MODULE 4: ARITHMETIC OPERATIONS

- Numbers, Arithmetic Operations And Characters
- Addition and subtraction of signed numbers
- Fast Adders
- Multiplication
- Division

NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

Computer stores information in binary form 0s and 1s. Information are stored in **bits –** binary digits . Common way to represent characters and numbers in a computer is in the form of string of bits.

NUMBER REPRESENTATION

- Numbers can be represented in 3 formats:
  1) Sign and magnitude
  2) 1's complement
  3) 2's complement

- In all three formats, MSB=0 for +ve numbers & MSB=1 for -ve numbers.
- Positive number have the same representation in all 2 systems, but representation of –ve number varies.

- In **sign-and-magnitude system**,
  negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value.
  For ex, +5 is represented by 0101 &
  -5 is represented by 1101.

- In **1's complement system**,
  negative values are obtained by complementing each bit of the corresponding positive number.
  For ex, -5 is obtained by complementing each bit in 0101 to yield 1010.

- In **2's complement system**,
  Negative values are obtained by complementing each bit and then adding 1 to complemented value.
  For ex, -5 is obtained by complementing each bit in 0101 & then adding 1 to yield 1011. 2's complement system yields the most efficient way to carry out addition/subtraction operations.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
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

Figure 1.3 Binary, signed-integer representations.

ADDITION OF POSITIVE NUMBERS

• Consider adding two 1-bit numbers.

• The sum of 1 & 1 requires the 2-bit vector 10 to represent the value 2. We say that sum is 0 and the carry-out is 1.

```
  0      1      0      1
+ 0    + 0    + 1    + 1
---    ---    ---    ---
  0      1      1     10
                       ↑
                   Carry-out
```

Figure 2.2 Addition of 1-bit numbers.

ADDITION & SUBTRACTION OF SIGNED NUMBERS

2's complement system is most efficient method for performing addition and subtraction operations.

• Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system .

Rule 1:

- **To Add** two numbers, add their n-bits and **ignore the carry-out signal** from the MSB position.
- Result is correct, if it lies in the range -2^{n-1} to $+2^{n-1}-1$.

Rule 2:

- **To Subtract** two numbers X and Y (that is to perform X-Y), take the 2's complement of Y and then add it to X as in rule 1.
- Result is correct, if it lies in the range (2^{n-1}) to $+(2^{n-1}-1)$.

• When the result of an arithmetic operation is outside the representable-range, an arithmetic **overflow** is said to occur.

• To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called **sign extension**.

OVERFLOW IN INTEGER ARITHMETIC

- When result of an arithmetic operation is outside the representable-range, an **arithmetic overflow** is said to occur.
• For example: If we add two numbers +7 and +4, then the output sum S is 1011(+0111+0100), which is the code for -5, an incorrect result.

• An **overflow occurs in following 2 cases**

1) Overflow can occur only when adding two numbers that have the same sign, but the result has the other sign. (Eg – overflow occurs when, Adding of 2 +ve numbers, gives –ve number as result).

2) When result of an arithmetic operation is outside the representable-range, an **overflow** is said to occur.

3) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow.

ADDITION & SUBTRACTION OF SIGNED NUMBERS n-BIT RIPPLE CARRY ADDER

- A cascaded connection of n full-adder blocks can be used to add 2-bit numbers.
- Each stage of addition takes two bits to be added along with the carry-in bit.
- Since carries must propagate (or ripple) through cascade, the configuration is called an n-bit ripple carry adder.
- C_i is the carry-in bit to the ith stage, which produces the sum Si and an carry-out bit C_{i+1}

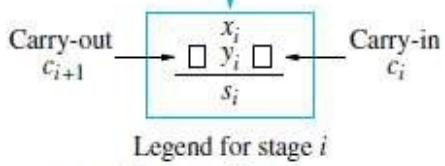
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

Example:

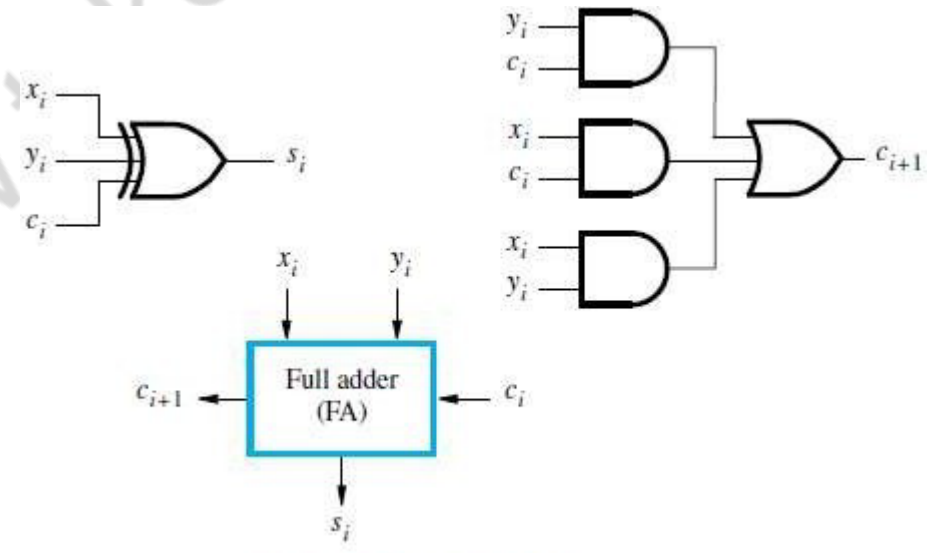
$$\begin{array}{r} X \\ +\,Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\,6 \\ \hline 13 \end{array} = \begin{array}{r} 0\ 1\ 1\ 1 \\ +\,0\ 0\ 1\ 1 \\ \hline 1\ 1\ 0\ 1 \end{array}$$



Legend for stage i

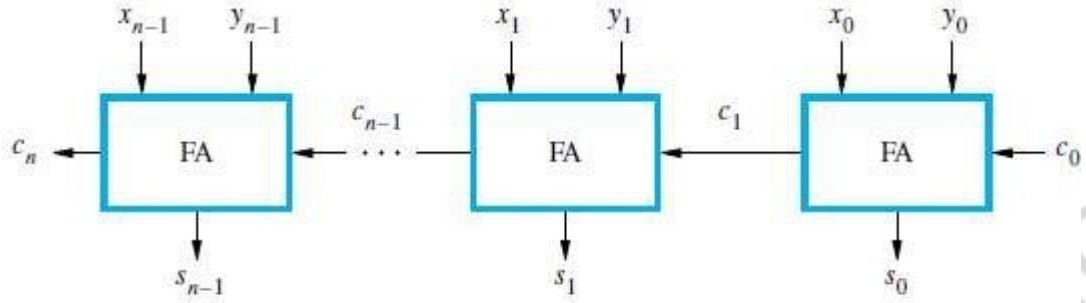
Figure 9.1 Logic specification for a stage of binary addition.

