Page No.
Date : 20/9/23

Unit-2

Computer Instructions

AND	BSA	CIR	SZE	SKO
ADD	ISZ	CIL	HLT	ION
LDA	CLA	INC	INP	IOF
STA	CLE	SPA	OUT	
BUN	CMA	SNA	SKI	
	CME	SZA		

Machine Language — binary (low level)

High-level Language — (software based lang)

Assembly level language (mnemonics based lang) — compiler / assembler

Machine Level (binary)

Difference b/w Machine & Assembly Language

	Machine Lang	Assembly Lang
1)	It is a low level progr-amming language consist of Binary numbers. It is also known as machine code or object code.	It is human only language that not understood by computer act a link b/w high level and -ne languages.
2)	No translator is required directly executed by CPU	An assembler is required to convert instructions into m code.

3)	It includes binary digits, hexadecimal and octal digits which can only be understood by computers	Mnemonics such as MOV, ADD, SUB etc. Make up the assembly language. Human can understand, utilize and apply this language
4)	Error fixing and modification cannot be done	It has a ability to correct errors and modify programs
5)	All data is present in binary format that makes it fast in execution.	As compare to machine lang execution spud of assembly lang is slow
6)	It is a first genration programing lang	It is a second genration programing lang
7)	It does not allow for modification	It can be modified easily
8)	There are more chances of error	There are few chances of error as compare to machine lang

Binary Program to add two numbers

Location	Ins. code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
110	0000 0000 0000 0000

Hexadecimal Program

Location	Ins
000	2004
001	1005
002	3006
003	7001
006	0000

DEC -4 Decimal Equivalent

Page No.
Date :

with Symbolic of^{n} code.

location	Instruction	Comments
000	LDA 004	load 1st operand in AC
001	ADD 005	Add 2nd operand to AC
002	STA 006	Store sum in location 006
003	HLT	Halt
006	0000	Store sum here
004	0100	
005	1100	

Assembly lang.

ORG	0	origin at 0
LDA	A	Load operand from A
ADD	B	Add operand from B
STA	C	Store sum in C
HLT		Halt
C, DEC	0	Sum stored in location C
END		end of Program

label

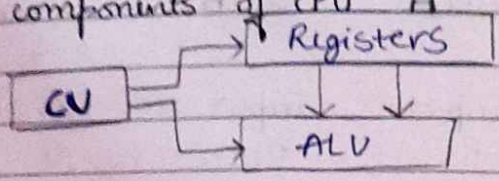
A, DEC -4

B, DEC -4

High level language (Python)

```
x = int (input (" Enter no. a :"))
y = int (input (" Enter no. b :"))
z = (x+y)
print (z)
```

Major components of CPU



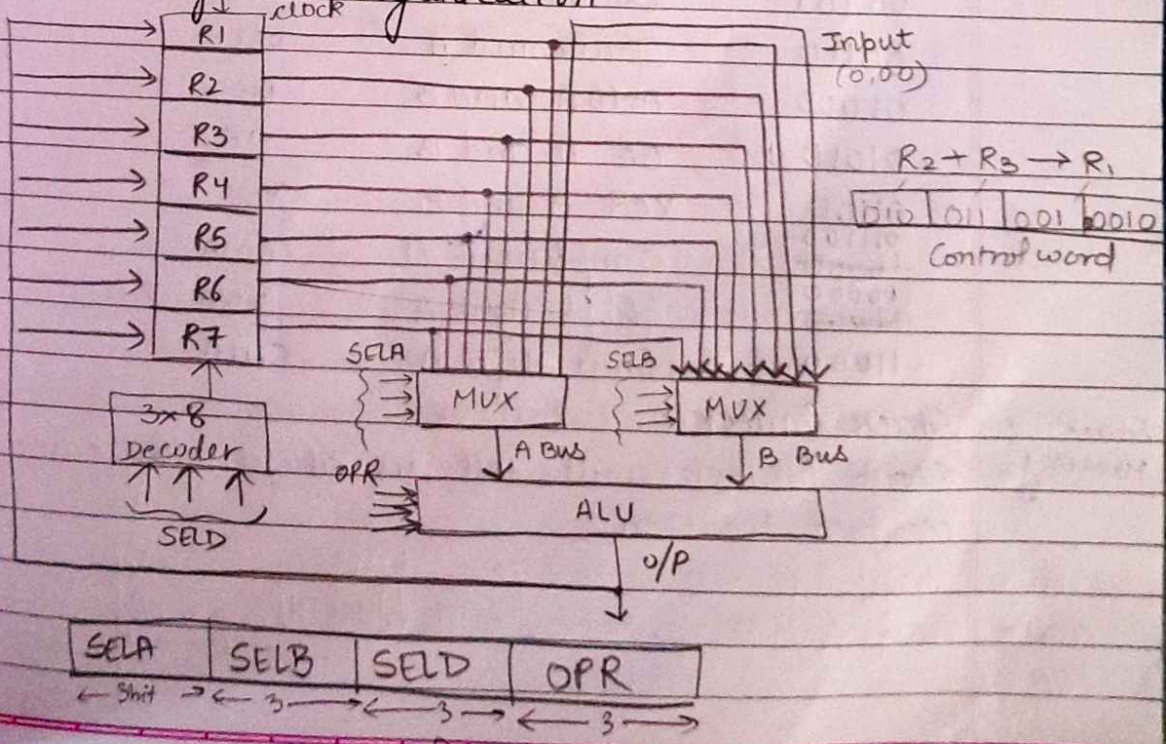
CPU organization

1. Single Accumulator organization → ADD X
2. General Register organization Ac ← Ac + M[x]
3. Stack Organization

1. Single accumulator organization
All operations are done ~~don~~ with the Accumulator

: ADD X
Ac ← Ac + M[x]

2. General Register organization



$R_4 - R_5 \rightarrow R_2$

100	101	010	00101

Page No.
Date :

Binary Code	~~SELA~~ SELA	SELB	SELD
	~~R1~~		
000	~~R2~~	Input	Input	None
001	~~R3~~	R1	R1	R1
010	~~R4~~	R2	R2	R2
011	~~R5~~	R3	R3	R3
100	~~R6~~	R4	R4	R4
101		R5	R5	R5
110		R6	R6	R6
111		R7	R7	R7

Encoding of ALU operations

OPR select	operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
~~01110~~ 01110	Complement A	COMA
~~10000~~ 10000	Shift right A	SHRA
~~11000~~ 11000	Shift left A	SHLA

Stack : Memory

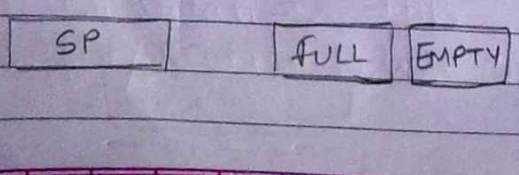
Stack Pointer

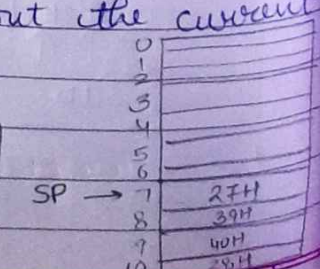
Stack Pointer will tell us about the current occupied location

LIFO

PUSH

POP





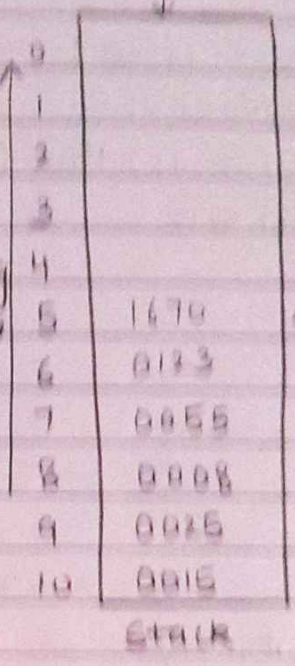
Push

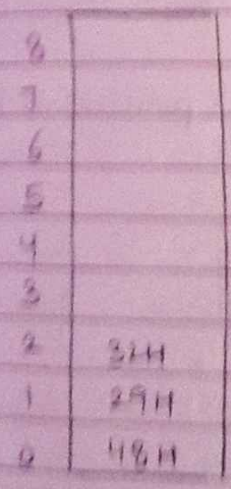
SP ← SP - 1

M[SP] ← DR

If (SP = 0) then (FULL ← 1)

