3

Arithmetic for Computers

Abstract

This chapter describes how computers handle arithmetic data. It explains that computer words (which are composed of bits) can be represented as binary numbers. While integers can be easily converted to decimal or binary form, fractions and other real numbers can also be converted, as this chapter shows. This chapter also explains what happens if an operation creates a number bigger than can be represented, and answers the question of how the computer hardware multiplies and divides numbers.

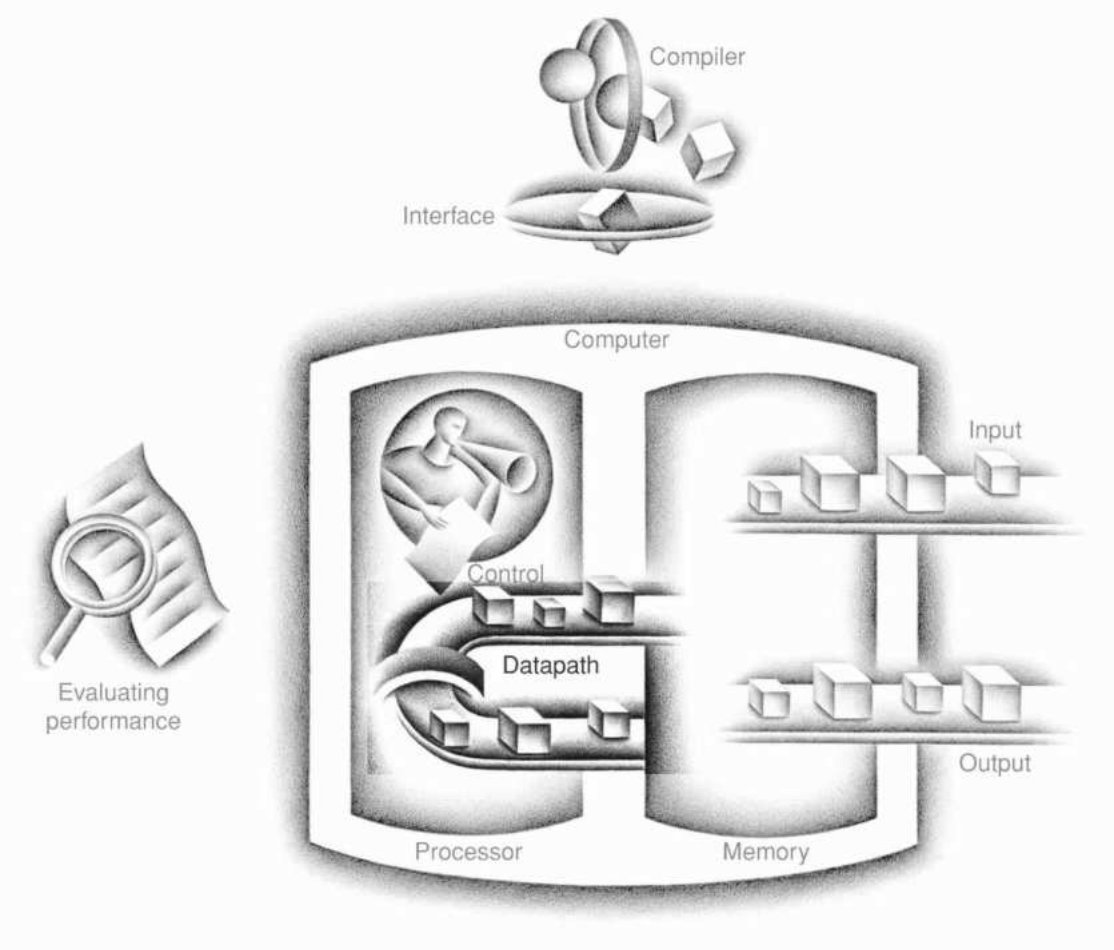
Keywords

Computer arithmetic; addition; subtraction; multiplication; division; associativity; x86; Arithmetic Logic Unit; ALU; exception; interrupt; dividend; divisor; quotient; remainder; scientific notation; normalized; floating point; fraction; exponent; overflow; underflow; double precision; single precision; guard; round; units in the last place; ULP; sticky bit; fused multiply add; matrix multiply; SIMD; subword parallelism

Numerical precision is the very soul of science.

Sir D'arcy Wentworth Thompson, On Growth and Form, 1917

OUTLINE



The Five Classic Components of a Computer

3.1 Introduction

Computer words are composed of bits; thus, words can be represented as binary numbers. Chapter 2 shows that integers can be represented either in decimal or binary form, but what about the other numbers that commonly occur? For example:

- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel these mysteries—including representation of real numbers, arithmetic algorithms, hardware that follows these algorithms—and the implications of all this for instruction sets. These insights may explain quirks that you have

already encountered with computers. Moreover, we show how to use this knowledge to make arithmetic-intensive programs go much faster.

3.2 Addition and Subtraction

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., Book of Top Ten Lists, 1990

Binary Addition and Subtraction

Example

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ + & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_{two} = 13_{ten} \end{array}$$

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. Parentheses identify the carries, with the arrows illustrating how they are passed.

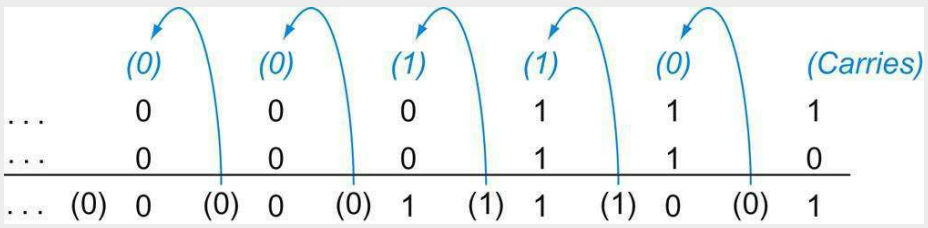


FIGURE 3.1 Binary addition, showing carries from

right to left.

The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 +1 +1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 +1 +1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 +0 +0, yielding a 1 sum and no carry.

Answer

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ - & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ + & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 64-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, −10 +4 =−6. Since the operands fit in 64 bits and the sum is no larger than an operand, the sum must fit in 64 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $c-a=c+(-a)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot

occur in this case either.

Knowing when an overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? Clearly, adding or subtracting two 64-bit numbers can yield a result that needs 65 bits to be fully expressed.

The lack of a 65th bit means that when an overflow occurs, the sign bit is set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a borrow occurred from the sign bit. Figure 3.2 shows the combination of operations, operands, and results that indicate an overflow.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

We have just seen how to detect overflow for two's complement numbers in a computer. What about overflow with unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

Fortunately, the compiler can easily check for unsigned overflow using a branch instruction. Addition has overflowed if the sum is less than either of the addends, whereas subtraction has overflowed if the difference is greater than the minuend.

Appendix A describes the hardware that performs addition and subtraction, which is called an **Arithmetic Logic Unit** or **ALU**.

Arithmetic Logic Unit (ALU)

Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

Hardware/Software Interface

The computer designer must decide how to handle arithmetic overflows. Although some languages like C and Java ignore integer overflow, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when an overflow occurs.

Summary

A major point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require recognizing overflow, so today all computers have a way to detect it.

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas RISC-V only has integer arithmetic operations on full words. As we recall from Chapter 2, RISC-V does have data transfer operations for bytes and halfwords. What RISC-V instructions should be generated for byte and halfword arithmetic operations?

1. Load with `lb, lh`; arithmetic with `add, sub, mul, div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb, sh`.
2. Load with `lb, lh`; arithmetic with `add, sub, mul, div`; then store using `sb, sh`.

Elaboration

One feature not generally found in general-purpose microprocessors is saturating operations. *Saturation* means that when a calculation overflows, the result is set to the largest positive number or the most negative number, rather than a modulo calculation as in two's complement arithmetic. Saturation is likely what you want for media operations. For example, the volume knob on a radio set would be frustrating if, as you turned it, the volume would get continuously louder for a while and then immediately very soft. A knob with saturation would stop at the highest volume no matter how far you turned it. Multimedia extensions to standard instruction sets often offer saturating arithmetic.

Elaboration

The speed of addition depends on how quickly the carry into the high-order bits is computed. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry. The most popular is *carry lookahead*, which Section A.6 in Appendix A describes.

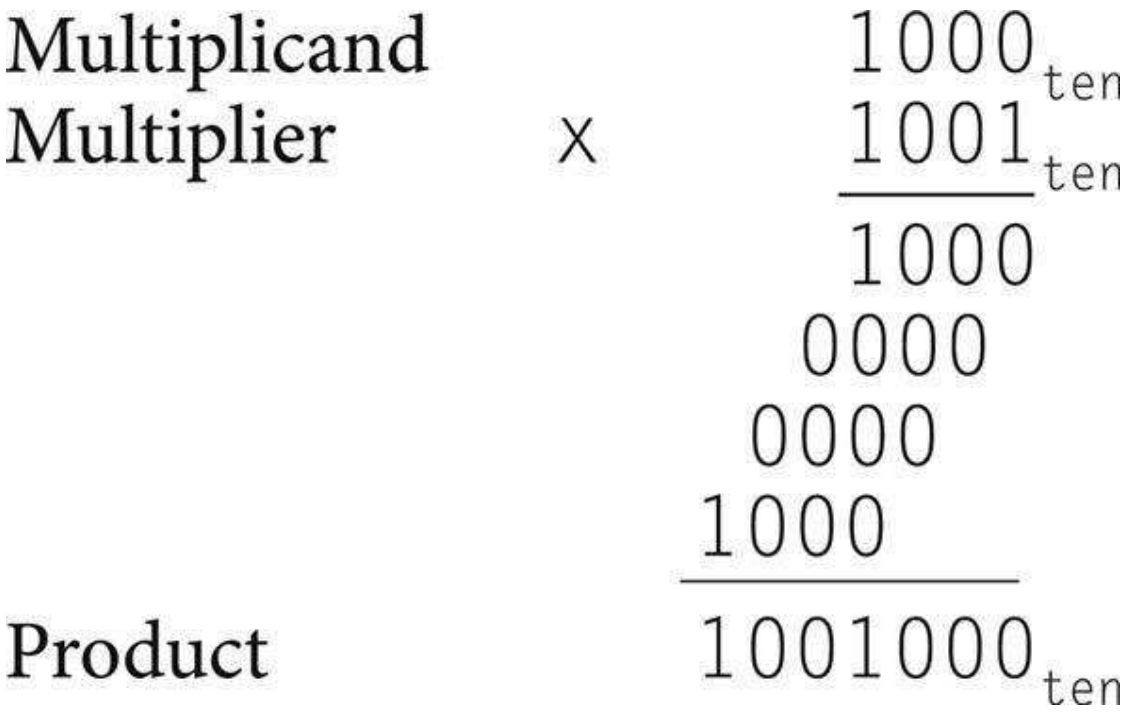
3.3 Multiplication

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiplication.

> *Multiplication is vexation, Division is as bad; The rule of three doth puzzle me, And practice drives me mad.*
>
> *Anonymous, Elizabethan manuscript, 1570*

First, let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps of multiplication 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}:



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 64-bit product as the result of multiplying two 64-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 happens 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 multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; Figure 3.3 shows the hardware. We have drawn the hardware so that data flow from top to bottom to resemble more closely the paper-and-pencil method.

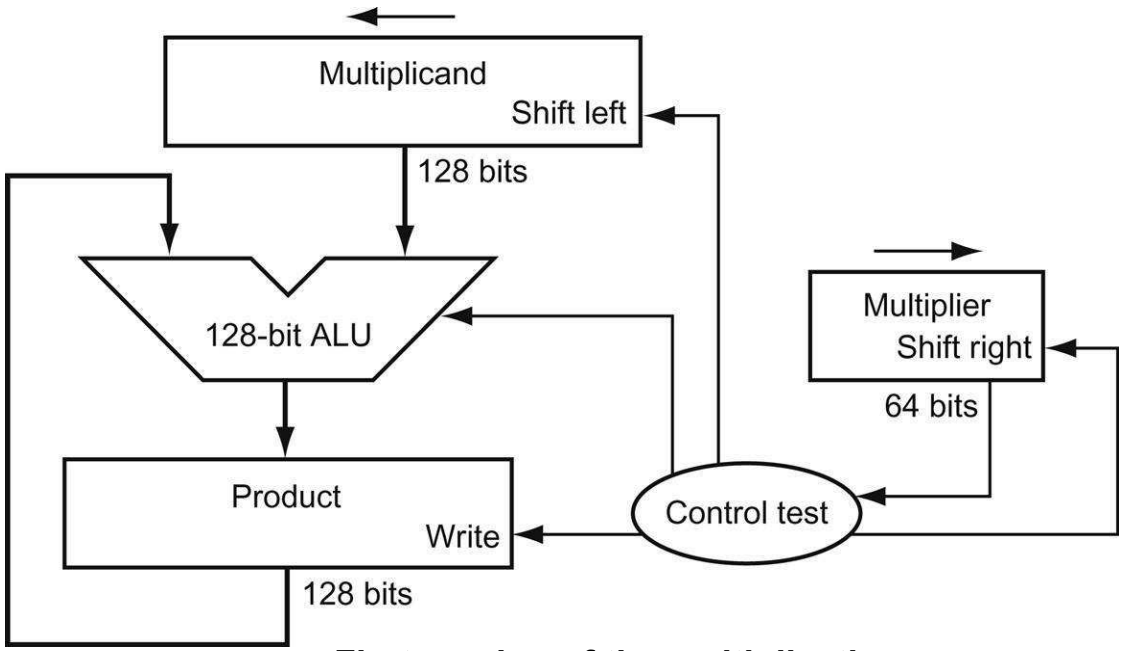


FIGURE 3.3 First version of the multiplication hardware.
The Multiplicand register, ALU, and Product register are all 128 bits wide, with only the Multiplier register containing 64 bits. (Appendix A describes ALUs.) The 64-bit multiplicand starts in the right half of the

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.

Let's assume that the multiplier is in the 64-bit Multiplier register and that the 128-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 64 steps, a 64-bit multiplicand would move 64 bits to the left. Hence, we need a 128-bit Multiplicand register, initialized with the 64-bit multiplicand in the right half and zero 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 128-bit Product register.

Figure 3.4 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is 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 with paper and pencil. 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 64 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 200 clock cycles to multiply two 64-bit numbers. 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 several clock cycles without significantly affecting performance. However, Amdahl's Law (see Section 1.10) reminds us that even a moderate frequency for a slow operation can limit performance.

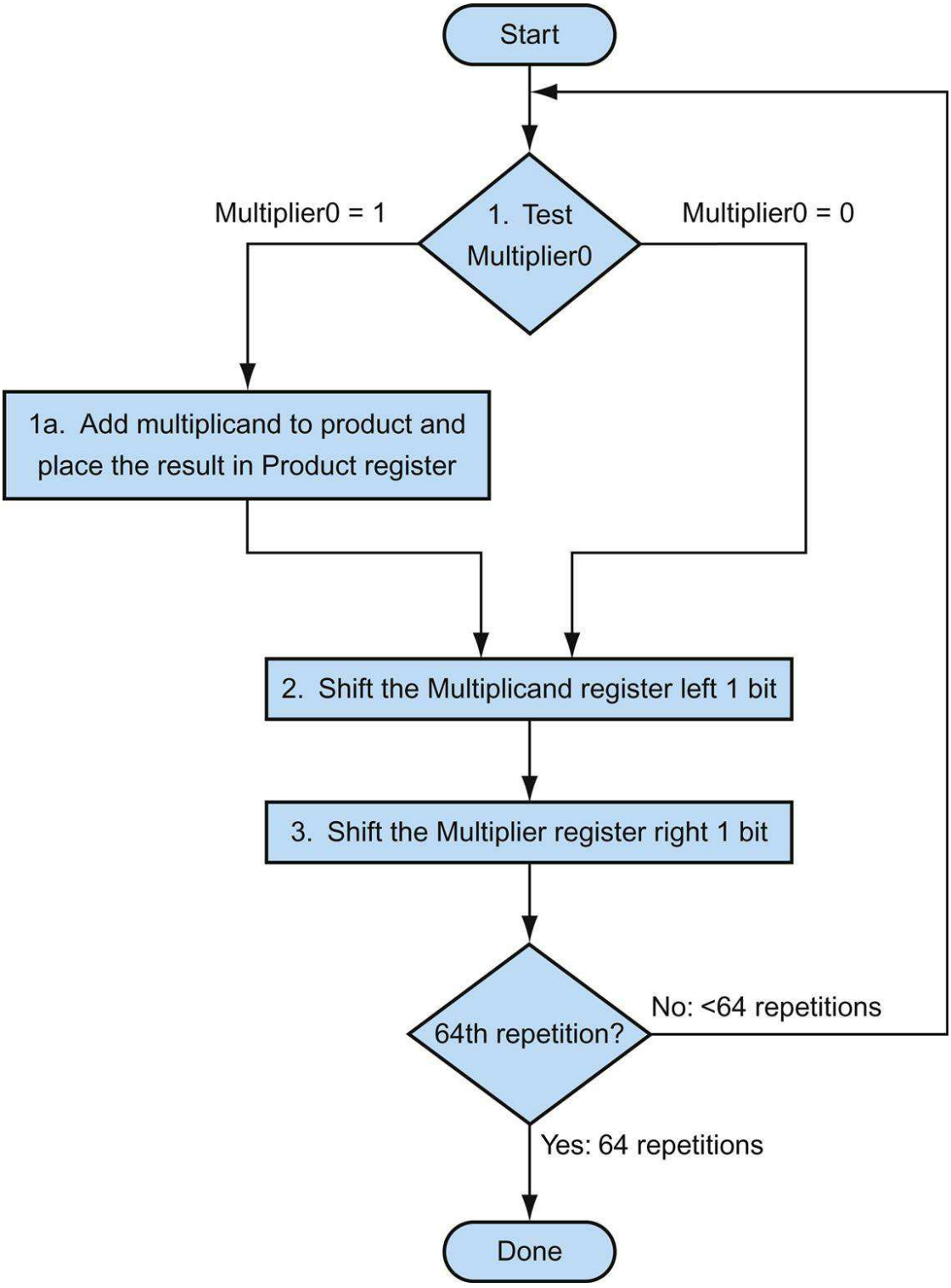


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3.
If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 64 times.