Circuits for Si and C_{i+1} can be represented as -



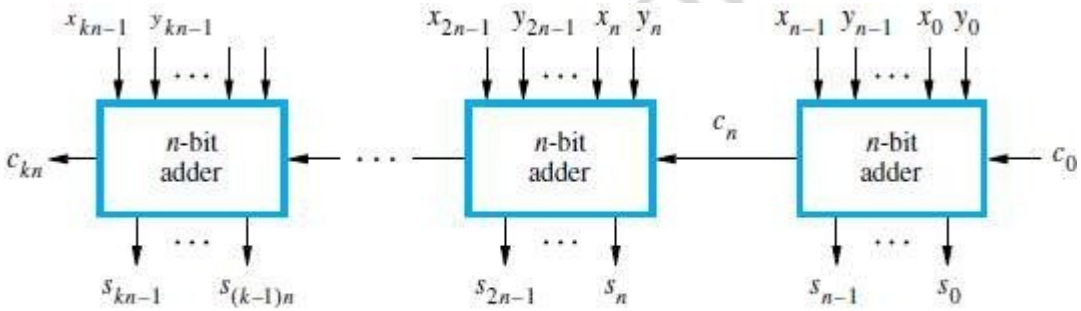
The circuits of Si and Ci+1 are in the Full Adder (FA).

The above circuit is to add two one bits. A number of such circuits are cascaded to add two 'n'bit numbers X and Y. Such a cascaded circuit where carry bit ripples from one FA to another is called a "n bit ripple – carry adder".



(b) An *n*-bit ripple-carry adder

To add k such n – bit numbers, the below circuit can be used -



(c) Cascade of *k n*-bit adders

Figure 9.2 Logic for addition of binary numbers.

ADDITION/SUBTRACTION LOGIC UNIT

- The n-bit adder can be used to add 2's complement numbers X and Y .
- **Overflow** can only occur when the signs of the 2 operands are the same.

  Overflow = $x_{n-1} \bar{y}_{n-1} \bar{S}_{n-1} + \bar{x}_{n-1} y_{n-1} S_{n-1}$
- In order to perform the subtraction operation X-Y on 2's complement numbers X and Y; we form the 2's complement of Y and add it to X.
- Addition or subtraction operation is done based on value applied to the **Add/Sub input control-line**.
- Control-line=0 for addition, so that Y value is unchanged and is sent as one of the adder inputs.
- Control-line=1 for subtraction, the Y value is complemented. Carry bit is also set to '1' so as to add '1' to the first bit to find the 2's complement of Y.

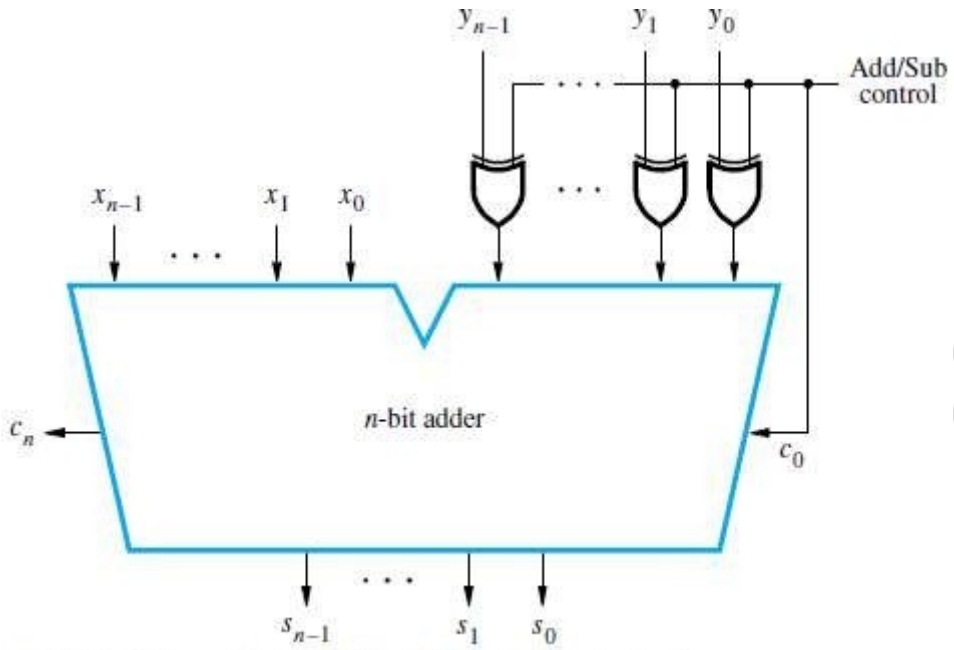


Figure 9.3 Binary addition/subtraction logic circuit.

DESIGN OF FAST ADDERS

• **Drawback of ripple carry adder:** Delay occurs in n-bit ripple carry adder structure. The delay depends on number of gates used in the path from inputs to outputs and also on the electronic technology used in the adders. If the adder is used to implement the addition/subtraction, all sum bits are available in 2n gate delays.

• Two approaches can be used to reduce delay in adders:

1) Use the fastest possible electronic-technology in implementing the ripple-carry design, use of carry-Look ahead addition.
2) Use an augmented logic-gate network structure.

CARRY-LOOKAHEAD ADDITIONS (CLA)

During addition, bits of two operands can be added instantly, but problem is with the carry bit. The previous carry-bit (carry-in bit) is required for operation of ith stage.

In carry-look ahead addition, the operation can take place simultaneously in any stage, ones the Co (carry bit of 1st stage) is known.

• The logic expression for si(sum) and ci+1(carry-out) of stage i are

$$s_i = x_i \oplus y_i \oplus c_i \quad \text{----------(1)}$$

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

- Factoring (2), gives

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

  Can be written as, $C_{i+1}=G_i+P_iC_i$

  where $G_i=x_iy_i$ and $P_i=x_i+y_i$

- The expressions G_i and P_i are called **generate and propagate functions** respectively.

- If $G_i=1$, then $C_{i+1}=1$, independent of the input carry c_i. This occurs when both x_i and y_i are 1. Propagate function means that an input-carry will produce an output-carry when either $x_i=1$ or $y_i=1$.
- All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay.

