

STUDY MATERIALS ON COMPUTER ORGANIZATION

(As per the curriculum of Third semester B.Sc.
Electronics of Mahatma Gandh University)

Compiled by Sam Kollannore U..

Lecturer in Electronics
M.E.S. College, Marampally

3. ARITHMETIC AND LOGIC UNIT

Basic operation of all digital computers is the addition or subtraction of two numbers. These operations are implemented along with basic logical functions such as AND, OR, NOT and Exclusive OR in the Arithmetic and Logic Unit (ALU) subsystem of the processor. The time needed to perform these operations affect the overall system performance. Compared with arithmetic operations, logic operations are simple to implement. They require only independent Boolean operations on individual bit positions of the operands, whereas carry/borrow lateral signals are required in arithmetic operations.

3.1 Number Representations

Binary number system is used in computers. Consider an n -bit vector (number)

$$B = b_{n-1}.b_{n-2}.....b_1.b_0 \quad \text{where } b_i = 0 \text{ or } 1 \text{ for } i = 0,1,..... n-1$$

This vector can represent positive integer values V in the range 0 to 2^n-1 , where

$$V(B) = b_{n-1} \cdot 2^{n-1} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Three systems are widely used for representing both positive and negative numbers

- 1) *Sign-and-magnitude*
- 2) *1's-compliment*
- 3) *2's-compliment*

- *In all the three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Table shows all the three types of representations using 4-bit numbers*
- *Positive numbers have identical representations in all systems, but negative values have different representations.*
- *In sign-and-magnitude system, negative values are represented by changing the most significant bit to 1. For example +5 is represented by 0101 and -5 is represented by 1101.*
- *In the 1's compliment representation, negative values are obtained by complimenting each bit of the corresponding positive number. E.g. Representation for -3 is obtained by complimenting each bit in the vector 0011 to yield 1100*
- *The operation of obtaining the 1's compliment of a number is equivalent to subtracting that number from $2^n - 1$ i.e. from 1111 (in the case of 4 bit numbers)*
- *In 2's compliment system, a negative number is obtained by subtracting the corresponding positive number from 2^n . Hence the 2's compliment representation is obtained by adding 1 to the 1's compliment representation.*

Note:-

- *There are distinct +0 and -0 representations in both the sign-and-magnitude and 1's compliment systems; but the 2's compliment system has only a +0 representation.*
- *For 4 bit numbers the value -8 is representable in the 2's compliment system but not in the other systems.*
- *2's compliment system yields the most efficient logic circuit implementation (most often used in computers for addition and subtraction)*

<i>B</i>	Values represented		
	$b_3b_2b_1b_0$	Sign and magnitude	1's compliment
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

Fig 3.1 Binary representations

3.2 Addition of Positive Numbers

Consider the case of unsigned numbers. Consider adding two 1-bit numbers. The results are shown in figure. The sum of 1 and 1 requires the 2-bit vector 10 to represent the value 2. We say the sum is 0 and the carry-out is 1.

$$\begin{array}{cccc}
 0 & 1 & 0 & 1 \\
 +0 & +0 & +1 & +1 \\
 \hline
 0 & 1 & 1 & 10 \\
 & & & \downarrow \text{Carry-out}
 \end{array}$$

3.3 RIPPLE - CARRY ADDERS

In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries towards the high-order (left) end. A straight forward 2-level combinational logic circuit implementation of the truth table for addition is shown along with a convenient symbol for the circuit called an Adder or sometimes called a Full-adder.

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x_i}\overline{y_i}c_i + \overline{x_i}y_i\overline{c_i} + x_i\overline{y_i}\overline{c_i} + x_iy_ic_i = x_i \oplus y_i \oplus c_i$$

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

Fig 3.2 (a)

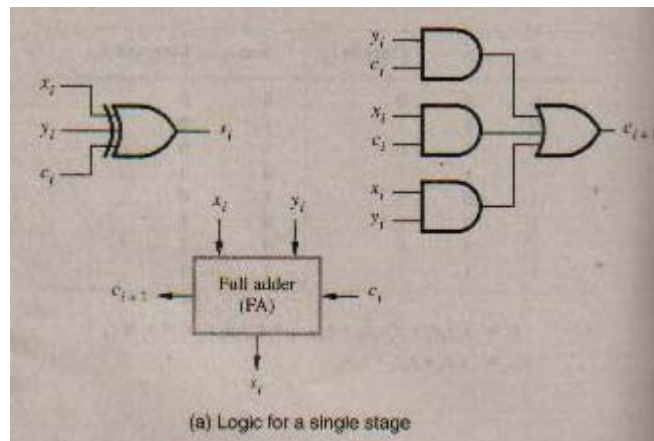
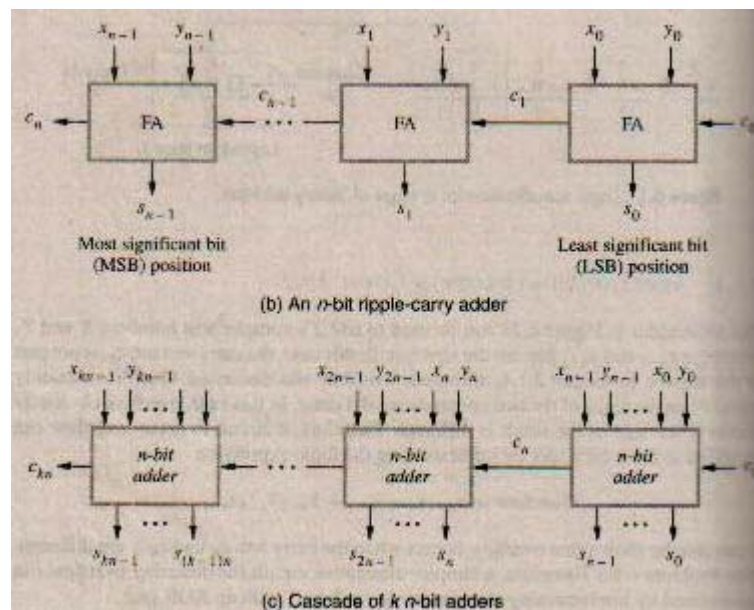


