MODULE 4: ARITHMETIC

NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

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.
• 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 other words, the operation of forming the 1's complement of a given number is equivalent to subtracting that number from 2^n-1).

• In **2's complement system**,
  forming the 2's complement of a number is done by subtracting that number from 2^n.
  For ex, -5 is obtained by complementing each bit in 0101 & then adding 1 to yield 1011.

(In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).

• 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

• Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system (Figure 1.6).

Rule 1:

- **To Add** two numbers, add their n-bits and ignore the carry-out signal from the MSB position.
- Result will be algebraically 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 will be algebraically 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**.

• In 1's complement representation, the result obtained after an addition operation is not always correct. The carry-out(c_n) cannot be ignored. If c_n=0, the result obtained is correct. If c_n=1, then a 1 must be added to the result to make it correct.

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.

2) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.

(a)	0010	(+2)		(b)	0100	(+4)	
	+ 0011	(+3)			+ 1010	(−6)	
	0101	(+5)			1110	(−2)	
(c)	1011	(−5)		(d)	0111	(+7)	
	+ 1110	(−2)			+ 1101	(−3)	
	1001	(−7)			0100	(+4)	
(e)	1101	(−3)			1101		
	− 1001	(−7)	⇒		+ 0111		
					0100	(+4)	
(f)	0010	(+2)			0010		
	− 0100	(+4)	⇒		+ 1100		
					1110	(−2)	
(g)	0110	(+6)			0110		
	− 0011	(+3)	⇒		+ 1101		
					0011	(+3)	
(h)	1001	(−7)			1001		
	− 1011	(−5)	⇒		+ 0101		
					1110	(−2)	
(i)	1001	(−7)			1001		
	− 0001	(+1)	⇒		+ 1111		
					1000	(−8)	
(j)	0010	(+2)			0010		
	− 1101	(−3)	⇒		+ 0011		
					0101	(+5)	

Figure 1.6 2's-complement Add and Subtract operations.

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.

• Since carries must propagate (or ripple) through cascade, the configuration is called an n-bit ripple carry adder (Figure 9.1).

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

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

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

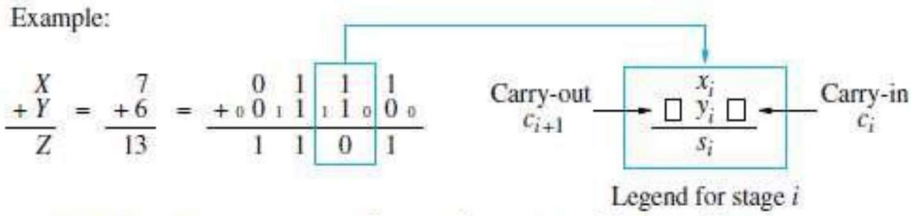
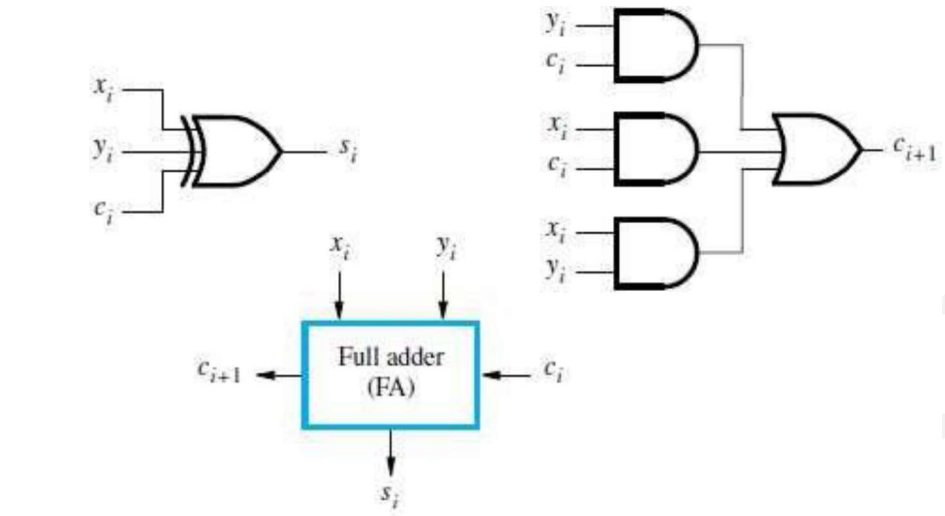
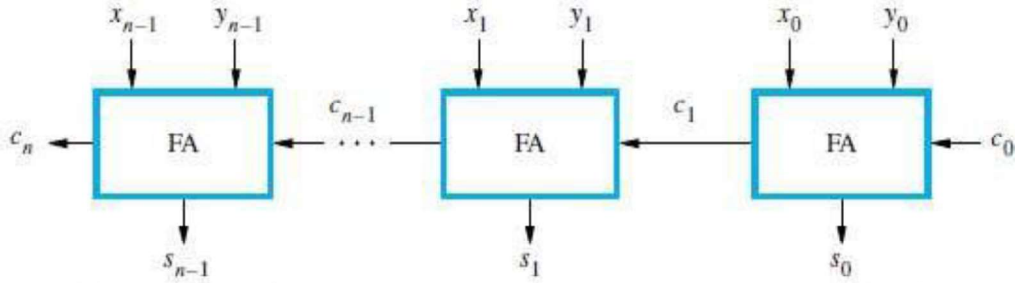