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

  $c1=G0+P0c0$

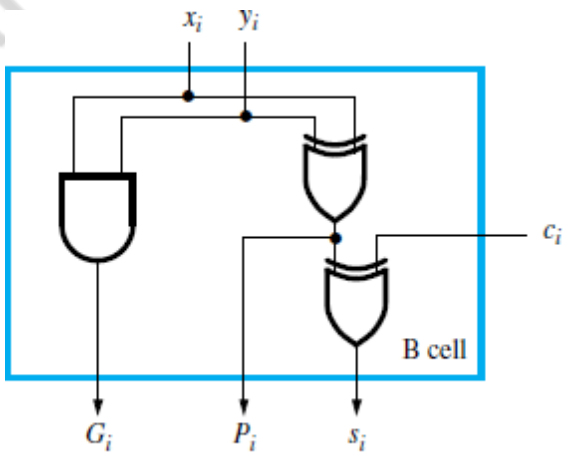
  $c2=G1+P1G0+P1P0c0$

  $c3=G2+P2G1+P2P1G0+P2P1P0c0$

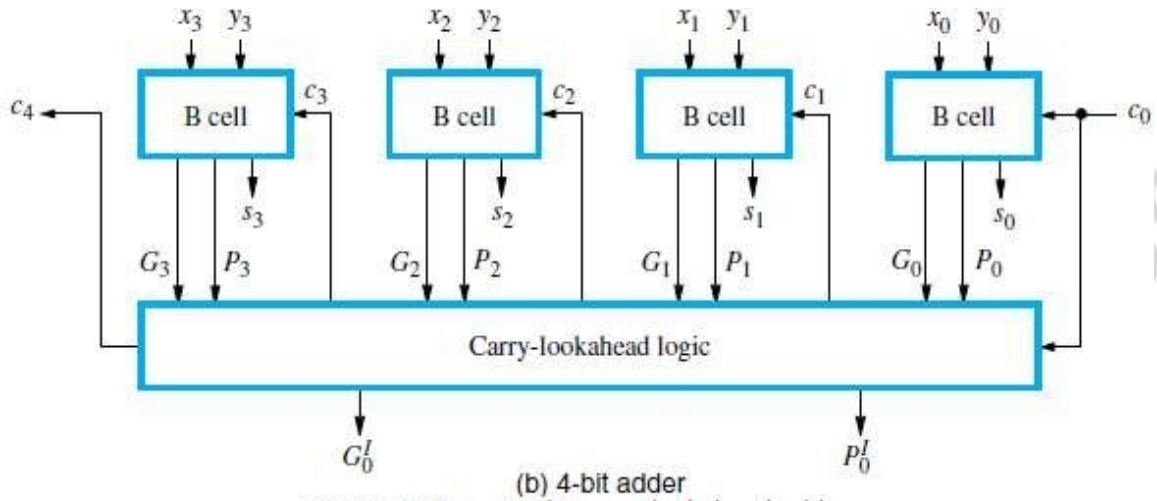
  $c4=G3+P3G2+P3P2G1+P3P2P1G0+P3P2P1P0c0$
- Expanding c_i terms of i-1 subscripted variables and substituting into the c_{i+1} expression, we obtain $C_{i+1}=G_i+P_iG_{i-1}+P_iP_{i-1}G_{i-2}.\ldots.+P_1G_0+P_iP_{i-1}\ldots P_0c_0$

Conclusion: Delay through the adder is 3 gate delays for all carry-bits & 4 gate delays for all sum-bits.

- The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a **Carry-Lookahead Adder**.
- Limitation: If we try to extend the carry-lookahead adder for longer operands, we run into a problem of gate fan-in constraints.



(a) Bit-stage cell



(b) 4-bit adder

Figure 9.4 A 4-bit carry-lookahead adder.

HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

- 16-bit adder can be built from four 4-bit adder blocks, as shown below.
- These blocks provide new output functions defined as $G_k{}^I$ and $P_k{}^I$, where k=0 for the first 4-bit block, k=1 for the second 4-bit block and so on.

- In the first block,

$P_0{}^I$=P3P2P1P0

& $G_0{}^I$=G3+P3G2+P3P2G1+P3P2P1G0

- The first-level Gi and Pi functions determine whether bit stage i generates or propagates a carry, and the second level Gk and Pk functions determine whether block k generates or propagates a carry.
- Carry c16 is formed by one of the carry-lookahead circuits as c16=G3+P3G2+P3P2G1+P3P2P1G0+P3P2P1P0c0
- Conclusion: All carries are available 5 gate delays after X, Y and c0 are applied as inputs.

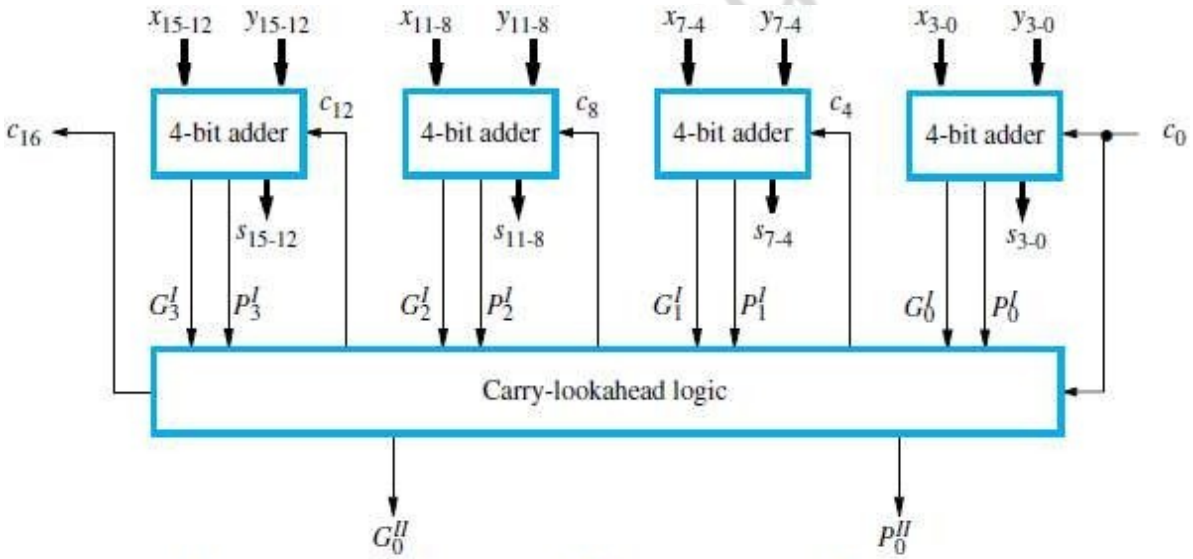


Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

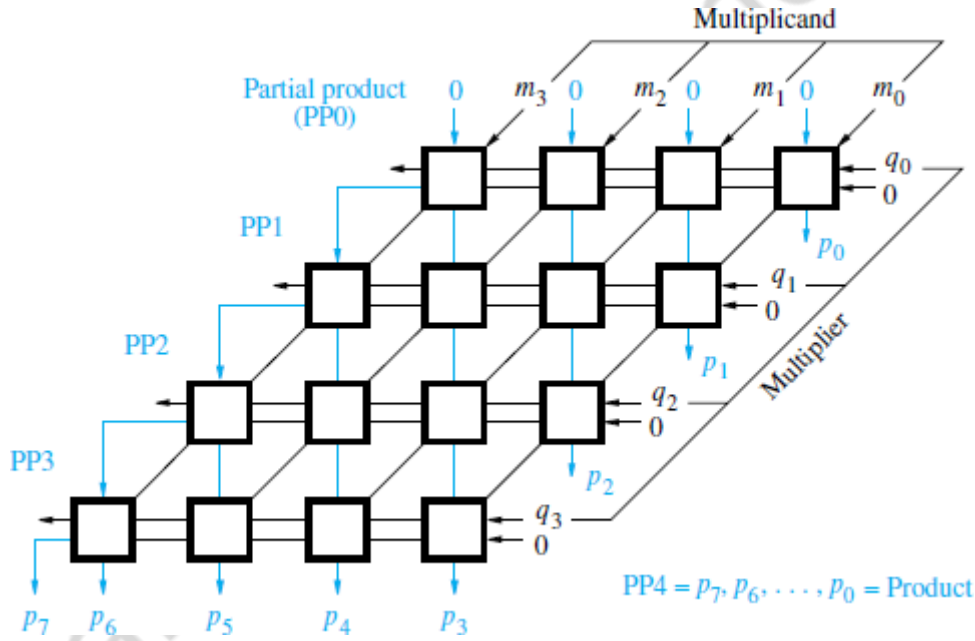
MULTIPLICATION OF POSITIVE NUMBERS

The product of two n – bit numbers is 2n digits, ie. the product of two 4 – bit numbers is 8 – bits.

```
                1 1 0 1     (13) Multiplicand M
              × 1 0 1 1     (11) Multiplier Q
              -------
                1 1 0 1
              1 1 0 1
            0 0 0 0
          1 1 0 1
        -----------
1 0 0 0 1 1 1 1     (143) Product P
```

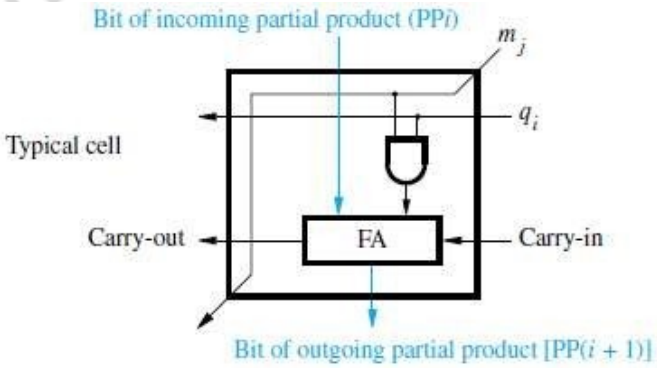
(a) Manual multiplication algorithm

The above method can be implemented as shown below, here m0, m1,m2,m3 are the multiplicands, q0,q1,q2,q3 are the multipliers, PP0,PP1,PP2,PP3 are the partial products and



p0,p1,p2,p3,p4,p5,p6,p7 are the partial products.

