20/12/21

Processor:-

1) General Pur Reg

2) Special Pur Reg :- 1) IR → Instruction
2) PC → execution sequence
Address of i/s
3) MAR → Address of Memory location
Read data → fun
4) MDR → consists of data
data can be instructions also
for Read & write

3) ALU :- perform arithmetic & logical operations.

4) CU :- Control unit sends signals for r/w

Read:-

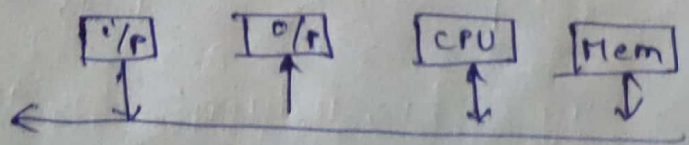
1) MAR → has address of what you want to read.
2) CU → send read signal.
3) Copy data to MDR.

Write:-

1) MAR → where to write
2) MDR → what
3) CU → write signal.

Bus:-

→ wires technically



1) Data bus lines → carry words
2) Address bus → carry address bits
3) Control bus → control bits

Scanned with CamScanner

→ To avoid the speed mismatch of i/p, o/p dev with CPU, we use buffer registers

→ These are a bit faster than i/p, o/p devices and a bit slower than CPU.

Software:-

→ 1) System software → OS, compilers, text editors

2) Application software

↳ user interacts with system to using Application software.

→ system software helps for the life of app software

→ Browser, word.

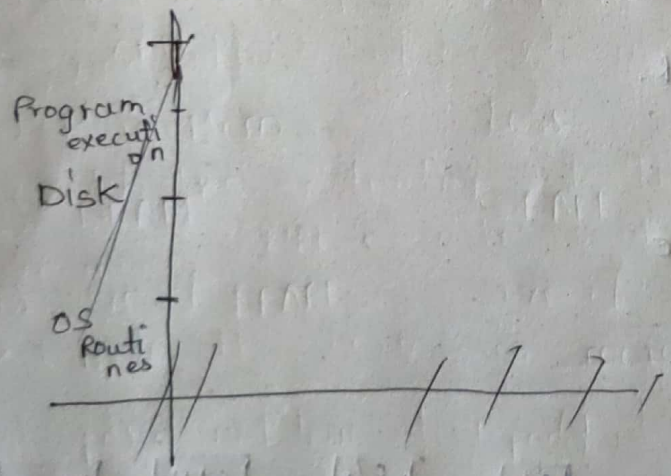
Steps to execute a prgrm

→ Load prgrm from disk to Mem

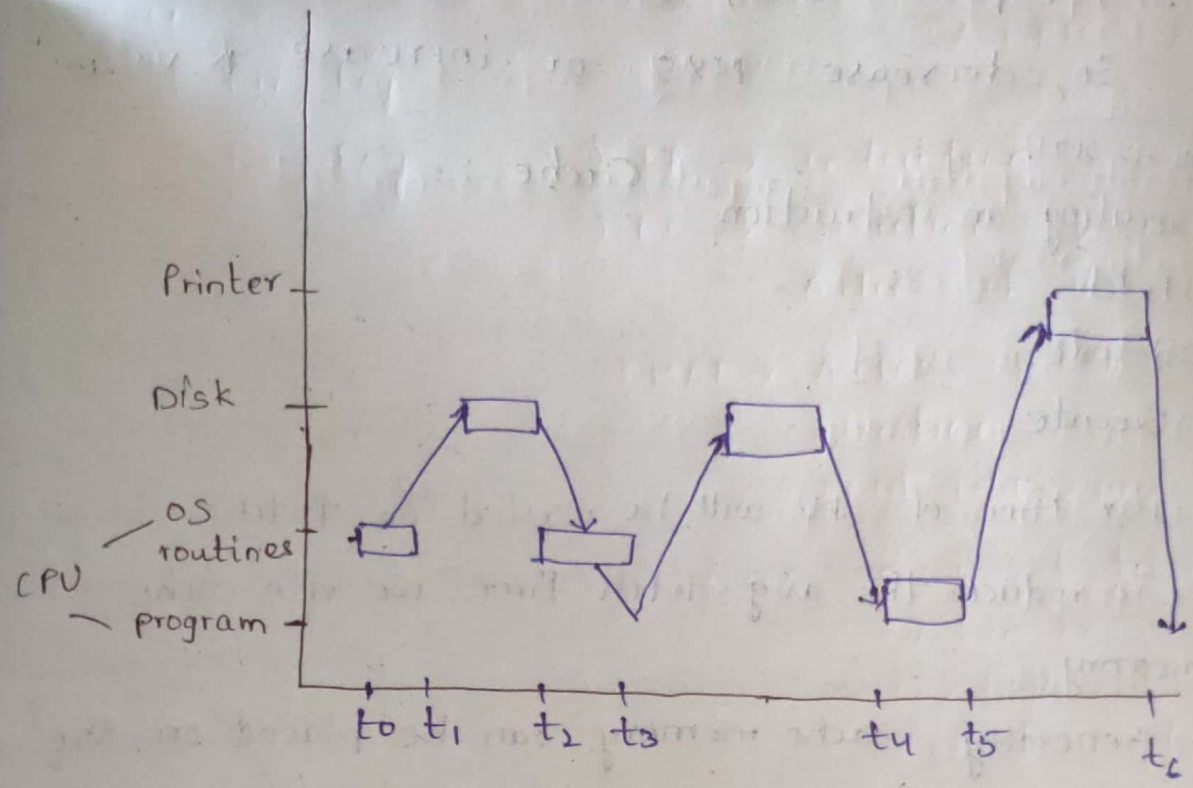
→ Read file from the disk.

→ Calculations on file data

→ Print the results.



Scanned with CamScanner



→ when CPU is free, use that time for another program → multiprgrmming

→ elapsed time → total time for prgrm execution

Processor time → CPU time

Performance:-

→ Measure efficiency of CPU

→ Measure using clock.

→ clock cycle length, P

→ Clock Rate, R = $\frac{1}{P}$ cycles per second. | Hertz

→ In each clock cycle, it executes certain tasks. (each basic inst)

Program → High level lang → N ins each has S basic steps

↓

Compiler

↓

Machine languages.

So, total steps, I = N x S x P

time $t = \frac{N \times S}{R}$ cycles / (cycles/sec) = sec

Scanned with CamScanner

→ For performance, decrease t

So, decrease NXS or increase R value.

24-12-2021

Cache

Executing an instruction

1) Fetch.
2) decode
3) Execute

→ Max time of CPU will be wasted on fetch.

→ To reduce the avg fetch time, we use cache memory.

→ Generally, cache memory can be placed on the CPU.

→ Cache can hold some data / instructions that are present in main memory.

→ Repeatedly accessible part of data from main memory is placed in cache.

	MM	Cache
1)	direct access by CPU	Not direct
2)	More Mem	Less
3)	DRAM	SRAM
4)	size of DRAM is more.	size of SRAM is more.
5)	speed is slow	speed is fast

Dynamic RAM static RAM

→ Using cache, reduces the avg fetch time

Scanned with CamScanner

Performance eqn = $\frac{N \times S}{R}$

→ We can decrease N & S (or) increase R.

For decreasing S:-

→ Using pipelining

→ Overlap stages of program's execution.

For decreasing N×S:-

→ Based on compiler

→ Compiler should be able to produce efficient Machine language.

For increasing R:-

→ Improving the IC of the system.

For reducing N:-

→ Reduced Instruction set computer → easy instructions and more steps.

→ Complex Instruction set computer → complex instructions and less steps.

Performance Measurement:-

→ SPEC → System Performance Evaluation Corporation.

→ set of standard programs to analyse the set of programs.

$$\text{SPEC rating} = \frac{\text{time taken on ref computer}}{\text{time taken on test computer.}}$$

$\text{SPEC} = (\text{SPEC}_i)^{1/n}$ to get final rating of our system.

Scanned with CamScanner

Multiprocessor :- More CPU's but same main memory.

Multicomputer :- computers are connected using networking.

Historical aspects:-

1) First gen comp:- 1945-55 → vacuum tubes, assembly level prgmng
2) Second gen comp:- 1955-65 → transistor, magnetic tape HLP, compilers
3) third :- 1965-75 → IC, OS, Multip, Cache
4) fourth :- 1975-present → VLSI, OS, MP, Vm, pipelining.

