Basic Computer Organization

Main parts of computer system

- **Processor** : executes program
- **Main memory** : holds program & data.
- **IO devices** : for communication with outside.

Machine instruction :

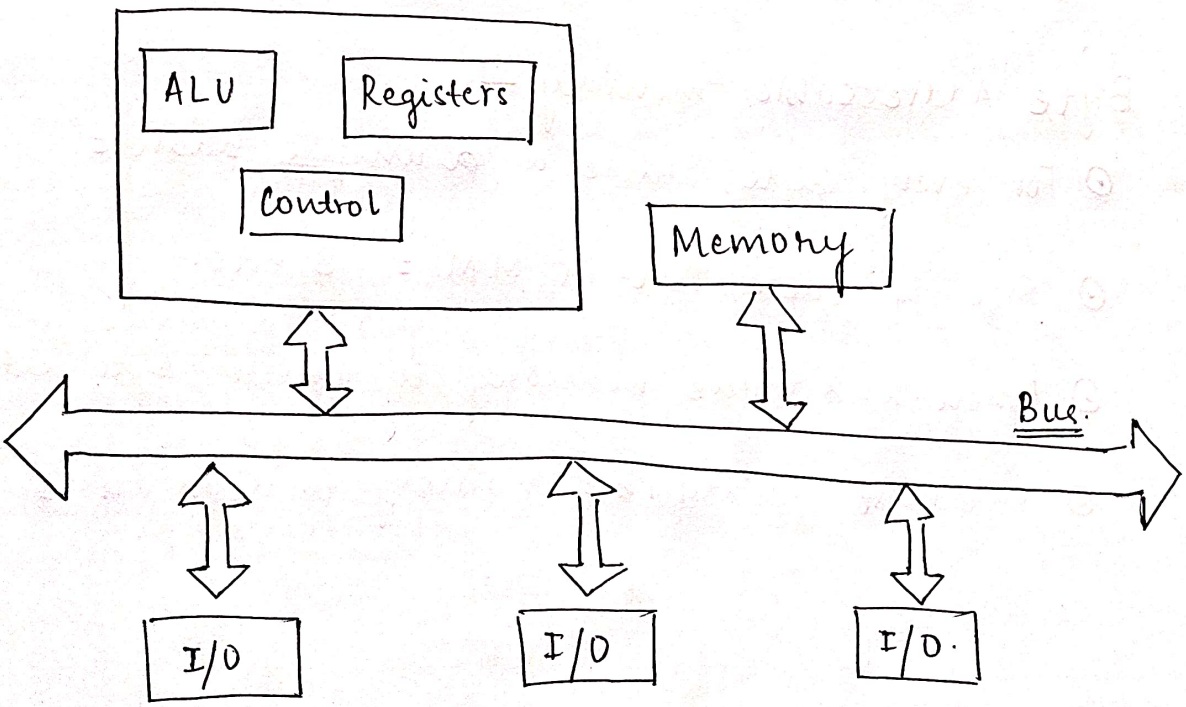
Description of premitive operation that machine hardware is able to execute.

inbuilt in h/w

eg → 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 : <number of cells> x <size of each cell>

$2^k \times x$.

⊙ Main memory is a collection of storage location, each with a unique identifier called address.

⊙ A program (data & instruction) should be stored in main memory in ~~org~~ order to get executed.

Address space

⊙ Total no. of uniquely identifiable locations in memory is called address space.

Byte Addressable memory –

⊙ For every byte, there is a unique address

⊙ Size of each cell in M.M. = 8 bits.

⊙ When $x=8$, the memory is called byte addressable.

⊙ Smallest identifiable memory location is 1 byte.

Word Addressable Memory.

Word – Concept.

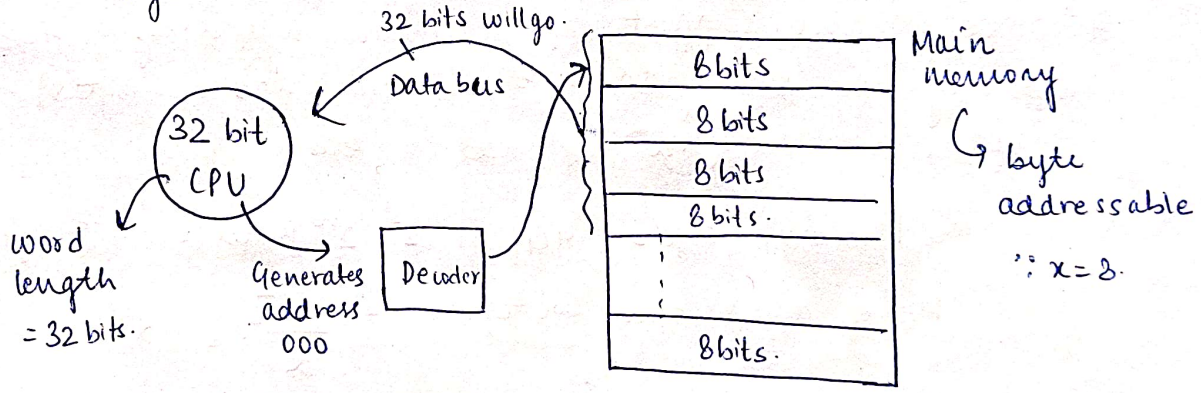
64 bit CPU } 64 and 32 are word size.
32 bit CPU } (word length of CPU)

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

Word
↳ Word is a property of the Processor (CPU) → not of memory.
↳ 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
∴ b 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.)

#Data lines = ⊠ Word length (property of CPU).

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

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

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

Number of data lines = 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

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

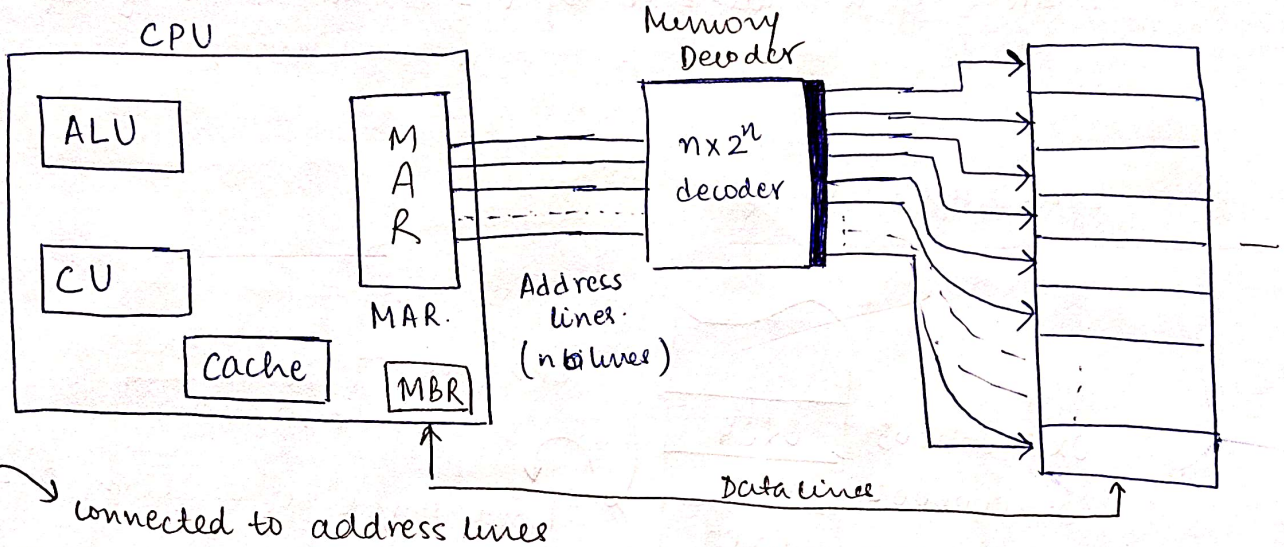
No. of addressable locations = $\frac{2^{32}}{2} = 2^{31}$

∴ No. of bits req. to identify each cell = 31

∴ No. of bits in address line = 31.

Memory Decoder

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



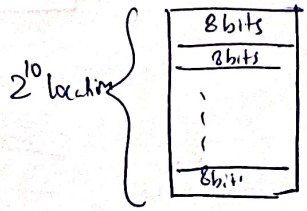
★ Memory Address Register → connected to address lines
- MAR stores the address to access main memory

Memory Buffer Register → connected to data lines.
- MBR contains value to be stored in memory or the last value read from the memory.

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 is _____ ?

byte addressable 1KB RAM.



Size of decoder

10×2^{10}

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

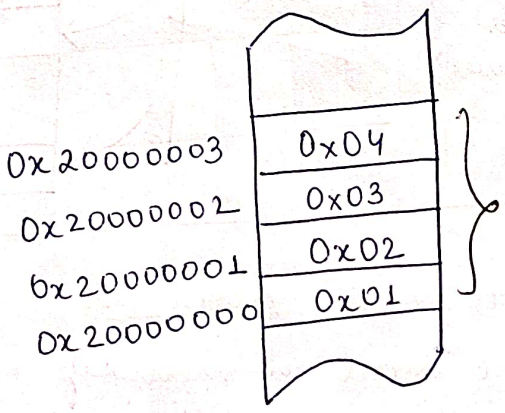
$\therefore m+n = 10 + 1024 = 1034$

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.



CPU generates the address 0x20000000

Word stored at location 0x20000000

The data that goes to CPU is 0x04030201 or 0x01020304.

Endian-ness is a property of CPU (system) (not of main memory)

Depends upon Endianness.

This depends upon the byte ordering used.

0x04 03 02 01

04 → Big end, Most significant byte

01 → little end, Least significant byte.

Endian-ness

→ Big Endian
Start with big end.
0x04030201
(to be stored in Mem.)

A	0x04
A+1	0x03
A+2	0x02
A+3	0x01

→ Little Endian.
Start with little end
0x04030201

A	0x01
A+1	0x02
A+2	0x03
A+3	0x04

1. Byte Ordering for a Multi-Byte data

(in CPU register)

Data to be stored = 0001 0001, 0001 1010, 0010 1011, 0011 1100.

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 ~~small~~ 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

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

4 bytes.

Big endian

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

little endian

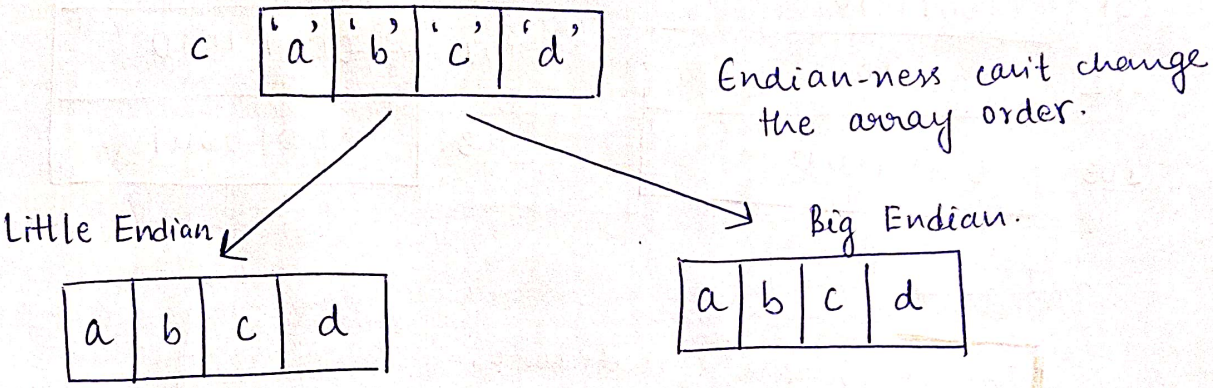
A	6D
A+1	3B
A+2	00
A+3.	00

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 premitive data type items, not to string, arrays or structs.

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



@ 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~~ b is `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.

@ 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 —

00000000 00000000

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

2. The maximum unsigned number.

The max unsigned no. with 32 bits —

11111111 11111111

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

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.

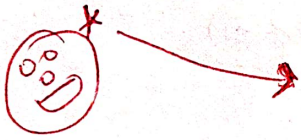
will be +ve.

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

∴ Maximum 2's complement number = 0111 ... 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 = 1000 0000 0000

Big endian = 80 00 00 00
little endian = 00 00 00 80.
} different.

System Bus

↳ Electrical wires / lines ↗ can carry 1 bit of data at any point of time.

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

System Bus → Address Bus
System Bus → Data Bus
System Bus → Control Bus.

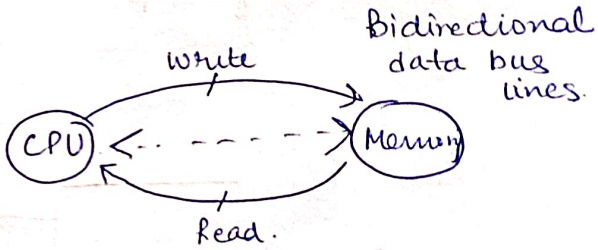
} classification is done based on the type of information they carry.

Address Bus

carry addresses generated by the CPU.
unidirectional from CPU to memory / I/O devices.

