3

SEQUENTIAL ALGORITHMS FOR MULTIPLICATION AND DIVISION

This chapter presents the basic sequential algorithms for multiplication, division, and square root extraction. Algorithms for high-speed multiplication are described in Chapter 6. Chapters 7, and 8 include algorithms for fast division and high-speed calculation of square roots.

3.1 SEQUENTIAL MULTIPLICATION

Let the multiplier and multiplicand be denoted by X and A, respectively, with the following sequences of digits:

$$X = x_{n-1}x_{n-2} \cdots x_1x_0 \ , \qquad A = a_{n-1}a_{n-2} \cdots a_1a_0$$

where x_{n-1} and a_{n-1} are the sign digits in either the signed-magnitude or the complement methods.

The sequential algorithm for multiplication consists of $n-1$ steps where in step j the multiplier bit x_j is examined and the product x_jA is added to the previously accumulated partial product, denoted by $P^{(j)}$. The appropriate expression for this recursive procedure is

$$P^{(j+1)} = (P^{(j)} + x_j \cdot A) \cdot 2^{-1} \ ; \qquad j = 0, 1, 2, \cdots, n-2 \qquad (3.1)$$

where in the first step $P^{(0)} = 0$. Multiplying the sum $(P^{(j)} + x_jA)$ by 2^{-1} shifts it by one position to the right, to align $P^{(j+1)}$ before adding the next product $x_{j+1}A$. This alignment is necessary, since the weight of x_{j+1} is double that of

x_j. To prove that the above procedure calculates the product of A and X, we repeatedly substitute into the recursive Equation (3.1), yielding

$$\begin{aligned} P^{(n-1)} &= (P^{(n-2)} + x_{n-2} \cdot A) \cdot 2^{-1} \\ &= \left((P^{(n-3)} + x_{n-3} \cdot A) \cdot 2^{-1} + x_{n-2} \cdot A\right) \cdot 2^{-1} = \cdots \\ &= \left(x_{n-2}2^{-1} + x_{n-3}2^{-2} + \cdots + x_0 2^{-(n-1)}\right) \cdot A \\ &= \left(\sum_{j=0}^{n-2} x_j \, 2^{-(n-1-j)}\right) \cdot A \; = \; 2^{-(n-1)} \left(\sum_{j=0}^{n-2} x_j \, 2^j\right) \cdot A \end{aligned}$$

If both operands are positive (i.e., $x_{n-1} = a_{n-1} = 0$), the product U is obtained from

$$U = 2^{n-1} \cdot P^{(n-1)} = \left(\sum_{j=0}^{n-2} x_j \, 2^j\right) \cdot A = X \cdot A \tag{3.2}$$

The result is a product consisting of $2(n-1)$ bits for its magnitude. To prove this, note that the maximum value of U is obtained when A and X assume their maximum value. Therefore,

$$U_{max} = (2^{n-1}-1)(2^{n-1}-1) = 2^{2n-2} - 2^n + 1 = 2^{2n-3} + (2^{2n-3} - 2^n + 1) \tag{3.3}$$

Since the last term in Equation (3.3) is positive for $n \geq 3$, the following inequality holds:

$$2^{2n-3} < U_{max} < 2^{2n-2} \; ; \qquad n \geq 3 \tag{3.4}$$

Thus, $(2n-2)$ bits are required to represent the value, producing a total of $(2n-1)$ bits when added to the sign bit.

For signed-magnitude numbers we multiply the two magnitudes using the above algorithm and generate the sign of the result separately (it is positive if both operands have the same sign and negative otherwise). For two's and one's complement representations we should distinguish between multiplication with a negative multiplicand A and multiplication with a negative multiplier X. If only the multiplicand is negative, there is no need to change the previous algorithm. We only to add some multiple of a negative number that is represented in either two's or one's complement. This is illustrated in the next example.

Example 3.1

In the following multiply operation, the multiplicand A is a negative number represented in the two's complement method, while the multiplier X is positive. Both are four bits long and the final product therefore has seven bits, including the sign bit. In an arithmetic unit for 4-bit operands, all

registers are four bits long, and consequently a double-length register is required for storing the final product. The vertical line in the table below separates the most significant half of the product, which can be stored in a single-length register (four bits long), from the least significant half, which can be stored in a second single-length register.

A		1	0	1	1				-5
X	$\times$	0	0	1	1				3
$P^{(0)} = 0$		0	0	0	0				
$x_0 = 1 \Rightarrow$ Add A	$+$	1	0	1	1				
		1	0	1	1				
Shift		1	1	0	1	1			
$x_1 = 1 \Rightarrow$ Add A	$+$	1	0	1	1				
		1	0	0	0	1			
Shift		1	1	0	0	0	1		
$x_2 = 0 \Rightarrow$ Shift only		1	1	1	0	0	0	1	-15

