UNIT — II

Machine Instruction and Programs

Instruction 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 program equation. component of Instructions: Logic Instructions shift and Rotate Instructions.

Instruction and Instruction sequencing:

→ The tasks carried out by a computer program consists of a sequence of small steps called instructions.

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

1. Data transfers between the memory and the processor registers.
2. Arithmetic and logic operations on data
3. Program sequencing and control
4. I/O transfers.

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.

→ Generally memory location names identified by symbolic names for its hardware binary address.

Example:

memory locations -> LOC, PLACE, A, VAR2

Processor registers -> R0, R3, R5

I/O registers -> DATAIN, OUTSTATUS.

-> the contents of a location are denoted by placing square brackets around the name of the location. thus, the expression

$$R1 \leftarrow [LOC]$$

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

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

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

-> this type of notation is known as Register Transfer Notation (RTN).

Assembly Language Notation:

-> ALN represents machine instructions and programs.

(1) Move LOC, R1

An instruction that causes transfer data from memory location LOC to processor register R1.

(2) Add R1, R2, R3

Adding two numbers contained in processor registers R1 & R2 and placing their sum in R3.

Basic Instruction Types / Instruction Formats:

1. three address Instructions
2. Two address Instructions
3. one address Instructions
4. Zero address Instructions
5. RISC Instructions.

① three-Address Instructions:

→ Each Instruction have three operand addresses.

→ A General Instruction format is

operation source1, source2, destination

Here A, B and C are memory addresses

Ex: Add A, B, C ; $C \leftarrow [A] + [B]$

→ operands A & B are called source operands, C is called destination operand.

② Two-address Instructions:

→ Each Instruction have only two operand addresses.

→ Instruction format is

operation source, destination

Ex: ① Add A, B ; $B \leftarrow [A] + [B]$

which performs the operation $B \leftarrow [A] + [B]$. when sum is calculated, the result is sent to the memory and stored in location B, replacing the previous data of this location. This means that operand B is both source and destination.

② Move B, C ; C ← [B]

This instruction copy contents of B to C leaving the contents of location B unchanged.

③ One-address Instruction:

→ Each instruction have only one memory operand.
→ when a second operand is needed, it is understood implicitly to be in accumulator (a processor register)
→ operand specified in the instruction may be a source (or) destination depending on the Instruction.

Ex: ① Add A ; AC ← [A] + [AC]

Add the contents of memory location A to the contents of Accumulator [AC] register and place the sum back into the Accumulator

② Load A ; AC ← [A]

copies the contents of memory location A into the accumulator.

③ store A ; A ← [AC]

copies the contents of accumulator into memory location A.

④ Zero-Address Instruction:

→ This type of Instructions are found in machine that store operands in stack.
→ Push and Pop instructions require one address field.

Ex: Add ; TOS ← [TOS] + [TOS-1]
Push A ; TOS ← [A]
Pop ; A ← [TOS]

TOS = Top of stack

(5) RISC Instruction:

→ In this memory restricted to LOAD & STORE Instructions.

→ All manipulation of data are allowed ~~only~~ operands that are in processor registors.

Ex: Add R_i, R_j ; $R_j \leftarrow [R_i] + [R_j]$

LOAD A, R_i ; $R_i \leftarrow [A]$

store R_i, A ; $A \leftarrow [R_i]$

Additional Instructions are

subtract R_i, R_j ; $R_j \leftarrow [R_j] - [R_i]$

multiply R_i, R_j ; $R_j \leftarrow [R_i] * [R_j]$

divide R_i, R_j ; $R_j \leftarrow [R_j] / [R_i]$

① perform the operation $[C = A+B]$

Ans: 1) Three address Instructions ; Add A,B,C ; $C \leftarrow [A] + [B]$

2) Two address Instructions ; move B,C ; $C \leftarrow [B]$

Add A,C ; $C \leftarrow [A] + [C]$

3) one-address Instruction: Load A ; $AC \leftarrow [A]$

Add B ; $AC \leftarrow [B] + [AC]$

store C ; $C \leftarrow [AC]$

④ Zero address Instruction: PUSH A ; $TOS \leftarrow [A]$

PUSH B ; $TOS \leftarrow [B]$

Add ; $TOS \leftarrow [TOS] + [TOS-1]$

$TOS \leftarrow [A] + [B]$

POP C ; $C \leftarrow [TOS]$

⑤ RISC Instruction: Load A, R_i ; $R_i \leftarrow [A]$

Load B, R_j ; $R_j \leftarrow [B]$

Add R_i, R_j, R_K ; $R_K \leftarrow [R_i] + [R_j]$

store R_K, C ; $C \leftarrow [R_K]$

Addressing modes:

→ In general, a program operates on data that reside in the computer's memory. these data can be organized in a variety of ways.

