08-02-18

UNIT-I : BASIC STRUCTURE OF COMPUTERS, ARITH

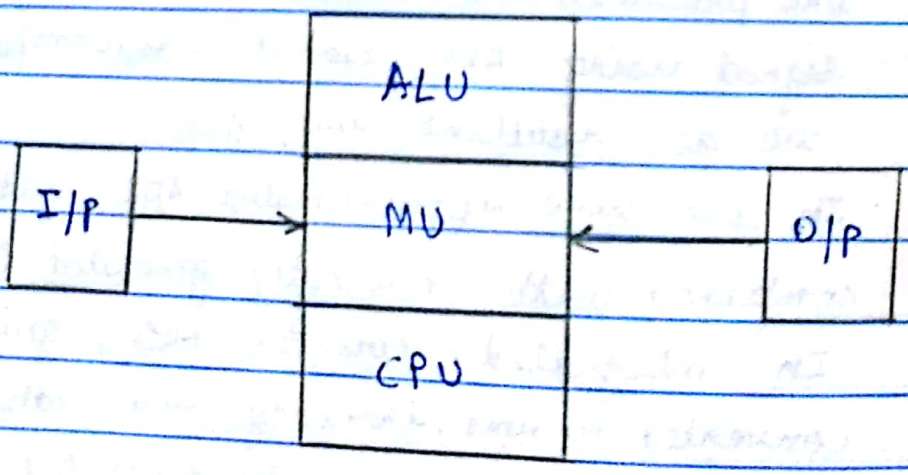
Computer Types

↳ Personal Computers, Desktop Computers, Laptops, Handhelds, Cell Phones, Work Stations, PDS, Super Computers

↳ So many computers developed for portability, Less size, more speed

Functional Units

↳ Input Unit
↳ Output Unit
↳ Central Processing Unit
↳ Arithmetic and Logic Unit
↳ Memory Unit

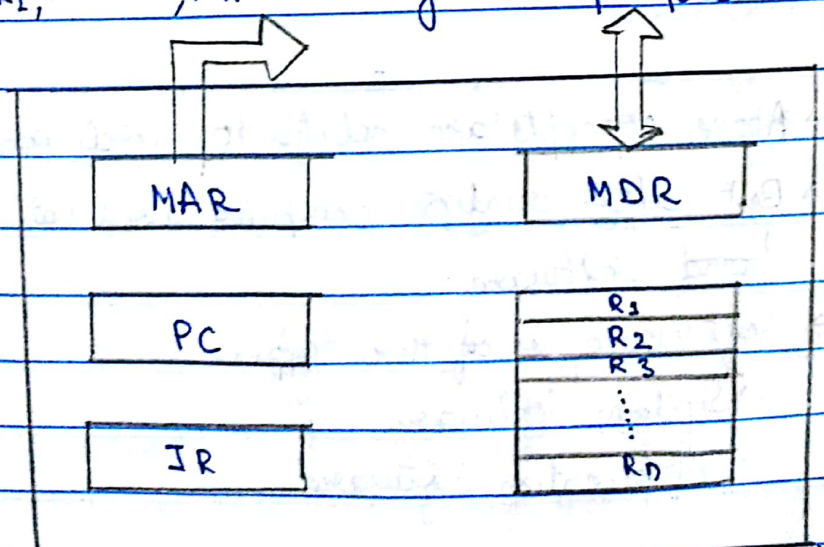


Basic Operational Concepts

i) Moving Operations
ii) General Purpose Register - To store results temporarily
iii) Accumulator - To perform arithmetic and logical operations

Eg. MOV R_d, R_s or MOV R_s, R_d ADD R_d, R_s or ADD R_s, R_d

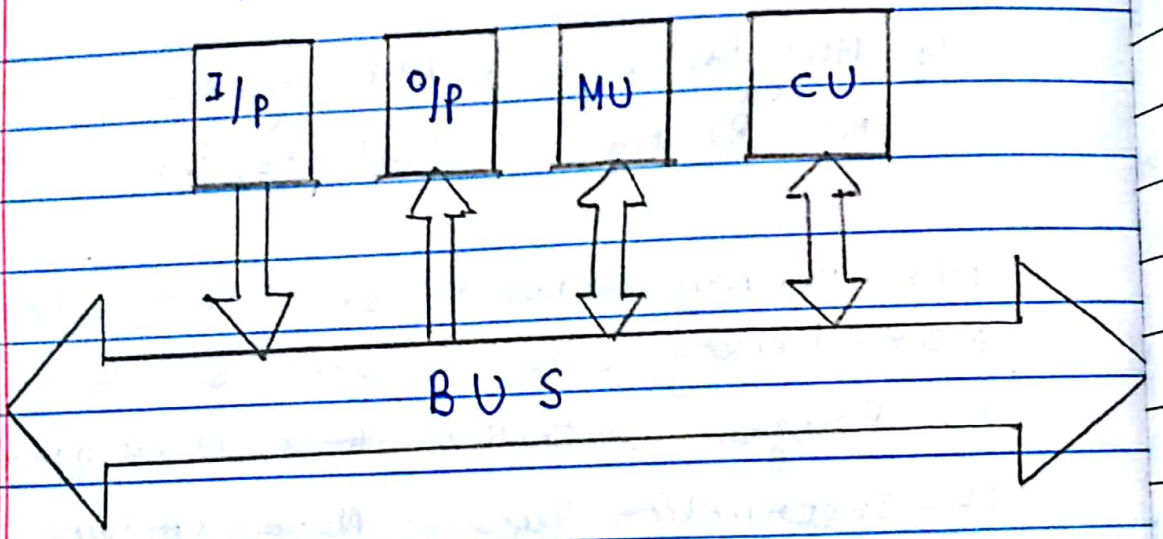
MAR - Memory Address Register (Unidirectional)
MDR - Memory Data Register (Bidirectional)
PC - Program Controller - Stores address of next instruction
IR - Instruction Register - Always contains the current instruction that is being executed
$R_1, R_2, \ldots, R_n$ - 'n' general purpose registers



Equation to find Time → $T = \frac{N \times S}{R}$ where (Basic Performance Equation) → units = sec
N = No. of instructions
S = Avg. no. of steps to execute 1 inst.
R = Clock Rate (1/2 or HZ)

BUS Structures

→ In 8085, we are using 8 files to transfer 8 bit data parallelly. So, 8085 has 8-bit BU
→ Group of files / lines used to transfer data parallelly is called BUS



Software

→ Above concepts are related to hardware.
→ But after creation, compiles does not work wit hard software.
→ Software is of two types:
i) System Software
ii) Application Software

Software

Computer Software or Software is a part of computer system that consists of data or computer instructions, in contrast to the physical hardware from which the system is built.

Software is of ~~two~~ three types:

i) Application Software - which is software that uses the computer system to perform special functions or provide entertainment functions beyond the basic operation of the computer itself. There are many different types of application software, because the range of tasks that can be performed with a modern computer is so large.

ii) System Software - which is software that directly operates the computer hardware, to provide basic functionality needed by users and other software, and to provide a platform for running application software. System software includes:

→ Operating Systems → Device Drivers → Utilities

iii) Malicious Software (or) Malware - which is software that is developed to harm and disrupt computers. As such, malware is undesirable. Malware is closely associated with computer related crimes, though some malicious programs may have been designed as practical jokes.

Types of Computers

A computer is a machine that can be programmed manipulate symbols. Its principal characteristics are
→ It responds to a specific set of instructions in a well
→ It can execute a prerecorded list of instructions (a pr
→ It can quickly store and retrieve large amounts of d

Computers are generally classified by size and pow
→ Personal Computer → ~~Microprocessor~~ Micro computer → Workstation
→ Minicomputer → Mainframe → Super Computer

Personal Computers are further classified as:
→ Tower Model → Desktop Model → Notebook Computer
→ Laptop Computer → Subnotebook Computer
→ Hand-held computer → Palmtop → Personal Digital Assis

P.T.O.

Performance

→ Initially only registers and secondary memory were available, but later on, 2 others were added.

→ Register → Cache → Primary → Secondary

→ Cache memory - Smallest memory placed in between processor and main memory, to improve the performance, by placing the data which is used regularly in it.

→ System Performance Evaluation Corporation
- Tests the computers and gives rating

$$\text{SPEC rating} = \frac{\text{Running Time of A Computer Under Test}}{\text{Running Time of A Reference Computer}}$$

- Check this for 'n' no. of problems.

- $\left(\prod_{i=1}^{n} \text{SPEC rating}\right)^{1/n}$

→ Throughput = Work Donce
= No. of instructions executed in given unit time

→ We must increase throughput to improve the performance of a computer

Multiprocessor Vs MultiComputer

→ Using one computer with one processor, there are some problems which we cannot solve.

→ Multiprocessor
- Two or more processors in a single computer

→ Multi Computer
- Two or more computers connected in a network

→ Multiprocessor
- Shared Memory
- Only one computer, only one memory, shared to all processors to solve a given problem.

→ Multi Computer
- Message Processing - to solve large problems
- Different computers communicate with each other by sending messages, to solve a task

→ In general, Multi-Computer System has more advantage over Multi-Processor System.

→ More complex and costly to develop Multi Processor System.

→ Using Multi Computer System, we can solve the same problem. It is Cheaper. It is less complex.