The three bits of the multiplier, x_2, x_1, and x_0, are examined one bit at a time, starting with the least significant bit x_0. An add-and-shift or shift-only operation is then performed accordingly. The final result is negative and is properly represented in two's complement. Note that the partial product bits in the least significant half do not participate in the add operation, and that all four bit positions in the first register (holding the most significant half of the final product) are utilized. However, only three bit positions in the second register are utilized, leaving the least significant bit position unused. This need not necessarily be the final arrangement. The three bits in the second register can afterwards be stored in the three rightmost positions, and the sign bit of the second register can then be set according to one of the following two possibilities: (1) Always set the sign bit to 0, irrespective of the sign of the product, since it is the least significant part of the result; (2) Set the sign bit equal to the sign bit of the first register. Another possible arrangement is to use all four bit positions in the second register for the four least significant bits of the product, use the rightmost two bit positions in the first register, and insert two copies of the sign bit into the remaining bit positions. □

The situation, however, is different when the multiplier is negative. Here, we consider each bit separately, and the sign bit (which has a negative weight) cannot be treated in the same way as the other bits. First consider two's complement numbers, which satisfy

$$X = -x_{n-1}\, 2^{n-1} + \widetilde{X} \tag{3.5}$$

where $\widetilde{X} = \sum_{j=0}^{n-2} x_j 2^j$.

If the sign bit of the multiplier in the previously presented procedure is ignored, then the final result U satisfies

$$U = \tilde{X} \cdot A = (X + x_{n-1} \cdot 2^{n-1}) \cdot A = X \cdot A + A \cdot x_{n-1} \cdot 2^{n-1}. \tag{3.6}$$

The term $X \cdot A$ is the desired product and hence, if $x_{n-1} = 1$, the following correction is necessary:

$$X \cdot A = U - A \cdot x_{n-1} \cdot 2^{n-1} \tag{3.7}$$

In other words, if $x_{n-1} = 1$, we must subtract the multiplicand A from the most significant half of U.

Example 3.2

The multiplier and multiplicand in this example are both negative numbers in the two's complement representation:

A		1	0	1	1					-5
X	$\times$	1	1	0	1					-3
$x_0 = 1 \Rightarrow$ Add A		1	0	1	1					
Shift		1	1	0	1	1				
$x_1 = 0 \Rightarrow$ Shift only		1	1	1	0	1	1			
$x_2 = 1 \Rightarrow$ Add A	$+$	1	0	1	1					
		1	0	0	1	1	1			
Shift		1	1	0	0	1	1	1		
$x_3 = 1 \Rightarrow$ Correct	$+$	0	1	0	1					
		0	0	0	1	1	1	1		$+15$

In the correction step, the subtraction of the multiplicand is performed by adding its two's complement. □

Similarly, when multiplying one's complement numbers, which satisfy

$$X = -x_{n-1}(2^{n-1} - ulp) + \tilde{X} \tag{3.8}$$

then,

$$X \cdot A = U - x_{n-1} \cdot 2^{n-1} \cdot A + x_{n-1} \cdot ulp \cdot A. \tag{3.9}$$

Thus, if $x_{n-1} = 1$, we start with $P^{(0)} = A$, which takes care of the second correction term, namely, $x_{n-1} \cdot ulp \cdot A$, and at the end of the process we subtract the first correction term, $A \cdot x_{n-1} \cdot 2^{n-1}$.

Example 3.3

The product of 5 and −3 in one's complement representation is

A		0	1	0	1				5
X	×	1	1	0	0				−3
$x_3 = 1 \Rightarrow P^{(0)} = A$		0	1	0	1				
$x_0 = 0 \Rightarrow$ Shift		0	0	1	0	1			
$x_1 = 0 \Rightarrow$ Shift		0	0	0	1	0	1		
$x_2 = 1 \Rightarrow$ Add A	+	0	1	0	1				
		0	1	1	0	0	1		
Shift		0	0	1	1	0	0	1	
$x_3 = 1 \Rightarrow$ Correct	+	1	0	1	0	1	1	1	
		1	1	1	0	0	0	0	−15

As in the previous example, the subtraction of the (first) correction term is accomplished by adding its one's complement. However, unlike the previous example, the one's complement has to be expanded to double size using the sign digit (see Section 1.6). This implies that a double-length binary adder is needed. □

3.2 SEQUENTIAL DIVISION

Division is the most complex of the four basic arithmetic operations and, consequently, the most time-consuming. Unlike the other three operations, division, in general, has a result consisting of two components. Given a dividend X and a divisor D, a quotient Q and a remainder R have to be calculated so as to satisfy

$$X = Q \cdot D + R \quad \text{with} \quad R < D. \tag{3.10}$$

We will assume at first, for simplicity, that the operands X and D and the results Q and R are positive numbers.

In many fixed-point arithmetic units, a double-length product is available after a multiply operation, and we wish to allow the use of this result in a subsequent divide operation. Thus, X may occupy a double-length register, while all other operands are stored in single-length registers. Consequently, we have to make sure that the resulting quotient Q is smaller than or equal to the largest number that we can store in a single-length register. If n is the number of bits in a single-length register, then every single-length integer is smaller than 2^{n-1}. Therefore, to ensure that the quotient is a single-length integer (i.e., the inequality $Q < 2^{n-1}$ is satisfied), we must require that

$$X < 2^{n-1}D.$$

If this condition is not satisfied, an *overflow* indication should be produced by the arithmetic unit. One should be aware that the above condition can always be satisfied by preshifting one of the operands X or D (or both). This preshifting is especially simple to apply when the operands are floating-point numbers. Another condition that has to be checked is that $D \neq 0$. If this is not the case, a *divide by zero* indication should be generated by the arithmetic unit. Unlike the previous condition, no corrective action can be taken when $D = 0$.

