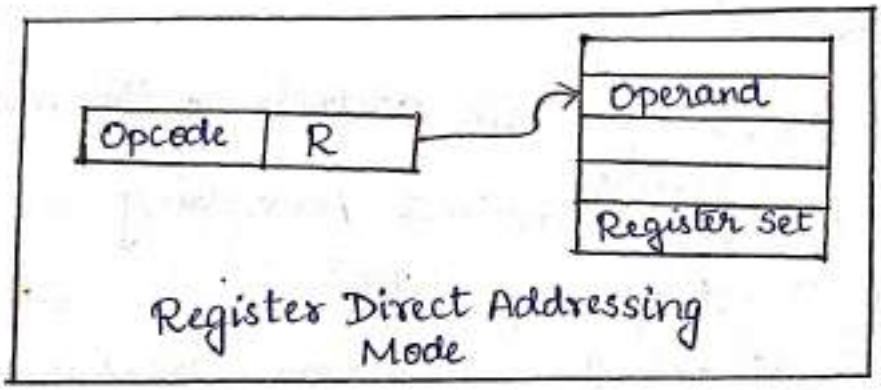
Addressing Modes -
The various formats of representing operand in an Instruction or location of an operand is called as "Addressing Modes".
The different types of Addressing Modes are.

a) Register Addressing
b) Direct Addressing
c) Immediate Addressing
d) Indirect Addressing
e) Index Addressing
f) Relative Addressing
g) Auto Increment Addressing
h) Auto Decrement Addressing

a) Register Addressing -
In this mode operands are sorted in the registers of CPU. The name of the register is directly specified in the instruction.

Ex: MOVE R1, R2

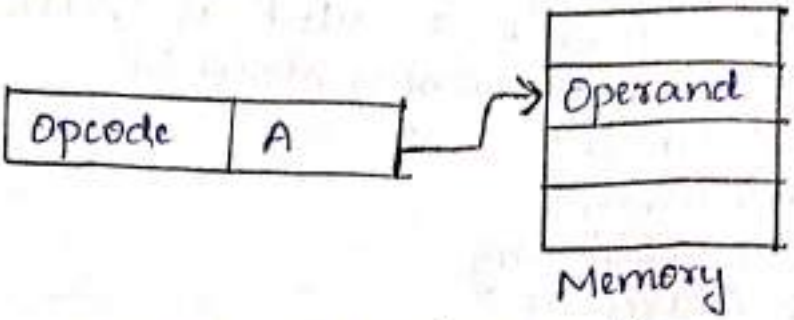
Where R1 and R2 are the Source and Destination registers respectively. This instruction transfers 32 bits of data from R1 register into R2 register. This instruction does not refer memory for operands. The operands are directly available in the registers.



Register Direct Addressing Mode

b) Direct Addressing -
It is also called as Absolute Addressing Mode. In this addressing mode operands are stored in the memory locations. The name of the memory location is directly specified in the instruction.
Ex: MOVE LOCA, R1 : Where LOCA is the memory location and R1 is the Register.

This instruction transfers 32 bits of data from memory location x into the General Purpose Register R1.

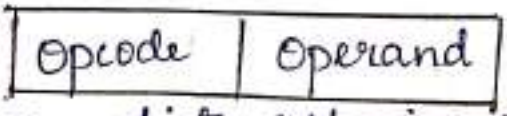


Direct Addressing Mode

c) Immediate Addressing -

In this Addressing Mode operands are directly specified in the Instruction. The source field is used to represent the operands. The operands are represented by #(hash) sign.

Ex: MOVE #23, R0



Immediate Addressing Mode

d) Indirect Addressing -

In this Addressing Mode effective address of an operand is stored in the memory location or General Purpose Register. The memory locations or GPR^S are used as the memory pointers.

Memory pointer: It stores the address of the memory location.

There are two types Indirect Addressing

i. Indirect through GPR^S

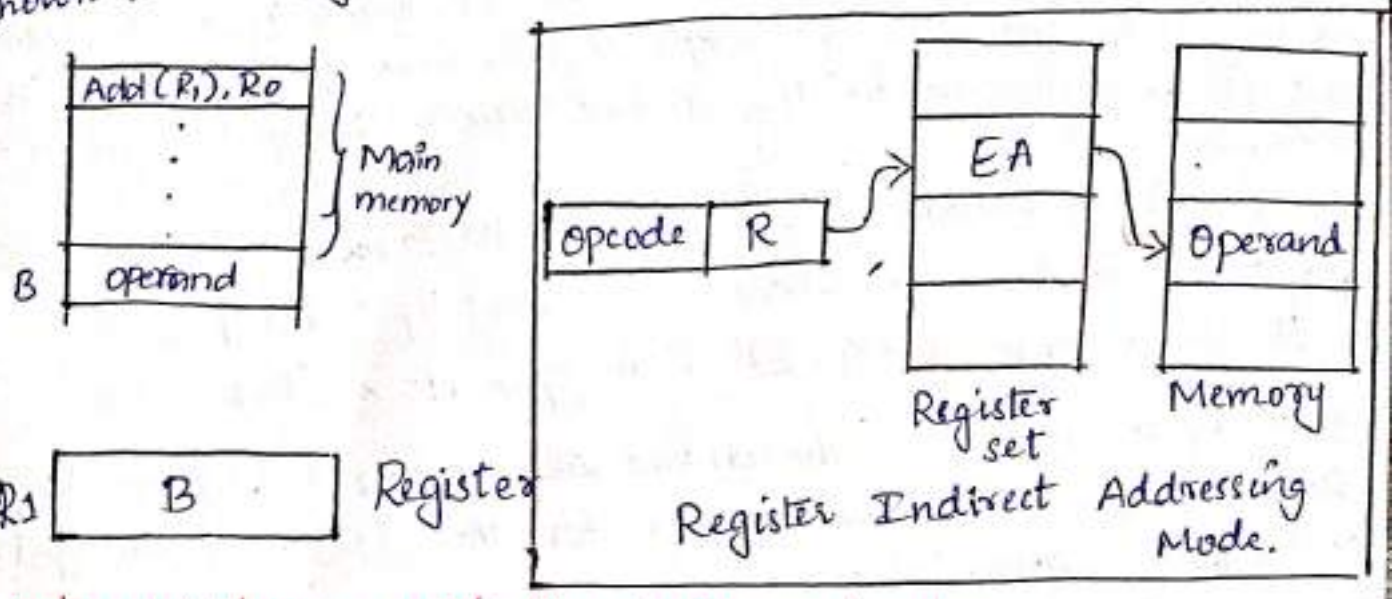
ii. Indirect through memory location

i) Indirect Addressing Mode through GPR^S.

In this Addressing Mode the effective address of an operand is stored in the one of the General Purpose Register of the CPU.

Ex: ADD (R1), R0; where R1 and R0 are GPR^S.

This instruction adds the data from the memory location whose address is stored in R1 with the contents of R0 Register and the ~~register~~ result is stored in R0 register as shown in the fig.