1) List the steps needed to execute machine instruction Add LOCA, R_0 interms of transfers between components processor and memory and some simple

Assume that ins is stored in memory at location instr and the address is initially in the reg PC
Assume address of next ins is instr+1

→ Fetch the instruction at the address instr to memory. (MAR ← instr, read by CU, MDR ← data in instr)
→ Decode the terms.
→ Address of location 'A' to MAR
→ Control unit issues read
→ Read the data to MDR
→ Addition is performed using ALU.
→ Store the result of addition to R_0

Scanned with CamScanner

PC → MAR

↓ CU read

MDR (data at location instr)

↓

IR (store it in IR)

↓

Decode (Identify operation, operands)

Add, loc A, R0

↓

Fetching from loc A

↳ MAR ← loc A

→ CU → read

→ MDR → data at A

↓

store that value in temporary reg

MDR → R1

↓

send the instruction to ALU

↓

perform addition in ALU

↓

store that value in R_0

2) Give a short sequence of Machine instructions for the task "Add the contents of Mem loc A to those of loc B and place the answer in loc C."

Instructions load loc, Ri and store Ri, loc are the only instructions available to transfer data b/w Mem & General purpose Reg. Ri.

Don't destroy contents of either loc A (or) loc B.

Scanned with CamScanner

load	locA, R_0	$R_0 \leftarrow m[locA]$
load	locB, R_1	$R_1 \leftarrow m[locB]$
Add	R_0, R_1	$R_1 \leftarrow R_0 + R_1$
store	R_1, locC	$m[locC] \leftarrow R_1$

27-12-2021

Suppose that move, and add instr are available with format move/add loc1, loc2.

The instr move/add a copy of the operand at 1st location to 2nd location, overwriting original operand at the 2nd location.

The loc i can be memory / process reg set.

Is it possible to use fewer instr. to accomplish the prev task? If yes, give the new set of instr.

move	locA, locC	$m[locC] \leftarrow m[locA]$
~~move~~ Add	locB, locC	$m[locC] \leftarrow m[locA] + m[locB]$

0

$2^n - 1$

256

3 2

2 8

8

2

10010101 0

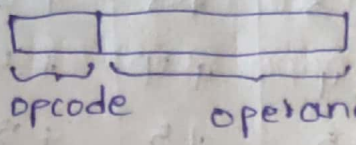
Scanned with CamScanner

27-12-2021

Machine Instructions:-

ISA → Instruction Set Architecture

Instructions:-



→ Types of opcode, No: of operands etc are under ISA.

Numbers:-

1) fixed-point representation (integers)

2) floating-point representation (float)

positive numbers → 0 (+) } signed bits
negative numbers → 1 (-) }

Unsigned :- All bits represent magnitude. (0 to $2^n - 1$) for n-bits

Signed :- One bit (MSB) sign part

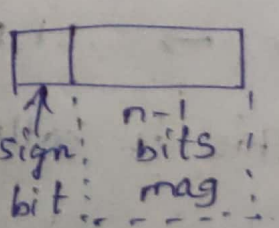
$+101 = 64 + 32 + 4 + 1$

$= 0 | 1100101$

0 ↙ sign, 1100101 ↘ Magnitude.

→ Range of n bit unsigned number is $[0, 2^n - 1]$

Signed :- 1) sign-magnitude
2) Signed 1's complement
3) Signed 2's complement



Scanned with CamScanner

0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

$[-(2^{n-1}-1),\ 2^{n-1}-1]$ sign Mag

$[-(2^{n-1}-1),\ 2^{n-1}-1]$ 1's complement

$[-2^{n-1},\ 2^{n-1}-1]$ 2's complement

+5 → 00000101

1's → 11111010

2's → 11111011

→ This is called sign extension.

-13 1's 2's

+13 → sign +1

+13 →

-13

1's 2's

(i) Represent -256 in 2's using 8 bits.

Not representable

(ii) -256 in 2's using 16 bits

(iii) -256 using 1's using 8 bits

Not possible.

(iv) 16384 using 16 bits in sign-mag, 1's, 2's

(v) -16384 " " " " "

(vi) +13 & -13 in all three systems using 16 bits

(ii) -256

+256 - 0000 0001 0000 0000

Sign Mag - 1000 0001 0000 0000 } -256

1's → 1111 1110 1111 1111 }

2's → 1111 1111 0000 0000

0100 0000 0000 0000

(iv) 16384 → 0 1 0 0 0 0 0 0 0 0 0 0 0 0

16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

1's → 0011 1111 1111 1111

2's → 0011 1111 1111 1111

2 | 16384
2 | 8192
2 | 4096
2 | 2048
2 | 1024
2 | 512

(v) -16384 → 1 0 1 1 1 1 1 1 1 1 1 1 1 1

1's → 1100 0000 0000 0000

2's → 1100 0000 0000 0001

(vi) +13 → 0000 0000 0000 1101

1's → 0000 0000 0000 1101

2's → 0000 0000 0000 1101

-13 → 1000 0000 0000 1101

1's → 1111 1111 1111 0010

Scanned with CamScanner

2's → 1111 1111 1111 0011

(iv)

16384 - 0100 0000 0000 0000

1's - 0100 0000 0000 0000

2's - 0100 0000 0000 0000

(v) -16384 - 1100 0000 0000 0000

1's - 1011 1111 1111 1111

2's - 1100 0000 0000 0000

30-12-2021

→ 2's complement is used because using this all the numbers in the range can be represented.

Addition/Subtraction on signed 2's complement data:-

4-bits:-

A, B A = +2 = 0010

B = +4 = 0100

A+B:- 0 0 1 0

+ 0 1 0 0

0 1 1 0

A = +2 = 0010

B = -4 = + 1 1 0 0

1 1 1 0

↓ binary

1010 = -2

A = -2 = 1110

B = +4 = 0100

+

1 0 0 1 0 → omit carry

= +2

Scanned with CamScanner

A = -2 = 1110

-4 = 1100

1 1010

↓ binary

1101

+1

1110 → -6

A = +4 = 0100

B = +5 = 0101

1001 → 1111 → -7

1010

A = +4 = 0100

B = -5 = 1011

1111 → 1 000

+1

1001 → -1

A = -4 = 1100

B = +5 = 0101

1 0001

A = -4 = 1100

B = -5 = 1011

1 0111 → +7 → This is not in the range of 2's complement representation with 4 bits.

→ When both numbers are having the same sign and if we are adding them, then we are in a state of overflow.

Scanned with CamScanner

Subtraction.

A = +2 = 0010

B = +4 = 0100

2's of 4 = 1100

```
+2   0010
-4  +1100
    ------
    1110   --2's--> 1010 (-2)
```

+8 → 0100

-8 → 01000 → 10111 → +1 → 1000 (1 discarded)

+2 = 0010

-4 = 1100

2's of -4 = 0100

```
+2 → 0010
+4 → 0100
     ----
     0110   +6
```

-2 → 1110

+4 → 0100

2's of +4 → 1100

```
-2 → 1110
+4 → 1100
     ----
  1  1010   --2's--> 1110
                    -6
```

-2 → 1110

-4 → 1100

2's of -4 = 0100

```
-2 → 1110
-4 → 0100
     ----
  1  0010 → +2
```

→ To perform subtraction of two signed numbers, take the 2's complement of second numbers and add them.

-4 → 1100

+5 → 0101

2's of +5 → 1011

```
 1100
 1011
 ----
1 0111 → +7
```

1010

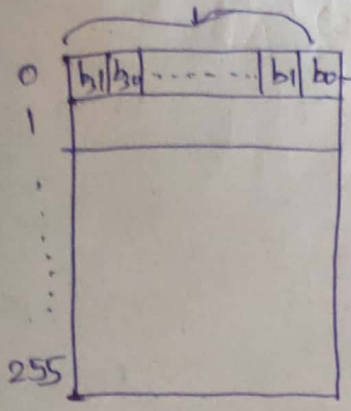
→ When two numbers of different signs are subtracted, then there is a chance of overflow.

Scanned with CamScanner

Instruction storage in Memory :-

Address 8-bits

word = 32 bit



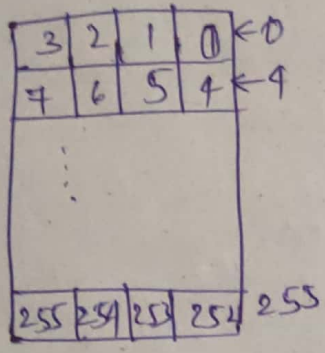
word length is given by system-architecture

16-bit or 32-bit

1 byte = 8 bits

$b_0 - b_7$ → 1 bytes (Byte 0)

Byte-Addressibility :-

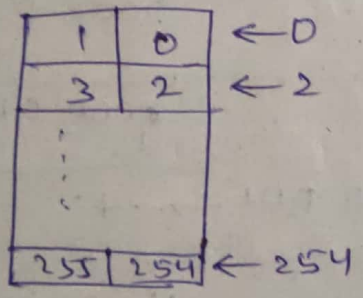


Address 0 means all bytes at address 0.

Right to left → little indian style

left to right → big indian style

For 16-bit :-



For 16-bit system.

→ Byte addressibility is easier access.

