The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of a (see page 170.) Thus the sum is further simplified to

$$b \times a$$

Hence Booth's algorithm does in fact perform two's complement multiplication of a and b.

Summary

Multiplication is accomplished by a simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even replace multiplications by powers of 2 with shift instructions. Signed multiplication is more challenging, with Booth's algorithm rising to the challenge with essentially a clever factorization of the two's complement number representation of the multiplier.

Elaboration: The original reason for Booth's algorithm was speed, because early machines could shift faster than they could add. The hope was that this encoding scheme would increase the number of shifts. This algorithm is sensitive to particular bit patterns, however, and may actually increase the number of adds or subtracts. For example, bit patterns that alternate 0 and 1, called *isolated 1s*, will cause the hardware to add or subtract at each step. If all combinations occur with uniform distribution, then on average there is no savings. Greater advantage comes from performing multiple bits per step, which we explore in Exercise 4.39.

4.7 Division

Divide et impera.

Latin for "Divide and rule," Ancient political maxim cited by Machiavelli, 1532

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For

reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

```
                         1001ten   Quotient
Divisor 1000ten  |1001010ten  Dividend
                 -1000
                    10
                    101
                    1010
                   -1000
                      10ten        Remainder
```

The two operands (*dividend* and *divisor*) and the result (*quotient*) of divide are accompanied by a second result called the *remainder*. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

where the Remainder is smaller than the Divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient. Note that the size of the dividend is limited by the sum of the sizes of the divisor and quotient.

The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

Once again textbooks traditionally jump to the refined division hardware, and once again we abandon tradition to offer insight into how that hardware evolved. The next three subsections examine three versions of the divide algorithm, refining the hardware requirements as we go. Let's assume that both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative.

First Iteration of the Division Algorithm and Hardware

Figure 4.31 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each step of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right one bit each step to align it with the dividend.

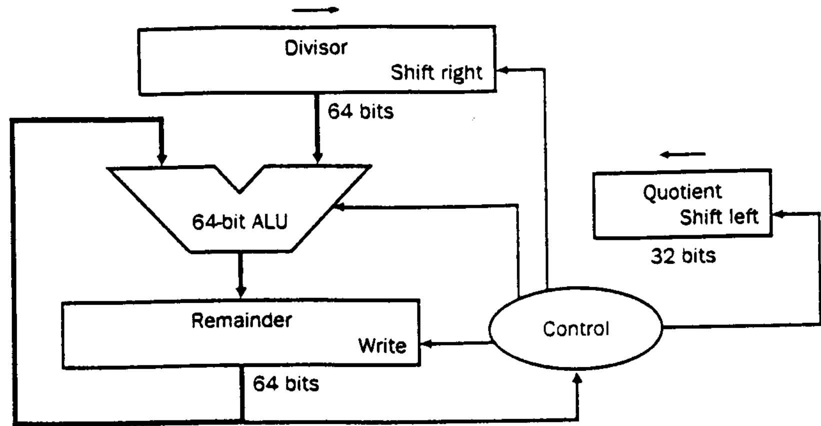

FIGURE 4.31 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each step. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Figure 4.32 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set-on-less-than instruction. If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder (step 2b). The remainder and quotient will be found in their namesake registers after the iterations are complete.