This algorithm and hardware are easily refined to take one clock cycle per step. The speed up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.5 shows the revised hardware.

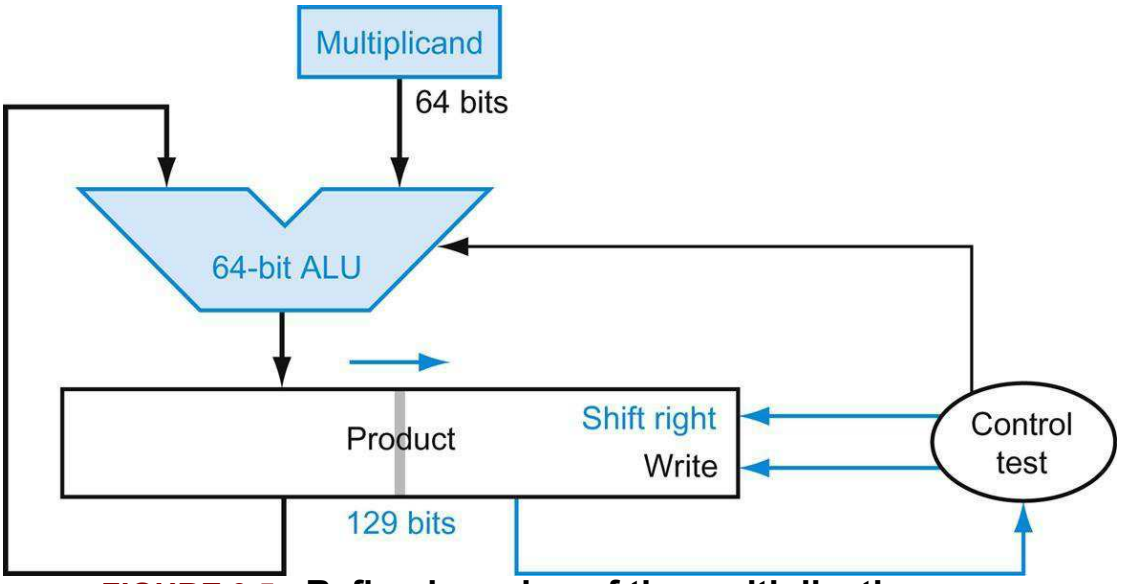


FIGURE 3.5 Refined version of the multiplication hardware.
Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 129 bits to hold the carry-out of the adder. These changes are highlighted 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 and adds. 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. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

A Multiply Algorithm

Example

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

Answer

Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, 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 3.6 Multiply example using algorithm in Figure 3.4.
The bit examined to determine the next step is circled in color.

Signed Multiplication

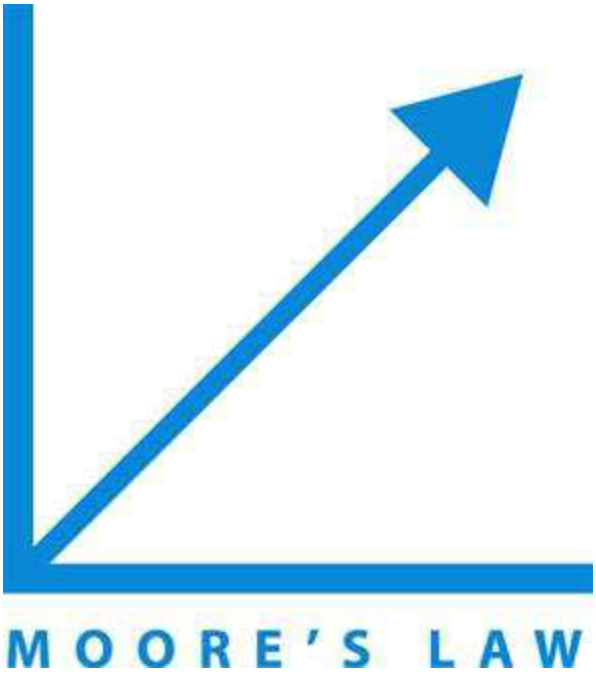
So far, we have dealt with positive numbers. The easiest way to

understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember their original signs. The algorithms should next 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, if we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 64 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower doubleword would have the 64-bit product.

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 64 multiplier bits. Faster multiplications are possible by essentially providing one 64-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.



A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 64 high. An alternative way to organize these 64 additions is in a parallel tree, as Figure 3.7 shows. Instead of waiting for 64 add times, we wait just the $\log_2$ (64) or six 64-bit add times.

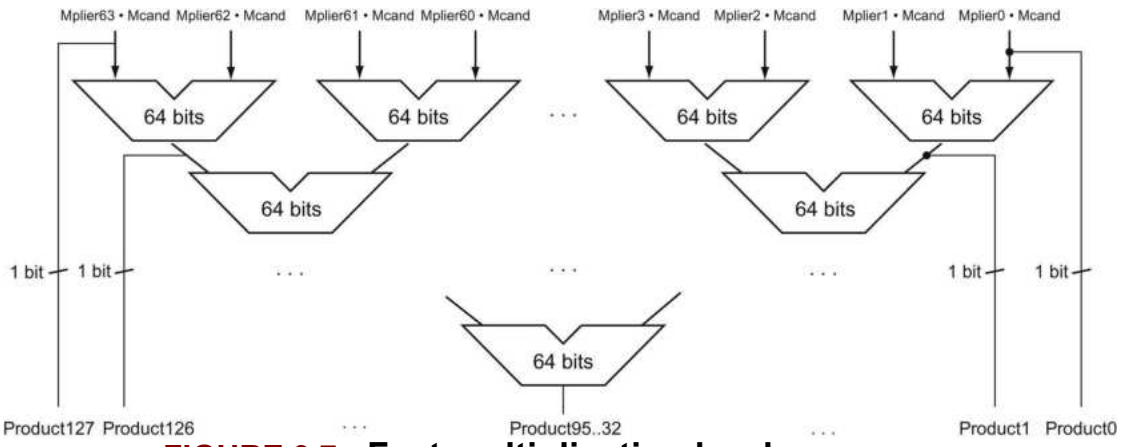
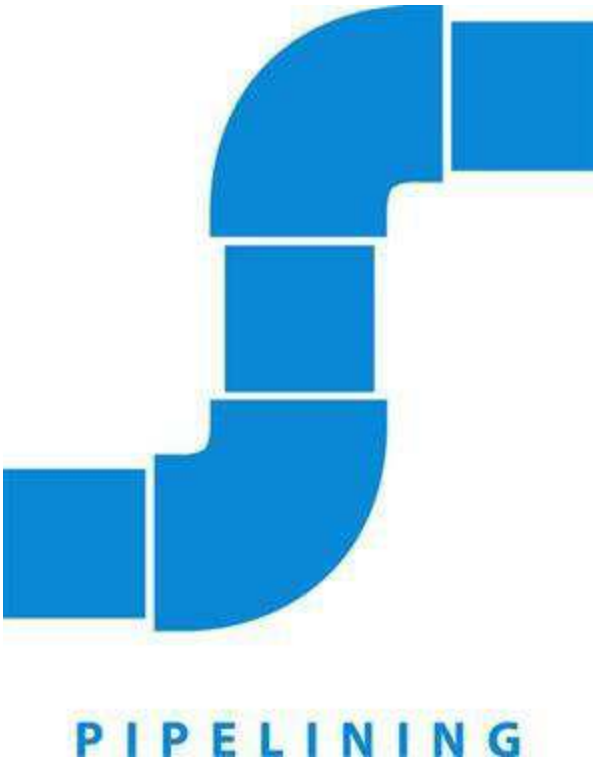


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 64-bit adder 63 times, this hardware "unrolls the loop" to use 63 adders and then organizes them to minimize delay.

In fact, multiply can go even faster than six add times because of the use of *carry save adders* (see Section A.6 in Appendix A), and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see Chapter 4).



Multiply in RISC-V

To produce a properly signed or unsigned 128-bit product, RISC-V has four instructions: *multiply* (`mul`), *multiply high* (`mulh`), *multiply high unsigned* (`mulhu`), and *multiply high signed-unsigned* (`mulhsu`). To get the integer 64-bit product, the programmer uses `mul`. To get the upper 64 bits of the 128-bit product, the programmer uses (`mulh`) if both operands are signed, (`mulhu`) if both operands are unsigned, or (`mulhsu`) if one operand is signed and the other is unsigned.

Summary

Multiplication hardware simply shifts and adds, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

Hardware/Software Interface

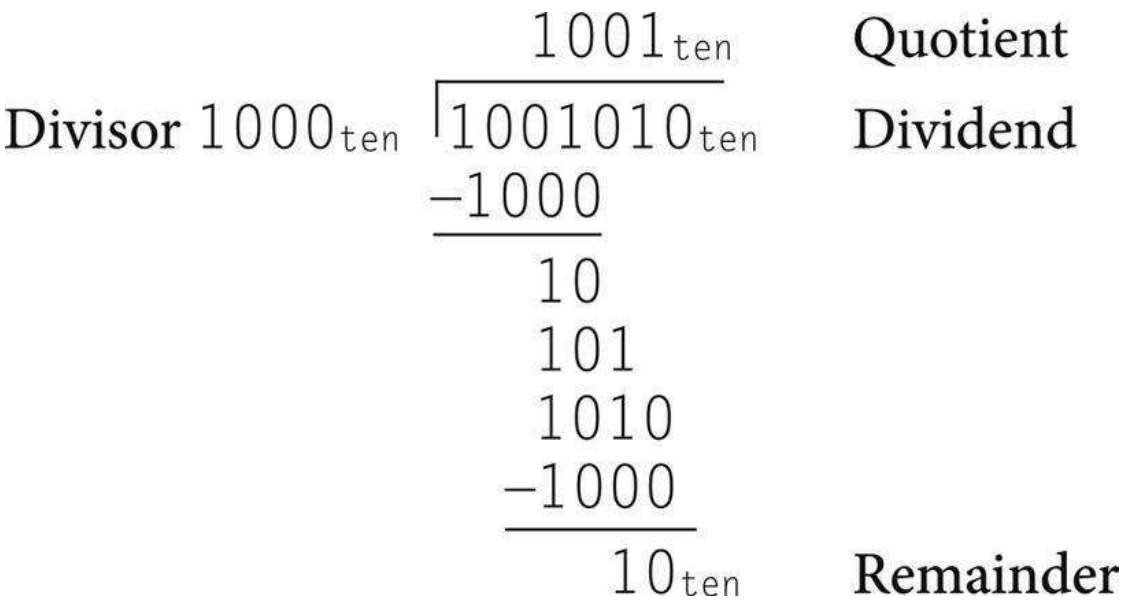
Software can use the multiply-high instructions to check for overflow from 64-bit multiplication. There is no overflow for 64-bit unsigned multiplication if `mulhu`'s result is zero. There is no overflow for 64-bit signed multiplication if all of the bits in `mulh`'s result are copies of the sign bit of `mul`'s result.

3.4 Division

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

> *Divide et impera.*
>
> *Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532*

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



Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

dividend

A number being divided.

divisor

A number that the dividend is divided by.

quotient

The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

remainder

The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

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

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder,

ignoring the quotient.

The basic division algorithm from grammar school 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 just 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 the divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 64-bit values, and we will ignore the sign for now.

A Division Algorithm and Hardware

Figure 3.8 shows hardware to mimic our grammar school algorithm. We start with the 64-bit Quotient register set to 0. Each iteration 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 128-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.

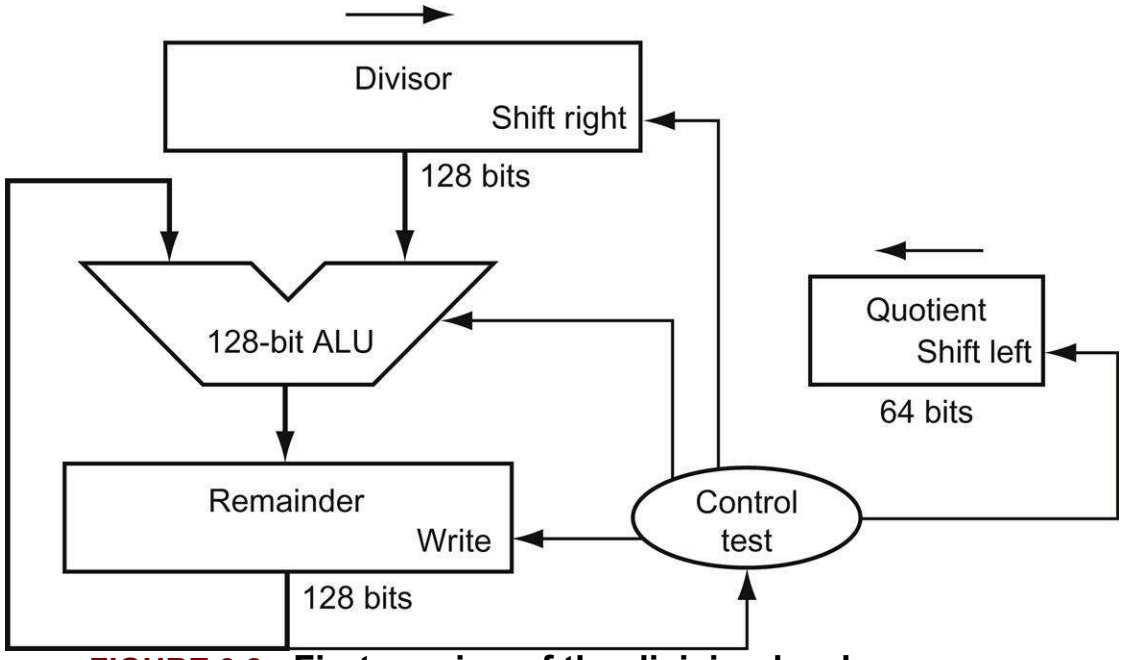


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are

all 128 bits wide, with only the Quotient register being 62 bits. The 64-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. 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 3.9 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 comparison. 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 complete.