2^{30} 64 bytes

64 bytes

$2^{30} - 4$

5 bytes

0
4
8
12
16
60 → 60 61 62 63

101101 1

1000010

Scanned with CamScanner

1.) Use 8 bits to perform addition of following sign -2's complement numbers.

(i) 63, 56 (all possible combinations)

+63 → 1111111 → 0011 1111

-63 → 1100 0001

1100 0000
+ 1
1100 0001

+56 → 0011 1000

-56 → 1100 1000

00111000
11000 111
+ 1
11001 000

111
+63 → 0011 1111

+56 → 0011 1000

0111 0111 → +119

+63 → 0011 1111

-56 → 1100 1000

1 0000 0111 → +7

-63 → 1100 0001

+56 → 0011 1000

1111 1001 → 2's → 1000 0111 → -7

0000 0110
+ 1
0000 0111

-63 → 1100 0001

-56 → 1100 1000

1 1000 1001 → 2's → 11110111 → -119

111 0110
+ 1
111 0111

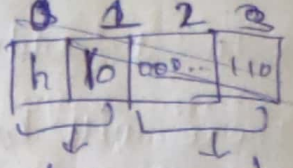
Scanned with CamScanner

(31-12-2021)

Add loc1, loc2 → 12

load loc1, R0 → 8

(5)



ISA:-

Memory operations:-

1.) load → read load locA, R0

2.) store → write store R0, locA

second operand:- destination.

Instructions:-

→ Categorize the instructions based on opcode (4 types)

Opcode:-

1.) Data transfer :- load, store, move

2.) ALU based operations :- Add, subtract, multiply, divide, and, or, not

3.) Program sequencing & control:- Branching, looping (If-else constructs, goto, for, while) jump Branching not equal (BNIE)

4.) I/O based operations:- in operation, out operation

Register transfer Language:- (RTL) [Arrow notation]

1.) $R_i \leftarrow m[loc]$

2.) $R_i \leftarrow R_j + R_k$

Assembly language notation:- (Machine language)

operation operands

1.) add loc, R_1

2.) add R_j, R_k, R_i

Instruction types based on number of operands:-

1.) 3-address instructions → Add R_i, R_j, R_k $R_k \leftarrow R_i + R_j$

2.) 2-address instructions → Add R_i, R_j $R_j \leftarrow R_i + R_j$

3.) 1-address instruction → Add R_i $AC \leftarrow AC + R_i$

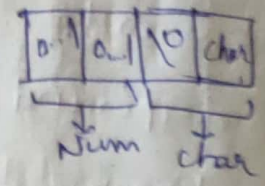
4.) 0-address instructions → push R_i, push R_j, Add
↳ stack based operations

Scanned with CamScanner

→ In 1-address instruction, for the other operand there will be default operand called accumulator.

$y = (A \times B) + (C \times D)$ → 3-Address

multiply	loc A, loc B, R_1	$R_1 \leftarrow A \times B$
multiply	loc C, loc D, R_2	$R_2 \leftarrow C \times D$
add	R_1, R_2, loc Y	loc Y $\leftarrow R_1 + R_2$



Multiply — 2-Address

load	loc A, R_1	$R_1 \leftarrow$ [loc A]
load	loc C, R_2	$R_2 \leftarrow$ [loc C]
multiply	loc B, R_1	$R_1 \leftarrow R_1 *$ [loc B]
multiply	loc D, R_2	$R_2 \leftarrow R_2 *$ [loc D]
Add	R_1, R_2	$R_2 \leftarrow R_1 + R_2$
store	R_2, loc Y	[loc Y] $\leftarrow R_2 +$ [loc Y]

---

load	loc A	ac $\leftarrow$ [loc A]
store	R_1	$R_1 \leftarrow$ ac
load	loc B	ac $\leftarrow$ [loc B]
Multiply	R_1	ac $\leftarrow R_1 \times$ ac
load	loc C	
store	R_2	
load	loc D	
Multiply	R_2	$R_2 \leftarrow R_2 \times$ ac
store	R_2	ac $\leftarrow R_2$
Add	R_1	$R_1 \leftarrow R_1 +$ ac
store	R_1	ac $\leftarrow R_1$
store	loc Y	

Scanned with CamScanner

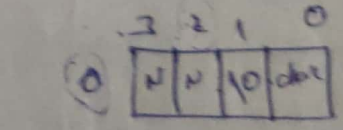
load locA	ac ← [locA]
store R_1	R_1 ← ac
load locB	ac ← [locB]
Multiply R_1	ac ← R_1 * ac
store R_1	R_1 ← ac
load locC	ac ← [locC]
store R_2	R_2 ← ac
load locD	ac ← [locD]
Multiply R_2	ac ← R_2 * ac
# store R_2	R_2 ← ac
Add R_1	ac ← ac + R_1
store locY	[locY] ← ac

1-Address

load locA	ac ← m[locA]	→ optimal.
Multiply locB	ac ← ac * m[locB]	
store R_1	R_1 ← ac	R_1 = A×B
load locC	ac ← m[locC]	
Multiply locD	ac ← ac * m[locD]	
Add R_1	ac ← ac + R_1	
Store locY	[locY] ← ac	

0 → Address

push locA
push locB
Multiply
push locC
push locD
Multiply
Add
pop locY



Scanned with CamScanner

05-01-2022

Default :- 2-address, only one memory location.

Program Sequencing:

Add A, B store to C $C \leftarrow A + B$

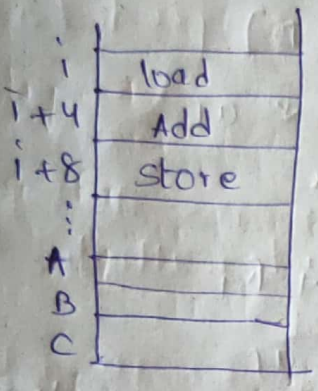
↳ only one mem loc.

load locA, R_1

Add locB, R_1

store R_1, locC

1 word = 32-bit

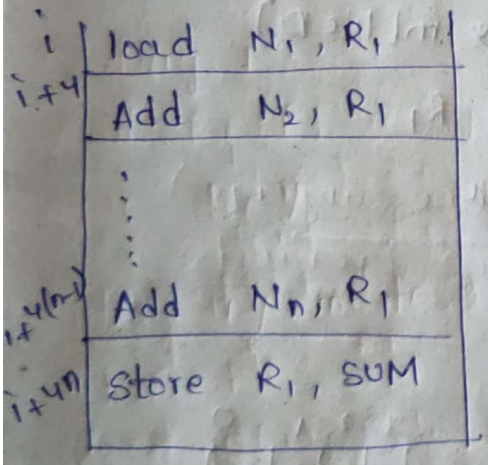


Assume instruction length = 4

→ Regs required :- PC, MAR, MDR, IR

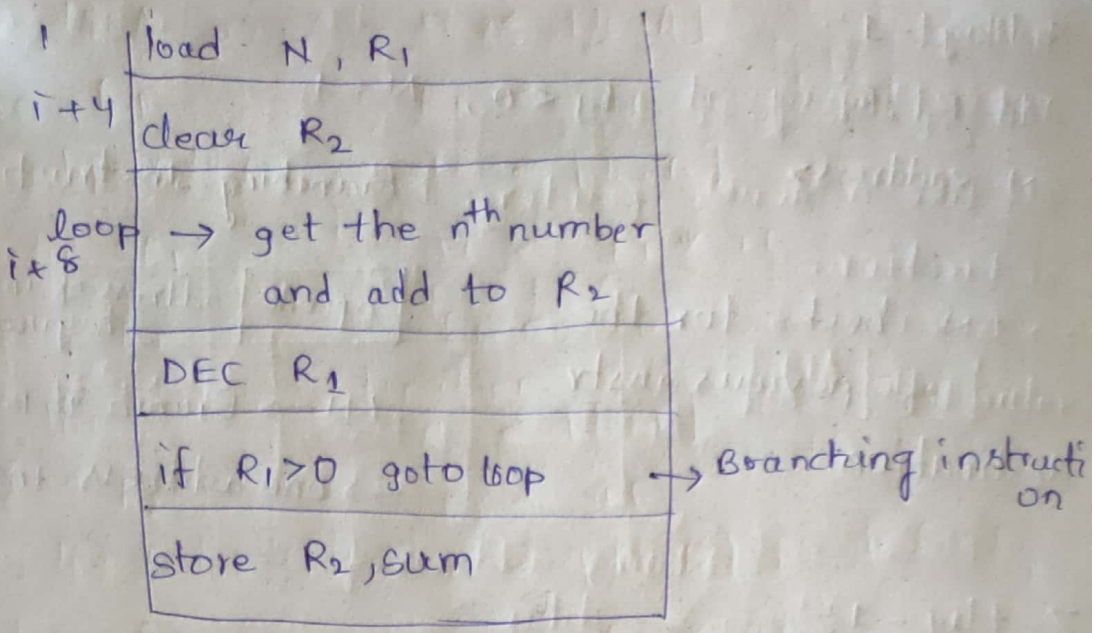
→ Based on value of PC, we say whether it is sequential or branching.