9394225111

Historical Perspective

→ It is related to generations:

1st Generation: Vaccum Tubes
2nd Generation: Silicon Chips
3rd Generation: Transistors
4th Generation: Integrated Circuits (IC),
5th Generation: Very Large Scale Integrated Circuits (VLSI)
Quardcore Processors,
Core Processors (I_3, I_7)

02-11

Programs

→ DAA - Decimal adjustment accumulator

1 byte instruction

Used to perform BCD addition only

6+4 = A => Hexa

6+4 = 10 => BCD addition

Write DAA after performing addition, adjusts to Hexa

If no < 9 after addition, same number is given, else, 06 is added and stored.

Eg. BCD => 6 + 4 = 10

Eg. Hexa => 6 + 4 = A => 0000 1010 = A

0000 0110 = 6

0001 0000

=10

1. Stack

↳ i) Write a program to clear the flags (all flags = 0)

ii) Move data (00) into accumulator and check if flags a

iii) Perform a logical operation and check the flag status (ORA A)

→ i) PSW - Program Status Word

Combination of two registers, A and F

PUSH PSW → Pushes data in ACC and Flags to stack

POP PSW → Pops data from stack to ACC and flags

Move 00 to L, don't worry about value in H

Push H to stack, POP to PSW from stack to clear the flags.

ii) Move data 00 to accumulator ~~it~~ then write PUSH PSW to check status of flags, then pop to HL and display L value. (O/P: 00)

iii) Perform ORA A with 0 and accumulator, then PUSH PSW to check status of flags, then pop to HL and display L value. (O/P: 40)

P.T.O.

→

2. Write a program to convert a given BCD number into Hexa number.

→ BCD => 19 Hexa => 13 19 (BCD) → (13)H (Hexa)

BCD => 43 Hexa => 2B 43 (BCD) → (2B)H (Hexa)

Using unpacking, we must split the number into 2 digits. We must multiply MSB 10 times. We must add LSB to it. Then, we get Hexa number.

Eg. 1×10 = A+9 = 13 => 0000 1010 => A

0000 1001 => 9

0001 0011 => 13

3. Write a program to convert a given Hexa number into BCD number.

→ Hexa => FF BCD => 1111 1111 Decimal = 255

For given two values, we need to display 3 values

(11)H → 017 (FF)H → 017 (01)H → 001.

We must use 3 memory locations to display output.

If given no. is >100 (64H), perform subtraction by 100 till it is less than 100, then perform subtraction by 10 (0AH) till it is less than 10, display BCD 1, BCD 2, BCD 3

Eg. (FF)H → 255 (BCD3 BCD2 BCD1)

>100

-100 => count = BCD 1

>10

-10 => count = BCD 2

Remainder => BCD 3

100)255(2 → BCD 3

-200

55

10)55(5 → BCD 2

-50

5 → BCD 1

→

4. Write a program to convert a given ASCII number to Hexa Number

→

	Decimal	Hexa
A	65	41
B	66	42
⋮		
F		
0	48	30
1	49	31
⋮		
9		

We give i/p only single values 00-0A, 01-09 If the value is less than 10, (given ASCII value) by adding 30, we get Hexa number, but if it is greater than 10, we add 30+7 to ASCII value to get Hexa value.

5. Write a program to convert a given Hexa number to ASCII number

→ We give i/p Hexa value and subtract 30 from it for 01 to 10 numbers, but if the number is beyond those, we must subtract 30 and 7 also from given Hexa value to get ASCII value

12-02-'17

Arithmetic

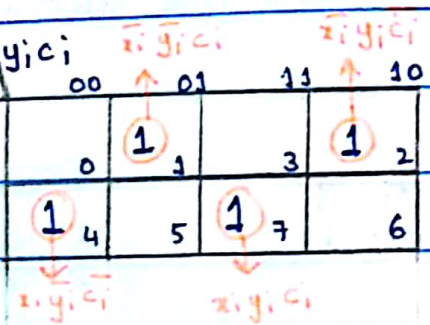
1. Addition / Subtraction of Signed Numbers

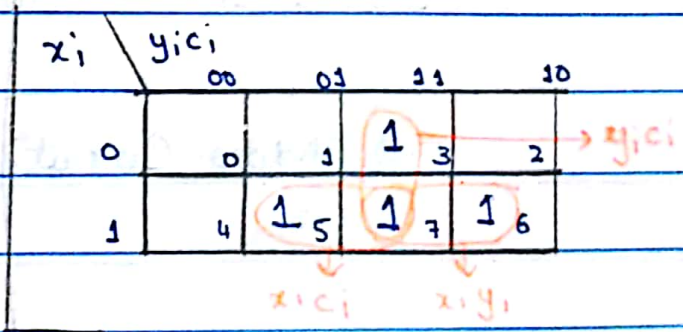
For 2-bit addition:

0	0	1	1
0	1	0	1
0	1	1	0

3-bit addition:

x_i	y_i	carry$_{in}$ (c_{in})	Sum s_i	Carry out (C_{out}) $= c_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





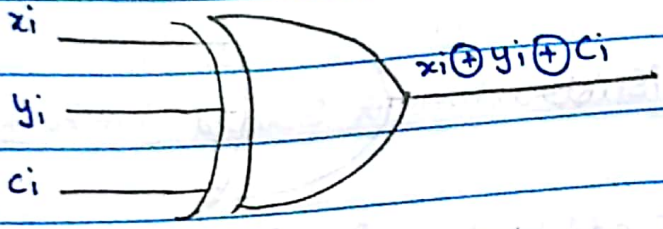
$S_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i$

$S_i = x_i \oplus y_i \oplus c_i$

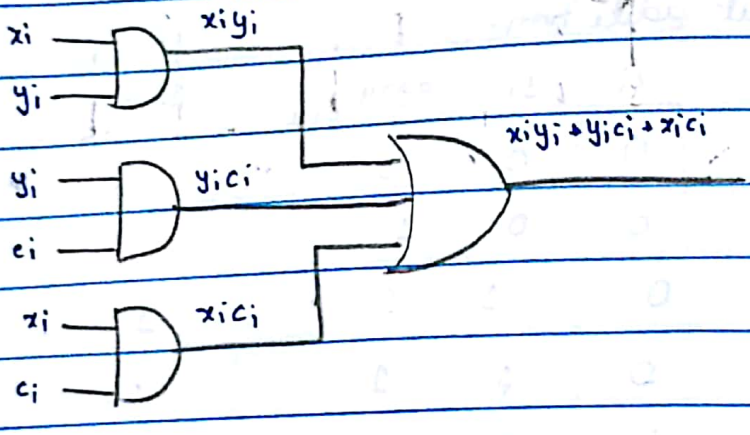
$C_{i+1} = y_i c_i + x_i c_i + x_i y_i$

$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$

S_i :

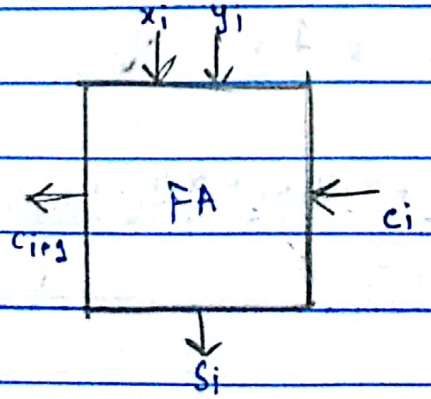


C_{i+1} :

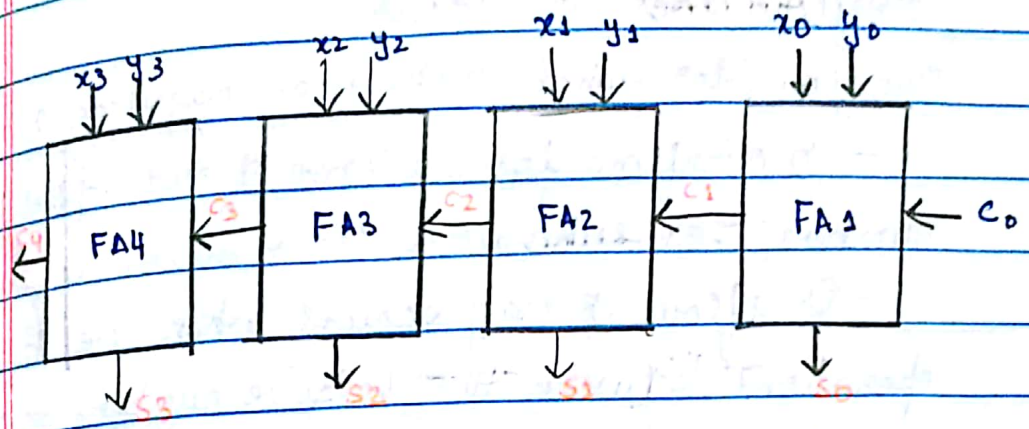


Half Adder (HA) takes two numbers as input and produces one sum bit and one carry bit as output.
Full Adder (FA) takes three numbers as input and produces one sum bit and one carry bit as output.