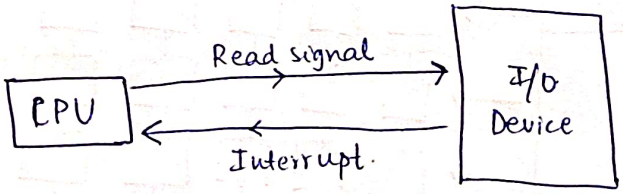
Data Bus

carries data / instructions
bidirectional



Control Bus

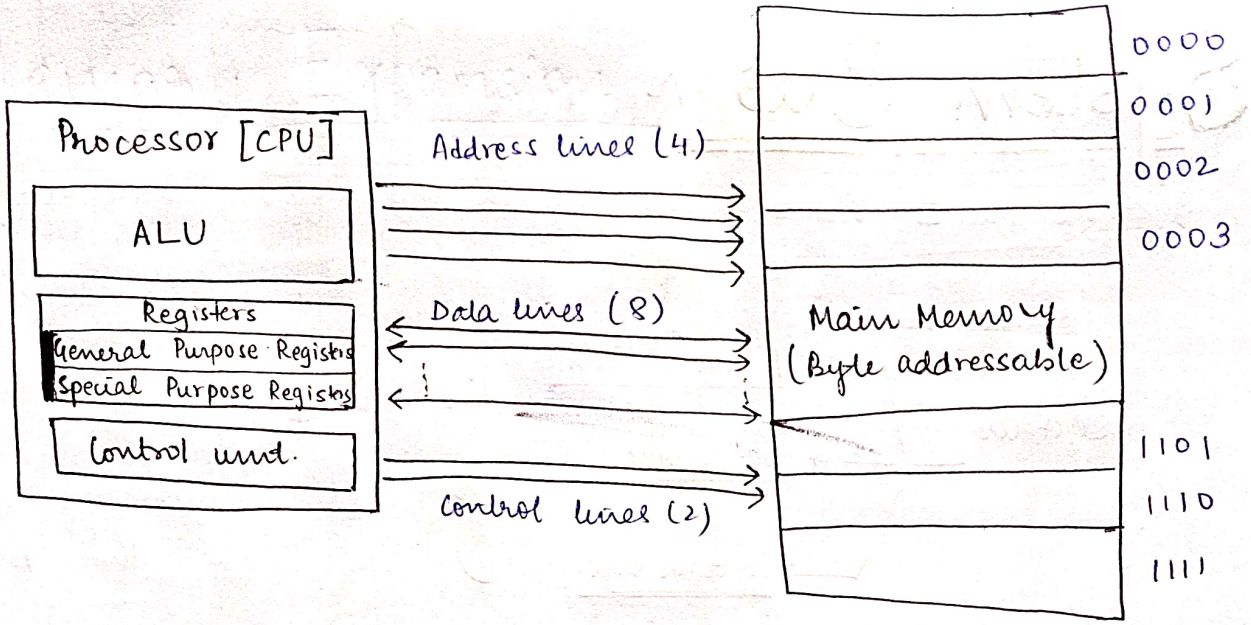
Carries control signals to/from CPU.
individual control lines are unidirectional but as a whole, control signals can be sent in both directions.
∴ partially bidirectional



MAR register size = # address lines

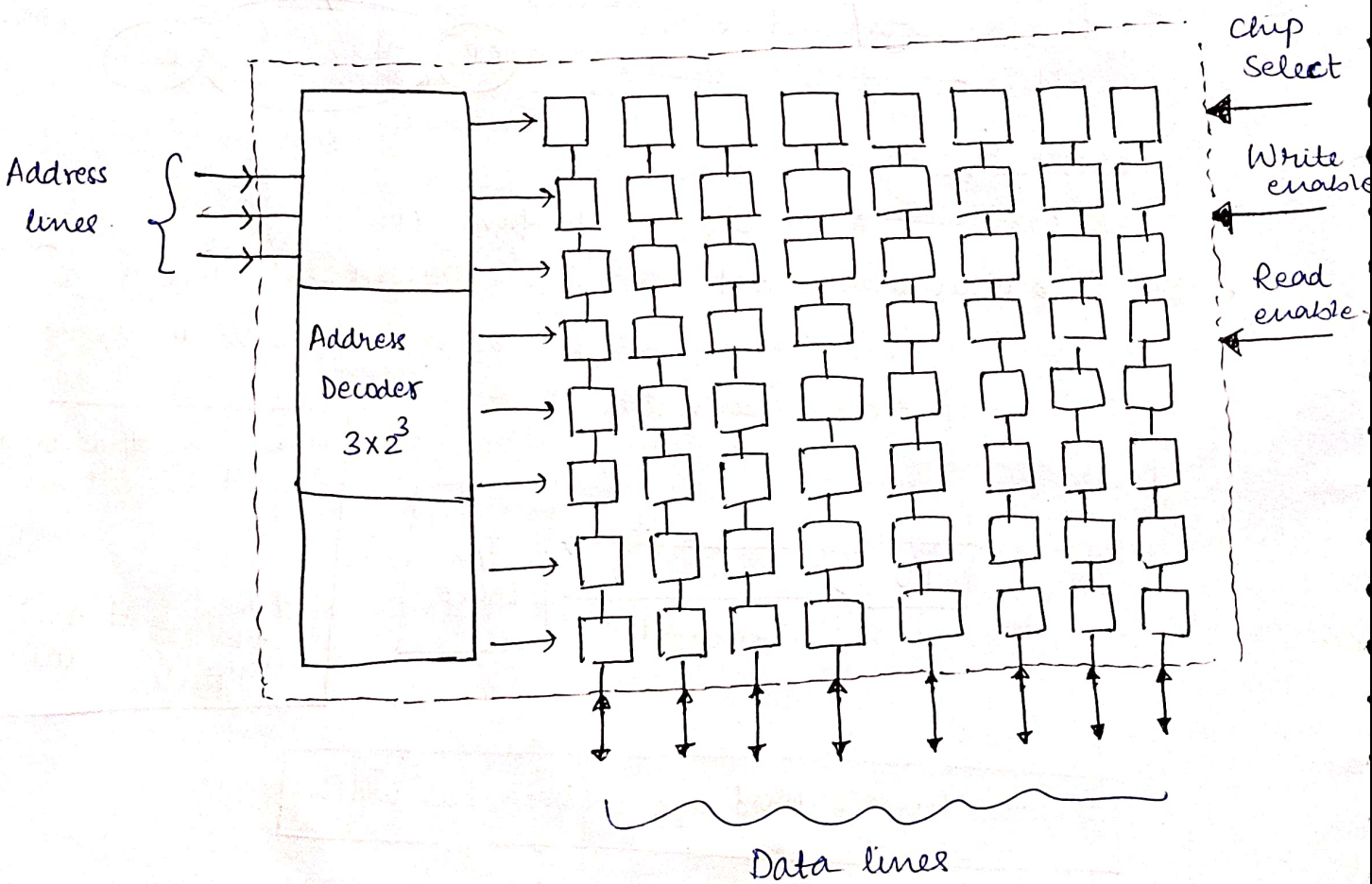
MBR register size = # data lines

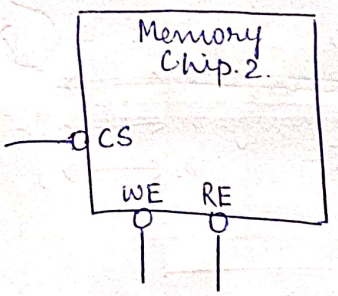
Data lines = word size = Data Bus width.
= MBR register size.



⊙ Three additional lines — chip select, read enable, write enable are used to control the transaction.

control lines connected to memory chip.





* Active low inputs.

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 CU that this never happens. (write is given preference if it happens)

↳ 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 i/ps are written to the addressed location.

↳ OE - Output Enable / Read Enable RE

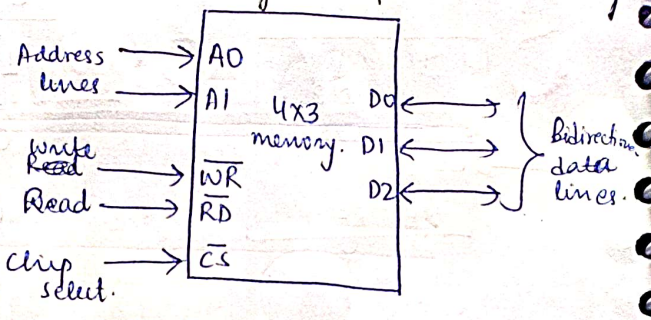
If the chip is selected and WE line is not asserted. and OE is asserted, data at the addressed location is driven on the data lines.

MBR ← <MAR> — Read from memory
<MAR> ← MBR — Write into memory.

<MAR> → memory location whose address is given by MAR. The MAR is said to point to the memory location.

Memory Expansion

Block diagram of 4x3 memory

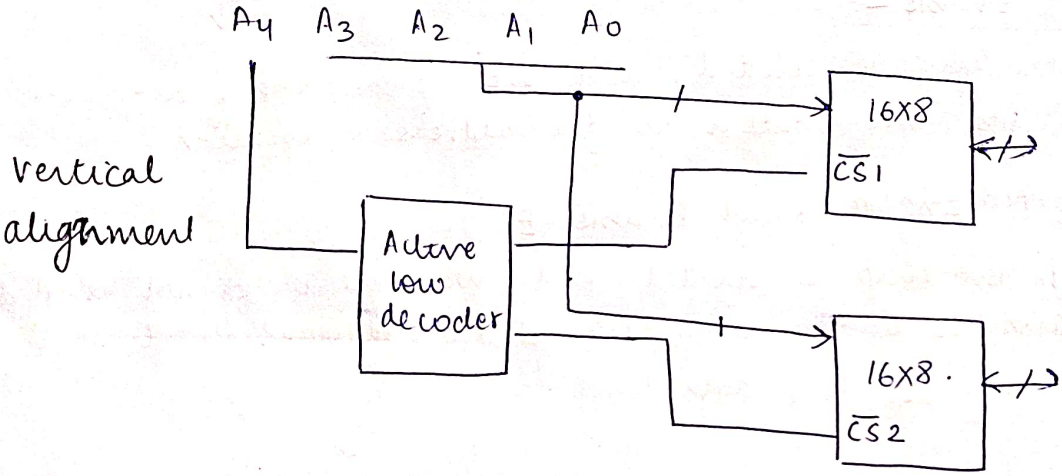


Designing circuit for a given memory capacity using smaller memory chips.

Organize 16x8 RAM chips to give 32x8 memory capacity

5 address lines $A_4\ A_3\ A_2\ A_1\ A_0$

If we use A_4 to decide which chip to select, then, it is called high order memory interleaving



If we use A_0 to decide which chip to select, it is called low order memory interleaving.

↳ it has its own advantages
word size = 2 bytes
1 byte from chip 1
1 byte from chip 2
can be read simultaneously

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 ~~verti~~ horizontal alignment. same address is sent as i/p into the address lines of each chip.

IMPORTANT

★ If ~~chip~~ 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

IMPORTANT

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

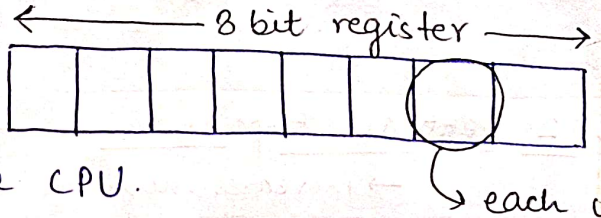
② Chip Select in Vertical Expansion.

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.

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

- Program counter register is used to store the address of the next instruction to be fetched ~~fo~~ for execution.
- connected to the internal address bus.
- 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).

Program Counter	≡	Instruction Address Register	≡	Instruction Pointer Register.

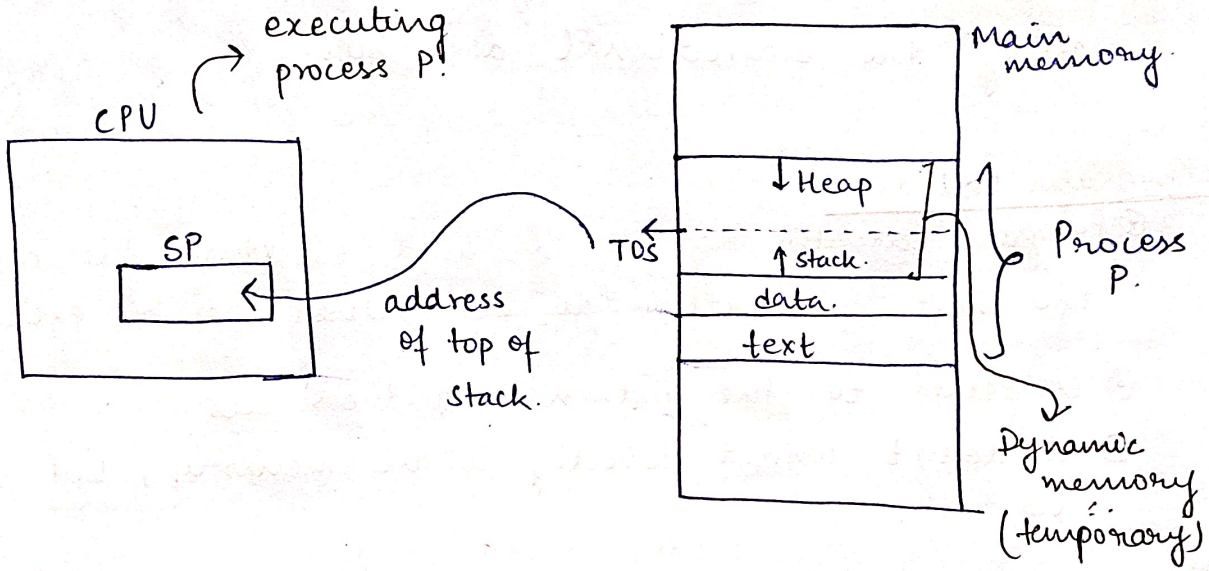
Instruction Register

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.

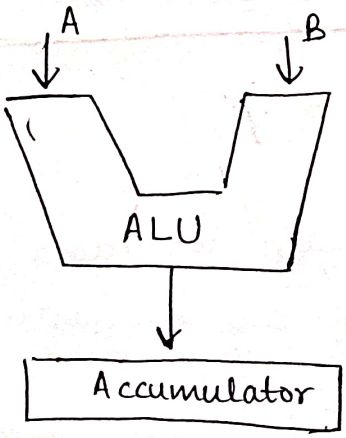
Stack Pointer (SP)

- stores address to top of stack of process.
- 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.