Empty ← 0





SP = 2

PUSH

(3) SP = SP + 1

DR ← M[SP]

First increment then store the value

first store value then increment

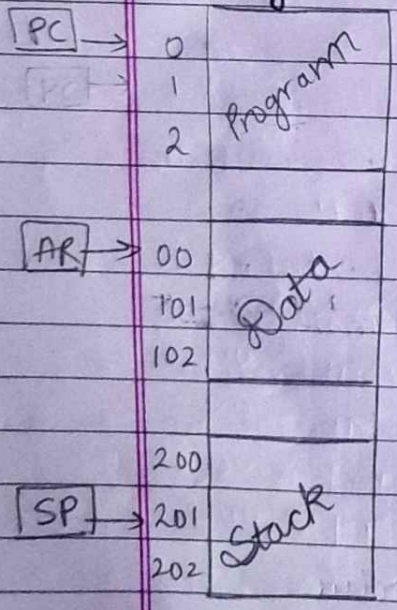
* POP

DR ← M[SP]
SP ← SP+1
If (SP=11) then (EMPTY←1)
FULL←0

	0	
	1	
	2	
	3	
	4	
→	5	1690
SP →	6	0623
	7	0055
	8	0008
	9	0025
Stack bottom →	10	0015

Stack

* Memory



Note: Data memory's pointer is kno
as AR

→ Reverse Polish Notation
* Infix Notation
A+B
* Prefix / Polish Notation +AB
* Postfix / Suffix / Reverse Polish Notation AB+

g: A * B + C * D $\xrightarrow{RPN}$ AB * CD * +

(2)(4) * (3)(3) * +

(8)(9) * +

(17)

g: (A+B) * [C * (D+E) + f]

AB+ C * * DE + + f

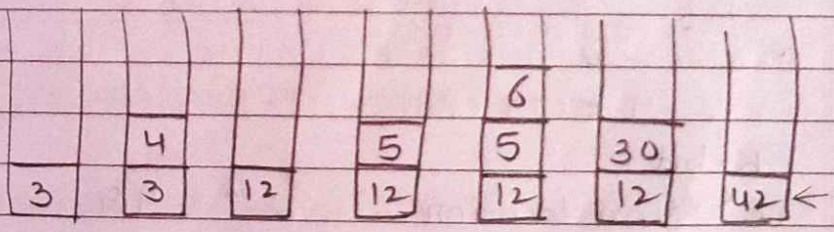
AB + C * DE + * + f

AB + CD * E + * f +

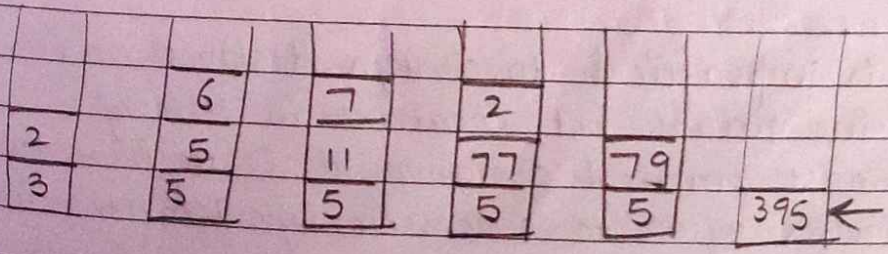
g: Stack op^n to evaluate 3 * 4 + 5 * 6

(3)(4) * (5)(6) * +

Push3
Push4
Mult
Push5
Push6
Mul
Add



j: (36) + (56+) 7 * 2 + *

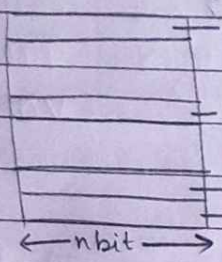


4th Oct 2023

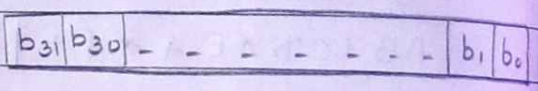
Page No.
Date :

Memory location, address & of n

- Memory consist of many millions of storage cells each which can store one bit.
- Data is usually accessed in 'n' bit groups and called word length.

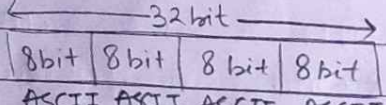


32 bit word length
$n = 32$



Signed Integer

b_{31} (sign) → +ve 0
-ve 1



4 ASCII char

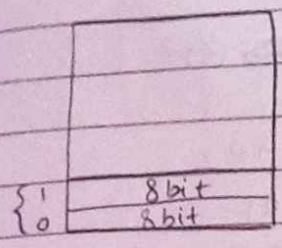
K-bit
2^K Mem location

$2^{10} = 1024 = 1K$
$2^{20} = 1M$
$2^{30} = 1G$
$2^{40} = 1T$

$2^{32} = 4G$
$2^{48} = 256T$
$2^{26} = 64M$

- It is impractical to assign distinct addresses to individual bit locations in Memory.
- The most practical assignment is to have successive addresses refers to successive byte locations in the memory
- It is known as Byte addressable memory

Byte addressable
1Byte = 8 bits



← 16 bit → 0, 2, 4, 6...
32 bit → 0, 4, 8, 12...
64 bit → 0, 8, 16...

Instruction format

- An operation code field that specifies the operation which is to be performed.
- An address field that designates a mem address or a processor register
- A mode field that specifies the way the operand or the effective address is determined.

op^n code	address	mode

- Three address instruction
- Two address instruction
- One address instruction
- Zero address instruction
- RISC instruction.

Three address instruction:

X = (A + B) * (C + D)

ADD R1, A, B (A+B) → R1
ADD R2, C, D (C+D) → R2
MUL X, R1, R2 R1 * R2 → M[X]

Advantage: Short programs when evaluating arithmetic operations.
Disadvantage: Binary coded ins require too many bits to specify 3 addresses.

Page No.
Date :

Two address instruction :

$X = (A+B) * (C+D)$

MOV	R1, A	M[A] → R1
ADD	R1, B	R1 + M[B] → R1
MOV	R2, C	M[C] → R2
ADD	R2, D	R2 + M[D] → R2
MUL	R1, R2	R1 * R2 → R1
MOV	X, R1	R1 → M[X]

One address instruction :

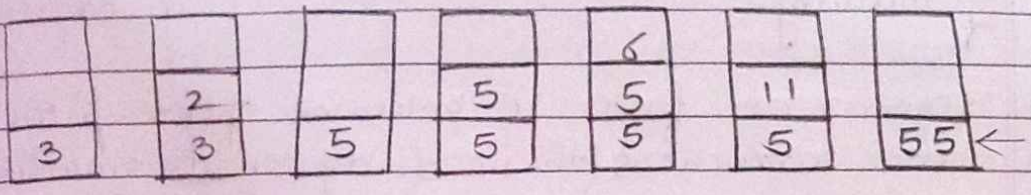
LOAD	A	M[A] → AC
ADD	B	AC + M[B] → AC
STORE	R1	AC → R1 or STORE T AC → M[T] (mem loc)
LOAD	C	M[C] → AC
ADD	D	AC + M[D] → AC
MUL	R1	AC * R1 → AC or MUL T AC * M[T] →
STORE	X	AC → M[X]

Zero address instruction :

- It use stack organized instruction set.
- The name zero address is given to this type of computer because of the absence of an address field in the computational instructions.
- However, push and pop instruction need an address field to specify the operand that communicate with stack.
- It is necessary to convert the expression into reverse polish notation for stack organized instructions.

