* Fetch – Decode – Execute Cycle

↳ Fetch current instruction, execute it & fetch the next instruction and continue

* hard disc → is not memory

↘ IO device

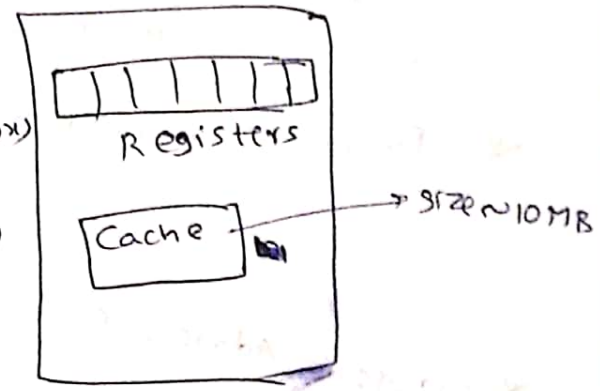
input output

* Cache → 3 levels (Multilevel cache)

Level 1 cache → 2 MB (approx)

Level 2 cache → 6 MB (")

Level 3 cache → 10 MB (")



* Main Memory ≐ DRAM

Secondary Memory = Floppy, Hard Disk, CD

Tertiary Memory = Tape, RAID

Cache " = SRAM

* MIPS → Unless otherwise mentioned, variables are always integers & signed.

→ comments start with #

* Name of a register starts with '$'

* Processor —(moving data)→ Memory : LOAD

* Memory → Processor : STORE

Only operations in MIPS involving Memory

* Register numbers

$t0 - $t7 → reg's 8 - 15

$t8 - $t9 → reg's 24 - 25

$s0 - $s7 → reg's 16 - 23

* $zero → we can't store anything in it
  ↓
  '0' is stored in it by default

* Barrel Shifter → shift any no. of bits in registers in '1' clock pulse

* LOOP: ← Address B

Address A ← bne $t1, $zero, LOOP

In the encoding of this instruction,

bne → 6 bits

$t1 → 5 bits

$zero → 5 bits

16 bits gone

⇒ Loop's address can't be stored in remaining 16 bits

∴ A + 4 − B is stored in the encoding

(called displacement)

* =, ≠ are faster than <, >, ≤, ≥ because to check if two numbers are equal, XOR them bitwise. To check if we got all 0's, NOR all the bits to see '1'.

* slt $rd, $rs, $rt

If $rs < $rt ⇒ $rd = 1
else ⇒ $rd = 0

* Procedure calling

$a0 - $a3 → arguments

$v0, $v1 → return values

$t0 - $t9 → temporaries

$s0 - $s7 → saved into memory (similar to static variables)

called callee saved registers

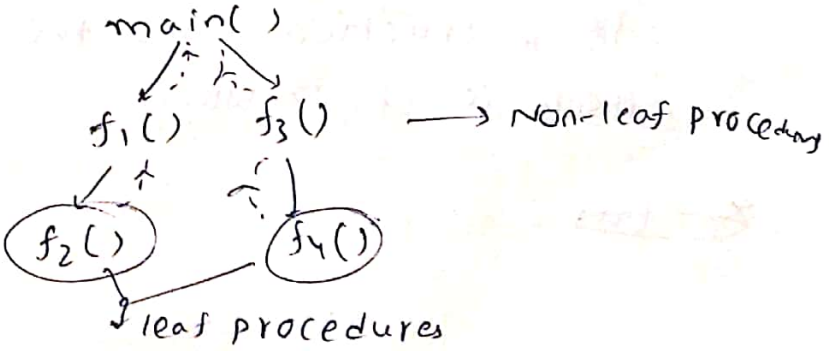
That which is called

Callee saves some values into memory. Before returning to the caller, it restores those values in these registers. [i.e they are needed for caller & they're passed from caller]

$sp → stack pointer

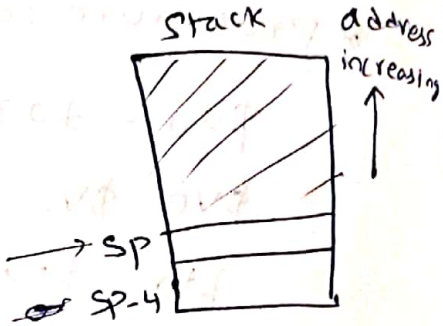
Memory works like a stack which grows downward.

~~→ leaf procedure → no need of stack pointer~~



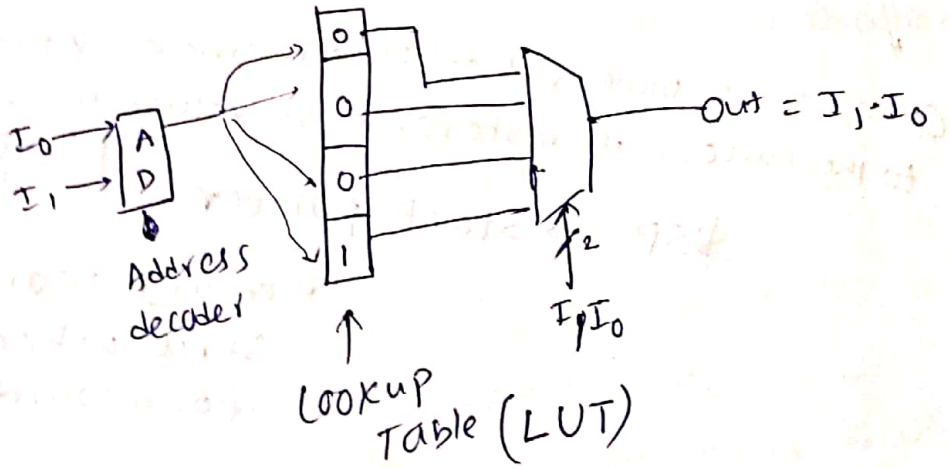
$ra → return address
[address of the instruction in the caller after the call to callee].

```
main()
    .
    .
    jal $2
    add $t1, $t2, $t3
    .
    .
$2  .
    .
    jr $ra
```



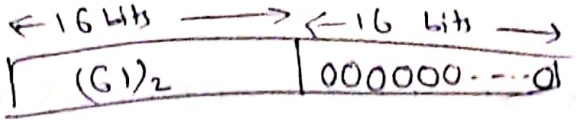
$sp → points to the last element inserted in stack.