The square box represents a single cell that implements partial product for one bit as sown:



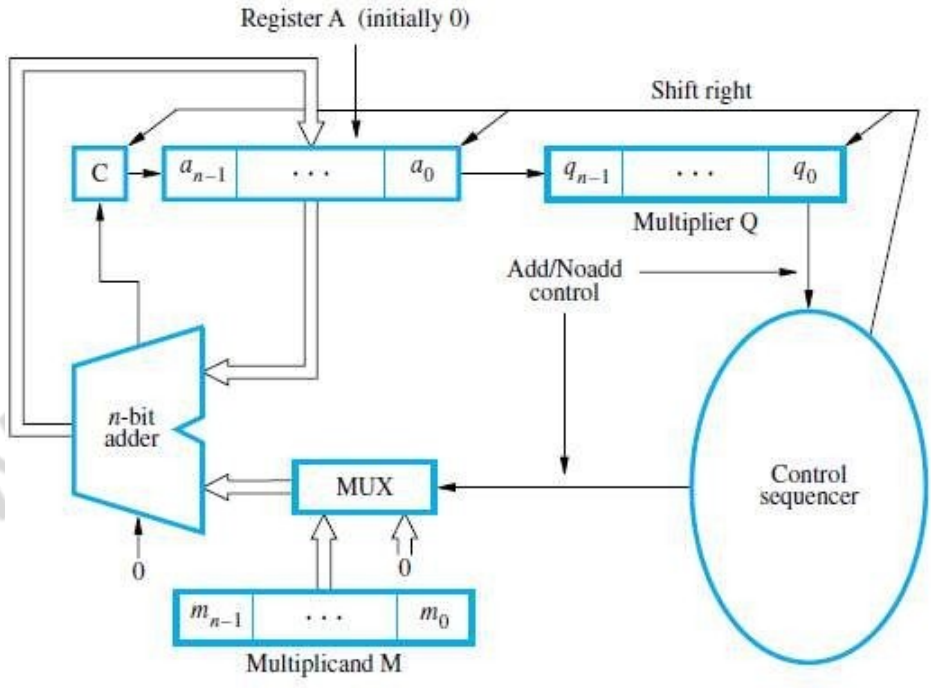
(b) Array implementation

Figure 9.6 Array multiplication of unsigned binary operands.

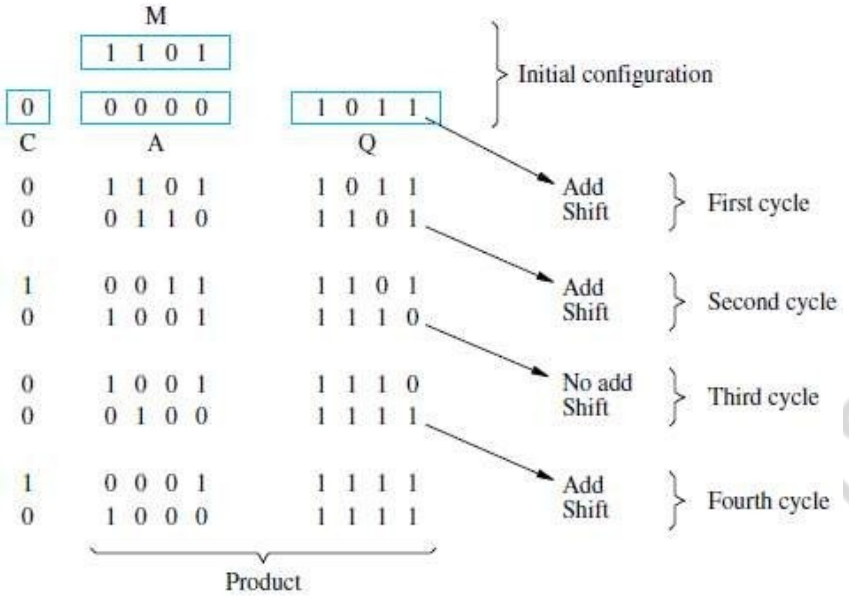
• The main component in each cell is a full adder(FA)..
• The AND gate in each cell determines whether a multiplicand bit mj, is added to the incoming partial- product bit, based on the value of the multiplier bit qi (Figure 9.6).

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- The simplest method to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps.
  • Registers A and Q combined hold PPi(partial product)
  while the multiplier bit qi generates the signal Add/Noadd.
  • The carry-out from the adder is stored in flip-flop C (Figure 9.7).
  • Procedure for multiplication:
  1) Multiplier is loaded into register Q, Multiplicand is loaded into register M and C & A are cleared to 0.
  2) If q0=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position. If q0=0, no addition performed and C, A & Q are shifted right one bit-position.
  3) Repeat step 2 for n cycles (where n is the number of bits in operand), the high-order half of the product is held in register A and the low-order half is held in register Q.



(a) Register configuration



(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

SIGNED OPERAND MULTIPLICATION BOOTH ALGORITHM

• This algorithm
→ generates a 2n-bit product
→ treats both positive & negative 2's-complement n-bit operands uniformly.
→ reduces the number of partial products, when there is continuous number of 1's.

• Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.
• Multiplication of 45 and 30 in normal and Booth algorithm method-

The multiplier is recoded using the below table -

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	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

While recoding the multiplier, assume '0' to the right of LSB.

Eg: if multiplier is 0011110, then for recoding, put extra '0' to the right of LSB.
Ie it becomes, 00111100, now recode as per the above table.
The recoded multiplier is, 0 +1 0 0 0 -1 0

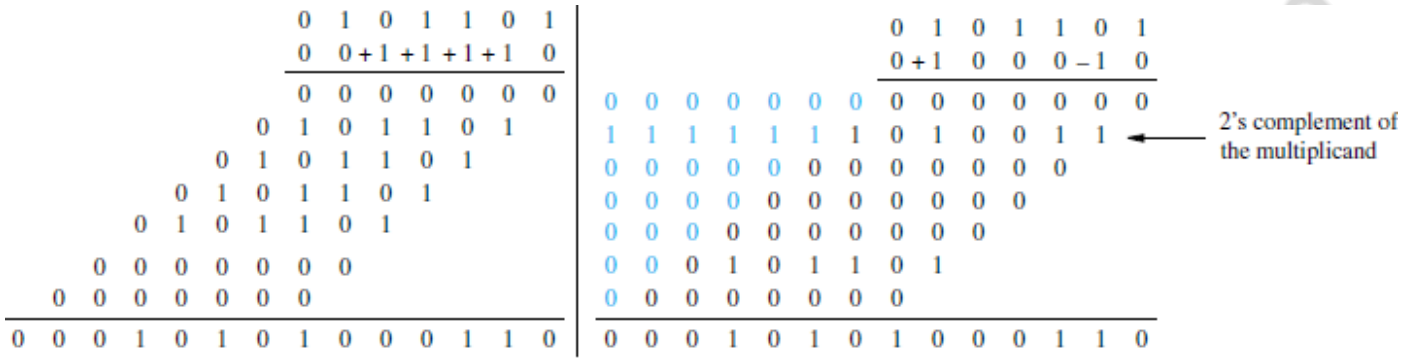


Figure 9.9 Normal and Booth multiplication schemes.