Page No.
Date :

Ex: (3+2) * (5+6)



PUSH	A	A → TOS
PUSH	B	B → TOS
ADD		(A+B) → TOS
PUSH	C	C → TOS
PUSH	D	D → TOS
ADD		(C+D) → TOS
MUL		(A+B) * (C+D) → TOS
POP	X	TOS → M[X]

RISC instruction:

- Arithmetic operations are executed within the registers of CPU without referring to the memory.
- Load and store instructions are used to transfer the operands from memory to CPU register.
- There is no restriction on number of addresses used in the instruction

LOAD	R1, A	M[A] → R1
LOAD	R2, B	M[B] → R2
LOAD	R3, C	M[C] → R3
LOAD	R4, D	M[D] → R4
ADD	R1, R1, R2	R1 + R2 → R1
ADD	R3, R3, R4	R3 + R4 → R3
MUL	R1, R1, R3	R1 * R3 → R1
STORE	X, R1	R1 → M[X]

Page No.
Date :

Advantages of using Registers in instructions inp of mumory:

- Registers are faster i.e potential speed up the pro
- Shorter instructions (the no. of register is smaller; for 32 registers need 5 bits)
- Minimize the frequency with which the data is moved back and forth b/w the memory and processor register.

9/10/23 Addressing Modes

- It refers to the way in which the operand of an instruction is specified.
- The addressing mode specifies a rule for interpreti or modifying the address field of the instruction before the operand is actually executed

1) Implied addressing Mode:

- In this mode the operand is specified in the in itself.
- Zero address instructions are designed with impli addressing modes.
- Eg: CMA → Complement Accumulator
  CLC → Clear carry flag

ins.

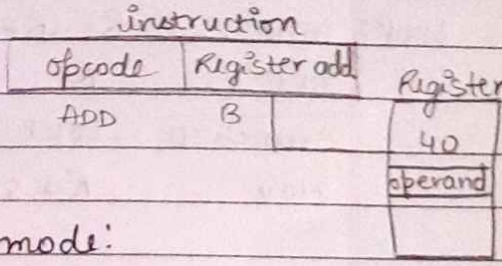
Data

2) Immediate addressing mode:

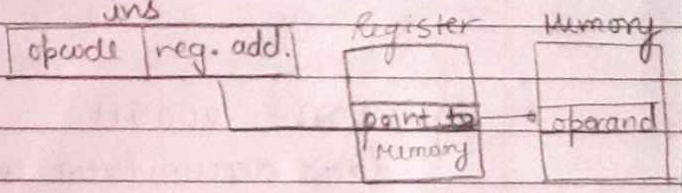
- Operand is a part of instruction
- No memory refurence to fetch data
- It is fast.
- It has limited range
- Eg: ADD 5

opcode	operand
ADD	5

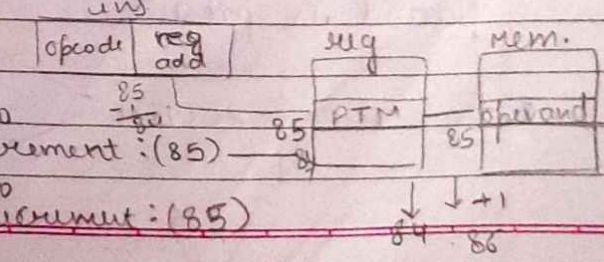
3) Register addressing mode:
- Operand is held in register named in address field
- Limited number of registers, so very small address field needed
- It has shorter instruction, so faster instruction execution.
- Eg: ADD B



4) Register indirect addressing mode:
- Operand is in memory cell pointed to by the content of Register 'R'.
- It has large address space
- Eg: ADD M[B]



5) ~~Addressing~~
5) Auto increment & auto decrement addressing mode:
- Auto increment mode is very similar to register indirect mode. The only exception is that the effective address of the operand is the content of a register specified in the instruction and after accessing the operand the content of this register are automatically incremented to point to the next item in a list.
- The increment is one for 8-bit operand, 2 for 16-bit operand and 4 for 32-bit operand.
- Auto decrement mode:
  - *Decrement first: Same as auto increment

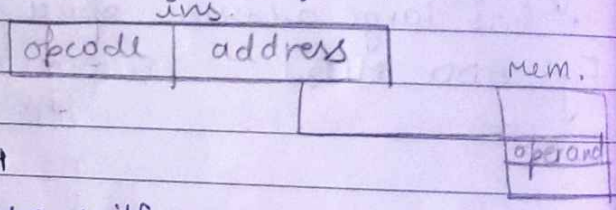


Eg of Auto increment :

```
        MOV   R1, N
        MOV   R2, NUM1
        CLR   R0
LOOP:   ADD   R0, (R2)+, R0
        DEC   R1
        Branch>0  LOOP
        MOV   R0, SUM
```

6) Direct addressing mode

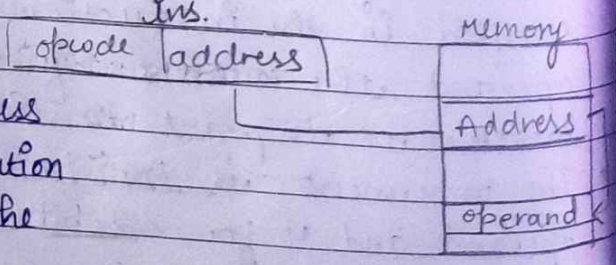
- Address field contains address of operands



- Eg: LDA 2005H
  load accumulator with a content available at 20 mem location

7) Indirect addressing mode

- Memory all pointed to by address field contains the address of the operand.

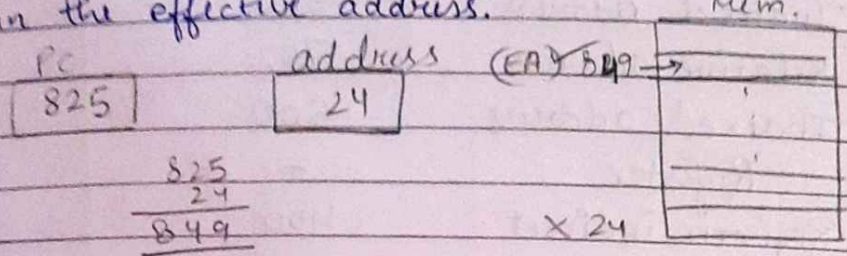


- Effective address: (EA) It is the address of exact memory location where the value of the operand is present.

- Eg: LDA 2005H → I(0) Direct
  → I(1) Indirect

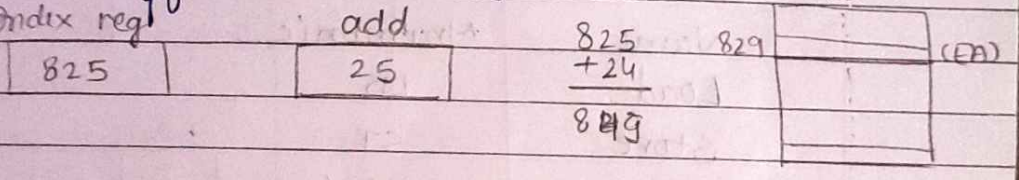
Relative addressing mode

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.



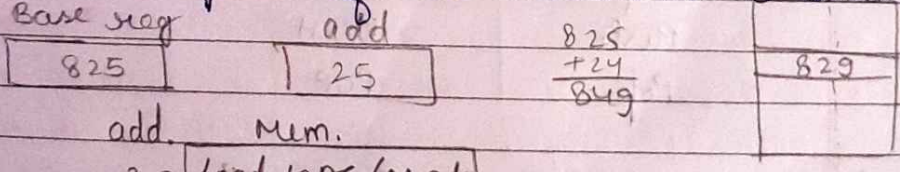
Index addressing mode.

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

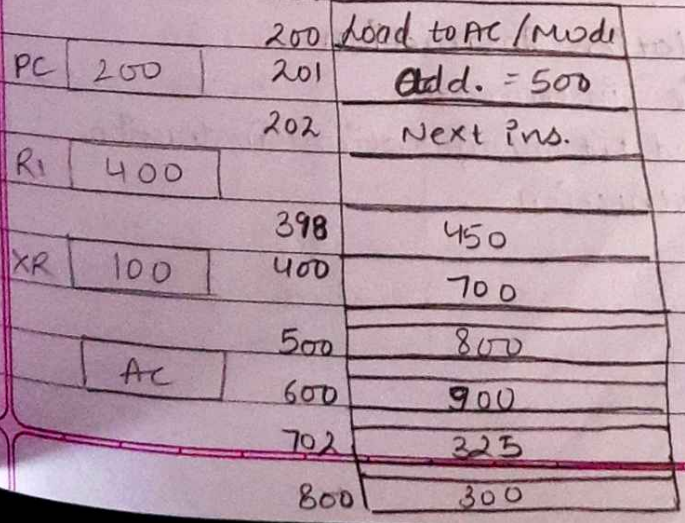


Base register addressing Mode.

