4.6 Multiplication

Multiplication is vexation,
Division is as bad;
The rule of three doth puzzle me,
And practice drives me mad.

Anonymous, Elizabethan manuscript, 1570

With the construction of the ALU and explanation of addition, subtraction, and shifts, we are ready to build the more vexing operation of multiply.

But first let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. Multiplying 1000_{ten} by 1001_{ten}:

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

The first operand is called the *multiplicand* and the second the *multiplier.* The final result is called the *product.* As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above happened to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through three generations. The rest of this section presents successive refinements of the hardware and the algorithm until we have a version used in some computers. For now, let's assume that we are multiplying only positive numbers.

First Version of the Multiplication Algorithm and Hardware

The initial design mimics the algorithm we learned in grammar school; the hardware is shown in Figure 4.25. We have drawn the hardware so that data flows from top to bottom to more closely resemble the paper-and-pencil method.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step as it may be added to the intermediate products. Over 32 steps a

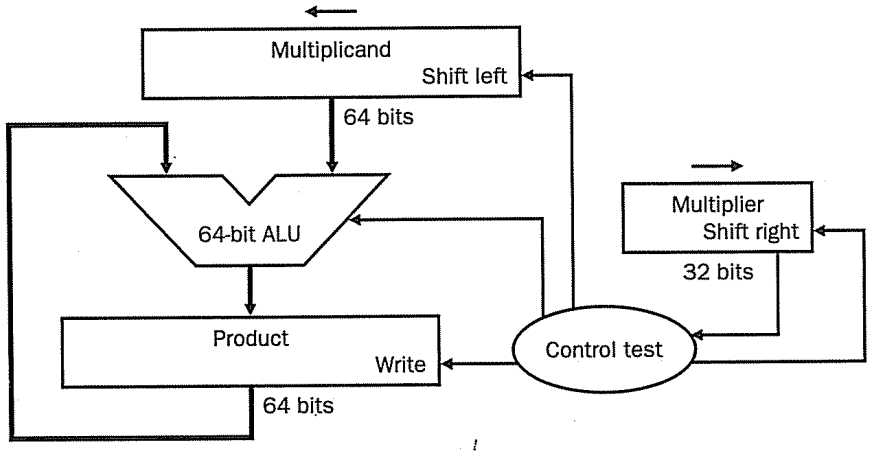


FIGURE 4.25 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register, and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

32-bit multiplicand would move 32 bits to the left. Hence we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and 0 in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