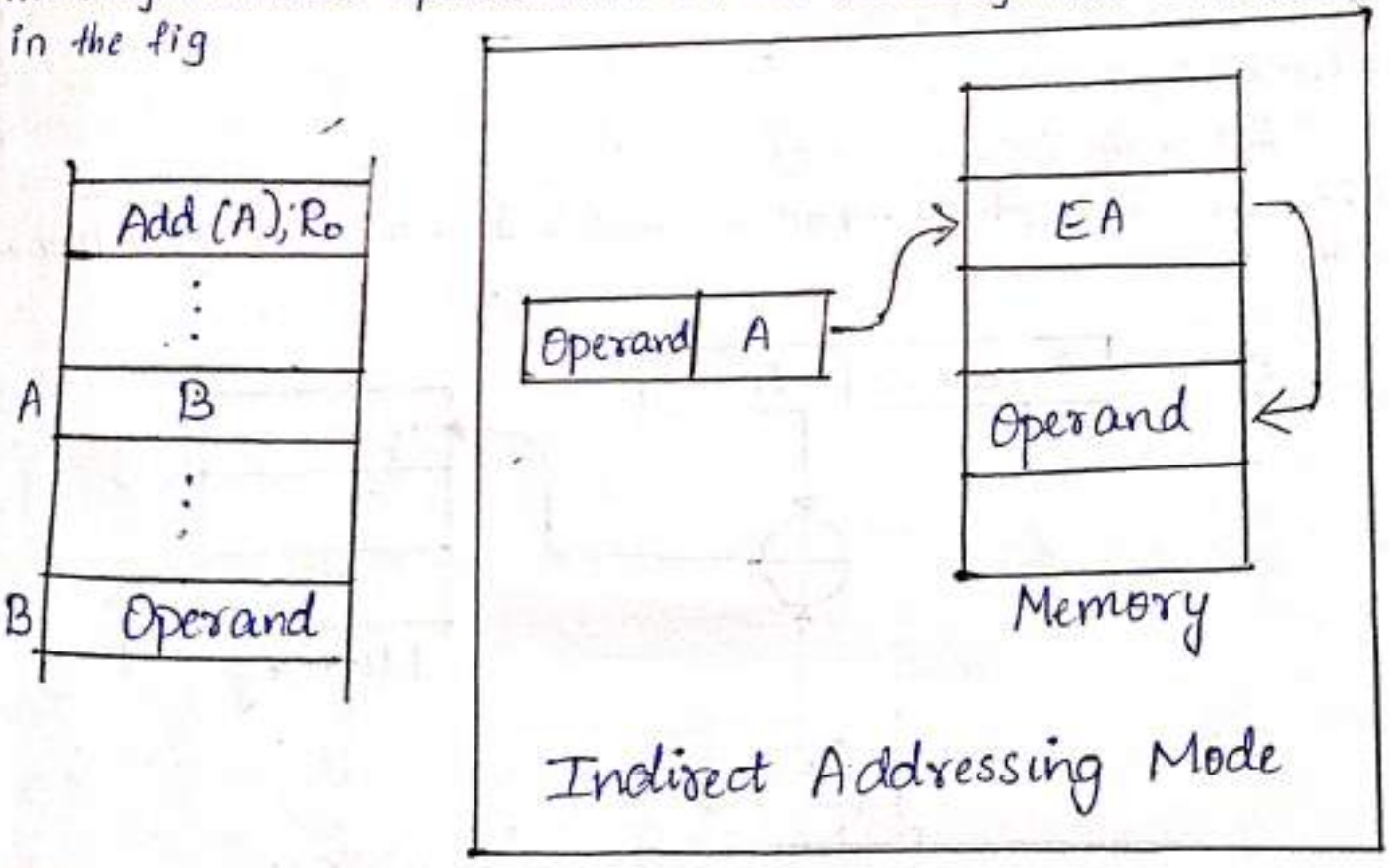
ii) Indirect Addressing Mode thraugh Memory Location :-

In this Addressing Mode, effective address of an operand is stored in the memory location.

Ex:- ADD (X), R0

This instruction adds the data from the memory location whose address is stored in 'X' memory location with the contents of R0 and result is stored in R0 register.

The diagrammatic representation of this addressing mode is as shown in the fig



e. Index Addressing Mode :-

In this addressing mode, the effective address of an operands is computed by adding constant value with the contents of index register and any one of the general purpose ~~regsi~~ register namely R_0 to R_{n-1}. Can be used as the index register. The constant value is directly specified in the instruction.

The Symbolic representation of this mode are as follows

1. $X(R_i)$ Where X is constant value and R_j is the GPR
   It can be represented as EA of an operand = $X+(R_i)$
2. $X(R_i, R_j)$ Where X is the constant value and R_I and R_J are the general purpose registers used to store the addresses of the operands. It can be represented as
   The EA of an operand is given by $EA = (R_i) + (R_j) + X$

f. Relative addressing Mode

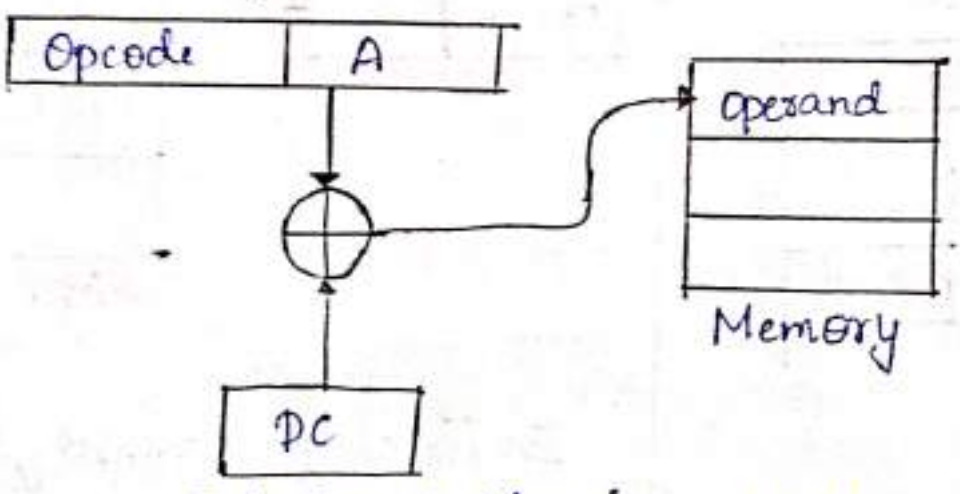
In this addressing mode EA of an operand is computed by the index addressing mode. This addressing mode uses PC [program counter] to store the EA of the next instruction instead of CIPR.