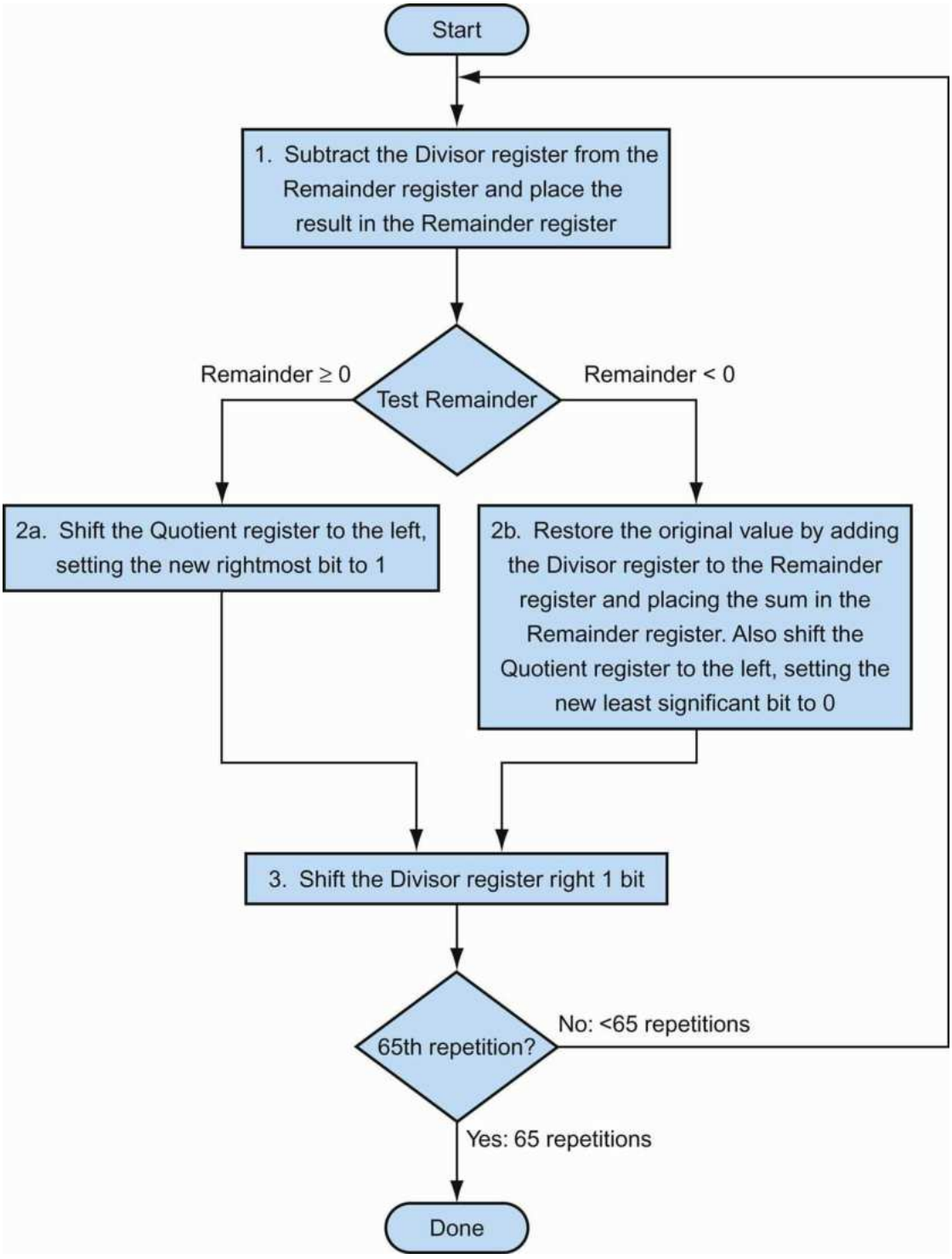


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8.

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 65 times.

A 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 3.10 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 checks 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	0000	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 3.10 Division example using the algorithm in Figure 3.9.
The bit examined to determine the next step is circled in color.

This algorithm and hardware can be refined to be faster and cheaper. The speed-up comes from shifting the operands and the quotient simultaneously with the subtraction. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.11 shows the revised hardware.

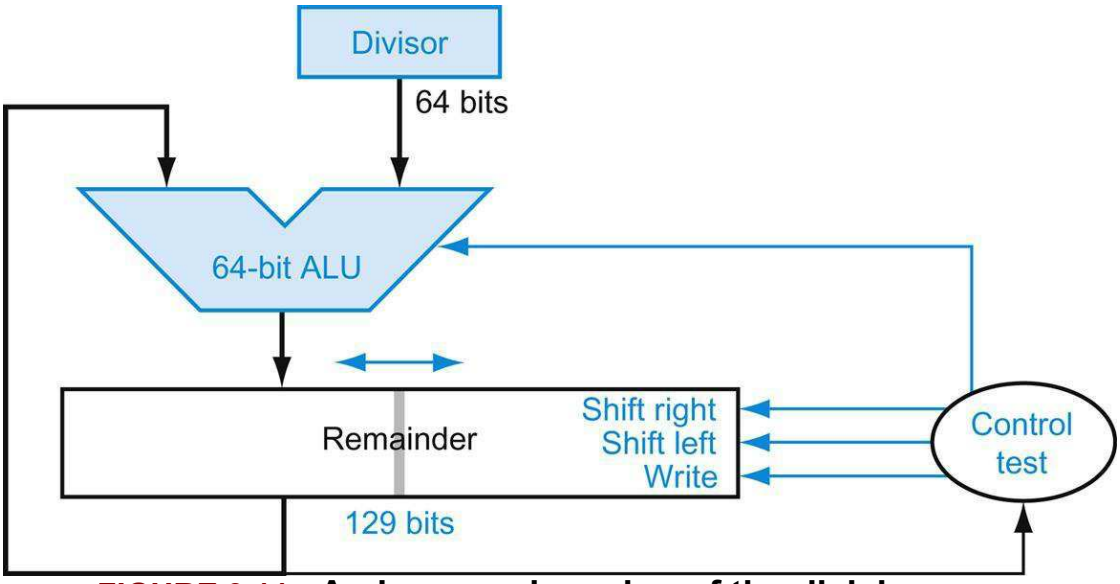


FIGURE 3.11 An improved version of the division hardware.
The Divisor register, ALU, and Quotient register are all 64 bits wide. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. As in Figure 3.5, the Remainder register has grown to 129 bits to make sure the carry out of the adder is not lost.

Signed Division

So far, we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration

The one complication of signed division is that we must also set the

sign of the remainder. Remember that the following equation must always hold:

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

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3, + \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2) \\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3,\ \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the

dividend and the divisor! Clearly, if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have identical signs, no matter what the signs of the divisor and quotient.

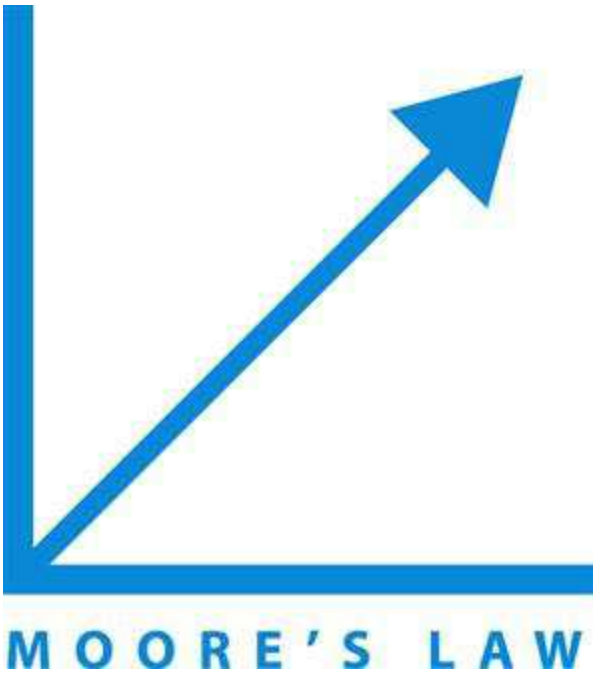
We calculate the other combinations by following the same rule:

```
+7 ÷ −2: Quotient = −3, Remainder = +1
−7 ÷ −2: Quotient = +3, Remainder = −1
```

Thus, the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division

Moore's Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 64 partial products immediately.



There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to **predict** several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The *Fallacy* on page 224 in Section 3.8 shows what can happen if the table is incorrect.

Divide in RISC-V

You may have already observed that the same sequential hardware can be used for both multiply and divide in Figures 3.5 and 3.11. The only requirement is a 128-bit register that can shift left or right and a 64-bit ALU that adds or subtracts.

To handle both signed integers and unsigned integers, RISC-V has two instructions for division and two instructions for remainder: *divide* (`div`), *divide unsigned* (`divu`), *remainder* (`rem`), and *remainder unsigned* (`remu`).

Summary

The common hardware support for multiply and divide allows RISC-V to provide a single pair of 64-bit registers that are used both for multiply and divide. We accelerate division by predicting

multiple quotient bits and then correcting mispredictions later. Figure 3.12 summarizes the enhancements to the RISC-V architecture for the last two sections.

Hardware/Software Interface

RISC-V divide instructions ignore overflow, so software must determine whether the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. RISC-V software must check the divisor to discover division by 0 as well as overflow.

Elaboration

An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step, since $(r+d)\times 2 - d = r - 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes one clock cycle per step, is explored further in the exercises; the algorithm above is called *restoring* division. A third algorithm that doesn't save the result of the subtract if it's negative is called a *nonperforming* division algorithm. It averages one-third fewer arithmetic operations.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
	Set if less than	slt x5, x6, x7	x5 = 1 if x5 < x6, else 0	Three register operands
	Set if less than, unsigned	sltu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Three register operands
	Set if less than, immediate	slti x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Set if less than immediate, uns.	sltiu x5, x6, x7	x5 = 1 if x5 < x6, else 0	Comparison with immediate
	Multiply	mul x5, x6, x7	x5 = x6 × x7	Lower 64 bits of 128-bit product
	Multiply high	mulh x5, x6, x7	x5 = (x6 × x7) >> 64	Upper 64 bits of 128-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	x5 = (x6 × x7) >> 64	Upper 64 bits of 128-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 64	Upper 64 bits of 128-bit signed-unsigned product
	Divide	div x5, x6, x7	x5 = x6 / x7	Divide signed 64-bit numbers
	Divide unsigned	divu x5, x6, x7	x5 = x6 / x7	Divide unsigned 64-bit numbers
	Remainder	rem x5, x6, x7	x5 = x6 % x7	Remainder of signed 64-bit division
	Remainder unsigned	remu x5, x6, x7	x5 = x6 % x7	Remainder of unsigned 64-bit division
Data transfer	Load doubleword	ld x5, 40(x6)	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte halfword from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	auipc x5, 0x12345	x5 = PC + 0x12345000	Used for PC-relative data addressing
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Uncondit-ional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

FIGURE 3.12 RISC-V core architecture.
RISC-V machine language is listed in the RISC-V Reference Data Card at the front of this book.

3.5 Floating Point

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

> *Speed gets you nowhere if you're headed the wrong way.*
>
> American proverb

3.14159265…$_{ten}$ (pi)
2.71828…$_{ten}$ *(e)*
0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)
$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

scientific notation
A notation that renders numbers with a single digit to the left of the decimal point.

normalized
A number in floating-point notation that has no leading 0s.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{two} \times 2^{-1}$$

To keep a binary number in the normalized form, we need a base that we can increase or decrease by exactly the number of bits the

number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called **floating point** because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

floating point

Computer arithmetic that represents numbers in which the binary point is not fixed.

A standard scientific notation for reals in the normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since real digits to the right of the binary point replace the unnecessary leading 0s.

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent**, because a fixed word size means you must take a bit from one to add a bit to the other. This tradeoff is between precision and range: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design

guideline from Chapter 2 reminds us, good design demands good compromise.

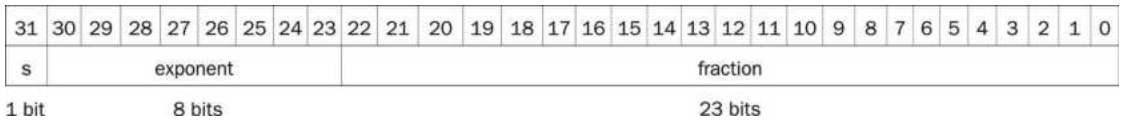
fraction

The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the *mantissa*.

exponent

In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Floating-point numbers are usually a multiple of the size of a word. The representation of a RISC-V floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the exponent), and *fraction* is the 23-bit number. As we recall from Chapter 2, this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the number.



In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that RISC-V does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give RISC-V computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus,

overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

overflow (floating-point)

A situation in which a positive exponent becomes too large to fit in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, we call this event **underflow**. This situation occurs when the negative exponent is too large to fit in the exponent field.

underflow (floating-point)

A situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce the chances of underflow or overflow is to offer another format that has a larger exponent. In C, this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

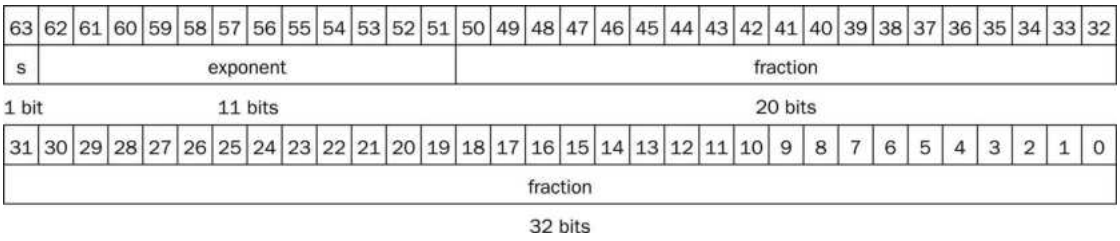
double precision

A floating-point value represented in a 64-bit doubleword.

single precision

A floating-point value represented in a 32-bit word.

The representation of a double precision floating-point number takes one RISC-V doubleword, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction field.



RISC-V double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

Exceptions and Interrupts

What should happen on an overflow or underflow to let the user know that a problem occurred? Some computers signal these events by raising an **exception**, sometimes called an **interrupt**. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 4.9 covers exceptions in more detail; Chapter 5 describes other situations where exceptions and interrupts occur.) RISC-V computers do *not* raise an exception on overflow or underflow; instead, software can read the *floating-point control and status register* (fcsr) to check whether overflow or underflow has occurred.

exception

Also called **interrupt**. An unscheduled event that disrupts program execution; used to detect overflow.

interrupt

An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

IEEE 754 Floating-Point Standard

These formats go beyond RISC-V. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the number, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or 53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus 00 … 00_{two} represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s1, s2, s3, …, then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

Figure 3.13 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing +∞ or −∞; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will output an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 IEEE 754 encoding of floating-point numbers.
A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 216. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when they are convenient.

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a quick test of less than, greater than, or equal to 0. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Placing the exponent before the significand also simplifies the sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{two} \times 2^{-1}$ would be represented in a single precision as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so an exponent of −1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

Let's demonstrate.

Example

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

Answer

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

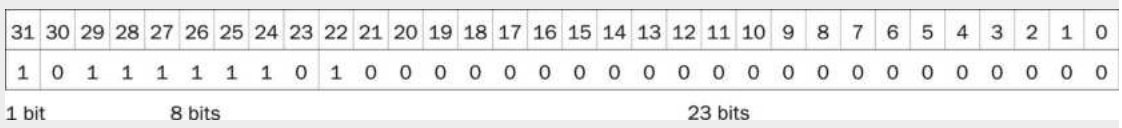
The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

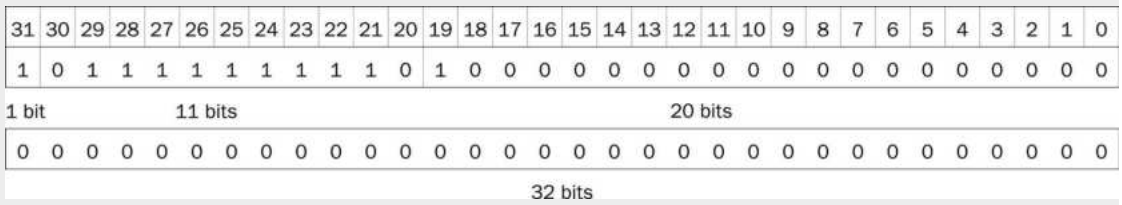
$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then



The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

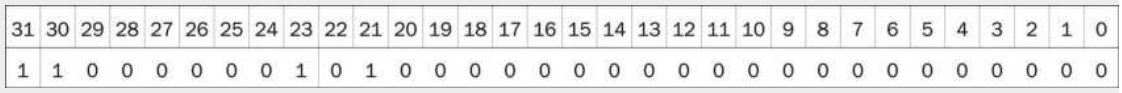


Now let's try going the other direction.

Converting Binary to Decimal Floating Point

Example

What decimal number does this single precision float represent?



Answer

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

In the next few subsections, we will give the algorithms for floating-point addition and multiplication. At their core, they use the corresponding integer operations on the significands, but extra bookkeeping is necessary to handle the exponents and normalize the result. We first give an intuitive derivation of the algorithms in decimal and then give a more detailed, binary version in the figures.

Elaboration

Following IEEE guidelines, the IEEE 754 committee was reformed 20 years after the standard to see what changes, if any, should be made. The revised standard IEEE 754-2008 includes nearly all the IEEE 754-1985 and adds a 16-bit format ("half precision") and a 128-bit format ("quadruple precision"). The revised standard also adds decimal floating point arithmetic.

Elaboration

In an attempt to increase range without removing bits from the significand, some computers before the IEEE 754 standard used a base other than 2. For example, the IBM 360 and 370 mainframe computers use base 16. Since changing the IBM exponent by one means shifting the significand by 4 bits, "normalized" base 16 numbers can have up to 3 leading bits of 0s! Hence, hexadecimal digits mean that up to 3 bits must be dropped from the significand, which leads to surprising problems in the accuracy of floating-point arithmetic. IBM mainframes now support IEEE 754 as well as the old hex format.

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$.

Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus, the first step shifts the

significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$0.016 \times 10^1$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^1$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \times 10^1 = 1.0015_{ten} \times 10^2$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand could be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{ten} \times 10^2$$

is rounded to four digits in the significand to

$$1.002_{ten} \times 10^{2}$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Figure 3.14 shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the *Elaboration* on page 216). For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is −126.