Figure 4.26 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is

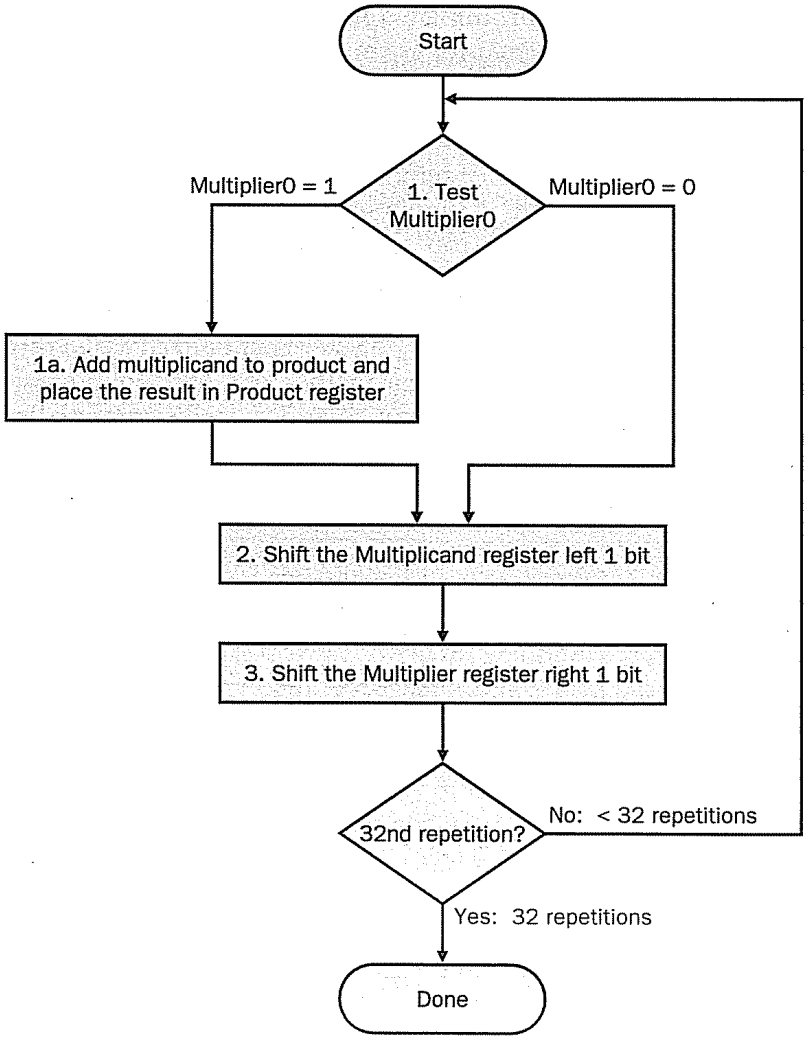


FIGURE 4.26 The first multiplication algorithm, using the hardware shown in Figure 4.25. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product.

First Multiply Algorithm

Example Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$, or $0010_{two} \times 0011_{two}$.

Answer Figure 4.27 shows the value of each register for each of the steps labeled according to Figure 4.26, with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000(1)	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 4.27 Multiply example using first algorithm in Figure 4.26. The bit examined to determine the next step is circled in color.

If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Chapter 2, page 75) reminds us that even a moderate frequency for a slow operation can limit performance.

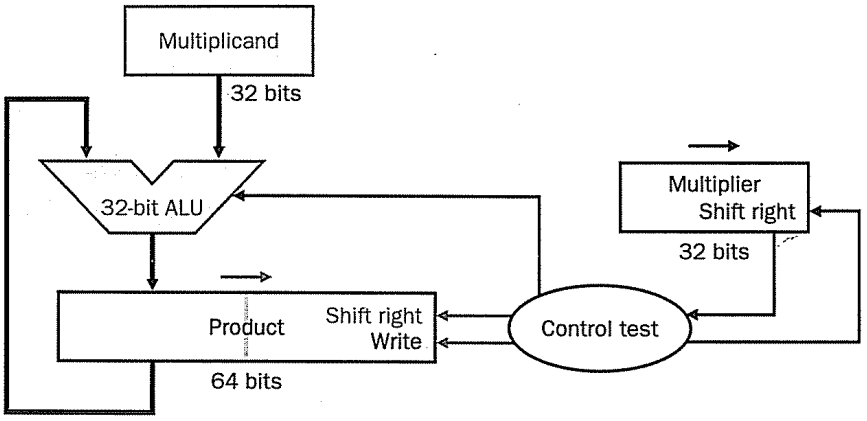


FIGURE 4.28 Second version of the multiplication hardware. Compare with the first version in Figure 4.25. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. These changes are highlighted in color.

Second Version of the Multiplication Algorithm and Hardware

Computer pioneers recognized that half of the bits of the multiplicand in the first algorithm were always 0, so only half could contain useful bit values. A full 64-bit ALU thus seemed wasteful and slow since half of the adder bits were adding 0 to the intermediate sum.

The original algorithm shifts the multiplicand left with 0s inserted in the new positions, so the multiplicand cannot affect the least significant bits of the product after they settle down. Instead of shifting the multiplicand left, they wondered, what if we shift the *product right*? Now the multiplicand would be fixed relative to the product, and since we are adding only 32 bits, the adder need be only 32 bits wide. Figure 4.28 shows how this change halves the widths of both the ALU and the multiplicand.

Figure 4.29 shows the multiply algorithm inspired by this observation. This algorithm starts with the 32-bit Multiplicand and 32-bit Multiplier registers set to their named values and the 64-bit Product register set to 0. This algorithm only forms a 32-bit sum, so only the left half of the 64-bit Product register is changed by the addition.