→ the different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

~~→ Programs programmer~~

→ programs are normally written in high level language, which enables the programer to use constants, local & global variables, pointers and arrays.

Types: ① Immediate addressing mode ② Register

③ Absolute ④ Indirect

⑤ Index ⑥ Base with index

⑦ Base with index & offset ⑧ Relative

⑨ Auto Increment ⑩ Auto decrement.

→ variables and constans are the simplest data types and are found in almost every computer program. In assembly language a variable is represented by allocating a register (or) a memory

location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Register mode: the operand is the contents of a processor register the name of the register is given in the instruction.

→ processor registers are used as temporary storage locations where the data in the register are accessed using Register mode

Ex: Add R_1, R_2 ; $R_2 \leftarrow [R_1] + [R_2]$

Absolute mode: The operand is in a memory location; the address of this location is given explicitly in the instruction. this mode is also called as direct addressing mode.

→ the Absolute mode can represent global variables in a program.

Ex:- move B, C ; $C \leftarrow [B]$

Note: move A, R_1 ; $R_1 \leftarrow [A]$

the instruction uses these two modes. [Register & Absolute Addressing modes].

Immediate mode: the operand is given explicitly in the Instruction.

→ the Immediate mode is only used to specify the value of a source operand. A common convention is to use the sharp sign [#] infront of the value to indicate an Immediate operand.

Ex: Move #200, R0

→ The value 200 is placed in R0.

→ Addres and data constants can be represented in Assembly language using the Immediate mode.

Ex: A = B+6

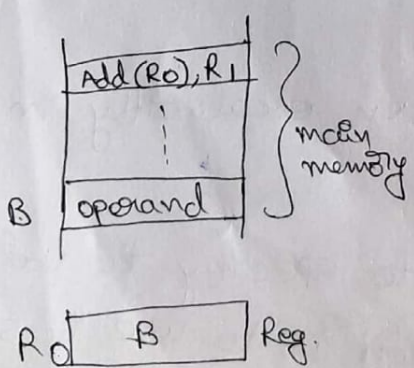
```
Move  B, R1
Add   #6, R1
move  R1, A
```

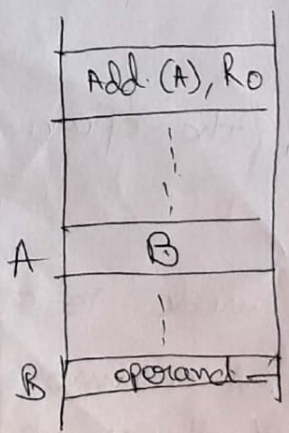
Indirection and pointers:

Indirect mode: the effective address of the operand is the contents of a register (or) memory location whose address appears in the instruction.

→ In the addressing mode, the instruction does not give the operand or its address explicitly.

→ the name of the ~~operan~~ register (or) memory address is denoted in parentheses in an instruction.





→ To execute the add instruction in above figure, the processor uses the value B, which is in register R_1, as the effective address of the operand

→ It requests a read operation from the memory to read contents of location B. the value read is the desired operand, which the processor adds to the contents of register R0.

→ Indirect addressing through a memory location is also possible is also shown in figure.

→ In this case, 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.

Ex:

```
        move    N, R1
        move    #NUM1, R2      } Initialization.
        clear   R0
LOOP →  Add     (R2), R0
        Add     #4, R2
        decrement R1
        Branch>0 LOOP
        move    R0, SUM
```

→ This is program for adding a list of numbers. Register R_2 is used as a pointer to the numbers in list and operands are accessed indirectly through R_2.

→ Program loads the counter value n from memory location N into R1 & uses immediate addressing mode to place address of NUM1 into R_2.

→ Then it clears R0 to 0. the first time through the loop, add. instruction fetches the operand at location NUM1 and adds it to R0.

→ The second Add instruction adds 4 to the contents of the pointer R_2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Index Addressing mode;

→ It is useful in dealing with lists and arrays.