Full Adder Circuit:



4-bit Ripple Carry Adder / 4-bit Addition using 4 Full Adders / Ripple Carry Adder:

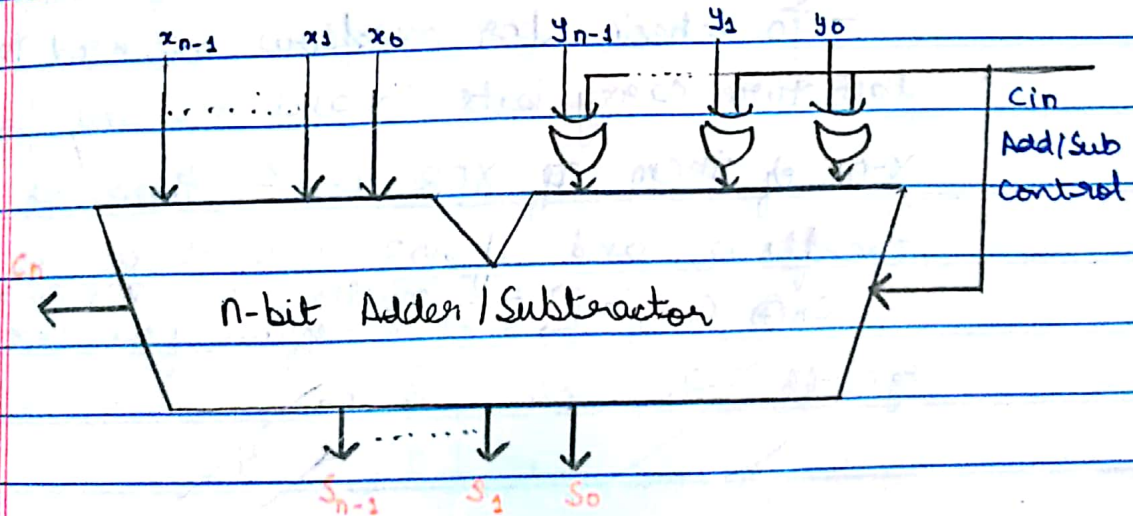


→ 4 full adders using cascading / cascaded together.

→ $C_0 = 0$ => Addition $C_0 = 1$ => Subtraction.

But, for subtraction, we must give $C_0 = 1$ and y input and complement these two for y-input. (y is complemented, and we add 1 to x => $x - y = x + \bar{y} + 1$)

General Addition / Subtraction Circuit (n-bit)



$c_{in} = 0$ => Add $c_{in} = 1$ => Sub

Overflow (for unsigned numbers (0-7 = magnitude) ∀ := range 0 to 255 00000000 to 11111111 or FF)

- While performing n-bit addition, if (n+1)th bit occurs, overflow occurs, if (n+1)th bit does not occur overflow does not occur.

Overflow (for signed numbers (0-6 = magnitude, 7 = MSB = sign = 1 = -ve, 0 = +ve)

- No overflow for addition of two opposite sign numbers, i.e., subtraction. No overflow in subtraction
- Overflow ∄ may occurs when we perform operations between two positive numbers or two negative ng

- Range of Signed numbers:

1 1 1 1 1 1 1 1 (-127) to 0 1 1 1 1 1 1 1 (+127)

Eg. 65 + 63 = 128 → overflow

Eg. 65 + 43 = 108 → no overflow

Eg. -66 - 66 = -132 → overflow

Eg. -66 - 26 = -92 → no overflow

- To check for overflow, we need to observe last two carry bits C_n and C_{n-1} and perform X-OR of them. If XOR is 1, there is an overflow, and if XOR is 0, there is no overflow

$C_n \oplus C_{n-1} = \begin{cases} 1 \Rightarrow \text{overflow (if } C_n, C_{n-1} \text{ are not same)} \\ 0 \Rightarrow \text{no overflow (if both } C_n, C_{n-1} \text{ are same)} \end{cases}$

Eg: -66 -66 16 66 2 4

1 100 0010
128 64 32 16 8 4 2 1 (64 + 2 = 66)

Eg. -66 - 66 = 1 100 0010
1 1 100 0010
1 0 0000100
↳ overflow

Eg. 65 + 63 = 0 1000001
0 0111111
1 0000000
different, overflow occurs

Eg. -66 - 66

+66 ⇒ 0100 0010
+66 1's ⇒ 1 0 11 11 0 1
-66 ⇒ 1 011 11 10
(2's)
-66 (unsigned) ⇒ 1100 0010
1 1111 111
0 1000 0000
different, overflow occurs

$x - y = x + \bar{y} + 1$

Fast Adder:

→ To generate c_1, we use 2 gate, c_2, 4 gates, c_3, 6 gates, c_n, $2n$ gates in ripple adder.

→ To generate s_{n-1}, we use $2n-1$ gates.

→ Time delay is $2n$ for carry.

8 bit addition = 16 delay.

→ Performance can be improved in 2 ways:

i) Using High End Circuits (fast AND, OR, XOR gates)

ii) Using Augmented Network Gate Structure

↳ circuit is comparitively big, but time delay is reduced, performance is increased.

→ Using Augmented Network Gate Structure - Fast Adder

$s_i = x_i \oplus y_i \oplus c_i$

$c_{i+1} = x_i y_i + y_i c_i + x_i c_i$

$= x_i y_i + (x_i + y_i) c_i$

$= G_i + P_i c_i$

G_i = Generate Function for stage 'i' = $x_i y_i$

P_i = Propogate Function for stage 'i' = $x_i + y_i$

→ When $x_i = 1$ and also $y_i = 1$, $G_i = 1$

→ $c_{i+1} = 1$ when $G_i = 1$, irrespective of c_i

→ Propogate function is going to produce output carry for input carry (when $c_i = 1$) if $x_i = 1$ or $y_i = 1$

→ Both G_i and P_i do not depend on carry, so we can generate them parallely, at a single delay. So, the delay is reduced, performance is increased.

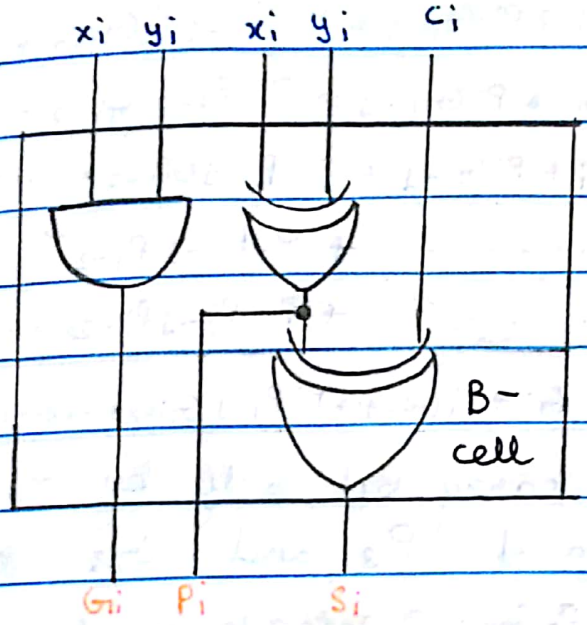
Basic structure:

$P_i = x_i + y_i = x_i \oplus y_i$

But for bot $x_i y_i = 1$

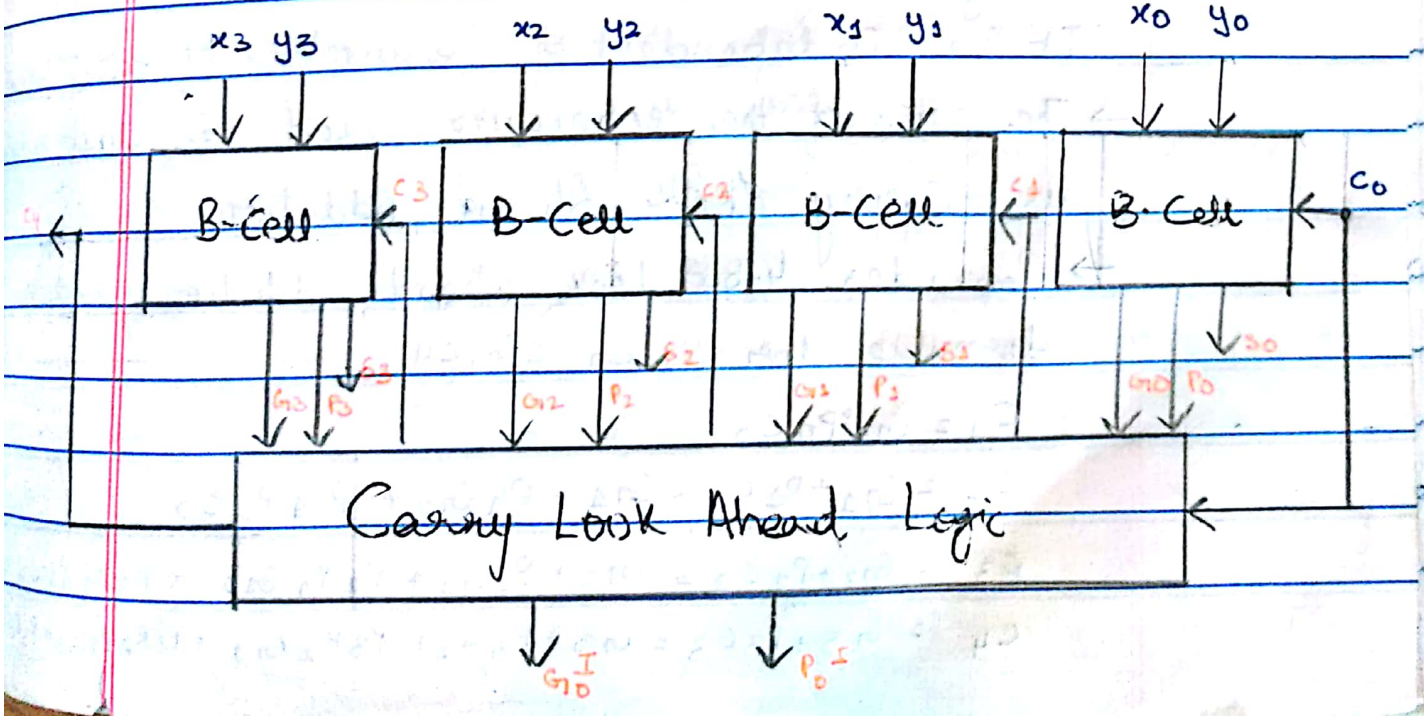
$G_i = 1$, so $c_{i+1} = 1$ irrespective of P_i => reversability of $x_i \oplus y_i$