The presentation of algorithms for division is simpler when the dividend and divisor, as well as the quotient and remainder, are interpreted as fractions. In this case, the divide overflow condition becomes $X < D$ to ensure that the quotient is a fraction. The division procedure that is presented next assumes that all operands and results are fractions, but is clearly also valid for integers, as will become apparent later on.

To obtain the fractional (positive) quotient $Q = 0.q_1 \cdots q_m$ (where $m = n - 1$), we perform the division as a sequence of subtractions and shifts. In step i of the process the remainder is compared to the divisor D. If the remainder is the larger of the two, then the quotient bit q_i is set to 1. If not, it is set to 0. The equation for the ith step is

$$r_i = 2r_{i-1} - q_i \cdot D\ ; \qquad i = 1, 2, \cdots, m \tag{3.11}$$

where r_i is the new remainder and r_{i-1} is the previous remainder. The first remainder is $r_0 = X$. Thus, q_i is determined by comparing $2r_{i-1}$ to D. This comparison is the most complicated operation in the division process.

We will now prove that the above procedure indeed calculates the quotient and the final remainder. The remainder in the last step is r_m and repeated substitution of Equation (3.11) yields

$$\begin{aligned} r_m &= 2r_{m-1} - q_m \cdot D \\ &= 2(2r_{m-2} - q_{m-1} \cdot D) - q_m \cdot D = \cdots \\ &= 2^m r_0 - (q_m + 2q_{m-1} + \cdots + 2^{m-1} q_1) \cdot D. \end{aligned}$$

Substituting $r_0 = X$ and dividing both sides by 2^m results in

$$r_m 2^{-m} = X - (q_1 2^{-1} + q_2 2^{-2} + \cdots + q_m 2^{-m}) \cdot D;$$

hence

$$r_m\ 2^{-m} = X - Q \cdot D \tag{3.12}$$

as required. Note that the true final remainder is $R = r_m 2^{-m}$.

Example 3.4

Let $X = (0.100000)_2 = 1/2$ and $D = (0.110)_2 = 3/4$. The dividend occupies a double-length register. The condition $X < D$ is clearly satisfied.

$r_0 = X$			0	.1	0	0	0	0	0	
$2r_0$		0	1	.0	0	0	0	0		set $q_1 = 1$
Add $-D$	+	1	1	.0	1	0				
$r_1 = 2r_0 - D$		0	0	.0	1	0	0	0		
$2r_1$		0	0	.1	0	0	0			set $q_2 = 0$
$r_2 = 2r_1$		0	0	.1	0	0	0			
$2r_2$		0	1	.0	0	0				set $q_3 = 1$
Add $-D$	+	1	1	.0	1	0				
$r_3 = 2r_2 - D$		0	0	.0	1	0				

Note that the generation of $2r_0$ should not result in an overflow indication (multiplying a positive number by 2 should result in a positive number), since the quotient and remainder are within the proper range for the given dividend and divisor. Hence, an extra bit position in the arithmetic unit is needed.

The final results are $Q = (0.101)_2 = 5/8$ and $R = r_m 2^{-m} = r_3 2^{-3} = 1/4 \cdot 2^{-3} = 1/32$. (The precise quotient is the infinite binary fraction $2/3 = 0.1010101\cdots$.) The quotient and final remainder satisfy the equation $X = Q \cdot D + R = 5/8 \cdot 3/4 + 1/32 = 16/32 = 1/2$. □

Exactly the same procedure should be followed if the operands and results are integers. In this case we may rewrite Equation (3.10) as follows:

$$2^{2n-2} X_F = 2^{n-1} Q_F \cdot 2^{n-1} D_F + 2^{n-1} R_F \tag{3.13}$$

where X_F, D_F, Q_F, and R_F are fractions. Dividing Equation (3.13) by 2^{2n-2} yields

$$X_F = Q_F \cdot D_F + 2^{-(n-1)} R_F. \tag{3.14}$$

The above mentioned condition $X < 2^{n-1} D$, when divided by 2^{2n-2}, now takes the form $X_F < D_F$.

Example 3.5

We repeat the previous example with all operands and results being integers. In this case the double-length dividend is $X = 0100000_2 = 32$, and the divisor is $D = 0110_2 = 6$. The overflow condition $X < 2^{n-1} D$ is tested by comparing the most significant half of X, 0100, to D, 0110. The results of the division are $Q = 0101_2 = 5$ and $R = 0010_2 = 2$. Observe that in the final step of the process the true remainder R is generated and, as can be verified from Equation (3.14), there is no need to further multiply it by $2^{-(n-1)}$. □

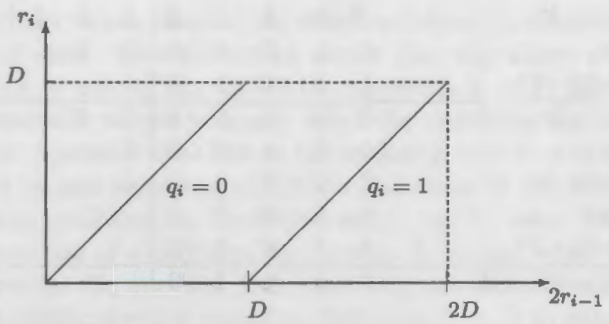

FIGURE 3.1 Restoring division.