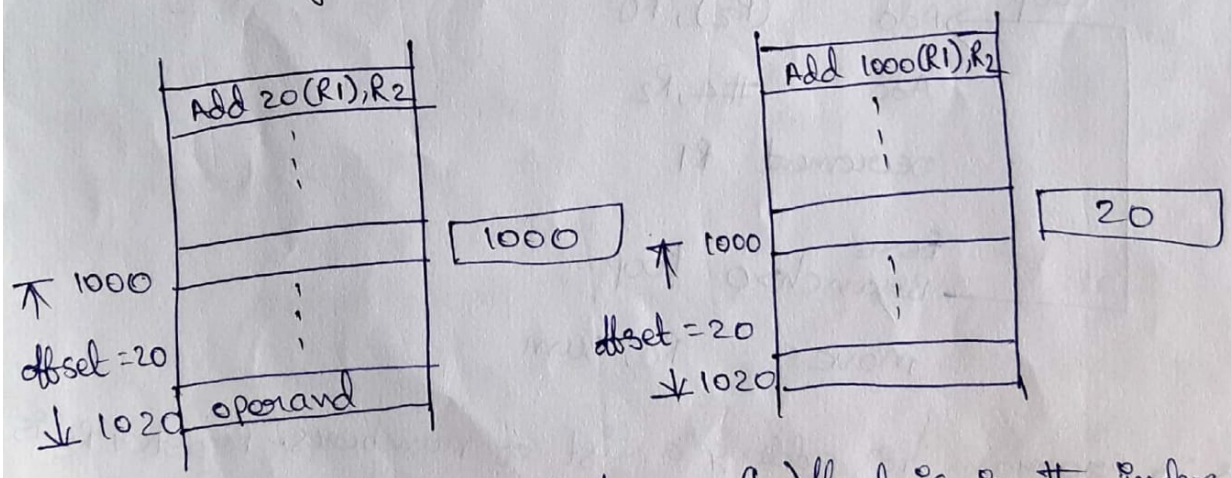
→ the effective address of the operand is generated by adding a constant value to the contents of a register ~~(or) memory location whose address appears in the instruction~~.

→ Index mode symbolically represented as X(Ri) where X denotes the constant value contained in the instruction and Ri is name of register involved.

→ the effective address of the operand is given by

$$EA = X + [Ri]$$

→ Below figure illustrates two ways of using the index mode.



(a) offset is given as a constant (b) offset is in the index register.

→ the index register R1, contains the address of memory location and value X defines an offset from this address to the location

where the operand is found.

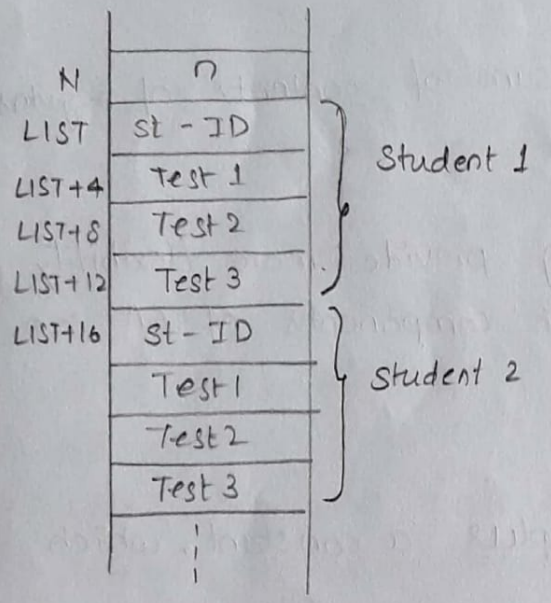
→ The constant X corresponds to a memory address, and contents of index register define offset to operand.

→ Considered a simple example involving a list of test scores for students taking a given course.

→ Each record consists of students Identification no (ID), followed by the scores student earned on three tests.

→ There are n students in the class, and value 'n' is stored in location N immediately infront of the list.

→ Suppose that, wish to compute sum of all scores obtained on each of the tests and store these three sums in memory locations sum1, sum2 & sum3.




```
        move    #LIST, R0
        clear   R1
        clear   R2
        clear   R3
        move    N, R4
Loop    Add     4(R0), R1
        Add     8(R0), R2
        Add     12(R0), R3
        Add     #16, R0
        Decrement R4
        Branch>0 Loop
        Move    R1, Sum1
        Move    R2, Sum2
        Move    R3, Sum3
```

→ R_0 is used as index register, first R_0 is set to point of ID location of 1st student record. Thus, it contains the address list.

→ Test scores of the first student are added to the running sums held in register R_1, R_2 & R_3 which are initially cleared to zero.

→ These scores are accessed using the index addressing modes 4(R_0), 8(R_0) & 12(R_0). The index register R_0 is then incremented by 16 to point to the ID location of the second student.

→ Register R_4, initialized to contain the value n, is decremented by 1 at the end of each pass through loop.
→ When the contents of R_4 reach 0, all student records have been accessed and loop terminates.
→ The last three instructions transfer accumulated sums from registers $R1, R2$ & $R3$ into memory locations sum1, sum2 & sum3 respectively.

Base with Index

→ In this mode, offset X is second register called base register symbolically indicated as

$$(R_i, R_j)$$

→ the effective address is the sum of contents of registers R_i & R_j $\quad EA = [R_i] + [R_j]$

This sum of index addressing provide more flexibility in accessing operands, because both components of EA can be changed.

