOVE  R1, A
```

(ii) Indirection and pointers:

→ In Addressing modes, the instruction does not give the operand (or) its address explicitly.

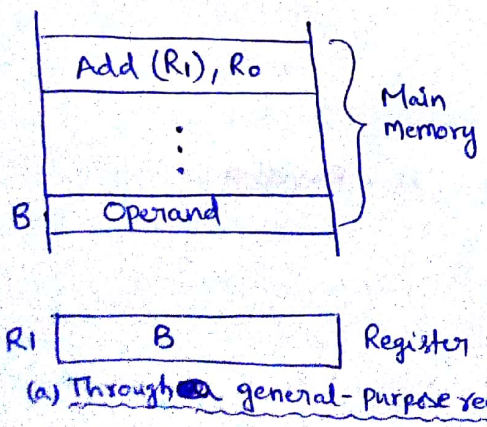
→ Instead, it provides information from which the memory address of the operand can be determined. This address is called as Effective Address(EA)

*) Indirect Mode:

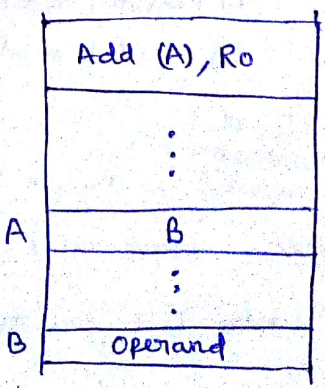
→ The effective address of the operand is the contents of a register (or) memory location whose address appears in the instruction

→ We denote indirection by placing the name of the register (or) the memory address given in the instruction in parantheses.

→ Indirect Addressing is depicted as



(a) Through a general-purpose register



(b) Through a memory location

(a) To execute the Add instruction, the processor uses the value of B, which is register R1, as the effective address of the operand.
- It requests a read operation from the memory to read the contents of location B.
- The value read is the desired operand, which the processor adds to the contents of register R0.

(b) - In Indirect Addressing through a memory location, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand.
- The register (or) memory location that contains the address of an operand is called a pointer.

→ Consider the C-language statement

A = *B;

- where B is a pointer variable
- This statement may be compiled into

