UNIT II

Arithmetic Operations :-

ALU – Addition and Subtraction – Multiplication – division – floating point operation – Subword parallelism.

① ALU (Arithmetic & Logic Unit)

1.) Arithmetic & Logic Unit :-

A basic operation in all digital computers is the addition or subtraction of Two numbers. Arithmetic operations occur at the machine instruction level. They are implemented along with basic logic functions such as AND, OR, NOT and Exclusive OR (XOR), in the Arithmetic and logic unit (ALU) subsystem of the processor.

The time needed to perform an addition operation affects the processors performance. Multiply and devide operation which require more complex circuitry than either addition or subtraction operation also effect the performance.

1.2) Signed and Unsigned Numbers :-

Computer represent numbers in binary. The hexadecimal notation is sometimes used to represent binary number as it is more concise. Binary numbers has two possible digits or bit : 0 and 1. So they are considered base 2 number. (base 10 for decimal). Generalizing the point in any number base the value of i th digit d is

$$d \times base^i$$

where i starts at 0 and increases from right to left.

In real world of Mathematics (computers) must represent both positive and negative binary numbers. For eg: even when dealing with positive arguments, Mathematical operations may produce a negative result. (ie) When we subtract larger number from smaller number

eg: $124 - 237 = -113$.

Thus a consistent method for representing negative numbers in binary Computer arithmetic operation is needed.

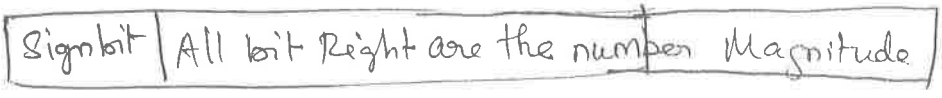
There are various approach but they all involve using one of the digit of the binary number to represent the sign of the number. There are three methods, they are the ① Sign and Magnitude representation.

② One's Complement Representation

③ Two's Complement Representation.

1.2.1) Sign and Magnitude Representation :-

In sign and Magnitude representation, the left bit is used to reserved for sign bit (0 indicates positive number & 1 indicates negative numbers).



if 0 +ve numbers
if 1 -ve numbers.

Advantages of Sign - Magnitude :-

① It is very simple to implement

② Useful for floating point representation.

Disadvantages of Sign - Magnitude :-

① Sign bit is independent of Magnitude; can be both +0, -0 difficult to represent.

1.2) Ones Complement Representation:

: The negative number of the same Magnitude as any given positive number is its one's complement. If m = 01001100 then m complement is 10110011. The most significand bit is the sign and is 0 for positive binary number and 1 for negative number.

Drawbacks:-

The drawback in one's complement method has two zeros (+0 & -0) ie

m 00000000 = +0
11111111 = -0

it is hard to implement in hardware.

1.2.3) Two's Complement Representation:-

Since there is drawbacks in sign & Magnitude representation and ones complement, another approach called the 2's complement has become a standard for representing the sign of a fixed point binary number in computer circuits. The 2's Complement representation is widely used in computers and it is a signed representation as it can represent both positive and

negative numbers:

In general, most significant bit (MSB or S) is 0 indicate a positive number and it is represented by its binary equivalent directly. If S=1 indicate a negative number or represent 2's Complement Value. The 2's complement of a negative number can be obtained by adding one to the one's complement.

eg:- The two's complement of -13 can be Obtained as follow

(X) Step1 :- Obtain the binary of +13 [8 bit representation]

+13 = 0000 1101

(X) Step2 : Take one's complement

1's complement is : 1111 0010

(X) Step3 : Add 1 to 1's complement.

```
  1111 0010
+        1
-----------
  1111 0011
```

$-13 = 1111\ 0011_2$

Which is the Value of '-13'!

The 2's complement representation of a negative number always has '1' as its MSB.

2 | 13
2 | 6 - 1
2 | 3 - 0
1 - 1

$(13_{10}) = 1101_2$

In a n-bit representation, if we add 2 n-bit numbers result is n+1 bits the left most bit is discarded. Since in computer there are no extra columns. If there are only 8 bit the 9th bit if resulted will be discarded.

eg: Consider 4 bit Addition.

$12_{10} + 14_{10}$

1100_2

$+ 1110_2$

11010

1 discarded

because it can hold only 4 bit.

1.3 Addition and Subtraction:

Addition is carried out similar to decimal addition (ie) start adding bit by bit from right to left, carry generated during addition, ~~then~~ it is taken to next digit to the left (higher order bit).

Subtraction uses addition (2's complement) The appropriate operand is negated before being added.

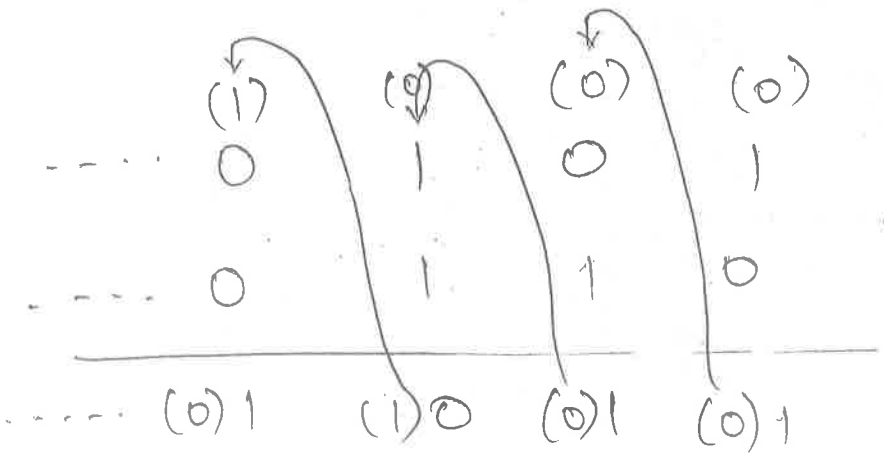
3.1) Boolean Addition :-

When adding two numbers, if the sum of the digit in a given position equals or exceeds, Then a carry is propagated. For eg: if two one's are added sum is 10_2 thus record a 0 for the sum and propagate a carry value 1 into next higher significant bit.

eg:

$5_{10} + 6_{10}$

```
                                                    1
  0000 0000 0000 0000  0000 0000 0000  0101₂  (5₁₀)
(+) 0000 0000 0000 0000  0000 0000 0000  0110₂  (6₁₀)
-------------------------------------------------------
  0000 0000 0000 0000  0000 0000 0000  1011₂  (11₁₀)
```



Carry propagation.

3.2) Boolean Subtraction :-

There are two ways of performing boolean subtraction.

i) Using normal subtraction procedure.

ii) negate the subtrahand (a-b ie b is subtrahand)

then perform addition.

eg: $5_{10} - 6_{10}$

$\Downarrow$

$5_{10} + (-6_{10})$

$\hookrightarrow$ -6 can be obtained by 2's complement method.

eg:-

i) $8_{10} - 5_{10}$

```
  0000 0000 0000 0000 0000 0000 0000 0111 0110 1010 (8_10)
(-) 0000 0000 0000 0000 0000 0000 0000 0101 (5_10)
-------------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 0011_2 (3_10)
-------------------------------------------------
```

(or) by using 2nd Method. (ie 2's complement)

ii) $8_{10} - 5_{10} \Rightarrow 8_{10} + (-5_{10})$

find -5_{10} $\Rightarrow$ 0000 0000 0000 0000 0000 0000 0000 0101 (+5

ones complement $\Rightarrow$ 1111 1111 1111 1111 1111 1111 1111 1010

+1

1111 1111 1111 1111 1111 1111 1111 1011 (-

negative numbers:

In general, most significant bit (MSB or S) is 0 indicate a positive number and it is represented by its binary equivalent directly. If S=1 indicate a negative number or represent 2's Complement Value. The 2's complement of a negative number can be obtained by adding one to the one's complement.

eg:- The two's complement of -13 can be Obtained as follow

(X) Step1 :- Obtain the binary of +13 [8 bit representation]

+13 = 0000 1101

(X) Step2 : Take one's complement

1's complement is : 1111 0010

(X) Step3 : Add 1 to 1's complement.

```
  1111 0010
+        1
-----------
  1111 0011
```

$-13 = 1111\ 0011_2$

Which is the Value of '-13'!

The 2's complement representation of a negative number always has '1' as its MSB.

2 | 13
2 | 6 - 1
2 | 3 - 0
1 - 1

$(13_{10}) = 1101_2$

In a n-bit representation, if we add 2 n-bit numbers result is n+1 bits the left most bit is discarded. Since in computer there are no extra columns. If there are only 8 bit the 9th bit if resulted will be discarded.

eg: Consider 4 bit Addition.

$12_{10} + 14_{10}$

1100_2

$+ 1110_2$

11010

1 discarded

because it can hold only 4 bit.

1.3 Addition and Subtraction:

Addition is carried out similar to decimal addition (ie) start adding bit by bit from right to left, carry generated during addition, it is taken to next digit to the left (higher order bit).

Subtraction uses addition (2's complement) The appropriate operand is negated before being added.

now add 8_{10} and (-5_{10})

```
(1) . . . .  . . . .  . . . .  . . . .  . . . .  . . . .  . . . .  . . . .  (1)(1)(1)(1)
 0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   1000   (8 10)
(+)1111 1111   1111 1111   1111 1111   1111 1111   1111 1111   1111 1111   1111 1111   1011   (-5 10)
-------------------------------------------------------------------------------------------------
 1  0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   0000 0000   0000 0011   (3 10)
discard
```

4) Overflow :

" Overflow occurs when there is insufficient bit in the binary number representation to hold the result of an arithmetic operation.

When a 32 bit number is added or subtracted a result produced may need 33 bit to store the result. This condition is called overflow.

eg: Consider 4 bit

8_{10} 1000 → 4 bit
9_{10} (+)1001 → 4 bit

(1)0001 → 5 bit (n+1) bit is needed)

This is called overflow bit.

To detect or compensate for overflow one need n+1 bits, if an n-bit number representation is employed.

1.4.1) Conditions where Overflow cannot occur :-

① In addition if the operands are of different sign: [eg: (+5) + (-4)] ⇒ 0101 (5) + 1010 (-4) = 1111 (no overflow)

-4 ⇒ 0110 1001 + 1 = 1010

② In subtraction if the operands are of same sign: [eg: (+5) − (+2) or (−5) − (−2)]

1.4.2) Conditions where overflow occurs :-

Operation		Operand A	Operand B	result Indicating Overflow
addition	A + B	≥ 0	≥ 0	< 0
	A + B	< 0	< 0	≥ 0
Subtraction	A − B	≥ 0	< 0	< 0
	A − B	< 0	≥ 0	≥ 0

(i) Overflow occurs when we adding two positive numbers and sum is a negative or adding two negative numbers and result is positive.

ii) When we subtract a negative number from a positive number & get a negative result or when we subtract a positive number from a negative number and get a positive result.

1.4.3) MIPS overflow Handling :-

→ It raises a exception when overflow occurs

→ 2's Complement arithmetic operation (add, addi, sub raise exception overflow).

→ Addu and addiu not raise exception, since they are used for arithmetic operation on adders.

2. Addition & Subtraction.

Addition and subtraction are the Arithmetic operations performed in the ALU (Arithmetic & Logic Unit). Addition is performed by adding bit by bit digits from right to left, Carry generated during the addition is taken to next digit to the left. Subtraction can be performed by subtracting the digits or adding the 2's complement of the subtrahand).

eg:

$5_{10} + 6_{10}$

```
                                                        1
  0000 0000 0000 0000 0000 0000 0000 0101₂
+ 0000 0000 0000 0000 0000 0000 0000 0110₂
-------------------------------------------
  0000 0000 0000 0000 0000 0000 0000 1011  (11₁₀)
```

Carry propagation

```
(1) (0) (0)
  0   1   0   1
  0   1   1   0
-----------------
(0) 1 (1) 0 (0) 1 (0) 1   => (1011)₂
```

eg: $8_{10} - 5_{10} = (3)_{10}$

```
0000 0000 0000 0000 0000 0000 0000 1000₂
0000 0000 0000 0000 0000 0000 0000 0101₂
-------------------------------------------
0000 0000 0000 0000 0000 0000 0000 0011₂
                                        ↓
                                        3₁₀
```

There are some Adders, High speed Adders for performing the operation.