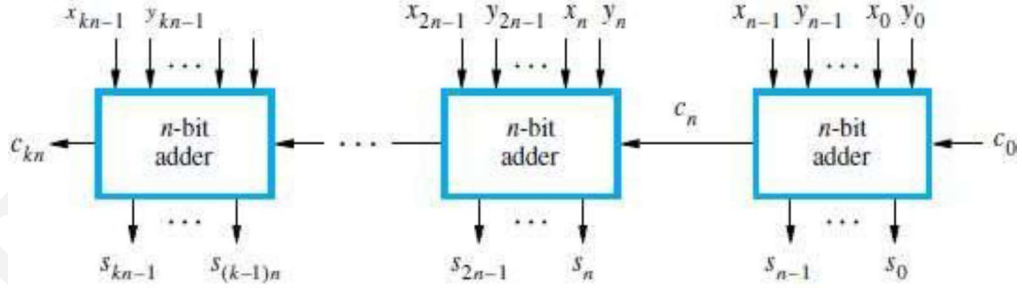
Figure 9.1 Logic specification for a stage of binary addition.



(a) Logic for a single stage



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



(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 (Figure 9.3).
- **Overflow** can only occur when the signs of the 2 operands are the same.
- 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, applying the Y vector unchanged to one of the adder inputs.
  Control-line=1 for subtraction, the Y vector is 2's complemented.

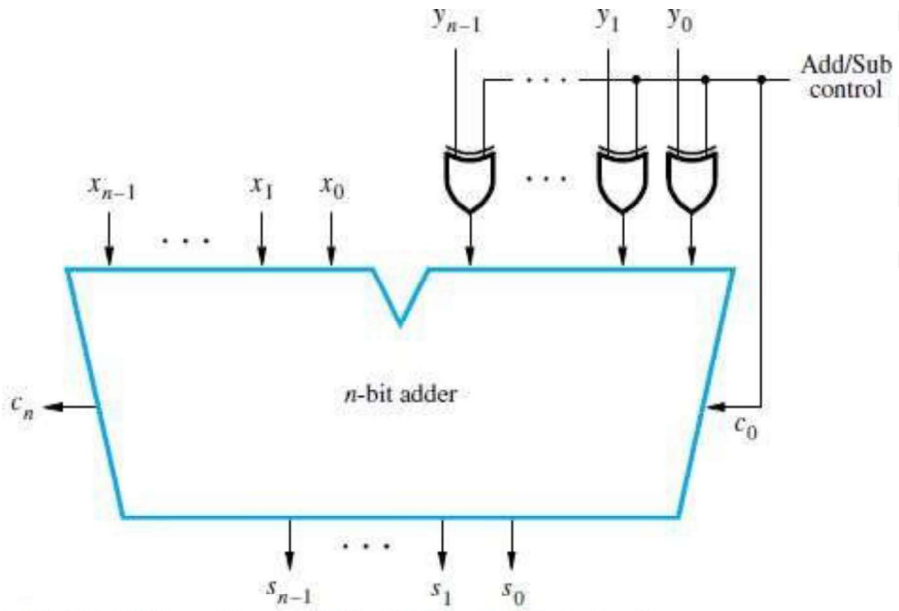


Figure 9.3 Binary addition/subtraction logic circuit.

DESIGN OF FAST ADDERS

- **Drawback of ripple carry adder:** 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.
  2) Use an augmented logic-gate network structure.

CARRY-LOOKAHEAD ADDITIONS

- The logic expression for s_i(sum) and c_{i+1}(carry-out) of stage i are

  $s_i = x_i + y_i + c_i$ ------(1) $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$---------(2)

- Factoring (2) into

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

  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 (Figure 9.4).
- 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.
- Expanding c_i terms of i-1 subscripted variables and substituting into the c_{i+1} expression, we obtain

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

- Conclusion: Delay through the adder is 3 gate delays for all carry-bits & 4 gate delays for all sum-bits.
- Consider the design of a 4-bit adder. The carries can be implemented as

  $c_1 = G_0 + P_0 C_0$

  $c_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

  $c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

  $c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

- 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.

