Unit-II

D. Venkata Vara Prasad

Session Objectives

- To explain the fixed point addition and subtraction operation handled by the processor.
- To explain the fixed point Multiplication operation handled by the processor
- To explain the fixed point Division operation handled by the processor
- To explain the floating point addition/subtraction operation handled by the processor
- To explain the floating point Multiplication operation handled by the processor
- To explain the floating point Division operation handled by the processor

ARITHMETIC AND LOGIC UNIT

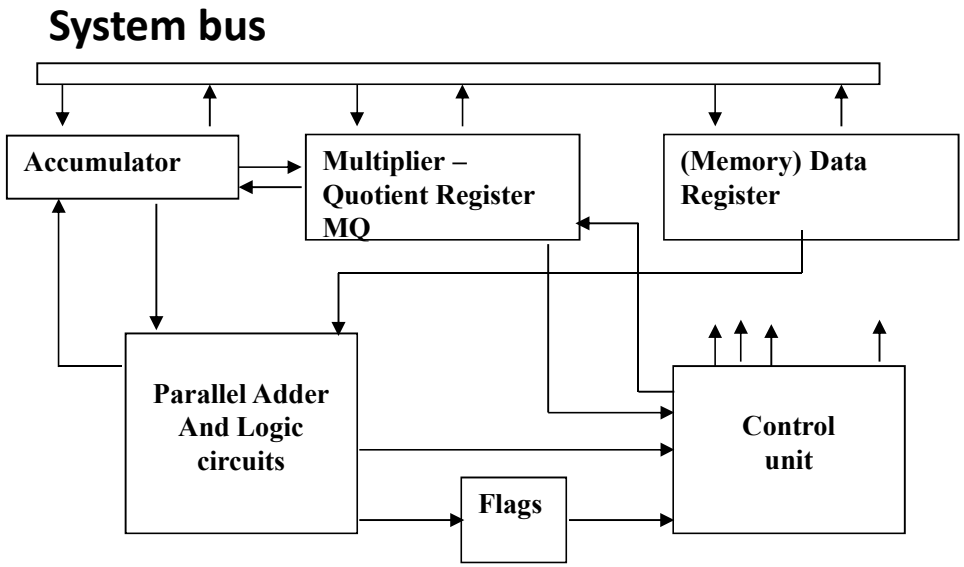
- The ALU performs arithmetic and logical operations on data.
- Data is stored in registers and the results of an operation are stored in registers.
- The control unit coordinates the operations of the ALU and the movement of the data in and out of the ALU.
- Most computers have registers called Accumulator(AC), or general purpose registers.
- The accumulator is the basic register containing one of the operand during operation in ALU.
- If the operation is add, the number stored in the AC is augend and the addend will be located and these operands (addend & augend) are added up and the result is stored back in the AC.
- The original augend will be lost in AC after addition.

Arithmetic operations

- The 4 basic arithmetic operations are
  - Addition
  - Subtraction
  - Multiplication
  - Division

  - The arithmetic operations are
    - Fixed point Arithmetic
    - Floating point Arthritic

Fixed point ALU



Block diagram of fixed point ALU

Fixed point ALU

- Three one word registers are used for operand storage:
  - Accumulator AC,
  - Multiplier Quotient register MQ and
  - Data register DR.
  - parallel adder that receives inputs from AC and DR and places its results in AC
  - Ac and MQ are organized as a single register AC.MQ capable of left and right shifting
  - MQ register is so called because it stores the multiplier during the multiplication operation, and quotient during the division operation.
  - DR stores the multiplicand or the divisor while the result is stored in AC.MQ.

Fixed point ALU

- Addition AC ← AC + DR
- Subtraction AC ← AC - DR
- Multiplication AC.MQ ← DR * MQ
- Division AC.MQ ← MQ / DR
- AND AC ← AC ^ DR
- OR AC ←AC V DR
- XOR AC ←AC .XOR. DR
- NOT AC ← AC *

Bit -sliced ALU

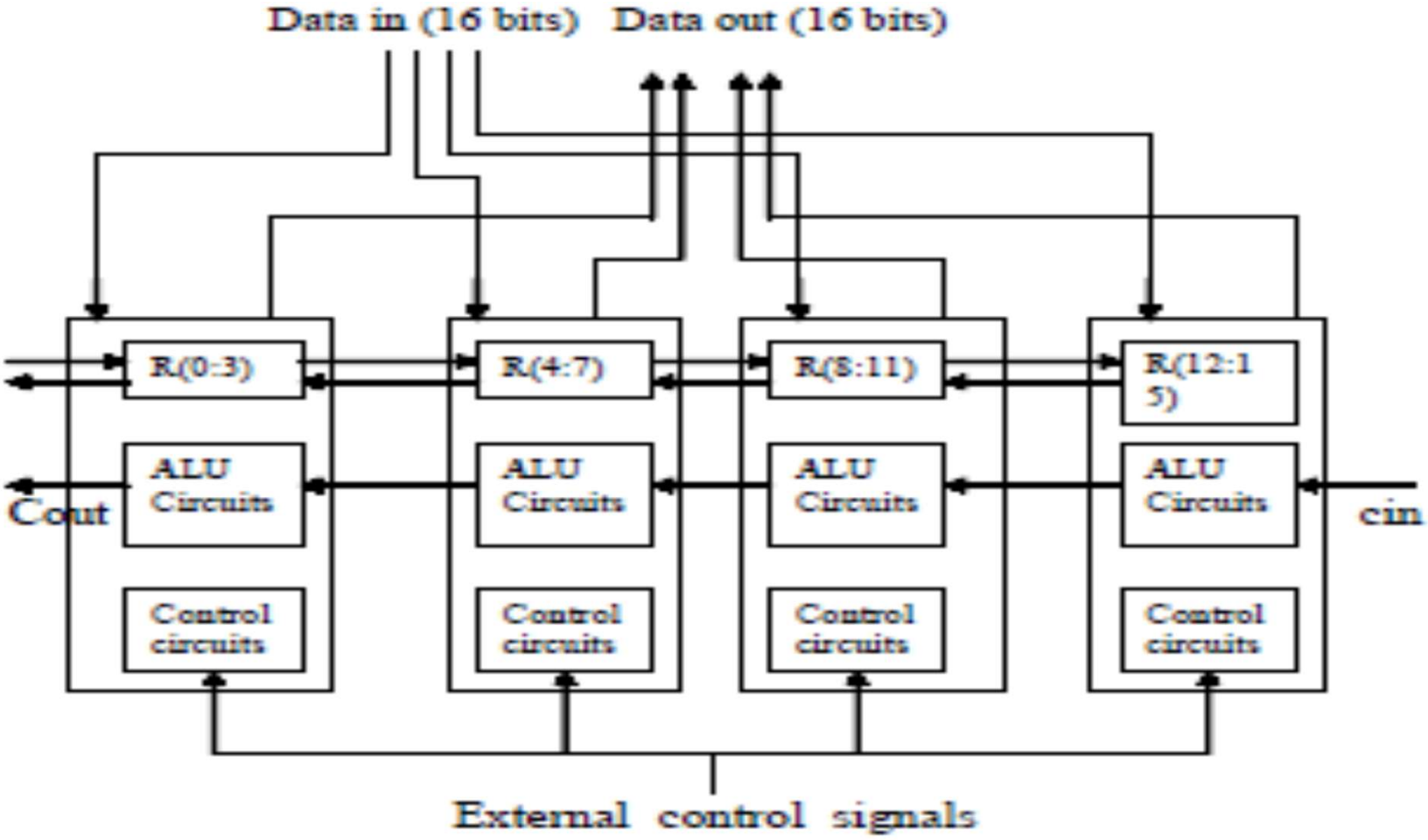


Fig: Bit Sliced ALU

Bit -sliced ALU

- Bit Sliced ALU
  - construct an entire fixed point ALU on a single IC chip if the word size **m** is kept small ex: 4 or 8 bits.
  - **m-bit ALU** can be designed to be expandable in that **k** copies of the ALU , can be connected to form a single ALU processing **km**-bit operands.
  - This array like circuit is called **bit sliced ALU** because each component chip processes an independent slice of m bits from each km bit operand.

- **Advantage** :
- any desired word size or even several different word sizes can be handled by selecting the appropriate number of components (bit slices).

Bit -sliced ALU

- Data buses & registers of the individual slices are placed to increase their sizes from 4 to 16 bits.
- The control lines that select and sequence the operation to be performed are connected to every slice.
- so that all slices execute the same operation in step with one another.
- Each slice performs the same operation on a different 4-bit part (slice) of the input operands ,and produces only the corresponding part of the results.
- Certain operations require information to be exchanged between slices.
- Ex:
  - shift operation to be implemented then each slice must send a bit to and receive a bit from left or right neighbors
  - addition the carry bits may have to be transmitted between the neighboring slices.

INTEGER REPRESENTATION FIXED POINT

- **Signed-Magnitude Representation**
- **Two's Complement Representation**

Signed-Magnitude Representation

- **+ve and –ve** numbers are differentiated by most significant bit in the word as a **sign bit**.
  - If the sign bit is 0, the number is positive
  - If the sign bit is 1, the number is negative
- +18 = 0 0 0 1 0 0 1 0

  ↓ ↓

  one sign bit seven bit magnitude

- Range : $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
  - Eg. A 7 bit register can store numbers from –63 to +63
- **Drawbacks**

1. Addition and subtraction requires both the sign bit and magnitude bits to be considered.
2. Two representation for zero (0)

- +0 → 0 0 0 0 0 0 0 0
- –0 → 1 0 0 0 0 0 0 0

Two's Complement Representation

- In two's complement system, forming the 2's complement of a number.
- Eg. Representation of –4
  - In Sign-magnitude 1 1 0 0
  - In 1's complement 1 0 1 1
  - In 2's complement 1 1 0 0
- **Advantages**

i. Only one arithmetic operation is required while subtracting using 2's complement notation.

ii. Used in arithmetic applications

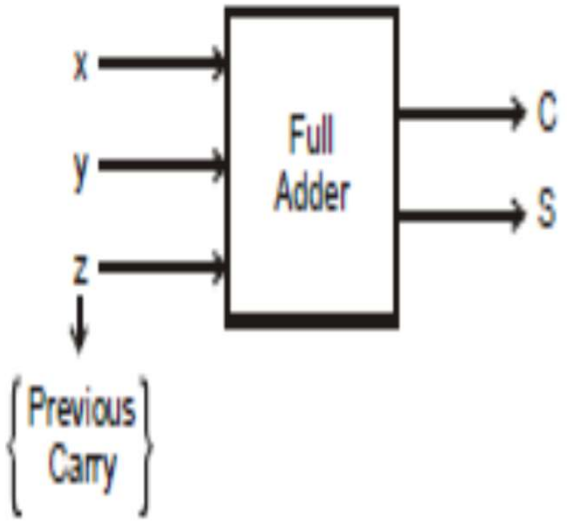
Sign Extension

- **Sign Extensions**
- Sometimes it is desirable to take an **n-bit** integer and store it in **m bits**, where **m > n**.
- **In sign-magnitude**:
- simply move the sign bit to the new leftmost position and fill in with zeros.
- **In 2's complement :**
- Move the sign bit to the new leftmost position and fill it with copies of the sign bit.
  - For +ve no's fill it with zeros.
  - For –ve no's fill it with ones.
- Eg.

+18 = 0 0 0 1 0 0 1 0 ← 8 bit notation
= 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 ← 16 bit notation in sign magnitude
–18 = 1 1 1 0 1 1 1 0 ← 2's complement 8 bit notation
111111111101110 ← 2's complement 16 bit notation

Full Adder

- Full adder is a combinational circuit that performs the sum of three input bits.
- It consists of three inputs and two outputs.
- Two input variables denoted by x and y, represent the two significant bits to be added.
- Third input, z represents the carry from the previous lower significant position.
- The two outputs are designated by symbols S for sum and C for carry.



Full Adder

Truth Table

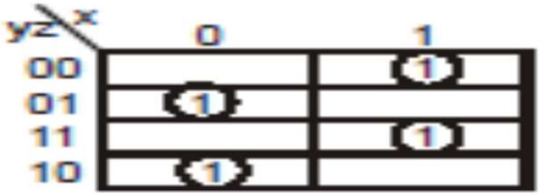
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder

Hardware Implementations

When k-map is drawn for S and C, we have

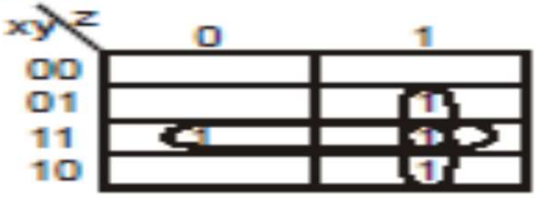
For S



$$\therefore \quad S = x\bar{y}\bar{z} + \bar{x}\bar{y}z + xyz + \bar{x}y\bar{z}$$

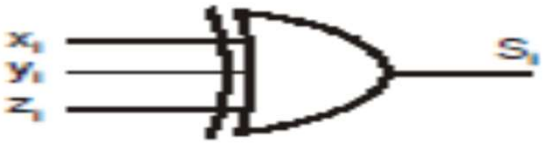
$$S = x \oplus y \oplus z$$

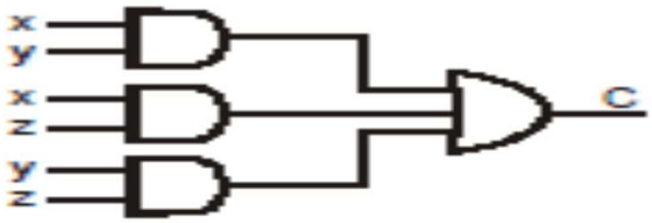
For C



$$C = xy + xz + yz$$

z is carry-in, therefore





4-bit binary adder

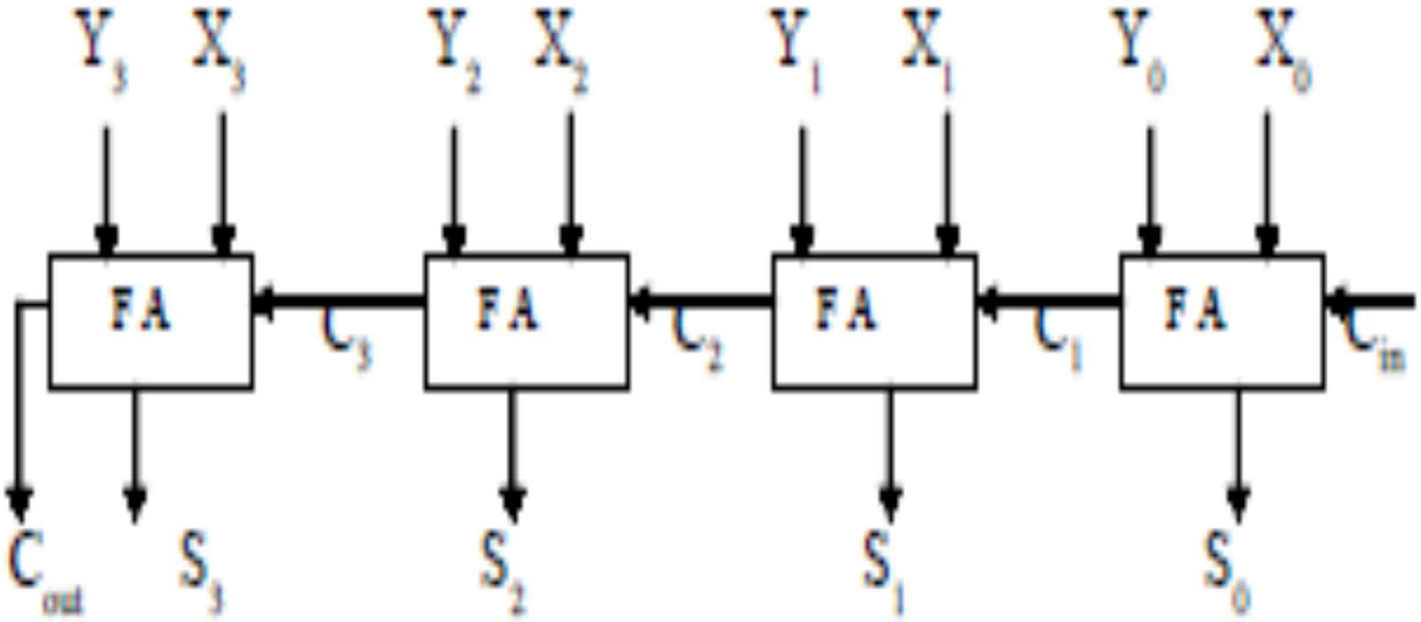


Fig: 4-Bit Binary Adder

4-bit binary adder

- Digital circuit that generates the arithmetic sum of two binary numbers of any length is called the binary adder.
- Binary adder circuit can be constructed with full adder circuits connected in cascade.
- Output carry from one full adder can be connected to the input carry of the next full adder.
- Augends bits of X and the Addend bits of Y are designated by subscript numbers from right to left.
- Carries are connected in chain through to full adders.
- Input carry to the binary adder is the C0 and the output carry to the binary adder is the C4.
- The S outputs of the full adders generates the required sum bits.