$x_i + y_i$	$x_i \oplus y_i$
0 0 0	0 0 0
0 1 1	0 1 1
1 0 1	1 0 1
1 1 1	1 1 0



~~4-bit fast adder~~:

4-bit carry look ahead adder (CLA):



$c_i = G_{i-1} + P_{i-1} c_{i-1}$ $\quad c_{i-1} = G_{i-2} + P_{i-2} c_{i-2}$

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

$= G_i + P_i (G_{i-1} + P_{i-1} c_{i-1})$

$= G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$

$= G_i + P_i G_{i-1} + P_i P_{i-1} (G_{i-2} + P_{i-2} c_{i-2})$

$= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} c_{i-2}$

$= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots \cdots$

$\cdots \cdots + P_i P_{i-1} P_{i-2} \cdots G_0 + \cdots$

$\cdots \cdots + P_i P_{i-1} P_{i-2} \cdots P_i G_0 c_0$

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} P_{i-2} \cdots G_0 + P_i P_{i-1} P_{i-2} \cdots P_i G_0 c_0$$

→ We get carry bit only by multiplication and addition of P_s and G_s, so at (1+2+1 = 4)

1 gate (delay) = P_0, G_0; 2 gate (delays) = to generate carrys (AND + OR gates)

1 gate (delay) = to get all sums

So, by using 4 gate delays we get the result using carry look ahead addition.

It is independent of the number of gates used.

→ The name of the techniques used in fast adder is - Carry Look Ahead Addition

→ Consider 4-bit look ahead addition, write the formula for c_1, c_2, c_3, c_4.

$c_1 = G_0 + P_0 c_0$

$c_2 = G_1 + P_1 c_1 = G_1 + P_1 G_0 + P_1 P_0 c_0$

$c_3 = G_2 + P_2 c_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$

$c_4 = G_3 + P_3 c_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$

→ For 4-bit carry look ahead addition, we have 5 inputs to 1 AND gate: ($P_3 P_2 P_1 P_0 C_0$)

For 8-bit carry look ahead addition, we have many more constraints.

→ Fan-in constraint condition

- Maximum number of bits that can be given to a single gate.

- Maximum 5-input gate is easy to develop, so more than that, it is difficult to generate and more costly.

→ So, if we use group of given B-cells as 1 unit, we can use many units to generate higher bit carry look ahead addition, since we can perform only 4-bit carry look ahead addition by satisfying fan in constraint.

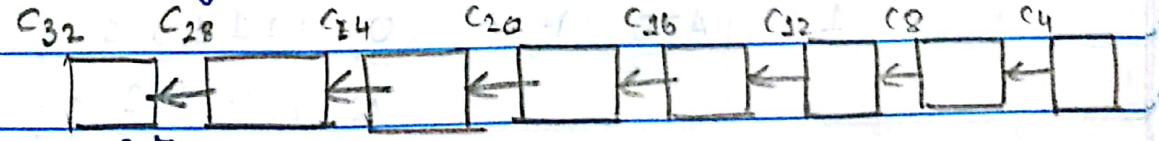
8-bit addition = 2× 4-bit addition

16-bit addition = 4× 4-bit addition

32-bit addition = 8 × 4-bit addition

→ 32 bit ripple adder — Final carry = 64 delays, Final Sum = 63 delays

32 bit fast adder — Final carry = (7×2) + 3 = 17 delays



C_4 = 3 gate delays (1+2, no sum)

$C_8, C_{16} \cdots$ = 2 gate delays ($P_i G_i$ already generated)

→ We may reduce the gate delays further by generating all carries $C_4, C_8, C_{12} \ldots$ parallelly. We can perform this by generating higher Order Generate and Propogate functions.

15-02-'18

Multiplication (unsigned number)

Unsigned Numbers

M ← 1101 ← 13 ← M ← Multiplicand

Q ← 1011 ← 11 ← Q ← Multiplier

Multiplicand and multiplier must have same no. of bits

$Q_n = 1$ => Add M & A, Shr CAQ

$= 0$ => Shr CAQ

Shr : Shift rigght CAQ : combined CAQ is left while Shr CAQ, LSB (Q_n) is lost, 0 is always placed at MSB, Sc : Serial counter : no. of bits

Q_n = last bit = lsb of Q, process Stops when Sc = 0

Q_n	Operation	C	A	Q (Q_n)	Sc
1		0	0000	1011	4
	ADD M		1101		
	~~Shr CAQ~~	0	1101	1011	
	Shr CAQ	0	0110	1101	3
1	ADD M	①0	1101		
		1	0011	1101	
	Shr CAQ	0	1001	1110	2

0	Shr CAQ	0	0100	1111	1
1	ADD M	①0	11101		
		1	0001	1111	
	Shr CAQ	0	1000	1111	0

Process stops

$10001111 = 143$, $11 \times 13 = 143$

Hence, the process is right

Eg: Multiplicand ← M ← 12 = 1100

Multiplier ← Q ← 9 = 1001

Qn	Operation	C	A	Q Qn ↓	Sc
1		0	0000	1001	4
	ADD M		1100		
		0	1100	1001	
	Shr CAQ	0	0110	0100	3
0	Shr CAQ	0	0011	0010	2
0	Shr CAQ	0	0001	1001	1

Qn	Operation	C	A	Q Qn	Sc
0		0	0001	1001	4-1
1	ADD M	0	1100		
		0	1101	1001	3
	Shr CAQ	0	0110	1100	

Process stops

0110 1100 = 108 , 12 × 9 = 108

Hence, the process is right

While performing Shr CAQ

→ lsb is lost

→ 0 is always placed at MSB

Eg. To find magnitude:

→ 11110 0 → $-2^4 = -2$

→ 10101 0 → $-2^4+2^3-2^2+2^1-2^0$ ± 1

$= -16+8-4+2-1 = -21+10 = -11$

→ 11011 0 → $-0+-2^3+2^2+0-2^0 = -8+4-1 = -5$

01010

01011

Signed Numbers Multiplication - Booth Algorithm

→ We must consider an extra register Q_{n+1} that can store one bit.

→ Q_n is the lsb of the multiplier.

→ We must compare the values of Q_n and Q_{n+1}.

→ $Q_n Q_{n+1}$ = 00 ⟶ ashr AQ

= 10 ⟶ Subtract M from A, ashr AQ (= ADD $\bar{M}+1$ to A; ashr AQ)

= 01 ⟶ Add M to A, ashr AQ

= 11 ⟶ ashr AQ

→ ashr AQ

⇒ Arithmetic shift right the values in register AQ

⇒ MSB is the same, shift right is performed.

lsb is lost.

→ While addition, if we encounter any carry, we must neglect it.

→ SC = Shift Counter

We must initialize it to no. of bits in multiplicand or multiplier.

→ M ← Multiplicand

Q ← Multiplier

Multiplicand and multiplier must have same no. of bits

→ Initially, A = 0 (n no. of bits), $Q_{n+1} = 0$

Eg. M ← Multiplicand ← 13

Q ← Multiplier ← −6

M = 0 1 1 0 1 (signed)

$\bar{M}$ = 1 0 0 1 0

$\bar{M}+1$ = 1 0 0 1 1

+6 ⇒ 0 0 1 1 0

$+\bar{6}$ = 1 1 0 0 1

+ 1

1 1 0 1 0 = −6

0 0 0 0 0 (2's 6+1)

$q_n q_{n+1}$	Operation	A	Q q_n	q_{n+1}	SC
		0 0 0 0 0	1 0 1 0	0	5
0 0	ashr AQ	0 0 0 0 0	0 1 1 0 1	0	4
1 0	sub M=add $\bar{M}+1$	1 0 0 1 1			
		1 0 0 1 1	0 1 1 0 1	0	
	ashr AQ	1 1 0 0 1	1 0 1 1 0	1	3
0 1	Add M	0 1 1 0 1			
		0 0 1 1 0	1 0 1 1 0	1	
	ashr AQ	0 0 0 1 1	0 1 0 1 1	0	2
1 0	sub M=add $\bar{M}+1$	1 0 0 1 1			
		1 0 1 0 0	0 1 0 1 1	1	1
	ashr AQ	1 1 0 1 0	0 0 1 0 1	1	1
1 1	ashr AQ	1 1 0 1 0	1 0 0 1 0	1	0

11101 10010

$\bar{A}$ = 00010 01101

+ 1

00010 01110 = -78

512 256 128 64 32 16 8 4 2 1 = -78

13x-6 = -78 00010 01110 = -78

Hence, Process is right

Q. Multiplicand ← M ← 15

Multiplier ← Q ← -3

M = 01111 (signed) +3 = 00011

$\bar{M}$ = 10000 +$\bar{3}$ = 11100

$\bar{M}$+1 = 10001 + 1

11101 = -3 (2's)