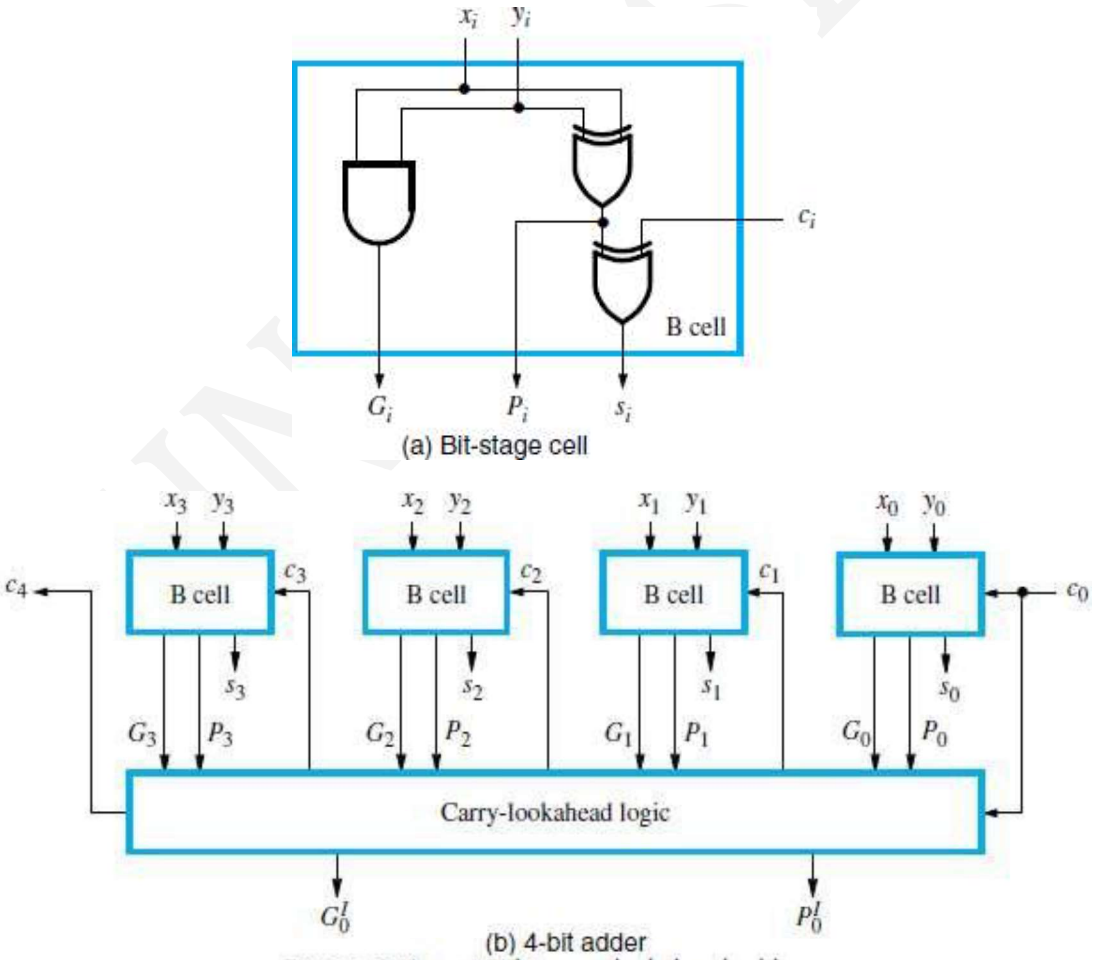


Figure 9.4 A 4-bit carry-lookahead adder.

HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

- 16-bit adder can be built from four 4-bit adder blocks (Figure 9.5).
- These blocks provide new output functions defined as G_k and P_k,
  where k=0 for the first 4-bit block,
  k=1 for the second 4-bit block and so on.
- In the first block,

$$P_0=P_3P_2P_1P_0$$

&

$$G_0=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$$

- The first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry, and the second level G_k and P_k functions determine whether block k generates or propagates a carry.
- Carry c_{16} is formed by one of the carry-lookahead circuits as

$$c_{16}=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$$

- Conclusion: All carries are available 5 gate delays after X, Y and c_0 are applied as inputs.