The most difficult step in the division procedure is the comparison between the divisor and the remainder to determine the quotient bit. If this is done by subtracting D from $2r_{i-1}$, then in the case of a negative result we set $q_i = 0$, and we must restore the remainder to its previous value. This method is therefore called *restoring division*, and can be diagrammed, as shown in Figure 3.1. Such a diagram is sometimes called a Robertson diagram [7].

This diagram illustrates the fact that if $r_{i-1} < D$, q_i should be selected so as to ensure $r_i < D$. Since $r_0 = X < D$, we are guaranteed to obtain $R < D$. In summary, a division performed by the restoring method uses m subtractions, m shift operations, and an average of $m/2$ restore operations. The latter can be implemented either by adding D or by retaining a copy of the previous remainder, thus avoiding the time penalty involved in the restore operations.

3.3 NONRESTORING DIVISION

An alternative scheme for sequential division is the *nonrestoring* division algorithm, in which the quotient bit is not corrected and the remainder is not restored immediately if it is negative. These corrections are instead postponed to later steps. In the restoring method, if $2r_{i-1} - D$ is negative, the remainder is restored to $2r_{i-1}$. It is then shifted and D is once again subtracted, obtaining $4r_{i-1} - D$. This process is repeated as long as the remainder is negative. In the nonrestoring method we avoid the restore operation, stay with a negative remainder $2r_{i-1} - D < 0$, shift it, and then attempt to correct it by *adding* D, obtaining $2(2r_{i-1} - D) + D = 4r_{i-1} - D$. Thus, this algorithm produces a remainder equal to the one we would generate using restoring division.

Consider now the resulting quotient. To enable the correction of a "wrong" selection of the quotient bit in step i, we must allow the next quotient bit, q_{i+1}, to assume a negative value. In other words, the allowed values for q_i are 1 and $\bar{1}$ where $\bar{1}$ represents -1. If q_i was incorrectly set to 1, resulting in a negative remainder, we would then select $q_{i+1} = \bar{1}$ and *add* D to the remainder. Hence,

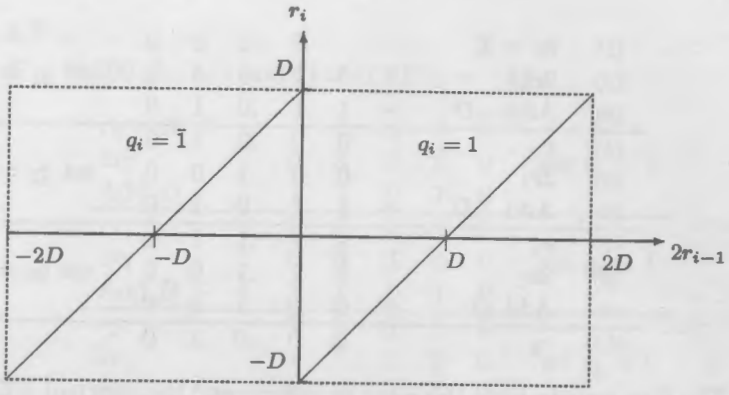

FIGURE 3.2 Nonrestoring division.

instead of $q_i q_{i+1} = 10$ (which is too large), we would get $q_i q_{i+1} = 1\bar{1} = 01$. Further correction, if needed, would be done in the next steps. Consequently, the quotient bit is determined in the nonrestoring scheme by the following rule:

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 0 \\ \bar{1} & \text{if } 2r_{i-1} < 0 \end{cases} \tag{3.15}$$

This rule is simpler (and faster to execute) than the selection rule for restoring division since it requires the comparison of $2r_{i-1}$ to 0 rather than D. The remainder is computed using the same equation

$$r_i = 2r_{i-1} - q_i \cdot D; \tag{3.16}$$

in other words, subtract the divisor D if $2r_{i-1}$ is positive and add it otherwise. The nonrestoring division is diagrammed in Figure 3.2. Here, $|r_{i-1}| < D$ and q_i is selected to ensure $|r_i| < D$. Note that $q_i \neq 0$ and therefore, at each step, either an addition or subtraction is performed. This is not an SD representation, and there is no redundancy in the representation of the quotient in the nonrestoring division. In summary, the nonrestoring method requires exactly m add/subtract and shift operations. Its main advantage is its simpler selection rule.

Example 3.6

Let $X = (0.100)_2 = 1/2$, and $D = (0.110)_2 = 3/4$, as in Example 3.4.

