Arithmetic

Addition and Subtraction of Signed Numbers

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

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

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

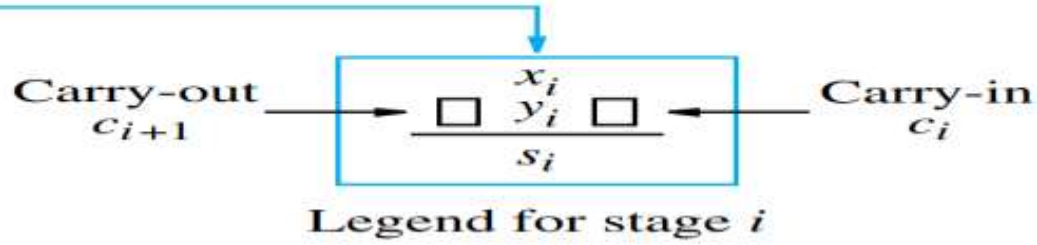
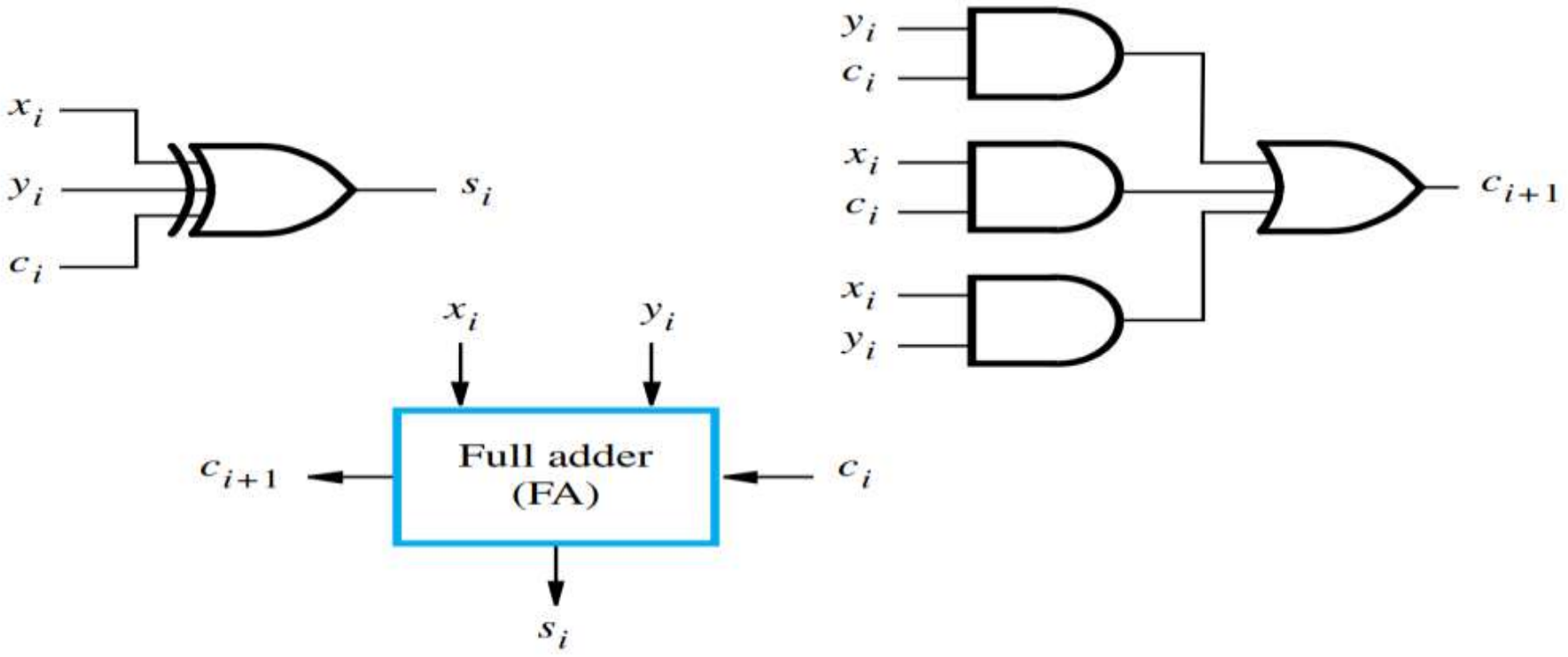


Figure 9.1 Logic specification for a stage of binary addition.

Figure shows the truth table for the sum and carry-out functions for adding equally weighted bits xi and yi in two numbers X and Y .

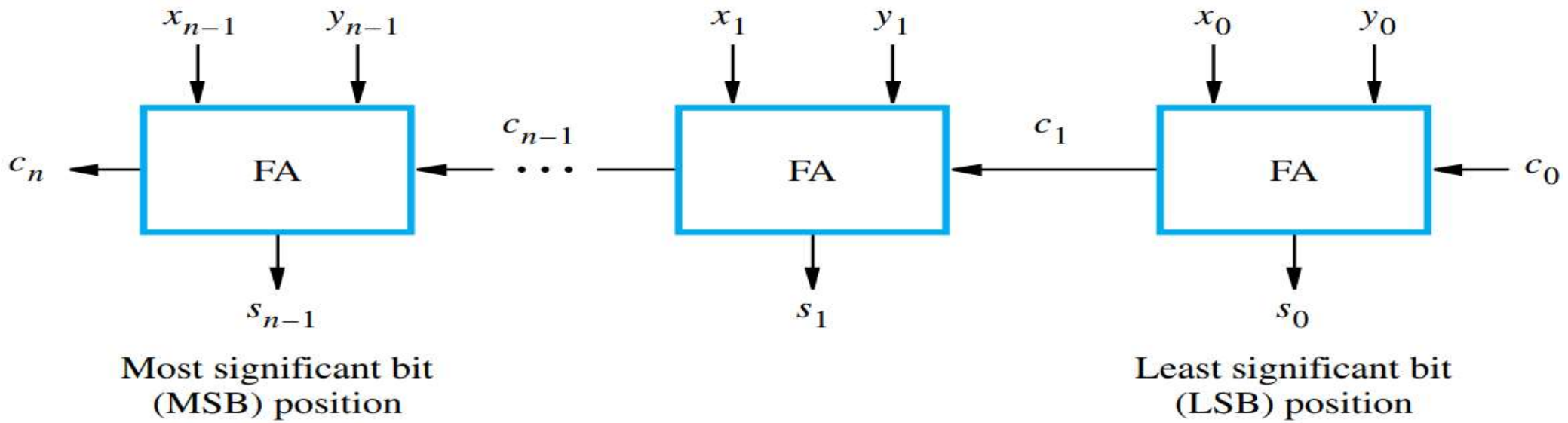
The logic expression for $s_i = \overline{x}_i \overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$

in can be implemented with a 3-input XOR gate, used in Figure 9.2a as part of the logic required for a single stage of binary addition. The carry-out function, ci+1, is implemented with an AND-OR circuit, as shown. A convenient symbol for the complete circuit for a single stage of addition, called a full adder (FA), is also shown in the figure.



(a) Logic for a single stage

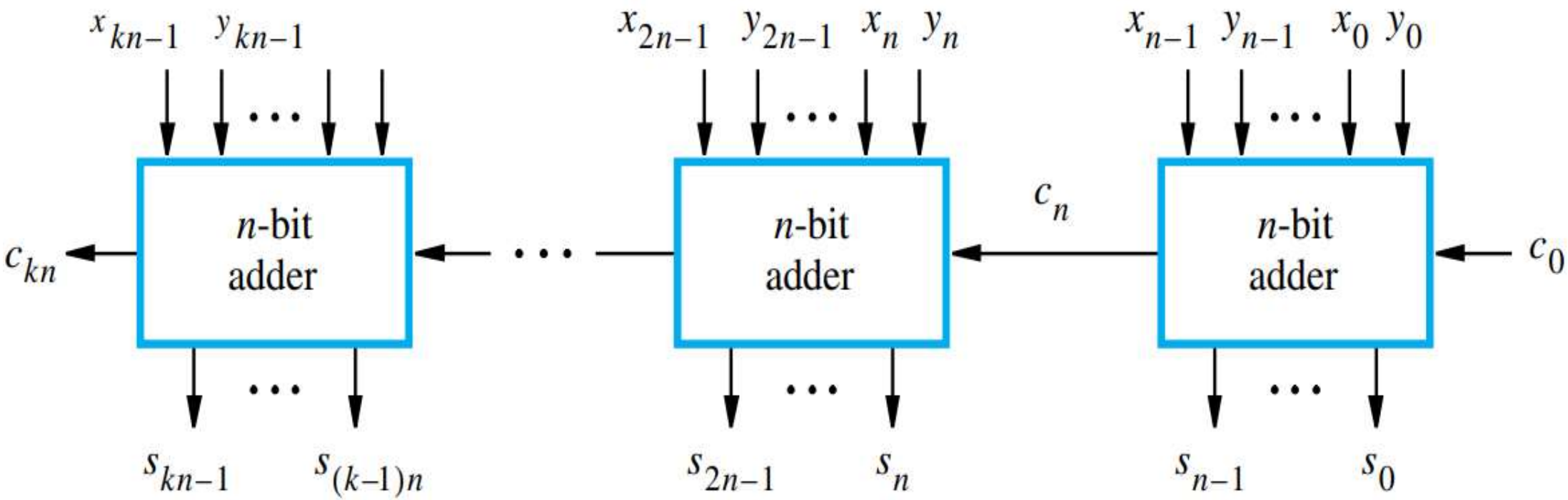
A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, as shown in Figure 9.2b. Since the carries must propagate, or ripple, through this cascade, the configuration is called a ripple-carry adder.



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

The carry-in, c_0, into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number. For instance, forming the 2's-complement of a number involves adding 1 to the 1's-complement of the number.

The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are kn bits long,



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

Addition/Subtraction Logic Unit

The n-bit adder can be used to add 2's-complement numbers X and Y , where the x_{n-1} and y_{n-1} bits are the sign bits.

The carry-out bit cn is not part of the answer. Because Arithmetic overflow occurs when the signs of the two operands are the same, but the sign of the result is different. Therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

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

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

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

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 .

The logic circuit can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line.

This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, c0, of 0.

When the Add/Sub control line is set to 1, the Y number is 1's-complemented (that is, bit-complemented) by the XOR gates and c0 is set to 1 to complete the 2's-complementation of Y .

An XOR gate can be added detect the overflow condition cn $\oplus$ cn−1

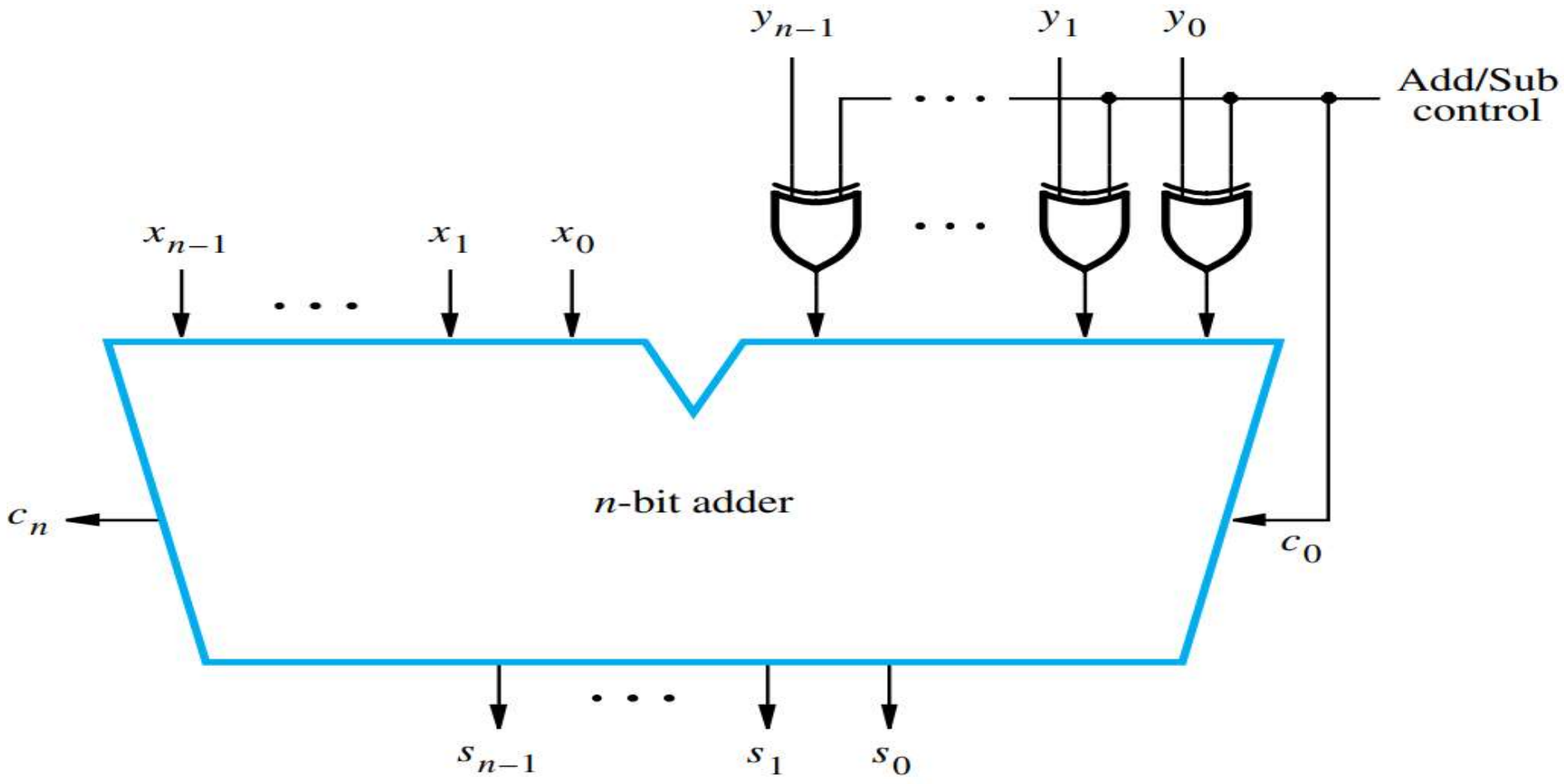


Figure 9.3 Binary addition/subtraction logic circuit.