- In this mode the content of a base register is added to the address part of the instruction to obtain EA.

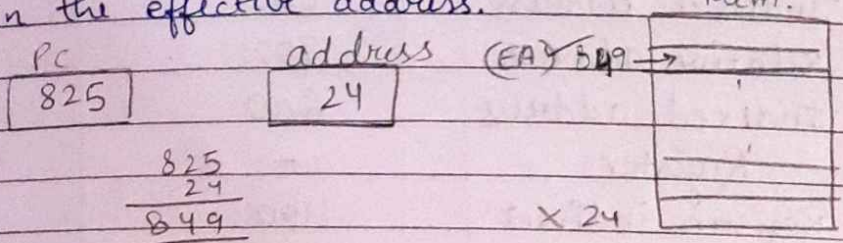


Ex:



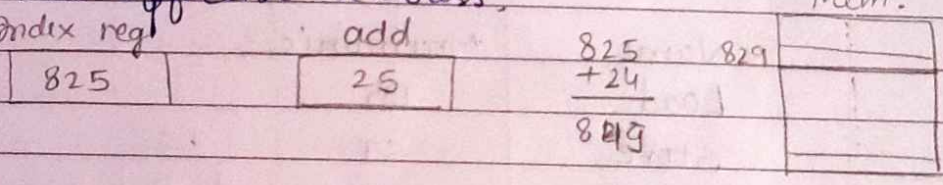
8) Relative addressing mode

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.



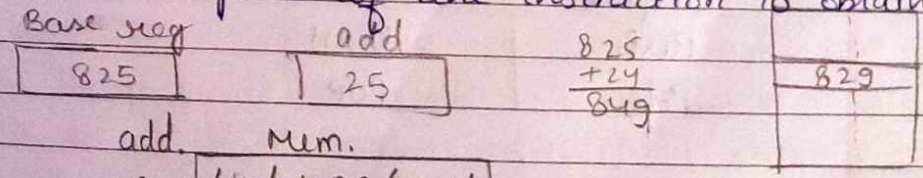
9) Index addressing mode.

- In this mode the content of an index register is added to the address part of the Instruction to obtain the effective address.

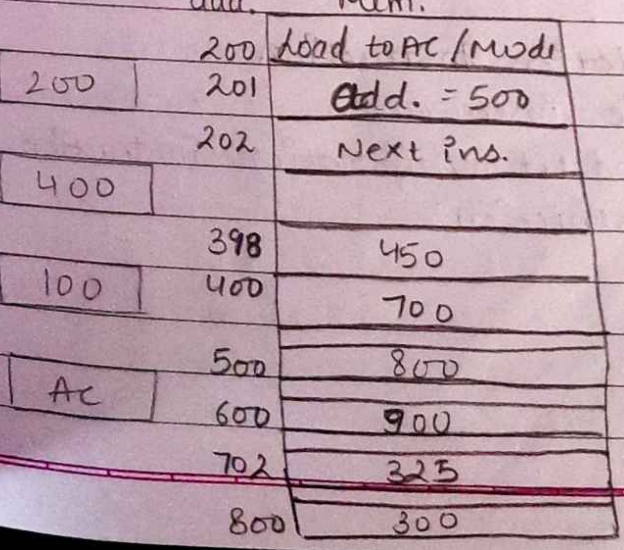


10) Base register addressing Mode.

- In this mode the content of a base register is added to the address part of the instruction to obtain EA.



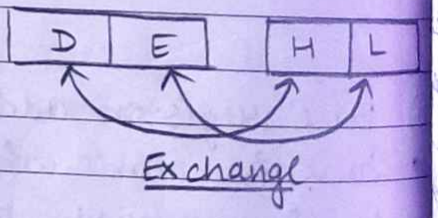
Ex:



Addressing mode	Effective address	Content of AC
Direct Address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	500 PC 202 702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Auto increment	400	700
Auto decrement	399	450

1) Data Transfer instruction

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP



Exchange

2) Data Manipulation Instruction
   - → Arithmetic instruction
   - → Logical and bit manipulation instruction
   - → Shift instruction

Page No.
Date :

* Arithmetic instruction

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
add with Carry	ADDC
Subtract with Borrow	SUBB
Negate (2's comp)	NEG
Add two binary int numbers	ADD I
Add two floating point numbers	ADD F
Add two decimal numbers in BCD	ADD D

* Logical and bit manipulation instruction

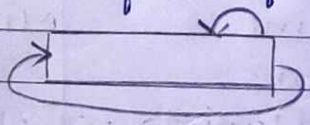
Name	Mnemonic
Clear	CLR
Complument	COM
and	AND
OR	OR
Exclusive ~~Not~~ OR	XOR
Clear Carry	CLRC
Set Carry	SETC
Comp. Carry	COMC
Enable Interupt	EI
Disable Interupt	DI

* Shift Instruction.

Name	Mnemonic
Logical shift right	SHR
Logical Shift left	SHL
Arithmetic shift right	SHR A
Arithmetic shift left	SHL A
Rotate Right	ROR
Rotate Left	ROL
Rotate Right through carry	ROR C
Rotate Left through Carry	ROL C

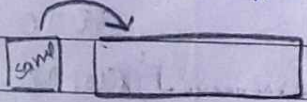
Logical Shift:

Logical shift insert zero to the end bit position. The end position is the left most bit for shift right and right most bit position for shift left



Arithmetic Shift:

Arithmetic shift right instruction must preserve the sign bit in left most position. The sign bit is shifted to the right but the sign bit itself remains unchanged.



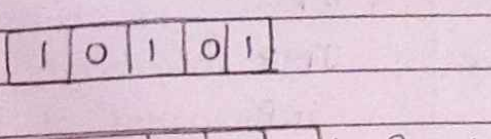
Eg: 1 00001110

1 10000111

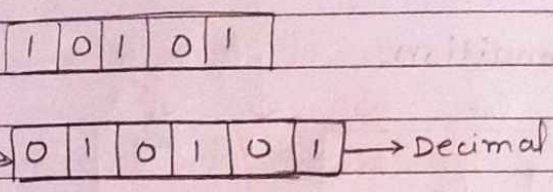
Arithmetic shift left instruction insert zero to end position and is identical to the logical shift left instruction.

Page No.
Date :

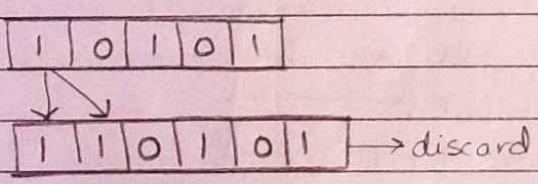
Logical Shift left
Arithmetic Shift left



