

Basic Computer Organization

Main parts of computer system

- Processor: executes program
- Main memory: holds program & data.
- I/O devices: for communication with outside.

Machine instruction:

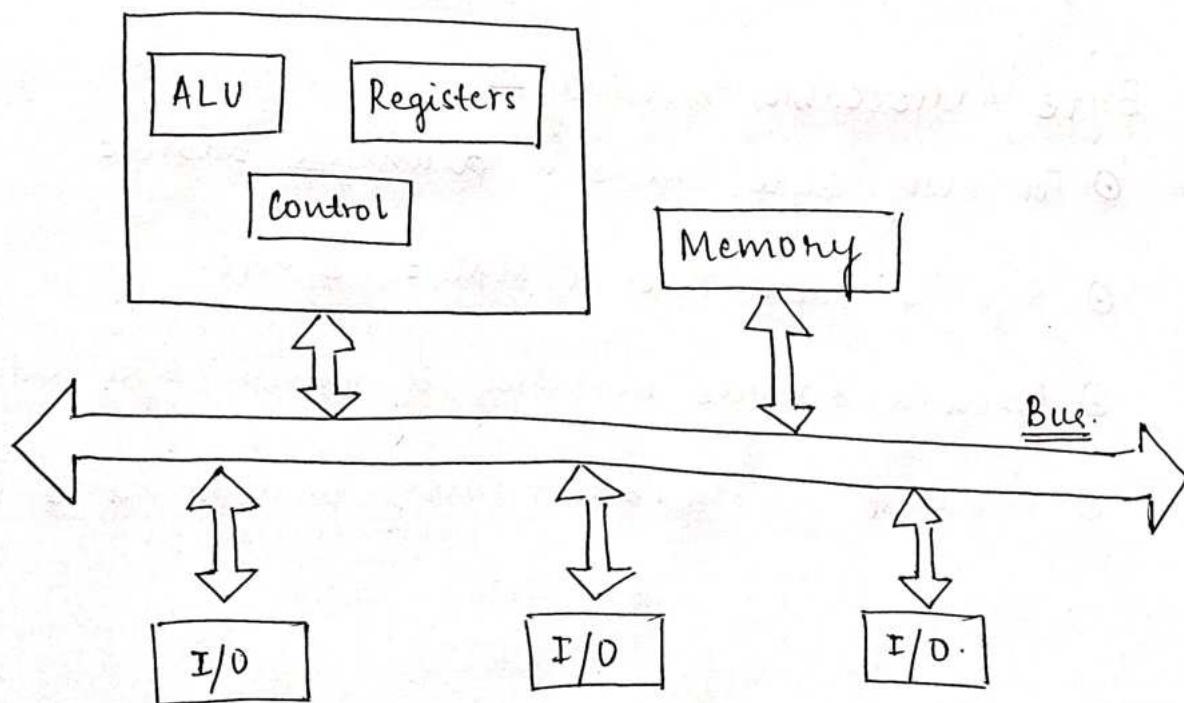
Description of primitive operation that machine hardware is able to execute.

inbuilt in h/w

e.g. ADD 2 integers.

Instruction set:

Complete specification of all the kinds of instructions that the processor hardware was built to execute.



Main Memory (RAM)

- ↳ divided equally into small units called cells.
- ↳ can be byte addressable or word addressable.
- ↳ cell is uniquely identified by binary no. called address.
- ↳ Representation of memory chip : $\langle \text{number of cells} \rangle \times \langle \text{size of each cell} \rangle$
$$2^k \times x.$$

- ① Main memory is a collection of storage location, each with a unique identifier called address.
- ② A program (data & instruction) should be stored in main memory in ~~seq~~ order to get executed.

Address space

- | ① Total no. of uniquely identifiable locations in memory is called address space.

Byte Addressable memory -

- ① For every byte, there is ~~a~~ unique address
- ② Size of each cell in M.M. = 8 bits.
- ③ When $x=8$, the memory is called byte addressable.
- ④ Smallest identifiable memory location is 1 byte.

Word Addressable Memory

Word - Concept

64 bit CPU } 64 and 32 are word size.
32 bit CPU } (word length of CPU)

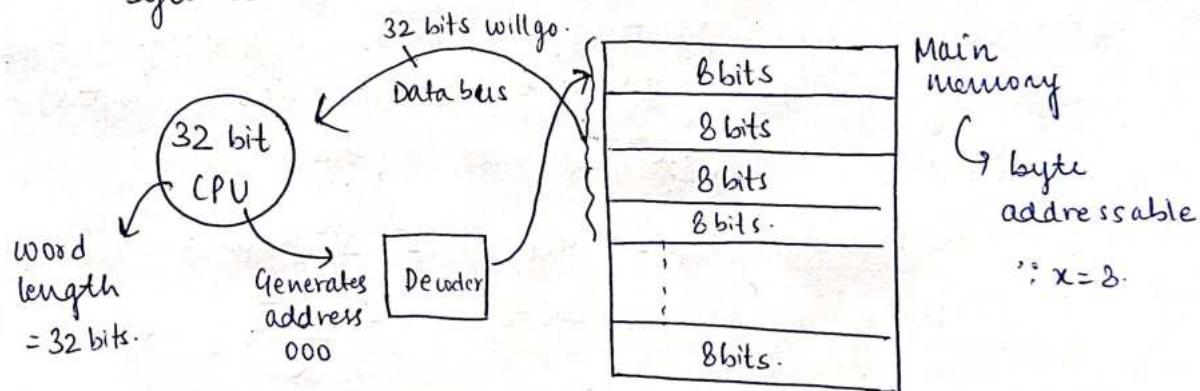
A word can be stored or retrieved in a single operation.

Word

- Word is a property of the Processor (CPU)
- fixed size data that is considered as a unit by the processor.

Word size refers to the number of bits processed by a computer's CPU in one go.

CPU does not care memory is word addressable / byte addressable.



A single bit represents a very small amount of info so bits are not handled individually.

The usual approach is to deal with them in groups of fixed size (called word length).

When cell capacity is 1 word, then the memory is word addressable.

Word length depends upon CPU

∴ memory is generally designed as byte addressable (so that it can be paired with any CPU).

(If CPU word = 32 bits and M.M. word = 64 bit then, cannot be used.)

Memory can be byte addressable. But, if 32 bit CPU is used, it requires 32 bit of info in one go.
 $\therefore \# \text{data lines} = 32$ (not 8).

$\boxed{\# \text{Data lines} = \text{Word length} \text{ (property of CPU)}}.$

Ques
How many address and data lines will be there for a $16M \times 32$ memory system?

$$\text{Number of address lines} = \log_2 (16M) = \log_2 (2^4) = \underline{\underline{24}}$$

$$\text{Number of data lines} = \underline{\underline{32}}$$

GATE 2016

A processor can support a maximum memory of 4GB where memory is word addressable (word size = 2 bytes).
What is the ^{min} size of address bus

$$\text{Memory size} = 4\text{GB} = 2^2 \cdot 2^{30} \text{ Bytes} = 2^{32} \text{ bytes.}$$

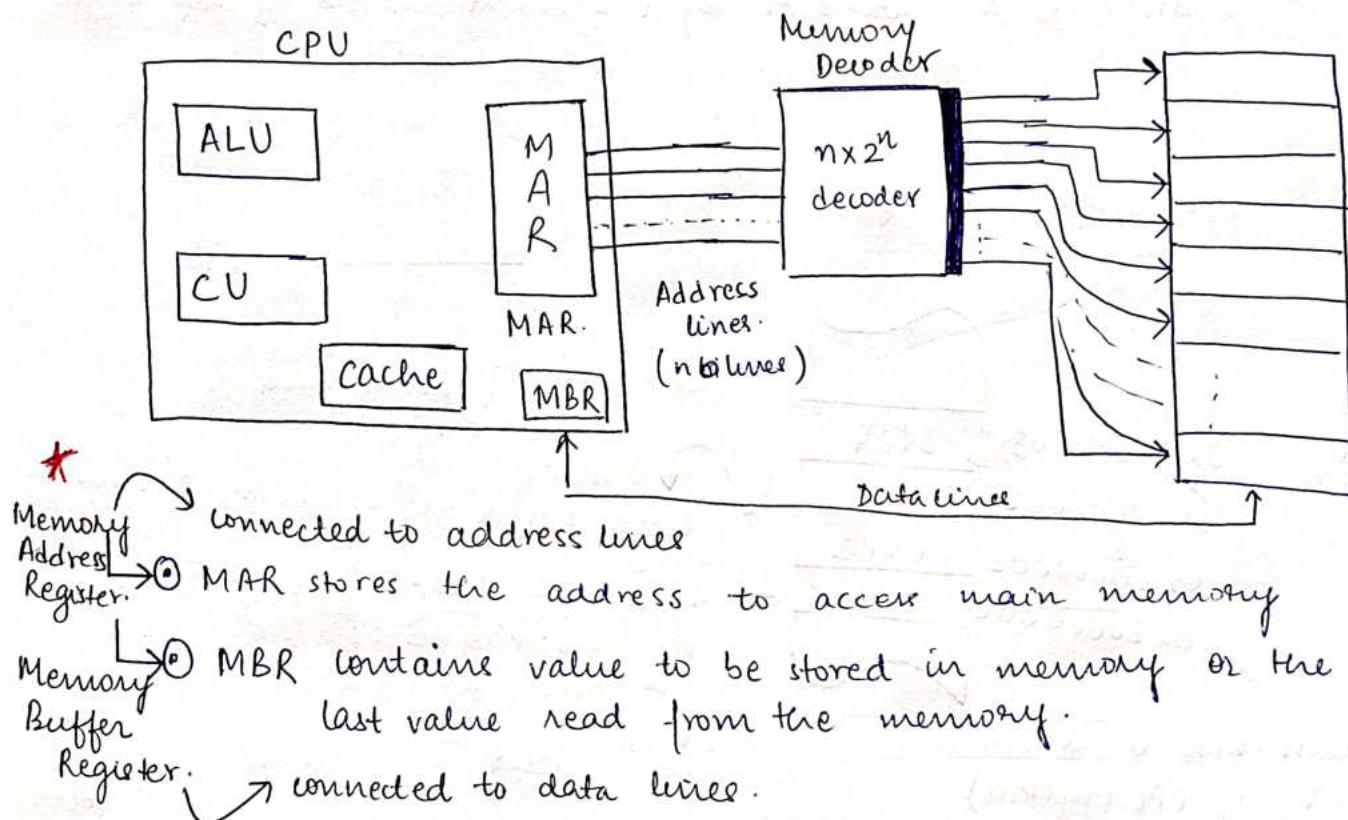
$$\text{No. of addressable locations} = \frac{2^{32}}{2} = 2^{31}$$

\therefore ~~No.~~ No. of bits req. to identify each cell = 31

\therefore No. of bits in address line = 31.

Memory Decoder

Memory Address Register (MAR) is connected to the address lines.

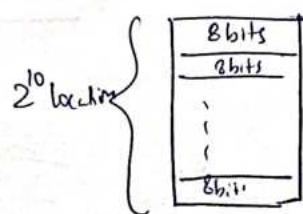


GATE
2020

If there are m input lines and n output lines for a decoder that is used to uniquely address a byte addressable 1KB RAM then the minimum value of

$$m+n \text{ is } \underline{\quad}?$$

byte addressable 1KB RAM.



Size of decoder

$$10 \times 2^{10}$$

$$m=10 \quad n=2^{10} = 1024$$

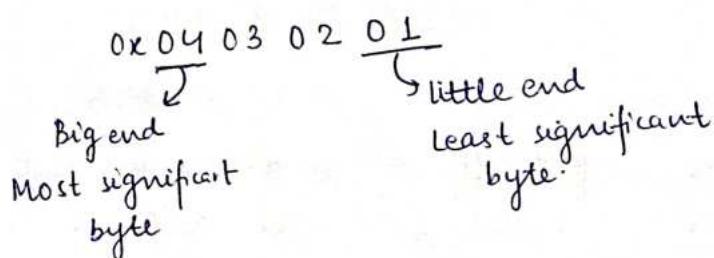
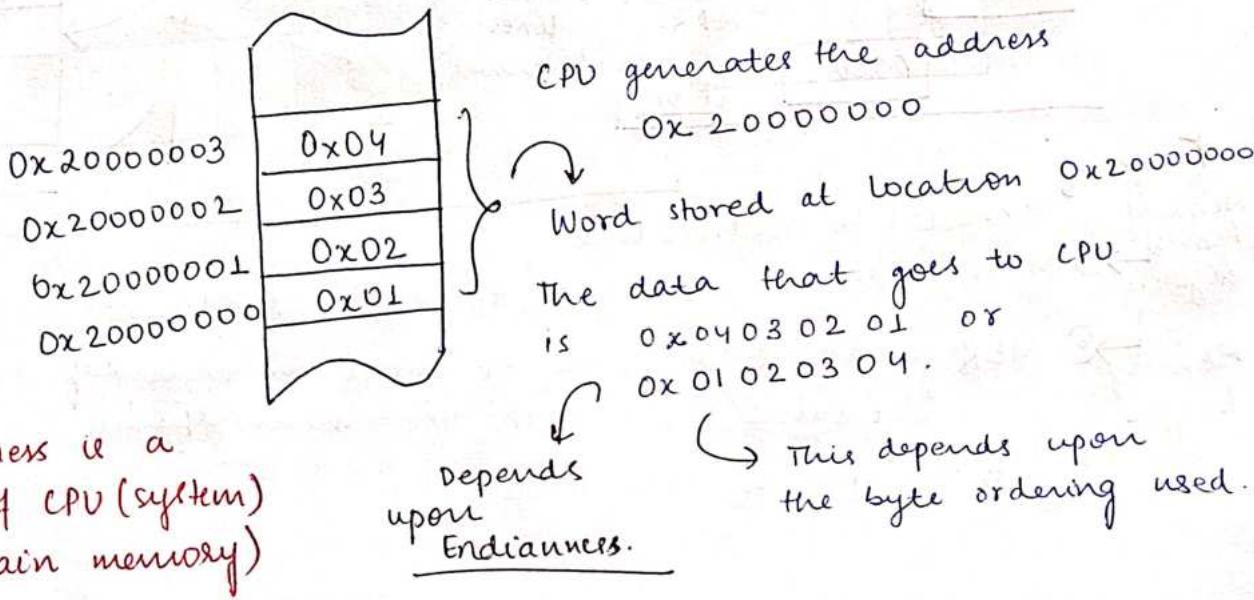
$$\therefore m+n = 10 + 1024 = \underline{1034}$$

Byte Ordering

(This comes into play when memory is byte addressable)
(default)

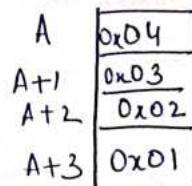
Address of a multi-byte object = Lowest address of all bytes it contains.

Word size = 4 bytes



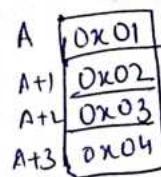
Endian-ness

- Big Endian
Start with big end.
0x04030201
(to be stored in m.m.)



Endian-ness

- Little Endian.
Start with little end
0x04030201



1. Byte Ordering for a Multi-Byte data

in CPU register

Data to be stored = 00010001, D001 1D10, 0010 1011, 00111100.

Big endian

100	0001 0001
101	0001 1010
102	0010 1011
103	0011 1100.

Little Endian.

100	0011 1100
101	0010 1011
102	0001 1010
103.	0001 0001

In Big Endian architecture,
the most significant byte of data goes to lowest address of bytes.

the least significant byte of data goes to highest address of bytes.

In little Endian architecture,
the most significant byte of data goes to highest address of bytes in M.M.

the least significant byte of data goes to lowest address of bytes in M.M.

Interesting

$$\text{int } A = 15213 = (3B6D)_{16}$$

↓
4 bytes.

Big endian

A	00
A+1	00
A+2	3B
A+3	6D

little endian

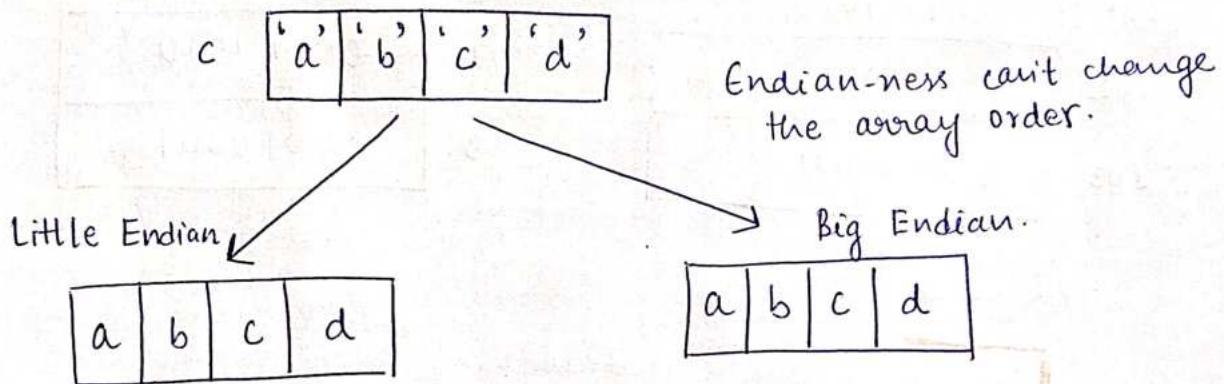
A	6D
A+1	3B
A+2	00
A+3	00

Endian-ness comes into picture only when we have multi-byte of data

for single byte of data, both are same representations.

★ Endian-ness concept only applies to multi-byte primitive data type items, not to string, arrays or structs.

char c[] = { 'a', 'b', 'c', 'd' }



- ① Endian-ness matters when user accesses different amount of data than how he/she stored it.
(e.g. store int, access byte as a char)

00000101 00000000 00000000 00000000 5 in little endian
00000000 00000000 00000000 00000101 5 in big endian.

Acc $\&$ int v=5;

5 is stored as int.

If it is accessed as char (1 byte) then, the result printed will depend upon endianness used.

00000101 in case of little endian

00000000 in case of big endian.

- ② Endian-ness also matters when data is transmitted over a communication channel.

Network layer takes care of this.

Sharing files and data b/w different computers can result in mis-interpretation.

2. Byte Ordering within a word

Big endian → lower byte addresses are used for more significant bytes (leftmost bytes of word).

little endian → lower byte addresses are used for less significant bytes (rightmost bytes of word)

<same as before>



GOOD QUESTIONS

Tell if the big endian and little endian representations are same or different (32 bit integers).

1. The minimum unsigned number.

The min unsigned no. with 32 bits -

0000 0000 0000 0000

Big endian → 0x00000000
little endian → 0x00000000 } same.

2. The maximum unsigned number -

The max unsigned no. with 32 bits -

1111 1111 1111 1111

Big endian → 0xFFFFFFFF
little endian → 0xFFFFFFFF } same.

3. +1 in 2's complement.

+1 = 0000 0001 (2's complement)

Big endian = 0x00000001

little endian = 0x01000000

} different.

4. -1 in 2's complement.

-1 = 1111 1111 (2's complement)

Big endian = 0xFFFFFFFF

little endian = 0xFFFFFFFF

} same

v) Maximum 2's complement number.

will be +ve.

For +ve numbers, 2's complement is same as sign magnitude.

∴ Maximum 2's complement number = 0 111 ... 1111

Big endian → 7F FF FF FF

little endian → FF FF FF 7F

} different.

GOOD
QUESTION

* vi) The minimum (most negative) 2's complement signed number



Minimum 2's complement no = 1 000 0000 ... 0000

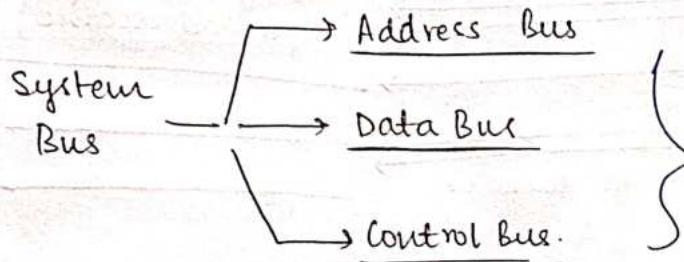
Big endian = 80 00 00 00

little endian = 00 00 00 80.

} different.

System Bus

System Bus \equiv Transmission media b/w different components of a computer



→ can carry 1 bit of data at any point of time.

classification is done based on the type of information they carry.

Address Bus

carries addresses generated by the CPU.
unidirectional from CPU \rightarrow memory / I/O devices.