Base with Index & offset

This mode uses two registers plus a constant, which can be denoted as $X(R_i, R_j)$.
→ the effective address is sum of constants X & contents of register R_i & R_j

$$EA = [R_i] + [R_j] + X$$

Relative Mode

→ It is useful version of Index mode, where PC is used instead of general purpose register (or) index register.
→ Denoted as $X(PC)$. The effective address is determined by index mode using the PC (Program counter) in place of the general purpose register R_i

$$EA = [PC] + X$$

→ The most common use of this mode to specify the target address is branch instructions.

Auto-Increment mode

→ The EA of the operand is contents of register specified in instruction. After accessing operand, the contents of this register automatically incremented to the next item in a list.

→ Auto increment mode is denoted by putting specified register in parantheses, to show that the contents of register are used as EA, followed by a plus sign to indicate that these contents are ~~the~~ to be incremented after operand is accessed.

(Rp)+
EA = [Ri] Increment Ri

→ The increment amount is 1, for Byte sized operands; 2 for 16-bit operand and 4 for 32-bit operand.

Auto-Decrement mode

→ In this mode, the contents of a register specified in instruction are first automatically decremented and then used as effective addres of operand.

→ Auto decrement mode denoted by puting the specified register in paranthesis ~~pred~~ preceded by a minus sign to indicate that the contents of the register are to be decrement-ed before being used as the EA.

-(Ri) Decrement Ri
EA = [Ri]

→ In this mode, operands are accened independing order.

Generic Addressing Modes

Name	Addsersing function	Assembler syntax
1. Immediate	operand = value	# value
2. Register	EA = Ri	Ri

Name	Addressing functions	Assembler syntax
3. Absolute (Direct)	EA = LOC	LOC
4. Indirect (Ri)	EA = [Ri]	(Ri)
(LOC)	EA = [LOC]	(LOC)
5. Index	EA = [Ri] + X	X(Ri)
6. Base with Index	EA = [Ri] + [Rj]	(Ri, Rj)
7. Base with Index & offset	EA = [Ri] + [Rj] + X	X(Ri, Rj)
8. Relative	EA = [PC] + X	X (PC)
9. Auto-Increment	EA = [Ri]; increment Ri	(Ri)+
10. Auto-Decrement	Decrement Ri EA = [Ri]	-(Ri)

Stack and Queue's

→ A computer program often needs to perform a particular subtask using familiar subroutine structure. In order to organize the control & information linkage between the main program & subRoutine, a data structure called a stack is used.

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

→ This end is called top of the stack, and the other end is called bottom of stack. The structure is sometimes referred as Pushdown Stack and Last-In-First-out [LIFO] Stack.

→ LIFO means the last data item placed on the stack is first on removed when retrieval begins.

→ The terms PUSH & POP are used to describe placing a new item on the stack & removing top item from stack respectively.

→ Below figure shows a stack of word data items in the memory of computer

→ A processor register is used to keep track of address of the elements of stack that is at the top at any given time. This register is called 'Stack pointer' (SP).