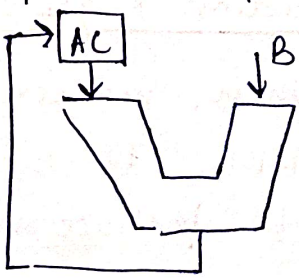


Accumulator (AC)

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



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

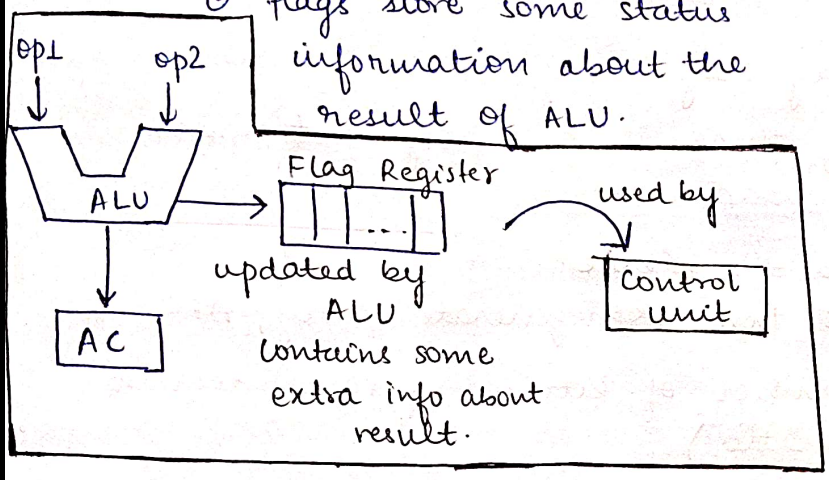


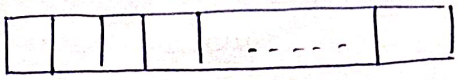
Flag Registers

Status Register ≡ Flag Register ≡ Program Status Word ≡ Program Status Register ≡ Status Word.

① associated to ALU

② flags store some status information about the result of ALU.





every bit is a flag

An important function of the ALU is to set up bits 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..

④ Six status flags monitor the outcome of arithemetic, logical, and related ope

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

Ex→ AL = 0x80 BL = 0x80.

AL+BL = 1000 0000 + 1000 0000 = 1 0000 000 0

Accumulator contains 0000 0000.

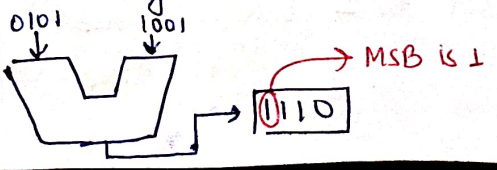
∴ ZF is set to 1.

2. Negative Flag (Sign flag)

SF = { 1 if MSB of result is 1; 0 if MSB is 0.

Ex→ unsigned int A = 0101
unsigned int B = 1001

A+B = 0101 + 1001 = 1110



∴ SF is set to 1.

3. Carry Flag

- CF is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
- Set to 1 when there is a carry from the adder.

4. Auxiliary Carry Flag

- 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).

- 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

- PF = 1 if the low byte of a result has an even number of one bits.
- PF = 0 if the low byte has odd parity.
  - only least significant 8 bits are taken into account.

Ex → FFFE

low byte → FE = 11111110

7 ones (odd)

∴ PF = 0.

6. Overflow Flag

overflow occurs in 2 cases

- → neg + neg gives positive
- → +ve + +ve gives negative.

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

ALU does not know why it is setting the flag. Rule ¥ तो follow करना ¥ बात खत्म!!

Example ①

AX = 0xFFFF BX = 0xFFFF

AX + BX =

```
  1 1111 1111 1111 1111
  1 1111 1111 1111 1111
-------------------------
1 1111 1111 1111 1110
```

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

Example ②

AX = 0x80 BX = 0x80

AX + BX =

```
  (1) 1000 0000
      1000 0000
-----------------
    0000 0000
```

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

- ISA defines the permissible instructions (allowed instructions)
- ISA is a view of computer system from Assembly Language Programmer's point of view or compiler's point of view.
- ISA is a boundary / interface b/w software and hardware.

Instruction set design issues —

ISA of a processor can be described using 5 categories.

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

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

0-address instruction.

ADD, SUB, MUL } operands are taken from stack (stack section of process in M.M.)

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

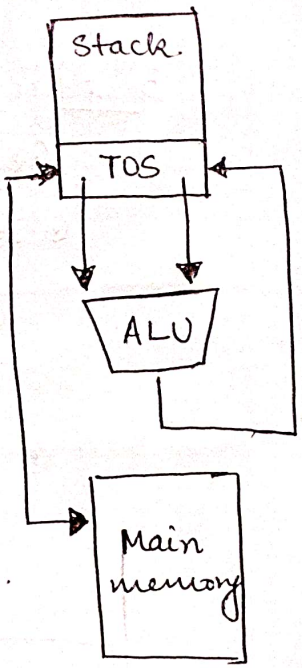
A lot of 3 memory cycles needed — time consuming.

Stack Based ISA

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

→ Data Transfer instructions
1 address instruction.
add/ delete elements from stack.

PUSH loc1 — TOS ← mem[loc1]
POP loc2 — mem[loc2] ← TOS.



C ← A+B

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

2. Accumulator ISA

① In arithematic instructions, one operand is implicit (from accumulator).

② arithematic instructions are 1 address instructions.
ADD loc1 — Acc ← Acc + mem[loc1].

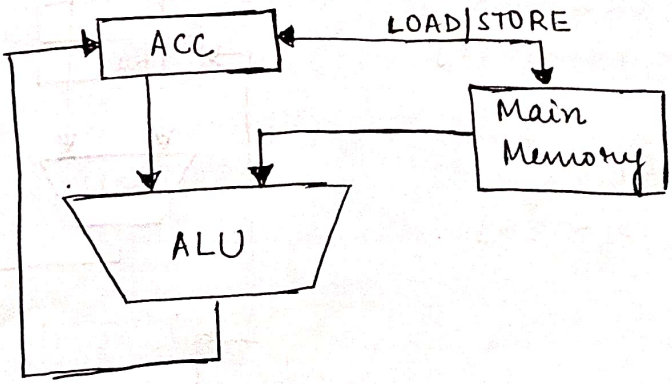
③ Data transfer instructions — LOAD and STORE are also 1 address instructions.

C ← A+B.

```
LOAD A   ← 1
ADD B    ← 1
STORE C. ← 1
```

LOAD loc1 — Acc ← mem[loc1]
STORE loc2 — mem[loc2] ← Acc
(loc2 → memory address.)

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

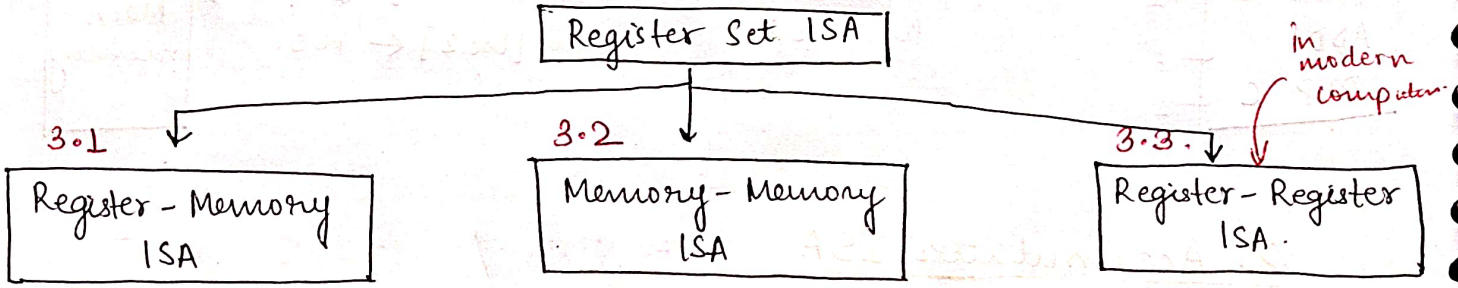


3. Register Set ISA

General Purpose Register ((GPR) ISA).

① Its 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.



3.1 Register - Memory ISA

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

Arithematic instructions are 2-address instructions.

ADD R, loc1
[R] ← [R] + mem[loc1].

Data transfer instructions are 2-address instructions

LOAD R, loc1
[R] ← mem[loc1].

STORE loc1, R
mem[loc1] ← [R].

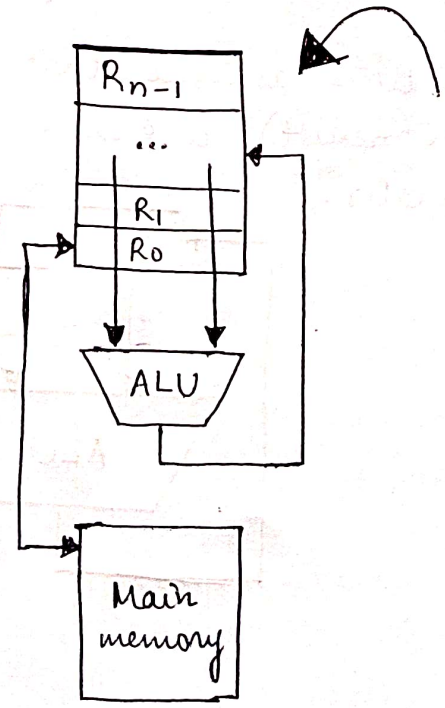
3.2 Memory - Memory ISA

all operands are from memory.

Most CISCs use this architecture.

ADD B, C, A
mem[A] = mem[B] + mem[C].

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



3.3 Register - Register ISA

All operands in arithematic operation come from registers.

ADD R1, R2, R3.
R1 ← R2 + R3

Data transfer instructions are 2-address instructions

LOAD R1, loc1;
[R1] ← mem[loc1]

STORE loc1, R1
mem[loc1] ← [R1].

* Arithematic ins do not access memory!!
* only load and store can access the main memory.

MOST RISCs adopt this architecture.

NOTE

⊙ 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

ISA

- RISC — ~~Regi~~ Reduced Instruction set Computer
- CISC — Complex Instruction Set Computer

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

Time per instructions must be less → Register - Register ISA is used.

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
} only 2 ins in assembly level code.

6 memory accesses

- 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.

- focuses on minimizing execution time.

- 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 $\quad R_1 \leftarrow R_2 + R_3$ → 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 $\quad mem[A] \leftarrow mem[B] + mem[C]$.

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

Simple Instructions	Complex Instructions
① Register - Register instructions.	① Memory - Memory instructions.
② can be executed in a single clock cycle.	② needs to access memory for operands and hence takes many clock cycles.
③ used in RISC	③ used in CISC.
④ Hardware is not so complicated.	④ Complicated hardware (coz it is h/w heavy).

① For CISC architectures, compilation is easy (since assembly language code is small, compiler has to do less work).

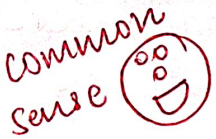
For RISC architectures, compilation is hard / heavy (large no. of instructions in assembly language are ~~gen~~ to be generated by the compiler).

- CISC CPU — Hardware does more work.
  RISC CPU — Software does more work. (compiles).

The Performance ~~Execu~~ Equation.

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

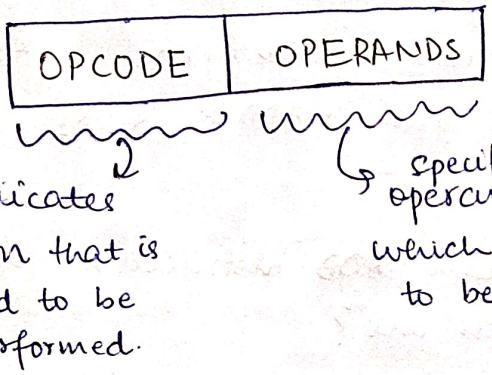
CISC minimizes # instructions in program.
RISC minimizes # cycles per instructions.



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

Instruction Format...

Each instruction has 2 parts – opcode and operands.

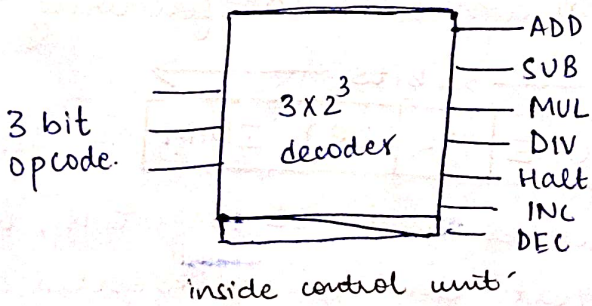


Instruction set –
The complete collection of instructions that are understood by the CPU.

Elements of an instruction →
- Operation code
  ADD, SUB, MUL ...
- Source operand reference
- Result memory operand reference.

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



⊙ 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 arithematic instructions

Accumulator based CPU :- 1 address arithematic instructions

2 address instructions : ADD R_1, R_2 $R_1 \leftarrow [R_1] + [R_2]$.

3 address instructions : ADD R_1, R_2, R_3 $R_1 \leftarrow [R_2] + [R_3]$.

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 R_1, R_2, R_3

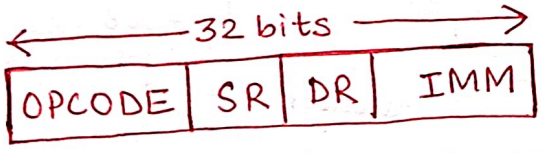
2 address ins :- LOAD R_1, loc $R_1 \leftarrow [loc]$

1 address ins :- INC R_1

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 → 5 bits

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

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

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

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.

No. of bits needed for opcode = $\lceil \log_2 150 \rceil$ = 8 bits.

No. of bits left for address part of ins = 24 - 8 = 16 bits.

Maximum allowable memory size = $2^{16} * 1B$ By default, memory is byte addressable

Largest unsigned binary number that can be accomodated in one word of memory = $2^{24} - 1$.

Expanding opcode (& Instruction Length)

Instructions → Fixed length instructions
Instructions → Variable length instructions.