4-bit binary adder

- The n-bit binary adder requires n-full adders.
- The n-data bits for the X inputs are from one register R1 and n-data bits for the Y input come from another register R2.
- The sum can be transferred to the third register or one of the source register. (R1 or R2)

4- bit Binary adder / sub tractor

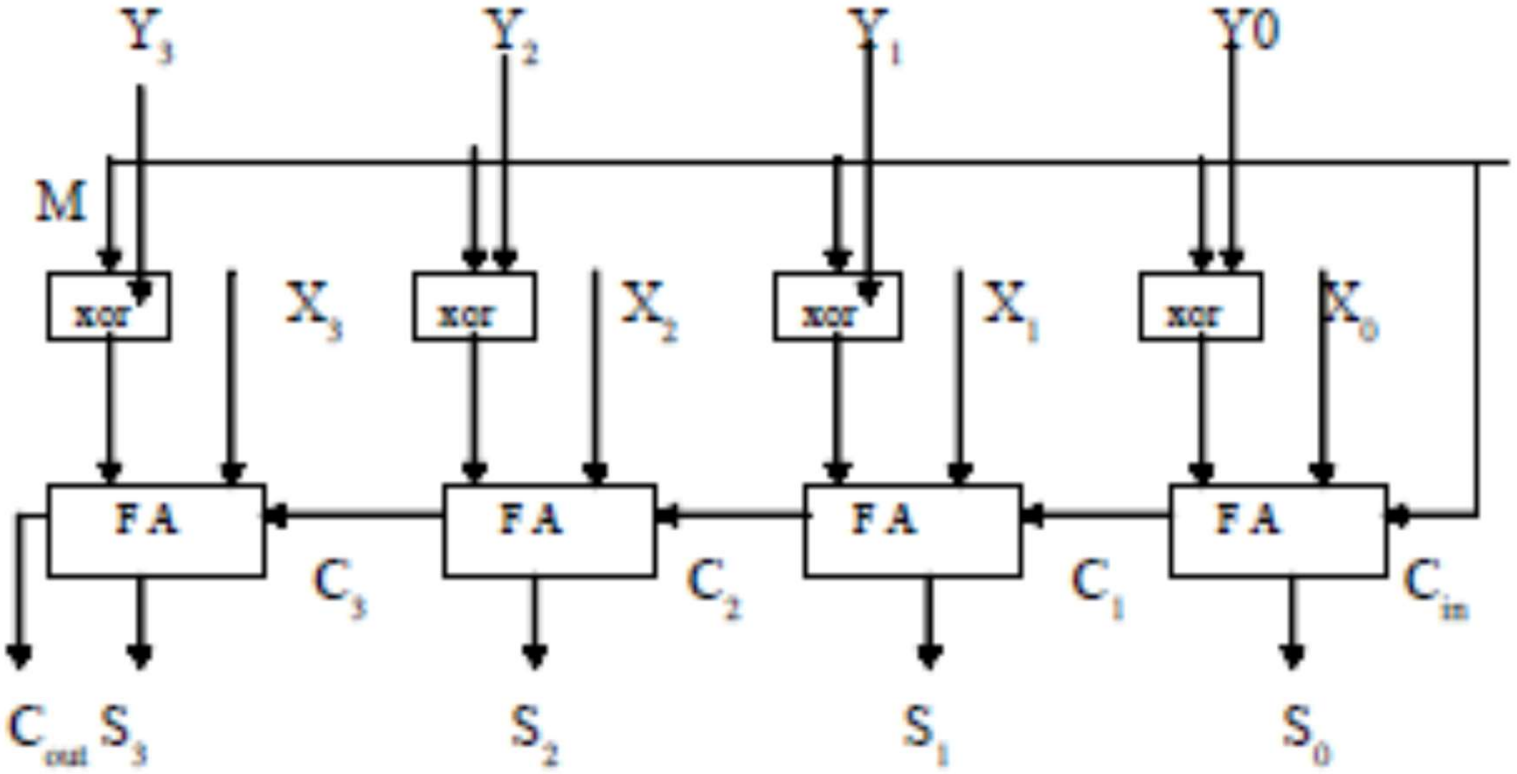


Fig: Binary Adder/ Sub tractor

4- bit Binary adder / sub tractor

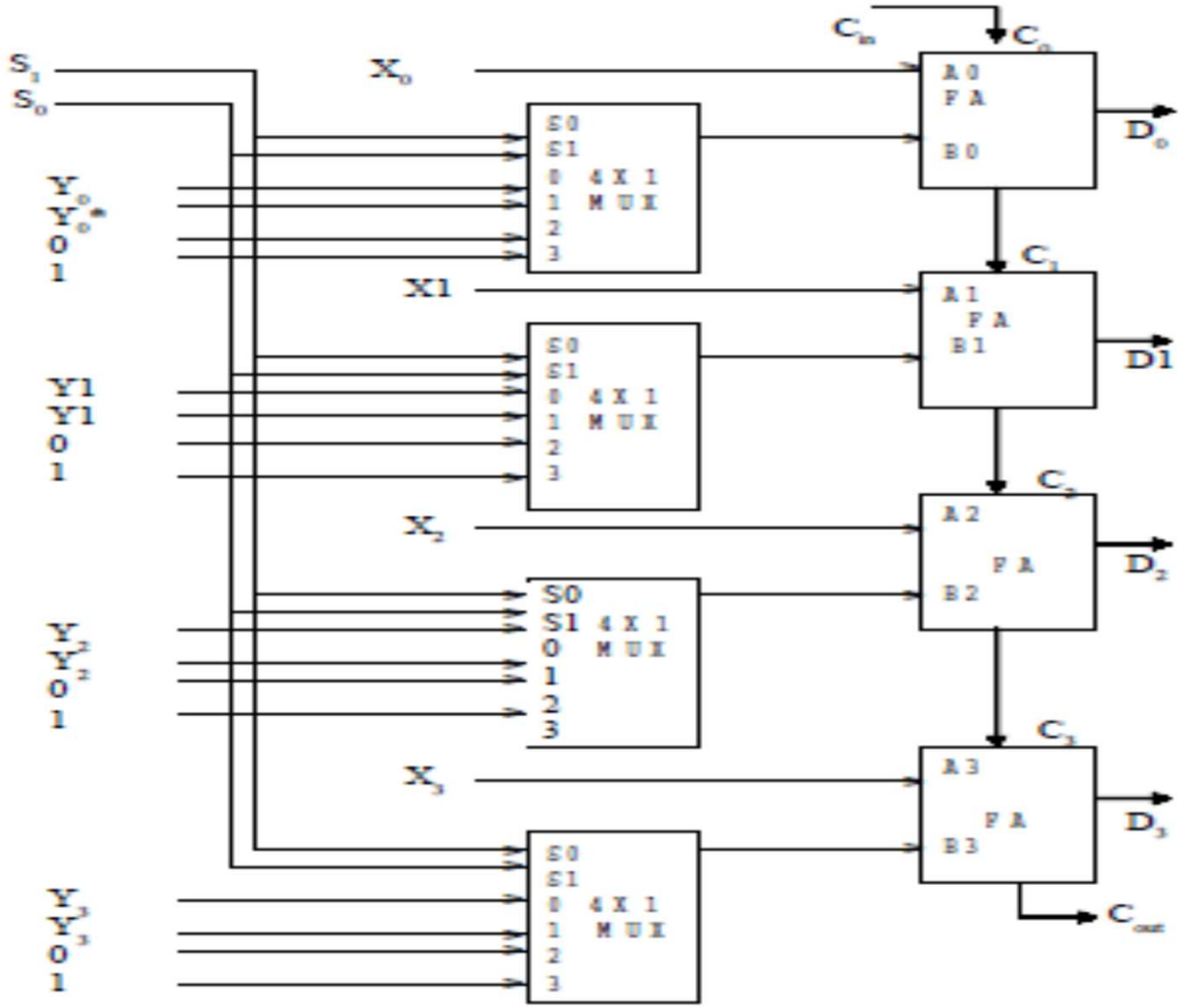
- The subtraction of binary numbers can be done by means of 2's complements.
- Addition and Subtraction can be combined in one circuit by including the **XOR** gate with each full adder.
- The mode input M controls the operation.

  If M = 0 → Adder,

  If M = 1 → subtractor.
- Each XOR gate receives input M and one of the inputs of Y.
- When M = 0 , Y **XOR** 0 = Y. The circuit performs the X + Y.
- When M = 1 . Y **XOR** 1 = $\overline{Y}$ and C0 = 1. The circuit performs X-Y

  i.e X plus 2's complement of Y.

4-bit binary arithmetic circuit



4-bit binary arithmetic circuit

- The 4-bit arithmetic circuit has:
  - four full adders.
  - 4-bit adder and 4 MUX for choosing different operation.
  - There are two 4-bit inputs A and B and 4-bit output D.
  - The 4 inputs go directly to the X input of the binary adder.
  - the 4 inputs of the B are connected to the data input of the MUX.
  - The MUX data inputs also receive the complement of B.
  - The other two data inputs are connected to logic 0. and logic 1.
  - The 4 MUX are controlled by selection inputs S0 and S1.
  - The input carry Cin goes to the carry input of the FA in the LSB.
  - The other carries are connected from one stage to another

4-bit binary arithmetic circuit

- D = X + B + Cin
  - X →4 bit binary number at A inputs.
  - Y → 4 bit binary number at B inputs.
  - Cin → Input carry which can be 0 or 1.

- When S_1S_0 = 00
  - the value of B is applied to the Y input of the adder.
  - If C_{in} = 0 the output D = X + Y.
  - If C_{in} = 1 then the output is D = X + Y + 1.
  - Both cases perform the add micro operation with or with out carry.

4-bit binary arithmetic circuit

- When S_1S_0 = 01
  - the complemented value of Y is applied to the B input of the adder.
  - If Cin = 0 the output D=X+(Comp)Y. This is equivalent to D = X–Y-1.
  - If Cin = 1 then the output is D = X+(Comp)Y+1.
  - This produces Xplus the 2's complement of Y, which is equivalent to X–Y.
  - Both cases perform the subtraction micro operation with or with out borrow.
- When S_1S_0 = 10
  - input from the B are neglected all 0's are applied to the B input
  - The output becomes D = X+1

4-bit binary arithmetic circuit

- When $S_1S_0 = 11$
  - all 1's are inserted in to B input of the adder to produce the Decrement operation. D = X-1, When Cin = 0.
  - This is because the number with all 1's equal to 2's complement of 1. (2's complement of 0001 = 1111). Adding a number A to the 2's complement of 1 produce D= X-1.
  - When Cin = 1 then D = X-1+1 = X

Addition/Subtraction

- When the signs of X and Y are identical add the two magnitudes and attach the sign of X to the result.
- When the signs of X and Y are different compare the magnitude and subtract the smaller number from the larger.
- Choose the sign of the result as X if X > Y or complement of the sign of X if X < Y.
- If the two magnitudes are equal, subtract Y from X and make the result positive

Addition/ Subtraction: Hardware Implementation

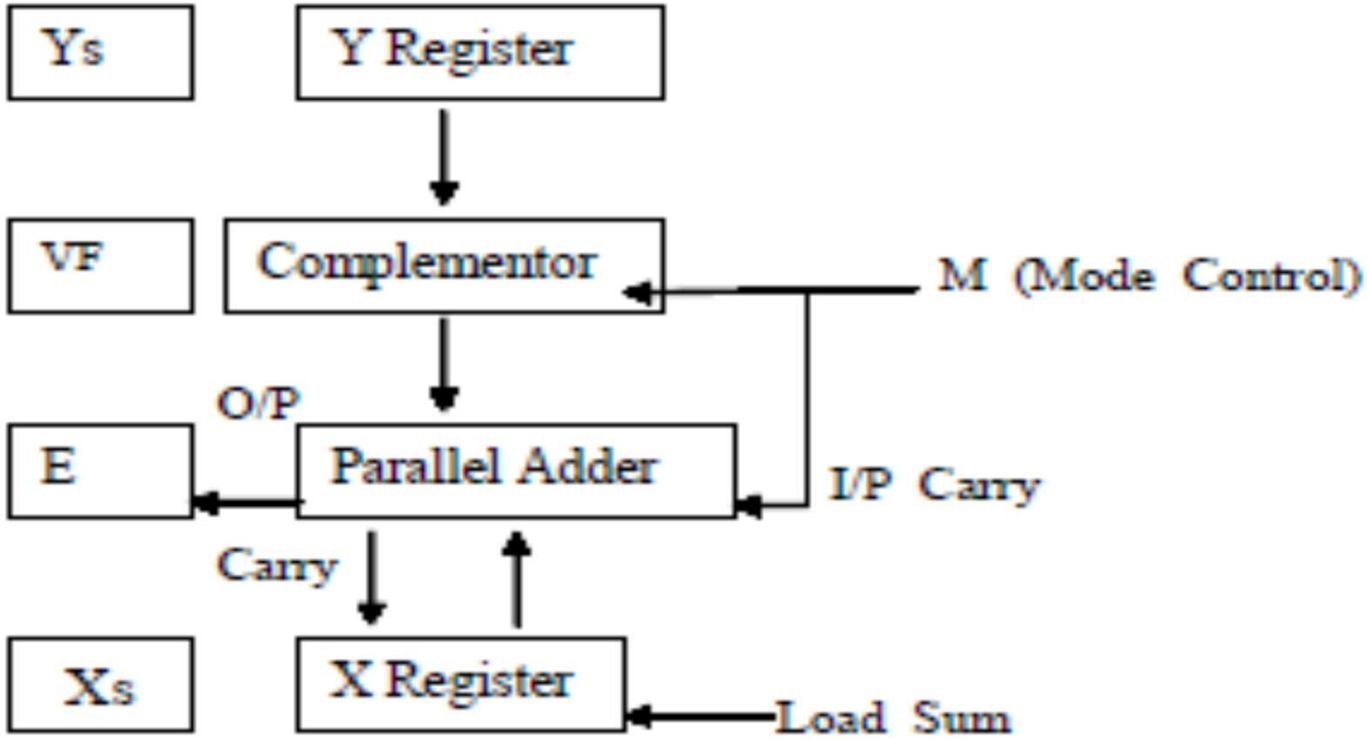


Fig:- H/W for Signed-Magnitude addition and Subtraction

Addition/ Subtraction: Hardware Implementation

- To implement the two arithmetic operations with hardware
  - It consists of registers X and Y and sign flip flop Xs and Ys.
  - Subtraction is done by adding X to the 2's complement of Y.
  - The output carry is transferred to flip flop E.
  - The add-overflow flip flop VF holds the overflow bit when X and Y are added.
  - The addition of X plus Y is done through the parallel adder.
  - The S (Sum) output of the adder is applied to the input of the X register.
  - The complement provides an output of Y or complement of Y depending on the state of mode control M.
  - The complementor consists of XOR gates and the parallel adder consists of full-adder. The M signal is applied to the input carry of the adder.

Addition/ Subtraction: Hardware Implementation

- When M = 0, the output of Y is transferred to the adder, the input carry is 0 and the output of the adder is the sum X + Y.
- When M=1, the 1's complement of Y is applied to the adder, the input carry is 1 and the output S = X + Y' +1. This is equal to A plus 2's complement of B. Which is equal to the subtraction X – Y.

Addition/Subtraction :H/W Algorithm

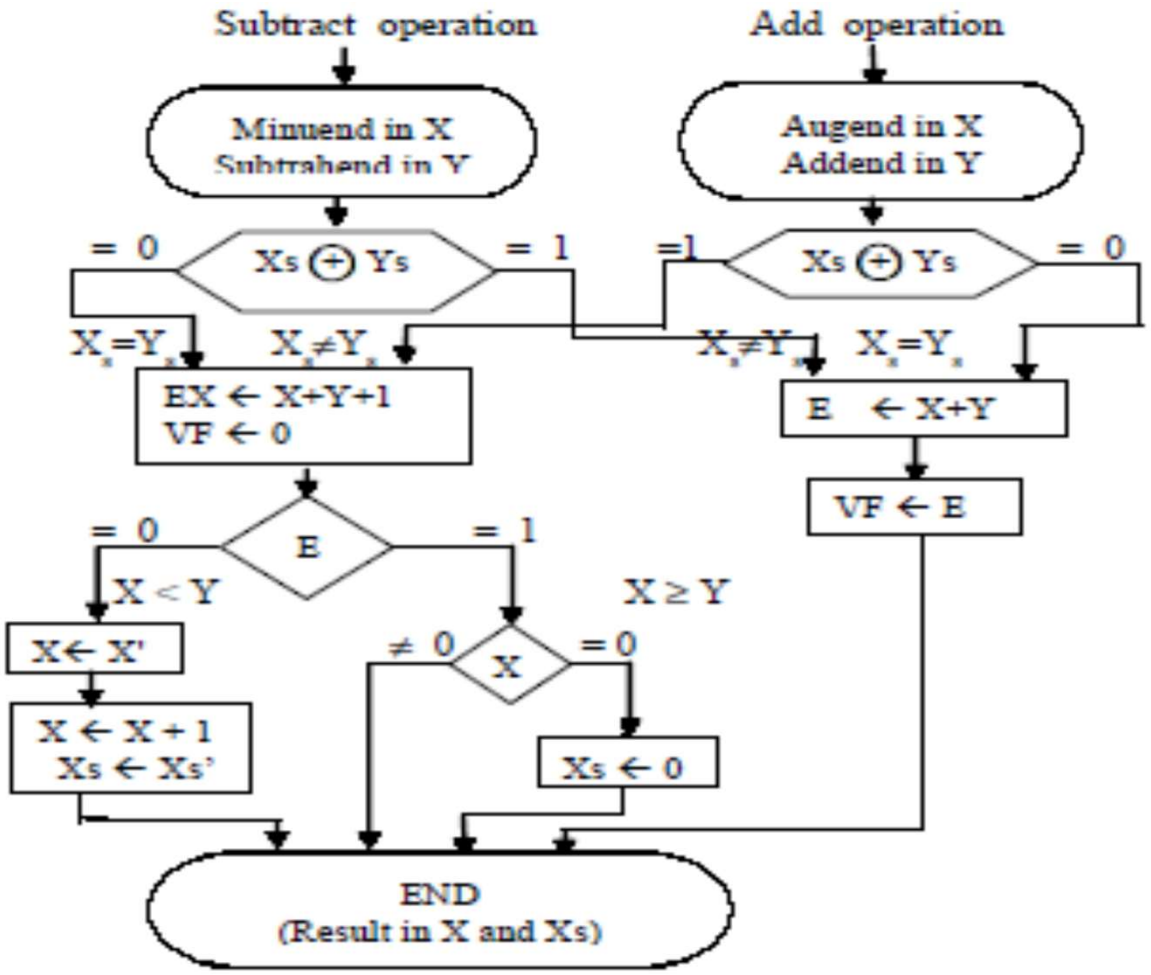


Fig: - Flow chart for add and subtract operation

Addition/Subtraction :H/W Algorithm

- For an addition operation the identical signs dictate that the magnitude be added.
- For subtraction operation different signs dictate that magnitude be added..
- Magnitude are added with a micro operation
  - EX← X + Y, Where EX ←Reg that combines that combines E and X.
  - The carry in E after the addition constitutes an overflow if it is equal to 1.
  - The value of E is transferred into the add-overflow flip-flop VF.
- Two magnitude are subtracted if the two signs are different for an addition operation or identical for subtract operation. The magnitude is subtracted by adding X to the number are subtracted so VF is cleared to 0.
- A 1 in E indicates that X ≥ Y and the number in X is the correct result. If this number is zero the sign X_s must be made positive to avoid a negative zero.