Design of Fast Adders

A carry-look ahead adder (CLA) or fast adder is a type of adder used in digital logic.

A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.

- If n-bit ripple carry adder is used in Add/ Sub Circuit, it may have too much delay in producing output.
- This delay is mainly because of carry.
- So we need to find a better parallel method to calculate carry.

Ex: rather than waiting for c3 directly generate C4 , so we need to design a circuit.

A fast adder circuit must speed up the generation of the carry signals. The logic expressions for si (sum) and ci+1 (carry-out) of stage i are

$$s_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

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

We can write:

$$c_{i+1} = G_i + P_i c_i$$
$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

G_i is called generate function and P_i is called propagate function for stage

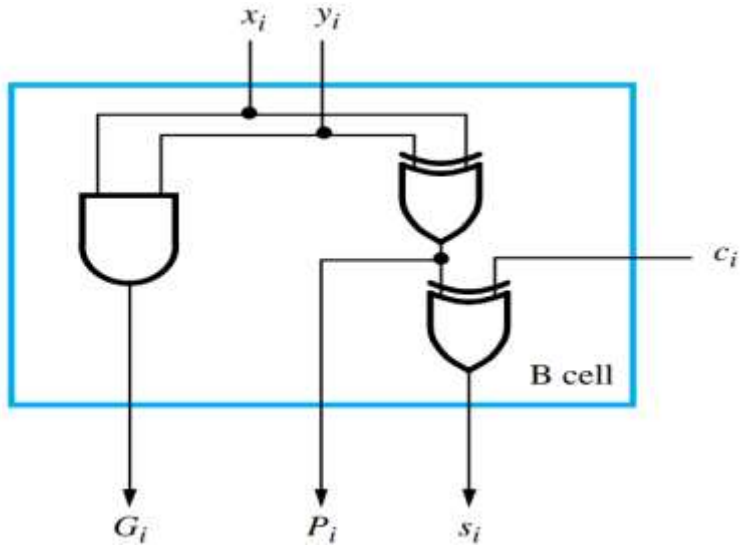
generate function for stage i is equal to 1, then ci+1 = 1, independent of the input carry, ci. This occurs when both xi and yi are 1.

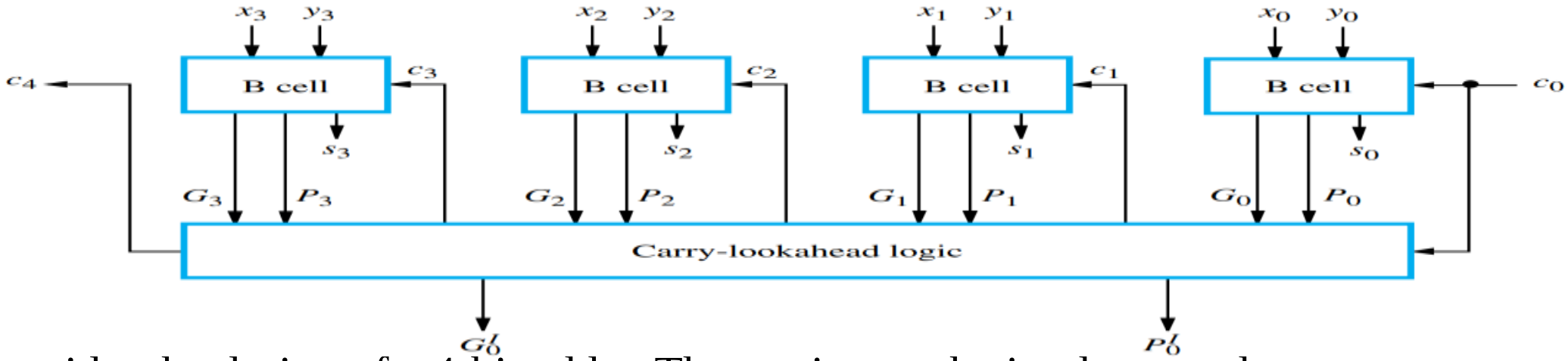
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.
- Expanding ci in terms of i − 1 subscripted variables and substituting into the ci+1 expression, we obtain ci+1 = Gi + PiGi−1 + PiPi−1ci−1

Continuing this type of expansion, the final expression for any carry variable is

$$ci+1 = Gi + PiGi-1 + PiPi-1Gi-2 + \cdots + PiPi-1 \cdots P1G0 + PiPi-1 \cdots P0c0$$

- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n, n-bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.





Let us consider the design of a 4-bit adder. The carries can be implemented as

$C_1=G_0+P_0C_0$

$C_2=G_1+P_1C_1=G_1+P_1(G_0+P_0C_0)=G_1+P_1G_0+P_1P_0C_0$

$C_3=G_2+P_2C_2=G_2+P_2(G_1+P_1G_0+P_1P_0C_0)$

$=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

$C_4=G_3+P_3C_3=G_3+P_3(G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0)$

$=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

- These equations suggest that C1,C2,C3 &C4 can be generated directly from C0.

i.e. these 4 carries depends on initial carry C0.

For this reason these equations are called carry-look ahead equations.

Performing *n*-bit addition in 4 gate delays independent of *n* is good.

CLA is a high speed adder , which adds 2 numbers without waiting for the carries from the previous stages.

In CLA, carry inputs of all stages are generated simultaneously, without using carries from the previous stages.

Higher-Level Generate and Propagate Functions

a 16-bit adder built from four 4-bit adder blocks.

These blocks provide new output functions defined as GI k and PI k , where k = 0 for the first 4-bit block, k = 1 for the second 4-bit block, and so on,

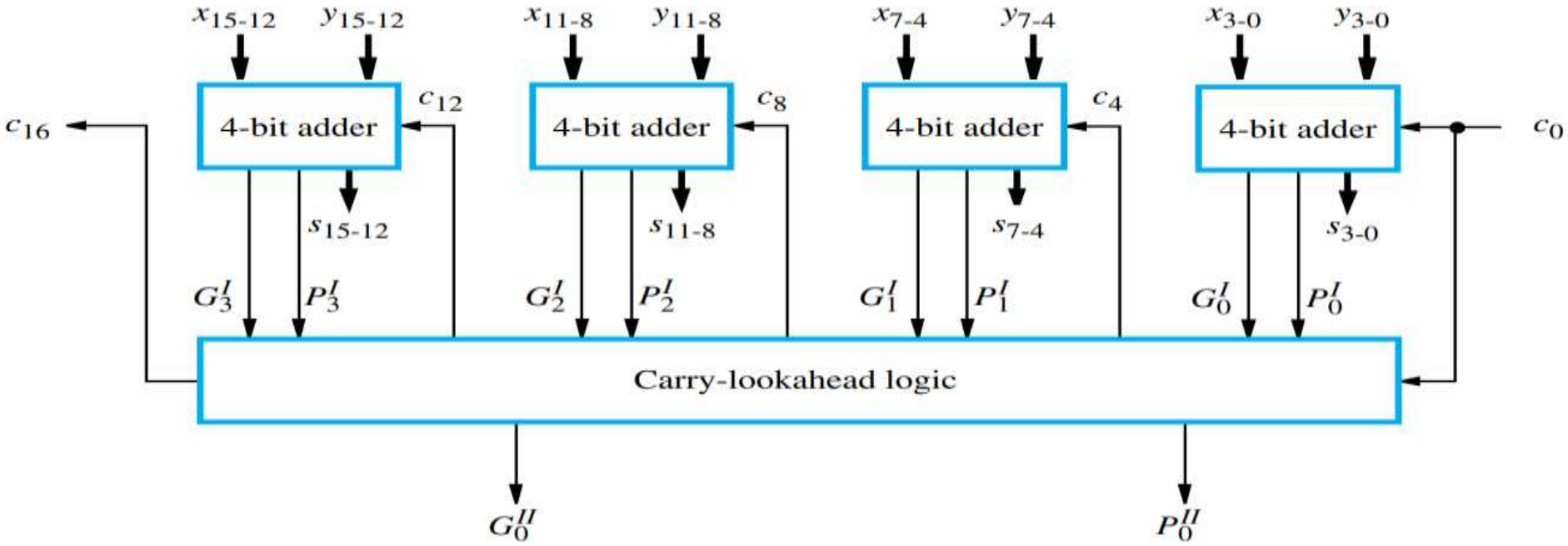
In the first block, $P_0^I = P_3P_2P_1P_0$

and $G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

Subscript I denotes the blocked carry look ahead and identifies the block

Carry c16 is formed by one of the carry-look ahead circuits in fig

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I c_0$$



After x_i, y_i and c_0 are applied as inputs:

- - G_i and P_i for each stage are available after 1 gate delay.
- - P^I is available after 2 and G^I after 3 gate delays.
- - All carries are available after 5 gate delays.
- - c_{16} is available after 5 gate delays.
- - s_{15} which depends on c_{12} is available after 8 (5+3)gate delays
- (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

Multiplication of Unsigned Numbers

- Multiplication of two unsigned binary numbers of n bit size results into 2n bit result.
- In binary system, multiplication of the multiplicand by multiplier, if multiplier is 1 then multiplicand is entered in appropriate shifted position and if the multiplier is 0 then 0s are entered in appropriate shifted position.
- Let us consider a Multiplicand of 4 bit size as (A3, A2, A1, A0) and Multiplier of 4 bit size as (B3, B2, B1, B0). Multiplication of these two is as shown below

A3 A2 A1 A0 - Multiplicand

X B3 B2 B1 B0 - Multiplier

3B0 A2B0 A1B0 A0B0 Partial Products (PP)

+ A3B1 A2B1 A1B1 A0B1

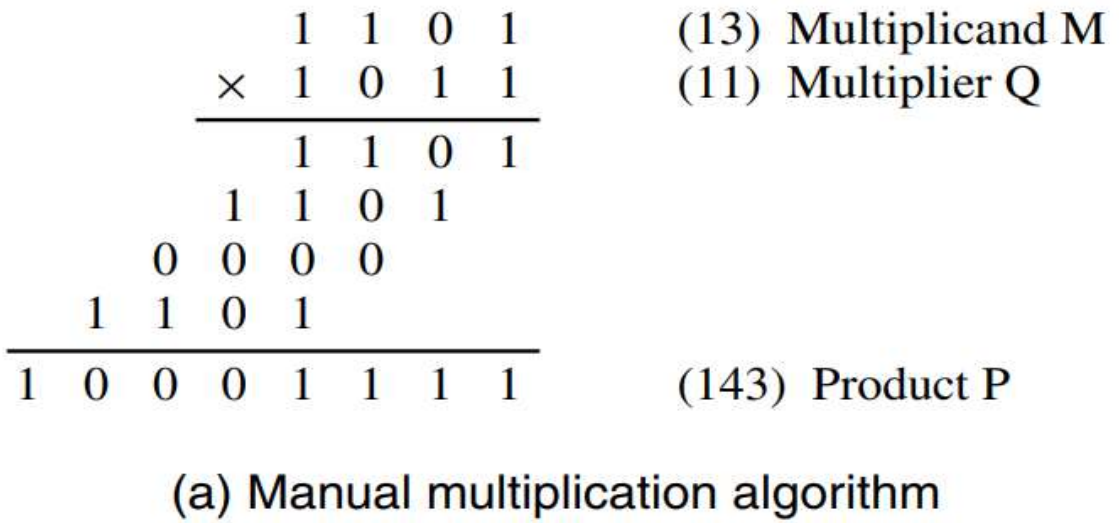
+ A3B2 A2B2 A1B2 A0B2

+ A3B3 A2B3 A1B3 A0B3

P7 P6 P5 P4 P3 P2 P1 P0 - Products

Where (P7,P6……P0) are product terms generated by adding the partial products as below