(1)	$r_0 = X$			0	.1	0	0	
(2)	$2r_0$		0	1	.0	0	0	set $q_1 = 1$
(3)	Add $-D$	+	1	1	.0	1	0	
(4)	r_1		0	0	.0	1	0	
(5)	$2r_1$		0	0	.1	0	0	set $q_2 = 1$
(6)	Add $-D$	+	1	1	.0	1	0	
(7)	r_2		1	1	.1	1	0	
(8)	$2r_2$		1	1	.1	0	0	set $q_3 = \bar{1}$
(9)	Add D	+	0	0	.1	1	0	
(10)	r_3		0	0	.0	1	0	

The final remainder is the same as before, and the quotient is $Q = 0.11\bar{1} = 0.101_2 = 5/8$. □

The nonrestoring division process in the previous example can be represented graphically using a diagram similar to the one depicted in Figure 3.2. The resulting diagram is shown in Figure 3.3. The horizontal lines correspond to the Add $\pm D$ operation in lines 3, 6 and 9 in Example 3.6, and the diagonal lines correspond to the Multiply by 2 operation in lines 2, 5 and 8.

A very important feature of nonrestoring division is that it can easily be extended to two's complement negative numbers. The generalized selection rule for q_i is

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \text{ and } D \text{ have the same sign} \\ \bar{1} & \text{if } 2r_{i-1} \text{ and } D \text{ have opposite signs} \end{cases} \tag{3.17}$$

Since the remainder changes signs during the process, there is nothing special about a negative dividend X. The following example illustrates the case of a negative divisor in two's complement.

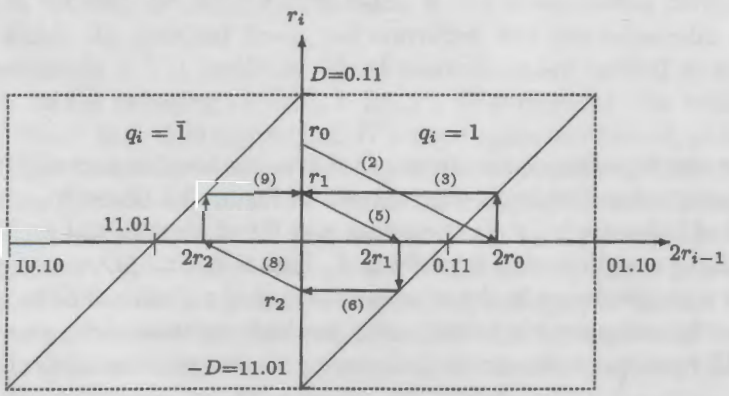

FIGURE 3.3 The nonrestoring division in Example 3.6.

Example 3.7

Let $X = (0.100)_2 = 1/2$ and $D = (1.010)_2 = -3/4$.

$r_0 = X$			0	.1	0	0	
$2r_0$		0	1	.0	0	0	set $q_1 = \bar{1}$
Add D		1	1	.0	1	0	
r_1		0	0	.0	1	0	
$2r_1$		0	0	.1	0	0	set $q_2 = \bar{1}$
Add D	+	1	1	.0	1	0	
r_2		1	1	.1	1	0	
$2r_2$		1	1	.1	0	0	set $q_3 = 1$
Add $-D$	+	0	0	.1	1	0	
r_3		0	0	.0	1	0	

Finally, $Q = 0.\bar{1}\bar{1}1 = 0.\bar{1}0\bar{1} = -(0.101)_2 = -5/8$, or in two's complement, 1.011. Note that the final remainder is 1/32 and has the same sign as the dividend X. □

By definition, the sign of the final remainder must equal that of the dividend. For example, when dividing 5 by 3 we should obtain a quotient of 1 and a final remainder of 2, and not a quotient of 2 and a final remainder of -1, although this remainder still satisfies $|R| < D$. Consequently, if the sign of the final remainder is different from that of the dividend, a correction of both the final remainder and quotient is needed. This situation, requiring a correction step, arises since the quotient digits in the nonrestoring division algorithm are restricted to $\{1, \bar{1}\}$. The last digit can not be set to 0 and therefore an "even" quotient can not be generated.

Example 3.8

Let $X = (0.101)_2 = 5/8$, and $D = (0.110)_2 = 3/4$. Then

$r_0 = X$			0	.1	0	1	
$2r_0$		0	1	.0	1	0	set $q_1 = 1$
Add $-D$	+	1	1	.0	1	0	
r_1		0	0	.1	0	0	
$2r_1$		0	1	.0	0	0	set $q_2 = 1$
Add $-D$	+	1	1	.0	1	0	
r_2		0	0	.0	1	0	
$2r_2$		0	0	.1	0	0	set $q_3 = 1$
Add $-D$	+	1	1	.0	1	0	
r_3		1	1	.1	1	0	

The final remainder is negative, while the dividend is positive. We must correct the final remainder by adding D to r_3, yielding 1.110+0.110=0.100, and then correct the quotient:

$$Q_{corrected} = Q - ulp$$

where $Q = 0.111$, and therefore $Q_{corrected} = 0.110_2 = 3/4$. □

In general, if the final remainder and the dividend have opposite signs, a correction step is needed. If the dividend and divisor have the same sign, then the remainder r_m is corrected by adding D and the quotient is corrected by subtracting *ulp*. If the dividend and divisor have opposite signs, we subtract D from r_m and correct the quotient by adding *ulp*.