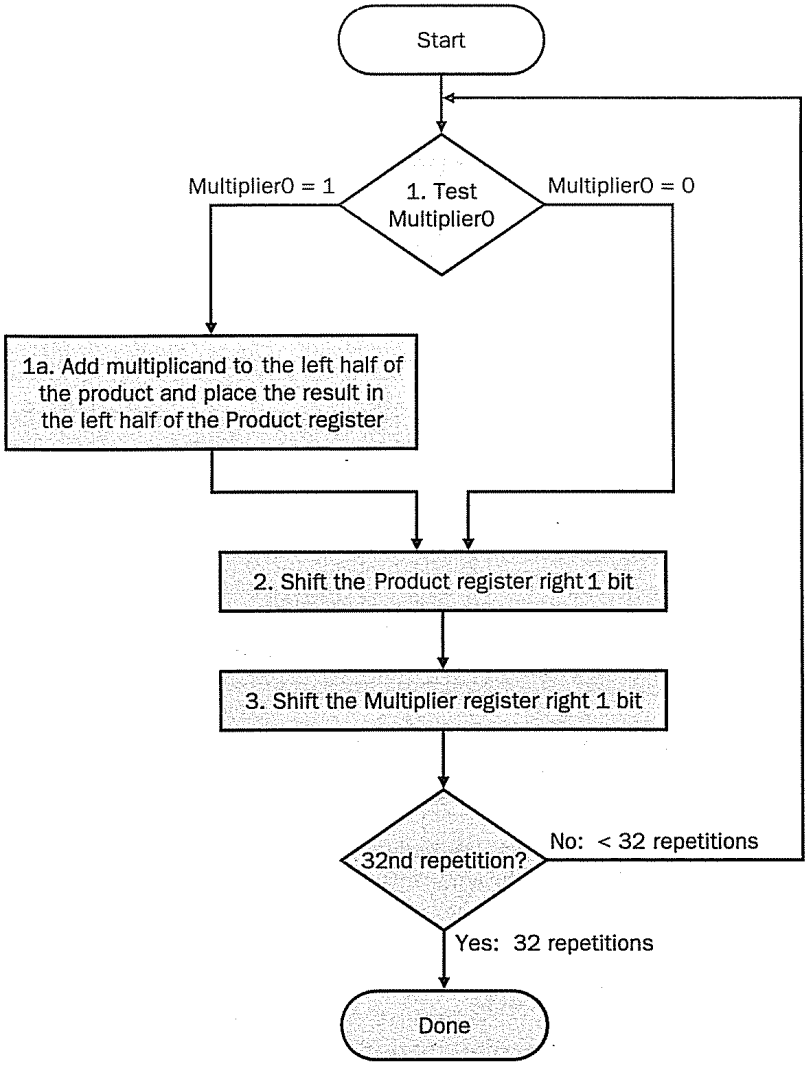


FIGURE 4.29 The second multiplication algorithm, using the hardware in Figure 4.28. In this version, the Product register is shifted right instead of shifting the multiplicand. Color type shows the changes from Figure 4.26.

Second Multiply Algorithm

Example Multiply $0010_{two} \times 0011_{two}$ using the algorithm in Figure 4.29.

Answer Figure 4.30 shows the revised 4-bit example, again giving a product of $0000\ 0110_{two}$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	000(1)	0010	0001 0000
2	1a: 1 => Prod = Prod + Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	000(0)	0010	0001 1000
3	1: 0 => no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	000(0)	0010	0000 1100
4	1: 0 => no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

FIGURE 4.30 Multiply example using second algorithm in Figure 4.29. The bit examined to determine the next step is circled in color.

Final Version of the Multiplication Algorithm and Hardware

The final observation of the frugal computer pioneers was that the Product register had wasted space that matched exactly the size of the multiplier: As the wasted space in the product disappears, so do the bits of the multiplier. In response, the third version of the multiplication algorithm combines the rightmost half of the product with the multiplier. Figure 4.31 shows the hardware. The least significant bit of the 64-bit Product register (Product0) now is the bit to be tested.

The algorithm starts by assigning the multiplier to the right half of the Product register, placing 0 in the upper half. Figure 4.32 shows the new steps.

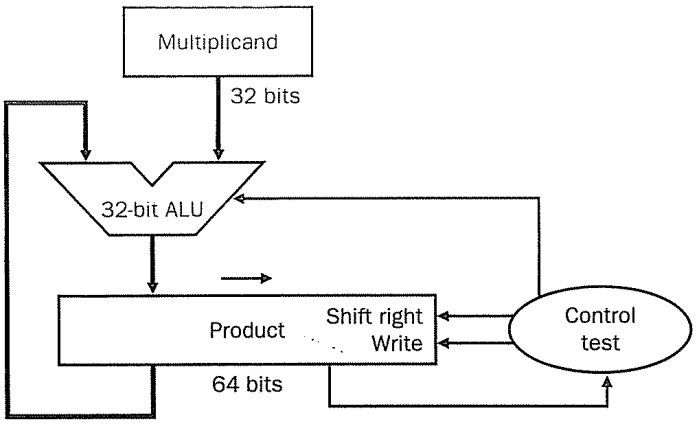


FIGURE 4.31 . Third version of the multiplication hardware. Comparing with the second version in Figure 4.28 on page 254, the separate Multiplier register has disappeared. The multiplier is placed instead in the right half of the Product register.

Third Multiply Algorithm

Example Multiply $0010_{two} \times 0011_{two}$ using the algorithm in Figure 4.32.