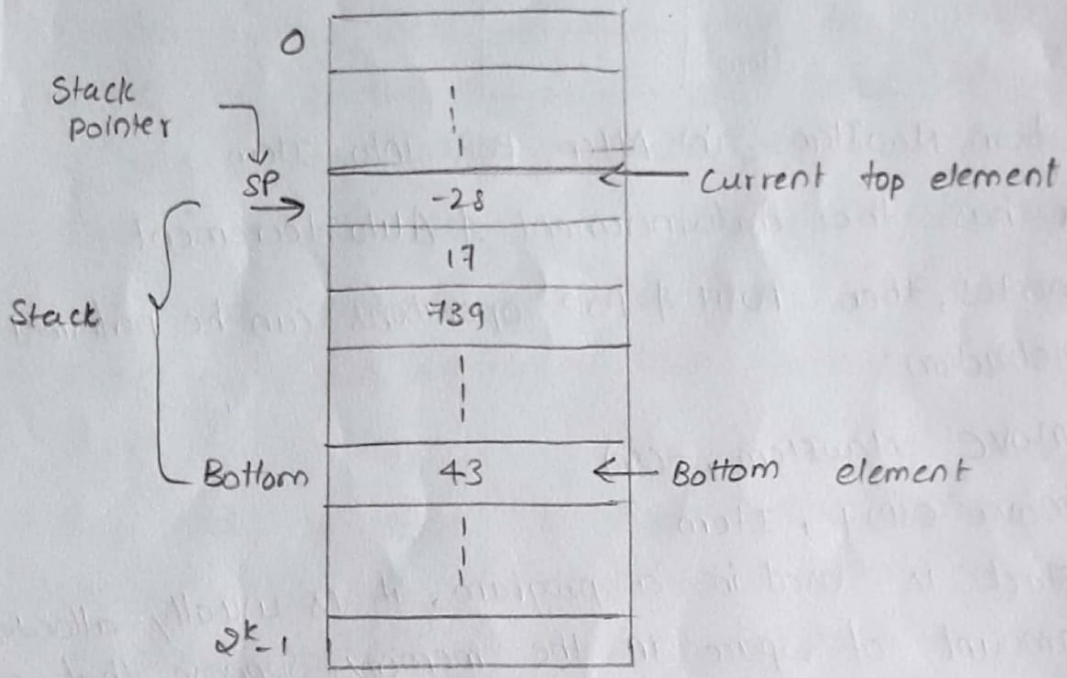


Fig: A stack of words in the memory.

→ The PUSH operation can be implemented as

```
Subtract   #4, SP
Move       NewItem, (SP)
```

Here assumed 32-bit word length.

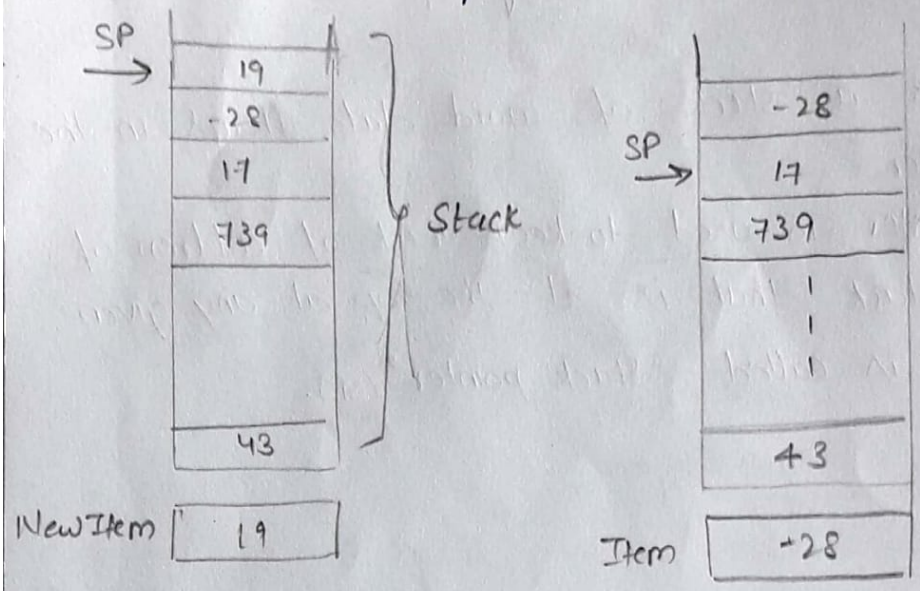
→ These two instructions move the word from location New Item onto the top of the stack, decreasing the stack pointer by 4 before the move

→ The POP instruction can be implemented as

```
Move   (SP), ITEM
Add    #4, SP
```

→ These 2 instructions move the top value from the stack into

location Item and then increment the stack pointer by 4 so that it points to the new top element.



(a) After PUSH from NewItem (b) After POP into Item

→ If processor has the Auto increment & Auto decrement addressing modes, then PUSH & POP operations can be performed by single instruction

PUSH ⇒ MOVE NewItem, -(SP)

POP ⇒ MOVE (SP)+, Item

→ When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. Suppose that a stack runs from location 2000 (Bottom) down no further than location 1500 (TOP).

→ Avoid pushing an item onto stack when stack is reached maximum & avoid attempting to pop an item off an empty stack, which result programming error.

→ For avoiding this problem, a single instruction can replace the PUSH & POP operations that is

compare src, dst

performs [dst] - [src]

src - source
dst - destination

Routine for Safe POP operation

```
SAFEPOP   compare   #2000, SP
          Branch>0  Empty Error
          Move      (SP)+, Item
```

Routine for Safe PUSH operation

```
SAFEPUSH  compare   #1500, SP
          Branch≤0  Full Error
                    NewItem, -(SP)
```

→ Another useful data structure that is similar to the stack is called a queue & data are stored in and retrieved from a queue on First-in-First-out (FIFO) basics.

→ In queue, new data are added at the back (high) address end) and retrived from the front (low address) of the queue.

→ There are two differences b/w Stack & Queue.

Stack

→ One end of stack is fixed (bottom), while the other end rises & falls as data pushed & popped.

→ A single pointer is needed to point to the top of the Stack at any given time.