Another consequence of the fact that 0 is not an allowed digit in non-restoring division, is the need for a correction if a zero remainder is generated in an intermediate step. This case is illustrated in the next example.

Example 3.9

Let $X = (1.101)_2 = -3/8$ and $D = (0.110)_2 = 3/4$. The correct result of this division is $Q = -1/2$ with a zero remainder.

$r_0 = X$			1	.1	0	1	
$2r_0$		1	1	.0	1	0	set $q_1 = \bar{1}$
Add D	+	0	0	.1	1	0	
r_1		0	0	.0	0	0	zero remainder
$2r_1$		0	0	.0	0	0	set $q_2 = 1$
Add $-D$	+	1	1	.0	1	0	
r_2		1	1	.0	1	0	
$2r_2$		1	0	.1	0	0	set $q_3 = \bar{1}$
Add D	+	0	0	.1	1	0	
r_3		1	1	.0	1	0	

Note that although the final remainder r_3 and the dividend X have the same sign, a correction step is needed, since the quotient we get is $Q = 0.\bar{1}1\bar{1} = 0.\bar{1}01_2 = -3/8$ instead of $-1/2$. We must therefore detect the occurrence of a zero intermediate remainder and correct the final remainder (to obtain a zero remainder):

$$r_3(corrected) = r_3 + D = 1.010 + 0.110 = 0.000$$

We have to then correct the quotient $Q = 0.\bar{1}1\bar{1} = 0.\bar{1}01$ by subtracting *ulp*, yielding $Q_{corrected} = 0.\bar{1}00_2 = -1/2$. □

3.3.1 Generating a Two's Complement Quotient

The nonrestoring division, as previously presented, generates a quotient that uses the digits 1 and $\bar{1}$ and might therefore be incompatible with the representation used for the dividend and divisor. If X and D are represented in two's complement, then there is a need for a conversion from the above representation to two's complement. We may, in principle, use one of the algorithms presented in Section 2.4 for converting a SD number to its two's complement representation. These algorithms however, require that all the digits of the quotient be known before the conversion can be performed thus increasing the total execution time of the divide operation. We prefer therefore to employ an algorithm that performs the conversion *on the fly*, as the digits of the quotient become available, in a serial fashion from the most to the least significant digit. Such an on-the-fly conversion algorithm from SD to two's complement representation has been presented in [3].

We can however, take advantage of the fact that the quotient digit in the nonrestoring division can assume only the values 1 and $\bar{1}$ (i.e., $q_i \neq 0$) and derive a simpler algorithm that requires a less complex circuit for its implementation. Since the quotient digit can assume only two values, a single bit is sufficient for representing it, and we may assign the digits 0 and 1 to the values $\bar{1}$ and 1, respectively. Let the resulting binary number be denoted by $(0.p_1 \cdots p_m)$ where $p_i = \frac{1}{2}(q_i + 1)$. This number can be converted to two's complement using the following algorithm:

Step 1: Shift the given number one bit position to the left.

Step 2: Complement the most significant bit.

Step 3: Shift a 1 into the least significant position.

The result of this algorithm is the sequence

$$(1 - p_1) \,.\, p_2 p_3 \cdots p_m 1.$$

We will now prove that the above sequence, when interpreted as a number in two's complement, has the same numerical value as the original quotient Q. The value of the above sequence in two's complement is

$$-(1 - p_1)2^0 + \sum_{i=2}^{m} p_i 2^{-i+1} + 2^{-m}. \tag{3.18}$$

Substituting $p_i = \frac{1}{2}(q_i + 1)$ yields

$$q_1 2^{-1} - 2^{-1} + \sum_{i=2}^{m} (q_i + 1)2^{-i} + 2^{-m}$$

$$= q_1 2^{-1} - (2^{-1} - 2^{-m}) + \sum_{i=2}^{m} q_i 2^{-i} + \sum_{i=2}^{m} 2^{-i}.$$

The last term equals $(2^{-1} - 2^{-m})$ and therefore,

$$= q_1 2^{-1} + \sum_{i=2}^{m} q_i 2^{-i} = \sum_{i=1}^{m} q_i 2^{-i} = Q.$$

The above conversion algorithm can be executed in a bit-serial fashion; that is, we can generate the appropriate bit of the quotient, when represented in two's complement, at each step of the nonrestoring division. For example, in the last division with $X = 1.101$ and $D = 0.110$, instead of generating the quotient bits $.\bar{1}1\bar{1}$, we can generate the bits $(1-0).101 = 1.101$. After the correction step we obtain $(Q - ulp) = 1.100$, which is the correct representation of $-1/2$ in two's complement. The same on-the-fly conversion algorithm can be derived from the general *SD* to two's complement conversion algorithm presented in Section 2.4. This is left as an exercise for the reader.