List of n numbers addition




```
for(i=1, i<n, i++)
    sum = sum + n[i]
```

Scanned with CamScanner



→ Branching alterates the PC of the instruction.

PC ← i
i+4
i+8
i+12
i+16
(i+20) → i+8
i+12
i+16
(i+20) → i+8

1) Conditional Branch → check for the condition
2) Uncondition Branch - jump, goto

Conditional Codes:-

1) Negative - N
2) Zero - Z
3) Overflow - O
4) Carry - C

→ These are like flags, if satisfies the condition it will be set to 1.

Scanned with CamScanner

N-flag -1:-

Add Ri, Rj if Rj < 0, N = 1

BN <address>

Branching Instruction

opcode	address of instr to jump

→ BN checks for the value of previous instr destination.

BN → Branch if negative

→ one address instr.

Z-flag - 1:-

Clear R0

BZ loc Y

DEC R1

BNZ loc loop

O-flag - 1:-

Add R1, R2

BNO loc Y

C-flag-1:-

Add R1, R2

BNC loc Y

BN

BNN

BZ

BNZ

BO

BNO

BC

BNC

~~Addressing modes~~

R1 ← N

R2 ← 0

R3 ← get & Fetch

R2 ← R2 + R3

R1 ← R1 - 1

sum ← R2

Scanned with CamScanner

06-01-2022

Addressing Modes:-

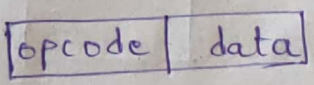
1) Immediate Addressing mode.
2) Register Addressing mode.
3) Direct Addressing Mode
4) Indirect → Memory Addressing Mode
   → Register Addressing Mode.
5) Index Addressing Mode
6) Index with base
7) Index with base & offset
8) Relative / PC
9) Auto-Increment
10) Auto decrement

→ Specifies where exactly operands are presents.

1) Variables, constants

H.L.L:- int x;

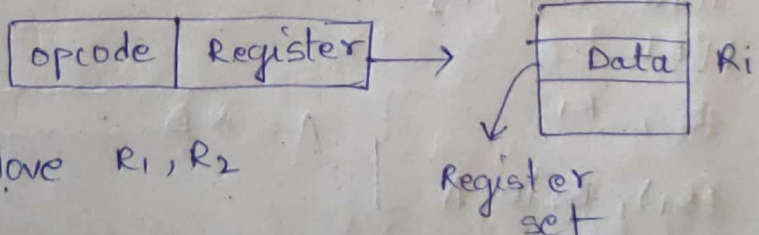
Immediate :-



Move #2, R1

→ indicates that it is data but not address.

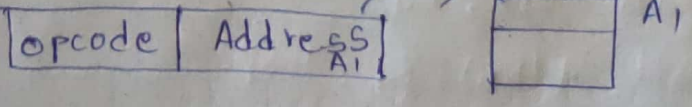
Register:-



Move R1, R2

→ Register has data directly.

Direct/Absolute:-



→ Data lies in memory.

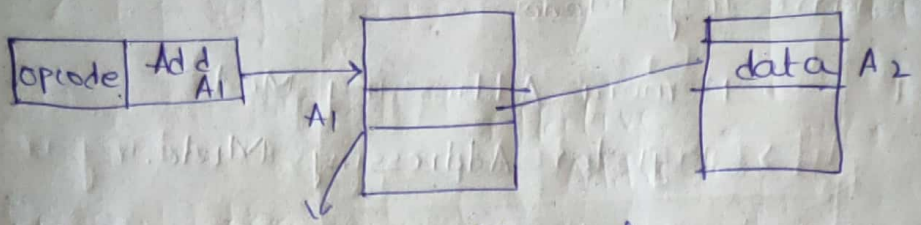
load loc A, R1

Scanned with CamScanner

Indirect :-

WRT Memory :-

→ Like a double pointer, has one more level of indirection.



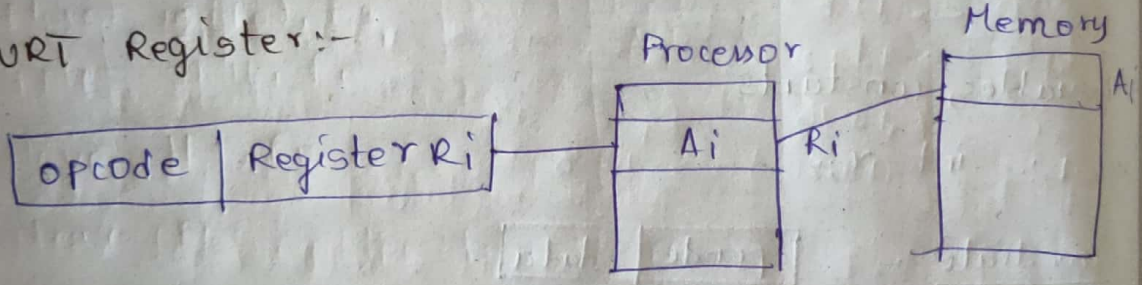
contains address of data

Ex :- load (LOC), R1

( ) → indicate the data is not present in that address but another address.

→ Effective Address is the address from where the data is obtained.

WRT Register :-



→ Single Memory access.

Ex :- load (R1), R2

MLL

load B, R1

load (R1), LOC A

} A = *B

Scanned with CamScanner

load N_1, R_1 → clear R_2, load size, R_3

Add (R_1), R_2

INCR R_1 by 4

if (R_3) > 0 goto loop

store R_2, sum

=

load size, R_3

clear R_2

load N_1, R_1

Add (R_1), R_2

INCR R_1 by 4 → DECR (R_3)

if (R_3) > 0 goto loop

store R_2, sum

=

load size, R_3

clear R_2

load #N_1, R_1 : loop → N_1 → data at N_1
#N_1 → address of N_1

Add (R_1), R_2

~~Sub~~ ~~R~~

Add #4, R_1 → #4 exact value 4

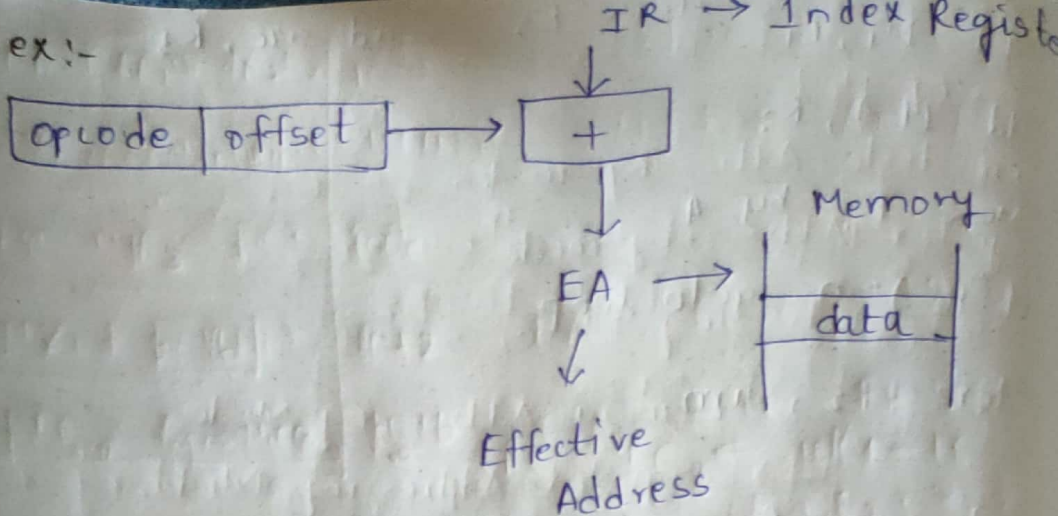
DEC R_3

BNZ loop

Store R_2, sum

Scanned with CamScanner

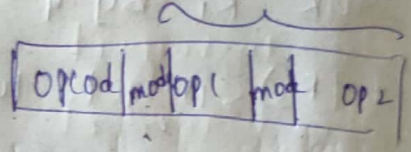
Index :-



Move 20(R1), R2

offset — Index Register.

Move #i, R0
Move 4(R0), R1
Move 8(R0), R2
Move 12(R0), R3
INC #16, R0



Move #1, R0
clear R5
Move #3, R5
Move 4(R0), R1

Move N, R0
clear R1
clear R2
clear R3
load #i, R4
Add 4(R4), R1 : loop
Add 8(R4), R2
Add 12(R4), R3

Scanned with CamScanner

DEC R0

Add #16, R4

BN2 loop

store R1, sum1

store R2, sum2

store R3, sum3

07-01-2022 Index with base.