- P0 = A0B0
- P1 = A1B0 + A0B1 + Carryout of P0
- P2 = A2B0 + A1B1 + A0B2 + Carryout of P1
- P3 = A3B0 + A2B1 + A1B2 + A0B3 + Carryout of
- P4 = A3B1 + A2B2 + A1B3 + Carryout of P3
- P5 = A3B2 + A2B3 + Carryout of P4
- P6 = A3B3 + Carryout of P5
- P7 = Carryout of P6

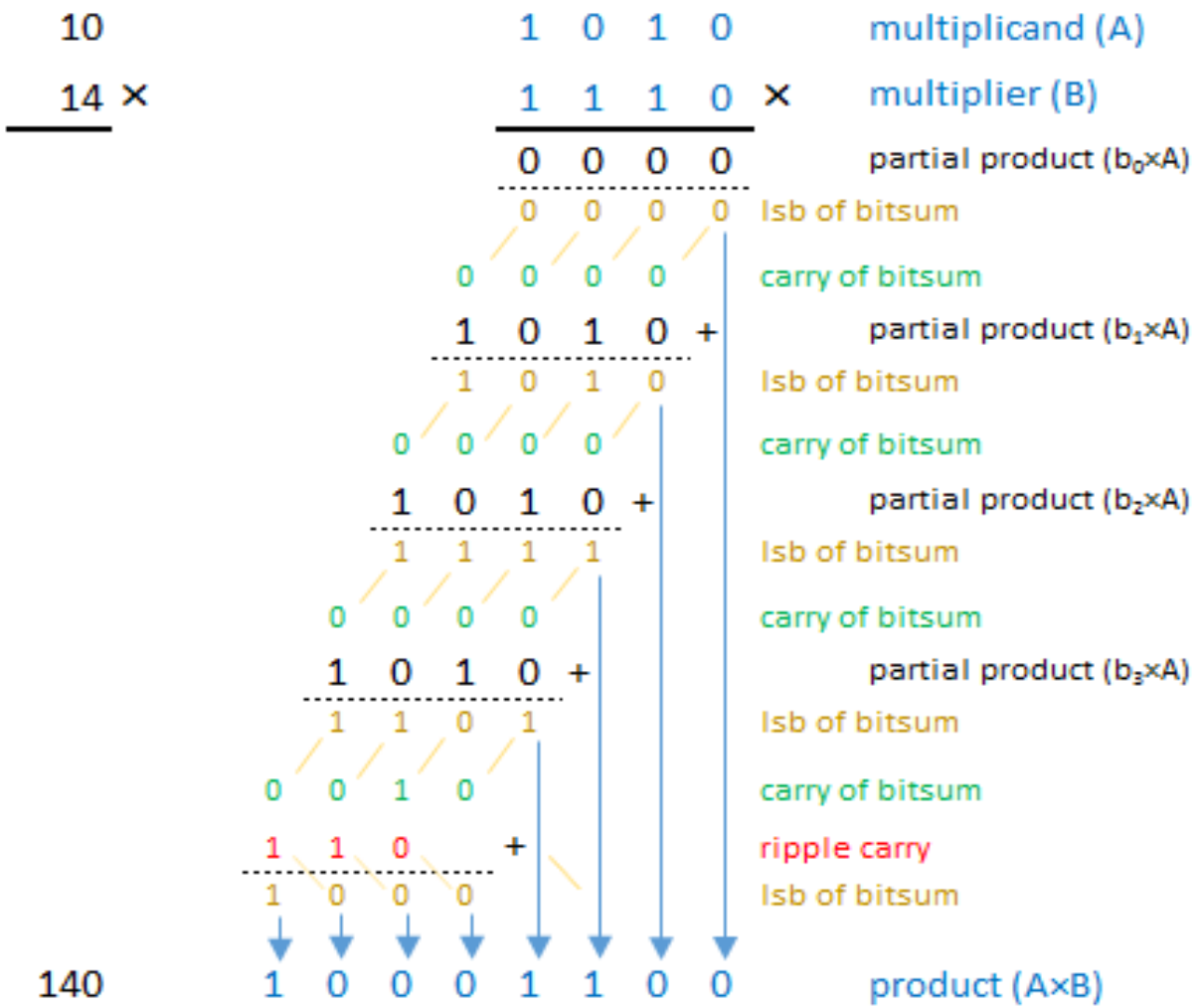


(a) Manual multiplication algorithm

Array multiplier

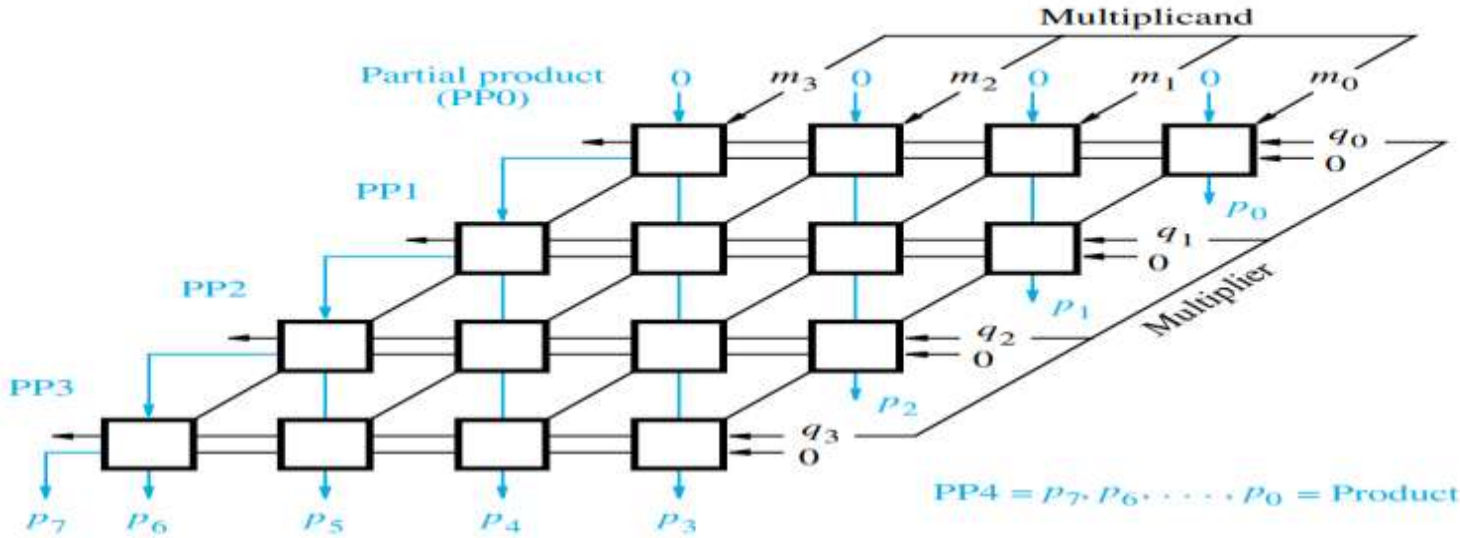
- Array multiplier is well known due to its regular structure. Multiplier circuit is based on add and shift algorithm.
- Each partial product is generated by the multiplication of the multiplicand with one multiplier bit. The partial product are shifted according to their bit orders and then added.

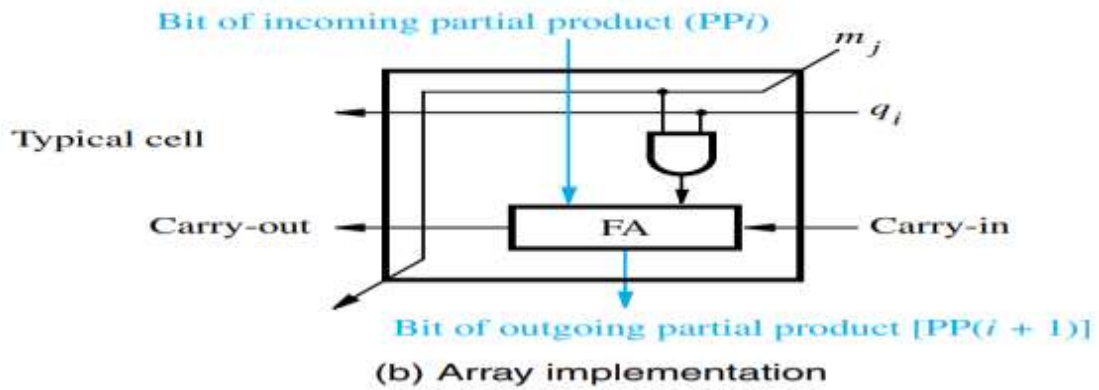
The addition can be performed with normal carry propagate adder. N-1 adders are required where N is the multiplier length.



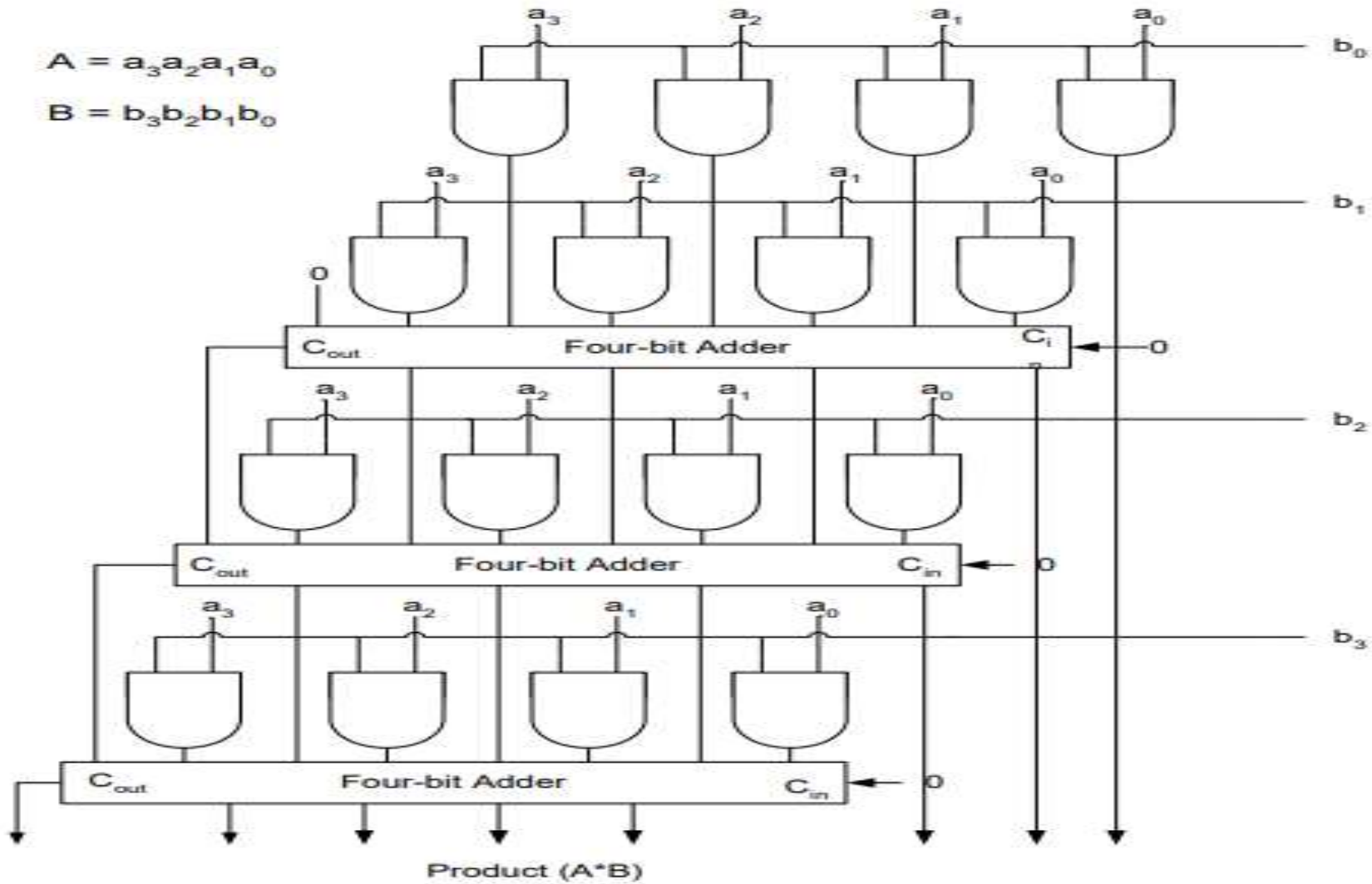
				A3	A2	A1	A0	Inputs
			x	B3	B2	B1	B0	
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0	
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0		
		C	sum	sum	sum	sum		
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0			Internal Signals
	C	sum	sum	sum	sum			
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0				
C	sum	sum	sum	sum				
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Outputs

Binary Multiplication product terms can be implemented by using cells consisting of
a) AND gate – Used for producing Partial Products (PP) qi.mj &
b) Full Adder (FA) – Used to produce product terms by summation of PPs





(b) Array implementation


A = a3a2a1a0
B = b3b2b1b0
a3
a2
a1
a0
b0
b1
b2
b3
0
Cout
Four-bit Adder
Ci
Cin
Product (A*B)

Advantages:

- Array Multipler is a Parallel Multiplier where the worst time propagation delay is through critical
- path with 6(n-1)-1 gate delay including the initial AND gate delay in all cells, for an n × n array.

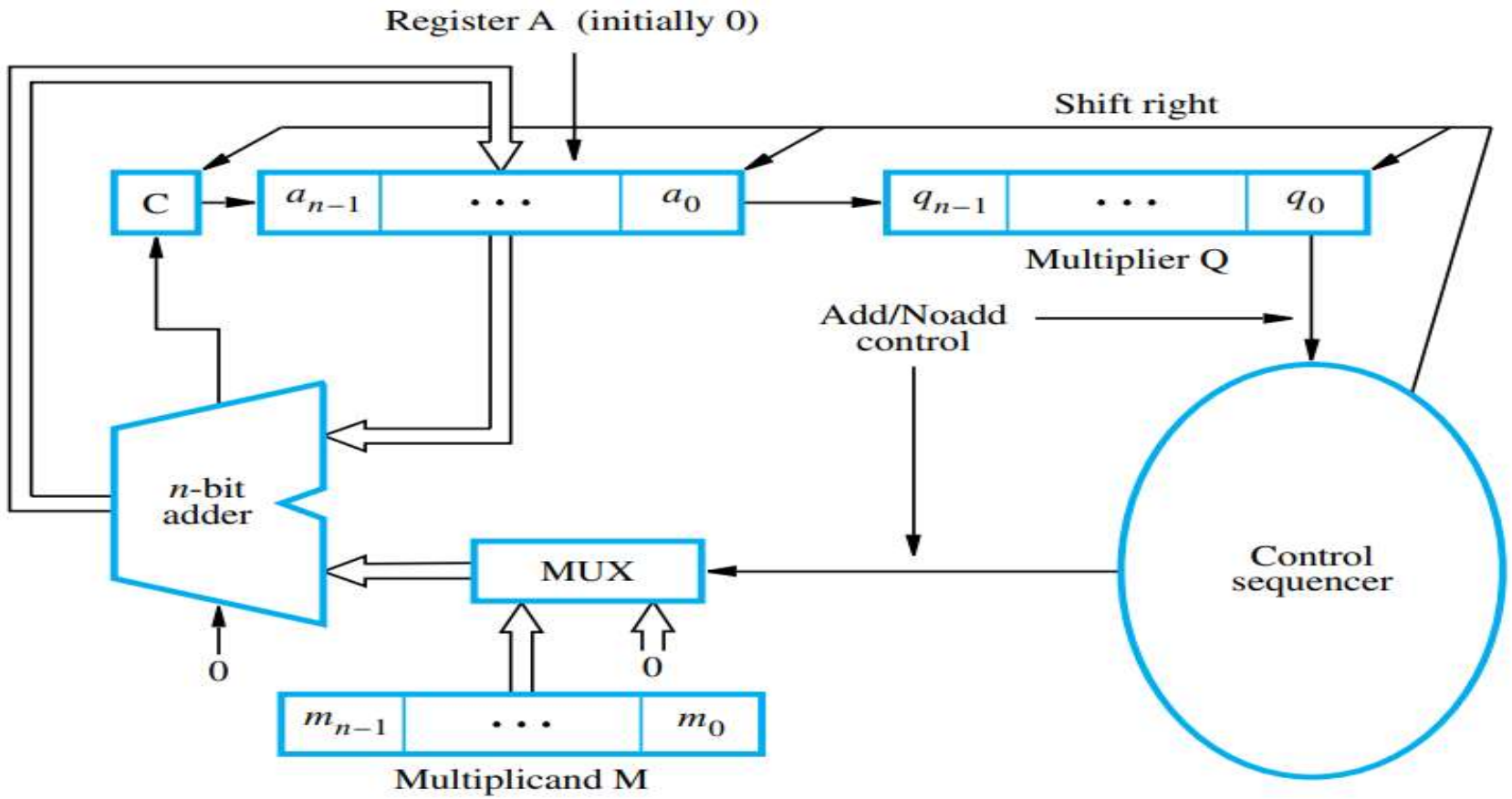
Drawback:

- Hardware resource used is high i.e., it uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers

Sequential Circuit Multiplier

- This circuit performs multiplication by using
- Single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders in array multiplier.
- Registers A and Q are shift registers, concatenated as shown, they hold partial product PPi while multiplier bit qi generates the signal Add/No add.

- This signal causes the multiplexer MUX to select 0 when qi = 0, or to select the multiplicand M when qi = 1, to be added to PPi to generate PP(i + 1). The product is computed in n cycles. Initially Register A is loaded with PPi as 0s (n bit 0s)
- The Sequential Circuit Multiplier is as shown in Figure

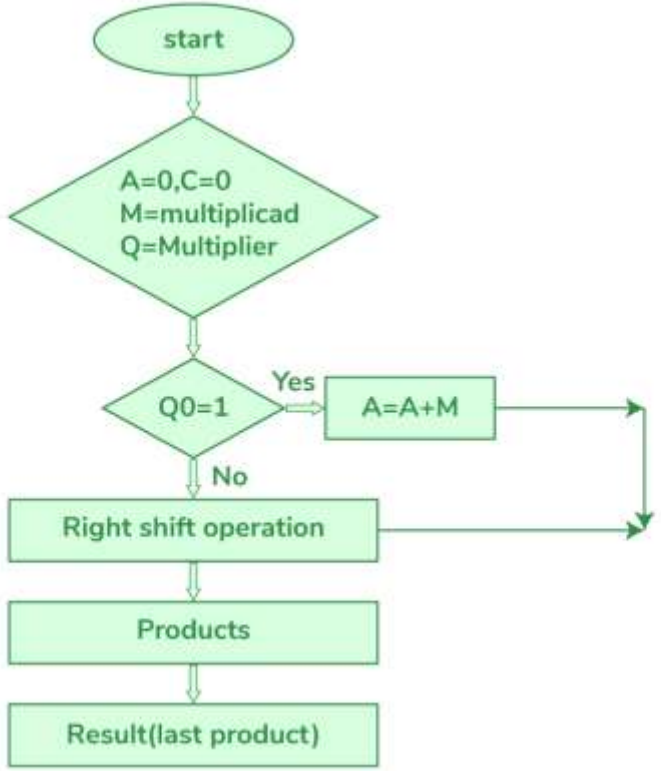


(a) Register configuration

Sequential Circuit Multiplier

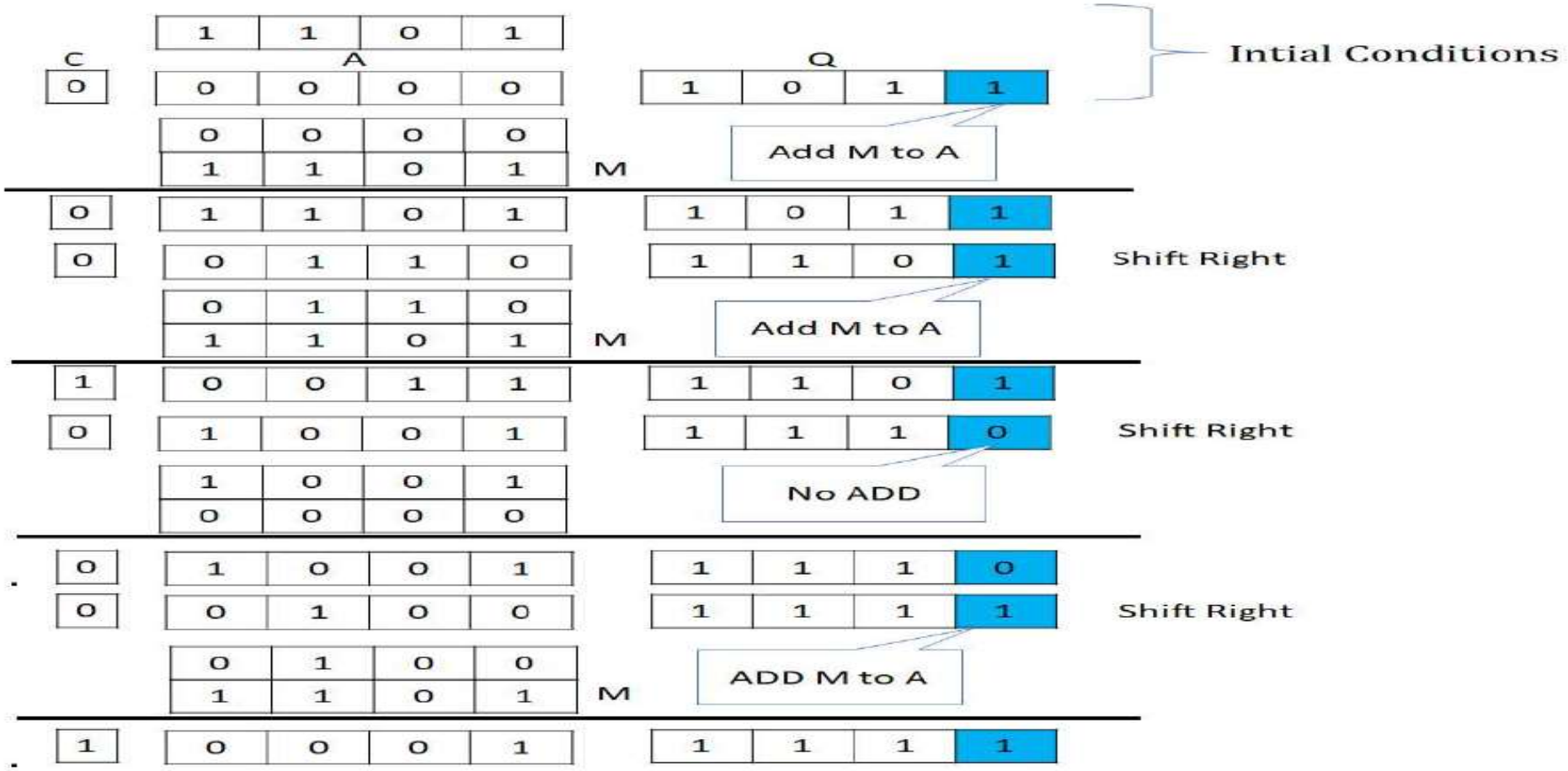
The flow chart explains the whole operation of the sequential binary multiplier in a simple manner. First, assign 0 to the accumulator(A) and carry(C) values. then check the LSB of Q(multiplier) i.e Q_0, if q0 is 0 then perform only the right shift operation and if q0 is 1 then perform the addition of the accumulator and multiplicand, store the result in the accumulator then perform the right shift operation.

We have to continue this process based on the number of bits in the multiplier.



- Example: Let us consider two 4 bit number multiplication (1 1 0 1)2 X (1 0 1 1)2

```
          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
```

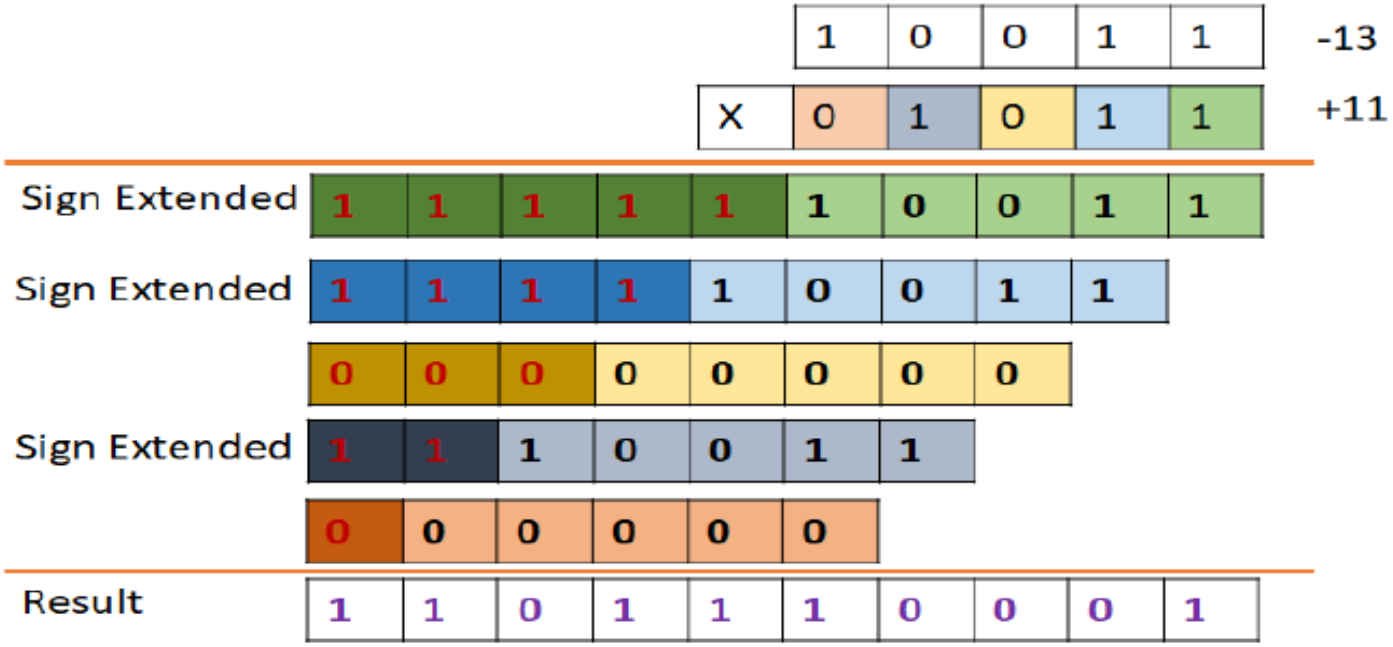


Multiplication of Signed Numbers

- Case (1): Consider a positive multiplier and a negative multiplicand.

When Multiplier bit is 1, the Multiplicand is added to Partial Product (PPi). In this case, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend.

- Example: Let us consider a 5-bit signed operands: – A multiplicand: (-13) and a multiplier: (+11) produces 10-bit product, (−143). The Multiplication is



Case II : Both are non negative numbers (+ve) ,then same multiplication as using unsigned numbers

- Case III : Consider a Negative Multiplier and a Negative Multiplicand

In this case, need to take 2’s-complements of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.

This is possible because complementation of both operands does not change the value or the sign of the product.

Case IV : both negative and negative take 2’s complement of both(-3*-7 = +3 * +7)

Booths multiplication algorithm

- The Booth Algorithm treats both positive and negative 2's complement n-bit operands uniformly.
- Basic principle used in Booth Algorithm is to reduce the number of operations is as follows A multiplier may be positive / negative.
- In Booth Algorithm, the multiplier is modified such that the number of operation is reduced as follows.

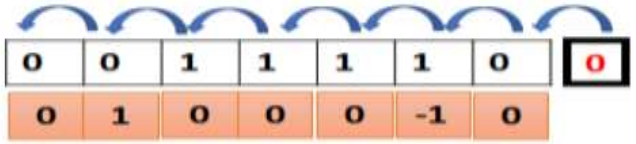
Booth Recoding of Multiplier

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	$0 \times M$

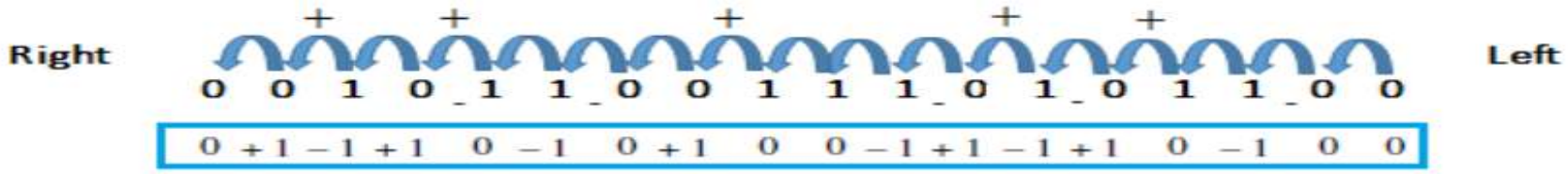
In general, in the Booth algorithm, the recoding of the Multiplier is done as follows

- −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.

Example1: Recoding of Multiplier (0 0 1 1 1 1 0)2

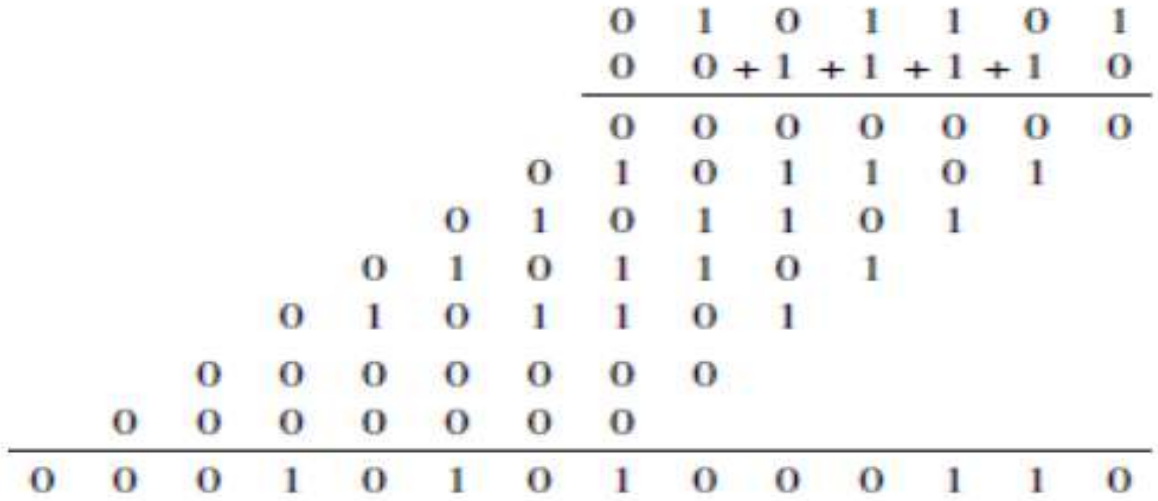


Example 2: Recoding of Multiplier (0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0)2

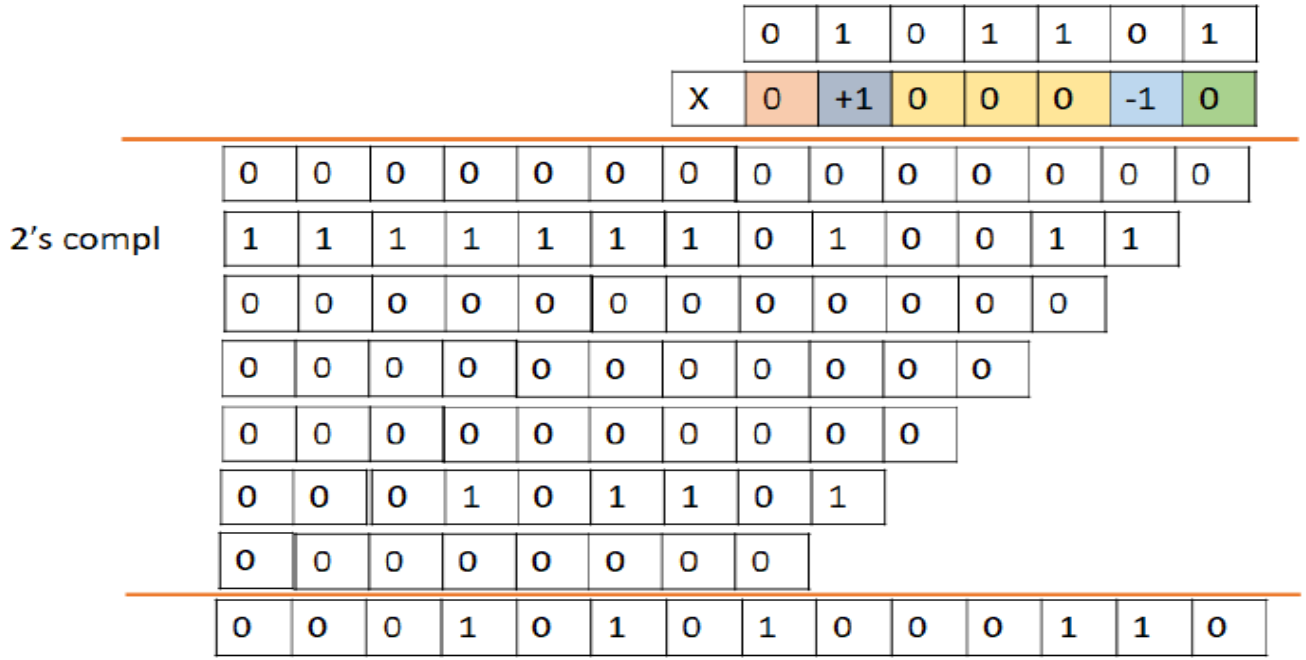


Let us take a Multiplicand – (0101101) and Multiplier (0011110)

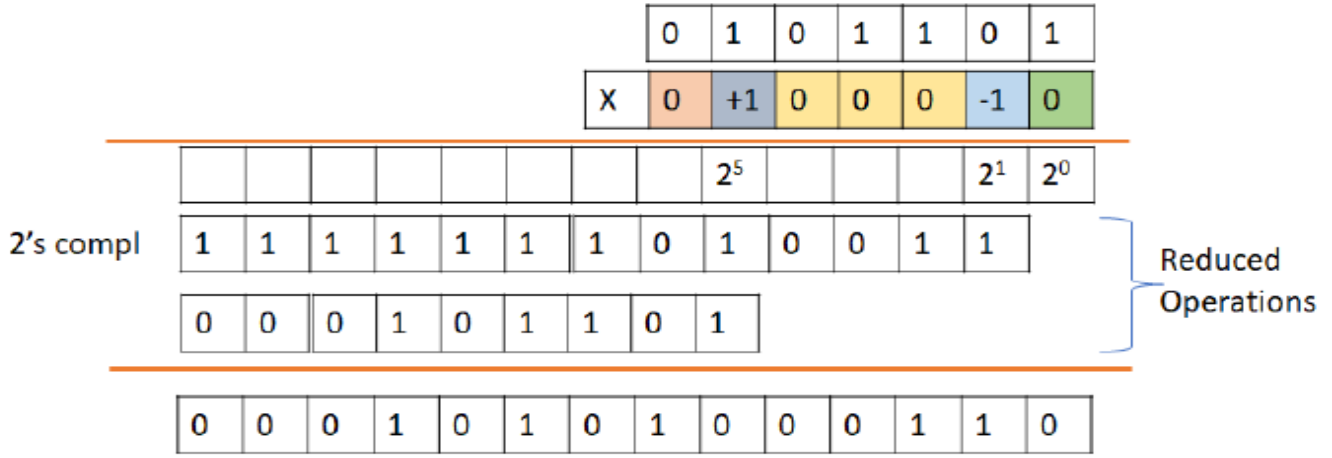
Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure?



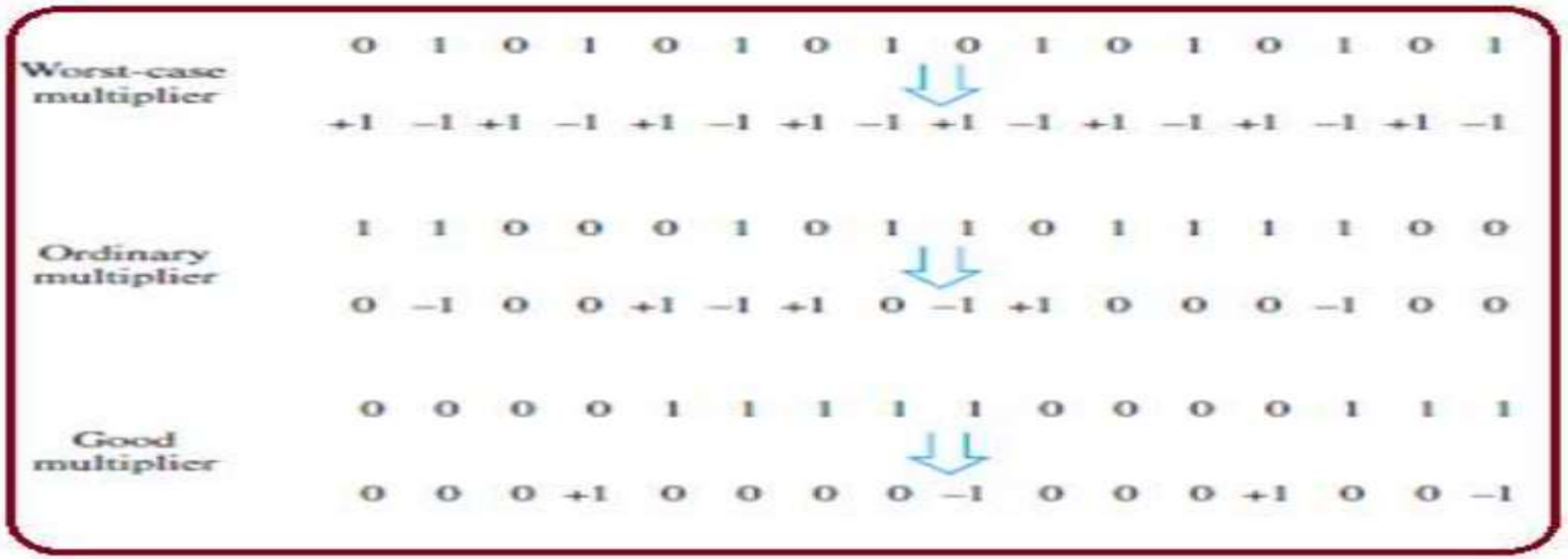
in case -1 occurs takes the 2's- complement of 21 times the multiplicand.