Binary Floating-Point Addition

Example

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.14.

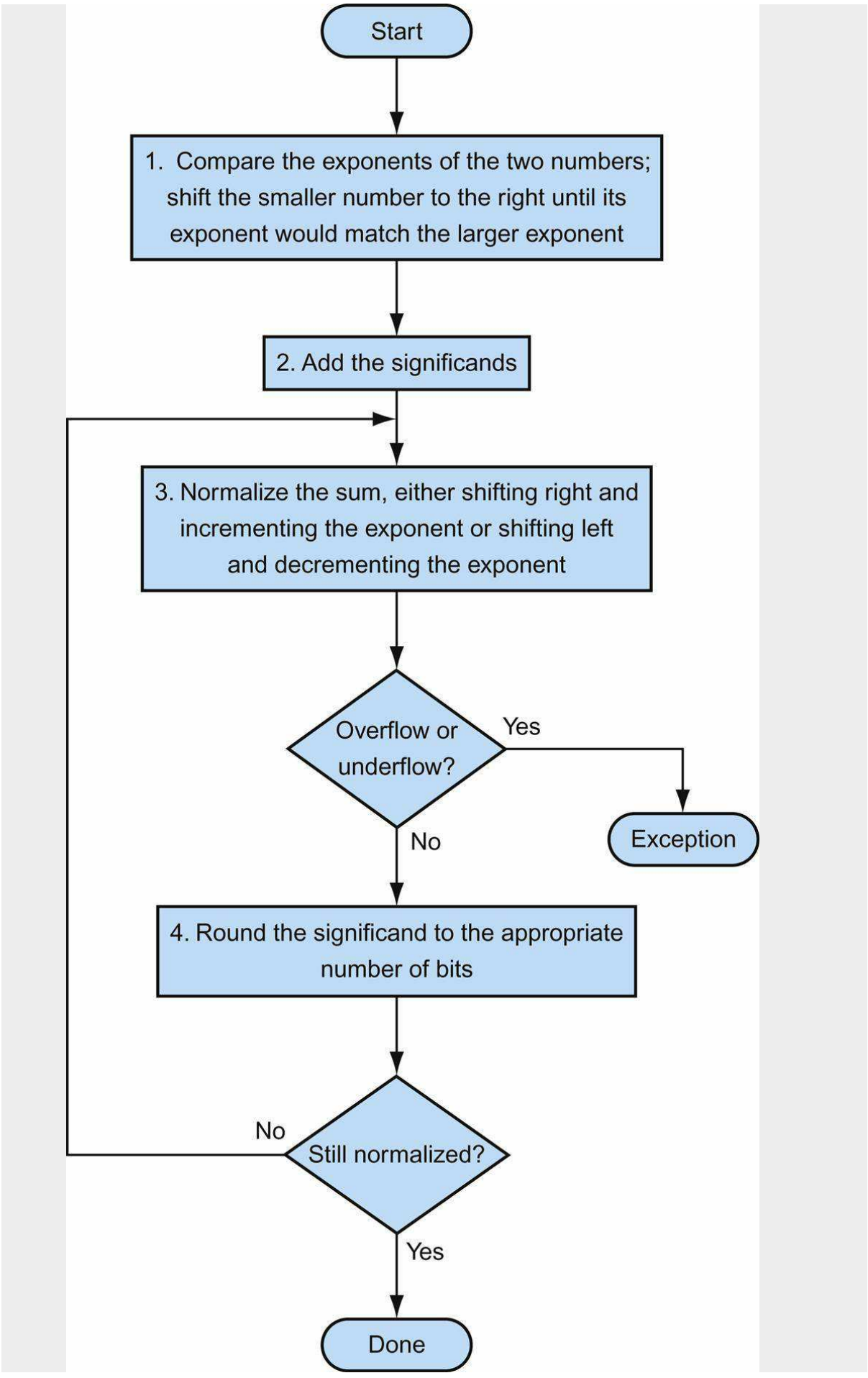

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

FIGURE 3.14 Floating-point addition.
The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Answer

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{array}{lll} 0.5_{ten} & = 1/2_{ten} & = 1/2^{1}_{ten} \\ & = 0.1_{two} & = 0.1_{two} \times 2^{0} \qquad = 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} & = -7/16_{ten} & = -7/2^{4}_{ten} \\ & = -0.0111_{two} & = -0.0111_{two} \times 2^{0} \qquad = -1.110_{two} \times 2^{-2} \end{array}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{array}{l} 0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} \\ \qquad\qquad = 1.000_{two} \times 2^{-4} \end{array}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be −4 +127, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.
This sum is then

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two}$$
$$= 1/2^{4}_{ten} = 1/16_{ten} = 0.0625_{ten}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure 3.15 sketches the basic organization of hardware for floating-point addition.

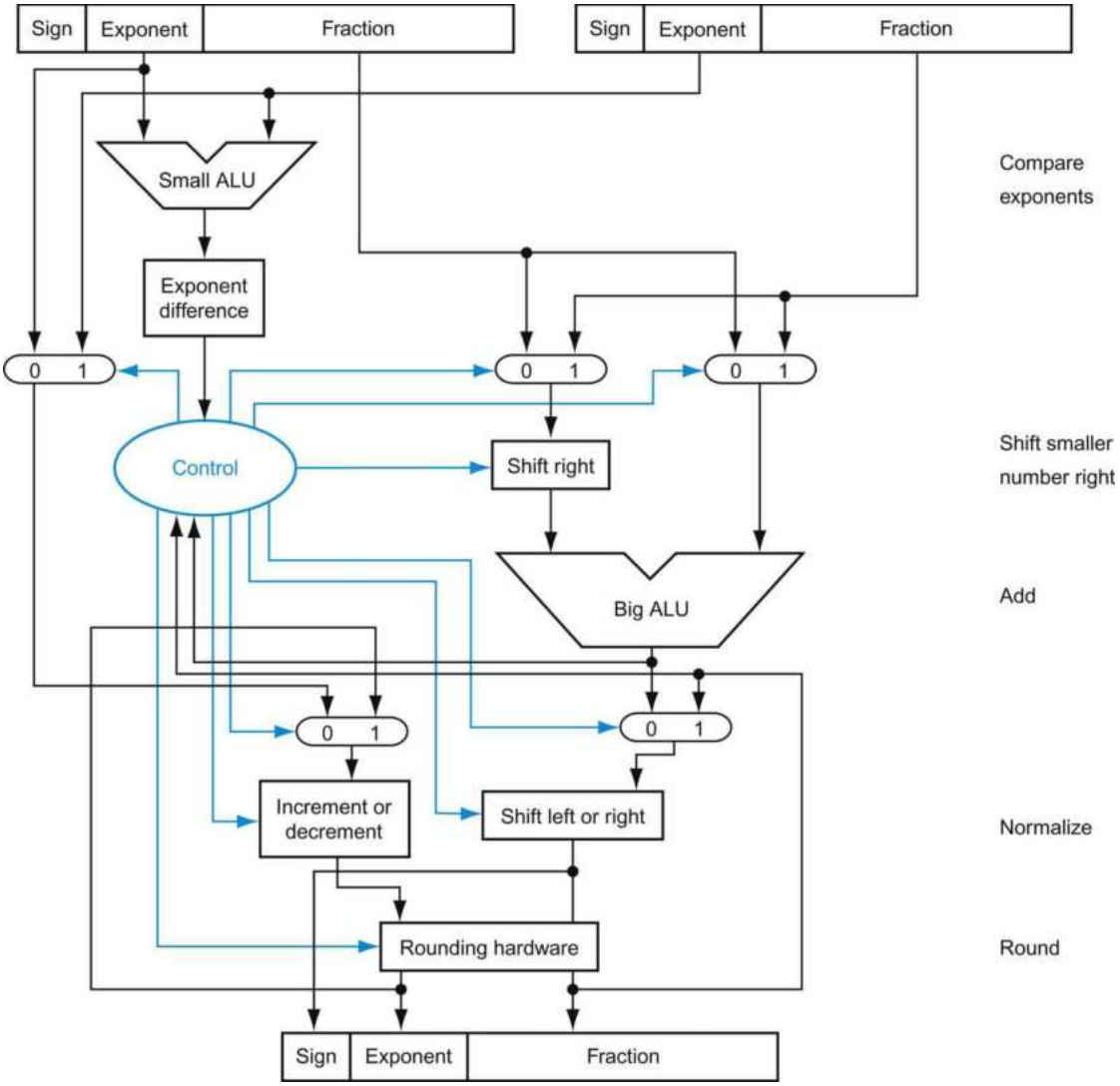


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition.

The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: 10 +127 =137, and −5 +127 =122, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$
\begin{array}{r}
1.110_{ten} \\
\times \quad 9.200_{ten} \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
1110000_{ten}
\end{array}
$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{ten}$$

If we can keep only three digits to the right of the decimal point, the product is 10.212×10^5.

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{ten} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{ten} \times 10^{6}$$

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$$+1.021_{ten} \times 10^{6}$$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication, the signs of the operands determine the sign of the product.

Once again, as Figure 3.16 shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

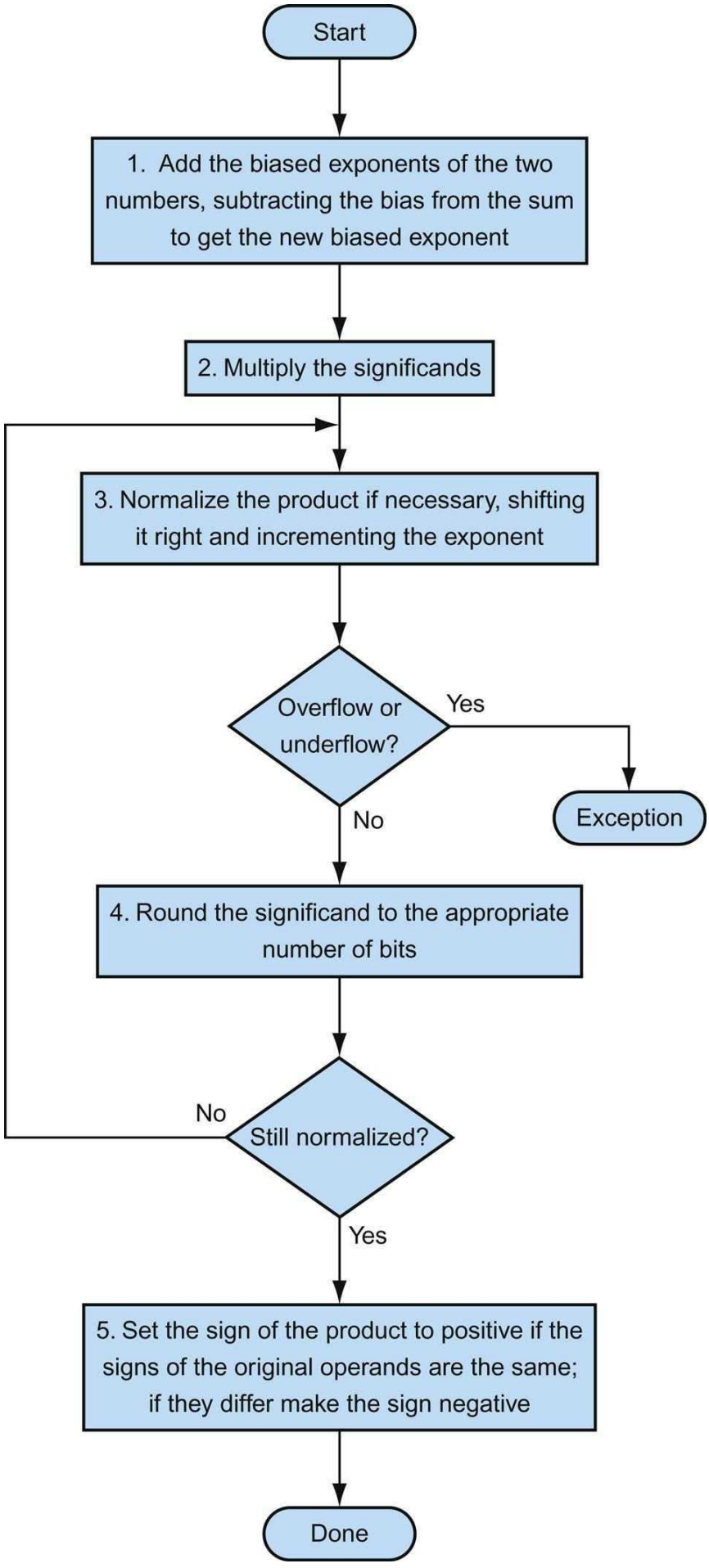

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

FIGURE 3.16 Floating-point multiplication.
The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Binary Floating-Point Multiplication

Example

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.16.

Answer

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.
Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$
\begin{array}{r}
1.000_{\text{two}} \\
\times \quad 1.110_{\text{two}} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{\text{two}}
\end{array}
$$

The product is $1.110000_{\text{two}} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{\text{two}} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{\text{two}} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$-1.110_{two} \times 2^{-3} = -0.001110_{two} = -0.00111_{two}$$
$$= -7/2^{5}_{ten} = -7/32_{ten} = -0.21875_{ten}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

Floating-Point Instructions in RISC-V

RISC-V supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point *addition, single* (`fadd.s`) and *addition, double* (`fadd.d`)
- Floating-point *subtraction, single* (`fsub.s`) and *subtraction, double* (`fsub.d`)
- Floating-point *multiplication, single* (`fmul.s`) and *multiplication, double* (`fmul.d`)
- Floating-point *division, single* (`fdiv.s`) and *division, double* (`fdiv.d`)
- Floating-point square root, single (`fsqrt.s`) and square root, double (`fsqrt.d`)
- Floating-point equals, single (`feq.s`) and equals, double (`feq.d`)
- Floating-point less-than, single (`flt.s`) and less-than, double (`flt.d`)
- Floating-point less-than-or-equals, single (`fle.s`) and less-than-or-equals, double (`fle.d`)

The comparison instructions, `feq`, `flt`, and `fle`, set an integer register to 0 if the comparison is false and 1 if it is true. Software can thus branch on the result of a floating-point comparison using the integer branch instructions `beq` and `bne`.

The RISC-V designers decided to add separate floating-point registers. They are called `f0`, `f1`, …, `f31`. Hence, they included separate loads and stores for floating-point registers: `fld` and `fsd` for double-precision and `flw` and `fsw` for single-precision. The base registers for floating-point data transfers which are used for addresses remain integer registers. The RISC-V code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
flw f0, 0(x10) // Load 32-bit F.P. number into f0
flw f1, 4(x10) // Load 32-bit F.P. number into f1
fadd.s f2, f0, f1 // f2 = f0 + f1, single precision
fsw f2, 8(x10) // Store 32-bit F.P. number from f2
```

A single precision register is just the lower half of a double-precision register. Note that, unlike integer register `x0`, floating-point register `f0` is *not* hard-wired to the constant `0`.

Figure 3.17 summarizes the floating-point portion of the RISC-V architecture revealed in this chapter, with the new pieces to support floating point shown in color. The floating-point instructions use the same format as their integer counterparts: loads use the I-type format, stores use the S-type format, and arithmetic instructions use the R-type format.

RISC-V floating-point operands

Name	Example	Comments
32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{61} memory double words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	fadd.s f0, f1, f2	f0 = f1 + f2	FP add (single precision)
	FP subtract single	fsub.s f0, f1, f2	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	fmul.s f0, f1, f2	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	fdiv.s f0, f1, f2	f0 = f1 / f2	FP divide (single precision)
	FP square root single	fsqrt.s f0, f1	f0 = √f1	FP square root (single precision)
	FP add double	fadd.d f0, f1, f2	f0 = f1 + f2	FP add (double precision)
	FP subtract double	fsub.d f0, f1, f2	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	fmul.d f0, f1, f2	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	fdiv.d f0, f1, f2	f0 = f1 / f2	FP divide (double precision)
	FP square root double	fsqrt.d f0, f1	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	flw f0, 4(x5)	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	fld f0, 8(x5)	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	fsd f0, 8(x5)	Memory[x5 + 8] = f0	Store double-precision from memory

FIGURE 3.17 RISC-V floating-point architecture revealed thus far.
This information is also found in column 2 of the RISC-V Reference Data Card at the front of this book.

Hardware/Software Interface

One issue that architects face in supporting floating-point arithmetic is whether to select the same registers used by the

integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a distinct set of data transfer instructions to move data between floating-point registers and memory.

The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point; for example, some computers convert all sized operands in registers into a single internal format.

Compiling a Floating-Point C Program into RISC-V Assembly Code

Example

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
 {
  return ((5.0f/9.0f) *(fahr - 32.0f));
 }
```

Assume that the floating-point argument `fahr` is passed in `f10` and the result should also go in `f10`. What is the RISC-V assembly code?