$q_n q_{n+1}$	Operation	A	Q q_n	q_{n+1}	Sc
			↓		
		00000	11101	0	5
10	sub M = add $\bar{M}$+1	10001			
		10001	11101	0	
	ashr AQ	11000	11110	1	4
01	Add M	01111			
		① 00111	11110	1	-
	ashr AQ	00011	11111	0	3

Q Qn+1	Operation	A	Q Qn	Qn+1	SC
		00011	11111	0	3
10	Sub M = Add $\bar{M}$+1	10001	11111	0	
		10100	11111	0	
	ashr AQ	11010	01111	1	2
11	ashr AQ	11101	00111	1	1
11	ashr AQ	11110	10011	1	0

Result: 11110 10011

Result $\Rightarrow$ 00001 01100

$+ \quad 1$

00001 01101 $\Rightarrow -45$

$-128\ 64\ 32\ 16\ 8\ 4\ 2\ 1$

$\therefore$ 15x-3 = -45 = 00001 01101

and result = -45

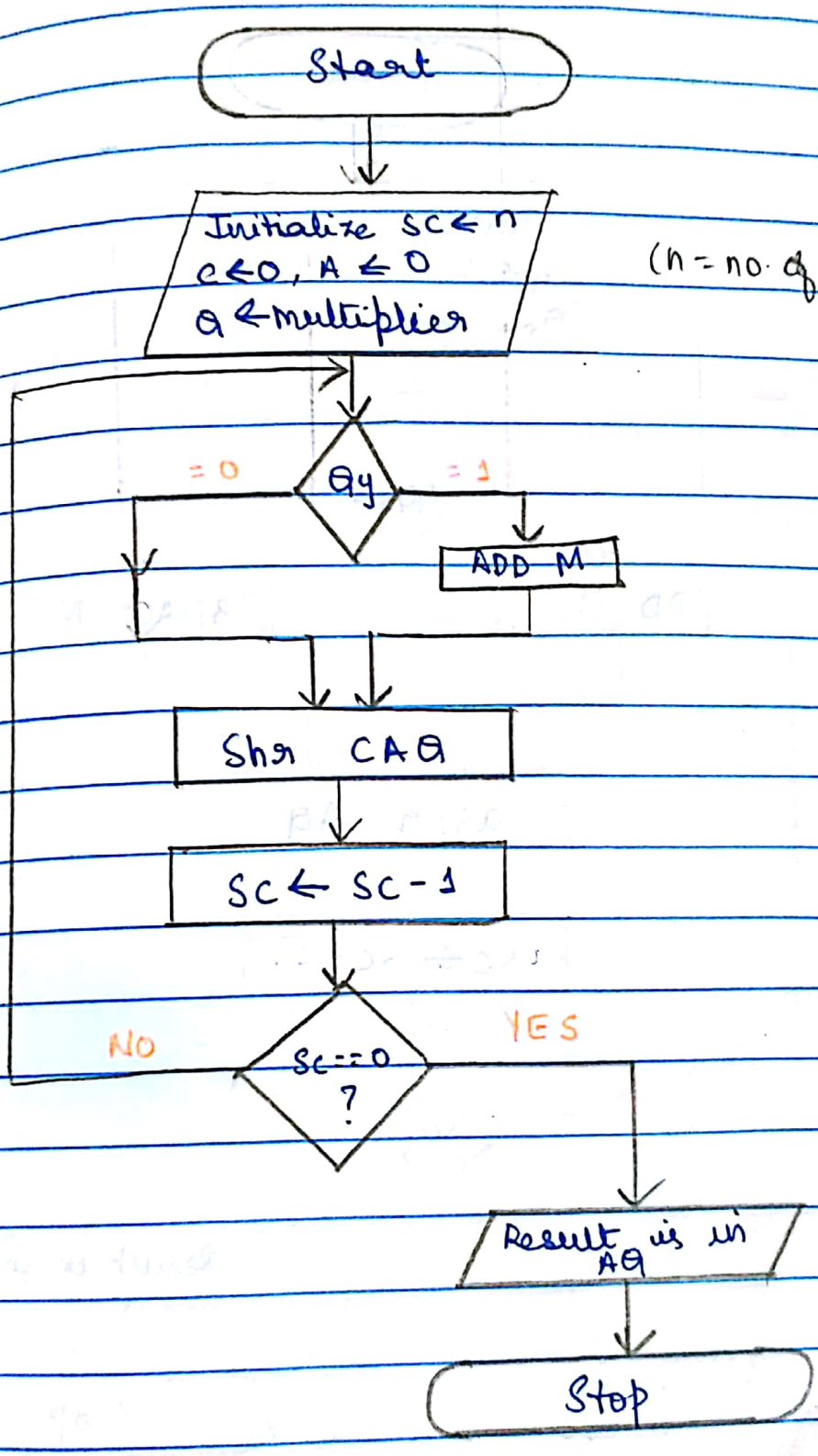
Hence, the process is right

Eg. $\overset{2^4}{0}\ \overset{2^3}{1}\ \overset{2^2}{1}\ \overset{2^1}{1}\ \overset{2^0}{0}\ \boxed{0} = +14$ $\Rightarrow$ 00 $\Rightarrow$ no change 01 $\Rightarrow$ -ve no

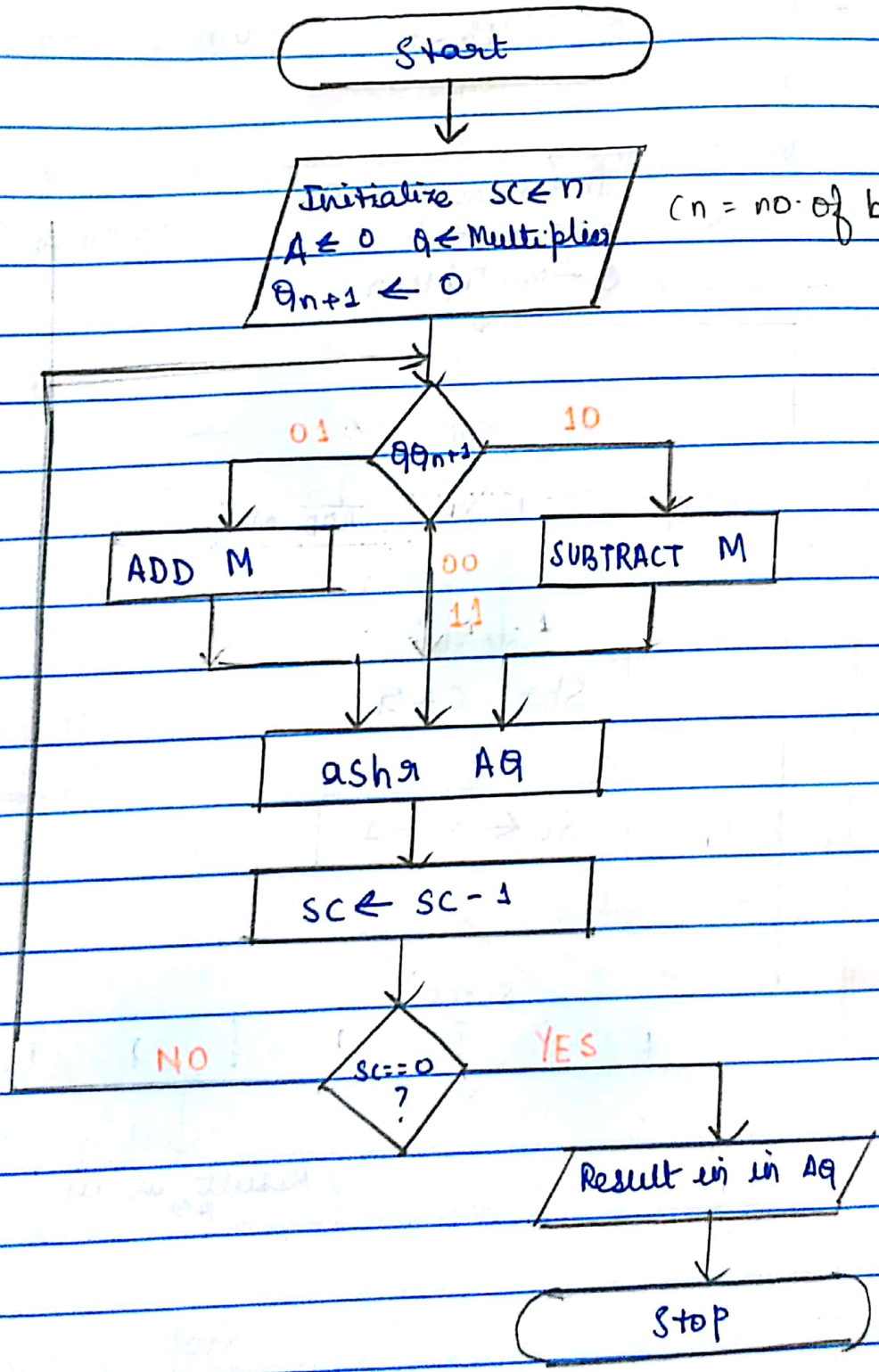
$\rightarrow$ in no format, assume 10 $\Rightarrow$ +ve no 11 $\Rightarrow$ no change

$+2^4 \quad -2^1 \quad = +16 - 2 = +14$

→ Flowchart for unsigned multiplication



→ Flowchart for Signed Multiplication (Booth Algorithm)



