8/12/18

Unit-2

Machine Instruction & program

Instruction & Instructing sequences:- Instructions are needed to perform following operations

(i) Data transform between memory & processor like instructions

(ii) Arthimetic & logic operations on data

(iii) program sequencing and control.

(iv) Input, output transfer.

→ To know abore operations we should learn register transfer language notation. (REN) (RTN)

→ We need to describe possible locations that are involved in data transfer.

→ Those are memory locations, processor registers, Input/output systems or registers

→ LOC, PLACE, A, VAR → related to memory location values

processor registers are reffered with R_0, R_1, R_2 ----

I/o registers are reffered with DATAIN, DATAOUT, OUTSTATUS

Eg:- $R_1 \leftarrow [Loc]$ [The contents of memory location with name loc are transefend to register R_1]

2) $R_3 \leftarrow [R_1] + [R_2]$ [The content of processor register R_1 add to regsiter R_2 stored in processor Register R_3]

In RTN

R.H.S contains contents and L.H.S values are corresponding locations

(i.e, memory locations, processor registers loc), variable names

Assembly language Notation

1) MOVE LOC, R_1 → moving content of loc to processor Register R_1 by overwriting R_1

2) ADD R_1, R_2, R_3

Basic Instruction Types:-

Syntax:-
Operation source1, source2, destination.

Eg:- $C = a+b$

In RTN $C \leftarrow [A]+[B]$

In assembly level language ADD A, B, C

→ The above instruction will not execute in single unit of clk cycle since it contain 3 words

The possible solution is.

ADD A, B
MOVE B, C
} It contains 2 words

→ They introduced accumlator (AC) to execute in single unit of clk cycle.

These are called nemonics.

→ Load A [load A content to accumlator]
Add B
Store C [store accumlator value to memory loc C]

116 → How it stores in modren computers

Move A, R_1 (or) Move A, R_1
Move B, R_2 Add B, R_1
ADD R_1, R_2 Move R_1, C
Move R_2, C

Addressing Modes:-

→ In how many ways assembly language notati-ons can refer memory locations is given by addressing modes concept.

Different types of addressing modes:-

(i) Register mode.
(ii) Direct / obslute mode
(iii) Immediate mode.

Scanned by CamScanner

(iv) Indirect mode

(v) Relative mode

(vi) Index mode

(i) **Register mode:-** In register mode the operand is the contents of processor register. The name of the register is given in the instruction.

(ii) **Direct mode:-** This mode directly transfers the contents of memory location value, to destination (processor register)

Eg:- Move loc, R_1 → Register mode.
↓
Direct mode.

(iii) **Immediate mode:-**

Move 200, R_0 — In this mode the value is stored in register without help of memory location.
(or)
Move #200, R_0

Eg:- A = B+6 → How this instruction executed by processor.

~~Move A, R_1~~

Add locB, #6, LocA
(or)
Add B, #6, A

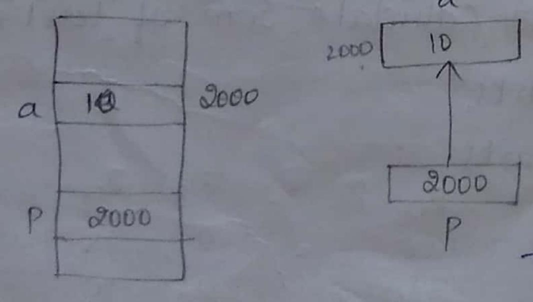
This takes more than one unit of clk cycle.

Move locB, R_1
Add R_1, #6, LocA.
↳ This also doesnot happen exactly in one unit of clk cycle.

→ Move locB, R_1
Add #6, R_1
Move R_1, locA

} → This simple instructions can be done in one unit of clk cycle.