Variable length Instructions	Fixed length instructions
⊙ Decoding is complicated	⊙ Decoding of instructions is easier.
⊙ wastage of memory/space in main memory due to alignment.	⊙ more wastage of space. (space is wasted in register in the form of unused bits).
⊙ Pipelining is difficult	⊙ Pipelining is easy.

① Fixed length instructions ⇒ can support / maximize various types of instruction formats without wasting much ~~spce~~ 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 ~~op~~ ~~opcode~~ 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.
Encodings used = 4×2^9

1 address instructions — opcode length = $12 - 3 = 9$ bits
Encodings used = 255×2^3

0 address instructions — opcode length = 12
Encodings used = 16×2^0.

Total encodings $\leq 2^{12}$

$\Rightarrow 4 \times 2^9 + 255 \times 2^3 + 16 \times 1 \leq 2^{12}$

$\Rightarrow 4 \times 512 + 255 \times 8 + 16 \leq 2^{12}$ $\Rightarrow 2048 + 2040 + 16 \leq 4096$

$\Rightarrow 4104 \leq 4096$ false.

$\therefore$ 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 = $8 - 6 = 2$

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

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

Encodings used = $2 \times 2^4 = 2 \times 16 = 32$

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

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

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

$\Rightarrow 256 \leq 256$ true.

$\therefore$ 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.
$\therefore$ 4 combinations possible — 2 are used (say 1100, 1101)

Type 3 ins — 5 length opcode —
11100 11101 11110 11111.

Instruction Types

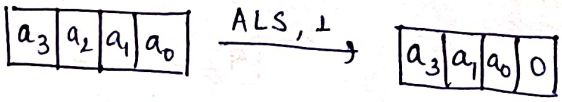
Increment, Decrement instructions do not affect the carry flag.
ADD R, 1 affect CF.

① direct memory to memory move operations are not possible.

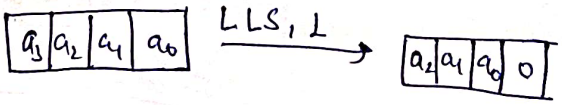
Computer instructions

- **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 –
    - ↳ Arithematic
    - ↳ Logical & bit manipulation
    - ↳ Shift ins.
- **Program Control instructions**
  - BRANCH & JUMP instructions.
  - used to transfer the control from one part of the program to another part.
  - (in detail next page)

Arithematic 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 arithematic shift = Division by 2 of signed numbers in 2's complement form.

Right logical shift = Division by 2 with unsigned numbers.

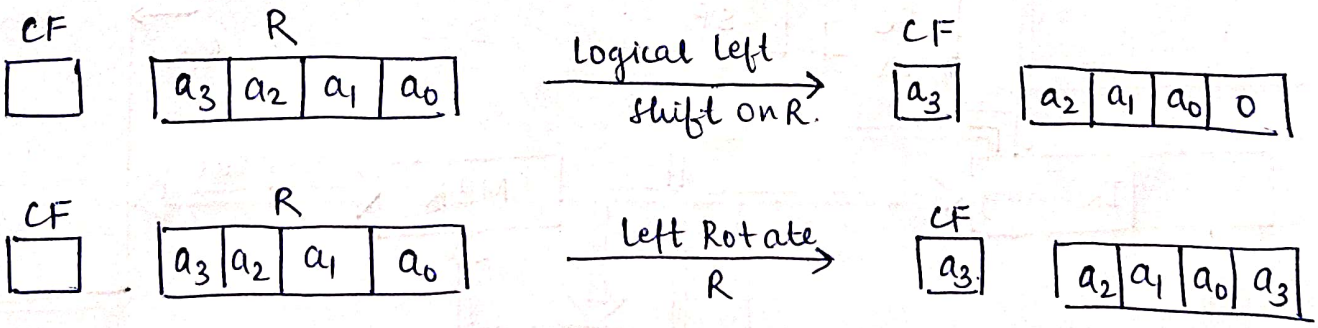
* With signed numbers in 2's complement notation, both ~~propagated~~ logical left shift – ×2 unsigned nos. ALS and LLS correspond to ×2 (when no overflow).

Important summary
(* Try out with examples)

	Multiplication by 2 ($\times 2$)	Division by 2 ($\div 2$)
Signed no. in 2's complement.	Both arithematic left shift and logical left shift.	Arithematic 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! to 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.

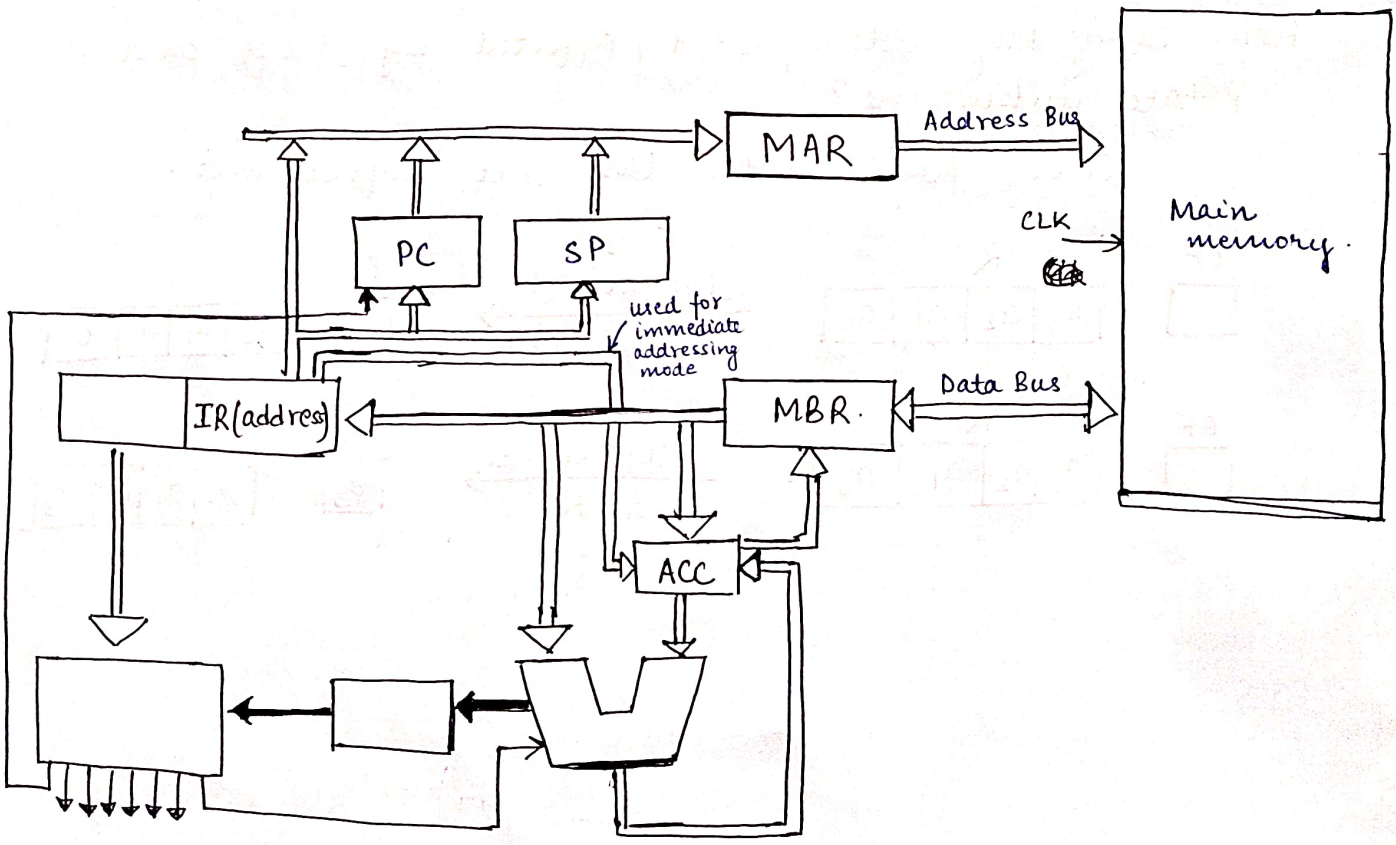
JNZ M — jump to M if ZF is not set.

Instruction Execution Cycle

↳ FETCH the next instruction from memory into the instruction register.

↳ DECODE the instruction (operations & operands).

↳ EXECUTE the instructions.



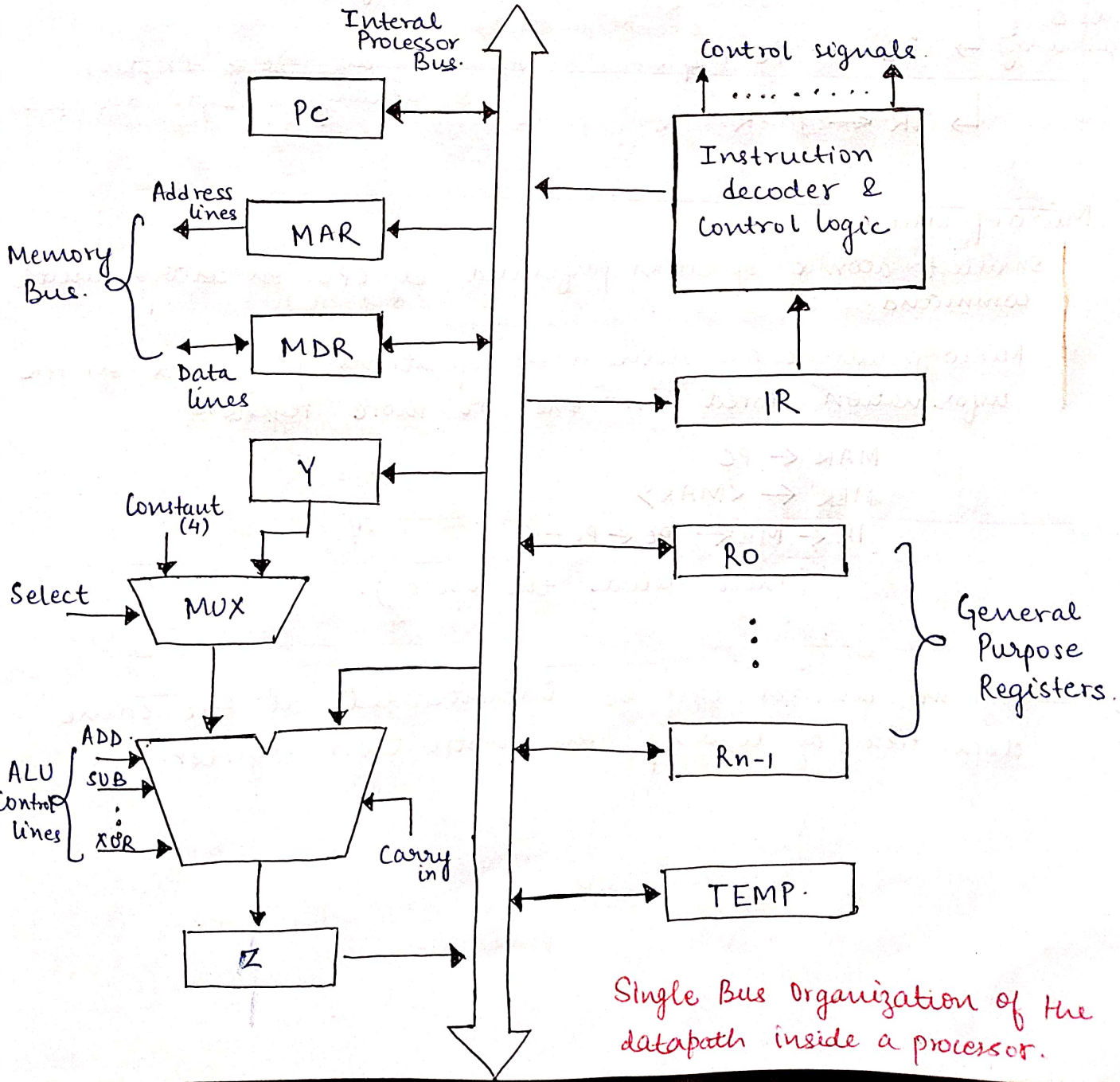
⊙ In order to communicate with memory, a processor needs three types of connections — data, address and control.

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.



Single Bus Organization of the datapath inside a processor.

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 –

↳ MAR ← [PC] Register Transfer.

PC_{out} , MAR_{in}

These 2 steps can be done simultaneously.

↳ MBR ← <MAR> Content of memory at address held by MAR.

↳ PC ← PC + 4 (does not need ALU)

Assume ins size = 4 bytes & memory is byte addressable

↳ IR ← 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.

MAR ← PC
MBR ← <MAR>
IR ← MBR ; PC ← 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 ins 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 ← <x>

① MAR ← IR(address)
② MBR ← <MAR>
③ ACC ← MBR. → 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 ← IR(address)
② MBR ← ACC
③ <MAR> ← MBR ; → fetch.

③ ADD instruction

ADD x → Add the content of memory location x to the accumulator and store the result in accumulator.

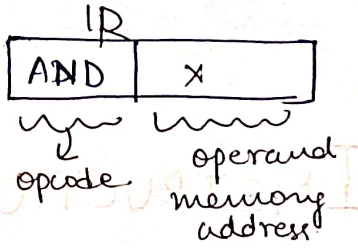
ACC ← ACC + <x>.

① MAR ← IR (address)
② MBR ← <MAR>
③ ACC ← ACC + MBR ; → t_{fetch}.

④ AND instruction

AND x → ACC ← ACC AND <x>

① MAR ← IR (address)
② MBR ← <MAR>
③ ACC ← ACC ∧ MBR.



⑤ JMP x.

unconditional jump to x.

① PC ← IR(address); → t_{fetch}.

⑥ BZ x

conditional jump

BZ x
Branch if zero flag is set
memory address.