Fig 3.2 (b)



A cascaded connection of n adder blocks can be used to add two n -bit numbers. Since the carries must propagate or ripple through this cascade, the configuration is called an ***n-bit-ripple-carry adder***. When the carry-out c_n , from the MSB position equals 1, an overflow from the operation occurs (i.e. the result cannot be represented in n bits)

The carry-in, c_0 into the LSB position provides a convenient means of adding 1 to a number. (for eg. Adding 1 to the 1's complement of a number forms a 2's complement

3.4 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

Rules for addition and subtraction of n -bit signed numbers using 2's complement representation

1. To add two numbers, add their n -bit representations, ignoring the carry-out signal from the MSB position. The sum will be the algebraically correct value in the 2's complement representation as long as the answer is in the range -2^{n-1} through $+2^{n-1} - 1$.
2. To subtract two numbers X and Y , that is to perform $X-Y$, find out the 2's complement of Y and then add it to X . Again the result will be algebraically correct value in the 2's complement representation if the answer is in the range -2^{n-1} through $+2^{n-1} - 1$.

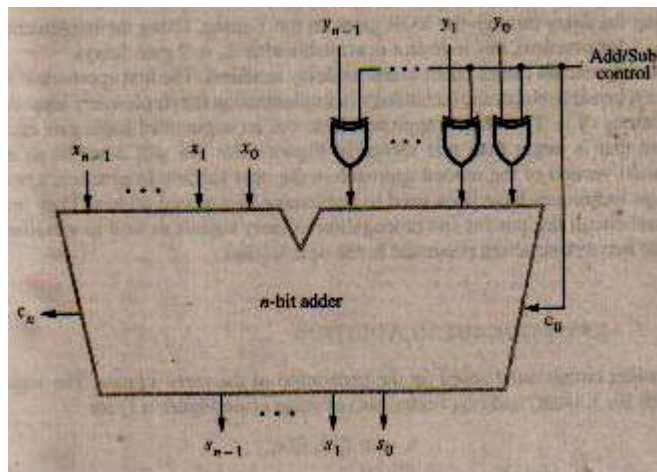
Examples:-

<i>Addition</i>	<i>Subtraction</i>
$\begin{array}{r} 0010 \quad (+2) \\ + 0011 \quad (+3) \\ \hline 0101 \quad (+5) \end{array}$	$\begin{array}{r} 1101 \quad (-3) \\ - 1001 \quad (-7) \\ \hline \end{array} \Rightarrow \begin{array}{r} 1101 \\ + 0111 \\ \hline 0100 \quad (+4) \end{array}$
$\begin{array}{r} 1011 \quad (-5) \\ + 1110 \quad (-2) \\ \hline 1001 \quad (-7) \end{array}$	$\begin{array}{r} 0010 \quad (+2) \\ - 0100 \quad (+4) \\ \hline \end{array} \Rightarrow \begin{array}{r} 0010 \\ + 1100 \\ \hline 1110 \quad (-2) \end{array}$
$\begin{array}{r} 0111 \quad (+2) \\ + 1101 \quad (-3) \\ \hline 0100 \quad (+4) \end{array}$	$\begin{array}{r} 1001 \quad (-7) \\ - 0001 \quad (+1) \\ \hline \end{array} \Rightarrow \begin{array}{r} 1001 \\ + 1111 \\ \hline 1000 \quad (-8) \end{array}$

Note: In all these 4-bit examples, the answer falls into the representable range of -8 and $+7$. When answers do not fall within the representable range, we say that arithmetic overflow has occurred.

3.5 BINARY ADDITION – SUBTRACTION LOGIC NETWORK

Fig 3.3



The subtraction operation requires the subtrahend i.e. the bottom value to be 2's complemented before the addition is performed. The Add/Sub control wire is set to 0 for addition and 1 for subtraction. So during addition, the Y vector is applied without change to one of the adder inputs along with a carry in signal $c_0 = 0$. During subtraction, the Y vector is 1's complemented by the EXOR gate and c_0 is set to 1 to complete the 2's complementation of Y.