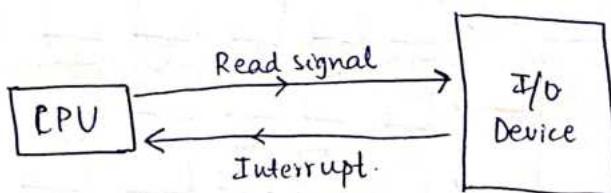
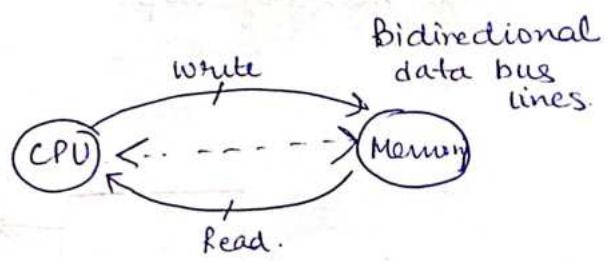
Data Bus

carries data/instructions
bidirectional

Control Bus

carries control signals to/from CPU.

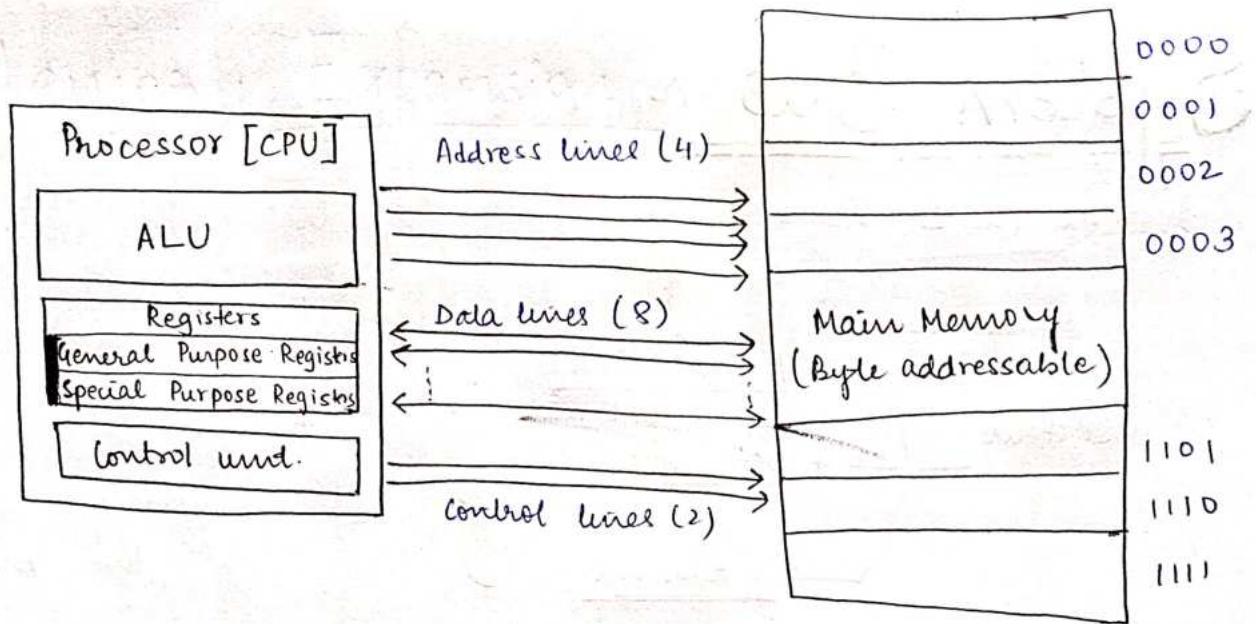
- individual control lines are unidirectional but as a whole, control signals can be sent in both directions.
∴ partially bidirectional.



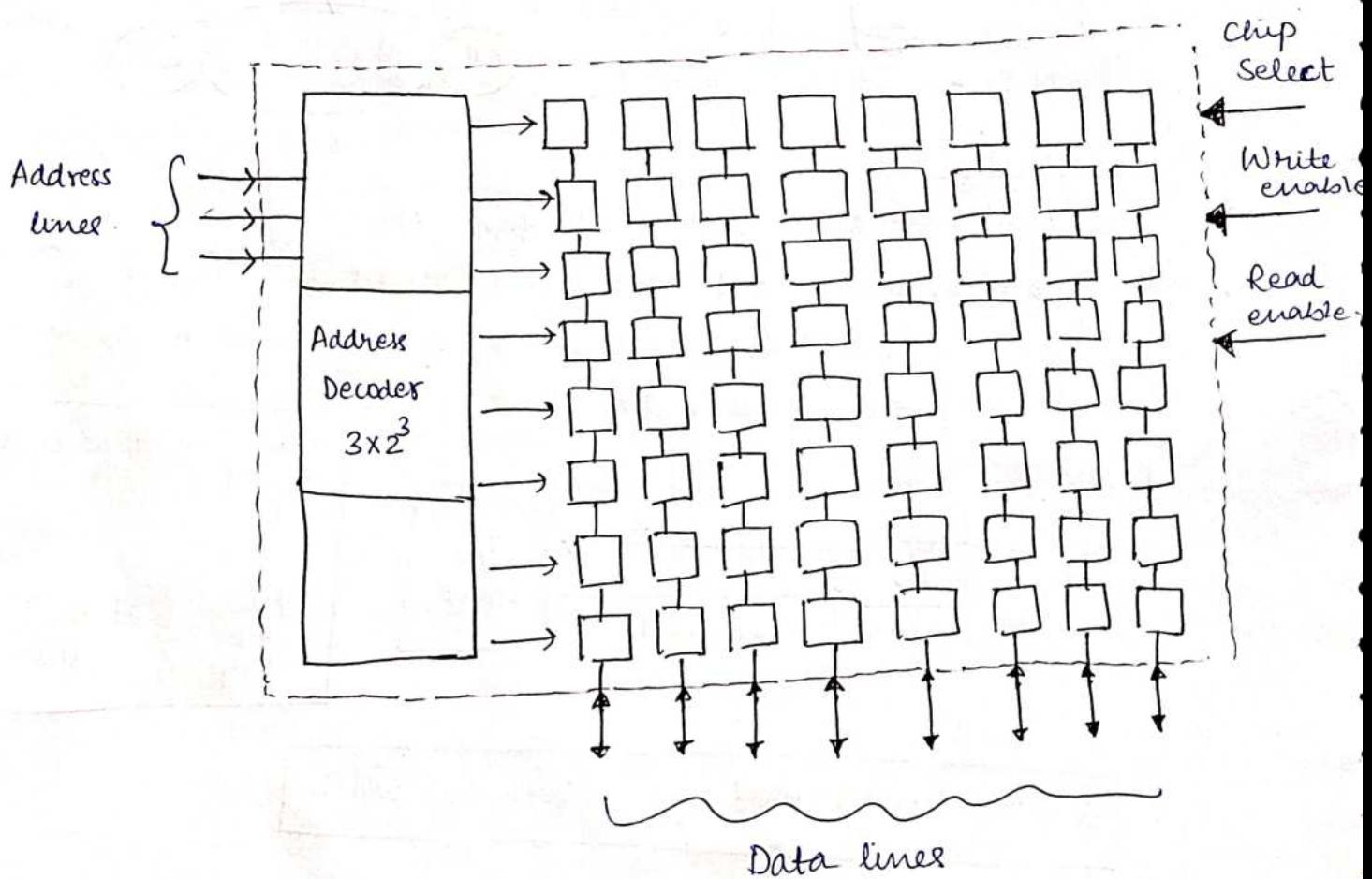
MAR register size	= # address lines
MBR register size	= # data lines

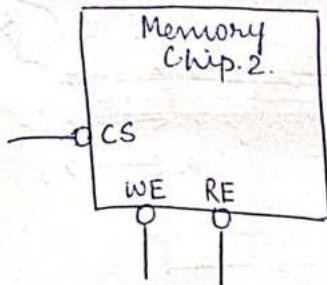
Data lines = word size = Data Bus width.

= MBR register size.



- ① Three additional lines — chip select, read enable, write enable
 ↘ are used to control the transaction.
 control lines connected
 to memory chip.





CS	RE	WE	Action
1	x	x	Memory chip 2 is idle.
0	0	1	Memory Read.
0	1	0	Memory write
0	1	1	No read & write
0	0	0	Responsibility of CPU that this never happens. (write is given preference if it happens)

* Active low inputs.

↳ CS - Chip Select -

No operation can be performed unless the chip is selected.

The data lines are not driven by this chip if it is not selected.

↳ WE - Write Enable -

If the chip is selected, and WE is asserted, data on the data pins are written to the addressed location.

↳ OE - Output Enable / Read Enable RE

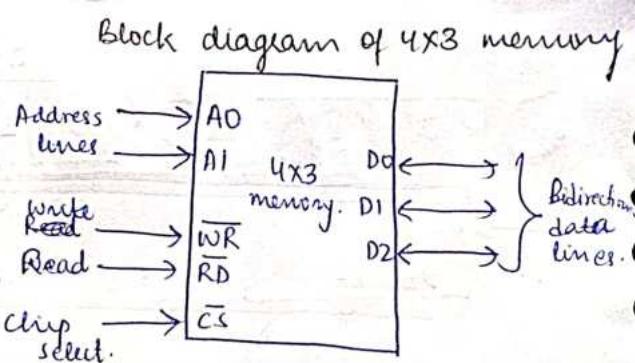
If the chip is selected and WE line is not asserted, and OE is asserted, data at the addressed location is driven on the data lines.

MBR \leftarrow $\langle MAR \rangle$ — Read from memory
 $\langle MAR \rangle \leftarrow MBR$ — Write into memory.

$\langle MAR \rangle \rightarrow$ memory location whose address is given by MAR. The MAR is said to point to the memory location.

Memory Expansion

Designing circuit for a given memory capacity using smaller memory chips.

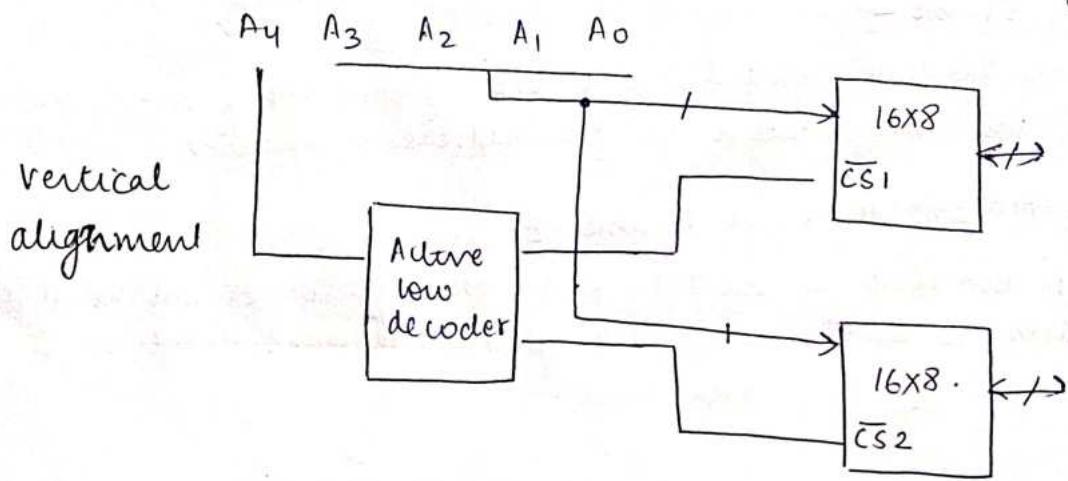


Organize 16x8 RAM chips to give 32x8 memory capacity

5 address lines

A₄ A₃ A₂ A₁ A₀

If we use A₄ to decide which chip to select, then, it is called high order memory interleaving.



- ① In horizontal alignment, we need more data lines.

(when no. of data lines required is greater than the size of cell in chip, then, chips are arranged in horizontal alignment)

- ② No decoder is required for vertical alignment. same address is sent as input into the address lines of each chip.

IMPORTANT

- * If chip select is active low, then, active low decoder (NAND) decoder is used.
- If chip select is active high, then, active high (normal) decoder is used.

① Larger memories can be built by

I M P O R T A N T | 1. Horizontal Expansion

→ increases word size

→ increases the number of bits per address location.

2. Vertical Expansion

→ increases number of words

→ increases the number of address lines / locations

② Chip Select in Vertical Expansion.

1. High order memory interleaving

→ consecutive addresses available within a memory module.

→ higher order address lines (bits) are used for chip select.

2. Low order memory interleaving

→ consecutive memory addresses are available across the memory modules (not within a single module).

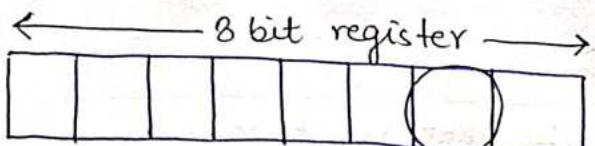
→ lower order address lines (bits) are used for chip select.

The Central Processing Unit - CPU

Registers in CPU

storage devices inside the CPU.

Register is a collection of flipflops.



each cell is a
flipflop.

One bit
storage device

Types of registers in CPU

Based on the information they store.

- Data Registers
- Address Registers

Based on tasks assigned to them.

- General Purpose Registers
- Special Purpose Registers.

Special Purpose Registers -

1. Memory Address register (MAR)
2. I/O Address register (I/O AR)
3. Program Counter (PC)
4. Stack Control Register.

Address Register
(store addresses)

5. Memory Buffer Register (MBR)

6. I/O Buffer Register (I/O BR)

7. Instruction Register (IR)

8. Accumulator Register (AC)

9. Flag Register / Program Status Word

Data Registers
(store data or instructions)

Memory Address Register and Memory Buffer Register

MAR - The Memory Address Register is used to store the address to access memory.

MBR - The Memory Buffer Register stores information that is being sent to, or received from, the memory along the bidirectional data bus.

Program Counter

① Program counter register is used to store the address of the next instruction to be fetched for execution.

② connected to the internal address bus.

③ it doesn't connect directly to the memory, but must go via the MAR.

↳ It is both - a register and a counter.
(we don't go to ALU to increment program counter).

(PC can increment itself by a fixed value).

$$\boxed{\text{Program Counter} \equiv \text{Instruction Address Register} \equiv \text{Instruction Pointer Register}}$$

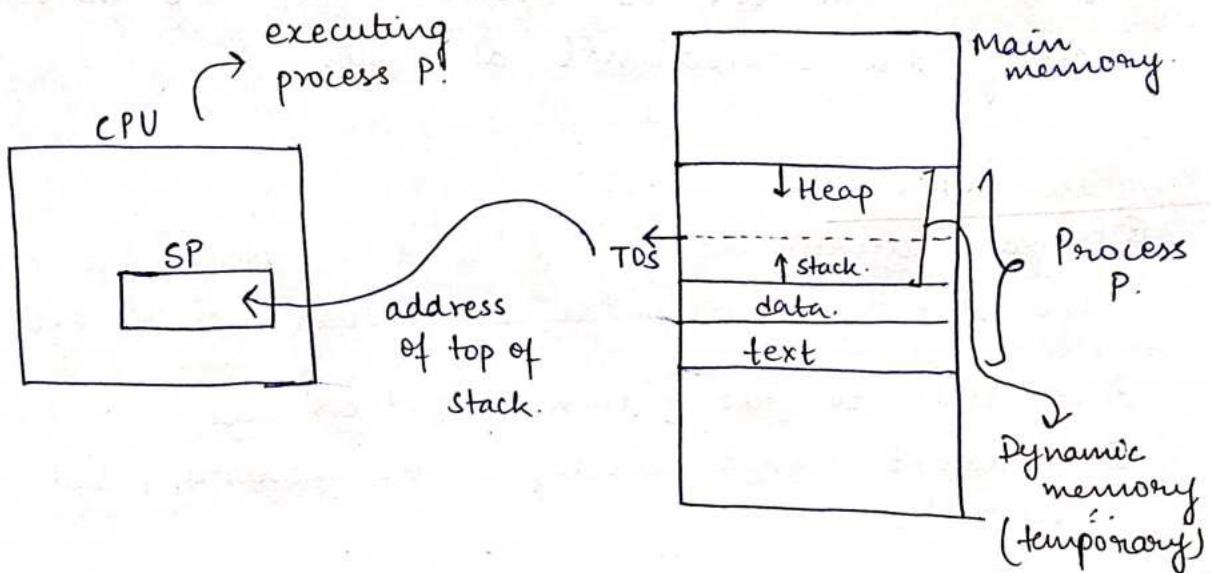
Instruction Register

When memory is read, the data first goes to the MBR. If the data is an instruction, it gets moved to Instruction Register.

IR holds the current instruction being executed or that needs to be executed.

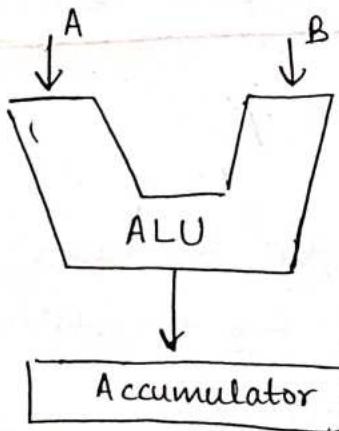
Stack Pointer (SP)

- ① stores address to top of stack of process.
- ② SP is connected to internal address bus and is used to hold the address of a special chunk of main memory used for temporary storage during program execution.

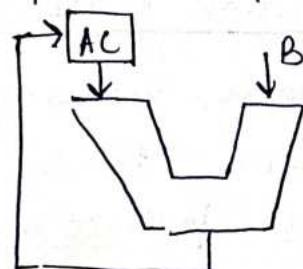


Accumulator (AC)

- ① associated with ALU (inside ALU)
- ② The accumulator is used to store data that is being worked on by the ALU.
- ③ stores and holds the result of operation performed by ALU.



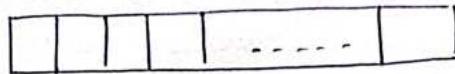
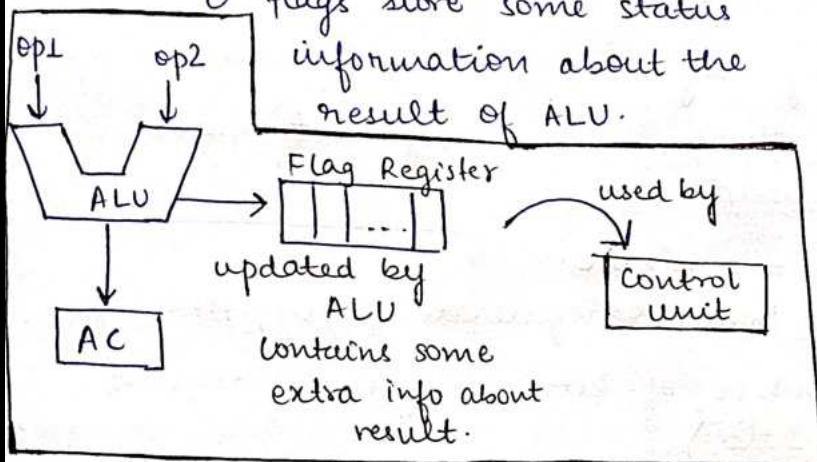
- ④ may also be used to hold one of the operands of ALU.



Flag Registers

Status Register = Flag Register = Program Status Word = Program Status Register = status word.

① associated to ALU



every bit is a flag

An important function of the ALU is to set up flags or flags which give information to the control unit about the result of the operation.

These flags are grouped together to form the status word.

② Six status flags monitor the outcome of arithmetic, logical, and related ops

1. Zero Flag (ZF)

This is set to 1 whenever the output from the ALU is zero.

$ZF = 1$ for zero result $ZF = 0$ for non zero result

$$\underline{\text{Ex}} \rightarrow AL = 0x80 \quad BL = 0x80.$$

$$AL + BL = \begin{array}{r} 1000\ 0000 \\ 1000\ 0000 \\ \hline 1\ 0000\ 0000 \end{array}$$

Accumulator contains 0000 0000.

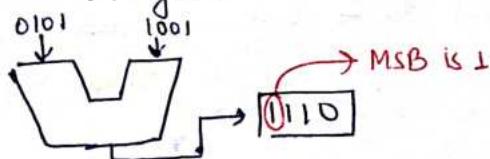
$\therefore ZF$ is set to 1.

2. Negative Flag (Sign flag)

$$SF = \begin{cases} 1 & \text{if MSB of result is 1} \\ 0 & \text{if MSB is 0.} \end{cases}$$

$$\underline{\text{Ex}} \rightarrow \text{unsigned int } A = 0101 \quad \text{unsigned int } B = 1001$$

$$A + B = \begin{array}{r} 0101 \\ 1001 \\ \hline 1110 \end{array}$$



$\therefore SF$ is set to 1.