This leads to reduced operations



- Best case – a long string of 1's (skipping over 1s)
- Worst case – 0's and 1's are alternating



Fast Multiplication

- There are two techniques discussed for speeding the Multiplication Operation

i. Reducing Maximum number of Summands: - Bit Pair Recoding of Multipliers reduces the maximum number of summands (versions of multiplicand) that must be added to n/2 for n bit operands.

ii. Faster Summands addition: - Summand addition uses

- a) Carry save – In this carry generated by Full Adders in ith row propagated to (i+1)th row Full Adders instead of rippling through adder in same row and
- b) Summands Reduction Technique - Techniques like 3 to 2, 4 to 2 reduction of summands .

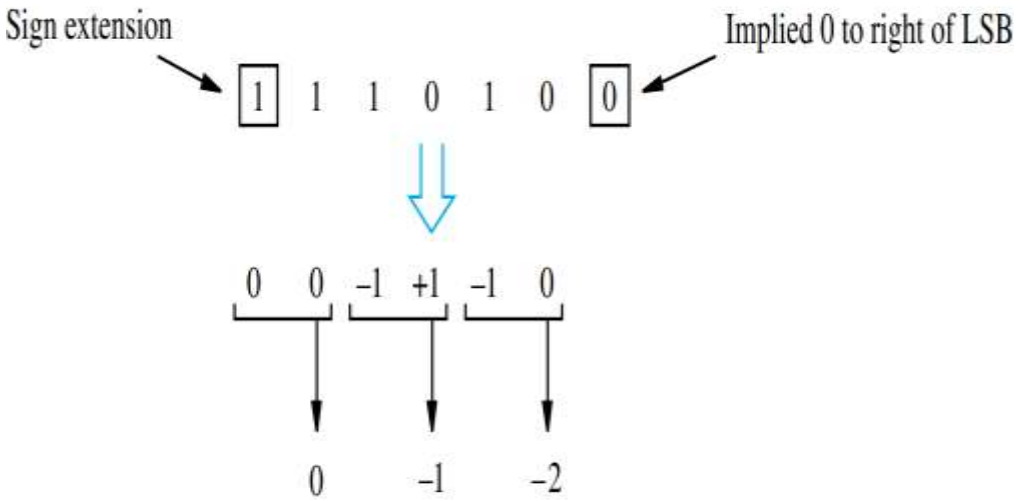
Reducing Maximum number of Summands using Bit Pair Recoding of Multipliers

- Bit-pair recoding of the multiplier – It is a modified Booth Algorithm, In this it uses one summand for each pair of booth recoded bits of the multiplier. (reduce the multiplier bits if n bits are there in multiplier we get n summands , n summands are reduced to n/2)

Step 1: Convert the given Multiplier into a Booth Recode the Multiplier . In booths recoding we get 2 bits i.e..bit i &bit i-1, but in bit pair reducing we get 3 bits i.e.. i-1,I,i+1

Step 2: Group the recoded Multiplier bits in pairs and observe the following

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	$0 \times M$



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

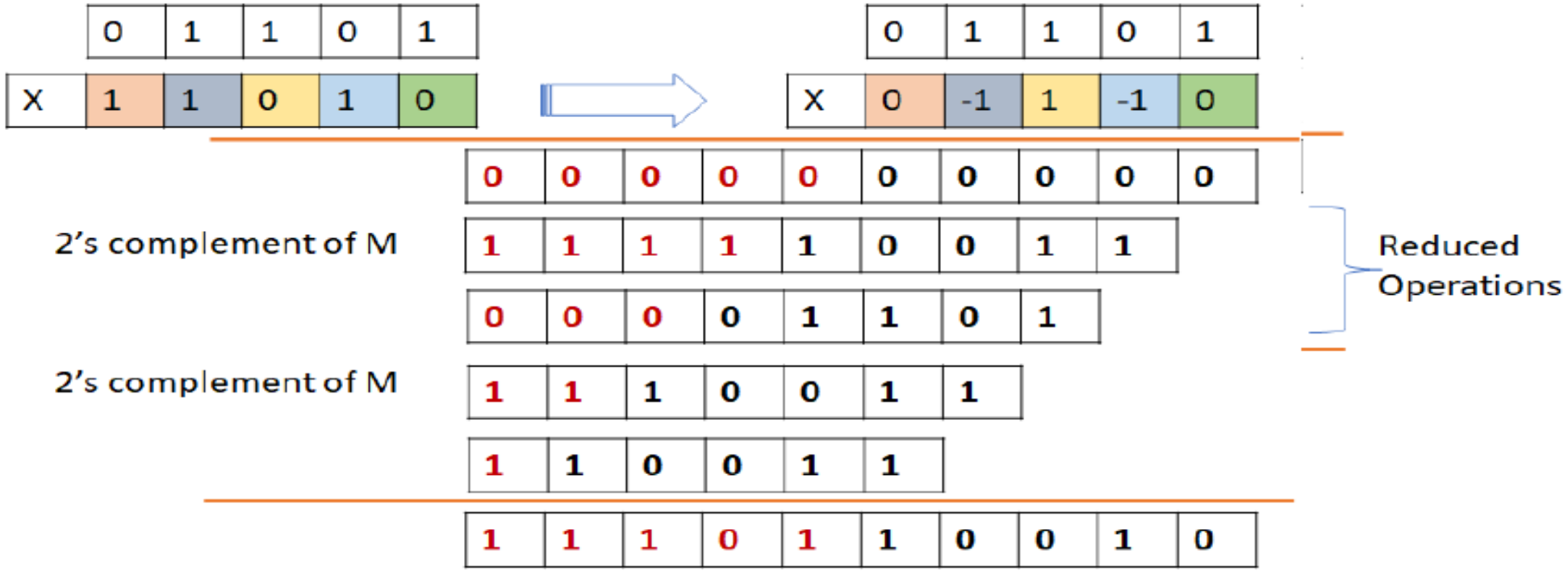
Here -1 *M means 2's complement of M
+2*M means first place 0 in LSB and multiply M with 1
-2*M means first place 0 in LSB and 2's complement of M

- Now let us see how bit pair Recoding Multiplier (Modified Booth) reduces the number of summands in comparison with Booth Recoding Multiplier algorithm

Example : Multiplication of (+13) = (01101)2 by (-6) = (11010)2

By Booth Recoding of Multiplier (1 1 0 1 0) = (0 -1 +1 -1 0)

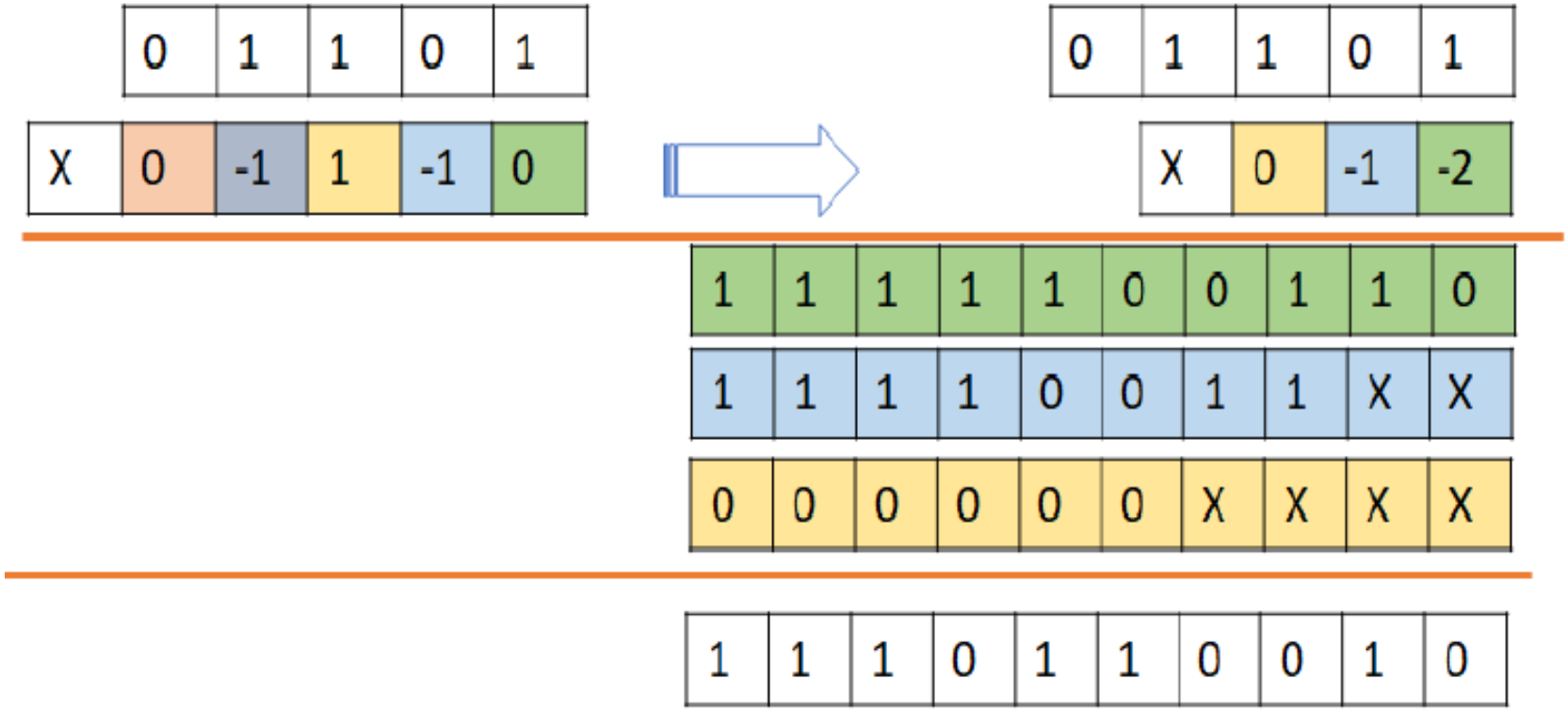
Booth's multiplication(for understanding)



By Bit Pair Booth Recoding of Multiplier (this is main solution for problem)

(0 -1 +1 -1 0) → By bit pairing 0 [-1 +1][-1 0]→ 0 [-21 + 20] [-21 + 0] → 0 [-1] [-2]

- Now the Multiplicand is multiplied by Bit pair recoded multiplier (0, -1, -2)

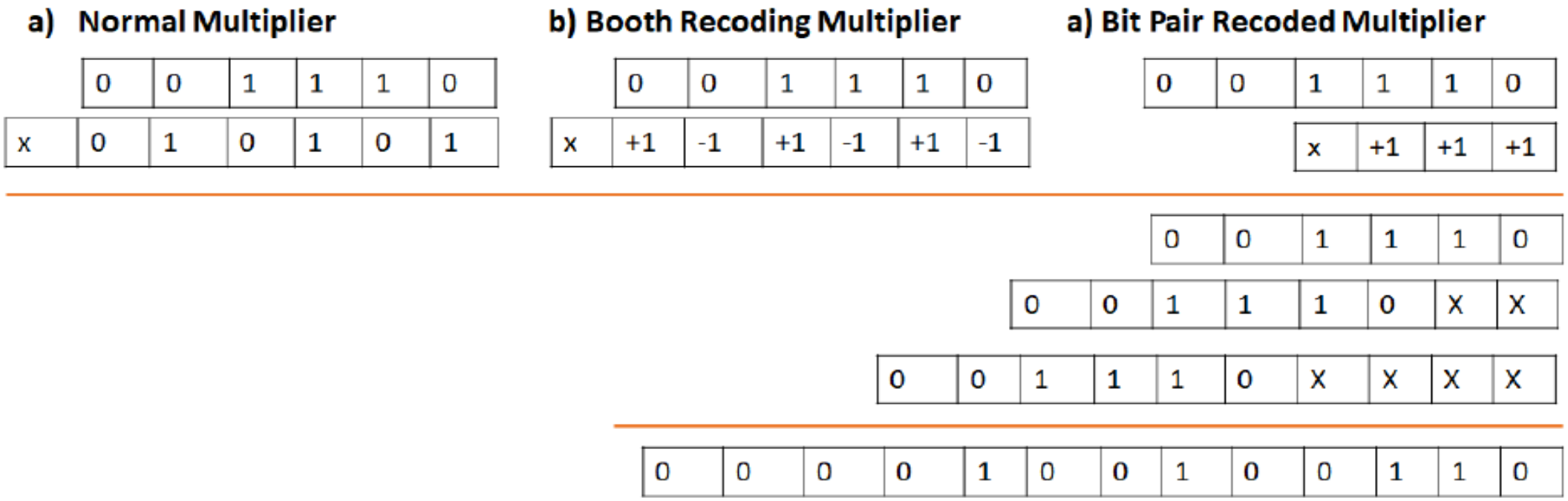


Example 2: The worst case of Booth Recoding Multiplier.

- In this example the given Multiplier is (0 1 0 1 0 1) and Multiplicand is (0 0 1 1 1 0).
- The recoding of Normal Multiplier using Booth Recoding and Bit pair is as shown below

[0 1 0 1 0 1] ---> [+1 -1 +1 -1 +1 -1 +1] --- >[(+1,-1) (+1,-1) (+1,-1)] --> [+1 +1 +1]

- It is clear the worst case of booth recoded Multiplier is also reduced to n/2 summands in Bit pair Booth algorithm



Integer Division

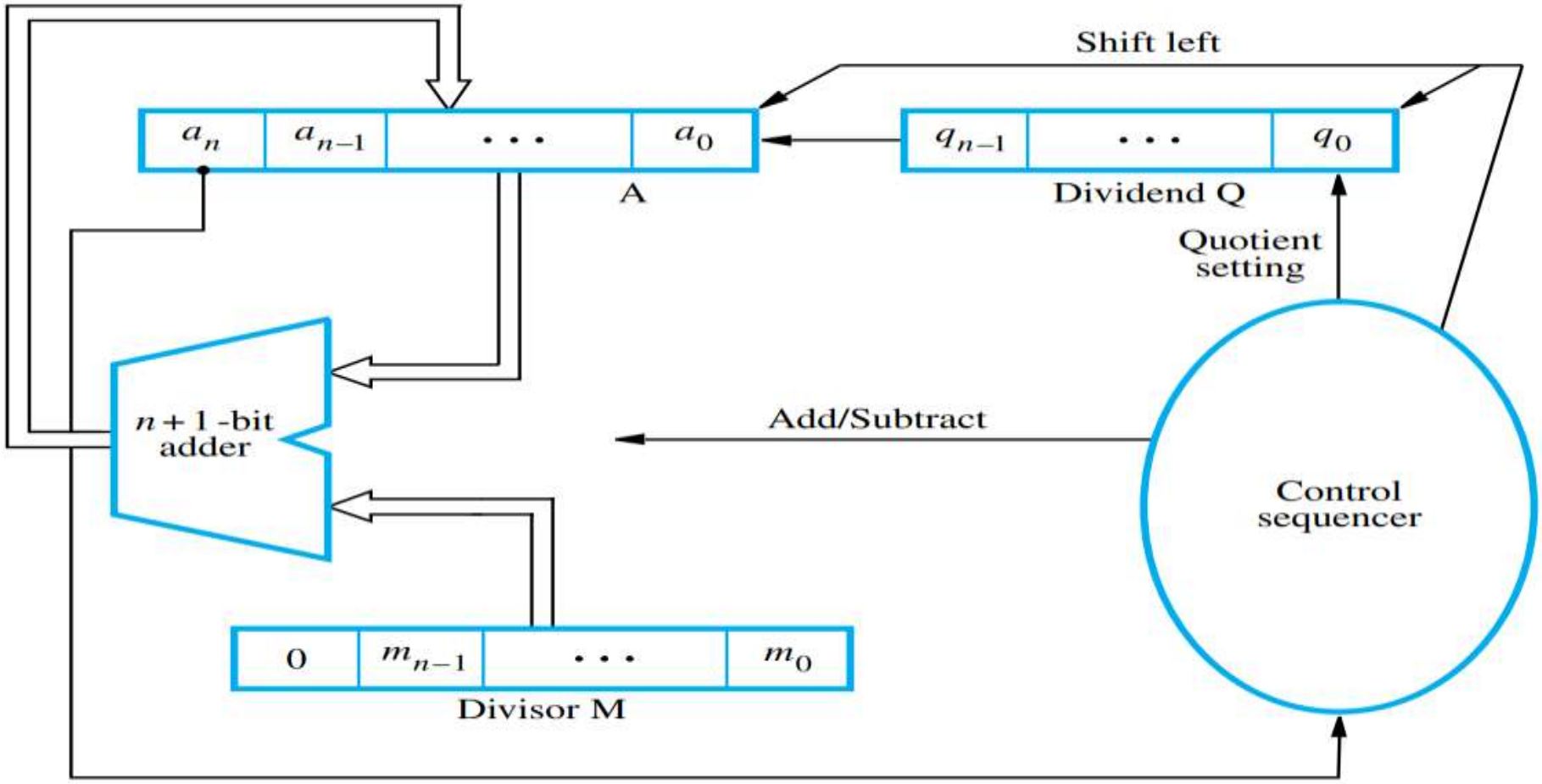
examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27.

We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that $27 - 26 = 1$ is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1.

```
      21
  ____
13 ) 274
     26
     --
     14
     13
     --
      1
```

```
          10101
      ________
1101 ) 100010010
       1101
       ----
       10000
        1101
        ----
        1110
        1101
        ----
           1
```

Circuit arrangement for binary division



Restoring Division

Step 1: In this step, the corresponding value will be initialized to the registers, i.e., register A will contain value 0, register M will contain Divisor, register Q will contain Dividend, and N is used to specify the number of bits in dividend.

Step 2: In this step, register A and register Q will be treated as a single unit, and the value of both the registers will be shifted left.

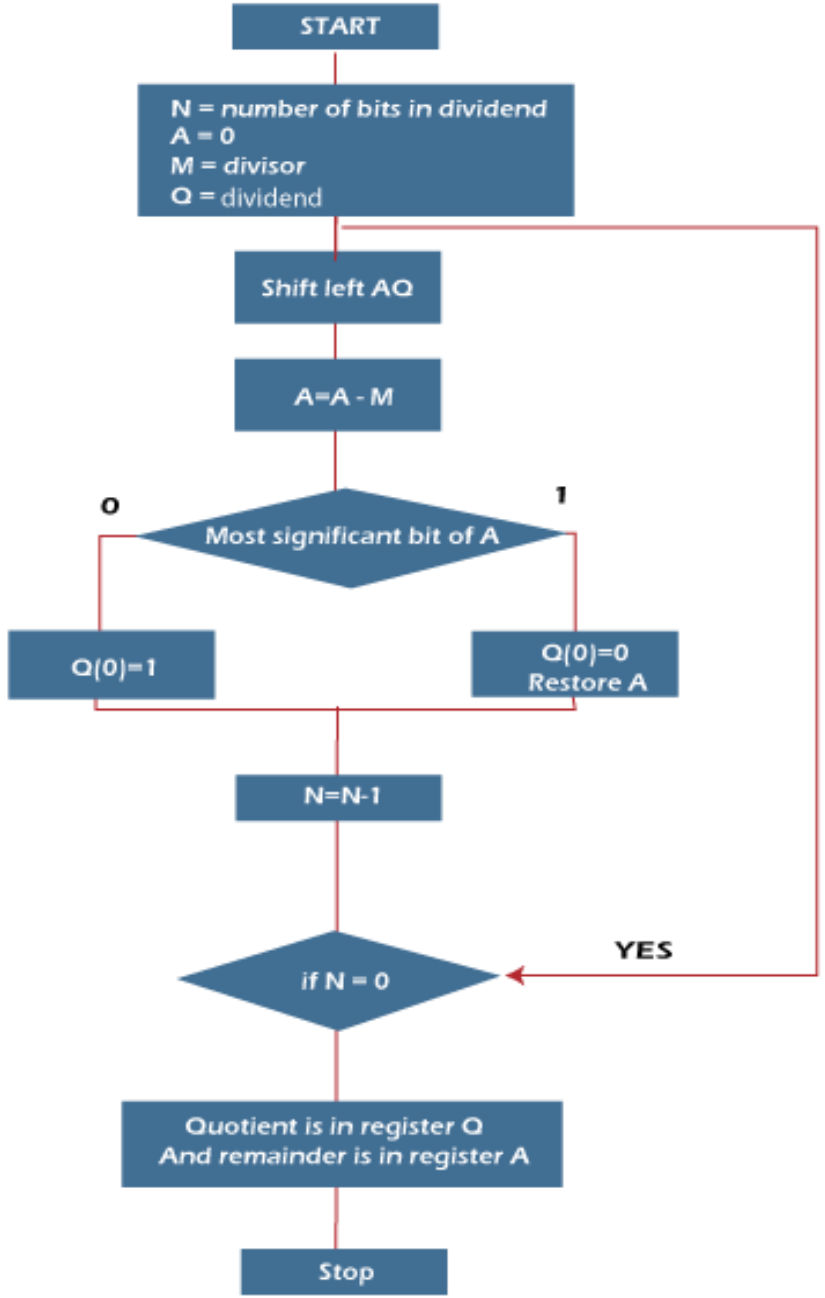
Step 3: After that, the value of register M will be subtracted from register A. The result of subtraction will be stored in register A.

Step 4: Now, check the most significant bit of register A. If this bit of register A is 0, then the least significant bit of register Q will be set with a value 1. If the most significant bit of A is 1, then the least significant bit of register Q will be set to with value 0, and restore the value of A that means it will restore the value of register A before subtraction with M.

Step 5: After that, the value of N will be decremented. Here n is used as a counter.

Step 6: Now, if the value of N is 0, we will break the loop. Otherwise, we have to again go to step 2.

Step 7: This is the last step. In this step, the quotient is contained in the register Q, and the remainder is contained in register A.



```
        1 0
    ---------
1 1 ) 1 0 0 0
      1 1
      -----
        1 0
```

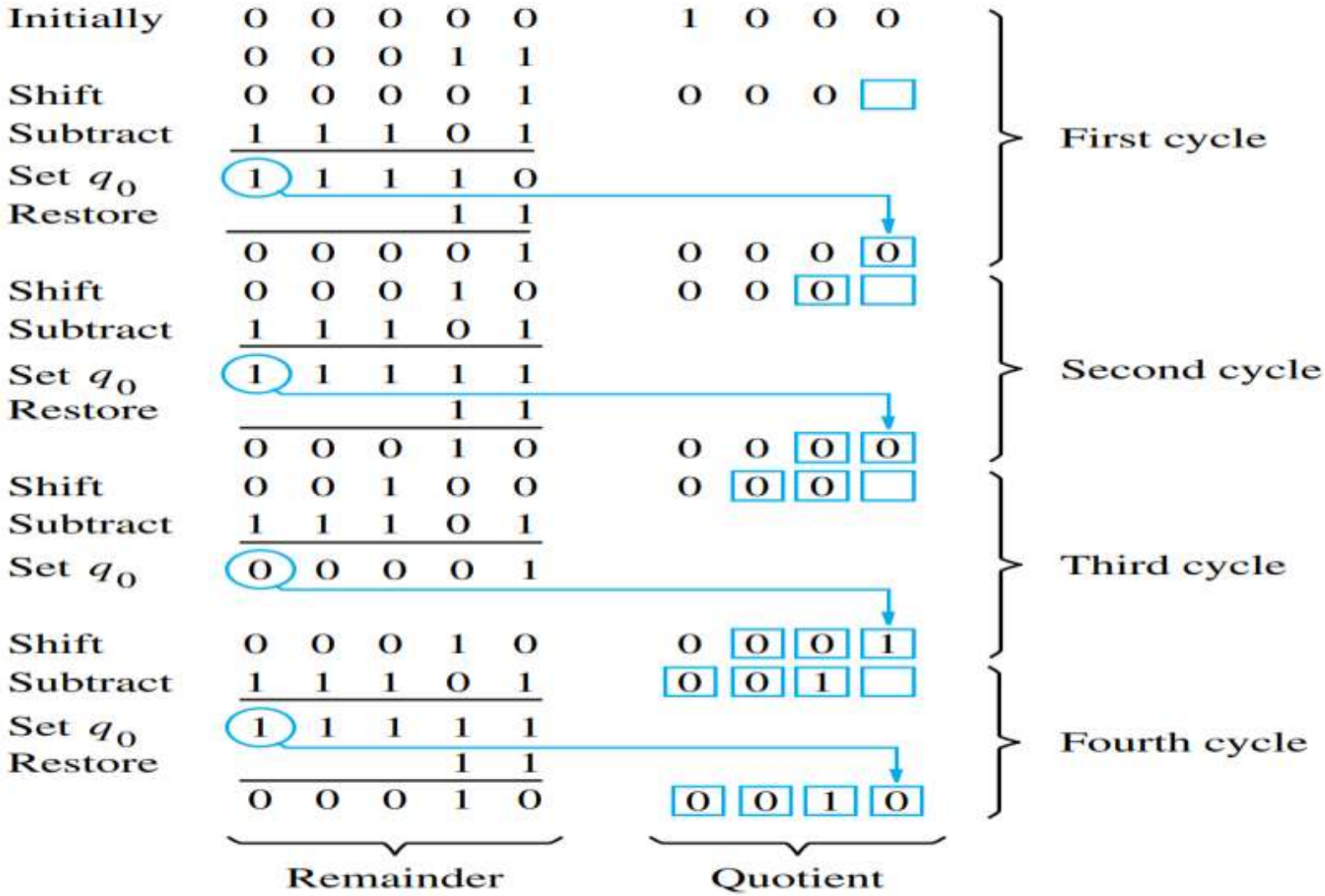
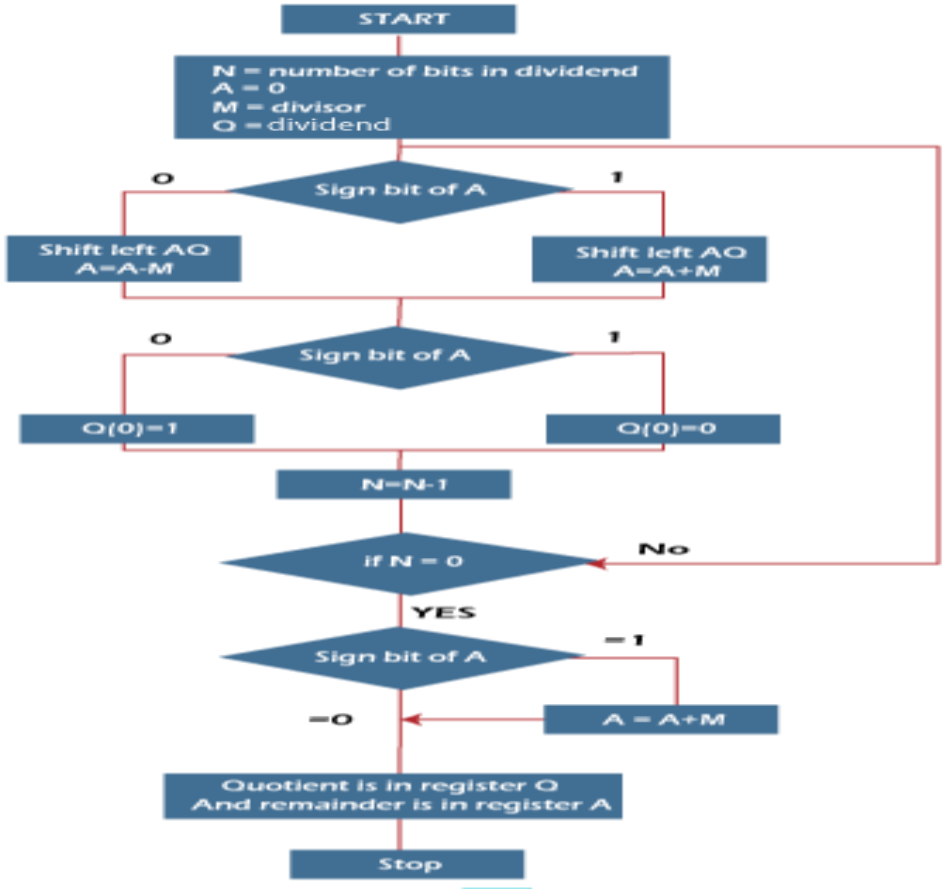


Figure 9.24 A restoring division example.

non-restoring division algorithm

The non-restoring division algorithm is more complex as compared to the restoring division algorithm. But when we implement this algorithm in hardware, it has an advantage, i.e., it contains only one decision and addition/subtraction per quotient bit. After performing the subtraction operation, there will not be any restoring steps. Due to this, the numbers of operations basically cut down up to half. Because of the less operation, the execution of this algorithm will be fast.

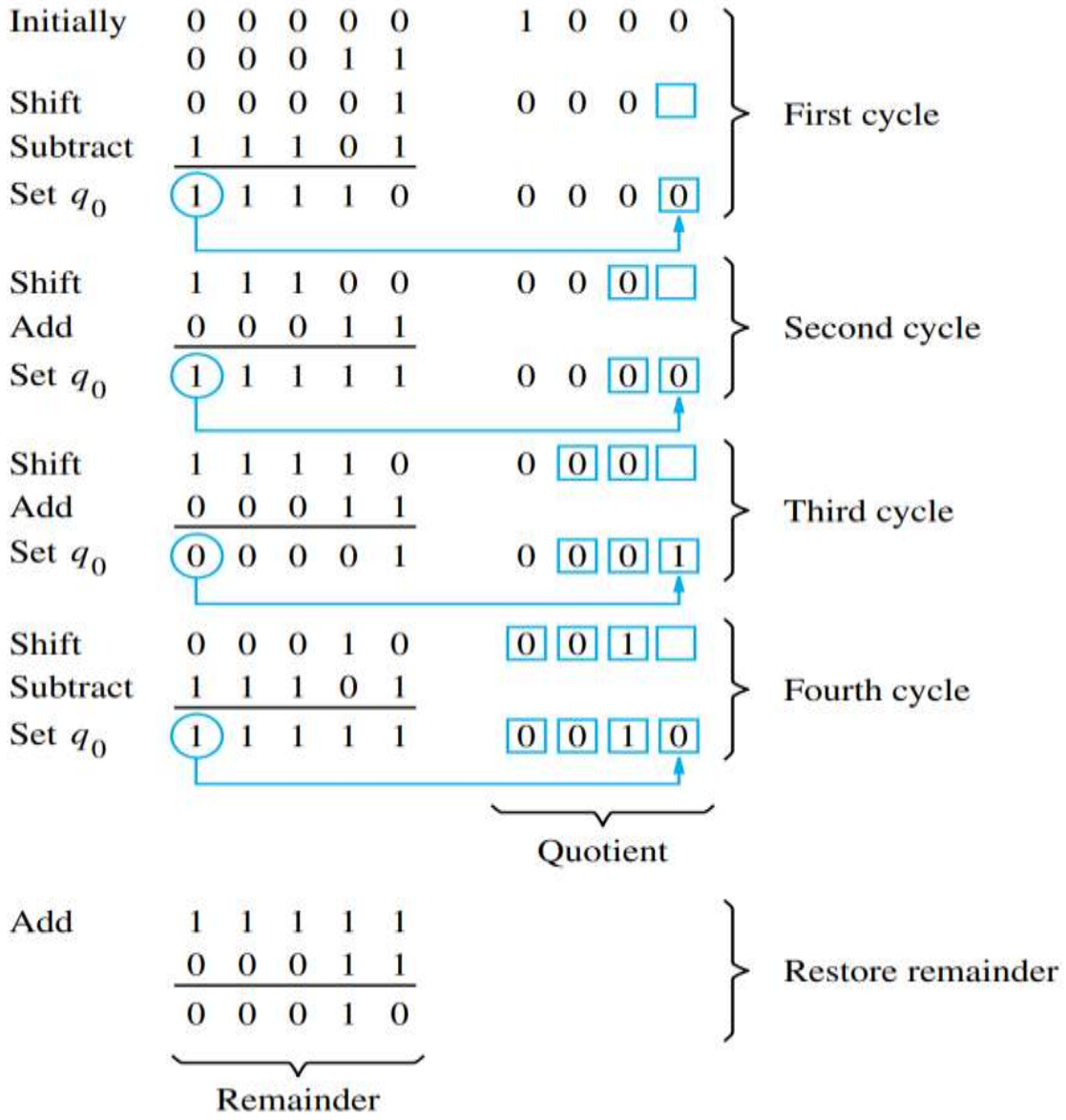


- Step 1: (Repeat *n* times)

➢ 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.

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

- Step2: If the sign of A is 1, add M to A



Floating-Point Numbers and Operations

Go to unit 1 chapter 1 last topic

Arithmetic Operations on Floating-Point Numbers

When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ.

Consider a decimal example in which we wish to add 2.9400 × 10^2 to 4.3100 × 10^4. We rewrite 2.9400 × 10^2 as 0.0294 × 10^4 and then perform addition of the mantissas to get 4.3394 × 10^4.

The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

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

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127 to maintain the excess-127 representation.

2. Divide the mantissas and determine the sign of the result.

3. Normalize the resulting value, if necessary.

Implementing Floating-Point Operations

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines.

In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers.

In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware

An example of the implementation of floating-point operations is shown in Figure

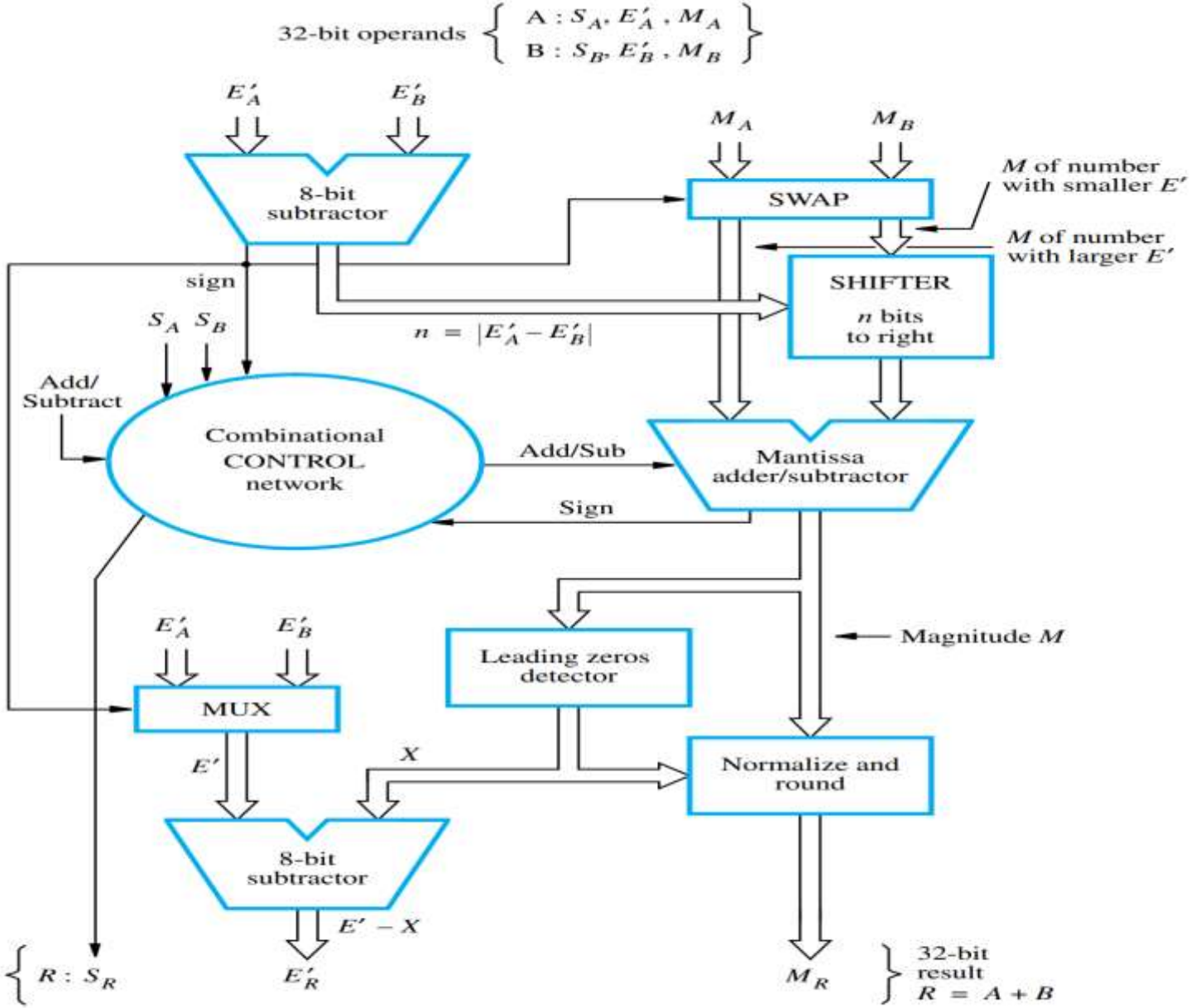


Figure 9.28 Floating-point addition-subtraction unit.