Figure 9.10 Booth recoding of a multiplier.

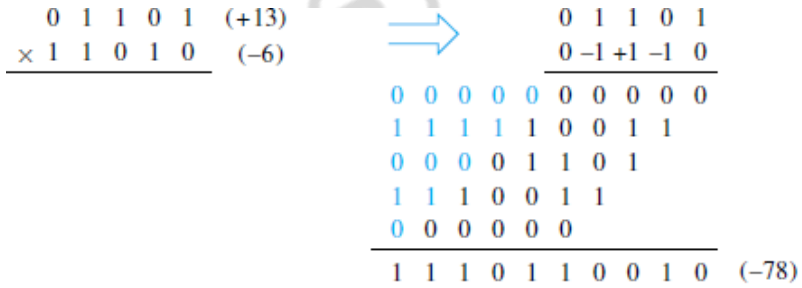
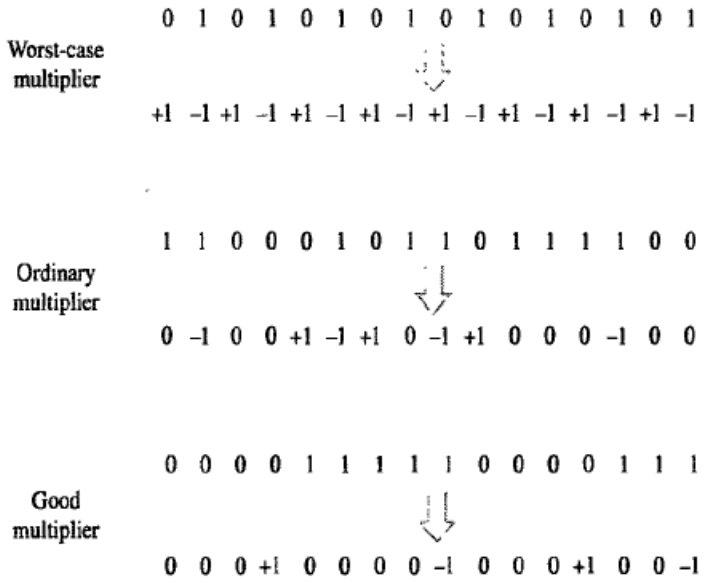


Figure 9.11 Booth multiplication with a negative multiplier.

When multiplier is ,
+1 – do the usual multiplication, ie. partial product is the multiplicand.
-1 – partial product is the 2's compliment of the multiplicand.

Note: Sign extension to be done, for the partial products.



FAST MULTIPLICATION

Two techniques to speed up the multiplication process –

1. Bit-Pair Recoding Of Multipliers – reduces the maximum number of summands (partial products) to n/2 for n-bit multiplier.
2. Carry-Save addition of summands – reduces the time needed to add the summands.

BIT-PAIR RECODING OF MULTIPLIERS

- This method

→ derived from the booth algorithm

→ reduces the number of summands by a factor of 2

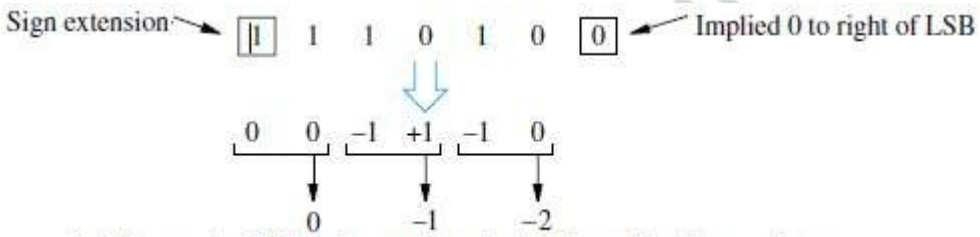
- Group the Booth-recoded multiplier bits in pairs. Suppose into [i j]
- Then the bit-pair recoded multiplier is obtained by (2*i +j)
- The pair (+1 -1) is equivalent to the pair (0 +1).

Direct recoding from multiplier to Bit-pair bits is done by using the below table -

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

(b) Table of multiplicand selection decisions

Figure 9.14 Multiplier bit-pair recoding.



(a) Example of bit-pair recoding derived from Booth recoding

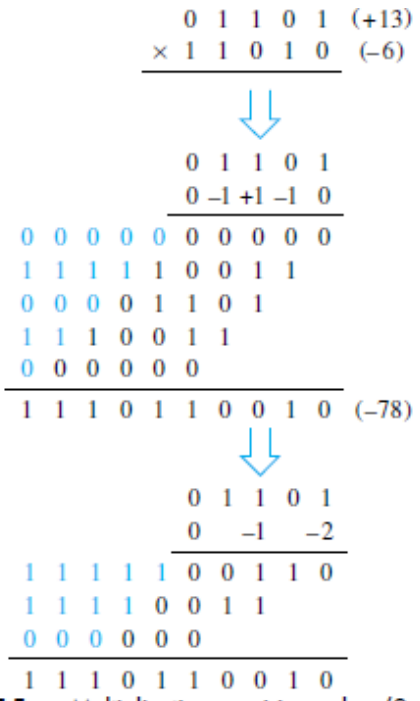


Figure 9.15 Multiplication requiring only $n/2$ summands.

When multiplier is,

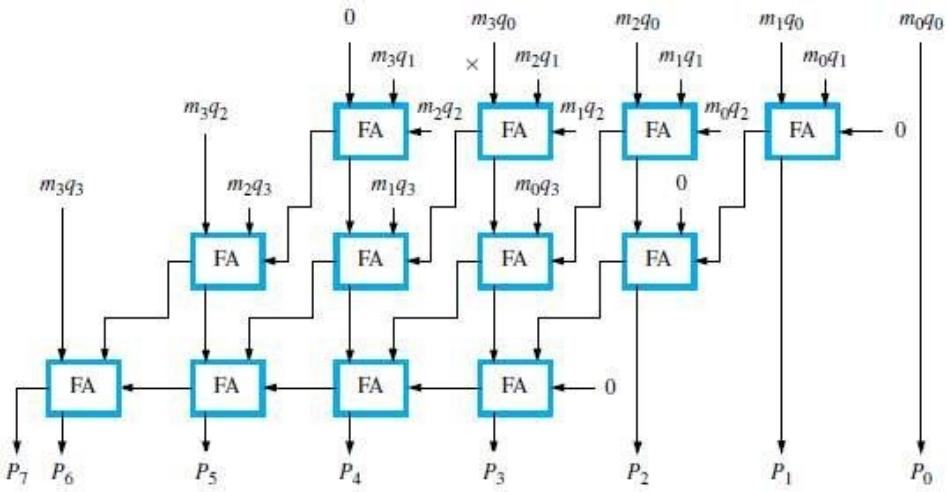
- -2 – put '0' as LSB & then 2's complement of multiplicand.
- -1 – 2's complement of multiplicand.
- +1 – only multiplicand (usual).
- +2 – put '0' as LSB & then multiplicand.