```
MOVE  B,R1
MOVE  (R1),A
```

- Using indirect addressing through memory,

```
MOVE  (B),A
```

(iii) Indexing and Arrays:

→ It is useful in dealing with lists and Arrays.

*) Index Mode:

→ The effective address of the operand is generated by adding a constant value to the contents of a register
→ The register used may be either a special register (or) general-purpose register called Index register.
→ Index mode symbolically represented as,

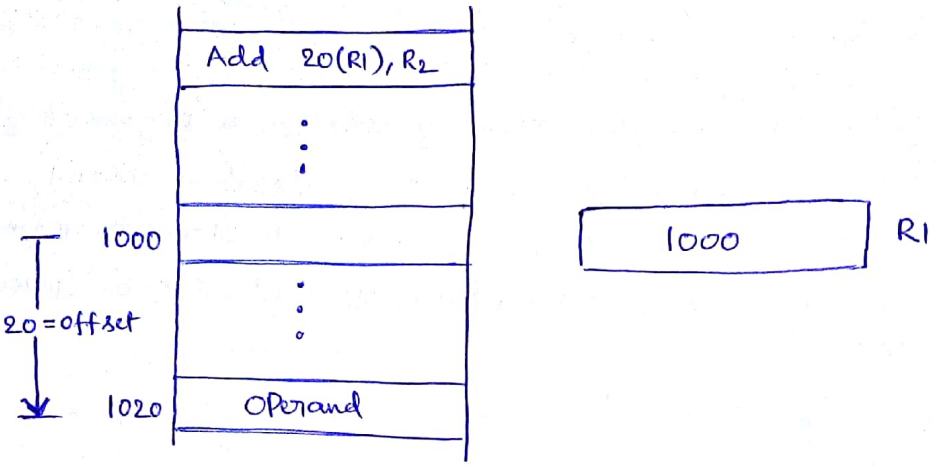
X(Ri)

where X denotes the constant value contained in the instruction
Ri denotes the name of the register involved.
→ The effective address of the operand is given by

EA = X + [Ri]

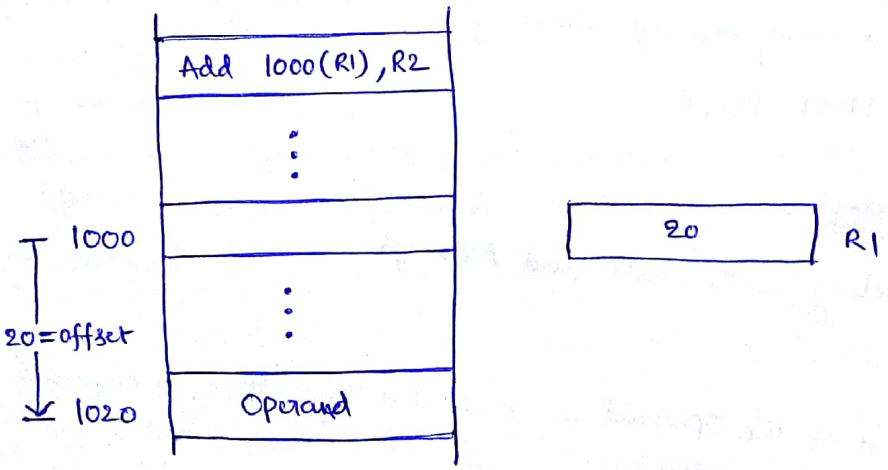
→ The contents of the index register are not changed in the process of generating the effective address.

→ Two ways of using Index mode (Index Addressing) is depicted as,



(a) Offset is given as a Constant

→ The index register R1 contains the address of a memory location, and the value X defines an offset (or) displacement) from this address to the location where the operand is found.



(b) offset is in the index register

→ Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.

→ In either case, the effective address is the sum of two values
- One is given explicitly in the instruction, and
- the other is stored in a register.

(iv) Relative Addressing:

→ Index mode is used for general-purpose processor registers

→ In Relative Addressing, the program counter (PC) is used instead of a general purpose register.

*) Relative Mode:

→ The effective address is determined by the Index mode using the program counter in place of general purpose register R_i

→ Relative mode Symbolically represented as,

$$X(PC)$$

→ The effective address of the operand is given by

$$EA = X + [PC]$$

→ This mode can be used to access data operands

(v) Additional Modes:

(a) Autoincrement Mode

(b) Autodecrement Mode

*) Autoincrement Mode:

→ The effective address of the operand is the contents of a register specified in the instruction

→ After Accessing the operand, the contents of this register are automatically incremented to point to the next item in the list.

→ Symbolically represented as

$$(R_i)+$$

- It normally increments one, but in byte-sized operands (or) byte-addressable memory

  1 for 8-bit operands

  2 for 16-bit operands

  4 for 32-bit operands.

→ The effective Address of the operand is

$$EA = [R_i];$$
$$\text{Increment } R_i$$

(*) Autodecrement Mode:

→ The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand

→ Symbolically represented as,

$$-(R_i)$$

→ The effective address of the operand is

Decrement R_i;
$EA = [R_i]$

(3) Basic Input/Output Operations:

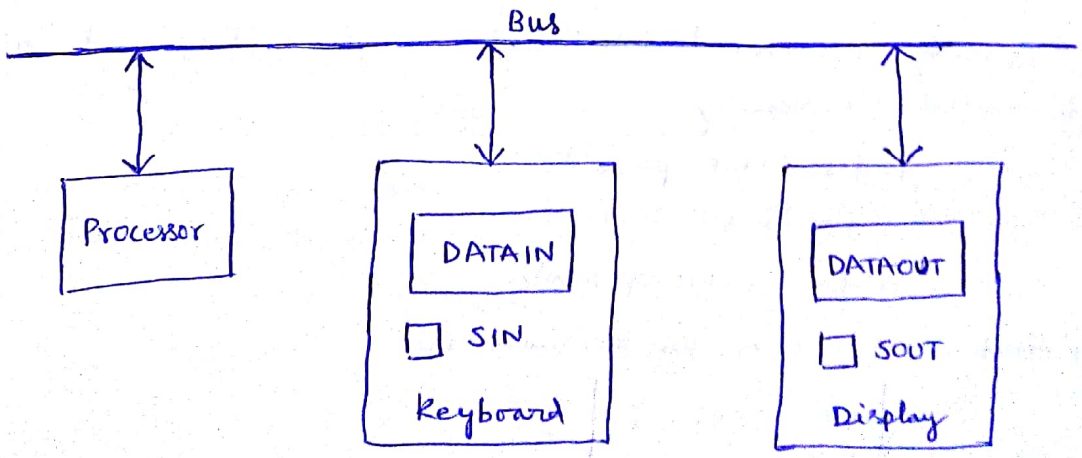
→ I/O operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

→ Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as program-controlled I/O.

→ The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second.

→ The rate of output transfers from the computer is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second.

→ Bus connection for processor, keyboard, and display are depicted as,



→ The Keyboard and the display are separate devices.
→ Consider the problem of moving a character code from the Keyboard to the processor.
→ Striking a key stores the corresponding character code in an 8-bit buffer register associated with the Keyboard.
→ Let us call this register DATAIN
→ 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 the second character is entered at the keyboard, SIN is again set to 1 and the process repeats.
→ When characters are transferred from the processor to the display, a buffer register, DATAOUT, and the status control flag, SOUT, are used for this transfer.
→ The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a device Interface.

Example:

→ The processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to Register R1 by the following sequence of operations