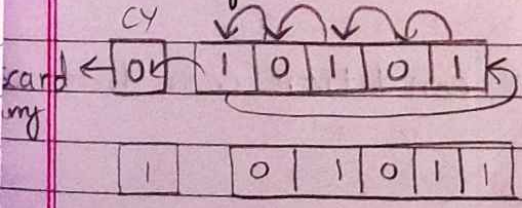
Logical Right



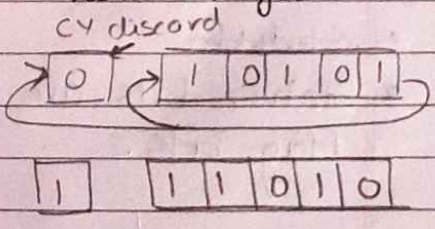
Arithmetic Right



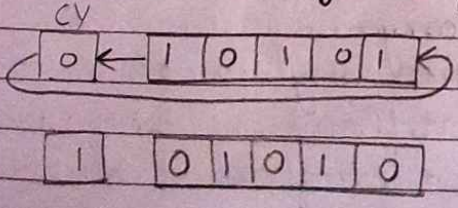
Rotate left



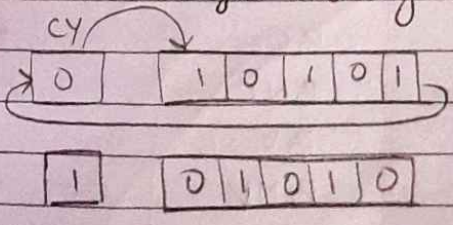
Rotate Right



Rotate Left through Carry



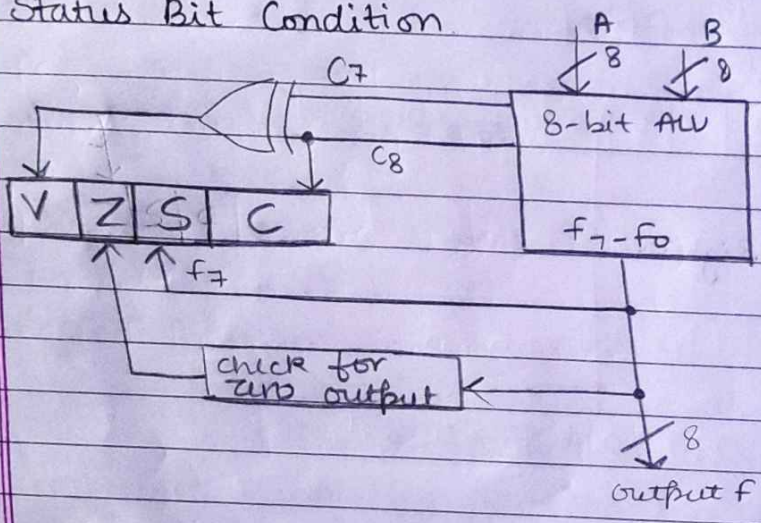
Rotate Right through Carry



3) Program Control Instruction

Name	Mnemonics
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
return	RET
compare	CMP

- Status Bit Condition



V: overflow
Z: zero
S: Sign
C: carry

- ~~Status bits~~
- Status bits are called as condition code bits or Flag bits.

BZ	Branch if zero
BNZ	" not zero
BC	" carry
BNC	" not carry
BV	" overflow
BNV	" not overflow
BP	" positive
BM	" negative

Page No.
Date :

Subroutine Call and return

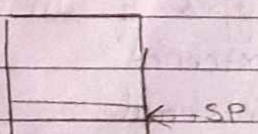
A subroutine is self contained sequence of instructions that perform certain computational task, during a execution of a program a subroutine may be called to perform its functions many times at various ~~time~~ points in main program.
Each time a subroutine is called a branch is executed to the beginning of the subroutine and after the subroutine has been executed branch is made back to the main program.

Two operations which is to be done in subroutine:

1) The address of the next instruction available in PC is stored in a temporary location. So the subroutine know where to return.
2) Control is transferred to the beginning of Subroutine

Temporary loc : Stack mem.

~~SP←~~ SP → SP-1
PC → M[SP]
PC ← EA (branch)
PC ← M[SP]
SP → SP+1



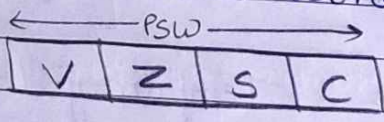
2/23 Program Interrupt

- Program interrupt refers to the trasfer of program control from a currently running program to another service program as a result of an external or internal generated request.
- Interrupt procedure is quite similar to a subroutine call except for three variations:

1) The interrupt is usually initiated by an Internal or exter signal, rather than from the execution of an instruction
2) The address of the interrupt servicing program is determ by the hardware rather than from the address fi of instruction
3) An Interrupt procedure usually stores all the informa necessary to define the state of the CPU rather tha storing only the Program Counter.

The information which is to be stored is content PC, content of all processor registers and content of certain status conditions.

Program Status Words (PSW)
The collection of all status bit conditions in CPU called Program Status Word



Types of Interrupt:
1) External
2) Internal
3) Software

External Interrupt:
It comes from I/O devices, from a timing device, from circuit monitoring the power supply and from any other external source
Eg; I/O device requesting transfer of data, I/O finish transfer of data, Elapsed time of an event or power failure.

Page No.
Date :

Internal Interrupt:
It arises from erroneous of an instruction or data. Also known as 'Traps'
Eg: Register overflow, Attempt to Divide by Zero, Invalid op code, Stack overflow and protection violation

Difference b/w internal and external interrupt is that: the internal interrupt is initiated by some exceptional con" caused by the program itself rather than by an external event

Software Interrupt:
A Software interrupt is initiated by executing an instruction.
It is a special call ins. that behaves like a interrupt rather than a subroutine call.
It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.
Eg; INT (interrupt)

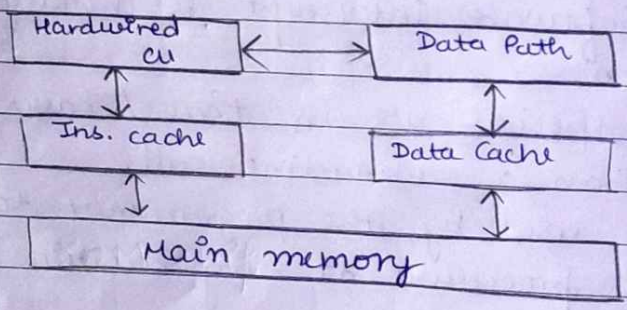
RISC
It has a simple collection and highly customized set of instruction. It is build to minimize the instruction execution time by optimizing and limiting the no. of instruction.
Each ins. cycle requires only one clock cycle and each cycle contains 3 parameters: Fetch, Decode & Execute
It is used to perform various complex instruction by combining them into simpler one.

Features of RISC processor:

- One cycle execution time
- By Pipelining technique
- A large no. of registers
- It supports simple addressing mode & fixed length of ins for executing the pipeline.
- If you deduce load & store ins to access the mem. ~~ins~~ location

(Google 4-4 points) Advan / Disad of RISC processor:

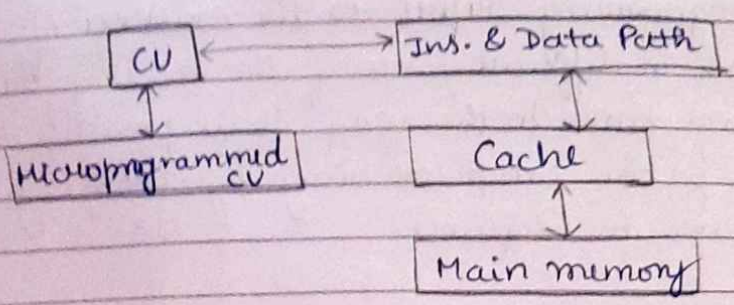
RISC Architecture:



CISC Features:

1) The length of code is short so it requires very RAM
2) CISC may take longer than a single clock cycle to execute the code.
3) Less ins is needed to write an application
4) Provides easier programming
5) Supports for complex data structure
6) It is composed of fewer registers & more addr Modes

CISC Architecture:



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Ins}}{\text{Program}} \times \frac{\text{cycles}}{\text{Ins}} \times \frac{\text{Seconds}}{\text{cycles}}$$

CISC ↑ RISC ↓ RISC ↑ CISC ↓

RISC

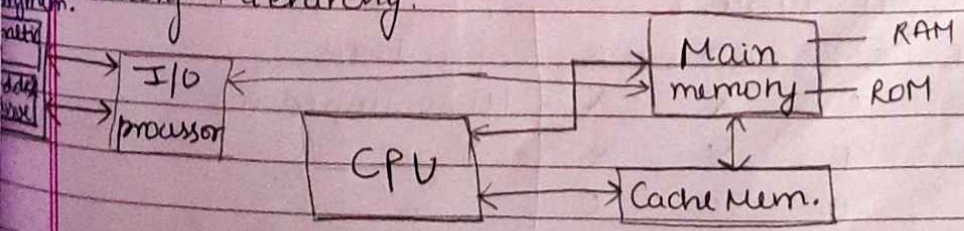
It reduce the cycle per instruction at a cost of the no. of instructions per program.

CISC

It attempts to minimize the no. of instructions per program but at the cost of an increase in the no. of cycles per instruction.

t-1) Differna b/w RISC, CISC
(4) Advan & Disadvan of CISC

23 Memory Organication
Memory Hierarchy.



need of sec mem:
1) cost less
2) consistency more

Page No.
Date :

Multiprogramming

Multiprogramming refers to the existence of 2 or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence.