*



* We cannot load a 32-bit constant in one instruction since the instruction itself is of 32 bits.

* lui $s0, 61



* ori $s0, $s0, 2304 (decimal number)

↓

bitwise or of $s0, with $(2304)_2$ and store the result in $s0

* ori $s0, $s1, 0x2121

↓

0x is written to tell the assembler, it is hex notation.

* MIPS → belongs to RISC Architecture

Reduced Instruction Set Computers

* Response time (or) Latency

→ After start, the time after which results appear

* Throughput → Total work done per unit time

* Instructions per Clock Cycle (IPC):

→ The average # of instructions executed per clock cycle.

For processors with a single execution unit,

$|IPC|_{max} = 1$

In practice, IPC is always < 1

$$CPI = \frac{1}{IPC} \qquad CPI|_{ideal} = 1,$$

Clock cycles Per Instruction. $CPI|_{actual} > 1$

→ Goal of Computer architecture → IPC ↑ ≡ CPI ↓

* CCT ← clock cycle Time

IC ← No. of instructions in the program

Then, CPU-time = ~~IC x CCT x CPI~~

= IC x CPI x CCT

* MIPS → another MIPS.

(Millions of Instructions per Second)

$$= \frac{\#\text{ of instructions executed}}{\text{Ex}|_{CPU} * 10^6}$$

Ex|CPU ↙ execution time of CPU

not used

Since, even if both # instructions & Ex|CPU increases MIPS remains same

* ¢ → clock

* Amdahl's Law ("Law of Diminishing Returns"):

Time taken to execute a program using

one CPU: T_1

" " " same " "

'n' CPUs : T_n

Then, speed up = $\frac{T_1}{T_n}$

For one processor situation, let

Time taken to execute parallelizable part : P

(They can be run in parallel)

" " " non-parallelizable part : S

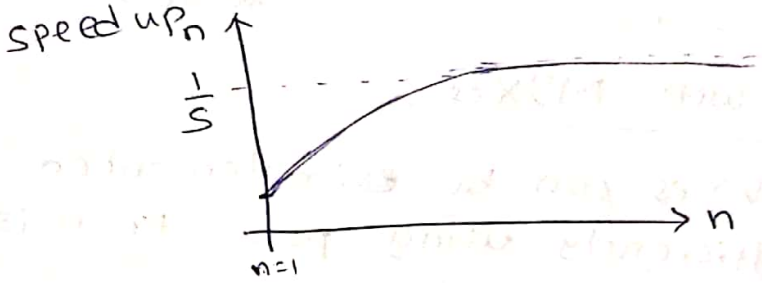
* $S+P = 1$ (normalized)

$\therefore T_1 = P+S = 1$

$T_n = \frac{P}{n} + S$

$(\text{Speed up})_n = \frac{1}{\frac{P}{n}+S} = \frac{1}{S+\frac{P}{n}}$ when $P+S=1$ (normalised)

If $n \to \infty$ $\text{Speedup}_n = \frac{1}{S}$



* SATA (Serial) Advanced Technology Attachment

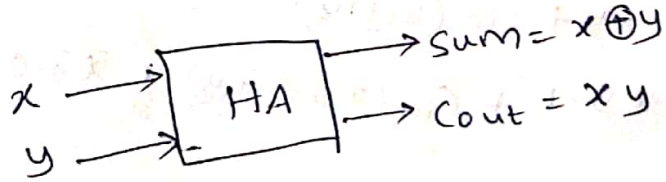
↓

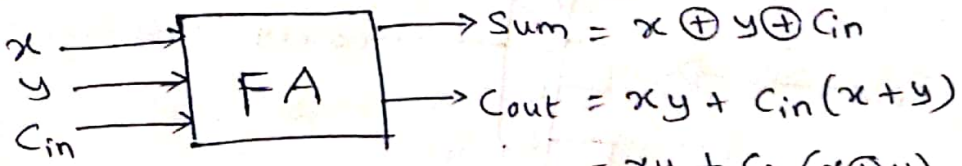
a fuzzy guide to the controller's instruction set in Hard disk

* SCSI → Small computer System Info.

pronounced as skuzy (or) Iscazee

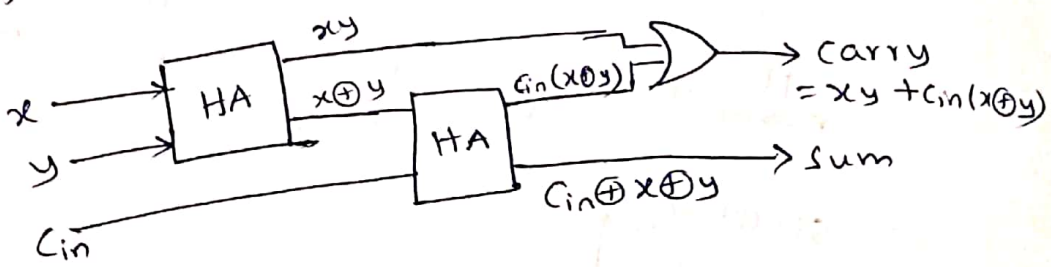
* Binary (Integer) Addition:





xy	x+y	x⊕y
00	0	0
01	1	1
10	1	1
11	1	0

If we use $Cout = xy + C_{in}(x \oplus y)$,



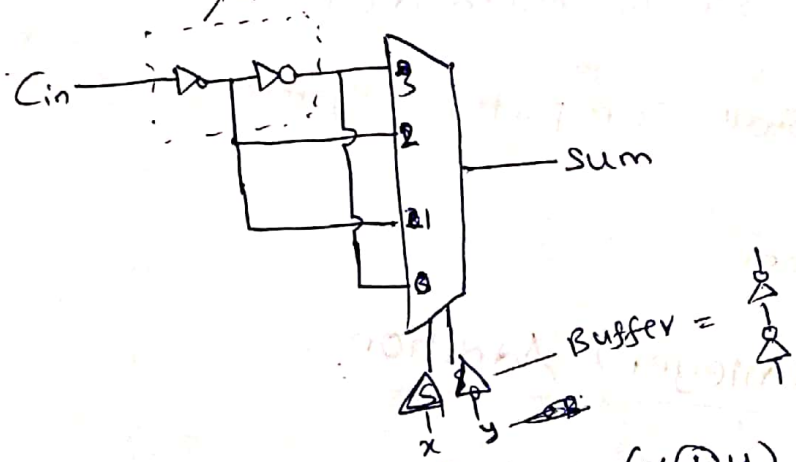
* FA with MUXes

→ MUXes can be implemented very efficiently using pass transistor logic (CMOS circuits).

$$Sum = (xy)C_{in} + (x\bar{y})\bar{C_{in}} + (\bar{x}y)\bar{C}_{in} + (\bar{x}\bar{y})C_{in}$$

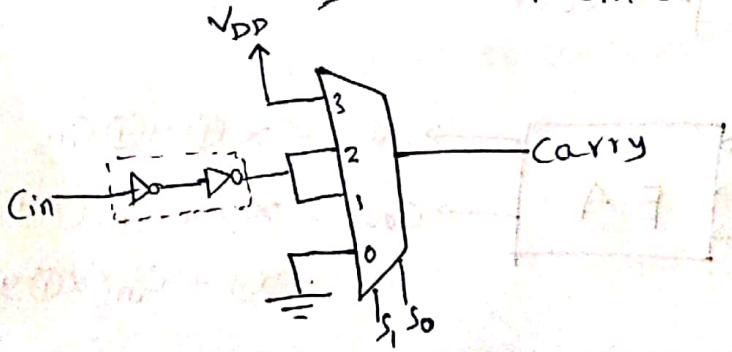
S_1 S_0

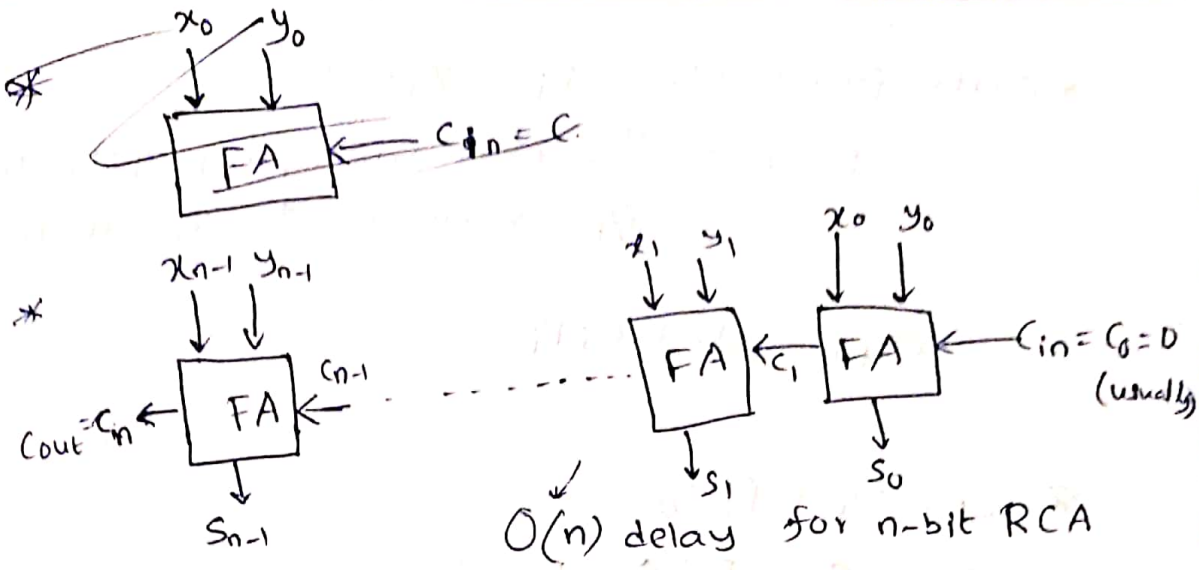
optional for non-pass transistor style.



$$Carry = xy + C_{in}(x \oplus y)$$

$$= xy \cdot 1 + C_{in}(x\bar{y} + \bar{x}y) + \bar{x}\bar{y} \cdot 0$$





$O(n)$ delay for n-bit RCA

* **overflow bits**

→ For 2's complement number addition, usually Cout is discarded. However, ~~cout~~ Cout is useful in detecting overflows.

* **Overflow**

→ Two numbers of same sign are added but the result is of opposite sign.

x_{n-1} . . . . . x_1 x_0

A 0 x x x x x

B 0 x x x x x

① . . . . . .

$OV \overset{def}{=} x_{n-1} y_{n-1} \bar{S}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} S_{n-1}$

$= C_{n-1} \oplus C_n$ (Commonly) → Homework

* **Carry Lookahead Adder (CLA):-**

→ Idea: Unroll c_i recurrence relation to remove dependency on $c_1, c_2, \ldots c_{i-1}$

→ Ideally, leads to $O(1)$ delay addition of two n-bit numbers.
But not feasible practically.

→ Carry generate: $g_i \overset{def}{=} x_i y_i$.

$g_i = 1 \Rightarrow$ i-th stage generated a carry of its own.