Queue

→ Both ends of a queue move to higher addresses as data are added at back & removed from the front.

→ So two pointers are needed to keep track of two ends of the queue.

→ Another difference between a stack & a queue is that, without further control a queue would continuously move through memory of computer in direction of higher addresses.

→ One way to limit the queue to a fixed region in memory is to use a circular buffer.

→ Let us assume BEGINNING is starting address of queue and END is ending address of queue.

→ When the data reached to END address, space will have been created at the beginning if some items have been removed from the queue.

→ Hence, Back pointer is reset to the value beginning and process continuous.

Logic Instructions

→ Logic operations such as AND, OR, & NOT applied to individual bits, are the basic building blocks of digital circuits.

→ These logic operations are done using instructions that apply these operations to all bits of a word (or byte) independently and in parallel.

Not dst

→ The NOT instruction complements all bits contained in the destination operand, changing 0's to 1's and 1's to 0's.

1's complement

Not R_0

2's complement

Not R_0

Add #1, R_0

(or)

Negate R_0

$R_0 = 0011$

1's com = 1100

2's com = 1100 + 1

1101

→ Now consider an application for the logic instruction and, which performs the bitwise AND operation on the source and destination operands.

→ Suppose that four ASCII characters are contained in the 32-bit register R_0. In this task, wish to determine if the left most character is z. If it is a conditional branch to Yes is to be made.

→ ASCII code for z is 01011010, which is expressed in hexa-decimal notation as 5A.

AND #$FF000000, R0
Compare #$5A000000, R0
Branch=0 Yes

→ The three instruction sequence implements the desired action.
→ And instruction clears all bits in the right most three character positions of R0 to zero, leaving the leftmost character unchanged.
→ The compare instruction compares the remaining character at the left end of R0 with the binary representation for the character Z. The branch instruction causes a branch to YES if there is a match.

Shift & Rotate Instructions

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

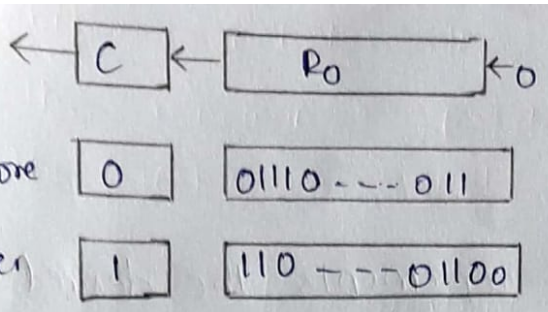
Logical Shifts

Two logical shift instructions are needed, one for shifting left (LShift L) & another for shifting right (LShift R).

→ These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of Logical Shift Left instruction is

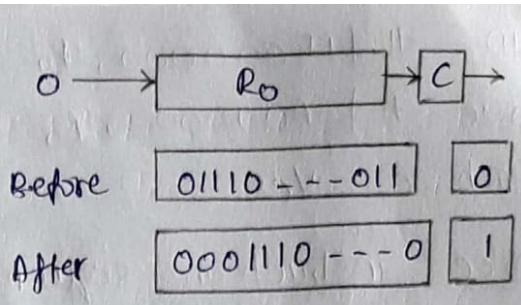
LShift L count, dst

→ The count operand may be given as an immediate operand, or it may be contained in a procesor register. Vacated positions are filled with zero's, and the bits shifted out are passed through the carry flag 'c', and then dropped.
→ Below figure shows an example of shifting the contents of register R0 left by two bit positions.



@ Logical shift left

LShiftL #2, R0



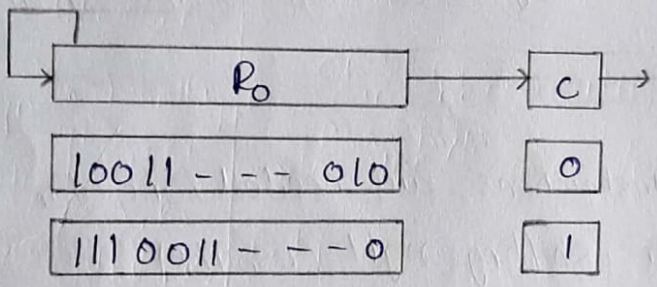
Logical shift Right

LShiftRR #2, R0

Arithmatic shifts

→ Shifting a number one bit position to the left is equivalent to multiplying it by 2 and shifting it to the right is equivalent to dividing it by 2.

→ Another observation is that on right shift the sign bit must be repeated as the fill-in bit for vacated position where in logical shifts vacated positions are filled with zero's.