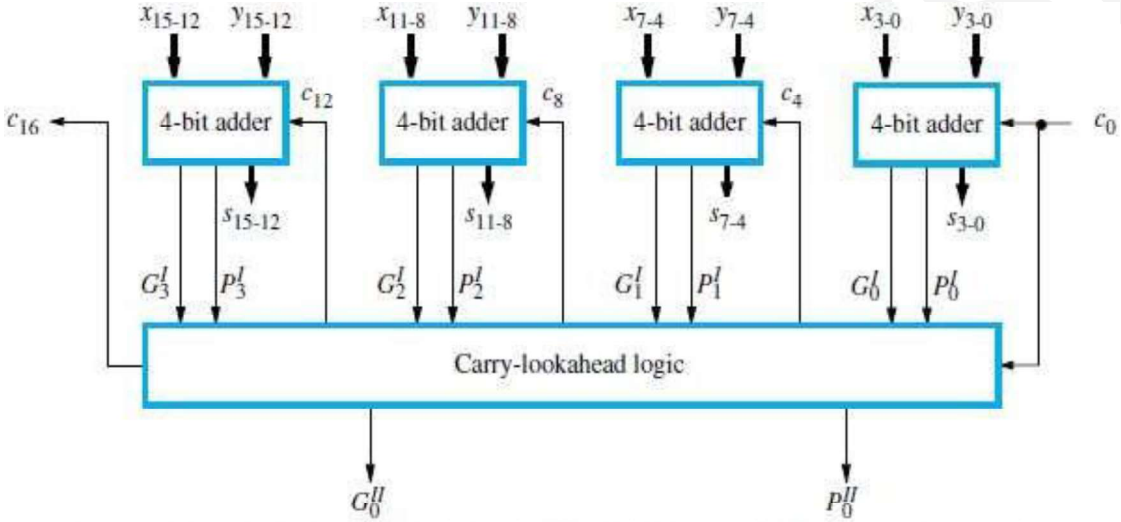
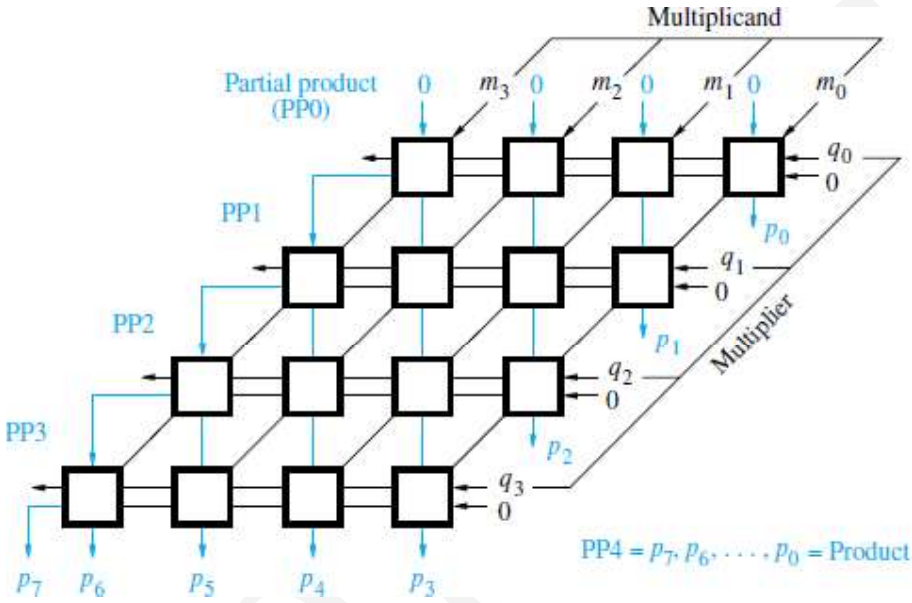


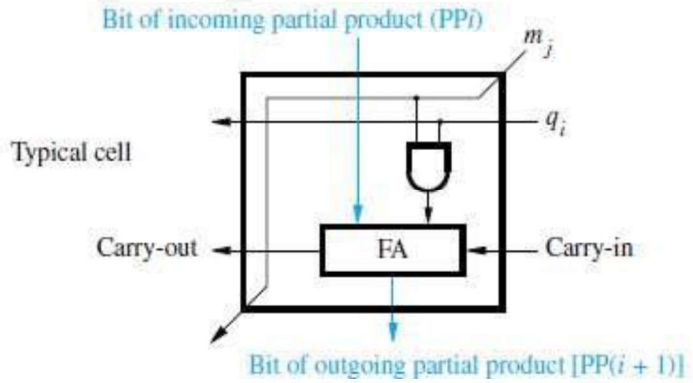
Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

MULTIPLICATION OF POSITIVE NUMBERS

				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





(b) Array implementation

Figure 9.6 Array multiplication of unsigned binary operands.