The Symbolic representation of this mode is X [pc]. Where X is offest value and PC is the program counter to store the address of the next instruction to be executed

It can be represented as

EA of the operand = $X + (pc)$

This addressing mode is useful to calculate the EA of the target memory location



Relative Addressing Mode

g. Auto Increment Addressing Mode:-

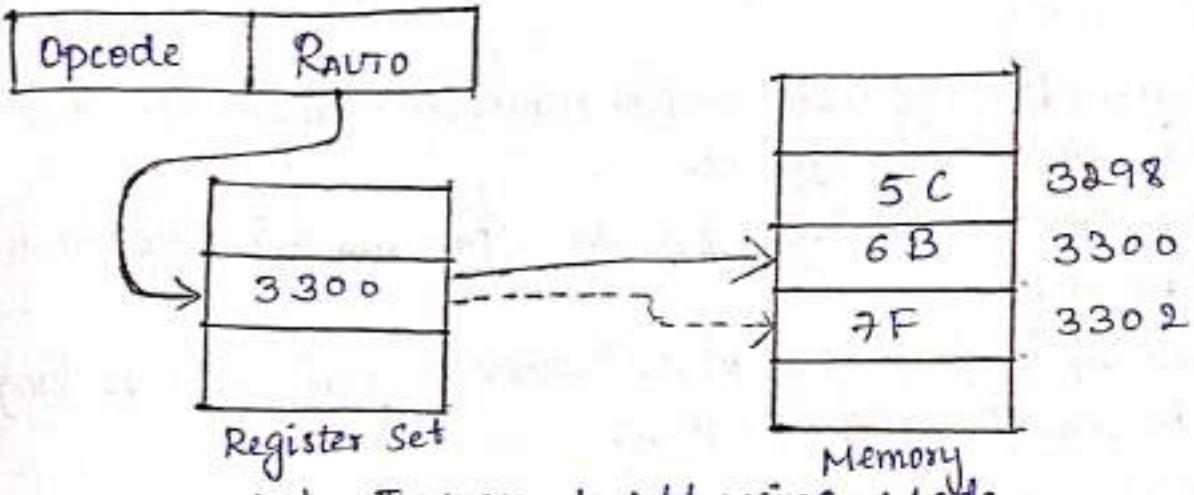
In this Addressing mode, EA is of an operand is stored in the one of the GPR's of the CPU. This Addressing mode increment the contents of the memory register by 4 memory locations after operand access.

The Symbolic representation is

(RI)+ , Where RI is one of the GPR.

Ex: MOVE $(R_1)+, R_2$

This instruction transfers data from the memory location whose address is stored in R_1 into R_3 register and then it increments the contents of R_1 by 4 memory location.



Auto-Increment Addressing Mode.

h. Auto Decrement Addressing Mode:-

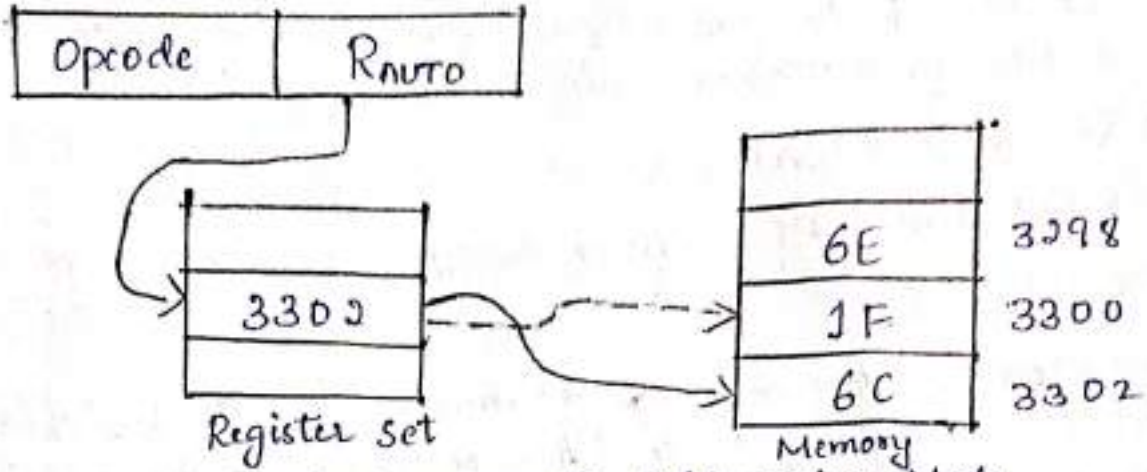
In this Addressing mode, EA of an operand is stored in the one of the GPR's of the CPU. The Addressing mode decrement the contents of the memory register by 4 memory locations and then transfer the data to destination.

The Symbolic representation is

-(RI) Where Ri is the one of the GPR.

Ex: MOVE $-(R_1), R_2$

This instruction first decrements the contents of R_1 by 4 memory locations and then transfers data of that location to destination register.



Auto-Decrement Addressing Mode

Assembly Language

*The Assembly language uses symbolic names to represent opcodes, memory locations and registers.

*The Assembler convert Assembly language program into machine level language programs.

*The Assembly language is called as "source program" and m/c language Program is called As "object program".

The Assembler converts source program into object program.
A set of Symbolic names and set of rules of their use forms a progr-amming language and is called as "Assembly Language"

Ex:- MOV, ADD, LOAD → opcodes.
X, Y, AMOUNT → Memory locations.

Where R_0 to R_7 are the registers.

The examples for Assembly language instructions are as follows
MOV R_0, R_1 : Register Addressing
MOV #23, R_0 : Immediate Addressing

Assembler Directives

There are two types of instructions namely i) processor Instructions & ii) Assembler Instructions

* The processor instruction are converted into m/c instructions by means of Assembler. Hence Assembler generate m/c code for processor instructions
* The Assembler instructions are not m/c intructions & hence Assembler does not generate the m/c code for Assembler instructions

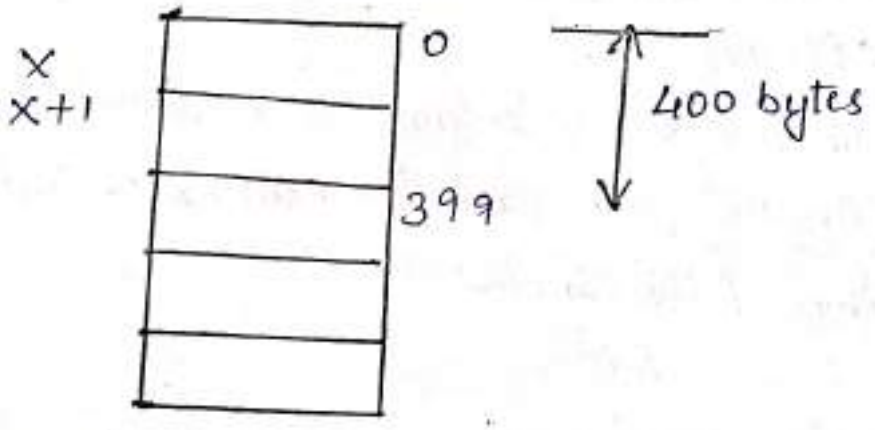
"A set of commands given to Assembler While converting source program into object program is called as Assembler directive".

Types of Assembler directives:

1. Reserve:- This directive is used to allocate a block of memory. This block of memory is used only for data.

```
X   RESERVE   400
```

This directive reserve 400bytes of memory whose Symbolic name is X as shown in fig.



2. EQU:- This directive is used to assign numerical values to Symbolic names during the execution of the assembly languages program

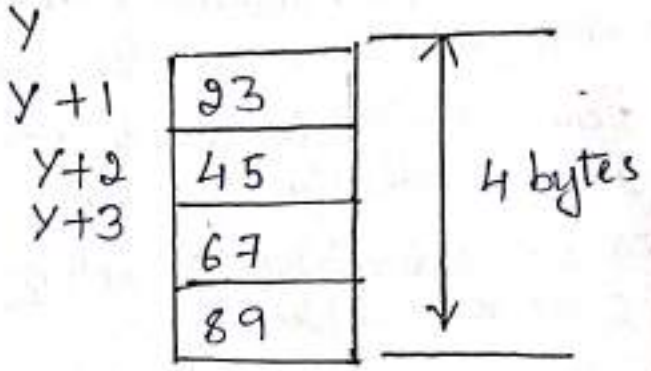
The following code describes the use of EQU directive

```
X       EQU     32
MOVE    #23     R0
MOVE    X,      R1
ADD     R0      R1
```

3. DATA WORD:- This directive is used to allocate by 4 bytes of memory

Y DATA WORD 23456789

The memory representation of this directive is as shown in the fig.