(iv) **Indirect mode:-**



→ Indirect mode since we used pointers

Eg:- int *p;
int a=10, c;
p=&a;
c=*p+5

The indirect addressing mode for the above code is

Move (p), R_1 // Move (2000), R_1 // Move #10, R_1 (x)
↓
The contents of p at 2000 address 10 is stored in R_1

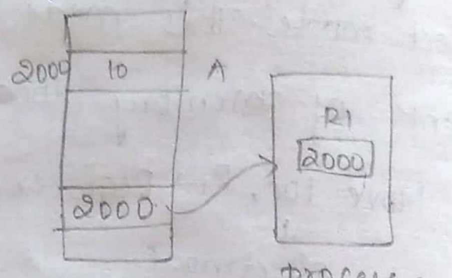
Scanned by CamScanner

C = *P+5 → Move #5, R_1
Add (2000), R_1
Move R_1, C

12/12/18 → Two forms of indirect mode.

1) Add (P), R_1
→ Here p holds adress

2) Add (R_2), R_1
→ registers can hold address not only values.

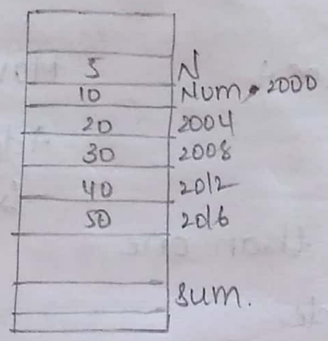


processor

→ In indirect mode effective address of the operand is the contents of the register (or) the memory location (or)

Eg:- caluclate sum of elements in a given list
→ variable name → straight line sequencing

Move N, R_1
Move #Num, R_2 → array name
clear R_0
Add (R_2), R_0 // loop
Add #4, R_2
Decrement R_1
Branch>0 loop
Move R_0, sum.



R_0	R_1	R_2
0	5	2000
10+0=10	4	2004
20+10=30	3	2008
30+30=60	2	2012
40+60=100	1	2016
50+100=150	0	2020

Tracing:- updated values of registers

14/12/18 (vi) Index Mode:-

→ write a program to caluclate sum of all student test 1 marks
(ii) sum of test 2 marks,
(iii) sum of test 3 marks.

Sol Move #list, R_0
Clear R_1
Clear R_2
Clear R_3

R_0	R_1	R_2	R_3	R_4
1000	0	0	0	15
1016	20	30	40	14
1032	35	40	65	13

→ sum of test marks of 2 students

Scanned by CamScanner

Move N, R_4

loop Add 4(R_0), R_1

Add 8(R_0), R_2

Add 12 (R_0), R_3

Add #16, R_0

Decrement R_4

Branch>0 loop

N	15
(1000) List	Student ID
list+4	T_1-20
list+8	T_2-30
list+12	T_4-40
+16	Student ID
+20	T_1-15
+24	T_2-10
+28	T_3-25
	,
	,

continuation

Move R_1, sum1

Move R_2, sum2

Move R_3, sum3

→ General form of index mode is X (R)

where x is displacement / offset.

X (R) is similar to (X + R)

→ effective address = X + [R]

→ There are two ways of using index mode.

(i) Offset is given as a constant (i.e., X(R)) → X+R

(ii) offset is in the register (i.e; (x, R))

(V) Relative mode:-

syntax: X(pc)

Here pc → program counter, which contains next instruction address.

→ As X is displacement we can have -ve aswell as +ve displacement.

→ In Relative modes effective address is calculated by adding offset to program counter value.

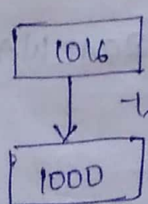
→ In general in memory we need to jump from a memory location to another memory location i.e; 1016 to 1000

→ There should be an instruction to do this

Eg:- present PC value is 1016, next instruction address is 1000

$X(pc) = -16(pc) = (-16 + 1016)$

$= (1000)$



17/12/18 Auto incrementmode.

Move N_1, R_1

Move # Num, R_2

clear R_0.

loop: Add $(R_2)+, R_0$

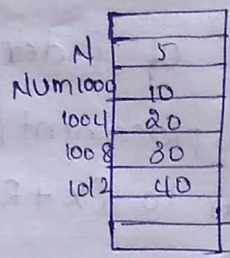
Decrement R_1

Branch>0 loop

Move R_0, SUM.

After accessing the operand, the contents of this register are automatically incremented to next memory location.