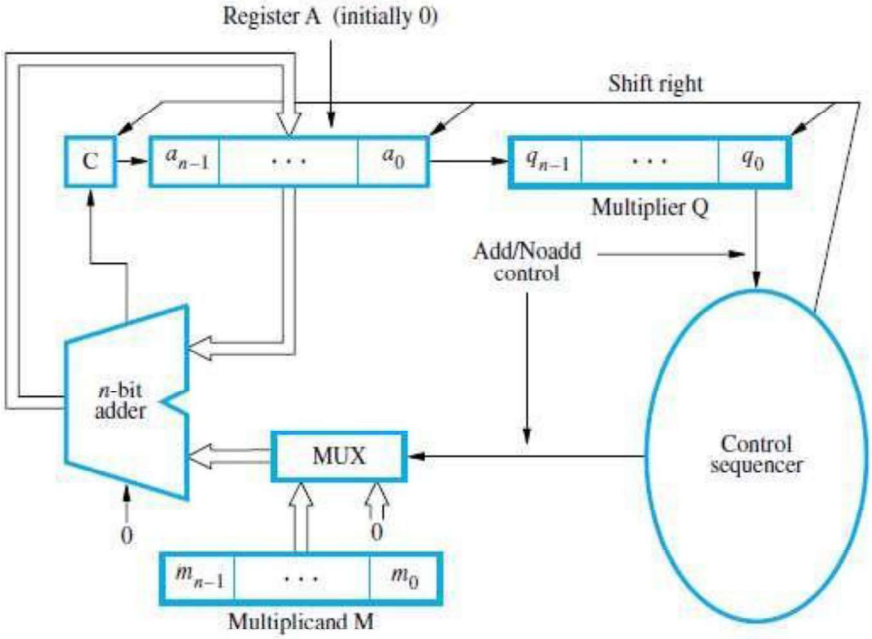
ARRAY MULTIPLICATION

• The main component in each cell is a full adder(FA)..

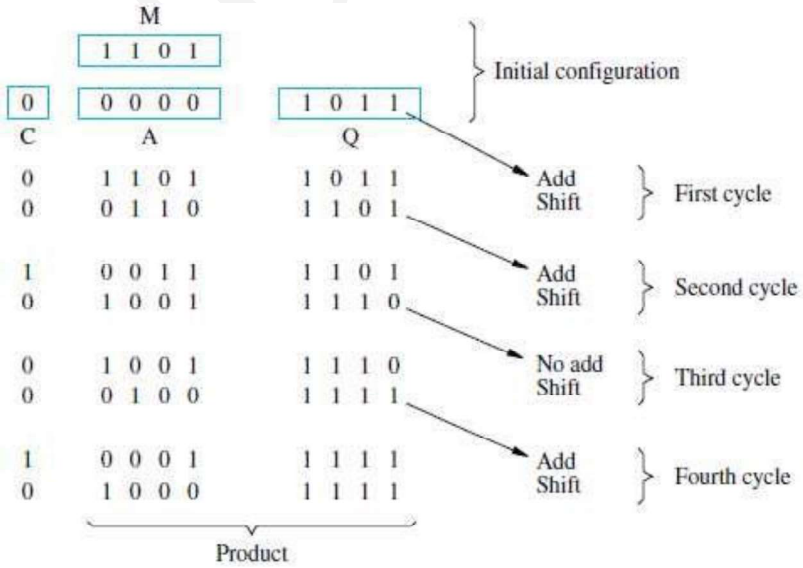
• The AND gate in each cell determines whether a multiplicand bit m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit q_i (Figure 9.6).

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold PP_i(partial product)
  while the multiplier bit q_i 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 q_0=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position.
     If q_0=0, no addition performed and C, A & Q are shifted right one bit-position.
  3) After n cycles, 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(Figure 9.9-9.12).

• Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.

• This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

For e.g. multiplier(Q) 14(001110) can be represented as
010000 (16)
-000010 (2)
001110 (14)

• Therefore, product P=M*Q can be computed by adding 2^4 times the M to the 2's complement of 2^1 times the M.

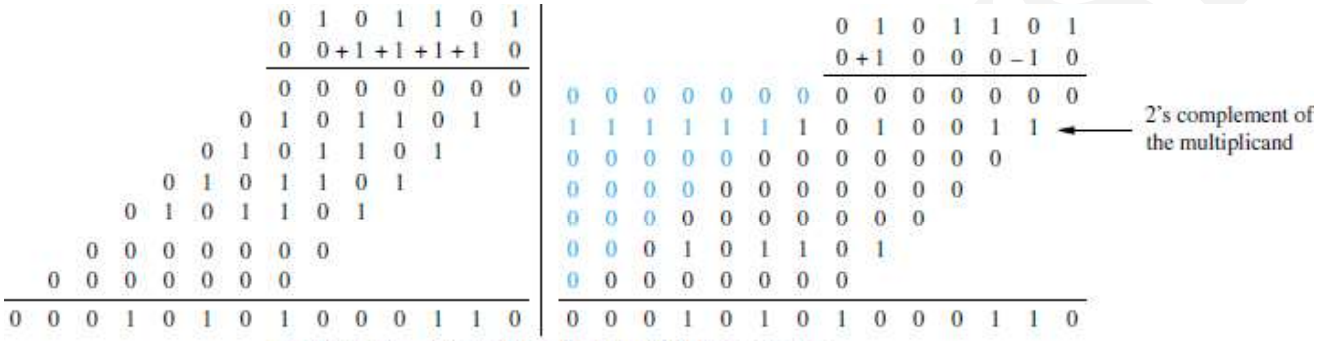


Figure 9.9 Normal and Booth multiplication schemes.