4. ORIGN :- This directive converts source program into object Program is loaded into memory for execution by means of loader. The origin directive is used to assign the successive addresses for sequence of instructions or operands.

ORG 100

5. END:- This directive is used to terminate the assembly level language program. The Assembler ignore the execution of instructions after the execution of the directive

END

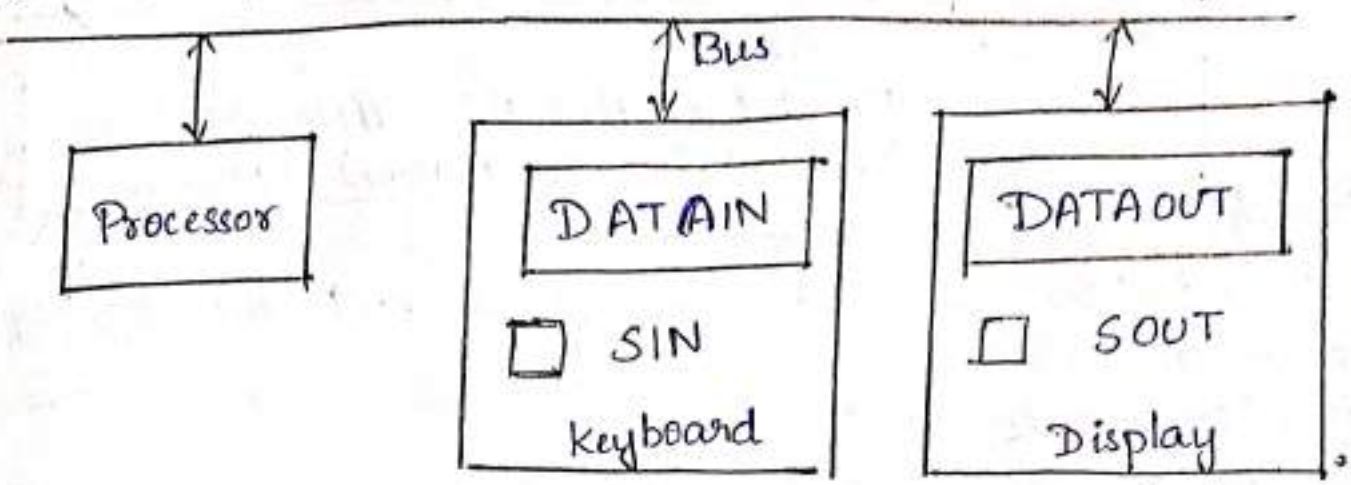
Basic Input and output operations:-

The simple arrangement of conneding i/p and o/p devices into the Processor is as shown in the Fig. The processor performs two operations with respect to i/o device namely

i. Input operation
ii. output operation

Input operation: It is the process of reading the data or instructions from the input device. The I/O subsystem consist of block of instructions to perform i/p operation.

output operation: It is the process of writing the data or instruction into the output device. The I/O subsystem consist of block of instruction to perform o/p operation.



consider a problem of transferring 1 byte of data from i/p device to o/p device. The i/p device transfer few characters/sec. The data transfer rate of i/p device is expressed in terms of few characters/sec. Similarly o/p device transfers thousands of characters to o/p device for display. The processor is capable of executing millions of instructions per second. From the above analysis it is clear that the processing and transfer speed varies in different device. So the devices must be synchronized.

To provide the synchronization between processor, i/p device and o/p device it is necessary to follow the several steps are as follow

Steps to provide Synchronization between processor and i/p device

1. When a character is pressed on the keyboard, the ASCII value of a character is stored in DATAIN register and hence SIN flag is set to 1.
2. When SIN=1, the processor reads the ASCII value of a character from DATAIN register into the processor register.
3. After reading, the SIN flag is reset to 0.

```
READWAIT  if SIN=0
              Branch to READWAIT   // No data to read
              Input data from DATAIN to R1   // Data is read
```

Steps to provide Synchronization b/w processor & o/p device

1. When the o/p device is ready to display the character, the processor transfer the character code from the processor register into DATAOUT register.
2. The SOUT Flag is set to 1 When DATAOUT register holds the character code.
3. The SOUT flag is cleared When the character code is transferred to o/p device.

```
WRITEWAIT  if SOUT=0
           Branch to WRITEWAIT  // No data to read
           Input data from R1 to DATAIN  // Data is
```

The i/p operation can be implemented as follows

Let R0 be the processor register and DATAIN be the interal register of the i/p device

```
MOVE DATAIN, R0
```

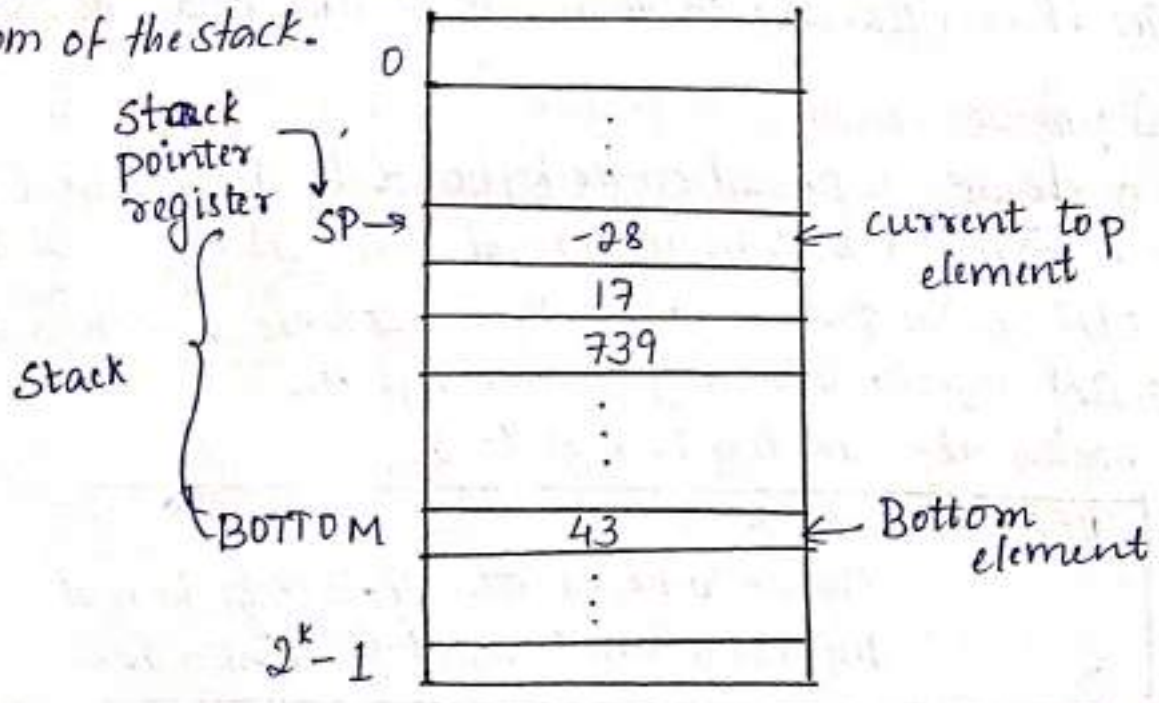
The o/p operation can be implemented as follows