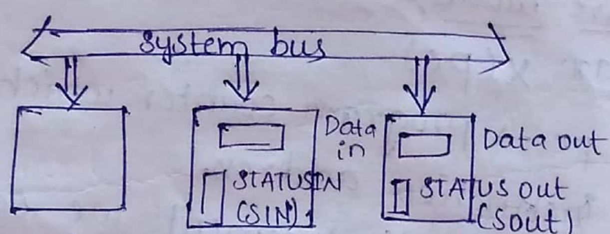
First addition happens after that increment to next memory address happens



R_0	R_1	R_2
0	5	1000
10	4	2004

Basic input/output operations:-/memory communication with other world)

So far we discussed the instructions exists between memory to processor and processor to memory. There is also instruction between input/output devices to memory and processor.



Processor.

When SIN is '0' buffer can't contain the data & SIN is 1 buffer having data.
DATAIN

process is fast when i/o devices are slow.

There are 3 instructions for processing the data i.e, in b/w i/o & processor.

(i) Testbit (ii) Branch>0 (iii) Move byte.

READWAIT Testbit # 3, INSTATUS

Branch =0 READWAIT

Move Byte DATAIN, R_1

Scanned by CamScanner

WRITE WAIT Test bit #3, OUT STATUS.

Branch =0 Read WAIT

Move byte R1, DATAOut

→ In READWAIT, processor has to wait for reading character until the data is ready in DATAIN register (buffer) using SIN (STATUS LAG) This is called processor READWAIT State.

→ In WRITE WAIT, processor having the data and it will send to output device otherwise processor has to wait if DATAOut is empty using Sout this is called processor WAIT State.

19/12/18 **Role of Stacks & Queue's**

→ Stacks & Queue's are memory organisation techniques

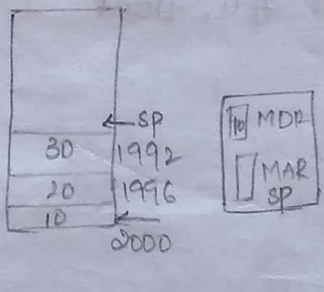
→ Stack organises memory using Last in First out (or) First in last out.

→ Queue is First in First out.

→ Two basic operations for stack PUSH & POP.

→ Eg:- Array Array is organised using stack.

{10, 20, 30, 40, 50, 60}



push → Subtract #4, SP

Move NEWITEM, (SP)

(or)

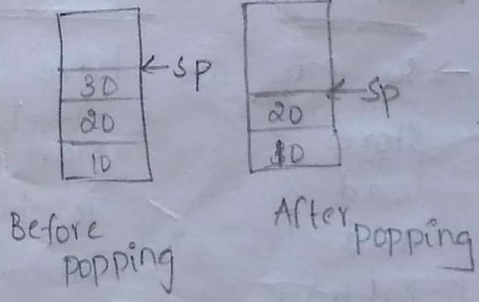
Move NewITem, -(SP) # using auto decrement.

pop → Move (SP), ITEM.

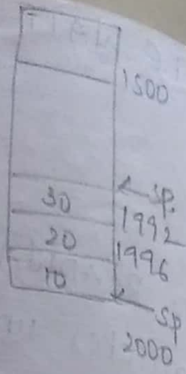
Add #4, SP

(or)

Move(SP)+, ITEM



Safe push → Compare #1500, sp
branch ≤0 FULLERROR
Move NEWITEM, -(sp)

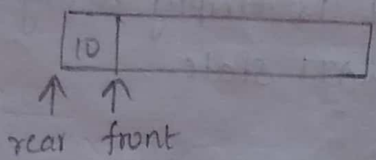


Safe pop → Compare # 2000, sp
branch >0 EMPTYERROR
Move (sp)+, ITEM.

Queue

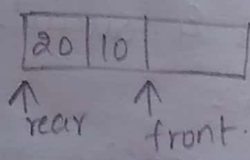
enqueue → Insertion at rear end
dequeue → Deletion at front end.