3.4 SQUARE ROOT EXTRACTION

The conventional "completing the square" method for square root extraction is conceptually similar to the restoring division scheme. Let the given radicand X be a positive fraction, and let $Q = (0.q_1 q_2 \cdots q_m)$ denote its square root. The bits of Q are generated in m steps, one bit per step. We use the notation

$$Q_i = \sum_{k=1}^{i} q_k 2^{-k}$$

for the partially developed root at step i. Thus, $Q_m = Q$. We also denote the remainder in step i by r_i. The next remainder, in general, is calculated from

$$r_i = 2r_{i-1} - q_i \cdot (2Q_{i-1} + q_i 2^{-i}). \tag{3.19}$$

Comparing the above equation to Equation (3.11) suggests that the square root extraction can be viewed as division with a changing divisor, i.e., $\hat{D}_i = (2Q_{i-1} + q_i 2^{-i})$.

In the first step the remainder is the radicand X and $Q_0 = 0$. The performed calculation is therefore

$$r_1 = 2r_0 - q_1(0 + q_1 2^{-1}) = 2X - q_1(0 + q_1 2^{-1}) \tag{3.20}$$

To determine the square root digit q_i in the restoring scheme, a tentative remainder,

$$2r_{i-1} - (2Q_{i-1} + 2^{-i})$$

is calculated. Note that the term $(2Q_{i-1} + 2^{-i})$ is equal to $(q_1.q_2 \cdots q_{i-1}01$ and is very simple to calculate. If the above tentative remainder is positive, we store its value in r_i and set q_i equal to 1. Otherwise, we set $r_i = 2r_{i-1}$ and $q_i = 0$.

To prove that the above procedure yields the required square root, we repeatedly substitute Equation (3.19) in the expression for r_m, obtaining

$$\begin{aligned} r_m &= 2r_{m-1} - q_m(2Q_{m-1} + q_m 2^{-m}) \\ &= 2^2 r_{m-2} - 2q_{m-1}(2Q_{m-2} + q_{m-1}2^{-(m-1)}) - q_m(2Q_{m-1} + q_m 2^{-m}) \\ &\vdots \\ &= 2^m \cdot r_0 - 2^m\left[(q_1 2^{-1})^2 + (q_2 2^{-2})^2 + \cdots + (q_m 2^{-m})^2\right] \\ &\quad -2^m\left[2q_2 2^{-2} q_1 2^{-1} + \cdots + 2q_m 2^{-m}\sum_{i=1}^{m-1} q_i 2^{-i}\right] \\ &= 2^m X - 2^m\left(\sum_{i=1}^{m} q_i 2^i\right)^2 = 2^m(X - Q^2). \end{aligned}$$

Dividing by 2^m results in the expected relation with $r_m 2^{-m}$ as the final remainder.

Example 3.10

Let $X = 0.1011_2 = 11/16 = 176/256$. Its square root is

$r_0 = X$			0	.1	0	1	1	
$2r_0$		0	1	.0	1	1	0	
$-(0 + 2^{-1})$	$-$	0	0	.1	0	0	0	
r_1		0	0	.1	1	1	0	set $q_1 = 1$, $Q_1 = 0.1$
$2r_1$		0	1	.1	1	0	0	
$-(2Q_1 + 2^{-2})$	$-$	0	1	.0	1	0	0	
r_2		0	0	.1	0	0	0	set $q_2 = 1$, $Q_2 = 0.11$
$2r_2$		0	1	.0	0	0	0	is smaller than $(2Q_2 + 2^{-3})$ $= 1.101$
$r_3 = r_2$		0	1	.0	0	0	0	set $q_3 = 0$, $Q_3 = 0.110$
$2r_3$		1	0	.0	0	0	0	still a positive number
$-(2Q_3 + 2^{-4})$	$-$	0	1	.1	0	0	1	
r_4		0	0	.0	1	1	1	set $q_4 = 1$, $Q_4 = 0.1101$

Finally, $Q = 0.1101_2 = 13/16$ and the final remainder is $2^{-4}r_4 = 7/256 = X - Q^2 = (176 - 169)/256$. □

The above procedure is similar to the restoring division algorithm. A method similar to the nonrestoring division algorithm can be employed with the following selection rule for q_i:

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq 0 \\ \bar{1} & \text{if } 2r_{i-1} < 0 \end{cases} \tag{3.21}$$

This algorithm is illustrated in the next example.

Example 3.11

Let $X = 0.011001_2 = 25/64$.

$r_0 = X$		0	.0	1	1	0	0	1	
$2r_0$		0	.1	1	0	0	1	0	set $q_1=1$, $Q_1=0.1$
$-(0+2^{-1})$	−	0	.1	0	0	0	0	0	
r_1		0	.0	1	0	0	1	0	
$2r_1$		0	.1	0	0	1	0	0	set $q_2=1$, $Q_2=0.11$
$-(2Q_1+2^{-2})$	−	0	1	.0	1	0	0	0	0
r_2		$\bar{1}$	1	.0	1	0	1	0	0
$2r_2$		1	0	.1	0	1	0	0	0	set $q_3=\bar{1}$, $Q_3=0.11\bar{1}$
$+(2Q_2-2^{-3})$	+	0	1	.1	0	$\bar{1}$	0	0	0
r_3		0	0	.0	0	0	0	0	0

The square root is $Q = 0.11\bar{1} = 0.101_2 = 5/8$. □

The digits of the square root Q can be converted to two's complement representation by the same method used for the quotient in the nonrestoring division algorithm. Faster algorithms for square root extraction have been developed and implemented. Some of them are introduced in Chapter 7.