→ carry propagate: $P_i = x_i \oplus y_i$

$P_i = 1 \Rightarrow$ i^{th}-stage will propagate the i/p carry c_i to $(i+1)$-th stage.

$$c_{i+1} = g_i + c_i P_i$$

* $c_1 = g_1 + P_1 c_0$

$c_2 = g_2 + P_2 (g_1 + P_1 c_0)$

$$c_4 = g_4 + P_4 [g_3 + P_3 \{g_2 + P_2 (g_1 + P_1 c_0)\}]$$

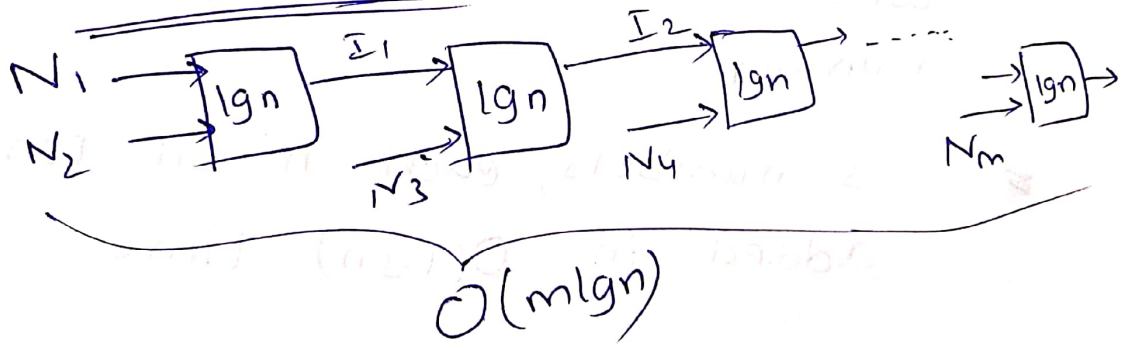
* carry absorb, $a_i = \bar{x}_i \bar{y}_i$

* Carry-save Adders :

→ Goal → assuming I have a fast carry lookahead adder with logarithmic delay, which I call "Logarithmic carry (Look ahead) Adder" (LCA)

gives $O(\lg n)$ delay for n-bit addition.

→ I will add 'm' numbers of n-bits in time $O(\lg m + \lg n)$.



x :	1	2	3	4	5
y :	3	8	1	7	2
+ z :	2	0	5	8	7
S :	6	0	9	9	4

Not considering carry

x :	1	2	3	4	5
y :	3	8	1	7	2
z :	2	0	5	8	7
C :	1	0	1	1	0 ← c_{in}

$\therefore$ Sum = S + C.

$S_i = (x_i + y_i + z_i) \% 10$

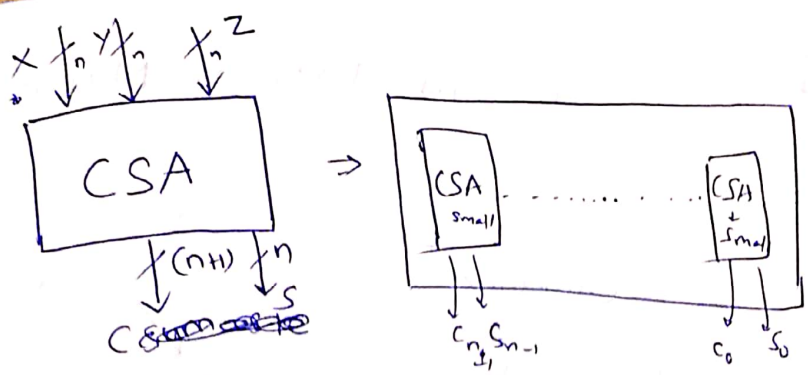
$C_{i+1} = (x_i + y_i + z_i) / 10$

($\%2$, $/2$ for binary)

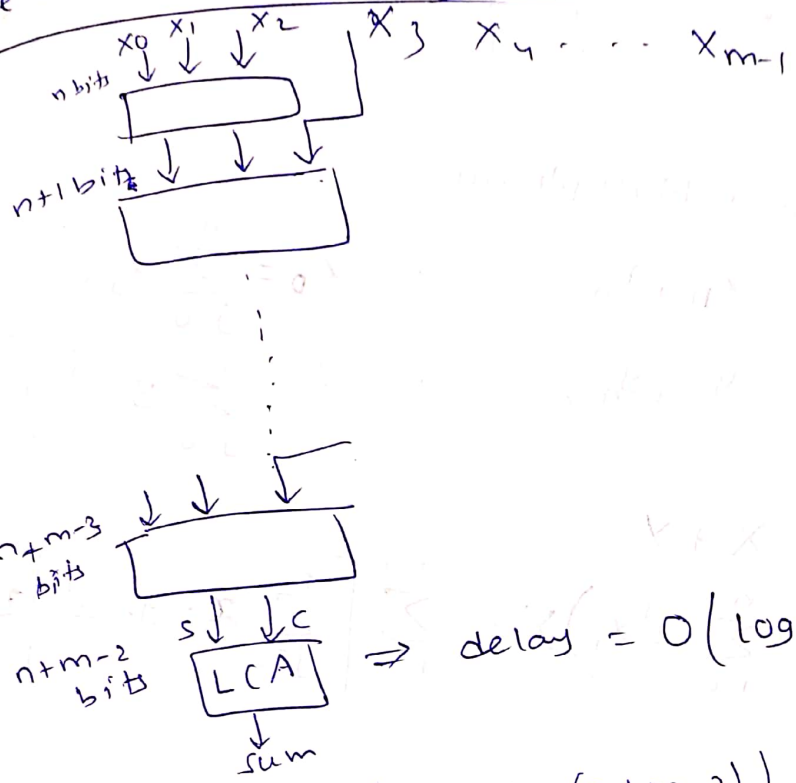
* {C, S} generation of 3 binary numbers with any no. of bits can be done in $O(1)$ time
  - delay of one full adder.

* C + S = Sum can happen in $O(\lg n)$ time where 'n' is the no. of bits in each number.

● ⇒ 3 numbers, each 'n' bit long can be added in $O(\lg n)$ time.