Answer Figure 4.33 shows the revised 4-bit example for the final algorithm.

Signed Multiplication

So far we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers provided that we remember that the numbers we are dealing with have infinite digits, and that we are only representing them with 32 bits. Hence the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

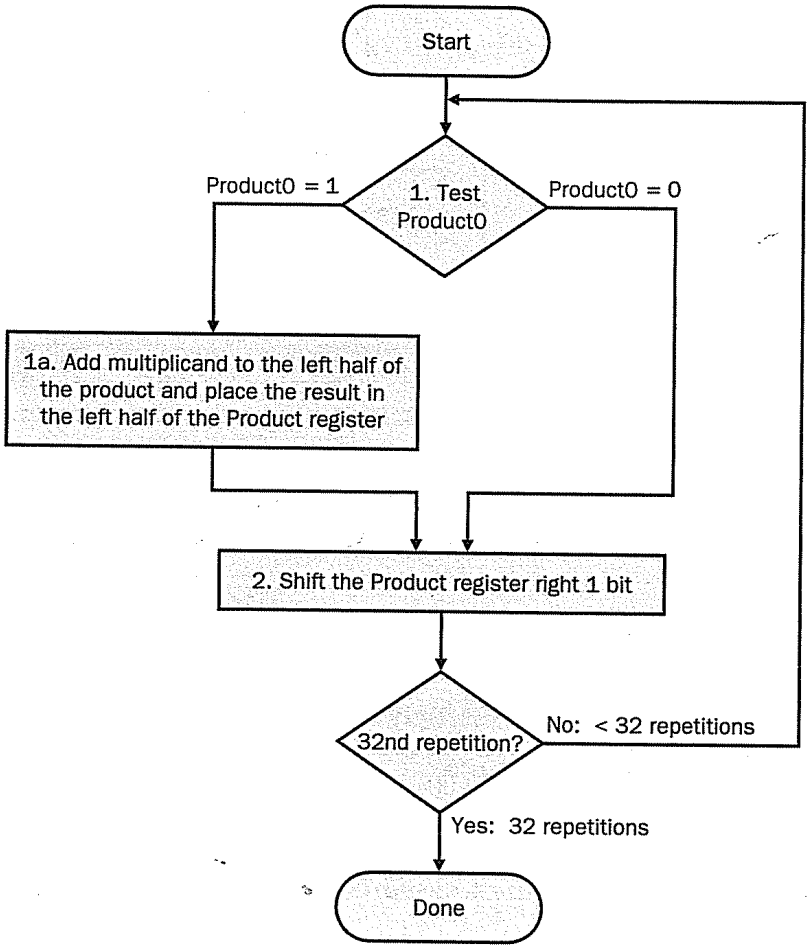


FIGURE 4.32 The third multiplication algorithm. It needs only two steps because the Product and Multiplier registers have been combined. Color type shows changes from Figure 4.29.

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

FIGURE 4.33 Multiply example using third algorithm in Figure 4.32. The bit examined to determine the next step is circled in color.

Booth's Algorithm

A more elegant approach to multiplying signed numbers than above is called *Booth's algorithm*. It starts with the observation that with the ability to both add and subtract there are multiple ways to compute a product. Suppose we want to multiply 2_{ten} by 6_{ten}, or 0010_{two} by 0110_{two}:

```
             0010two
x            0110two
        ---------
+            0000   shift (0 in multiplier)
+           0010    add   (1 in multiplier)
+          0010     add   (1 in multiplier)
+         0000      shift (0 in multiplier)
        -------------
          00001100two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$$6_{ten} = -2_{ten} + 8_{ten}$$

or

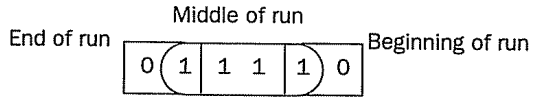
$$0110_{two} = -0010_{two} + 1000_{two}$$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