Let R0 be the processor registes and DATAOUT be the internal register of the o/p device.

```
MOVE R0, DATAOUT
```

Stacks & Queues

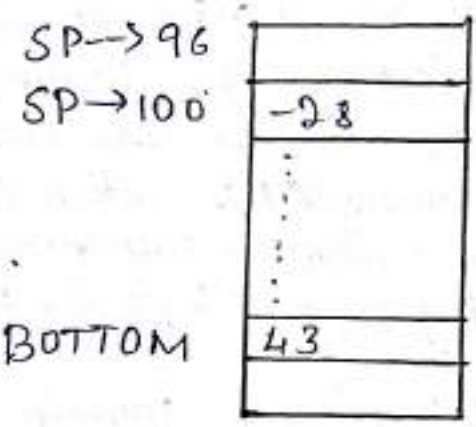
STACK: A stack is a Data structure, in which the accessing is restricted at only one and of the stack. It is similar to a bottle, in which elements can be added and removed from the same end. The end of the stack, from which elements can be added or removed is called the top of the stack and the other end is called the bottom of the stack.



It works on the principle of LIFO[Last in first out]. the last item placed on the stack is the first to be removed. The term 'push' and 'pop' are used to describe the placing a new item on stack and removing the top item from the stack.
Assume that the first element is placed in the location BOTTOM, and when they are placed in successive lower addresses.
A processor register is used to keep track of the address of the element that is the top of stack at any time. This register is called stack pointer (SP)
In the above figure, the SP is currently pointing to the topmost value -28. To add a new element, the SP will decrement its value by 1 address. so as to point at next location and add the new value.

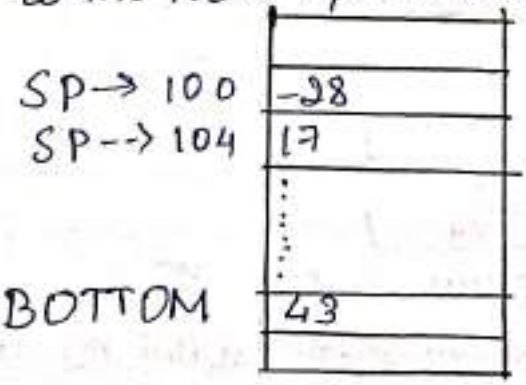
PUSH operation – Subtract #4, SP
MOV NEWITEM, (SP)

The Subtract instruction subtracts the SP value by 4, now SP pointer to the next lower address. The MOV instruction moves the new element to the address location stored in SP.



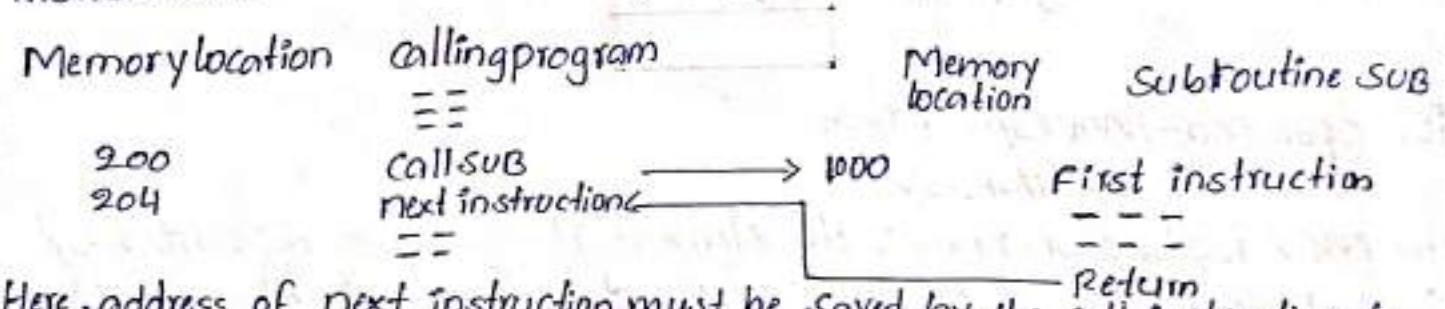
POP operation – MOV (SP), ITEM
ADD #4, SP

The MOV instruction moves the element at the location pointed by SP to ITEM & the SP pointer is moved to the next higher address so that it points to the new top element.



Queue:- A queue is a data structure that works on the principal of FIFO[First In First Out] ie.. data that are stored first are retrived first on FIFO basis.
The elements are added at one end and retrieved from Other end. In stack, one end is fixed Where as in queue both ends are pointed by Pointers & both end changes its location. one end is used to add items and other end is used to delete items.

Subroutine:- Sub function in a program necessary to perform a particular subtask are called a Subroutine.
Eg: Subroutine to sort a list of numbers, Subroutine to add the given number etc.
In a program, the subroutines can be called from different locations and different functions. When a program branches to a Subroutine, then it is calling the Subroutine. The instruction that perform this branch operation is called call instruction.
Whenever the subroutine is called, the execution starts from the starts from the starting address of Subroutine. After its execution, the execution of calling function is resumed from the location Where it Subroutine. After its execution, the execution of calling function is resumed from the location Where it called the Subroutine. Hence the content of PC is stored before moving to the Subroutine.
The way in Which a Computer calls & returns from a Subroutine are called Subroutine linkage method. The return address is stored in link registes. After the execution of subroutine, the return instruction returns to the calling program by using the link register.



Here, address of next instruction must be saved by the call instruction to enable returning to calling program

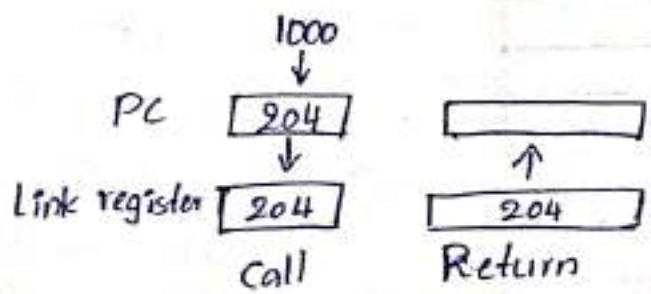


Figure Subroutine Linkage using a link register

The call instruction is a Special branch instruction -
* It stores the content of pc in link register
* stores the specified Subroutine address in pc and branch to that address.

The return instruction of Subroutine is a Special branch instruction -
* The branches to the address contained in link register.

Parameter passing : Example for Subroutine

Calling program	MOVE N, R_1
	MOVE #NUM$_1$, R_2
	CALL List ADD
	MOVE R0, Sum
	...

Subroutine	
LIST ADD	CLEAR 0 ADD (R_2)+, R_0 Decrement R_1 Branch>0 Loop Return

*passing by value :- The actual values are passed to the Subroutine. The original values remain unchanged outside the Subroutine.

*passing by reference :- The address of variables are passed. The Subroutine can modify the original variables directly.