Note: Sign extension to be done, for the partial products.

CARRY-SAVE ADDITION OF SUMMANDS

• Consider the array for 4*4 multiplication.

• Instead of letting the carries ripple along the rows, they are "saved" and introduced into the next row, at the correct weighted positions. Thus reduces the number of summands and speeds up the addition process.



(b) Carry-save array

Figure 9.16 carry-save arrays for a 4 × 4 multiplier.

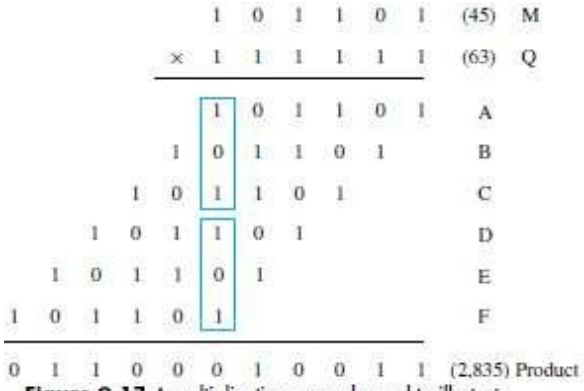


Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

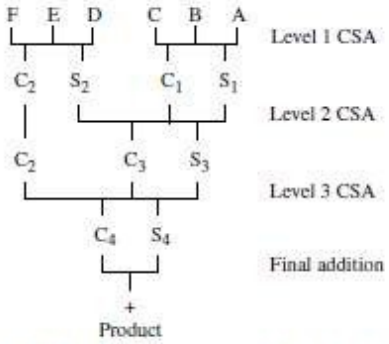


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

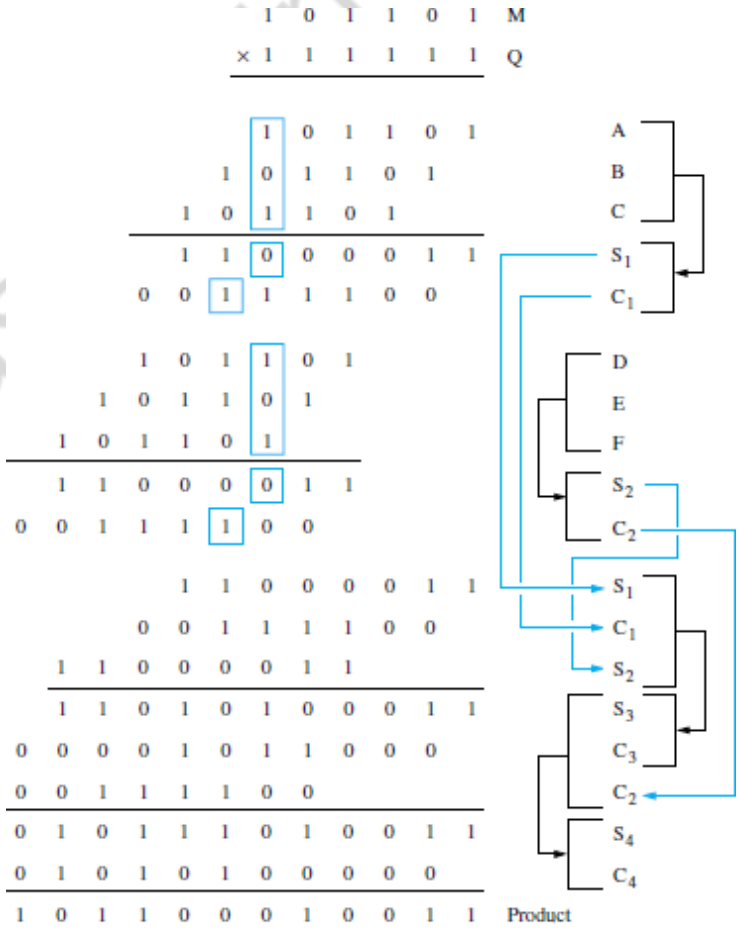


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

- The full adder is input with three partial bit products in the first row.
- Multiplication requires the addition of several summands.
- Carry- Save Addition (CSA) speeds up the addition process.

- Consider the array for 4x4 multiplication shown in fig 9.16.

- First row consisting of just the AND gates that implement the bit products m3q0, m2q0, m1q0 and m0q0.
  - The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
  - Consider the addition of many summands in fig 9.18.
  - Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
  - Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
  - Continue with this process until there are only two vectors remaining
  - The final 2 vectors, are added in a RCA (Ripple Carry Adder) or CLA (Carry Look-ahead Adder) to produce the desired product.
  - When the number of summands is large, the time saved is proportionally much greater.
  - Delay: AND gate + 2 gate/CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay, array 6x6 require 6(n-1)-1 = 29 gate Delay.
- In general, CSA takes 1.7 $log2$k-1.7 levels of CSA to reduce k summands.

INTEGER DIVISION

- An n-bit positive-divisor is loaded into register M.

An n-bit positive-dividend is loaded into register Q at the start of the operation.
Register A is set to 0 (Figure 9.21).

- After division operation, the n-bit quotient is in register Q, and the remainder is in register A.

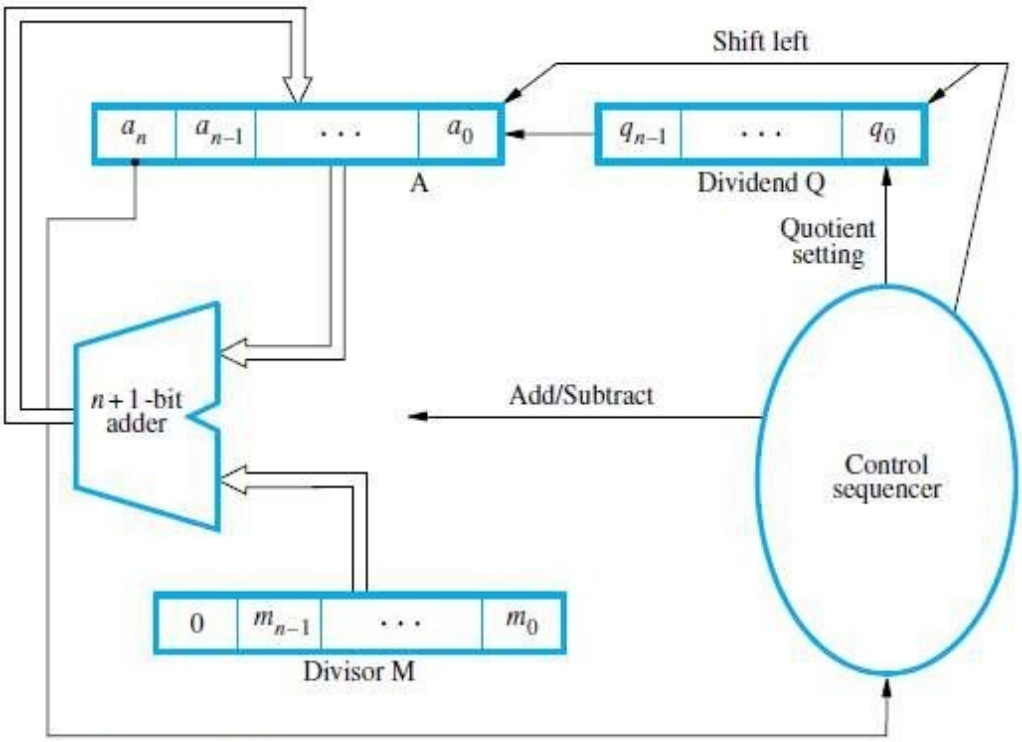


Figure 9.23 Circuit arrangement for binary division.

```
      21                    10101
13 ) 274          1101 ) 100010010
     26                     1101
     14                     10000
     13                      1101
      1                      1110
                             1101
                                1
```

Figure 9.22 Longhand division examples.