```
          0010two
x         0110two
        ---------
+         0000 shift (0 in multiplier)
-        0010  sub (first 1 in multiplier)
+       0000   shift (middle of string of 1s)
+      0010    add (prior step had last 1)
      ----------
      00001100two
```

Booth invented this approach in a quest for speed because in machines of his era shifting was faster than addition. Indeed, for some patterns his algorithm would be faster; it's our good fortune that it handles signed numbers as

well, and we'll prove this later. The key to Booth's insight is in his classifying groups of bits into the beginning, the middle, or the end of a run of 1s:



Of course, a string of 0s already avoids arithmetic, so we can leave these alone.

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	00001111000_{two}
1	1	Middle of a run of 1s	00001111000_{two}
0	1	End of a run of 1s	00001111000_{two}
0	0	Middle of a run of 0s	00001111000_{two}

Booth's algorithm changes the first step of the algorithm in Figure 4.32—looking at 1 bit of the multiplier and then deciding whether to add the multiplicand—to looking at 2 bits of the multiplier. The new first step, then, has four cases, depending on the values of the 2 bits. Let's assume that the pair of bits examined consists of the current bit and the bit to the right—which was the current bit in the previous step. The second step is still to shift the product right. The new algorithm is then the following:

1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

Now we are ready to begin the operation, shown in Figure 4.34. It starts with a 0 for the mythical bit to the right of the rightmost bit for the first stage. Figure 4.34 compares the two algorithms, with Booth's on the right. Note that

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod − Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

FIGURE 4.34 Comparing algorithm in Figure 4.32 and Booth's algorithm for positive numbers. The bit(s) examined to determine the next step is circled in color.

Booth's operation is now identified according to the values in the 2 bits. By the fourth step, the two algorithms have the same values in the Product register.

The one other requirement is that shifting the product right must preserve the sign of the intermediate result, since we are dealing with signed numbers. The solution is to extend the sign when the product is shifted to the right. Thus, step 2 of the second iteration turns $1110\ 0011\ 0_{two}$ into $1111\ 0001\ 1_{two}$ instead of $0111\ 0001\ 1_{two}$. This shift is called an *arithmetic right shift* to differentiate it from a logical right shift.