3. Carry Flag

- OCF is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.

- ④ Set to 1 when there is a carry from the adder.

4. Auxiliary Carrying Flag

- ④ $AF = 1$ if there is a carry out from bit 3 on addition.

4 bits = 1 nibble.

if 4 bits from LSB generate carry, AF = 1.

(Number of bits in operand does not matter).

- ⑥ Indicates whether an operation produced a carry or borrow in the low order 4 bits of 8-, 16-, or 32-bit operands.

5. Parity Flag

- ① PF = 1 if the low byte of a result has an even number of one bits.

- ④ PF = 0 if the low byte has odd parity

Ex → FFFE

- ⑥ only least significant 8 bits are taken into account.

low byte \rightarrow FE = 1111110
7 ones (odd)

$$\therefore PF = 0.$$

6. Overflow Flag

ALU does not know why it is setting the flag. Rule \overline{E}
it follows करना \overline{E}
तो क्या !! 

overflow occurs
in 2 cases

→ neg + neg gives positive

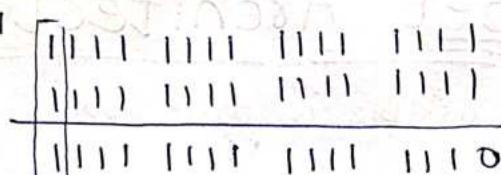
→ tve + tve gives negative.

- ④ set to L when $A_{MSB} = 1$, $B_{MSB} = 1$ but $O_{MSB} = 0$
 or $A_{MSB} = 0$, $B_{MSB} = 0$ but $O_{MSB} = 1$

Example AX = 0xFFFF BX = 0xFFFF

①

AX+BX =



ZF = 0 ✓
SF = 1 ✓
CF = 1 ✓
PF = 0 ✓
AF = 1 ✓
OF = 0 ✓

Example

②

AX = 0x80

BX = 0x80

$$\begin{array}{r} \text{AX+BX} = \textcircled{1} \quad 1000 \quad 0000 \\ \quad 1000 \quad 0000 \\ \hline \quad 0000 \quad 0000 \end{array}$$

ZF = 1 ✓
SF = 0 ✓
CF = 1 ✓
AF = 0 ✓
PF = 1 ✓
OF = 1 ✓

IMPORTANT

MOV, PUSH, POP do not affect any flag.

Increment, Decrement affect all flags except carry flag(CF)

ADD, SUB affect all flags.

Instruction Set Architecture

- ① ISA defines the permissible instructions (allowed instructions)
 - ② ISA is a view of computer system from Assembly language programmer's point of view or compiler's point of view.
 - ③ ISA is a boundary / ~~b/w~~ interface b/w software and hardware.

ISA of
a processor
can be
described
using 5
categories

Instruction set design issues -

1. Number of explicit operands.
 2. Location of operands. (Registers | accumulator | memory)
 3. Specification of operand locations (addressing modes)
 4. Size of operands supported (byte, word, ...).
 5. Supported operations (ADD, MUL, SUB, AND, OR, ...).

1. Stack Based ISA

Arithmetic instructions — specify 0 explicit operands.
(all operands are implicit).

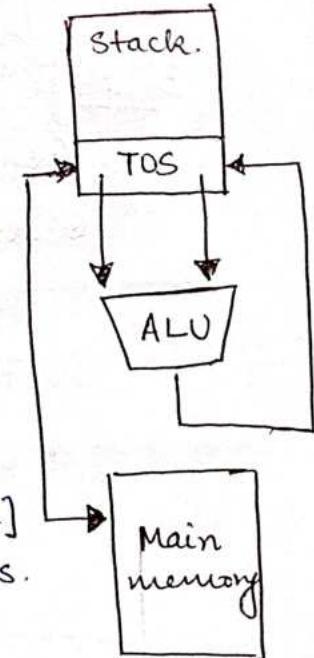
0 - address instruction .

- ④ To execute ADD instruction, 3 time memory needs to be accessed - 2 for fetching operands from stack and 1 time for writing results.

A lot of do 3 memory cycles needed - time consuming.

Stack Based ISA

→ Arithmetic instructions
0 address instructions
operands are fetched from stack.
ADD, SUB. (needs 3 memory accesses)



$$C \leftarrow A + B$$

PUSH A	1 mem cycle
PUSH B	1
ADD	3
POP C	1

→ Data Transfer instructions

1 address instruction.
add/ delete elements from stack.

$$\text{PUSH loc1}$$

$$\text{POP loc2}$$

$$\text{TOS} \leftarrow \text{Mem[loc1]}$$

$$\text{mem[loc2]} \leftarrow \text{TOS.}$$

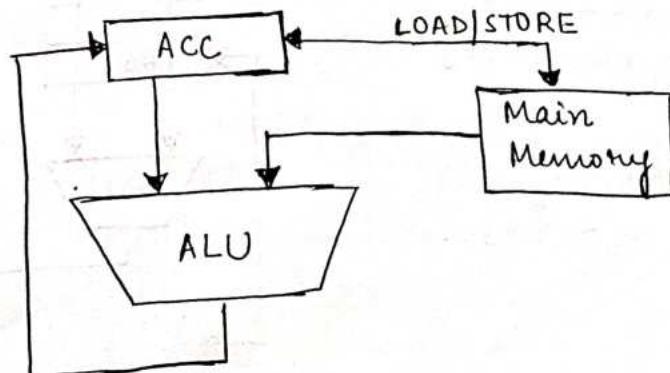
2. Accumulator ISA

- ① In arithmetic instructions, one operand is implicit (from accumulator).
- ② arithmetic instructions are 1 address instructions.
 $\text{ADD loc1 Acc} \leftarrow \text{Acc} + \text{mem[loc1]}$.
- ③ Data transfer instructions — LOAD and STORE are also 1 address instructions.

C \leftarrow A + B.	1
LOAD A	1
ADD B	1
STORE C.	1

LOAD loc1 $\text{Acc} \leftarrow \text{mem[loc1]}$
 STORE loc2 $\text{mem[loc2]} \leftarrow \text{Acc}$
 ↴ memory address.

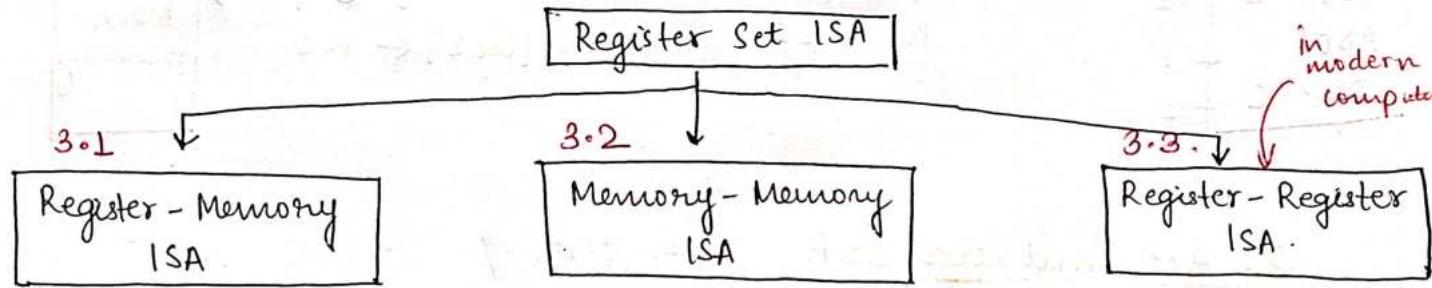
- ④ All instructions assume that one of the operands (and also the result) is in a special register called accumulator.



3. Register Set ISA

General Purpose Register ((GPR) ISA).

- ① It's the dominant architecture used in modern computers.
(coz modern CPUs have a number of GPRs for internal storage (8-64)).
- ② Operands are specified explicitly.



One operand will come from register, the other will come from memory.

Arithmetic instructions are 2-address instructions.

ADD R, loc1
[R] \leftarrow [R] + mem[loc1].

Data transfer instructions are 2-address instructions

LOAD R, loc1
[R] \leftarrow mem[loc1].

STORE loc1, R
mem[loc1] \leftarrow [R].

All operands are from memory.

Most CISCs use this architecture.

ADD B, C, A

$$\text{mem}[A] = \text{mem}[B] + \text{mem}[C].$$

3 address instructions or 2 address instructions both can be used.

All operands in arithmetic operation come from registers.

$$\text{ADD } R_1, R_2, R_3.$$

$$R_1 \leftarrow R_2 + R_3$$

Data transfer instructions are 2-address instructions

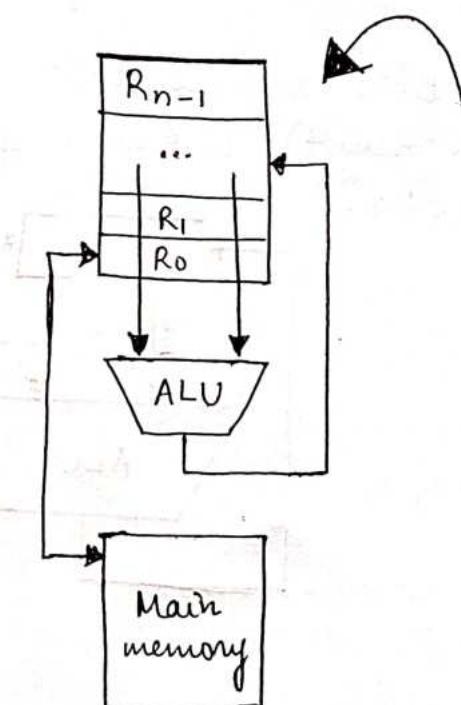
LOAD R1, loc1;
[R1] \leftarrow mem[loc1]

STORE loc1, R,
mem[loc1] \leftarrow [R].

* Arithmetic ins do not access memory!!

* only load and store can access the main memory.

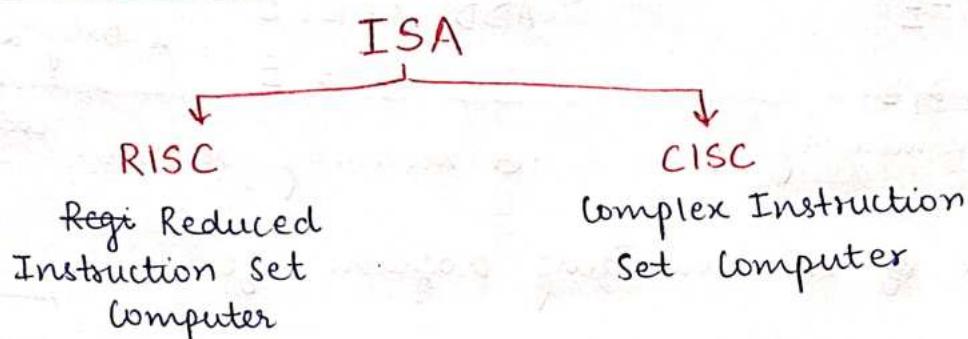
MOST RISCs adopt this architecture.



NOTE

① Registers are faster than the memory, the more data that can be kept internally in the CPU, the faster the program will run.

RISC v/s CISC



In old days, (in the time of Adam),
Memory was very expensive... time wasn't. So CISC architecture was used.

Memory expensive → Program must be short → No. of instructions per program must be less → CISC ISA architecture used.
(Memory - Memory ISA)

In modern days, (in the time of atom),
Memory is not so expensive... time is. (So, RISC architecture is used.)

Memory cheap → Fast execution needed → Optimization needed → Program must take less time to execute
Register-Register ISA is used. ← Time per instruction must be less ←

The CISC Architecture

The primary goal of CISC architecture is to complete a task in as few lines of assembly code as possible.

$$A = B + C$$

$$D = B - E$$

A, B, C, D, E

are in memory.

Mem - Mem. Instructions

$$ADD\ A, B, C$$

$$SUB\ D, B, E$$

6 memory accesses

} only 2 ins in assembly level code.

- ① focuses on minimizing program size.

The RISC Architecture

RISC processors only use simple instructions that can be executed within one clock cycle.

$$A = B + C$$

$$D = B - E$$

5 memory accesses.

$$LOAD\ R_1, B$$

$$LOAD\ R_2, C$$

$$ADD\ R_3, R_1, R_2$$

$$STORE\ A, R_3$$

$$LOAD\ R_4, E$$

$$SUB\ R_5, R_1, R_4$$

$$STORE\ D, R_5$$

} 7 instructions in assembly level code.

- ① focuses on minimizing execution time.

- ② RISC uses Register-Register ISA

coz by using reg-reg architecture, optimization is easy and possible.

(same variable can be reused in multiple instructions by storing its value in a register)

(load data from memory into registers, then, use registers again and again).

Execution time is minimized
coz less no. of memory accesses required.

IMPORTANT

Why CISC architecture is called 'Complex' Instruction Set Architecture and why RISC architecture is called 'Reduced' Instruction Set Architecture?

Register - Register instruction

Simple instructions (can be executed in few clock cycles)	ADD R ₁ , R ₂ , R ₃	R ₁ ← R ₂ + R ₃	fast devices available inside CPU.
	No memory access is needed. can execute in 1 clock cycle.		

Memory - Memory instruction

Complex instructions (can be executed in many clock cycles)	ADD A, B, C	mem[A] ← mem[B] + mem[C].
	Internally, hardware will fetch the values of B and C from memory then add them up and store the result back into the memory.	3 clock Many clock cycles are required.

Simple Instructions v/s Complex Instructions

- | | |
|--|---|
| <ul style="list-style-type: none"> ① Register - Register instructions. ② can be executed in a single clock cycle. ③ used in RISC ④ Hardware is not so complicated. | <ul style="list-style-type: none"> ① Memory - Memory instructions. ② needs to access memory for operands and hence takes many clock cycles. ③ used in CISC. ④ complicated hardware (coz it is u/w heavy). |
|--|---|
-
- ⑤ For CISC architectures, compilation is easy (since assembly language code is small, compiler has to do less work).
 - For RISC architectures, compilation is hard / heavy (large no. of instructions in assembly language are gen to be generated by the compiler).

- ① CISC CPU — Hardware does more work.
 RISC CPU — software does more work.
 (compiler).

The Performance Execut Equation.

$$\frac{\text{Time}}{\text{program}} = \frac{\# \text{instructions}}{\text{in program}} * \frac{\# \text{cycles per}}{\text{instruction}} * \frac{\text{Time taken}}{\text{for each cycle}}$$

CISC minimizes # instructions in program.

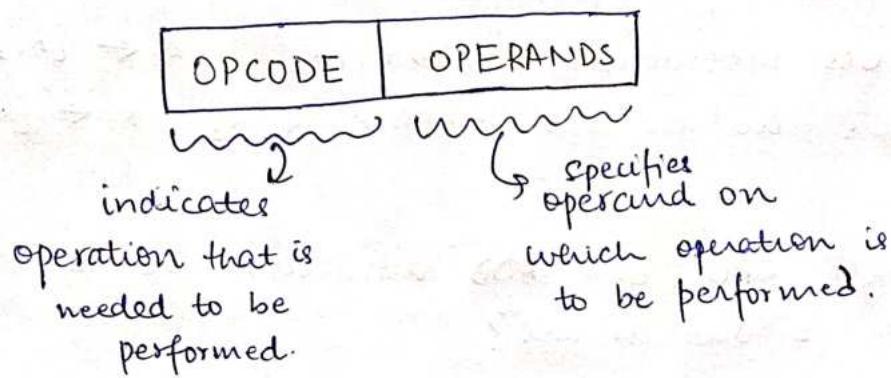
RISC minimizes #cycles per instruction.



<u>CISC</u>	v/s	<u>RISC</u>
① Memory to memory instruction set.		① Register - register instruction set.
② Requires less number of registers.		② Requires more number of registers.
③ Load and Store are incorporated in instructions.		③ LOAD and STORE are independent instructions
④ Emphasis is on hardware		④ Emphasis is on software.
⑤ Slower.		⑤ Faster
⑥ Less instructions per program.		⑥ More instructions per program.
⑦ Includes multi clock complex instructions.		⑦ Includes single clock reduced instructions only
⑧ Pipelining is difficult		⑧ Pipelining is easy.
⑨ More addressing modes.		⑨ Fewer addressing modes.

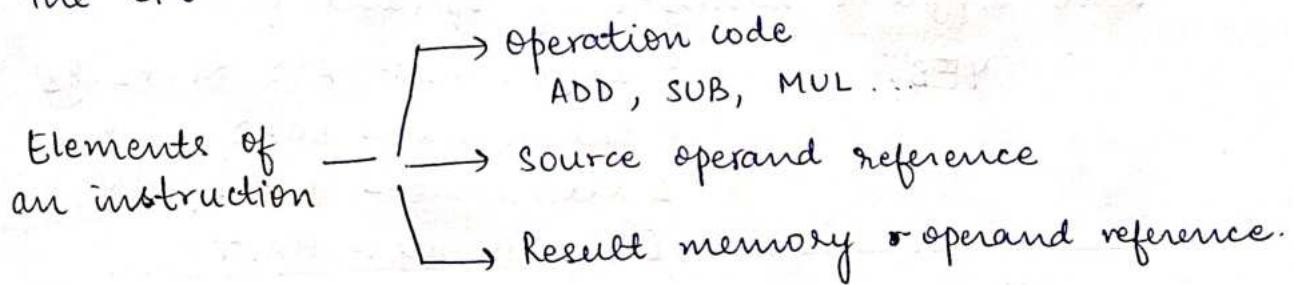
Instruction Format...

Each instruction has 2 parts - opcode and operands.



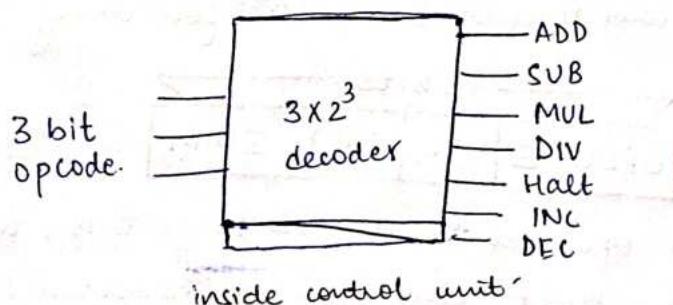
Instruction set -

The complete collection of instructions that are understood by the CPU.



Opcode - encoding of the operations that are implemented in the hardware.

Control unit has a decoder that decodes the opcode to find out which operation is to be performed



① To each of the 2^K instructions a unique binary bit pattern of length K is assigned.

A K -to- 2^K decoder can then be used to decode all the instructions.

An instruction can have zero or more operands. (max 3 operands).

Stack Based CPU :- 0 address arithmetic instructions

Accumulator based CPU :- 1 address arithmetic instructions

2 address instructions : ADD R₁, R₂ R₁ \leftarrow [R₁] + [R₂].

3 address instructions : ADD R₁, R₂, R₃ R₁ \leftarrow [R₂] + [R₃].

In a single CPU, can ADD instruction be 0 address ins as well as 1 address ins?

NO.

In a single CPU, can different instructions / operations have different number of operands?

YES.

3 address ins :- ADD R₁, R₂, R₃

2 address ins :- LOAD R₁, loc R₁ \leftarrow [loc]

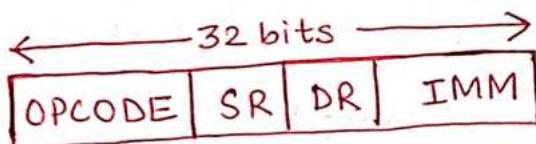
1 address ins :- INC R₁

0 address ins :- HALT.

The total number of instructions and the types and formats of the operands determine the length of the instruction.

Ques

Suppose a 32 bit instruction takes the following format.



If there are 60 opcodes and 32 registers, what is the range of values that can be represented by immediate (IMM)? Assume IMM is a 2's complement number.

Opcode = 6 bits. Registers \rightarrow 5 bits

No. of bits for IMM = $32 - (5 \times 2 + 6) = 32 - 16 = 16$ bits.

$$\text{Min no.} = (1000\ldots0)_2 = -2^{n-1} = -2^{15} \text{ } \checkmark$$

$$\text{Max no.} = (0111\ldots1)_2 = 2^{n-1} - 1 = 2^{15} - 1. \text{ } \checkmark$$

Ques A digital computer has a memory unit with 24 bits per word. The instruction set consists of 150 different operations. All instructions have an operation code part (opcode), and an address part (allowing for only one address). Each instruction is stored in 1 word of memory.

$$\text{No. of bits needed for opcode} = \lceil \log_2 150 \rceil = \underline{\underline{8 \text{ bits}}}$$

$$\text{No. of bits left for address part of ins} = 24 - 8 = \underline{\underline{16 \text{ bits}}}$$

$$\text{Maximum allowable memory size} = \underline{\underline{2^{16} \times 1B}}$$

By default,
memory is
byte addressable

$$\text{Largest unsigned binary number that can be accommodated in one word of memory} = \underline{\underline{2^{16} - 1}}$$

Expanding opcode (Instruction length)

Instructions → Fixed length instructions

Variable length instructions

- ① Decoding is complicated

Fixed length instructions

- ② Decoding of instructions is easier.