20/12/18



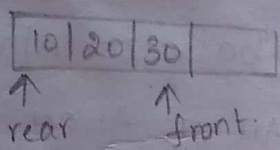
Enqueue(20)

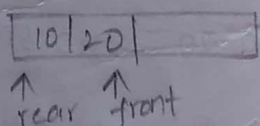
→ Enqueue(20)
Add #4, front
Move (rear), (front)
Move NEWITEM, (rear)

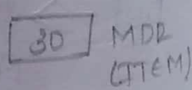


Dequeue

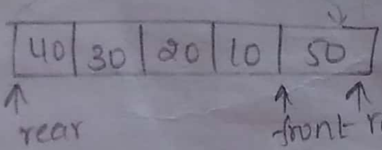
Move (front), ITEM
Subtract #4, front → clear(front)







→



Enqueue(50) → If this was the case we use circular queue to use memory efficiently.

Additional Instructions

→ So far we have learned instructions like Move,
Store
load
Add
Subtract
clear
Branch
compare
Movebite
~~Move byte~~.
Test bit.
Autoincrement & decrement.

Scanned by CamScanner

→ ① logical Instructions → we have not seen instructions

② shift & Rotate Instructions

These instru are basically

→ there a five types of logical Instructions

(i) AND (ii) OR (iii) NOT (iv) XOR (V) TEST

The syntax for all these instructions are:

instruction variable = Operand 1 Instruction operand 2

→ both operands are register values (or) one is register value and the other is memory.

→ Another name for AND operation is "masking"

" " OR operation is "setting."

21/12/18

→ Another name for XOR operation is "clear".

(i) AND (Bitwise AND)

consider R_0 = 00000010

AND R_0, $(01)_H$.

→ 00000010
00000001
00000000

, R_0 = 0000 0101

AND R_0, $(01)_H$

00000101
00000001
00000001

If we want to mask these

→ 1010 1101
0000 1111
0000 1101

we use AND operation (i.e, masking)

If R_0 : 1010 1101

AND R_0, $(0F)_H$

→ write assembly language code to check whether the given number is even (or) odd.

If the given number is 'x'

Move x, R_0

AND R_0, 01H

~~Branch = 0 Even~~

~~TEST~~ :-

→ In logical instructions the relation between operand1 & operand2 is not like source & destination it is like operand1 operation operand2 result is stored in operand 1

TEST:- It is used to say whether given value is even or odd.

→ It is similar to AND operation but R_0 is not updated with result. eg. TEST $R_0,(01)_H$.

OR

OR $R_0,(01)_H$

R_0 : 0000 0010
0000 0001
0000 0011

OR R_0,(

R_0: 1010 1101
0101 0000
1111 1101

XOR

XOR R_0 / clear R_0

R_0 :- 1010 1100
1010 1100
0000 0000

NOT:- NOT R_0

R_0:- 1010 1100
Not R_0- 0101 0011

→ Not R_0 is different from negat-ion R_0

→ negative number (negation) get stored in memory using 2's complement

→ NOT get stored in memory using 1's complement.

1010 1100
1's 0101 0011
1
2's - 01010100 84 (negation R_0)

22/1/18

1) logical-shift left
2) logical shift Right
3) Arithmetic Shift left
4) Arithmetic shift Right

1) logical Shift left:-

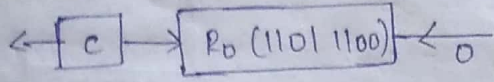
R_0 → 1101 1100
LshiftL #2, R_0

→ Assume R0 value
→ daigram
→ Assembly language notation
→ explanation
→ updated R0

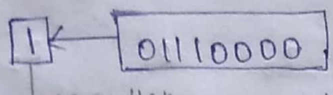
11|0 11100.

01110000

→ In logical shift left
it appends with
"zero".



L ShiftL #2, R0



↓ carry flag
the ~~first~~ lastest shifted bit value.

→ suppose consider R_0 is 0000 0111

R_0 → 0000 0111

LshiftL #1, R_0 // $7 \times 2^1 = 14$

L shift #2, R_0 // $7 \times 2^2 = 28$

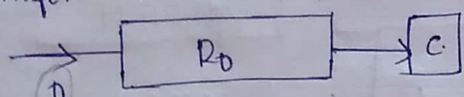
0|000 0111

00001110 =

2) Logical Shift Right

R_0 → 0000 0111

R ShiftR #1, R_0 // $\frac{7}{2^1} = 3$



Here also it appends with '0'.

0000 011|1

0000 011 = 3

0000 01|11

0000 01 = 1

→ RshiftR by 2

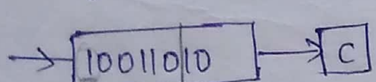
3) Arithmetic Shift left

→ It exactly works like logical shift left.

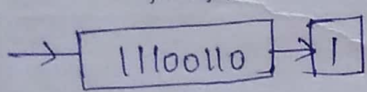
AShiftL (notation)

4) Arithmetic Shift Right.

→ R_0 → 100 11010



AshifR.



→ Arithmetic shift right preserves sign bit.

→ It appends sign bit of a given binary number (i.e., most significant bit)

Imp

→ Application on shift instruction & logical instruction

Digit packing example

Aim:- Extract low order 4-bits in memory locations LOC & LOC+1 & concatenate them into single byte at PACKED

24/12/18 Move #loc, R_0

MoveByte $(R_0)+$, R_1

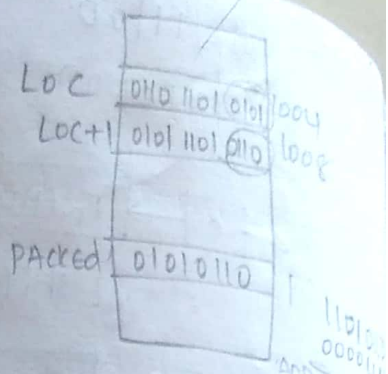
LshiftL #4, R_1

MoveByte (R_0), R_2

AND R_2, OFH

OR R_1, R_2

MoveByte R_2, PACKED



R_0	R_1	R_2
1004	11010101	
1008	01010000	11010110
		0000 0110
		01010110

Rotate Instructions:-

→ Rotate left without carry

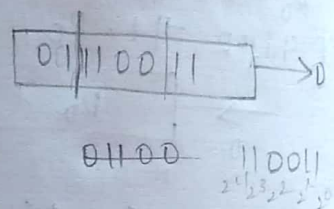
→ Rotate left with carry

→ Rotate Right without carry.

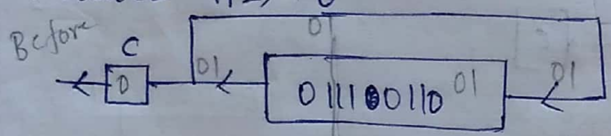
→ Rotate Right with carry.

(i) Rotate left without carry:-

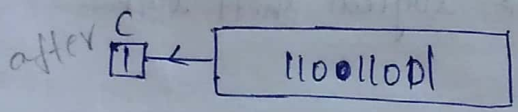
R_0 = 01110011



RotateL #2, R_0

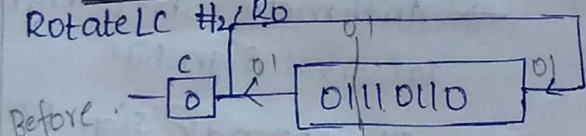


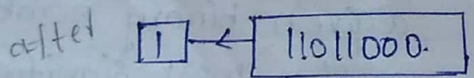
R_0 → 11001101



(ii) Rotate left with carry:-

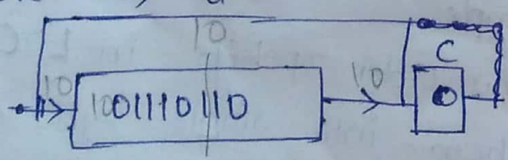
RotateLC #2, R_0





(iii) Rotate Right without carry:-

→ RotateR #2, R_0



R_0 → 10011101

Scanned by CamScanner

C

$\boxed{10011101} \rightarrow \boxed{1}$

(iv) Rotate left with carry:-

→ RotateRC , #2, R_0

$\boxed{01110110} \rightarrow \boxed{}$

unit-II

① Explain the need of Regist

② Addressing modes

③ Role of stacks & Queue.

④ Discuss about shift & Rotate instructions with digit packing example.

Scanned by CamScanner