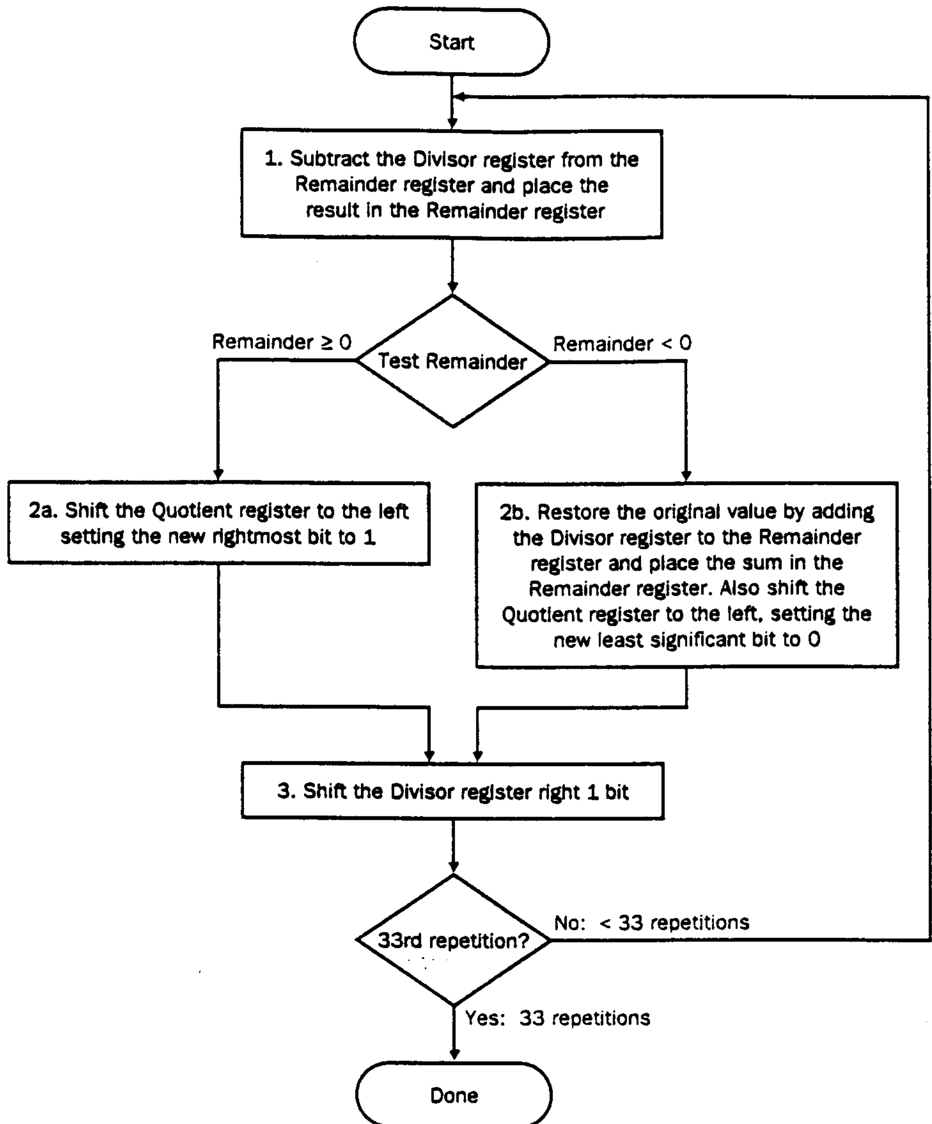

FIGURE 4.32 The first division algorithm, using the hardware in Figure 4.31. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after this step means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times; the reason for the apparent extra step will become clear in the next version of the algorithm.

Example

Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten} or $0000\ 0111_{two}$ by 0010_{two}.

Answer

Figure 4.33 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes n + 1 steps to get the proper quotient and remainder.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem<0 => +Div, sll Q, Q0 = 0	0011	0010 0000	0000 0111
	3: shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	(1)111 0111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	(1)111 1111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	(0)000 0011
	2a: Rem 0 => sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	(0)000 0001
	2a: Rem 0 => sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: shift Div right	0011	0000 0001	0000 0001

FIGURE 4.33 Division example using first algorithm in Figure 4.32.

Second Version of the Division Algorithm and Hardware

Once again the frugal computer pioneers recognized that, at most, half the divisor has useful information, and so both the divisor and ALU could potentially be cut in half. Shifting the remainder to the left instead of shifting the divisor to the right produces the same alignment and accomplishes the goal of simplifying the hardware necessary for the ALU and the divisor. Figure 4.34 shows the simplified hardware for the second version of the algorithm.