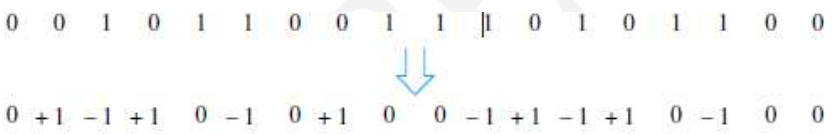


Figure 9.10 Booth recoding of a multiplier.

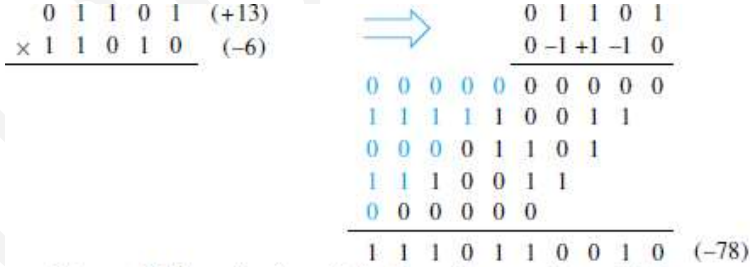


Figure 9.11 Booth multiplication with a negative multiplier.

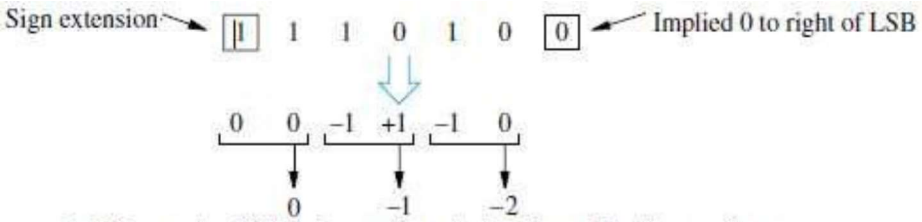
Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

FAST MULTIPLICATION

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. (Figure 9.14 & 9.15).
- The pair (+1 -1) is equivalent to the pair (0 +1).



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

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.

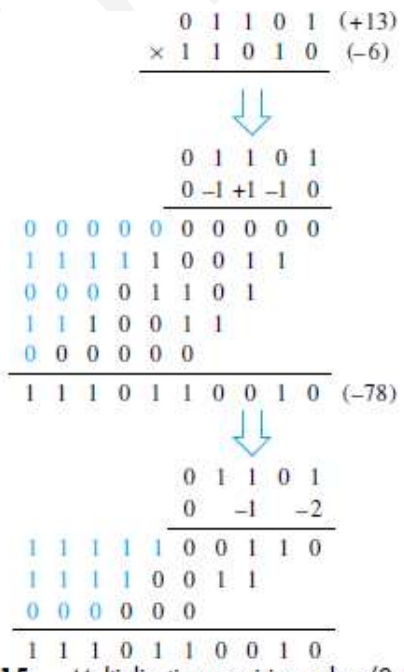
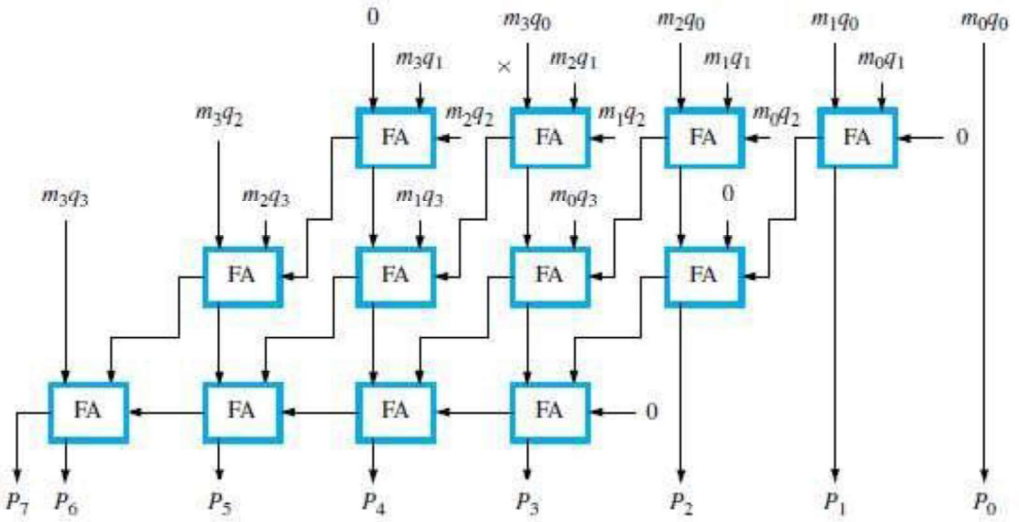


Figure 9.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

• Consider the array for 4*4 multiplication. (Figure 9.16 & 9.18).

• Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.