Parameter passing example such that, Let's assume we have a Subroutine that adds two numbers. Let's

1. Register :- * R_1 & R_2 : Hold parameters or intermediate values.
* R_3 : will hold the result.

2. Instruction :- * MOV : Move value from one place to another
* ADD : Add values.
* DEC : Decrement value.

Case1 : passing parameters by value : In this case, the actual value directly to the Subroutine

```
Move 5, R1 /// R1 = 5
Move 10, R2 /// R2 = 10
Call-Add-Subroutine
Add-Subroutine : Add R2, R1 //Add R1 & R2, result in R2
MOV R2, R3 ///Move result to R3
RET  /// Return from Subroutine
```

*After execution, R_3 will hold 15 (i.e, 5+10).

Case2: Passing parameters by Reference

```
MOVE 5, R1 ; R1 = 5
MOVE 10, R2 ; R2 = 10

MOVE ADD-VAR1, R3 : Address of R1
MOVE ADDR-VAR2, R4 ; Address of R2

CALL ADD_BY_REFERENCE

ADD_BY_REFERENCE:
MOV R3 ,[R5]  ; Load value from address in R3 (R1) to R5
MOV R4 ,[R6]  ; Load value from address in R4 (R2) to R6
ADD R5, R6    ; Add R5 & R6, result in R6
MOV R6, [R3]  ; Store result back at address in R3 (R1)
RET           ; Return the Subroutine
```

* After execution, R_1 will be modified to 15 (i.e, 5+10), and R_2 remain 10 (unchanged):

Multiplication and Division:-

Format: opcode Source operand, destination operand

Ex:- Multiply R_i, R_j ; $R_i = [R_i] * [R_j]$

Multiply R_3, R_0

Perform the operation as $R_0 = [R_3] * [R_0]$

Ex:- Divide R_i, R_j ; $R_i = [R_j] / [R_j]$

Divide R_3, R_0

Perform the operation as $R_0 = [R_0] / [R_3]$

Addition Instruction:-

They are 3 types

a) Logical instruction - NOT, AND, OR

b) shift instruction - Logical shift left, Logical shift right

c) Rotate instruction - Rotate left with carry, Rotate left without carry, Rotate Right with carry, Rotate Right without carry.

Logical instructions :- The processors are designed to perform logical operator such as AND, OR & NOT operations.

i) NOT instruction

Format: opcode destination

The opcode specifies operation to be performed and destination specifies the operand. The operand can be registes operand

Ex: NOT R_0

It performs the function of complementation. It is the process of converting binary bit 1 & vice verse

The illustration of this instruction is as follows

Before instruction execution	After instruction execution
R_0 = 10101100	01010011

ii) AND instruction :- It perfroms the function of logical AND operation

Format: opcode Source, destination

Ex:- AND R_3, R_0

This instruction logically ANDs the contents of R_3 with the contents of R_0 & result is stored in R_0 register.

iii) OR instruction :- It performs the function of logical OR operation.

Format: opcode source, destination

Ex: OR R_3, R_0

This instruction logically OR^s the contents of R_3 with the contents of R_0 & result is stored in R_0 registes.

Shift instructions:- The shift instruction are designed to the shift the contents of processor register or memory location of Left or right according to the number of bits Specified in the count

a. Logical shift Left:-

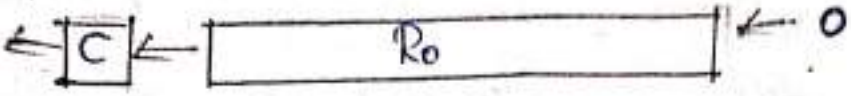
Format: opcode count, destination

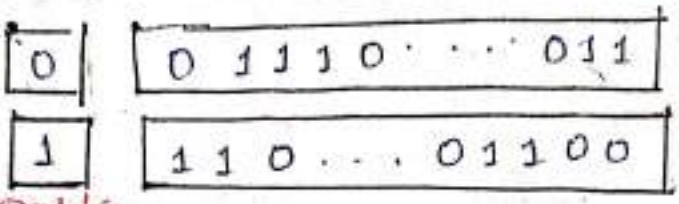
The opcode indicates operation to be performed. The count can be either immediate operand or the content of processor registes, The destination can be either registes operand

Ex: Lshift #2, R_0

This instruction shift the contents of register R_0 to left through Carry by 2 bits. The count value directly specified in the instruction as an immediate operand.

The contents of R_0 before & after the execution of this instruction are as shown in the fig. The shifted positions are filled with Zeros from right side as shown in the fig





Logical shift Right:

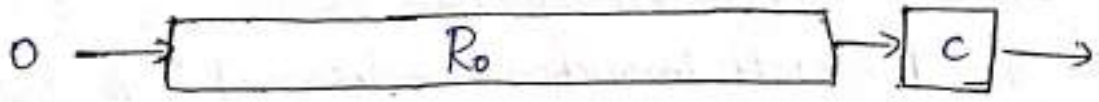
Format: opcode count, destination

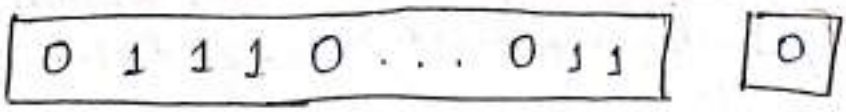
The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand.

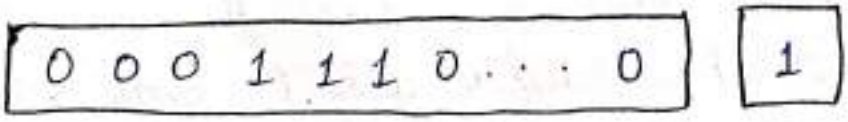
Ex: Lshift #2, R_0

The instruction shifts the contents of register R_0 to right through carry by 2 bits. The count value directly specified in the instruction as an immediate operand.

The content of R_0 before and after the execution of this instruction are as shown in fig. The shifted positions are filled with Zeros from left side as shown in the fig.







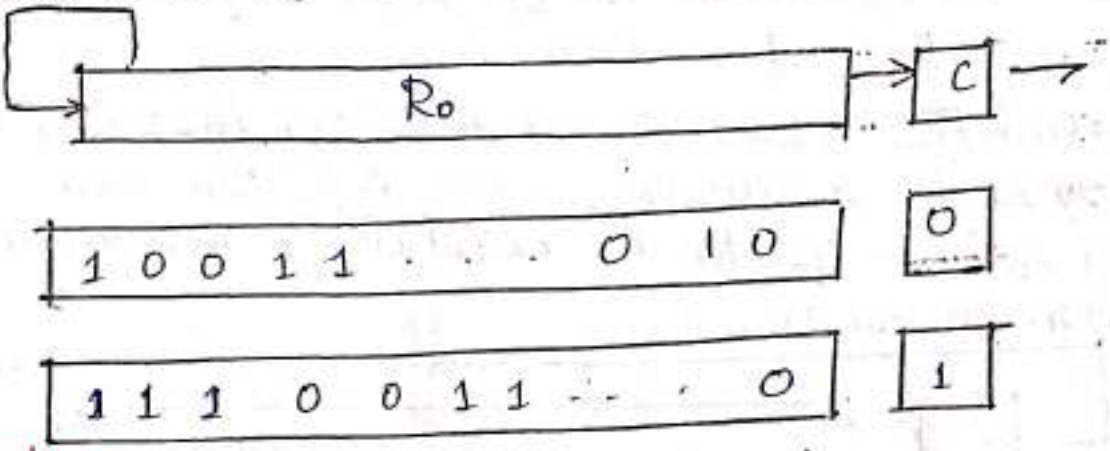
Arithmetic Shift Right:

Format: opcode count, destination

The opcode indicates operation to be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand

Ex:- AshiftR #2, R0

This instruction is designed to preserve the sign bit. This instruction shifts the contents of register or memory location to right through carry, by number of bits specified in the count. After each shift it copies leftmost bit to most significant bit. The contents of R0 before & after the execution of this instruction are as shown in the fig



Rotate instructions:- The rotate instructions are designed to rotate the contents of register or memory location to left or right according to the number of bits specified in the count.