Answer

We assume that the compiler places the three floating-point constants in memory within easy reach of register `x3`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
 flw f0, const5(x3) // f0 = 5.0f
 flw f1, const9(x3) // f1 = 9.0f
```

They are then divided to get the fraction 5.0/9.0:

```
fdiv.s f0, f0, f1 // f0 = 5.0f / 9.0f
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the

divide at runtime.) Next, we load the constant 32.0 and then subtract it from `fahr` (`f10`):

```
flw f1, const32(x3) // f1 = 32.0f
fsub.s f10, f10, f1 // f10 = fahr - 32.0f
```

Finally, we multiply the two intermediate results, placing the product in `f10` as the return result, and then return

```
fmul.s f10, f0, f10 // f10 = (5.0f / 9.0f)*(fahr - 32.0f)
jalr x0, 0(x1) // return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into RISC-V

Example

Most floating-point calculations are performed in double precision. Let's per-form matrix multiply of C =C +A * B. It is commonly called *DGEMM*, for Double precision, General Matrix Multiply. We'll see versions of DGEMM again in Section 3.8 and subsequently in Chapters 4, 5, and 6. Let's assume C, A, and B are all square matrices with 32 elements in each dimension.

```
void mm (double c[][], double a[][], double b[][])
{
 size_t i, j, k;
 for (i = 0; i < 32; i = i + 1)
 for (j = 0; j < 32; j = j + 1)
 for (k = 0; k < 32; k = k + 1)
  c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

The array starting addresses are parameters, so they are in `x10`, `x11`, and `x12`. Assume that the integer variables are in `x5`, `x6`, and `x7`, respectively. What is the RISC-V assembly code for the body of the procedure?

Answer

Note that `c[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `c[i][j]`, so we can avoid loading and storing `c[i][j]` each iteration. Instead, the compiler loads `c[i][j]` into a register outside the loop, accumulates the sum

of the products of `a[i][k]` and `b[k][j]` in that same register, and then stores the sum into `c[i][j]` upon termination of the innermost loop. We keep the code simpler by using the assembly language pseudoinstruction `li`, which loads a constant into a register.

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
 mm:...
  li x28, 32 // x28 = 32 (row size/loop end)
  li x5, 0 // i = 0; initialize 1st for loop
L1: li x6, 0 // j = 0; initialize 2nd for loop
L2: li x7, 0 // k = 0; initialize 3rd for loop
```

To calculate the address of `c[i][j]`, we need to know how a 32 ×32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimensional arrays, each with 32 elements. So the first step is to skip over the `i` "single-dimensional arrays," or rows, to get the one we want. Thus, we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
slli x30, x5, 5 // x30 = i * 2^5(size of row of c)
```

Now we add the second index to select the `j`th element of the desired row:

```
add x30, x30, x6 // x30 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by three:

```
slli x30, x30, 3 // x30 = byte offset of [i][j]
```

Next we add this sum to the base address of `c`, giving the address of `c[i][j]`, and then load the double precision number `c[i][j]` into `f0`:

```
add x30, x10, x30 // x30 = byte address of c[i][j]
fld f0, 0(x30) // f0 = 8 bytes of c[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number `b[k][j]`.

```
L3: slli x29, x7, 5 // x29 = k * 2^5(size of row of b)
 add x29, x29, x6 // x29 = k * size(row) + j
 slli x29, x29, 3 // x29 = byte offset of [k][j]
```

```
add x29, x12, x29 // x29 = byte address of b[k][j]
fld f1, 0(x29) // f1 = 8 bytes of b[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number `a[i][k]`.

```
slli x29, x5, 5 // x29 = i * 2^5(size of row of a)
add x29, x29, x7 // x29 = i * size(row) + k
slli x29, x29, 3 // x29 = byte offset of [i][k]
add x29, x11, x29 // x29 = byte address of a[i][k]
fld f2, 0(x29) // f2 = a[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `a` and `b` located in registers `f2` and `f1`, and then accumulate the sum in `f0`.

```
fmul.d f1, f2, f1 // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1 // f0 = c[i][j] + a[i][k] * b[k][j]
```

The final block increments the index `k` and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in `f0` into `c[i][j]`.

```
addi x7, x7, 1 // k = k + 1
bltu x7, x28, L3 // if (k < 32) go to L3
fsd f0, 0(x30) // c[i][j] = f0
```

Similarly, these final six instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
addi x6, x6, 1 // j = j + 1
bltu x6, x28, L2 // if (j < 32) go to L2
addi x5, x5, 1 // i = i + 1
bltu x5, x28, L1 // if (i < 32) go to L1
. . .
```

Looking ahead, Figure 3.20 below shows the x86 assembly language code for a slightly different version of DGEMM in Figure 3.19.

Elaboration

C and many other programming languages use the array layout discussed in the example, called *row-major order*. Fortran instead uses *column-major order*, whereby the array is stored column by column.

Elaboration

Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence, the floating-point unit, including the floating-point registers, was optionally available as a second chip. Such optional accelerator chips are called *coprocessor chips*. Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and thus the term *coprocessor chip* joins *accumulator* and *core memory* as quaint terms that date the speaker.

Elaboration

As mentioned in Section 3.4, accelerating division is more challenging than multiplication. In addition to SRT, another technique to leverage a fast multiplier is *Newton's iteration*, where division is recast as finding the zero of a function to produce the reciprocal 1/*c*, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solved this problem by calculating an extra-precise reciprocal.

Elaboration

Java embraces IEEE 754 by name in its definition of Java floating-point data types and operations. Thus, the code in the first example could have well been generated for a class method that converted Fahrenheit to Celsius.

The second example above uses multiple dimensional arrays, which are not explicitly supported in Java. Java allows arrays of arrays, but each array may have its own length, unlike multiple dimensional arrays in C. Like the examples in Chapter 2, a Java version of this second example would require a good deal of checking code for array bounds, including a new length calculation at the end of row accesses. It would also need to check that the object reference is not null.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are

normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 1 and 2, but no more than 2^{53} can be represented exactly in double precision floating point. The best we can do is getting the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly, if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps two extra bits on the right during intervening additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate their value.

guard

The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

round

Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format. It is also the name of the second of two extra bits kept on the right during intermediate floating-point calculations, which improves rounding accuracy.

Rounding with Guard Digits

Example

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

Answer

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} 2.3400_{ten} \\ +0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Thus the sum is $2.3656_{ten} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{ten} \\ +0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

The answer is $2.36_{ten} \times 10^2$, off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number were off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there are no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

units in the last place (ulp)

The number of bits in error in the least significant bits of the

significand between the actual number and the number that can be represented.

Elaboration

Although the example above really needed just one extra digit, multiply can require two. A binary product may have one leading 0 bit; hence, the normalizing step must shift the product one bit left. This shifts the guard digit into the least significant bit of the product, leaving the round bit to help accurately round the product.

IEEE 754 has four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The U.S. *Internal Revenue Service* (IRS) always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

The goal of the extra rounding bits is to allow the computer to get the same results as if the intermediate results were calculated to infinite precision and then rounded. To support this goal and round to the nearest even, the standard has a third bit in addition to guard and round; it is set whenever there are nonzero bits to the right of the round bit. This **sticky bit** allows the computer to see the difference between $0.50 \ldots 00_{ten}$ and $0.50 \ldots 01_{ten}$ when rounding.

sticky bit

A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \times$

10^{-1} to $2.34_{ten} \times 10^{2}$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set, since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000 … 00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000 … 00, we round instead to 2.35.

Elaboration

RISC-V, MIPS-64, PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures all provide a single instruction that does a multiply and add on three registers: $a=a+(b\times c)$. Obviously, this instruction allows potentially higher floating-point performance for this common operation. Equally important is that instead of performing two roundings—after the multiply and then after the add—which would happen with separate instructions, the multiply add instruction can perform a single rounding after the add. A single rounding step increases the precision of multiply add. Such operations with a single rounding are called **fused multiply add**. It was added to the revised IEEE 754-2008 standard (see  **Section 3.11**).

fused multiply add

A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

Summary

The *Big Picture* that follows reinforces the stored-program concept from Chapter 2; the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

$$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

is almost always an approximation of the real number. Computer systems must take care to minimize this gap between computer arithmetic and arithmetic in the real world, and programmers at times need to be aware of the implications of this approximation.

The BIG Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, character strings, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and hence limited precision; it's possible to calculate a number too big or too small to be represented in a computer word. Programmers must remember these limits and write programs accordingly.

C type	Java type	Data transfers	Operations
long long int	long	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned long long int	–	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	–	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

Hardware/ Software Interface

In the last chapter, we presented the storage classes of the programming language C (see the *Hardware/Software Interface* section in Section 2.7). The table above shows some of the C and Java data types, the data transfer instructions, and instructions that operate on those types that appear in Chapter 2 and this chapter. Note that Java omits unsigned integers.

Check Yourself

The revised IEEE 754-2008 standard added a 16-bit floating-point format with five exponent bits. What do you think is the likely range of numbers it could represent?

1. $1.0000\ 00 \times 2^{0}$ to $1.1111\ 1111\ 11 \times 2^{31}$, 0
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}$, ± 0, $\pm \infty$, NaN

3. ±1.0000 0000 00 ×2^{-14} to ±1.1111 1111 11 ×2^{15}, ±0, ±∞, NaN
4. ±1.0000 0000 00 ×2^{-15} to ±1.1111 1111 11 ×2^{14}, ±0, ±∞, NaN

Elaboration

To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. RISC-V does not provide instructions for unordered comparisons, but a careful sequence of ordered comparisons has the same effect. (Java does not support unordered compares.)

In an attempt to squeeze every bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero fraction. They allow a number to degrade in significance until it becomes 0, called *gradual underflow*. For example, the smallest positive single precision normalized number is

$$1.0000\,0000\,0000\,0000\,0000\,000_{two} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\,0000\,0000\,0000\,0000\,001_{two} \times 2^{-126}, \text{ or } 1.0_{two} \times 2^{-149}$$

For double precision, the denorm gap goes from 1.0 ×2^{-1022} to 1.0 ×2^{-1074}.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Moreover, if programmers do not expect denorms, their programs may surprise them.

3.6 Parallelism and Computer Arithmetic: Subword Parallelism

Since every microprocessor in a phone, tablet, or laptop by definition has its own graphical display, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of these data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.

The cost of such partitioned adders was small yet the speedups could be large.

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They are known as well as vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily compute in parallel. As of this writing, RISC-V does not have additional instructions to exploit subword parallelism, but the next section presents a real-world example of such an architecture.

3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) for the x86 included instructions that operate on short vectors of integers. Later, SSE (*Streaming SIMD Extension*) provided instructions that operate on short vectors of single-precision floating-point numbers. Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It included eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.18 summarizes the SSE and SSE2 instructions.

Data transfer	Arithmetic	Compare
MOV[AU]{SS|PS|SD|PD} xmm, {mem|xmm}	ADD{SS|PS|SD|PD} xmm,{mem|xmm}	CMP{SS|PS|SD|PD}
	SUB{SS|PS|SD|PD} xmm,{mem|xmm}	
MOV[HL]{PS|PD} xmm, {mem|xmm}	MUL{SS|PS|SD|PD} xmm,{mem|xmm}	
	DIV{SS|PS|SD|PD} xmm,{mem|xmm}	
	SQRT{SS|PS|SD|PD} {mem|xmm}	
	MAX{SS|PS|SD|PD} {mem|xmm}	
	MIN{SS|PS|SD|PD} {mem|xmm}	

FIGURE 3.18 The SSE/SSE2 floating-point instructions of the x86.
xmm means one operand is a 128-bit SSE2 register, and {mem|xmm} means the other operand is either in memory or it is an SSE2 register. The table uses regular expressions to show the variations of instructions. Thus, `MOV[AU]{SS|PS|SD|PD}` represents the eight instructions `MOVASS`,`MOVAPS`,`MOVASD`,`MOVAPD`,`MOVUSS`,`MOVUPS`,`MOVUSD`, and `MOVUPD`. We use square brackets `[]` to show single-letter alternatives: `A` means the 128-bit operand is aligned in memory; `U` means the 128-bit operand is unaligned in memory; `H` means move the high half of the 128-bit operand; and `L` means move the low half of the 128-bit operand. We use the curly brackets `{}` with a vertical bar `|` to show multiple letter variations of the basic operations: `SS` stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; `PS` stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; `SD` stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; `PD` stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can

compute simultaneously on four singles (PS) or two doubles (PD).

In 2011, Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions* (AVX). Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter "v" (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point additions

```
addpd %xmm0, %xmm4
```

becomes

```
vaddpd %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies. Intel has announced plans to widen the AVX registers to first 512 bits and later 1024 bits in later editions of the x86 architecture.

Elaboration

AVX also added three address instructions to x86. For example, `vaddpd` can now specify

```
vaddpd %ymm0, %ymm1, %ymm4 // %ymm4 = %ymm0 + %ymm1
```

instead of the standard, two address version

```
addpd %xmm0, %xmm4 // %xmm4 = %xmm4 + %xmm0
```

(Unlike RISC-V, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

3.8 Going Faster: Subword Parallelism and Matrix Multiply

To demonstrate the performance impact of subword parallelism, we'll run the same code on the Intel Core i7 first without AVX and then with it. Figure 3.19 shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in Section 3.5, this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled "Going Faster" to demonstrate

the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in Chapters 3, 4, 5, and 6 will incrementally improve DGEMM performance using the ideas that each chapter introduces.

```
1.  void dgemm (size_t n, double* A, double* B, double* C)
2.  {
3.    for (size_t i = 0; i < n; ++i)
4.      for (size_t j = 0; j < n; ++j)
5.      {
6.        double cij = C[i+j*n]; /* cij = C[i][j] */
7.        for(size_t k = 0; k < n; k++ )
8.          cij += A[i+k*n] * B[k+j*n]; /*cij+=A[i][k]*B[k][j]*/
9.        C[i+j*n] = cij; /* C[i][j] = cij */
10.     }
11. }
```

FIGURE 3.19 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply. Because we are passing the matrix dimension as the parameter `n`, this version of DGEMM uses single-dimensional versions of matrices `C`, `A`, and `B` and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

Figure 3.20 shows the x86 assembly language output for the inner loop of Figure 3.19. The five floating point-instructions start with a `v` like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include `sd` in the name, which stands for scalar double precision. We'll define the subword parallel instructions shortly.

```
1.  vmovsd (%r10),%xmm0              // Load 1 element of C into %xmm0
2.  mov    %rsi,%rcx                 // register %rcx = %rsi
3.  xor    %eax,%eax                 // register %eax = 0
4.  vmovsd (%rcx),%xmm1              // Load 1 element of B into %xmm1
5.  add    %r9,%rcx                  // register %rcx = %rcx + %r9
6.  vmulsd (%r8,%rax,8),%xmm1,%xmm1  // Multiply %xmm1,element of A
7.  add    $0x1,%rax                 // register %rax = %rax + 1
8.  cmp    %eax,%edi                 // compare %eax to %edi
9.  vaddsd %xmm1,%xmm0,%xmm0         // Add %xmm1, %xmm0
10. jg     30 <dgemm+0x30>           // jump if %eax > %edi
11. add    $0x1,%r11                 // register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)              // Store %xmm0 into C element
```

FIGURE 3.20 The x86 assembly language for the body of the nested loops generated by compiling the unoptimized C code in Figure 3.19. Although it is dealing with just 64 bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the *Elaboration* in Section 3.7).

While compiler writers may eventually be able to produce high-quality code routinely that uses the AVX instructions of the x86, for now we must "cheat" by using C intrinsics that more or less tell the compiler exactly how to produce good code. Figure 3.21 shows the enhanced version of Figure 3.19 for which the Gnu C compiler produces AVX code. Figure 3.22 shows annotated x86 code that is the output of compiling using gcc with the –O3 level of optimization.

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t k = 0; k < n; k++ )
8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                  _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                 _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }
```

FIGURE 3.21 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86.

Figure 3.22 shows the assembly language produced by the compiler for the inner loop.