19-20 - SB

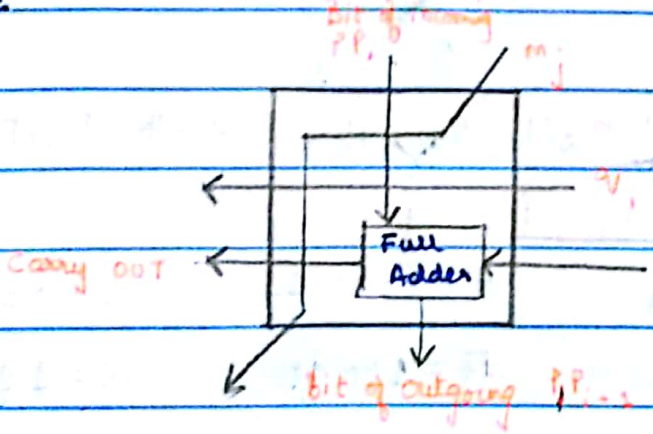
→ Block Diagram for Simple Multiplication Algorithm

M ← M_{n-1} ..... $M_3 M_2 M_1 M_0$ ← Multiplicand

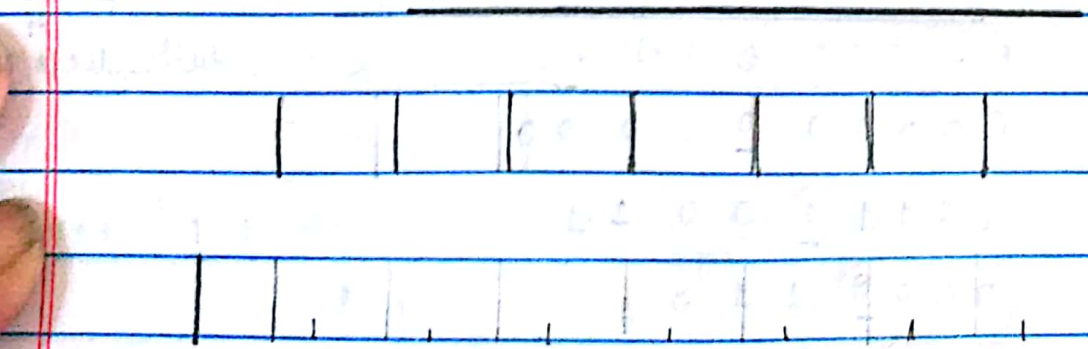
Q ← Q_{n-1} ..... $Q_3 Q_2 Q_1 Q_0$ ← Multiplier

[Both M and Q must have same no. of bits. else, make them same by adding a '0' at the L.H.S.).

Cell:



Simple Multiplication:



→ Finding the Quantity of a Number

Bit i	Bit i-1	
0	0	0 × M = 0
0	1	+1 × M = M
1	0	−1 × M = −M = M̄ + 1
1	1	0 × M = 0

2nd method of Booth Algorithm :→ Booth Multiplier Bit Pairs

13 = M −6 = Q M̄ + 1 = 10011 M = 01101

M̄ = 10010 −6 = 11010

01101 // Multiplicand

11010 0 // Multiplier

0 −1 +1 −1 0 // Multiplier after recoding

00000 0 0 0 00 // 0

11111 1 0 0 11 // 2's of M = M̄ + 1

0000 0 1101 // M

111 0 0 11 // 2's of M = M̄ + 1

(1) 0 0 0 0 0 0 // 0

1 1 0 1 1 0 0 1 0 1 1 0 1 1 0 0 1 0 0

⇒ −78 = 137 − 6. 0 0 −128 64 0 16 8 4 −2 0 = −78

Eg. Booth Algorithm

M ← Multiplicand ← 12

Q ← Multiplier ← −10

M = 01100 (signed)

$\bar{M}$ = 10011

$\bar{M}$ + 1 = 10100

+10 => 01010

$\overline{10}$ => 10101

\+ 1

10110 = −10

Q Qn+1	Operation	A	Q Qn ↓	Qn+1	SC
		00000	10110	0	5
0 0	ashr AQ	00000	01011	0	4
1 0	Sub M = add $\bar{M}$ + 1	10100			
		10100	01011	0	
	ashr AQ	11010	00101	1	3
1 1	ashr AQ	11101	00010	1	2
0 1	add M	01100			
		(1)01001	00010	1	1
	ashr AQ	00100	10001	0	

01		00100	10001	1	1
10	sub M = add $\bar{M}+1$	10100			
		11000	10001	1	
	ashr AQ	11100	01000	1	0

Result: 11100 01000

Result : 00011 10111

Result + 1 : + 1

00011 11000 = -120

512 256 128 64 32 16 8 4 2 1

∴ 12 x -10 = -120.

Hence, the process is right

Eg. M ← Multiplicand ← 15

Q ← Multiplier ← 6

M = 01111 Q = 00110

$\bar{M}$ = 10000

$\bar{M}+1$ = 10001

$q_{n-1}q_n$	Operation	A	$q_{n-1}q_n$	q_{n+1}	Sc
		00000	00110	0	5
00	ashr AQ	00000	00011	0	4
10	sub M = add $\bar{M}$+1	10001			
		10001	00011	0	
	ashr AQ	11000	10001	1	3
11	ashr AQ	11100	01000	1	2
01	sub M = add $\bar{M}$+1	10001			
		⊕01101	01000	1	
	ashr AQ	00110	10100	0	1
00	ashr AQ	00011	01010	0	0

00011 01010

64 32 16 8 4 2 1

26-02-'18

Eg. Booth Algorithm - Bit Pair Recoding of Multipliers (Booth multiplication)

M ← 13 M ← 01101 $\bar{M}$ ← 10010 $\bar{M}$+1 = 10011

Q ← −5 Q ← 00101 $\bar{Q}$ ← 11010 $\bar{Q}$+1 = 11011

01101

11011 0

0-1 +1 0 -1

11111 1 00111

00000 0 0000

00001 101

11100 11

00000 0

1 1 1 1 1 1

1 1 1 1 0 1 1 1 1 1 1 0

-128 64 32 16 8 4 2 1

0 0 -128 +64 0 0 0 0 0 -1

= -128 + 64 - 1 = -64 - 1 = -65

∴ 13 × -5

= -65

Hence Proved

Advantages of Bit Pair Recording of Multipliers (Booth Multiplier)

i) Multiplier = +1 -1 +1 -1 +1 -1 +1 -1

⇒ Worst Case

ii) Multiplier = -1 00000 +1 00000

(continuous 1s or continuous 0s)

⇒ Best Case

iii) Multiplier = +1 +1 00 +1 -1 00 -1 +1 00

⇒ Average Case

Fast Multiplication - Bit Pair Recording of Multipliers

$i+1$	i	$i-1$	
0 (0)	0 (0)	0	0 × M
0 (0)	0 (0)	1 (+1)	+1 × M
0 (+1)	1 (-1)	0 (-1)	+1 × M
0 (+1)	1 (-1)	1 (0)	+2 × M
1 (-1)	0 (0)	0 (0)	-2 × M
1 (-1)	0 (0)	1 (+1)	-1 × M
1 (0)	1 (-1)	0 (-1)	-1 × M
1 (0)	1 (-1)	1 (0)	0 × M

→ In booth multiplication algorithm, for 10 bits, we must add 10 times, but here, we add only 5 times, it reduces time.

Eg. 13 x -6

13 ← Multiplier ← M (Card) M = 01101 $\bar{M}$ = 10010 $\bar{M}$+1 = 10011

-6 ← Multiplier ← Q 6 = 00110 $\bar{6}$ = 11001 $\bar{6}$+1 = 11010

-6 = $\bar{6}$+1 = 1 1 0 1 0 0

0 -1 +1 -1 0 => -6 Recoded

0 -1 -2

+1 = M -1 = $\bar{M}$ +2 = RLC M -2 = RLC $\bar{M}$+1 0 = 0

same M add LSB+0 same $\bar{M}$ add LSB+0

0 1 1 0 1

× 0 -1 -2

1 1 1 1 1 0 0 1 1 0

1 1 1 1 0 0 1 1

0 0 0 0 0 0

1 1 1 1 1 1

1] 1 1 1 0 1 1 0 0 1 0 [0

512 256 128 64 32 16 8 4 2 1

0 0 -128 +64 0 -16 0 +4 -2 0

= -78

= 13 x -6

Hence Proved

Leave 2 gaps while writing ∵ we consider 2 bits

eg 13×-5

13 ← Multiplicand ← M M = 01101 $\bar{M}$ = 10010 $\bar{M}+1$ = 10011

-5 ← Multiplier ← Q 5 = 00101 $\bar{5}$ = 11010 $\bar{5}+1$ = 11011

$-5 = \bar{5} + 1 = 11011\ 0$

0-1+1 0-1 ⇒ -5 Recoded

0 -1 -1

+1 = M -1 = $\bar{M}$ +2 = RLC M -2 = RLC $\bar{M}+1$ 0 = 0

same M add lsb+0 same $\bar{M}$ add lsb+0

```
  01101
x 0-1-1
-------
 11111 10011
 11110 0 11
 00000 0
```

Leave 2 gaps while writing ∵ we consider 2 bits

```
    1 1 1 1
-------------
1 1 1 1 0 1 1 1 1 1 [0
```

512 256 128 64 32 16 8 4 2 1

0 0 -128 64 0 0 0 0 -1

= -126 + 64 - 1

= -65

= 13 x -5

Hence Proved

27-02-'18

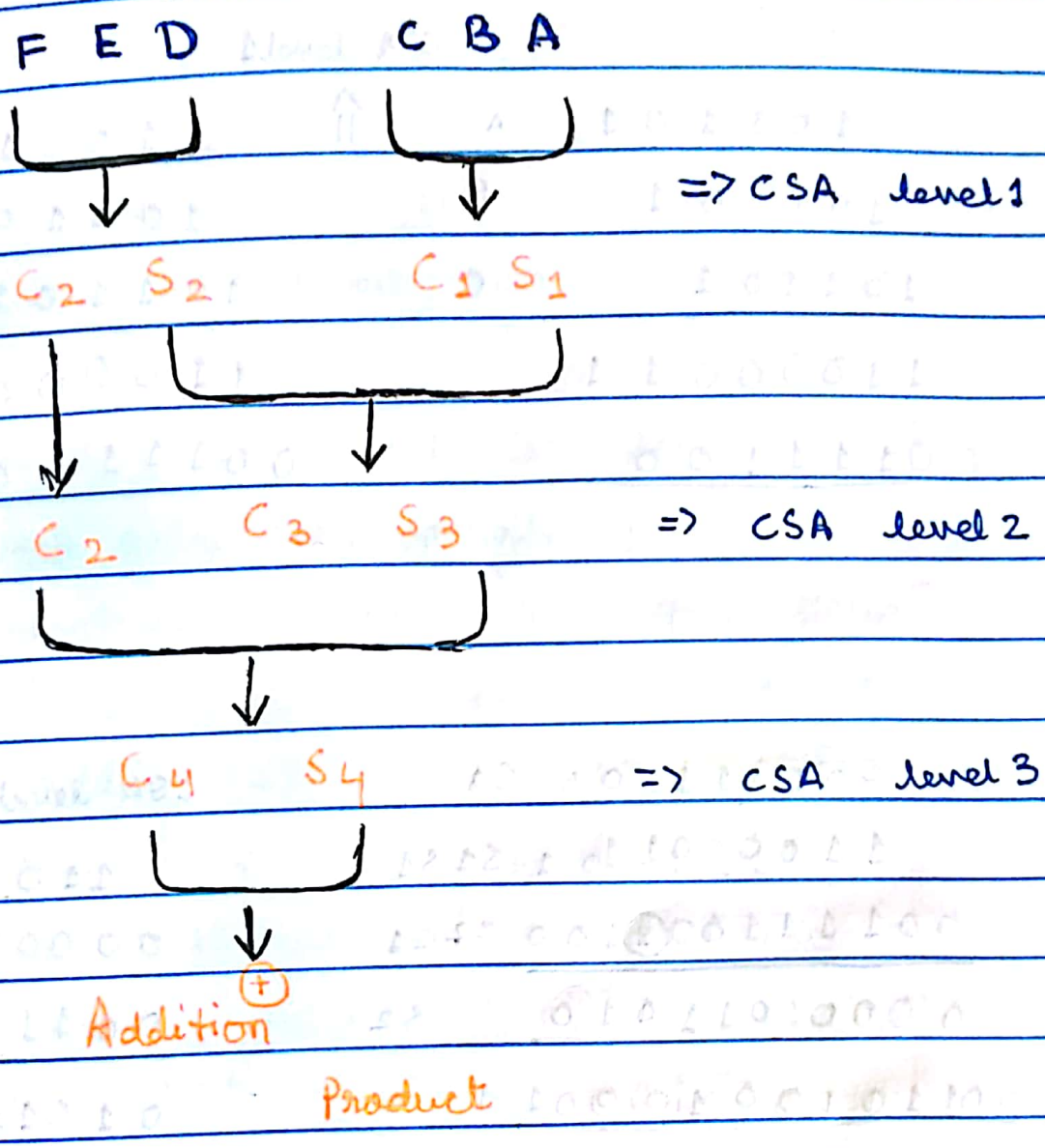
Fast Multiplication - Carry Save Addition (CSA Algorithm)

45 (M)
× 63 (Q)

```
        1 0 1 1 0 1
      × 1 1 1 1 1 1        ← Summands
        1 0 1 1 0 1    A
      1 0 1 1 0 1      B
    1 0 1 1 0 1        C
  1 0 1 1 0 1          D
1 0 1 1 0 1            E
1 0 1 1 0 1            F
  1 1 1 1 1 1 1
(1)↓
1 0 1 1 0 0 0 1 0 0 1 1
```

(If the carry is 12th bit, consider it, else, if the carry is 13th bit, neglect it in 6x6)

Schematic Representation of CSA



→ We are saving the carry and sending it to the next level.

P.T.O.

Eg. 45 (M)
×63 (Q)

CSA level 1

```
  1 0 1 1 0 1   A          1 0 1 1 0 1
  1 0 1 1 0 1   B        1 0 1 1 0 1
1 0 1 1 0 1     C      1 0 1 1 0 1
1 1 0 0 0 1 1   S1   1 1 0 0 0 0 1 1   S2
0 0 1 1 1 1 0 0 C1   0 0 1 1 1 1 0 0   C2
```

//While addition, do not do normal addition with carry, Write sum and carry separately

CSA level 2:

```
1 1 0 0 0 0 1 1 1 S1
0 0 1 1 1 1 0 0   C1
1 1 0 0 0 0 1 1 ⊗   S2
1 1 0 1 0 1 0 0 0 1 1   S3
0 0 0 0 1 0 1 1 0 0 0   C3
```

//While addition, do not do normal addition with carry, write sum and carry separately. While adding S2, leave one extra gap ⊗ apart from the normal one

CSA level 4:

```
1 1 0 1 0 1 0 0 0 1 1   S3
0 0 0 0 1 0 1 1 0 0 0   C3
0 0 1 1 1 1 0 0 ⊗ ⊗   C2
0 1 0 1 1 1 0 1 0 0 1 1   S4
0 0 1 0 1 0 1 0 0 0 0 0   C4
```

//While addition, do not do normal addition with carry, write sum and carry separately. While adding C2, leave two extra gaps ⊗⊗ apart from the normal ones.

Addition Product:

```
  0 1 0 1 1 1 0 1 0 0 1 1
  0 0 1 0 1 0 1 0 0 0 0 0
    1   1 1 2
  ---------------------------
  0 1 0 1 1 0 0 0 1 0 0 1 1
```

=> Final Product

(If the carry is 13th bit, neglect it, else, consider it)

Total number of logic gates for 6 by 6 Multiplication:

i) One AND gate for generating six summands.

ii) To generate S4, C4; six logic gate delays (assume each CSA level requires two gate delays.

iii) For adding S4, C4 using carry look ahead addition, we require 8 logic gate delays.

=> Total = 15 gate delays (using CSA)

=> For normal multiplication, total logic gate delays = $6(n-1)-1 = 6(6-1)-1 = 29$ gate delays

Total number of logic gates for 32 by 32 multiplication

i) One AND gate for generating all summands.

ii) For 8 CSA levels, we require $8 \times 2 = 16$ logic gate delays to get Sn, Cn.

iii) For adding Sn, Cn using carry look ahead addition, we require 12 logic gate delays.

=> Total = 29 gate delays (using CSA)

=> For normal multiplication, total gate delays = $6(n-1)-1 = 6(32-1)-1 = 185$ gate delays

21-02-18

Integer Division - Restoring Division, Non Restoring Division

Steps (Restoring Division)

i) Initialization (Q = dividend, A = (n+1) 0s)

ii) Shift left (msb of A is lost, lsb of Q is kept empty)

iii) Subtract M(divisor) = Add $\bar{M}+1$

iv) If msb of A is 0, place 1 in lsb of Q and restore the value in A (add M to A).

If msb of A is 1, place 0 in lsb of Q and do not restore the value in A, leave it as it is.

v) Perform 'n' such cycles, where 'n' is no. of bits in Q

vi) Finally, A will have the remainder and Q will have the quotient.

Eg. 11/3 (Restoring Division)

Q ← 11 ← 1011 ← divident M ← 3 ← divisor ← 00011

M = 00011 $\bar{M}$ = 11100 $\bar{M}+1$ = 11101

	A	Q	
Initialization	00000	1011	
Shift	00001	011□	
Subtract	11101		First Cycle
Set q_0 to 0 ←	(1) 1110	0110	
Restore	(1) 00011	0110	
	00001		

	A	Q	
	00001	0110	
Shift	00010	110□	Second Cycle
Subtract	1101		
Set q0 to 0	11111	1100	
Restore	(1) 0 0 0 1 1		
	00010	1100	
Shift	00101	100□	Third Cycle
Subtract	(1) 1 1 1 0 1		
Set q0 to 1	00010	1001	
Shift	00101	001□	Fourth Cycle
Subtract	(1) 1 1 1 0 1		
Set q0 to 1	00010	0011	
	Remainder	Quotient	

∴ Remainder = 10 = 02

Quotient = 11 = 03

3)11(3 → Quotient

-9

2

↳ Remainder

Hence Proved

Restoring Division Algorithm:

→ Initializing Q, A and M.

→ Do the following 'n' times: (n = no. of bits)

i) Shift 'A' and 'Q' left one binary position
= Shift left AQ.

ii) Subtract M from A and place the answer back in A = Add $\bar{M}+1$.

iii) If the sign (msb) of A is 1, set Q_0 to 0 and add M back to A; i.e., restore A.
Otherwise, set Q_0 to 1; do not restore

Non Restoring Division Algorithm:

→ Initializing Q, A and M.

i) Do the following 'n' time: (n = no. of bits)

(a) If the sign (msb) of A is 0, shift A and Q left one bit position and Subtract M from A.
Otherwise, shift A and Q left and add M to A.

(b) Now, if the sign of A is 0, set Q_0 to 1, otherwise, set Q_0 to 0.

ii) If the sign of A is 1, add M to A.

Eg. 10/3 (Non Restoring Division)

Q ← 10 ← 1010 ← divident M ← 3 ← divisor ← 00011

M = 00011 $\bar{M}$ = 11100 $\bar{M}+1$ = 11101

	A	Q	
Initialization	00000	1010	
Shift	00001	010□	First Cycle
Subtract	11101		
Set q0 to 0 ←	1 11 10	0100	
Shift	11100	100□	Second Cycle
Add	00011		
Set q0 to 0 ←	1 1 1 1 1	1000	
Shift	11111	000□	Third Cycle
Add	① 0 0 0 1 1		
Set q0 to 1 ←	0 0 0 1 0	0001	
Shift	00100	001□	Fourth Cycle
Subtract	① 1 1 1 0 1		
Set q0 to 0 ←	0 0 0 0 1	0011	

* = 0 = no change

Remainder Quotient

∴ Remainder = 01 = 1

Quotient = 11 = 3

Hence Proved

3)10(3 → Quotient
-9
1
↳ Remainder

Eg. 9/4 (Non Restoring Division)

Q ← 9 ← 1001 ← divident M ← 00100 ← 4 ← divisor

M = 00100 $\bar{M}$ = 11011 $\bar{M}$ + 1 = 11100

	A	Q	
Initialization	00000	1001	
Shift	00001	001□	First Cycle
Subtract	11100		
Set q0 TO 0 ←	11101	0010	
Shift	11010	010□	Second Cycle
Add	00100		
Set q0 TO 0 ←	11110	0100	
Shift	11100	100□	Third Cycle
Add	(1) 00100		
Set q0 TO 1 ←	00000	1001	
Shift	00001	001□	Fourth Cycle
Subtract	11100		
Set q0 TO 0 ←	11101	0010	

↓ r = 1 => Add M

(1) 00100

00001 0010

Remainder Quotient

∴ Remainder = 01 = 1

Quotient = 10 = 2

4)9(2 → Quotient

−8

1 → Remainder

Hence Proved

Eg. 10/3 (Restoring Division)

9 ← 10 ← 1010 ← dividend M ← 00011 ← 3 ← divisor

M = 00011 $\bar{M}$ = 11100 $\bar{M}$+1 = 11101

	A	Q	
Initialization	00000	1010	
Shift	00001	010 □	First Cycle
Subtract	1 1 1 0 1		
Set Q0 to 0 ←	① 1 1 1 1 0	0101	
Restore	① 0 0 0 1 1		
	00001	0101	
Shift	00010	101 □	Second Cycle
Subtract	1 1 1 0 1		
Set Q0 to 0 ←	① 1 1 1 1 1	1010	
Restore	① 0 0 0 1 1		
	00010	1010	
Shift	00101	010 □	Third Cycle
Subtract	① 1 1 1 0 1		
Set Q0 to 1 ←	00010	0101	
Shift	00100	101 □	Fourth Cycle
Subtract	① 1 1 1 0 1		
Set Q0 to 1 ←	00001	1011	
		Quotient	Remainder

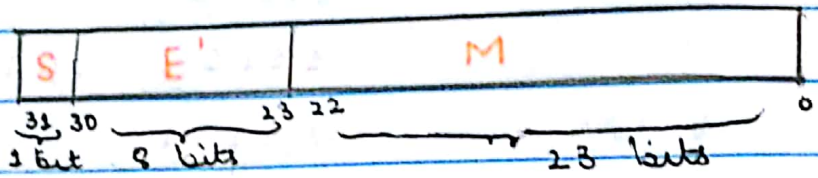
28-02-'18

Floating Point Numbers and Operations

→ IEEE Standards

- IEEE - 32 Single Precision

- Can display upto 8 to 10 bits after decimal point



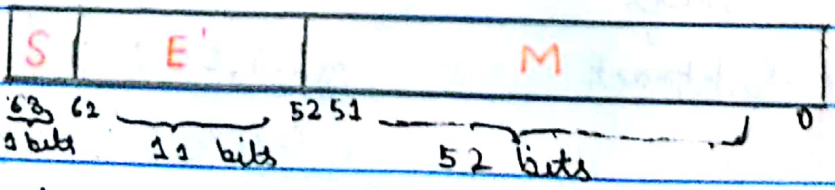
(bias) $E' = E + 127$ $\quad 0 \le E' \le 255$

[excess −127] $\quad -127 \le E \le 128$

$$\text{Value} = \pm 1.M \times 2^{E'-127}$$

- IEEE - 64 Double Precision

- Can display upto 16 or more bits after decimal p



(bias) $E' = E + 1023$ [excess - 1023]

$$\text{Value} = \pm 1.M \times 2^{E'-1023}$$

Eg. $102.56 = 1.0256 \times 10^2$ $\quad 0.0123 = 1.23 \times 10^{-2}$

(move left(point): +ve power, move right (point): -ve power)

1.23×10^{-2} → Exponent (E)

Mantissa (M)

Normalization:

Eg. 12.0625

12 = 1100

0.625 = 101

0.625 × 2 = 1.250

0.250 × 2 = 0.5

0.5 × 2 = 1.0

∴ 12.0625

= 1100.101

= 1.100101×2^3

E' = E + 127 = 3 + 127 = 130

0	10000010	10010100 ···· 00

(= 130) (= .100101)

Eg. 23.0125

23 = 10111

0.125:

2 | 23
2 | 11 - 1
2 | 5 - 1
2 | 2 - 1
1 - 0

0.125 × 2 = 0.2500

0.25 × 2 = 0.5

0.5 × 2 = 1.0

∴ 23.0125

= 10111.0001

= 1.0111001×2^4

E' = E + 1023 = 1027

0	10000000011	01110010000 ···· 00

If 0.0125 is there,

0.0125 × 2 = 0.0250

0.0250 × 2 = 0.0500

0.05 × 2 = 0.1000

0.1 × 2 = 0.200

0.2 × 2 = 0.40

0.4 × 2 = 0.80

0.8 × 2 = 1.6

0.6 × 2 = 1.2

0.2 × 2 = 0.4

→ Special Values

In IEEE standards (IEEE-32), we represent normal value only from $1 \leq E' \leq 254$.
We represent/use 0 and 255 for special values.

Special Values:

Not possible to normalize these numbers (all 4 types)

- $E' = 0$ and $M = 0$ => ± 0
- $E' = 255$ and $M = 255$ => $\pm \infty$
- $E' = 0$ and $M \neq 0$ => Denormal Numbers (smaller than smallest number)
- $E' = 255$ and $M \neq 0$ => NaN (Not a number, Eg. 0/0, $\sqrt{-1}$)

→ Exceptions

i) Underflow Exception (E < -127, Not possible to represent using IEEE-32)
ii) Overflow Exception (E > 128, Not possible to represent using IEEE-32)
iii) Divide By Zero Exception (Not possible to divide by zero)
iv) Inexact Exception (we need to round of the value to normalize, we do not get exact value when we try to say 0.0125)
v) Invalid Exceptions (Invalid numbers, Eg. 0/0, $\sqrt{-1}$)

→ Guard Bits and Truncation

- In IEEE-32 bit format, to represent Mantissa, we have only 23 bits. But, if we get only more than 23 bits in Mantissa, the extra bits are called Guard Bit.
- To solve this problem, we use Truncation - More bits setting in 23 bits is called Truncation. Using rounding off techniques, we eliminate odd bits
- We have three techniques for truncation:
  i) Chopping ii) Von-Neuman iii) Rounding

i) Chopping

- Eliminate any extra bits after 23, irrespective of them being 1's or 0's, more error rate
- $0 \cdot b_{-1} b_{-2} b_{-3} \underbrace{000}_{b_{-4} b_{-5} b_{-6}}$ to $0 \cdot b_{-1} b_{-2} b_{-3} \underbrace{111}_{b_{-4} b_{-5} b_{-6}} = 0 \cdot b_{-1} b_{-2} b_{-3}$
- Error is not symmetric, it is 0 or +ve, not -ve

ii) Von-Neuman

- If all are 0's in guard bits, follow chopping, if (atleast any) 1 is present, make lsb of retained bits to 1.
- $b_{-4} b_{-5} b_{-6} = 000 \Rightarrow 0 \cdot b_{-1} b_{-2} b_{-3}$, $b_{-4} b_{-5} b_{-6} \neq 0 \Rightarrow 0 \cdot b_{-1} b_{-2} 1$

iii) Rounding

- If guard bit starts with 1, add $0.00\cdots1$, if guard bit starts with 0, no change
- $0.b_{-1} b_{-2} b_{-3} 01\cdots = 0.b_{-1} b_{-2} b_{-3}$
  $0.b_{-1} b_{-2} b_{-3} 1\cdots = 0.b_{-1} b_{-2} b_{-3}$
  $+0.0\ 0\ 1$

→ Arithmetic Operations

* Algorithm for Addition / Subtraction
  i) Exponents must be made equal
  ii) Perform addition or subtraction
  iii) Perform normalization if required

* Algorithm for Multiplication
  i) Add exponents
  ii) Subtract 127 from resulting exponent
  iii) Multiply Mantissas
  iv) Perform normalization if required

* Algorithm for Division
  i) Subtract exponents
  ii) Add 127 to resulting exponent
  iii) Divide Mantissas
  iv) Perform normalization if required