① $\overline{Z}$ → t_{fetch}
② PC ← IR(address)

⑦ NOT instruction

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

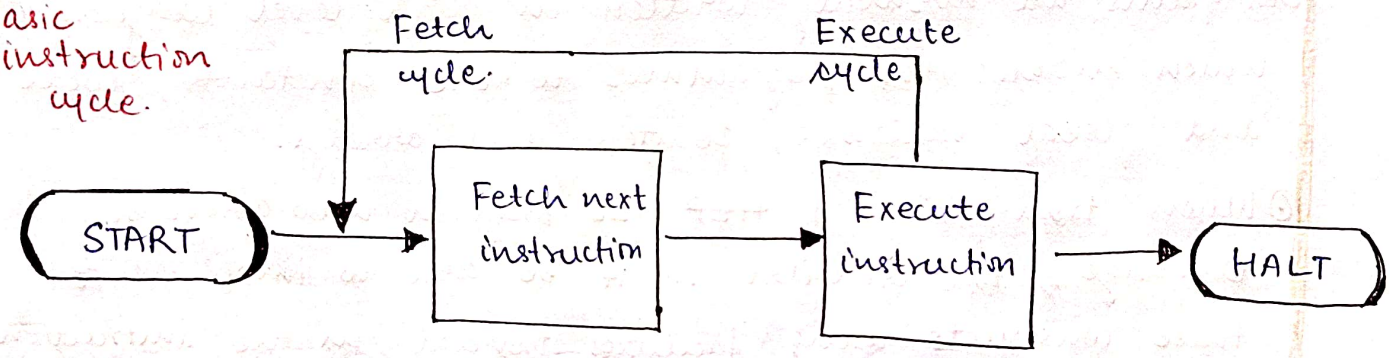
① $ACC \leftarrow \overline{ACC}$

⑧ SHR instruction

Shift Right instruction.

① $ACC \leftarrow ShiftRight(ACC)$ // 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 –

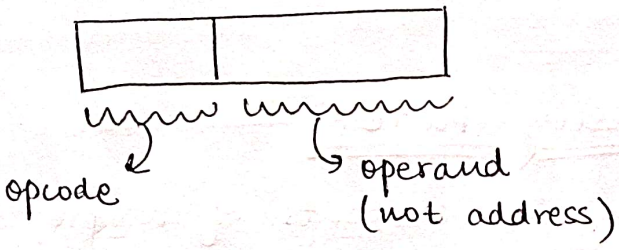
- Programs are normally written in high level languages, which enables the programmer to use constants, local and global variables, pointers and arrays.
- When translating a ~~HLP~~ HLL program into assembly language, the compiler must be able to implement these constructs using facilities provided in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

1. Immediate addressing mode

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



int A = 5 → LOAD R1, #5;

→ 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

- opcode tells about the location of operand.

< Stack based ISA CPU >

ins: ADD → TOP ← [TOP] + [TOP]

< Accumulator CPU >

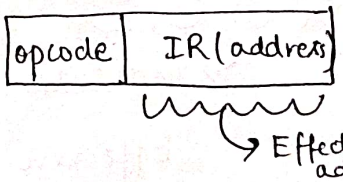
ADD LOCA → ACC ← [ACC] + memory[LOCA].

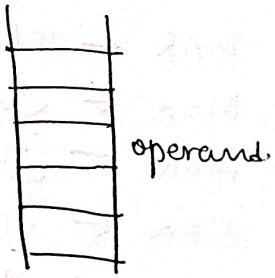
first operand is implied.

3. Direct / Absolute Addressing mode

* Simple. Requires only one memory reference and no special calculations.

- Address field of instruction contains the address of operand.





< Accumulator based CPU >

ADD LOCA;

first operand → implied addressing mode.

2nd operand → direct addressing mode

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

Register Addressing Mode

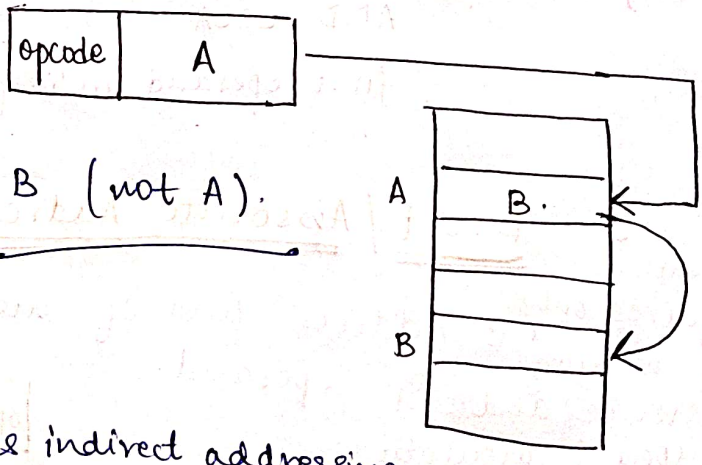
- The operand is the contents of a processor register.
- The address of the register is given in the instruction.

ADD R1, R2, R3 $\quad$ R1 ← R2 + R3

Advantages compared to Direct Addressing mode –
→ small address field in the instruction.
→ no memory reference.

Indirect Addressing Mode

- 2 memory accesses required.
- The address part of instruction contains the address of memory location ~~of~~ where the effective address of an operand is present.
- used in pointers.



Effective address of operand = B (not A).

μ instructions.

LDA x $\quad$ (m) and @M

ACC ← mem[mem[x]]

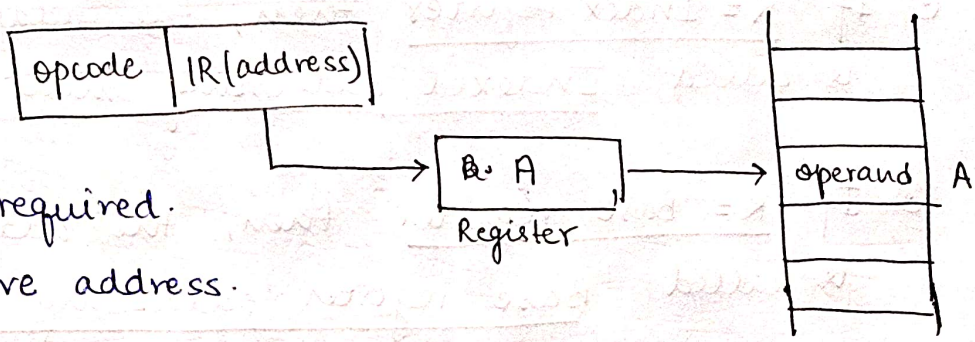
↳ indicates indirect addressing mode

MAR ← IR(address)
MBR ← <MAR>
MAR ← [MBR]
MBR ← <MAR>
ACC ← MBR.

Effective address = [M].

Register indirect addressing mode

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



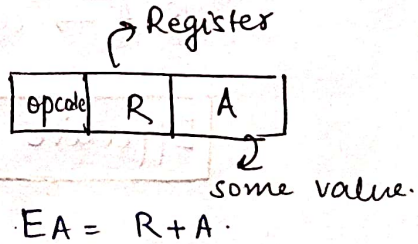
⊙ 1 memory access is required.

A → effective address.

Advantages w.r.t. Indirect addressing mode
- ↳ one less memory reference is required.
- ↳ smaller address field.

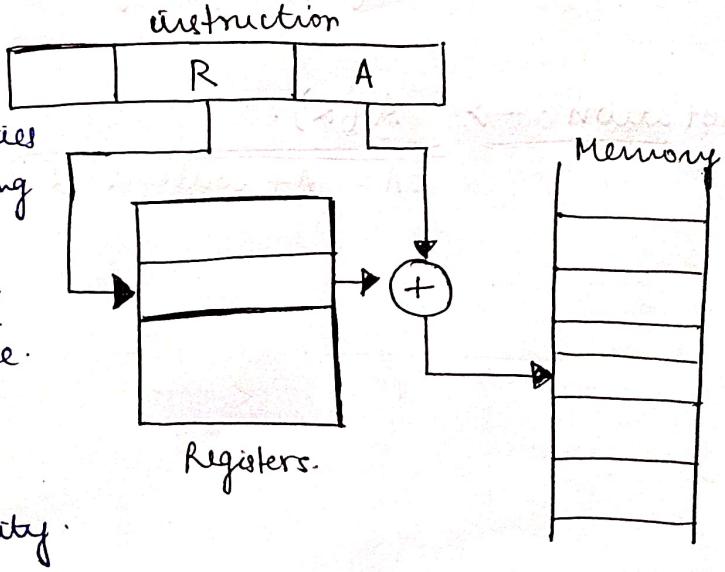
Displacement Based Addressing Modes

① Relative Addressing Mode.
② Indexed Addressing Mode.
③ Base Register Addressing Mode.



EA = R + A.

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



⊙ combines capabilities of direct addressing mode and register indirect addressing mode.

Adv → flexibility
Disadv → complexity.

⊙ If R = Program counter then, the addressing mode is called Relative addressing mode (PC relative).

⊙ If R = Index register then, the addressing mode is called Indexed addressing mode.

⊙ If R = Base register then, the addressing mode is called Base register addressing mode.

Indexed Addressing mode.

Arrays are implemented using indexed addressing mode.

⊙ Index register holds index value → not the actual index values as in HLL it is HLL * size of datatype
↳ it keeps on changing.

⊙ A → contains base address of array.

Effective address = A + Content in index register.

⊙ 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 → A(R).
EA = A + content of R.

Relative / PC relative addressing mode.

PC: special purpose register.
∴ implied.

EA = A + content of PC

A → signed value in 2's complement.

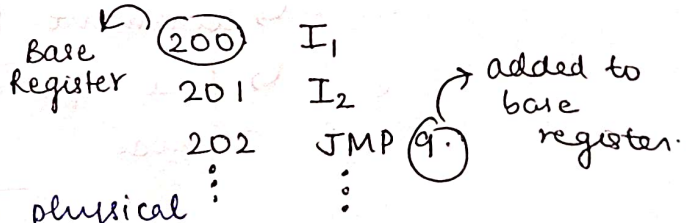
- used in JUMP instructions
- using relative addressing mode, we can write relocatable code.
  ↳ position independent code.
- In relative addressing mode, the (offset is displacement) from the next instruction.

Base Register Addressing mode.

- better and easier way to write relocatable code.

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

- 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

PC Relative addressing mode	Base Register addressing mode
① Offset value can be -ve (signed no. in 2's complement)	① Offset is always +ve. (unsigned)
② Used for Branch instructions	② 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 } No requirement of main memory

MULT : 3 cycles

LOAD / STORE : 100 cycles } Main memory required.

- Cache Memory → not as fast as registers → not as slow as main memory.
- 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.

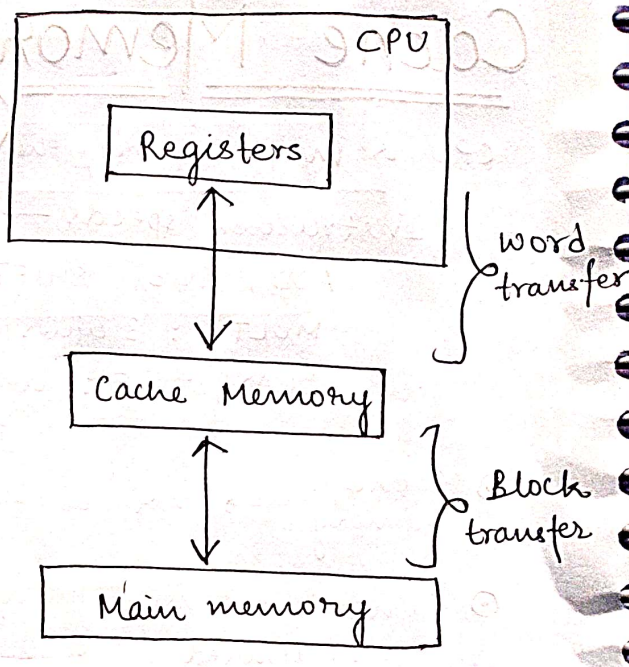
Cache Memory

- Direct Mapped cache
- Set Associative
- Fully Associative

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

Direct Mapped Cache

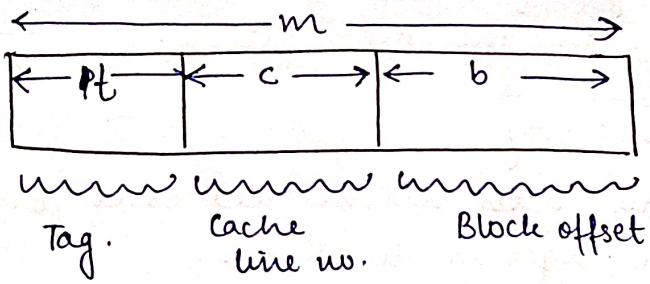
- simplest approach.
- Each main memory address maps to exactly one cache block.
- Nearby blocks are mapped to different lines
  → spatial locality is maintained.



Mapping function

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

CPU generated main memory address.



Main memory size = 2^m bytes

Block size = 2^b bytes

No. of lines in cache = 2^c

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

$$t = m - (b+c).$$

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.