Arithemehic shift.Right AShiftR #2, R0

Rotale operations

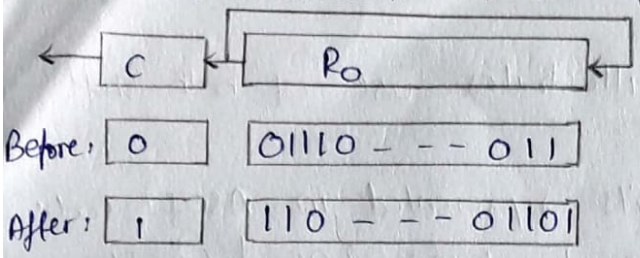
→ In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the carry flag 'c'.

→ To preserve all bits, a set of rotale instructions can be used. they move the bits that are shifted out of one end of operand back into other end.

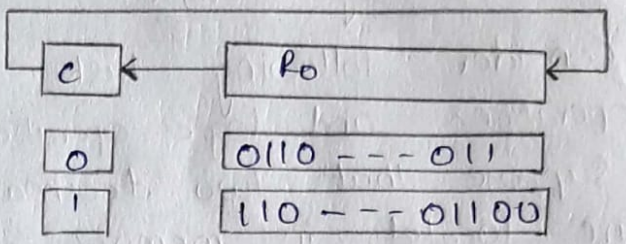
→ Two versions of both the left & right rotale instructions are usually provided. In one version, the bits of the operand are simply rotated. In the other version, the

rotation includes the C flag.

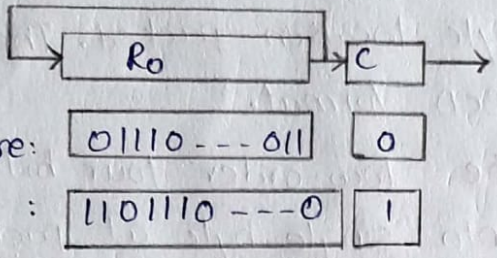
→ The mnemonics RotateL, RotateLC, RotateR, RotateRC denote the instructions that perform the rotate operations.



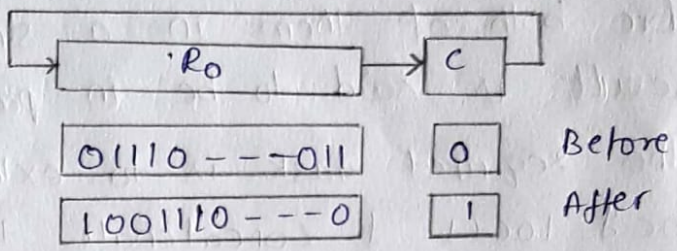
a) RotateL #2, R0



(b) RotateLC #2, R0



c) RotateR #2, R0



d) RotateRC #2, R0

→ Multiplication and Division

→ Two signed integers can be multiplied or divided by machine instruction with the same format like Add instruction.

Multiply Ri, Rj

Performs operation $R_j \leftarrow [R_i] \times [R_j]$

→ The product of two n-bit numbers can be as large as 2n bits. Therefore, the answer will not necessarily fit into register Rj.

→ A number of instruction sets have a multiply instruction that computes the low-order n bits of the product & places it in reg Rj.

→ Higher-order n-bits are placed in R(j+1).

→ Signed integer divide instruction is

Divide Ri, Rj

which performs the operation

$R_j \leftarrow [R_j] / [R_i]$

→ Placing the quotient in R_j. The remainder may be placed in $R(j+1)$, or it may be lost.

Digit packing example

→ Consider following task that illustrates use of both shift operations and logic operation.

→ Suppose that two decimal digits represented in ASCII code are located in memory at byte locations LOC & LOC+1.

→ To represent each of these digits in the 4-bit BCD code & store both of them in a single byte location packed. The result is said to be in packed-BCD format.

→ The required task is to extract the loco-order four bits in loc & loc+1 & concatenate them into single byte at packed.

```
Move      #LOC, R0
Move      (R0)+, R1
LShiftL   #4, R1
MoveByte  (R0), R2
AND       #$F, R2
OR        R1, R2
MoveByte  R2, PACKED
```

Basic Input / Output operations

→ Input / Output (I/O) operations are essential, and I/O device's performance can effect the performance of computer

→ Consider a task that reads in character input from a key board 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 Input and Output.

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

→ The rate of output transfer from computer to display is typically several thousand characters per second and the processor executes millions of instructions per second.

→ The difference in speed b/w the processor and I/O devices creates need for mechanism to synchronize transfer of data between them.

→ A solution for this problem is using Buffer registers at keyboard & display unit.

→ The keyboard & display are seperate devices as shown in below figure

→ Consider problem of moving character code from keyboard to processor.