MOVE (Ri, Rj), Rk

opcode	BR, IR

BR+IR → Effective Address

Move (R1, R2), R3

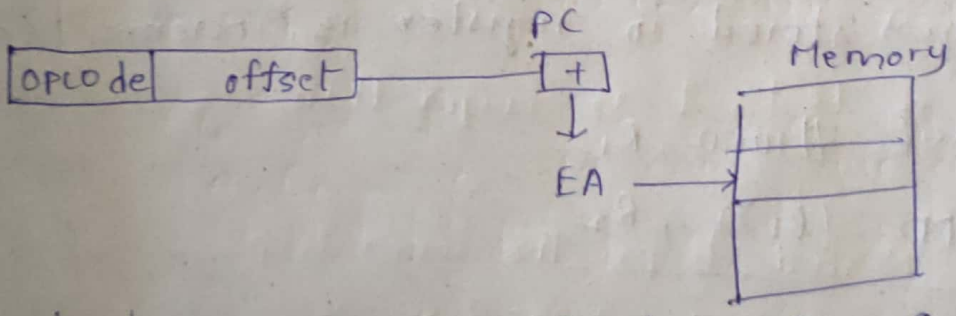
Index with base and offset

Move 10(Ri, Rj), Rk EA = Ri + Rj + 10

opcode	off (IR, BR)

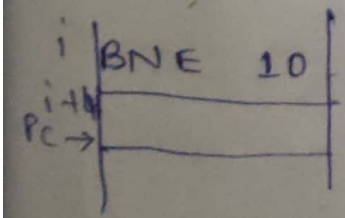
→ If offset is huge, then pass it into another and use that.

PC related:- (Relative)



→ This is generally used for instruction fetching.

→ Used with branching instruction.



If the above condition is true

PC ← i+4 + (10) → offset.

Scanned with CamScanner

ADD	(R_1), R_2	1000
INC	#4, R_1	1004
DEC	R_3	1008
BNZ	-16	1012
STORE	R_2, sum	1016

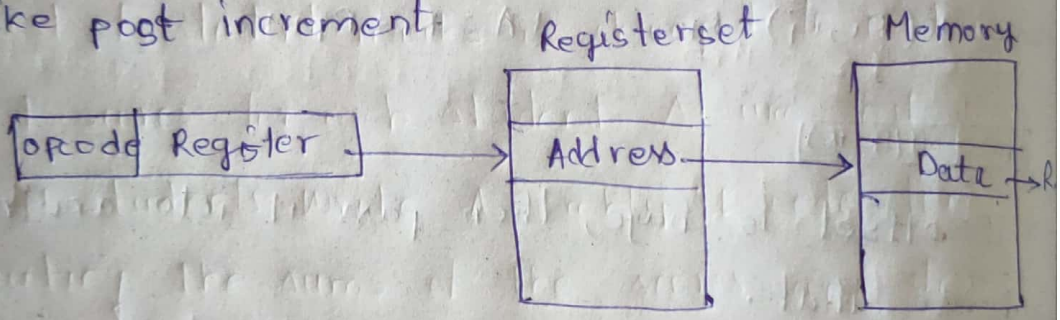
← PC PC ← 1060 − 16

→ Even if the program is replaced to another memory location, then also it performs correctly.

Auto Increment Addressing Mode:-

→ used with loops

→ like post increment



After fetching the data from address, automatically the address stored in register is incremented.

Ex:- MOVE #1000, R_1

ADD $(R_1)+$, R_2

$R_1 = 1004$

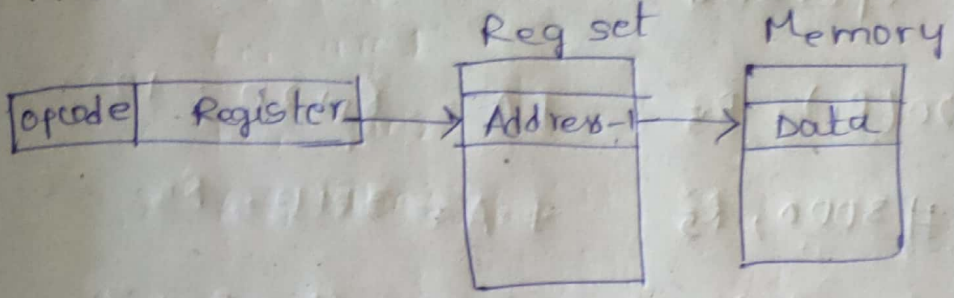
$R_2 \leftarrow R_2 + R_1$

Scanned with CamScanner

Auto Decrement

→ Pre Decrement

→ First decrement and then execute.




```
Move   N, R0      0
clear  R1         4
clear  R2         8
clear  R3         12
load   (#i), R4   16
Add    (R4)+, R1  20
Add    (R4)+, R2  24
Add    (R4)+, R3  28
DEC    R0         32
BNZ    -12
store  R1, locA
store  R2, locB
store  R3, locC
```

(#i)+

INCR (R1)+

+1 +

Scanned with CamScanner

Regs R_1 and R_2 of comp contain the dec values 1200, 4600. What is the effective Add of memory operand?

1.) Load 20(R_1), R_5 EA = R_1+20 = 1220

2.) Move #3000, R_5 ~~EA = 3000 = R5 = 3000~~

3.) Store R_5, 30(R_1, R_2) E.A = (R_1 + R_2 + 30) = 5830

4) ADD -(R_2), R_5 E.A = R_2 - 4 = 4596.

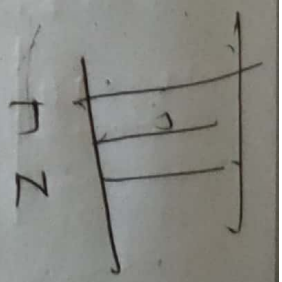
5) SUB (R_1)+, R_5 E.A = R_1 = 1200

~~(1) E.A~~

The list of student marks is changed to contain I-test scores for each student Assume that there are N students. Write Assembly L code for computing the sums of the scores and store these sums at at Addresses, sum, sum+4, sum+8, ...

The No: of tests I is larger than the No: of regs.

Use two Nested loops. The inner loop → sum for one test and outer loop for No: of tests I. Assume that I is stored in Memory locati I placed ahead of N.



Scanned with CamScanner

load locJ , R0
load loc N , R1
load #i , R2
~~Add #4, (R2)~~ Add #4, R2
Move (R2)+, R3

clear R4
L Add 4J(R3)

---

load	locJ , R0	R0 ← [J] i
Mv(load)	#J, R1	R1 ← J
8 Add	#1, R1, R2	R2 ← R1+1
12 Mul	#4, R2	R2 ← R2*4
16 Mv(load)	#0, R3 | clear R3	R3 ← 0
20 load	locN, R4	R4 ← [N] n
24 (Move/	12(R1)/ R5)	R5 ← R1+12
28 Add	#12, R1, R5	
32 Add	R3, R5	R5 ← R3+R5
36 clear	R6	R6 ← 0
40 Add	(R5), R6	R6 ← R6+[R5]
44 Add	R2, R5	R5 ← R5+R2
48 DEC	R4	R4 ← R4-1
52 BNZ	-12	loop to 40
56 store	R6, (sum, R3)	[sum+R3] ← R6
60 Add	#4, R3	R3 ← R3+4
64 DEC	R0	R0 ← R0-1
68 BNZ	-48	loop to 20

Scanned with CamScanner

0	load	[N], R0	DIR	R0 ← [J]
4	move	#J, R1	IMM	R1 ← J
8	Add	#1, R1, R2	IMM	R2 ← R1+1
12	Mul	#4, R2	IMM, Reg	R2 ← R2*4
16	clear	R3	Reg	R3 ← 0
20	load	(R1, #1/AD), 4(R1), R4	Index	R4 ← [R1+4]
24	Add	#12, R1, R5	IMM	R5 ← R1+12
28	Add	R3, R5	Reg	R5 ← R5+R3
32	clear	R6	Reg	R6 ← 0
36	Add	(R5), R6	Reg Indirat	R6 ← R6+[R5]
40	Add	R2, R5	Reg	R5 ← R5+R2
44	DEC	R4	Reg	R4 ← R4-1
48	BNZ	-16	PC	loop to 36
52	Store	R6, (sum, R3)	Index with base	[sum+R3] ← R6
56	Add	#4, R3	IMM	R3 ← R3+4
60	DEC	R0	Reg	R0 ← R0-1
64	BNZ	-48	PC	loop to 20

Last instruction Branch, then value of PC

Scanned with CamScanner

Assembly Language:- (17-01-2023)

→ Source language is used for source program

→ Assembly language consists of some mnemonics like opcode, operands

→ Object programming → code in 0's & 1's

HLP —compiler→ byte code ←— assembler.

↳ Assembly language

computer organization 5th ed pdf

Syntax rules for A.L:-

→ opcode → capital

→ Right operand → destination.

→ # → used for immediate data

→ opcodes have some fixed notations in binary form

if 15 codes are there, we can use 4 bit code

opcode □ operands (M op1 M op2)

↓ mode bits to give Addressing mode.

bit codes

→ ADDI → Add immediate data to destination.

→ EQU, ORIGIN, RESERVE → directives.

→ The above words are assembly directives.

ORIGIN → Next storage of data. (Next address)

Reserve → store / require that much storage

Return → prgrm completed

start, end

Scanned with CamScanner

Assembly language Instruction:-

Label + operations + operands

→ We can have comments too.

Symbol table:

→ used by assembler for storing values of the labels in source program.

→ 2-pass assembler
1st pass go through names and store its addresses.

→ Loader

A.L $\xrightarrow{\text{Assembler}}$ M.L

→ Loader loads the M.L program from hard disk to Main Memory.

→ Loader should know the length of program and starting address of program.

→ Debugger will help to recover the errors.

→ # 10 → decimal Immediate

#% 10 → binary Imm

#$ 10 → Hexadecimal.

Scanned with CamScanner

21-01-2022

Input/output:-

→ speed of I/O devices is small when compared to processor.

→ So, we use some interfaces to match the speed of these i/o with processor.

→ Modes of I/O:-

1) Polling

2) Interrupt

3) DMA

→ There will be some bits like sin, sout.

sin $\xrightarrow{0}$ indicates that processor should wait for the data transfer from device to device buffer.

sin $\xrightarrow{1}$ processor can access data in buffer.

sout $\xrightarrow{1}$ data transfer from buffer to display.

sout $\xrightarrow{0}$ processor can't transfer, monitor displays

Readwait Branch to readwait if sin = 0

Input from datain to R_1

writewait Branch to writewait sout = 0

(Buffer/out) Send data to dataout.
Output from R_1

Data Transfer b/w Peripheral devices & processor Regs.

1) Memory Mapped I/O :- MOVEBYTE R_1 DATAOUT
MOVEBYTE DATAIN R_1

2) Isolated I/O

① → Buffer Regs are taken as Memory addresses.

Scanned with CamScanner

INSTATUS → registers (4 bits)

↳ 3rd bit (sin bit)

```
WW    Testbit  #3  OUTSTATUS
      Branch=0   WW
      MOVEBYTE   R1, DATAOUT

RW    Testbit  #3  INSTATUS
      Branch=0   RW
      MOVEBYTE   DATAIN, R1

RW    Test bit  #3  INSTATUS
      Branch=0   RW
      MOVEBYTE   DATAIN, R1
      CMP   #CR, R1
      BNZ   EXIT
      STORE  R1  (LOCA, R3)  ──→ Add #1, R3
WW    Test bit  #3 OUTSTATUS
      Branch=0   WW
      MOVEBYTE   R1, DATAOUT
      BNZ   RW

EXIT
```

Scanned with CamScanner

25-01-2022

Stacks:-

Subroutine:-

1.) function calls

2) Loopings

→ stacks store from higher addr to lower addr.

→ Stack pointer (SP) points to the top of the stack.

→ So, to insert into stack

1.) update SP

2) push new item to updated S.P.

→ PUSH

```
SUB  #4 , SP
MOVE  R0 , (SP)
```

→ MOVE R0, -(SP)

→ POP

```
MOVE (SP), R0
Add  #4, SP
```

→ MOVE (SP)+, R0

```
SUB  #4 , SP
CMP  #1500, SP
BZN  EXIT
MOVE R0, (SP)
EXIT
```

SAFEPUSH

```
CMP  #1500 SP
BZ/BN  EXIT
MOVE R0, -(SP)
```

```
CMP #2000 , SP
BNN  EXIT
MOVE (SP)+, R0
```

SAFE POP

Scanned with CamScanner

Queues:-

Subroutine:-

Call:- store PC to link register.
Branch to the sub routine by loading address of subroutine to PC.

LR ← PC
PC ← Address of starting instr of sub routine.

Return:- Load PC with Link reg.

In case of sub-routines nesting; the link register may not be helpful.
Because the LR will get updated to latest SR PC; which leads to loss of previous SR PC.

Scanned with CamScanner

In case of sub-routines nesting; the link register may not be helpful.
Because the LR will get updated to latest SR PC; which leads to loss of previous SR PC

28-01-2022

→ To pass parameters to sub routine from main method, we use registers.

```
call ListAdd
```

Here listadd is subroutine and we use this instruction for calling subroutine

→ If there is a need to transfer more data between main and subroutine, then reg set may not be sufficient.

Scanned with CamScanner

Logical operations (01-02-2022)

NOT, AND, OR

NOT R_1

AND R_1, R_2

AND #$FFFF 0000 , R_1

The above instruction converts the last 16 bits of R_1 to zeroes.

32-bit ASCII code → R0

Check if most significant byte is Z

Z ← 0101 1010 B

AND #$FF00 0000 , R_1

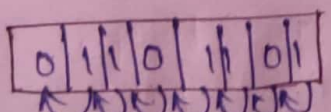
CMP #$5A00 0000 , R_1

OR R_1, R_2

Shift Operations:-

1) Logical (Left / Right)
2) Arithmetic (Left / Right)
3) Rotate (Left Rotate carry / Left Rotate)

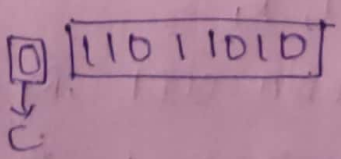
Logical shift:-



Here we lose MSB

LSHIFTL count, Rj

LSHIFTR count, Rj

Logical left shift:- 

Scanned with CamScanner

Arithmetic Right shift:-

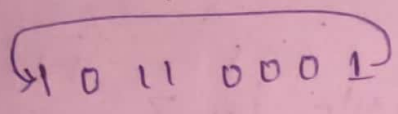
→ Sign of the number will not change

```
1 0 1 1 0 0 0 1
↓   1 0 1 1 0 0 0
1
```

1101 1000

Rotate operation:-

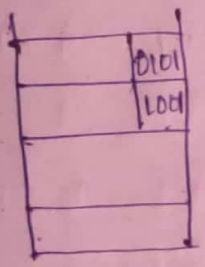
Rotate Right:-



1 0 11 0001

1101 1000

LSB → MSB




```
MOVEBYTE loc, R0        R0 = 0000 0101
MOVEBYTE loc+1, R1      R1 = 0000 1001
LSHIFTL #4, R1
ADD R0, R1, loc

LOAD loc, R0
LOAD loc+1, R1
AND #$0000 000F, R0
AND #$0000 000F, R1
LSHIFTL #4, R1
ADD, R1, R0, PACKED
```

Scanned with CamScanner

1) Vector dot product

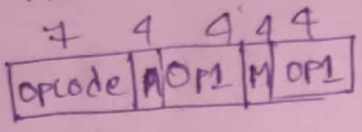
2) Sorting

3) Linked List

02-02-2022

Encoding of Instructions

Add R_1, R_2



No: of opcodes supported

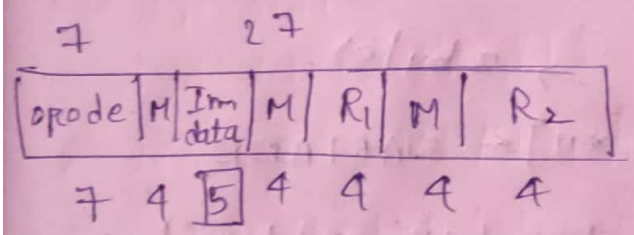
No: of Regs present

This information is required.

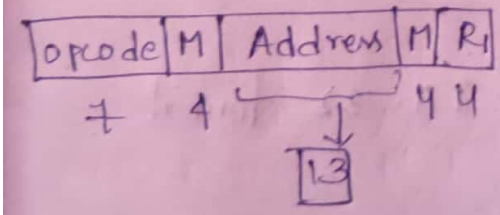
To store this information, we require

23 bits

Add $5(R_1), R_2$



Add Loc, R_1



But 13 bits are not enough to represent the loc address.

→ So, we use 2 words to store this instr.

→ This is called as CISC architecture.
↳ Instruction can take More than 1 word.

Scanned with CamScanner

RISC → 1 WORD (only 1 word instruction)

```
0   MOV   EAX , N
4   clear EBX
8   MOV   ECX , N+1H
12  Add   EBX , [ECX]
16  Add   ECX , 4D
20  DEC   EAX
24  JNZ   -16
28  MOV   SUM, EBX
```

04-02-2022

```
LEA   EBX, NUM1
SUB   EBX, 4
MOV   ECX, N
MOV   EAX, 0
ADD   EAX, [EBX + ECX *4]   ← STARTADD   (3 bytes)
LOOP  STARTADD              - Dec ECX and branch back if [ECX] > 0
MOV   SUM, EAX
```

LOOP instr is supported in IA-32 format

OPCODE	Add mode	Displace	Imm
1 or 2 bytes	1 or 2	1 or 4	1 or 4

→ Min Instr length is 1 byte.