1. Full Adder
2. Ripple Carry Adder
3. Carry Lookahead Adder" — Fast Adders.

Full Adder :-

Full Adder is a circuit for performing 1 bit addition, if two bits x_i and y_i are added Sum S_i and C_{i+1} is generated. A full adder uses three input one is X_i, another is Y_i third input is the C_i represents the carry-in to the i^{th} stage.

The Logic truth table for Full adder is given below.

Truth Table for Full Adder:

Input			output	
x_i	y_i	Carry-in c_i	Sum s_i	Carry 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

The logical expression:-

$$\text{Sum } s_i = \bar{x_i}\bar{y_i}c_i + \bar{x_i}y_i\bar{c_i} + x_i\bar{y_i}\bar{c_i} + x_iy_ic_i$$

$$= \bar{x_i}(\bar{y_i}c_i + y_i\bar{c_i}) + x_i(\bar{y_i}\bar{c_i} + y_ic_i)$$

$$= \bar{x_i}(y_i \oplus c_i) + x_i(\overline{y_i \oplus c_i})$$

$\because A \oplus B = \bar{A}B + A\bar{B}$

$$\boxed{\text{Sum } s_i = x_i \oplus y_i \oplus c_i}$$

$$\text{carry } c_{i+1} = \bar{x_i}y_ic_i + x_i\bar{y_i}c_i + x_iy_i\bar{c_i} + x_iy_ic_i$$

(1) (3) (2)

$$= y_ic_i(\bar{x_i} + \bar{x}) + x_ic_i(\bar{y_i} + \bar{y}) + x_iy_i(\bar{c_i} + c_i)$$

$\because A + A = A$

$\because A + \bar{A} = 1$

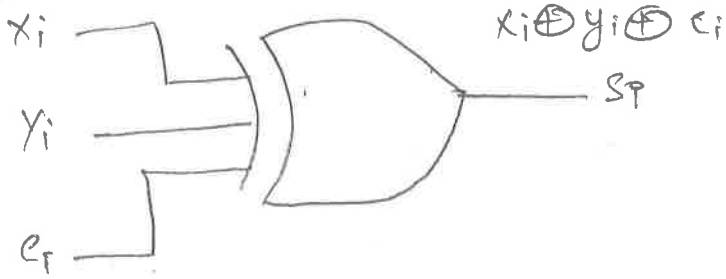
$$\boxed{c_{i+1} = y_ic_i + x_ic_i + x_iy_i}$$

The logical Expression for sum can be implemented using a 3-input XOR gate. And carry C_{i+1} can be implemented with two-level AND-OR logic circuit.

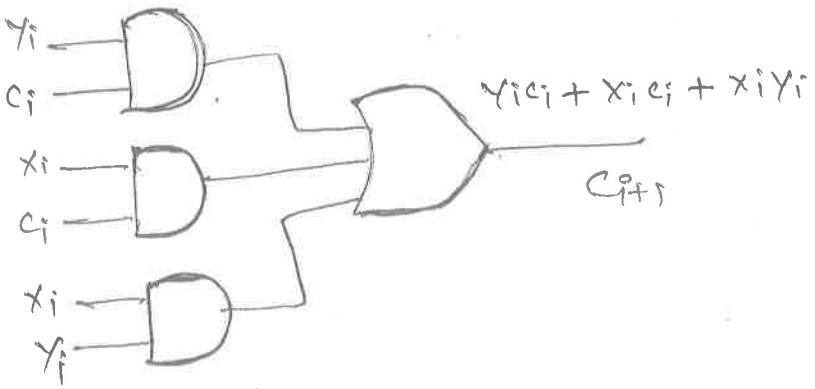
A conventional Symbol for the complete circuit for a single stage of addition called Full Adder (FA).

for sum:

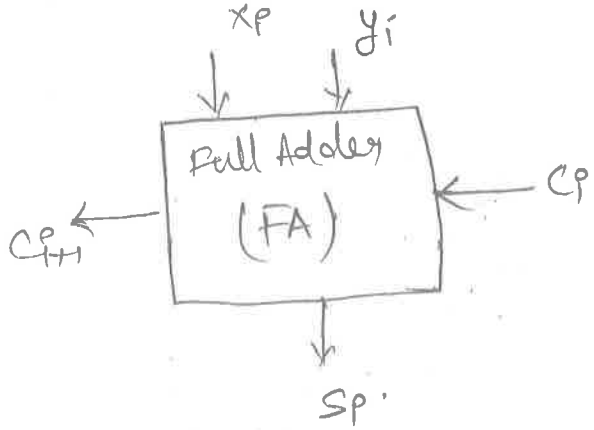
(i)



(ii) for carry:



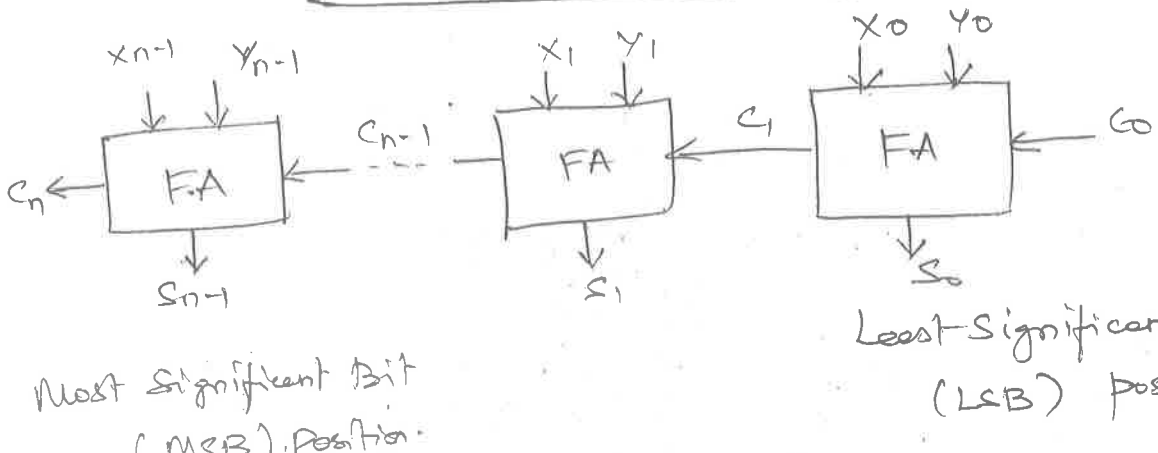
(iii)



2) Ripple Carry Adder :-

A single full adder is capable of adding two one-bit numbers and an input carry. In order to add binary numbers with more than one bit, additional full adders must be employed. The n-bit parallel adder can be constructed using number of full adder circuit connected in parallel.

An n-bit ripple Carry Adder.



Each 1 bit adder stage supplies a carry bit to the stage on its left, Hence carry signals propagate through the adder from Right to Left giving rise to the name "ripple-carry adder".

In the worst case a carry signal can ripple through all n stage of adder. The input carry is normally set to 0

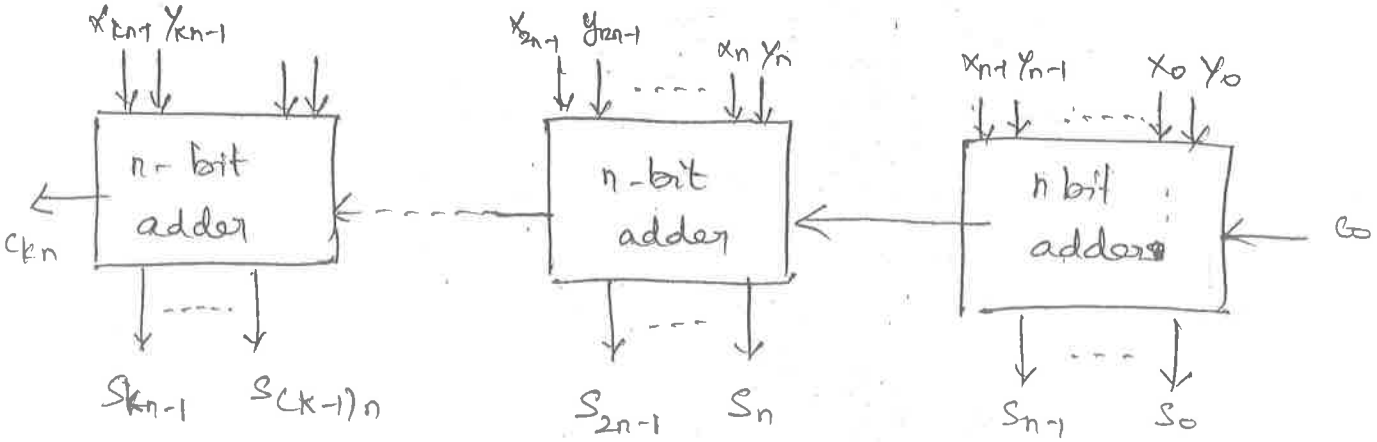
for addition.

The Maximum propagation delay of an n-bit ripple carry adder, the operating speed is nd, where d is the delay of a full adder stage.

Fast Adder Circuit:-

In an N-bit parallel adder (ripple carry adder there is

The carry signals are also useful for interconnecting k-adders to form an adder capable of handling input numbers that are kn bits long.



Cascade of k n-bit adders.

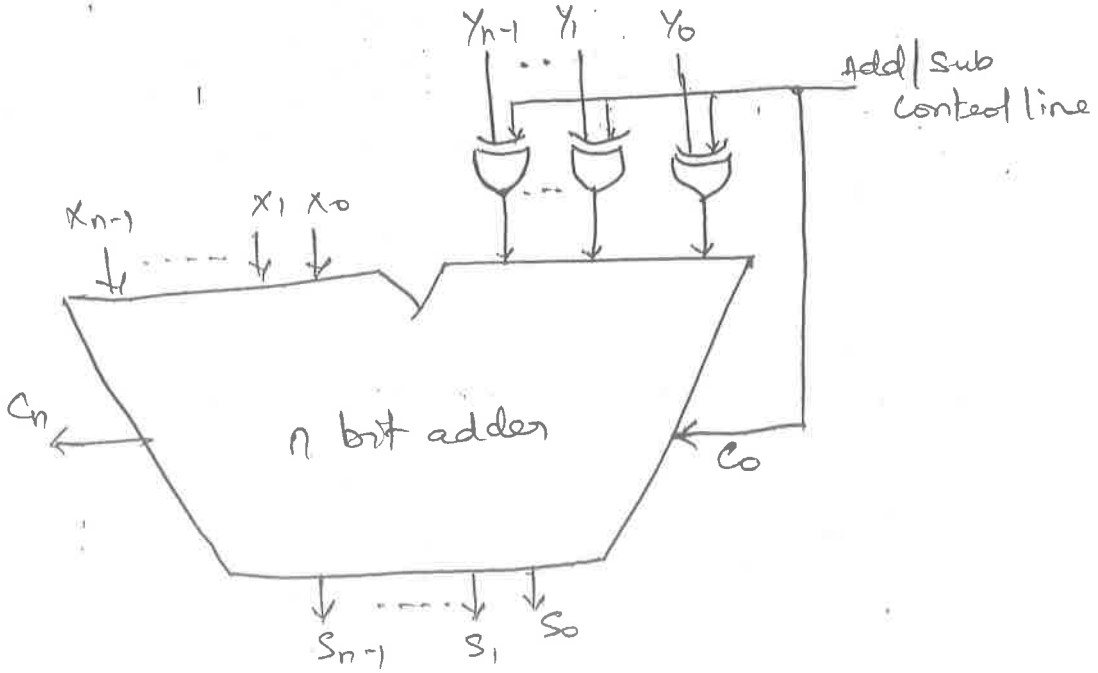
Addition or Subtraction logical Unit:-

The addition / subtraction logic circuit is used to perform either addition or subtraction based on the value applied to Add / Sub input control line.