Booth's Algorithm

Example Let's try Booth's algorithm with negative numbers: $2_{ten} \times -3_{ten} = -6_{ten}$, or $0010_{two} \times 1101_{two} = 1111\ 1010_{two}$.

Answer Figure 4.35 shows the steps.

Our example multiplies one bit at a time, but it is possible to generalize Booth's algorithm to generate multiple bits for faster multiplies (see Exercise 4.53).

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

FIGURE 4.35 Booth's algorithm with negative multiplier example. The bits examined to determine the next step are circled in color.

Hardware Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts, adds, and subtracts. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2, so almost every compiler will substitute a left shift for a multiply by a power of 2.

Multiply by 2^i via Shift

Example Let's multiply 5_{ten} by 2_{ten} using a left shift by 1.

Answer Given that

$$101_{two} = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)_{ten} = 4 + 0 + 1_{ten} = 5_{ten}$$

if we shift left 1 bit, we get

$$1010_{two} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)_{ten}$$
$$= 8 + 0 + 2 + 0_{ten} = 10_{ten}$$

and

$$5 \times 2^1{}_{ten} = 10_{ten}$$

Hence the MIPS sll instruction can be used for multiplies by powers of 2.

Now that we have seen Booth's algorithm work, we are ready to see *why* it works for two's complement signed integers. Let a be the multiplier and b be the multiplicand and we'll use a_i to refer to bit i of a. Recasting Booth's algorithm in terms of the bit values of the multiplier yields this table:

a_i	a_{i-1}	Operation
0	0	Do nothing
0	1	Add b
1	0	Subtract b
1	1	Do nothing

Instead of representing Booth's algorithm in tabular form, we can represent it as the expression

$$(a_{i-1} - a_i)$$

where the value of the expression means the following actions:

0 : do nothing
+1: add b
–1: subtract b

Since we know that shifting of the multiplicand left with respect to the Product register can be considered multiplying by a power of 2, Booth's algorithm can be written as the sum

$$\begin{array}{lll} & (a_{-1} - a_0) \times b \times 2^0 \\ + & (a_0 - a_1) \times b \times 2^1 \\ + & (a_1 - a_2) \times b \times 2^2 \\ \ldots & \ldots \\ + & (a_{29} - a_{30}) \times b \times 2^{30} \\ + & (a_{30} - a_{31}) \times b \times 2^{31} \end{array}$$

We can simplify this sum by noting that

$$-a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = (2a_i - a_i) \times 2^i = a_i \times 2^i$$