Morris Mano Ch-12

Main memory

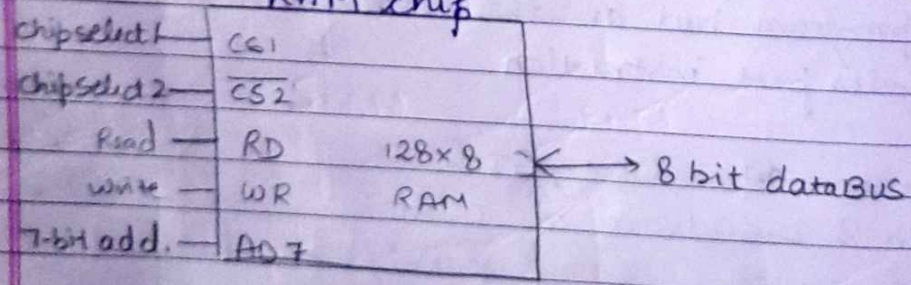
RAM (volatile) (SRAM, DRAM)
ROM (nonvolatile) (ePROM etc)

Bootstrap loader

The ROM portion of main mem. is used for storing an initial program is called a Bootstrap loader.

RAM and ROM chips

RAM chip

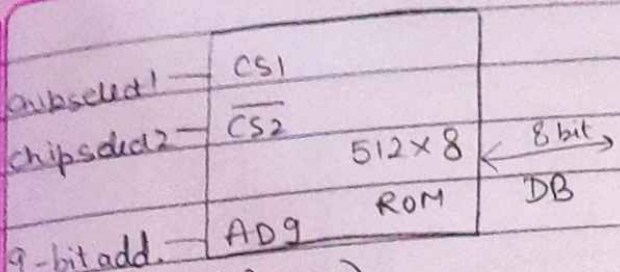


Function Table

CS1	CS2	RD	WR	operations
0	0	X	X	High Impedence
0	1	X	X	"
1	0	0	0	"
1	0	0	1	Write in RAM
1	0	1	0	Read from RAM
1	1	X	X	High impedence

Page No.
Date :

ROM chip



$(2^9 = 512)$

CS1	CS2	Op^n
0	0	HI
1	0	Read
1	1	HI
0	1	HI.

Memory address Map

Components	Hexadecimal address	Address 10 9 8 7 6 5 4 3 2 1
RAM 1	0000-007F	0 0 0 x x x x x x x
RAM 2	0080-00FF	0 0 1 x x x x x x x
RAM 3	0100-017F	0 1 0 x x x x x x x
RAM 4	0180-01FF	0 1 1 x x x x x x x
ROM	0200-03FF	1 x x x x x x x x x

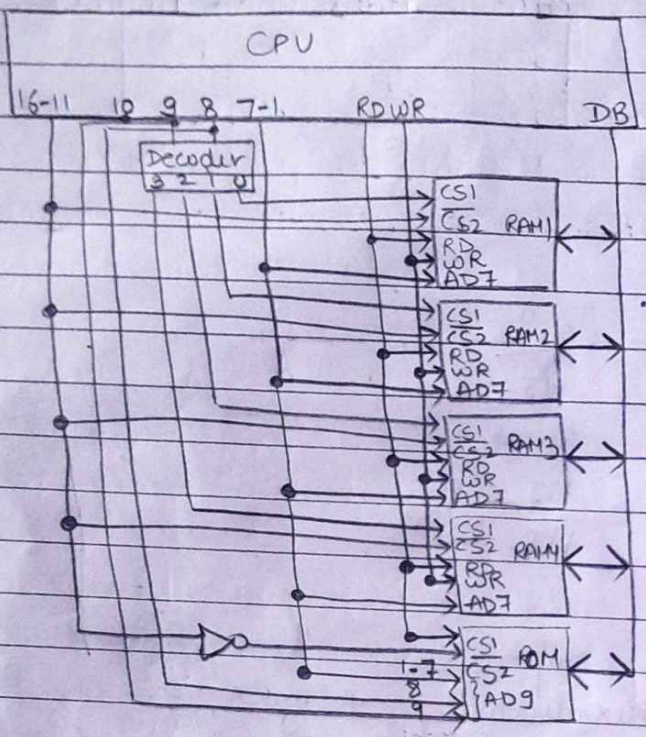
Put all zero's & all one's in x

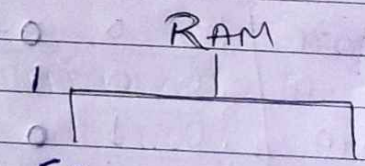
The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. This table is called memory address map

25/10/23

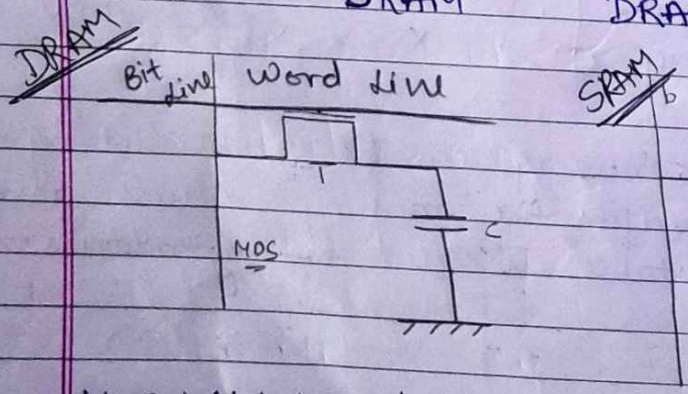
Page No.
Date :

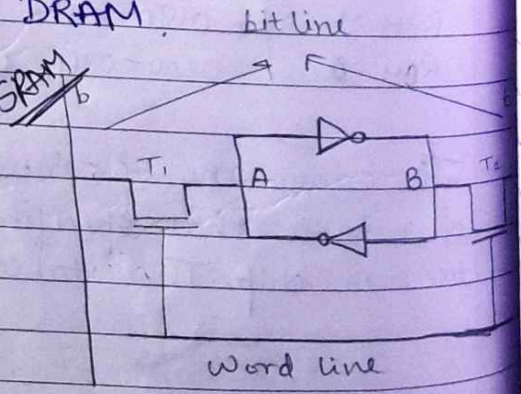
Memory Connection to CPU





SRAM DRAM

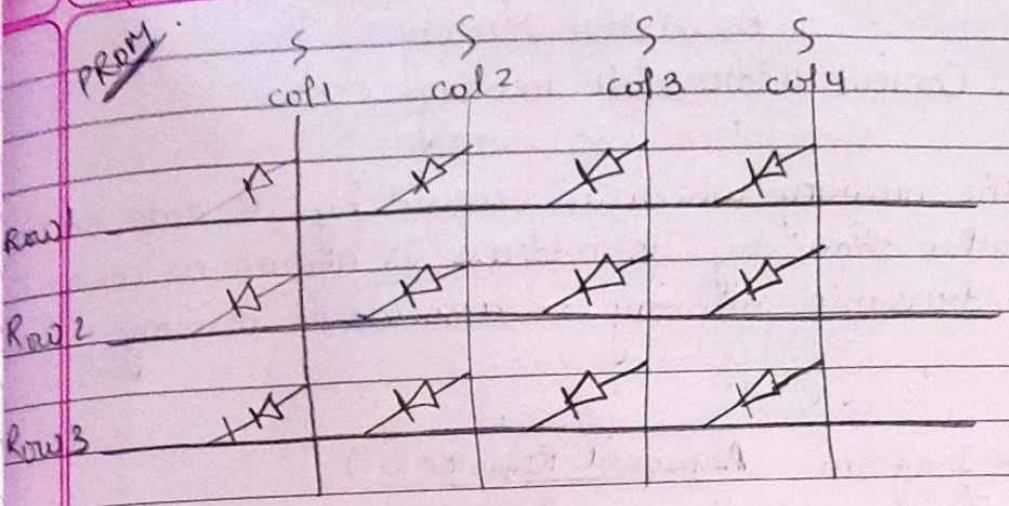


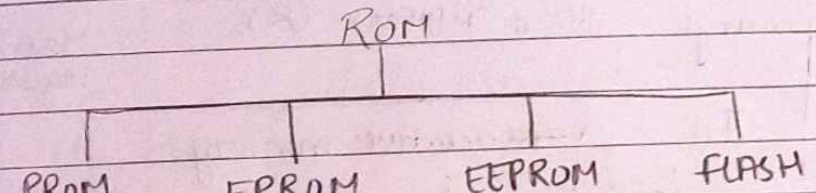


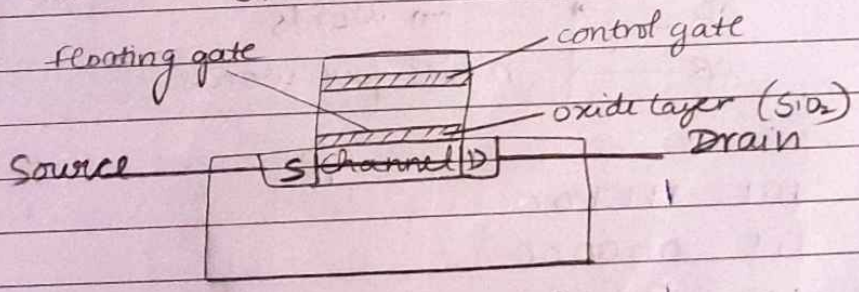
MOS : Metal Oxide Semiconductor.

~~Difference b/w SRAM & DRAM~~

PROM

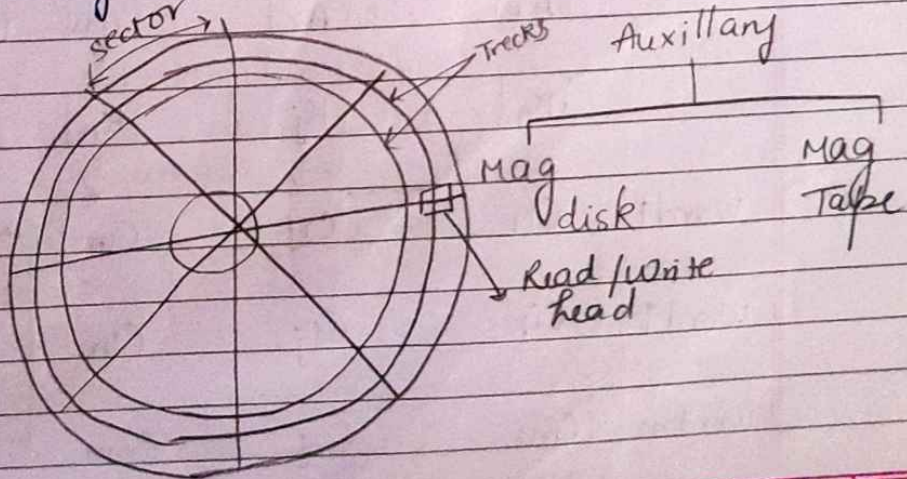






EPROM : Erasable PROM (erasing using UV)

EEPROM : Electrically Erasable PROM (tunneling)



30/10/20

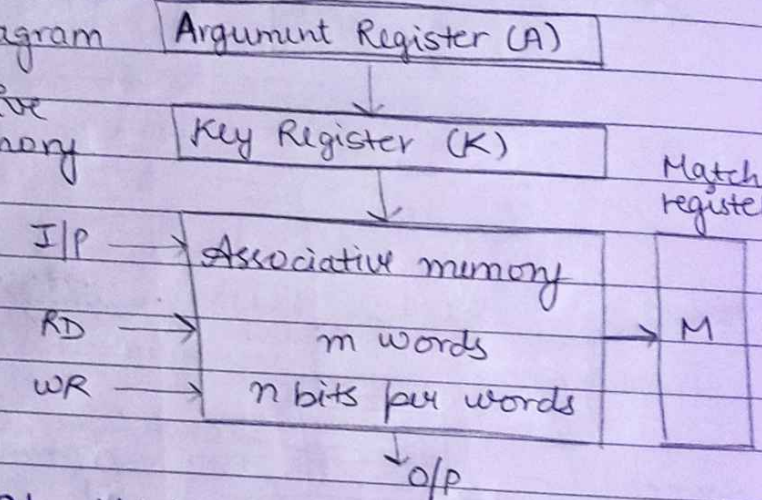
Page No.
Date :

Associative Memory

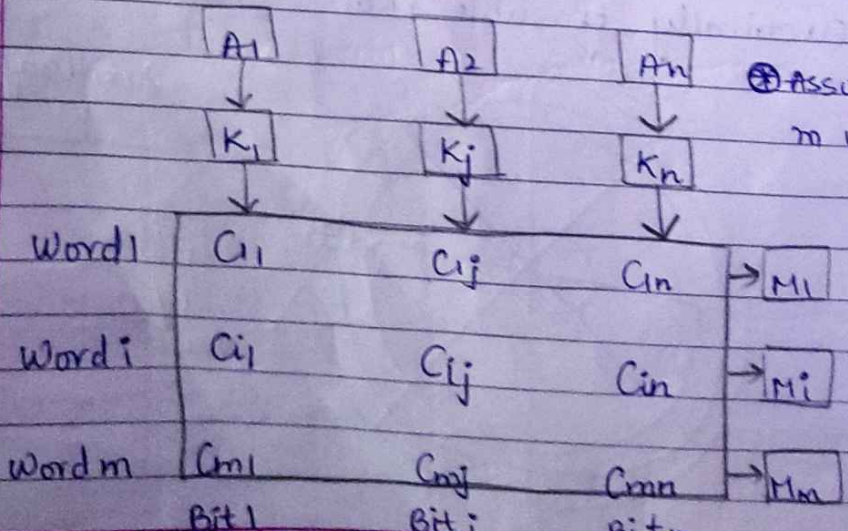
CAM : Content Addressable memory
(accessed by its data)

↳ The memory which is accessed by its data or co
rather than by its address is known as content
addressable memory or associative memory.

Block Diagram of associative memory

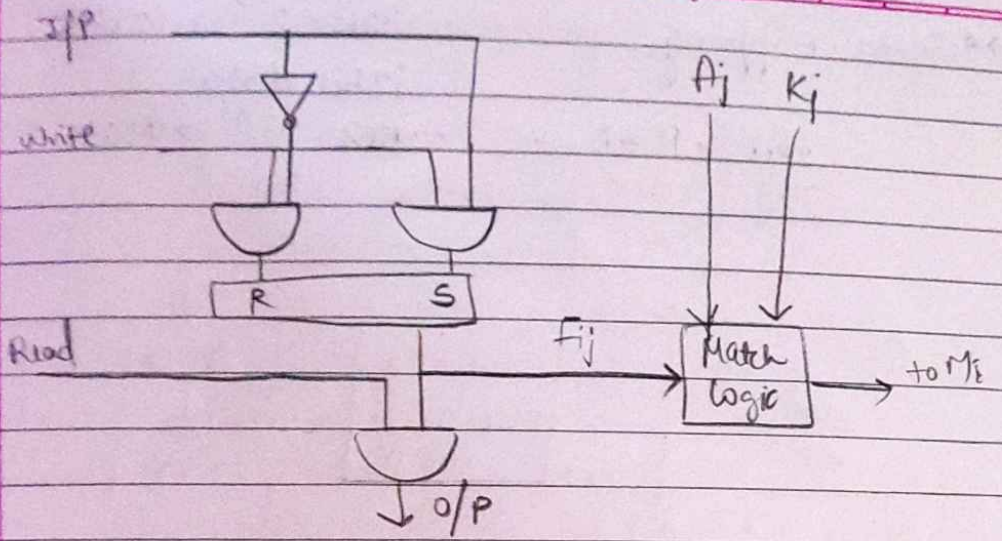


A	101	111100
K	111	000000
Word1	100	111100
Word2	101	000001



* Associative memo
m word, n cells
word.

One cell of associative memory



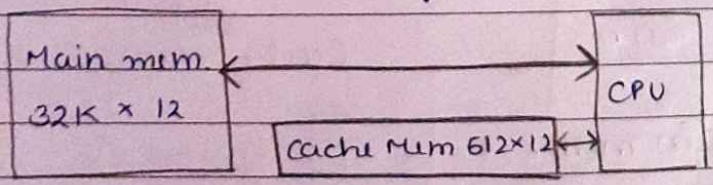
Cache Memory.