```
1.  vmovapd (%r11),%ymm0             // Load 4 elements of C into %ymm0
2.  mov     %rbx,%rcx                // register %rcx = %rbx
3.  xor     %eax,%eax                // register %eax = 0
4.  vbroadcastsd (%rax,%r8,1),%ymm1  // Make 4 copies of B element
5.  add     $0x8,%rax                // register %rax = %rax + 8
6.  vmulpd (%rcx),%ymm1,%ymm1        // Parallel mul %ymm1,4 A elements
7.  add     %r9,%rcx                 // register %rcx = %rcx + %r9
8.  cmp     %r10,%rax                // compare %r10 to %rax
9.  vaddpd %ymm1,%ymm0,%ymm0         // Parallel add %ymm1, %ymm0
10. jne     50 <dgemm+0x50>          // jump if not %r10 != %rax
11. add     $0x1,%esi                // register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)             // Store %ymm0 into 4 C elements
```

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21.

Note the similarities to Figure 3.20, with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for packed double precision

instead of the sd version for scalar double precision.

The declaration on line 6 of Figure 3.21 uses the `__m256d` data type, which tells the compiler the variable will hold four double-precision floating-point values. The intrinsic `_mm256_load_pd()` also on line 6 uses AVX instructions to load four double-precision floating-point numbers in parallel (`_pd`) from the matrix `C` into `c0`. The address calculation `C+i+j*n` on line 6 represents element `C[i+j*n]`. Symmetrically, the final step on line 11 uses the intrinsic `_mm256_store_pd()` to store four double-precision floating-point numbers from `c0` into the matrix `C`. As we're going through four elements each iteration, the outer *for* loop on line 4 increments `i` by 4 instead of by 1 as on line 3 of Figure 3.19.

Inside the loops, on line 9 we first load four elements of A again using `_mm256_load_pd()`. To multiply these elements by one element of B, on line 10 we first use the intrinsic `_mm256_broadcast_sd()`, which makes four identical copies of the scalar double precision number—in this case an element of `B`—in one of the YMM registers. We then use `_mm256_mul_pd()` on line 9 to multiply the four double-precision results in parallel. Finally, `_mm256_add_pd()` on line 8 adds the four products to the four sums in `c0`.

Figure 3.22 shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with `v` and four of the five use `pd` for packed double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in Figure 3.20 above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (`sd`) using XMM registers to *packed double* (`pd`) with YMM registers. The one exception is line 4 of Figure 3.22. Every element of A must be multiplied by one element of B. One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register, which is just what the instruction `vbroadcastsd` does.

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in Figure 3.19 runs at 1.7 GigaFLOPS (FLoating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in Figure 3.21 performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of

4.0 increase that you might hope for from performing four times as many operations at a time by using **subword parallelism**.

Elaboration

As mentioned in the *Elaboration* in Section 1.6, Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of 3.3/2.6 =1.27 to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.

3.9 Fallacies and Pitfalls

Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

> *Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.*

Recall that a binary number x, where xi means the ith bit, represents the number

$$\ldots + (x^3 \times 2^3) + (x^2 \times 2^2) + (x^1 \times 2^1) + (x^0 \times 2^0)$$

Shifting the bits of c right by n bits would seem to be the same as dividing by $2n$. And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}. The two's complement representation of -5_{ten} is

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111011_{two}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$00111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $4{,}611{,}686{,}018{,}427{,}387{,}902_{ten}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two}$$

The result is -2_{ten} instead of -1_{ten}; close, but no cigar.

Pitfall: Floating-point addition is not associative.

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number. For example, let's see if $c+(a+b)=(c+a)+b$. Assume $c=-1.5_{ten} \times 10^{38}$, $a = 1.5_{ten} \times 10^{38}$, and $b = 1.0$, and that these are all single precision numbers.

$$\begin{aligned} c + (a + b) &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38} + 1.0) \\ &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38}) \\ &= 0.0 \\ c + (a + b) &= (-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}) + 1.0 \\ &= (0.0_{ten}) + 1.0 \\ &= 1.0 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{ten} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{ten} \times 10^{38} + 1.0$ is still $1.5_{ten} \times 10^{38}$. That is why the sum of c, a, and b is 0.0 or 1.0, depending on the order of the floating-point additions, so $c+(a+b) \neq (c+a)+b$. Therefore, floating-point addition is *not* associative.

Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, "Do the two versions get the same answer?" If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you

were to add a million numbers together, you would get the same results whether you used one processor or 1000 processors. This assumption holds for two's complement integers, since integer addition is associative. Alas, since floating-point addition is not associative, the assumption does not hold.

A more vexing version of this fallacy occurs on a parallel computer where the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer. As the varying number of processors from each run would cause the floating-point sums to be calculated in different orders, getting slightly different answers each time despite running identical code with identical input may flummox unaware parallel programmers.

Given this quandary, programmers who write parallel code with floating-point numbers need to verify whether the results are credible, even if they don't give the exact same answer as the sequential code. The field that deals with such issues is called numerical analysis, which is the subject of textbooks in its own right. Such concerns are one reason for the popularity of numerical libraries such as LAPACK and ScaLAPAK, which have been validated in both their sequential and parallel forms.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.23). The following is the inside story behind the headlines.

FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times, San Jose Mercury News, San Francisco Chronicle, and Infoworld*.
The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing −2, −1, 0, +1, or +2. The guess is multiplied by the divisor and subtracted from the remainder to generate a new remainder. Like nonrestoring division, if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently, there were five elements of the table from the 80486 that Intel engineers thought could never be accessed, and they optimized the PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

A math professor at Lynchburg College in Virginia, Thomas Nicely, discovered the bug in September 1994. After calling Intel technical support and getting no official reaction, he posted his discovery on the Internet. This post led to a story in a trade magazine, which in turn caused Intel to issue a press release. It called the bug a glitch that would affect only theoretical mathematicians, with the average spreadsheet user seeing an error every 27,000 years. IBM Research soon counterclaimed that the average spreadsheet user would see an error every 24 days. Intel soon threw in the towel by making the following announcement on December 21:

> *We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer.*

Analysts estimate that this recall cost Intel $500 million, and Intel engineers did not get a Christmas bonus that year.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

3.10 Concluding Remarks

Over the decades, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's

complement binary integer arithmetic is found in every computer sold today, and if it includes floating point support, it offers the IEEE 754 binary floating-point arithmetic.

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapters 4 and 5 discuss exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation of floating point are part of the inspiration for the field of numerical analysis. The switch to **parallelism** will shine the searchlight on numerical analysis again, as solutions that were long considered safe on sequential computers must be reconsidered when trying to find the fastest algorithm for parallel computers that still achieves a correct result.

Data-level parallelism, specifically subword parallelism, offers a simple path to higher performance for programs that are intensive

in arithmetic operations for either integer or floating-point data. We showed that we could speed up matrix multiply nearly fourfold by using instructions that could execute four floating-point operations at a time.

With the explanation of computer arithmetic in this chapter comes a description of much more of the RISC-V instruction set.

Figure 3.24 ranks the popularity of the twenty most common RISC-V instructions the for SPEC CPU2006 integer and floating-point benchmarks. As you can see, a relatively small number of instructions dominate these rankings. This observation has significant implications for the design of the processor, as we will see in Chapter 4.

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load doubleword	ld	8.27%	22.63%
Load fl. pt. double	fld	6.83%	29.46%
Add registers	add	6.23%	35.69%
Load word	lw	4.38%	40.07%
Store doubleword	sd	4.29%	44.36%
Branch if not equal	bne	4.14%	48.50%
Shift left immediate	slli	3.65%	52.15%
Fused mul-add double	fmadd.d	3.49%	55.64%
Branch if equal	beq	3.27%	58.91%
Add immediate word	addiw	2.86%	61.77%
Store fl. pt. double	fsd	2.24%	64.00%
Multiply fl. pt. double	fmul.d	2.02%	66.02%
Load upper immediate	lui	1.56%	67.59%
Store word	sw	1.52%	69.10%
Jump and link	jal	1.38%	70.49%
Branch if less than	blt	1.37%	71.86%
Add word	addw	1.34%	73.19%
Subtract fl. pt. double	fsub.d	1.28%	74.47%
Branch if greater/equal	bge	1.27%	75.75%

FIGURE 3.24 The frequency of the RISC-V instructions for the SPEC CPU2006 benchmarks. The 20 most popular instructions, which collectively account for 76% of all instructions executed, are included in the table. Pseudoinstructions are converted into RISC-V before execution, and hence do not appear here, explaining in part the popularity of `addi`.

No matter what the instruction set or its size—RISC-V, MIPS, x86—never forget that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, string, instruction, and so on. In stored-program computers, it is the operation on the bit pattern that determines its meaning.

Historical Perspective and Further Reading

> *Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."*
>
> W. Kahan, 1992

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the x86. See the rest of  **Section 3.11** online.

Historical Perspective and Further Reading

> *Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."*
>
> W. Kahan, 1992

This section surveys the history of the floating point going back to

von Neumann, including the surprisingly controversial IEEE standards effort, the rationale for the 80-bit stack architecture for floating point in the IA-32, and an update on the next round of the standard.

At first it may be hard to imagine a subject of less interest than the correctness of computer arithmetic or its accuracy, and harder still to understand why a subject so old and mathematical should be so contentious. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical reuse of registers during serial multiplication and division, still command respect today. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest stored-program computer had been built:

> *… a project under von Neumann was to be set up at the Institute of Advanced Studies in Princeton.… Goldstine explained to me the principal features of the design, including the device whereby the digits of the multiplier were put into the tail of the accumulator and shifted out as the least significant part of the product was shifted in. I expressed some admiration at the way registers and shifting circuits were arranged … and Goldstine remarked that things of that nature came very easily to von Neumann.*

There is no controversy here; it can hardly arise in the context of exact integer arithmetic, so long as there is general agreement on what integer the correct result should be. However, as soon as approximate arithmetic enters the picture, so does controversy, as if one person's "negligible" must be another's "everything."

The First Dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann was aware of Konrad Zuse's proposal for a computer in Germany in 1939 that was never built, probably because the floating point made it appear too complicated to finish before the Germans expected World War II to end. Hence, von Neumann refused to include it in the computer he built at Princeton. In an influential report coauthored in 1946 with H. H. Goldstine and A. W. Burks, he gave the arguments for and against

floating point. In favor:

> *… to retain in a sum or product as many significant digits as possible and … to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplication constants which serve to keep numbers within the limits of the machine.*

Floating point was excluded for several reasons:

> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

The argument seems to be that most bits devoted to exponent fields would be bits wasted. Experience has proven otherwise.

One software approach to accommodate reals without floating-point hardware was called *floating vectors*; the idea was to compute at runtime one scale factor for a whole array of numbers, choosing the scale factor so that the array's biggest number would barely fill its field. By 1951, James H. Wilkinson had used this scheme extensively for matrix computations. The problem proved to be that a program might encounter a very large value, and hence the scale factor must accommodate these rare sizeable numbers. The common numbers would thus have many leading 0s, since all numbers had to use a single scale factor. Accuracy was sacrificed, because the least significant bits had to be lost on the right to accommodate leading 0s. This wastage became obvious to practitioners on early computers that displayed all their memory bits as dots on cathode ray tubes (like TV screens) because the loss of precision was visible. Where floating point deserved to be used,

no practical alternative existed.

Thus, true floating-point hardware became popular because it was useful. By 1957, floating-point hardware was almost ubiquitous. A decimal floating-point unit was available for the IBM 650, and soon the IBM 704, 709, 7090, 7094 … series would offer binary floating-point hardware for double as well as single precision.

As a result, everybody had floating point, but every implementation was different.

Diversity versus Portability

Since roundoff introduces some error into almost all floating-point operations, to complain about another bit of error seems picayune. So for 20 years, nobody complained much that those operations behaved a little differently on different computers. If software required clever tricks to circumvent those idiosyncrasies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a long time, matrix computations mystified most people who had no notion of error analysis; perhaps this continues to be true. That may be why people are still surprised that numerically stable matrix computations depend upon the quality of arithmetic in so few places, far fewer than are generally supposed. Books by Wilkinson and widely used software packages like Linpack and Eispack sustained a false impression, widespread in the early 1970s, that a modicum of skill sufficed to produce *portable* numerical software.

"Portable" here means that the software is distributed as source code in some standard language to be compiled and executed on practically any commercially significant computer, and that it will then perform its task as well as any other program performs that task on that computer. Insofar as numerical software has often been thought to consist entirely of computer-independent mathematical formulas, its portability has commonly been taken for granted; the mistake in that presumption will become clear shortly.

Packages like Linpack and Eispack cost so much to develop—over a hundred dollars per line of Fortran delivered—that they could not have been developed without U.S. government subsidy; their portability was a precondition for that subsidy. But nobody

thought to distinguish how various components contributed to their cost. One component was algorithmic—devise an algorithm that deserves to work on at least one computer despite its roundoff and over-/underflow limitations. Another component was the software engineering effort required to achieve and confirm portability to the diverse computers commercially significant at the time; this component grew more onerous as ever more diverse floating-point arithmetics blossomed in the 1970s. And yet scarcely anybody realized how much that diversity inflated the cost of such software packages.

A Backward Step

Early evidence that somewhat different arithmetics could engender grossly different software development costs was presented in 1964. It happened at a meeting of SHARE, the IBM mainframe users' group, at which IBM announced System/360, the successor to the 7094 series. One of the speakers described the tricks he had been forced to devise to achieve a level of quality for the S/360 library that was not quite so high as he had previously achieved for the 7094.

Von Neumann could have foretold part of the trouble, had he still been alive. In 1948, he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it. It did predict correctly, however, that computations with larger arrays of data would probably fall prey to roundoff more often. IBM S/360s had bigger memories than 7094s, so data arrays could grow larger, and they did. To make matters worse, the S/360s had narrower single precision words (32 bits versus 36) and used a cruder arithmetic (hexadecimal or base 16 versus binary or base 2) with consequently poorer worst-case precision (21 significant bits versus 27) than the old 7094s. Consequently, software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on S/360s. The quickest way to recover adequate accuracy was to replace old codes' single precision declarations with double precision before recompilation for the S/360. This practice exercised S/360 double precision far more than had been expected.

The early S/360's worst troubles were caused by lack of a guard

digit in double precision. This lack showed up in multiplication as a failure of identities like 1.0* x=x because multiplying x by 1.0 dropped x's last hexadecimal digit (4 bits). Similarly, if x and y were very close but had different exponents, subtraction dropped off the last digit of the smaller operand before computing $x-y$. This final aberration in double precision undermined a precious theorem that single precision then (and now) honored: If $1/2 \le x/y \le 2$, then no rounding error can occur when $x-y$ is computed; it must be computed exactly.

Innumerable computations had benefited from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof. We had been taking all this stuff for granted.

The identities and theorems about exact relationships that persisted, despite roundoff, with reasonable implementations of approximate arithmetic were not appreciated until they were lost. Previously, it had been thought that the things to matter were precision (how many significant digits were carried) and range (the spread between over-/underflow thresholds). Since the S/360's double precision had more precision and wider range than the 7094's, software was expected to continue to work at least as well as before. But it didn't.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on S/360s. A small subcommittee of SHARE proposed improvements to the S/360 floating point. This committee was surprised and grateful to get a fair part of what they asked for from IBM, including all-important guard digits. By 1968, these had been retrofitted to S/360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility (a lesson learned by Intel 30 years later). IBM employees who can remember the incident still shudder.

The People Who Built the Bombs

Seymour Cray was associated for decades with the CDC and Cray computers that were, when he built them, the world's biggest and fastest. He always understood what his customers wanted most: *speed*. And he gave it to them even if, in so doing, he also gave them arithmetics more "interesting" than anyone else's. Among his

customers have been the great government laboratories like those at Livermore and Los Alamos, where nuclear weapons were designed. The challenges of "interesting" arithmetics were pretty tame to people who had to overcome Mother Nature's challenges.

Perhaps all of us could learn to live with arithmetic idiosyncrasy if only one computer's idiosyncrasies had to be endured. Instead, when accumulating different computers' different anomalies, software dies the Death of a Thousand Cuts. Here is an example from Cray's computers:

```
if (x == 0.0) y = 17.0 else y = z/x
```

Could this statement be stopped by a divide-by-zero error? On a CDC 6600 it could. The reason was a conflict between the 6600's adder, where `x` was compared with 0.0, and the multiplier and divider. The adder's comparison examined `x`'s leading 13 bits, which sufficed to distinguish zero from normal nonzero floating-point numbers `x`. The multiplier and divider examined only 12 leading bits. Consequently, tiny numbers existed that were nonzero to the adder but zero to the multiplier and divider! To avoid disasters with these tiny numbers, programmers learned to replace statements like the one above with