recalling that $a_{-1} = 0$ and by factoring out b from each term:

$$b \times ((a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + (a_{29} \times 2^{29}) + \ldots + (a_1 \times 2^1) + (a_0 \times 2^0))$$

The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of a (see page 213.) 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.

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Hardware Software Interface

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. To avoid overflow, Hi must be 0 for `multu` or must be the replicated sign of Lo for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer Hi to a general-purpose register to test for overflow.

Summary

Multiplication is accomplished by simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of two. 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. Looking at more bits to carefully avoid isolated 1s can reduce the number of adds in the worst case. Greater advantage comes from performing multiple bits per step, which we explore in Exercise 4.53.

Even faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder. When adding such a large column of numbers, a *carry save adder* is useful (see Exercises 4.49 to 4.52).

Elaboration: The replacement of a multiply by a shift, as in the example on page 262, is an instance of a general compiler optimization strategy called *strength reduction*.

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}:

$$
\begin{array}{rrl}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{ten} & \text{Remainder}
\end{array}
$$

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.

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.

Let's assume that both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values, and we will ignore the sign for now. Rather than make

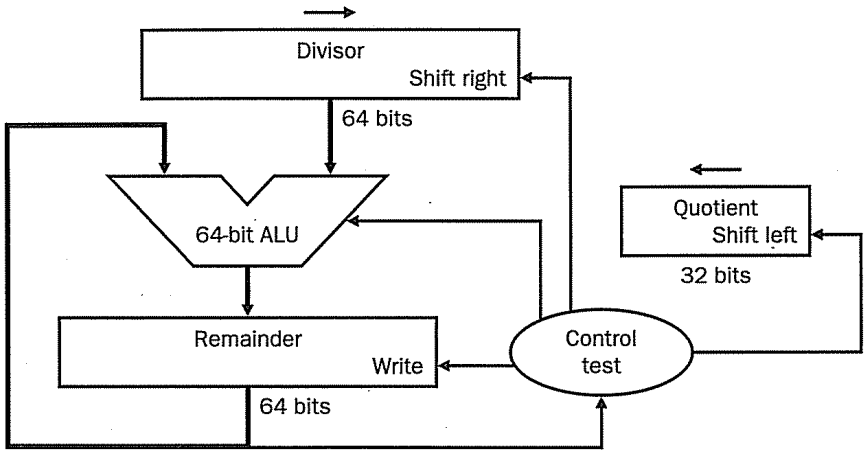


FIGURE 4.36 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each step. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

each of the three evolutionary steps explicit with drawings and examples, as we did for multiply, we will save space by giving a sketch for the two intermediate steps and then give the final algorithm in detail.

First Version of the Division Algorithm and Hardware

Figure 4.36 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 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 4.37 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 positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