Scanned with CamScanner

Consider the instruction

INC EDI

1 byte

So, for mentioning the register number, 3-bits are enough. (There are only 8 regs in IA-32)

5-bits for identifying opcode.

MOV EAX, 1000 - 5 bytes

1 byte → Opcode + Reg

IMM data → 4 bytes

MOV ECX, LOC

↓ ↓ ↓

1 byte 1 byte 32 bits

4 bytes

32 + 3

4

Total :- 6 bytes.

1000, 1003, 1004, 1005

ADD EAX, [EBX + EDI * 4] 3 + 1 + 1 + 2

↓ ↓

1 bytes 2 bytes

MOV DWORD PTR [EBP + ESI * 4 + DISP], 10

↓ ↓ ↓

1 2 + 4 4

Scaling factor can be of only 1, 2, 4 (or) 8

DD → double word

Flags:- ZF, CF, SF, OF (Zero, Carry, Sign, Overflow)

Scanned with CamScanner

(09-02-2022)

Interrupt Nesting:-

→ Use priorities to handle multiple interrupts.

→ To change the priority, switch from user to privelege mode.

→ Use multi-level priorities to handle

→ Use multiple bus lines to handle interrupts properly.

→ To handle simultaneous interrupt, we use daisy chain.

→ Only IA bus line and it passes through all devices.

→ So, the device which electrically nearer to processor will be served first.

Controlling device requests:-

→ If we want any i/o device not give any interrupts, there will be some bits like enable bits which when set to 0, then that will n't be able to give interrupts

Consider a processor that uses vectored Interr
where starting address of is stored
at INTVEC. Interrupts are enabled by
setting to 1. and
which we assume it is bit 9. A keyboard and

Scanned with CamScanner

display unit cncted to the processor have the status control and data regs as we discussed. Assume that at some point in a prgrm called main, we read input line from keyboard and store the line in succesive bytes in the mem starting at loc 9.

→ List out steps for initializing interrupt process.

→ steps by ISR.

1.) store pc and regs to stack.

2.) Update PC to INTVEC

3.) Load the data / instr to Mem.

1.) Load the address in INTVEC to PC.

2.) Enable keyboard interrupt by setting bit 2 in reg CONTROL to 1.

3) Set IE bit to 1 in processor status register.

Scanned with CamScanner

02-2022

Exceptions:-

→ Interrupts are a form of exceptions.

→ The current instruction which is causing excep will not be completely executed.

Debugging:-

→ Finding out errors.

1.) Trace :- Every instruction is going to executed and going to cause excepti

2.) Break points to check for errors.

Stop execution after every regular interval of instructions.

Scanned with CamScanner

07/03/2022

Cache Memory :-

→ used to store the frequentely accessed data.

→ Fetching data from cache reduces the overhead.

→ smaller than M.M but faster

→ If data is present in cache, it is called hit

→ If not miss.

→ If data is missed, then a copy of the data is stored in cache memory.

→ To map addresses from MM to cache :-

3 - tech

(i) Direct Map

(ii) Associative Map

(iii) set - associative Mapping

Address

Tag	Block	data

$$\text{Tag} = \frac{\text{M.M blocks}}{\text{Cache M block}}$$ gives the No: of bits in tag field.

→ In cache memory, there is a tag field for every block in cache field.

① In direct mapping, using the tag field, we directly store block there.

Check videos for Memory Mapping

Scanned with CamScanner

25-03-2022

Computer Arithmetic:-

Consider two integers A & B in 2's-co

$A = (00010001)_2$ $B = (00010110)_2$

Show A+B

A = 00010001

B = 00010110

R = A+B = 00100111

$A = (00011000)_2$ $B = (11101111)_2$

A = 00011000

B = 11101111

1 00000111

omit carry

Subtraction

$A = (00011000)_2$ $B = (11101111)_2$

2's of $B = (00010001)_2$

A = 00011000

B = 00010001

00101001

Scanned with CamScanner

$A = (11011101)_2$ $B = (00010100)_2$

2's of B :- $(11101100)_2$

```
  1111
A = 11011101
B = 11101100
  ----------
1 | 11001001
  ----------
```

omit carry

Overflow:-

→ when the signs of the addends are different, there is no chance of overflow.

Multiplication:-

Unsigned binary integers:-

M - Multiplicand

Q - Multiplier

n - counter = no. of bits in Q

A ← temp reg to hold sum.

Ex:-

M = 1011 , Q = 1101

```
 0110   1011
 1011   0010
-----  -----
1 0001  1101
-----  -----
```

C

~~A = 1011~~

If right most bit of Q is 0, shift right the reg C, A, Q

① C=0 A = 1011 Q = 1101 M = 1011

Shift R A = 0101 Q = 1101

② $Q_0 = 0$ shift R C=0 A = 0010 Q = 1111

③ $Q_0 = 1$ Add A = 1101 Q = 1111 C=0

shift R C=0 A = 0110 Q = 1111

Scanned with CamScanner

④ C = 1 A = 0001 Q = 1111

shift R C = 0 A = 1000 Q = 1111

Product = A Q

= 1000 1111

M = 1101 Q = 1111 A = 0000

① C = 0 A = 1101 Q = 1111

shift R C = 0 A = 0110 Q = 1111

```
 0110
 1101
1 0011
```

② C = 1 A = 0011 Q = 1111

shift R C = 0 A = 1001 Q = 1111

```
 1001
 1101
1 0110
```

③ C = 1 A = 0110 Q = 1111

shift R C = 0 A = 1011 Q = 0111

```
 1011
 1101
1 1000
```

④ C = 1 A = 1000 Q = 0111

shift R C = 0, A = 1100 Q = 0011

P = 1100 0011

128+64+2+1

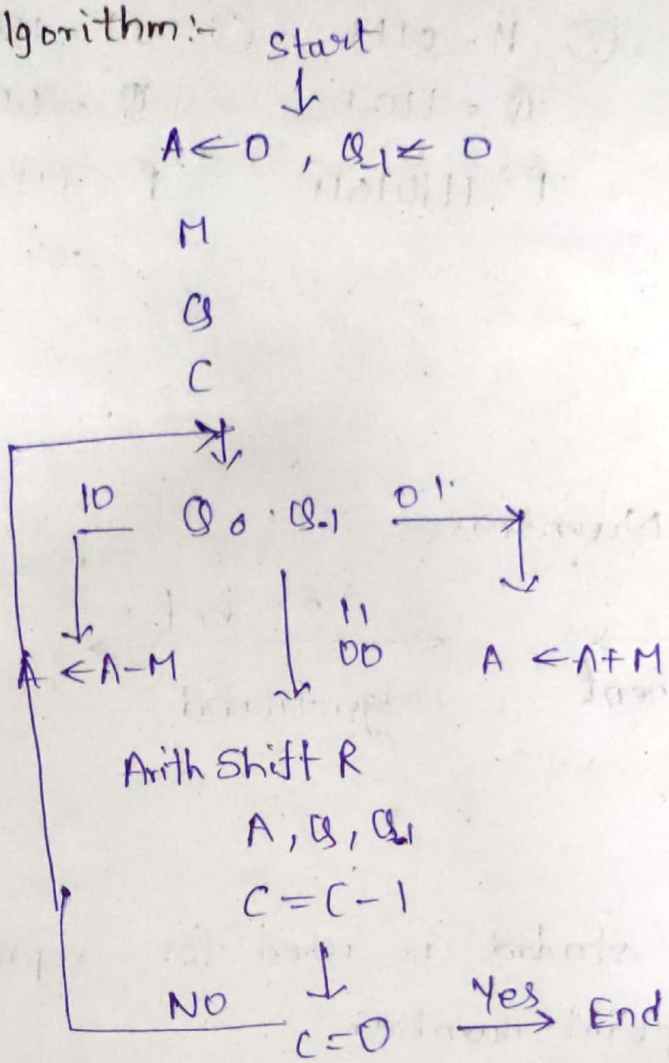
195

```
    1101
    1111
   1
    1101
   1101
  1101
 1101
11000011
```

```
    1101
    1101
1 00111
```

Scanned with CamScanner

Booth's Algorithm:-



→ * In Arithmetic shift Right, the sign bit is conserved.