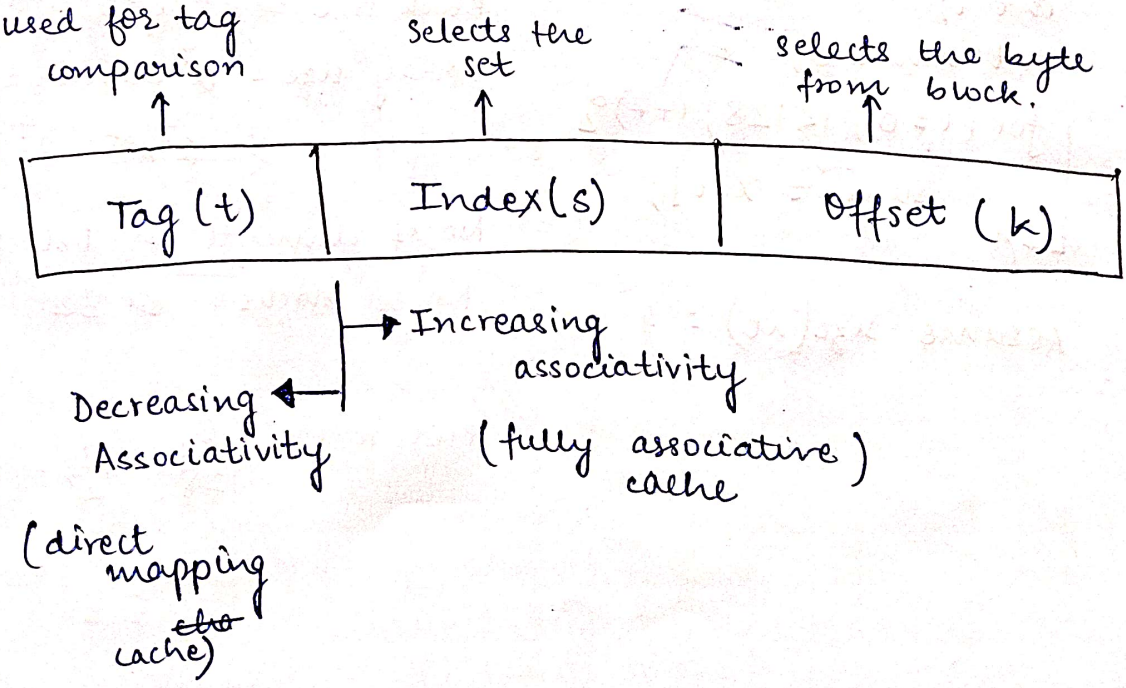
Line number – Represents to which line number of cache memory the address belongs.

Set Associative Mapping

- A block can be placed anywhere within a set of locations in the cache.
- Cache blocks are grouped together into sets.

  2-way set ≡ set size is 2 ≡ # lines in a set = 2.

Cache set No. = Main memory Block No. % Number of sets.



Fully Associative Cache Mapping –

⊙ Each memory block can map anywhere in the cache (fully associative)
- ↳ most efficient use of space
- ↳ least efficient to check.

⊙ No index bits.
(just tag and block offset).

Tag	Block offset

↳ Block No.

Check a fully associative cache –
1) 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 too 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
= $2^7 \times 2^2 = 2^9$ Bytes
No. of elements in 1 block = $2^2 = 4$
No. of blocks = 32.

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

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

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

1 line can store 2 integer variables.

x[0]	M	
x[1]	H	
x[2]	M	
x[3]	H	
x[4]	M	
x[5]	H	
x[6]	M	
x[7]	H	

×2

0	x[0]	x[1]
1	x[2]	x[3]
2	x[4]	x[5]
3	x[6]	x[7]

Main memory

No. of misses = 4×2 = 8

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

0	x[4]	x[5]
1	x[6]	x[7]

Cache Memory (at the end)

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.

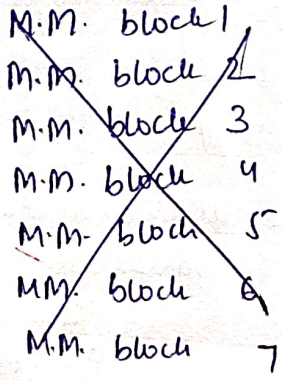
ii) 2-way set associative
8 byte block size
2 sets.

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

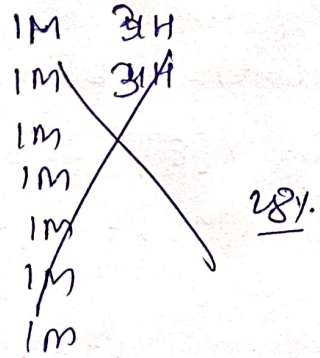
0	a(0,0)	a(0,1)	a(0,2)	a(0,3)	1M 3H
1	a(0,4)	a(1,0)	a(1,1)	a(1,2)	1M 3H
2	a(1,3)	a(1,4)	a(2,0)	a(2,1)	1M 3H
3	a(2,2)	a(2,3)	a(2,4)	a(3,0)	1M 3H
4	a(3,1)	a(3,2)	a(3,3)	a(3,4)	1M 3H
5	a(4,0)	a(4,1)	a(4,2)	a(4,3)	1M 3H

Miss rate = $\frac{1}{25}$ = 28%

(ii)



1 elmt.



8 byte block size ⇒ 2 integers in 1 block.
2 sets.

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

No. of misses = 12+1 = 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 IM } Iteration 1
1 — next 9 elements IM

In 2nd iteration
everything is already there so 0 misses

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

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

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	

∴ No. of misses = $\frac{8}{25}$ = 32%

Ques

128 x 64 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];
    }
}
```

4KB 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].
```

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

No. of blocks needed to store 128 x 64 integers = $\frac{128 \times 64}{8}$ (26, 8)

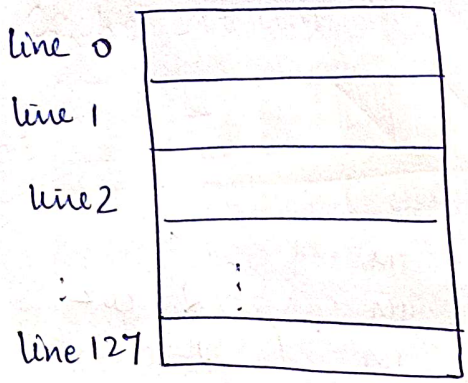
No. of cache lines = $\frac{2^{12}}{2^5} = 2^7$ = 1024

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, ...

∴ No. of misses
= No. of elements
= 64 × 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

A CPU has a 32KB 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];
    }
}
```

P2:

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

No. of cache misses?

blocks occupied in MM. = $\frac{512 \times 512}{16} = 2^{14}$

integers in 1 block = $\frac{2^7}{2^3} = 2^4 = 16$.

cache lines = $\frac{2^{15}}{2^7} = 2^8 = 256$

Main memory Organization:

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

P1: access sequence —
~~0, 1, 2, ...~~
0, 0, 0, ... (16 times)
1, 1, 1, ... 16 times

No. of misses = $\frac{512 \times 512}{16} = \frac{2^{18}}{2^4} = 2^{14}$

P2: access sequence —

0, 32, 64, 96, ... 0, 32 ....

(0,0) (0,1)

by the time (0,1) is accessed
line 0 of cache contains block 2^8.

∴ # misses = # elements = 512 × 512 = 2^{18}

Ratio of no. of cache misses = $\frac{2^{14}}{2^{18}} = \frac{1}{2^4} = \frac{1}{16}$

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

integers in 1 block = 64

(0,0) (0,49) (1,0) ... (1,13) (1,14) ...

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

Access sequence →

0, 0, 0, 0, ... 0, 1, 1 1

64 times 64 times.

No. of lines in cache = 32 No. of blocks = 40

first 8 lines will be replaced. Remaining same

No. of cache misses = 40 + 8 = 48 + 8 = 56

↳ in 1st iteration ↳ in 2nd iteration. ↳ coz starts from 1100H

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 20 ns to bring the first word of block from main memory while each subsequent word takes 5ns. The word size is 64 bits. Avg memory access time?

Block size = 256 bytes Word size = 64 bits = 8 bytes

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

Time taken to transfer 1 block from M.M. to cache = $20 + 31 \times 5 = 20 + 155 = 175$ ns.

Avg memory access time = $0.94 \times 3 + 0.06 \times (3 + 175)$

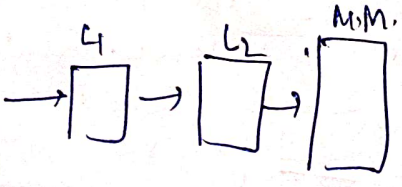
$= 0.94 \times 3 + 0.06 \times 178$

$= 2.82 + 10.68 = \underline{13.50}$ ns.

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 $L_1 = 2x$



AMAT = ~~xxxxxxxx~~

AMAT = $(1-2x) \times 1 + 2x\left[(1-x) \times (1+8) + x \times (1+8+18)\right.$

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

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

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

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

16
16
9 6
16

256
144
400

$\Rightarrow x = \frac{-16 \pm \sqrt{16^2 + 4 \times 36}}{2 \times 36} = \frac{-16 \pm 20}{72} = \frac{20-16}{72} = \frac{4}{72} = \frac{1}{19}$

$= 0.0526.$

In multilevel cache,

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

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 Miss

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 —

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

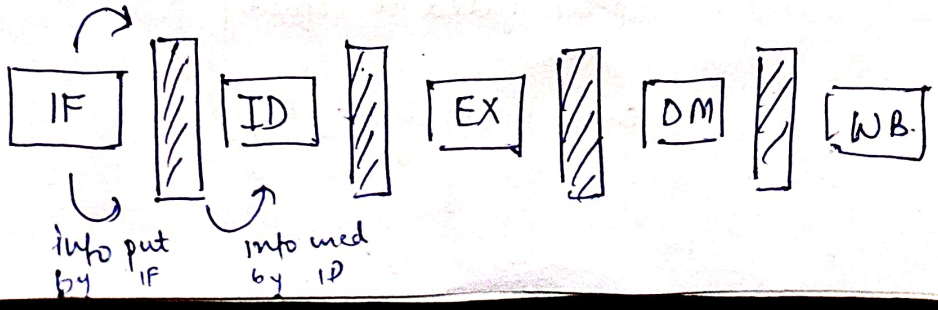
Pipeline system —

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

Inter stage Buffer

→ used to store information.

→ information is ~~stor~~ used 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.

I/O Devices
- ↳ Input Devices — Devices that allow the user to send data or commands to the computer.
- ↳ Output Devices — Devices that display or output data from the computer to the user.

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

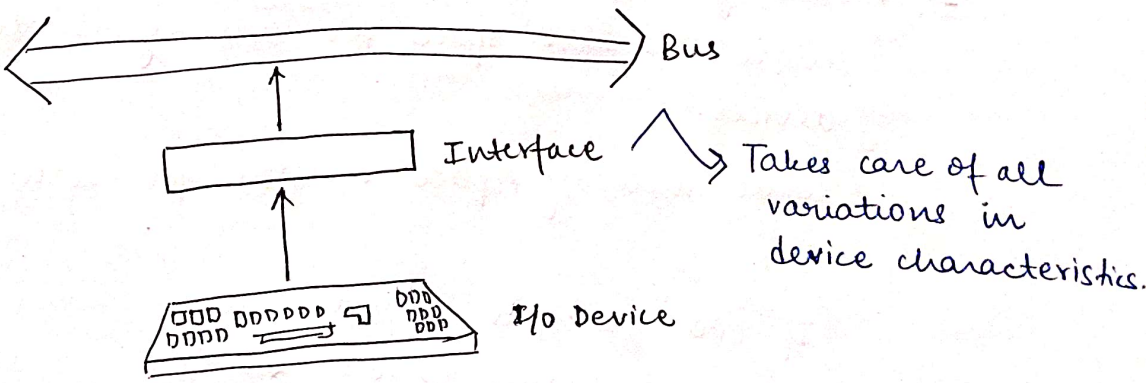
↳ wide variety of peripherals with widely varying speeds (I/O devices are very much slower than CPU).

↳ Data transfer rate is irregular or regular.

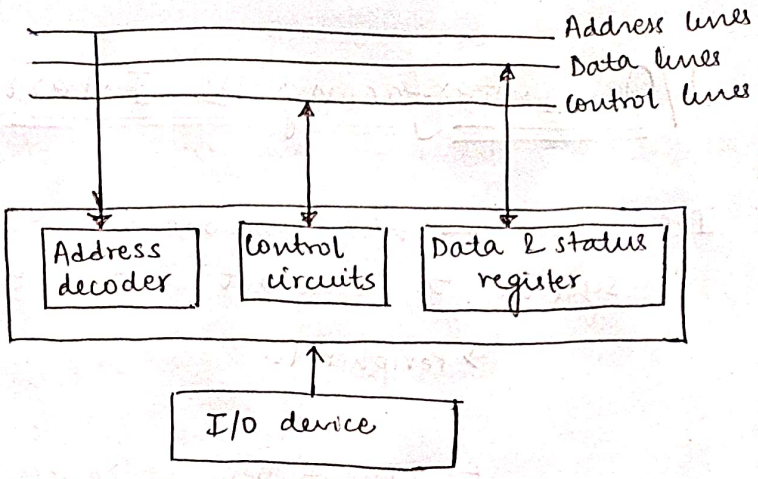
↳ Different I/O devices have different data formats.

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.

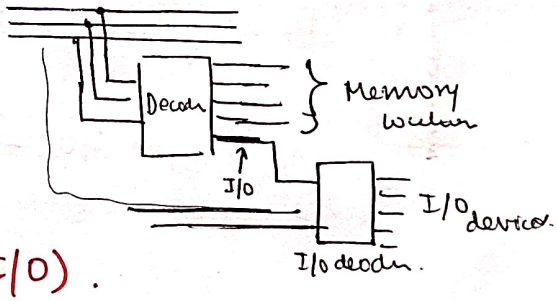
Ex – printer assigned address 0xFF00.
data sent to 0xFF00 will be treated as i/p to printer

Advantages

- 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).

~~Disag~~ Disadvantage.
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.



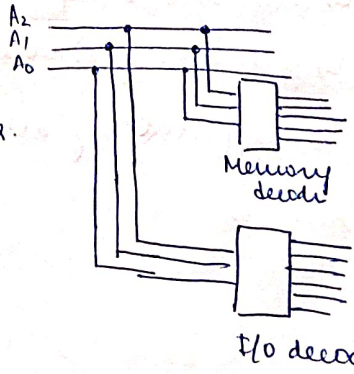
2. I/O Mapped I/O (Isolated I/O).

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 devices.



⑤ CPU generates a control signal $IO/\bar{M}$ along with address

- 0 → access main memory
- 1 → access I/O

enable signal for each decoder

(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

- Memory mapped I/O is supported by every processor. (coz no special treatment by processor)
  while I/O mapped I/O require IO/$\overline{M}$ signal, additional decoder, IN, OUT instructions.

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

- Memory mapped I/O is faster than I/O mapped I/O.
  Memory mapped I/O → use LOAD & STORE → memory instructions ∴ faster.
  I/O Mapped I/O → use IN & OUT → 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

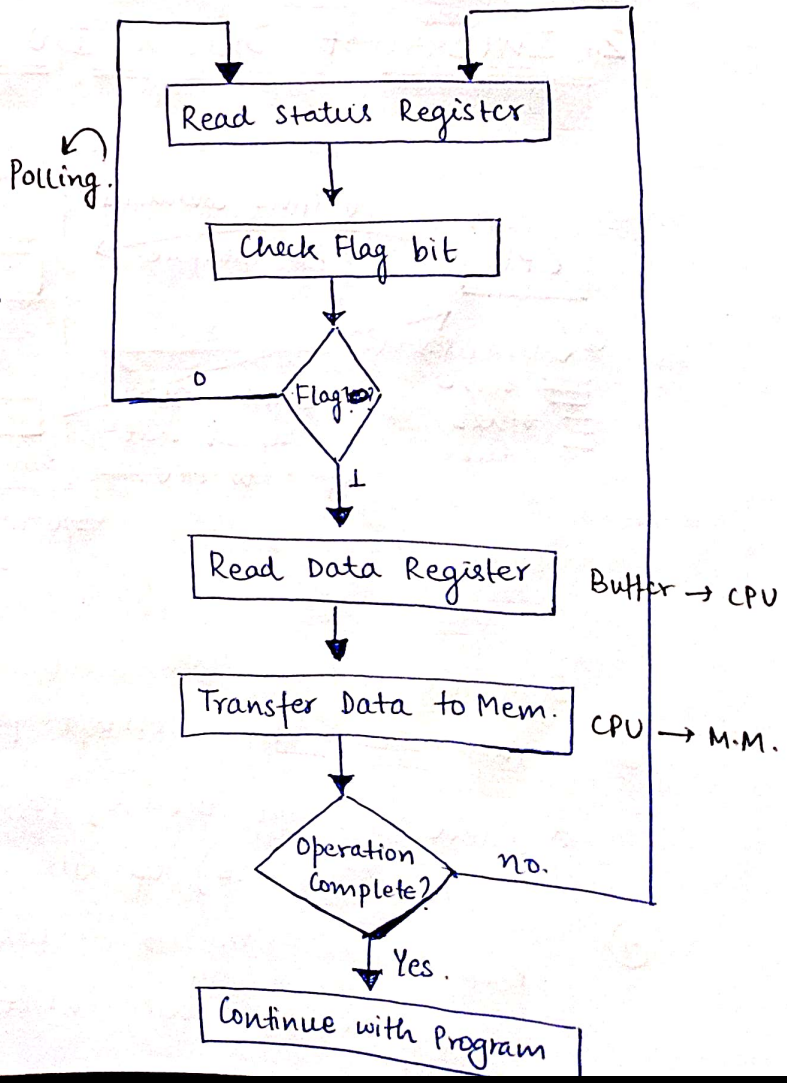
} (CPU is involved). Data transfer is done by the help of CPU.

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)

- ⊙ CPU executes a program that transfers data b/w I/O device and main memory.
- ⊙ CPU does not do any useful work during Polling.
- ⊙ Data transfer is initiated by the program run by the CPU.
- ⊙ Processor is totally in control and does all the work.
- ⊙ I/O module does not inform CPU directly.
- ⊙ I/O module does not interrupt CPU.
- ⊙ CPU checks status bits periodically.



⓪ 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 –

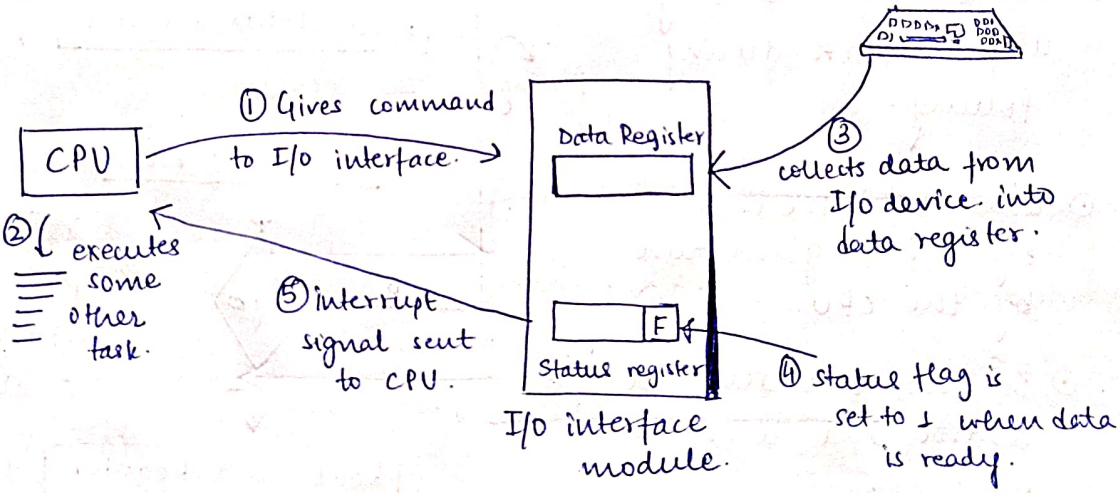
↳ simple – processor is totally in control and does all the work.

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 ⓪ At the end of each instruction cycle, the processor checks for interrupts (on interrupt pin).

⑦ ⓪ 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).

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

⊙ In case of small and frequent data transfers, programmed IO is faster than interrupt driven I/O. } for fast I/O devices.

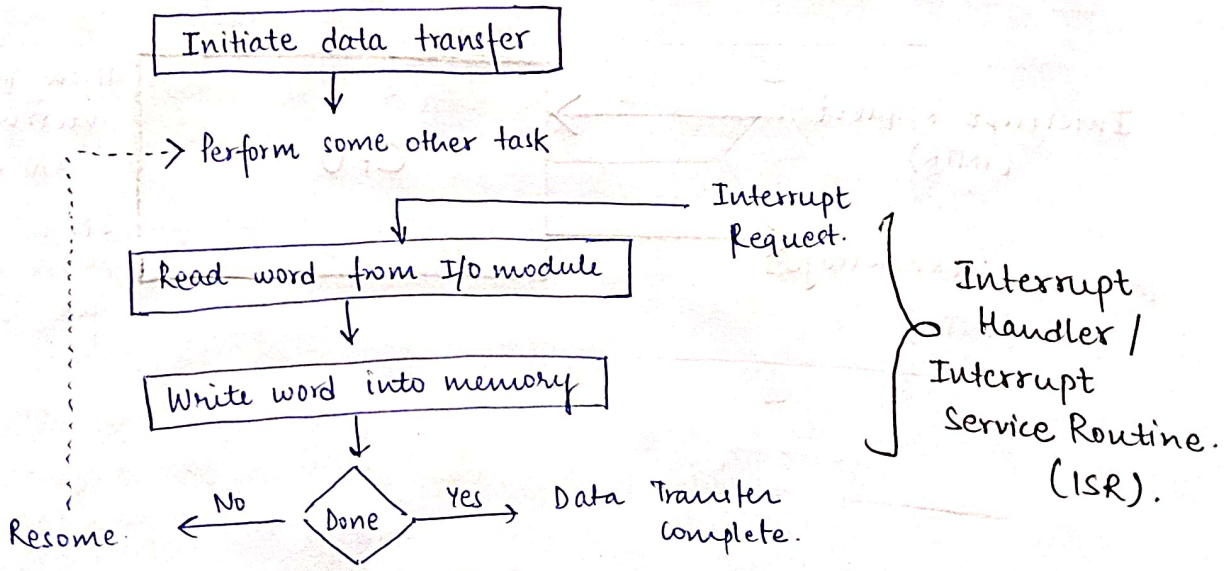
↳ a lot of context switches.

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.

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



Interrupt

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

Interrupt Service Routine –

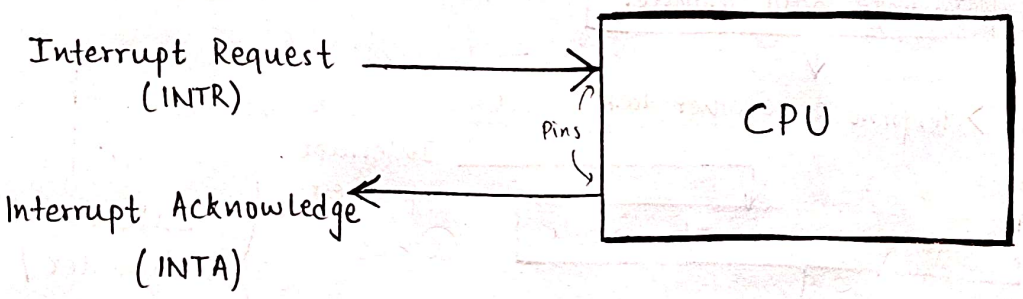
↳ Interrupt handler, Interrupt service subroutine (ISS)

- ISR is the program i.e. instructions to handle a particular sort of interrupt.
- 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

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



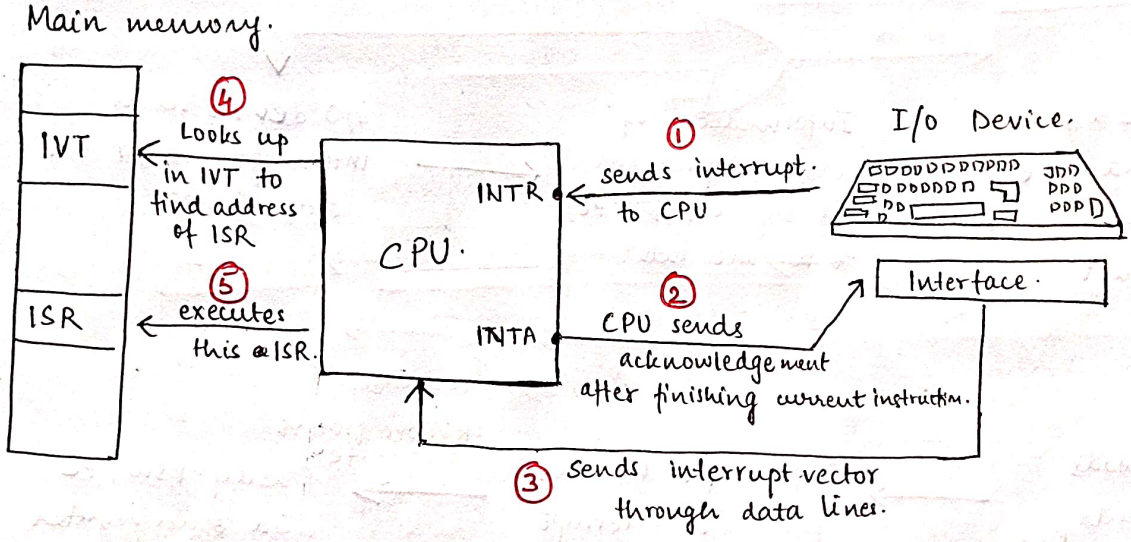
INTR pins receives interrupt request from I/O device.

INTA pin sends acknowledge before ISR is executed to I/O device that (~~after~~)

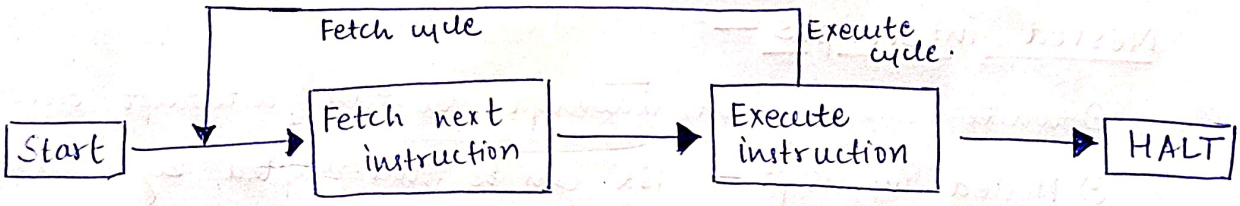
Vectored Interrupts –

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).

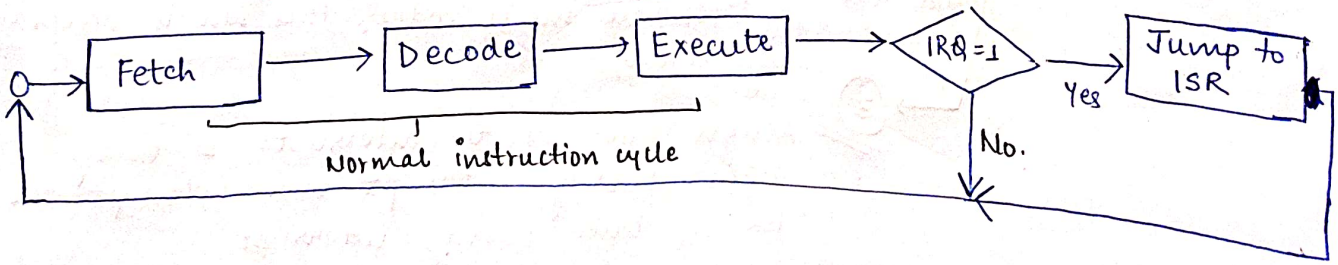


Interrupt cycle –



Basic instruction cycle.

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



Interrupt Processing —

I/O device issues interrupt signal → Processor finishes execution of current instruction before responding → Acknowledgement signal is sent to I/O device (Then I/O device removes interrupt signal)

↓

I/O device sends interrupt vector via data lines → Information of current program is stored (PSW & PC) to resume later → Processor check IVT to find location of ISR for interrupt

↓

Processor loads PC with entry location of ISR. → Interrupt handler performs interrupt processing → Restore process state. → Finally PSW, PC and other register values are restored.

Nested interrupts —

- ⊙ while servicing an interrupt, another interrupt occure.
- ⊙ Nested interrupt — ISR can be interrupted by another interrupt.

Solutions

① Allow any type and level of nesting.
Starvation possible (high priority interrupt may starve).

② Disable any type of nesting of interrupt.
(sequential interrupt processing).
(∵ ISR are small codes, this soln is acceptable).

does not take into account relative priority or time critical needs.

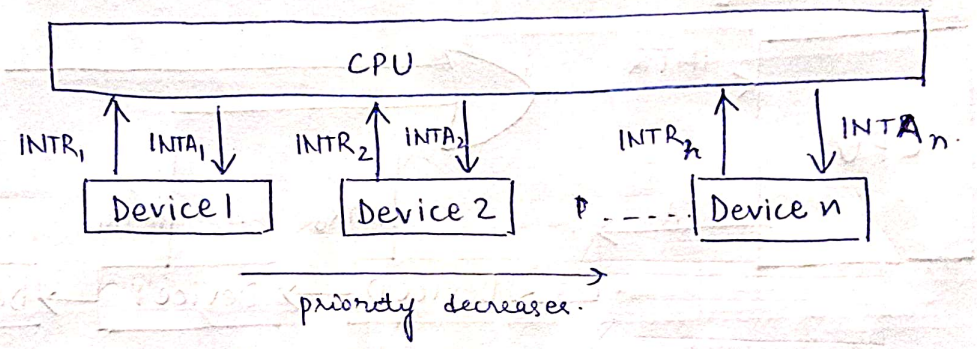
③ Assign priorities to interrupts & only higher priority interrupt can interrupt the ISR of lower priority interrupt.

Multiple interrupts at same time.

© 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 I/O device one by one if it caused the interrupt using system bus.

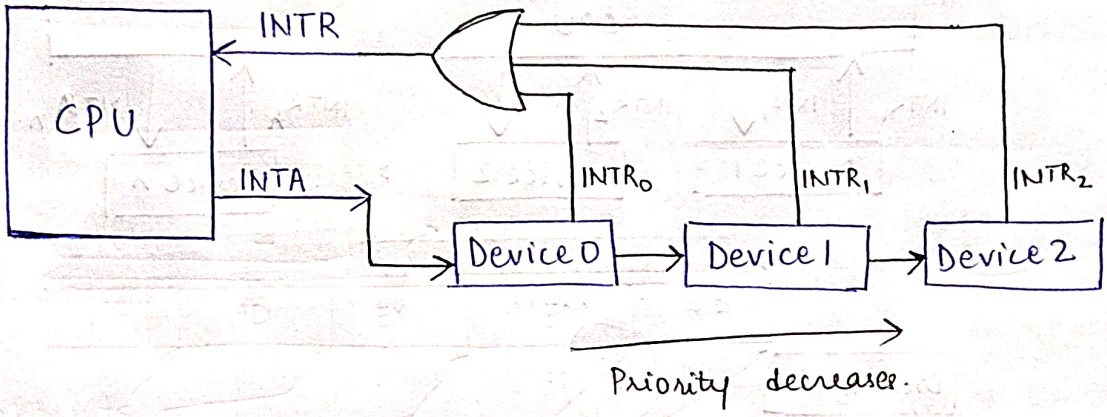
Each I/O device will have one system bus cycle to answer yes.

Drawback

↳ Time consuming.

Solution 3: Daisy Chaining arrangement for assigning priority.

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



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

3. Direct Memory Access (DMA)

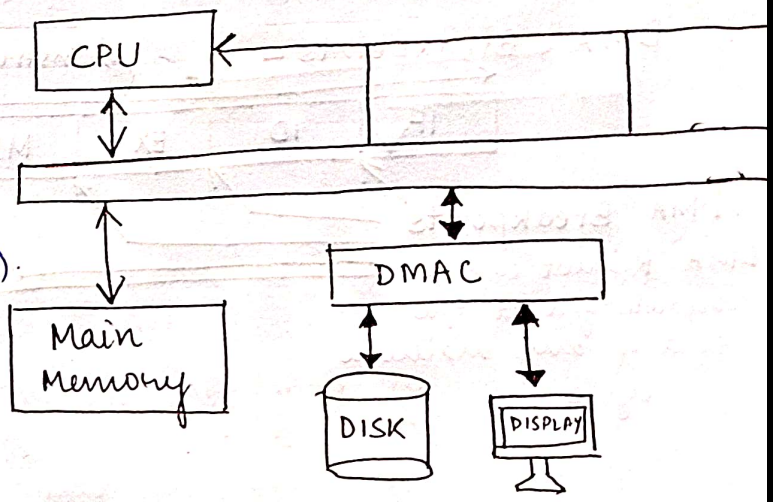
DMA mechanism can be used for data transfer, without involvement of CPU.

⊙ 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.

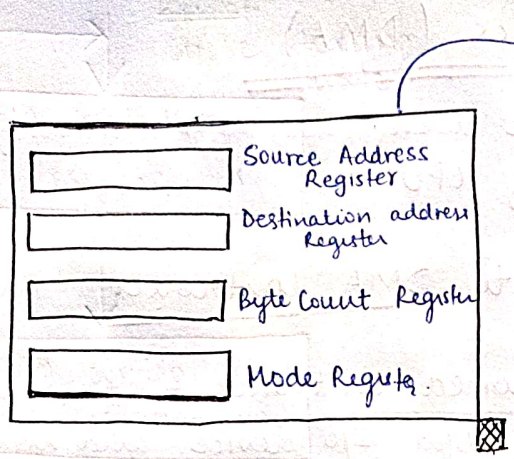
→ as DMA controller transfers data, the value in this register is decremented.

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

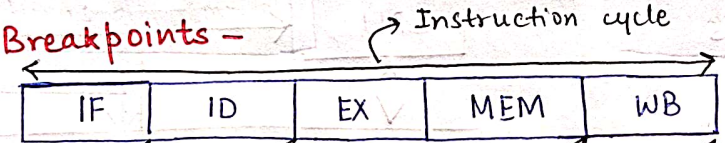
↳ Cycle Stealing mode (Transfers 1 byte at a time while alternating bus control with CPU)



DMA Controller
(mini processor)
only used for data transfer.

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

DMA Breakpoints –

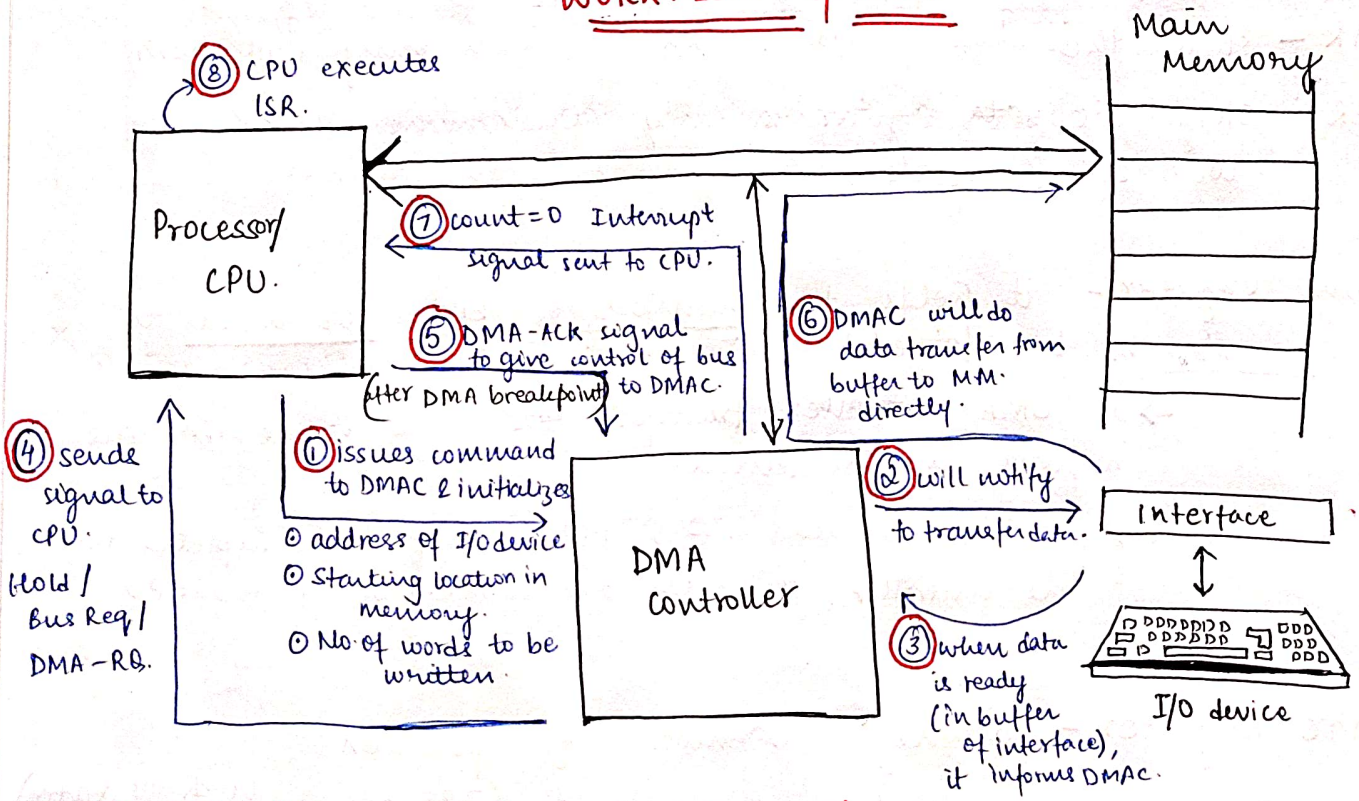


DMA Breakpoints

DMA Request can be acknowledged at the end of any machine cycle.

Interrupt breakpoint (only at the end of instruction)

WORKFLOW of DMA



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

CPU provides SAR, DAR, count register to DMAC.

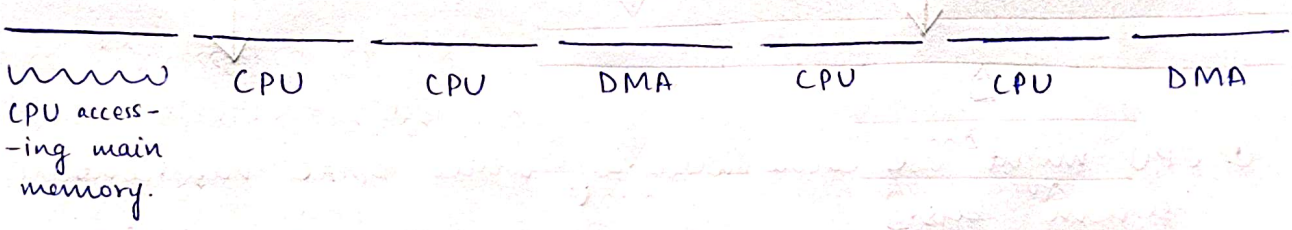
⊙ Processor is involved only at the beginning and at the end of the transfer.

⊙ DMA can be acknowledged during instruction execution but interrupt is acknowledged only at the end of instruction execution.
(coz no context switch in case of DMA-ACK).

Modes of DMA Transfer –

① Cycle Stealing Mode –

⊙ 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.

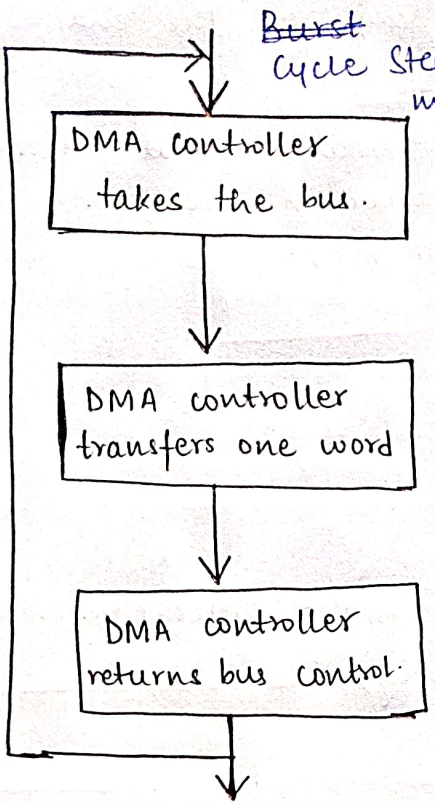
② Burst Mode

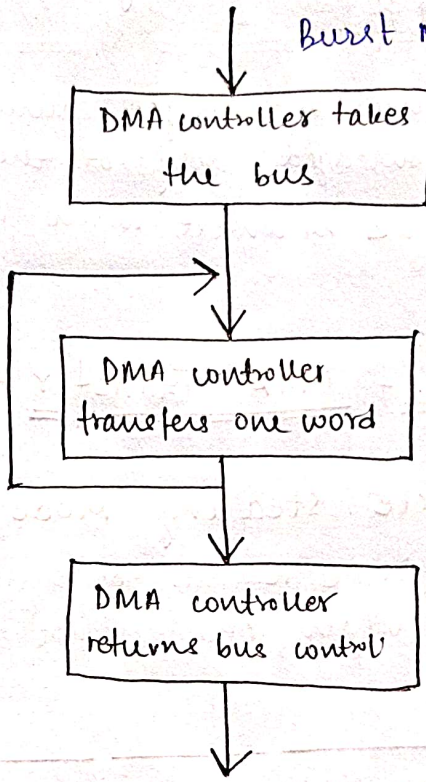
⊙ DMA Controller transfers the whole block of data without interruption.

⊙ CPU lies idle during this period as it cannot fetch new instruction from memory.

⊙ Results in max. possible data transfer rate.

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





⊙ 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.

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

↳ DMA initialization takes lot of time + interrupt at the end

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

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 ins 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

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

3hr video watch all ques. { I/O Lec 6 – DMA all GATE PYQs.
90 classes for GATE CS.

Priyanshu