RESTORING DIVISION

The above logic circuit arrangement implements non-restoring and restoring division. The non-restoring division algorithm is as follows –

Step 1: Initialize M – Divisor, Q – Dividend (n bits), A with 0s
Step 2: Repeat step 3 to step 5 - *n* times
Step 3: Shift A and Q left, by one binary position
Step 4: Subtract M from A, (A=A-M)
Step 5: If sign of A is 1, set q_0 to 0 and add M back to A (restore A);
Elseif sign of A is 0, set q_0 to 1
Step 6: Finally n bit Quotient is present in register Q and Remainder is present in register A.

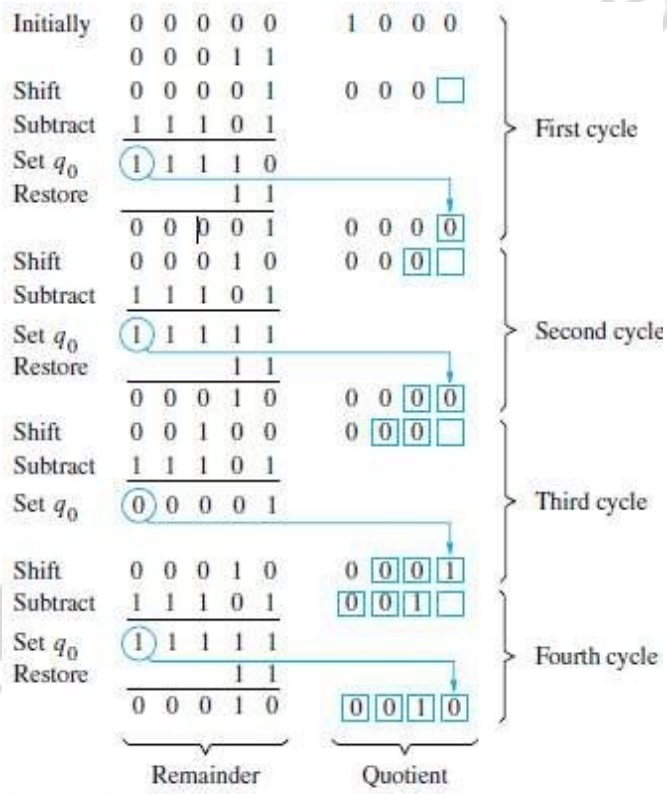


Figure 9.24 A restoring division example.

NON - RESTORING DIVISION

The above logic circuit arrangement implements non-restoring and restoring division

Step 1: Initialize M – Divisor, Q – Dividend (n bits), A with 0s

Step 2: Repeat step 3 - *n* times

Step 3: If sign of A is 0, shift A and Q left by 1 bit and subtract M from A, accordingly set q0 bit;
Elseif sign of A is 1, shift A and Q left and add M to A, accordingly set q0 bit

Step 4: After n cycles, if sign of A is 1, add M to A.

Step 5: Finally Quotient is present in register Q and Remainder is present in register A.

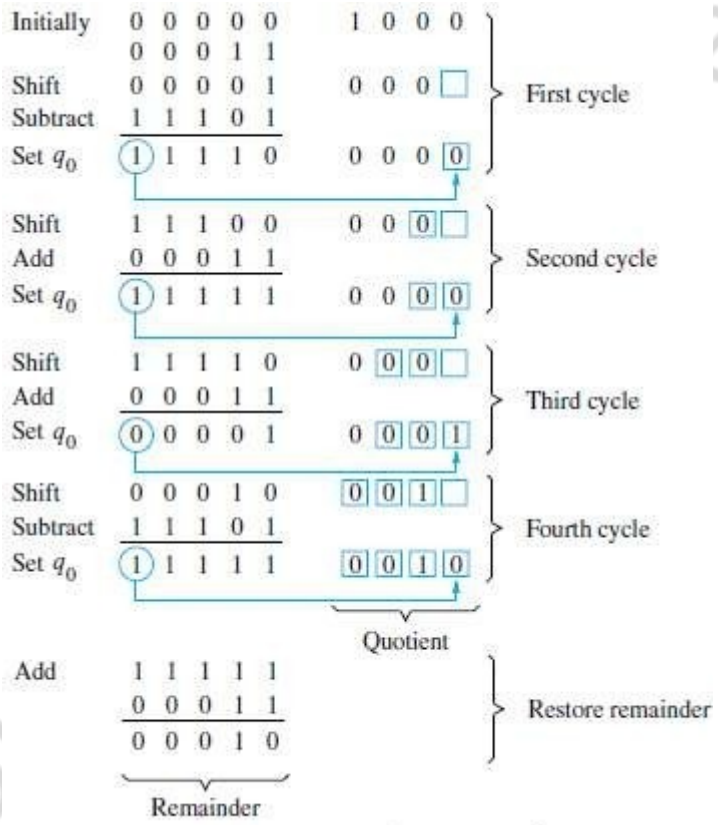


Figure 9.25 A non-restoring division example.

Module - 4

ARITHMETIC

1. Explain the design of a 4-bit carry-look ahead adder?
2. Design a logic circuit to perform addition/subtraction of two 'n' numbers X and Y?
3. Given A=10101 and B=00100, perform A/B using restoring division algorithm?
4. Perform signed multiplication of numbers (-12) and (-11) using Booth's algorithm?
5. Design 4 bit carry look ahead logic and explain how it is faster them 4 bit ripple adder?
6. Problems on Booth algorithm, Bit-pair recoding, restoring division and non-restoring division.
7. Perform following operations on the 5-bit signed numbers using 2's complement representation system. Also indicate whether overflow has occurred? i) (-10)+(-13) ii)(-10)-(+4) iii)(-3)+(-8) iv) (-10)-(+7)
8. Explain the circuit arrangement for binary division.
9. Explain the 16 bit carry look ahead adder using 4 – bit adder. Also unite the expression for Ci+1.
10. Explain the concept of carry save addition for multiplication operation M xQ =P for 4-bit operands with diagram and suitable example.
11. Explain Generate and Propagate functions in carry look ahead adder.
12. Explain the design of a 4-bit carry look-ahead adder.
13. Design a logic circuit to perform addition/subtraction of two n bit numbers X and Y.
14. Design a 'n' bit carry propagation adder circuit to add 'k' n bit numbers.

Problem 1:

Represent the decimal values 5, -2, 14, -10, 26, -19, 51 and -43 as signed 7-bit numbers in the following binary formats:

(a) sign-and-magnitude
(b) 1's-complement
(c) 2's-complement

Solution:

The three binary representations are given as:

Decimal values	Sign-and-magnitude representation	1's-complement representation	2's-complement representation
5	0000101	0000101	0000101
−2	1000010	1111101	1111110
14	0001110	0001110	0001110
−10	1001010	1110101	1110110
26	0011010	0011010	0011010
−19	1010011	1101100	1101101
51	0110011	0110011	0110011
−43	1101011	1010100	1010101

Problem 2:

(a) Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case.

a) 5 and 10 b) 7 and 13
c) –14 and 11 d) –5 and 7
e) –3 and –8

(b) Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

Solution:

(a)

(a)	(b)	(c)
00101 + 01010	00111 + 01101	10010 + 01011
01111	10100	11101
no overflow	overflow	no overflow

(d)	(e)	(f)
11011 + 00111	11101 + 11000	10110 + 10011
00010	10101	01001
no overflow	no overflow	overflow

(b) To subtract the second number, form its 2's-complement and add it to the first number.