```
if (1.0 * x == 0.0) y = 17.0 else y = z/x
```

But this statement is unsafe to use in would-be portable software because it malfunctions obscurely on other computers designed by Cray, the ones marketed by Cray Research, Inc. If `x` was so huge that 2.0 * `x` would overflow, then 1.0 * `x` might overflow too! Overflow happens because Cray computers check the product's exponent before the product's exponent has been normalized, just to save the delay of a single AND gate.

Rounding error anomalies that are far worse than the over-/underflow anomaly just discussed also affect Cray computers. The worst error came from the lack of a guard digit in add/subtract, an affliction of IBM S/360s. Further bad luck for software is occasioned by the way Cray economized his multiplier; about one-third of the bits that normal multiplier arrays generate have been left out of his multipliers, because they would contribute less than a unit to the last place of the final Cray-rounded product. Consequently, a Cray multiplier errs by almost a bit more than might have been expected. This error is compounded when division takes three multiplications to improve an approximate reciprocal of the divisor and then

multiply the numerator by it. Square root compounds a few more multiplication errors.

The fast way drove out the slow, even though the fast was occasionally slightly wrong.

Making the World Safe for Floating Point, or Vice Versa

William Kahan was an undergraduate at the University of Toronto in 1953 when he learned to program its Ferranti-Manchester Mark-I computer. Because he entered the field early, Kahan became acquainted with a wide range of devices and a large proportion of the personalities active in computing; the numbers of both were small at that time. He has performed computations on slide rules, desktop mechanical calculators, tabletop analog differential analyzers, and so on; he has used all but the earliest electronic computers and calculators mentioned in this book.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two years of postdoctoral study in England, where he became acquainted with Wilkinson. In 1960, he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free rein to tinker with its operating system, Fortran compiler, and runtime library. (He denies that he ever came near the 7090 hardware with a soldering iron but admits asking to do so.) One story from that time illuminates how misconceptions and numerical anomalies in computer systems can incur awesome hidden costs.

A graduate student in aeronautical engineering used the 7090 to simulate the wings he was designing for short takeoffs and landings. He knew such a wing would be difficult to control if its characteristics included an abrupt onset of stall, but he thought he could avoid that. His simulations were telling him otherwise. Just to be sure that roundoff was not interfering, he had repeated many of his calculations in double precision and gotten results much like those in single; his wings had stalled abruptly in both precisions. Disheartened, the student gave up.

Meanwhile Kahan replaced IBM's logarithm program (ALOG) with one of his own, which he hoped would provide better accuracy. While testing it, Kahan reran programs using the new

version of ALOG. The student's results changed significantly; Kahan approached him to find out what had happened.

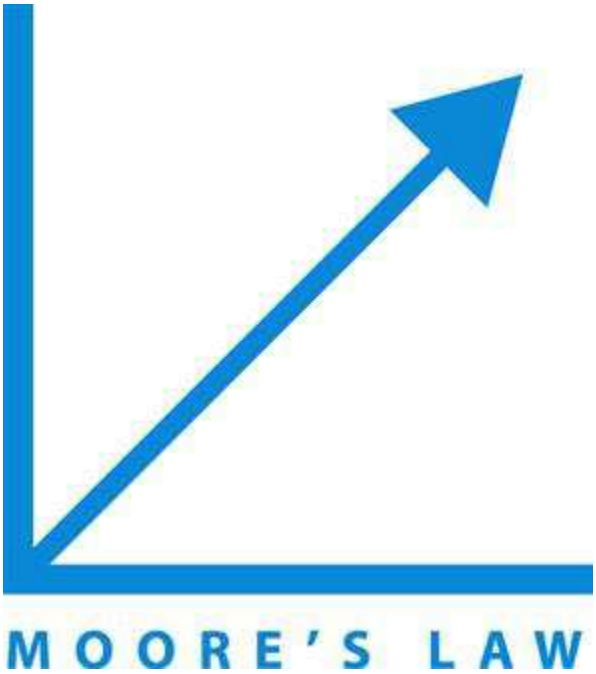
The student was puzzled. Much as the student preferred the results produced with the new ALOG—they predicted a gradual stall—he knew they must be wrong because they disagreed with his double precision results. The discrepancy between single and double precision results disappeared a few days later when a new release of IBM's double precision arithmetic software for the 7090 arrived. (The 7090 had no double precision hardware.) He went on to write a thesis about it and to build the wings; they performed as predicted. But that is not the end of the story.

In 1963, the 7090 was replaced by a faster 7094 with double precision floating-point hardware but with otherwise practically the same instruction set as the 7090. Only in double precision and only when using the new hardware did the wing stall abruptly again. A lot of time was spent to find out why. The 7094 hardware turned out, like the superseded 7090 software and the subsequent early S/360s, to lack a guard bit in double precision. Like so many programmers on those computers and on Cray's, the student discovered a trick to compensate for the lack of a guard digit; he wrote the expression `(0.5 − x) + 0.5` in place of `1.0 − x`. Nowadays we would blush if we had to explain why such a trick might be necessary, but it solved the student's problem.

Meanwhile the lure of California was working on Kahan and his family; they came to Berkeley and he to the University of California. An opportunity presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and his work dramatically improved the accuracy of HP calculators, but that is another story. Fruitful collaboration with congenial coworkers, however, fortified him for the next and crucial opportunity.

It came in 1976, when John F. Palmer at Intel was empowered to specify the "best possible" floating-point arithmetic for all of Intel's product line, as **Moore's Law** made it now possible to create a whole floating-point unit on a single chip. The floating-point standard was originally started for the iAPX-432, but when it was late, Intel started the 8086 as a short-term emergency stand-in until the iAPX-432 was ready. The iAPX-432 never became popular, so the emergency stand-in became the standard-bearer for Intel. The

8087 floating-point coprocessor for the 8086 was contemplated. (A *coprocessor* is simply an additional chip that accelerates a portion of the work of a processor; in this case, it accelerated floating-point computation.)



Palmer had obtained his Ph.D. at Stanford a few years before and knew whom to call for counsel of perfection—Kahan. They put together a design that obviously would have been impossible only a few years earlier and looked not quite possible at the time. But a new Israeli team of Intel employees led by Rafi Navé felt challenged to prove their prowess to Americans and leaped at an opportunity to put something impossible on a chip—the 8087.

By now, floating-point arithmetics that had been merely diverse among mainframes had become chaotic among microprocessors, one of which might be host to a dozen varieties of arithmetic in ROM firmware or software. Robert G. Stewart, an engineer prominent in IEEE activities, got fed up with this anarchy and proposed that the IEEE draft a decent floating-point standard. Simultaneously, word leaked out in Silicon Valley that Intel was going to put on one chip some awesome floating point well beyond anything its competitors had in mind. The competition had to find a way to slow Intel down, so they formed a committee to do what Stewart requested.

Meetings of this committee began in late 1977 with a plethora of competing drafts from innumerable sources and dragged on into 1985, when IEEE Standard 754 for Binary Floating Point was made official. The winning draft was very close to one submitted by Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based on the Intel design, with Intel's permission, of course, as simplified by Coonen. Their harmonious combination of features, almost none of them new, had at the outset attracted more support within the committee and from outside experts like Wilkinson than any other draft, but they had to win nearly unanimous support within the committee to win official IEEE endorsement, and that took time.

The First IEEE 754 Chips

In 1980, Intel became tired of waiting and released the 8087 for use in the IBM PC. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." (Kahan's [1990] history of the stack architecture selection for the 8087 is entertaining reading.)

Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was that the restriction of the top of stack as one operand was not so bad since it only required the execution of an `FXCH` instruction (which swapped registers) to get the same result as a two-operand instruction, and `FXCH` was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence, he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data.

The Intel 8087 was implemented in Israel, and 7500 miles and 10

time zones made communication from California difficult. According to Palmer and Morse (*The 8087 Primer*, J. Wiley, New York, 1984, p. 93):

> *Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid operation is indeed due to a stack overflow.… Also lacking is the ability to restart the instruction that caused the stack overflow …*

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> *Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/under-flow.…*
>
> *I still regret that the 8087's stack implementation was not quite so neat as my original intention.… If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.*

In 1982, Motorola announced its 68881, which found a place in Sun 3s and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Another Berkeley graduate student, George S. Taylor, had soon designed a high-speed implementation of the proposed standard for an early superminicomputer (ELXSI 6400). The standard was becoming de facto before its final draft's ink was dry.

An early rush of adoptions gave the computing industry the false impression that IEEE 754, like so many other standards, could be implemented easily by following a standard recipe. Not true. Only the enthusiasm and ingenuity of its early implementors made it look easy.

In fact, to implement IEEE 754 correctly demands extraordinarily

diligent attention to detail; to make it run fast demands extraordinarily competent ingenuity of design. Had the industry's engineering managers realized this, they might not have been so quick to affirm that, as a matter of policy, "We conform to all applicable standards."

IEEE 754 Today

Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied.

The successful parts of IEEE 754 are that it is a widely implemented standard with a common floating-point format, that it requires minimum accuracy to one-half ulp in the least significant bit, and that operations must be commutative.

The IEEE 754/854 has been implemented to a considerable degree of fidelity in at least part of the product line of every North American computer manufacturer. The only significant exceptions were the DEC VAX, IBM S/370 descendants, and Cray Research vector supercomputers, and all three have been replaced by compliant computers.

IEEE rules ask that a standard be revisited periodically for updating. A committee started in 2000, and drafts of the revised standards were circulated for voting, and these were approved in 2008. The revised standard, IEEE Std 754-2008 [2008], includes several new types: 16-bit floating point, called *half precision*; 128-bit floating point, called *quad precision*; and three decimal types, matching the length of the 32-bit, 64-bit, and 128-bit binary formats. In 1989, the Association for Computing Machinery, acknowledging the benefits conferred upon the computing industry by IEEE 754, honored Kahan with the Turing Award. On accepting it, he thanked his many associates for their diligent support, and his adversaries for their blunders. So . . . not all errors are bad.

Further Reading

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 2002] are good starting points; they abound with pointers to further reading. Several of the stories told in this section come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the most recent IEEE-sponsored Symposium on Computer Arithmetic, held every two years; the 23rd was held in 2016.

Burks, A.W., H.H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97–146.

This classic paper includes arguments against floating-point hardware.

Goldberg, D. [2002]. "Computer arithmetic". Appendix A of *Computer Architecture: A Quantitative Approach*, third edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.

A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4–3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* 23(1), 5–48.

Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing 71* (Proc. IFIP Congress 71 in Ljubljana), Vol. 2, North-Holland Publishing, Amsterdam, 1214–1239.

This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, 12–26.

The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University

of California, Berkeley.

What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available at *http://www.cims.nyu.edu/~dbindel/class/cs279/87stack.pdf*.

A collection of memos related to floating point, including "Beastly numbers" (another less famous Pentium bug), "Notes on the IEEE floating point arithmetic" (including comments on how some features are atrophying), and "The baleful effects of computing benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, …).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.

A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.

This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

3.12 Exercises

> *Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.*
>
> Winston Churchill, address at Harrow School, 1941

3.1 [5] <§3.2>What is 5ED4 −07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

3.2 [5] <§3.2>What is 5ED4 −07A4 when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

3.3 [10] <§3.2>Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

3.4 [5] <§3.2>What is 4365 −3412 when these values represent unsigned 12-bit octal numbers? The result should be written in

octal. Show your work.

3.5 [5] <§3.2>What is 4365 −3412 when these values represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

3.6 [5] <§3.2>Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate 185–122. Is there overflow, underflow, or neither?

3.7 [5] <§3.2>Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 +122. Is there overflow, underflow, or neither?

3.8 [5] <§3.2>Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 −122. Is there overflow, underflow, or neither?

3.9 [10] <§3.2>Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151 +214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.10 [10] <§3.2>Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151 −214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.11 [10] <§3.2>Assume 151 and 214 are unsigned 8-bit integers. Calculate 151+214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.12 [20] <§3.3>Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

3.13 [20] <§3.3>Using a table similar to that shown in Figure 3.6, calculate the product of the hexadecimal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.

3.14 [10] <§3.3>Calculate the time necessary to perform a multiply using the approach given in Figures 3.3 and 3.4 if an integer is 8 bits wide and each step of the operation takes four time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If

this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

3.15 [10] <§3.3>Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes four time units.

3.16 [20] <§3.3>Calculate the time necessary to perform a multiply using the approach given in Figure 3.7 if an integer is 8 bits wide and an adder takes four time units.

3.17 [20] <§3.3>As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9 ×6, for example, can be written (2 ×2 ×2 +1)×6, we can calculate 9 ×6 by shifting 6 to the left three times and then adding 6 to that result. Show the best way to calculate 0 ×33 ×0 ×55 using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

3.18 [20] <§3.4>Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.8. You should show the contents of each register on each step. Assume both inputs are unsigned 6-bit integers.

3.19 [30] <§3.4>Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.11. You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers. This algorithm requires a slightly different approach than that shown in Figure 3.9. You will want to think hard about this, do an experiment or two, or else go to the web to figure out how to make this work correctly. (Hint: one possible solution involves using the fact that Figure 3.11 implies the remainder register can be shifted either direction.)

3.20 [5] <§3.5>What decimal number does the bit pattern `0×0C000000` represent if it is a two's complement integer? An unsigned integer?

3.21 [10] <§3.5>If the bit pattern `0×0C000000` is placed into the Instruction Register, what MIPS instruction will be executed?

3.22 [10] <§3.5>What decimal number does the bit pattern `0×0C000000` represent if it is a floating point number? Use the

IEEE 754 standard.

3.23 [10] <§3.5>Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

3.24 [10] <§3.5>Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

3.25 [10] <§3.5>Write down the binary representation of the decimal number 63.25 assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

3.26 [20] <§3.5>Write down the binary bit pattern to represent -1.5625×10^{-1} assuming a format similar to that employed by the DEC PDP-8 (the leftmost 12 bits are the exponent stored as a two's complement number, and the rightmost 24 bits are the fraction stored as a two's complement number). No hidden 1 is used. Comment on how the range and accuracy of this 36-bit pattern compares to the single and double precision IEEE 754 standards.

3.27 [20] <§3.5>IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

3.28 [20] <§3.5>The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in two's complement format, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the rightmost 8 bits representing the exponent. However, in an interesting twist, the exponent was stored in sign-magnitude format with the sign bit on the far right! Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. No hidden 1 is used. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

3.29 [20] <§3.5>Calculate the sum of 2.6125×10^{1} and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the

steps.

3.30 [30] <§3.5>Calculate the product of -8.0546875×10^{0} and $-1.79931640625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision format described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps; however, as is done in the example in the text, you can do the multiplication in human-readable format instead of using the techniques described in Exercises 3.12 through 3.14. Indicate if there is overflow or underflow. Write your answer in both the 16-bit floating point format described in Exercise 3.27 and also as a decimal number. How accurate is your result? How does it compare to the number you get if you do the multiplication on a calculator?

3.31 [30] <§3.5>Calculate by hand 8.625×10^{1} divided by -4.875×10^{0}. Show all the steps necessary to achieve your answer. Assume there is a guard, a round bit, and a sticky bit, and use them if necessary. Write the final answer in both the 16-bit floating point format described in Exercise 3.27 and in decimal and compare the decimal result to that which you get if you use a calculator.

3.32 [20] <§3.10>Calculate $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.33 [20] <§3.10>Calculate $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.34 [10] <§3.10>Based on your answers to Exercises 3.32 and 3.33, does $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3} = 3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$?

3.35 [30] <§3.10>Calculate $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2}$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and

write your answer in both the 16-bit floating point format and in decimal.

3.36 [30] <§3.10>Calculate $3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.37 [10] <§3.10>Based on your answers to Exercises 3.35 and 3.36, does $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2} = 3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$?

3.38 [30] <§3.10>Calculate $1.666015625 \times 10^{0} \times (1.9760 \times 10^{4} + -1.9744 \times 10^{4})$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.39 [30] <§3.10>Calculate $(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}) + (1.666015625 \times 10^{0} \times -1.9744 \times 10^{4})$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.40 [10] <§3.10>Based on your answers to Exercises 3.38 and 3.39, does $(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}) + (1.666015625 \times 10^{0} \times -1.9744 \times 10^{4}) = 1.666015625 \times 10^{0} \times (1.9760 \times 10^{4} + -1.9744 \times 10^{4})$?

3.41 [10] <§3.5>Using the IEEE 754 floating point format, write down the bit pattern that would represent −1/4. Can you represent −1/4 exactly?

3.42 [10] <§3.5>What do you get if you add −1/4 to itself four times? What is −1/4 ×4? Are they the same? What should they be?

3.43 [10] <§3.5>Write down the bit pattern in the fraction of value 1/3 assuming a floating point format that uses binary numbers in the fraction. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.44 [10] <§3.5>Write down the bit pattern in the fraction of value

1/3 assuming a floating point format that uses Binary Coded Decimal (base 10) numbers in the fraction instead of base 2. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.45 [10] <§3.5>Write down the bit pattern assuming that we are using base 15 numbers in the fraction of value 1/3 instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 15 numbers would use 0–9 and A–E.) Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.46 [20] <§3.5>Write down the bit pattern assuming that we are using base 30 numbers in the fraction of value 1/3 instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 30 numbers would use 0–9 and A–T.) Assume there are 20 bits, and you do not need to normalize. Is this representation exact?

3.47 [45] <§§3.6, 3.7>The following C code implements a four-tap FIR filter on input array `sig_in`. Assume that all arrays are 16-bit fixed-point values.

