

Module IV

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 the sign-and-magnitude system, negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value. For example, +5 is represented by 0101 & -5 is represented by 1101.
- In 1's complement representation, negative values are obtained by complementing each bit of the corresponding positive number. For example, -3 is obtained by complementing each bit in 0011 to yield 1100. (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 the 2's complement system, forming the 2's complement of a number is done by subtracting that number from 2^n .
- (In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).
- The 2's complement system yields the most efficient way to carry out addition and subtraction operations.

b	Values represented			
	$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 7	- 8
1 0 0 1	- 1	- 6	- 6	- 7
1 0 1 0	- 2	- 5	- 5	- 6
1 0 1 1	- 3	- 4	- 4	- 5
1 1 0 0	- 4	- 3	- 3	- 4
1 1 0 1	- 5	- 2	- 2	- 3
1 1 1 0	- 6	- 1	- 1	- 2
1 1 1 1	- 7	- 0	- 0	- 1

Figure 2.1 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.

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.

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \end{array}
 \quad
 \begin{array}{r} 1 \\ + 0 \\ \hline 1 \end{array}
 \quad
 \begin{array}{r} 0 \\ + 1 \\ \hline 1 \end{array}
 \quad
 \begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$

↑
Carry-out

Figure 2.2 Addition of 1-bit numbers.

1. To add two numbers, add their n-bits and ignore the carry-out signal from the MSB position. The sum will be algebraically correct value as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$ (Figure 2.4).
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 $+(2^{n-1}-1)$.

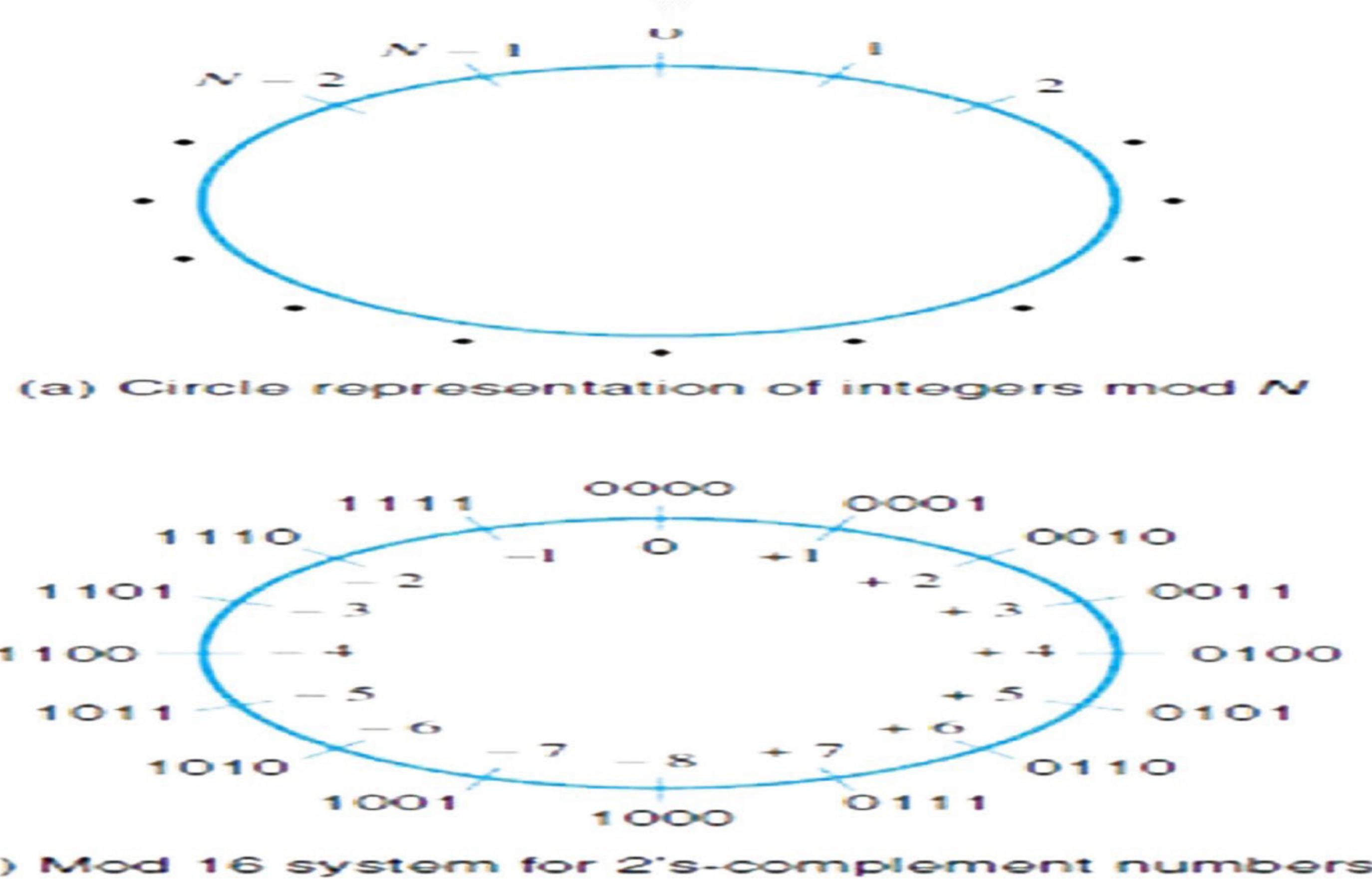


Figure 2.3 Modular number systems and the 2's-complement system.

- 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 the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.
- For example, when using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output sum S is 1011, 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 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.(Fig 9.2).

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:

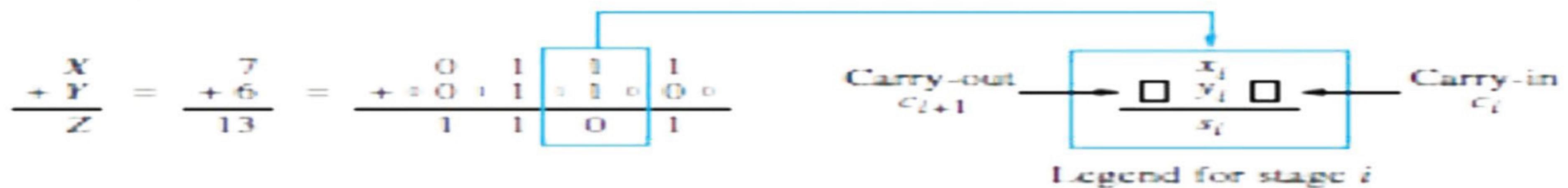


Figure 9.1 Logic specification for a stage of binary addition.

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.