(b) Carry-save array

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

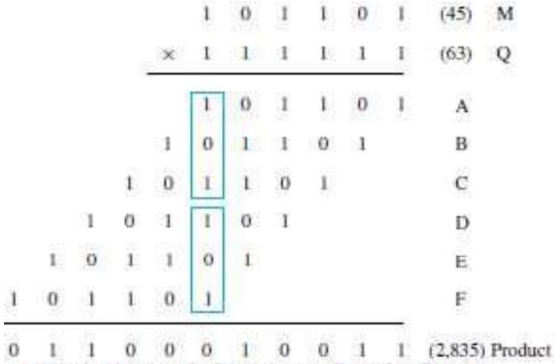


Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

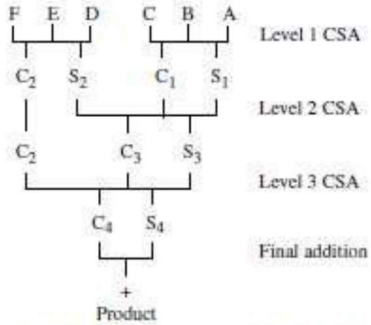


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

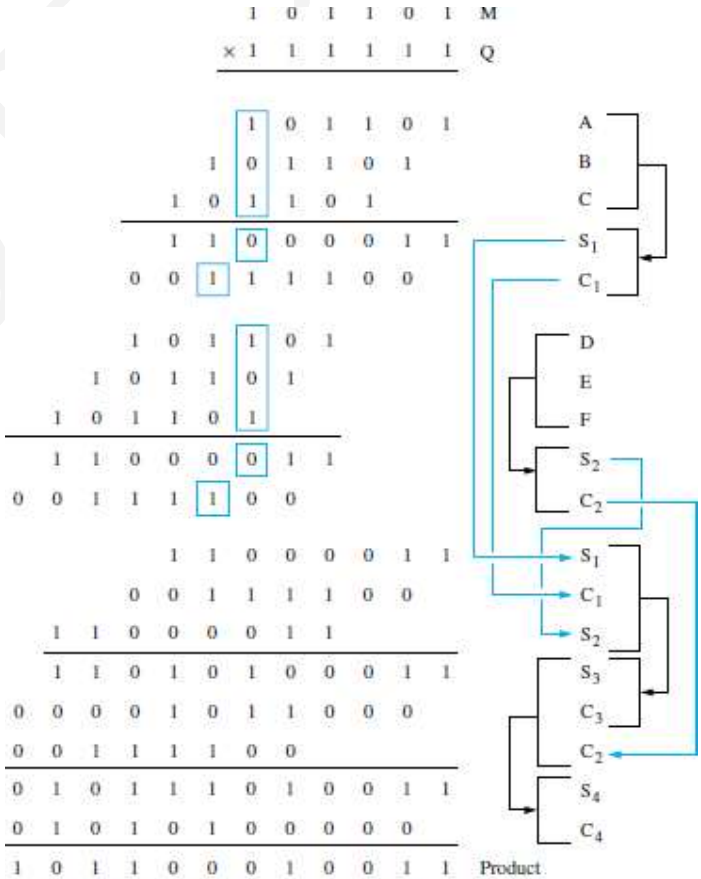


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

• The full adder is input with three partial bit products in the first row.
• Multiplication requires the addition of several summands.
• 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 m_3q_0, m_2q_0, m_1q_0 and m_0q_0.
• 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
• They can be added in a RCA or CLA 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 log_2k-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.

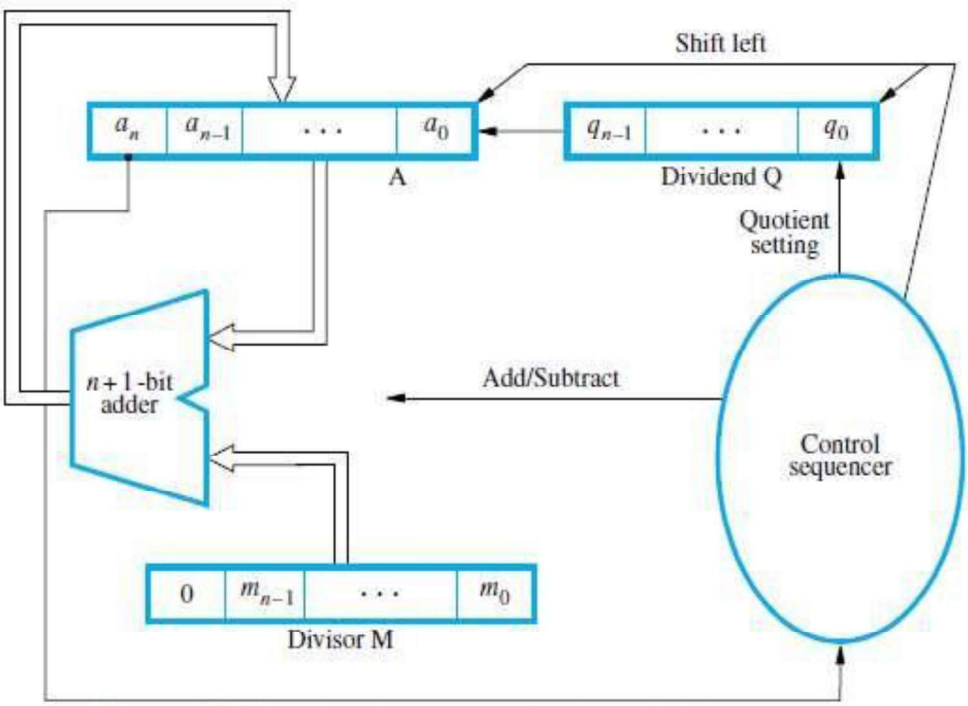


Figure 9.23 Circuit arrangement for binary division.

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

Figure 9.22 Longhand division examples.

NON-RESTORING DIVISION

- Procedure:

Step 1: Do the following n times

i) 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 (Figure 9.23).

ii) Now, if the sign of A is 0, set q_0 to 1; otherwise set q_0 to 0.

Step 2: If the sign of A is 1, add M to A (restore).

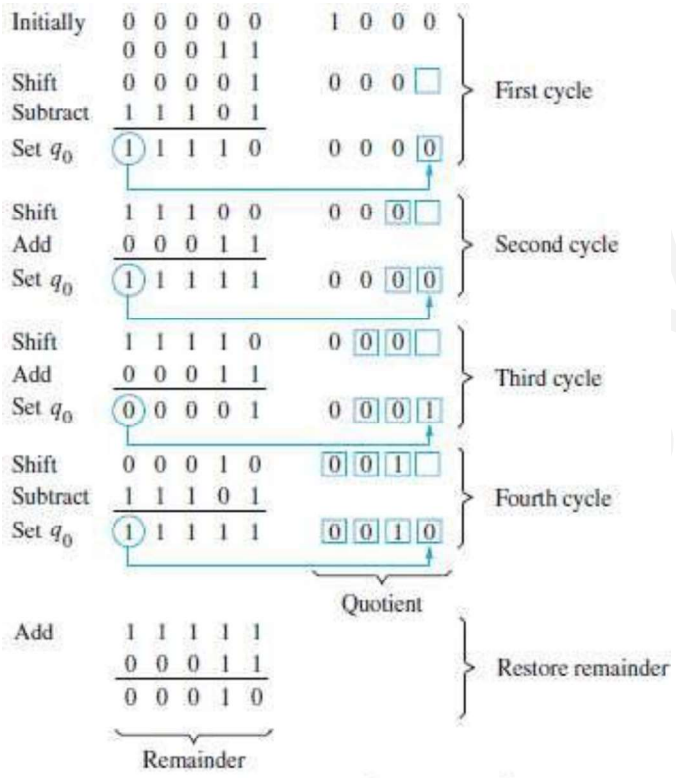


Figure 9.25 A non-restoring division example.

RESTORING DIVISION

- Procedure: Do the following n times
  1) Shift A and Q left one binary position (Figure 9.22).
  2) Subtract M from A, and place the answer back in A
  3) If the sign of A is 1, set q_0 to 0 and add M back to A(restore A).
     If the sign of A is 0, set q_0 to 1 and no restoring done.

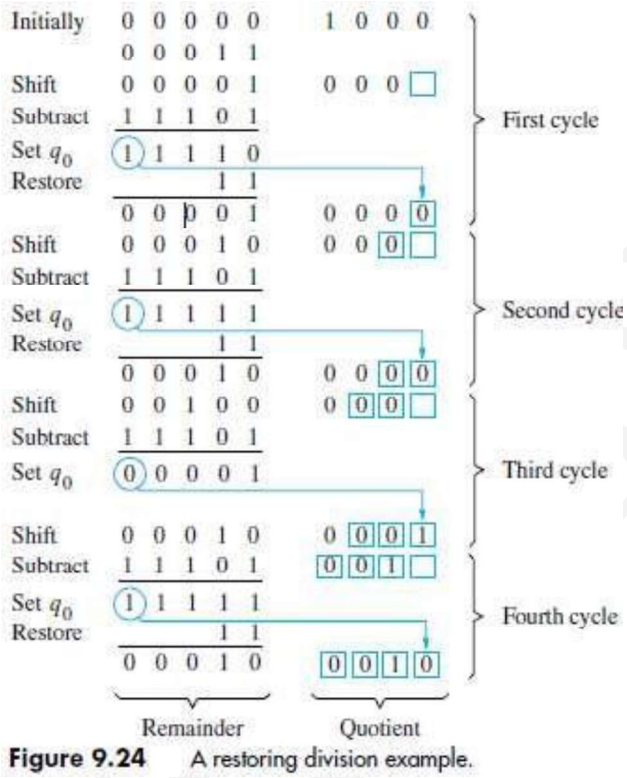


Figure 9.24 A restoring division example.