- ③ wastage of memory (space in main memory due to alignment).

④ more wastage of space.
(space is wasted in register in the form of unused bits).

- ⑤ Pipelining is difficult

⑥ Pipelining is easy.

- ④ Fixed length instructions can support / maximize various types of instruction formats without wasting much spec space → By using Expand opcode technique.

Expanding opcode technique -

- ① used in fixed length instructions
- ② maximizes various types of instructions.
ie. more instructions can be allowed in instruction set.
- ③ Decoding logic does not become too complex.
(still better, simpler and faster than variable length instructions).

→ Always start from the instruction having smallest length of expand opcode.

3 address instruction → 2 address instruction → 1 address instruction → 0 address instruction

Ques

Is it possible to design an expanding opcode to allow the following to be encoded in a 12 bit instruction? Assume a register operand requires 3 bits and this instruction set does not allow memory addresses to be directly used in an instruction.

- ① 4 instructions with 3 registers
- ② 255 instructions with 1 register
- ③ 16 instructions with 0 registers.

3 address instructions — opcode length = $12 - 9 = 3$ bits.

$$\text{Encodings used} = 4 \times 2^9$$

1 address instructions — opcode length = $12 - 3 = 9$ bits

$$\text{Encodings used} = 255 \times 2^3$$

0 address instructions — opcode length = 12

$$\text{Encodings used} = 16 \times 2^0$$

$$\begin{aligned} \text{Total encodings} &\leq 2^{12} \\ \Rightarrow 4 \times 2^9 + 255 \times 2^3 + 16 \times 1 &\leq 2^{12} \\ \Rightarrow 4 \times 512 + 255 \times 8 + 16 &\leq 2^{12} \quad \Rightarrow 2048 + 2040 + 16 \leq 4096 \\ \Rightarrow 4104 &\leq 4096 \quad \text{false.} \end{aligned}$$

∴ not possible

Ques

Given 8 bit instructions, is it possible to use expanding opcodes to allow the following to be encoded?

- 3 instructions with 2 3 bit operands. (T-1)
- 2 instructions with 1 4 bit operand. (T-2)
- 4 instructions with 1 3 bit operand. (T-3)

Type 1 ins — opcode length = $8 - 6 = 2$

$$\text{Encodings used} = 3 \times 2^6 = 3 \times 64 = 192$$

Type 2 ins — opcode length = $8 - 4 = 4$

$$\text{Encodings used} = 2 \times 2^4 = 2 \times 16 = 32$$

Type 3 ins — opcode length = $8 - 3 = 5$

$$\text{Encodings used} = 4 \times 2^3 = 4 \times 8 = 32$$

$$192 + 32 + 32 \leq 2^8 \quad \Rightarrow 192 + 64 \leq 256$$

$$\Rightarrow 256 \leq 256 \quad \text{true.}$$

∴ possible.

Type 1 ins — 2 length opcode — 4 combinations possible
3 are used (say 00, 01, 10).

Type 2 ins — 4 length opcode — first 2 should be 11.

∴ 4 combinations possible — 2 are used (say 1100, 1101)

Type 3 ins — 5 length opcode —

11100 11101 11110 11111.

Instruction Types

① direct memory to memory more operations are not possible-

Computer instructions

Data Transfer instructions

Transfer data from one location to another.

most fundamental type of machine instruction.

LOAD, STORE, PUSH, POP, etc.

Data Manipulation instructions

These instructions perform operations on data and provide the computational capabilities of the computer.

3 types of data manipulation instructions -

- ↳ Arithmetic
- ↳ Logical & bit manipulation
- ↳ Shift ins.

Increment, Decrement instructions do not affect the carry flag.

ADD R, L affect CF.

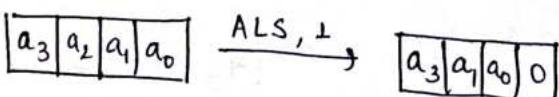
Program control instructions

BRANCH & JUMP instructions.

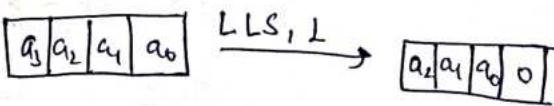
used to transfer the control from one part of the program to another part.

(in detail next page)

Arithmetic shift → sign bit (MSB) remains unchanged.
LSB/bit next to MSB is lost



logical shift → each bit is shifted
MSB/LSB is lost



IMPORTANT Right arithmetic shift = Division by 2 of signed numbers in 2's complement form.

Right logical shift = Division by 2 with unsigned numbers.

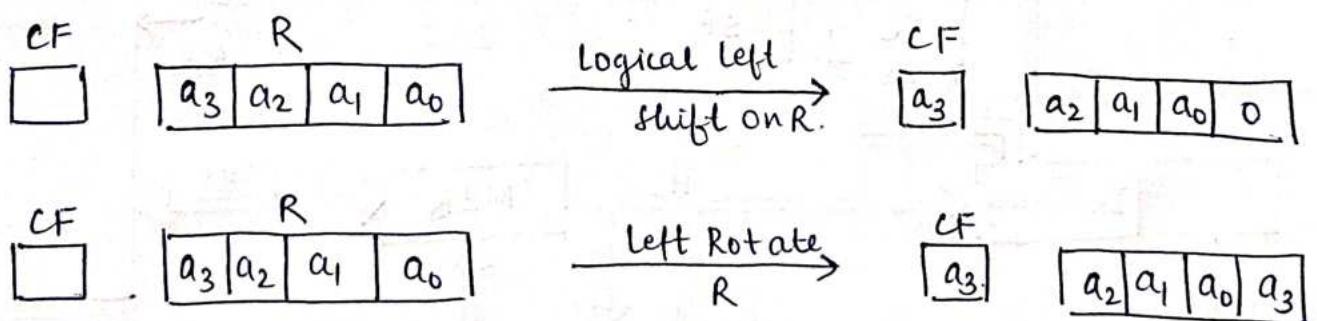
* With signed ~~numbers~~ numbers in 2's complement notation, both logical left shift - $\times 2$ unsigned nos. ALS and LLS correspond to $\times 2$ (when no overflow).

Important summary
 (* Try out with examples)

	Multiplication by 2^k ($\times 2^k$)	Division by 2^k ($\div 2^k$)
Signed no. in 2's complement.	Both arithmetic left shift and logical left shift.	Arithmetic Right shift.
Unsigned number.	Logical left shift.	logical right shift.

How Carry Flag gets updated / affected by shift and Rotate instructions?

Carry flag contains last bit shifted out.



Branch instruction

→ Unconditional Branch instruction

JMP M just jump to M man!
so don't worry!
no overthinking.

→ Conditional Branch instruction

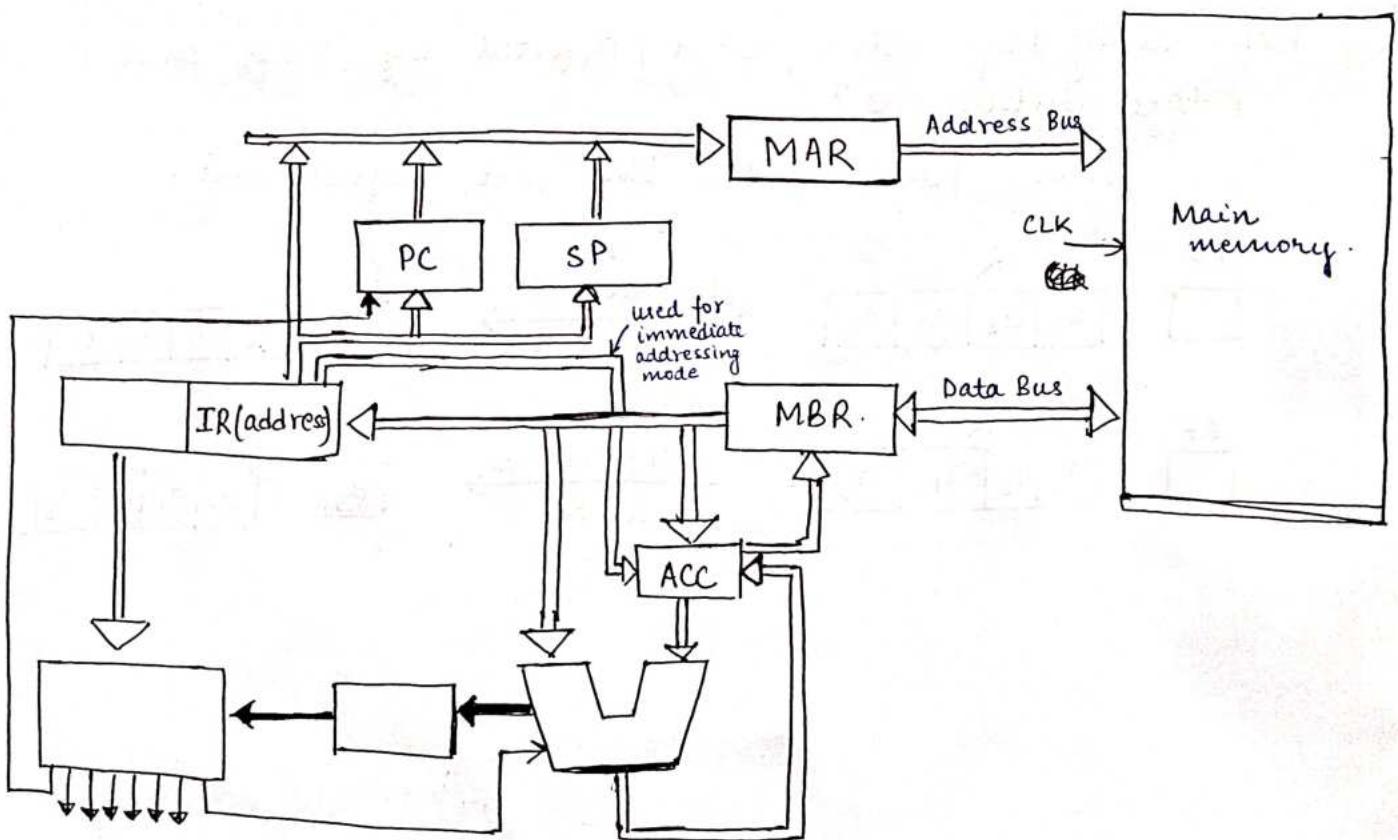
JZ M Jump to M if ZF is set.

JC M Jump to M if CF is set.

JNZ M jump to M if ZF is not set.

Instruction Execution Cycle

- ↳ FETCH the next instruction from memory into the instruction register.
- ↳ DECODE the instruction (operations & operands).
- ↳ EXECUTE the instructions.



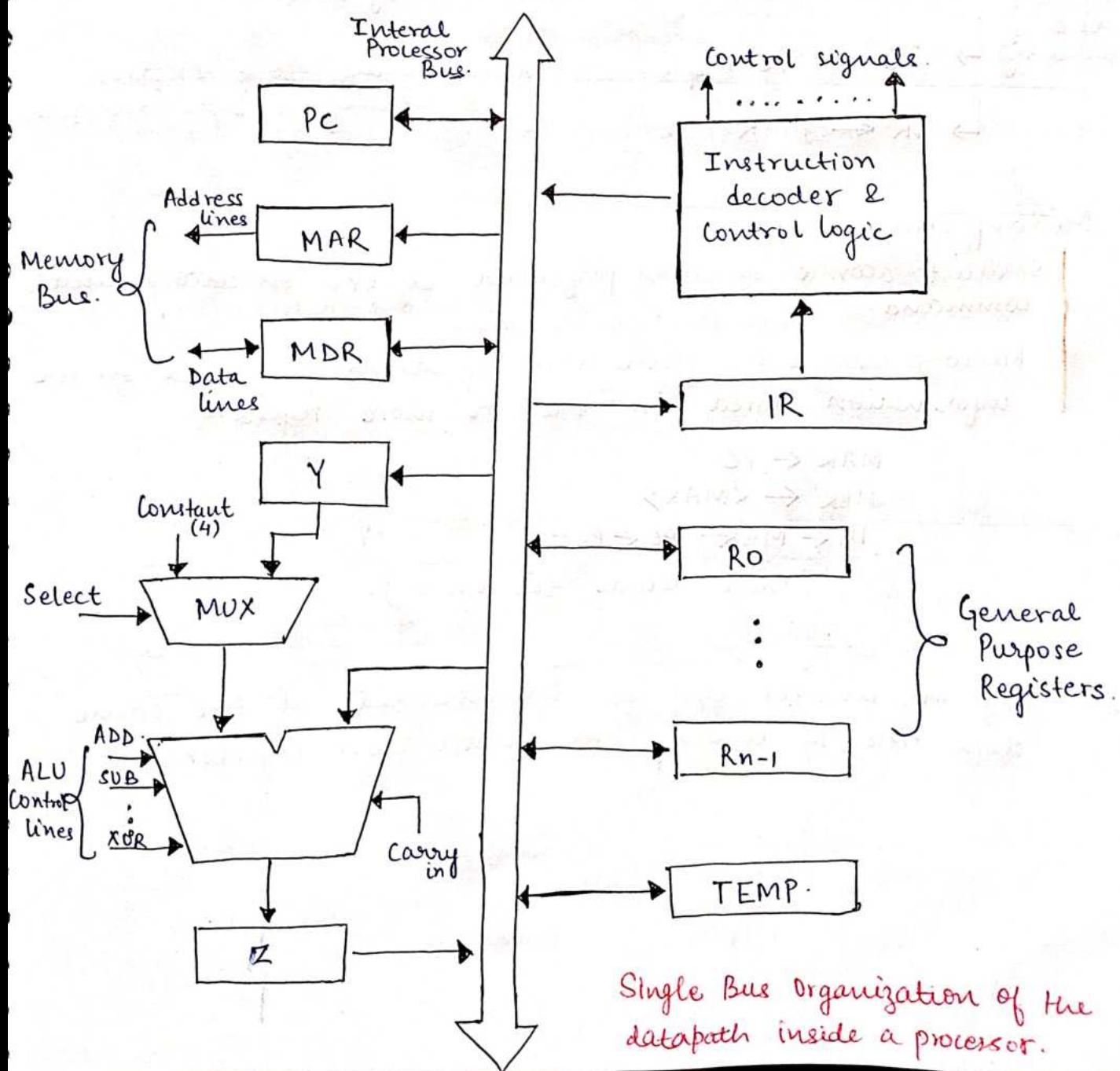
- ① In order to communicate with memory, a processor needs three types of connections — data, address and control.

WORD

A word refers to how much data width the CPU can process at a time.

Based on processor's data bus width.

The registers, the ALU, and the interconnecting bus are collectively referred to as datapath.



INSTRUCTION FETCH CYCLE

Instruction Fetch

Initially, the program counter PC is loaded with the address of the first instruction in the program.

Instruction fetch steps -

↳ $\text{MAR} \leftarrow [\text{PC}]$ Register Transfer
 $\text{PC}_{\text{out}}, \text{MAR}_{\text{in}}$

These 2 steps can be done simultaneously.

↳ $\text{MBR} \leftarrow \langle \text{MAR} \rangle$ Content of memory at address held by MAR.
↳ $\text{PC} \leftarrow \text{PC} + 1$ (does not need ALU) Assume inc size = 4 bytes
2 memory is byte addressable
↳ $\text{IR} \leftarrow \text{MBR}$.

Micro operation -

smallest atomic operation performed by CPU on control unit's command.

Microoperations are elementary operations performed on the information stored in one or more registers.

$\text{MAR} \leftarrow \text{PC}$
 $\text{MBR} \leftarrow \langle \text{MAR} \rangle$
 $\text{IR} \leftarrow \text{MBR}; \text{PC} \leftarrow \text{PC} + 1$
(Then decode the opcode).

Program counter can be incremented at the same clock tick as loading the instruction register.

Instruction Decode

Word size = 2 Bytes ; Instruction I = 4 bytes = 2 words.

In instruction fetch, we get one word of memory i.e.
After decoding, remaining part of instruction is fetched
from memory.

Instruction decode is also part of instruction fetch cycle.

Instruction Fetch cycle

- ↳ Instruction fetch
- ↳ Instruction decode

opcode is decoded using decoder.

INSTRUCTION EXECUTE CYCLE

① LOAD instruction

LDA x → copy the contents of memory at address x
into the accumulator.

LDA x ACC \leftarrow <x>

- ① MAR \leftarrow IR(address)
- ② MBR \leftarrow <MAR>
- ③ ACC \leftarrow MBR. \rightarrow l_{fetch}.
next instruction.

② STORE instruction

STA x → Store the contents of the ACC in the
memory location which is held in the IR
(operand).

- ① MAR \leftarrow IR(address)
- ② MBR \leftarrow ACC
- ③ <MAR> \leftarrow MBR ; \rightarrow l_{fetch}.

③ ADD instruction

ADD $x \rightarrow$ Add the content of memory location x to the accumulator and store the result in accumulator.

$$ACC \leftarrow ACC + <x>$$

① MAR \leftarrow IR (address)

② MBR \leftarrow <MAR>

③ ACC \leftarrow ACC + MBR ; \rightarrow lfetch.

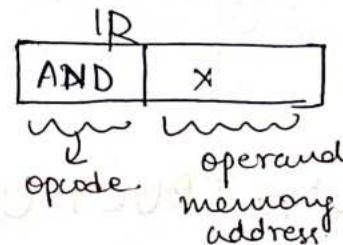
④ AND instruction

$$AND x \rightarrow ACC \leftarrow ACC \text{ AND } <x>$$

① MAR \leftarrow IR (address)

② MBR \leftarrow <MAR>

③ ACC \leftarrow ACC \wedge MBR.



⑤ JMP x .

unconditional jump to x.

① PC \leftarrow IR(address); \rightarrow lfetch.

⑥ BZ x

conditional jump

BZ x
Branch if
zero flag is
set
memory
address.

① $\bar{Z} \rightarrow$ lfetch

② PC \leftarrow IR(address)

⑦ NOT instruction

$ACC \leftarrow \overline{ACC}$ complement of accumulator

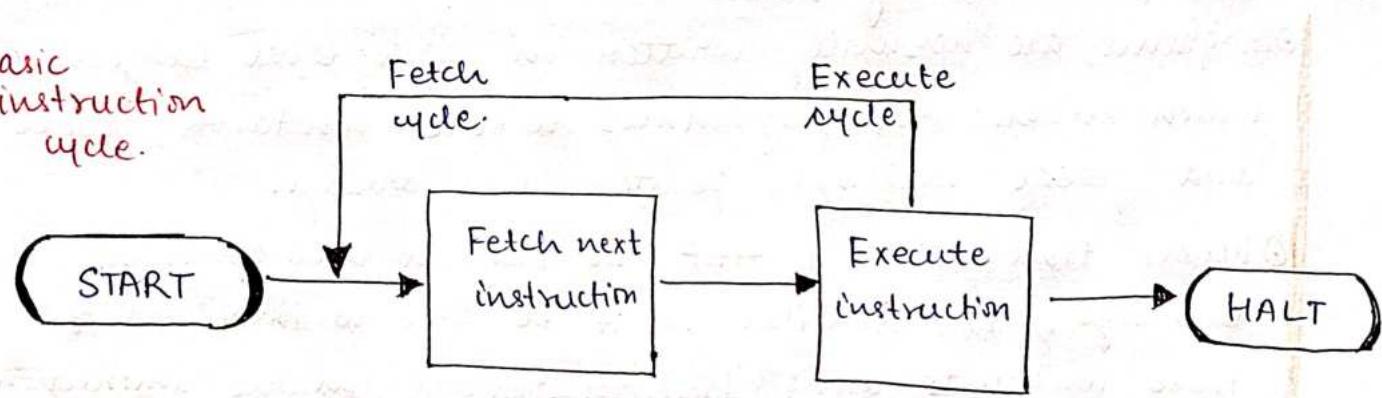
① $ACC \leftarrow \overline{ACC}$

⑧ SHR instruction

Shift Right instruction.

① $ACC \leftarrow \text{ShiftRight}(ACC) \quad // ACC_{out}, ALU_{SR}, ACC_{in}$

Basic instruction cycle.



HALT instruction -

- Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or HALT program instruction is encountered.
- It breaks the FETCH - EXECUTE cycle.
- Meaning - stop the CLOCK.
- HALT tells the CPU to stop executing altogether.

Addressing modes

Addressing modes are the ways by which the location of operands is specified in the instruction.

Different addressing modes are used for different instructions.

Need for addressing modes -

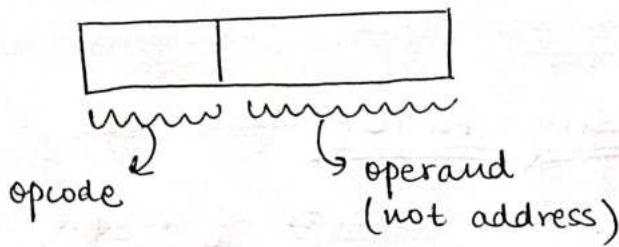
- ① Programs are normally written in high level languages, which enables the programmer to use constants, local and global variables, pointers and arrays.
- ② When translating a HLP HLL program into assembly language, the compiler must be able to implement these constructs using facilities provided in the instruction set of the computer in which the program must run.
- ③ The different ways in which location of an operand is specified in an instruction are referred to as addressing modes.

Effective address - EA of an operand is the actual address where we can find the operand value.

$[EA]$ - content at address EA \rightarrow operand.

1. Immediate addressing mode

- ① operand value is present in the instruction itself.
- ② used to define constants or set initial values of variables.



int A=5

LOAD R1, #5;
S0A

→ The value is stored in 2's complement.

Advantage - no memory reference.

Disadvantage - size of no. is restricted by the size of address field.

2. Implied / Implicit | Stack Addressing mode

- ① opcode tells about the location of operand.

< Stack based ISA CPU >

ins: ADD TOP $\leftarrow [TOP] + [TOP]$

< Accumulator CPU >

ADD LOCA ACC $\leftarrow [ACC] + \text{memory[LOCA]}$.

first operand is implied.

3. Direct / Absolute Addressing mode

* Simple.

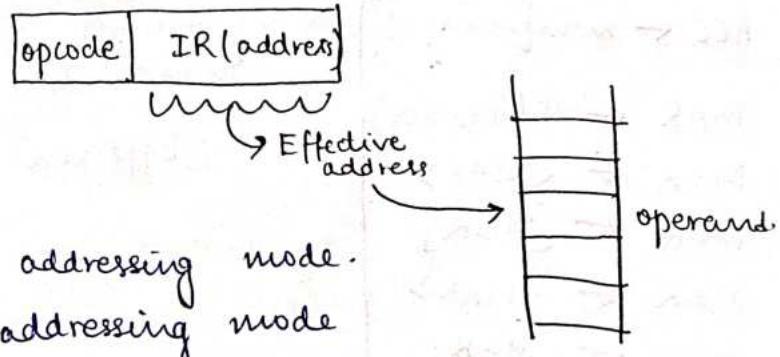
Requires only ① Address field of instruction contains the address of memory reference and of operand.
no special calculations.

< Accumulator based CPU >

ADD LOCA;

first operand \rightarrow implied addressing mode.

and operand \rightarrow direct addressing mode



Absolute addressing mode can be used to represent / access LOCAL VARIABLES, GLOBAL VARIABLES, STATIC VARIABLES in the program.

Register Addressing Mode

- ① The operand is the contents of a processor register.
- ② The address of the register is given in the instruction.

ADD R1, R2, R3 $R1 \leftarrow R2 + R3$

Advantages compared to Direct Addressing mode -

- small address field in the instruction.
- no memory reference.

Indirect Addressing Mode

- ① 2 memory accesses required.

- ① The address part of instruction contains the address of memory location where the effective address of an operand is present.

- ① used in pointers.



Effective address
of operand = B (not A).

All instructions.

LDA X (m) and @M

ACC \leftarrow mem[mem[X]]

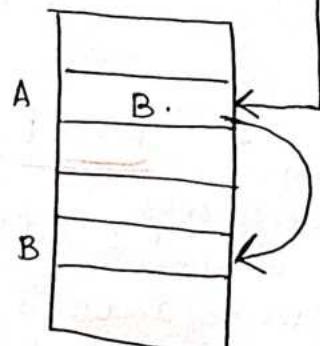
MAR \leftarrow IR(address)

MBR \leftarrow <MAR>

MAR \leftarrow [MBR]

MBR \leftarrow <MAR>

ACC \leftarrow MBR.

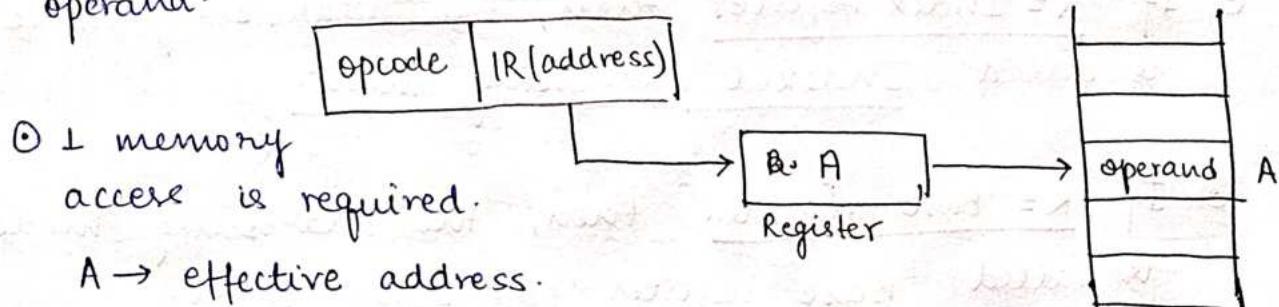


indicates indirect addressing mode

Effective address = [M].

Register indirect addressing mode

- The address part of instruction contains the address of register which contains the effective address of operand.

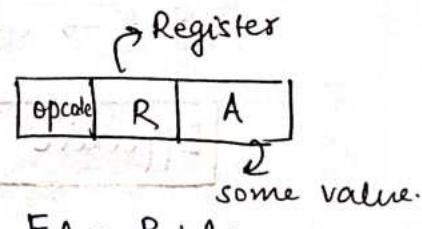


Advantages w.r.t. Indirect addressing mode

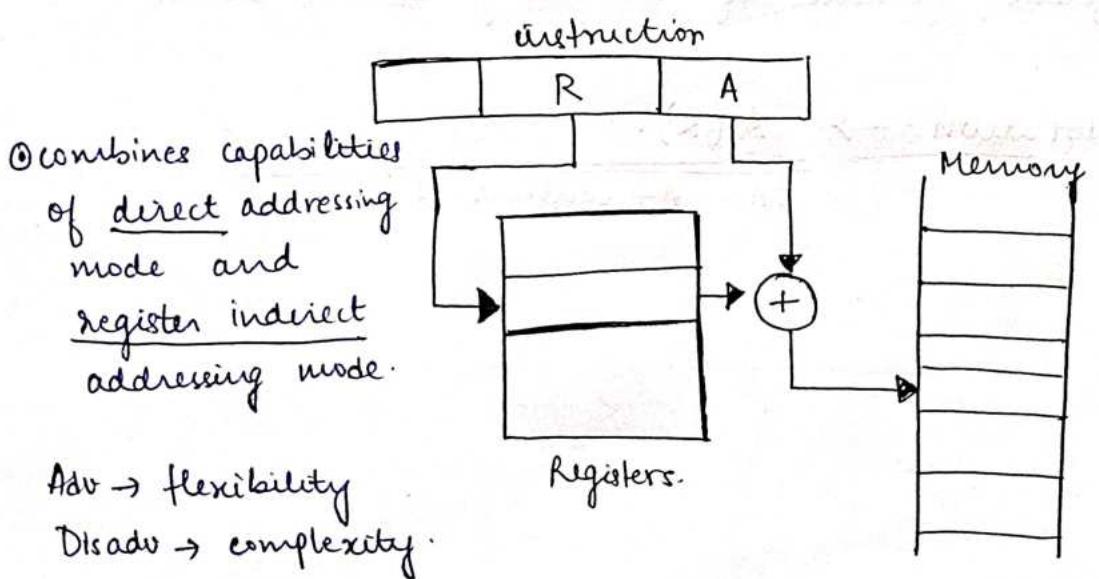
- one less memory reference is required.
- smaller address field.

Displacement Based Addressing Modes

- Relative Addressing Mode.
- Indexed Addressing Mode.
- Base Register Addressing Mode.



These addressing modes require that the addressing field of the instruction to be added to the content of a specific register in CPU.



- ⑥ If R = Program counter then, the addressing mode is called Relative addressing mode (PC relative).
- ⑦ If R = Index register then, the addressing mode is called Indexed addressing mode.
- ⑧ If R = Base register then, the addressing mode is called Base register addressing mode.

Indexed Addressing mode.

Arrays are implemented using indexed addressing mode.

- ① Index register holds index value → not the actual index values as in HLL
it is $HLL \times$ size of datatype
↳ it keeps on changing.
- ② A → contains base address of array.

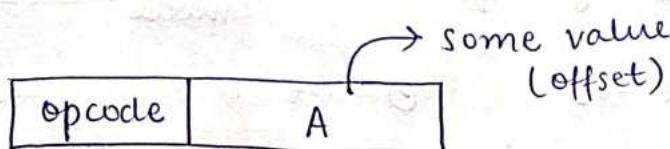
$$\boxed{\text{Effective address} = A + \text{content in index register.}}$$

- ③ used to access an array whose elements are in successive memory locations.
By incrementing / decrementing index register value, different element of the array can be accessed.

Notation $\rightarrow A(R)$.

$$EA = A + \text{content of } R.$$

Relative / PC relative addressing mode.



PC: special purpose register.
∴ implied.

$$EA = A + \text{content of PC}$$

$A \rightarrow$ signed value in 2's complement.

- ① used in JUMP instructions
- ② using relative addressing mode, we can write relocatable code.
↳ position independent code.
- ③ In relative addressing mode, the displacement ^{offset is} from the next instruction.

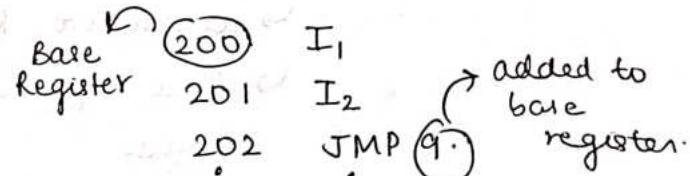
Base Register Addressing mode.

- ④ better and easier way to write relocatable code.

Base Register -

contains base address of program where OS has loaded/relocated the program.

- ⑤ When the OS loads the program, it sets the base register to the physical address where the program should start and the CPU adds this value to every address that the process uses when it is executing.



PC Relative addressing mode vs Base Register addressing mode

- ① offset value can be -ve
(signed no. in 2's complement)
- ② offset is always +ve.
(unsigned)
- ③ used for Branch instructions
- ④ used for Relocatable code

Autoincrement and autodecrement addressing modes

Autoincrement addressing mode - $(R) +$

after calcul. same as register indirect addressing mode.
(only difference is after calculating EA,
content in register R is incremented).
by operand size.

Autodecrement addressing mode - $-(R)$

↳ decrement R by operand size.

↳ find effective address as in register indirect mode

→ used in loops.

Cache Memory

CPU is much faster than ~~main~~ memory.

Instruction speeds —

ADD, SUB, SHIFT : 1 cycle

MULT : 3 cycles

LOAD | STORE : 100 cycles

} no requirement of main memory

} main memory required.

- ① Cache Memory → not as fast as registers → not as slow as main memory.
- ② Cache is expensive than main memory.
∴ limited.

Why caching works?

Cache works because it solves locality problem.

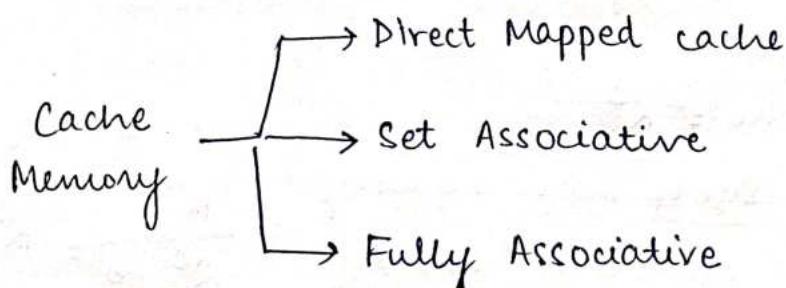
Locality

↳ Temporal Locality

Recently referenced items are likely to be referenced again in the near future.

↳ Spatial Locality

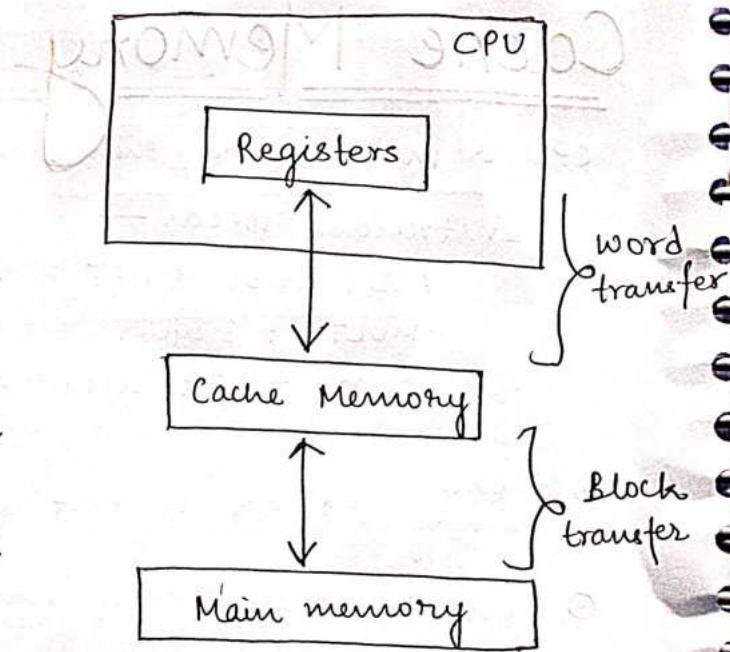
Items with nearby addresses tend to be referenced close together in time.



based on the location where data is kept from M.M. to cache.

Direct Mapped Cache

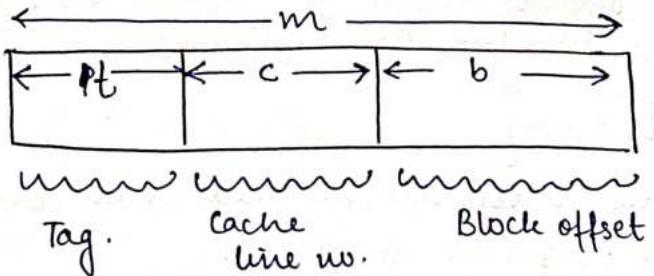
- ① simplest approach.
- ② Each main memory address maps to exactly one cache block.
- ③ Nearby blocks are mapped to different lines
→ spatial locality is maintained.



Mapping function

$$\text{Cache Line No.} = \frac{\text{Main memory Block No.}}{\% \text{ Number of cache blocks.}}$$

CPU generated main memory address.



$$\text{Main memory size} = 2^m \text{ bytes}$$

$$\text{Block size} = 2^b \text{ bytes}$$

$$\text{No. of lines in cache} = 2^c$$

$$\text{No. of tag bits} = \frac{\text{Size of main memory}}{\text{Size of cache memory}} = \frac{2^m}{2^c \cdot 2^b} = 2^{m-(b+c)}$$

$$\underline{t = m - (b+c)}.$$

Block offset - Denotes a particular word (or byte) within that block.

Tag - Used to identify whether the requested memory block is present in cache or not.

Line number - Represents to which line number of cache memory the address belongs.

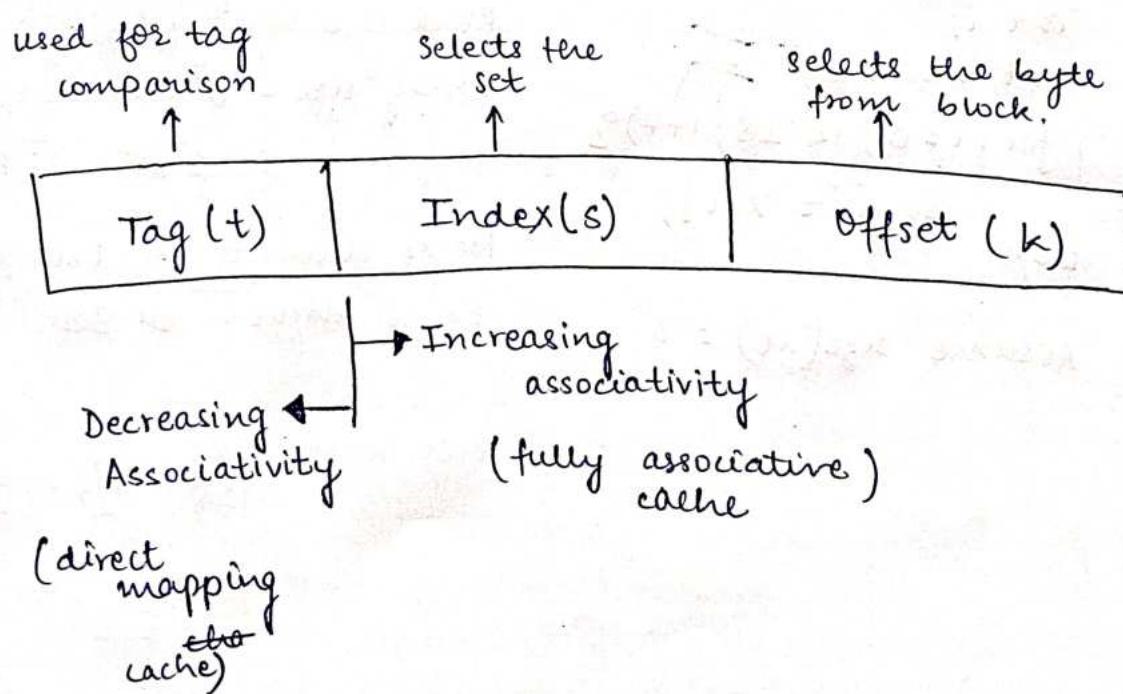
Set Associative Mapping

① A block can be placed anywhere within a set of locations in the cache.

② Cache blocks are grouped together into sets.

2-way set \equiv Set size is 2 \equiv #lines in a set = 2.

$$\text{Cache set No.} = \frac{\text{Main memory Block No.}}{\text{Number of sets}}$$

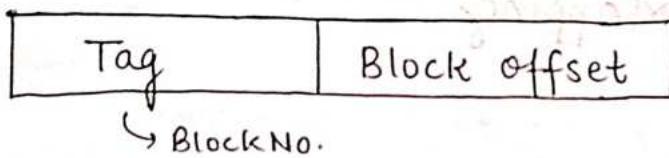


Fully Associative Cache Mapping -

- ① Each memory block can map anywhere in the cache (fully associative)

↳ most efficient use of space
↳ least efficient to check.

- ② No index bits.
(just tag and block offset).



↳ Block No.

check a fully associative cache -

→ Look at all cache slots in parallel.

2) If valid bit = 0,
then ignore.

3) If valid bit is 1
and Tag matches
then return that data.

Questions on loops & arrays

Solve this by starting with access sequence of main memory blocks — then mapping to cache lines.

Ques

What is the miss rate for a cache of size 512 bytes that is direct mapped and has 16 byte cache blocks.

```
int x[128];  
int i;  
int sum=0;  
for(i=0; i<128; i++) {  
    sum+= x[i];  
}
```

Assume size(int) = 4

Cache size = 2^9 bytes = 2^5 lines.

Block size = 2^4 bytes.

Array size = 2^7 elements

$$= 2^7 \times 2^2 = 2^9 \text{ bytes}$$

No. of elements in 1 block = $2^2 = 4$

No. of blocks = 32.

$$\therefore \text{Miss rate} = \frac{32}{128} = \frac{2^5}{2^7} = \frac{1}{2^2} = \underline{\underline{0.25}}$$

Ques A very small direct mapped 16 byte data cache with 2 cache lines
size(int) = 4 bytes.

```
int x[8], t=0;
for (int j=0; j<2; j++) {
    for (int i=0; i<8; i++)
        t+=x[i];
}
```

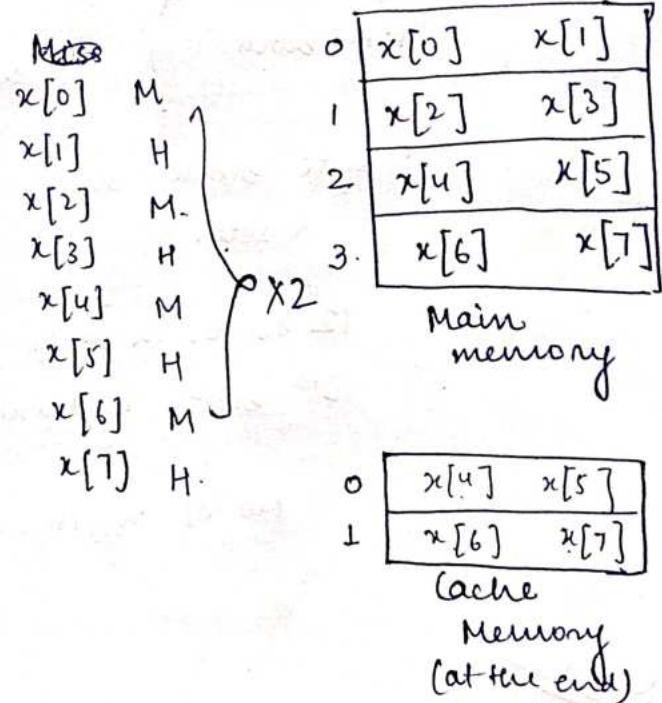
What is no. of misses

$$\text{No. of misses} = \frac{4 \times 2}{2} = 8$$

$$\text{Miss rate} = \frac{8}{16} = \frac{1}{2} = 50\%$$

$$\text{Block size / line size} = \frac{16}{2} = 8 \text{ bytes}$$

1 line can store 2 integer variables.



Ques

The matrix is declared as arr[5][5].

Code -

```
int i, j;
int sum=0;
for(i=0; i<5; i++) {
    for(j=0; j<5; j++)
        sum+=arr[i][j];
}
```

(i) Direct mapped

16 Byte Block size
4 blocks.

(ii) 2-way set associative
8 byte block size
2 sets.

(i) 1 block can contain $\frac{16}{4} = 4$ elements

$$\begin{aligned} \text{Miss rate} &= \frac{1}{25} \\ &\equiv \\ &= 28\% \end{aligned}$$

0	a(0,0)	a(0,1)	a(0,2)	a(0,3)	1M	3H
1	a(1,0)	a(1,1)	a(1,2)	a(1,3)	1M	3H
2	a(2,0)	a(2,1)	a(2,2)	a(2,3)	1M	3H
3	a(3,0)	a(3,1)	a(3,2)	a(3,3)	1M	3H
4	a(4,0)	a(4,1)	a(4,2)	a(4,3)	1M	3H

(ii) M.M. block 1
 M.M. block 2
 M.M. block 3
 M.M. block 4
 M.M. block 5
 M.M. block 6
 M.M. block 7

1 elem.

1M 3H
 1M 3H
 1M
 1M
 1M
 1M
 1M

28%

8 byte block size \Rightarrow 2 integers in 1 block.
 2 sets.

12 blocks in M.M. containing 2 integers
 13th block containing 1 integer

No. of misses = $12 + 1 = \underline{\underline{13}}$

Ques

```
for (k=0; k< ITERATIONS; k++) {  

    for (i=0; i< 5; i++) {  

        for (j=0; j< 5; j++) {  

            sum += arr[i][j];  

        }  

    }  

}
```

ITERATIONS = 2

Direct Mapped

64 byte block size
 2 sets. 2 lines.

No. of misses?

No. of integers in 1 block = $\frac{64}{4} = 16$

In main memory

0 - first 16 elements
 1 - next 9 elements

1M 1M } Iteration 1

In 2nd iteration

everything is already there so 0 misses

Total miss = $\frac{2}{50} = \frac{1}{25} = \underline{\underline{4\%}}$

ii) 2 way set associative
32 byte block size; 1 set

$$\text{No. of integers in 1 block} = \frac{32}{4} = 8$$

In main memory

Block 0	- First 8 elements	1M	X2 coz 2 iterations
Block 1	- Next 8 elements	1M	
Block 2	→ Next 8 elements	1M	
Block 3	→ 1 element	1M	

$$\therefore \text{No. of misses} = \frac{8}{25} = \underline{\underline{32\%}}$$

Ques

128x64 matrix of 32 bit integers

LOOP A

```
sum=0
for(i=0; i<128; i++) {
    for(j=0; j<64; j++) {
        sum += A[i][j];
    }
}
```

4 KB direct mapped cache with 32 byte cache lines.

LOOP B

```
sum=0
for(j=0; j<64; j++) {
    for(i=0; i<128; i++) {
        sum += A[i][j];
    }
}
```

$$\text{No. of integers in 1 line} = \frac{32}{4} = 8$$

$$\text{No. of blocks needed to store } 128 \times 64 \text{ integers} = \frac{128 \times 64}{8} = 1024$$

$$\text{No. of cache lines} = \frac{128 \times 64}{2^5} = 2^7 = 1024$$

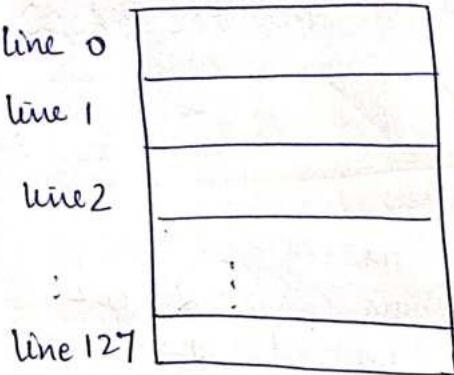
For LOOP A, No. of misses = No. of blocks = 1024

For LOOP B -

- A[0][0] → Block 0 in M.M.
- A[1][0] → Block 8 in MM.
- A[2][0] → Block 16 in MM.

Access pattern -

0, 8, 16, 24, 32, 40, 48, 56, ..., 104, 112, 120, 0, 8, ...



0, 128, 256, ...

\therefore No. of misses

= No. of elements

$$= 64 \times 128$$

$$= 2^6 \times 2^7 = 2^{13} = 8 \times 10^3 \\ = 8192$$

Access pattern of MM blocks in

1st iteration — 0, 8, 16, 24, ...

2nd iteration — 0, 8, 16, 24, ...

:

8th iteration — 0, 8, 16, 24, ...

9th iteration — 1, 9, 17, 25, ...

coz 8
elements are
in 1 block.

Ques
GATE 2006

32KB

A CPU has a direct mapped cache with 128 byte block size. Suppose a 2-D array of size 512×512 with elements that occupy 8 bytes each.

P1:

```
for(i=0; i<512; i++) {  
    for(j=0; j<512; j++) {  
        x += A[i][j];  
    }  
}
```

blocks occupied in MM. = $\frac{512 \times 512}{16} = 2^{14}$ No. of cache misses?

integers in 1 block = $\frac{128}{8} = 2^7 = 16$.

Main memory organization

(0,0)	(0,15)	B(0)
(0,16)	(0,31)	B(1)
:		
(1,0)	(1,15)	B(32)

P2:

```
for (i=0; i<512; i++) {  
    for (j=0; j<512; j++) {  
        x += A[j][i];  
    }  
}
```

cache lines = $\frac{2^7}{2^4} = 2^3 = 8$

P1: access sequence —

0, 1, 2, ...

0, 0, 0, ... (16 times)

1, 2, 1, ... (16 times)

No. of misses = $\frac{512 \times 512}{16} = \frac{2^{14}}{2^4} = 2^{10} = 1024$

P2: access sequence —

$$0, 32, 64, 96, \dots \quad 0, 32, \dots$$

$$d(0,0) = 0$$

10,321--

(0,1)

by the time $(0, 1)$ is asserted

line 0 of cache contains block 216.

$$\therefore \# \text{ misses} = \# \text{ elements} = 512 \times 512 = \underline{\underline{2^{18}}}$$

$$\text{Ratio of} \quad \frac{\alpha^{14}}{\alpha^{18}} = \frac{1}{\alpha^4} = \underline{\underline{\frac{1}{16}}}$$

GATE CSE
2007

Consider a machine with a main memory (byte addressable) of 2^{16} bytes. Assume that a direct mapped data cache consisting of 32 lines of 64 bytes each is used in the system. A 50×50 2-D array of bytes is stored in main memory starting from main memory location 1100H. The complete array is accessed twice.

Good
question!

No. of misses?

lines in cache = 32

(0,0)

$$(0, \frac{49}{50}) \quad (1, 0) \dots (1, 13)$$

$$\# \text{ integers in 1 block} = 64$$

(114) . . .

$$\# \text{ blocks occupied in main memory} = \left\lceil \frac{50 \times 56}{64} \right\rceil = \left\lceil \frac{2500}{64} \right\rceil = 40.$$

Access ~~req~~ sequence →

$0, 0, 0, 0, \dots \dots \quad 0, 1, 1, \dots \quad 1$

64 times 64 times.

No. . No. of blocks in LRU in cache = 32 No. of blocks = 40
first 8 blocks will be replaced. Rem

$$\text{No. of Cache misses} = \frac{40}{2} + 8 = \underline{\underline{48}} + 8 = 56 \quad \begin{matrix} \text{in 1st iteration} \\ \text{in 2nd iteration.} \end{matrix} \quad \begin{matrix} \text{as 4,} \\ \text{it is 4,} \\ \text{coz it is} \\ \text{from 110011} \end{matrix}$$

Cache Latency Good questions —

GATE 2020

A direct mapped cache memory of 1MB has a block size of 256 bytes. The cache has an access time of 3ns and a hit rate of 94%. During cache miss, it takes 20ns to bring the first ^{word of block} _{1 block} from main memory while each subsequent word takes 5ns. The word size is 64 bits. Avg memory access time?

$$\text{Block size} = 256 \text{ bytes} \quad \text{Word size} = 64 \text{ bits} = 8 \text{ bytes}$$

$$\therefore \# \text{ words in a block} = \frac{256}{8} = \frac{2^8}{2^3} = 32$$

$$\begin{aligned} \text{Time taken to see transfer 1 block from M.M. to} \\ \text{cache} &= 20 + 31 \times 5 = 20 + 155 = 175 \text{ ns.} \end{aligned}$$

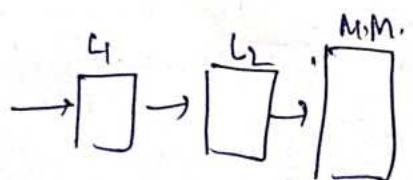
$$\begin{aligned} \text{Avg memory access time.} &= 0.94 \times 3 + 0.06 \times (3 + 175) \\ &= 0.94 \times 3 + 0.06 \times 178 \\ &= 2.82 + 10.68 = \underline{\underline{13.50 \text{ ns}}}. \end{aligned}$$

GATE 2017

In a 2 level cache system, the access times of L_1 and L_2 caches are 1 and 8 clock cycles. The miss penalty from the L_2 cache to main memory is 18 clock cycles. The miss rate of L_1 cache is twice miss rate of L_2 . The avg. memory access time of this cache system is 2 cycles. What is the miss rate of L_1 & L_2 ?

Let miss rate of $L_2 = x$

Miss rate of $L_1 = 2x$



$$\begin{aligned} \text{TMAT} &= \cancel{\text{misses}} + \cancel{\text{hit}} \\ \text{AMAT} &= (1 - 2x) \times 1 + 2x [(1 - x) \times (1 + 8) \\ &\quad + x \times (1 + 8 + 18)] \end{aligned}$$

$$\Rightarrow 2 = (1-2x) + 2x[9(1-x) + 27x]$$

$$\Rightarrow 2 = 1 - 2x + 18x(1-x) + 54x^2$$

$$\Rightarrow 2 = 1 - 2x + 18x - 18x^2 + 54x^2 = 1 + 16x + 36x^2$$

$$\Rightarrow 36x^2 + 16x - 1 = 0$$

$$\begin{array}{r} 16 \\ 16 \\ \hline 96 \\ 256 \\ \hline 144 \\ 400 \\ \hline \end{array} \Rightarrow x = \frac{-16 \pm \sqrt{16^2 + 4 \times 36}}{2 \times 36} = \frac{-16 + 20}{72} = \frac{20 - 16}{72} = \frac{4}{72} = \frac{1}{18}$$

0.0526.

In multilevel cache,

$$\text{The average memory access time} = H_1 \times \frac{L_1 \text{ cache access time}}{\text{access time}} + (1-H_1)(H_2) \frac{(L_1 + L_2)}{\text{access time}} + (1-H_1)(1-H_2) \frac{\text{Main memory access time}}{\text{access time}}$$

Cache Miss

Compulsory Miss
(cold miss)

occurs when data is accessed for the 1st time.

unavoidable miss

Capacity Miss

occurs when cache can't contain all the data needed by the system.

occurs when working set is larger than cache size.

cache filled to capacity.

can be reduced by increasing cache size.

Conflict Misses

multiple data items get mapped to same cache location.

can be reduced by increasing associativity of cache

PIPELINING

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

Separating the hardware to execute multiple instructions at the same time.

Improves efficiency.

Instruction cycle — IF ID EX . DM WB .

5 stage pipeline (1 clock cycle / stage)

10 instructions in the program.

How many cycles in Non pipelining v/s pipelining

Non pipeline system —

$$\text{No. of cycles} = 10 \times 5 = \underline{\underline{50}}$$

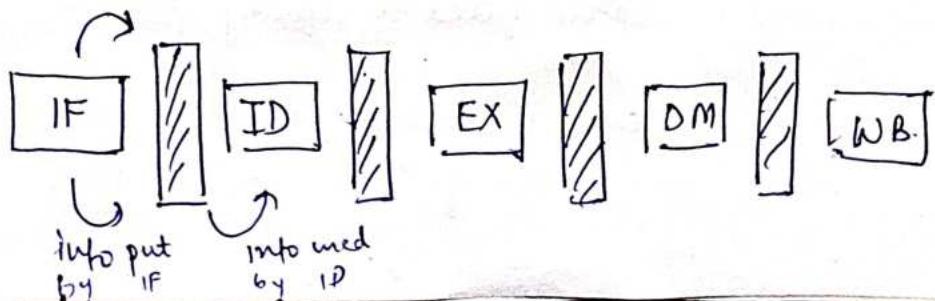
Pipeline system —

$$5 + 10 - 1 = \underline{\underline{14}}$$

Inter stage Buffer

→ used to store information.

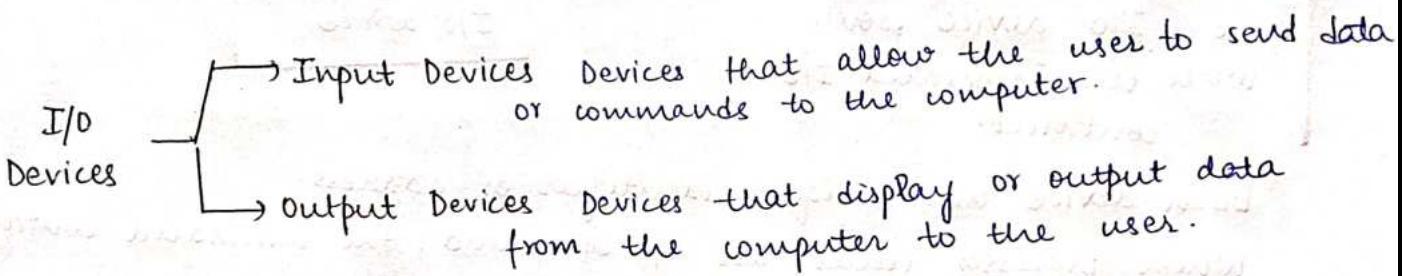
→ information is stored by later stages.



I/O Interfacing & Interrupts

I/O devices — h/w components that enable a computer system to interact with the external world.

→ Peripherals.

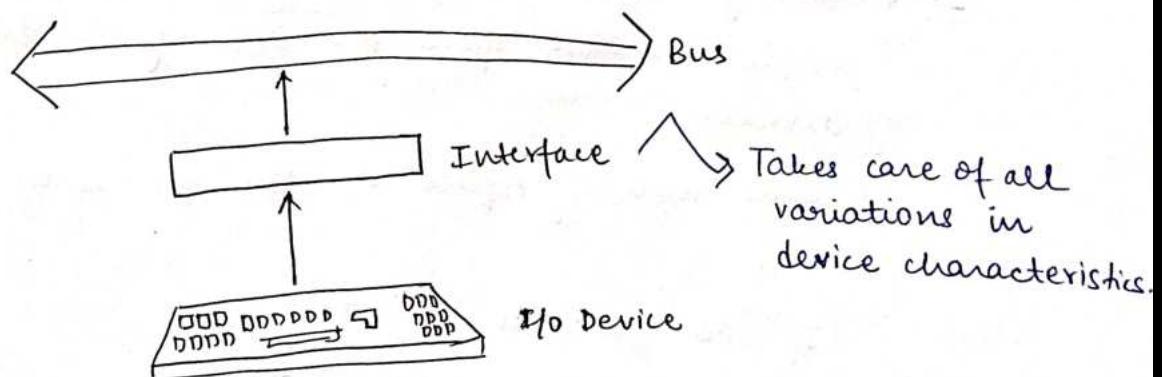


The reasons why peripherals are not directly connected to system bus —

- ↳ wide variety of peripherals with widely varying speeds (I/O devices are very much slower than CPU).
- ↳ Data transfer rate is irregular or regular.
- ↳ Different I/O devices have different data formats.

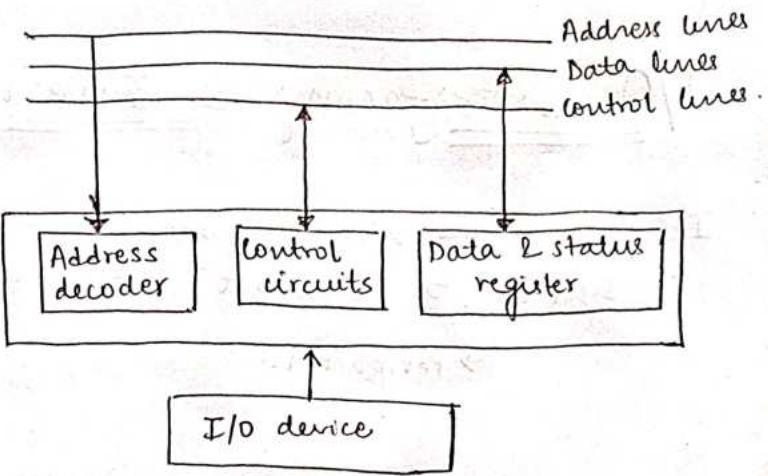
I/O Interface

Device interface contains Device Controller.



I/O interface contains

- ↳ Address Decoder
- ↳ control circuits
- ↳ Data & status Registers.



CPU communicates with the I/O device ~~only~~ with the associated I/O controller.

Each device has unique identifier or address.

When processor issues an I/O command, the command contains address of I/O device.

I/O Port

- ↳ physical point at which the device connects to a computer
- ↳ address used by CPU to communicate with peripheral devices.

I/O port = Address of I/O device.

Connecting I/O devices to system bus - (Determines how the devices are addressed)

1. Memory Mapped I/O

single address space for memory locations and I/O devices.

same memory space is used for both.

If CPU generates n bit address.

Then, address space = 2^n

Out of these 2^n addresses, some addresses are reserved for I/O devices.

Every device is assigned a specific memory address and the CPU communicates with the devices by reading from or writing to those memory addresses.

Ex - printer assigned address 0xFF00.

Data sent to 0xFF00 will be treated as I/P to printer

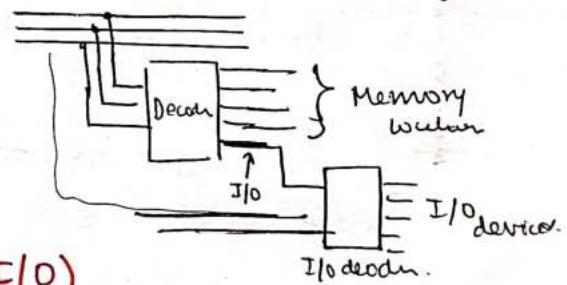
Advantages

- ① Same instructions used for accessing memory (LOAD, STORE etc.) are also used for accessing I/O devices (No need for special ins. to handle I/O).

Disadvantages

- ② Reduced memory space. (since memory & I/O share same address space available memory is reduced).

→ A single I/O device can have multiple addresses in memory mapped I/O.



2. I/O Mapped I/O (Isolated I/O).

- ① I/O Devices have their own address space, different from memory address space.
- ② special I/O instructions are needed to access the I/O address space : IN and OUT (to access I/O ports).

③ same address lines.

④ separate decoder for I/O device.