The line is set to 0 for addition, applying the Y vector unchanged to one of the adder inputs along with a carry-in signal, C_0, of 0.

When the add/sub control line is set to 1, the Y vector is 1's complemented (that is bit complemented) by XOR gates and C_0 is set to 1 to complete the 2's complementation of Y.

An XOR gate can be added to detect the overflow condition $C_n \oplus C_{n-1}$.



Fast Adders Circuit :-

In an n-bit parallel adder (ripple carry adder), there is too much propagation delay in developing the outputs.

Delay is not acceptable, in comparison with the speed of other processor components and speed of data transfers between registers & cache memories.

The delay through the circuit depends on number of gates in the path from input to outputs. In case of the ripple carry adder, the longest path is from inputs at the LSB position to output at the MSB position.

The propagation delay can be reduced by using a carry look-ahead adder in ALU.

* Carry Look ahead adder.

Carry look ahead Adder :- (Fast Adders)

The Main purpose of design fast adders is to reduce the time required to 'carry' signals from input to output. One way to Compute the input carry needed by stage i directly from a Carry like 'signals obtained from all the preceeding stages i-1, i-2 ... 0, rather than waiting for normal carries to ripple slowly from stage to stage. Adders that uses this principle is called Carry :- Look ahead 'adder.) Definition

A n bit Carry look ahead adder is formed from n stages, each of which is basically a full adder modified by replacing its Carry output line c_i by two auxillary signal called g_i and P_i called generate & propagate.

A fast adder circuit must speed up the generation of the Carry signal. the logical expression for Sum (S_i) and Carry C_{i+1} (carry out) of stage i is

$$S_i = x_i \oplus y_i \oplus c_i \quad \text{— (1)}$$

and

$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i \quad \text{— (2)}$$

factoring (2) nd eqn

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

We can write

$$C_{i+1} = G_i + P_i C_i \quad \text{— (3)}$$

Where

$$G_i = x_i y_i \quad \text{— (4)}$$

$$P_i = x_i + y_i \quad \text{— (5)}$$

(These two expression 4 & 5 is called as Generate and propagate function of stage i.)

If generate function is 1, then $C_{i+1} = 1$ independent of 'c_i' which is the input carry. This occurs when x_i & y_i are 1.

The propagate function will produce output carry when either x_i or y_i is 1.

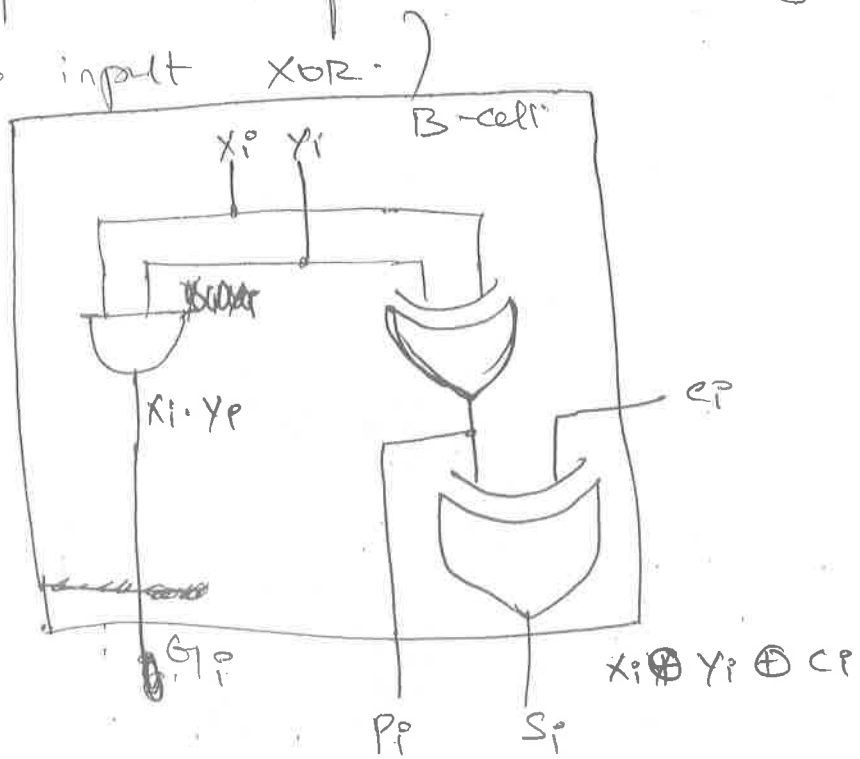
All G_i & P_i function can be formed independently and in parallel in one logic gate delay after x and y therefore are

applied to the inputs of an n-bit adder.

Each Bit Stage contain an AND Gate to form G_i, an OR gate to for P_i and three input XOR gate to form S_i.

A Simple Circuit can be devised called B-cell, where $P_i = x_i \oplus y_i$ which differs $P_i = x_i + y_i$ when $x_i = y_i = 1$.

3 input is XOR function is realized to two input XOR.



from (3) eqn c_i can be obtained, by ~~replacing~~ subtracting 1 from i+1

$$c_i = G_{i-1} + P_{i-1} c_{i-1} \quad \text{---(6)}$$

Sub (6) in (3) we get

$$C_{i+1} = G_i + P_i C_i \longrightarrow \text{Level 1}$$

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1} \quad \text{—(7)} \quad \text{Level 2}$$

Sub the ~~value~~ expression of C_{i-1} in (7)

C_{i-1} is obtained from 6 eqn by substracting 1.

$$C_{i-1} = G_{i-2} + P_{i-2} C_{i-2} \quad \text{—(8)}$$

Sub 8 in (7) we get

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} C_{i-2} \quad \text{—(9)}$$

Continuing expanding the final expression for any Carry Variable is general form:

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} \cdots P_i P_{i-1} P_{i-2} \cdots P_0 C_0 \quad \text{—(10)}$$

Let Us consider design of 4 bit adder, the Carries are implemented as -

Consider eqn (3), (7), (9)

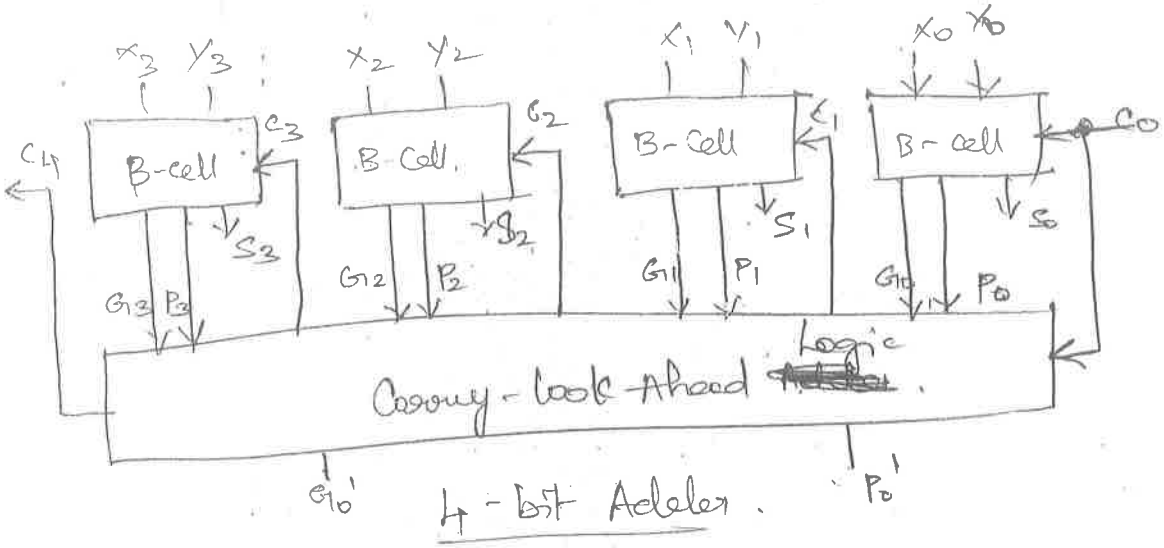
Sub $i=0$, $i=1$, $i=2$, $i=3$ respectively we get

$i=0$ in (3) $\Rightarrow C_1 = G_0 + P_0 C_0$

$i=1$ in (7) $\Rightarrow C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

$i=2$ in (9) $C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

$i=3$ $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$



4-bit Adder.

In This Diagram the carries are implemented in a block called "Carry Look ahead Logic". An adder implemented in this form is called Carry look ahead adder.

Delay through adder is 3 gate delay for all carry bits and 4 gat delay for all sum bits.

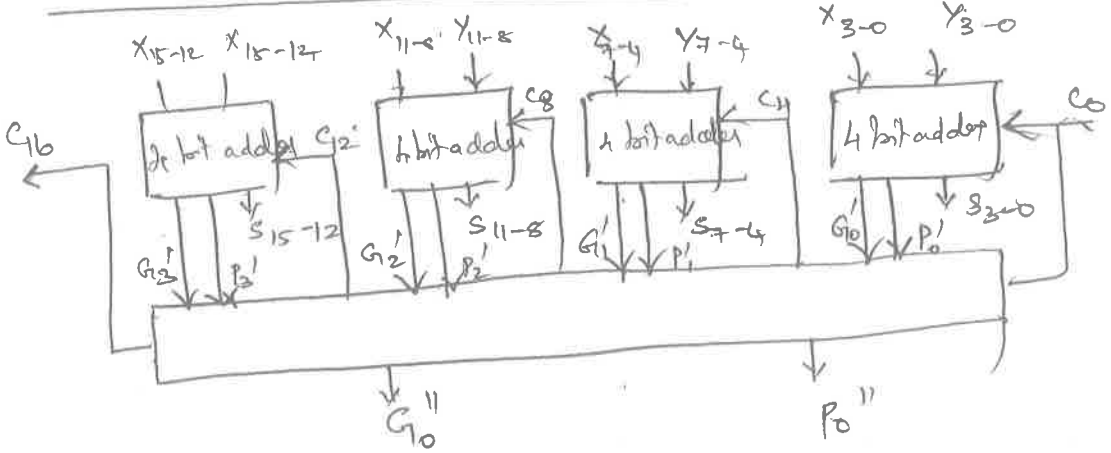
If we try to extend the carry-lookahead adder for more operands

We run into a problem called gate fan in* constraints. From the general expression we see that last AND and OR gate requires i+2 fan in in generating C_{i+1}. So this is a limitation. So the adder cannot be extended directly from the previous B-cells, so it is possible to build longer adders.

Fan in :- no of inputs that logic gate can accept, If input exceeds, the output will be undefined.

Eight 4-bit Carry-Look ahead adders can be connected to form a 32 bit adder. Inorder to extend adder, key idea is to generate carries C_4, C_8 ... in parallel similar way of C_1, C_2, C_3 & C_4.

(Higher Level Generate & propagate functions



This is a 16 bit adder built from four 4 bit adder blocks, these block provide new output function defined as G_k' and P_k' where $k = 0$ for 1st 4 bit block. $k = 1$ for second 4-bit block and so on.

In the first block,

$$P_0' = P_3 P_2 P_1 P_0 \quad \text{—— (A)}$$

$$G_0' = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \quad \text{— (B)}$$

∴ Carry C_{16} is formed by one of carry-lookahead circuits as

$$C_{16} = G_3' + P_3' G_2' + P_3 P_2 G_1' + P_3' P_2' P_1' G_0' + P_3' P_2' P_1' P_0' C_0$$

∴ We can extend the expands in the similar way.

A 64 bit adder can be built from four of the 16 bit adders.

∴ Delay through the carry lookahead adder is 3 gate delay for carry, 4 gate delay for sum, which is Less when compared to Ripple Carry adder.

Multiplication

Multiplication is a slightly more complex operation than addition or subtraction, being implemented by shifting as well as addition. Multiplying two n-bit values together can result in a value of up to 2n bits. Because of the partial products involved in most multiplication algorithms more time and more circuit area is required to compute, allocate, and sum the partial product to obtain the multiplication result.

Consider a normal Binary Multiplication

Multiplicand	1000_2	(8_{10})
Multiplier	1001_2	(9_{10})

```
        1000
       0000
      0000
     1000
-----------------
1001000
```

Product 1001000_2 $(72)_{10}$

$72/2$
$36 - 0$
$18 - 0$
$9 - 0$
$4 - 1$
$2 - 0$
$1 - 0$

The first operand is called the multiplicand, and the second operand is called as the multiplier.

In this example, we restricted ~~as the result of multiplying two 32 bit numbers~~ the decimal digits to 0 and 1. with only two choices; each step of multiplication is simple.

① Just place a copy of multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1 (or)

② Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Some of the highly optimized multiplication hardware are.

① Sequential Version (first Version) of the multiplication Algorithm & Hardware.

② Refined Version of the Multiplication Hardware

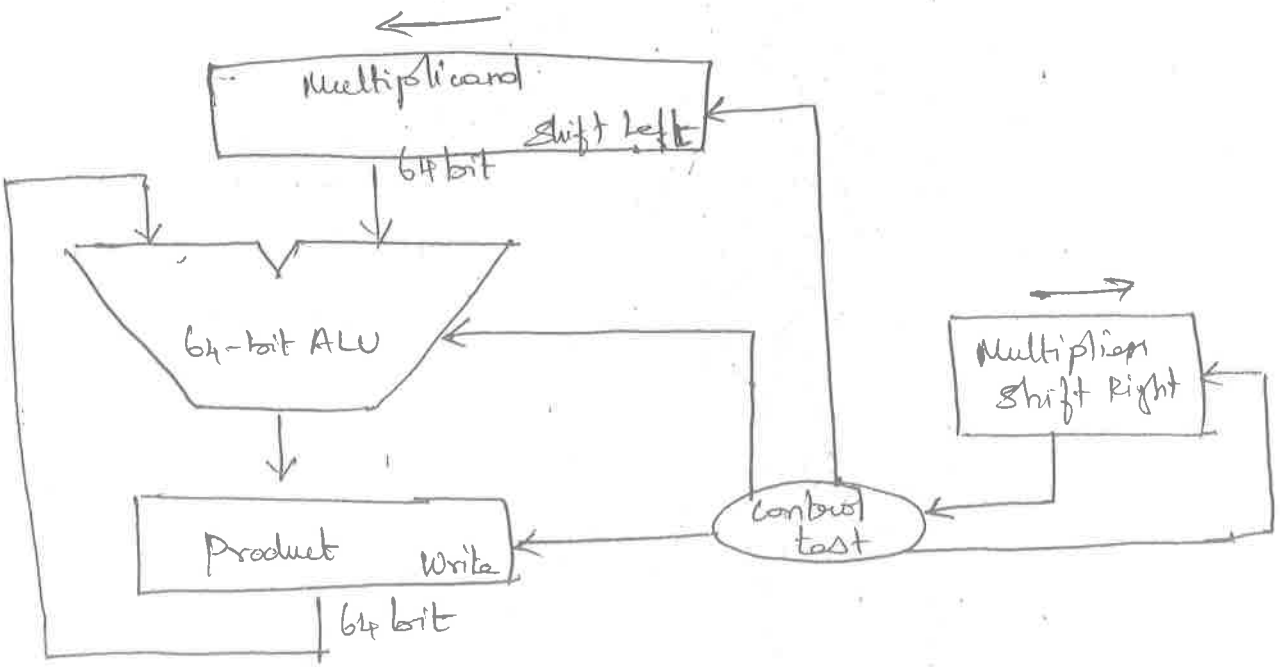
③ Signed Multiplication → (Booth's Algorithm)

④ Faster Multiplication → Carry Save adders

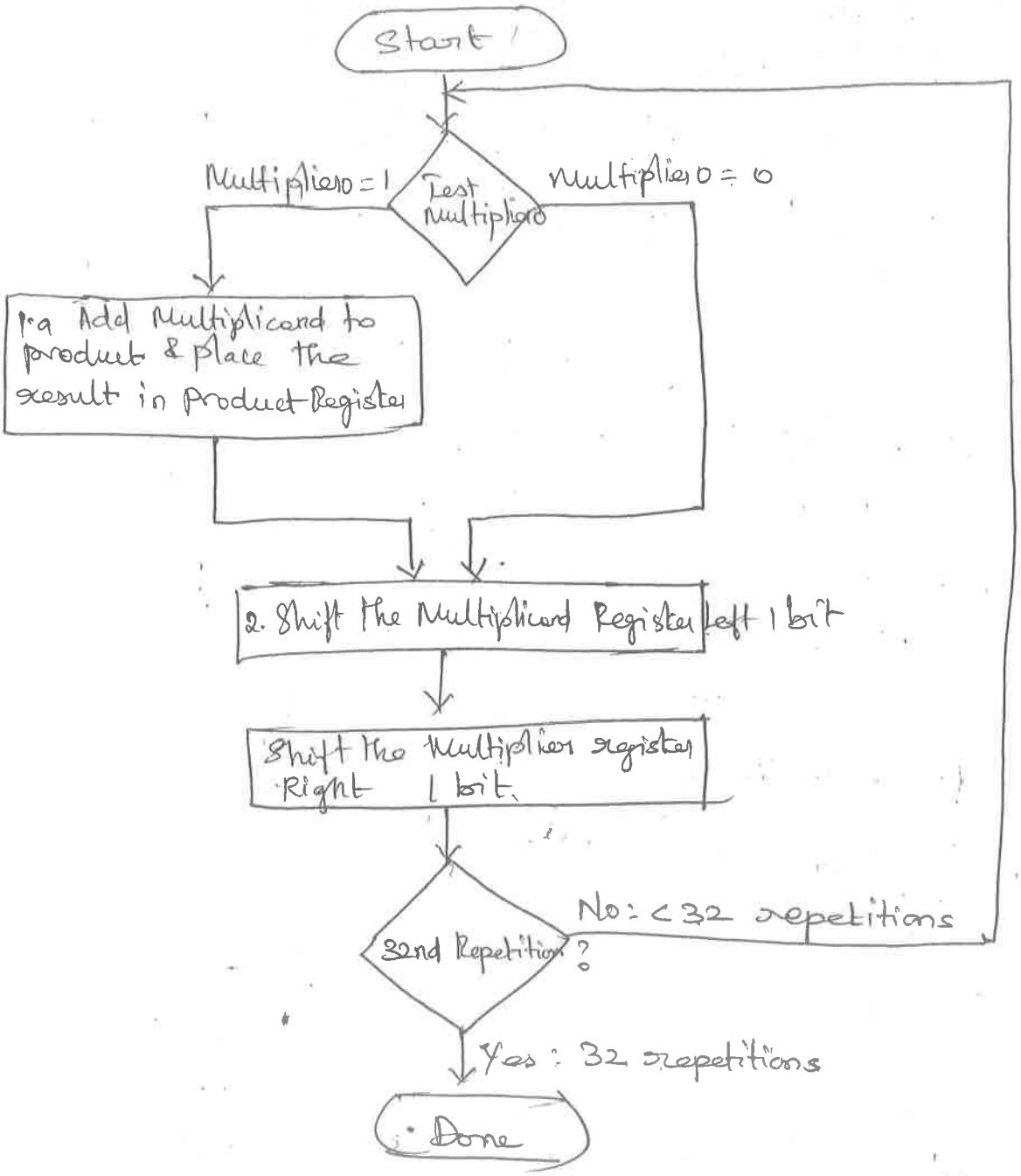
⑤ Multiply in MIPS.

Sequential Version of the Multiplication Algorithm:-

The hardware is made up of 32 bit multiplier register and 64 bit product register that is initialized to '0'. Over 32 steps a 32 bit multiplicand would move 32 bit to the left. Hence we need 64 bit multiplicand register initialized with 32 bit multiplicand in the right half and zero in the left half. The register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64 bit product register. The ALU is 64 bit wide to perform 64 bit addition.



The first Multiplication ~~Hardware~~ Algorithm using the hardware.



The three basic step needed to perform multiplication of each bit

(17)

Step 1: The LSB bit of the multiplier determines whether the multiplicand is added to the the product register.

Step 2: Shift the multiplicand by 1 bit left, that will have a effect of moving the intermediate operands to the left.

Step 3: Shift the multiplier right by 1 bit that will give the next bit of the multiplier to be examined.

These 3 steps are repeated for 32 times to obtain the product.

eg: Consider 4 bit Multiplication

$4_{10} \times 3_{10} = 12_{10}$ or $0100_2 \times 0011_2$

Iteration	Steps	Multiplier	Multiplicand	Product
~~Initial~~ 0	Initial Values	0011	0000 0100	0000 0000
1.	1. 1 → prod = prod + Mcand	0011	0000 0100	0000 0100
	2. Shift Left Multiplicand	0011	0000 1000	0000 0100
	3. Shift Right Multiplier	0001	0000 1000	0000 0100
2.	1. 1 → prod = prod + Mcand	0001	0000 1000	0000 1100
	2. Shift Left Multiplicand	0001	0001 0000	0000 1100
	3. Shift Right Multiplier	0000	0001 0000	0000 1100
3.	1. 0 => No operation	0000	0001 0000	0000 1100
	2. Shift Left Multiplicand	0000	0010 0000	0000 1100
	3. Shift Right Multiplier	0000	0010 0000	0000 1100
4	1. 0 => No operation	0000	0001 0000	0000 1100
	2. Shift Left Multiplicand	0000	0100 0000	0000 1100
	3. Shift Right Multiplier	0000	0100 0000	0000 1100

It is observed that

① Each step took a clock cycle.

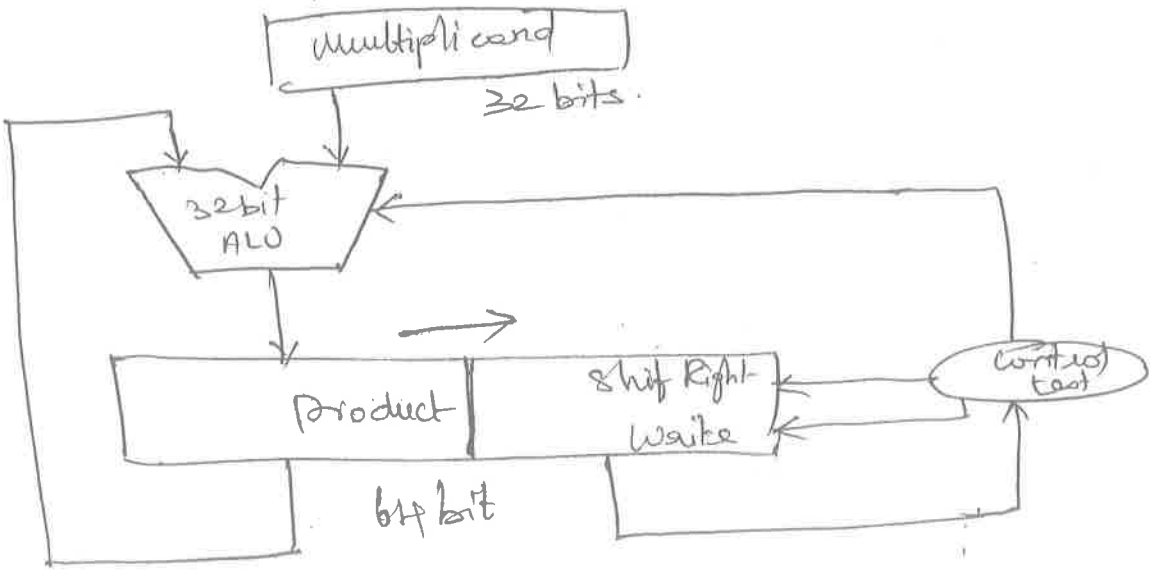
② Half bits in Multiplicand is always 0.
∴ 64-bit adder is wasted.

③ LSB of the product is never changed once formed, instead of shifting multiplicand to left, shift product to right :

These drawback were considered for the another version of the multiplier

Refined Version of the Multiplication Hardware :-

In comparison to the first Version, the Multiplicand Register, ALU and Multiplier Register are all 32 bit wide with only the product register is 64 bit. The product is shifted right. The separate Multiplier is disappeared and it is placed in the Right half of the product register.



Faster Multiplication :-

→ Fast Multiplier Hardware
→ Carry save adders

→ Moore's law provided so much more in resources that hardware designers can now build much faster Multiplication Hardware.

→ Rather than to use a single 32-bit adder 31 times, the hardware "Unrolls the loop" ie it uses 32 bit adder for each bit of Multiplier and Organize them inorder to minimize the delay.

→ It provides a 32 bit adder for each bit of Multiplier, one input is multiplicand `AND`ed with a multiplier bit and other is the output of a prior adder.

→ Straight forward approach is to connect the outputs of the adders on the right, to the inputs of the adders on the left making a stack of adders 32 bit high.

→ An alternate way to organize these 32 additions in a "parallel tree" as shown in

Faster Multiplication :-

→ Fast Multiplier Hardware
→ Carry save adders

→ Moore's law provided so much more in resources that hardware designers can now build much faster Multiplication Hardware.

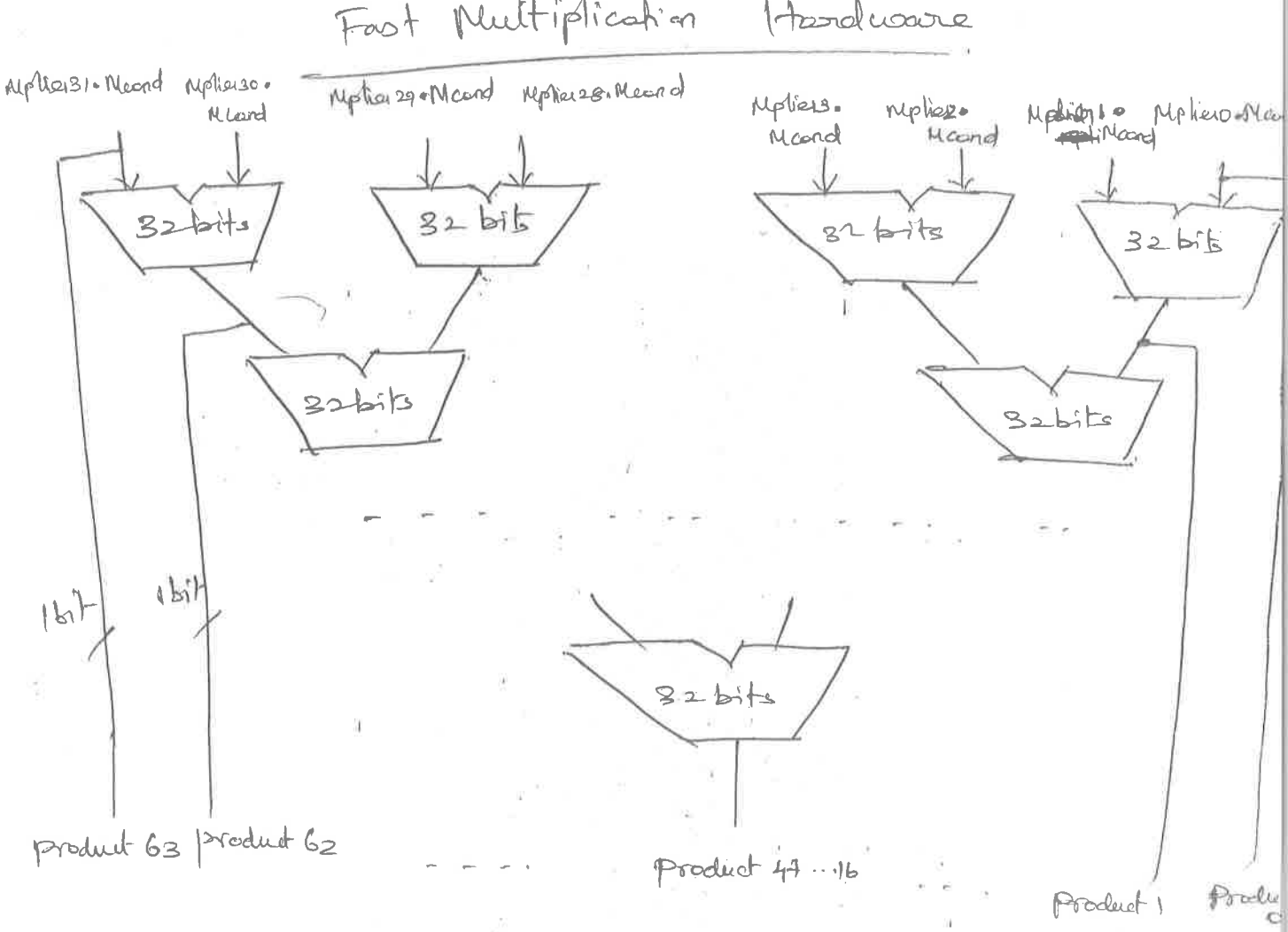
→ Rather than to use a single 32-bit adder 31 times, the hardware "Unrolls the loop" ie it uses 32 bit adder for each bit of Multiplier and Organize them inorder to minimize the delay.

→ It provides a 32 bit adder for each bit of Multiplier, one input is multiplicand 'AND'ed with a multiplier bit and other is the output of a prior adder.

→ Straight forward approach is to connect the outputs of the adders on the right, to the inputs of the adders on the left making a Stack of adders 32 bit high.

→ An alternate way to organize these 32 additions in a "parallel tree" as shown in

the diagram.



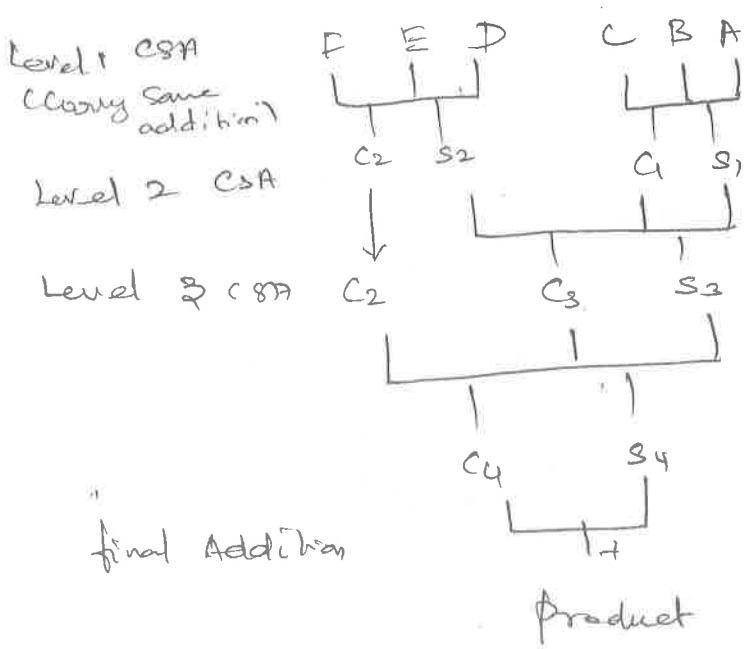
→ Instead of waiting 32 add times, we just wait $\log_2(32)$ or fine 32-bit add times.

* Multiplication can go even faster than fine add times because of the use of "Carry Save adders"

Carry Save Adders :- (for Multiplication)

"A technique called carry save addition (CSA) speeds up the addition process by saving the carry and introducing it at corrected weighted position. in the next row instead of letting the carry to ripple along the "row".

Schematic Representation of The carry save addition



→ (Summonds for 6 bit Multiplication)

eg. 1 1 1
1 0 1

A → 1 1 1
B → 0 0 0
C → 1 1 1

→ Summonds

Signed Multiplication :- (Booths Algorithms).

In signed number there is a need to take the sign of the number into consideration. The Easiest way to deal with signed number is first convert the multiplier and multiplicand to positive and then remember the original signs. The algorithm should then run for 32 iterations leaving the signs out of calculation.

→ We need to negate the product only if the original signs disagree (ie signs differ).

eg: Consider the case of a (1) positive multiplier & negative multiplicand, when we add a negative multiplicand to a partial product we must extend the sign-bit value of each multiplicand to the left as far as the product will extend.

eg: $-4_{10} \times 3_{10} \Rightarrow 1100_{10} \times 0011_{10}$

```
                 1100  (-4)
               x 0011  (3)
               ------------
Sign bit  <-- 1111 1100
extension.    1111 100
              0000 00
              0000 0
              ------------
              11111 0100  => -12₁₀
```

```
4  -> 0100
1s -> 1011
+1 ->    1
      ----
      1100

12 -> 01100
1s -> 10011
 1 ->     1
      -----
      10100
```

For negative Multiplier, a straight forward solution is to form the 2's complemet of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.

BOOTHS ALGORITHM :-

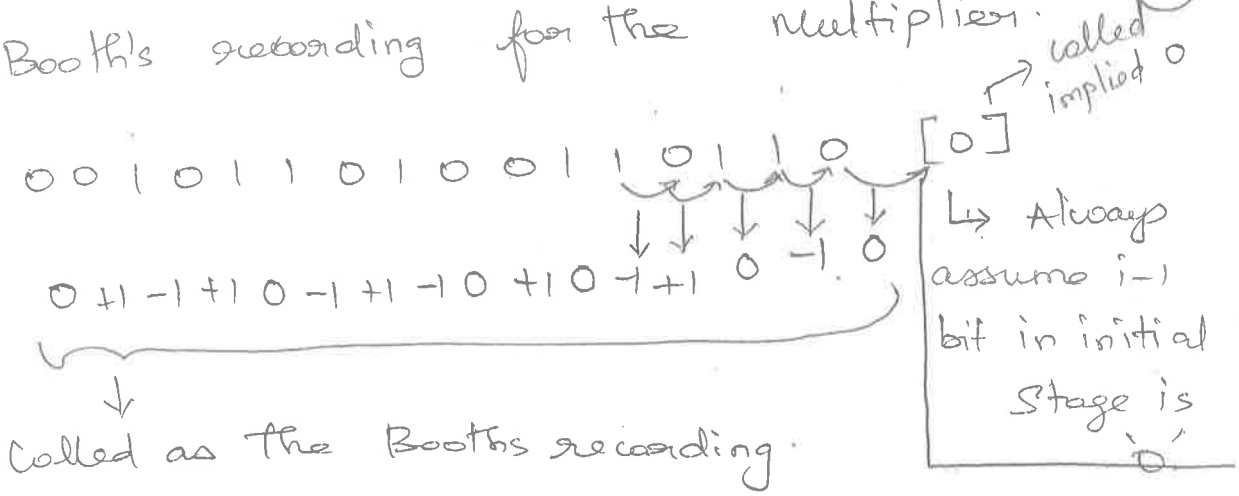
In order to take care of both positive and negative multipliers BOOTHS Algorithm can be used.

The Booth's Algorithm generates a 2n-bit product and treats both positive and negative 2's complement uniformly.

→ In the Booth's Scheme, booth's recording of a multiplier should be known.

Multiplier		Version of Multiplicand Selected by bit i
Bit i	Bit i-1	
0	0	0 × M
0	1	+1 × M
1	0	-1 × M
1	1	0 × M

eg: Booth's recording for the Multiplier.



⇒ Note: Join the two bits from Right to Left then look the Recording table for the two bit Combination if 0 1 [0] the 1 0 in table will correspond to "-1 × M" is the recording for 10.

Different Cases of Multiplier :-

i) Worst Case Multiplier

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

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

ii) Ordinary Case Multiplier

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

0 -1 0 0 +1 -1 +1 0 -1 +1 0 0 -1

iii) Good Multiplier

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

0 0 0 +1 0 0 0 0 -1 0 0 0

eg: Recode the Multiplier 101100 for Booth's Multiplication

Multiplier 1 0 1 1 0 0 [0] → Implied Zero

Recoded Multiplier

−1	+1	0	−1	0	0

"The speeding up of Multiplication is achieved depending upon the multiplier. If the Multiplier has few configuous blocks of 1's then Summands will be reduced, the speedup will be more.

eg:- Note ⇒ Always record the multiplier (if it +ve or Negative)

i) If the Multiplier is −ve and Multiplicand is +ve.

$6_{10} \times -5_{10} = -30$

$6_{10} \rightarrow$ 0 1 1 0

$-5 \rightarrow$ 1 0 1 1

−5 → 0101
1010
1011

Solution:-

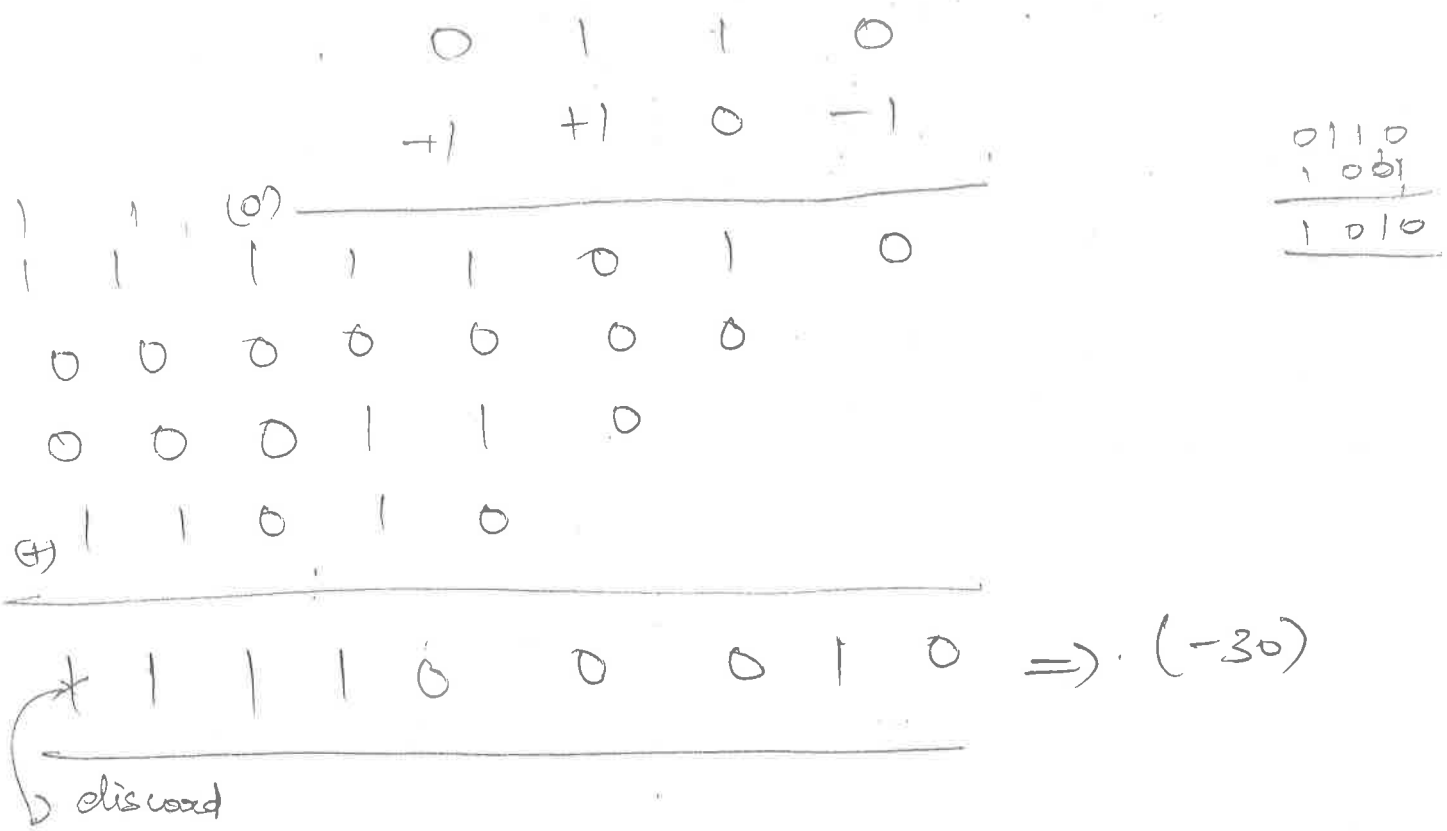
0 1 1 0 $(+6_{10})$ (Multiplicand)

1 0 1 1 (0) (-5_{10}) (Multiplier)

−1 +1 0 −1 (recorded Multiplier)

recorded Multiplier

Take the Multiplicand and recorded Multiplier and perform Multiplication.



0110
1001
1
1010

Check :- 30 → 00011110

1's comp 11100001

1
11100010 → (−30).

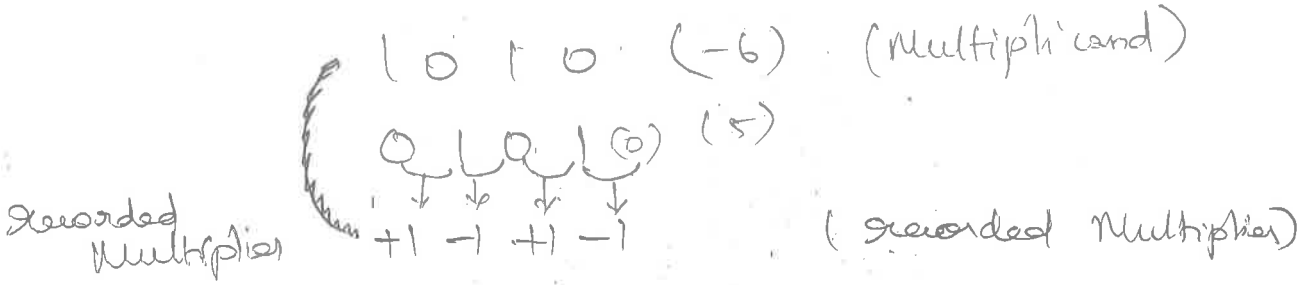
ii) If the Multiplicand is −ve and Multiplier is positive.

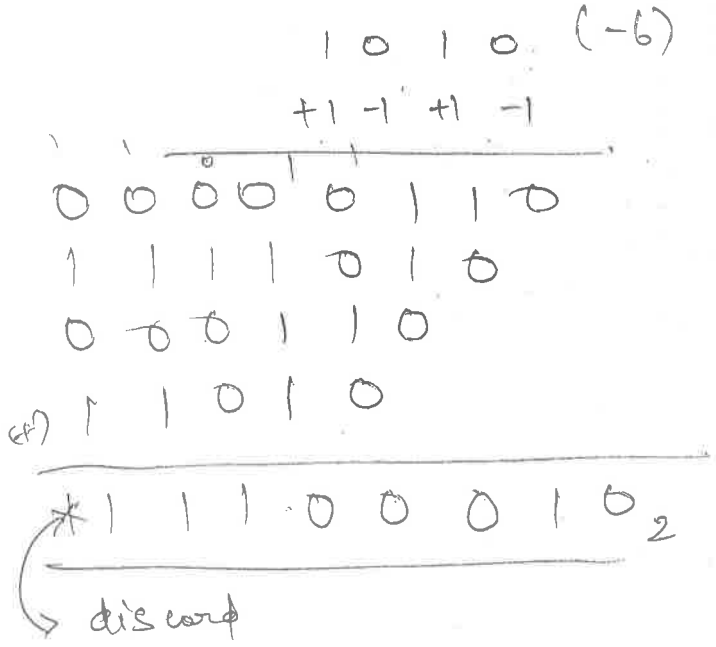
$-6 \times 5 = -30$
$_{10}\quad _{10}$

$-6 \rightarrow 1010_2$

$5 \rightarrow 0101_2$

0110
1001
1
1010





ii) If both the multiplier and multiplicand are negative:

$-6_{10} \times -5_{10} = +30_{10}$

-6)0110
1001
1
1010

$-6 \rightarrow 1010$ (Multiplicand)

$-5 \rightarrow (0\ 1)\ (0)$ (Multiplier)

-5)
0101
1010
1011

Booths recording of Multiplier $\longrightarrow -1\ +1\ 0\ -1$

1010_2 $(-6)_{10}$

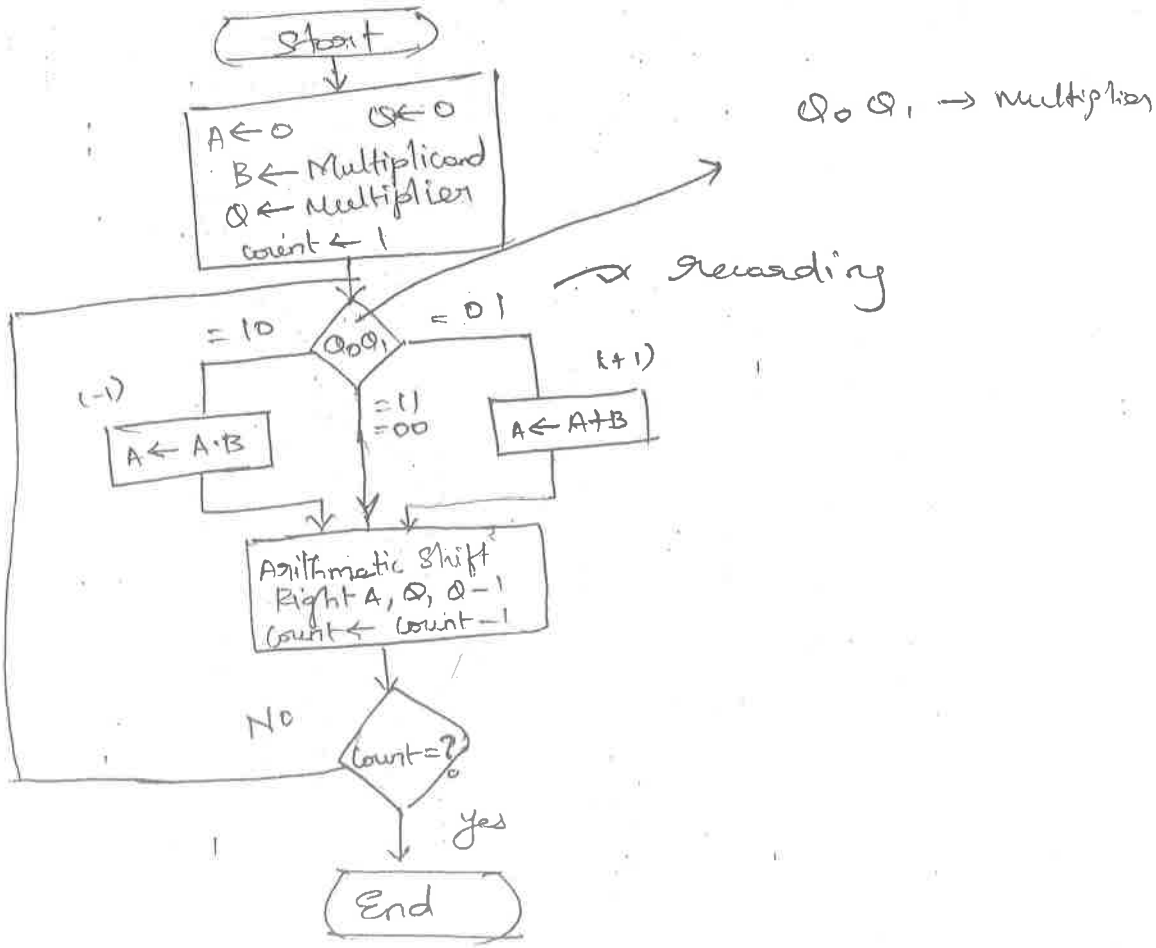
$-1+1\ 0-1$ (Booth's Recording of Multiplier)

```
  0000   0 1 1 0
  0000   0 0 0
  1110   1 0
+ 0011   1 0
-------------------
(1) 0001 1110
```

discard

$= (+30)_{10}$.