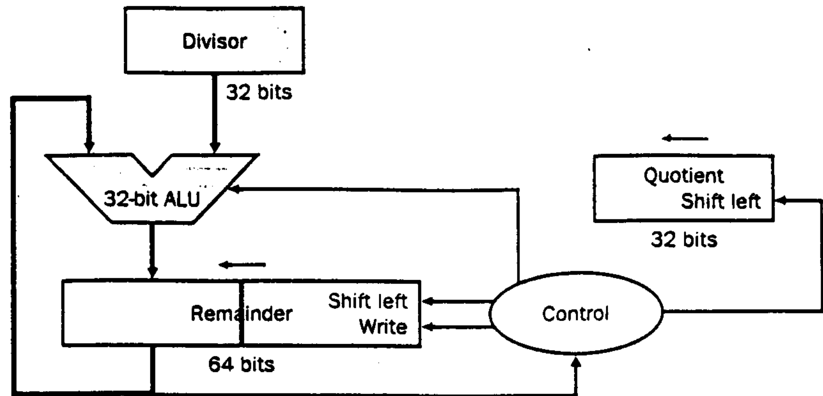

FIGURE 4.34 Second version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 4.31, the ALU and Divisor registers are halved and the remainder is shifted left. These changes are highlighted.

The second improvement comes from noticing that the first step of the current algorithm cannot produce a 1 in the quotient bit; if it did, then the quotient would be too large for the register. By switching the order of the operations to shift and then subtract, one iteration of the algorithm can be removed. Figure 4.35 shows the changes in this refined division algorithm. The remainder is now found in the left half of the Remainder register.