⑤ CPU generates a control signal

I_{O/M} along with address

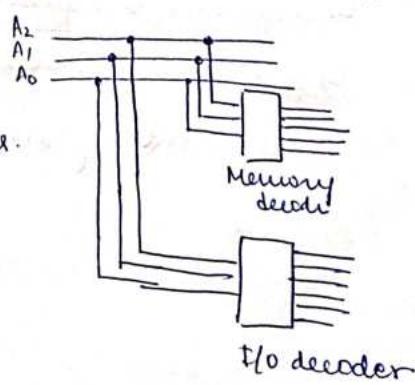
enable
signal for
each decoder

↓
0 → access main memory

↓
1 → access I/O

(One is active low other is active high.)

↓ or add NOT gate.



Advantage

Since I/O & memory addresses are separate, full memory is available for use by the system memory.

Disadvantage

require special I/O instructions - IN, OUT to access I/O devices



① Memory mapped I/O is supported by every processor.
(coz no special treatment by processor)

while I/O mapped I/O require $I/O/M$ signal, additional decoder, IN, OUT instructions.



② A system can use both memory mapped I/O and I/O mapped I/O (isolated I/O) at the same time.



③ Memory mapped I/O is faster than I/O mapped I/O.

Memory mapped I/O \rightarrow use LOAD & STORE \rightarrow memory instructions
:- faster!

I/O Mapped I/O \rightarrow use IN & OUT \rightarrow I/O instructions
:- slower

Benefits of memory mapped I/O over isolated I/O.

- ↳ cheaper
- ↳ faster
- ↳ easier to build
- ↳ consumes less power.
- ↳ physically smaller
- ↳ supported by all systems

I/O Data Transfer Techniques.

Data transfer b/w the main memory and the I/O device.

3 techniques

- ↳ Programmed I/O
- ↳ Interrupt Driven I/O
- ↳ Direct Memory Access

(CPU is involved). done
 Data transfer is by the help of CPU.

Issues in data transfer

- ↳ Data Transfer rate of peripherals is slower than the transfer rate of CPU.

1. Programmed I/O

(Program controlled I/O or Polled I/O)

- ① CPU executes a program that transfers data b/w I/O device and main memory.

- ② CPU does not do any useful work during polling.

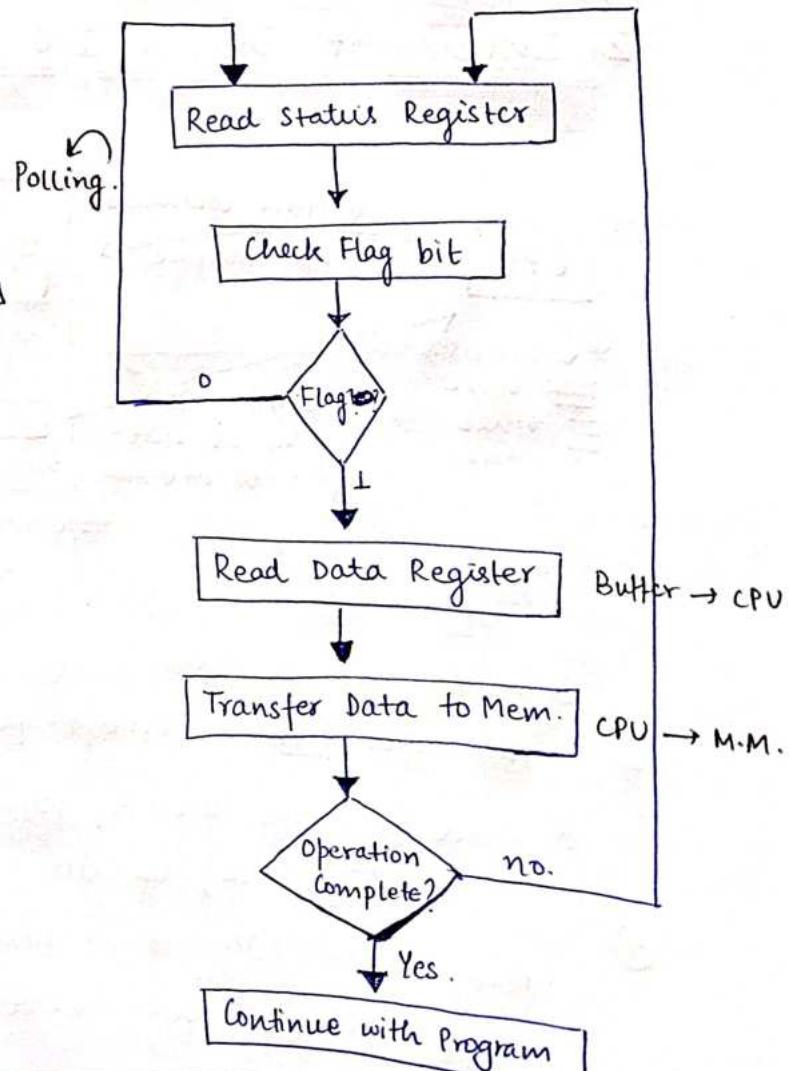
- ③ Data transfer is initiated by the program run by the CPU.

- ④ Processor is totally in control and does all the work.

- ⑤ I/O module does not inform CPU directly.

- ⑥ I/O module does not interrupt CPU.

- ⑦ CPU checks status bits periodically.



① The I/O device itself doesn't have the ability to signal the CPU for attention or start the transfer. The device passively waits for CPU to interact with it.

Advantage -

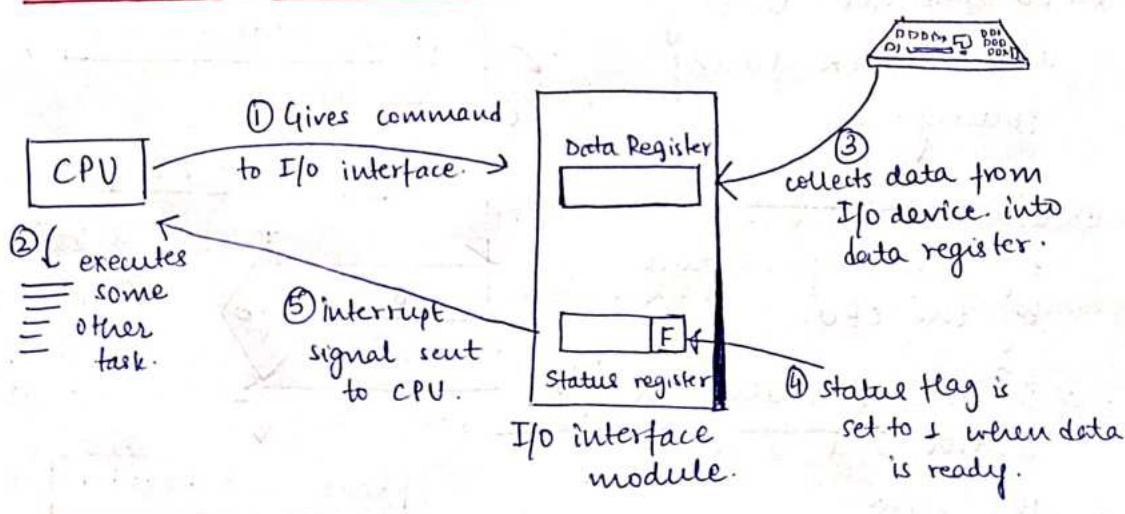
↳ simple - processor is totally in control and does all the work.

Disadvantage -

↳ Polling overhead can consume a lot of CPU time.

→ Programmed I/O is used in embedded systems (which perform simplest tasks) (no real time computation needed to be done).

2. Interrupt Driven I/O



- ⑥ At the end of each instruction cycle, the processor checks for interrupts (on interrupt pin).
- ⑦ When interrupt occurs, processor saves the current work (PC & registers) of current prg. and processes the interrupt.
 - Reads data from I/O module & stores in memory.
 - Then, restores the context of program it was working on.

Overheads in interrupt driven I/O -

- ↳ saving context of current work when interrupt occurs.
- ↳ context switch (after interrupt is handled).

① Interrupt driven I/O is generally more efficient than programmed I/O in reducing CPU idling.

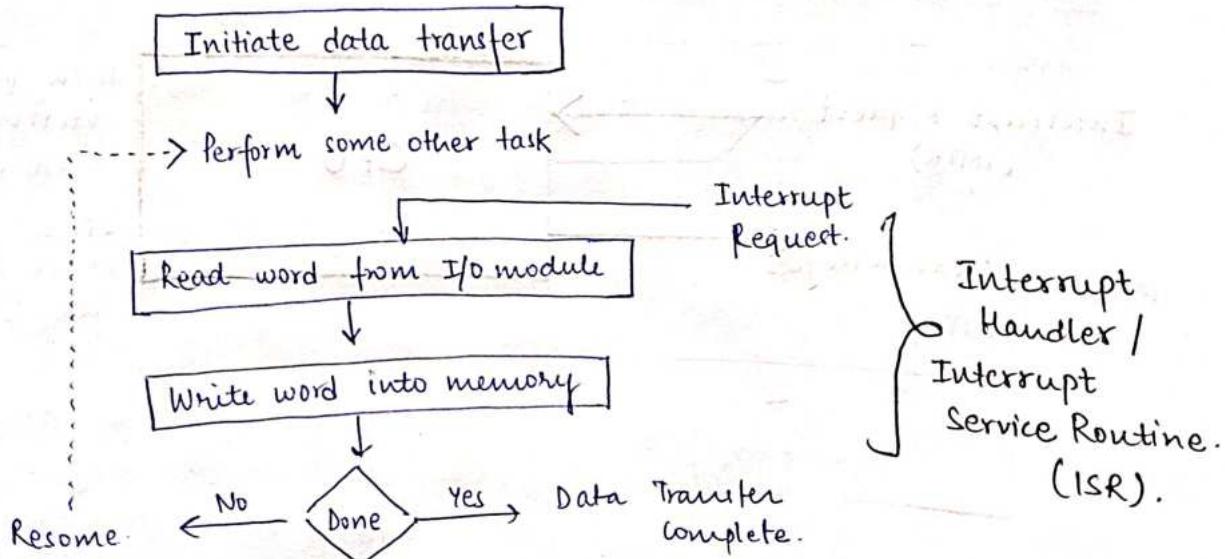
② In case of small and frequent data transfers, programmed I/O is faster than interrupt driven I/O.
↳ for fast I/O devices.
↳ a lot of context switches.

Interrupt driven I/O is usually faster than programmed I/O
not always.

SUMMARY — INTERRUPTS ARE NOT ALWAYS BETTER THAN POLLING.

If the rate of receiving data is high, context switching for interrupts adds more overhead than simply polling.

$$\text{Speedup of A over B} = \frac{\text{Execution time of B}}{\text{Execution time of A.}}$$



Interrupt

- ① Interrupts allow I/O devices to send signal to the CPU that they require attention.
- ② Signals sent to the processor to indicate that an event needs immediate attention.

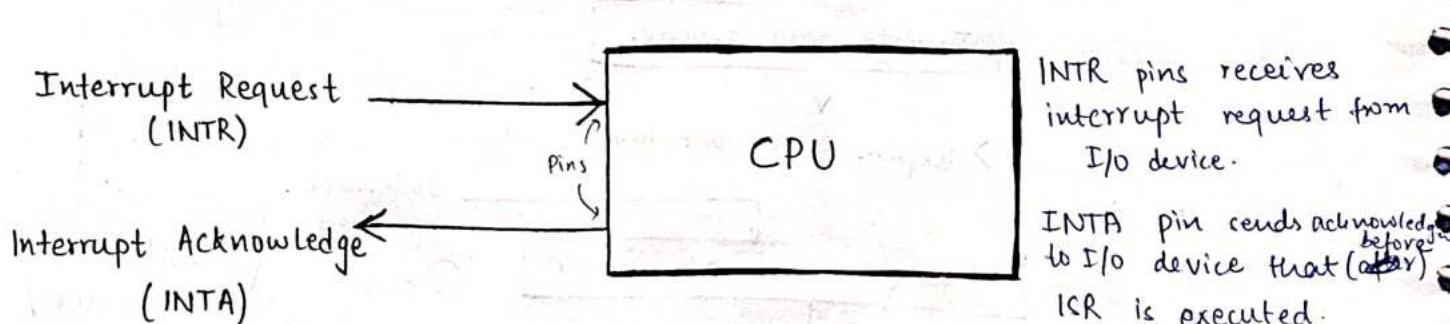
Interrupt Service Routine -

- ↳ Interrupt handler, Interrupt Service subroutine (ISR)
- ① ISR is the program i.e. instructions to handle a particular sort of interrupt.
- ② ISR is a special function / program that gets executed in response to an interrupt signal. (to service the interrupt)

Each I/O device has its own ISR.

IMPORTANT

- ① For all types of interrupts, their ISR are stored in main memory.
- ② ISR is executed by CPU just like any other program.

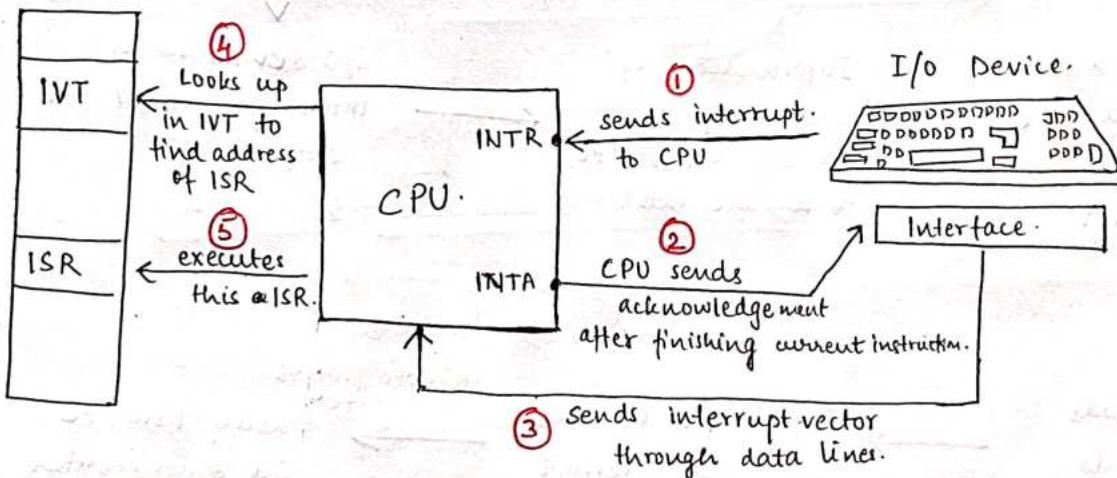


Vectored Interrupts -

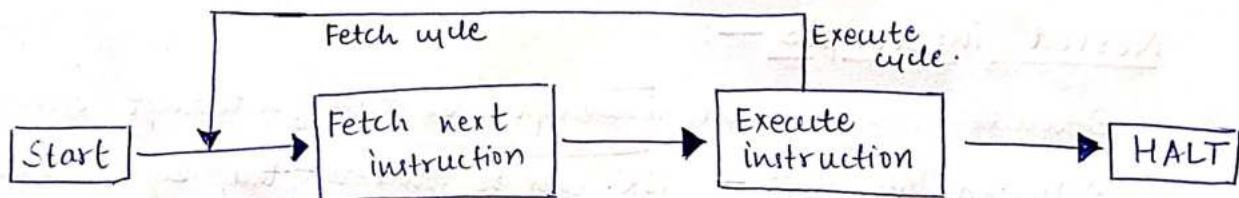
Each device is assigned a unique identification number called interrupt vector.

Interrupt Vector Table is a table that contains interrupt vector along with the corresponding ISR location in main memory (predefined area in main memory).

Main memory.

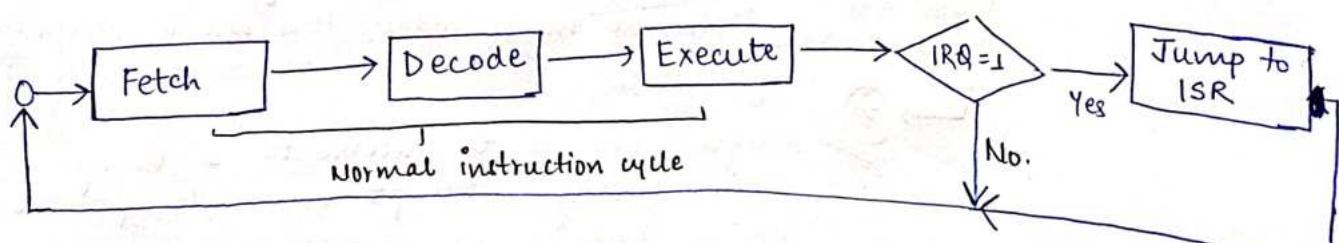


Interrupt cycle -

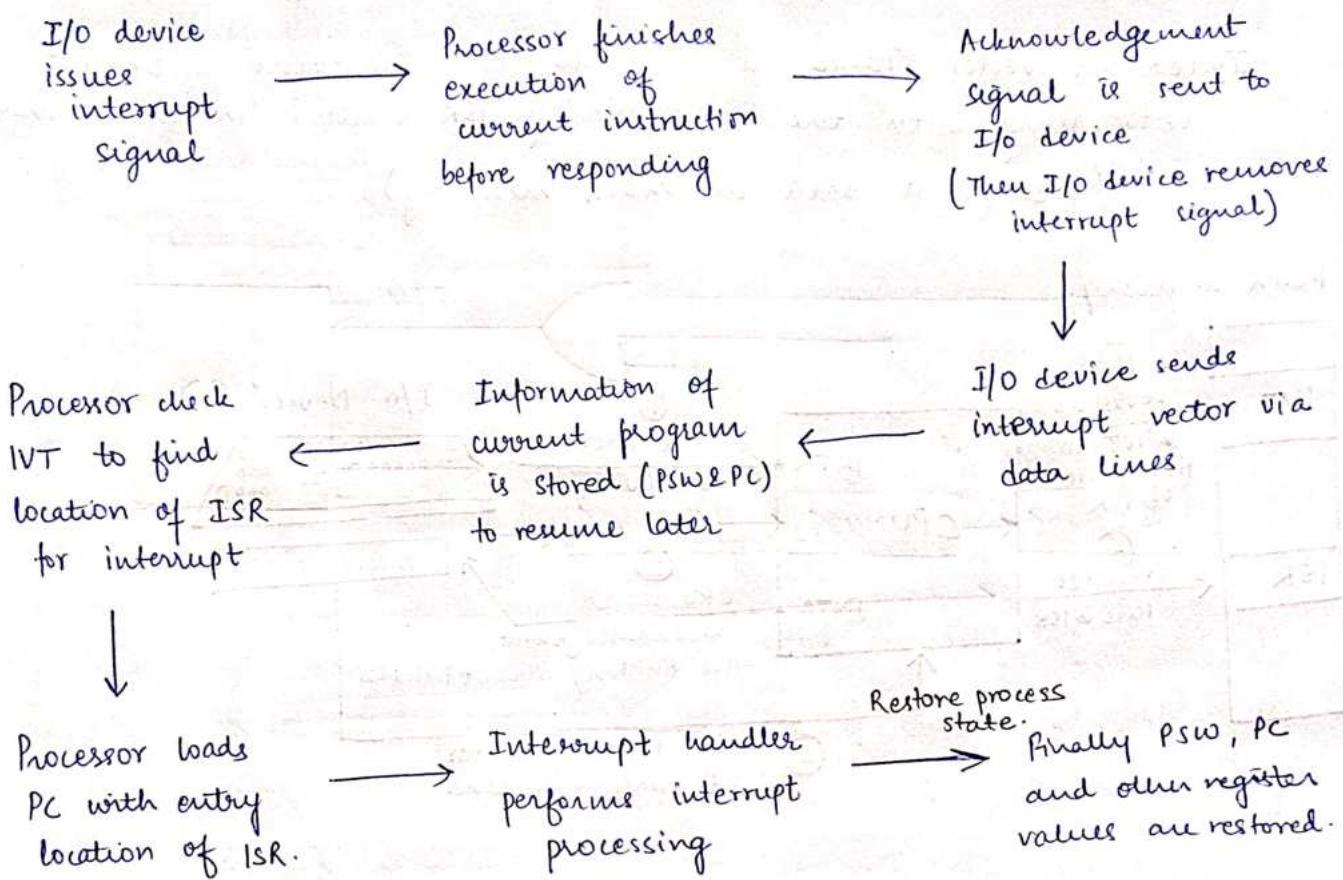


Basic instruction cycle.

To accommodate interrupts, an interrupt cycle is added into instruction cycle for checking the availability of interrupt.

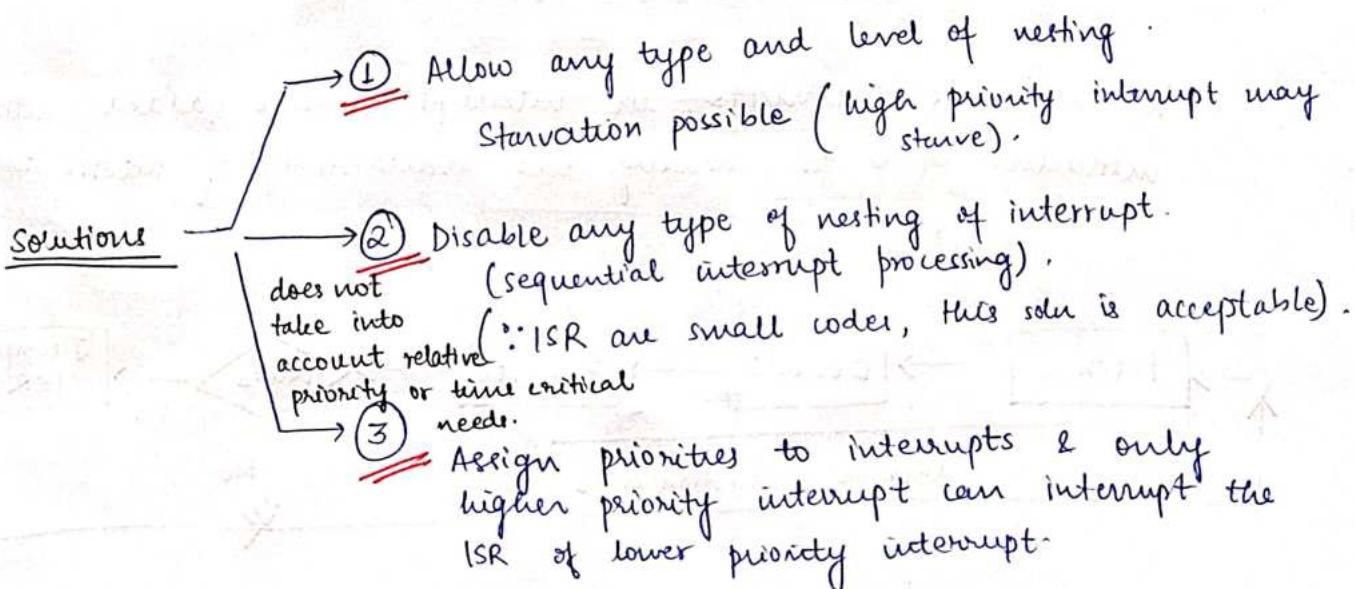


Interrupt Processing -



Nested interrupts -

- ① while servicing an interrupt, another interrupt occurs.
- ② Nested interrupt - ISR can be interrupted by another interrupt.

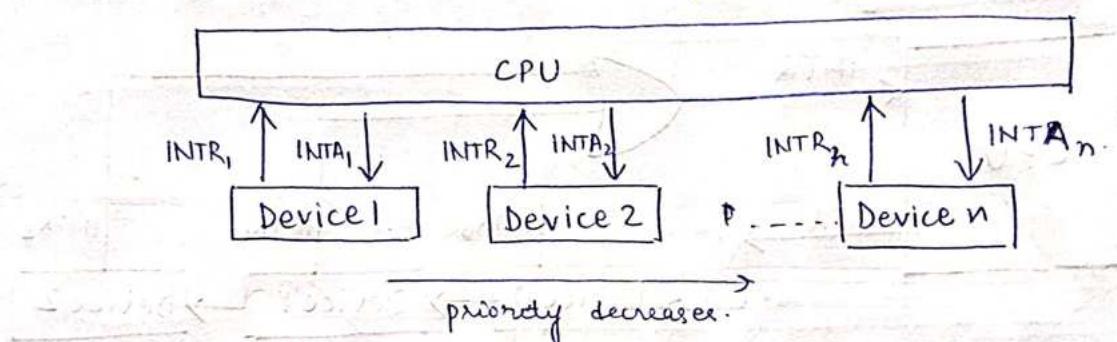


Multiple interrupts at same time.

⑥ CPU serves the highest priority interrupt first among all the current interrupts.

Solution 1:- Multiple interrupt lines in the order of priority.

INTR_i has higher priority than INTR_j where i < j



Drawback



↳ Large no. of dedicated lines required (burden on CPU in administrating all these lines).

(In this soln, we can have very limited number of I/O devices connected to CPU).

Solution 2:- Polling method (single interrupt line)

CPU asks each device one by one if it caused the interrupt using system bus.
^{I/O}

Each I/O device will have one system bus cycle to answer yes.

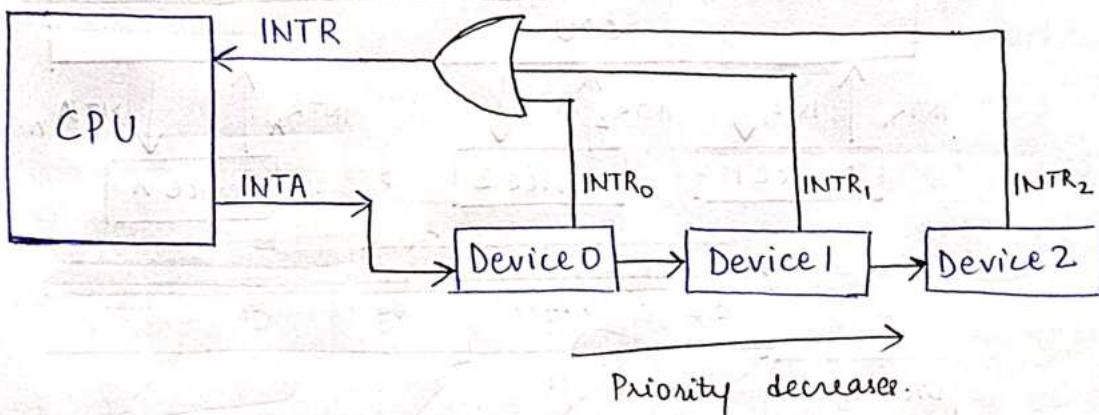
Drawback



↳ Time consuming.

Solution 3: Daisy chaining arrangement for assigning priority.

effective way of arranging devices physically in the order of priorities.



- ① The devices closer to CPU have higher priorities.
- ② I/O devices are physically ordered by ~~per~~ priority.
- ③ INTA propagates serially through all devices.
- ④ Low priority device may have a danger of STARVATION.

3. Direct Memory Access (DMA)

DMA mechanism can be used for data transfer, without involvement of CPU.

① I/O device is attached to DMA Controller.

DMA controller - Hardwired controller that enables direct data transfer b/w I/O device and memory without CPU intervention.

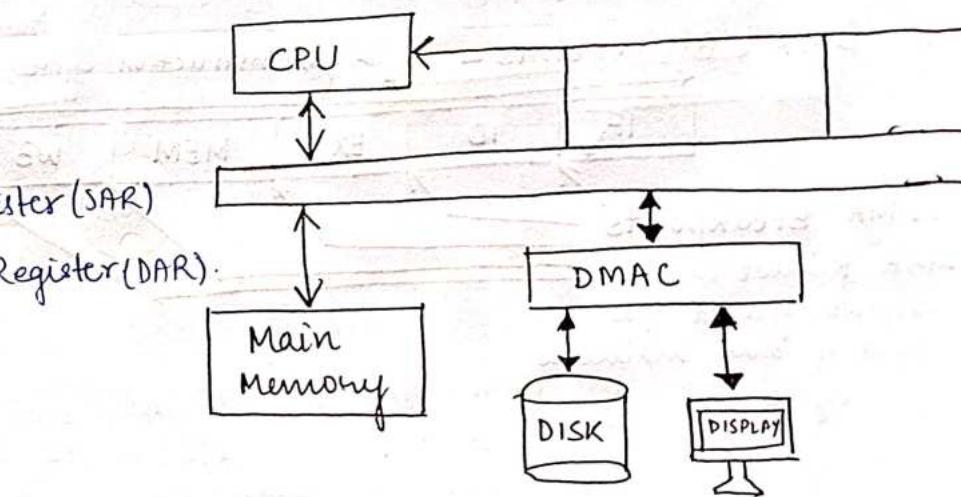
Important Components in DMA Controller.

1. Source Address Register (SAR)

2. Destination Address Register (DAR)

3. Count Register

4. Mode Register.



SAR - stores the memory address where data is being read from.

DAR - stores the data register memory address where the data will be written to.

Count Register - it holds the number of bytes or words to be transferred.

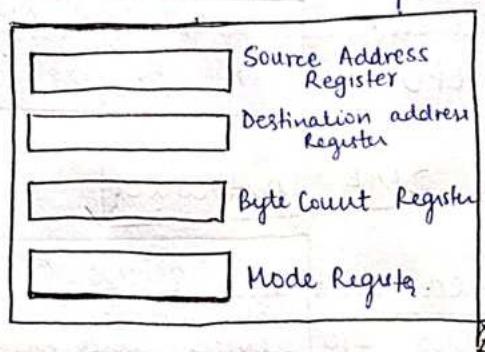
→ as DMA controller transfers data, the value in this register is decremented.

→ when count reaches 0, DMA transfer is complete and the controller generates an interrupt to the CPU.

Mode Register - defines the mode of operation

↳ Burst mode (Transfers a block of data in one go)

↳ Cycle Stealing mode (Transfers 1 byte at a time while alternating bus control with CPU)

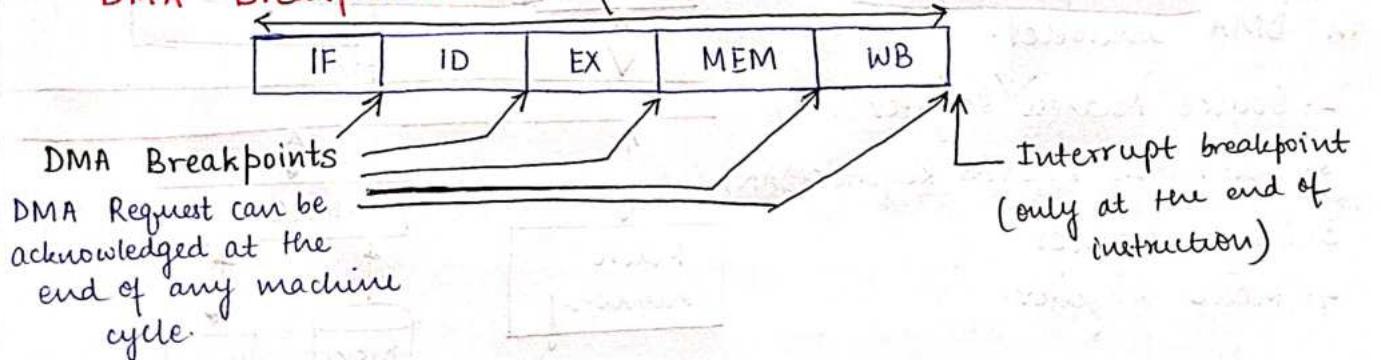


DMA controller
(mini processor)

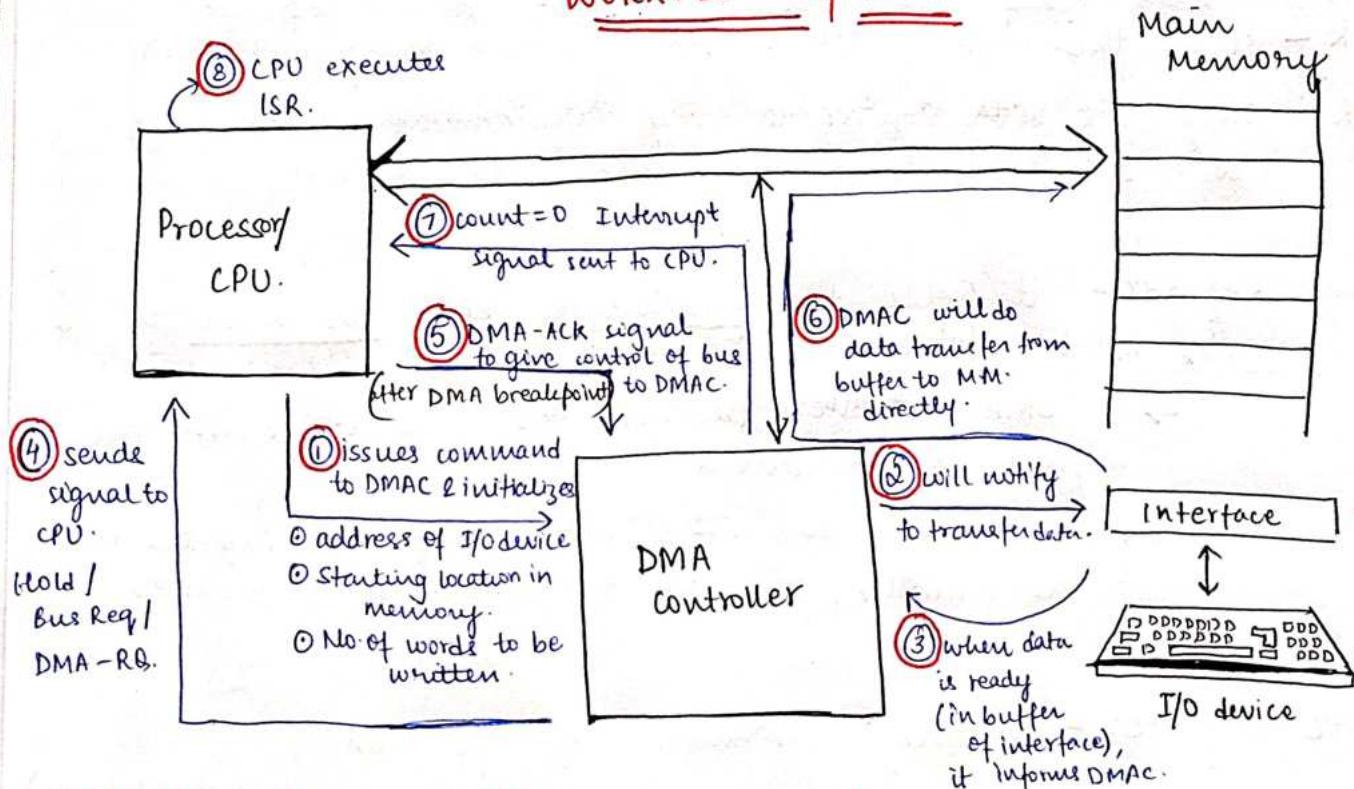
only used for data transfer.

⑥ CPU or DMAC — only one of them can be the Bus Master at any given time.

DMA Breakpoints -



WORKFLOW of DMA



④ CPU does other work (not involving bus) b/w steps 5 and 7.

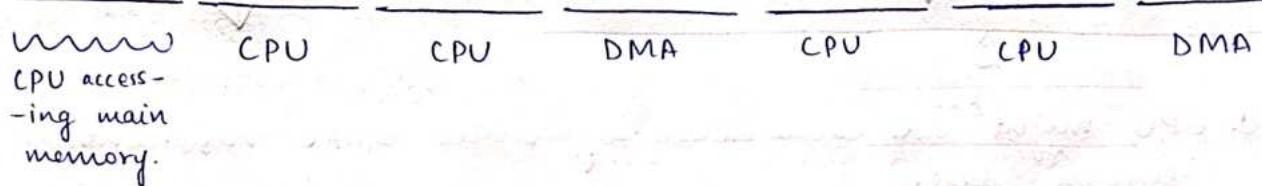
CPU provides SAR, DAR, count register to DMAC.

- ④ Processor is involved only at the beginning and at the end of the transfer.
- ⑤ DMA can be acknowledged during instruction execution but interrupt is acknowledged only at the end of instruction execution.
(wz no context switch in case of DMA-ACK).

Modes of DMA Transfer -

① Cycle Stealing Mode -

- ⑥ DMAC requests for system bus only for 1-2 cycles (preferably when CPU is not using memory)

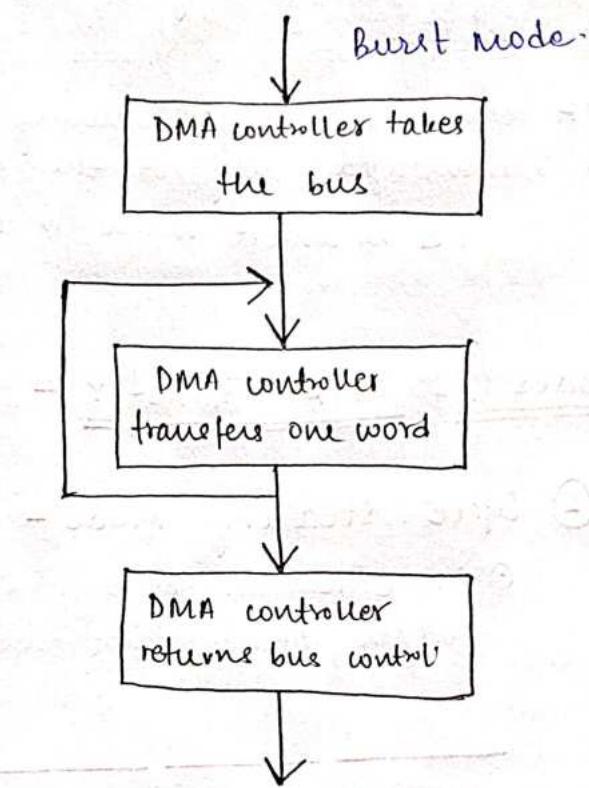
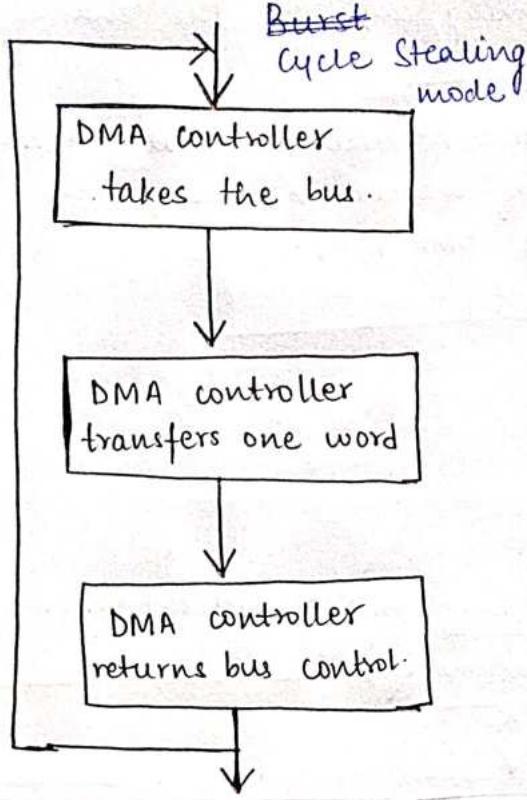


The DMA module transfers one word and returns control to the processor.

② Burst Mode

- ⑦ DMA controller transfers the whole block of data without interruption.
- ⑧ CPU lies idle during this period as it cannot fetch new instruction from memory.
- ⑨ Results in max. possible data transfer rate.

Slow I/O devices — cycle stealing mode
Fast I/O devices — burst mode } decided by CPU during DMA controller initialization.



- ① CPU might use cache memory while DMAC has control of the system bus.

Even if CPU may be idle during the DMA Transfer, DMA is still more efficient and better performing than interrupt driven I/O for large high speed data transfers.

↳ coz minimal CPU involvement & reduced overhead caused by frequent interrupts and context switching.

- * ① For small and infrequent data transfers, interrupt driven I/O is better than DMA.

↳ DMA initialization takes lot of time + interrupt at frequent

Slowdown of one activity due to another activity.

$$\text{Slowdown percentage} = \frac{\text{Time taken by interrupting activity}}{\text{Total time available}} \times 100\%$$

Ques ISRO 2013

Ques. DMA module is transferring characters to memory using cycle stealing from a device transmitting at 9600 bps. The processor is processing instructions at rate 1 MIPS (million instructions per second). By how much processor is slowed down due to DMA?

Good ques. // Processor speed = 1 MIPS = 10^6 bytes per second = $10^6 \times 8$ bits per second

If no DMA activity,

CPU in 1 second has 10^6 memory accesses.

DMA — 9600 bps.

cycle stealing mode ; 1 byte is transferred.

In 1 sec — 9600 bits = $\frac{9600}{8}$ bytes = 1200 bytes

∴ In 1 second, DMA accesses memory 1200 times

Ideally, CPU would have accessed memory 10^6 times

$$\therefore \text{Slowdown \% of CPU} = \frac{1200}{10^6} \times 100 = \underline{\underline{0.12\%}}$$

3hr video
watch all que.

I/O Lec 6 - DMA all GATE P/Qs.
90 classes for GATE CS.

Priyanshu