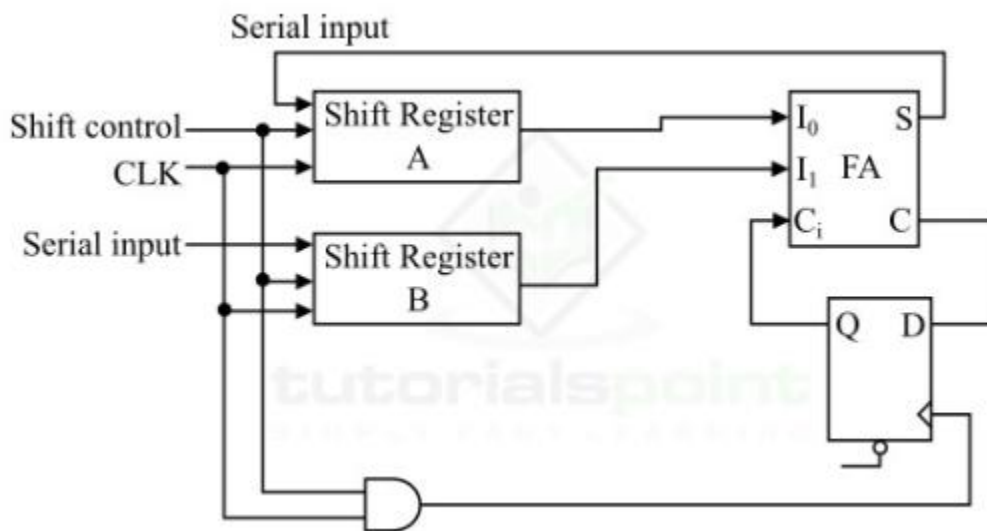


UNIT 1: Computer Arithmetic (10 Lectures)

Serial adder, Parallel adder, Ripple carry adder, Carry look-ahead adder, Multiplication of signed and unsigned numbers, Booth's algorithm, Division of integer, Floating point arithmetic.

Serial Adder

The **serial binary adder** or **bit-serial adder** is a synchronous digital circuit that performs binary addition bit by bit.



In the serial binary adder circuit, the binary digits (bits) are added one pair at a time using a full-adder circuit. The carry generated from the full adder is transferred to a D flip flop. Thus, the output of this D flip flop is then used as the carry input for the next pair of significant bits. The sum bit S is transferred to a third shift register. Now, let us understand the operation of the serial binary adder in the detail.

At beginning, the shift register A stores the augend bit of the given binary number, and the shift register B stores the addend bit. Initially, the D flip flop is cleared to 0, hence no carry bit is present. The outputs of shift registers A and B supply a pair of significant bits to the full adder circuit at inputs I_0 and I_1 . A shift control is used to enable the shift registers A and B and the carry flip-flop.

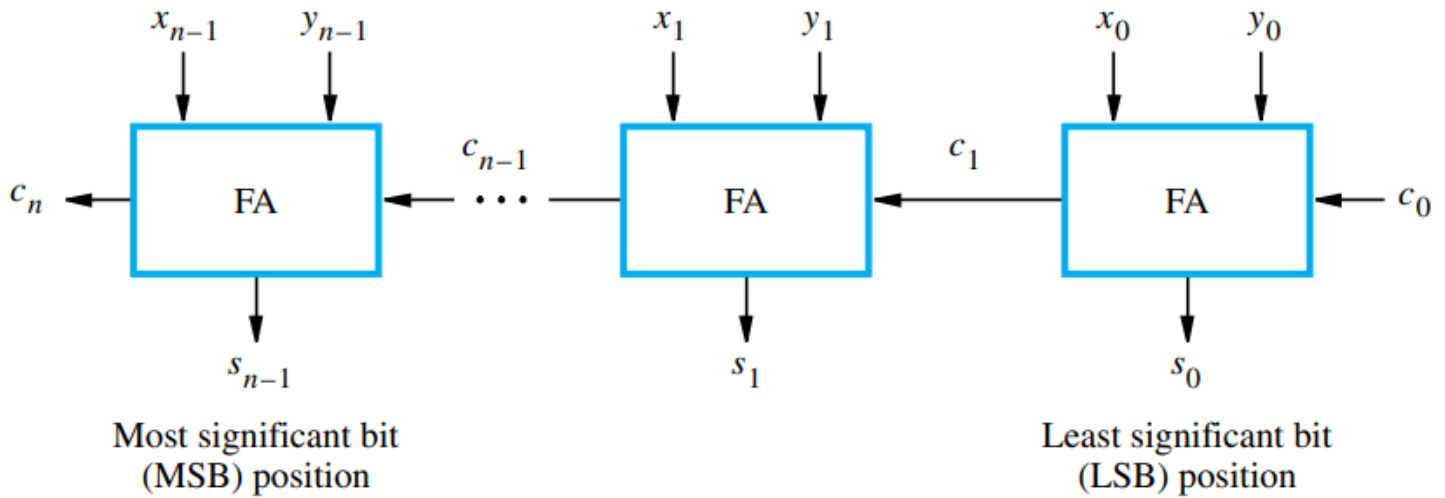
Therefore, at each clock pulse, the registers A and B are shifted to the right, and the sum bit from output S of the full adder circuit enters the left most of the shift register A. Thus, for each succeeding clock pulse a new sum bit is transferred to the shift register A and a new carry bit is transferred to the output Q of the D flip flop. This process continues until the shift control is disabled.

Hence, the addition of two binary numbers in serial form is accomplished by supplying a pair of bits together with the previous carry to a full adder circuit, and transferring a sum bit at a time into the shift register A.

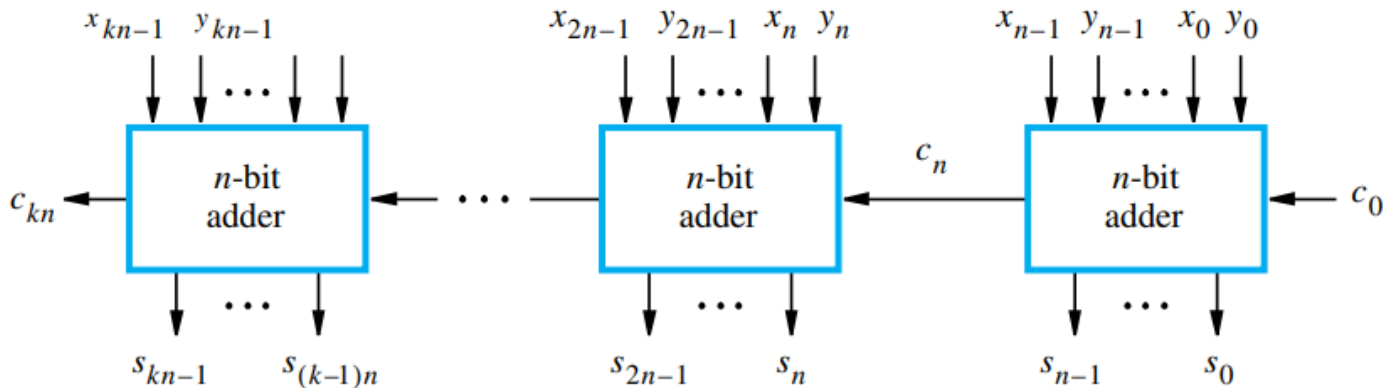
It is slower because of the sequential operation.

Parallel Adder:

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



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 c_n is not part of the answer. 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 c_n 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.

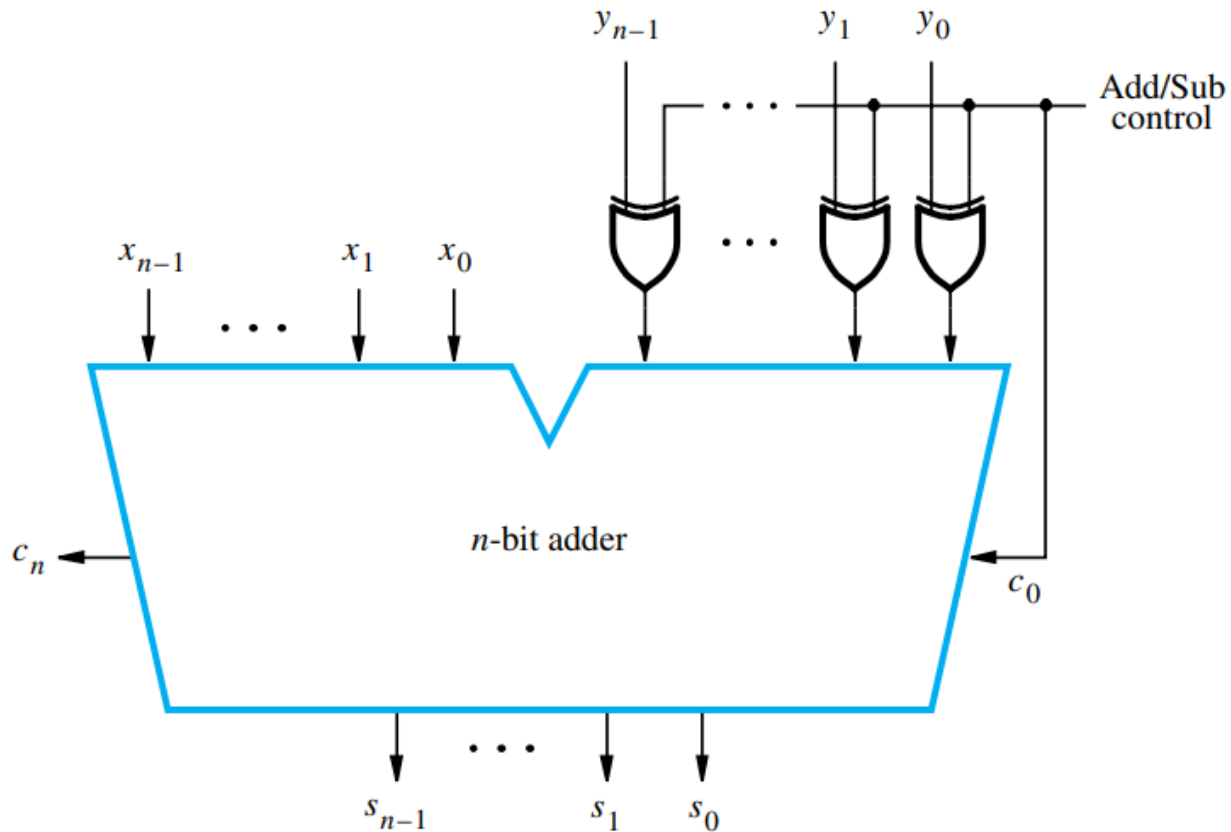


Figure 2.2

Binary addition/subtraction logic circuit.

Carry look-ahead adder:

The delay through any combinational circuit constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the circuit. In the case of the n -bit ripple-carry adder, the longest path is from inputs x_0 , y_0 , and c_0 at the LSB position to outputs c_n and s_{n-1} at the most-significant-bit (MSB) position. If a ripple-carry adder is used to implement the addition/subtraction, all sum bits are available in $2n$ gate delays, including the delay through the XOR gates on the Y input. Using the implementation $c_n \oplus c_{n-1}$ for overflow, this indicator is available after $2n + 2$ gate delays.

To reduce delay in adders, carry-look-ahead networks are used.

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

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

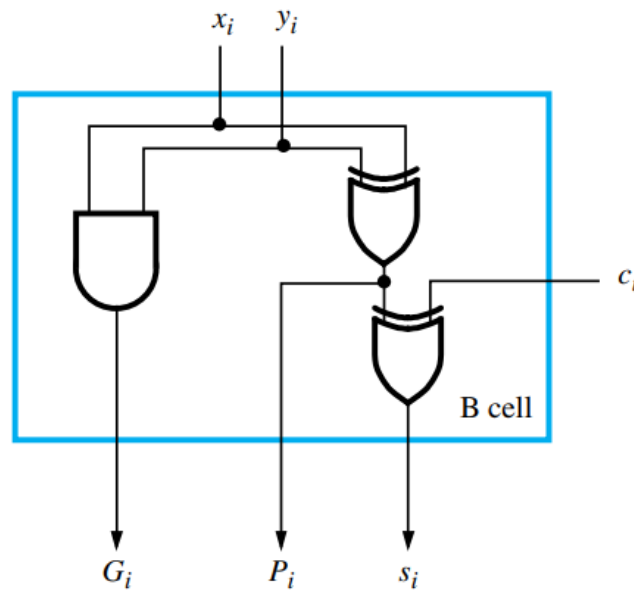
Factoring the second equation into $c_{i+1} = x_i y_i + (x_i + y_i) c_i$ we can write

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

The expressions G_i and P_i are called the generate and propagate functions for stage i .

COM0700504MJ: COMPUTER ARCHITECTURE UNIT-I

If the generate function for stage i is equal to 1, then $c_{i+1} = 1$, independent of the input carry, c_i . This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1. All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay after the X and Y operands are applied to the inputs of an n -bit adder. Each bit stage contains an AND gate to form G_i , an OR gate to form P_i , and a three-input XOR gate to form s_i . A simpler circuit can be derived by observing that an adequate propagate function can be realized as $P_i = x_i \oplus y_i$, which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$. But, in this case $G_i = 1$, so it does not matter whether P_i is 0 or 1. Then, using a cascade of two 2-input XOR gates to realize the 3-input XOR function for s_i , the basic B cell as shown below can be used in each bit stage.



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

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

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

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0$$

Thus, all carries can be obtained three gate delays after the input operands X , Y , and c_0 are applied because only one gate delay is needed to develop all P_i and G_i signals, followed by two gate delays in the AND-OR circuit for c_{i+1} . After a further XOR gate delay, all sum bits are available. In total, the n -bit addition process requires only four gate delays, independent of n .

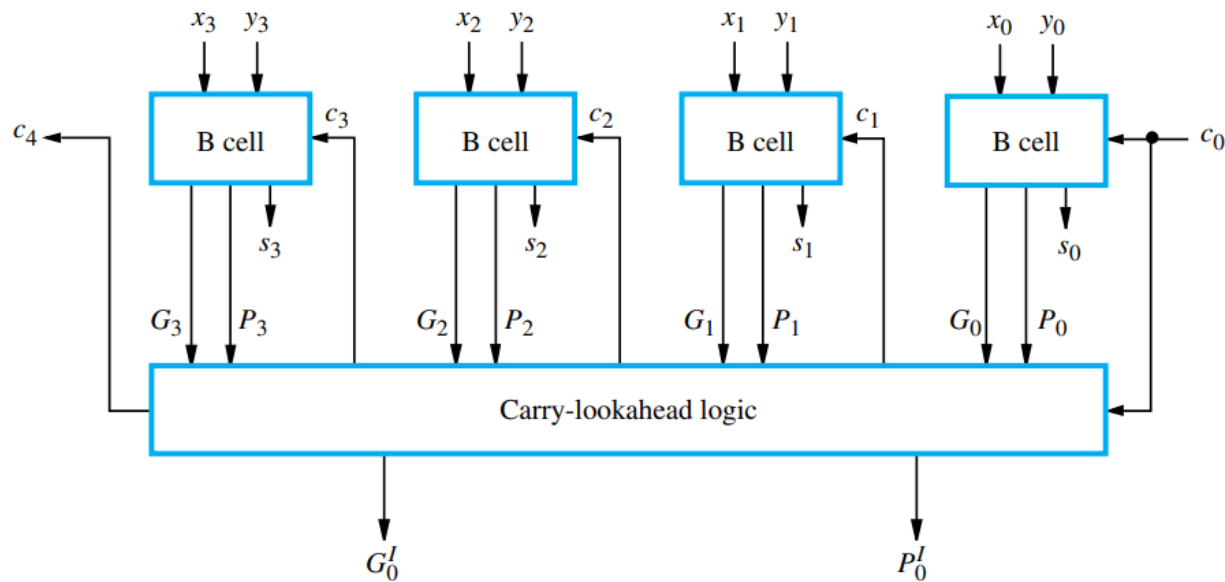
Let us consider the design of a 4-bit adder. The carries can be implemented as

$$c_1 = G_0 + P_0c_0$$

$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$

$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$



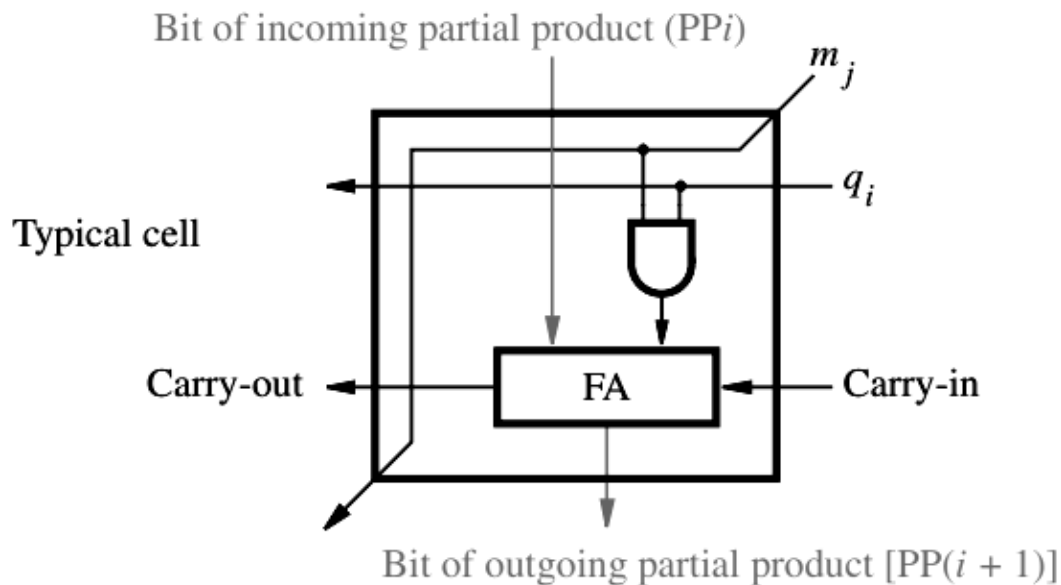
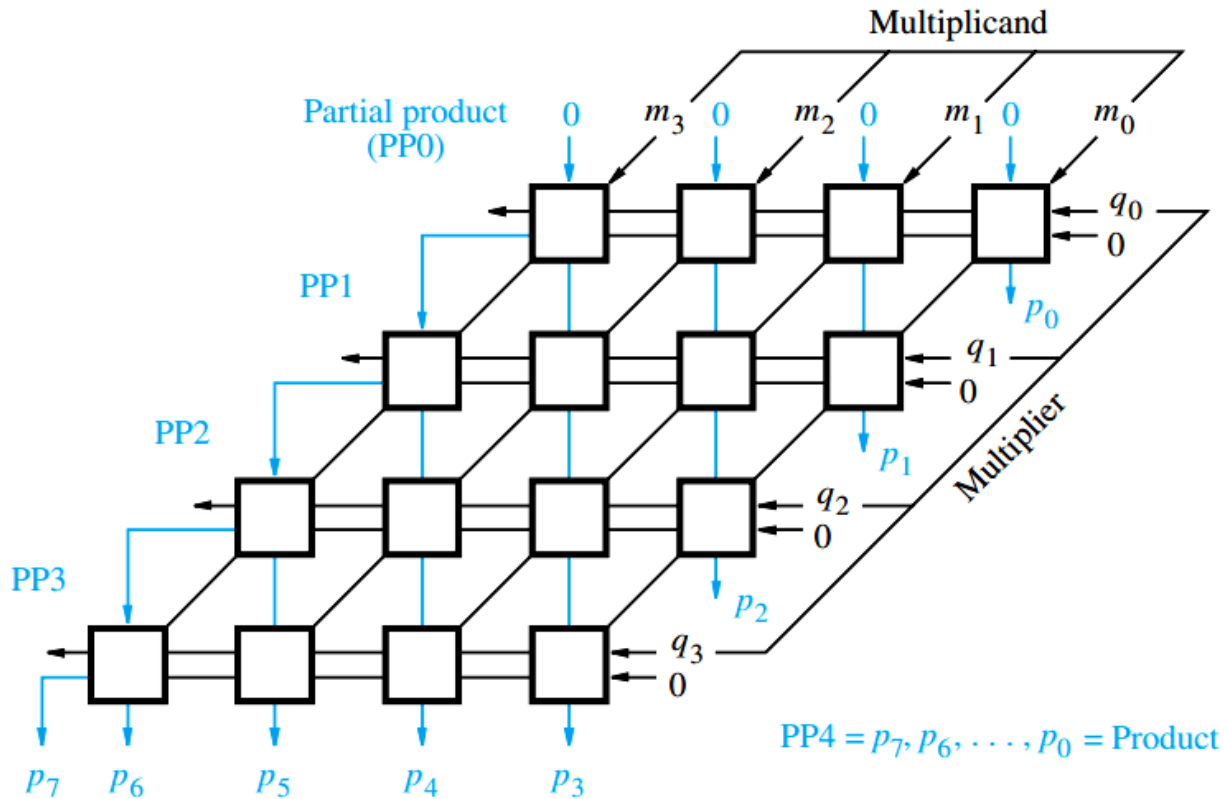
4-bit Carry look-ahead Adder

If we try to extend the carry-lookahead adder longer operands, we encounter the problem of gate fan-in constraints.

Multiplication of Unsigned Numbers

The product of two, unsigned, n -digit numbers can be accommodated in $2n$ digits.

Array Multiplier: Binary multiplication of unsigned operands can be implemented in a combinational, two dimensional, logic array, as shown below:

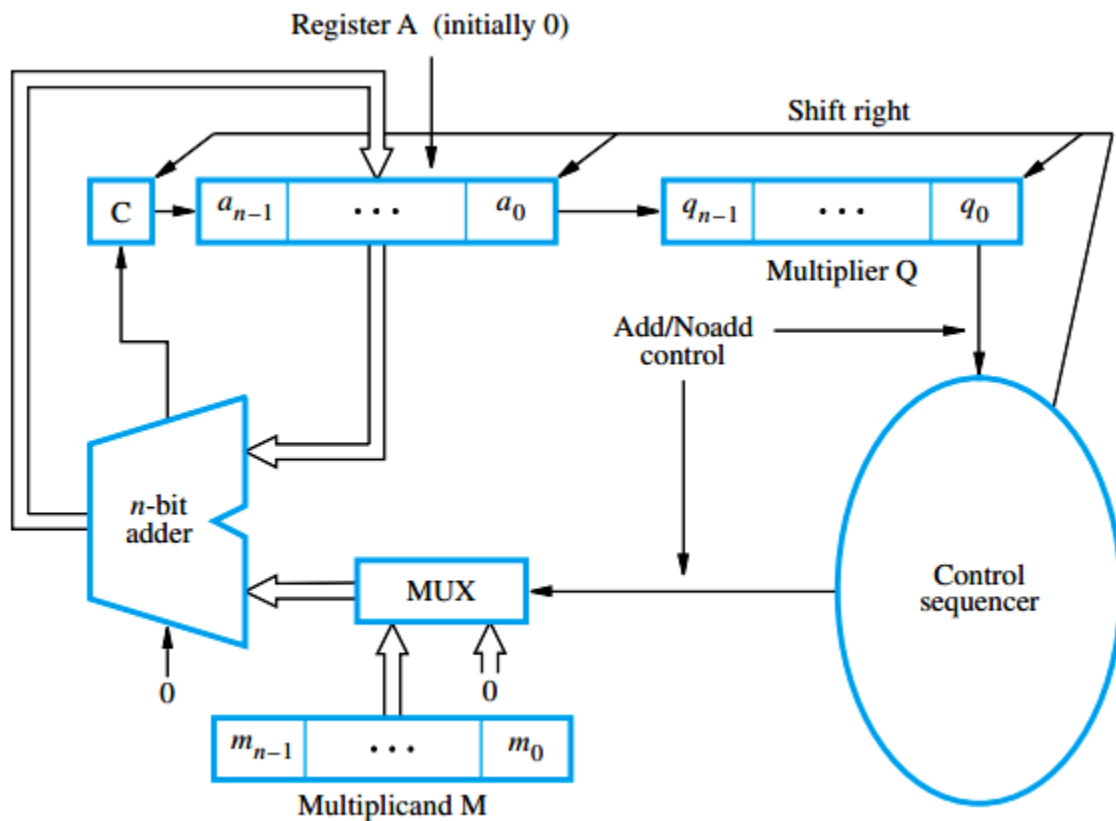


The main component in each cell is a full adder, FA. The AND gate in each cell determines whether a multiplicand bit, m_j , is added to the incoming partial-product bit, based on the value of the multiplier bit, q_i . Each row i , where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PP_i , to generate the outgoing partial product, $PP(i + 1)$, if $q_i = 1$. If $q_i = 0$, PP_i is passed vertically downward unchanged. PP_0 is all 0s, and PP_4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path.

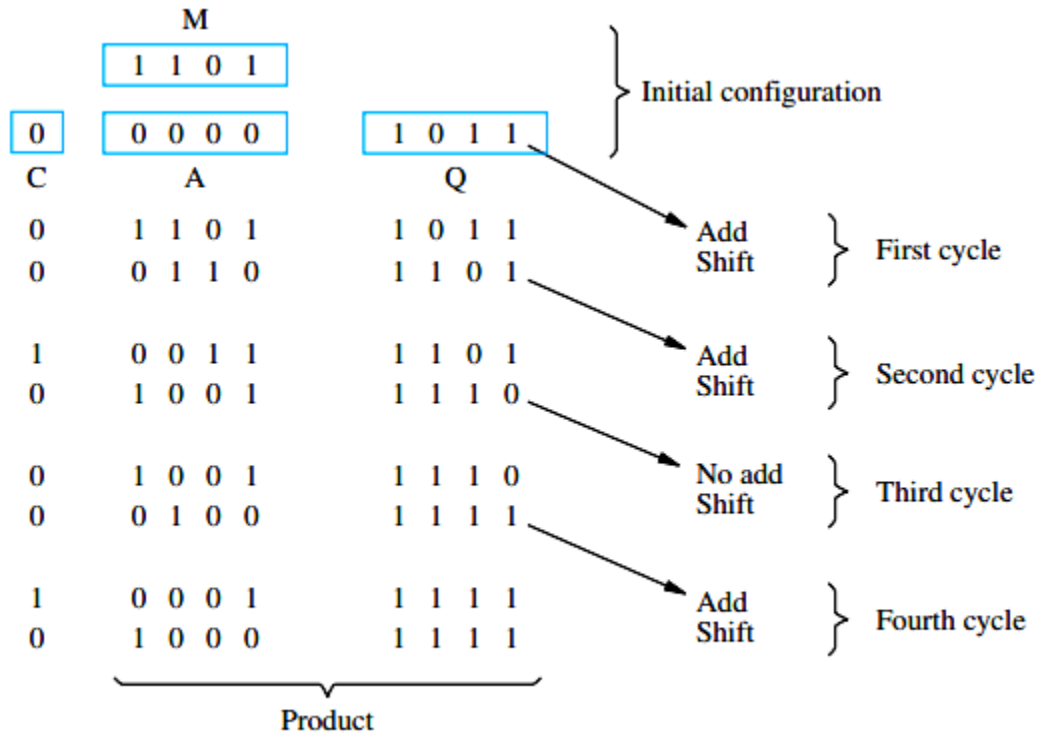
The worst-case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. This critical path

consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full-adder block, FA, the critical path has a total of $6(n - 1) - 1$ gate delays, including the initial AND gate delay in all cells, for an $n \times n$ array.

Sequential Circuit Multiplier: The combinational array multiplier uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication of two n -bit numbers can also be performed in a sequential circuit that uses a single n -bit adder. The hardware arrangement for sequential multiplication is shown below:



(a) Register configuration



(b) Multiplication example

This circuit performs multiplication by using a single n -bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PP_i while multiplier bit q_i generates the signal Add/Noadd. This signal causes the multiplexer MUX to select 0 when $q_i = 0$, or to select the multiplicand M when $q_i = 1$, to be added to PP_i to generate PP_{i+1} . The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP_0 , of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q_0 during the first cycle, q_1 during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP_{i+1} , and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q.

Multiplication of Signed Numbers

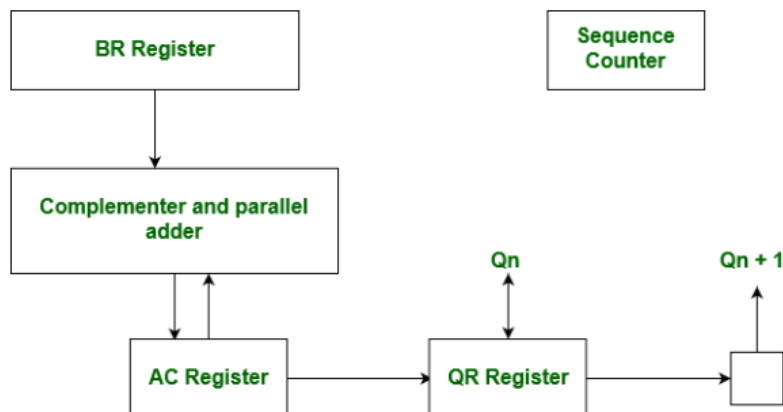
Let us consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. Example in which a 5-bit signed

COM0700504MJ: COMPUTER ARCHITECTURE UNIT-I

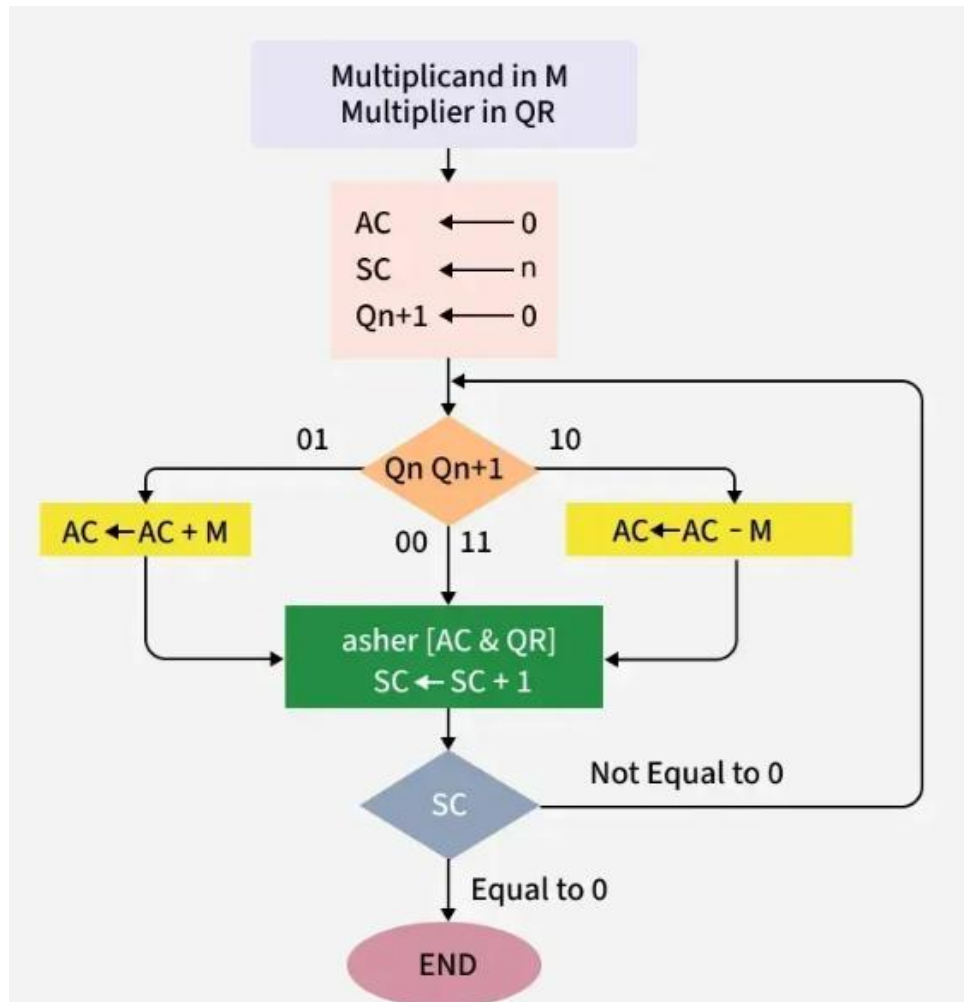
						1	0	0	1	1	(-13)
					×	0	1	0	1	1	(+11)
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0			
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(-143)

The Booth Algorithm

- The method works by examining pairs of adjacent bits in the multiplier and deciding whether to add, subtract, or do nothing with the multiplicand, followed by an arithmetic shift.
- This approach simplifies handling of consecutive sequences of 1s or 0s, making multiplication faster and more effective than the standard binary method.



The flowchart for the booth algorithm is shown below:



Example - A numerical example of booth's algorithm is shown below for $n = 4$. It shows the step by step multiplication of -5 and -7.

Step-by-Step Execution:

Iteration 1:

$$Q_n Q_{n+1} = 10$$

$$\text{Perform } AC = AC - BR = 0101$$

Arithmetic right shift of (AC, QR, Q_{n+1}):

$$AC = 0010, QR = 1100, Q_{n+1} = 1$$

$$SC = 3$$

Iteration 2:

$$Q_n Q_{n+1} = 01$$

$$\text{Perform } AC = AC + BR = 1101$$

Arithmetic right shift of (AC, QR, Q_{n+1}):

$$AC = 1110, QR = 1110, Q_{n+1} = 0$$

$$SC = 2$$

Iteration 3:

$Q_n Q_{n+1} = 00$

No operation

Arithmetic right shift of (AC, QR, Q_{n+1}):

AC = 1111, QR = 0111, $Q_{n+1} = 0$

SC = 1

Iteration 4:

$Q_n Q_{n+1} = 10$

Perform $AC = AC - BR \rightarrow AC = 0100$

Arithmetic right shift of (AC, QR, Q_{n+1}):

AC = 0010, QR = 0011, $Q_{n+1} = 1$

SC = 0

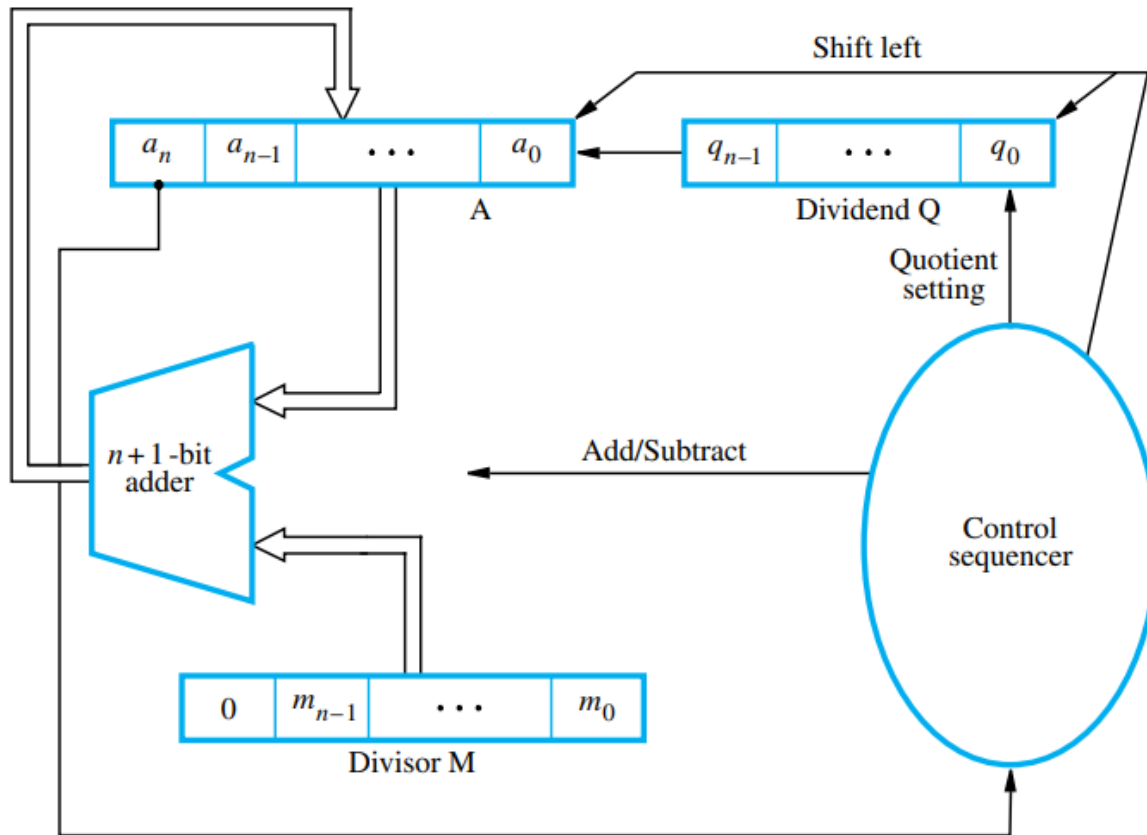
Product is calculated as follows:

Product = AC QR

Product = 0010 0011 = 35

Integer Division

Restoring Division: A circuit that implements division by longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the restoring division algorithm. Logic circuit arrangement that implements the restoring division algorithm is shown below:



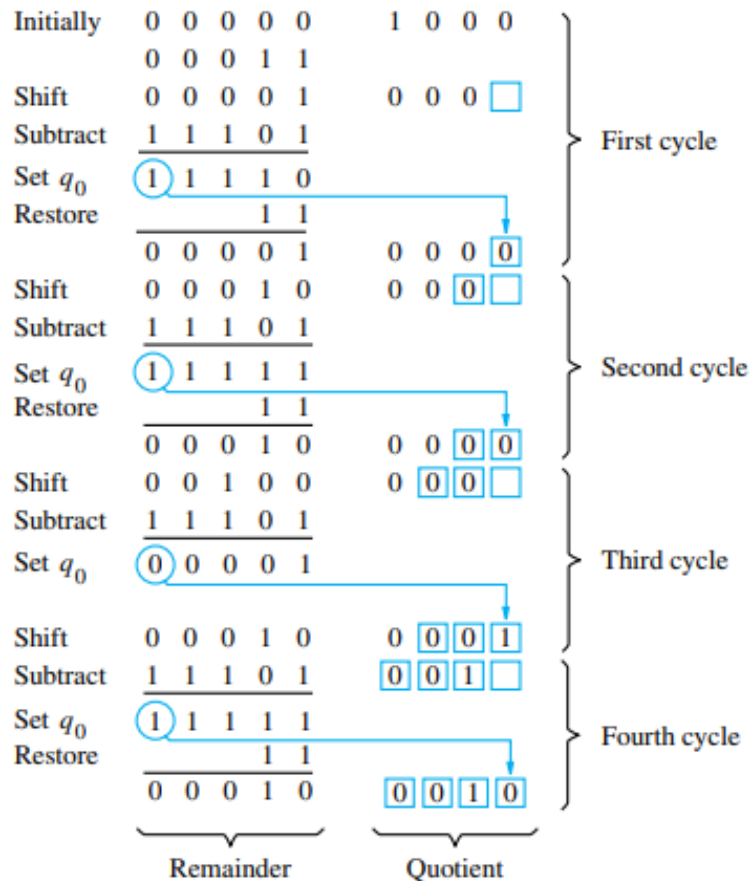
An n -bit positive divisor is loaded into register M and an n -bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n -bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps n times:

1. Shift A and Q left one bit position.
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 (that is, restore A); otherwise, set q_0 to 1.

Example:

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



Non-Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Algorithm for non-restoring division.

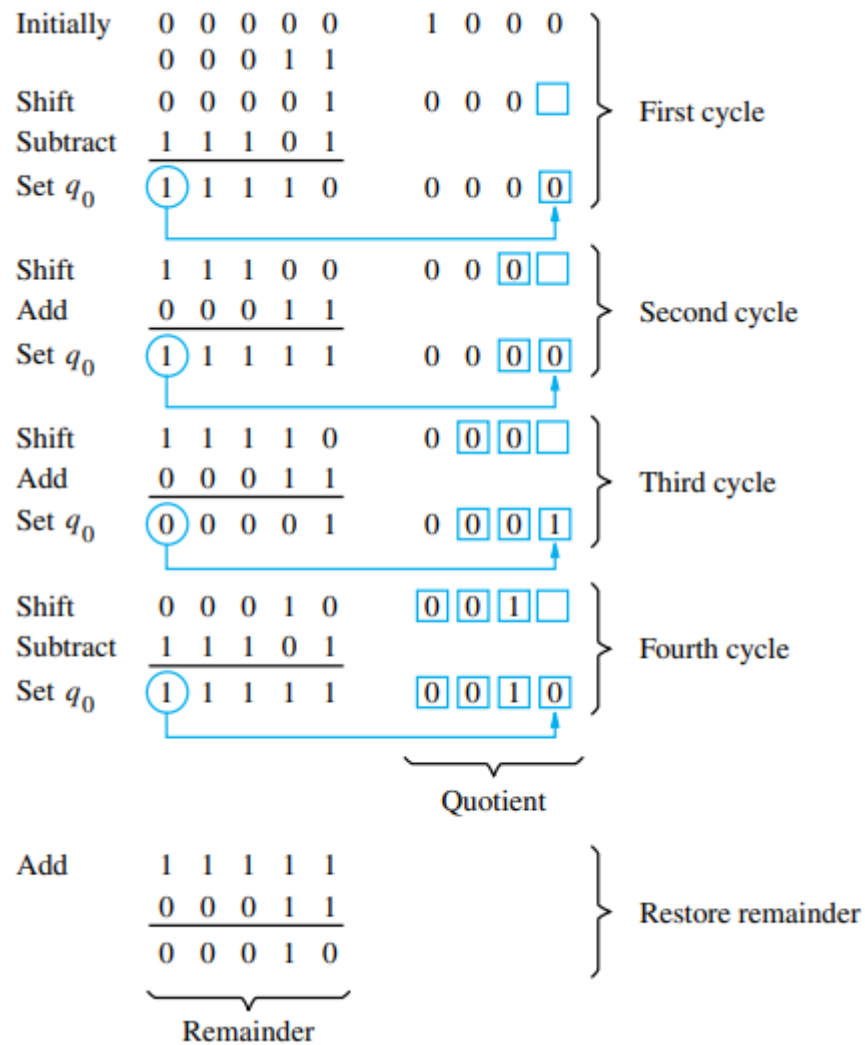
Stage 1: Do the following two steps n times:

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

Stage 2: If the sign of A is 1, add M to A.

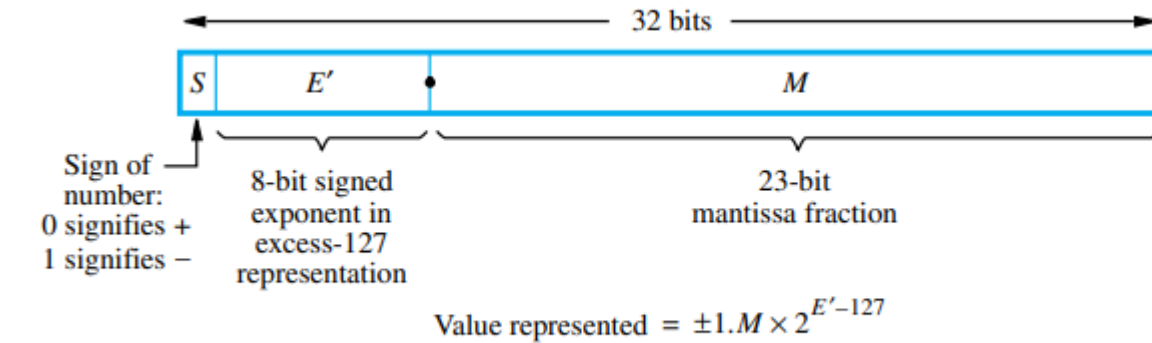
Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1.

Non-restoring division example

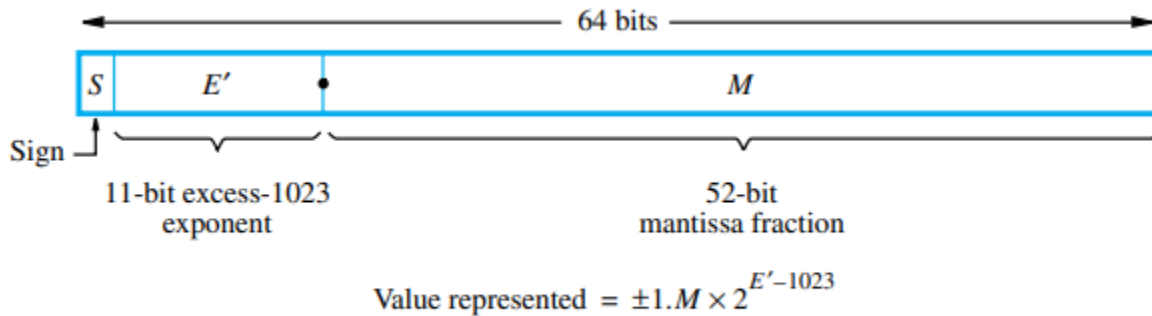


Floating point Arithmetic

The 2008 version of IEEE Standard 754 format for single precision and double precision floating point representation is shown below:



(a) Single precision



(b) Double precision

By convention, when the binary point is placed to the right of the first significant bit, the number is said to be normalized.

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.

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.

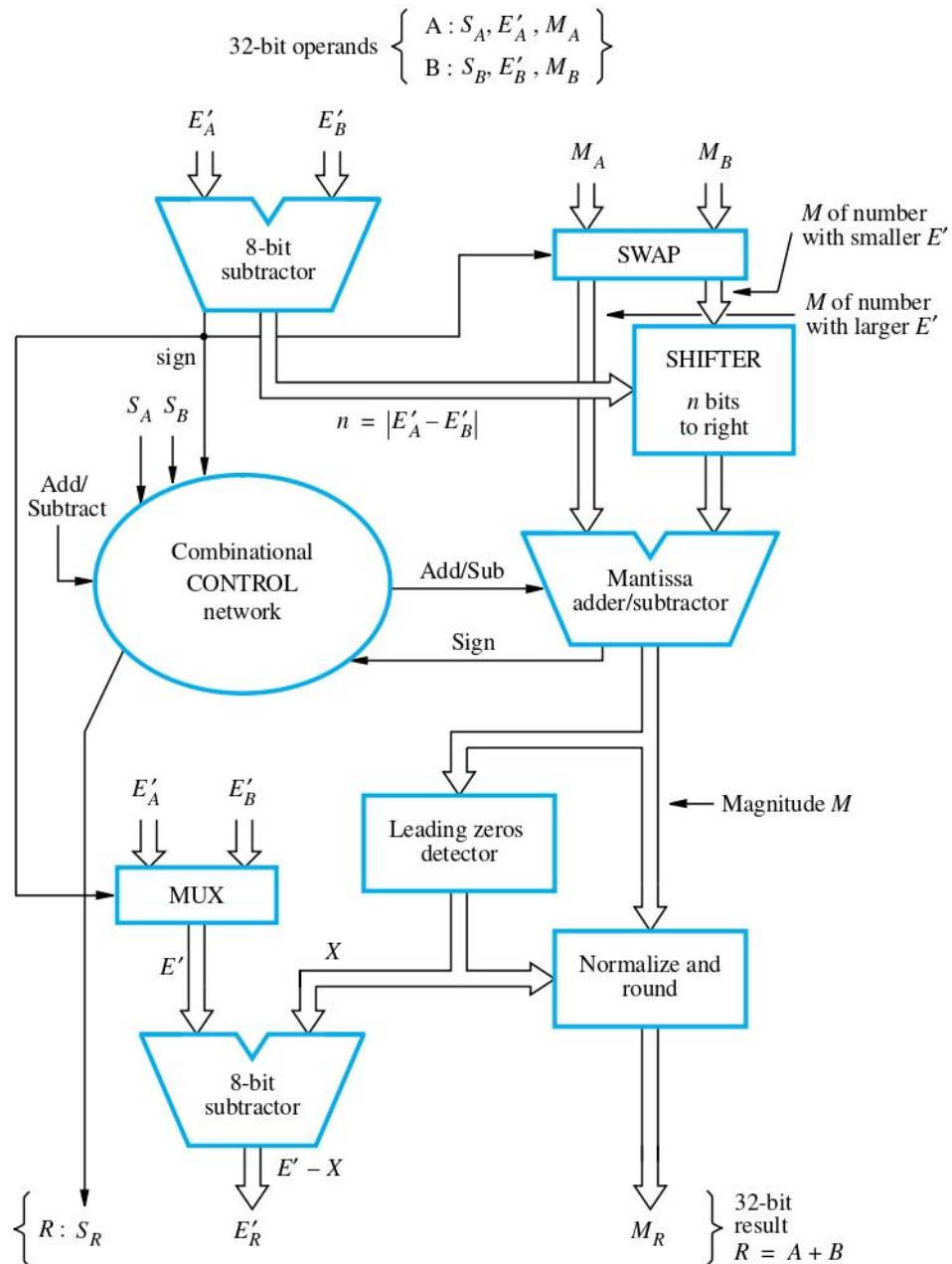
Divide Rule

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.

Hardware implementation of floating-point operations

370

CHAPTER 9 • ARITHMETIC



Floating point addition subtraction unit