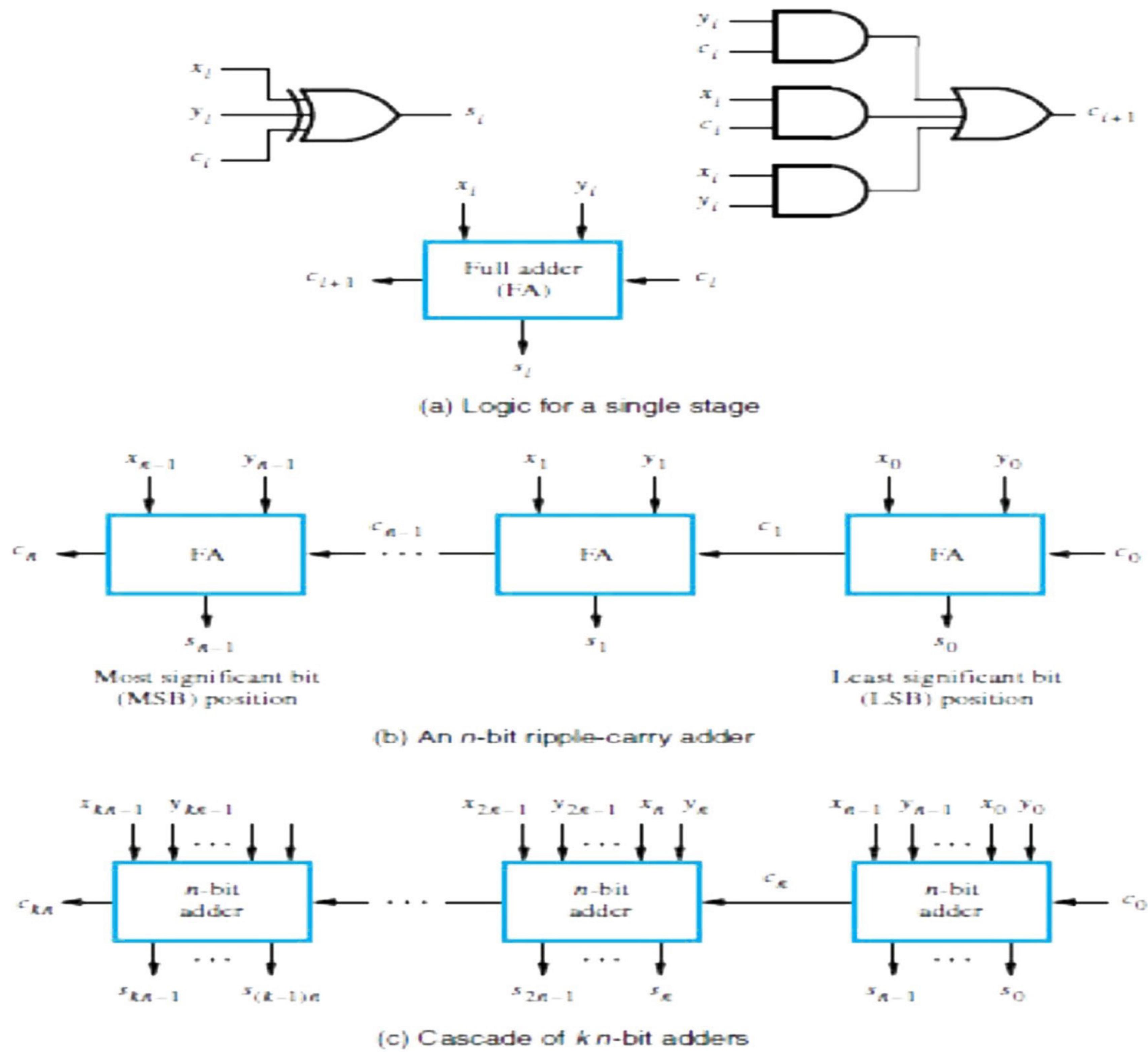


Figure 9.2 Logic for addition of binary numbers.

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

- Factoring (2) into

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

- this can be written as

$$c_{i+1} = G_i + P_i C_i$$

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

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i G_0 + P_i P_{i-1} \dots P_0 c_0$$

- 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

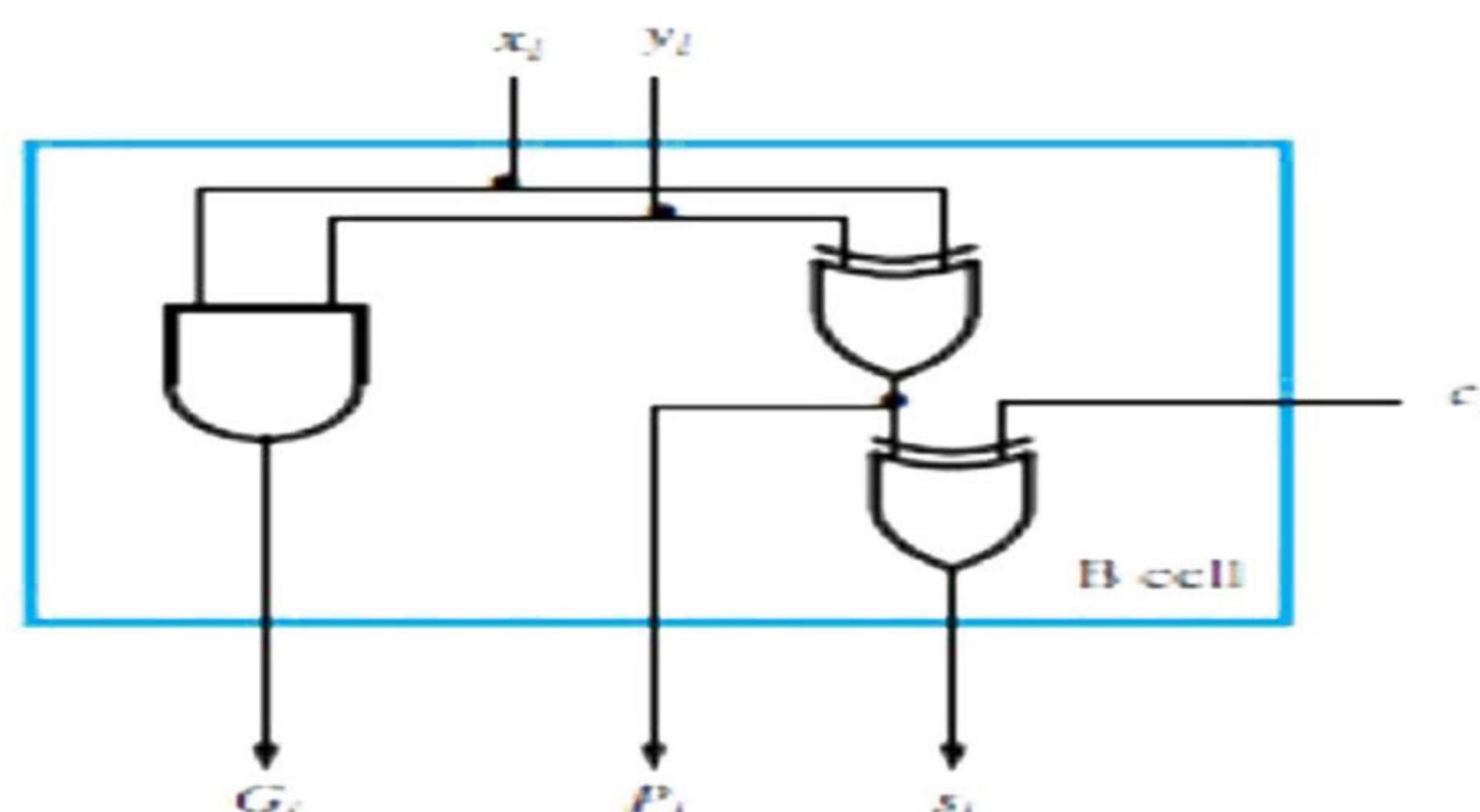
$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

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

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

- 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.



(a) Bit-stage cell

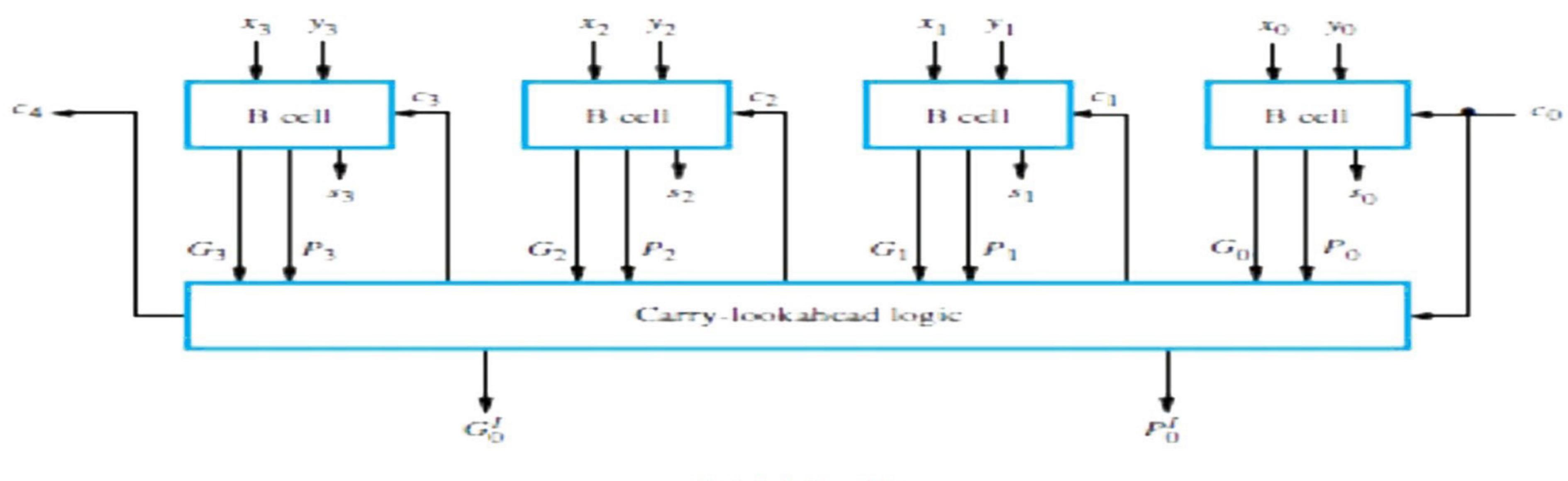


Figure 9.4 A 4-bit carry-lookahead adder.