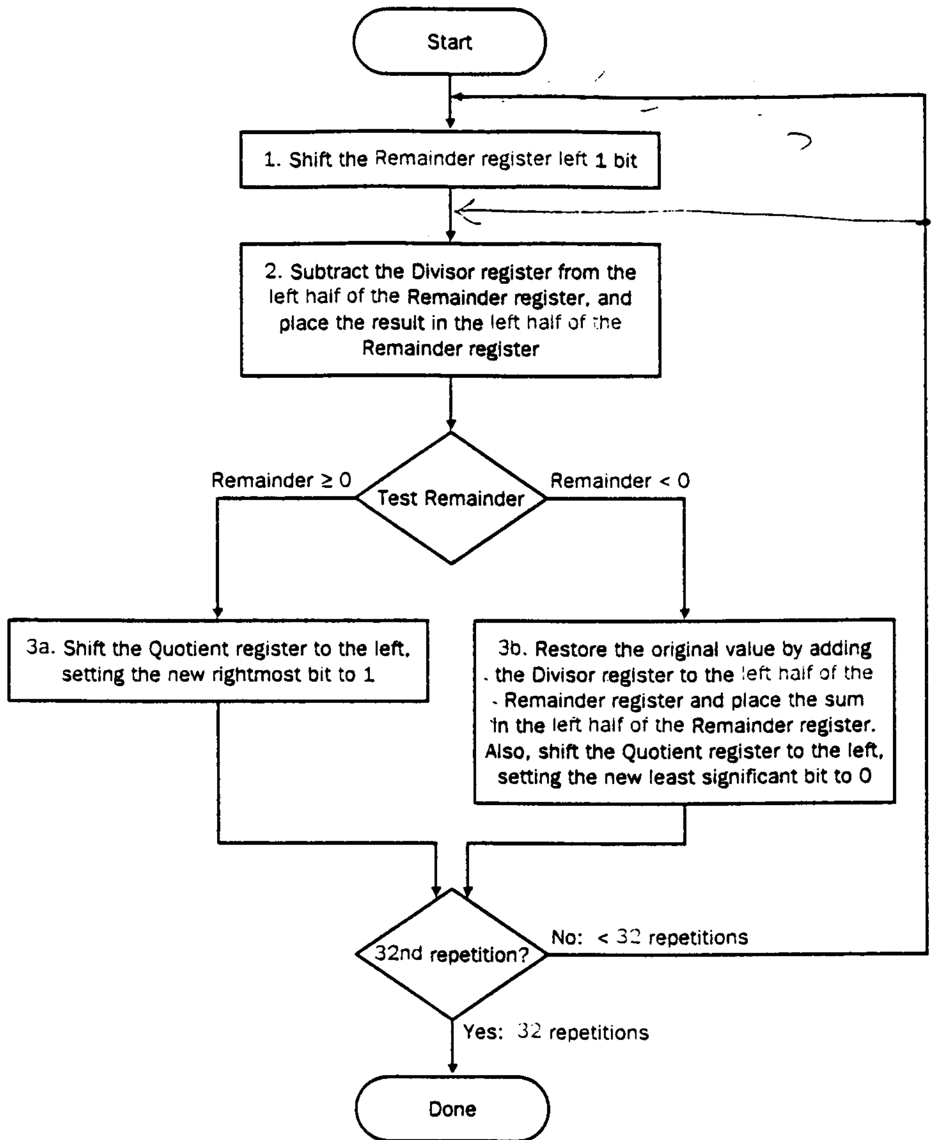

FIGURE 4.35 The second division algorithm, using the hardware in Figure 4.34. Unlike the first algorithm in Figure 4.32, only the left half of the remainder is changed, and the remainder is shifted left instead of the divisor being shifted right. Color type shows the changes from Figure 4.32.

Example Divide 0000 0111_{two} by 0010_{two} using the algorithm in Figure 4.35.

Answer The answer is summarized in Figure 4.36.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010	0000 0111
1	1: shift Rem left	0000	0010	0000 1110
	2: Rem = Rem – Div	0000	0010	1110 1110
	3b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0010	0000 1110
2	1: shift Rem left	0000	0010	0001 1100
	2: Rem = Rem – Div	0000	0010	1111 1100
	3b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0010	0001 1100
3	1: shift Rem left	0000	0010	0011 1000
	2: Rem = Rem – Div	0000	0010	0001 1000
	3a: Rem 0 => sll Q, Q0 = 1	0001	0010	0001 1000
4	1: shift Rem left	0001	0010	0011 0000
	2: Rem = Rem – Div	0001	0010	0001 0000
	3a: Rem 0 => sll Q, Q0 = 1	0011	0010	0001 0000

FIGURE 4.36 Division example using second algorithm in Figure 4.35.

Final Version of the Division Algorithm and Hardware

With the same insight and motivation as in the third version of the multiplication algorithm, computer pioneers saw that the Quotient register could be eliminated by shifting the bits of the quotient into the Remainder instead of shifting in 0s as in the preceding algorithm. Figure 4.37 shows the third version of the algorithm. We start the algorithm by shifting the Remainder left as before. Thereafter, the loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half (see Figure 4.38). The consequence of combining the two registers and the new order of the operations in the loop is that the remainder will be shifted left one time too many. Thus the final correction step must shift back only the remainder in the left half of the register.