```
READWAIT   Branch to READWAIT If SIN = 0
           Input from DATAIN to R1
```

→ Sequence of operations used for transferring output to the display is given as,

```
WRITEWAIT  Branch to WRITEWAIT If SOUT = 0
           Output from R1 to DATAOUT
```

→ The contents of the keyboard character buffer DATAIN can be transferred to Register R1 in the processor by the instruction

```
MOVEBYTE   DATAIN, R1
```

→ Similarly, the contents of Register R1 can be transferred to DATAOUT by the instruction

```
MOVEBYTE   R1, DATAOUT
```

(4) The Role of STACKS and QUEUES in Computer Programming Equations

→ Inorder to organize the control and information linkage between the main program and the subroutine, a datastructure called a stack is used.

→ A stack is a list of data elements, words (or) bytes, with the accessing restriction that elements can be added (or) removed at one end of the list only.

→ It uses a pointer variable called Top of the stack.

Example :- pile of Trays in a Cafeteria

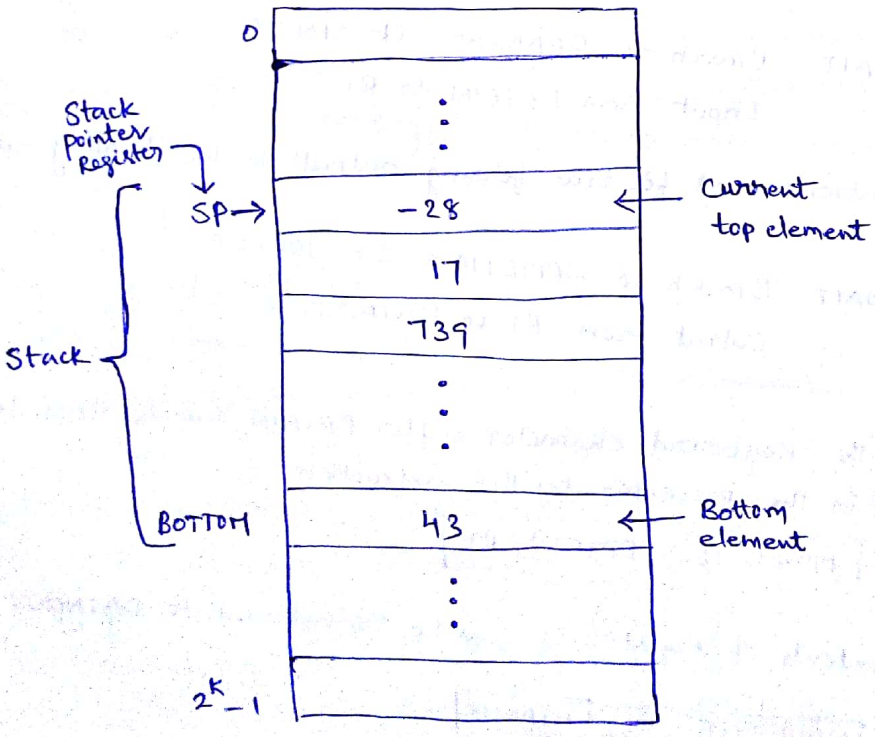
- Customer picks up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile.

→ Stack uses LIFO (Last In First Out) mechanism, which describes the last data item placed on the stack is the first one removed when retrieval begins.

→ PUSH and POP are the operations of stack, which describes placing the new item on the stack and removing the top item from the stack, respectively.

→ Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations.

→ A stack of words in the memory is depicted as



→ It contains numerical values, with 43 at the bottom and -28 at the top.
→ A processor register is used to keep track of the address of the element of the stack that is at the top at any given time, called Stack pointer (SP)
→ In a byte-addressable memory with a 32-bit word length, the PUSH operation can be implemented as,

```
SUB  #4, SP
MOVE NEWITEM, (SP)
```

→ The POP operation can be implemented as

```
MOVE (SP), ITEM
ADD  #4, SP
```

→ If the processor has the Autoincrement and Autodecrement addressing modes, then the PUSH operation can be performed by the single instruction

```
MOVE NEWITEM, -(SP)
```

→ The pop operation can be performed by the single instruction