(a)	00101	(b)	00111	(c)	10010
	+ 10110		+ 10011		+ 10101
	11011		11010		00111
	no overflow		no overflow		overflow
(d)	11011	(e)	11101	(f)	10110
	+ 11001		+ 01000		+ 01101
	10100		00101		00011
	no overflow		no overflow		no overflow

Problem 3:

Perform following operations on the 6-bit signed numbers using 2's complement representation system. Also indicate whether overflow has occurred.

010110	101011	111111
+001001	+100101	+000111
011001	110111	010101
+010000	+111001	+101011
010110	111110	100001
−011111	−100101	−011101
111111	000111	011010
−000111	−111000	−100010

Solution:

010110	(+22)	101011	(−21)	111111	(−1)
+ 001001	+ (+9)	+ 100101	+ (−27)	+ 000111	+ (+7)
011111	(+31)	010000	(−48)	000110	(+6)
		overflow			
011001	(+25)	110111	(−9)	010101	(+21)
+ 010000	+ (+16)	+ 111001	+ (−7)	+ 101011	+ (−21)
101001	(+41)	110000	(−16)	000000	(0)
overflow					

010110	(+22)	010110
− 011111	− (+31)	+ 100001
	(−9)	110111
111110	(−2)	111110
− 100101	− (−27)	+ 011011
	(+25)	011001
100001	(−31)	100001
− 011101	− (+29)	+ 100011
	(−60)	000100
		overflow
111111	(−1)	111111
− 000111	− (+7)	+ 111001
	(−8)	111000
000111	(+7)	000111
− 111000	− (−8)	+ 001000
	(+15)	001111
011010	(+26)	011010
− 100010	− (−30)	+ 011110
	(+56)	111000
		overflow

Problem 4:

Perform signed multiplication of following 2's complement numbers using Booth's algorithm. (a) A=010111 and B=110110 (b) A=110011 and B=101100

(c) A=110101 and B=011011 (d) A=001111 and B=001111 (e) A=10100 and B=10101

(f) A=01110 and B=11000

Solution:

010111 ×110110 = -230 (+23 × -10)

0 1 0 1 1 1 × 0 -1+1 0 -1 0

0

sign extension:

1 1 1 1 1 1 0 1 0 0 1

0

0 0 0 0 1 0 1 1 1

1 1 1 1 0 2 1 1 0 1 0 1 1

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

110011 ×101100 = 260 (-13 × -20)

1 1 0 0 1 1 × -1+1 0 -1 0 0

0

0

sign extension:

0 0 0 0 0 0 1 1 0 1

0

1 1 1 1 0 0 1 1

0 1 0 1 0 1 1 1 2 1 1

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

110101 ×011011 = -297 (-11 × 27)

1 1 0 1 0 1 × +1 0 -1+1 0 -1

0

sign extension:

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

0

1 1 1 1 1 1 0 1 0 1

0 0 0 0 0 1 0 1 1

0

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

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

001111 ×001111 = 225 (15 × 15)

0 0 1 1 1 1 × 0+1 0 0 0 -1

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

0 0 0 0 1 1 1 1

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

10100(-12) ×10101(-11) →

1 0 1 0 0

-1 1 -1 1 -1 (recoded multiplier)

0 0 0 0 0 1 1 0 0

0 -1 1 -1 0

01110(+14) ×11000(-8) →

0 -1 0 0 0 (recoded multiplier)

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

1 1 1 0 0 1 0

0 0 0 0 0 0

1 1 1 0 0 1 0 0 0 0 (-112)

Problem 5:

Perform signed multiplication of following 2's complement numbers using bit-pair recoding method.

(a) A=010111 and B=110110 (b) A=110011 and B=101100

(c) A=110101 and B=011011 (d) A=001111 and B=001111

Solution:

```
  010111                          0 1 0 1 1 1
× 110110                            -1  +2  -2
                    ┌─────────────────────────
  1 1 1 1 1 1 0 1 0 0 1 0
  0 0 0 0 1 0 1 1 1 0
  1 1 1 1 0 2 1 1 0 1 0 1
  ─────────────────────────
  1 1 1 1 0 0 0 1 1 0 1 0
```

```
  110011                          1 1 0 0 1 1
× 101100                            -1  -1   0
                                            0
                    ┌─────────────────────────
  0 0 0 0 0 0 1 1 0 1
  0 0 0 0 1 1 1 0 1 1
  ─────────────────────────
  0 0 0 1 0 0 0 0 0 1 0 0
```

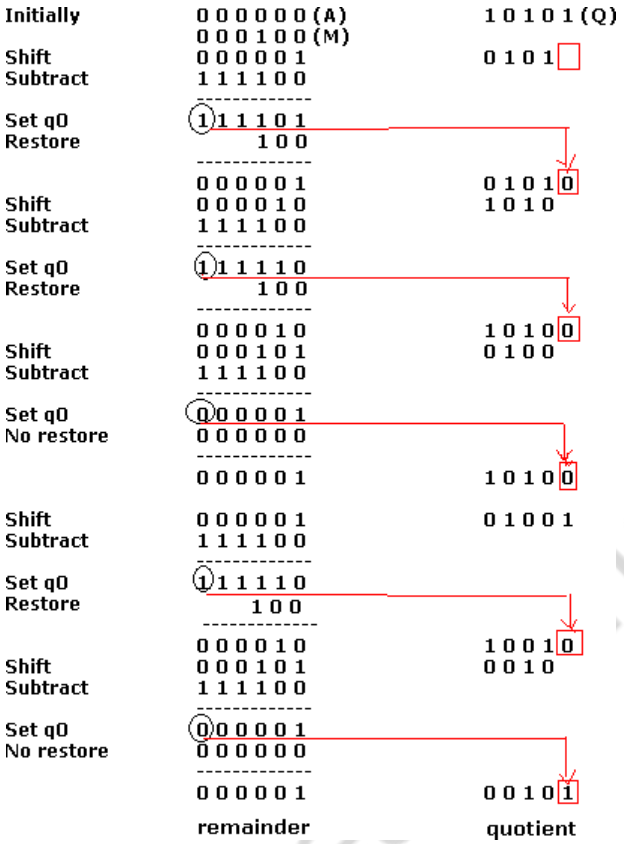
```
  110101                          1 1 0 1 0 1
× 011011                            +2  -1  -1
                    ┌─────────────────────────
  0 0 0 0 0 0 0 0 1 0 1 1
  0 0 0 0 0 0 1 0 1 1
  1 1 1 0 1 0 1 1
  ─────────────────────────
  1 1 1 0 1 1 0 1 0 1 1 1
```

```
  001111                          0 0 1 1 1 1
× 001111                            +1      -1
                    ┌─────────────────────────
  1 1 1 1 1 1 1 1 0 0 0 1
  0 0 0 0 1 1 1 1
  ─────────────────────────
  0 0 0 0 1 1 1 0 0 0 0 1
```

Problem 6:

Given A=10101 and B=00100, perform A/B using restoring division algorithm.

Solution:



Problem 7:

Given A=10101 and B=00101, perform A/B using non-restoring division algorithm.

Solution: