

CHAPTER III

ARITHMETIC UNIT

3.1 THE ARITHMETIC AND LOGIC UNIT

The ALU performs arithmetic and logical operations on data. All other components of computer system bring data into ALU for processing. The information handled in a computer is generally divided into “words”, each consisting of a fixed number of bits. For eg., if the words handled by a microcomputer is 32 bits in length, the ALU would have to be capable of performing arithmetic operations on 32 bits words. Data are presented to the ALU 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 one or more registers called accumulator, or general purpose registers. The accumulator is the basic register containing one of the operands during operations in ALU. If the computer is instructed to add, the number stored in the accumulator is augend and the addend will be located and these operands (addend & augend) are added up and the result is stored back in the accumulator. The original augend will be lost in accumulator after addition.

Arithmetic operations

Arithmetic instruction in digital computers manipulates data to produce result necessary for the solution of computational problem. These instructions perform arithmetic calculation and are responsible for bulk of activity in processing data in a computer. The 4 basic arithmetic operations are

- 1) Addition
- 2) Subtraction
- 3) Multiplication
- 4) Division

From the 4 basic operations it is possible to formulate other arithmetic functions and solve scientific problem by means of numerical analysis methods.

An arithmetic processor is the part of a processor unit that executes arithmetic operation. The data type assumed to reside in processor register during the execution of an arithmetic instruction, is specified in the definition of the instruction. An arithmetic instruction is specified in the definition of the instruction. An arithmetic instruction may specify binary or decimal data and in each case the data may be fixed point or floating point form. Fixed-point number may represent integers or fractions. Negative number may be in signed-magnitude or signed-complement form.

A step-by-step procedure involved in solving any problem is called an algorithm. In this chapter we develop the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We consider the addition, subtraction, multiplication and division for the following types of data.

1. Fixed-point (data) binary data in signed-magnitude representation.
2. Fixed-point binary data in signed 2's complement representation.
3. Floating-point binary data
4. Binary-coded decimal (BCD) data

Fixed point ALU

The following fig shows the most widely used ALU design. It is intended to implement multiplication and division using one of the sequential digit by digit shift and add/ subtract algorithms. Three one word registers are used for operand storage: Accumulator AC, the Multiplier Quotient register MQ and the data register DR. AC and MQ are organized as a single register

AC.MQ capable of left and right shifting. The main additional data processing capability is provided by the parallel adder that receives inputs from AC and DR and places its results in AC. The 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.

In many instances DR serves as a memory buffer register to store data addressed by the instruction address field ADR.

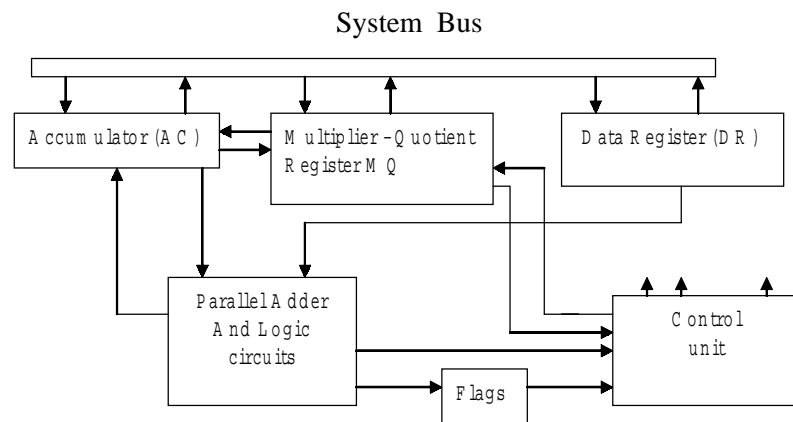


Fig: Block diagram of fixed point ALU

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

It is possible to construct an entire fixed point ALU on a single IC chip especially if the word size m is kept fairly small ex: 4 or 8 bits. Such an m -bit ALU can be designed to be expandable in that k copies of the ALU chip can be connected to form a single ALU capable of processing km -bit operands directly. The resulting array like circuit is called bit sliced because each component chip processes an independent slice of m bits from each km bit operand. Bit sliced ALU's have advantage that any desired word size or even several different word sizes can be handled by selecting the appropriate number of components (bit slices) to use.

The following fig shows a 16-bit ALU can be constructed from 4-bit ALU slices. The data buses and 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. The required control lines are derived from an external control unit which is usually micro programmed. Certain operations require information to be exchanged between slices. For ex if a shift operation is to be implemented then each slice must send a bit to and receive a bit from left or right neighbors .Similarly when performing addition the carry bits may have to be transmitted between the neighboring slices.

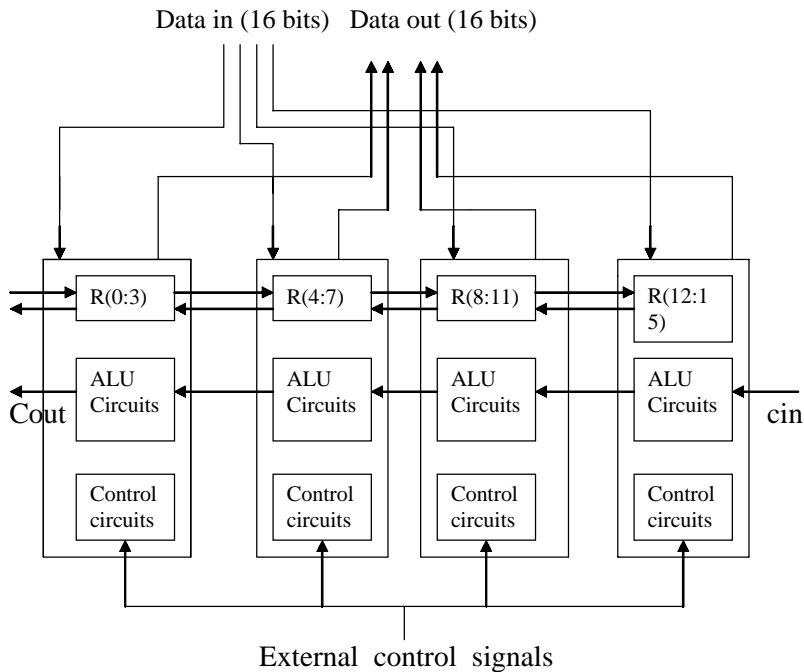


Fig: Bit Sliced ALU

3.2 INTEGER REPRESENTATION FIXED POINT

The numbers used in digital machines are represented using binary digits 0 and 1. The storage element called flipflops can hold this digits. Group of flip flops forms a register in computer systems.

Eg. The number 41 can be represented as

0 0 1 0 1 0 0 1 [8 bit representation]

Thus if a register contains 8 bits, a signed binary number in the system will have 7 magnitude bits or integer and a single sign bit and the system is called signed magnitude binary integer system.

Eg. +18 = 0 0 0 1 0 0 1 0
 -18 = 1 0 0 1 0 0 1 0

3.2.1 Signed-Magnitude Representation

The positive and negative numbers are differentiated by treating the 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 → 1 0 0 0 0 0 0

3.2.2 Two's Complement Representation

In two's complement system, forming the 2's complement of a number is done by subtracting that number from 2^N .

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 Extensions

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$.

To do this,

In sign-magnitude notation,

- simply move the sign bit to the new leftmost position and fill in with zeros.

In 2's complement notation

- 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 = 00010010 ← 8 bit notation
 = 0000000000010010 ← 16 bit notation in sign magnitude
 -18 = 11101110 ← 2's complement 8 bit notation
 = 100000000110110 ← 2's complement 16 bit notation

Both the sign-magnitude and 2's complement representation discussed above are often called as fixed point notation as the decimal point (binary) read point is fixed and assumed to be to the right of the rightmost digit.

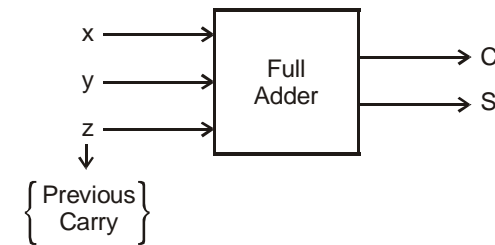
3.3 INTEGER ARITHMETIC

Arithmetic operations occur at the machine instruction level. These arithmetic operations are implemented in the ALU of the processor. As given above, arithmetic operations are performed on fixed point, floating point and binary coded decimal data.

In both fixed point signed magnitude notation and 2's complement notation, the sign bit is treated as the same as the other bits.

3.3.1 Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables denoted by x and y, represent the two significant bits to be added. The 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. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry.



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

In the truth table, when all input bits are 0's, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

Hardware Implementations

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

For S

yz \ x	0	1
00		1
01	1	
11		1
10	1	

For C

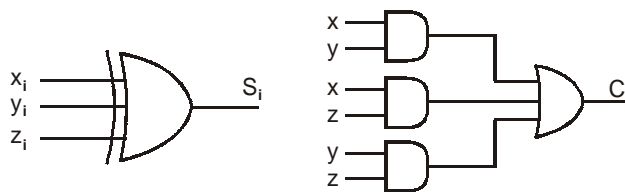
xy \ z	0	1
00		
01		1
11	1	1
10		1

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

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

$$C = xy + xz + yz$$

z is carry-in, therefore



Thus the logic expression for S_i can be implemented with a 3 input XOR gate and logic expression for $C_{out}(C)$ is implemented with a two-level AND-OR logic circuit.

A cascaded connection of n full adder can be used to add two n-bit numbers. Since the carry propagates or ripples, this cascade of full address is called n-bit ripple carry-adder.

To perform the subtraction operation $X-Y$ on 2's complement numbers X and Y, 2's complement of Y is added to X.

Note

Overflow occur

1. When the signs of the two operands are the same and if the sign of the result is different.

Two circuits are possible to check overflow.

1. Implementing the expression

$$\text{Overflow} = x_{n-1} y_{n-1} \bar{S}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} S_{n-1}$$

2. Implementing the expression

$$\text{Overflow} = C_n \oplus C_{n-1}$$

Note

- i) If the overflow occurs, the ALU must signal this fact so that no attempt is made to use the result.
- ii) Overflow can occur whether or not there is a carry.

3.3.2 4-bit binary adder

To implement the add micro operation with the hardware we need the register that hold the data and the digital component that perform the arithmetic addition. The digital circuit that performs the arithmetic sum of two bits and a previous carry is called a **Full adder**. The digital circuit that generates the arithmetic sum of two binary numbers of any length is called the binary adder.

The binary adder circuit can be constructed with full adder circuits connected in cascade. Here the output carry from one full adder can be connected to the input carry of the next full adder. The augends bits of X and the addend bits of Y are designated by subscript numbers from right to left as shown below. The carries are connected in chain through to full adders. The input carry to the binary adder is the C_0 and the output

carry to the binary adder is the C_4 . The S outputs of the full adders generates the required sum bits.

The n-bit binary adder requires n-full adders. The n-data bits for the X inputs are from one register R_1 and n-data bits for the Y input come from another register R_2 . The sum can be transferred to the third register or one of the source register. (R_1 or R_2)

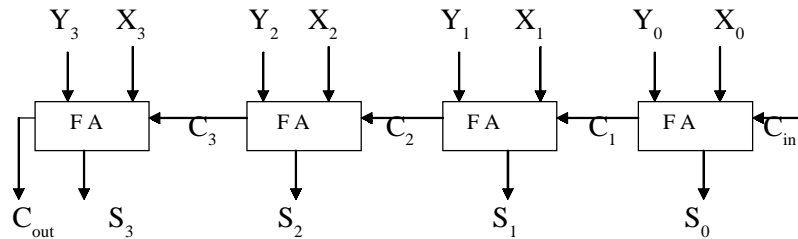


Fig: 4-Bit Binary Adder

3.3.3 4- bit Binary adder / sub tractor

The subtraction of binary numbers can be done by means of complements. The subtraction of X and Y i.e $X-Y$ can be done by taking the 2's complement of Y and adding it to X. The 2's complement can be obtained by taking the 1's complement and adding 1 to it. The 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. When $M=0$ the circuit is an adder, when $M=1$ the circuit is a subtractor. Each XOR \oplus gate receives input M and one of the inputs of B. When $M=0$ we have $Y \oplus 0 = Y$. The full adder receives the value of B. the input carry is 0, and the circuit performs the X plus Y.

When $M=1$ we have have $Y \oplus 1 = \bar{Y}$ and $C_0=1$. The Y inputs are all is complemented and 1 is added through the input carry. The circuit performs X plus 2's complement of Y.

For unsigned numbers this gives the $X - Y$ if $X \geq Y$ or the 2' s complement ($Y-X$) if $X < Y$.

For signed numbers the result is $(X-Y)$ provided there is no overflow.

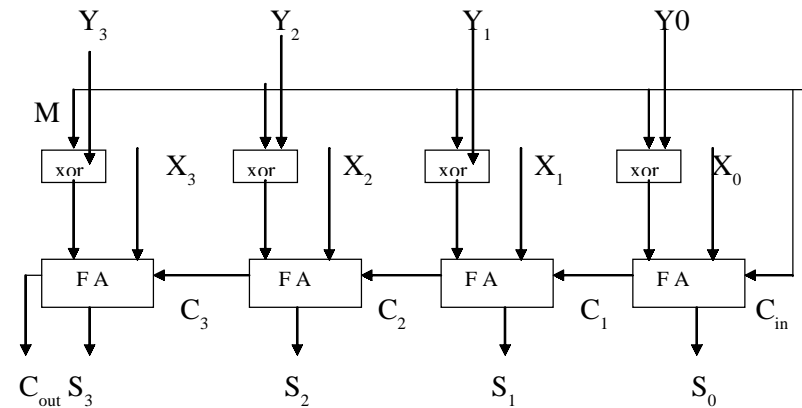


Fig: Binary Adder/ Sub tractor

3.3.4 4-bit binary arithmetic circuit

The 4-bit arithmetic circuit has four full adders. Constitutes a 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 from a go directly to the X input of the binary adder. Each of 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 S_0 and S_1 . The input carry C_{in} goes to the carry input of the FA in the LSB. The other carries are connected from one stage to another.

The output of the binary adder is calculated from the following arithmetic sum.

COMPUTER ARCHITECTURE

$$D = X + B + C_{in}$$

$X \rightarrow$ 4 bit binary number at A inputs.

$Y \rightarrow$ 4 bit binary number at B inputs.

$C_{in} \rightarrow$ Input carry which can be 0 or 1.

By controlling the value of Y with two selection inputs S_0 and S_1 and making C_{in} equal to 0 or 1 it is possible to generate 8 possible arithmetic operations.

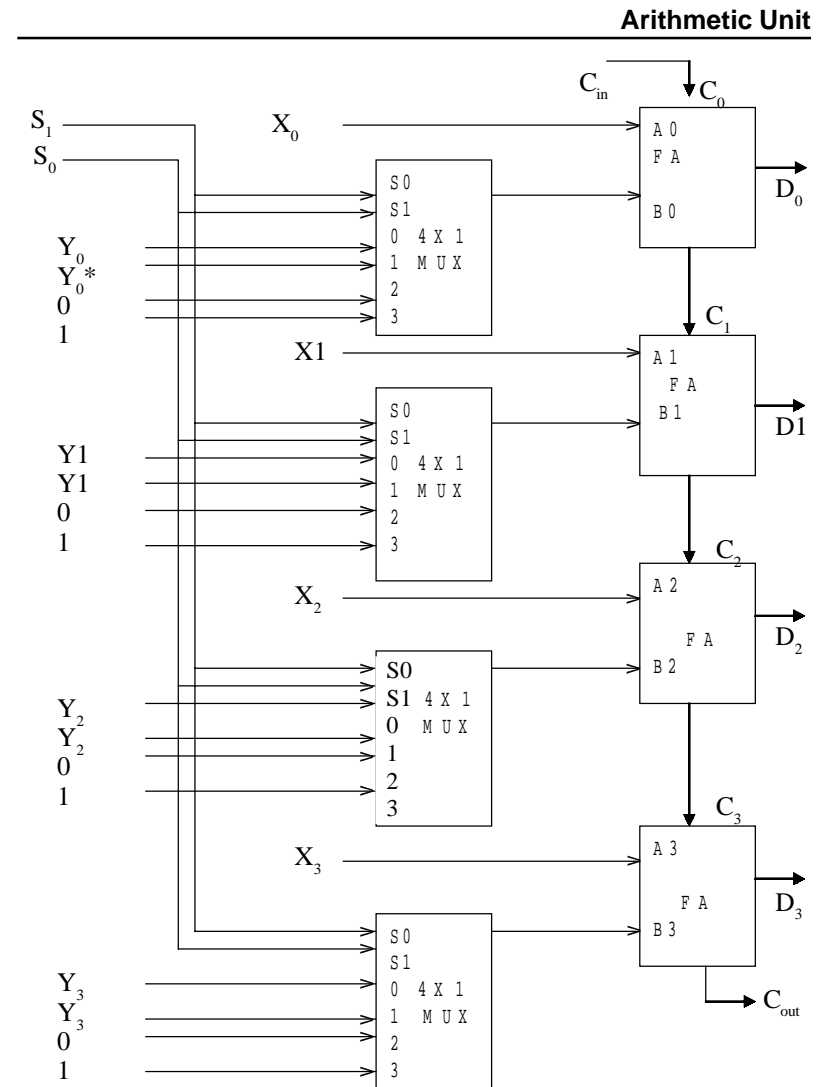
Addition: When $S_1S_0 = 00$, the value of B is applied to the Y input of the adder. If $C_{in} = 0$ the output $D = X + Y$. If $C_{in} = 1$ then the output is $D = X + Y + 1$. Both cases perform the add micro operation with or without carry.

Subtraction: When $S_1S_0 = 01$, the complemented value of Y is applied to the B input of the adder. If $C_{in} = 0$ the output $D = X + (\text{Comp})Y$. This is equivalent to $D = X - Y$. If $C_{in} = 1$ then the output is $D = X + (\text{Comp})Y + 1$. This produces X plus the 2's complement of Y , which is equivalent to $X - Y$. Both cases perform the subtraction micro operation with or without borrow.

When $S_1S_0 = 10$ the input from the B are neglected all 0's are applied to the B input. The output becomes $D = X + 1$.

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 $C_{in} = 0$. 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 $C_{in} = 1$ then $D = X - 1 + 1 = X$.



3.3.5 Addition & Subtraction of Signed Numbers

We designate the magnitude of two numbers by A and B . When signed numbers are added or subtracted we find that there are 8 different conditions to consider, depending on the sign of the number and the operation performed. These conditions are listed in the following table (1st column). The other columns

show the actual operation to be performed with the magnitude of the number. The last column is needed to prevent -0. When two equal number are subtracted the result should be +0 not -0.

Operation	Add Magnitudes	Subtract Magnitudes When		
		X > Y	X < Y	X = Y
(+X) + (+Y)	+(X+Y)			
(+X) + (-Y)		+ (X - Y)	- (X - Y)	+ (X - Y)
(-X) + (+X)		- (X - Y)	+ (X - Y)	+ (X - Y)
(-X) + (-Y)	-(X + Y)			
(+X) - (Y)		+ (X - Y)	- (Y - X)	+ (X - Y)
(+X) - (-Y)	+ (X + Y)			
(-X) - (+Y)	- (X + Y)			
(-X) - (-Y)		- (X - Y)	+ (Y - X)	+ (X - Y)

Addition Algorithm:

- 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.

Hardware Implementation:

To implement the two arithmetic operations with hardware it is first necessary that the two numbers be stored in register. Let X and Y be two register that hold the magnitude of the number and Xs and Ys be two flip flop that hold the corresponding signs. The result of operation may be transferred

to a 3rd register. However a saving is achieved by transferring the result into X and Xs. Thus X and Xs together form an accumulator register.

Consider now the hardware implementation of the algorithm above.

- First a parallel adder is needed to perform the micro operation $X + Y$.
- A comparator circuit is needed to establish if $X > Y$, $X = 0$ or $X < Y$.
- Two parallel subtract circuits are needed to perform the micro operation $X - Y$ and $Y - X$
- The sign relation can be determined from an XOR gate with Xs and Ys input.

This procedure required a magnitude comparator, an adder, and two subtractions. However a different procedure can be found that required less equipment.

- First we know that subtraction can be accomplished by means of complement and addition.
- The result of a comparison can be determined from the end carry after the subtraction.

The following fig shows the hardware implementation of the addition and subtraction operation. It consists of register 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. Where it can be checked to determine the relative magnitude of the two numbers. The add-overflow flip flop VF holds the overflow bit when X and Y are added. The register provides other micro operation that may be needed when we specify the sequence of steps in the algorithm.

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.

1. 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 $X + Y$.
2. 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 $X - Y$.

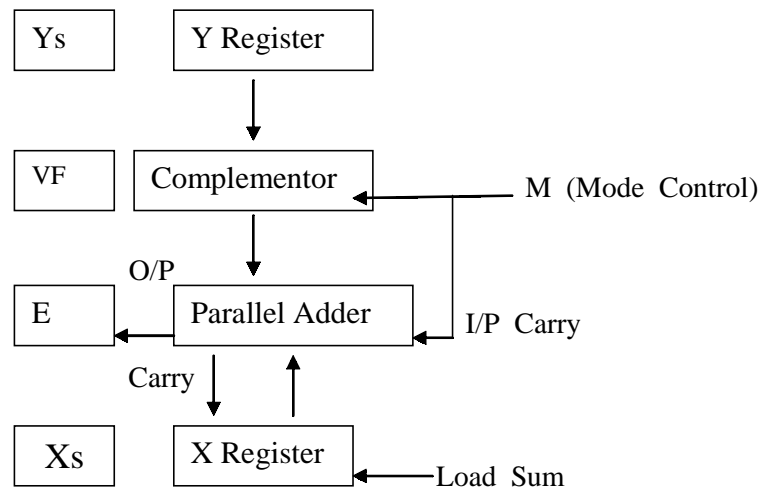


Fig:- H/W for Signed-Magnitude addition and Subtraction

H/W Algorithm

The flow chart for the H/W algorithm is shown below. The two signs A_s and B_s are compared by XOR gate. If the output of

XOR gate is 0 the signs are (equal) identical, if it is 1 the signs are different.

1. For an addition operation the identical signs dictate that the magnitude be added.
2. For subtraction operation different signs dictate that magnitude be added..
3. For magnitude are added with a micro operation.

$EX \leftarrow X + Y$, Where $EX \leftarrow$ 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.

4. The 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.
5. A 1 in E indicates that $X \geq Y$ 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.
6. 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 $X \leftarrow X + 1$. However we assume that the X register has circuits function micro operation complement and increment. So the 2's complement is obtained from these two micro operation. In other paths of the flow charts the sign of the result is the same as the sign of X, so no change in X_s is required. When $X < Y$ the sign of the - result is the complement of the original sign of X. It is then necessary to complement X_s to obtain the correct sign. The final result is found in register X and its sign in X_s . The value in VF provides an overflow indication.

The final value of E is immaterial.

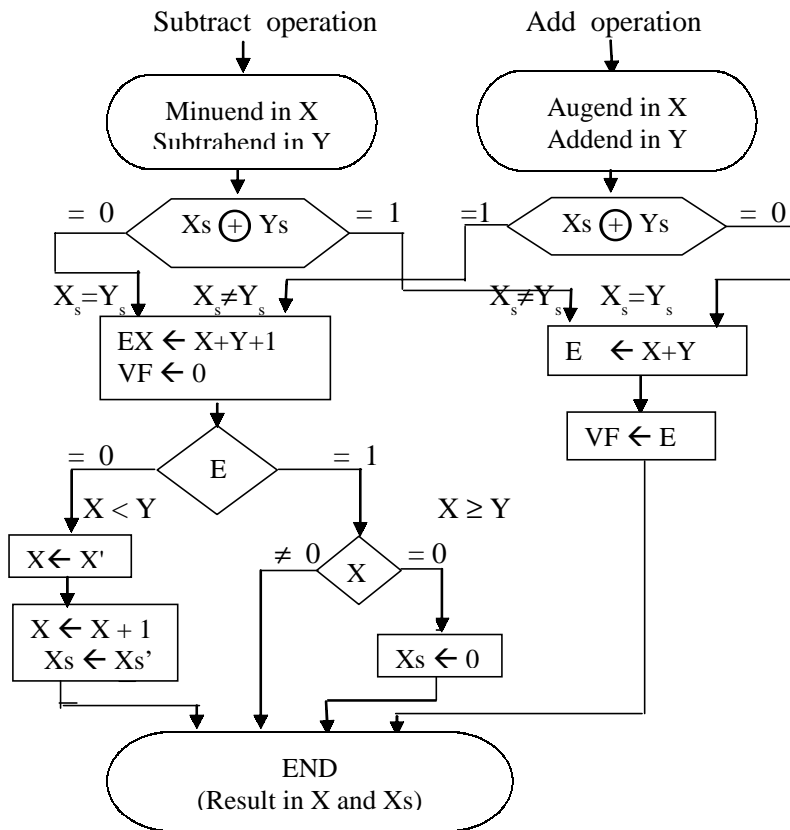


Fig: - Flow chart for add and subtract operation

3.3.5.1 Addition and subtraction with signed 2's complement data

The left most bit of a binary number represents sign bit 0 → positive, 1 → negative. If the sign bit is 1 the entire number is represented in 2's complement form.

Thus +33 Ex: is represented as 00100001 and

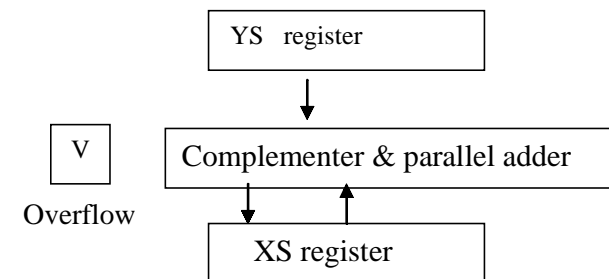
-33 as 11011111. Note that 11011111 is the 2's complement of 00100001 and vice versa.

The addition of two numbers in signed 2's complement form consists of 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 of first taking 2's complement of the subtrahend and then adding it to the minuend.

When two numbers of n digits each are added and the sum occupies n+1 digits. We say that an overflow occurred. 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.

The register configuration for hardware implementation is shown in the figure below. This is same as the figure above but the sign bits are not separated from the rest of the register. We name the X register X_s (Accumulator) and the Y register Y_s . The left most bit in X_s and Y_s represents the sign bits of the number. The two sign bits are added or subtracted together with the other bits in the completer 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.



The algorithm for adding and subtracting two binary number in signed 2's complement is shown below. The sum is obtained by adding the contents of X_s and Y_s (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 Ac to the 2's complement of Y_s . 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.

Comparing this algorithm to its signed magnitude counterpart we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed 2's complement. We adopt this representation over the more familiar signed magnitude.

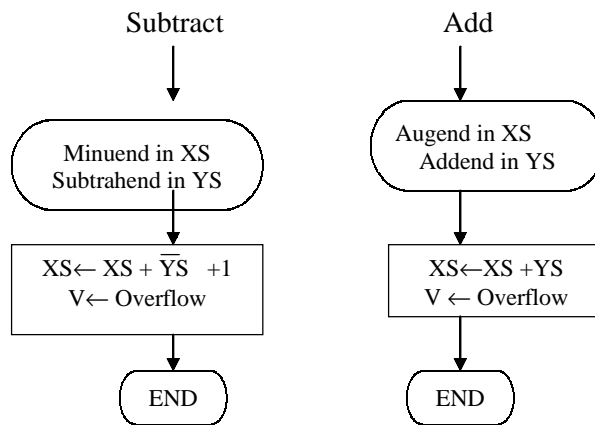
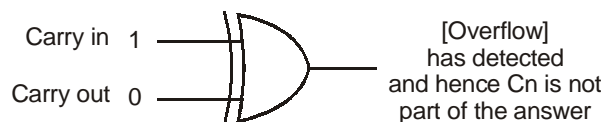


Fig: - Algorithm for addition & Subtraction-2's Complement representation

Eg.

Addition

$$\begin{array}{r} +74 \quad 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0 \\ + \quad \\ +69 \quad 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array}$$



Subtraction

$$\begin{array}{r}
 + 125 \\
 + 90 \\
 \hline
 + 35 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 (-) \quad 01111101 \\
 (+) \quad 10100110 \leftarrow \text{2's complement of 90.} \\
 \hline
 \boxed{1} \quad 00100011 \\
 \downarrow \text{discard Carry}
 \end{array}$$

3.3.6 Design of Fast address

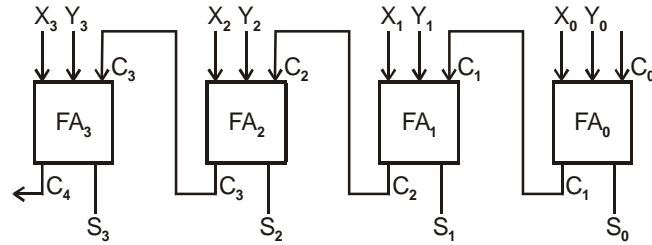
Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and the addend are available for computation at the same time.

The parallel address are ripple carry types in which the carry output of each full adder stage is connected to the carry input of the next higher-order stage. The sum of carry outputs 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.

Full adder 1 cannot produce a potential carry output until a carry input is applied. i.e. 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 carries 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.



3.3.6.1 The 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 this ripple carry delay is called Look-Ahead Carry addition and is based on two functions of the full adder called the carry generate and the carry propagate function

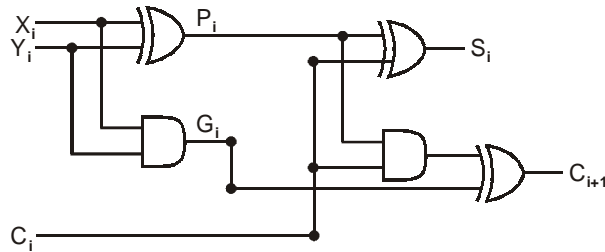


Fig. Full-Adder-Bit Stage Cell.

From the full adder circuit,

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

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

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

$$\text{Generate } G_i = x_i y_i$$

Now the expression for the carry out co. of each full adder stage for the four bit example.

$$\text{Propagate } P_i = x_i + y_i$$

FA₀ For first full adder

$$C_1 = G_0 + C_0 P_0$$

FA₁ For second full adder

$$C_2 = G_1 + C_1 P_1$$

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

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

Similarly

$$C_3 = G_2 + C_2 P_2$$

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

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

In all the above expressions, the carry output for each full-adder stage is dependent only on the initial input carry (C_0), its G_0 and P_0 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.

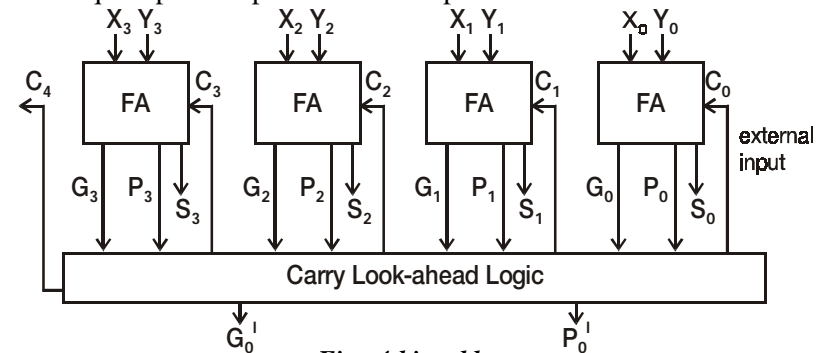


Fig. 4-bit adder.

In general, the final expression for any carry variable is

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_1 P_0 C_0$$

All carries can be obtained three gate delays after the input signals X, Y and C_0 are applied because only one gate delay is needed to develop all P_i and G_i signals, and followed by two gate delays in the AND-OR circuit for C_{i+1} (See fig. Full-Adder-Bit stage cell)

After a further XOR gate delay, all sum bits are available. Therefore, independent of n, the n-bit addition process requires only four gate delays.

Delays in Carry Look ahead adder

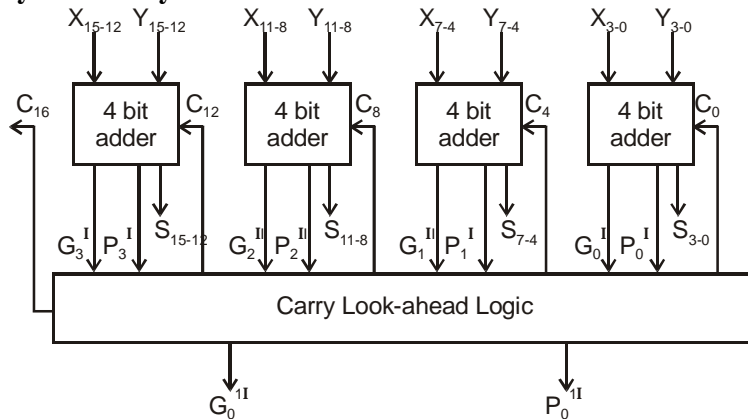


Fig. 16 bit Carry-Lookahead adder built from 4-bit adders.

In carry-look ahead 4-bit adder, all carries are obtained three gate delays after the input signals X, Y and C_0 are applied because one gate delay is needed to develop all P_i and G_i signals followed by two gate delays in the AND-OR circuit for C_{i+1} . After a further XOR gate delay, all sum bits are available. Therefore independent of n, the n-bit addition process requires only four gate delays.

In the above fig., the carryout C_4 from the low order adder is available 3 gate delays after the input operands X, Y and C_0 are applied to the 16 bit carry lookahead adder.

C_8 is available after a further 2 gate delays, C_{12} is available after a further 2 gate delays and C_{16} is available after a further 2 gate delays.

(i.e.) C_{16} is available after a total of

$$(3 \times 2) + 3 = 9 \text{ gate delays}$$

If a ripple carry adder is used, C_{16} is available only after 31 gate delays for S_{15} and 32 gate delays for C_{16} .

Higher-Level generate and propagate functions

By using Higher-Level block generate and propagate functions, it is possible to use the lookahead approach to develop the carries C_4 , C_8 , C_{12} in parallel in a higher-level carry lookahead circuit.

In Fig. 16 bit Carry Look Ahead Adder

$$P_0^I = P_3 P_2 P_1 P_0 \quad \&$$

$$G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

and

C_{16} can be

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

Gate-delays when higher level block generate and propagate functions are used

Carry generated internally by the 4-bit adder blocks are not needed because they are generated by the higher-level carry lookahead circuits.

G_k^I & P_k^I are produced after 3 gate delays after the generation of G_i and P_i .

The delay in developing the carries produced by the carry lookahead circuits is two gate delays more than delay needed to develop G_k^I & P_k^I

and hence totally 5 gate delays after X , Y and C_0 are applied as inputs. Then the sum is produced after further 3 gate delays (C_{15} after 2 gate delays when C_{12} is available and 1 further gate delay) thus totally 8 gate delays. Therefore when cascaded 4 bit adder is used, S_{15} and S_{16} are available after 10 and 9 gate delays. When higher level carry lookahead adder is used, S_{15} and C_{16} are available after 8 and 5 gate delays.

3.3.7 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.

Ex:-

11	1 0 1 1	Multiplicand
9	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	

The process consists of looking at successive bits of the multiplier, LSB first. If the multiplier bit is a 1 the multiplicand is copied down; otherwise zeroes are copied down. The number copied down in successive lines are shifted one position to the left from the previous number finally the number are added and their sum forms a product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike the sign of the product is positive. If they are unlike the sign of the product is negative.

Multiplication algorithm using signed - Magnitude data

- 1) Instead of providing register to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register.
- 2) 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.
- 3) 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.

The H/w implementation for the multiplication is shown below. 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.

Initially, the multiplicand is in register Y and the multiplier in Q. The sum of X and Y forms a partial product, which is transferred to the EX reg. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement Shr EXQ to designate the right shift depicted.

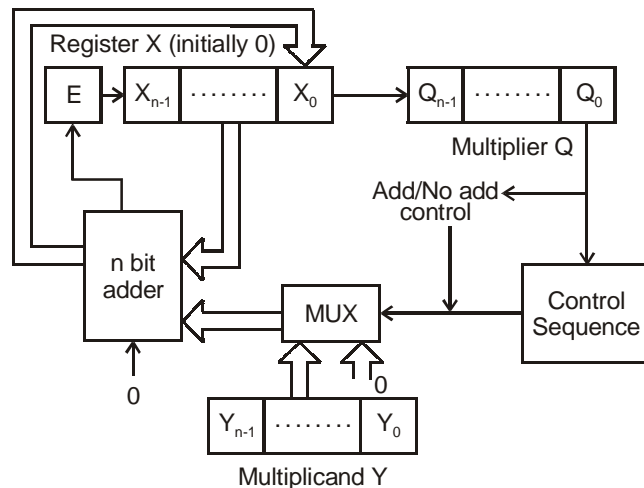


Fig:- H/w for multiply Operation

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. In this manner the right most FF in register Q, designated by Q_n ; will hold the bit of the multiplier, which must be inspected next.

H/W algorithm:-

Initially the multiplicand is in Y and the multiplier in Q. their corresponding signs are in Y_s and Q_s , respectively. The signs are compared, and 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 and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to reg. From a memory unit that has words of n bits. Since 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.

After the initialization the low order bit of the multiplier in Q_n 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 1 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 s to ps when $SC=0$ note that the 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

Multiply Operation

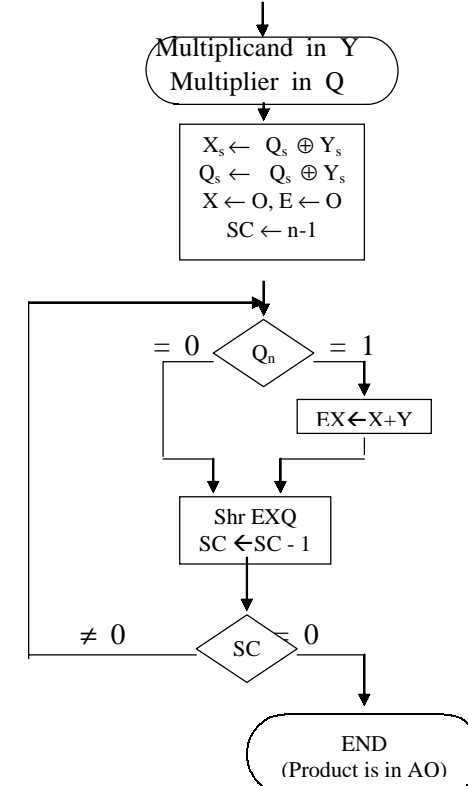


Fig: - Flow chart for the multiply operation.

Eg.

$$23 \times 19 = 437$$

Multiplicand Y = 1 0 1 1 1 \leftarrow 23

Multiplier Q = 1 0 0 1 1 \leftarrow 19

	E	X	Q	SCounter = no. of bits in the multiplier in the binary form
--	---	---	---	--

	0 0 0 0 0	1 0 0 1 1	1 0 1
--	-----------	-----------	-------

Q_n=1, ADD Y 1 0 1 1 1

First partial Product 0 1 0 1 1 1

Shift right EXQ 0 0 1 0 1 1 1 1 0 0 1 1 0 0

Q_n = 1 , Add y 1 0 1 1 1

Second partial Product 1 0 0 0 1 0

Shift right EXQ 0 1 0 0 0 1 0 1 1 0 0 0 1 1

Q_n = 0 , No add

Shr EXQ 0 0 1 0 0 0 1 0 1 1 0 0 1 0

Q_n = 0 , No add

Shr EXQ 0 0 0 1 0 0 0 1 0 1 1 0 0 1

Q_n = 1 ADD Y 1 0 1 1 1

1 1 0 1 1

Shr EXQ 0 0 1 1 0 1 1 0 1 0 1 0 0 0

Since SC becomes zero

Final product is in XQ

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

Using this sequential hardware structure, it is clear that a multiply instruction takes much more time to execute than an Add instruction.

3.3.7.1 Array multiplier

The multiplication of two binary number can be done

with one loop by means of a combinational circuit that forms the product bits all at once. This is a first way of multiplying two no since all it takes is the time for the signals to propagate through the gates that form a multiplication array however an array multiplier reg. A large no of gates and for this reason it was not economical until the development of integrated circuits.

To see how an array multiplier can be implemented with a combinational circuit consider the multiplication of two 2 - bit number as shown below.

The multiplicand bits are y_1 and y_0 and the multiplier bits are x_1 and x_0 and the product is $p_3 p_2 p_1 p_0$.

The first partial product is formed by multiplying a_0 by $y_1 y_0$. The multiplication of two bits such as x_0 and y_0 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. As shown in fig the first partial product is formed by means of two AND gates.

The second partial product is formed by multiply by x_1 by $y_1 y_0$ and is shifted one position to the left. The two partial products are added with two half adder (HA) circuits. Note that LSB of the product does not have to get through an adder since it is formed by o/p of first AND gate.

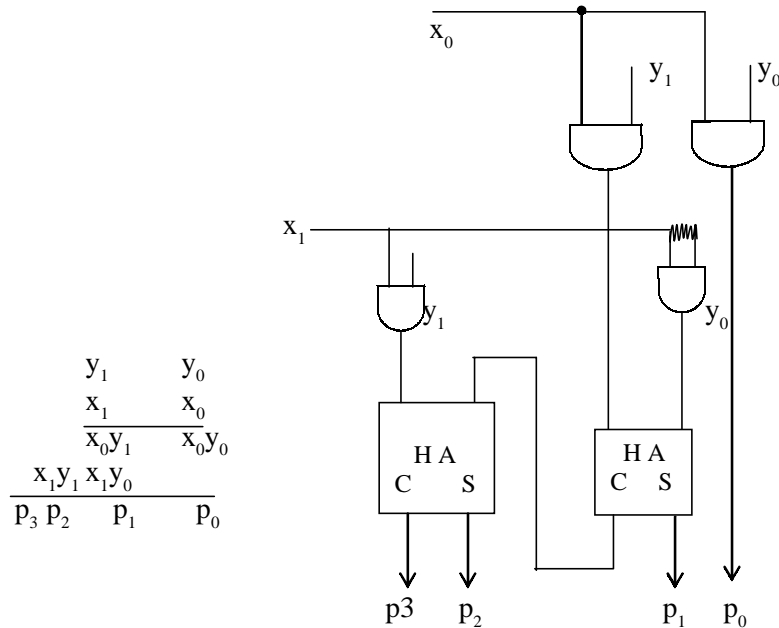


Fig: -2 - bit by 2 - bit array multiplier

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is AND ed with each bit of multiplicand in as many levels as there are bits in a multiplier.

For j multiplier bits and k multiplicand bits need $j \times k$ AND gates and $(j-1)$ k -bit adders to produce a product of $j + k$ bits.

3.3.8 Signed Operand Multiplication

This topic discusses multiplication of 2's complement signed operands, generating a double length product.

3.3.8.1 Booth Multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation. It operates on the fact that a string of 1's in the multiplier register on add but just

shifting and a string of 0's in the multiplier from bit wt 2^k to wt 2^m can be treated as $(2^{k+1} - 2^m)$ for ex the bin no 001110 (+14) has a string of 1's from 2^3 to 2^1 ($k=3, m=1$) the no can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$

Therefore Multiplication $M \times 14$ where $M \rightarrow$ multiplicand and 14 the multiplier; can be done as $(M \times 2^4 - M \times 2^1)$. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifting left ones.

As in all multiplication schemes, booth algorithm register examine of the multiplier bits and shifting of the partial product prior to the shifting the multiplicand may be added to the partial product, subtracted from the partial product or left unchanged according to the following rules

- 1) The multiplicand is sub from the partial product upon encountering the first LSB 1 in a string of 1's in the multiplier
- 2) 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.
- 3) 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. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight

Ex:- Multiplier equal to -14 in 2's complement form \rightarrow 110010 and is treated as

$$-2^4 + 2^2 - 2^1 = -14$$

The H/W implementation of Booth algorithm requires the register configuration as shown below. This is similar to H/W fig above except that the sign bits are not separated from the

rest of the reg. To show this different we rename the reg. A, B and Q as AC, BR and QR respectively. Q_n designates the LSB of multiplier in reg. QR. An extra FF Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier

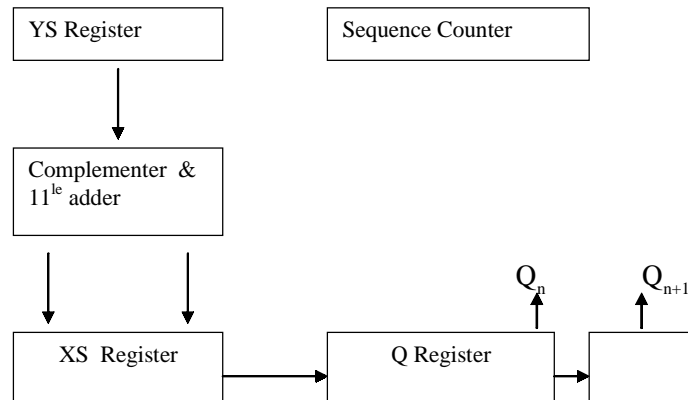


Fig:- H/W for Booth algorithm

The flow chart for Booth algorithm is shown below. XS and appended bit Q_{n+1} are initially cleared to 0 and the 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 the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a substitution of the multiplicand from the partial product in XS . Where the two bits are equal, the partial product does not change. An overflow can't occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including Bit Q_{n+1}). This is an arithmetic shift (ashr) operation which shifts XS and YS to the right and leaves the sign bit in XS unchanged. The sequence counter (SC) is decremented and the computational loop is repeated n times.

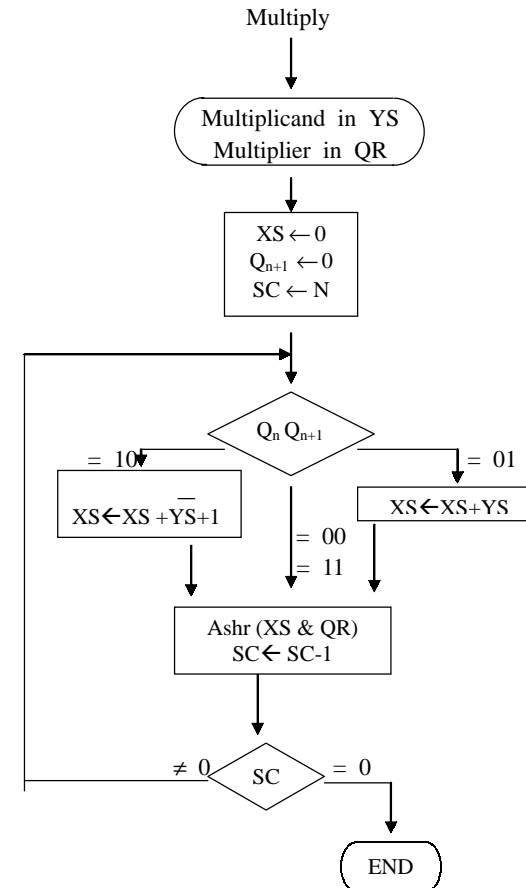


Fig:- Booth Algorithm for multiplication of signed - 2's complementer

A numerical example of Booth algorithm for $n = 5$ it shows the step-by-step multiplication of $(-9) \times (-13) = 117$. Note that the multiplier in QR is negative and that the multiplicand in YS is also negative. The 10-bit product appends in XS and QR and is positive. The final value of Q_{n+1} the original sign bit of the multiplier and should not be taken as part of the product

Example Case (i) $-ve \times -ve$
 -9×-13

$YS = 10111 = -9$

$\overline{YS} + 1 = 01001$

Eg 2: -9×-13

I Step find Booth recoding of a Multiplier

$$\begin{array}{ccccc} 1 & 0 & 1 & 1 & 1 \\ -1 & 0 & +1 & 0 & -1 \end{array} \quad \times$$

Arithmetic Unit

In the above example; the transformation

Multiplier		Version of Multiplicand selected by bit i
Bit i	Bit i-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Fig. Booth Multiplier recording table.

																	implied
Ordinary	1	1	0	0	0	1	0	1	1	0	1	1	1	0	0	1	0
Multiplr (average)	0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	-1	0	+1	-1	0

implied

Good	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	<div style="border: 1px solid black; padding: 2px;">0</div>
Multiplr	0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1	

Booth Algorithm Advantages

- Case (ii)** +ve \times +ve

$$\text{BR} = 0 \ 0 \ 1 \ 1 \ 0 = -9$$

$$\overline{\text{BS}}_{+1} = 1\ 1\ 0\ 0\ 1 + 1 \rightarrow 1\ 1\ 0\ 1\ 0$$

Arithmetic Unit

Qn	Q _{n+1}	Initial	AC	QR	Q _{n+1}	SC
			0 0 0 0 0	1 0 1 1 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 1 0 1 1	1	1 0 0
1	1	Ashr AC	1 1 1 0 1	0 1 0 1 1	1	0 1 1
1	1	Ashr AC	1 1 1 1 1	0 1 0 1 0	1	0 1 0
0	1	Add BR	0 0 1 1 0			
			0 0 1 0 1			
		Ashr	0 0 0 1 0	1 0 1 0 1	0	0 0 1
1	0	Sub BR	1 1 0 1 0			
			1 1 1 0 0			
		ashr	1 1 1 1 0	0 1 0 1 0	1	0 0 0
		Final Pdt in 2's comp form				
			0 0 0 0 0	1 0 1 0 1		
			0 0 0 0 1	1 0 1 1 0		

There are two techniques for speeding up the multiplication operation.

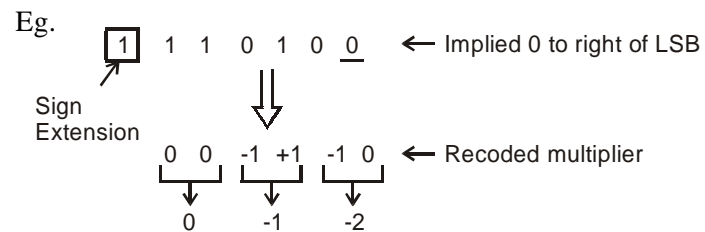
(i) Bit Pair Recoding of Multipliers

(ii) Carry Save Addition of Summands

3.3.9.1 Bit Pair Recoding of Multipliers

Guarantees that the max number of summands (versions of the multiplicand) that must be added is reduced by half and is derived from Booth's Alg.

(i) Group the Booth recoded Multipliers bits in pairs.



In the recoded multiplier bits,

(+1, -1) is equivalent to (0 +1) i.e. Instead of adding -1 times the multiplicand M at shift position i to +1×M at position i+1, the same result is obtained by adding +1×M at position i.

Similarly (+1 0) is equivalent to (0 +2)

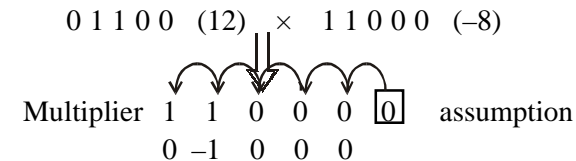
(-1 +1) is equivalent to (0 -1) So on.

(ii) 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.

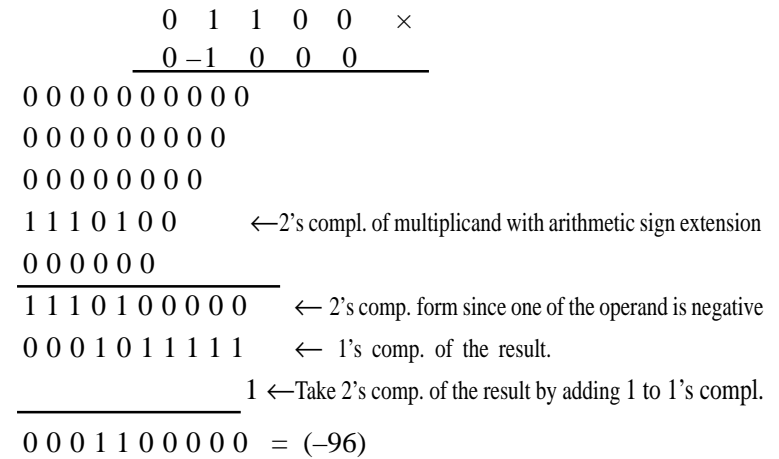
Table of Multiplicand Selection decisions.

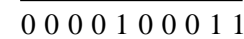
Multiplier bit Pair i+1 i	Multiplier bit on the right i-1	Multiplicand Selected at position i
0 0	0	$0 \times M$
0 0	1	$+1 \times M$
0 1	0	$+1 \times M$
0 1	1	$+2 \times M$
1 0	0	$-2 \times M$
1 0	1	$-1 \times M$
1 1	0	$-1 \times M$
1 1	1	$0 \times M$

Example 1



Method I : (Booth Multiplication)





3.3.9.2 Carry Save addition of summands

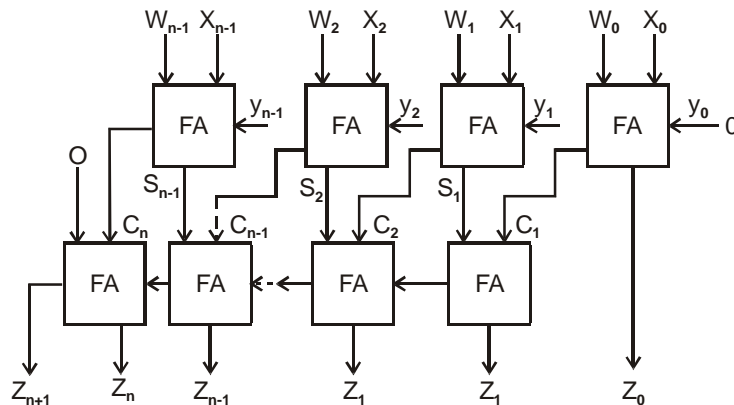
Multiplication of 2 numbers involves the addition of several summands (partial products). A technique called carry save addition speeds up the addition process.

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



Eg :

$$\begin{array}{rcl}
 W & = & 10101 \\
 X & = & 11011 \\
 \hline
 & & 11000 \\
 Y & = & 010100 \\
 \hline
 Z & = & 1000100
 \end{array}$$

Case (ii) Addition Using Carry save adder

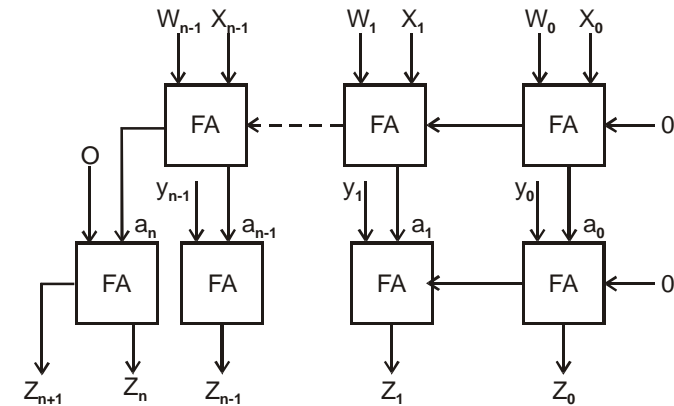


Fig. Addition using ripple carry adder.

Eg.

$$\begin{array}{rcl}
 W & = & 10101 \\
 X & = & 11011 \\
 Y & = & 10100 \\
 S & = & 11010 \\
 C & = & 10101 \\
 Z & = & 1000100
 \end{array}$$

In carry save addition, instead of letting the carries ripple along the rows, they are saved and introduced into the next row, at the correct weighted positions.

Carry save addition transforms W, X & Y into S & C. Its advantage is that all bits of S and C Vectors are produced in a short, fixed amount of time after W, X and Y are applied. Carry propagation takes place only in the second row.

Example 2

Consider the multiplication operation $M \times Q = P$ for 6 bit operands

1 0 1 1 0 1	(45) M
1 1 1 1 1 1	× (63) Q
1 0 1 1 0 1	W1
1 0 1 1 0 1	W2
1 0 1 1 0 1	W3
1 0 1 1 0 1	W4
1 0 1 1 0 1	W5
1 0 1 1 0 1	W6
1 0 1 1 0 0 0 1 0 0 1 1	

Fig. 2: Using Ripple Carry Addition

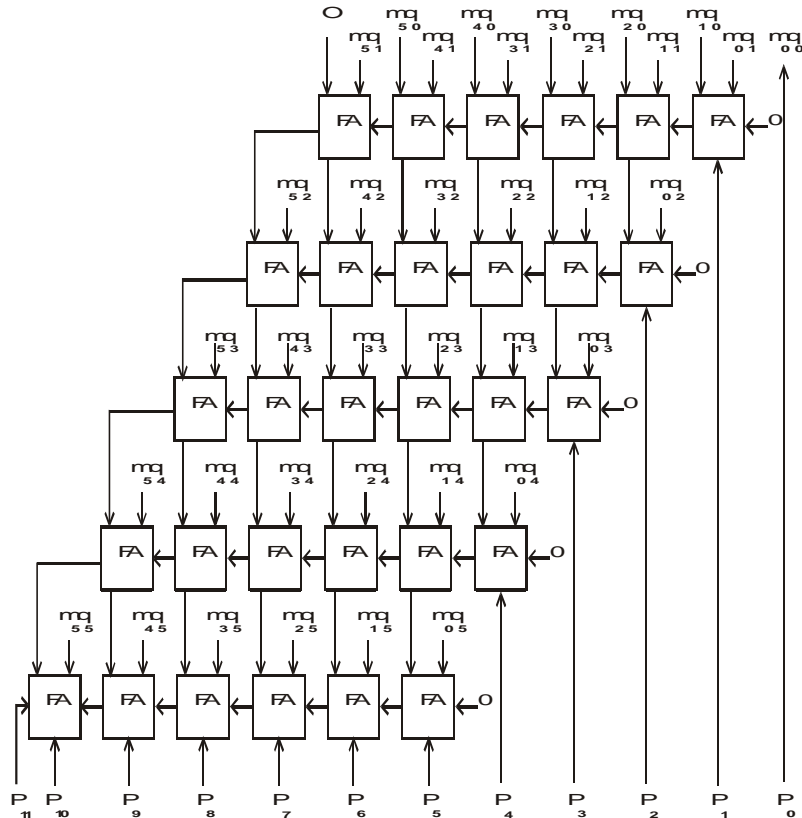
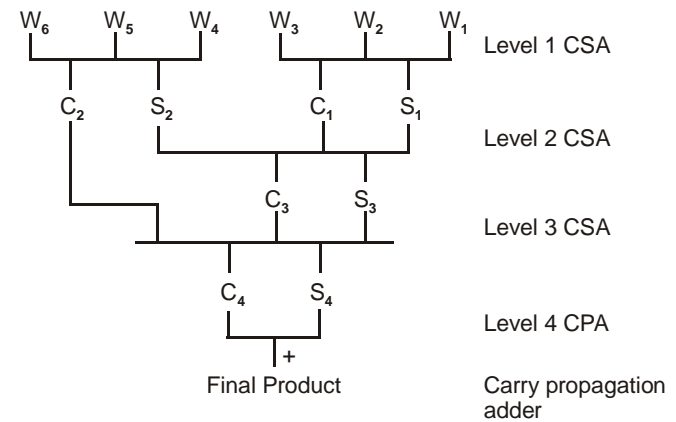


Fig. Using Ripple carry addition for 6×6 bits multiplication

Fig. 2: Using Carry Save Addition



Example 3

1 0 1 1 0 1	M
1 1 1 1 1 1	×
1 0 1 1 0 1	W1
1 0 1 1 0 1	W2
1 0 1 1 0 1	W3
1 1 0 0 0 0 1 1	S1
0 0 1 1 1 1 0 0	C1
1 0 1 1 0 1	W4
1 0 1 1 0 1	W5
1 0 1 1 0 1	W6
1 1 0 0 0 0 1 1	S2
0 0 1 1 1 1 0 0	C2
1 1 0 0 0 0 1 1	S1
0 0 1 1 1 1 0 0	C1
1 1 0 0 0 0 1 1	S2
1 1 0 1 0 1 0 0 0 0 1 1	S3
0 0 0 0 1 0 1 1 0 0 0	C3
0 0 1 1 1 1 0 0	C2
0 1 0 1 1 1 0 1 0 0 1 1	S4
0 1 0 1 0 1 0 0 0 0 0	C4
1 0 1 1 0 0 0 1 0 0 1 1	Product

Delay through the carry save array is somewhat less than delay through the ripple carry array. It is because the S and C vector outputs from each row are produced in parallel in one full adder delay.

Steps for addition of summands in the multiplication of longer operands

1. Group the summands into threes and perform carry save addition to produce S and C vectors in one full adder delay.
2. Group all the S and C vectors into threes and perform carry save addition to generate further set of S and C vectors in one more full adder delay.
3. Do the step 2 until there are only two vectors remaining.
4. Perform ripple carry addition or carry lookahead addition to produce the desired product.

Delays in carry save addition

For 6×6 array multiplier (Eg. given above)

1 gate delay to select the summands based on multiplier bits
 +
 6 gate delays (two gate delays for each CSA level) (3 CSA levels)
 +
 8 gate delays (for carry lookahead addition Final addt of C&S)
15 gate delay

For the same 6×6 array multiplication, the delay when ripple carry addition is used is

$$6(n-1)-1 = 6(6-1) = 6(5)-1$$

$$= 29 \text{ gate delay}$$

Thus carry save addition halves the delay when compared to ordinary array multiplier that uses ripple carry addition.

Note

No of CSA levels needed to
 reduce k summands to 2 vectors $= 1.7 \log_2 k - 1.7$

But if bit pair recoding of the multiplier is done instead of n summands for $n \times n$ multiplication, only $n/2$ summands are produced. This reduces the number of CSA levels required $= 1.7 \log_2 k - 3.4$.

Note

1. Bit pair recoding of the multiplier \rightarrow reduces the number of summands by a factor of 2
2. Carry save adder \rightarrow adds the summands and produces only Sum and Carry (2)
3. Carry lookahead adder \rightarrow used to add the final sum and carry.

3.3.10 Division

Division of two-fixed point binary number in signed magnitude representation is done by a process of compare, shift and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1.

Consider an example the divisor Y consists of five bits and the dividend X of 10 bits. The five MSB bits of the dividend are compared with the divisor. Since the five bit number is smaller than Y we try again by taking 6 MSB bits of X and compare this number with Y. The 6 bit number is greater than Y, so we place a 1 for the quotient bit in the 6th position above the dividend. The difference is called a partial remainder, because the division could have stopped here to obtain a quotient of 1 and the remainder is equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction

is needed. The divisor is shifted once to the right in any case. The result gives both a quotient and remainder.

Division Overflow

The division operation may result in a quotient with a overflow, i.e. when the operation is implemented with hardware this is because the length of the register is finite and will not hold the number that exceeds the standard length. The divide overflow condition must be avoided in normal computer operation because the entire quotient will be too long to transfer into memory unit that has words of standard length, that is, same as the length of register.

When the dividend is twice as long as the divisor the condition to overflow can be stated as follows. A divide-overflow condition occurs if the high order half bits of dividend constitute a number greater than or equal to the divisor. Another fact associated with division is the fact that the division with Zero is avoided. The divide overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor, which is equal to zero. Overflow condition is usually detected when the special FF is set. We will call it a divide-overflow FF and label it DVF.

- 1) The occurrence of divide overflow can be handled in a variety of ways. In some computer it is the responsibility of the programmer to check if DVF is set after each (division) divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow.
- 2) In some olden computers, the occurrence of divide overflow stopped the computer and this condition was referred to as divide stop. Stopping the operation of computer is not recommended because it is time consuming. The procedure in most computers is to provide an interrupt required, when DVF is set. The instruction causes the computer to suspend the current program and branch to a service routine to take a corrective measure.

The best way to avoid a divide overflow is to use floating-point data.

(i) Example 1

	1 1 0 1 0	Divisor B = 10001
10001) 0 1 1 1 0 0 0 0 0 0	Quotient Q
		Dividend A
	- 1 0 0 0 1	5 bits of A < B, take 6 bits of A
	- 0 1 0 1 1 0	→ shift right B & Sub; enter 1 in Q 6 bit position
	- - 1 0 0 0 1	
	- - 0 0 1 0 1 0	7 bits of remainder > B Shr B & subtract, enter 1 in Q
	- - - 0 0 1 0 1 0	Remainder < B, enter 0 in Q.
	- - - - 1 0 0 0 1	
	- - - - 0 0 0 1 1 0	Remainder > B Shr B and sub enter 1 in Q
	- - - - - 0 0 0 1 1	Remainder < B; enter 0 in Q
Quotient	→ 11010	Shr B, Final remainder
Remainder	→ 00011	

(ii) Example 2

	1 1 1 1 0 1	Divisor B = 0101
0101) 1 0 0 1 1 0 0 1 0	Dividend A
	0 1 0 1	Quotient Q
		4 bits of A > B; Sub enter 1 in Q Shr B
	0 1 0 0 1	5 bits of A > B sub enter 1 in Q
	- 0 1 0 1	6 bits of A > B; sub Shr B Enter 1 in Q
	- 0 1 0 0 0	7 bits of A > B; sub Shr B Enter 1 in Q
	- - 0 1 0 1	8 bits of A < B;
	- - 0 0 1 1 0	Enter 0 in Q
	- - - 0 1 0 1	Shr B
	- - - - 0 0 0 1 1	A > B, sub enter 1 in Q
	- - - - - 0 0 1 1 0	
	- - - - - 0 1 0 1	
	0 0 0 1	

Quotient 1 1 1 1 0 1	Remainder 0 0 0 1
----------------------	-------------------

3.3.10.1 Integer Division (Restoring Method)

When the division is implemented in a digital computer, it is convenient to change the process slightly instead of shifting the divisor to the right the dividend or partial remainder, is shifted to the left thus may 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 Q_n and previous value of E is lost.

Ex:- The divisor is stored in the Y register and the double length 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 $X \geq Y$. A quotient bit 1 is inserted into the Q_n 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 Q_n 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.

Method 1 : Restoring Division Method

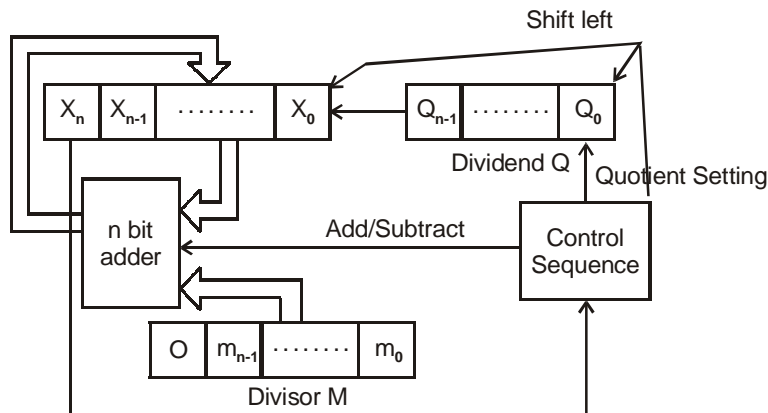


Fig. Register Configuration

H/W Algorithm

The dividend is in X and Q and the divisor in Y. The sign of the result is transferred to Q_s to be a part of quotient. A constant is set in to sequence counter sc to specify the no of bits in a quotient. The operands are transferred to register from a memory unit that has words to n-bits. Since 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 option is terminated permanently. If $X < Y$, no divide overflow occurs so the value of the dividend is restored by adding Y to X.

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 $EX > B$ 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 Q_n for the quotient bit. Since register X is missing the high order bit of the dividend,(which is in E) its value is $EX \cdot 2^{n-1}$. Adding to this value the 2's complement of Y result is

$$(EX \cdot 2^{n-1}) + (2^{n-1} - Y) = EX - Y$$

The carry from this addition is not transferred to E if we want E to remain a 1.

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 Q_n 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 Q_n (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 Q_s and sign of the remainder in X_s is same as the original sign of the dividend.

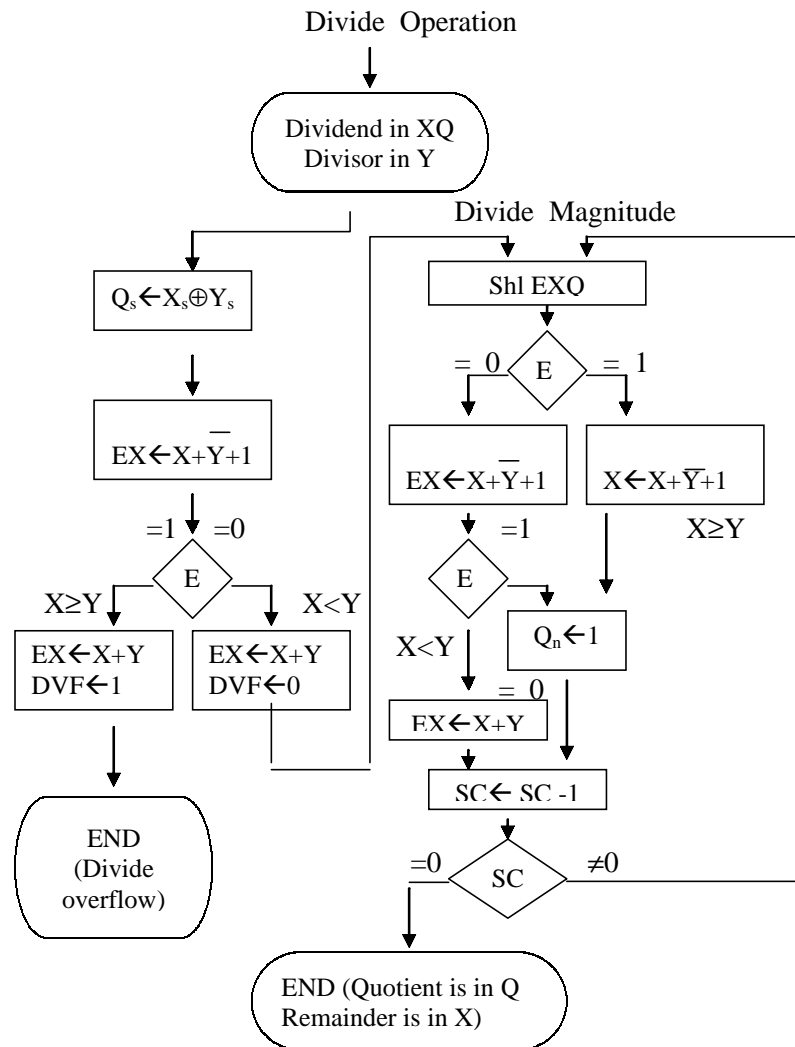


Fig: - Flow chart for divide operation (Restoring Method)

In this configuration,

Register M holds the divisor

Register Q holds the dividend at the start of operation.

Register X is initially set to 0.

After division Register Q contains quotient.

Register X contains Remainder

Algorithm

1. Shift X and Q left one binary position
2. Subtract M from X and place the answer back in X
3. If the sign of X is 1, set q_0 to 0 and add M back to X (restore X); otherwise set q_0 to 1.

$$11 \overline{) 1000} \Rightarrow \frac{8}{3} = 2.2 \quad \text{Quotient} = 2 \\ \text{Remainder} = 2$$

Eg.:

	X	Q	
Initial Cond	0 0 0 0	1 0 0 0	
M	0 0 0 1 1		
Shift X & Q	0 0 0 0 1	0 0 0 □	Cycle I
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	

COMPUTER ARCHITECTURE

Shift X & Q	0 0 1 0 0	0 0 0 □	} Cycle III
Sub M	<u>1 1 1 0 1</u>		
	0 0 0 0 1		
Set $q_0=1$		0 0 0 1	
Shift X & Q	0 0 0 1 0	0 0 1 □	} Cycle IV
Sub M	<u>1 1 1 0 1</u>		
	1 1 1 1 1		
Set $q_0=0$			
Restore M	<u>0 0 0 1 1</u>		
	0 0 0 1 0	0 0 1 0	
	Remainder	Quotient	

Method 2 : 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.
- In the restoring method
If A is positive, we shift A & Q left and sub M ie. we perform $2A - M$
If A is negative, we restore A by performing $A+M$ and then we shift it left and subtract M.

Non Restoring algorithm

Step 1 : Do the following n times

- 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.
- If the sign of A is 0, set q_0 to 1; otherwise, so q_0 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.

Arithmetic Unit

Eg.: $8/3 = 2.2$	Quotient	= 2	
	Remainder	= 2	
Initial Cond	A	Q	
	0 0 0 0 0	1 0 0 0	} Cycle I
M	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract M	<u>1 1 1 0 1</u>		
Set $q_0=0$	1 1 1 1 0		
		0 0 0 0	
Shift	1 1 1 0 0	0 0 0 □	} Cycle II
Add	<u>0 0 0 1 1</u>		
	1 1 1 1 1		
Set q_0		0 0 0 0	
Shift	1 1 1 1 0	0 0 0 □	} Cycle III
Add	<u>0 0 0 1 1</u>		
	0 0 0 0 1		
Set $q_0=1$		0 0 0 1	
Shift	0 0 0 1 0	0 0 1 □	} Cycle IV
Sub M	<u>1 1 1 0 1</u>		
	1 1 1 1 1	0 0 1 0	
		Quotient	
Add	1 1 1 1 1		
Restore M	<u>0 0 0 1 1</u>		
	0 0 0 1 0		
	Remainder		

3.3.11 Floating Point Numbers : Representation

The floating point numbers contains the binary point variable in its position and hence these numbers are called floating point numbers. Because the position of the binary point number is variable, it must be given explicitly in the floating point representation.

If the decimal point is placed to the right of the first significant digit, (non zero) the number is said to be normalized.

A floating point number consists of sign, mantissa and an exponent.

3.3.11.1 Standard for floating point numbers

The IEEE standard describes the floating point representations and the way in which the four basic arithmetic operations are to be performed on these floating point operands.

There are two types of representations for floating point numbers.

1. Single Precision
2. Double Precision

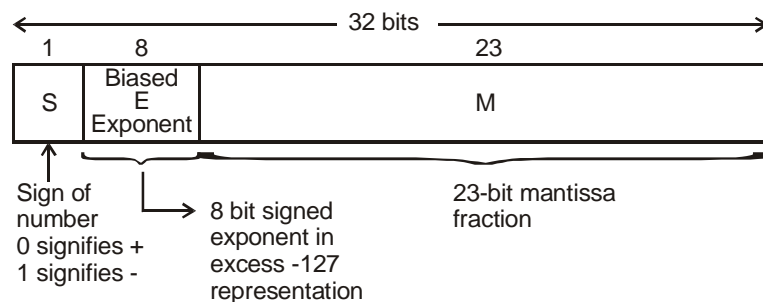
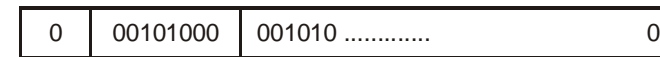


Fig. Single precision (32 bit).

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

Eg.:



$$\text{Value represented} = +1.001010 \dots \times 2^{-87}$$

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

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 53 bit mantissa provides a precision equivalent to about 16 decimal digits.

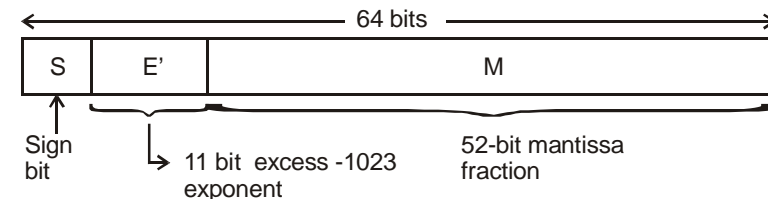


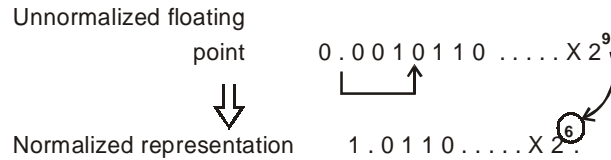
Fig. Double Precision.

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

If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent.

COMPUTER ARCHITECTURE

Eg



Since the scale factor is in the form 2^i , shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent respectively.

Special Values

When $E'=0$, the mantissa M is zero.

$E'=255$ and $M=0$, the value ∞ is represented where ∞ is the result of dividing a normal number by zero.

Similarly

When $E'=0$, and $M \neq 0$, denormal numbers are represented.

Eg. $+0.M \times 2^{-126} \rightarrow$ denormal numbers are allowed to represent very small numbers.

When $E'=255$ and $M \neq 0$, the value represented is Not-a-Number (NaN) \rightarrow result of invalid operation $0/0$, $\sqrt{-1}$.

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}

Arithmetic Unit

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}

- For exponent values in the range of 1 through 254 for single format and 1 through 2046 for double format, normalized non-zero floating point numbers are represented.
- The exponent is biased, so that the range of exponents is -126 thro +127 for single format and -1022 through +1023 for double format.
- A normalized number requires a 1 bit to the left of the binary point this bit is implied giving an effective 24-bit or 53-bit significant.

	Single Precision (32 bits)				Double Precision 64 bits)			
	Sign	Biased Exponent	Fraction	Value	Sign	Biased Exponent	Fraction	Value
Positive Zero	0	0	0	0	0	0	0	0
Negative zero	1	0	0	-0	1	0	0	-0
Plus Infinity	0	255(all 1's)	0	∞	0	2047 (all 1's)	0	∞
Minus Infinity	1	255(all 1's)	0	$-\infty$	1	2047 (all 1's)	0	$-\infty$
Positive Normalized non-zero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
Negative Normalized non-zero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
Positive denormalized	0	0	$f \neq 0$	$2^{-126}(0.f)$	0	0	$f \neq 0$	$2^{-1022}(0.f)$
Negative denormalized	1	0	$f \neq 0$	$-2^{-126}(0.f)$	1	0	$f \neq 0$	$-2^{-1022}(0.f)$

Exceptions

If a number has the exponent value less than -126 we say underflow has occurred.

If a number has the exponent value greater than $+127$, we say overflow has occurred.

Such conditions are called exceptions, Exception flag is set if underflow, overflow, divide by zero, in exact or invalid operations occur.

Overflow

When normalized mantissa are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent.

Underflow

When two numbers are subtracted, the result may contain significant zeros.

Eg.

$$\begin{array}{r} 0.56780 \times 10^5 \\ -0.56430 \times 10^5 \\ \hline 0.00350 \times 10^5 \end{array}$$

A floating point number that has a 0 in the most significant position of the mantissa is said to have an underflow.

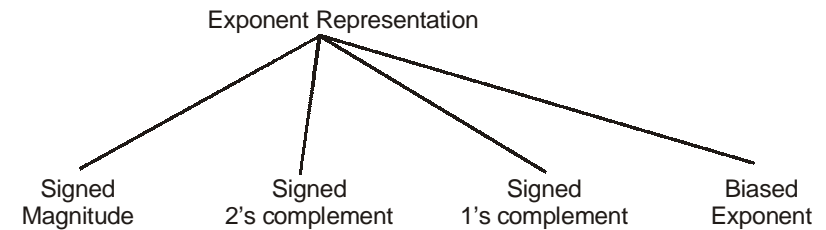
In most computers, a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating point multiplication and division do not require an alignment of the mantissa.

Multiplication \rightarrow Multiply two mantisas and add exponents.

Division \rightarrow Divide the mantisas and subtract exponents

Operations with mantisas are same as in fixed point representation. Operation performed with exponents are compare and increment (for aligning the mantisas) add and sub (for multiplication and division) and decrement (to normalize the result)

**Biased Exponent**

In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating point number is formed, so that internally all exponents are positive. Typically, the bias equals $2^{k-1}-1$ where k is the number of bits in the binary exponent.

Eg. Excess -127 representation

Advantages

- Contains only positive numbers
- simple to compare their relative magnitudes without being concerned with their sign.

3.3.11.2 Arithmetic operations on floating point numbers

In floating point operations, it is assumed that each floating point number has a mantissa in signed magnitude representation and biased exponent.

If the exponents differ, the mantissas of floating point numbers must be shifted with respect to each other before they are added and subtracted.

Eg.

$$2.3742 \times 10^2 + 5.6232 \times 10^4$$

Rewrite 2.3742×10^2 as 0.023742×10^4 and then add

Eg.

$$\begin{array}{r} 0.023742 \times 10^4 \\ 5.623200 \times 10^4 \\ \hline 5.646942 \times 10^4 \end{array}$$

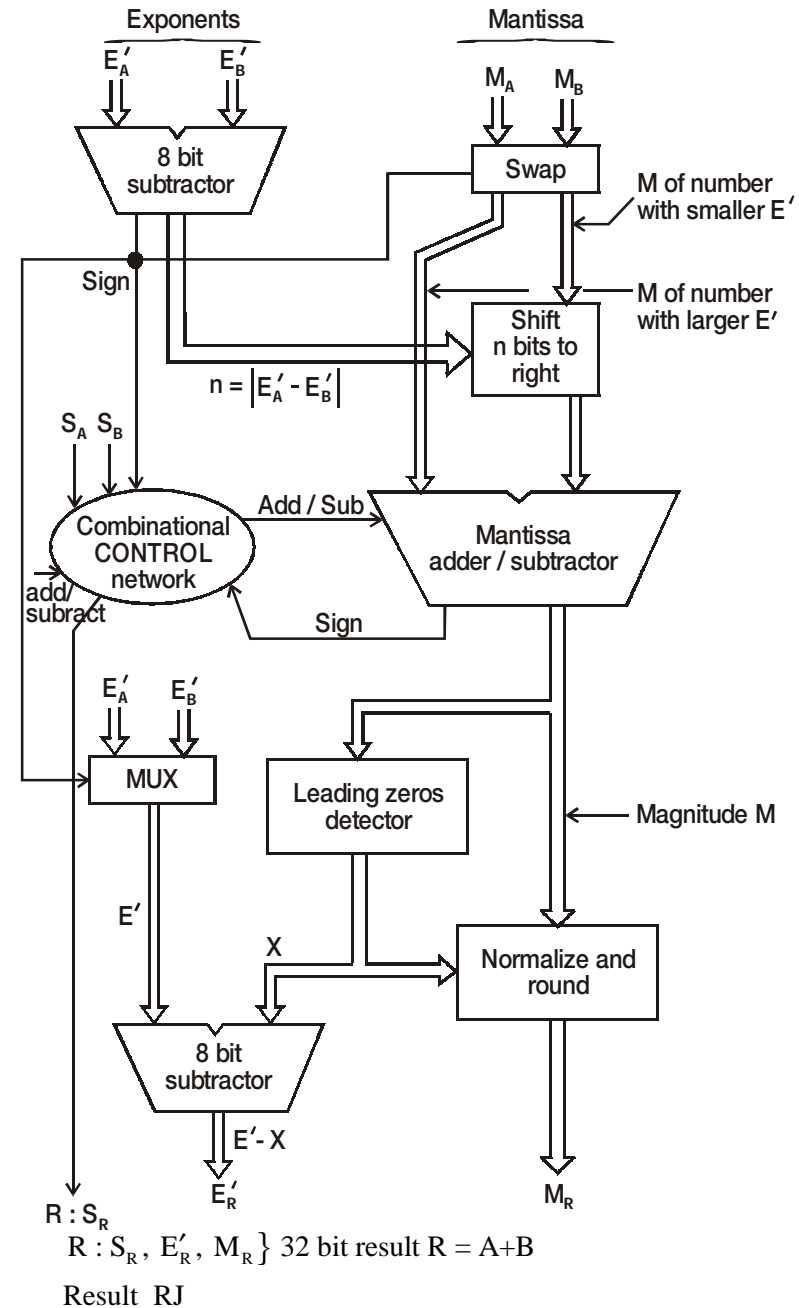
Addition / Subtraction

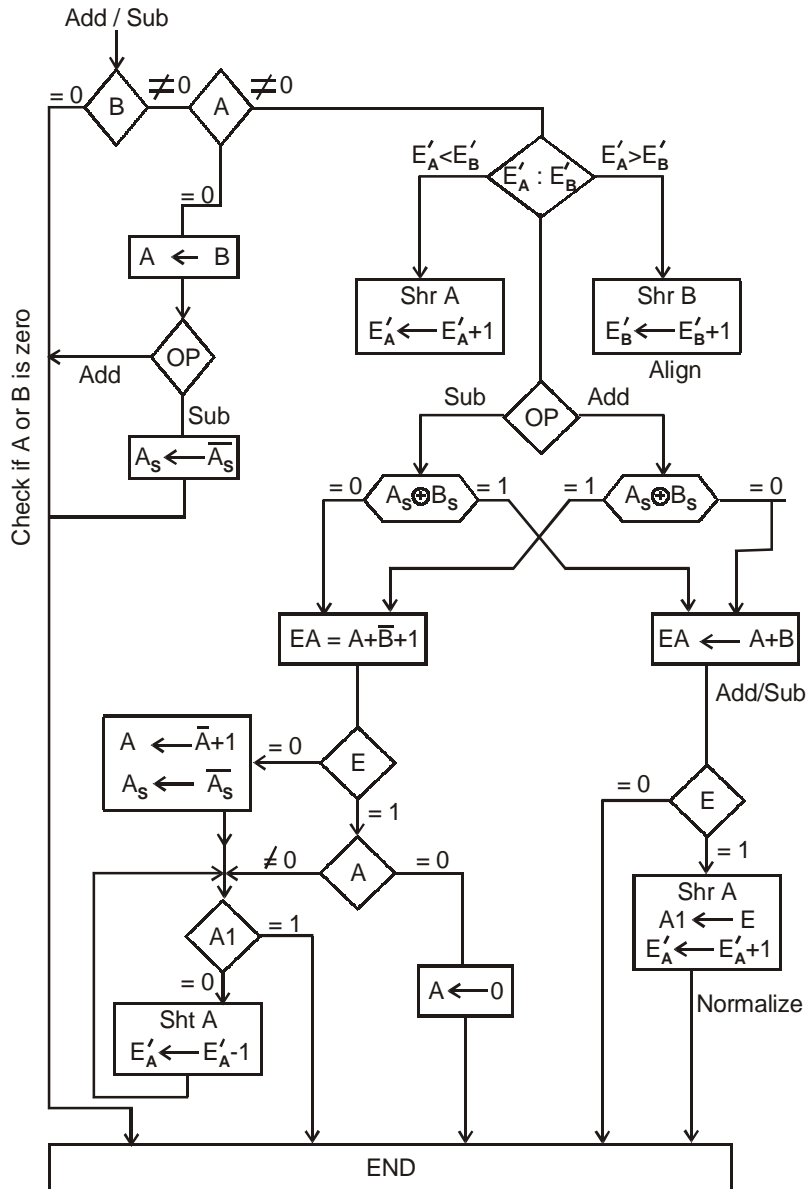
1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition / subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value if necessary

Arithmetic Operations on Floating Point Numbers

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left. \begin{aligned} X + Y &= (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y &= (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{aligned} \right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S} \right) \times B^{X_E - Y_E}$

32 bit operands $\left\{ \begin{array}{l} A : S_A, E'_A, A \\ B : S_B, E'_B, B \end{array} \right\}$





The flow chart depicts the generalized approach for add/sub of floating point numbers.

1. Check for zeros
2. Align the mantissas
3. Add or sub the mantissas
4. Normalize the result

Eg.

$$0.0523 \times 10^3 +$$

$$0.2751 \times 10^4$$

$$1. E'_A = 3$$

$$E'_B = 4$$

2. Shift right A

$$0.00523 \times 10^4$$

$$0.2751 \times 10^4$$

3. Since the exponents are equal

add

$$0.00523 \times 10^4$$

$$0.27510 \times 10^4$$

$$0.28033 \times 10^4$$

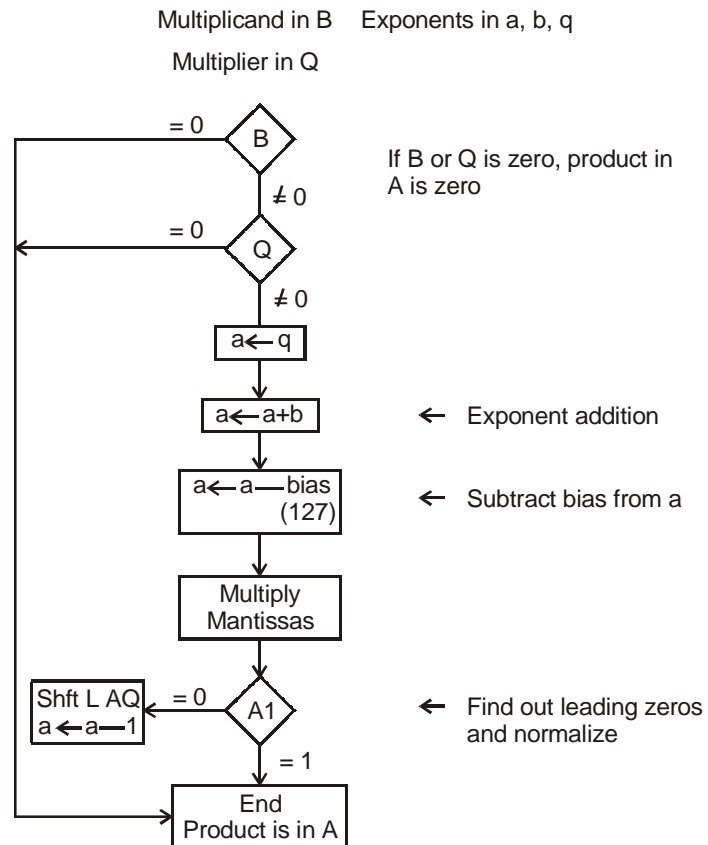
Implementation Steps

1. In the figure, 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 and thereby to shift the mantissa of the smaller number.

- The mantissas are added / subtracted. The sign of the result is determined by combinatorial control network. if $E'_A > E'_B$ then sign is positive or if $E'_A < E'_B$ then sign is negative.
- The result is normalized by truncating the leading zeros and by subtracting E' by X , the number of leading zeros.

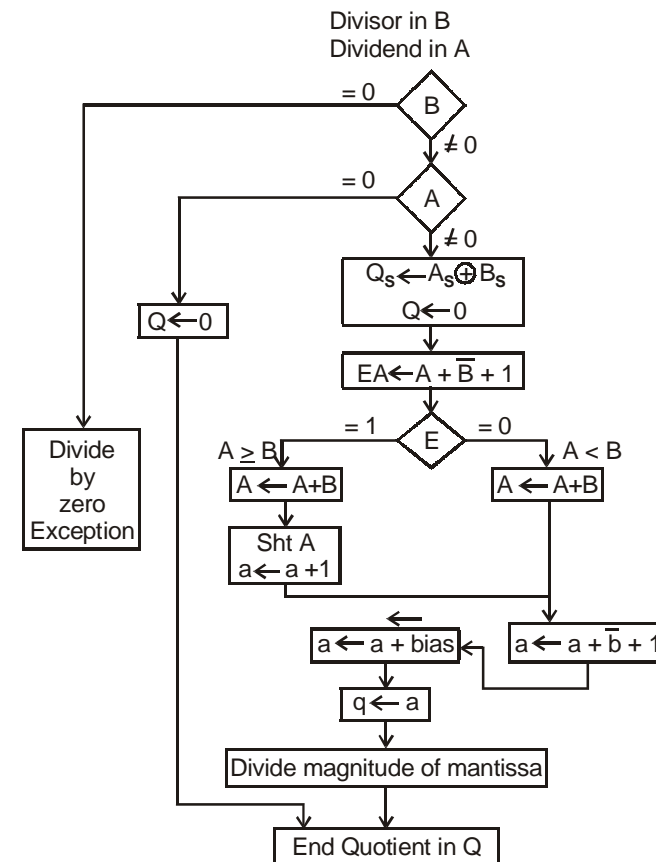
Multiplication

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value of necessary.



Division

- Subtract the exponents and add 127.
- Divide the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.



3.3.11.3 Guard bits and Truncation

In 32 bit single precision floating point representation the mantissa bits are limited to 24 bits including implicit leading 1. But some operations may result in extra bits called guard bits and these bits should be retained during the intermediate steps to increase the accuracy in final results.

Similarly, allowing guard bits during intermediate steps results in extended mantissa. Thus this extended mantissa should be truncated to 24 bits while generating final results.

There are several ways to truncate

- (i) Chopping
- (ii) Von-Neumann rounding
- (iii) rounding

(i) Chopping

Chopping is the simplest way to do truncation. i.e. Remove the guard bits and make no changes in the retained bits.

Eg.

$$0.b_{-1} b_{-2} b_{-3} 000$$

$$0.b_{-1} b_{-2} b_{-3} 111$$

are truncated to $0.b_{-1} b_{-2} b_{-3}$.

Error in chopping ranges from 0 to almost 1 and results in biased approximation of b_{-3} position.

(ii) Von-Neumann Rounding

In this method, if the bits to be removed are all 0's, they are simply dropped, with no changes to the retained bits.

If any of the bits to be removed are 1, the least significant bit of the retained bit is set to 1.

Eg.

$$\begin{array}{lcl} 0.b_{-1} b_{-2} b_{-3} 000 & \xrightarrow{\text{truncated to}} & 0.b_{-1} b_{-2} b_{-3} \\ 0.b_{-1} b_{-2} b_{-3} 100 & \xrightarrow{\hspace{1cm}} & 0.b_{-1} b_{-2} \frac{1}{\text{LSB to 1}} \end{array}$$

Error in this technique is larger than chopping and the approximation is unbiased and hence results in high probability of accuracy.

(iii) Rounding Procedure

Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. 1.A is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed.

Eg

$$0.b_{-1} b_{-2} b_{-3} 1..... \text{ is rounded to}$$

$$0.b_{-1} b_{-2} b_{-3} (+)$$

$$0.001$$

$$0.011100 \text{ is rounded to}$$

$$0.011 (+)$$

$$0.001$$

$$0.100$$

$$0.b_{-1} b_{-2} b_{-3} 0 \text{ is rounded to } 0.b_{-1} b_{-2} b_{-3}.$$

There can be a tie situation, that is the bits to be removed is halfway between the two closest truncated representations.

Eg.

$$\text{The value } 0.b_{-1} b_{-2} 0100 \text{ is truncated to}$$

$$\begin{array}{l} \text{and } 0.b_{-1} b_{-2} 0 \text{ } 0.b_{-1} b_{-2} 1100 \text{ is truncated} \\ \text{to } + 0.b_{-1} b_{-2} 0.001 \end{array}$$


To break the tie in an unbiased way, one possibility is to choose the retained bits to be the nearest even number.

- The error range is approximately $-1/2$ to $+1/2$ in LSB position of the retained bits.
- This rounding technique is the default mode for truncation specified in the IEEE floating point standard.


PROBLEMS

- Consider the binary numbers in the following addition and subtraction problems to be signed, 6 bit values in the 2's complement representation. Perform the operations indicated, specify whether or not arithmetic overflow occurs


(i)
$$\begin{array}{r} 010110 \\ 001001 \\ \hline 011111 \end{array} \Rightarrow \begin{array}{r} +22 \\ +9 \\ \hline +31 \end{array}$$

$$\begin{array}{c} C_{n-1} = 0 \\ C_n = 0 \end{array}$$
  Overflow does not occur


(ii)
$$\begin{array}{r} 110111 \\ 111001 \\ 1110000 \\ \hline \end{array} \Rightarrow \begin{array}{r} -9 \\ -7 \\ \hline -16 \end{array}$$

$$\begin{array}{c} C_{n-1} = 1 \\ C_n = 1 \end{array}$$
  Overflow does not occur

(iii)
$$\begin{array}{r} 000111 \\ (-)111000 \\ \hline \end{array} \Rightarrow \begin{array}{r} 000111 \\ 001000 \\ \hline 001111 \end{array} \Rightarrow \begin{array}{r} +7 \\ -8 \\ \hline -15 \end{array}$$

$$\begin{array}{c} C_{in-1} = 0 \\ C_n = 0 \end{array}$$
  Overflow does not occur

(iv)
$$\begin{array}{r} 011010 \\ (-)100010 \\ \hline \end{array} \Rightarrow \begin{array}{r} 011010 \\ 011110 \\ 111000 \\ \hline \end{array} \Rightarrow \begin{array}{r} +26 \\ -30 \\ \hline +56 \end{array}$$

$$\begin{array}{c} C_{n-1} = 1 \\ C_n = 0 \end{array}$$
  Overflow has occurred

- Multiply the following pairs of signed 2's comp numbers using the Booth algorithm.

a) $A = 110011$ & $B = 101100$

$= -13 \times -19 = +260$

Multiplicand BR = 110011

$\overline{BR} + 1 = 001100 + 1 = 001101$

Q_n	Q_{n+1}	Initial	AC	QR	Q_{n+1}	SC
			000000	101100	0	110
0	0	Ashr AQ	000000	010110	0	101
0	0	Ashr AQ	000000	001011	0	100
1	0	Sub BR	001101			
			001101			
		Ashr AQ	000110	100101	1	011
1	1	Ashr AQ	000011	010010	1	010
0	1	Add BR	110011			
			110110			
		Ashr AQ	111011	001001	0	001
1	0	Sub BR	001101			
			001000			
		Ashr AQ	000100	000100	1	000
<hr/>						
Final Product						
260						

COMPUTER ARCHITECTURE

3. Using bit-pair recoding, multiply the following

$$A = 110101, \quad B = 011011$$

$$(-11) \times (27)$$

Bit-pair recoding of the multiplier is

$$011011 \quad \boxed{0} \text{ implied zero}$$

$$+10-1+10-1$$

2's comp of the multiplicand (110101)

$$1's \text{ complement} = 001010+1$$

$$+1 = \underline{\quad 1 \quad}$$

$$2's \text{ complement} = \underline{001011}$$

Multiply:

$$\begin{array}{r} 110101 \times \\ +10-1+10-1 \\ \hline 00000001011 \\ 00000000000 \\ 111110101 \\ 00001011 \\ 0000000 \\ 110101 \\ \hline 111011010111 \end{array}$$

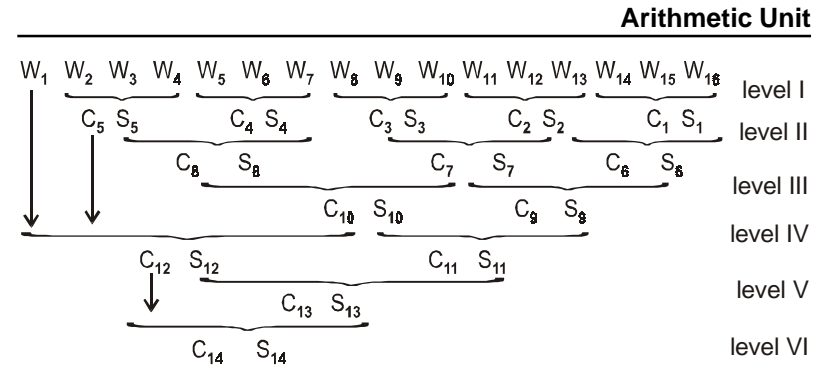
← Result is in 2's comp representing negative pdt

← take 2's comp & put -ve sign.

$$= -297$$

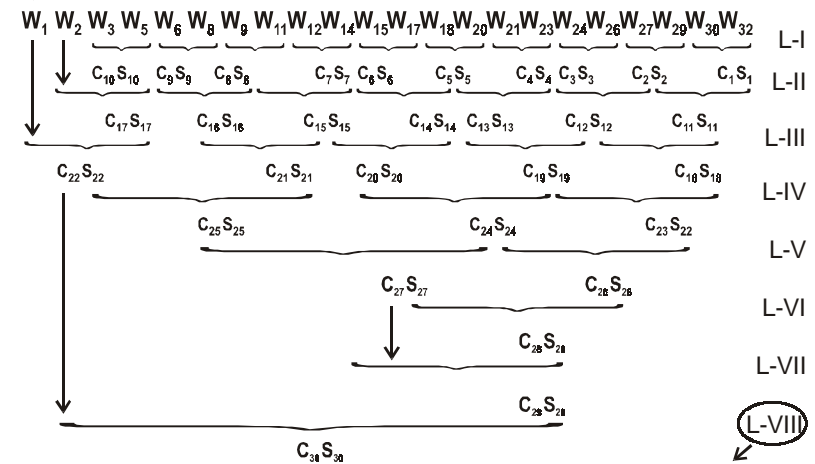
4. a) How many CSA levels are needed to reduce 16 summands to 2.

b) Draw the pattern for reducing 32 summands to 2 to prove that the claim of 8 levels is correct.



Totally 6 CSA levels are required to reduce 16 summands to 2.

b)



5. Eg. Floating Point Arithmetic Operations

Level 8

$$X = 0.3 \times 10^2 = 30$$

$$Y = 0.2 \times 10^3 = 200$$

$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$

to bring the exponent
equal and then add

$$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$

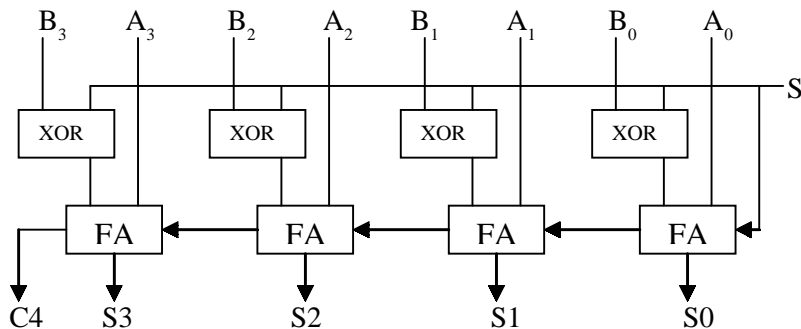
$$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$$

$$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$$

COMPUTER ARCHITECTURE

- 6) Design a 4 bit arithmetic circuit with one selection variable S and two 4-bit data inputs A & B. When S=0 the circuit performs addition A+B. When S=1, the circuit performs A-B (2's complement form).

Solution:

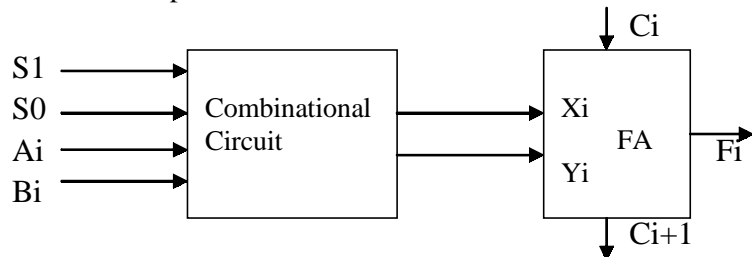


- 7) An arithmetic circuit has 2 selection variables S1 and S0. The arithmetic operation available in the unit are listed below. Determine the circuit that must be incorporated with the full adder in each stage of the arithmetic circuit.

S1	S0	Ci=0	Ci=1
0	0	$F = A+B$	$F = A+B+1$
0	1	$F = A$	$F = A+1$
1	0	$F = B$	$F = B + 1$
1	1	$F = A+B$	$F = A+B + 1$

Solution:

Let the input to the full adder circuit can be X_i & Y_i



Arithmetic Unit

$$X_i = A_i S_1 + A_i S_0$$

$$Y_i = B_i S_1 S_0 + B_i S_1$$

S1	S0	Ai	Bi	X_i	Y_i	Function
0	0	0	0	0	0	$X_i = A_i$ $Y_i = B_i$
0	0	0	1	0	1	
0	0	1	0	1	0	
0	0	1	1	1	1	
0	1	0	0	0	0	$X_i = A_i$ $Y_i = 0$
0	1	0	1	0	0	
0	1	1	0	1	0	
0	1	1	1	1	0	
1	0	0	0	0	1	$X_i = 0$ $Y_i = \overline{B_i}$
1	0	0	1	0	0	
1	0	1	0	0	1	
1	0	1	1	0	0	
1	1	0	0	0	1	$X_i = A_i$ $Y_i = B_i$
1	1	0	0	0	0	
1	1	0	0	1	1	
1	1	0	0	1	0	

- 8) Prove that the multiplication of two n-digit number in base r gives a product no more than 2n digits in length. Show that this statement implies that no overflow can occur in multiplication operation.

Solution:

Maximum value of number is $r^n - 1$. It's necessary to show that maximum product is less than or equal to $r^{2n} - 1$.

Maximum product is

$$(r^n - 1)(r^n - 1) = r^{2n} - 2r^n + 1 \leq r^{2n} - 1$$

$$\Rightarrow 2 \leq 2r^n \quad \text{or} \quad 1 \leq r^n$$

This is always true since $r \geq 2$ and $n \geq 1$

COMPUTER ARCHITECTURE

- 9) Derive an algorithm for evaluating the SQRT of a binary fixed point number.

Solution:

The algorithm for SQRT is similar to division with a radicand being equivalent to the dividend and the test value being equivalent to this divisor. Let A be the radicand, Q be the SQRT and R be the remainder such that $Q^2 + R = A$ (or) $\sqrt{A} = Q$ and a remainder.

Algorithm

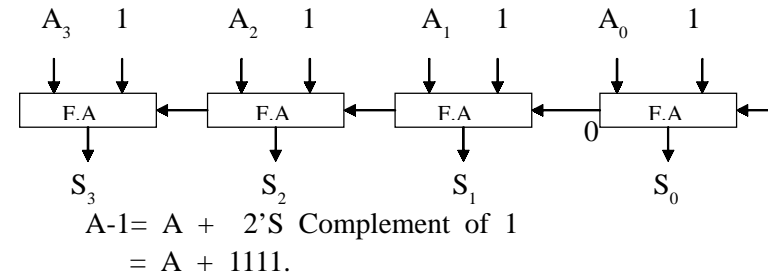
- For K bits in A (K even), Q will have K/2 bits
 $Q = q_1 q_2 q_3 \dots q_{k/2}$
- The 1st test value is 01
 2nd test value is $0q_1 01$
 3rd test value is $00q_1 q_2 01$
 4th test value is $000q_1 q_2 q_3 01$ etc
- Mark the bits of A in groups of two starting from left.
- The procedure is similar to the division restoring method as show below.

$$\begin{array}{r}
 1101 \\
 \sqrt{10101001} \\
 \underline{01} \\
 01 \\
 0110 \\
 \underline{0101} \\
 0001 \\
 000110 \\
 \underline{001101} \\
 -ve \\
 000110 \\
 00011001 \\
 \underline{00011001} \\
 00000
 \end{array}$$

Arithmetic Unit

- 10) Design a 4 bit combinational circuit decremter using 4 FA's

Solution



- 11) Derive an algorithm in flowchart form for non-restoring method of fixed point binary division.

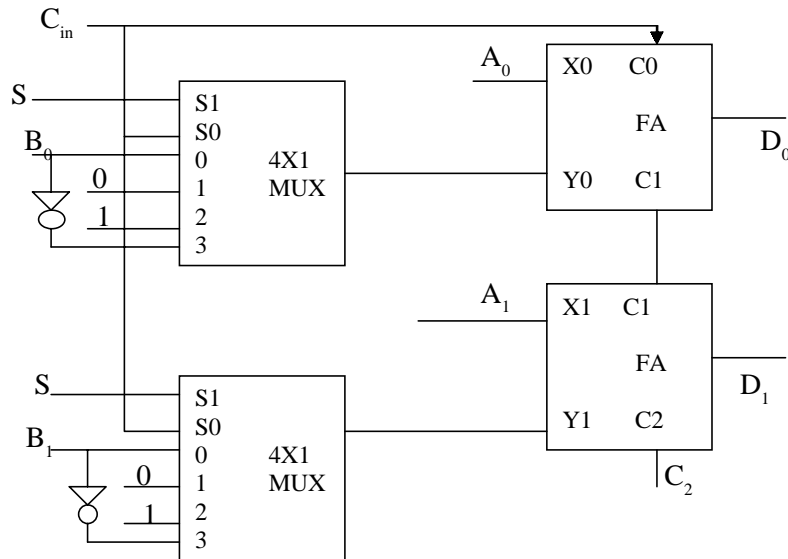
Refer

- 12) 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.

Solution:

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

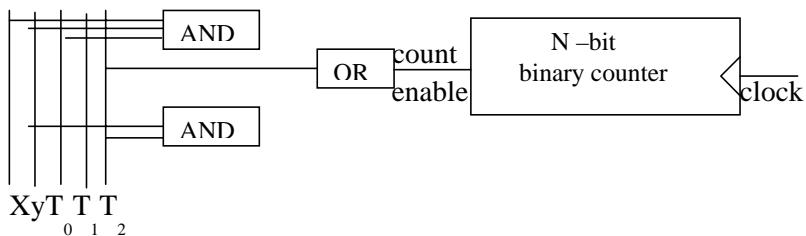
S	Cin	X	Y
0	0	A	B (A+B)
0	1	A	0 (A+1)
1	0	A	1(A-1)
1	1	A	B (A-B)



- 13) Show the hardware that implements the following statement. Include the logic gates for the control function and the Block diagram for the binary counter with the count enable input.

Solution:

$$xy T_0 + T_1 + \bar{y} T_2 : AR \leftarrow AR + 1$$



- 14) What is the principle of BOOTH's multiplication algorithm

Solution

Skipping over of 1's. Recording of multiplier such that when ever a series of 1's occur the multiplication process corresponding to it can be replaced by an addition of two numbers.

- 15) Derive an algorithm in flowchart form for the comparison of two signed binary numbers when negative numbers are in signed 2's complement representation.

By means of a subtraction operation with the signed 2's complement number.

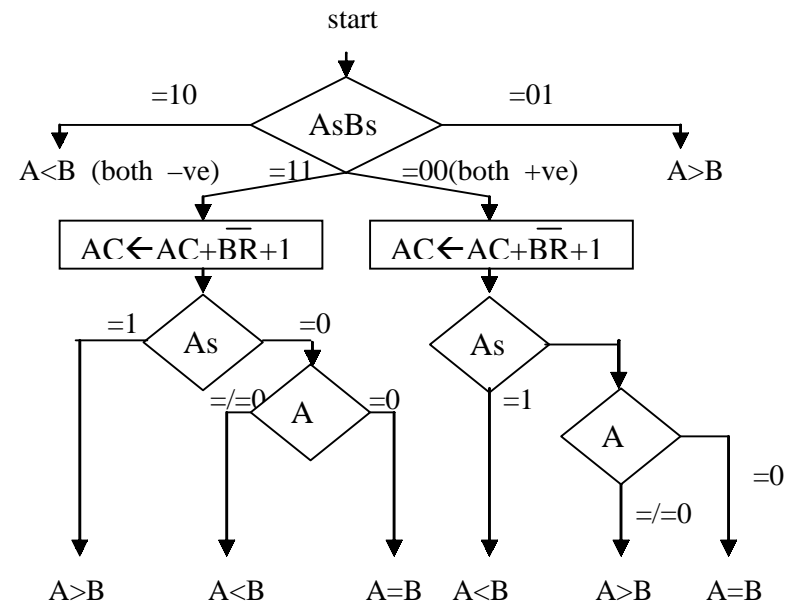
Solution:

$$AC = A_s A_1 A_2 A_3 \dots A_n$$

$$BR = B_s B_1 B_2 B_3 \dots B_n$$

If signs are unlike \rightarrow the one with 0 (plus) is larger.

If signs are alike \rightarrow both numbers are either +ve or -ve.



QUESTIONS & ANSWERS**1) What are the types of ALU?**

- Combinational ALU
- Sequential ALU.

2) What is the disadvantage of combinational ALU

It is costly in hardware.

3) Give any 2' complement multiplier algorithm

- Robertson's algorithm
- Booth's algorithm.

4) What is spatial expansion in ALU

In this expansion connect K copies of the m-bit ALU in the manner of a ripple carry adder to form a single ALU capable of processing Km bit words directly.

5) What is temporal expansion in ALU

In this expansion use one copy of the m-bit ALU chip in the manner of a serial adder to perform an operation on Km-bit words in K consecutive steps. In each step the ALU processes a separate m-bit slice of each operand. This processing is also called multi cycle or multi precision processing.

6) When a ALU is said to be bit sliced

An ALU is said to be bit sliced if each component ALU concurrently process a separate "slice" of m bits from each Km -bit operand.

7) Give the advanced features of ALU

- Floating point arithmetic circuit.

- Pipelined circuit
- Co-processor.

8) What is a co-processor

A co-processor a separate instruction set processor that is closely coupled to CPU and whose instructions and registers are direct extensions of CPU.

9) What is a co-processor trap

Even if no coprocessor is present, Co-processor instructions can be included in CPU programs, because if the CPU knows that no CO-processor is present it can transfer program to a predetermined memory location where a software routine implementing the desired co-processor instruction is stored. This CPU generated interrupt is called a Co-processor trap.

10) Define Micro operation?

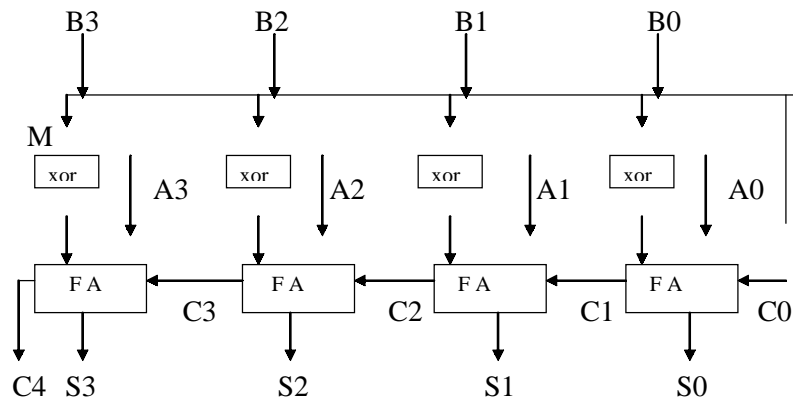
A micro operation is an elementary operation performed with data stored in the register.

11) What are the types of micro operations

There are four types of micro operations

- Arithmetic micro operations.→ perform arithmetic operations on the data stored in the registers.
- Logical micro operations → performs bit manipulation operations on the data stored in the registers.
- Register transfer micro operations → transfers binary information from one register to another register.
- Shift micro operations→ performs shift operations on the data stored in the registers.

- 12) Draw a logic circuit which performs both addition and subtraction.



When $M = 0$ The above circuit behaves like an adder.

$M = 1$ The above circuit behaves like a subtractor.

PART - B

- 1) Explain the 4-bit binary adder?
Refer Sec. 3.3.2
- 2) Explain the binary adder / subtractor with a neat block diagram
Refer Sec.
- 3) Explain the binary incrementer with a neat block diagram
Refer Sec.
- 4) Design a 4-bit binary arithmetic circuit
Refer Sec. 3.3.4
- 5) With a neat block diagram fixed point ALU
Refer Sec. 3.1
- 6) Explain the Bit-sliced ALU with a neat block diagram.
Refer Sec. 3.1

- 7) Explain briefly about arithmetic operations?
Refer Sec. 3.1
- 8) Write an algorithm for Addition & Subtraction?
Refer Sec. 3.3.5
- 9) Explain the Addition (Subtraction) Algorithm:
Refer Sec. 3.3.5
- 10) Explain Addition and subtraction with signed 2's complement data
Refer Sec. 3.3.5.1
- 11) Write an Multiplication Algorithm ?
Refer Sec. 3.3.7
- 12) Explain the multiplication algorithm using signed - Magnitude data
Refer Sec. 3.3.7
- 13) Write an Booth Multiplication Algorithm ?
Refer Sec. 3.3.8.1
- 14) Explain Multiplication with Booth Algorithm.
Refer Sec. 3.3.8.1
- 15) Write briefly about Array multiplier ?
Refer Sec. 3.3.7.1
- 15) Explain Division Algorithms.
Refer Sec. 3.3.10
- 16) Explain about Floating-point Arithmetic operations.
Refer Sec. 3.3.11
- 17) Explain about floating point addition/subtraction algorithm.
Refer Sec. 3.3.11.2
- 18) Explain the multiplication of floating point numbers.
Refer Sec. 3.3.11.2
- 19) Explain the Division of two floating point numbers.
Refer Sec. 3.3.11.2

20) Explain briefly about Decimal Arithmetic Unit?

A decimal Arithmetic unit is a digital function that performs digital micro operation. It can add or subtract decimal no. Usually by forming the 1's or 10's complement of the subtracted. The unit accepts the coded decimal no and generates

Results in the same adopted binary code.

A single stage decimal arithmetic unit consists of 9 binary input variable and 5-binary output variable, since a min of 4 bits is required to represent each coded decimal digit. Each stage must have 4 inputs for the augends digit, four inputs for the addend digit and an input carry. The output includes 4 terminals for the sum digit and one for the o/p carry.

BCD adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum can't be greater than 9+9+1=19, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a 4 bit binary adder. The adder will form the sum in binary and produce the result that may change from 0 to 19. These binary number are labeled by K, Z₈, Z₄, Z₂ and Z₁. K is the carry and the subscripts under the letter Z represents the wt. 8,4,2 and 1 that may be assigned to the 4 bits in the BCD code.

The first column in the table lists the binary sums as they appear in the output of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and appear in the 2nd column of the table. It is apparent that when the binary sum is equal to or less than 1001 the corresponding BCD number is identical and therefore no conversion is needed. When the binary sum is greater than 1001 we obtain a non-valid BCD representation

Derivation of BCD Adder

Binary SUM					BCD SUM					Decimal
K	Z ₈	Z ₄	Z ₂	Z ₁	C	S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	0	1	1	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	1	0	0	0	0	8
0	1	0	0	1	1	0	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

One method of adding decimal not in BCD could be to employ one 4 bit binary adder and perform the arithmetic operation one digit at a time. The lower pair of BCD digits is first added to produce a binary sum. If the result is equal to or greater than 1010, it is corrected by adding 0110 to the binary sum. This 2nd operation could automatically produce an output carry for the next pair of significant digits. The next higher order pair of digits, together with the input carry is then added to produce their binary sum. If this

result is greater than or equal to 1010, it is corrected by adding 0110. This produce is repeated until all decimal digits are added.

It is obvious that the correction is needed when the binary sum has an output carry $K=1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8 . To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8 , we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function.

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

When $C=1$ it is necessary to adder 0110 to the binary sum and provide an output carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a 2nd 4-bit binary adder shown in figure below. The two decimal digits together with the input carry are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output carry terminal.

A decimal parallel adder that adds n decimal digits needs n BCD adder stages with the output carry from one stage connected to the input carry of the next higher order stage. To achieve shorter propagation delays, BCD adders include necessary circuits for carry look a head.

Fig: - Block diagram of BCD adder

21) Explain about Decimal Arithmetic Operations?

The algorithms for arithmetic operation with decimal data are similar to the algorithms for the corresponding operation with binary data. Except for the slight modification in the multiplication and division algorithm the same flowcharts can be used for both types of data provided we interpret the micro operation symbols properly.

Decimal numbers in BCD are stored in computer register in groups of 4-bits. Each 4-bit group represents a decimal digit.

A bar over the register letter symbol denotes 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus for decimal number the symbol

$$A \leftarrow A + B + 1$$

denotes transfer of decimal sum formed by adding the original content of A to the 10's complement of B.

Decimal Arithmetic Micro Operation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal number and transfer sum into A.
\overline{B}	9's Complement of B

COMPUTER ARCHITECTURE

$A \leftarrow A + B + 1$	Contents of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
d Shr A	Decimal shift right A
d Shl A	Decimal shift left A

Incrementing or decrement the register is same for binary and decimal except for the number of states that the register is allowed to have. A decimal counter goes through 10 states from 0000 to 1001 and back to 0000, since 9 is the last count. A binary counter goes through 16 states from 0000 to 1111 when incremented and from 1111 to 0000 when decremented.

A decimal shift right or left is preceded by the letter d to indicate that a shift over the 4-bits that a shift is over the 4-bits that hold the decimal digits.

Ex:- Reg A \rightarrow 7860 in BCD. bit pattern is 0111 1000 0110 0000

Then the micro operation d shr A \rightarrow Shift the decimal number to the right to give 0786. This shift is over 4-bits and changes the content of the register.

000 0111 1000 0110

22) Explain BCD Addition & Subtraction in binary-signed magnitude form?

The algorithm for addition and subtraction of binary signed-magnitude number applies also to decimal signed-magnitude number provided that we interpret the micro operation symbols in the proper manner. Similarly the algorithm for binary 2's complement number applies to decimal signed 10's complement number.

Decimal data can be added in three different ways.

Arithmetic Unit

1. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and required only one micro operation.

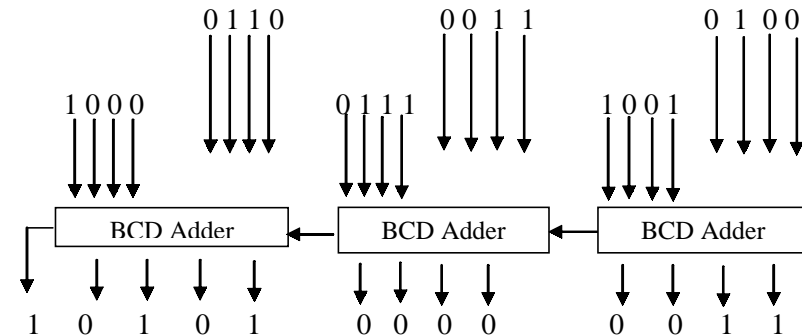


Fig: -Parallel addition (decimal) $624 + 879 = 1503$

2. In the digit – serial bit – parallel method the digits and applied to the single BCD adder serially; while the bits of each coded digit will be transferred in parallel. The sum is formed by shifting the decimal no thru BCD adder one at a time for k decimal digits this configuration required k micro operation one for each decimal shift.

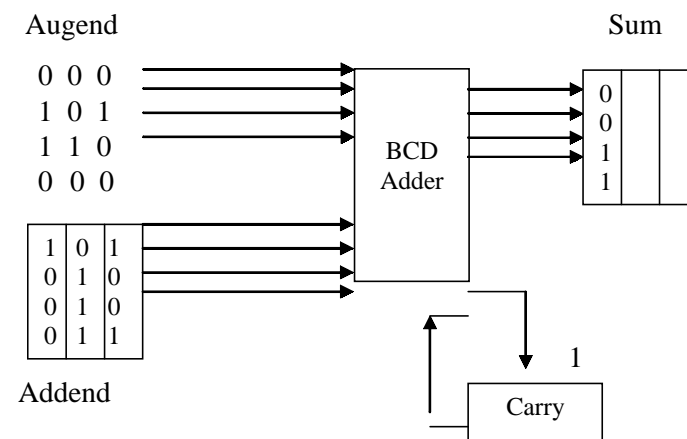


Fig:- Digit-Serial –Bit Parallel

3. In the all serial adder, the bits are shifted one at a time thru a full adder. The bin sum formed after 4 shifts must be corrected in to a valid BCD digit. This correct consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

Augend

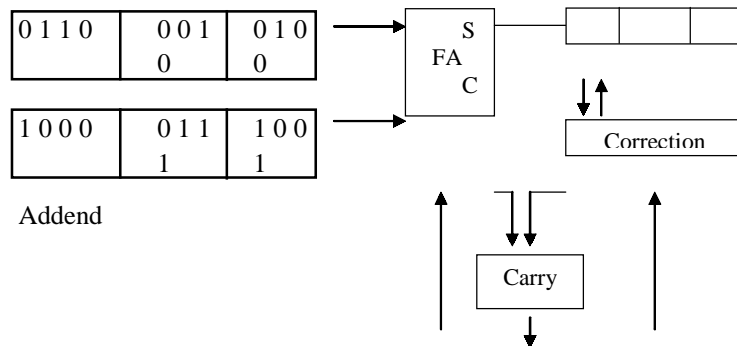


Fig: -All serial decimal add

The parallel method is fast but required large no of adders. The digit serial bit –parallel required only our BCD adder; which is shared by all the digits. It's slower than 11le method. The all-serial method requires minimum amount of equipment but is very slow.

- 23) Explain BCD Multiplication algorithm in binary signed-magnitude form?

The multiplication of fixed – point decimal no is similar to binary except the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9; where as bin multiplier has only 0 and 1 digits

- In the binary case the multiplicand is added to the partial product if the multiplier bits is 1.
- In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product.

The register organization for the decimal multiplication is shown below. We are assuming here 4 digits no., with each digit occupying four bits, for a total of 16 bits for each no. There are 3 register, A, B and Q each having corresponding sign FF As, Bs and Qs. Register A and B have four more bits designated by Ae and Be that provide an extension of one more digit to the Reg. The BCD arithmetic unit adds the 5 digits in 11le and places the sum in the 5-digit A Reg. The end carry goes to FF E. The purpose Ae is to accommodate an overflow while adding the multiplicand to the partial product during the multiplication. The purpose of digit Be is to form the 9's comp of the divisor when subtracted from the partial remainder during the division op~. The least significant digit in register Q is Q_l.

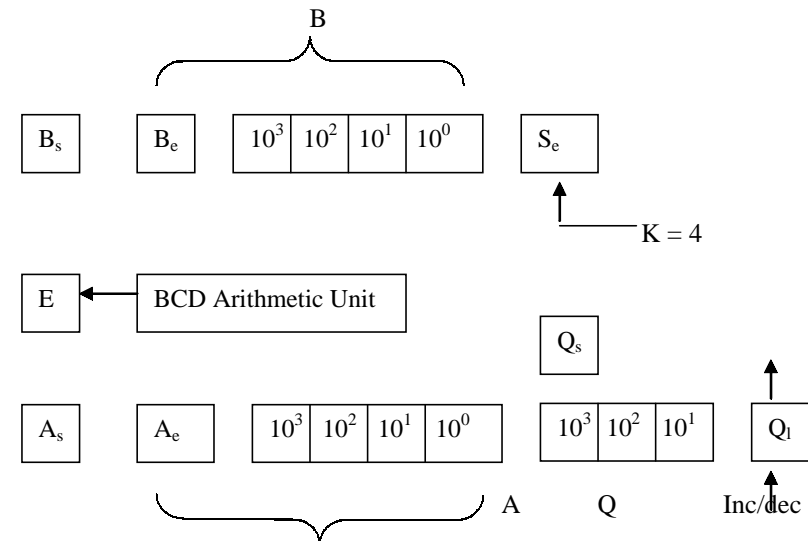


Fig: -Registers for Decimal Arithmetic Multiplication & Division

Algorithm

Initially the entire A register and Be are cleared and the sequence counter SC is set to a no. K equal to the no of digits in the multiplier. The low order digit of the multiplier in QL is checked. If it is not equal to 0, the multiplicand

in B is added to the partial product in A once and QL is decremented QL is checked again and the process is repeated until it is equal to 0. In this way the multiplicand in B is added to the partial product a no of times equal to the multiplier digit. Any temp overflow digit will reside in Ae and can range in value from 0 to 9.

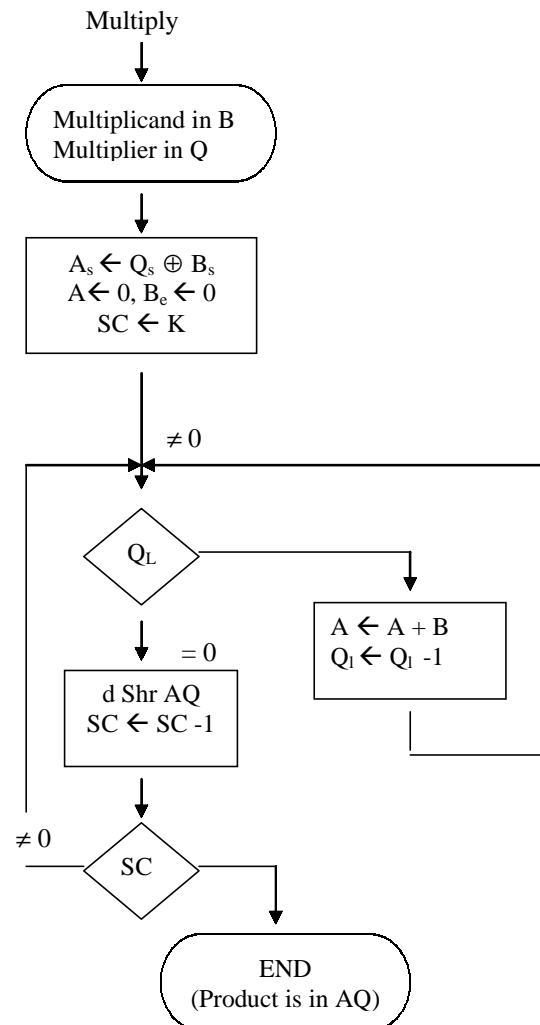


Fig: -Flow chart for Decimal Multiplication

Next the partial product and the multiplier are shifted once to the right. This places zero in A_e and transfers the next multiplier quotient in to Q_L . The process is then repeated k times to form a double length product in AQ.

24) Explain BCD Division algorithm in binary-signed magnitude form?

Decimal Division is similar to binary Division except that quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the division is subtracted from the dividend on partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The decimal Division algorithm is similar to binary algorithm except that the way the quotient bits are formed. The dividend is shifted to the left (or partial remainder) with its most significant digit placed in A_e . The divisor is then subtracted by adding the 10's complement value. Since B_e is initially cleared, its complement value is 9 as reg. The carry in E determine the relative magnitude of A and B. If $E = 0$, it signifies that $A < B$. In this case the divisor is added to restore the partial remainder and Q_L stays at 0. (Inserted there during the slight.). If $E = 1$ it signifies that $A > B$. The quotient digit in Q_L is incremented and divisor is subtracted again. This process is repeated until the sub~ results in a negative difference which is recognized by E being 0. When in this occurs the quotient restore the positive remainder. In this way the quotient digit is made equal to the number of times that the partial remainder 'goes' in to the divisor.

The partial remainder and the quotient bits are shifted once to the left and the process is repeated k times to form k quotient digits. The remainder is then found in the register A and the quotient is in register Q. The value of E is neglected.

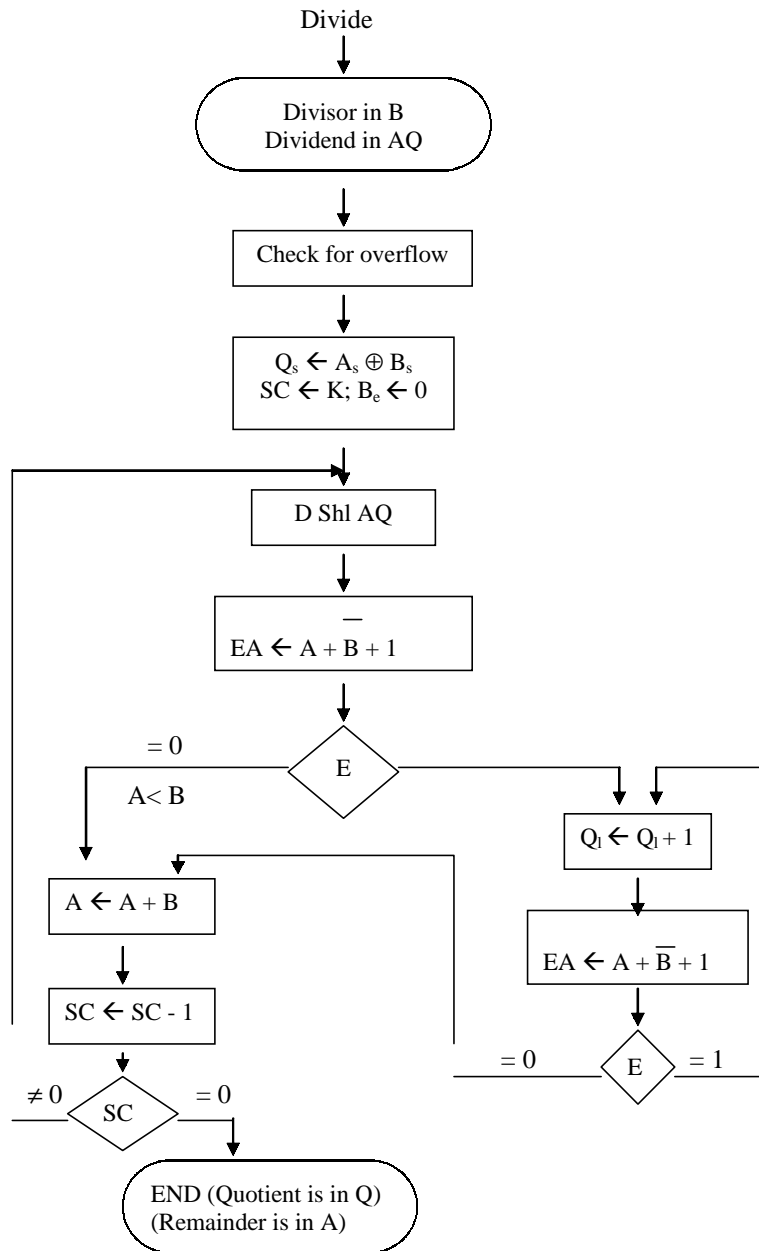


Fig: - Flow Chart for decimal Division

B = 470 x 152 = 0		Multiply by two decimal numbers		
A _e	A	Q	Q ₁	SC
Initial		0 0 0 0	1 5 2	3
Q _L ≠ 0		4 7 0		
	0	1 5 1		
Q _L ≠ 0		4 7 0		
		4 7 0		
Q _L = 0	0	1 5 0		
		9 4 0		
d Shr		0 0 9 4	0 1 5	
Q _L ≠ 0		4 7 0		
	0	0 1 4		
Q _L ≠ 0		5 6 4		
		4 7 0		
Q _L ≠ 0	1	0 1 3		
		0 3 4		
		4 7 0		
Q _L ≠ 0	1	0 1 2		
		5 0 4		
		4 7 0		
Q _L ≠ 0		1	0 1 1	
		9 7 4		
		4 7 0		
Q _L = 0		2	0 1 0	
		4 4 4		
d Shr AQ	0	2 4 4	4 0 1	1
Q _L ≠ 0		4 7 0		
	0	4 0 0		
Q _L = 0		7 1 4		
D Shr AQ	0	0 7 1	4 4 0	0
<div style="text-align: center;"> } Product </div>				