Addition/Subtraction :H/W Algorithm

- A zero in E indicates that X < Y. For this case it is necessary to take the 2's complement of the value in X. This operation can be done with the micro operation X ←X' + 1.
- When X < Y the sign of the - result is the complement of the original sign of X. It is necessary to complement Xs' to obtain the
- correct sign.
- The final result is found in register X and its sign in Xs.
- The value in VF provides an overflow indication

Addition and subtraction : signed 2's complement data

- The left most bit of a binary no. represents as sign bit :0→ positive,1→negative.
- If the sign bit is 1 the entire number is represented in 2's complement form.

Ex: +33 : is represented as 00100001 and

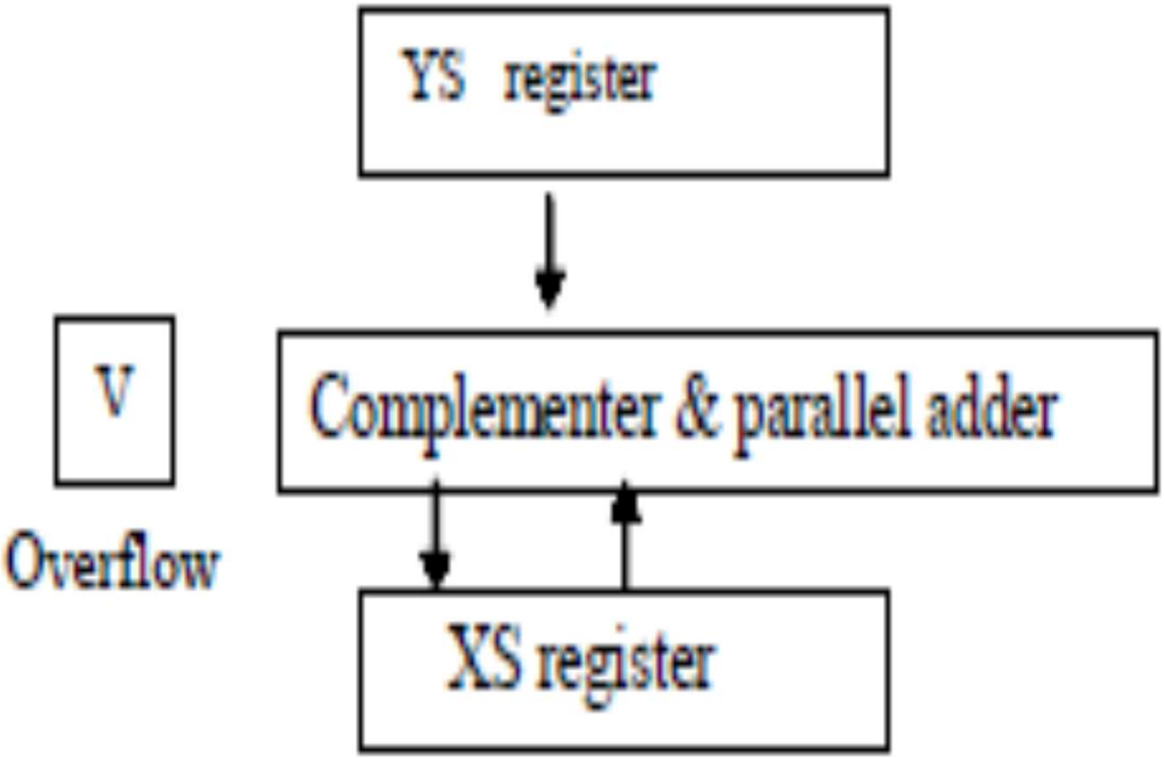
-33 as 11011111.

11011111 is the 2's complement of 00100001 and vice versa.

Addition and subtraction : signed 2’s complement data

- The addition of two numbers in signed 2’s complement form:
  - adding the number with the sign bits treated the same as the other bits of the number.
  - A carryout of the sign-bit position is discarded.
- The subtraction consists:
  - taking 2’s complement of the subtrahend and then adding it to the minuend.
- An overflow can be detected by inspecting the last two carries out of the addition.
- When the two carries are applied to the XOR gate the overflow is detected when the output of the gate is equal to 1.

Add/Sub signed 2's complement data :H/W Representation



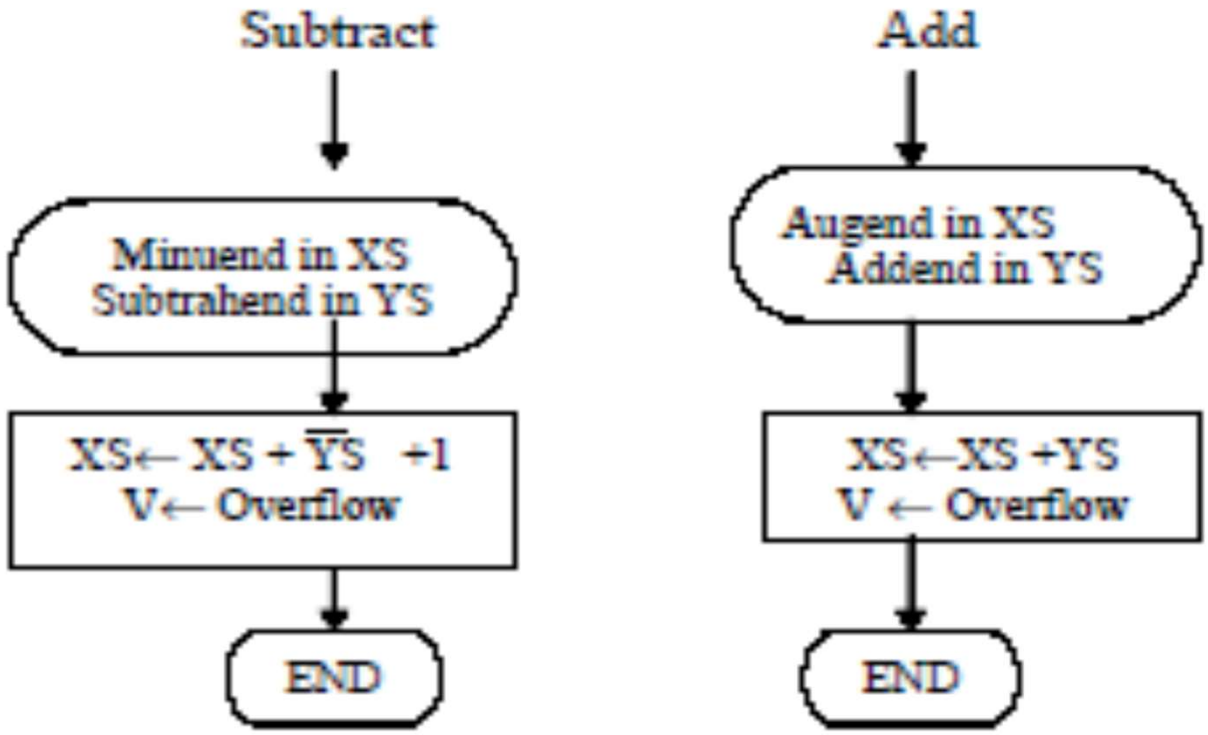
Add/Sub signed 2's complement data :H/W Representation

- The X register XS (Accumulator) and the Y register YS.
- The left most bit in XS and YS represents the sign bits of the number.
- The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder.
- The overflow flip-flop V is set to 1 if there is an overflow.
- The output carry in this case is discarded.

Add/Sub signed 2's complement data :Algorithm

- The sum is obtained by adding the contents of XS and YS (including their sign bits).
- The overflow bit V is set to 1 if XOR of the last two carries is 1 and it is cleared to 0 otherwise.
- The subtract operation is accomplished by adding the content of Xs to the 2's complement of YS. Taking the 2's complement of YS has the effect of changing a positive number to negative and vice versa.
- An overflow must be checked during these operation because the two number added could have the same sign.

Add/Sub signed 2's complement data :Algorithm



Algorithm for addition& Subtraction-2's Complement representation

Add/Sub signed 2's complement data :Algorithm

- Addition:

+74 0 1 0 0 1 0 1 0

+

+69 0 1 0 0 0 1 0 1

- 1 0 0 0 1 1 1 1

Subtraction :

+ 1 2 5 0 1 1 1 1 1 0 1

(-) (+)

+ 9 0 1 0 1 0 0 1 1 0 ← 2's complement of 90

+ 3 5 1 0 0 1 0 0 1 1

discard Carry

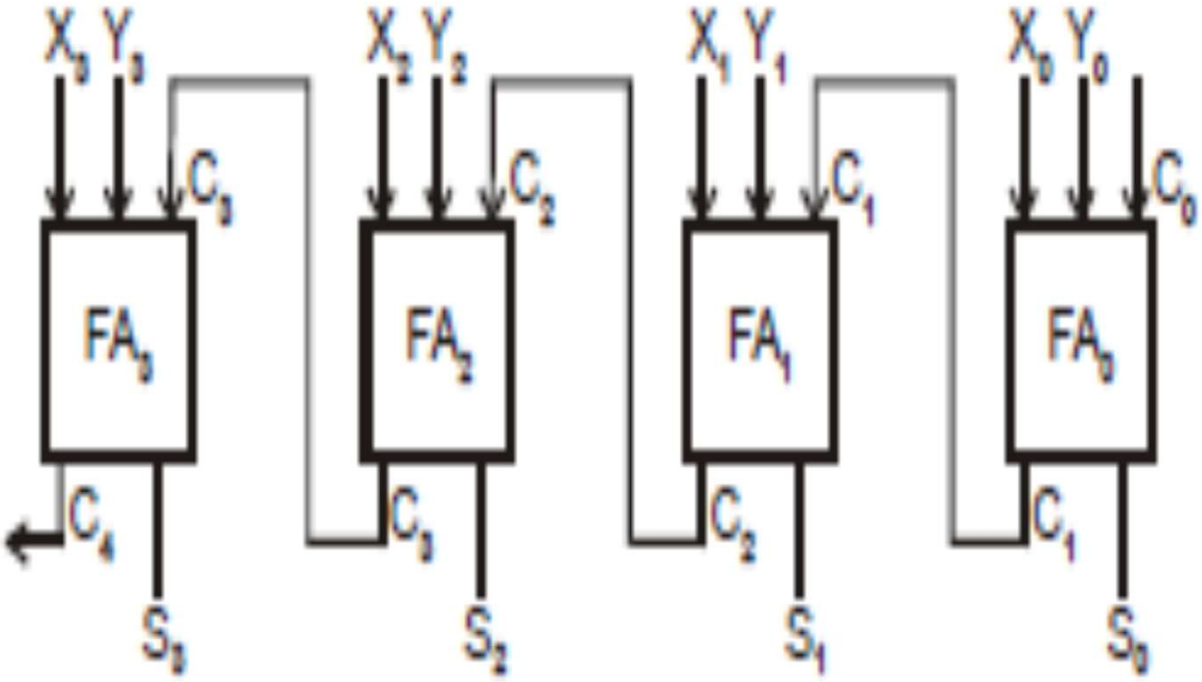
Design of Fast adders :Carry Propagation

- Addition of two binary numbers in parallel implies:
- All the bits of the augend & addend are available for computation at the same time.
- The parallel adders are ripple carry types, carry output of each full adder stage is connected to the carry input of the next higher-order stage.
- The sum of output of any stage cannot be produced until the input carry occurs.
- This lead to a time delay in the addition process.
- The carry propagation delay for each full adder is the time from the application of the input carry until the output carry occurs, assuming that the P and Q inputs are present

Design of Fast adders: Carry Propagation

- Full adder 1 cannot produce a potential carry output until a carry input is applied.
- The input carry to the least significant stage has to 'ripple' through all the adders before the final sum is produced.
- A cumulative delay through all of the adder stages is a 'worst-case' addition time.
- The total delay can vary, depending on the carries produced by each stage.
- If two numbers are added such that no carriers occurs between stages, the add time is simply the propagation time through a single full adder from the application of the data bits on the inputs to the occurrence of a sum output.

Design of Fast adders :Carry Propagation



Look Ahead Carry Adder

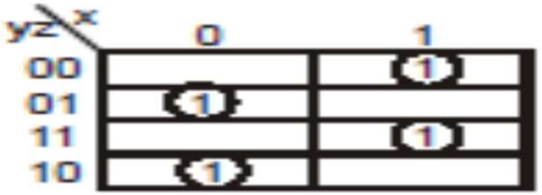
- In parallel adder, the speed with which an addition can be performed is limited by the time required for the carries to propagate or ripple through all of the stages of adder.
- One method of speeding up this process is by eliminating ripple carry delay is called Look-Ahead Carry addition.
- It is based on two functions of the full adder called the carry generate and the carry propagate function

Full Adder

Hardware Implementations

When k-map is drawn for S and C, we have

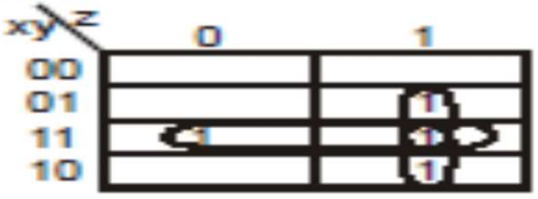
For S



$$\therefore \quad S = x\bar{y}\bar{z} + \bar{x}\bar{y}z + xyz + \bar{x}y\bar{z}$$

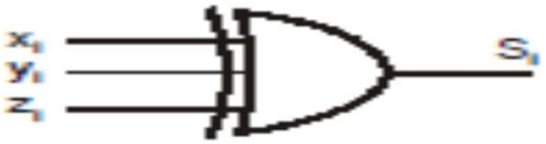
$$S = x \oplus y \oplus z$$

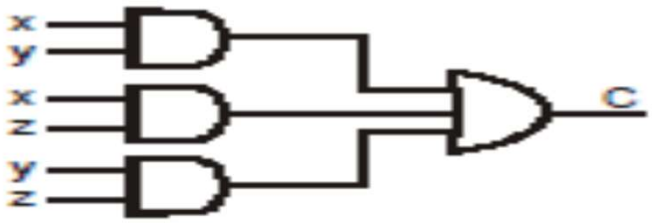
For C



$$C = xy + xz + yz$$

z is carry-in, therefore





Look Ahead Carry Adder

From the full adder circuit,

$$S_i = x_i \oplus y_i \oplus c_i$$

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

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

Generate $G_i = x_i y_i$

Look Ahead Carry Adder

Propogate $P_i = x_i + y_i$

FA_0 For first full addeer

$$C_1 = G_0 + C_0 P_0$$

FA_1 For second full addeer

$$C_2 = G_1 + C_1 P_1$$

$$= G_1 + P_1 (G_0 + C_0 P_0) \quad [\because \quad C_1 = G_0 + P_0 C_0]$$

$$\therefore \; C_2 = G_1 + G_0 P_1 + C_0 P_1 P_0$$

Similarly

$$C_3 = G_2 + C_2 P_2$$

$$= G_2 + (G_1 + G_0 P_1 + C_0 P_1 P_0) P_2$$

$$= G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_2 P_1 P_0$$

Look Ahead Carry Adder

- The carry output for each full adder stage is dependent only on the initial input carry (C0), its G0 and P0 functions and the G and P functions of the preceding stages.
- Since, each of the G and P functions can be expressed in terms of the x and y inputs to the full adders.
- All of the output carries are immediately available and the adder circuit need not have to wait for a carry to ripple through all of the stages before a final result is achieved.
- Thus the look ahead carry technique speeds up the addition process.

Look Ahead Carry Adder

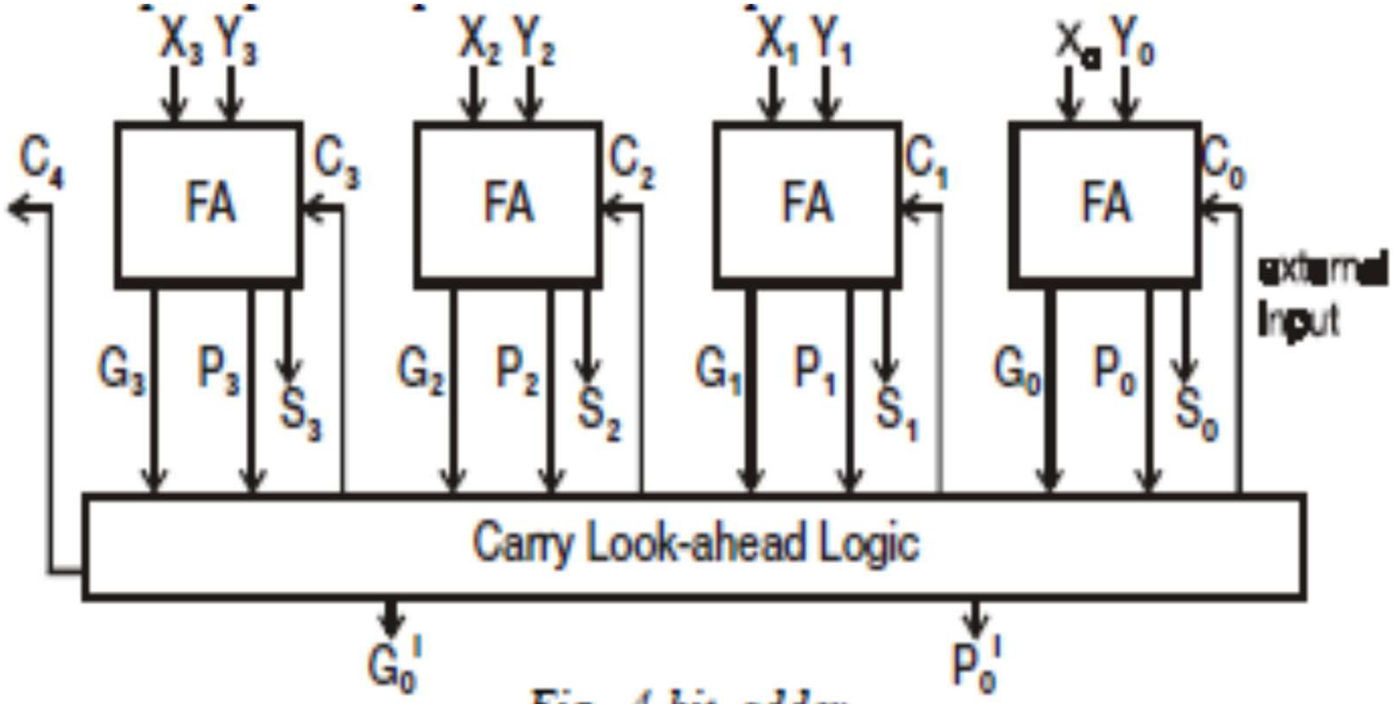


Fig. 4-bit adder.

Look Ahead Carry Adder

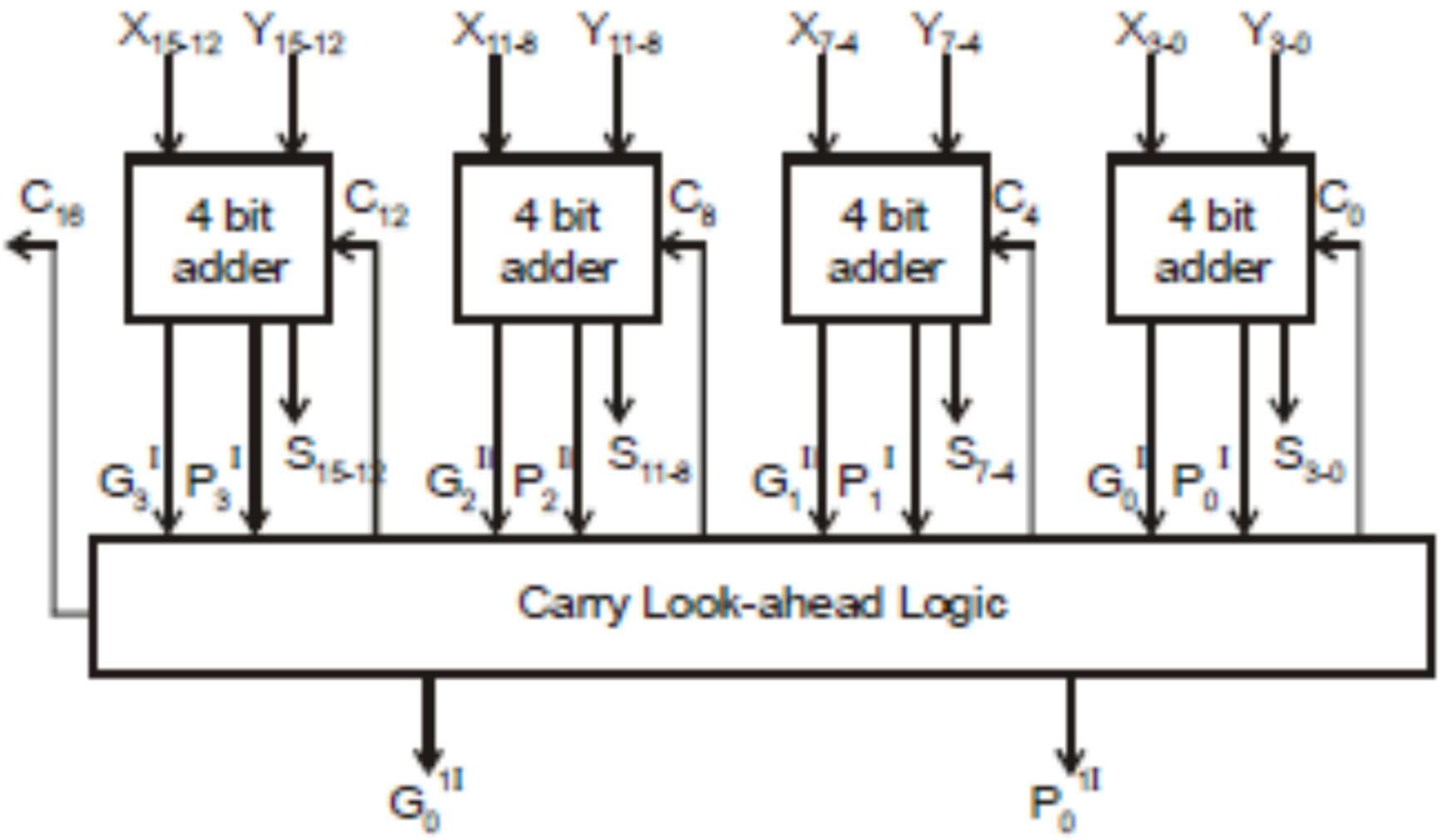
In general, the final expression for any carry variable is

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots\ldots\ldots$$
$$+ P_i P_{i-1} \ldots\ldots P_i G_0 + P_i P_{i-1} \ldots\ldots P_0 G_0$$

Delays in Carry Look ahead adder

- In carry-look ahead 4-bit adder, all carries are obtained three gate delays after the input signals X, Y and C0 are applied.
- One gate delay is needed to develop all Pi and Gi signals followed by two gate delays in the AND-OR circuit for Ci+1.
- After XOR gate delay, all sum bits are available.

Delays in Carry Look ahead adder



Delays in Carry Look ahead adder

- The carryout C4 from the low order adder is available 3 gate delays after the input operands X, Y and C0 are applied to the 16 bit carry look ahead adder.
- C8 is available after a further 2 gate delays, C12 is available after a further 2 gate delays and C16 is available after a further 2 gate delays.
- C16 is available after a total of (3×2) + 3 = 9 gate delays
- If a ripple carry adder is used, after 31 gate delays for S15 and 32 gate delays for C16.

Multiplication of positive numbers

- Multiplication of two fixed-point binary numbers in signed magnitude representations is done by a process of successive shift and add operations

- 11 1 0 1 1 Multiplicand
- 9 x 1 0 0 1 Multiplier

```
          1 0 1 1
        0 0 0 0 +
      0 0 0 0
    1 0 1 1
 99 1 1 0 0 0 1 1
```

Multiplication of positive numbers

- The process consists of looking at successive bits of the multiplier, LSB first.
- If the multiplier bit is a 1 → multiplicand is copied down

  0 → zeroes are copied down.
- The number copied down in successive lines are shifted one position to the left from the previous number finally the numbers are added and their sum forms a product.
- Sign of the product is determine from the signs of the multiplicand and multiplier.
  - If they are alike the sign of the product is positive.
  - It they are unlike the sign of the product is negative

Multiplication of positive numbers

- An adder for the summation of only two binary numbers and successively accumulate the partial products in a register.
- Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required (Position) relations.
- When the corresponding bit of the multiplier is 0 there is no need to add all zeros to the partial product since it will not alter its value.

Multiplication of positive numbers

- The multiplier is stored in the Q register and its sign in Qs.
- The sequence counter SC is initially set to a number equal to the number of bits in the multiplier.
- The counter is decremented by 1 after forming each partial product.
- When the content of the counter reaches zero, the product is formed and the process stops.
- multiplicand is in register Y and the multiplier in Q.
- sum of X and Y forms a partial product, & is transferred to the EX reg.
- Both partial product and multiplier are shifted to the right.
- It will be denoted by the statement Shr EXQ to designate the right shift depicted.

Multiplication of positive numbers

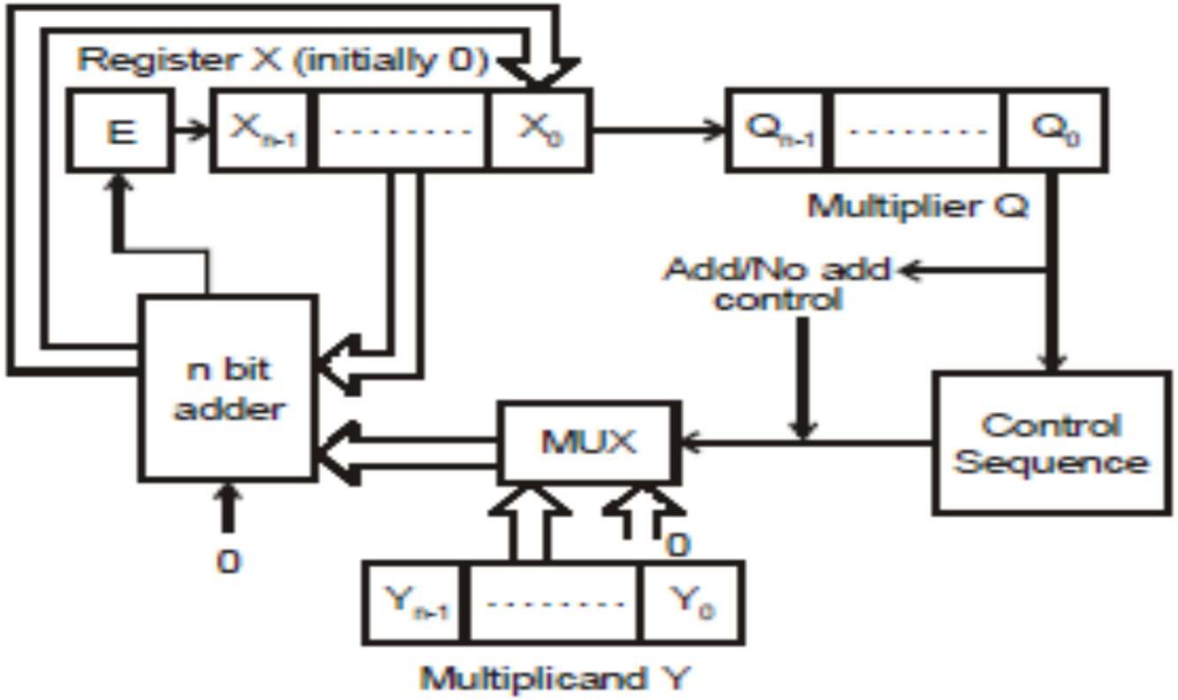


Fig:- H/w for multiply Operation

Multiplication of positive numbers

- The LSB of X is shifted in to the MSB position of Q., the bit from E is shifted in to MSB position of X.
- After the shift one bit of the partial product is shifted into Q, pasting multiplier bits one position to through right.
- The right most FF in register Q, designated by Qn; will hold the bit of the multiplier, which must be inspected next

Multiplication: H/W algorithm

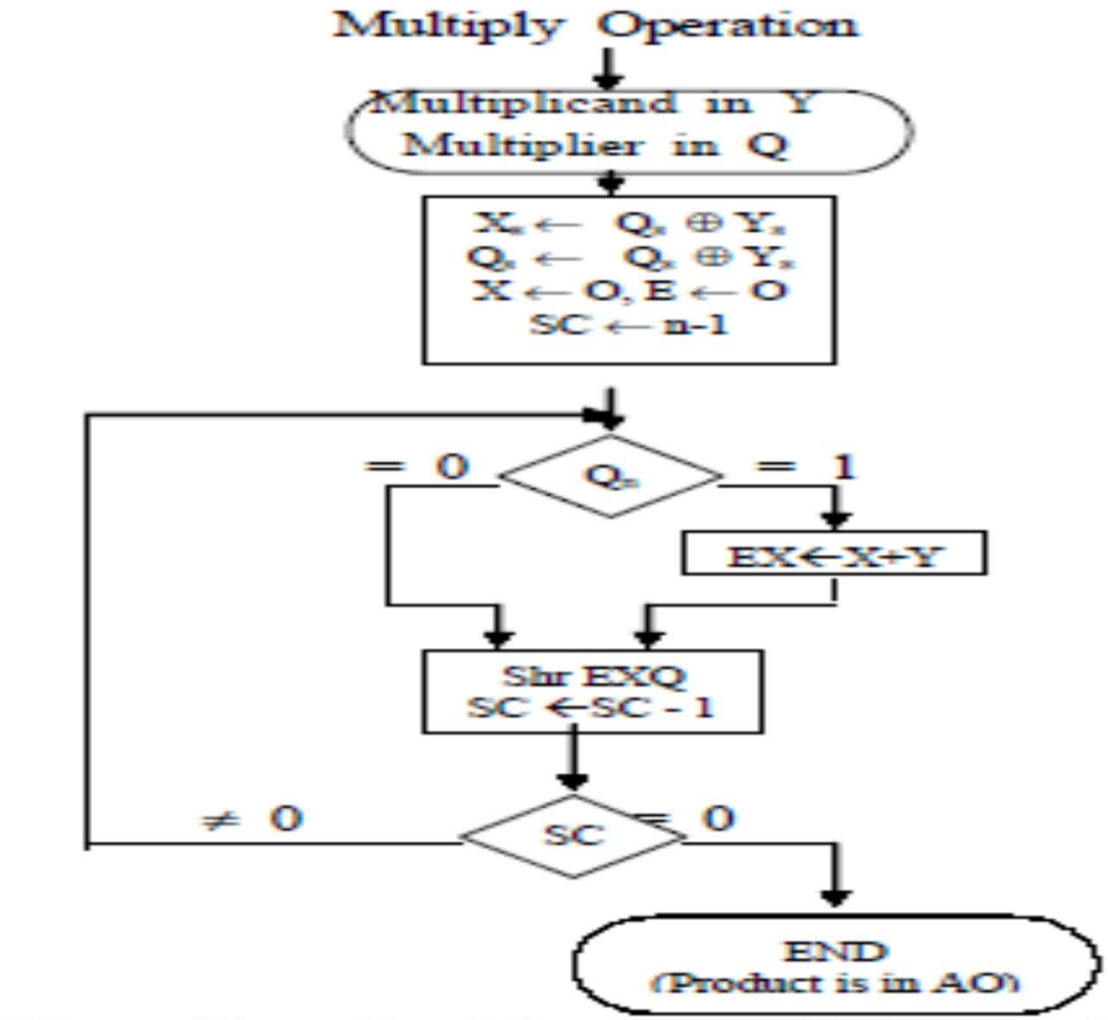


Fig: - Flow chart for the multiply operation.

Multiplication: H/W algorithm

- Initially the multiplicand is in Y, the multiplier in Q. their signs are in Ys & Qs.
- The signs are compared
- Both X and Q are set to correspond to the sign of the product since a double - length product will be stored in reg X and Q register X and E are cleared.
- sequence counter SC is set to a no. equal to the no. of bits of the multiplier.
- Operands are transferred to reg from a memory unit that has words of n bits.
- Operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consists of (n-1) bits.

Multiplication: H/W algorithm

- The low order bit of the multiplier in Qn is tested if it is a 1, the multiplicand in Y is added to the present partial product in X.
- If it is 0 nothing is done.
- Register EXQ is then shifted once to the right to form the new partial product.
- The sequence counter SC is decremented by I
- and its new value is checked if it is not equal to zero the process is repeated and the new partial is formed.
- The process stops when SC=0.
- partial product in X is shifted in to Q one bit at a time and eventually replaces the multiplier the final product is available in both X and Q.
- With A holding the MSBs and Q holding LSBs

Multiplication: H/W algorithm

Eg.

$23 \times 19 = 437$

Multiplicand $Y = 10111 \leftarrow 23$

Multiplier $Q = 10011 \leftarrow 19$

	E	X	Q	SCounter = no. of bits in the multiplier in the binary form
	0	00000	10011	101
Qn=1, ADD Y		10111		
First partial Product	0	10111		
Shift right EXQ	0	01011	11001	100
Q_n= 1 , Add y		10111		
Second partial Product	1	00010		
Shift right EXQ	0	10001	01100	011
Q_n= 0 , No add Shr EXQ	0	01000	10110	010
Q_n= 0 , No add Shr EXQ	0	00100	01011	001
Q_n= 1 ADD Y		10111		
		11011		
Shr EXQ	0	01101	10101	000

Since SC becomes zero

Final product is in XQ

$XQ = 0\,1\,1\,0\,1\,1\,0\,1\,0\,1 \leftarrow 437$

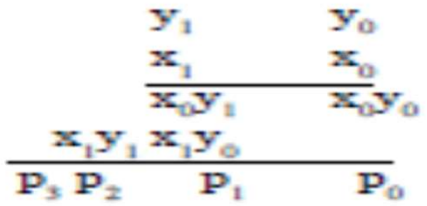
Array multiplier

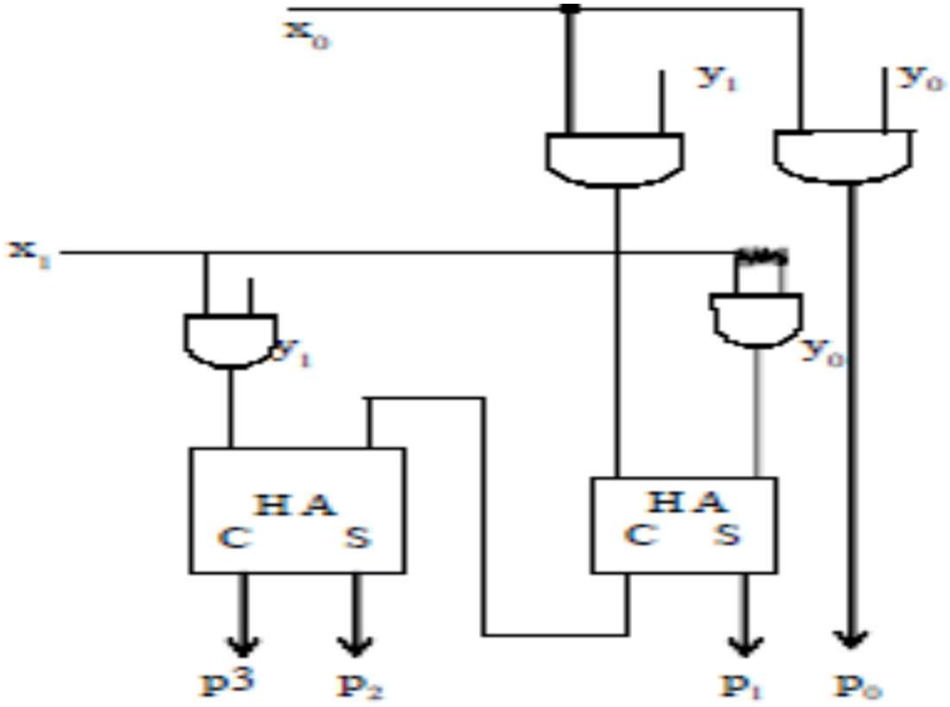
- The multiplicand bits are y1 and y0 and the multiplier bits are x1 and x0 and the product is p3 p2 p1 p 0.
- The first partial product is formed by multiplying x0 by y1y0.
- The multiplication of two bits such as x0 and y0 produces a 1 if both bits are 1; otherwise it produces a product 0.
- This is identical to AND operation and can be implemented with AND gate.
- the first partial product is formed by means of two AND gates.

Array multiplier

- The second partial product is formed by multiply by x1 by y1y0 and is shifted one position to the left.
- The two partial products are added with two half adder (HA) circuits.
- LSB of the product does not have to get through an adder since it is formed by o/p of first AND gate

Array multiplier





For j multiplier bits and k multiplicand bites need j x k AND gates and (j-1) k-bit address to produce a product of j + k bits.

Array multiplier

- For j multiplier bits and k multiplicand bits need j x k AND gates and (j-1) k-bit address to produce a product of j + k bits.

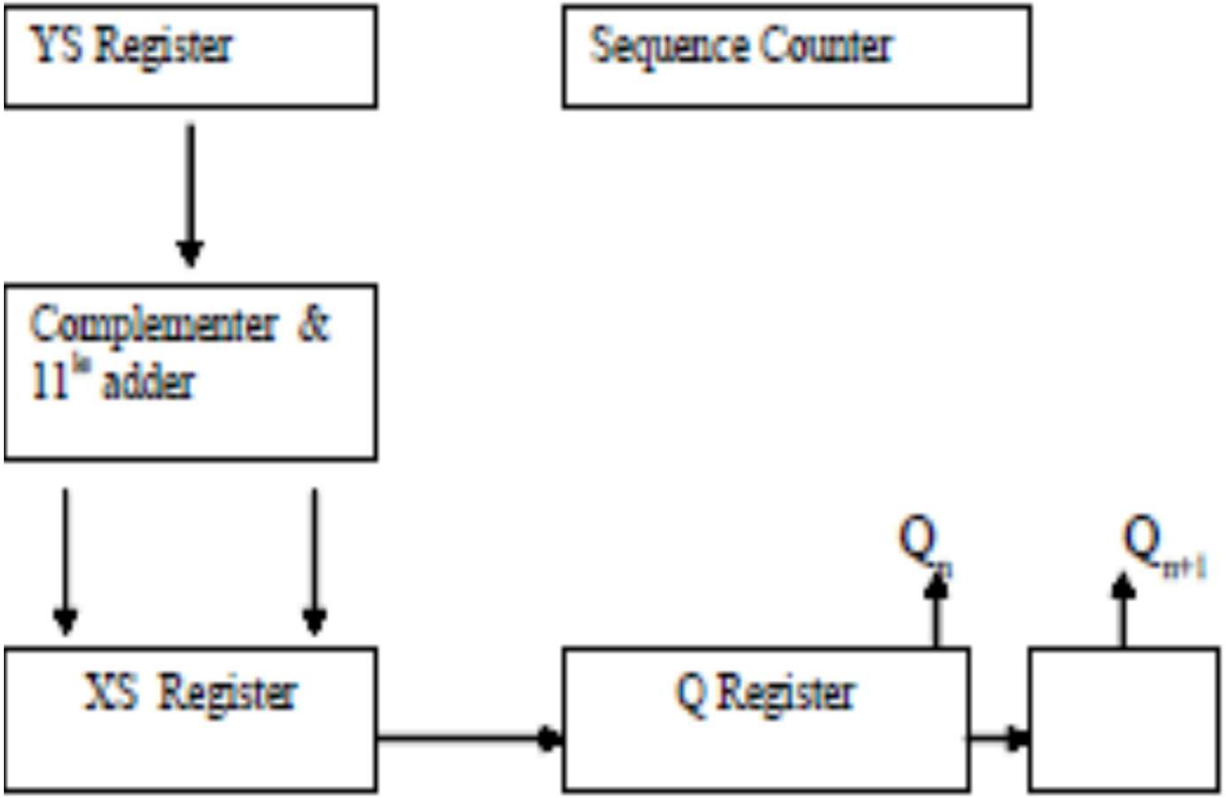
Booth's Multiplication Algorithm

- Booth's Multiplication algorithm gives a procedure for multiplying binary integers in signed 2' s complement representation.
- The multiplicand is subtracted from the partial product upon encountering the first LSB 1 in a string of 1's in the multiplier
- The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1 in a string of 0's in the multiplier).
- The partial product does not change when the multiplier bit is identical to the previous multiplier bit
- The algorithm works for positive (or) negative multiplier in 2's Comp representation.
- A negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight

Booth's Multiplication

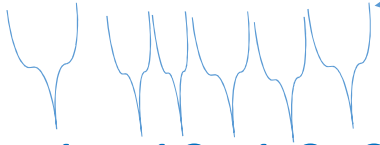
- H/W Requirement is similar to the H/W for normal multiplication process.
- The sign bits are not separated from the X, Y and Q as XS, YS and QR respectively.
- Qn designates the LSB of multiplier in reg. QR.
- An extra FF Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier

Booth's Multiplication



Booth's Multiplication

- -1 times shifted multiplicand is chosen when the combination is 1 0
- +1 times shifted multiplicand is chosen when the combination is 0 1
- 0 times shifted multiplicand is chosen when combination is 0 0 or 1 1
- Ex: Recode the multiplier 1 0 1 1 0 0 for booths multiplication
- Multiplier : 1 0 1 1 0 0 0 ← Implied zero



- Recoded multiplier : -1 +1 0 -1 0 0

Booth's Multiplication

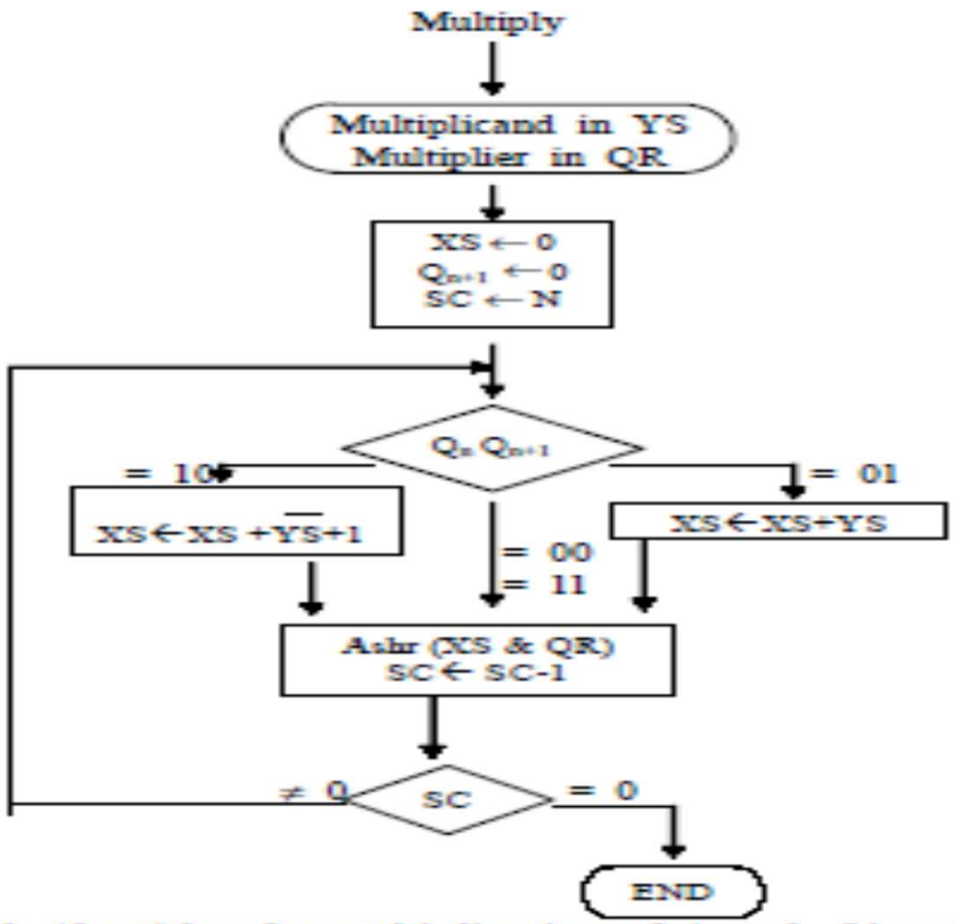


Fig:- Booth Algorithm for multiplication of signed - 2's complementer

Booth's Multiplication

- XS and appended bit Q_{n+1} are initially cleared to 0
- sequence counter SC is set to a number n equal to the number of bits in the multiplier.
- The two bits of the multiplier in Q_n and Q_{n+1} are inspected.
- If Q_n Q_{n+1} equal to 10, it means that the first 1 in a string of 1's has been encountered ➔subtraction of the multiplicand from the partial product in XS.
- If Q_n Q_{n+1} equal to 00 or 11, the partial product dose not change.
- If Q_n Q_{n+1} equal to 01 add the Multiplicand to partial product
- An overflow can't occur because the addition and subtraction of the multiplicand follow each other.
- The next step is to shift right the partial product and the multiplier (including Bit Qn+1).
- This is an arithmetic shift (ashr) operation which shift XS and QR to the right and leaves the sign bit in XS unchanged.
- The sequence counter (SC) is decremented and the computational loop in repeated n times.

Booth's Multiplication

- Step-by-step multiplication of (-9) x (-13)=117.
- multiplier in QR is negative & multiplicand in YS is also negative.
- The 10-bit product appends in XS and QR and is positive.
- The final value of Qn+1 should not be taken as part of the product

Booth's Multiplication

- Example : –ve × –ve

  –9 × –13
- Multiplicand :YS = 1 0 1 1 1 = –9
- $\overline{YS}$ +1 = 0 1 0 0 1
- Multiplier : QR = 1 0011 = – 13

Booth's Multiplication

Qn	Q_{n+1}	Initial	XS	QR	Q_{n+1}	SC
		→	0 0 0 0 0	1 0 0 1 1	0	1 0 1
1	0	Subtract YS (by 2's sompl addt)	0 1 0 0 1 0 1 0 0 1			
		Ashr	0 0 1 0 0	1 1 0 0 1	1	1 0 0
1	1	Ashr	0 0 0 1 0	0 1 1 0 0	1	0 1 1
0	1	Add YS	1 0 1 1 1 1 1 0 0 1			
		Ashr	1 1 1 0 0	1 0 1 1 0	0	0 1 0
0	0	Ashr	1 1 1 1 0	0 1 0 1 1	0	0 0 1
1	0	Sub YS	0 1 0 0 1 0 0 1 1 1			
		Ashr	0 0 0 1 1	1 0 1 0 1	1	0 0 0

Final Product

Booth's Algorithm Advantages

- It handles both positive and negative multipliers uniformly.
- It achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1's.
- The speed gained by skipping 1's depends on the data.

Booth's Algorithm : +ve numbers

Case (ii) +ve × +ve

$$\underbrace{00110}_{6} \times \underbrace{01001}_{9} \times \underbrace{110110}_{54}$$

BR = 0 0 1 1 0 = –9

$\overline{BS}+1$ = 1 1 0 0 1 +1 → 1 1 0 1 0

Qn	Q_{n+1}	Initial	AC	QR	Q_{n+1}	SC
			0 0 0 0 0	0 1 0 0 1	0	1 0 1
1	0	Sub BR	1 1 0 1 0			
			1 1 0 1 0			
		Ashr	1 1 1 0 1	0 0 1 0 0	1	1 0 0
0	1	Ashr BR	0 0 1 1 0			
			0 0 0 1 1			
		Ashr	0 0 0 0 1	1 0 0 1 0	0	0 1 1
0	0	Ashr	0 0 0 0 0	1 1 0 0 1	0	0 1 0
1	0	Sub BR	1 1 0 1 0			
			1 1 0 1 0			
		Ashr	1 1 1 0 1	0 1 1 0 0	1	0 0 1
0	1	add BR	0 0 1 1 0			
			0 0 0 1 1			
		ashr	0 0 0 0 1	1 0 1 1 0 0	0	0 0 0

Final Product

Bit Pair Recoding of Multipliers

- If the Booth recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial part for each pair of multiplier bits.

Table of Multiplicand Selection decisions.

Multiplier bit Pair		Multiplier bit on the right	Mulitplicand Selected at position i
i+1	i	i–1	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

Bit Pair Recoding of Multipliers

- Find the bit pair code for the multiplier 11010
- By referring the table:

Sign extension →1 1 1 0 1 0 0←implied 0 to right of LSB

0 -1 -2

Bit Pair Recoding of Multipliers

- Multiply given signed 2's complement no using bit pair recoding
- A= 110 101 multiplicand (-11)
- B= 011 011 multiplier (+27)
- Bit pair code for multiplier 0 1 1 0 1 1 0

+2 -1 -1

Multiplication: 1 1 0 1 0 1

+2 -1 -1

0 0 0 0 0 0 0 0 1 0 1 1 ←2's comp of multiplicand

0 0 0 0 0 0 1 0 1 1 ←2's comp of multiplicand

1 1 1 0 1 0 1 0 ← multiplicand X +2

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

Carry Save addition of summands

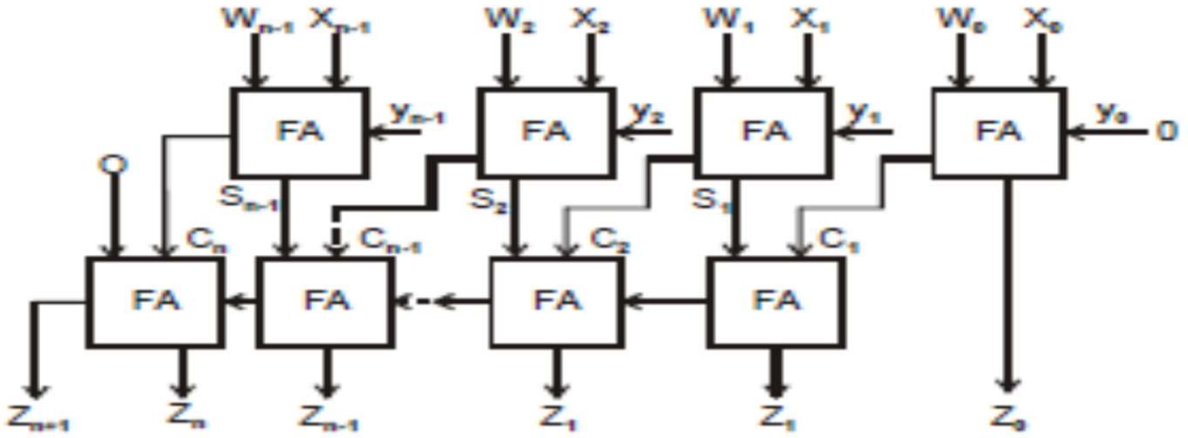
- Multiplication of 2 numbers involves the addition of several summands (partial products). A technique called carry save addition speeds up the addition process.
- In ordinary multiplication process, while adding the summands, the carry produced by second bit position has to be added with third bit position and so on.
- Thus the carry ripple along the rows.

Eg. Consider the addition of 3 numbers W, X, Y

i.e. Z = W + X + Y

- **Case (i)** Using ripple carry adders, Z = W + X + Y can be implemented as

Carry Save addition of summands



Eg :

W	=	1 0 1 0 1
X	=	1 1 0 1 1
		1 1 0 0 0 0
Y	=	0 1 0 1 0 0
Z	=	1 0 0 0 1 0 0

Division

- Instead of shifting the divisor to the right the dividend or partial remainder, is shifted to the left can be achieved by adding X to the 2's complement of Y.
- The H/W for implementing the division operation is identical to that required for multiplication.
- Register EXQ is now shifted to the left with 0 inserted into Qn and previous value of E is lost

Division

- Divisor is stored in the Y register(M register) & the dividend is stored in register X and Q.
- The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value.
- The information about the relative magnitude is available in E.
- If E=1 it signifies that X≥Y. A quotient bit 1 is inserted into the Qn and the partial remainder is shifted to the left to repeat the process.
- If E=0 it signifies that X<Y so that quotient in Qn is 0 (inserted during the shift).
- The value of Y is then added to restore partial remainder in X to its previous value.
- The partial remainder is shifted to the left and the process is repeated again until all five-quotient bits are formed.
- The quotient is in Q and the final remainder is in X.

Division

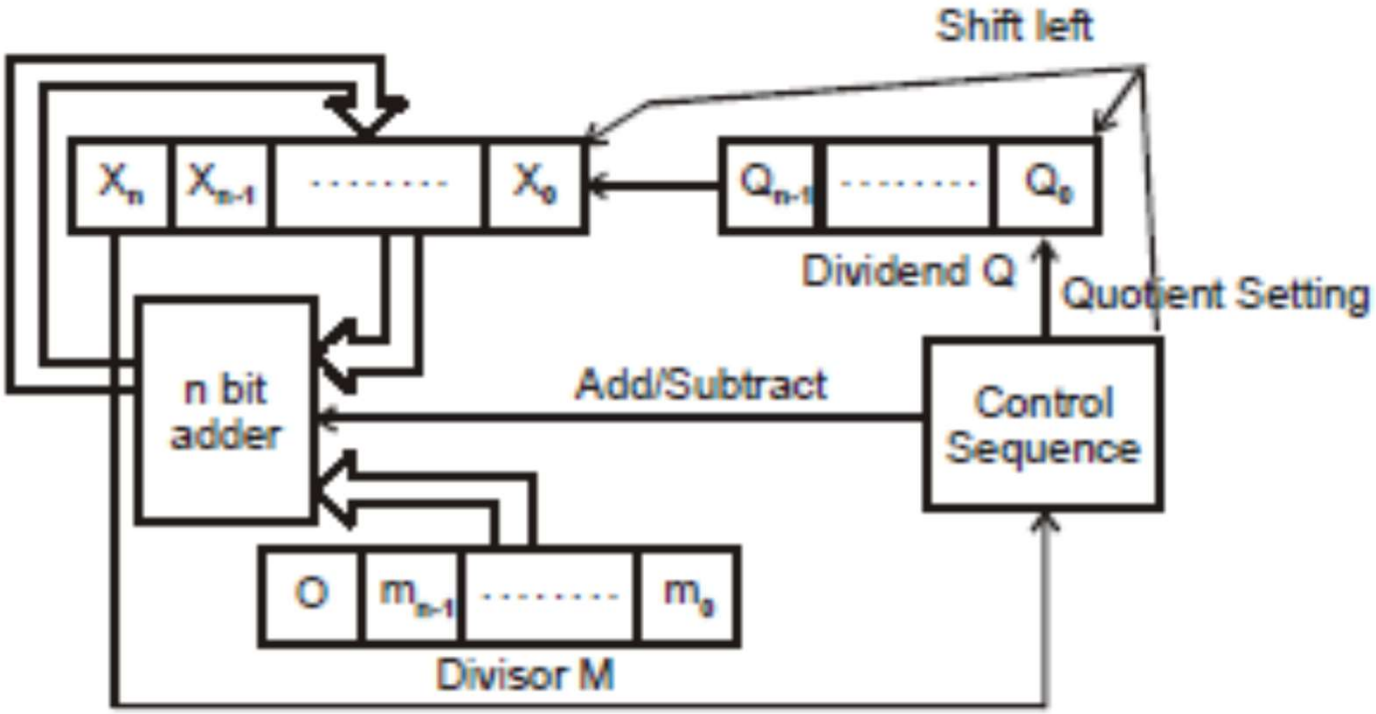


Fig. Register Configuration

Division Algorithm

- The dividend is in X and Q and the divisor in Y (M).
- The sign of the result is transferred to Qs to be a part of quotient.
- An operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consists of (n-1) bits.
- A divide overflow condition is tested by subtracting divisor in Y from half of the bits of the dividend stored in X.
- If X>Y, the divide-overflow FF DVF is set and the operation is terminated
- If X<Y, no divide overflow occurs so the value of the dividend is restored by adding Y to X.

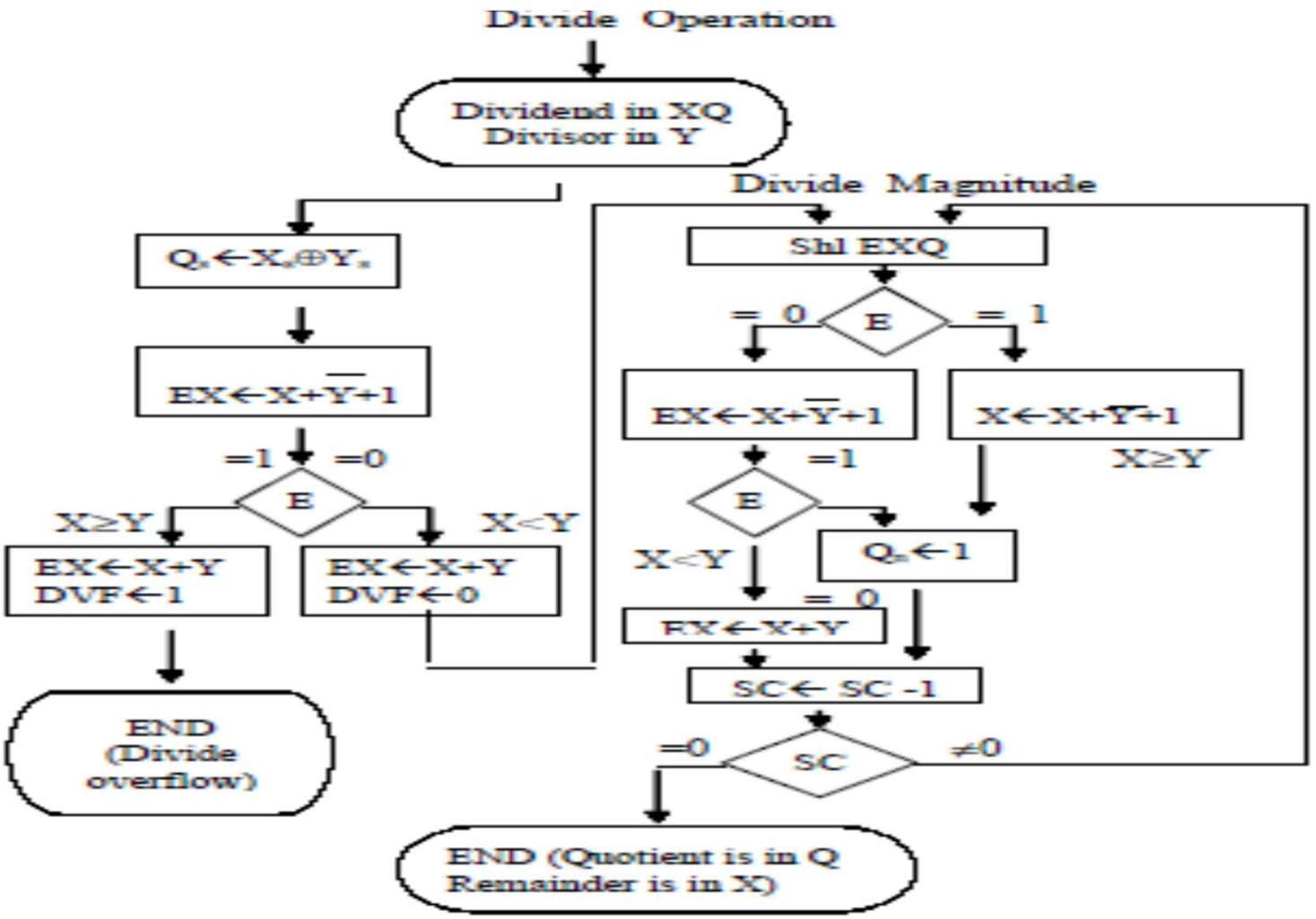
Division Algorithm

- The division of magnitude starts by shifting the dividend in XQ to the left with the high order bit shifted in to E.
- If the bit is shifted in to E is 1, we know that EX>Q because EX consists of 1 followed by (n-1) bits while Y consists of only (n-1) bits.
- In this case Y must be subtracted from EX and 1 inserted in to Qn for the quotient bit.

Division Algorithm

- If the shift left option inserts a 0 in to E the divisor is subtracted by adding its 2's complement value and the carry is transferred in to E.
- If E=1, it signifies that X>Y; therefore Qn is to 1.
- If E=0, it signifies that X<Y and the original no is restored by adding Y to X.
- In the later case we leave a 0 in Qn.(0 was inserted during the shift)
- This process is repeated again with register. X holding the partial remainder. After (n-1) times, the quotient magnitude is formed in register X.
- The quotient sign is in Qs and sign of the remainder in Xs is same as the original sign of the dividend.

Division Algorithm



Division Algorithm

- Register M holds the divisor
- Register Q holds the dividend at the start of operation.
- Register X is initially set to 0.
- After division Registrer Q contrains quotient Register X contains Remainder

Division Algorithm

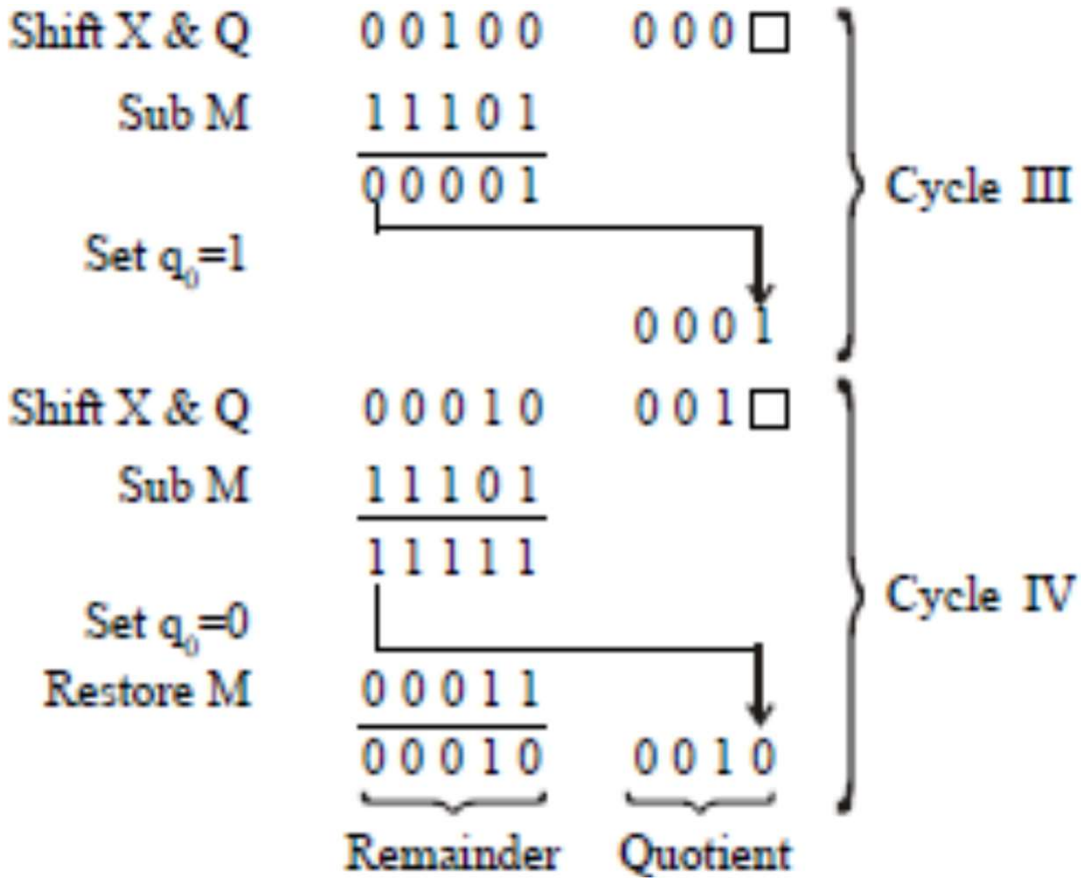
$$11\overline{)1000}\ (10 \qquad \Rightarrow \frac{8}{3} = 2.2$$

Quotient = 2
Reminder = 2

Eg.:

	X	Q	
Initial Condt	0 0 0 0 0	1 0 0 0	Cycle I
M	0 0 0 1 1		
Shift X & Q	0 0 0 0 1	0 0 0 □	
Sub M	1 1 1 0 1		
Set q_0=0	1 1 1 1 0		
Restore M	0 0 0 1 1		
	0 0 0 0 1	0 0 0 0	
Shift X & Q	0 0 0 1 0	0 0 0 □	Cycle II
Sub M	1 1 1 0 1		
Set q_0=0	1 1 1 1 1		
Restore M	0 0 0 1 1		
	0 0 0 1 0	0 0 0 0	

Division Algorithm



Non Restoring Division

- A method to improve restoring division by avoiding the need for restoring A after an unsuccessful subtraction.
- Subtraction is said to be unsuccessful if the result is negative.

Non Restoring Division

- **Non Restoring algorithm**

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(divisor) from A, otherwise, shift A and Q left and Add M(Divisor) to A.

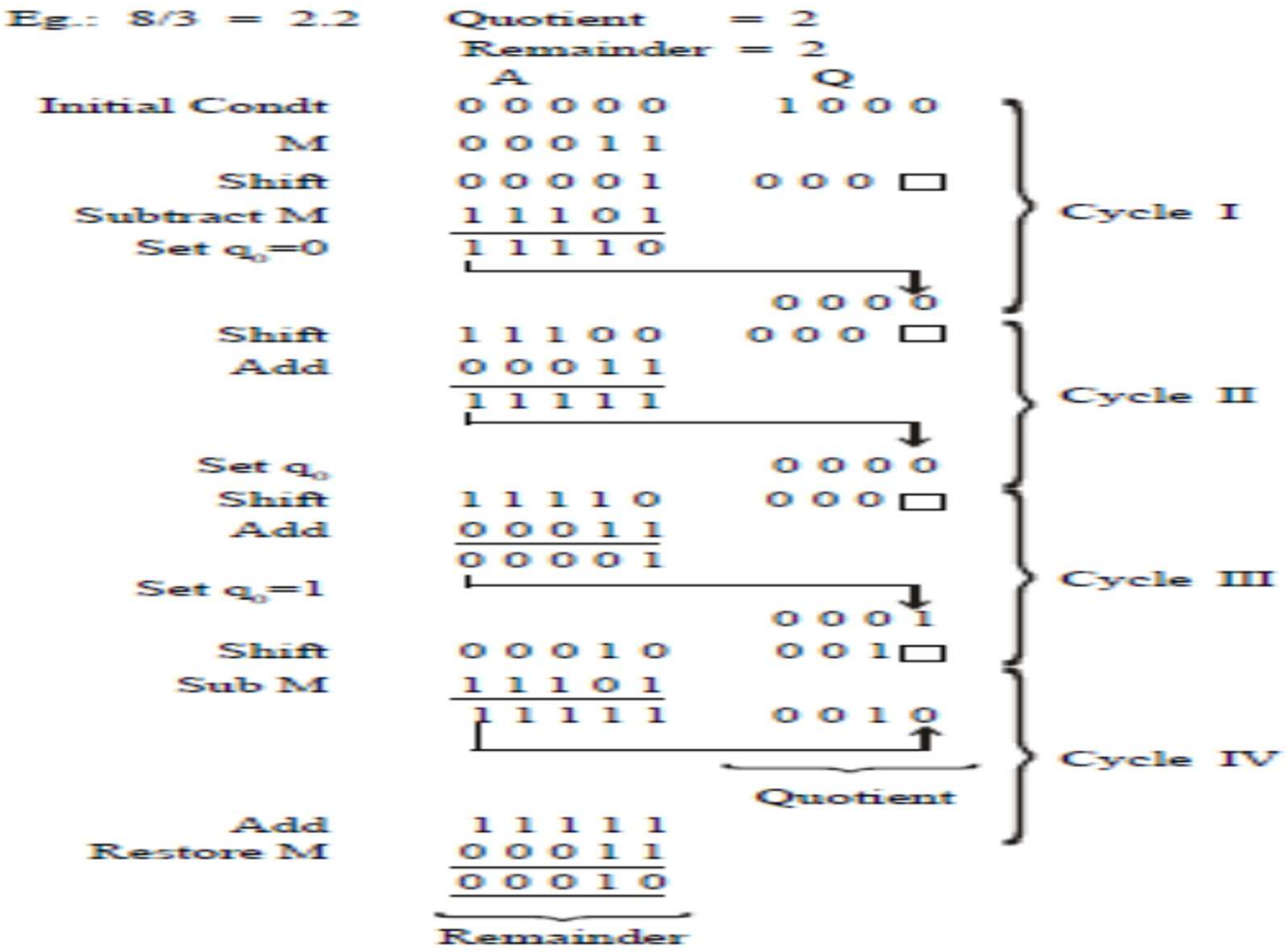
2. If the sign of A is 0, set q0 to 1; otherwise, so q0 to 0.

Step 2 : If the sign of A is 1, add M to A

Note : Restore operations are no longer needed and that exactly

one add or subtract operation is performed per cycle.

Non Restoring Division



Floating Point Numbers : Representation

- There are two types of representations for floating point numbers.

1. Single Precision
2. Double Precision

Floating Point Numbers : Representation

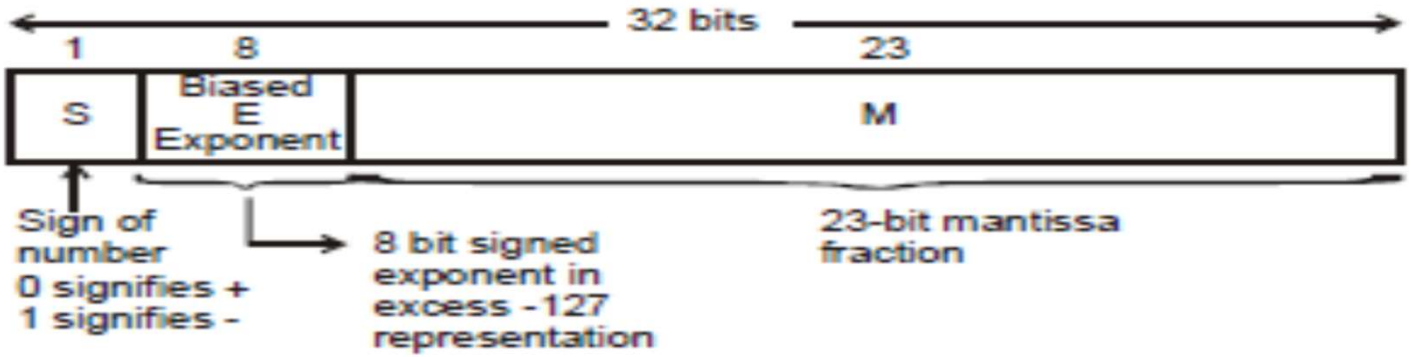


Fig. Single precision (32 bit).

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

Eg.:

0	00101000	001010 0

Value represented $= +1.001010 \ldots\ldots 0 \times 2^{-87}$

Floating Point Numbers : Representation

- In this representation, one bit is needed for the sign of the number.
- Since the leading non-zero bit of a normalized binary mantissa must be a 1, it does not have to be included explicitly in the representation.
- Instead of signed exponent E, the value actually stored in the exponent field is an unsigned integer E’ = E+127.
- This is called the excess 127 format.
- Therefore the range of E’ for normal values is $1 < E' < 254$.
- This means that the actual exponent E is in the range $-126 < E < 127$.

Double Precision Floating Point Number

- Double precision representation contains 11 bit excess 1023 exponent E’ which has the range $1 < E' < 2046$ for normal values.
- This means that the actual exponent E is in the range $-1022 < E < 1023$.
- The 52 bit mantissa provides a precision equivalent to about 16 decimal digits.
- If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent.

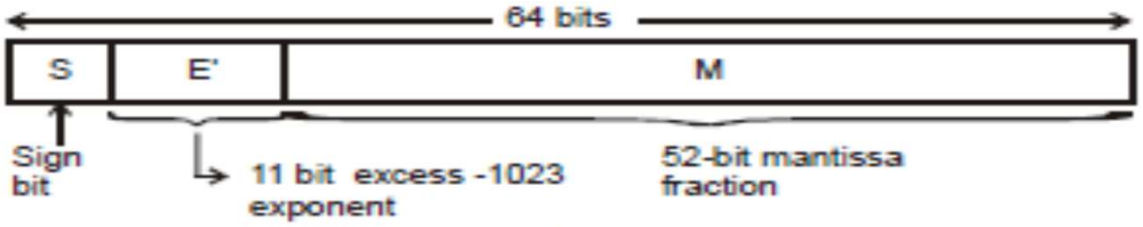


Fig. Double Precision.

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

Floating point representation :Example

- Represent 1259.12510
- Integer part:

  1259/ 16 = Quotient →78 Reminder →11 (0B)

  78 /16 = Quotient →4 Reminder → 14 (0E)

→ 4 E B = 100 1110 1011

- Fraction part: 0.125 x2 =0.25 0

  0.25 x 2 =0.5 0

  0.5 x 2 =1.0 1

Binary no → 100 1110 1011.001

Normalized form → 1.0011101011001 x 2 ^10

Floating point representation :Example

- Single precision
- For this no S=0 , E=10 and M= 001 110 101 1001
- Bias for the single precision floating point format =127
- E’ =E+127 =10+127=137 ➔ 1 0 0 0 1 0 0 1
- The no in single precision format is :
- 0 1000 1001 001 110 101 1001 … 0

  sign Exponent Mantissa

Floating point representation :Example

- Double precision
- For this no S=0 E=10 and M= 001 110 101 1001
- Bias for the Double precision floating point format =1023
- E’ =E+1023=10+1023=1033 ➔ 1 0 0 0 0 0 0 1 0 0 1
- the no in double precision format is :
- 0 1000 0001001 001 110 101 1001 … 0

  sign Exponent Mantissa

IEEE 754 Format Parameters

Parameter	Format	
	Single	Double
Word width (bits)	32	64
Exponent width (bits)	8	11
Exponent bias	127	1023
Max exponent	127	1023
Min exponent	–126	–1022
Number range (base 10)	10^{-38}, 10^{+38}	10^{-308}, 10^{+308}
Significand width (bits)	23 (Not including implied bits)	52 (Not including implied bits)
Number of exponents	254	2046
No. of fractions	2^{23}	2^{52}
No. of values	1.98×2^{31}	1.99×2^{63}

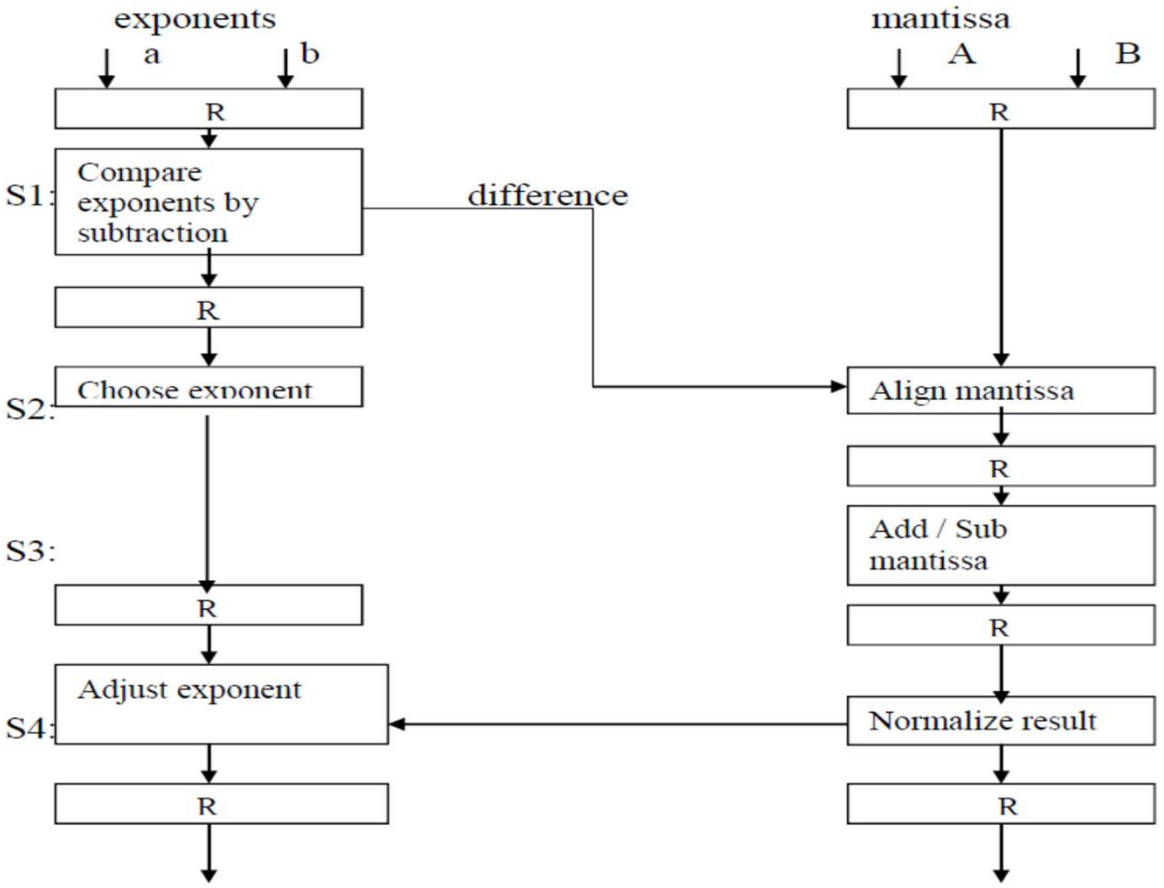
Floating Addition/Subtraction

- Compare the exponents
- Align the mantissa.
- Add / Sub the mantissa.
- Normalize the result.

Floating Addition/Subtraction

- Compared the exponents by subtracting them.
- Larger exponent is chosen as the exponent of the result.
- Exponent difference determine how many times the mantissa associated with the smaller exponent must be shifted to the right.
- This produces an alignment of the two mantissas .
- The two mantissas are added or subtracted.
- The result is normalized.
- If an overflow occurs the mantissa of the sum or the difference is shifted right and the exponent is incremented by one.
- If an underflow occurs the number of leading zeroes in the mantissa determine the number of left shifts in the mantissa and the number that must be subtracted from the exponent

Floating Addition/Subtraction



Where S1,S2,S3,S4 ➔ represent the segments

Fig: floating point add/ sub

Floating Addition/Subtraction

1. The first step is to compare exponents to determine the number of times the mantissa of the smaller exponent to be shifted.

2. The shift value n is then given to the shifter unit to shift the mantissa of the smaller number.

3. The sign of the exponent after subtraction determines which is smaller or which is larger no and thereby to shift the mantissa of the smaller number.

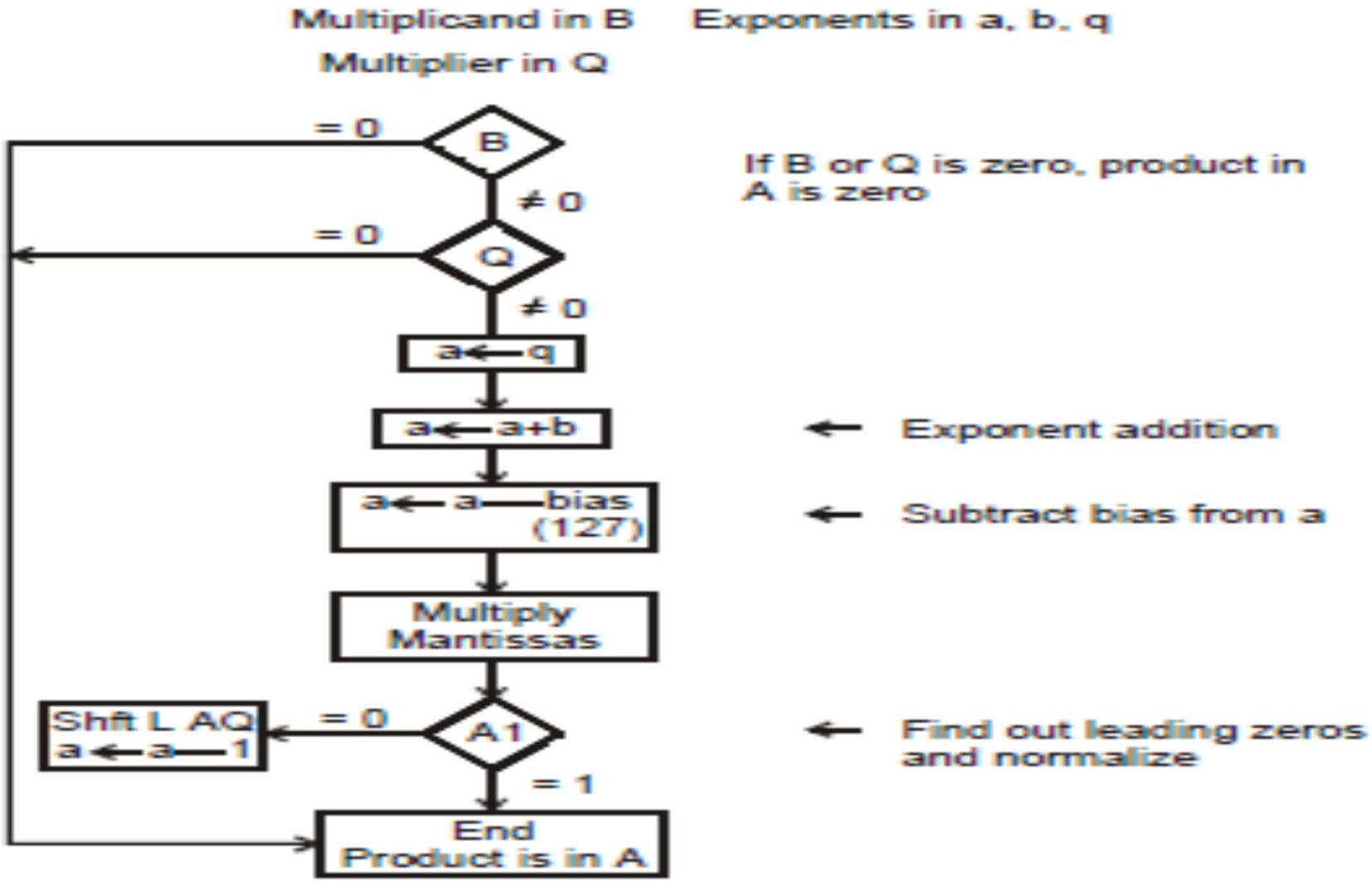
4. The mantissas are added / subtracted.

5. The result is normalized by truncating the leading zeros and by subtracting E’ by X, the number of leading zeros.

Floating point Multiplication

- Check for Zeros.
- Add the exponents.
- Multiply the mantissas.
- Normalize the product.
- steps (2) and (3) can be done (separately) simultaneously if separate adders are available for the mantissas and exponents.

Floating point Multiplication



Floating point Multiplication

- If either operand is equal to aero, the product in the AQ is set to zero.
- operation is terminated.
- If neither of the operands is equal to zero the process continues with the exponent addition.
- The exponent of the multiplier is in Q and the adder is between exponents a and b.
- Transfer the exponents from q to a add the two exponents transfer the sum in to a.
- since both exponents are biased by the addition of a constant, the exponent sum will have double this bias.

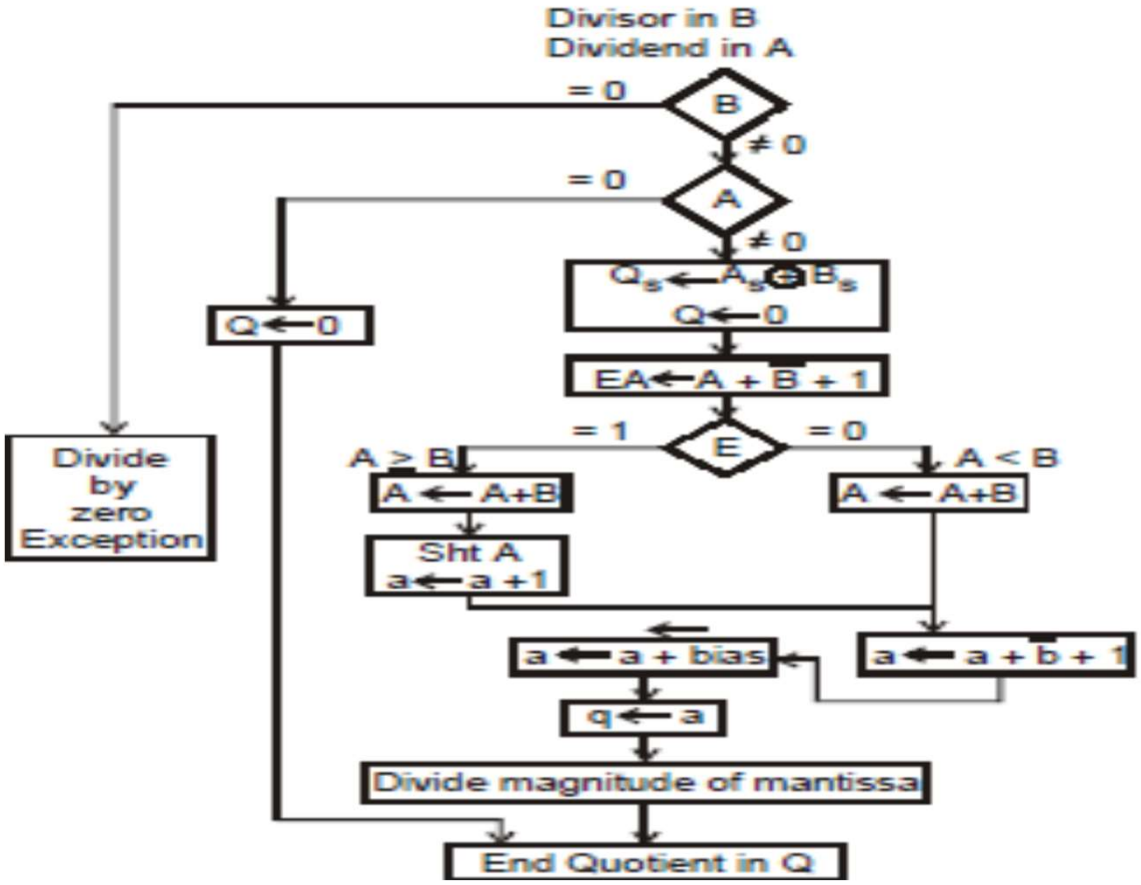
Floating point Multiplication

- The multiplication of the mantissas is done as in fixed-point case with the product residing in A and Q.
- Overflow can’t occur during multiplication so there is no need to check for it.
- The product may have an underflow, so MSB in A is checked.
- If it is a 1, the product is already normalized.
- If it is a 0 the mantissa in AQ is shifted left and the exponent decremented.

Floating point Division

- The division algorithm can be subdivided in to five parts.
- Check for zeros.
- Initialize register and evaluate the sign.
- Align the dividend.
- Subtract the exponent
- Divide the mantissa

Floating point Division



Floating point Division

- The two operands are checked for zeros.
- If the divisor is zero, It indicates and attempt to divide by zero.
- The operation is terminated with an error message.
- If the dividend in AC is zero the quotient in Q is made zero and the operation terminates.

Floating point Division

- If the operands are not zero, we proceed to determine the sign of the quotient and Store it in Q_s
- The Q register is cleared.
- Dividend alignment:
- The two fractions are compared by a subtraction test.
- The carry in E determines their relative magnitudes. The dividend fraction is restored to its original value by adding the divisor.
- If $A \geq B$ it is necessary to shift A once to the right and increment the dividend exponent.

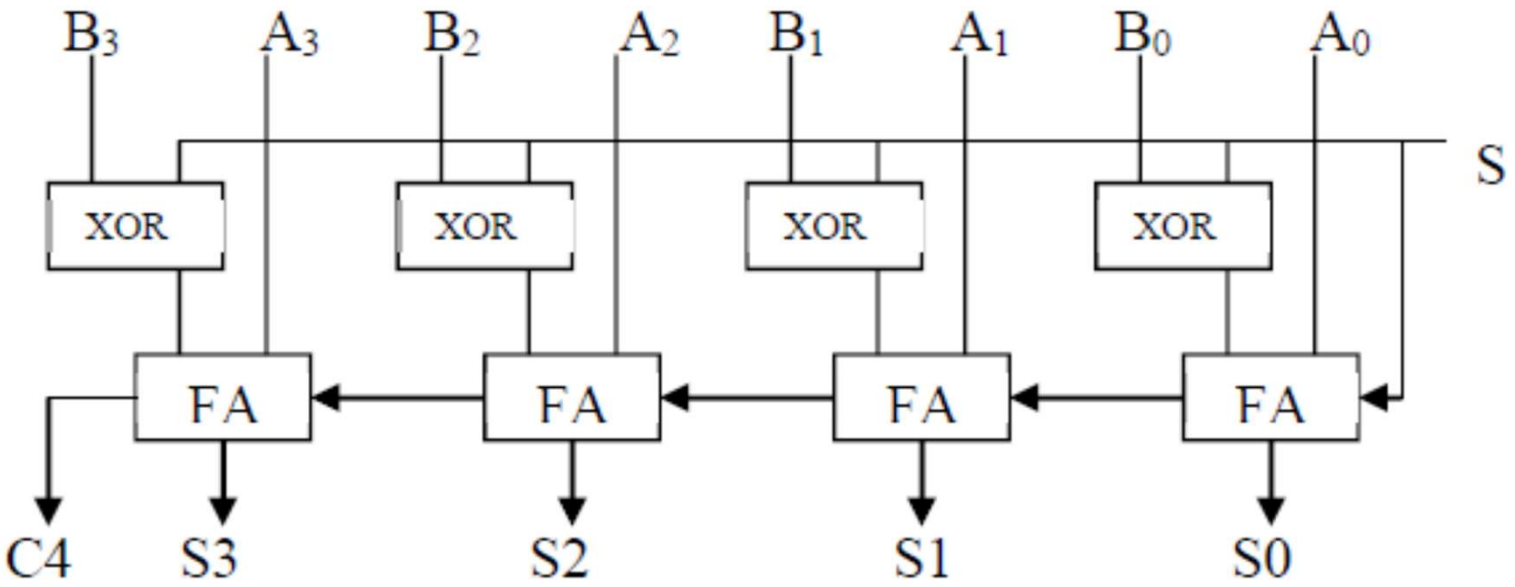
Floating point Division

- Divisor exponent is subtracted from the divided exponent.
- Since both exponents are originally biased, the subtract operation gives the difference with out the bias.
- The bias is then added and the result is transferred in to q because the quotient is formed in Q.
- The magnitude of mantissa quotient resides in Q and the remainder in A

Problem 1

- Design a 4 bit arithmetic circuit with one selection variable S and two 4-bit data inputs A & B.
- When S=0 ➔addition A+B.
- When S=1 ➔Subtraction A-B.

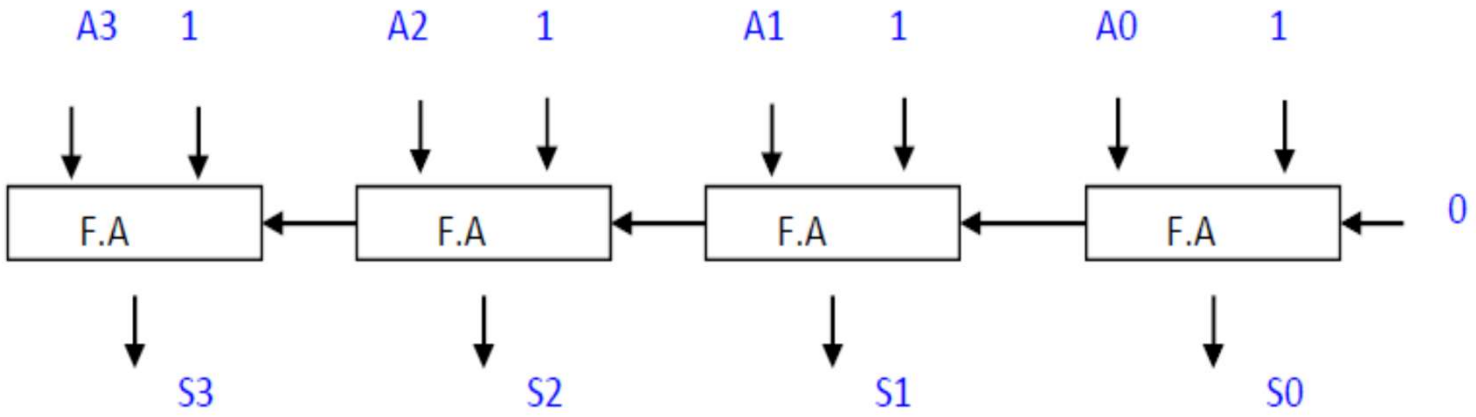
Solution 1



Problem 2

- Design a 4 bit combinational circuit decrementer using 4 FA’s

Solution 2



A-1=A + 2'S Complement of 1

= A +1111.

Problem 3

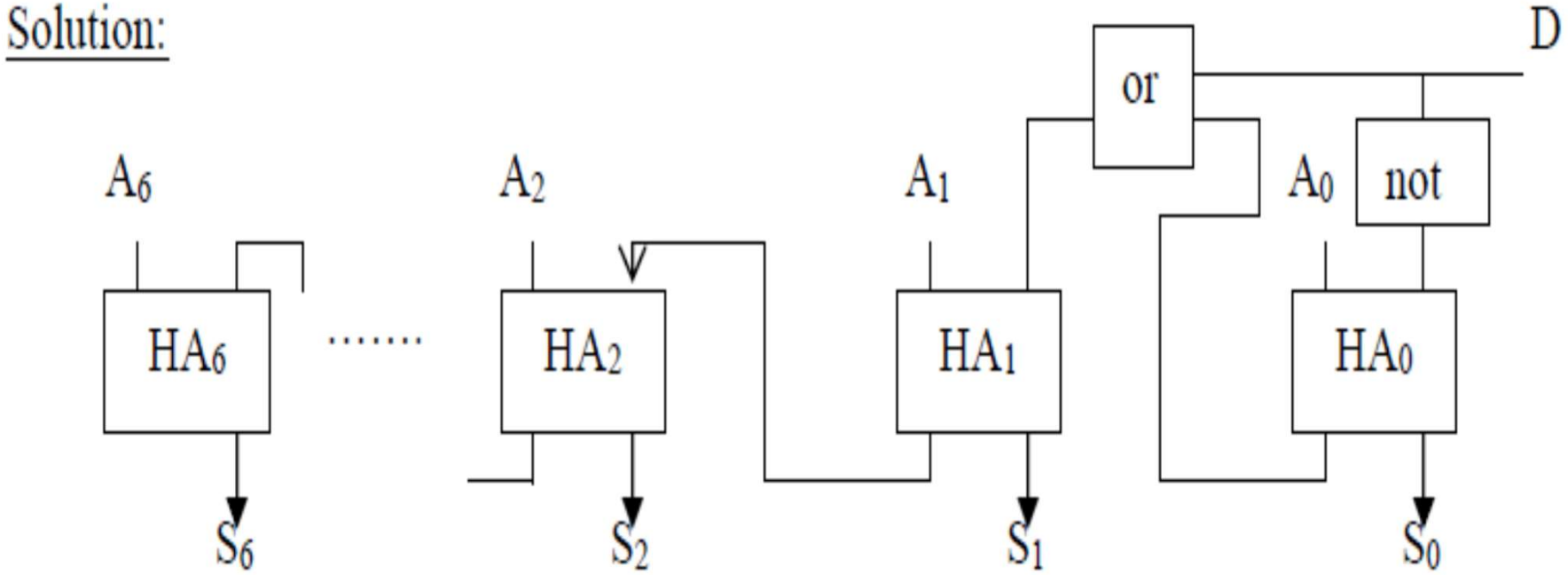
- Design a 7 bit combinational circuit incrementer for the micro program sequencer. Modify the incrementer by including a control input D

  when D = 0 ➔increments by 1,

  when D = 1 ➔ increments by 2.

Solution 3

Solution:



Problem 4

- Multiply the following pairs of signed 2's comp numbers
- using the Booth algorithm.
- $-13 \times -20 = +260$

Solution 4

- Multiplicand is in YS and Multiplier is in QR
- Product in XS.QR
- Need to inspect Qn Qn+1 bits
- YS= 1 1 0 0 1 1 & QR = 1 0 1 1 0 0
- Multiplicand YS = 1 1 0 0 1 1
- YS' +1= 0 0 1 1 0 0 + 1 = 0 0 1 1 0 1

Solution 4

Qn	Qn+1	Initial	XS	QR	Qn+1	SC
			000000	101100	0	110
0	0	ashr XS.QR	000000	010110	0	101
0	0	ashr XS.QR	000000	001011	0	100
1	0	SUB YS	001101			
			001101			
		ashr XS.QR	000110	100101	1	011
1	1	ashr XS.QR	000011	010010	1	010
0	1	add YS	110011			
			110110			
		ashr XS.QR	111011	001001	0	001
1	0	sub YS	001101			
			001000			
		ashr XS.QR	000100	000100	1	000

Problem 5

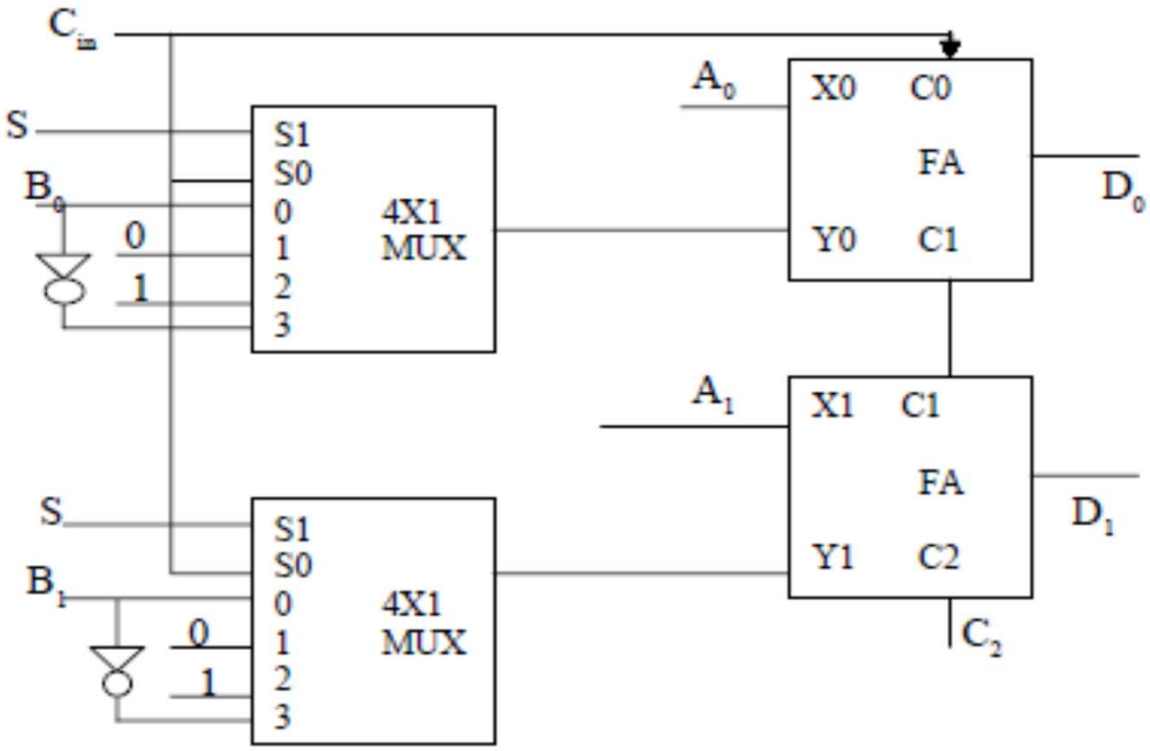
- Design an arithmetic circuit with one selection variable S and two data inputs A & B The circuit generates the following 4 arithmetic operations in conjunction with the input carry Cin . Draw the logic diagram for the first two stages.

S	Cin=0	Cin=1
0	D=A+B	D=A+1(increment)
1	D=A-1(decrement)	D=A+B'+1(sub)

Solution 5

S	Cin	X	Y	D
0	0	A	B	A+b
0	1	A	0	A+1
1	0	A	1	A-1
1	1	A	B’	A-B

Solution 5


C_{in}
S
B_0
0
1
S1
S0
0
1
2
3
4X1
MUX
A_0
X0
C0
FA
Y0
C1
D_0
A_1
X1
C1
FA
Y1
C2
D_1
C_2
S
B_1
0
1
S1
S0
0
1
2
3
4X1
MUX

Problem 6

- Bring out the differences between Restoring and Non-Restoring methods of Division.

Solution 6

Restoring Division	Non Restoring Division
1.Needs restoring of Register A if the result of the subtraction is negative	1. Does not need restoring
2.In each cycle the contents of register A is shifted left and then the divisor is subtracted from it.	2.In each cycle the contents of register A is shifted left and then the divisor is added or subtracted from the contents of register A depending on the sign of A.
3.Does not need restoring of reminder	3.Needs restoring of reminder if the reminder is negative.
4.Slower Algorithm	4.Faster Algorithm

Summary

- Discussed about Fixed point arithmetic operations
- Discussed about Floating point arithmetic operations

References

- David A. Patterson and John L. Hennessey, “Computer Organization and Design‟, Fifth edition, Morgan Kauffman / Elsevier, 2014.
- V.Carl Hamacher, Zvonko G. Varanesic and Safat G. Zaky, “Computer Organisation“, VI edition, Mc Graw-Hill Inc, 2012.
- William Stallings “Computer Organization and Architecture”, Seventh Edition , Pearson Education, 2006.
- Vincent P. Heuring, Harry F. Jordan, “Computer System Architecture”, Second Edition, Pearson Education, 2005.
- Computer System Architecture by Morris Mano