→ Striking a key stores corresponding character code in a 8-bit buffer register anociated with keyboard, called as DATAIN

→ To inform the processor that a valid character is in DATAIN, a status control flag SIN is set to '1'. when SIN is set to 1, processor reads contents of DATAIN.

→ When character is transferred to processor SIN is automatically cleared to '0' If second character is entered, SIN again set to 1 and proces repeats.

→ A Buffer register DATAOUT & status control flag sout are used

for transfer from processor to display.

→ When the display device is ready to receive a ~~second~~ character SOUT=1 and processor transfers characters code to DATAOUT after that SOUT=0.

→ When the display device is ready to receive a second character SOUT is again set to 1.

→ The Buffer registers DATAIN & DATAOUT & status flags SIN & SOUT are part of circuitary commonly known as device Interface

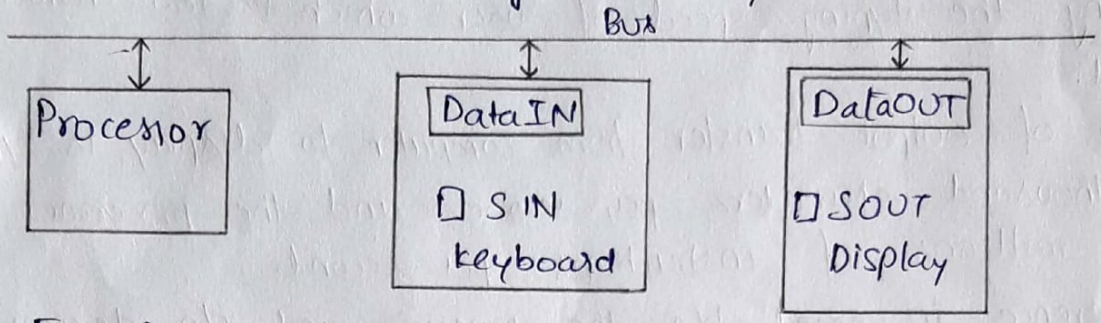


Fig: Bus connection for processor, keyboard, and display.

→ The circuitary for each device is connected to the processor via Bus indicated in figure.

→ For 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
```

→ The first instruction tests the status flag and second instruction transfers input data when SIN is set to 1.

→ An analogous sequence of operations is used for transferring o/p to display. Ex:

```
WRITEWAIT  Branch to writewait if SOUT=0
           Output from R1 to DataOUT
```

→ The wait loop is executed repeatedly until status flag SOUT is set to 1, when SOUT=1, second instruction transfers a

character from R1 to DataOut to be displayed, and clears SOUT to 0.

→ Many computers use an arrangement called memory-mapped I/O in which some memory address values are used to refer peripheral device buffer registers such as DataIN & DataOUT.

→ Data can be transferred b/w these registers & the processor using instructions like Move, Load & Store.

Eg: MoveByte DATAIN, R1
MoveByte R1, DATAOUT

→ The move byte operation code signifies that the operand size is a byte.

→ SIN & SOUT are device status registers one for each of the two devices. Let us assume that bit b3 in registers INSTATUS & OUTSTATUS corresponds to SIN & SOUT respectively.

Read operation

```
READWAIT    Testbit    #3, INSTATUS
            Branch=0   Readwait
            MoveByte   DataIN, R1
```

Write operation

```
WRITEWAIT   testbit    #3, OUTSTATUS
            Branch=0   WRITEWAIT
            MoveByte   R1, DATAOUT
```

→ When the device is ready, i.e., when the bit tested becomes equal to 1, the data are read from input buffers (or) written into the o/p buffers otherwise (Testbit is 0) then condition of branch is true & branch is made to beginning of WAIT LOOP.

→ The program shown in below uses two operations to read a line of characters typed at a keyboard & send them out to a display device.

→ As the characters are read in one by one, they are stored

in a data area in memory & then eehoed backout to the display.

```
Move    #Loc, R0 ; Initialize pointer register R0 to point to
                   the addren of the first location in
                   memory where characters are to be stored.

Read    TestBit  #3, INSTATUS   ; wait for character to be
        Branch=0 READ             entered in keyboard, & buffer
        MoveByte DATAIN, (R0)     (DATAIN) transfers char to
                                  memory, after that clears SIN=0

ECHO    TestBit  #3,OUTSTATUS   ; wait for display to become
        Branch=0 ECHO             ready
        MoveByte (R0), DATAOUT    Move char just read to the display
                                  buffer register (DATAOUT)

        Compare  #CR, (R0)+       check if CR is just read, if it
                                  is not CR then brach back &
                                  read another char.

        Branch≠0 READ
```

→ The program finishes when the carriage return character CR is Read, stored & send to display.

→ R0 is incremented for each character read & displayed by auto-increment addreming mode used in compare instructions