Flow Chart Booth's Algorithm for Signed Multiplication.



Features of Booths Algorithm :-

* It handles both positive and negative Multipliers uniformly.

* Second It achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1's.

Multiply in MIPS :-

MIPS has two instruction mult (multiply) and Multiply Unsigned (Multu). MIPS provide a seperate pair of 32 bit registers to hold 64 bit product.

Division

1. Division
2. First Version of the Division Hardware & Algorithm.
3. Refined Version of the Division Hardware.
4. Signed Division
5. Faster Division
6. Division in MIPS.

Division :-

→ The reciprocal operation of multiplication is divide.

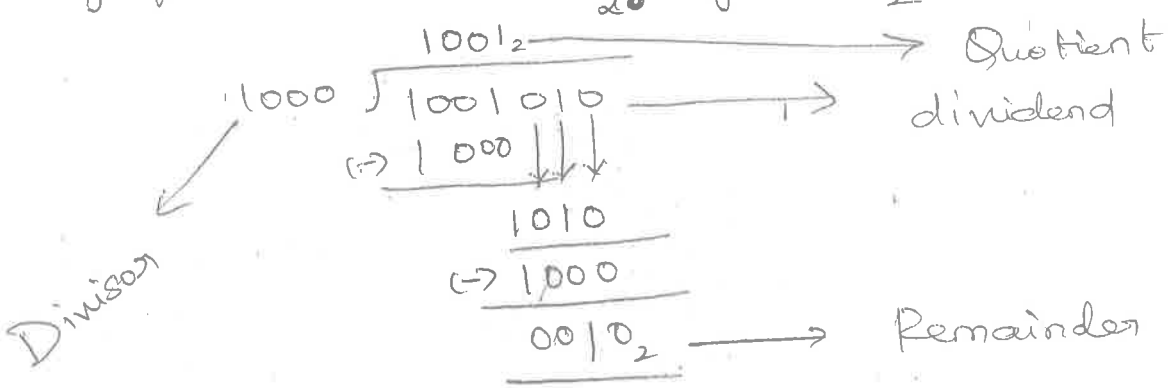
→ It is very complex similar to multiplication

→ Division can be achieved by successive subtraction.

eg :- Consider the binary division of

74_{10} by 8_{10}

binary form is 1001010_{20} by 1000_2



There are two operands called, the dividend and the divisor. The result is called as the Quotient, second result is called remainder

dividend - A number being divided.
divisor = A number that the dividend is divided by.
Quotient - The primary result of a division.
remainder - The secondary result of a division.

We can express the relation between these by,

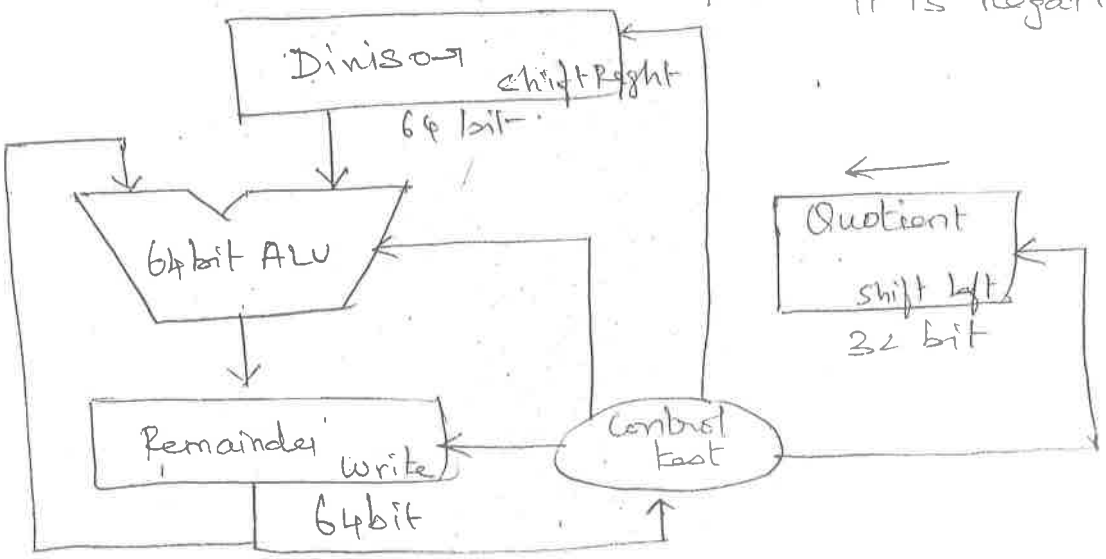
$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

where the remainder should be smaller than that of the divisor.

First Version of the Division Algorithm and Hardware :

Let us assume that the dividend and the divisor is positive, Hence the Quotient and the remainder is non-negative. The Hardware uses 32 bit Quotient Register and set to 0. It uses 64 bit ALU.

Each iteration of the algorithm needs to move the divisor to the right on digit, so the divisor is placed in the left half of 64 bit divisor register and shift it right 1 bit each step. The Remainder Register is initialized with the dividend. It is also called Restoring division because the remainder is restored when it is negative.



First Version of the division hardware

Steps involved in division Algorithm:-

The computer isn't smart enough to know in advance whether the divisor is smaller than the dividend.

Step 1: Test divisor < dividend (ie) Subtract the divisor from the dividend and place the result in the remainder register.

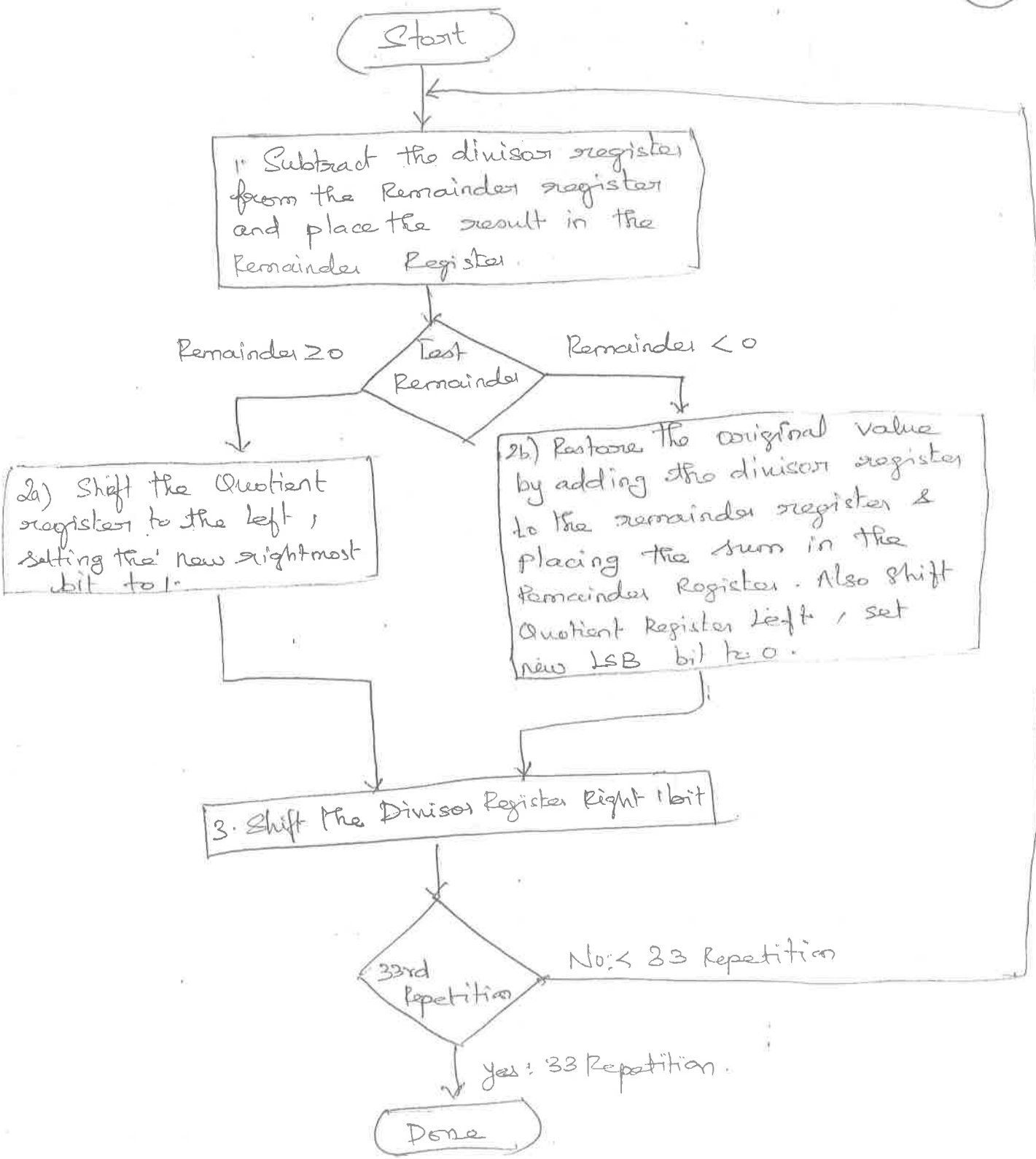
Step 2: a) if remainder $\geq$ 0 ~~divisor < dividend~~; shift the quotient register to the left setting the new right most bit to 1.

b) if remainder < 0, Restore the original value by adding the divisor to the dividend (Remainder Register) and placing the sum result in remainder register and also shift the Quotient register to left, setting the new least significant bit to '0'.

Step 3: Shift the divisor right by 1 bit.

These 3 steps are repeated for < 33 times to obtain the remainder and Quotient result.

The division algorithm, using the hardware is given below.



A division Algorithm, Using the hardware.

Eg:- divide 7_{10} by 2_{10}

the binary is $0000\ 0111_2$ by 0010_2

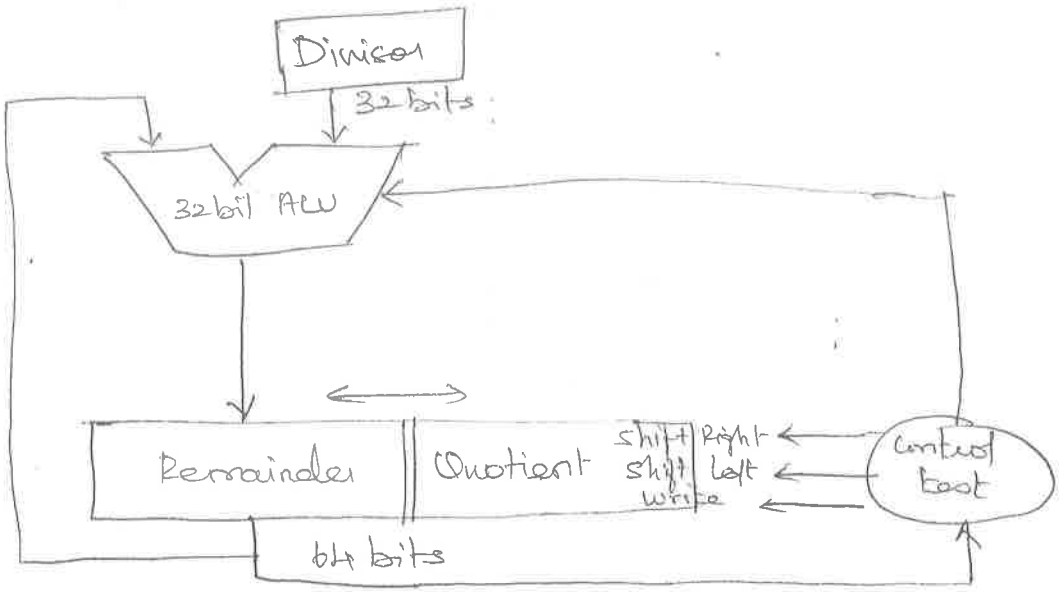
Iteration	Steps	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1.	1. Rem = Rem - Div 2.b Rem < 0 ⇒ +Div, sll Q, Q_0=0 3. Shift Div Right	0000 0000 0000	0010 0000 0010 0000 0001 0000	(1)110 0111 0000 0111 0000 0111
2.	1. Rem = Rem - Div 2.b Rem < 0 ⇒ +Div, sll Q, Q_0=0 3. Shift Div Right	0000 0000 0000	0001 0000 0001 0000 0000 1000	(1)111 0111 0000 0111 0000 0111
3.	1. Rem = Rem - Div 2.b Rem < 0 ⇒ +Div, sll Q, Q_0=0 3. Shift Div Right	0000 0000 0000	0000 1000 0000 1000 0000 0100	(1)111 1111 0000 0111 0000 0111
4.	1. Rem = Rem - Div 2.a Rem ≥ 0 ⇒ sll Q, Q_0=1 3. Shift Div Right	0000 0001 0001	0000 0100 0000 0100 0000 0010	(0)000 0011 0000 0011 0000 0011
5	1. Rem = Rem - Div 2.a Rem ≥ 0 ⇒ sll Q, Q_0=1 3. Shift Div Right	0000 0011 0011	0000 0010 0000 0010 0000 0001	(0)000 0001 0000 0001 0000 0001

A revised / Improved Version of Division Hardware :-

→ The Divisor Register, ALU and Quotient Register are all 32 bits wide

→ The Remainder Register alone is left as 64 bits

→ This Version also combines the Quotient register with the right half of the Remainder Register.



