* Computer Architecture ⇒ It refers to those attributes of a system visible to a programmer.

⇒ Attributes that have a direct impact on logical execution of a program.

⊙ Designing of Computer

⊙ Designer point of view

* Eg. of Architectural attributes ⇒ instruction set, no. of bits used to represent various datatypes, I/O mechanism, techniques for addressing memory etc.

* Computer Organisation ⇒ It refers to operational units and their interconnections that realize architectural specs.

⊙ Logic and circuits

⊙ Programmer point of view

* Eg. of Organizational attributes ⇒ Hardware details that are transparent to programmer, interfaces b/w peripherals and computer, memory technology used.

COMPUTER ⇒ Complex system, contemporary computers contains millions of elementary electronic components.

* Due to complexity ⇒ we need to know hierarchical nature of most complex systems.

* Designer is concerned with 2 aspects =>

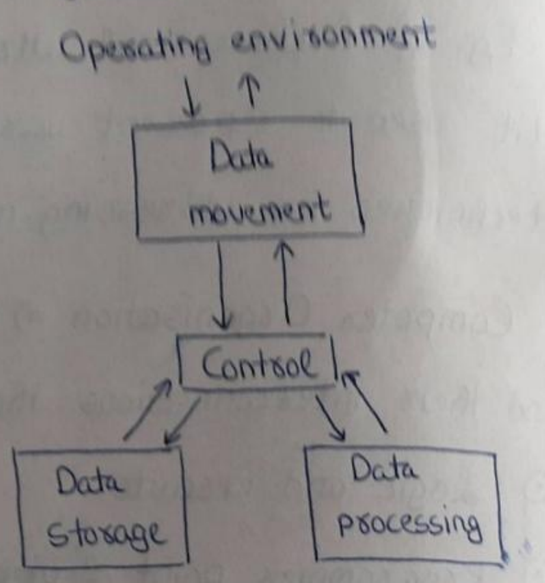
(A) Structure => the way in which components are interrelated

(B) Function => operation of each individual component as a part of structure.

Functional Description =>

* There are 4 basic functions of computer =>

(1) Data processing
(2) Data storage
(3) Data movement
(4) Control



* functions are ↴

Bidirectional

Structure Description =>

* 4 main structural components =>

(1) CPU => ⊙ controls the operation of computer
⊙ performs data processing through processor.

(2) Main memory => stores data

(3) I/O => moves data b/w computer & external environment.

(4) system interconnection => communication path (bus) b/w components of computer.

* Structural components of CPU =>

(1) Control unit => controls operations of CPU / computer.

(2) ALU => performs data processing functions.

(3) Registers => provides storage internal to CPU.

(4) CPU interconnection => communication among components of CPU

* Control Unit => ⊙ sequencing logic
⊙ Control unit registers & decoders
⊙ control memory.

3 principles characteristic of Computer =>

(A) It responds to a specific set of instructions in a well-defined manner.

(B) It can execute a pre-recorded list of instructions.

(C) It can quickly store & retrieve large amounts of data.

* First use of word "computer" => 1613

* History of calculating techniques ↴

=> Tally sticks → Abacus → Napier's Bones (1614) → Slide Rule (1622) ↓ Pascaline (1642) ← Stepped Reckoner (1672) ← Jacquard Loom (1861) ← Arithmometer (1820)

↳ Analytical Engine (1822-34) → etc....

* Computer Generations => (5)

(A) First Generation (1946-1958) => Vaccum tubes

(B) Second Generation (1959-1964) => Transistors (40 Vac. tubes)

(C) Third Generation (1965-1970) => Integrated circuits (ICs)

(D) Fourth Generation (1971-today) => Microprocessors

(E) Fifth Generation (today-future) => Artificial Intelligence

Moore's Law =>

* Gordon Moore (cofounder of Intel) in 1965 observed that the no. of transistors that could be put on a single chip was doubling every year.
* This pace slowed to 18 months, but has sustained it ever since.

⊙ Consequencies of Moore's law =>

(1) Cost of chip has remained virtually unchanged during period of rapid growth in density => It means cost of computer logic and memory circuitary has fallen at a dramatic rate.

(2) As logic and memory elements are placed closer together on more densely packed chips => Electrical path length is shortened => operating speed increases.

(3) Reduction in power and cooling requirements.

(4) Computer becomes smaller => more convenient to place in a variety of environments.

(5) Inter connections on IC are much more reliable than before.
=> With more circuitary on single chip, interchip connections are less.

Von Neumann =>

* John Mauchly and J. Presper Eckert conceived of an easier way to change the behaviour of their calculating system.

=> They reckoned that memory devices, in the form of MERCURY DELAY LINES, could provide a way to store program instructions.

=> They documented their idea proposing it as foundation of their next computer => EDVAC.

⊙ After reading their proposal, a Hungarian mathematician John von Neumann published the idea so effectively that all stored program computers have come to be known as von Neumann systems.

=> History credited him.

* Today's version of stored computers satisfies atleast these following characteristics =>

(1) Consists of these hardware systems ↴

=> CPU, ALU, Registers, main memory & an I/O system.

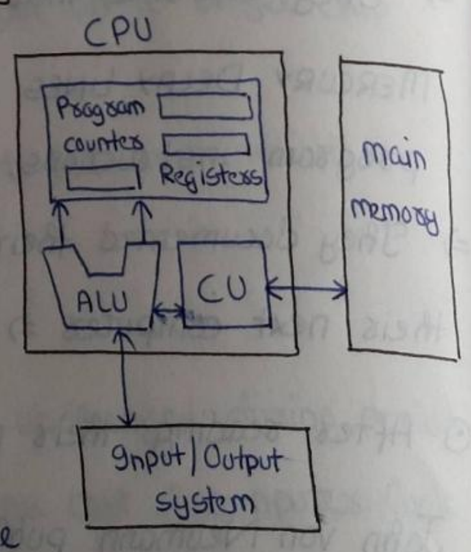
(2) Capacity to carry out sequential instruction processing.

(3) Contains a single path b/w main memory and CU of CPU, forcing alteration of instruction and execution cycles.

=> this single path known as => von Neumann Bottleneck.

* von Neumann Architecture & working =>

=> running of program is called (von Neumman execution cycle / fetch-decode execution cycle).



(1) CU fetches next program instruction from memory, using program counter where instruction is located.

(2) Instruction is decoded into a language that ALU can understand.

(3) Any data operands required to execute instruction is fetched from memory and placed into registers in CPU.

(4) ALU executes the instructions and places the results in memory or registers.

⊙ Drawback of earlier von Neumann Architecture =>

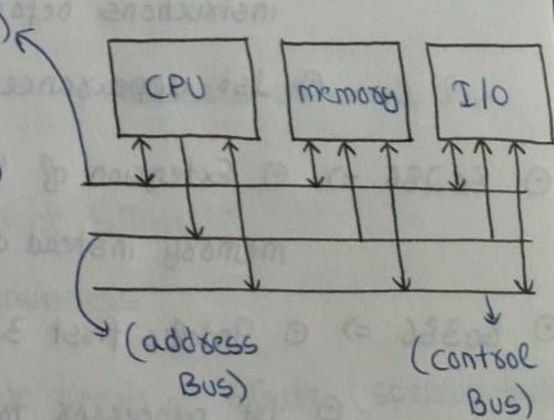
=> Only a single path (bus) was available. } slow working of system

* To overcome this ↴

Modified von Neumann Architecture =>

⊙ That architecture has been streamlined => System Bus Model

* Working ↴



(1) Data bus moves data from main-memory to CPU registers (vice versa)

(2) Address bus holds address of data that data bus is currently accessing.

(3) Control bus carries necessary control signals that signifies how info transfer is to take place.

Non-Von Neumann Architecture =>

* Uses the concept of Parallel Computing.

Concept => If computer is not powerful enough ↴

⊙ Instead of trying to develop a faster / powerful computer, use · MULTIPLE COMPUTERS.

Evolution of Intel x86 Architecture =>

⊙ 8080 => ⊙ World's first general purpose microprocessor.

⊙ 8-bit machine, 8 bit data path to memory.

⊙ 8086 => ⊙ 16-bit machine

⊙ supported an instruction cache (that prefetches the instructions before their execution).

⊙ 1st appearance of x86 arch.

⊙ 80286 => ⊙ Extension of 8086 enabled addressing a 16-Mbyte memory instead of 1-Mbyte.

⊙ 80386 => ⊙ Intel's first 32-bit machine => rivaled complexity

⊙ 1st processor to support multitasking, i.e. running multiple programs at same time.

⊙ 80486 => ⊙ introduced more powerful cache technology.

⊙ offered in-built to math coprocessor.

⊙ Pentinum => ⊙ Intel introduced use of superscalar techniques. that allow multiple instructions running in parallel.

⊙ Pentinum Pro => ⊙ Aggressive use of register naming, branch prediction, data flow analysis.

⊙ Pentinum II => ⊙ Incorporated MMX technology, designed to process video, audio and graphics data efficiently.

⊙ Pentinum III => ⊙ additional floating point instructions to support 3-D graphics.

⊙ Pentinum 4 => ⊙ more additional floating point.

⊙ Core => ⊙ first Intel up with dual core, 2 processors on single chip.

⊙ Core 2 => ⊙ 64 bits

⊙ Core 2 Quad provides 4 processors on single chip.

* In 1978, 8086 => ⊙ clock speed of 5MHz

⊙ 29000 transistess

In 2008, Core 2 Quad => ⊙ clock speed of 3GHz (speedup fac by 600)

⊙ 820 million transistors

(28000 times of 8086)

Data Representation in Computer Systems =>

* System can only understand 0/1.

* Signed Numbers Representation =>

⊙ n-bit vector, $B = b_{n-1} \cdots b_2 b_1 b_0$ => Eg. => 4 bit no.

=> value of b_i can be 0/1. => $b_3 b_2 b_1 b_0$

* We can represent both +ve and -ve nos.

=> for this we have three representations =>

(1) Sign-and magnitude

(2) 1's compliment

(3) 2's compliment.

* In sign magnitude => ⊙ if MSB is 0 => +ve no.

⊙ if MSB is 1 => -ve no.

(Problem)

↳ it represents 0 as +0(0000) & -0(1000)

which is not appropriate.

=> we find this same problem in 1's compliment.

* But in 2's comp. => no such problem

* ∴ 2's comp. is used efficiently in our systems for calculations.

=> In signed => 0, +ve, -ve nos. } everything same for

=> In unsigned => 0 & +ve nos. } +ve numbers.

Cases => (1) +65 01000001

+5 +00000101

+70 01000110 => $(70)_D$ => Ans.

(2) +65 01000001

-5 +11111011

+60 (1) 00111100 => Ans.

(discarded)

* 2's comp. of -5 ↴ 11111011

(3) -65 10111111

+5 +00000101

-60 11000100

00111011

+1

00111100 => Ans.

* 2's comp. of -65 ↴ 10111111

(4) -65 10111111

-5 11111010

-70 (1) 10111010

01000101

+1

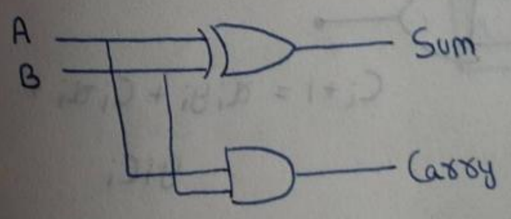
01000110 => Ans.

(discard)

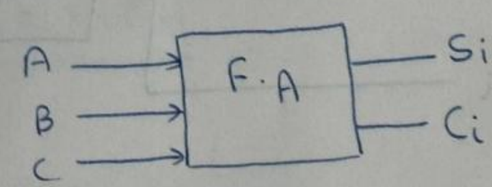
} (2's comp. of -70)

ADDER => * Basic unit of ALU.

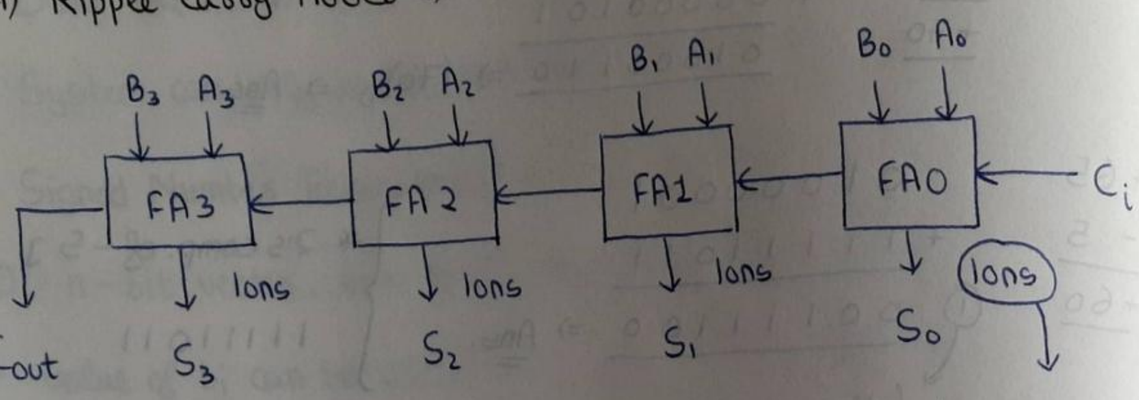
* Half Adder ↴



* Full Adder =>



(A) Ripple Carry Adder ⇒



(processing time of each FullA.)

* Here C_{output} of FA0 will become C_{input} for FA1 and so on.

⊙ ∴ FA1 has to wait for the output of FA0 ⇒ 1ons

* it will take, for full sum ⇒ 4ons = 4bit no.
  8ons = 8bit
  16ons = 16bit

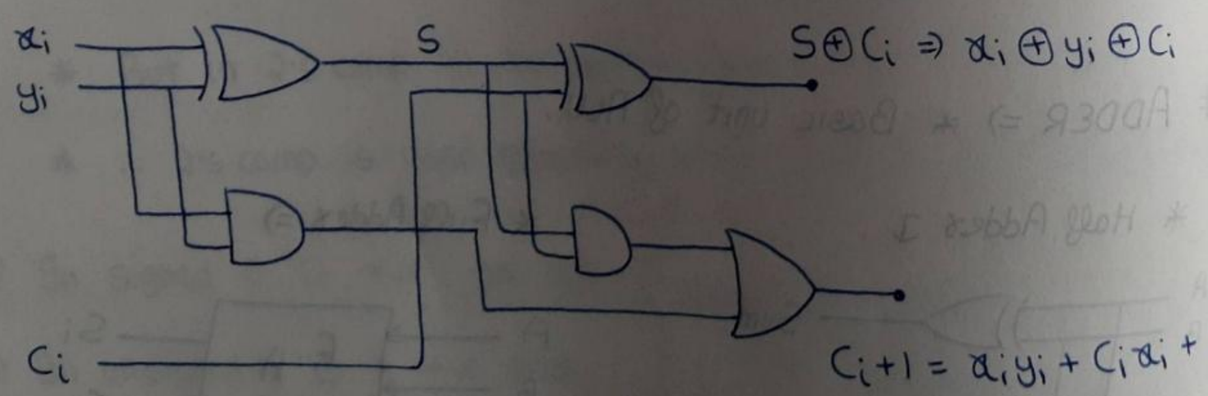
⇒ Processing speed is more

∴ cannot use this in system. ⊗

(B) Carry Look Ahead Adder / Carry Propagation Adder ⇒

⇒ propagates the carry

* 2 half adders ⇒ 1 full Adder



⇒ $C_{i+1} = x_i y_i + C_i x_i + C_i y_i$

⇒ $x_i y_i + C_i (x_i + y_i)$

G_i P_i

* $G_i = x_i y_i$
  $P_i = x_i + y_i$

* for $i=0$ ⇒ $C_1 = G_0 + C_0 P_0$ ⇒ depending on first carry C_0 (initial)

* for $i=1$ ⇒ $C_2 = G_1 + C_1 P_1$

⇒ $G_1 + (G_0 + C_0 P_0) P_1$

⇒ $G_1 + G_0 P_1 + P_0 P_1 C_0$ ⇒ still depending on C_0

⇒ Carry Look Ahead adder is better

⇒ can be used in systems.

⊙ Understanding ⇒ $x_i y_i + C_i (x_i \oplus y_i)$

of C_{i+1} eqn. ⇒ $x_i y_i + C_i (x_i \bar{y_i} + \bar{x_i} y_i)$

$(A + \bar{A}B = A + B)$

⇒ $x_i y_i + C_i x_i \bar{y_i} + C_i \bar{x_i} y_i$

⇒ $x_i (y_i + \bar{y_i} C_i) + C_i \bar{x_i} y_i$

⇒ $x_i (y_i + C_i) + C_i \bar{x_i} y_i$

⇒ $x_i y_i + x_i C_i + C_i \bar{x_i} y_i$

⇒ $x_i y_i + C_i (x_i + \bar{x_i} y_i)$ ⇒ $(A + \bar{A}B = A + B)$

⇒ $x_i y_i + C_i (x_i + y_i)$

⇒ $\boxed{x_i y_i + C_i x_i + C_i y_i}$

Arithmatic Operations ⇒

* 2 things needed to perform operation ⇒

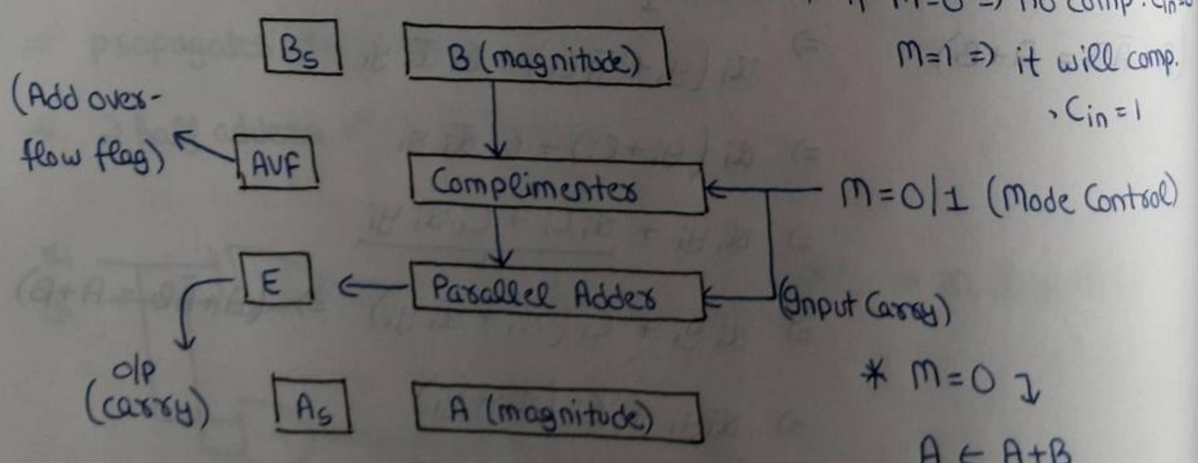
(1) Hardware

(2) Algorithm

* A-B ⇒ A ← A + $\bar{B}$ + 1

	Magnitude Addition	Magnitude Subtraction A>B	A<B	A=B
(+A) + (+B)	+ (A+B)			
(+A) + (-B)		+ (A-B)	- (B-A)	+ (A-B)
(-A) + (+B)		- (A-B)	+ (B-A)	± (A-B)
(-A) + (-B)	- (A+B)			
(+A) - (+B)		+ (A-B)	- (B-A)	+ (A-B)
(+A) - (-B)	+ (A+B)			
(-A) - (+B)	- (A+B)			
(-A) - (-B)		- (A-B)	+ (B-A)	+ (A-B)

* Hardware ⇒



* A_S / B_S ⇒ sign bits

* if M=0 ⇒ no comp. C_{in}=0

M=1 ⇒ it will comp. , C_{in} = 1

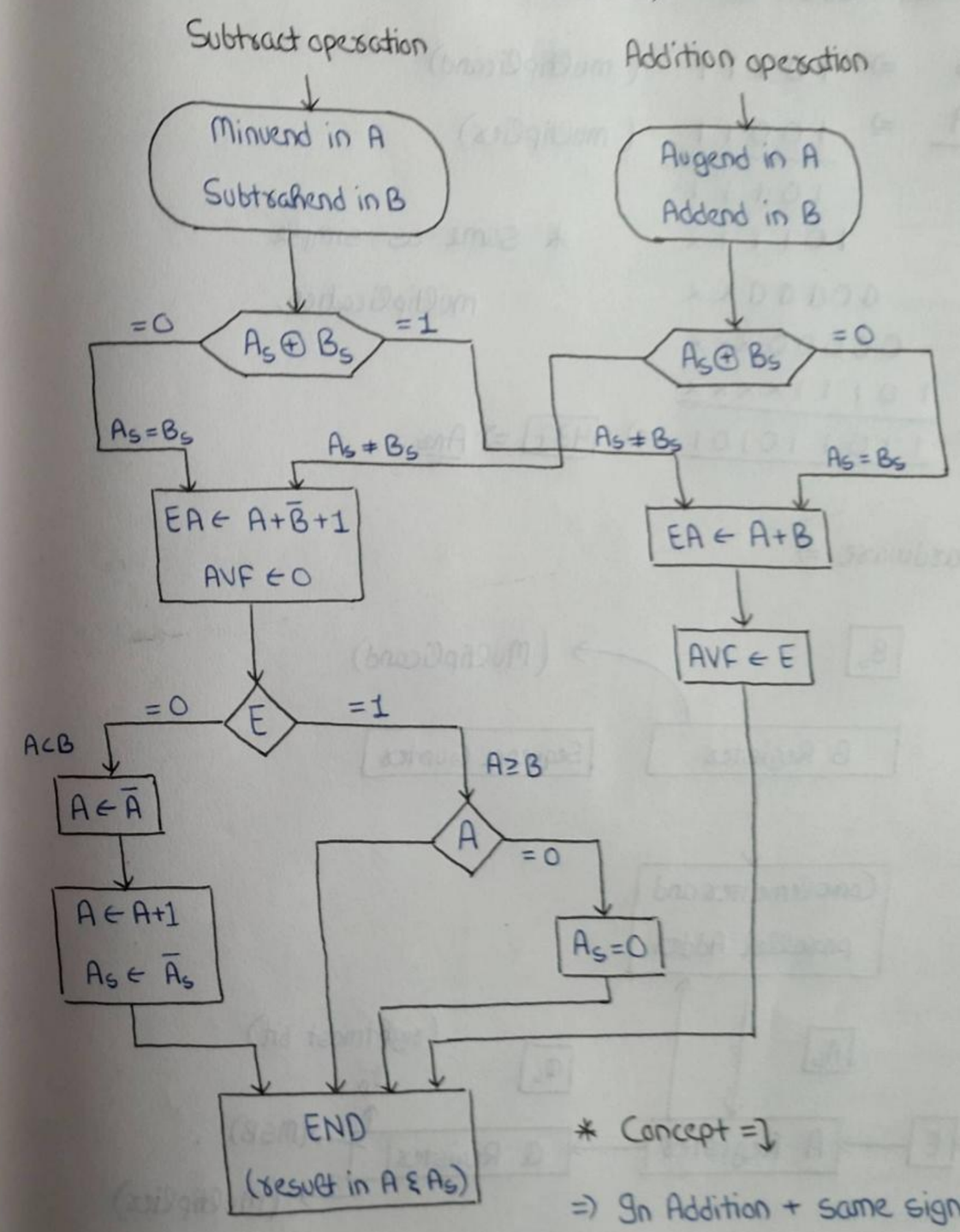
* M=0 ↴

A ← A+B

* M=1 ↴

A ← A + $\bar{B}$ + 1

* Algorithm ⇒ (Addition & Subtraction)



* Concept ⇒

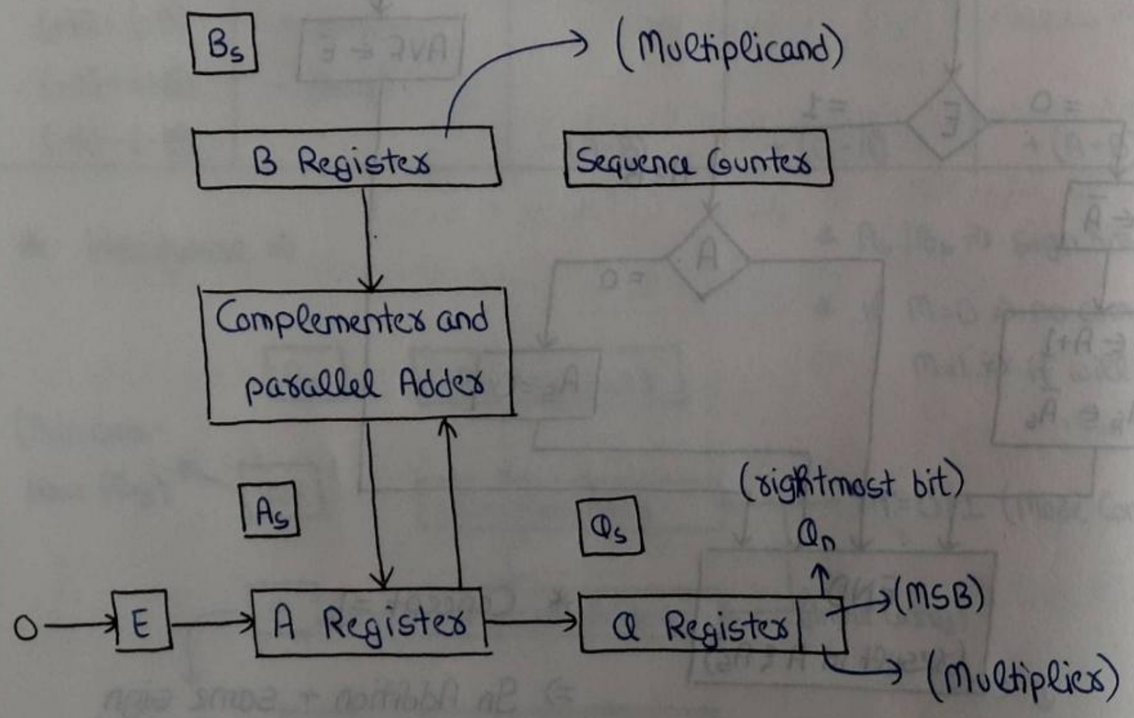
⇒ In Addition + same sign will follow same procedure as in Subtraction + different sign.

and vice versa.

* Multiplication ⇒

```
  23  ⇒  10111   (multiplicand)
× 19  ⇒  10011   (multiplier)
         -----
         10111
        10111x        * same as simple
       00000xx          multiplication.
      00000xxx
     10111xxxx
    ----------
    110110101  ⇒ [437] ⇒ Ans.
```

* Hardware ⇒



* we have taken 2 registers A and Q,
  as the product is very high i.e. more bits.

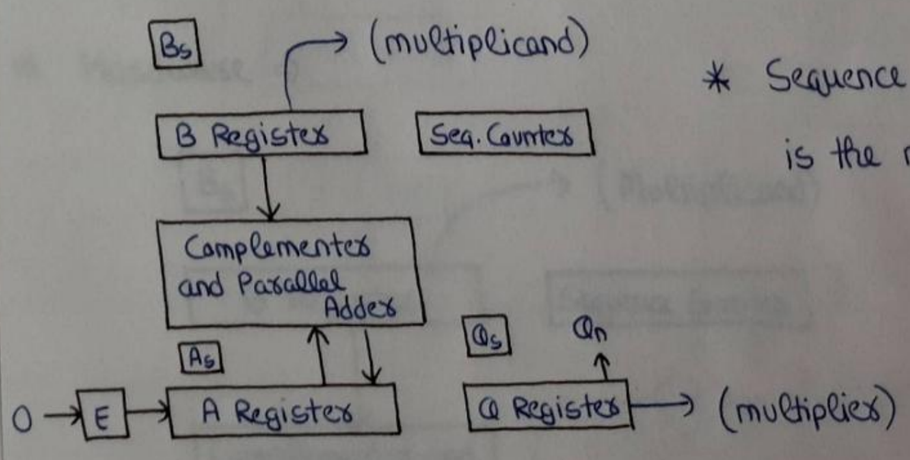
(Multiplication) ⇒ Binary multiplication is similar to simple multiplication on paper.

⊙ Eg ↴

```
  10111  (multiplicand)
x 10011  (multiplier)
---------
  10111
 10111
10111
---------
110110101  (product)
```

* Two things required are again same ⇒ Hardware and Algorithm.

* Hardware ↴

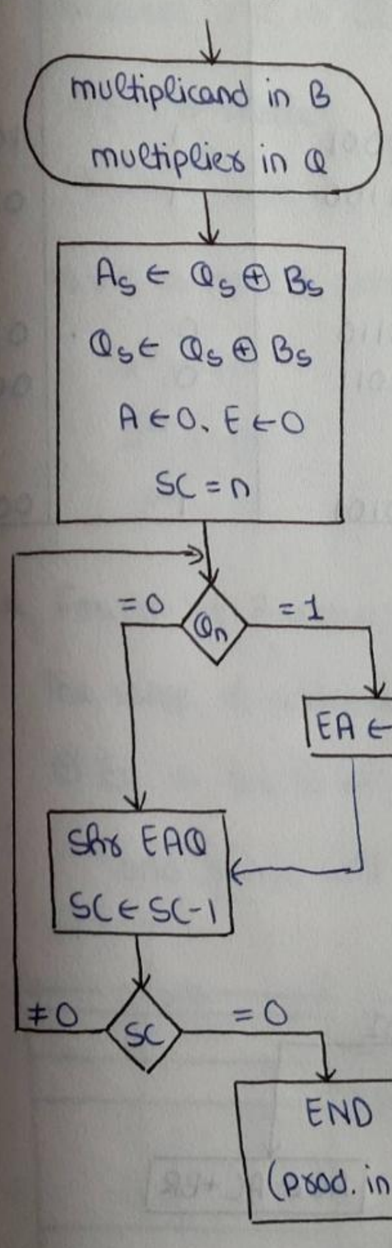


* Sequence counter (SC) is the no. of bits (n).

Multiplicand B = 10011	E	A	Q	SC
* multiplier in Q	0	00000	10011	101 (5)
Q_n = 1, Add B		10111		
* First partial product	0	10111		
* Shift right EAQ	0	01011	11001	100 (4)
Q_n = 1, Add B		10111		
* second partial product	1	00010		
* Shift right EAQ	0	10001	01100	011 (3)
* Q_n = 0, shift right EAQ	0	01000	10110	010 (2)
* Q_n = 0, shift right EAQ	0	00100	01011	001 (1)
* Q_n = 1, add B		10111		
* fifth partial product	0	11011		
* Shift right EAQ	0	01101	10101	000 (0)

* Final product in AQ = 0110110101

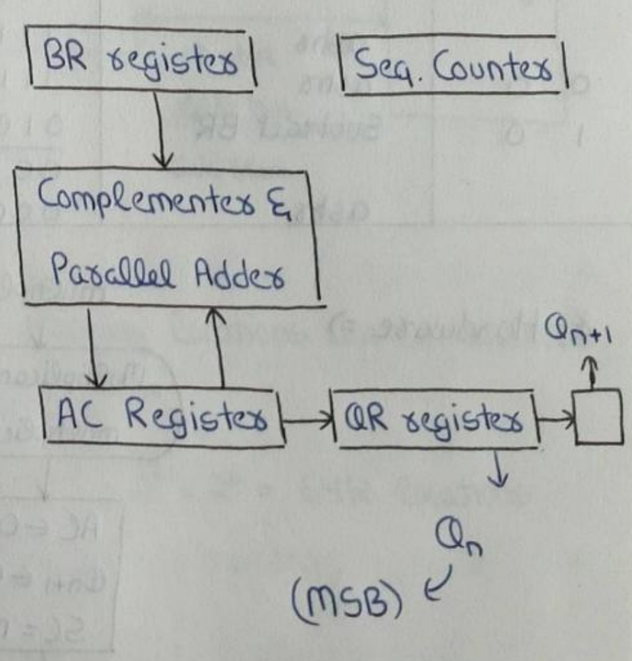
⊙ multiply Algorithm



* Another method ⇒

Booth Multiplication Algo. ↴

⊙ Hardware ↴



* Example Table on next Page - - - - -

* Difference b/w Normal & Arithmatic Shift right ⇒

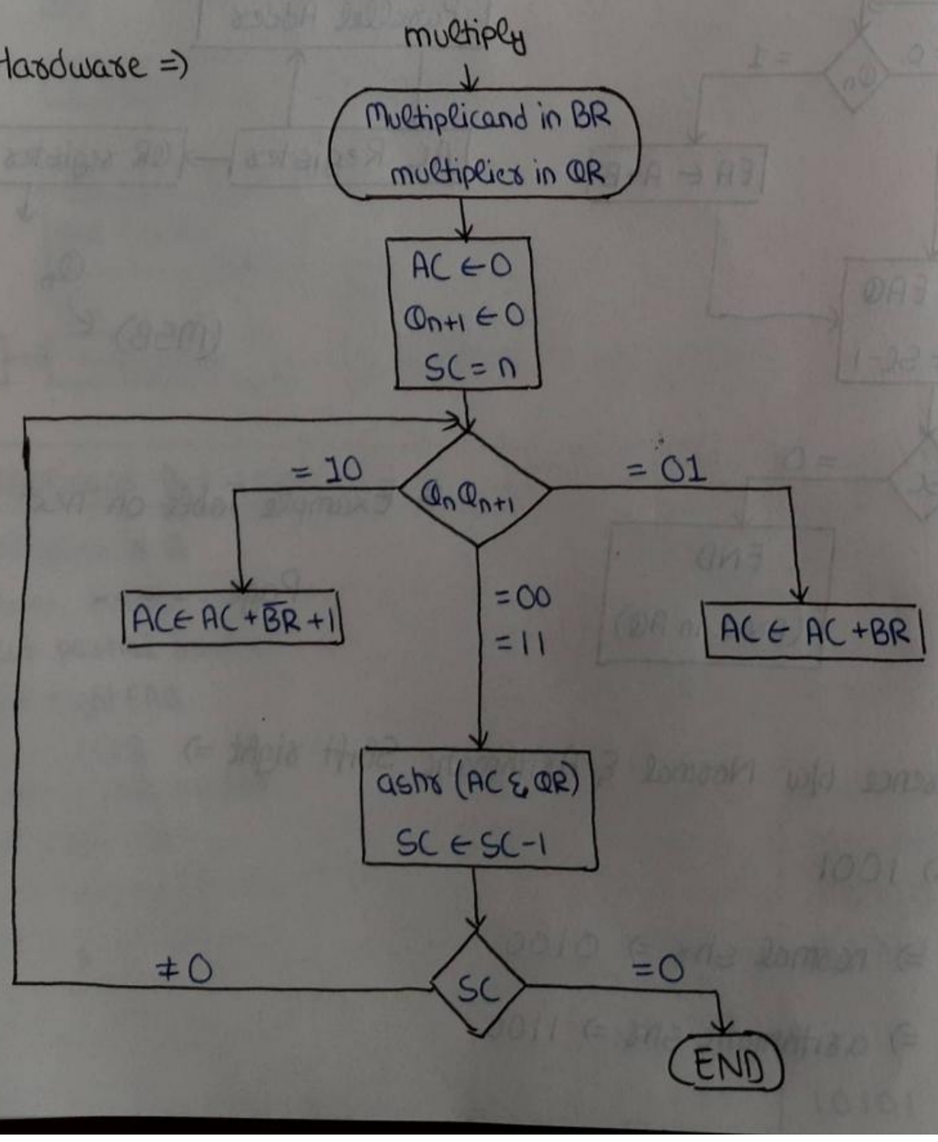
⊙ Eg. ⇒ 1001

⇒ normal shr ⇒ 0100

⇒ arithmatic shr ⇒ 1100

Q_n Q_{n+1}	BR = 10111 $\overline{BR}+1$ = 01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

* Hardware =>



UNIT 2 => Memory Location and Addresses =>

* smallest unit => 0/1 (bit)

4 bits = nibble

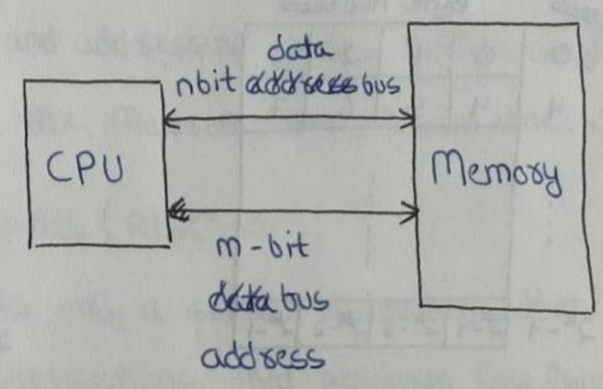
8 bits = byte

16 bits or more = word

2^{10} = 1K

2^{20} = 1M

2^{30} = 1G

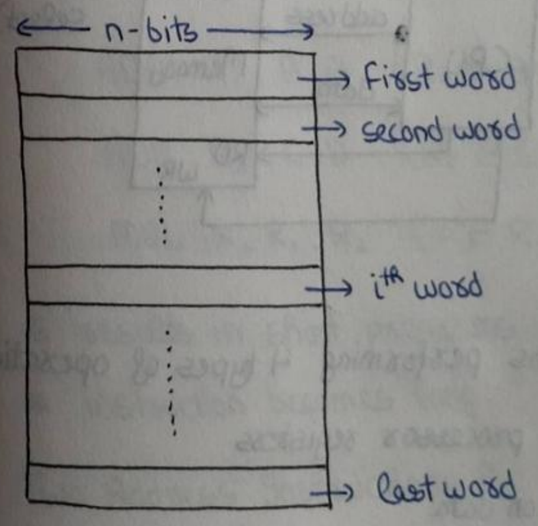


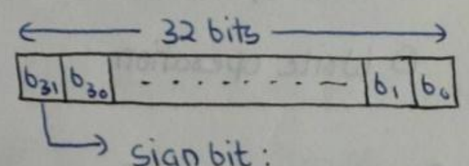
* For n-bit Address bus => 2^n memory locations can be address.

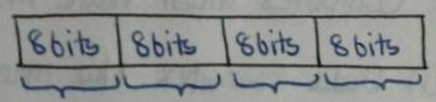
The range of addresses is from 0 to $2^n - 1$.

⊙ Eg. => for 16 bit address => $2^{16} = 2^{10} \times 2^6$ = 64K locations

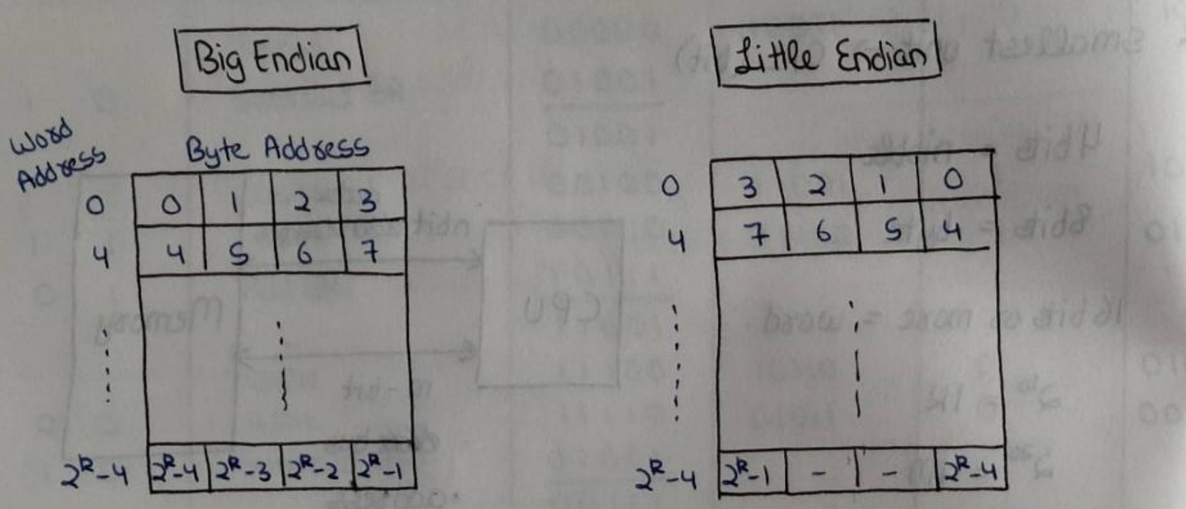
and range will be 0 to $(2^{16} - 1)$.







* 2 formats to Store Data =>



* Eg. => 10011 will be stored as =>

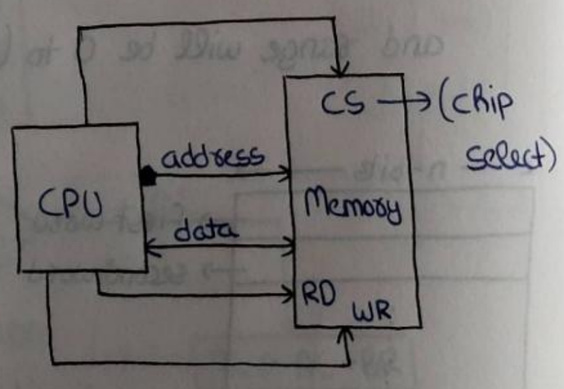
⊙ 10011 in Big Endian

⊙ 11001 in Little Endian.

* Memory Operation =>

⊙ Read operation

⊙ Write operation



* Instruction and Instruction Sequencing ↴

=> Computer must have instructions performing 4 types of operations =>

(1) Data transfer b/w memory & processor registers

(2) Arithmatic & Logic operations on data

(3) Program sequencing & control

(4) I/O transfers.

* Complex Instruction Set Computer (CISC) =>

=> CISC is a single instructions can execute several low-level instructions.

=> capable of multioperations and addressing modes within single instructions. Eg. => PDP-11, VAX, Motorola 68K, PCs on intel x86.

* Reduced Instruction Set Computer (RISC) =>

=> RISC is a computer that uses only a single instructions that can be divided into multiple instructions that perform low level operations within single clock.

=> Eg. => AMD 29R, ARC, intel i860, 960 etc.

Types of Instructions on basis of Address Used =>

(1) Three Address Instructions => (Opcode, Destination, Source)

⊙ Eg. => Program to evaluate => (A+B)*(C+D)

ADD R_1, A, B ($R_1 \leftarrow A+B$)

ADD R_2, C, D ($R_2 \leftarrow C+D$)

MUL X, R_1, R_2 ($X \leftarrow R_1 * R_2$)

* results in short programs

* instruction becomes long

(2) Two Address Instructions =>

⊙ Eg. => Program => (A+B)*(C+D)

MOV R1, A (R1 ← M[A])

ADD R1, B (R1 ← R1 + M[B])

MOV R2, C (R2 ← M[C])

ADD R2, D (R2 ← R2 + M[D])

MUL R1, R2 (R1 ← R1 * R2)

MOV X, R1 (M[X] ← R1)

(3) One Address Instructions ⇒ uses an implied AC register for all data manipulation.

⊙ Eg. ⇒ (A+B) * (C+D)

LOAD A (AC ← M[A])

ADD B (AC ← AC + M[B])

STORE T (M[T] ← AC)

LOAD C (AC ← M[C])

ADD D (AC ← AC + M[D])

MUL T (AC ← AC * M[T])

STORE X (M[X] ← AC)

(4) Zero Address Instructions ⇒ (TOS ⇒ Top of stack)

PUSH A (TOS A)

PUSH B (TOS B)

ADD (TOS (A+B))

PUSH C

PUSH D

ADD (TOS (C+D))

MUL (TOS (C+D) * (A+B))

POP X (M[X] ← TOS)

* stack is used.

* Notations ⇒

⊙ Arithmatic Expression ⇒ A+B

⊙ Infix Notation ⇒ A+B

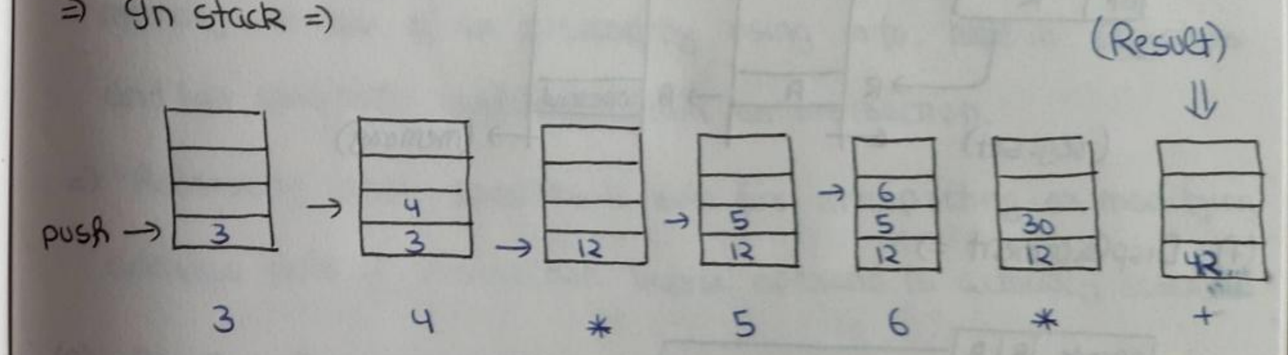
⊙ Postfix / Reverse Polish notation ⇒ AB+ (most suitable)

⊙ Prefix / Polish notation ⇒ +AB

* Eg. ⇒ A*B + C*D ⇒ AB*CD*+

* Eg. ⇒ (3*4) + (5*6) ⇒ 34*56*+

⇒ In stack ⇒



* Addressing Modes ⇒

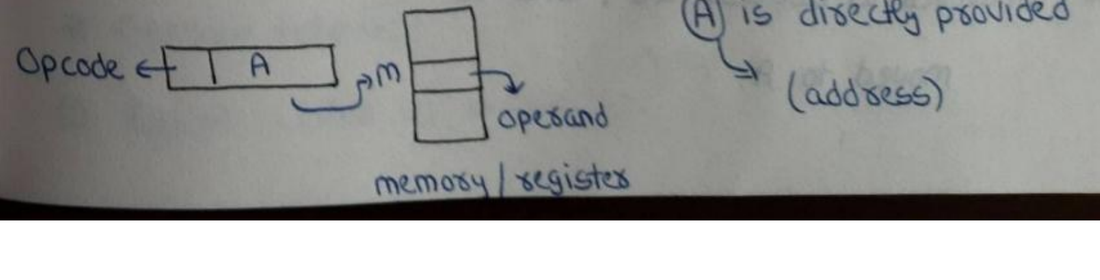
(1) Implicit ⇒ Eg. ⇒ INCR (increment)

COMP (compliment)

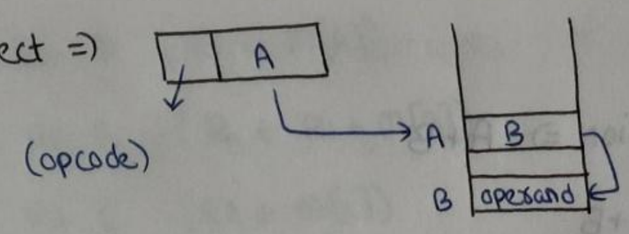
(2) Immediate ⇒ Data is directly transfered to a memory / register location. Eg. ⇒ MOV A, #32H

ADD A, #22H

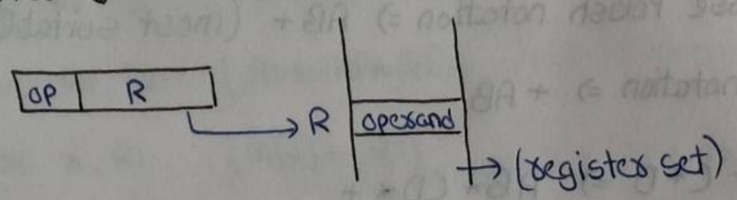
(3) Direct Addressing ⇒



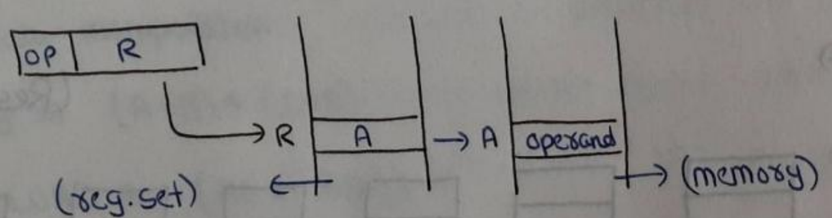
(4) Indirect =>



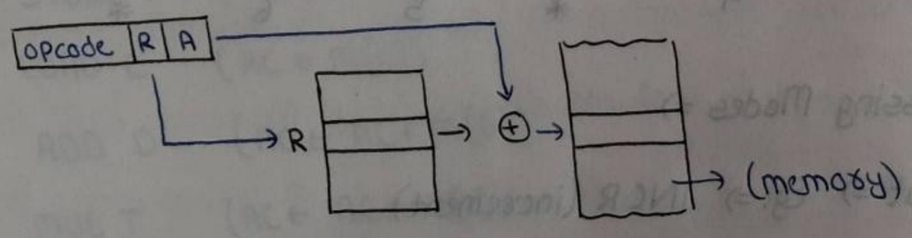
(5) Registers ↴



(6) Register Indirect ↴



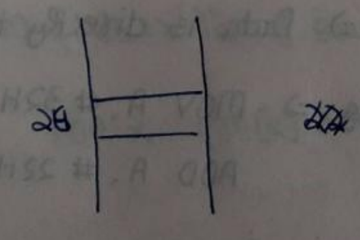
(7) Displacement =>



(8) Relative ↴

MOV A, [22], [4]

↙ (base) ↓ (relative address)



* whatever at address 26, will be moved to A.

(9) Base register => R → index

(10) Indexing => A → starting base add.

R → index.

* opcode | mode | operand

⊙ Mode => The way operands are chosen during program execution is dependent on addressing mode of instruction.

=> An addressing mode specifies how to calculate effective memory address of an operand by using info. held in registers and/or constants contained within an instruction.

=> Addressing mode specifies a rule for interpreting or modifying address field of instruction before operand is actually reserved.

(A) Implied Addressing Mode =>

=> Operands are specified implicitly in definition of instruction.

⊙ Eg => Complement Accumulator

Comp A => acc. is complemented

* All register reference instruction that use an Accumulator are Implied mode Instructions.

(B) Immediate Addressing Mode => No Address field ⊗

* Operand field ✓

⊙ Eg. => LOAD R1, #50H

⊙ Advantage ⇒ (1) fastest Execution

(2) no memory reference to fetch data.

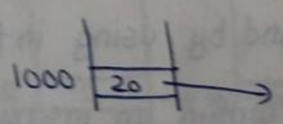
⊙ Disadvantage ⇒ (1) less flexible

(2) Low range

(C) Direct Addressing mode ⇒ Operands resides in memory, and its address is given directly by address field of instruction.

⊙ Eg. ⇒ LOAD R1, [1000]

R1 ← M[1000]



⊙ Advantage ↴

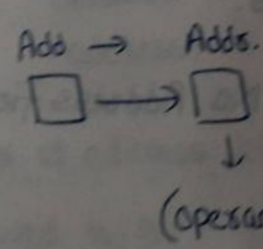
(1) single memory ref. to access data.

(2) no additional calc. to calculate effective address.

(D) Indirect Addressing Mode ⇒ Address field of instruction given the address where the effective address is stored. (address of operand).

⇒ Control fetches the instruction from memory and uses its address part to access memory again to read eff. address.

(E.A.)



* Indirect Addressing Mode

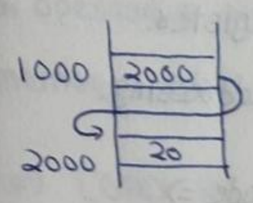
Memory Indirect A.M.

Register indirect A.M.

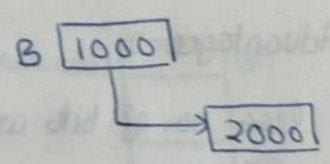
* Eg. ⇒ LOAD R1, (1000)

Or

LOAD R1, @1000



* Eg. ⇒ LOAD R1, (B)



(E) Register Addressing mode / Register Direct mode ⇒ Operands Registers are in the registers, that reside within CPU.

E.A. = R

* Eg. ⇒ ADD A, B ⇒ A ← A+B

⊙ Advantage ⇒ (1) no memory access, hence fast execution.

(2) very small address field needed.

⊙ Disadvantage ⇒ (1) limited no. of registers.

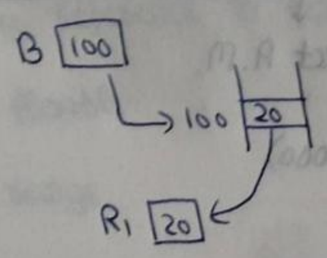
(2) Requires good assembly programming.

(F) Register Indirect Addressing Mode ⇒ The instruction specifies a register whose content gives the address of operand in memory.

⇒ Here, inst. uses fewer bits to select a register that would have been required to specify a memory address directly.

EA ⇒ address stored in register

* Eg. ⇒ LOAD R1, B



⊙ Advantages :

(1) less no. of bits are required to specify registers.

(2) one fewer memory access than indirect addressing.

(G) Auto increment / Autodecrement Addressing mode ⇒

```
MOV  R1, #R2
INR R1
DEC R1
```

(H) Relative Addressing Mode ⇒ Content of program counter (PC) is added to address part of inst. in order to obtain E.A.

E.A. = PC + Address part of instruction.

⇒ E.As position in memory is relative to the address of the next instruction.

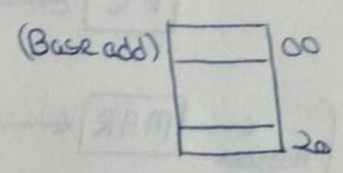
⇒ This mode is often used with branch type instruction where the branch address is in the area surrounding instruction.

⊙ Eg. ⇒ JUMP.

(I) Index Addressing Mode ⇒ Content of an index register is added to the address part of instruction to obtain two E.A.

⇒ Index register stores an index value.

⇒ Address field of instruction defines the starting address (base) of data array in memory.



⇒ Each operand in array is stored in memory relative to base address.

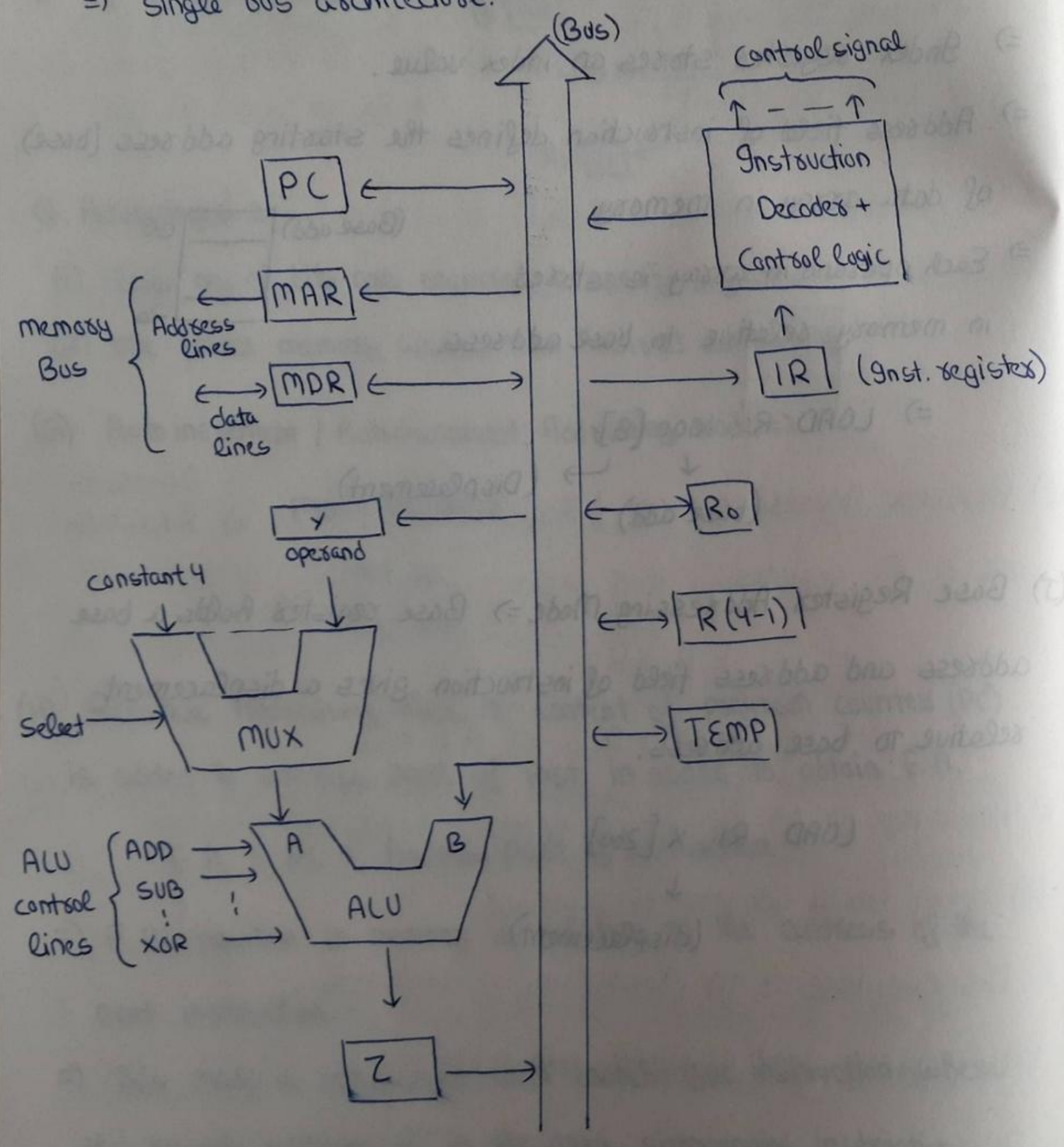
⇒ LOAD R, 1000 [B]
↓ (base add.) ↳ (Displacement)

(J) Base Register Addressing Mode ⇒ Base register holds a base address and address field of instruction gives a displacement relative to base address.

LOAD R1, X[200]
↓
(displacement)

* Execution of a Complete Instruction =>

=> single bus architecture.



* Execution => (1) Fetch the instruction from memory to IR.

(2) Fetch the 1st operand from memory.

(3) Perform addition operation

(4) Load result into R1.

ADD (R3), R1

R3 - 1000

R1 - 30

* Control Sequence =>

(1) PC_{out} , MAR_{in} Read . Select 4 , Add , Z_{in}

(2) Z_{out} , PC_{in} , WMFC (wait for memory functn. to complt.)

(3) MDR_{out} , IR_{in} .

(4) $R_3{}_{out}$, MAR_{in} , Read

(5) $R1_{out}$, Y , WMFC

(6) MDR_{out} , Select 4 , Add , Z_{in}

(7) Z_{out} , $R1_{in}$, EN (end).

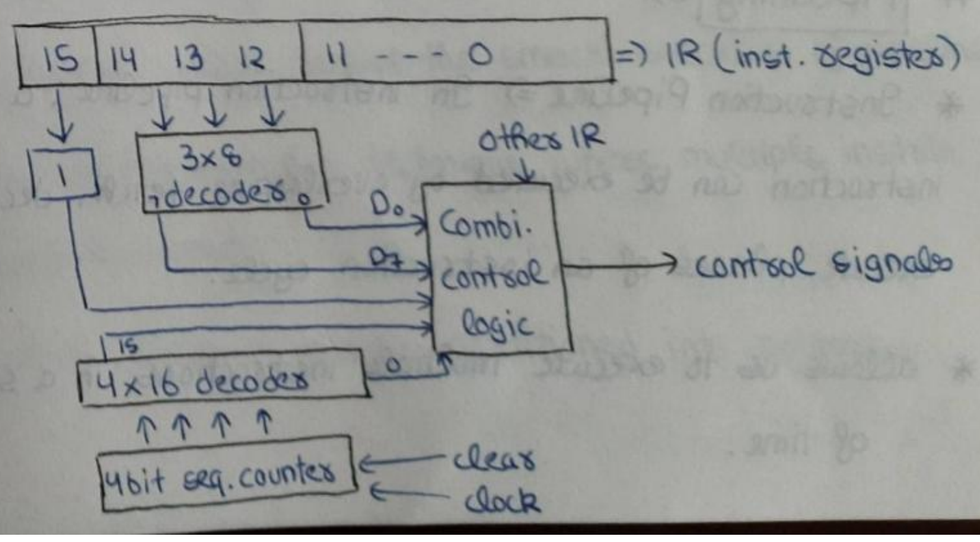
Hardwired v/s Microprogrammed CPU =>

(Design of control unit)

=> To execute an instruction , control unit of CPU must generate the req. signal in proper sequence.

* Hardwired CU => In this , control logic is implemented with gates , flipflops , decoders and digital circuits.

* fixed logic circuit



⇒ Hardwired CU consists of ⇒

(1) Instruction registers

(2) No. of control logic gates

(3) Two decoders

(4) 4-bit seq. counter

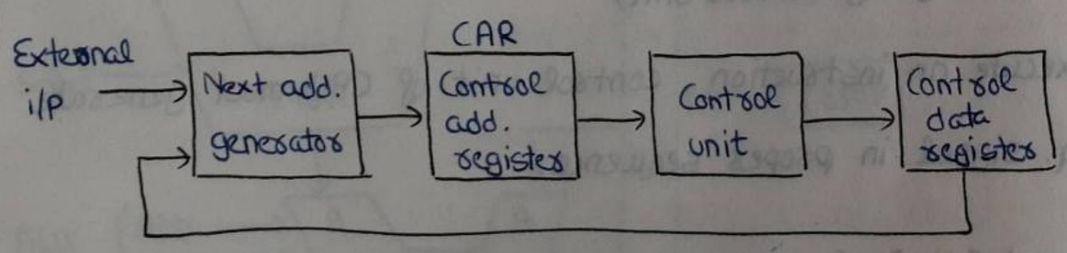
* Hardwired is faster than μ-prog. control.

* Controller using this approach can approach at high speed.

* For large CU ⇒ this organization is complicated.

* Modification of all combinational circuit is very difficult.

* Microprommed CU ⇒



Pipelining ⇒

* Instruction Pipeline ⇒ In instruction pipeline, a stream of instruction can be executed by overlapping fetch, decode and execute phases of an instruction cycle.

* allows us to execute multiple instructions in a single unit. of time.

⊙ throughput ⇒ no. of instructions | per unit of time.

⇒ This type of pipelining is used to increase the throughput of computer system.

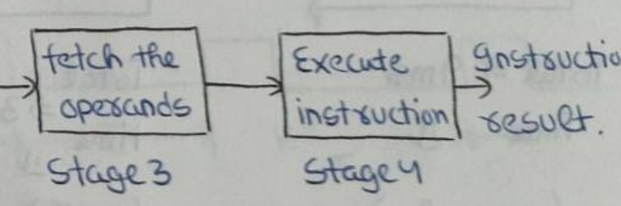
* Process of execution of instruction involves the foll. steps ⇒

(1) fetch the inst. from main memory.

(2) Decode the instruction.

(3) fetch the operands

(4) Execute the decoded instruction.



* Advantages ⇒ (1) more efficient use of processor.

(2) It can save circuitry.

* Disadvantages ⇒ (1) Pipelining involves adding H/w to the cip which increase cost of system.

(2) Inability to continously run the pipeline at full speed because of pipeline hazard, which disrupt the smooth execution of pipeline

⊙ Pipelining is an implementation technique where multiple instrtn. are overlapped in execution.

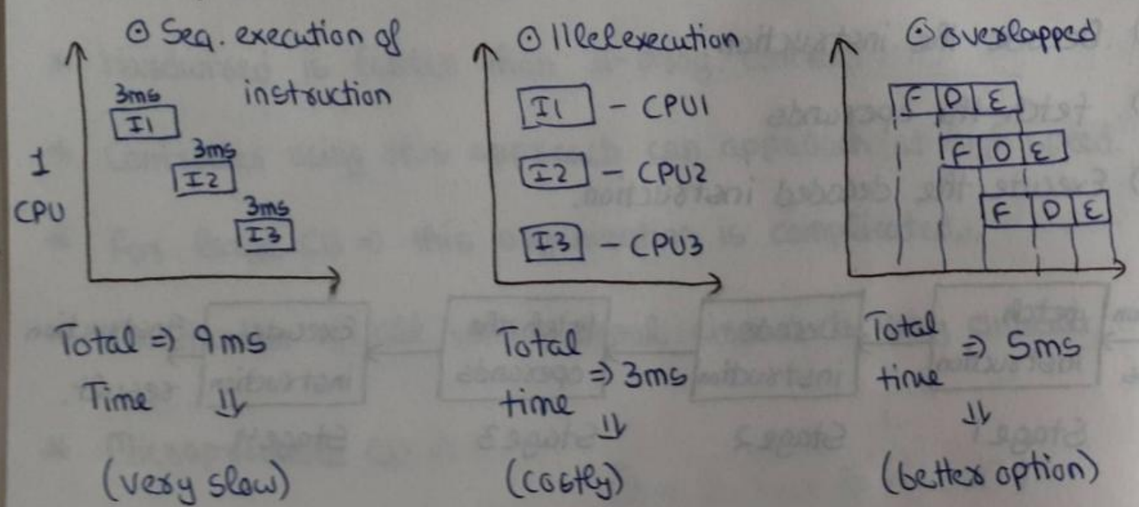
* used where operation can be partitioned into subtasks.

* To improve CPU performance ⇒ 2 options ⇒

(1) Improve hardware by introducing faster circuits, which is costly.

(2) Pipelining.

* Example ⇒



Total ⇒ 9ms
Time ⇓
(very slow)

Total time ⇒ 3ms ⇓
(costly)

Total time ⇒ 5ms ⇓
(better option)

* Design of Pipline ⇒ ↴

⇒ A pipeline has 2 ends ⇒ the i/p end and o/p end. B/w these ends there are multiple stages such that o/p of one stage is connected to i/p of next stage and each stage performs specific operation.

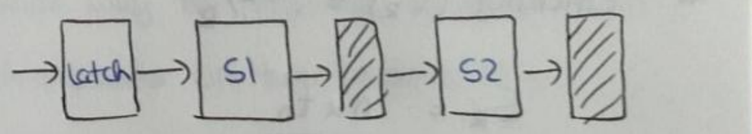
* Performance of Pipeline ⇒

* Speed up ratio (S_R)
* frequency (f)
* Efficiency (ε_R)
* Throughput (H_R)

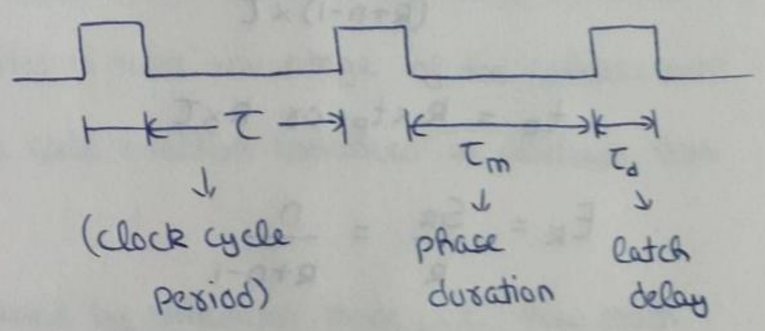
* Speed up ratio = non pipeline execution time / pipelined exe. time

$$S_R = \frac{n \times t_n}{(R+n-1) \times \tau}$$

IF ⇒ inst. fetch
ID ⇒ inst. decode
OF ⇒ operand fetch
OD ⇒ operand decode
OS ⇒ result store



$$\tau = \max\left(\overset{R}{\underset{i=1}{\tau_i}}\right) + \tau_D$$



Speed up ratio (S_R)

$$\frac{n \times t_n}{(R+n-1) * \tau}$$

* If $n \gg R$ so $R+n-1 \approx n$ ⇒ $(S_R) = \frac{n t_n}{n \tau} = \frac{t_n}{\tau}$

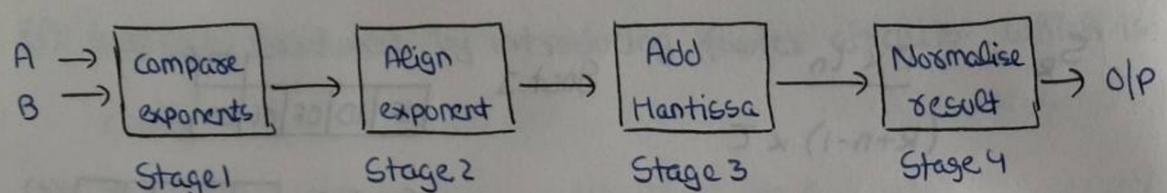
* Types of Pipelines ↴

(1) Arithmatic Pipelining ⇒ It is mostly used and found in most of the computers.

⇒ used for floating point arithmatic operation, multiplication of fixed point nos. etc.

* It can be applied to various complex and slow arithmatic op. to speed up processing time.

* Eg. ⇒ floating point Arithmatic pipelining.



* frequency(f) = $\frac{1}{\tau}$

* Efficiency (E_k) = S_k/k

$$S_k = \frac{n \times t_n}{(k+n-1) \times \tau}$$

$t_n = k \times t_p$ or $k \times \tau$

$$E_k = \frac{S_k}{k} = \frac{n}{k+n-1}$$

* Throughput (H_k) = $\frac{n}{(k+n-1)\tau}$

$$\Rightarrow \frac{nf}{k+n-1}$$

* **Amdahl's Law** ⇒ * used to obtain performance gain.

* It states that the performance improvement to be gained from using some faster mode of execution is limited by fraction of time the faster mode can be used.

* Speed up = $\frac{\text{Performance for entire task using enhancement when possible}}{\text{--- - - without using enhancement.}}$

OR

Speed up = $\frac{\text{Execution time without enhancement}}{\text{Exe. time using enhancement.}}$

⇒ Amdahl law gives a quick way to find the speedup from some enhancement which depends on two factors ⇒

(1) The functn. of the computation time in the original computer that can be converted to take advantage of the enhancement - ⇒ value which we call fraction enhanced is always less than 1.

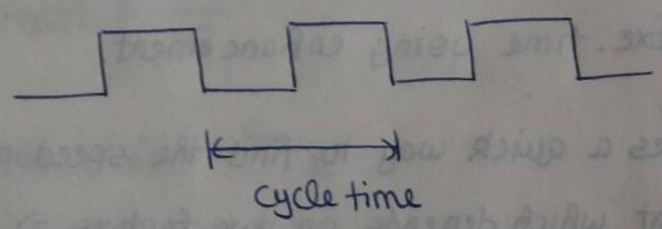
(2) The improvement gained by enhanced mode, i.e., how much faster the task would run if enhanced mode were used for entire program.

* This value is the time of the original mode over the time of enhanced mode. If enhanced mode takes 2 sec. for the portion of program, while it is 5 sec. in original mode, improvement is 5/2. we will call this value (greater than 1), speed up enhanced.

⊙ Execution Time (new) = Ex. time (old) × $\left[(1 - \text{fraction enhanced}) + \frac{\text{frac. enhanced}}{\text{speedup enhanced}} \right]$

* Speedup overall ⇒ $\frac{\text{Ex. time}_{old}}{\text{Ex. time}_{new}}$

Performance of Processor ⇒



F ⇒ freq. / clock rate

CC ⇒ cycle count

CT ⇒ cycle time

Time period = $1/f$

CPI ⇒ cycle per instruction

IC ⇒ inst. count

MFLOPS ⇒ million floating point operation per sec.

MIPS ⇒ millions inst. per second.

Execution time ⇒ CC × CT

⇒ IC × CPI × CT

⇒ $\frac{IC \times CPI}{f}$

CPI ⇒ Total CPU clk cycle for program / Instruction count.

⇒ $\sum_{i=1}^{n} \frac{CPI_i \times I_i}{IC}$

MFLOPS ⇒ no. of floating point operations in prog. / Ex. time × 10^6

MIPS ⇒ $\frac{IC}{\text{Execution time} \times 10^6} = \frac{F}{CPI \times 10^6}$

⊙ Eg. ⇒ Calculate CPI and MIPS for a CPU with 200MHz freq. which is executing a performance program with full. inst. axe ⇒

Inst. Category	Performance of occurence	No. of cycle per instruction
ALU	38	1
Load & Store	15	3
Branch	42	4
Others	5	5
	100	

Soln. ⇒ $\boxed{CPI} = \sum_{i=1}^{n} \frac{CPI_i \times I_i}{\text{Instruction Count}}$

⇒ $\frac{38\times1 + 15\times3 + 42\times4 + 5\times5}{100}$ ⇒ $\frac{276}{100} = 2.76$

$\boxed{MIPS}$ ⇒ $\frac{\text{clock rate}(f)}{CPI \times 10^6}$ ⇒ $\frac{200 \times 10^6}{2.76 \times 10^6}$ ⇒ 70.24