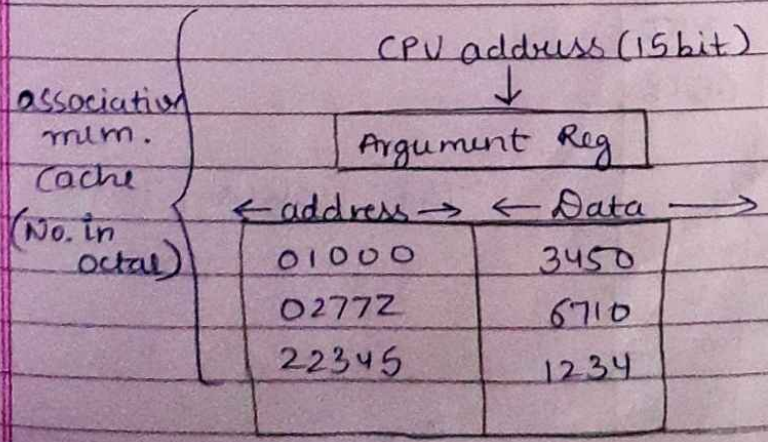
→ locality of reference
→ hit ratio
→ mapping

hit / miss → $\frac{\text{No. of hits}}{\text{No. of hits} + \text{no. of miss}} \times 100$

↳ The transformation of data from main memory to cache memory is known as mapping.

↳ • Associative Mapping. → Drawback: Cost is more.





: copy comp block instead of single data

→ Drawback: searching is more

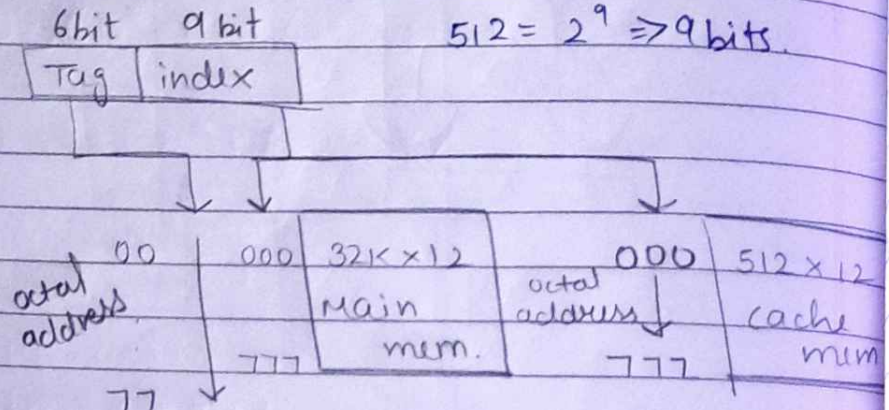
Page No.
Date :

↳• Direct Mapping.

$32K = 2^{15} \Rightarrow 15$ bits

12 bit data

$512 = 2^9 \Rightarrow 9$ bits.



Mem. add | Memory Data

00000	1220
00777	2340
01000	3450
01777	4560
02000	5670
02777	6710

Main mem.

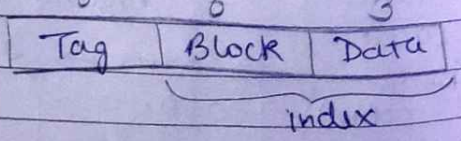
Single Block Mapping

Index add.	Tag	Data
000	00	1220
777	02	6710

(4 full use FIFO

Cache Mem.

	Index	Tag	Data
Block 0	000	01	3450
	007	01	6578
Block 1	010		
	017		
Block 63	770	02	
	777	02	6710



↳ • Set Associative Mapping fig 12-15 MM

* Writing into Cache
  ↳ Write Through
  ↳ Write Back

* Virtual Memory

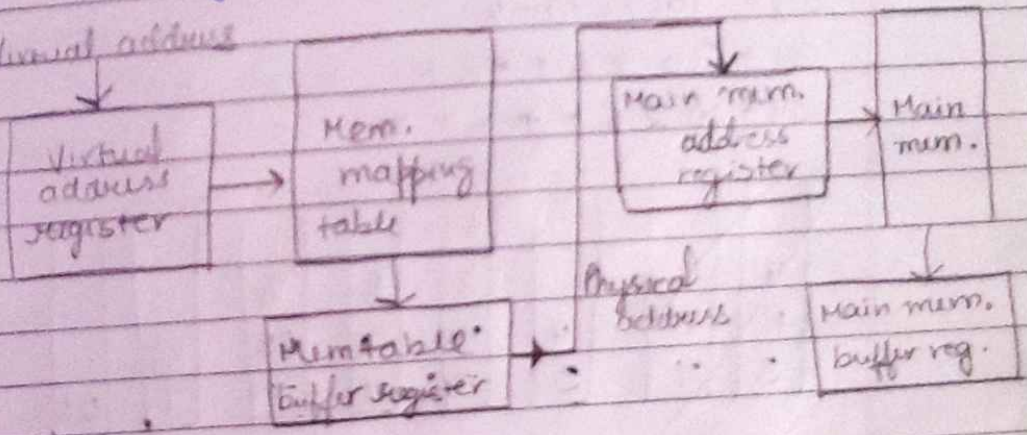
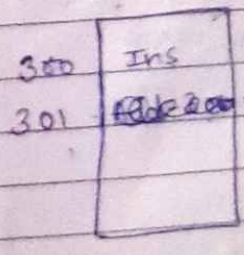


fig 12-18
fig 12-19
↓ page fault
fig 12-20

Q
a) 400
b) 301
c) 702
d) 200 + content at 200
e) 600

R1
200



a) 400
b) 300
c) 700
d) 200
e) 500

Q
A * B + C * D + E * f
AB* + CD* + Ef*
AB* CD* + Ef* +

Q
A * B + A * (B * D + C * E)
A * B + A * (BD* CE* +)
AB* + ABD* CE* + *
AB* ABD* CE* + * +

Page No.
Date :

Q1 $A * [B + C * (D + E)] / F * (G + H)$

$A * (B C (D * + E +) / F * G H +$

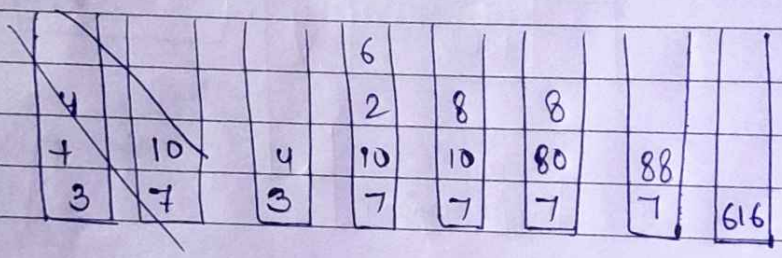
$A B C (C D * + E + * F / G H + * /$

Q2 $(3+4) * [10 * (2+6) + 8]$

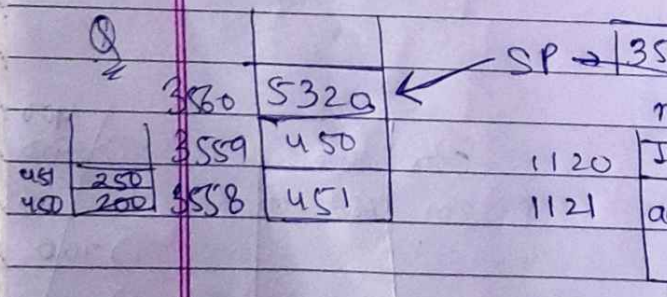
$34 + 10 * (26+) + 8$

$34 + 10\ 26 + * 8 +$

$34 + 10\ 26 + * 8 + *$



Q3



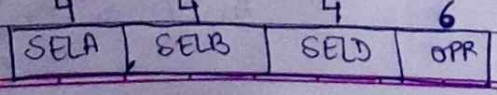
Q4 a) 32 multiplexers each of size 16:1

b) 4, 4

c) input : 4
output : 16

d) input : 65
output : 33

e)



140ns

R1 ← R2 + R3

010 011 001 00010

R4 ← R4

~~100 100 00000~~ 100 xxx 100 00000

R5 ← R5 - 1

101 xxx 101 00100

R6 ← shl R1

001 xxx 110 11000

R7 ← input

000 xxx 111 00000

R3 ← R1 - R2

Output ← Input

R2 ← R2 ⊕ R2 OR R2 ← 0

R0 ← R0 + R1

R3 ← R7 shr R4