An Improved Version of the Division Hardware.

Signed Division :-

Remember the sign of the divisor and dividend and then negate the Quotient if signs disagrees (ie if they are not same).

eg: $-6_{10} \div 3_{10} = -2_{10}$ (since diff signs)

The one complication is that sign of the remainder.

> Dividend = Quotient x Divisor + Remainder

eg: Consider the combinations of $\pm 7_{10}$ by $\pm 2_{10}$ to find the sign.

(i) $+7 \div +2$: Quotient = +3 , Remainder = +1

Checking :-

$7 = 3 \times 2 + (+1) = 6 + 1$.

(ii) $+7 \div +2$: Quotient $= -3$,

Remainder can be calculated Using this

Rem = Divident $-$ (Quotient $\times$ Divisor)

$= -7 - (-3 \times +2) = -7 - (-6) = -1$

So

$-7 \div +2$ Quotient $= -3$, Remainder $= -1$.

here , -4 and Remainder $+1$ also fits this formula.

this anomalous behaviour is avoided by following the rule that the divident & Remainder must have same signs. no matter what the signs of the divisor or Quotient is.

$+7 \div -2$: Quotient $= -3$ Remainder $= +1$

$-7 \div -2$: Quotient $= +3$, Remainder $= -1$.

Faster Division :-

-> We can use adders to speed up multiply , but we cannot use the same for division. The reason is that we need to know the sign of the difference before performing the next step of the Algorithm.

→ There are techniques to produce more than one bit of the quotient per step.

(Sweeney, Robertson & Tocher) division

→ The SRT technique tries to guess several 4bit quotient bits per step using a table lookup based on the upper bits of the dividend and remainder.

→ The accuracy of this fast method depends on having proper values in the lookup table.

→ This technique tries to guess several 4 bit Quotient bits per step, using the lookup table based on upper bits of dividend and remainder.

→ These algorithm use 6 bits from the remainder and 4 bits from divisor to index a table.

Division in MIPS :-

* MIPS supports multiplication and division using the existing hardware, ALU & shifters.

→ It needs one extra hardware component - a 64 bit register able to support sll and sra instructions.

→ Signed division is supported by 'div' instruction and unsigned is by 'divu' instruction.

-> MIPS hardware doesn't check for division by zero. Thus 'divide by zero' exception must be handled in system software.

Non Restoring Division :-

2m (The restoring division algorithm can be improved by avoiding the need for restoring the Remainder Register 'A' after an unsuccessful subtraction.)

(Subtraction is unsuccessful if the result is negative.)

Remainder is in 'A' register.

nbit, Divisor is loaded in M register.

nbit dividend is loaded in to Register Q at the start of the operation.

Non Restoring division Algorithm :-

① Step 1 :- Do the following n times.

1. If the sign 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.
2. Now if the sign of A is 0, Set q_0 to 1 otherwise Set q_0 to 0.

Step 2 :- If the sign of A is 1, Add M to A.

∴ Step 2 is needed to leave the proper positive remainder in A at the end of the n cycles of step 1.

---

eg: $8_{10} \div 3_{10}$

$1000_2 \div 0011_2$.

Non-Restoring division :

	A	Q	
Initial step	00000 00011 → M.	1000	
Shift A & Q left	00001 A (−) 00011 M	000□	step 1
Sub M from A set q_0	11110	0000'	(step 2)
Shift A & Q left	11100	000□	step 1
Add M to A Set q_0	(+) 00011 11111	000[0]	(step 2)
Shif A & Q left	11110	000□	step 1
Add M Set q_0	(+) 00011 00001	0001	step 2
Shift A & Q left Sub M set q_0	00010 (−) 00011 11111	0010 Quotient 0010	step 1 step 2

Finally Restore Remainder

```
 1111
 0011
------
 0010  → Remainder.
------
```

Floating point

Floating point representations of decimal numbers are essential to scientific computation using scientific notation with a fraction (F) an exponent (E) and certain radix (R) in the form $F \times R^E$

Decimal numbers use radix of 10
eg: 1.2×10^{-2}.

Binary numbers use radix of 2.
eg: 1.1×2^1

Representation of floating point is not unique eg: 55.66 can be represented as

1. 5.566×10^1
2. 0.5566×10^2
3. 0.05566×10^3 and so on.

Some examples of ~~floating point fractions are~~ decimal point numbers are

$3.14159\ldots_{10}$ (Pi)

0.000000001_{10} or 1.0×10^{-9} → Scientific Notation

$3,155760000$ or 3.15576×10^9

Scientific Notation - It has a single digit to the left of the decimal point.

eg: $3.1459_{10} \times 10^3$

Normalized Notation:- A number is scientific and has no leading 0's is called Normalized number.

eg: $1.0_{10} \times 10^{-9}$ → (normalized)

10.0×10^{-10} & 0.1×10^{-8}

(Not Normalized)

Binary numbers can also be represented in scientific Notation

eg: $1.01010_2 \times (2^3)$ → base 2.

Floating point :-

It represents number in which the binary point is not fixed (because they are shifted to have one non zero digit to left) they are floating hence it is called floating point.

In binary the form is

$1.xxxxx_2 \times 2^{yyyy}$

2nd

Scientific notation for reals is normalized form offers 3 advantages :-

(1) It simplifies exchange of data that include floating - point numbers.

(2) It simplifies floating point arithmetic algorithms to know that numbers will always be in this form.

(3) It increases the accuracy of the numbers that can be stored in a word.

Floating - point Representation :-

A designer of a floating point representation must find compromise between the size of the fraction and the size of the exponent

There are three fields in floating - point representation

(1) Fraction - Value generally between 0 and 1

2) Exponent - It is numerical representation system of floating point

(3) Sign - This indicate the sign of the number if it is positive the bit is '0' if negative it is set to 1.

(ie) In 32 bit Representation :-

Here S the sign of floating point is 1 bit.

Exponent - Value including the sign of the exponent represented in 8 bit.

Fraction :- 23 bit number.

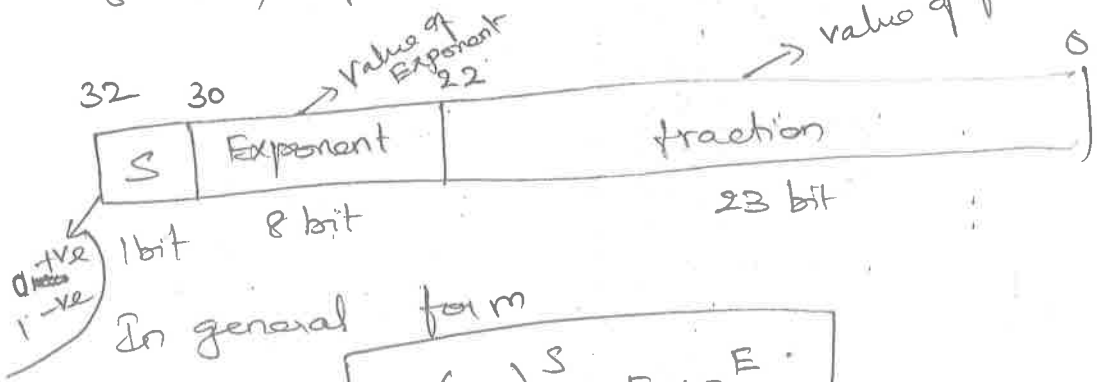
The representation used for Floating point is called "Sign & Magnitude" representation since sign is a seperate bit.

There are two representation :-

1. Single precision
2. Double Precision.

Single Precision :-

It uses a 32 bit word to represent a floating point value. Where S uses 1 bit, exponent uses 8 bit, fraction uses 23 bit.



In general form

$$(-1)^S \times F \times 2^E$$

Fractions almost as small as $2.0_{10} \times 10^{-38}$ and numbers almost as large as $2.0_{10} \times 10^{38}$ can be represented in computer.

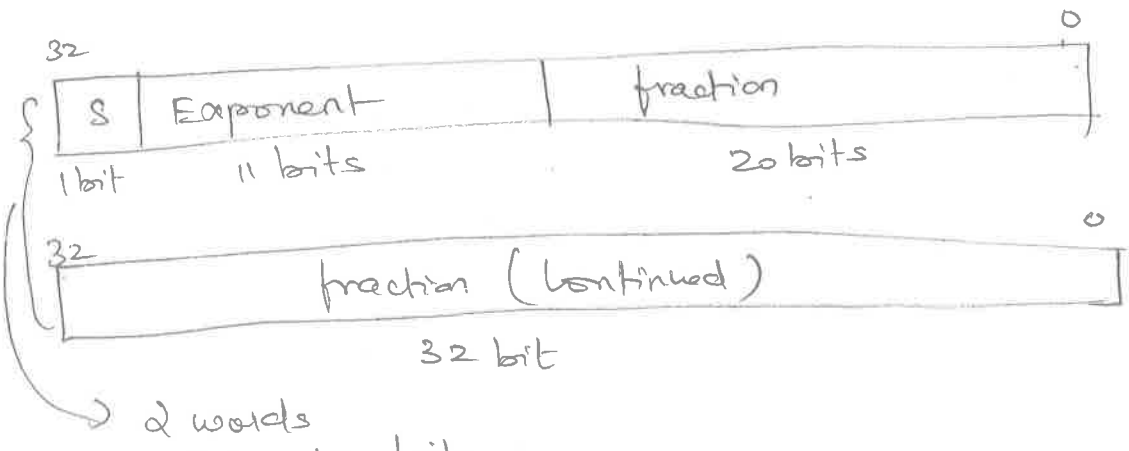
(31)

⇒ Overflow & underflow Interrupt can occur in the floating point arithmetic

Overflow :- A situation which positive exponent becomes too large to fit in the exponent field.

Underflow: A situation which negative exponent becomes too large to fit in the exponent field.

One way to reduce Overflow & underflow is to offer format that has larger exponent. So double precision is used eg: like double ~~to~~ datatype in c



The double floating point number takes two words. S is sign bit, exponent value of 11 bit exponent field, fraction is the 52 bit number in the fractional field.

It allows numbers almost as small as $2.0_{10} \times 10^{-308}$ and almost as large as $2.0_{10} \times 10^{308}$.

IEEE 754 Standard :-

To pack even more bits into the significant, it makes the leading 1-bit of normalized binary number implicit. Hence number is 24 bit long in single precision (implied 1 & 23 bit fraction) Similarly for double precision is 53 bit (1 implied, 52 fraction bits).

Thus $00\cdots00_2$ represent 0, the representation except for 0 is

$$(-1)^S \times (1 + \text{fraction}) \times 2^E$$