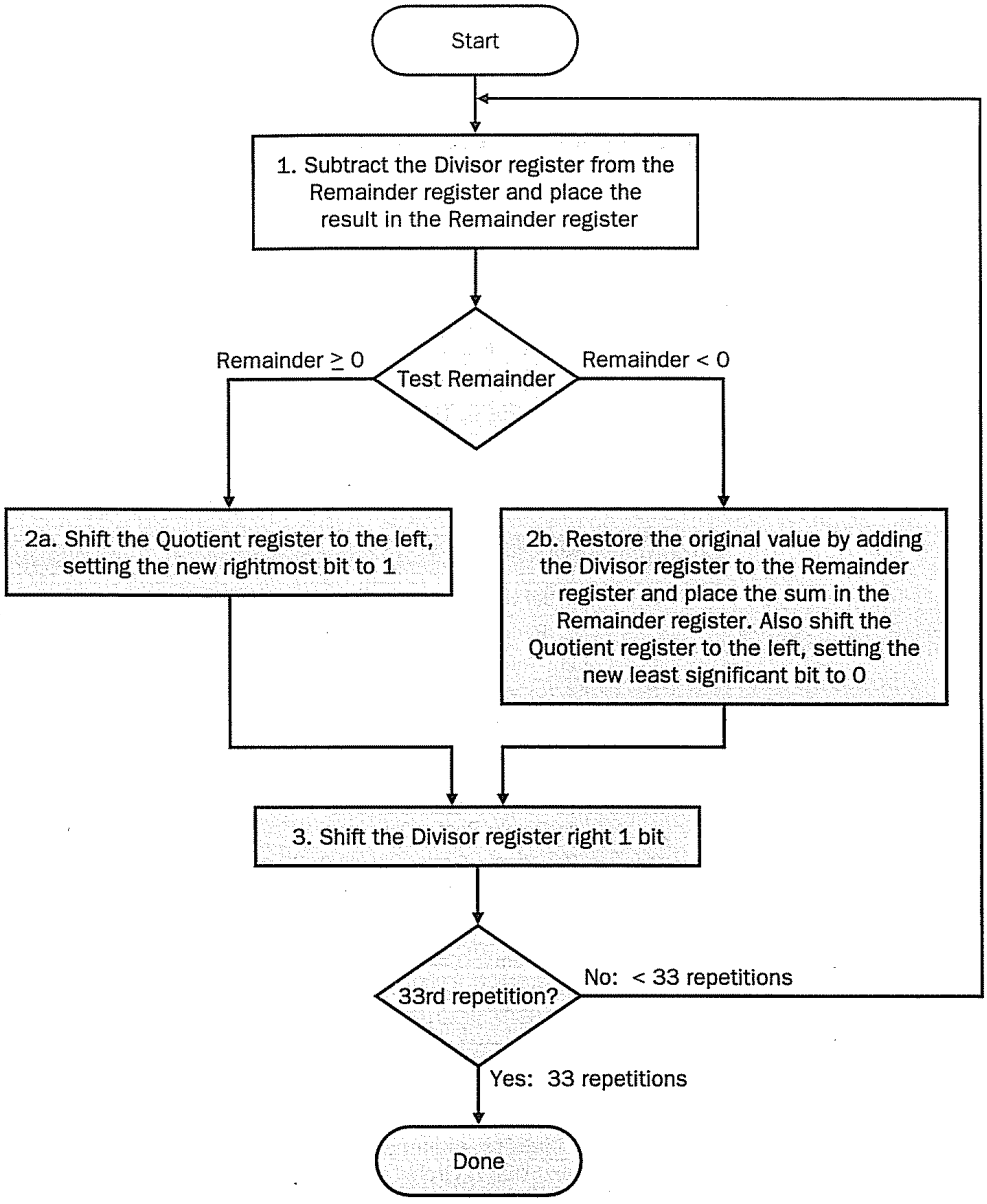


FIGURE 4.37 The first division algorithm, using the hardware in Figure 4.36. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in 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.

First Divide 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 4.38 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	①110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 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	①111 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	⓪000 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	⓪000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 4.38 Division example using first algorithm in Figure 4.37. The bit examined to determine the next step is circled in color.

Second Version of the Division Algorithm and Hardware

Once again the frugal computer pioneers recognized that, at most, half of 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 4.39 shows the simplified hardware for the second version of the algorithm.

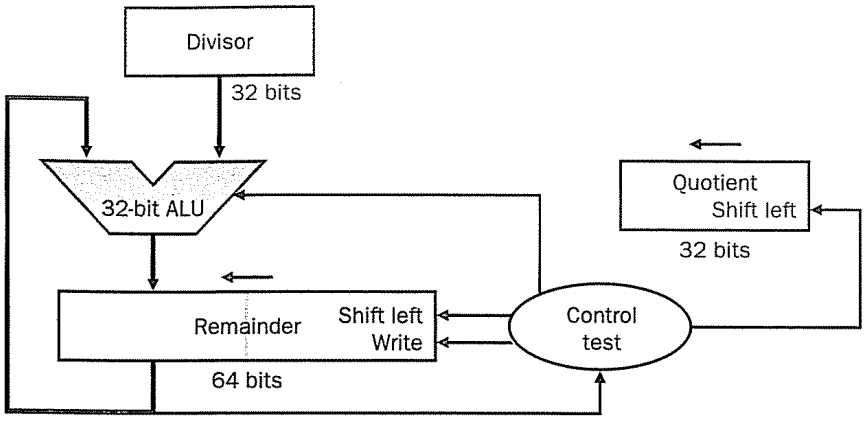


FIGURE 4.39 Second version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 4.36, the ALU and Divisor registers are halved and the remainder is shifted left. These changes are highlighted.

Another change 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. When the algorithm terminates, the remainder will be found in the left half of the Remainder register.

Final Version of 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 4.40 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 4.41). 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.

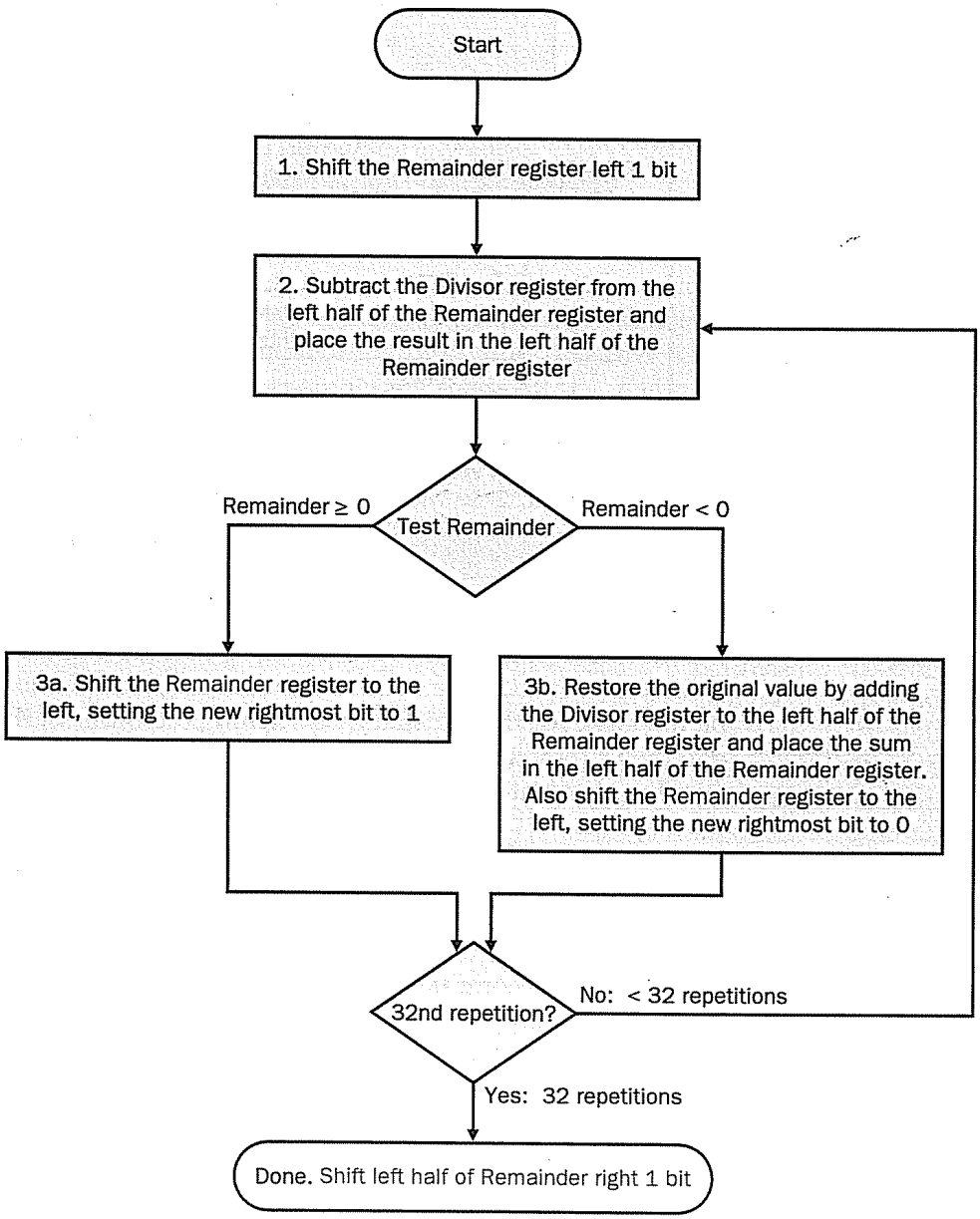


FIGURE 4.40 The third division algorithm has just two steps. The Remainder register shifts left.

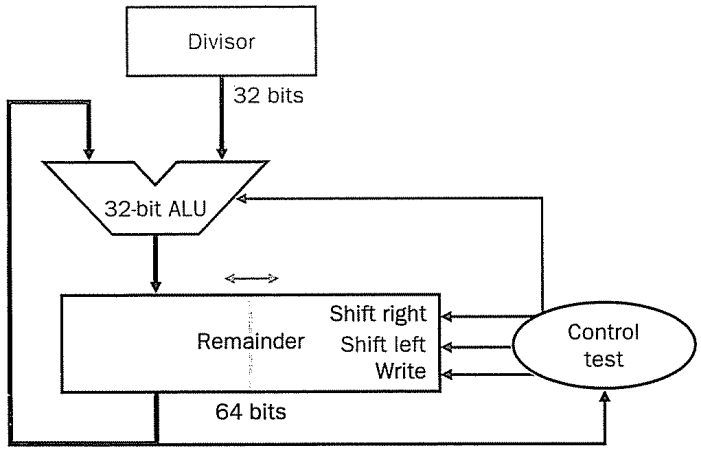


FIGURE 4.41 Third version of the division hardware. This version combines the Quotient register with the right half of the Remainder register.

Third Divide Algorithm

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

Answer Figure 4.42 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	2: Rem = Rem – Div	0010	①110 1110
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	①111 1100
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	⓪001 1000
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	⓪001 0001
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

FIGURE 4.42 Division example using third algorithm in Figure 4.40. The bit examined to determine the next step is circled in color.