```
MOVE (SP)+, NEWITEM
```

→ Another useful datastructure that is similar to the stack is called Queue
→ Data are stored in and retrieved from a queue on a First In First Out (FIFO) basis.
→ Here, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.
→ It uses two pointers FRONT and REAR.
→ INSERT and DELETE are the operations of queue.

⑤ Components of Instructions :

(i) Logic Instructions
(ii) Shift and Rotate Instructions

(i) Logic Instructions :

→ Logical operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits
→ It is also useful to be able to perform logic operations in software

Example : 1's complement

```
NOT DST
```

– Complements all bits contained in the destination operand, changing 0's to 1's and 1's to 0's

Example : 2's Complement

```
NOT  R0
ADD  #1, R0
```

→ Many computers have a single instruction for 2's complement

```
NEGATE R0
```

→ Logical operators AND, OR, NOT represented as bits in a table as,

AND

Operand1	Operand2	AND
0	0	0
0	1	0
1	0	0
1	1	1

OR

Operand1	Operand2	OR
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Operand	NOT
0	1
1	0

(ii) Shift and Rotate Instructions:

(a) Shift Instructions:

→ There are many applications that require the bits of an operand to be shifted right (or) left some specified number of bit positions.

→ There are 2 types of Shift Instructions

(*) Logical Shift Instructions

(*) Arithmetic Shift Instructions

(*) Logical Shift Instructions:

→ There are 2 Logical Shift Instructions

- Logical Shift Left (LShiftL)
- Logical Shift Right (LShiftR)

→ These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

• Logical Shift Left (LShiftL):

→ The general form of Logical Shift Left Instruction is

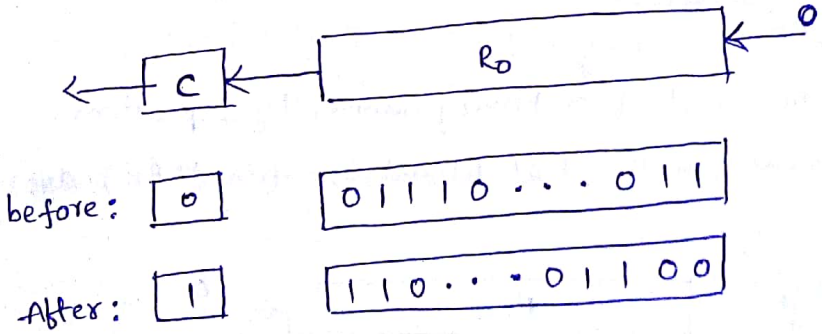
```
LShiftL  count, DST
```

→ The count operand may be an immediate operand (or) it may be contained in a processor register.

Example:

```
LShiftL  #2, R0
```

→ This is depicted as,



Logical Shift Left

→ It shifts the contents of Register R0 left by two positions

→ Vacated positions are filled with 0's.

- **Logical Shift Right:**

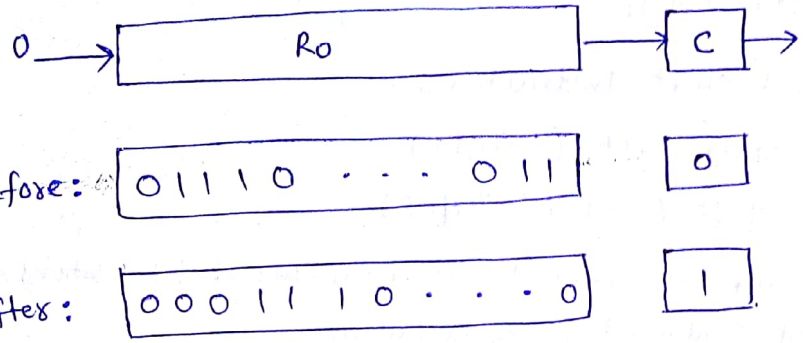
→ The general form is,

`LShiftR Count, DST`

Example :

`LShiftR #2, R0`

→ This is depicted as,



Logical Shift Right

→ It shifts the contents of register R0 right by two bit positions.

→ Vacated positions are filled with zeros.

(*) **Arithmetic Shift Instructions:**

→ There are two types in Arithmetic shift Instructions

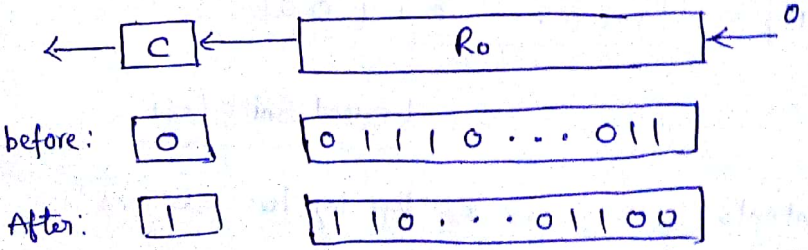
- Arithmetic Shift Left (AShiftL)
- Arithmetic Shift Right (AShiftR)

- **Arithmetic ShiftLeft:**

`AShiftL #2, R0`

→ A Left Arithmetic shift of a binary number by 2 positions

→ The empty positions in the LSB (Least Significant Bit) are filled with zeros.



Arithmetic Shift Left

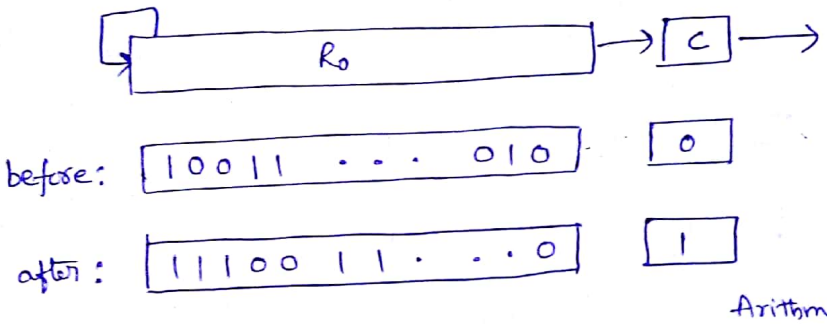
→ It works same like Logical Shift Left operation

• Arithmetic Shift Right:

AShiftR #2, R0

→ A right arithmetic shift of a binary number by 2 positions.

→ The empty positions in the MSB (Most Significant Bit) are filled with copies of the original MSB bit.



Arithmetic shift Right

(b) Rotate Instructions:

→ In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the carry flag C.

→ To preserve all bits, a set of rotate instructions can be used.

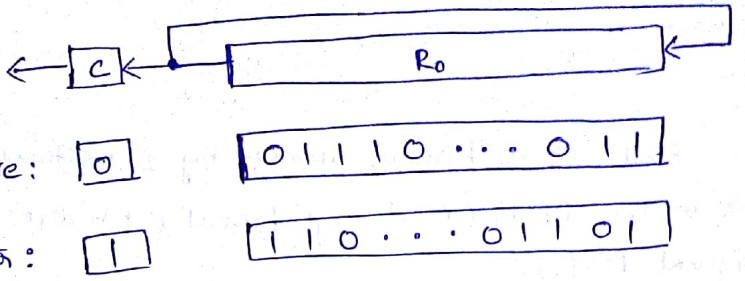
→ They move the bits that are shifted out of one end of the operand back into the other end

→ The different Rotate Instructions are,

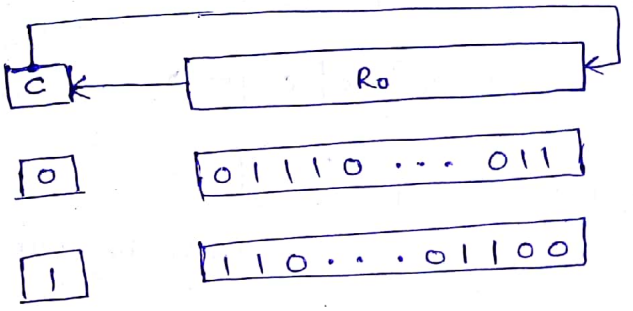
* Rotate Left without Carry (RotateL)
* Rotate Left with Carry (RotateLC)
* Rotate Right without Carry (RotateR)
* Rotate Right with Carry (RotateRC

→ The different Rotate Operations are depicted as,

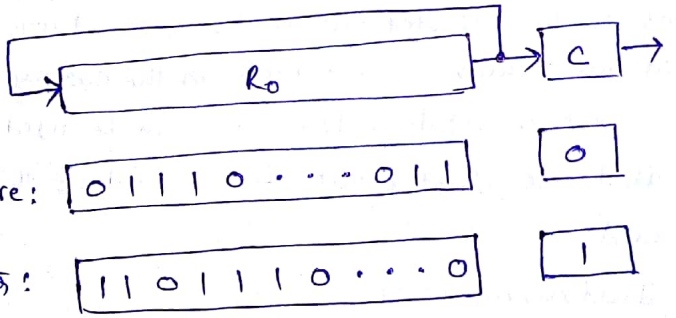
(*) Rotate Left without Carry : RotateL #2, R0



(*) Rotate Left with carry : RotateLC #2, R0



(*) Rotate Right without Carry : RotateR #2, R0



(*) Rotate Right with Carry : RotateRC #2, R0