Note:- An EXOR gate can be added to detect the overflow condition $c_n \oplus c_{n-1}$

3.5.1 Overflow Logic

In all the 4 bit examples the answers fall into the representable range of -8 through +7. When answers do not fall within the representable range, we say that arithmetic overflow has occurred. When adding unsigned numbers the carry out c_n from the MSB position serves as the overflow indicator. However this does not work for adding signed numbers.

Eg.

Adding +7 and +4

0111	
0100	
01011	

which is -5 ; a wrong result. The carry out signal from MSB position is 0

Similarly adding -4 and -6

1100	
1010	
10110	

which is +6; again a wrong result. The carry out signal is 1.

Conclusions:-

Overflow can occur only when adding two numbers that have the same sign and the sign of the sum is different from them.

A simple way to detect overflow is to examine the signs of the two summands X and Y and the sign of the result. In an n-bit adder, we can define a signal overflow by the logical expression

$$\text{Overflow} = x_{n-1}y_{n-1}s_{n-1} + x_{n-1}y_{n-1}\bar{s}_{n-1}$$

It can also be shown that overflow occurs when the carry bits c_n and c_{n-1} are different

Therefore a simple alternative circuit for detecting overflow can be obtained by implementing the expression $c_n \oplus c_{n-1}$ with an EXOR gate.

3.6 DESIGN OF FAST ADDERS

Drawback of n -bit ripple-carry adder is the too much delay in developing its output. This delay is considerable only by considering the speed of other processor components and the data transfer times of registers and cache memory.

The length of the delay through a network of logic gates depends on

- i) IC fabrication technology and
- ii) number of gates in the path from input to output.

After a particular technology is chosen, the delay caused with any combinational logic circuit can be calculated by adding the number of logic gate delays along the longest path through the network. In the above example longest signal propagations from inputs x_0, y_0 and c_0 at the LSB position to outputs c_n, s_{n-1} at the MSB position.

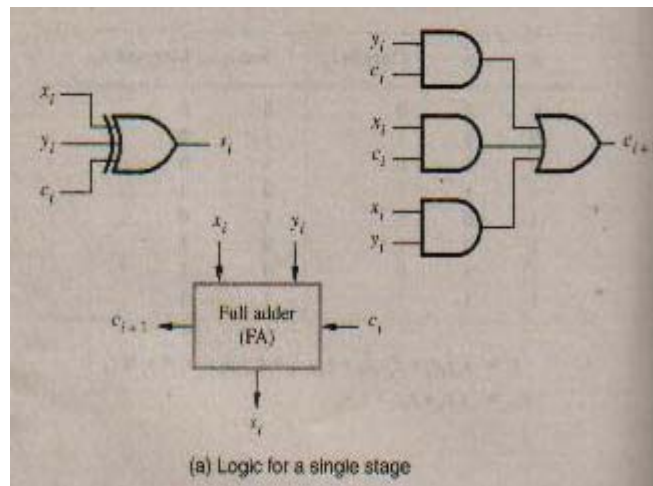
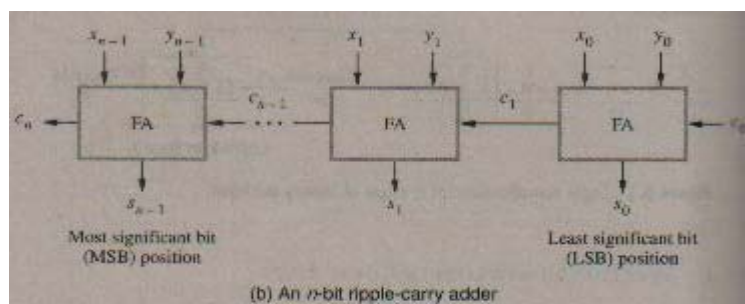


Fig 3.4



Using the logic gate implementation c_{n-1} is available in $2(n-1)$ gate delays, and s_{n-1} is correct one XOR gate delay later. The final carry-out, c_n is available after $2n$ gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit (as shown in the previous topic), all sum bits are available in $2n$ gate delays, including the delay through the XOR gates on the Y input. Using the implementation $c_n \oplus c_{n-1}$ for overflow, this indicator is available after $2n + 2$ gate delays.

Two approaches can be taken to reduce delay in adders.

- i) To use the fastest possible electronic technology in implementing the ripple-carry logic design
- ii) To use an augmented logic gate network structure that is larger than that shown in figure 3.4(b)

3.7 CARRY-LOOKAHEAD ADDITION

A fast adder circuit must speed up the generation of the carry signals. We have

$$s_i = x_i \oplus y_i \oplus c_i$$

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

Factoring the second equation

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

We can write $c_{i+1} = G_i + P_i c_i$; where $G_i = x_i y_i$ and $P_i = x_i + y_i$

The expressions G_i and P_i are called Generate and Propagate functions for stage i

If the generate function for stage i is equal to 1, then $c_{i+1} = 1$, independent of the input carry c_i . This occurs when both x_i and y_i are 1.

The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1.

All G_i and P_i functions can be formed independently and in parallel in one logic gate delay after the X and Y vectors are applied to the inputs of an n -bit adder. Each bit stage contains:

- i) an AND gate to form G_i
- ii) an OR gate to form P_i
- iii) a 3- input XOR gate to form s_i

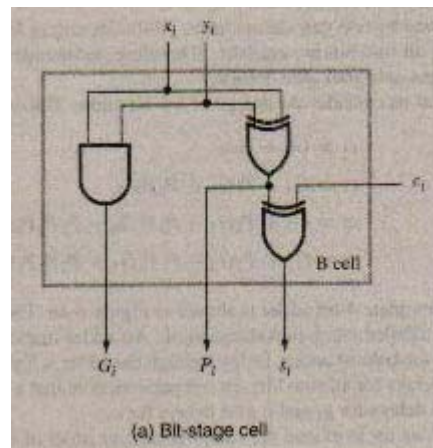
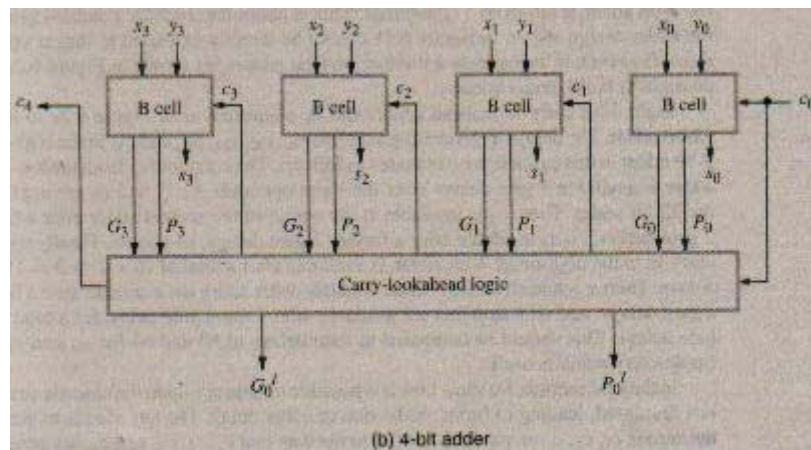


Fig 3.5



Note:- Here P_i is realized as $P_i = x_i \oplus y_i$, which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$. This does not effect the final c_{i+1} , since when $x_i = y_i = 1$; $G_i = 1$. Thus using a cascade of 2-input XOR gates to realize the 3-input XOR function, the basic cell B in the figure can be used in each bit stage.

Expanding c_i in terms of $i-1$ subscripted variables and substituting into the c_{i+1} expression, we obtain

$$\begin{aligned} c_{i+1} &= G_i + P_i c_i \\ &= G_i + P_i (G_{i-1} + P_{i-1} c_{i-1}) \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1} \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} (G_{i-2} + P_{i-2} c_{i-2}) \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} c_{i-2} \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} P_{i-2} \dots P_0 c_0 \end{aligned}$$

Thus 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, 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.

Example: Consider the design of a 4-bit adder. The carries can be implemented as

$$c_1 = G_0 + P_0 c_0$$

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

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

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

Carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a **carry-look ahead adder**.

Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, note that a 4-bit ripple-carry adder requires 7 gate delays for s_3 and 8 gate delays for c_4

Sign extension:-

We often need to represent a given number in 2's compliment system by using a large number of bits. For a positive number, this is achieved by adding 0s to the left. Similarly for a negative number, longer number with the same value is obtained by replicating the sign bit to the left as many times as desired. This operation is called Sign extension.

3.8 MULTIPLICATION OF POSITIVE NUMBERS

The product of two n -digit numbers can be accommodated in $2n$ digits as shown in the manual multiplication example.

$$\begin{array}{r} 1101 \quad (13) \text{ Multiplicand M} \\ \times 1011 \quad (11) \text{ Multiplier Q} \\ \hline 1101 \\ 1101 \\ 0000 \\ 1101 \\ \hline 10001111 \quad (143) \text{ Product P} \end{array}$$

} Partial products **Fig 3.6 (a)**

In binary system, if the multiplier bit is 1, the multiplicand is entered in the appropriate position to be added to the partial product. If the multiplier bit is 0, then 0s are entered.

3.8.1 Method I – Using combinational logic circuit

Binary multiplication of positive operands can be implemented in a pure combinational two dimensional logic array as shown. The main component in any cell is an Adder circuit. The AND gate in any cell determines whether the multiplicand bit m_j , is added to the partial product bit, based on the value of the multiplier bit q_i . The multiplicand is shifted left one position per row by the diagonal signal path. PP4 is the desired product.

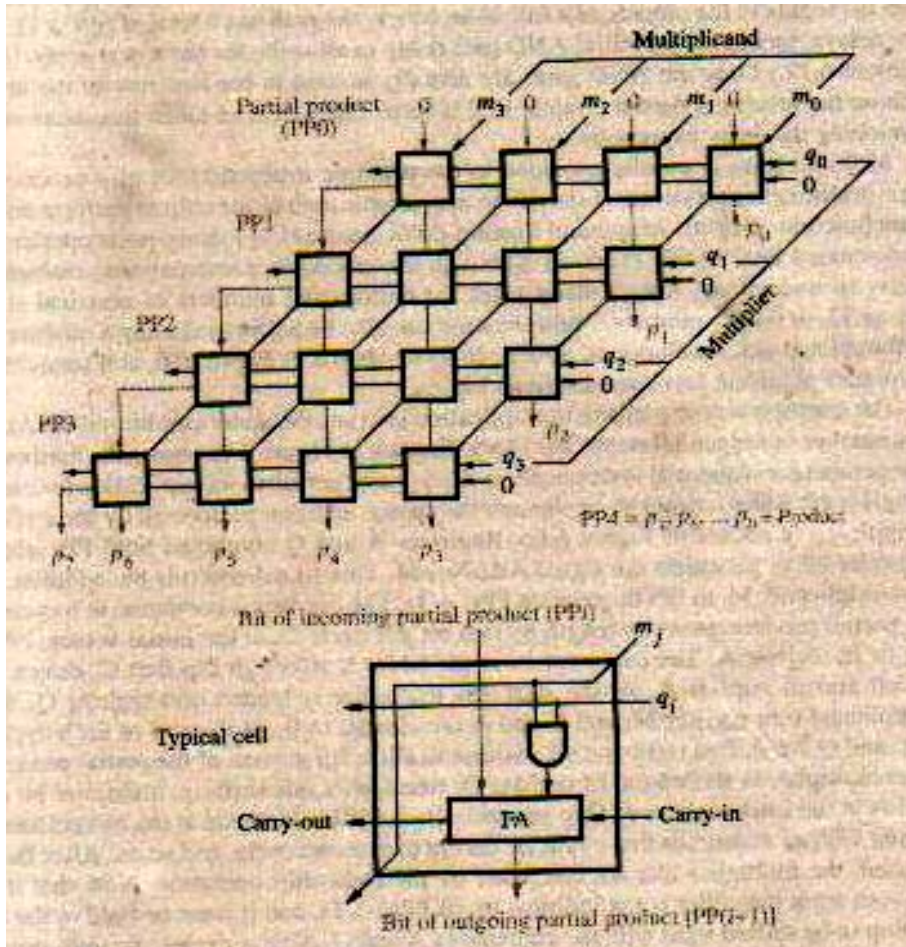


Fig 3.6 (b)

3.8.2 Method II – Using combinational logic circuit & sequential techniques

Multiplication can also be performed using a mixture of combinational array techniques as shown above and sequential techniques. The simplest hardware arrangement for sequential multiplication is as shown.

The circuit performs the multiplication by using a single n -bit adder n time. Register A and Q combined to hold the Partial Product i (PP_i) while the multiplier bit q_i generates the signal Add/Noadd. This signal controls the addition of multiplicand M to PP_i to generate PP_{i+1} . The product is computed in n cycles. The partial product grows in length by one bit in each cycle. The carry-out from the adder is stored in flip flop C.

Fig 3.7(a)

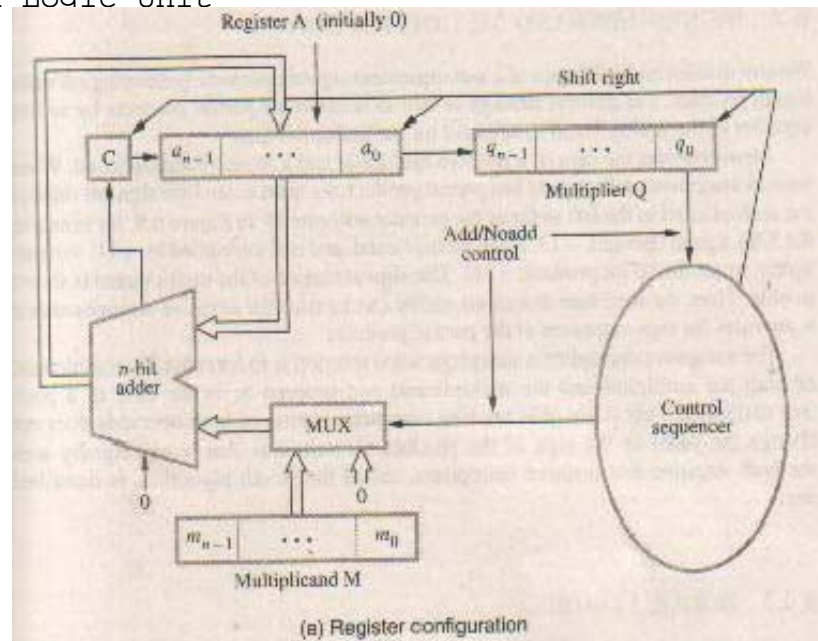
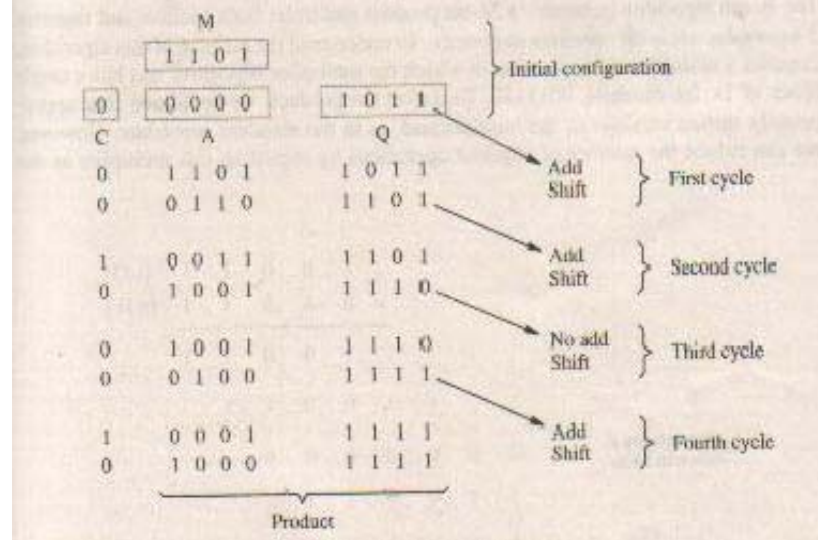


Fig 3.7(b)



At the start the multiplier is loaded into register Q, multiplicand into register M and C & A are cleared to 0. At the end of each cycle, C, A and Q are shifted right one bit position to allow for growth of the partial product. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at correct time in each cycle (with q_0 during the 1st cycle, q_1 during the 2nd cycle etc.). After they are used, the multiplier bits are discarded by the right shift operation. Carry-out from the adder is held in C and this is shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half in register Q. The multiplication example of Fig 3.6(a) is shown in Figure 3.7(b) as it would be performed by this hardware arrangement.

In this Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1 and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left

Example for Booth multiplication algorithm:

$$\begin{array}{r}
 0101101 \text{ (Multiplicand)} \\
 \times 0011110 \text{ (Multiplier)} \\
 \hline
 0000000 \\
 0101101 \\
 0101101 \\
 0101101 \text{ (No sign extension required because the multiplicand is positive)} \\
 0101101 \\
 0000000 \\
 0000000 \\
 \hline
 00010101000110
 \end{array}$$

	0 1 0 1 1 0 1	(Multiplicand)
	\times 0 +1 0 0 0 -1 0	(Multiplier)
<i>0 0 0 0 0 0 0</i>	<u>0 0 0 0 0 0 0</u>	
<i>1 1 1 1 1 1 1</i>	0 1 0 0 1 1	(2's compliment of the multiplicand)
<i>0 0 0 0 0 0 0</i>	0 1 0 1 1 0 1	
<i>0 0 0 0 0 0 0</i>	1's - 1 0 1 0 0 1 0 +	
<i>0 0 0 0 0 0 0</i>	<u>1</u>	
<i>0 0 0 1 0 1 1 0 1</i>	2's - 1 0 1 0 0 1 1	
<i>0 0 0 0 0 0 0</i>		
<u><i>0 0 0 1 0 1 0 1 0 0 0 1 1 0</i></u>		Sign extension bits shown in <i>italics</i>

Note:

- 1) Booth algorithm can be extended to any number of blocks of 1s in a multiplier including the situation in which a single 1 is considered as a block.
- 2) If the least significant bit is 0 or 1, then we must assume that an implied 0 lies to its right.
- 3) Booth algorithm can be used for negative multipliers also

16-bit number

$$\begin{array}{r}
 001011001110101100 \\
 0+1-1+10-10+100-1+1-1+10-100
 \end{array}$$

For negative multipliers:

$$\begin{array}{r}
 01101 \quad (+13) \\
 \hline
 11010 \quad (-6)
 \end{array}$$

$$\begin{array}{r}
 01101 \quad (+13) \\
 \times 0\bar{1}+1\bar{1}0 \quad (-6) \\
 \hline
 0000000000 \\
 1111110011 \\
 00001101 \\
 1110011 \\
 \hline
 0000000 \\
 \hline
 1110110010 \quad (-78)
 \end{array}$$

(2's complement of the multiplicand 01101)
 01101
 1's - 10010 +
 1
 2's - 10011

Sign extension bits shown in *italics*

Booth multiplier recoding table		
Multiplier		Version of multiplicand selected by Bit <i>i</i>
Bit (<i>i</i>)	Bit (<i>i</i> -1)	
0	0	0 × M
0	1	+1 × M
1	0	-1 × M
1	1	0 × M

Fig 3.8

The transformation from original multiplier to booth recoded multiplier is called skipping over 1's

16-bit worst case multiplier

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 *(no blocks of 1's)*

16-bit ordinary multiplier

1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0 *(some blocks of 1's)*

16-bit good multiplier

0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 *(large blocks of 1's)*

Attractive features of Booth Algorithm

1. *It handles both positive and negative multipliers uniformly*
2. *It achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1's.*

3.11 FAST MULTIPLICATION

Two techniques to speed-up the multiplication operation

3.11.1 BIT-PAIR RECODING OF MULTIPLIERS

$$\begin{array}{r}
 W \qquad \qquad 1\ 0\ 1\ 0\ 1 \\
 + X \qquad \qquad 1\ 1\ 0\ 1\ 1 \\
 \hline
 A \qquad \qquad 1\ 1\ 0\ 0\ 0\ 0 \\
 + Y \qquad \qquad 1\ 0\ 1\ 0\ 0 \\
 \hline
 Z \qquad \qquad 1\ 0\ 0\ 0\ 1\ 0\ 0
 \end{array}$$

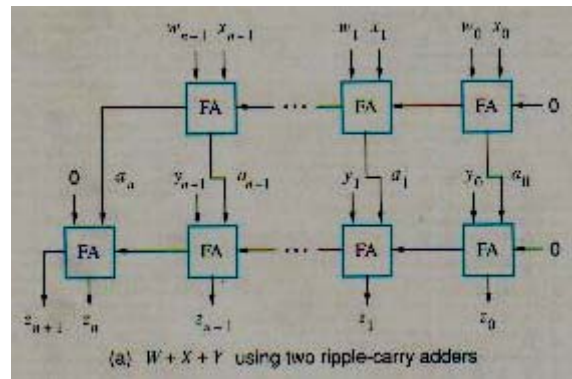


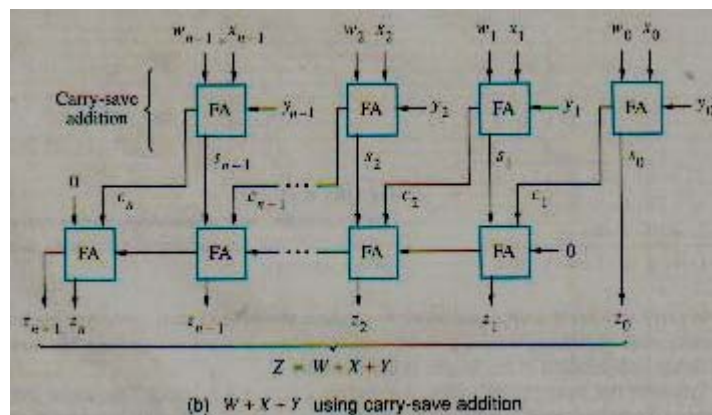
Fig 3.9

Method II: In a second method, instead of adding W and X to produce A , we introduce the bits of Y into the carry inputs. This generates the vectors S and the “saved” carries C as the outputs of the upper row. In the second row, S and C are added in a ripple-carry adder to produce the desired sum Z .

$$\begin{array}{r}
 W \qquad \qquad 1\ 0\ 1\ 0\ 1 \\
 X \qquad \qquad 1\ 1\ 0\ 1\ 1 \\
 + Y \qquad \qquad 1\ 0\ 1\ 0\ 0 \\
 \hline
 S \qquad \qquad 1\ 1\ 0\ 1\ 0 \\
 + Y \qquad \qquad 1\ 0\ 1\ 0\ 1 \\
 \hline
 Z \qquad \qquad 1\ 0\ 0\ 0\ 1\ 0\ 0
 \end{array}$$

Here the intermediate sum A is not generated in the second method. Carry-save addition transforms W , X and Y into S and C , because all bits of S and C are available in parallel.

Fig 3.10



Consider the addition of many summands. We can group the summands into threes and performs carry-save addition on them. We continue this process until there are only two vectors remaining. They can be added in a ripple-carry adder or a carry-lookahead adder to produce the desired sum. Each carry-save addition reducing three to two is done in only one FA unit delay independent of the length of the number.

Consider the example of adding 9 numbers using this method:-

Step I : initial 3 groups of three numbers each is reduced to 6 numbers.

Step II: These 6 nos. are reduced to 4.

Step III: 4 reduced to 3

Step IV: then 3 reduced to 2 nos.

Step V: Final 2 numbers are added to produce the desired sum.

The complete operation requires 4 full adder unit delays to do the carry-save additions, followed by a full addition operation on the final 2 vectors.

3.12 Applications of High-Speed Multiplication Techniques

- The addition of 32 numbers using the carry-save addition method requires only 7 full adder unit delays before the final Add operation. In general about $1.7 \log_2 K$ full adder unit delays (i.e. carry-save addition steps) are needed to reduce K summands to 2 vectors which when added, produce the desired sum.
- Again a carry-lookahead adder can be used effectively to add the final 2 vectors because all bits of these vectors are produced in parallel as the result of the carry-save addition operation. This method saves time and enhances performance

Conclusion:

- Bit-pair recoding of the multiplier derived from the Booth algorithm, reduces the number of summands by up to a factor of 2.
- These summands can then be reduced to only 2 by using a relatively small number of carry-save addition of steps.
- The final product can be generated by one addition operation which is speeded up by using a lookahead adder.

In summary, all the 3 basic techniques – bit-pair recoding of the multiplier, carry-save addition of the summands and lookahead addition – have been used in high-performance processors to reduce the time needed to perform multiplication.

3.13 FLOATING POINT NUMBERS

Integers are considered of having an implied binary point at the right end of the number (Fixed point numbers). It is also possible to assume that the binary point is just to the right of the sign bit, thus representing a fraction.