3.5 EXERCISES

3.1. Given the following three pairs of binary multiplicand and multiplier:
(i) +.1001 and −.0101 (ii) −.1001 and +.0101 (iii) −.1001 and −.0101.
(a) Represent the numbers in the two's complement form and multiply them. Check your results.
(b) Repeat (a) for the one's complement form.

3.2. Can the sequential multiplication algorithm be modified so that the multiplier bits are examined starting with the most significant bit? What might be a major disadvantage to this modified algorithm?

3.3. Multiply the binary *SD* numbers $A = 10\bar{1}01$ (the multiplicand) and $X = 0\bar{1}10\bar{1}$ (the multiplier). Perform all intermediate steps in *SD* arithmetic.

3.4. Given the following three pairs of binary dividend and divisor:
(i) +.1010 and −.1101 (ii) −.1010 and +.1101 (iii) −.1010 and −.1101.

Represent the numbers in the two's complement form and perform the division by the nonrestoring method. The quotient should also be represented in two's complement.

3.5. Devise an algorithm for dividing numbers in one's complement representation. Illustrate your algorithm using the three pairs of numbers in problem 3.4.

3.6. Write the rules for nonrestoring division for decimal fractions. Illustrate the procedure using a positive dividend and positive and negative divisors.

3.7. Explain the need for a correction step in the nonrestoring division if a zero remainder is encountered.

3.8. Show that if the quotient bits q_i $(i = 1, 2, \cdots, m)$ in the nonrestoring division are set according to the rule

$$q_i = \begin{cases} 1 & \text{if the signs of the remainder and divisor agree} \\ 0 & \text{if the signs of the remainder and divisor differ} \end{cases}$$

and subtraction (addition) is performed when $q_1 = 1$ $(q_1 = 0)$, then the correction term $(1 + 2^{-m})$ has to be added to $q_1.q_2 \cdots q_m$ to obtain a quotient represented in two's complement.

3.9. Can the algorithm for converting the quotient bits generated in nonrestoring division into two's complement representation be modified for converting binary *SD* numbers to two's complement? Explain.

3.10. To speed up the nonrestoring division, it has been suggested to allow 0 to be a quotient bit for which no add/subtract operation is needed in order to calculate a new remainder. The modified selection rule is

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq D \\ \bar{1} & \text{if } 2r_{i-1} < -D \\ 0 & \text{otherwise} \end{cases}$$

Apply this new algorithm to calculate the quotient of the dividend $X = 0.101$ and the divisor $D = 0.110$. Would you recommend the use of this new algorithm? Explain.

3.11. The on-the-fly conversion algorithm in subsection 3.3.1 is a special case of the *SD*-to-two's complement conversion algorithm presented in Section 2.4. Since just the values 1 and $\bar{1}$ are allowed we need only use the last four rows in Table 2.4. The resulting table for converting the quotient $0.p_1p_2 \cdots p_m$ to its two's complement equivalent $z_0.z_1z_2 \cdots z_m$ is shown below, with the indices changed to match the different indexing used here.

p_j	c_j	z_j	c_{j-1}
1	0	1	0
1	1	0	0
0	0	1	1
0	1	0	1

Show that $.z_1 z_2 \cdot z_m = .p_2 \cdots p_m 1$. Also show that based on the first two rows in Table 2.4 $z_0 = 1 - p_1$.

3.12. Find the square root of 0.011111 using the nonrestoring algorithm.

3.6 REFERENCES

[1] J. J. F. CAVANAGH, *Digital computer arithmetic: Design and implementation,* McGraw-Hill, New York, 1984.

[2] Y. CHU, *Computer organization and microprogramming,* Prentice Hall, Englewood Cliffs, NJ, 1972, chap. 5.

[3] M. D. ERCEGOVAC and T. LANG, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. on Computers, C-36* (July 1987), 895-897.

[4] K. HWANG, *Computer arithmetic: Principles, architecture, and design,* Wiley, New York, 1978.

[5] O. L. MACSORLEY, "High-speed arithmetic in binary computers," *Proc. of IRE, 49* (Jan. 1961), 67-91.

[6] G. W. REITWIESNER, "Binary arithmetic," in *Advances in computers,* vol. 1, F. L. Alt, (Editor), Academic, New York, 1960, pp. 231-308.

[7] J. E. ROBERTSON, "A new class of digital division methods," *IRE Trans. on Electronic Computers, EC-7* (Sept. 1958), 218-222.

[8] N. R. SCOTT, *Computer number systems and arithmetic,* Prentice Hall, Englewood Cliffs, NJ, 1985.

[9] C. TUNG, "Arithmetic," in *Computer science,* A. F. Cardenas et al. (Eds.), Wiley-Interscience, New York, 1972, chap. 3.

[10] S. WASER and M. J. FLYNN, *Introduction to arithmetic for digital system designers,* Holt, Rinehart, Winston, New York, 1982.