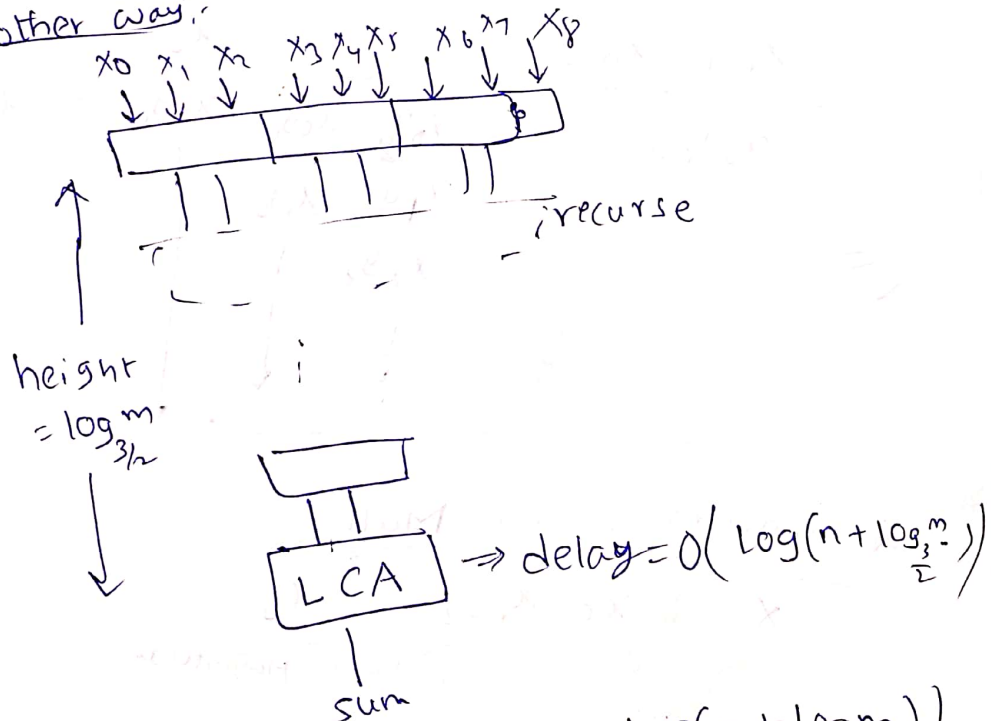
* One way of implementation :-



$\Rightarrow$ delay $= O(\log(n+m-2))$

Total delay $= O(m + \log(n+m-2))$

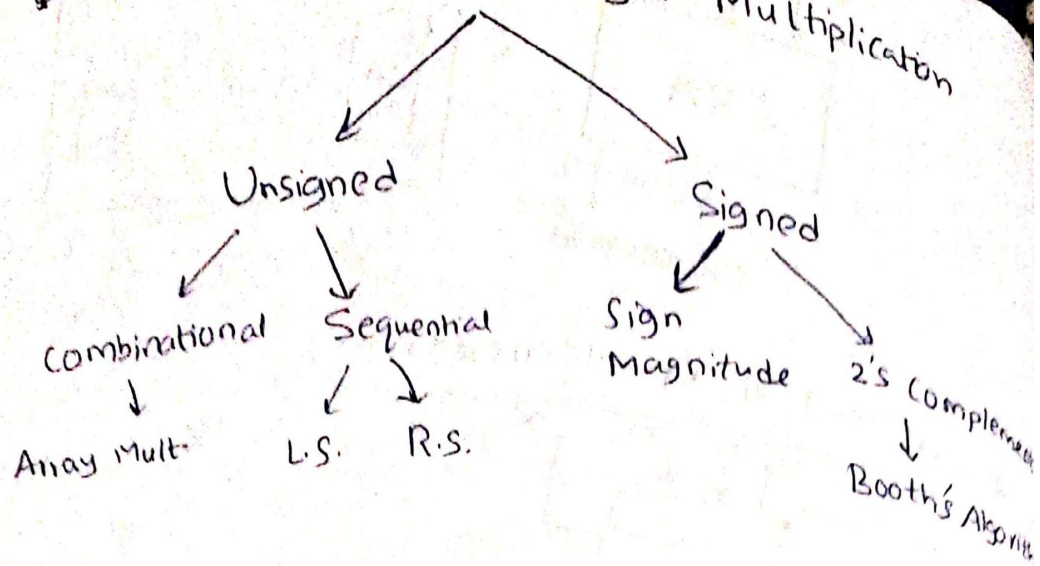
* Another way :-



$\Rightarrow$ delay $= O(\log(n + \log_{\frac{3}{2}} m))$

$\therefore$ Total delay $= O(\log m + \log(n + \log m))$

Binary Integer Multiplication



* Unsigned Multiplication

$$X = x_{n-1}x_{n-2}\cdots\cdots x_0 = \sum_{j=0}^{n-1} x_j 2^j$$

$$Y = y_{n-1}y_{n-2}\cdots\cdots y_0 = \sum_{j=0}^{n-1} y_j 2^j \quad \left.\right\} x_j, y_j \in \{0,1\}$$

$$P = X * Y$$

$$= \left(\sum_{i=0}^{n-1} 2^i x_i\right) Y = \sum_{i=0}^{n-1} \left[\sum_{j=0}^{n-1} x_i y_j 2^j\right] 2^i$$

$$\begin{array}{l} y_{n-1}\ y_{n-2}\ \cdots\cdots\ y_0 \\ x_{n-1}\ x_{n-2}\ \cdots\cdots\ x_0 \\ \hline x_0 y_{n-1}\ \cdots\ x_0 y_2\ \ x_0 y_1\ \ x_0 y_0 \Rightarrow x_0\ 2^0 \\ x_1 y_2 \quad x_1 y_0 \\ x_2 y_0 \end{array}$$

* Sign magnitude Mult.

$$X = x_s\ \underbrace{x_{n-2}\cdots x_0}_{(n-1)\text{ bit Magnitude}}$$

x_s : sign bit

$$Y = y_s\ y_{n-2}\cdots y_0$$

$P = XY$

$P = \{P_S, P_M\}$ → Concatenate

$P_S = x_S \oplus y_S$

$P_M = x_M y_M$

magnitude

* Unsigned sequential Multiplication

1. Left-shift version

~~In i th iteration~~

In $(i+1)^{th}$ iteration, calculate

$$P_{i+1} = P_i + x_i 2^i Y \quad , P_0 = 0$$

then $P_n = P$ ----> Prove

2. Right-shift scheme

$$P_0 = 0, \quad P_{i+1} = \left(P_i + x_i Y 2^n\right) 2^{-1}$$

with $P_n = P$ ---> Prove

x_0 terms: LS by n, RS by n => zero shift

x_1 terms: LS by n, RS. by n-1 => 1 shift

$P = XY$

$P = \{P_S, P_M\}$ → concatenate

$P_S = x_S \oplus y_S$

$P_M = X_M Y_M$

magnitude

* Unsigned sequential Multiplication

1. Left-shift Version

~~In i-th iteration~~

In $(i+1)^{th}$ iteration, calculate

$$P_{i+1} = P_i + x_i 2^i Y \quad , P_0 = 0$$

then $P_n = P$ ----→ Prove

2. Right-shift scheme

$$P_0 = 0, \quad P_{i+1} = \left(P_i + x_i Y 2^n\right) 2^{-1}$$

with $P_n = P$ ---→ Prove

x_0 terms: LS by n, RS by n ⇒ zero shift

x_1 terms: LS by n, RS. by n-1 ⇒ 1 shift

* Signed (2's complement) Integer Mult. (sequential):-

→ Booth's multiplication Algo (1951):-

→ Main idea is to reduce the no. of addition/subtraction steps needed to be performed in n-bit multiplication

↓ useful in circuits that cannot perform add/sub & shift in the same clock cycle

↓

In ckts, where this can be done, B.M. does not save anything.

→ Skips over range of 0's or 1's.

→ Uses right shifts

→ $n * n = n^2$ multiplication ⇒ n clock cycles.

Let,

multiplier $X = x_{n-1} x_{n-2} \cdots x_0$

$= -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$

Multiplicand $Y = y_{n-1} y_{n-2} \cdots y_0$

both X, Y in 2's complement form

Consider bit strings in X

case (i) :- $X^* = x_i x_{i-1} x_{i-2} \cdots x_{i-k-1}$

Substring of X of length 'k+2'

$= 0\,1\,1\,1\,1 \cdots 1\,0$ (x_i … x_{i-k-1})

k 1's flanked by two zeroes

case (ii) : $X^* = x_i x_{i-1} x_{i-2} \cdots x_{i-k-1}$

$= 1\,0\,0\,0\,0 \cdots 0\,1$

k 0's flanked by two 1's.

* Booth's multiplication Rules

1. If $x_j x_{j-1} = 01$, add Y to partial product & Right shift with sign extension

$= 10$, Sub Y from partial product (i.e add $-Y$) & Right shift with sign extension

$= 00$ or 11, only Right shift.

We need to justify the corrections of the rules for case-(i) and case(ii).

Case(i):-

Contribution of X^* in the product

$$= \sum_{j=i-k}^{i-1} 2^j \cdot y$$

According to Booth's rules, contribution of X^* to the product is

$$= 2^i \cdot y - 2^{i-k} \cdot y$$

$$= 2^{i-k}(2^k - 1) y$$

$$= 2^{i-k} \left(\sum_{j=0}^{k-1} 2^j \right) y$$

$$= \left(\sum_{j=0}^{k-1} (2^{j+i-k}) \right) y = \sum_{j=i-k}^{i-1} 2^j y$$

Case(ii):-

contribution of X^* in the product

$$= -2^i y + 2^{i-k-1} y$$

$$= (-2^i + 2^{i-k-1}) y$$

According to Booth's rules, contribution of X^* to the product is

$$= -2^i y + 2^{i-k-1} y = (-2^i + 2^{i-k-1}) y$$

Examples:-

4-bit representation

$X = -5 = 1011$, $Y = 0111 = 7 \Rightarrow -Y = 1001$

Accumulator → n-bit register

Q → (n+1)-bit register.

Step No.	Action	Partial product: Accumulator (A)	Q	Q[-1]
(init) 0	init	0000	10116) = {x, 1'b0}	
1	sub y, from A & RS	1001 → 1100	10110 → 11011	
2	RS	1110	01101	
3	A+y RS	0101 → 0010	01101 → 10110	
4	A-y RS	1011 → 1101	10110 → 11011	1

$P = X * Y = \{A, Q[n+1:1]\}$

$= 1101\ 1101$

$-P = 0010\ 0011 = (23)_{16} = 35$

$\therefore \boxed{P = -35}$

$*X = 2 = 0010, \quad Y = 6 = 0110 \Rightarrow -Y = 1010$

Step No.	Action	Accumulator (A)	Q
0	init	0000	00100
1	RS	0000	00010
2	A-y RS	1010 → 1101	00010 → 00001
3	A+y RS	0011 → 0001	00001 → 10000
4	RS	0000	11000

$\therefore P = X * Y = 0000\ 1100 = 12$

* $X = -77 = 1011\ 0011 \quad Y = -43 = 1101\ 0101$

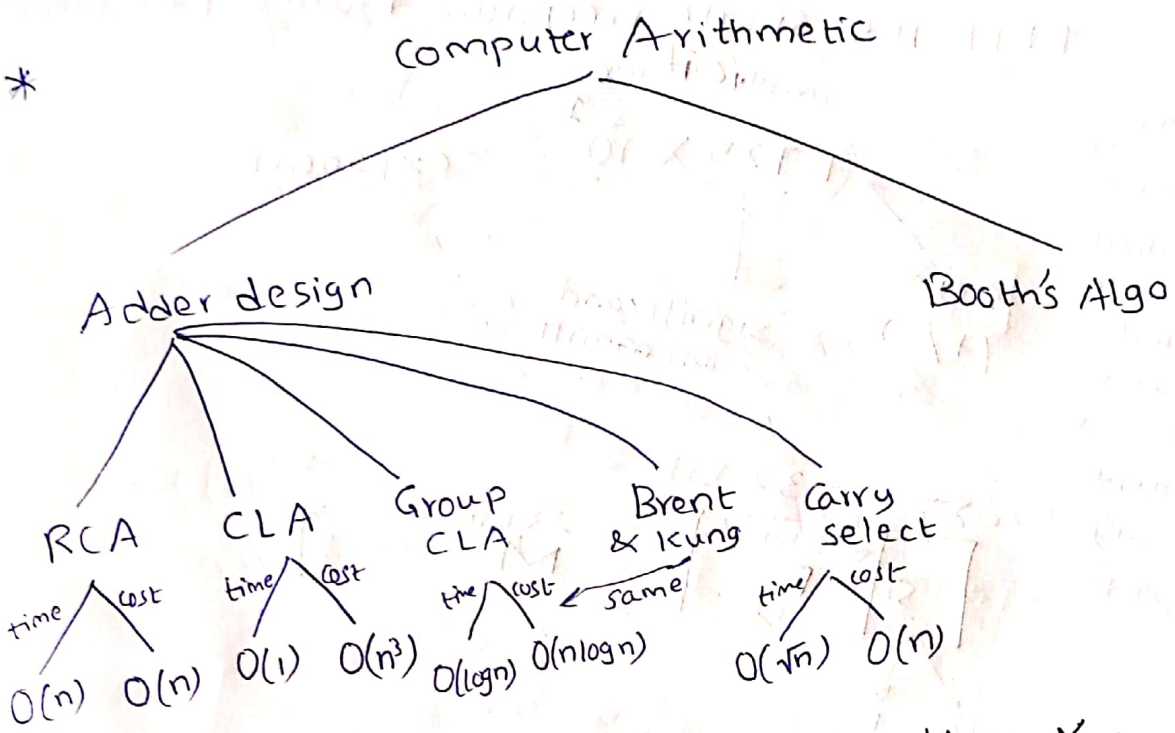
$P = 3311$

$= 0000\ 1100\ 1110\ 1111$

→ Quiz-2 Syllabus

1. Computer Arithmetic
2. Hard disk performance

*



Q. Let the content of memory Location X in MIPS architecture be

0x8FEFC000 . Who am I?

In binary, 1000 1111 1110 1111 1100 0000 0000 0000

X => −1880,113,152 ← 2's complement number

X => 2,414,854,144 ← unsigned number

X => Lw $t7 , -16384($ra)

$t7 → Reg. 15

$ra → Reg. 31

Lw → op code is 1000 11

$X \Rightarrow (1.873 \times 2^{-96})_{10}$ (In ~~fo~~ floating point)

* Representing real numbers in binary form

N may be very large $+93 \times 10^{32}$

N maybe very small -93×10^{-32}

$N = 17.25$

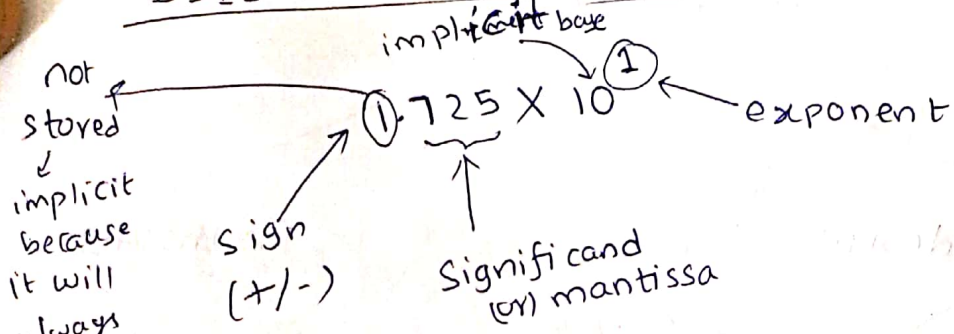
$= 172.5 \times 10^{-1}$

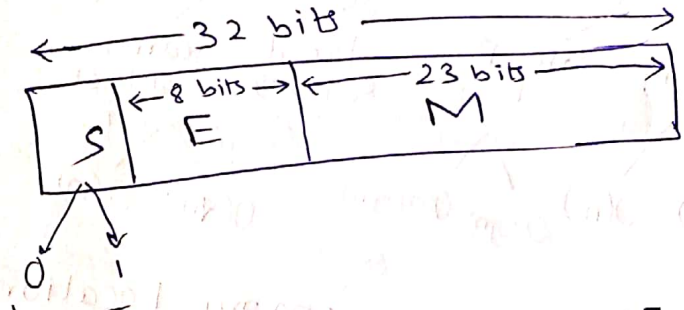
$= 1725 \times 10^{-2}$

$= 1.725 \times 10^{1}$ ✓

$= .1725 \times 10^{2}$

IEEE 754 Single precision Floating point format





$\therefore N = (-1)^S \, 1.M \times 2^{E-127}$ ↖ biased excess-127 coding

In FP-arithmetic, no role of 2's complement

Q. Which decimal number is represented by:

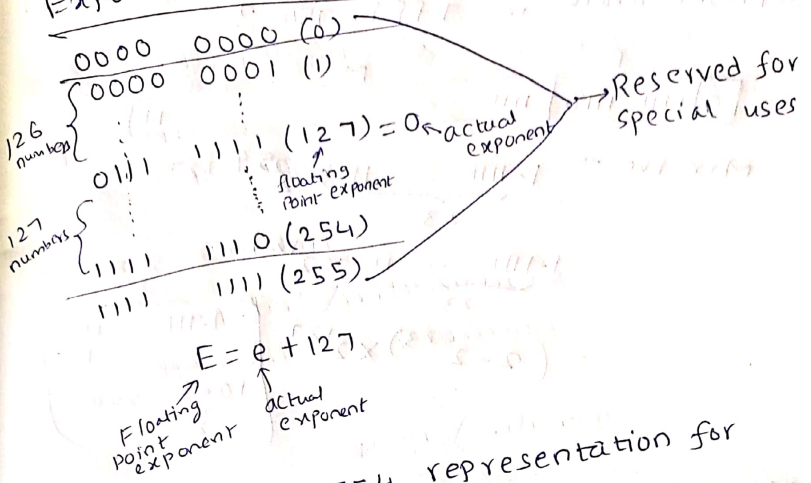
N =	1	1000 0001	01000...00

&

$N = -1.01 \times 2^{129-127} = -1.01 \times 2^{2}$

$N = -101 = -5$

Exponent 'E' => 8 bits



$E = e + 127$

Floating point exponent ↑ actual exponent

Q. Derive IEEE 754 representation for
$N = -575 \times 10^{-2}$

Steps

$N = -5.75$

Convert to binary

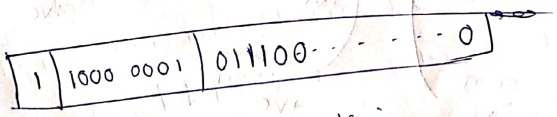
$N = -101.11$

Normalize =>

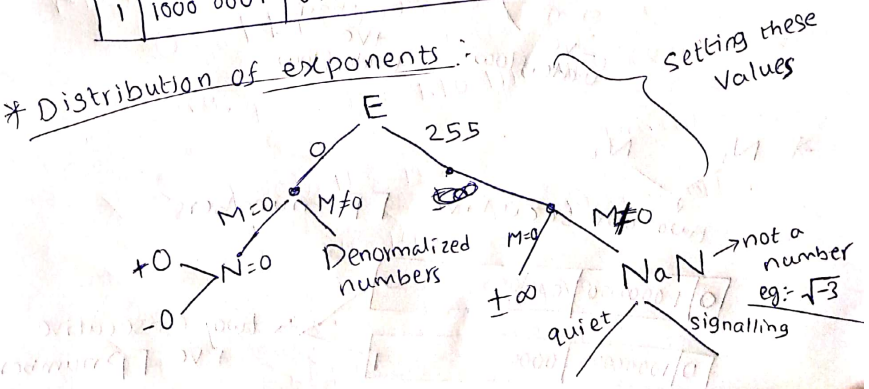
$N = -1.0111 \times 2^2$

FP exponent $E = 2 + 127 = 129$

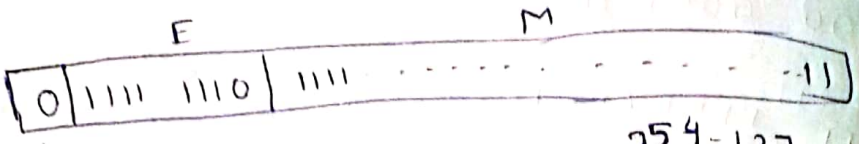
$\Rightarrow E = 1000\ 0001$



* Distribution of exponents :-



Q. What is the largest positive normalized number representable in IEEE 754 ?



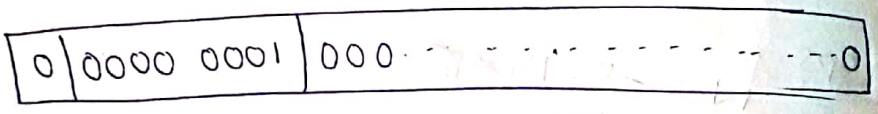
Max. +ve $\Rightarrow 1.\underbrace{1111\ldots11}_{23} \times 2^{254-127}$

$= 1.\underbrace{1111\ldots11}_{23} \times 2^{127}$

$= (2 - 2^{-23}) \times 2^{127}$

$\left(\because 1.\overbrace{111\ldots11}^{23} + 1 = 10.0\ldots000 = 2\right)$

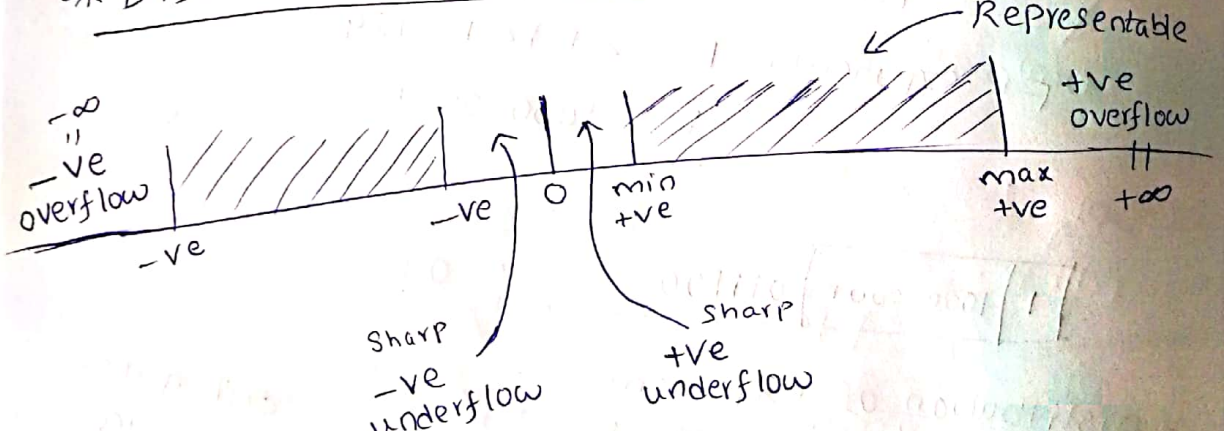
Min. +ve



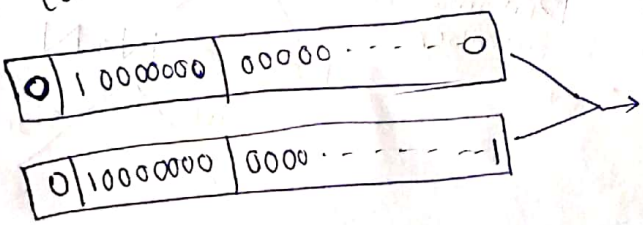
Min. +ve $\Rightarrow +1.0 \times 2^{1-127}$

$= 2^{-126}$

* Distribution of FP numbers on Real line



* N_1 N_2 two consecutive FP numbers



Because of the exponent, the separation b/w consecutive FP #'s increases.