M = 0111 Q = 0011

A = 0000 + 1001 = 1001

	A	Q	Q-1	M	
①	0000	0011	0	0111	Q_0Q_{-1} = 10 , A = A-M
A-M	1001	0011	0	0111	
ASR	1100	1001	1	0111	
② ASR	1110	0100	1	0111	Q_0Q_{-1} = 11
③ A+M	0101	0100	1	0111	Q_0Q_{-1} = 01
ASR	0010	1010	0	0111	
④ ASR	0001	0101	0	0111	Q_0Q_{-1} = 00

1110
0111
1 0101

Scanned with CamScanner

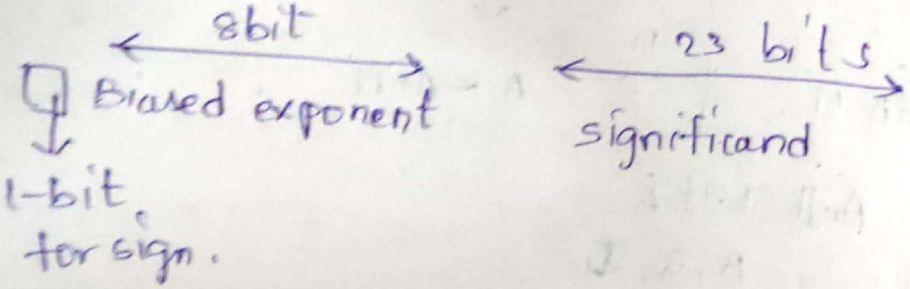
① 0111 = M
0011 = Q
P = 00010101

② M = 0111
Q = 1101
P = 11101011

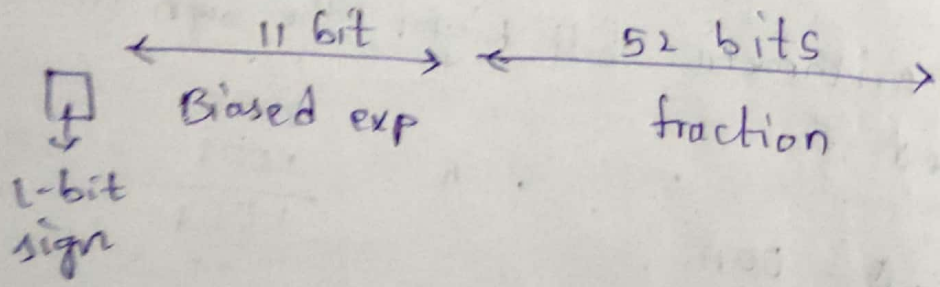
③ M = 1001
Q = 0011
P = 11101011

④ M = 1001
Q = 1101
P = 00010101

Floating point Numbers :-



→ IEEE 754 stndrd is used for representation of floating point number.



Scanned with CamScanner

① M = 0111 , Q = 0011 , A = 0000

	A	Q	Q_{-1}	M	
	0000	0011	1	0111	
① A.S.R	0000	0001	1	0111	$Q_0Q_{-1} = 11$
② A.S.R	0000	0000	1	0111	$Q_0Q_{-1} = 11$
③ A+M	0111	0000	1	0111	$Q_0Q_{-1} = 01$
A.S.R	0011	1000	0	0111	A = A+M
④ ASR	0001	1100	0	0111	Q_0Q_{-1} 00

	A	Q	Q_{-1}	M
	0000	0011	0	0111

0000
1001
1001

① A-M $Q_0Q_{-1} = 10$

② M = 0111 Q = 1101

0001
1001
1010

1
1100
0111
0011

	A	Q	Q_{-1}	M	
	0000	1101	0	0111	
① A-M	1001	1101	0	0111	$Q_0Q_{-1} = 10$
ASR	1100	1110	1	0111	
② A+M	0011	1110	1	0111	$Q_0Q_{-1} = 01$
ASR	0001	1111	0	0111	
③ A-M	1010	1111	0	0111	$Q_0Q_{-1} = 10$
ASR	1101	0111	1	0111	
④ ASR	1110	1011	1	0111	

∴ QM = 11101011

Scanned with CamScanner

③ M = 1001 Q = 0011

		A	Q	Q-1	M
		0000	0011	0	1001
①	A-M	0111	0011	0	1001
	ASR	0011	1001	1	1001
②	ASR	0001	1100	1	1001
③	A+M	1010	1100	1	1001
	ASR	1101	0110	0	1001
④	ASR	1110	1011	0	1001

QM = 11101011

④ M = 1001 Q = 1101

		A	Q	Q-1	M
		0000	1101	0	1001
①	A-M	0111	1101	0	1001
	ASR	0011	1110	1	1001
②	A+M	1100	1110	1	1001
	ASR	1110	0111	0	1001
③	A-M	0101	0111	0	1001
	ASR	0010	1011	1	1001
④	ASR	0001	0101	1	1001

QM = 00010101

Scanned with CamScanner

Pipelining:- (30-03-2022)

→ It means parallel execution.

→ when we execute prgms using pipelining, the no: of clock cycles taken to completely execute them is called pipelining. less

→ decompose sequential process into sub-operations where each sub-process executes on separate segment so that concurrent execution is possible.

→ This pipelining is implemented in processor.

→ Instruction

1) Fetch IF
2) Decode ID
3) Execute IE
4) Memory Access MA
5) Write back WB

} 5-stage pipeline

4-stages.

1) Fetch
2) Decode
3) Execute
4) Write Back

→ Instr can be divided into above seq.

→ These segments are called stages of pipeline.

→ Every stage has a separate dedicated reg.

Non-pipeline ← sequential execution of instr.

Ex:- Consider an assembly program with 5 instr which are to be executed on a 3 segment processor. Compare No: of cycles req using both pipelined and non-pipeline instr.

Scanned with CamScanner

→ Assume every instr is taking 3 clock cycles

→ So, without pipelining, the program takes 15 clock cycles.

Pipelining :-

→ With pipelining it takes lesser time.

→ It is found out using phase-time diagram

Phase-time diagram :-

	Segments			
Clock cycle	S_1	S_2	S_3	
1.	I_1	-	-	
2.	I_2	I_1	I_2	
3.	I_3	I_2	I_1	→ I_1
4.	I_4	I_3	I_2	→ I_2
5.	I_5	I_4	I_3	→ I_3
6.	-	I_5	I_4	→ I_4
7	-	-	I_5	→ I_5

For execution with pipelining, it takes less number of cycles.

Execution time & Speed up :-

→ A pipeline with k-stages

→ No: of instr executed = n

→ Non-pipelined : n x k cycles

→ Pipelined = n+k-1

Speed up = $\frac{nk}{n+k-1}$

Scanned with CamScanner

(2)

Instr	S_1	S_2	S_3	S_4
I_1	2	1	3	2
I_2	1	1	2	3
I_3	3	2	2	2
I_4	1	1	1	1

→ This gives how many clock cycles an instr spends in each seg.

Non-pipelining = 8 + 7 + 9 + 4

= 28

1	I_1	–	–	–
2	I_1	–	–	–
3	I_2	I_1	–	–
4	I_3	I_2	I_1	–
5	I_3	"	I_1	"
6	I_3	"	I_1	"
7	I_4	I_3	I_2	I_1
8	"	I_3	I_2	I_1 → I_1
9	–	I_4	I_3	I_2
10	–	"	I_3	I_2
11	–	"	"	I_2 → I_2
12	–	–	I_4	I_3
13	–	–	"	I_3 → I_3
14	–	–	–	I_4 → I_4

Scanned with CamScanner

Speed up = $\frac{14}{7}$ = 2

31-03-2022

Cycle time:-

→ Cycle time is value of one clock cycle.

Case-i:

All stages offer same delay

- cycle time = delay offered by one stage including delay due to register.

Case-ii:-

All stages don't offer same delay

- cycle time = Max delay.

A 4 stage pipeline has 150, 120, 160, 140ns respectively. Reg in each stage have a delay of 5ns. Assuming constant clocking rate. Total time to process 1000 datatems/ instructions on pipeline will be.

Total time for 1000 cycles

= (1 × 4 clock cycles + 999 × 1 clock cycles.

= 16595ns

Data hazards:-

→ Using pipelining reduces clock cycles.

→ DisAdv is possibility of hazards.

i) Data hazards

ii) Instruction/control hazards

iii) Structural hazards.

Scanned with CamScanner

Data Hazards:-

→ occurs when the pipeline is stalled because the data to be operated on is delayed due to some reason.

→ These hazards are mainly prone in decoding stage.

Ex:-

Consider A = 5 and two instr^n^s

I_1: A = 3 + A

I_2: B = 4 x A

The expected result is 32 in B

But due to abnormality, 20 is stored in B

→ If the instrn's are independent, then there will be no data hazards.

Instruction hazards:-

→ Pipeline is stalled due to unavailibility of instruction.

→ occurs during instruction fetch stage.

Control Hazards:-

→ These occur due to branching instr^n^.

Structural Hazards:-

→ when more than one instr requires the use of same hardware resource, then these hazards occur.

Scanned with CamScanner