There are 4 types with respect to them.

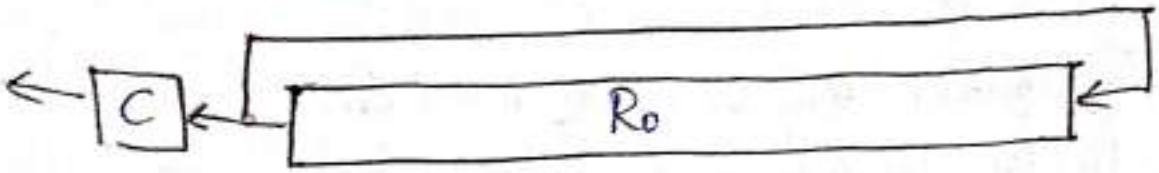
a. Rotate left without carry:-

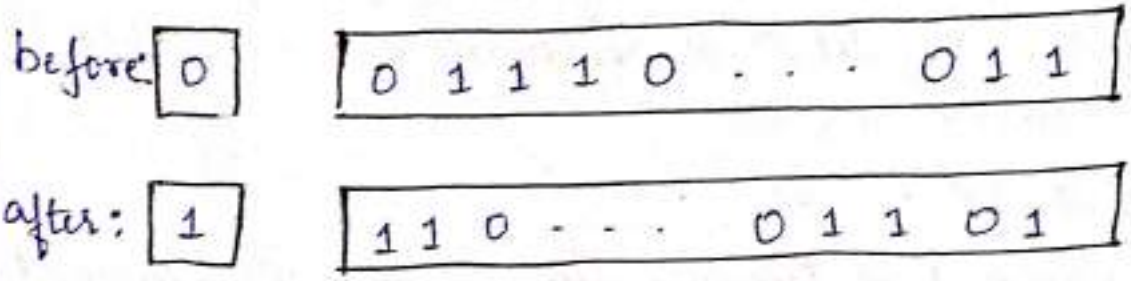
Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the content of processor register. The destination can be either register operand

Ex: RotateL #2, R0

This instruction rotates the contents of register R0 to left without carry by 2 bits as shown in the fig. The most significant bits are transferred to least significant bits are as shown in fig. The contents of register R0 before & after the execution of this





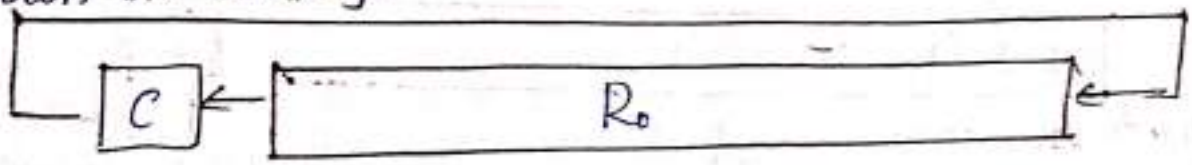
Rotate left with carry:-

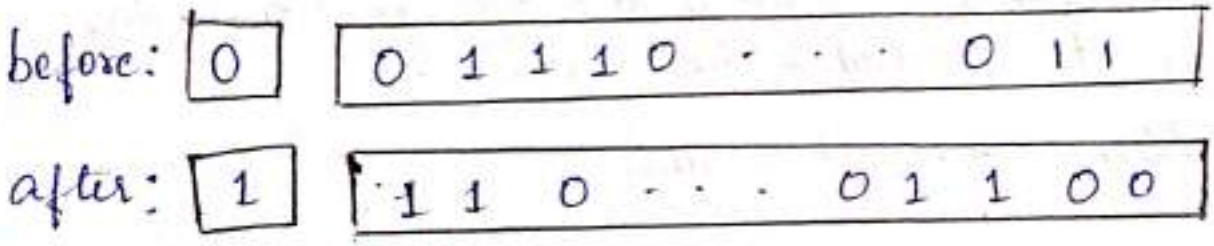
format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand

Ex: Rotate LC#2, R_0

This instruction rotates the contents of register R_0 to left with carry by 2 bits as shown in the fig. The contents of register R_0 before & after the execution of this instruction is as shown in the fig.





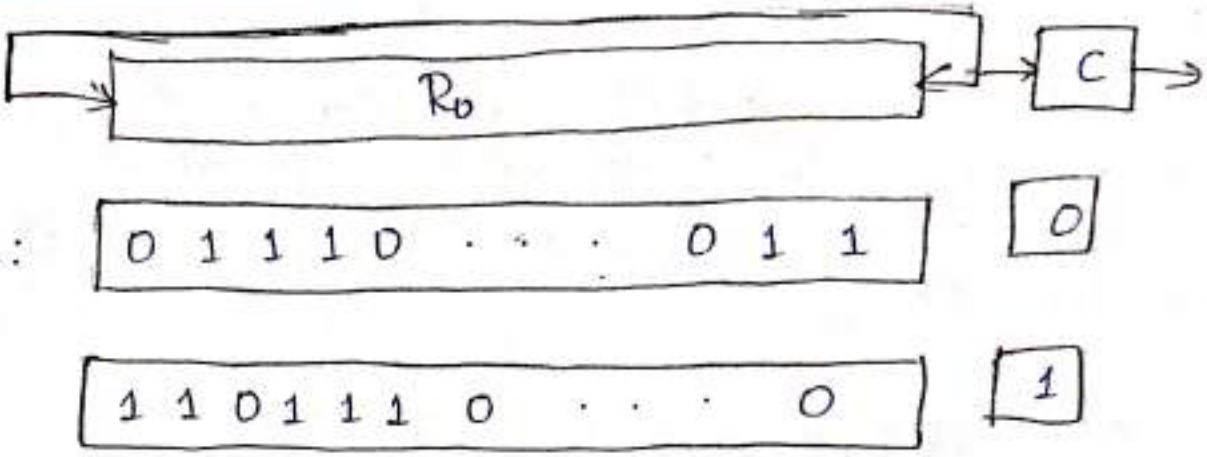
Rotate right without carry:-

Format: opcode count, destination.

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand

Ex: RotdeR#2, R_0

The instruction rotates the content of register R_0 to right without carry by 2 bits as shown in the fig. The least significant bits are transferred to most significant bits are also shown in fig.



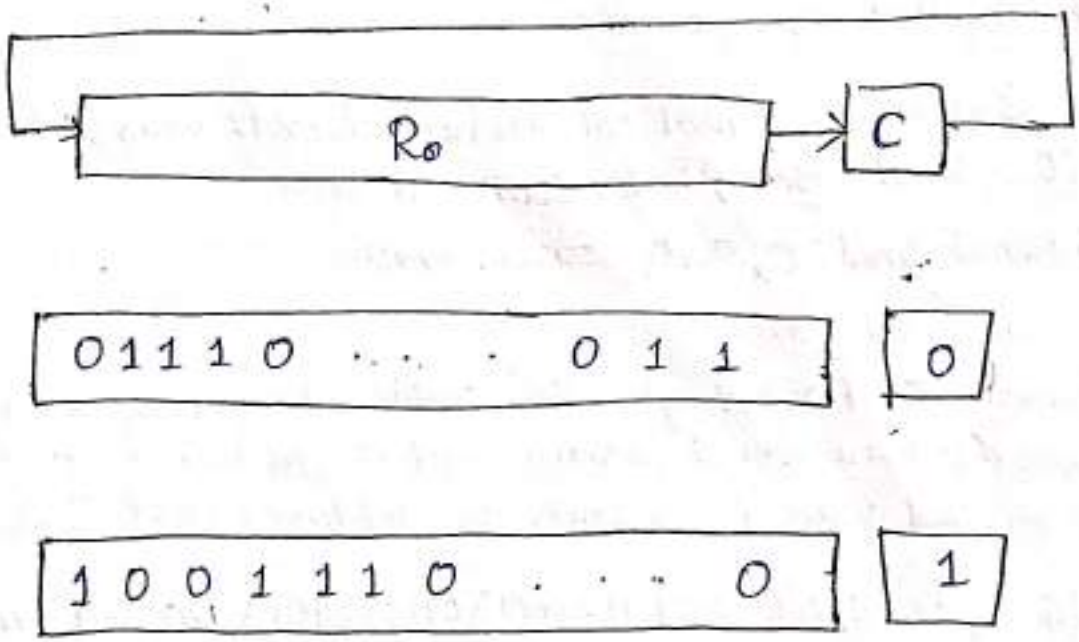
Rotate right with carry:-

Format: opcode count, destination

The opcode indicates operation to be performed. The count can be either immediate operand or the contents of processor register. The destination can be either register operand or memory operand.

Ex: Rotate R0 #2, R0

This instruction rotates the contents of register R0 to right with carry by 2 bits as shown in the fig. The least significant bits are transferred to carry & then transferred to most significant bits are also shown in fig. The content of registes R0 before & after the execution of this instruction are as shown in the fig.



Problem :-

Consider a register R_1 to size 16-bits with initial data 5876d. With neat diagram, depict the output in each case after performing the following operations. i) Lshift L#2, R_1 ii) Ashift R#1, R_1 iii) Rotate R#1, R_1

Convert $5876_{10}5876_{10}$ to Binary;

$5876_{10} = 0001\ 0110\ 0101\ 1100$ $5876_{10} = 0001\backslash 0110\backslash 0101\backslash 1100$

$5876_{10} = 0001\ 0110\ 0101\ 1100_2$

Performing the operations:

i) Left shift Logical (Lshift L#2, R_1);

The left shift logical operation shifts all bits to the left by the specified number of position. The vacant positions on the right are filled with 0s.

Original binary :- 0001 0110 0101 11000001\0110\0101\11000001 0110 0101 1100

After shifting left by 2 position:-

Shifting Binary: 0101 1001 0111 0000 \text{Shifted Binary} 0101\1001\0111\0000 Shifted Binary: 0101 1001 0111 0000

Result in Hexadecimal: $0x5970_0x5970_0x5970$

ii) Arithmetic shift Right:-

The arithmetic shift right operation shifts all the bits to the right by the specified number of position but it preserves the sign bit. Since our original value is positive, the leftmost bit is 0.

Original binary: 0001 0110 0101 11000001\0110\0101\11000001 0110 0101 1100

After shifting right by 1 position;

Shifted binary : 0000 1001 1010 1110 \text{shifted binary:} 0000\1001\1010\1110 ~~Rotated~~ 110 Shifted Binary: 0000 1001 1010 1110

Result in hexadecimal: $0x09AE_0x09AE_0x09AE$

iii) Rotate Right (Rotate R#1, R1):

The rotate right operation shifts all bits to the right by the specified number of position, but the bits that fall off the end are wrapped around to the start.

Original binary: 0001 0110 0101 1100\0001\0110\0101\1100 0001 0110 0101 1100

After rotating right by 1 position:

Rotated binary: 0000 1001 1010 1110\text{ Rotated binary: } 0000\1001\1010\1110 Rotated binary: 0000 1001 1010 1110

Result in hexadecimal: 0x09AE0x09AE0x09AE

*Summary with diagrams:-

Let's summarize the operations with diagrams:

Original value:

Binary: 0001 0110 0101 1100\text{ Binary: } 0001\0110\0101\1100 Binary: 0001 0110 0101 1100 Hex: 0x5876\text{ Hex: } 0x5876 Hex: 0x5876

i) LshiftL#2:

Binary: 0101 1001 0111 0000\text{ Binary: } 0101\1001\0111\0000 Binary: 0101 1001 0111 000. Hex: 0x5970\text{ Hex: } 0x5970 Hex: 0x5970

ii) AshiftR#1:

Binary: 0000 1001 1010 1110\text{ Binary: } 0000\1001\1010\1110 binary: 0000 1001 1010 1110. Hex: 0x09AE\text{ Hex: } 0x09AE Hex: 0x09AE

iii) RotateR#1:

Binary: 0000 1001 1010 1110\text{ Binary: } 0000\1001\1010\1110 binary: 0000 1001 1010 1110. Hex: 0x09AE\text{ Hex: } 0x09E Hex: 0x09AE

Diagram Representation:

* Original binary: 0001 0110 0101 1100
* LshiftL #2 : 0101 1001 0111 0000
* AshiftR#1: 0000 1001 1010 1110
* RotateR#1 : 0000 1001 1010 1110

Problem :- Consider a database of marks scored by #Students in 3 test stored in memory starting at address List. each student record consis-t of student ID followed by marks in 3 tests. Assume each of these to the 4 bytes in size There are 50 students in class & their value is stored at location NUM i) Sketch the memory map showing all details, ii) develop an ALP using indexed addressing mode to compute the sum of score by all the student in Test 2 & store the results in location SUM. write appropriate comment

Each student's record consist of:

* 4 bytes for student ID
* 4 bytes for Test 1 mark
* 4 bytes for Test 2 mark
* 4 bytes for Test 3 mark

* There are 50 students, so that the total number of record is 50.
* Each record is 16 bytes
* The total memory required for all student is 50 students x 16 bytes/student = 800 bytes.
* Location List is the starting address of the student records
* Location NUM holds the number of students, which is 50.
* Location SUM will be used to store the Sum of the scores in Test2 for all students

Memory Map:

Memory address	content
LIST	Student Record 0 (ID, Test1, Test2, Test3)
LIST+16	Student Record1 (ID, Test1, Test2, Test3)...
.....	
LIST+784	Student Record A9 (ID, Test1, Test2, Test3)
NUM	50
SUM	0 (intialSum)

Program:

```
MOVE NUM, R1
MOVE # LIST, R2
Clear R0
BACK: ADD (R2), R1
ADD #16, R2
DECREMENT R1
BRANCH TO BACK
MOV R0, SUM
END
```