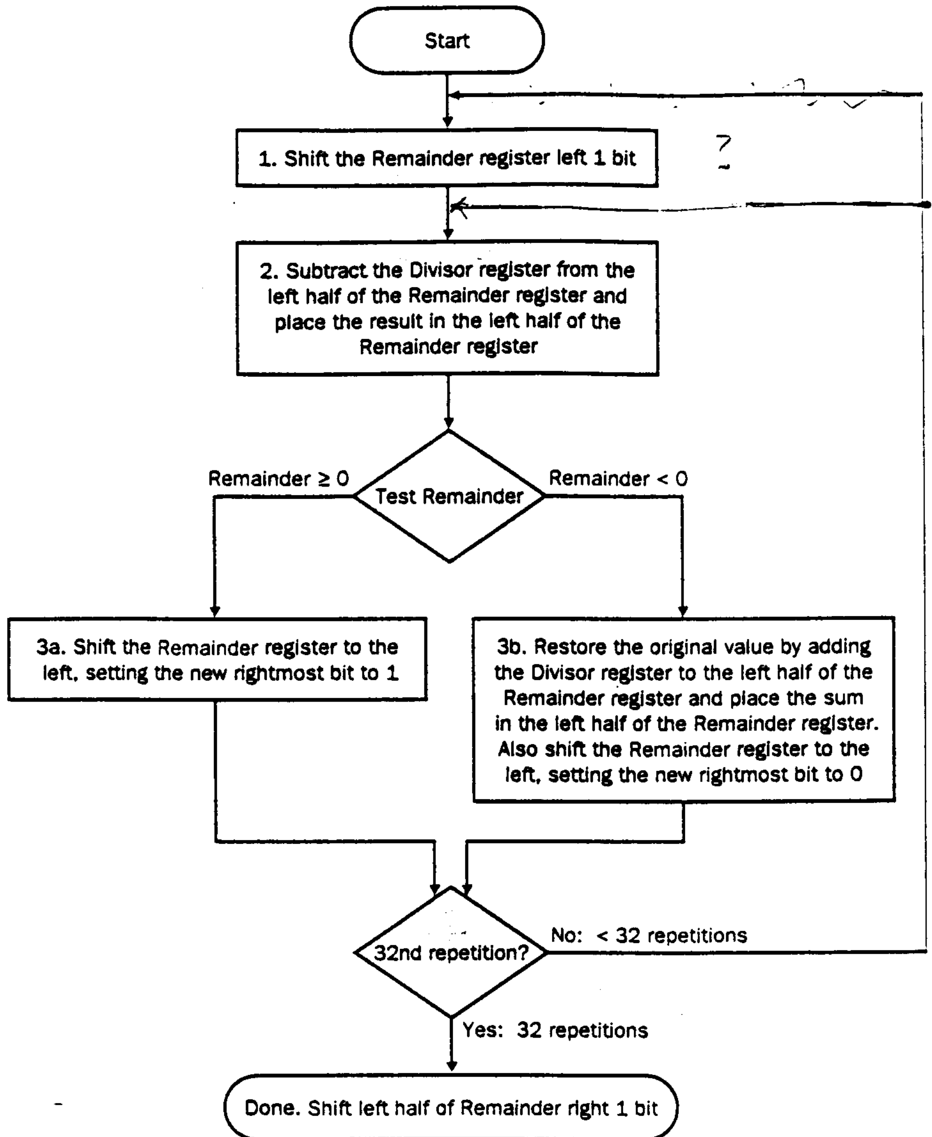

FIGURE 4.37 The third division algorithm has just two steps. The Remainder register shifts left, combining steps 1 and 3 in Figure 4.35 on page 218.

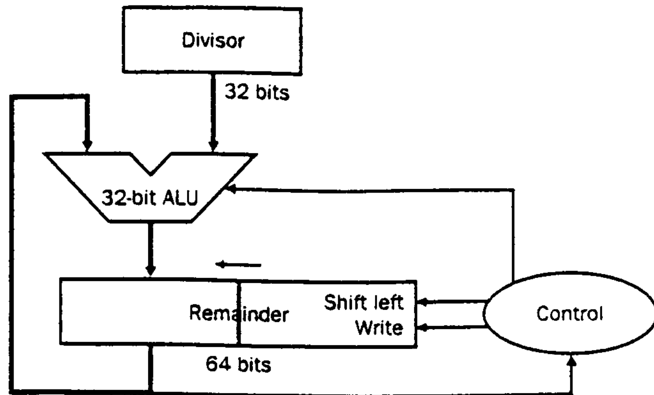

FIGURE 4.38 Third version of the division hardware. This version combines the Quotient register with the right half of the Remainder register.

Example

Use the third version of the algorithm to divide $0000\ 0111_{two}$ by 0010_{two}.

Answer

Figure 4.39 shows how the quotient is created in the bottom of the Remainder register and how both are shifted left in a single operation.

Iteration	Step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	1: Rem = Rem – Div	0010	0001 1110
	2b: Rem < 0 => +Div, sll R, R0 = 0	0010	0001 1100
2	1: Rem = Rem – Div	0010	0111 1100
	2b: Rem < 0 => +Div, sll R, R0 = 0	0010	0011 1000
3	1: Rem = Rem – Div	0010	0001 1000
	2a: Rem 0 => sll R, R0 = 1	0010	0011 0001
4	1: Rem = Rem – Div	0010	0001 0001
	2a: Rem 0 => sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

FIGURE 4.39 Division example using third algorithm in Figure 4.32.