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,

$$P_0 = P_3 P_2 P_1 P_0 \quad \& \\ G_0 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

- 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
- $$c_{16} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$
- Conclusion: All carries are available 5 gate delays after X, Y and c_0 are applied as inputs.

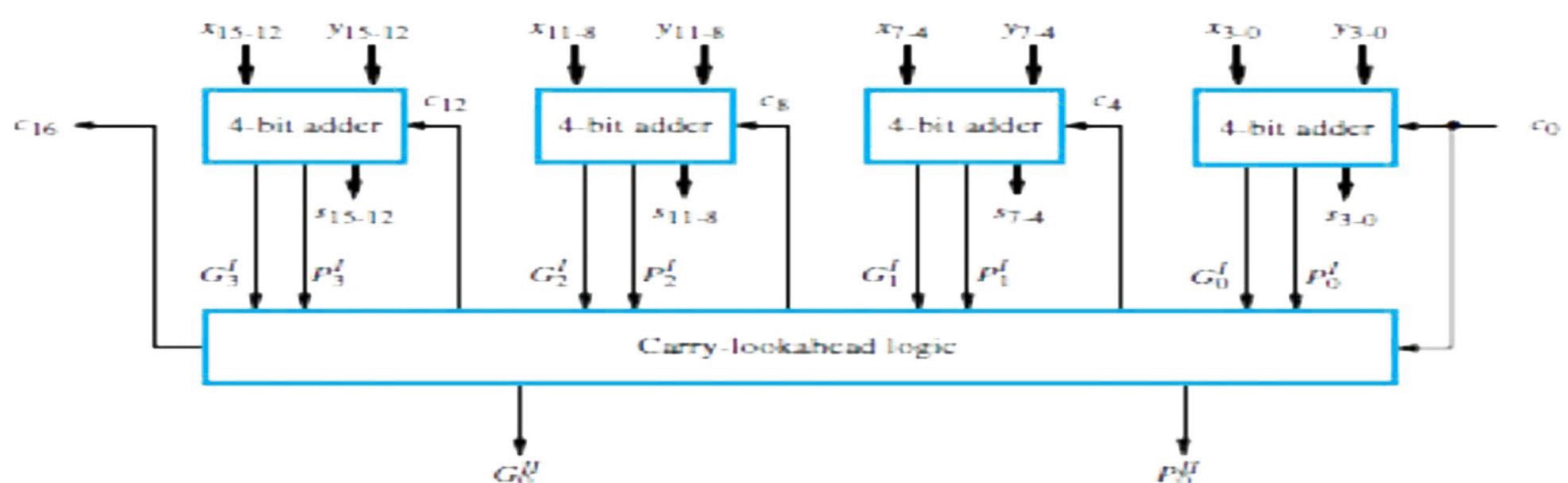


Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4b).

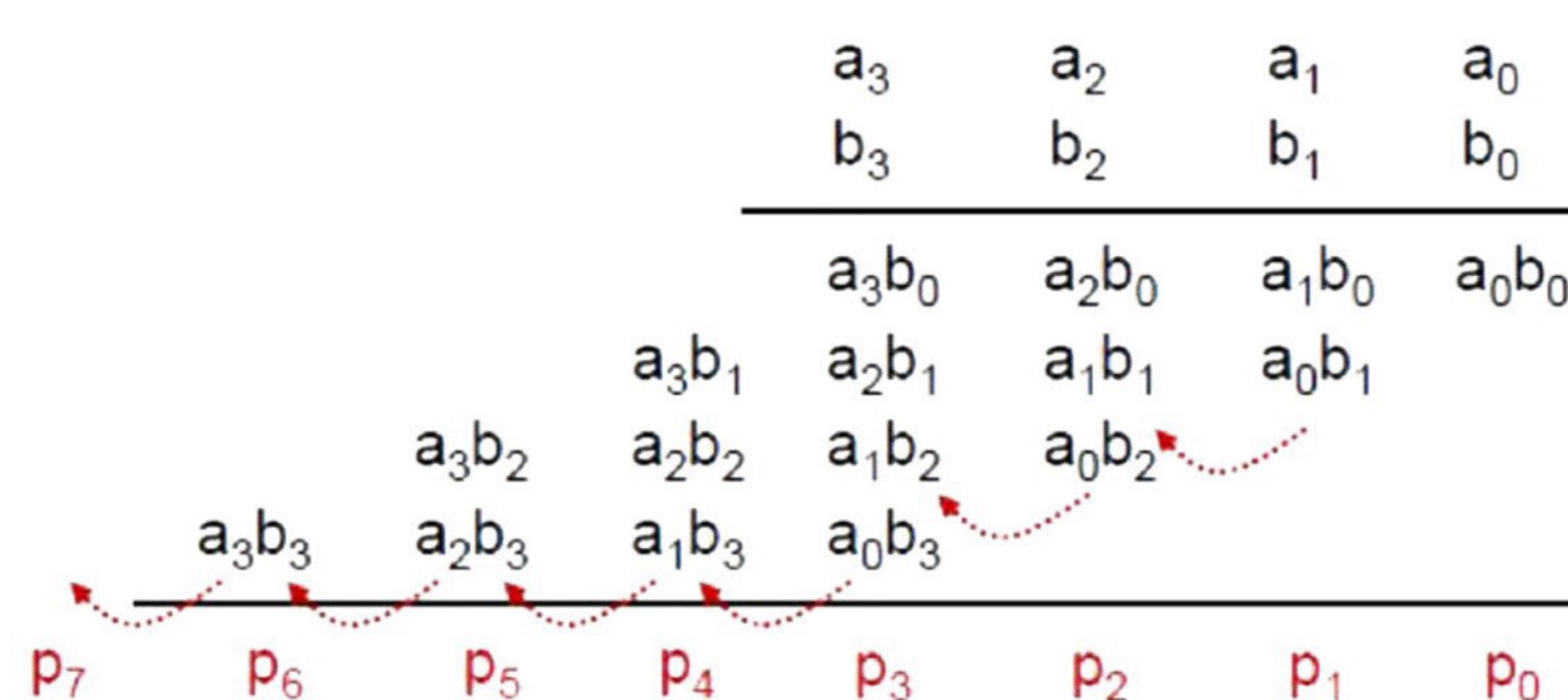
MULTIPLICATION OF POSITIVE NUMBERS

Bit-level multiplier

a	b	axb
0	0	0
0	1	0
1	0	0
1	1	1



Multiplication of two 4-bit words



ARRAY MULTIPLICATION

- The main component in each cell is 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).

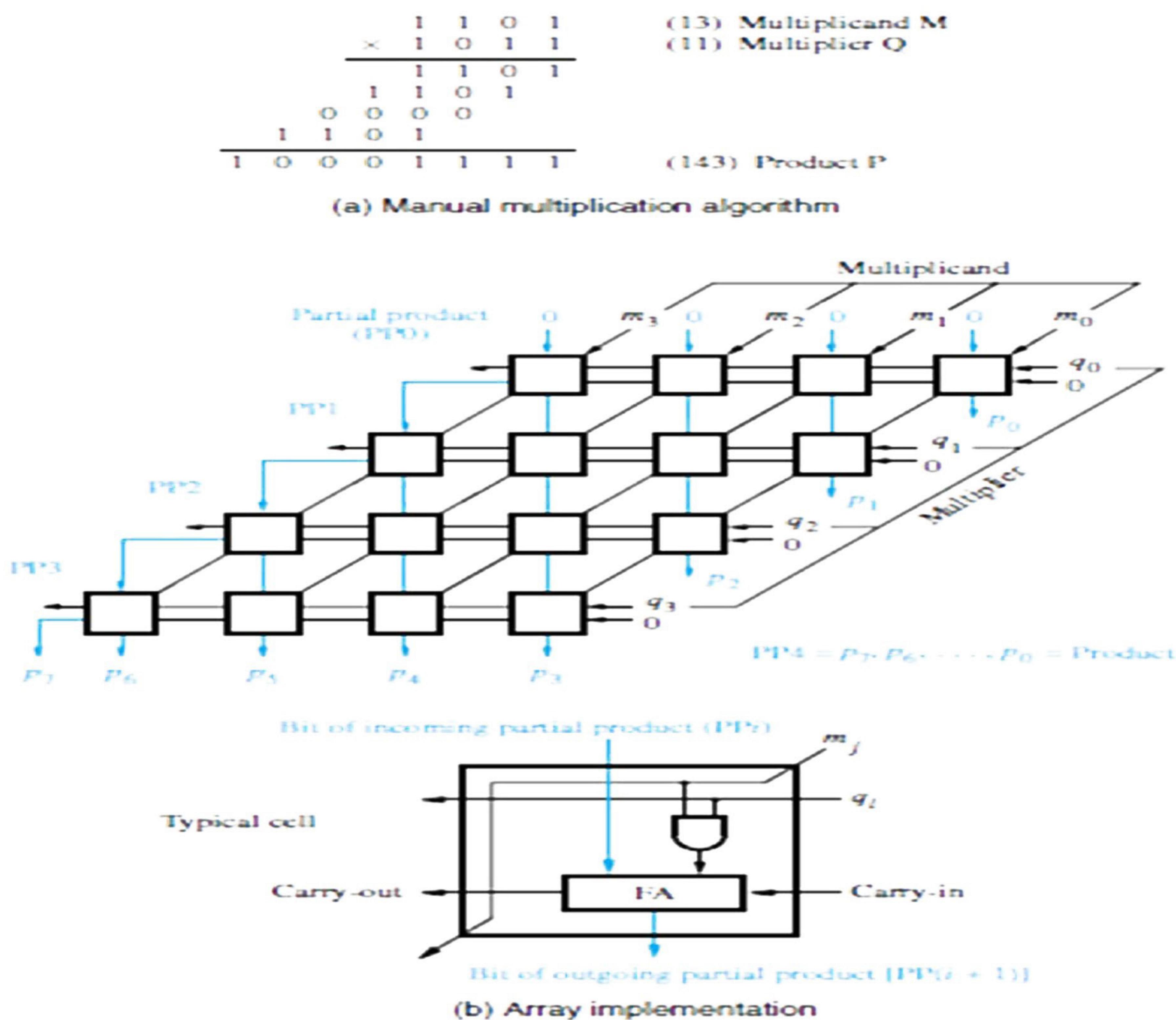


Figure 9.6 Array multiplication of unsigned binary operands.

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:
 - Multiplier is loaded into register Q, Multiplicand is loaded into register M and C & A are cleared to 0.
 - 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.
 - 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.

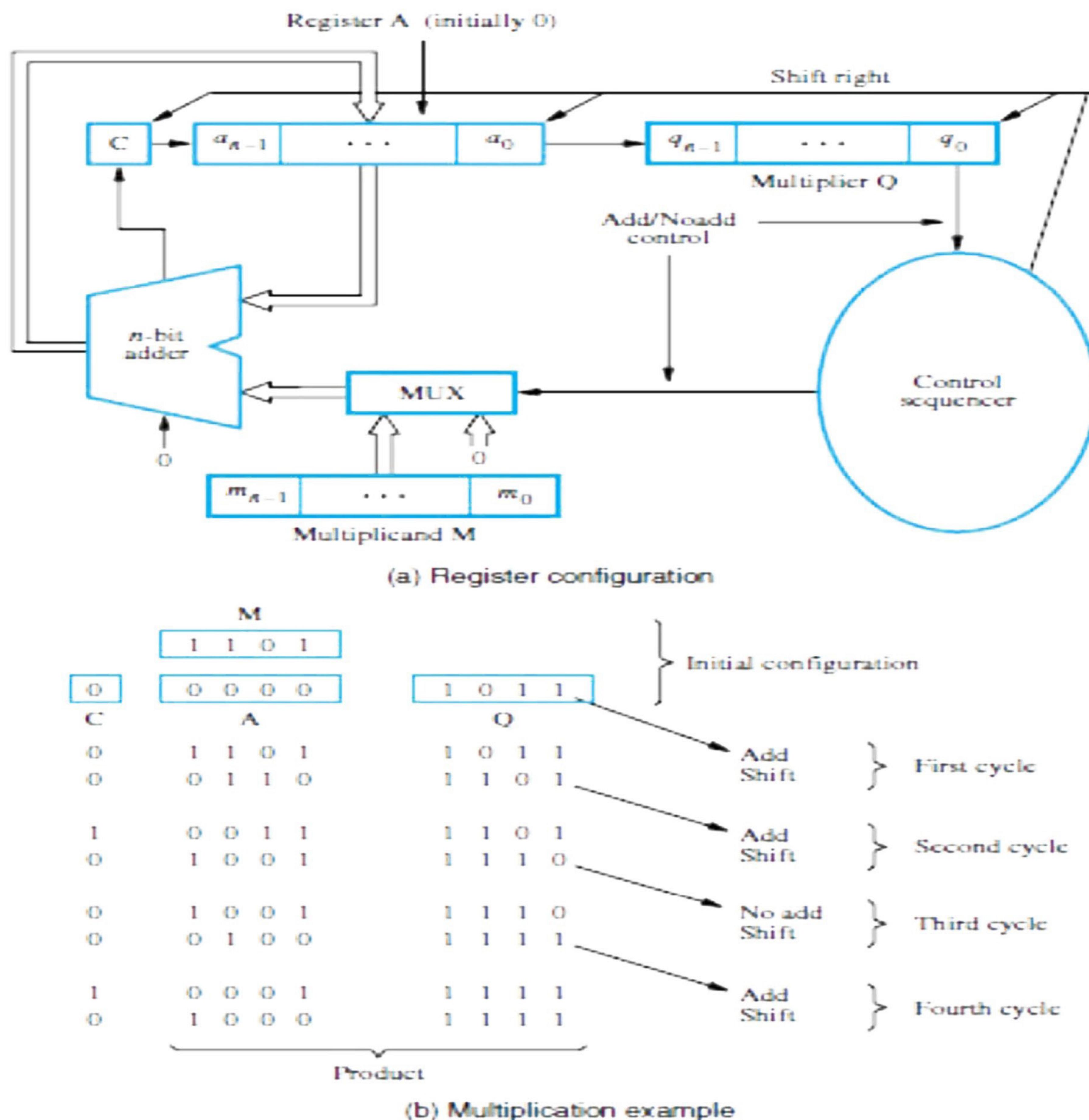


Figure 9.7 Sequential circuit binary multiplier.

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)
- 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

Algorithm:

- Step1:** Find the 2's compliment of -ve number
- Step2:** Find the new Q by applying booth recoding technique (use Figure 9.12).
- Step3:** Multiply M with new Q for summands of 2n-bits,
where after n-bit the n+1 bit onwards bits til 2n-bit are called sign extension bit, which will have the value of sign bit of the partial product.
 $+1^*M$ means M as it is and -1^*M means 2's compliment of M
- Step4:** The final product should be in **2n-bits** more than bits can be ignored.

$$\begin{array}{r}
 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & 0 + 1 & + 1 & + 1 & + 1 & 0 \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0
 \end{array}$$

$$\begin{array}{r}
 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 & 0 + 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0
 \end{array}$$

2's complement of the multiplicand

Figure 9.9 Normal and Booth multiplication schemes.

$$\begin{array}{cccccccccccccccc}
 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & & & & & & & & & & & & & & & \\
 & 0 & + 1 & - 1 & + 1 & 0 & - 1 & 0 & + 1 & 0 & 0 & - 1 & + 1 & - 1 & + 1 & 0 & - 1 & 0 & 0
 \end{array}$$

Figure 9.10 Booth recoding of a multiplier.



$$\begin{array}{r}
 0 1 1 0 1 (+13) \\
 \times 1 1 0 1 0 (-6) \\
 \hline
 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{r}
 0 1 1 0 1 \\
 0 - 1 + 1 - 1 0 \\
 \hline
 0 0 0 0 0 0 0 0 0 0 \\
 1 1 1 1 1 0 0 1 1 \\
 0 0 0 0 1 1 0 1 \\
 1 1 1 0 0 1 1 \\
 0 0 0 0 0 0 0 \\
 \hline
 1 1 1 0 1 1 0 0 1 0 (-78)
 \end{array}$$

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.

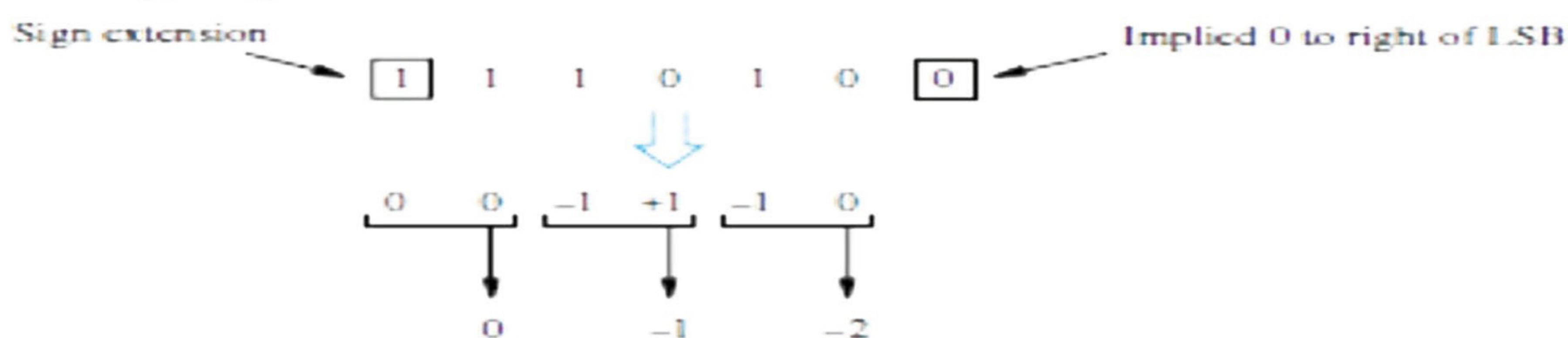
Worst-case multiplier	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1
Ordinary multiplier	1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
	0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0
Good multiplier	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Figure 9.13 Booth recoded multipliers.

FAST MULTIPLICATION

BIT-PAIR RECODING OF MULTIPLIERS

- This method
 - ✓ Derived from the booth algorithm so it also called as modified 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

- **Algorithm:**
- **Step1:** Find the 2's compliment of of –ve number
- **Step2:** Find the new Q by applying bit-pair recoding technique.

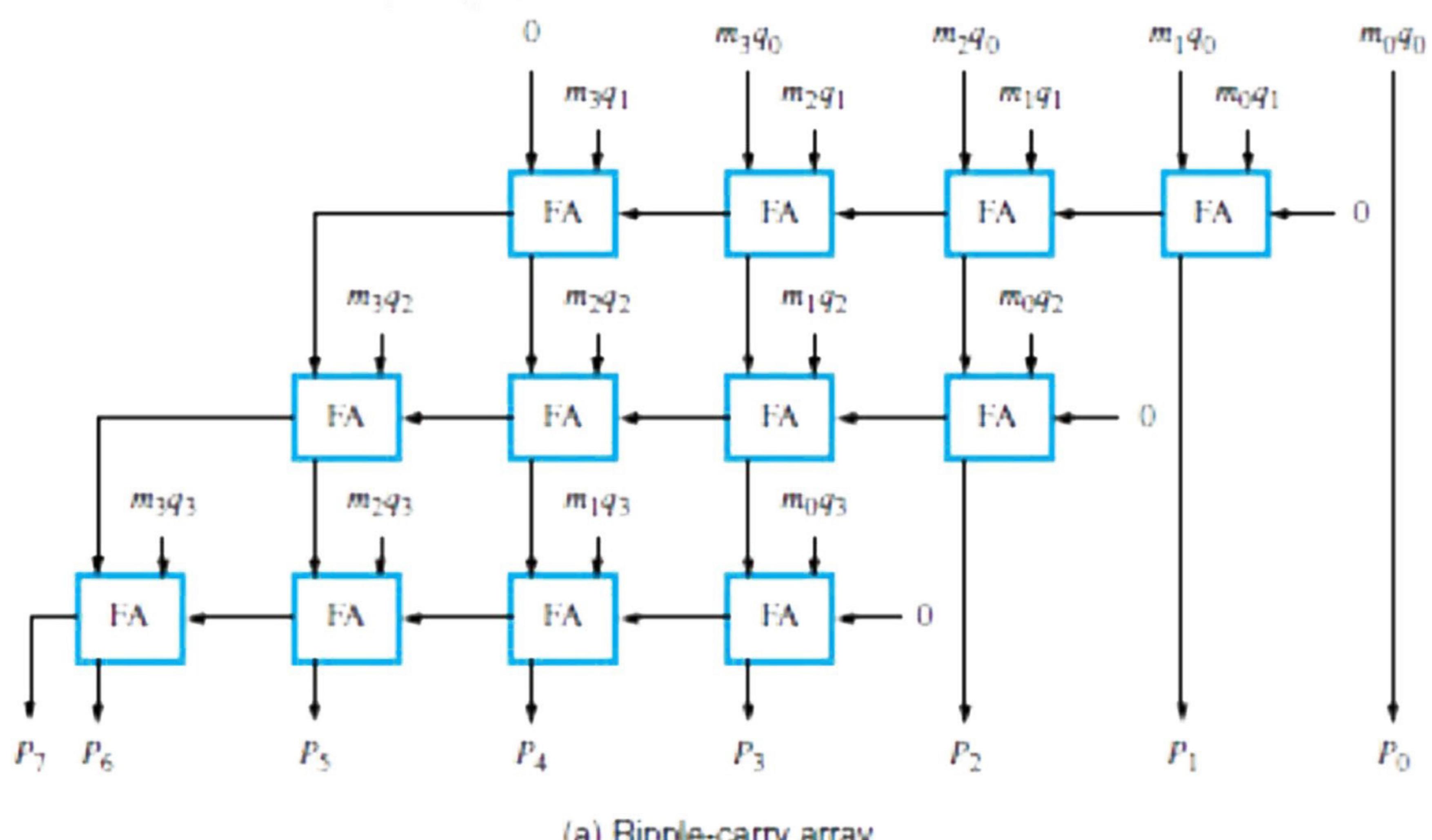
- **Step3:** Multiply M with new Q for summands of $2n$ -bits, and each summands will be having 2-bit difference, sign extension bits are treated as like booth algorithm.
+ 2^kM means $2M$ and - 2^kM means 2's compliment of $2M$
 - **Step4:** The final product should be in **$2n$ -bits** more than bits can be ignored.

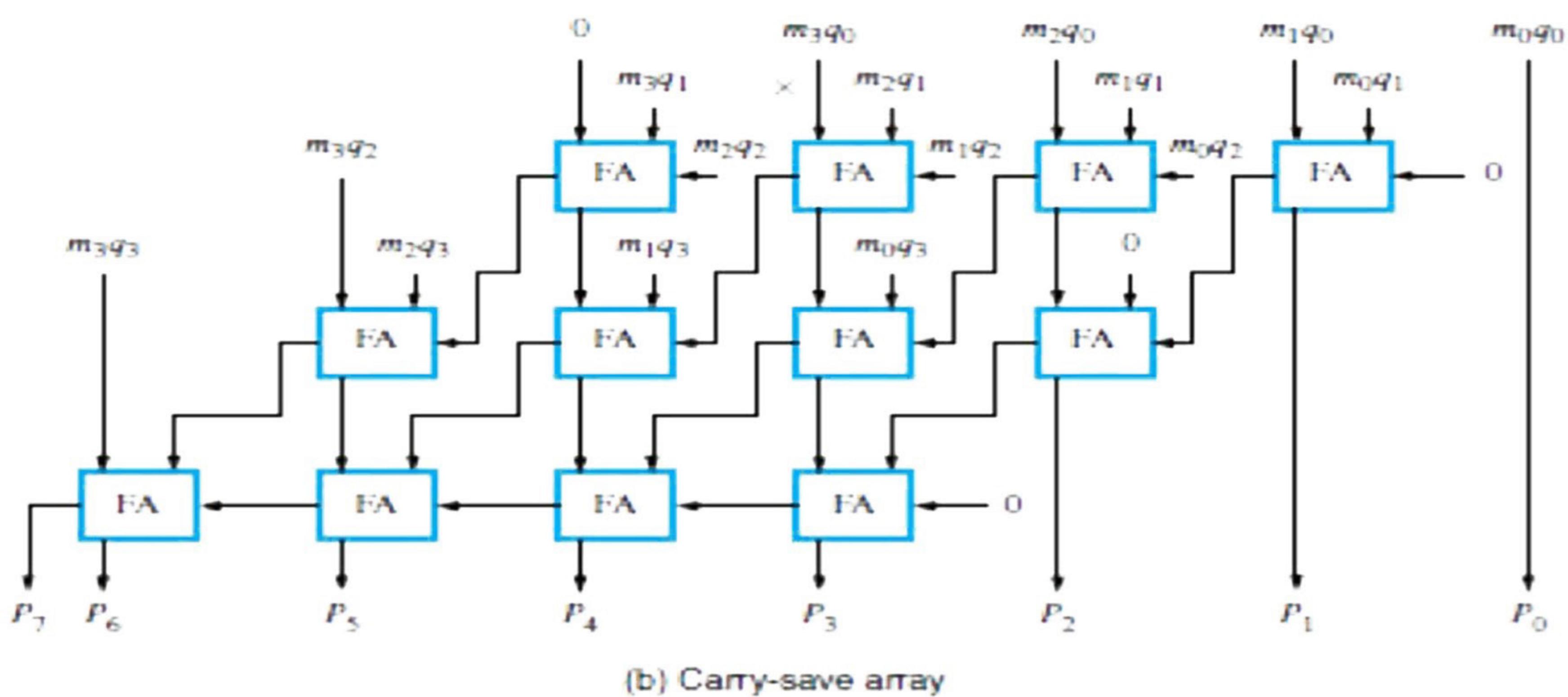
$$\begin{array}{r}
 & 0 & 1 & 1 & 0 & 1 & (+13) \\
 \times & 1 & 1 & 0 & 1 & 0 & (-6) \\
 \hline
 & & & & & & \\
 & 0 & 1 & 1 & 0 & 1 & \\
 & 0 & -1 & +1 & -1 & 0 & \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & \\
 \hline
 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & (-78)
 \end{array}$$

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

CARRY-SAVE ADDITION OF SUMMANDS

- Consider the array for 4*4 multiplications. (Figure 9.16 9.17 & 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 Ripple-carry and carry-save arrays for a 4×4 multiplier.

$$\begin{array}{r}
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & (45) & \mathbf{M} \\
 \times & \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} & (63) & \mathbf{Q} \\
 \hline
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{A} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{B} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{C} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{D} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{E} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{F} \\
 \hline
 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & (2,835) & \text{Product}
 \end{array}$$

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

$$\begin{array}{r}
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} & \mathbf{M} \\
 \times & \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} & \mathbf{Q} \\
 \hline
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \\
 & \begin{array}{ccccccc} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 & \begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \\
 & \begin{array}{ccccccc} 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{array} \\
 + & \begin{array}{ccccccc} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} \\
 \hline
 & \begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array} & \text{Product}
 \end{array}$$

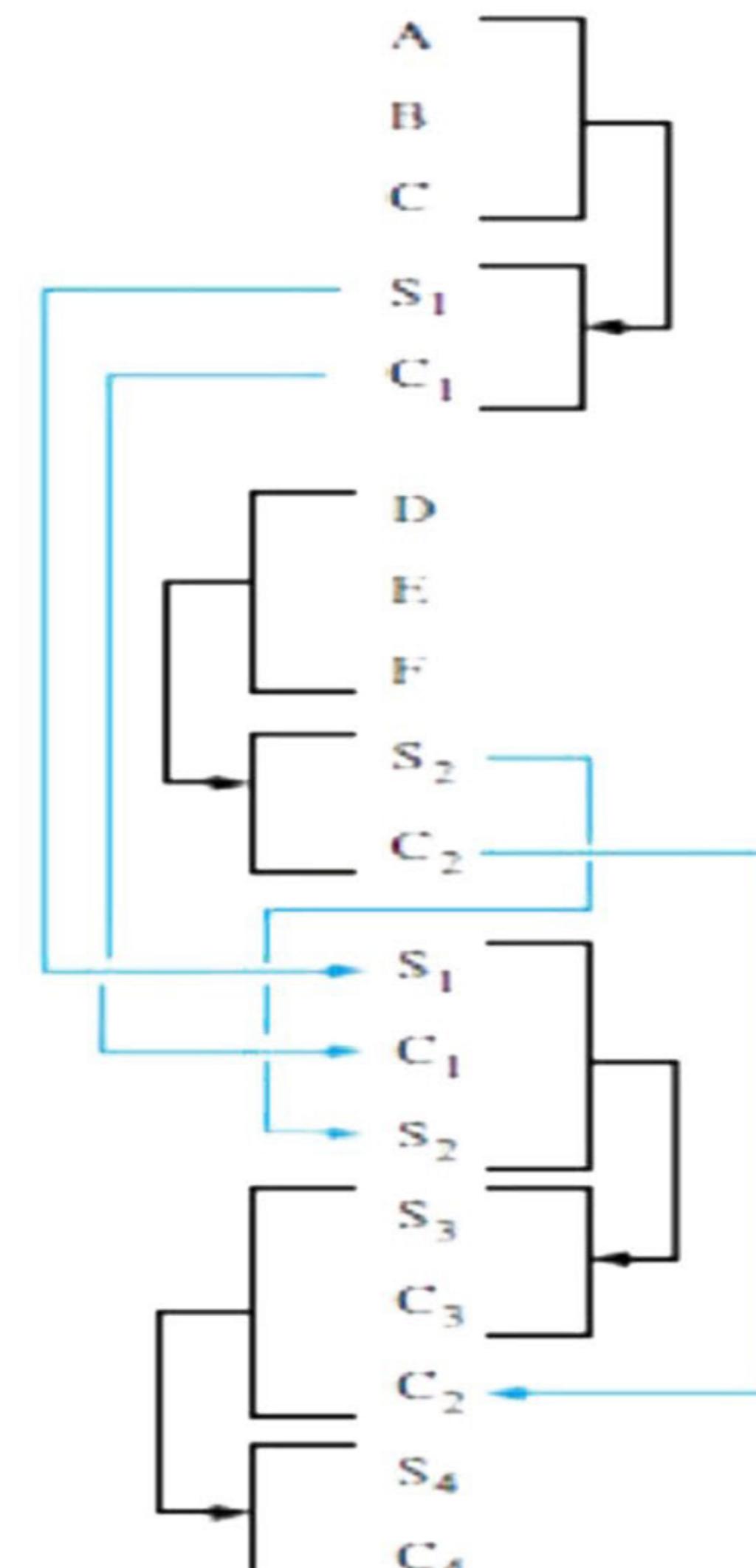


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

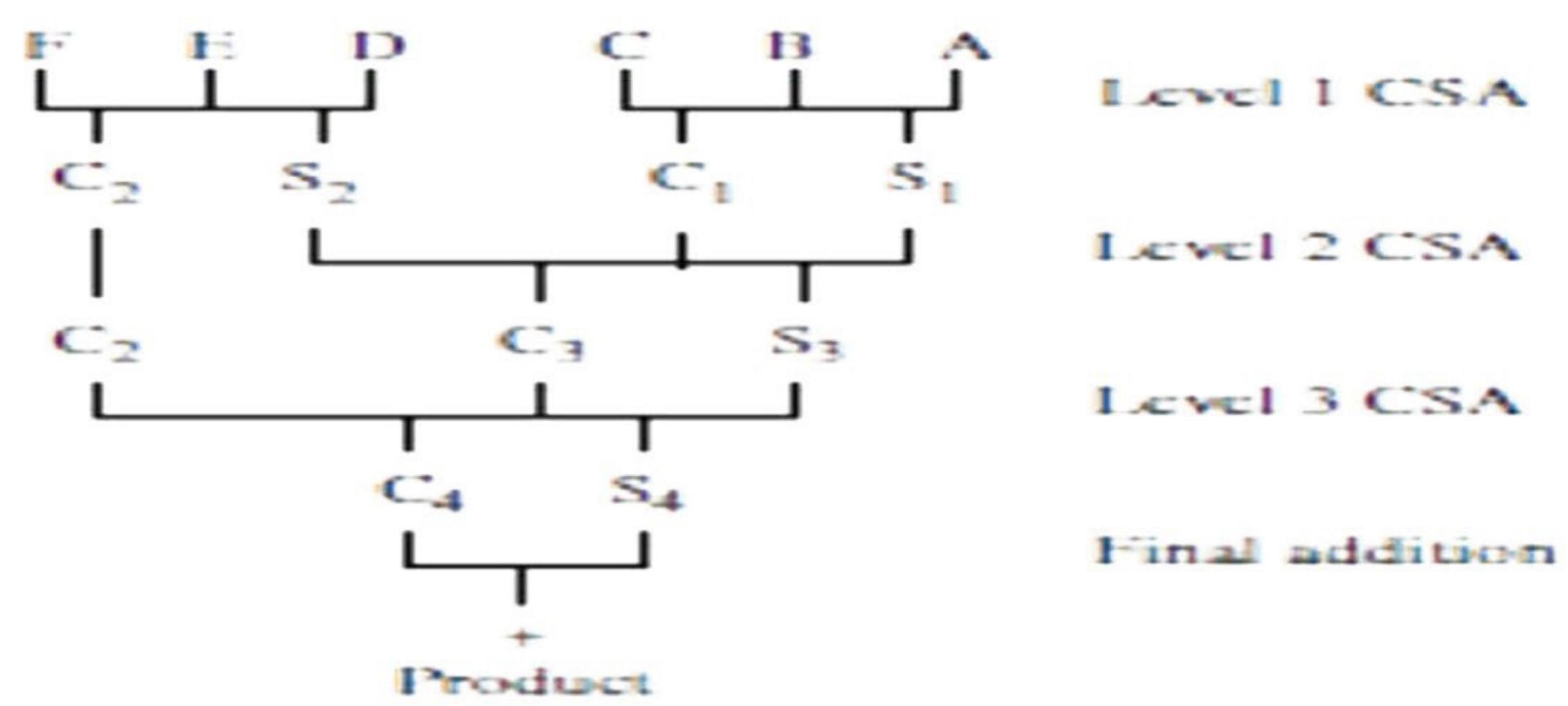


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

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 9.23).
 - After division operation, the n-bit quotient is in register Q, and the remainder is in register A

$$\begin{array}{r}
 & 10101 \\
 & \overline{)100010010} \\
 1101 & \overline{)100010010} \\
 & \overline{1101} \\
 & \overline{10000} \\
 & \overline{1101} \\
 & \overline{1110} \\
 & \overline{1101} \\
 & \overline{1}
 \end{array}$$

Figure 9.22 Longhand division examples.

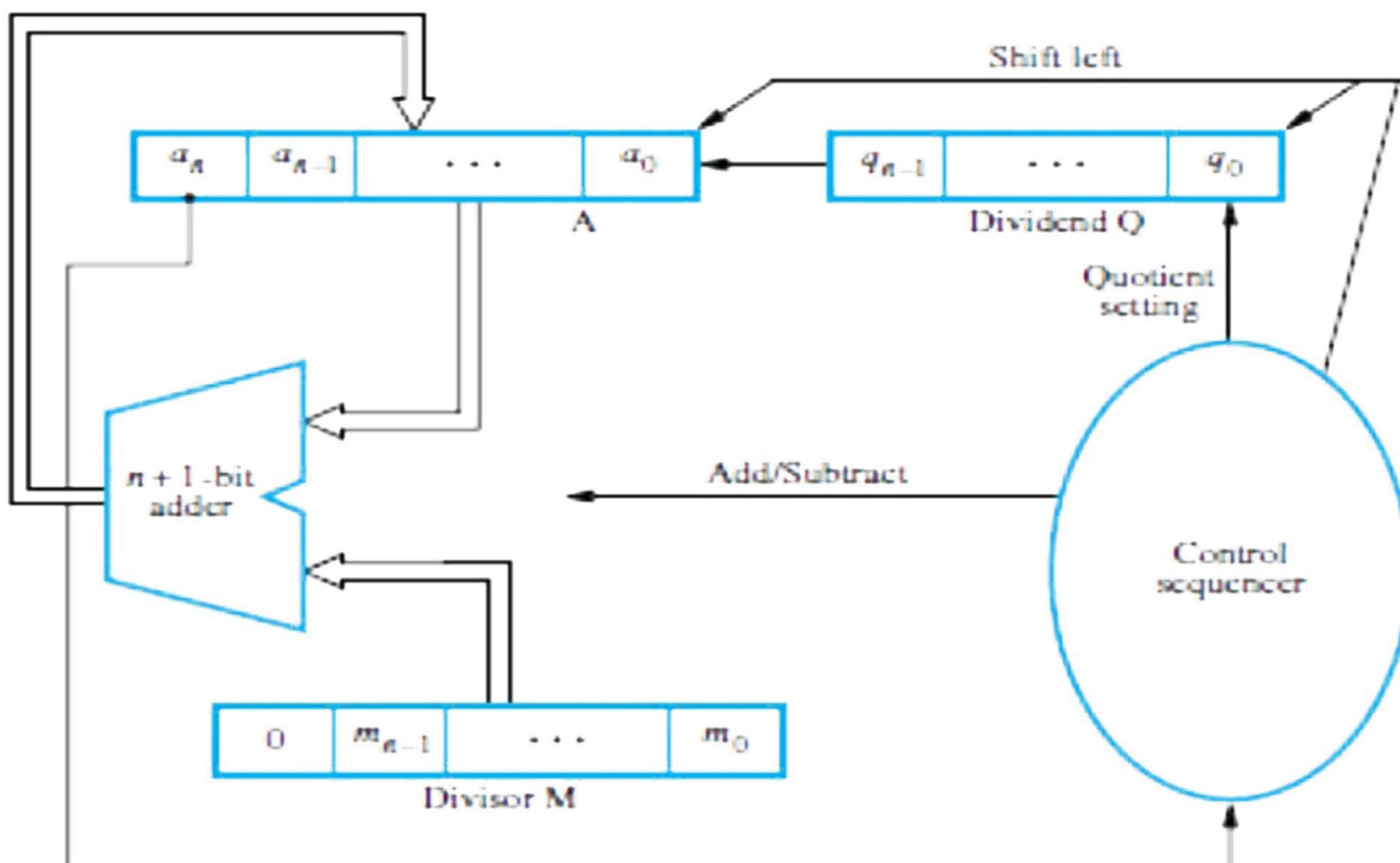


Figure 9.23 Circuit arrangement for binary division.

RESTORING DIVISION

- Procedure: Do the following n times
 - 1) Shift A and Q left one binary position (Figure 9.24).
 - 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), otherwise if the sign of A is 0, set q_0 to 1 and no restoring done.

$$\begin{array}{r} 10 \\ \overline{11) 1000} \\ 11 \\ \hline 10 \end{array}$$

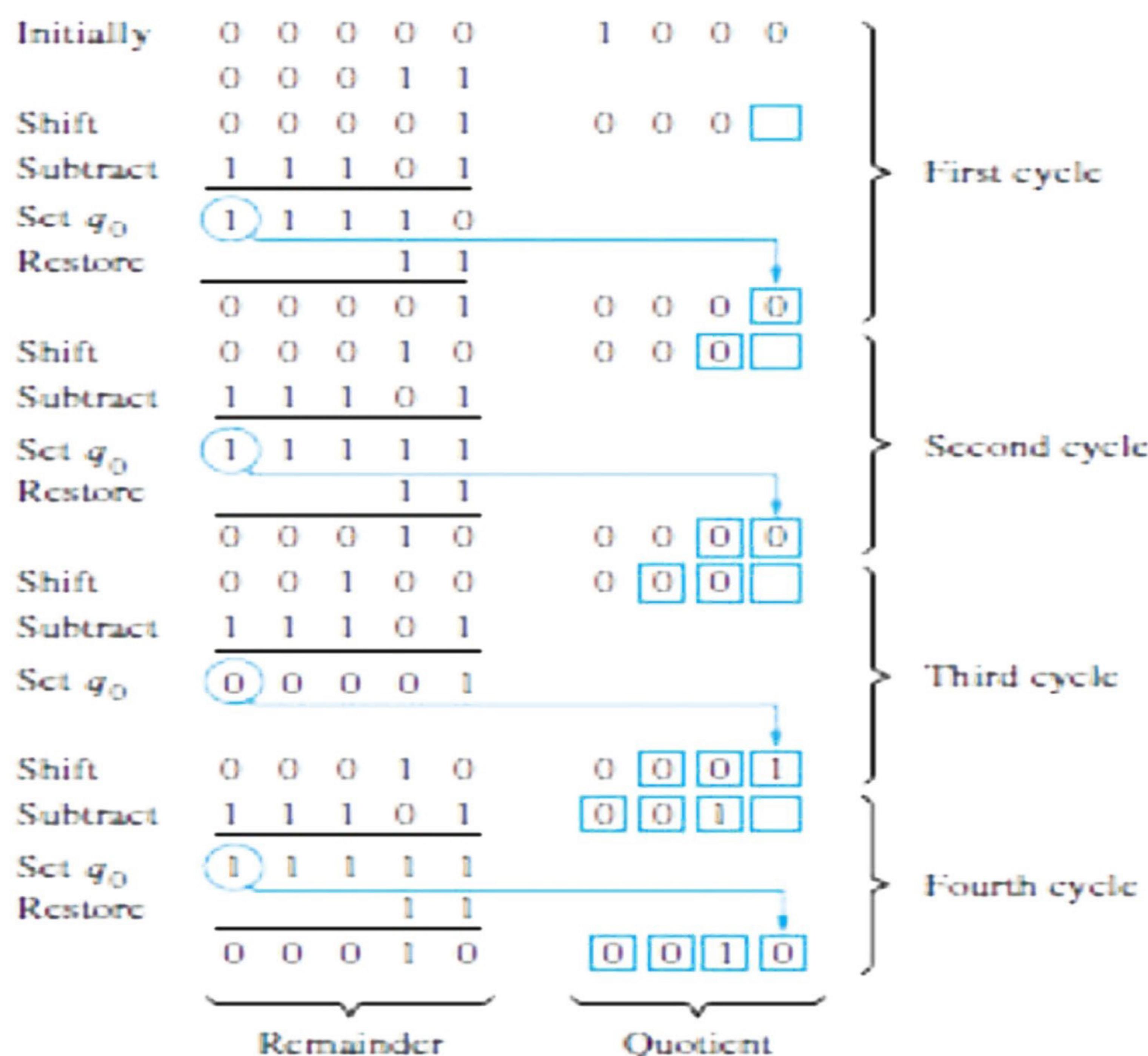


Figure 9.24 A restoring division example.

NON-RESTORING DIVISION

- Procedure:
 - 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 (Figure 9.25).
 - 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).

Initially	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 1 \ 1 \end{array}$	$\begin{array}{r} 1 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ \square \end{array}$	First cycle
Shift	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 0 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ \square \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Subtract	$\underline{\begin{array}{r} 1 \ 1 \ 1 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \ 0 \end{array}}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Set q_0	$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 0 \\ 1 \ 1 \ 1 \ 1 \ 0 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Shift	$\begin{array}{r} 1 \ 1 \ 1 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 1 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ \square \\ 0 \ 0 \ 0 \ 0 \end{array}$	Second cycle
Add	$\underline{\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 1 \\ 1 \ 1 \ 1 \ 0 \ 0 \end{array}}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Set q_0	$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 0 \ 0 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Shift	$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 0 \\ 0 \ 0 \ 0 \ 0 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ \square \\ 0 \ 0 \ 0 \ 0 \end{array}$	Third cycle
Add	$\underline{\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \ 0 \end{array}}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Set q_0	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \\ 1 \ 1 \ 1 \ 1 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \end{array}$	
Shift	$\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 0 \\ 1 \ 1 \ 1 \ 0 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 1 \ \square \\ 0 \ 0 \ 0 \ 1 \end{array}$	Fourth cycle
Subtract	$\underline{\begin{array}{r} 1 \ 1 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 1 \ 0 \end{array}}$	$\begin{array}{r} 0 \ 0 \ 1 \ 0 \\ 0 \ 0 \ 0 \ 1 \end{array}$	
Set q_0	$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 0 \ 1 \ 0 \end{array}$	$\begin{array}{r} 0 \ 0 \ 1 \ 0 \\ 0 \ 0 \ 0 \ 1 \end{array}$	
		$\overbrace{\quad\quad\quad\quad\quad}$ Quotient	
Add	$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 0 \ 1 \ 1 \end{array}$	$\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 0 \\ 0 \ 0 \ 0 \ 1 \ 0 \end{array}$	Restore remainder
	$\underline{\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 1 \\ 1 \ 1 \ 1 \ 1 \ 1 \end{array}}$	$\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 0 \\ 0 \ 0 \ 0 \ 1 \ 0 \end{array}$	
	$\overbrace{\quad\quad\quad\quad\quad}$ Remainder		

Figure 9.25 A non-restoring division example.

Question Bank

1. Perform the following operations on the 5-bit signed numbers using 2's complement representation system. Further indicate whether overflow has occurred.
 - i) $(-10)+(-13)$, ii) $(-10)-(14)$, iii) $(+7)-(-15)$, iv) $(+8)+(+10)$,
 - v) $(-3) + (-8)$, vi) $(-10) - (+7)$
2. Multiply i) $(+14)$ and (-6) , ii) (-12) and (-11) , iii) 14 and (-8) , iv) 1100 and 1001 , v) 10111 and 1000 , vi) 1111 and 1110 , vii) 1011 and 1100 , viii) 1111 and 1011 ,
3. ix) 25 and 31 , x) 17 and 15 using Booth's algorithm and Bit pair recoding technique and also write the algorithm.
4. Write the steps and perform the division i) 8 by 3 ($8/3$), ii) $12/4$, iii) $1101/101$, iv) $1111/110$, v) $1110/10$ using restoring division method and non-restoring division method.
5. Given $A = 10101$ and $B = 00100$ perform A/B using non-restoring division algorithm.
6. Design a logic circuit to perform addition/subtraction of two 'n' bit numbers X and Y .
7. Design a 4 bit carry lookahead logic & explain how it is faster than 4 bit ripple adder.

9. Explain different arithmetic operation on floating point number.
10. Multiply i) 15 and 9 , ii) 14 and 6 , iii) 1100 and 1001 , iv) 10111 and 1000 , v) 1111 and 1110 , vi) 1011 and 1100 , vii) 1111 and 1011 , viii) 25 and 31 , ix) 17 and 15 , x) 12 and 6 using sequential multiplication technique and also write the algorithm.
11. Explain with a neat diagram implementation of floating point addition and subtraction unit.