Consider an integer in 32 bit, signed, fixed-point format. The approximate range of value represented by this system is from 0 to $\pm 2.15 \times 10^9$. If we consider them to be fractions, the range is approximately $\pm 4.55 \times 10^{-10}$ to ± 1 . This range is not sufficient for scientific calculations involving parameters like Avogadro's number (6.0247×10^{23} mole⁻¹) or Planck's constant (6.6254×10^{-27} erg.s). Therefore we need to accommodate both very large numbers and very small fractions. This means that the computer must be able to represent numbers and operate on them in such a way that the position of binary point is variable and is automatically adjusted as computation proceeds. In such a case the binary point is said to float and the numbers are called floating point numbers. (Where as in the fixed-point numbers, the binary point is always in the same position)

3.13.1 FLOATING-POINT REPRESENTATION

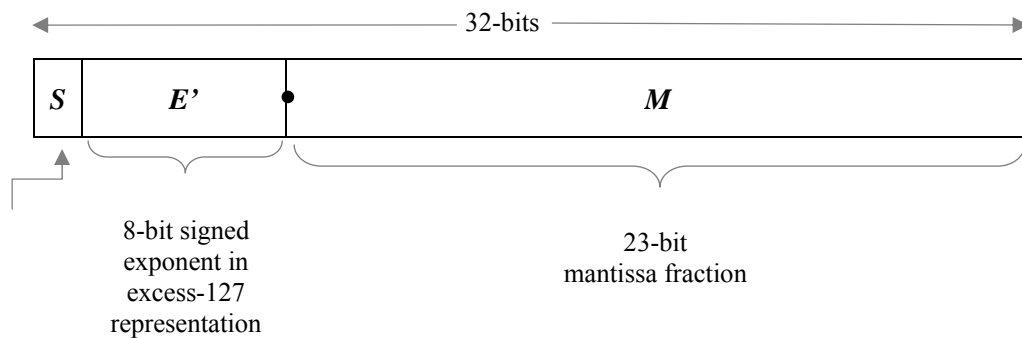
In decimal scientific notation, numbers may be written as 6.0247×10^{23} , 6.6254×10^{-27} and so on. These numbers are said to have five significant digits. Scale factors (10^{23} , 10^{-27} and so on) indicate

the position of the decimal point with respect to the significant digits. By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be normalized.

A floating point representation is one in which a number is represented by its sign, a string of significant digits commonly called mantissa and an exponent to an implied base for the scale factor.

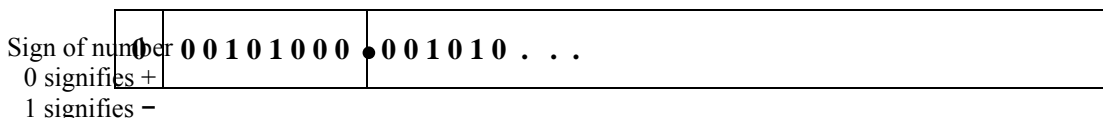
Eg:- $\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y_1Y_2}$ where X_i and Y_i are decimal digits.

Here both the number of significant digits (7) and the exponent range (± 99) are sufficient for a wide range of scientific calculations. It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits which is a standard computer word length. Out of this 32 bits, 24 bits for mantissa (which can approximately represent a 7 digit number) and 8 bits for exponent to an implied base 2. Since the leading nonzero bit of a nonzero mantissa must be a binary 1, it is not included explicitly in the representation. Anyway one bit is needed for the sign of the number. This standard is developed by IEEE.



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

Fig (a) Single-precision



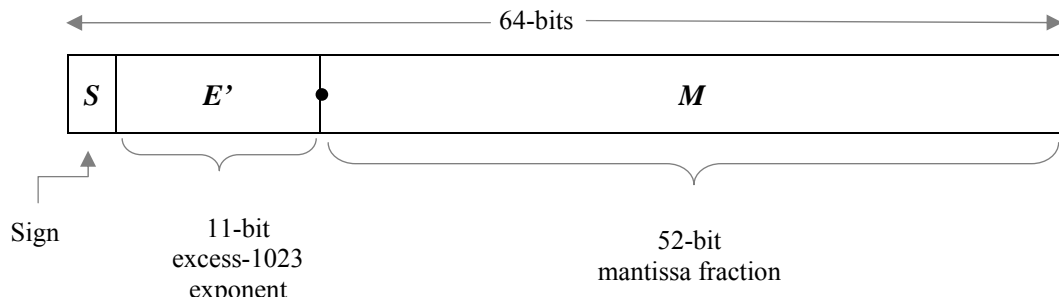
$$\text{Value represented} = 1.001010 \dots 0 \times 2^{-87}$$

Fig (b) Example of a single-precision number

Fig 3.11 (a & b)

Here 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. Thus E' is in the range $0 \leq E' \leq 255$. Actual exponent is in the range $-126 \leq E \leq +127$ (0 and 255 are used to represent special values). Thus scale factor has a range of 2^{-126} to 2^{+127} which is equal to $10^{\pm 38}$. Since binary normalization is used, the most significant bit of the mantissa is always equal to 1. This bit is not explicitly represented- it is assumed to be at immediate left of the binary point. The 23 bits stored in the M field represent the fractional part of the mantissa i.e. the bits to the right of the binary point. This 32 bit standard representation is called a single precision representation because it occupies a single 32-bit word.

To provide more precision and range for floating point numbers IEEE standard specifies a double precision format as shown below



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

Fig 3.11(c) Double precision

Here 11 bit Excess-1023 exponent E' has the range $0 < E < 2047$. Actual exponent is in the range $-1022 \leq E \leq 1023$ providing scale factors of 2^{-1022} to 2^{1023} (which is equal to approximately $10^{\pm 308}$) 53 bit mantissa provides a precision equivalent to about 16 decimal digits

Note:- Double precision representation is optional. Any commercial computer implementation must provide at least single-precision representation to confirm to the IEEE standard.