```
for (i = 3;i< 128;i+ +)
sig_out[i] = sig_in[i - 3] * f[0] + sig_in[i - 2] * f[1]
 + sig_in[i - 1] * f[2] + sig_in[i] * f[3];
```

Assume you are to write an optimized implementation of this code in assembly language on a processor that has SIMD instructions and 128-bit registers. Without knowing the details of the instruction set, briefly describe how you would implement this code, maximizing the use of sub-word operations and minimizing the amount of data that is transferred between registers and memory. State all your assumptions about the instructions you use.

Answers to Check Yourself

§3.2, page 177: 2.

§3.5, page 215: 3.

CHAPTER 10

COMPUTER ARITHMETIC

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the distinction between the way in which numbers are represented (the binary format) and the algorithms used for the basic arithmetic operations.
- Explain **twos complement representation**.
- Present an overview of the techniques for doing basic arithmetic operation in two complement notation.
- Understand the use of significand, base, and exponent in the representation of **floating-point numbers**.
- Present an overview of the IEEE 754 standard for floating-point representation.
- Understand some of the key concepts related to floating-point arithmetic, including guard bits, rounding, subnormal numbers, underflow and overflow.

We begin our examination of the processor with an overview of the arithmetic and logic unit (ALU). The chapter then focuses on the most complex aspect of the ALU, computer arithmetic. The implementations of simple logic and arithmetic functions in digital logic are described in Chapter 11, and logic functions that are part of the ALU are described in Chapter 12.

Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point. In both cases, the representation chosen is a crucial design issue and is treated first, followed by a discussion of arithmetic operations.

This chapter includes a number of examples, each of which is highlighted in a shaded box.

10.1 THE ARITHMETIC AND LOGIC UNIT

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 10.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.3). The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

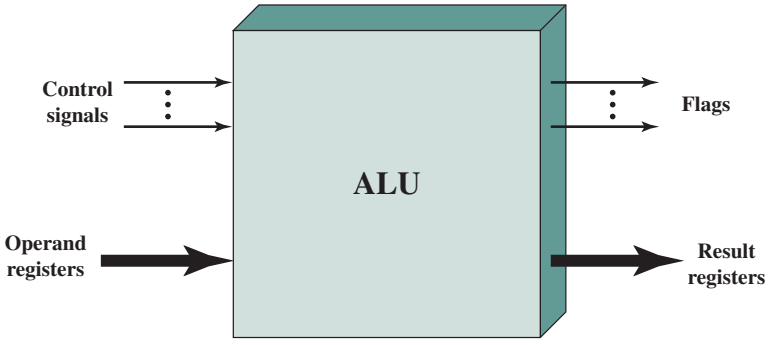


Figure 10.1 ALU Inputs and Outputs

The flag values are also stored in registers within the processor. The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

10.2 INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign (for negative numbers), and the period, or **radix point** (for numbers with a fractional component).

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of special symbols for the minus sign and radix point. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, such as

$$\begin{aligned} 00000000 &= \quad 0 \\ 00000001 &= \quad 1 \\ 00101001 &= \quad 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

[1]See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \quad \text{(sign magnitude)} \end{aligned}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \tag{10.1}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 10.3. Another drawback is that there are two representations of 0:

$$\begin{aligned} +0_{10} &= 00000000 \\ -0_{10} &= 10000000 \quad \text{(sign magnitude)} \end{aligned}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 10.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Most treatments of twos complement representation focus on the rules for producing negative numbers, with no formal proof that the scheme is valid. Instead,

[2]In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

our presentation of twos complement integers in this section and in Section 10.3 is based on [DATT93], which suggests that twos complement representation is best understood by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number $A(A < 0)$, the sign bit, a_{n-1}, is one. The remaining $n - 1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}. We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, as we will see in Section 10.3, if the weight of the most significant bit is -2^{n-1}. This is the convention used in twos complement representation, yielding the following expression for negative numbers:

Twos Complement

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textbf{(10.2)}$$

Equation (10.2) defines the twos complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines

Table 10.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

a nonnegative integer. When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, yielding a negative integer.

Table 10.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is 1 (2^0) and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 10.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 10.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

Range Extension

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. This expansion of bit length is referred to as **range extension**, because the range of numbers that can be expressed is extended by increasing the bit length.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Figure 10.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 10.2. The last line can be verified using Equation (10.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

To see why this rule works, let us again consider an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ interpreted as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with $m > n$. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

The two values must be equal:

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$-2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow a_{m-2} = \ldots = a_{n-2} = a_{n-2} = 1$$

In going from the first to the second equation, we require that the least significant $n - 1$ bits do not change between the two representations. Then we get to the next to last equation, which is only true if all of the bits in positions $n - 1$ through $m - 2$ are 1. Therefore, the sign-extension rule works. The reader may find the rule easier to grasp after studying the discussion on twos complement negation at the beginning of Section 10.3.

Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

10.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

$$\begin{array}{rcll} +18 & = & 00010010 & \text{(twos complement)} \\ \text{bitwise complement} & = & 11101101 & \\ & & \underline{+\qquad 1} & \\ & & 11101110 = -18 & \end{array}$$

As expected, the negative of the negative of that number is itself:

$$\begin{array}{rcll} -18 & = & 11101110 & \text{(twos complement)} \\ \text{bitwise complement} & = & 00010001 & \\ & & \underline{+\qquad 1} & \\ & & 00010010 = +18 & \end{array}$$

We can demonstrate the validity of the operation just described using the definition of the twos complement representation in Equation (10.2). Again, interpret an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Now form the bitwise complement, $\overline{a_{n-1}a_{n-2}} \ldots \overline{a_0}$, and, treating this as an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer B, so that its value is

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Now, we want $A = -B$, which means $A + B = 0$. This is easily shown to be true:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i})\right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

$$\begin{array}{rcl} 0 & = & 00000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 11111111 \\ & & \underline{+\qquad 1} \\ & & 100000000 = 0 \end{array}$$

There is a *carry* out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

$$\begin{array}{rcl} +128 & = & 10000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 01111111 \\ & & \underline{+\qquad 1} \\ & & 10000000 = -128 \end{array}$$

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is $2n$, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

Addition and Subtraction

Addition in twos complement is illustrated in Figure 10.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Figure 10.3 Addition of Numbers in Twos Complement Representation

Figures 10.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 10.4. The last two examples demonstrate that the overflow rule still applies.

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M − S)

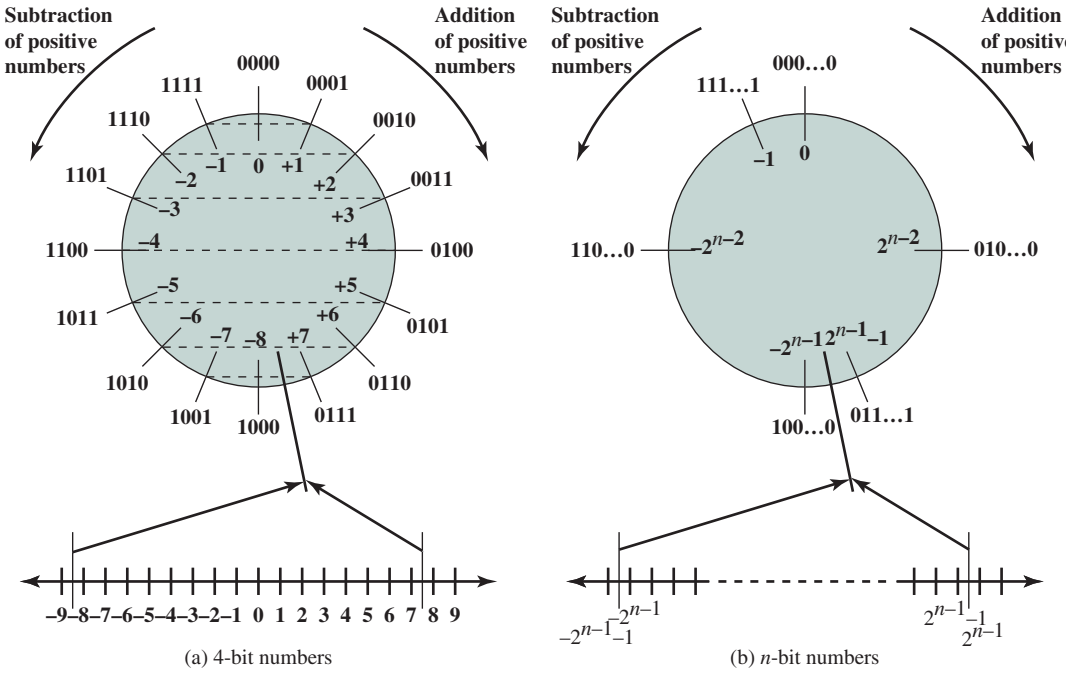


Figure 10.5 Geometric Depiction of Twos Complement Integers

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 10.5. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive k (or subtract negative k) to that number by moving k positions clockwise, and we can subtract positive k (or add negative k) from that number by moving k positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

ALL OF the examples of Figures 10.3 and 10.4 are easily traced in the circle of Figure 10.5.

Figure 10.6 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Chapter 11.) For addition, the two numbers are presented to the adder from two registers, designated in this case as **A** and **B** registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (**B** register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure 10.6 only shows the

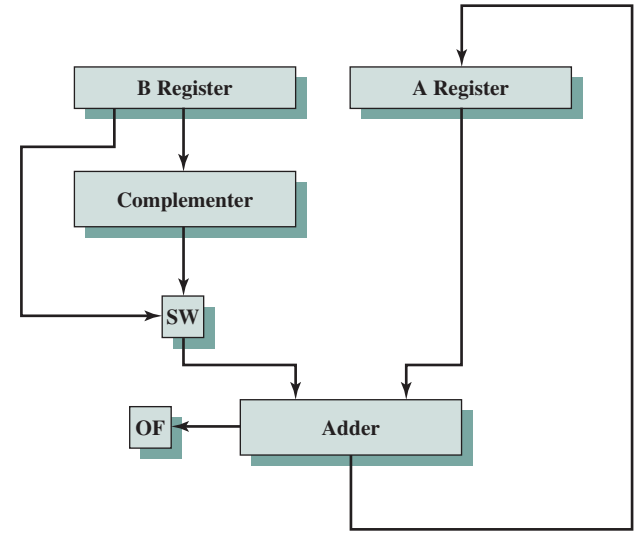


Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

UNSIGNED INTEGERS Figure 10.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.

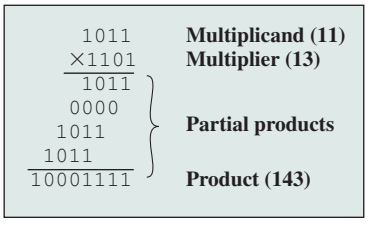
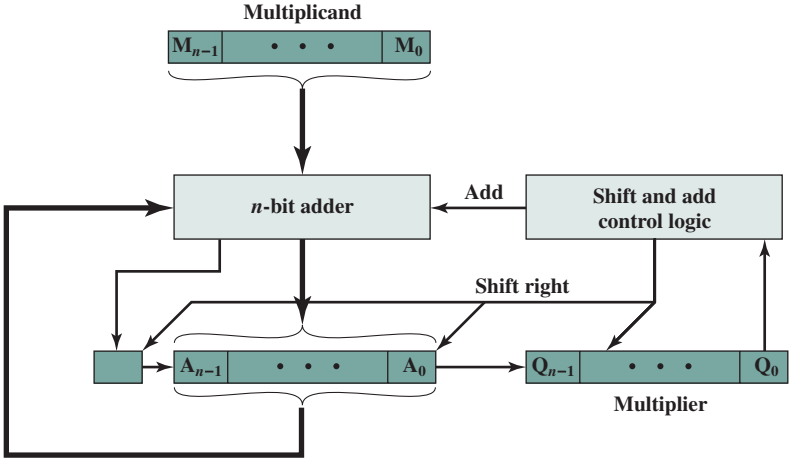


Figure 10.7 Multiplication of Unsigned Binary Integers

2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 10.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	**Initial values**	
0	1011	1101	1011	**Add**	**First cycle**
0	0101	1110	1011	**Shift**	
0	0010	1111	1011	**Shift**	**Second cycle**
0	1101	1111	1011	**Add**	**Third cycle**
0	0110	1111	1011	**Shift**	
1	0001	1111	1011	**Add**	**Fourth cycle**
0	1000	1111	1011	**Shift**	

(b) Example from Figure 10.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers. A flowchart of the operation is shown in Figure 10.9, and an example is given in Figure 10.8b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

TWOS COMPLEMENT MULTIPLICATION We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\,0011 \\ \hline 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7(1001) to 3 (0011) to get -4(1100).

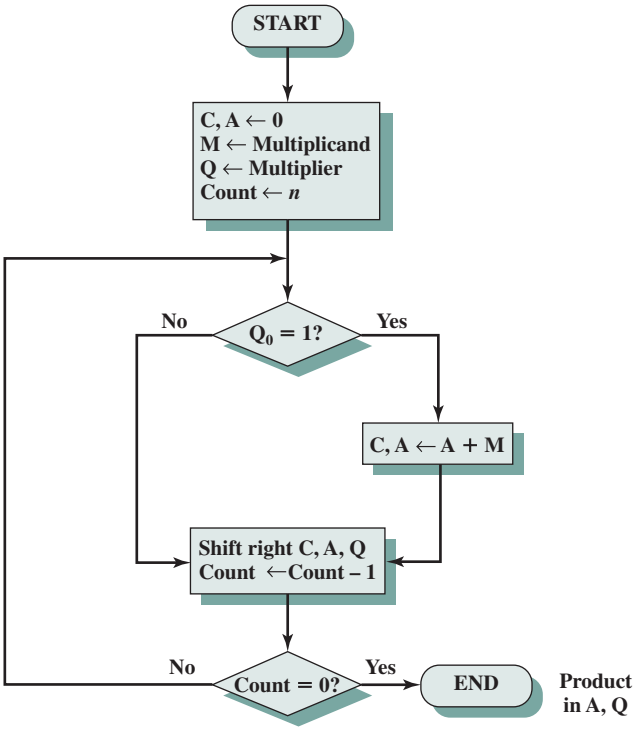


Figure 10.9 Flowchart for Unsigned Binary Multiplication

```
      1011
    × 1101
  --------
  00001011   1011 × 1 × 2^0
  00000000   1011 × 0 × 2^1
  00101100   1011 × 1 × 2^2
  01011000   1011 × 1 × 2^3
  --------
  10001111
```

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 10.7. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have −5(1011) times −3 (1101) equals −113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 10.7 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 2^3 + 2^2 + 2^0$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 10.10 recasts Figure 10.7 to make the generation of partial products by multiplication explicit. The only difference in Figure 10.10 is that it recognizes that the partial products should be viewed as $2n$-bit numbers generated from the n-bit multiplicand.

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This is demonstrated in Figure 10.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted

```
    1001  (9)                  1001  (-7)
  × 0011  (3)                × 0011  (3)
  --------                   --------
  00001001  1001 × 2^0       11111001  (-7) × 2^0 = (-7)
  00010010  1001 × 2^1       11110010  (-7) × 2^1 = (-14)
  --------                   --------
  00011011  (27)             11101011  (-21)
```

(a) Unsigned integers (b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

as the twos complement value −7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in Figure 10.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number −3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \leftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm [BOOT51]. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 10.12 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers,

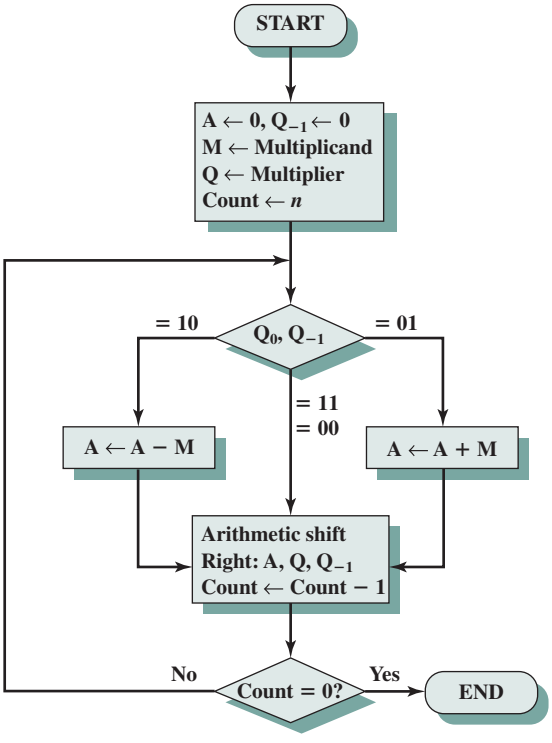


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	**Initial values**	
1001	0011	0	0111	**A←A – M**	**First cycle**
1100	1001	1	0111	**Shift**	
1110	0100	1	0111	**Shift**	**Second cycle**
0101	0100	1	0111	**A←A + M**	**Third cycle**
0010	1010	0	0111	**Shift**	
0001	0101	0	0111	**Shift**	**Fourth cycle**

Figure 10.13 Example of Booth's Algorithm (7 × 3)

respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit.

Figure 10.13 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3. More compactly, the same operation is depicted in Figure 10.14a. The rest of Figure 10.14 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

```
   0111
 × 0011    (0)
11111001   1-0
0000000    1-1
000111     0-1
00010101   (21)
```

(a) (7) × (3) = (21)

```
   0111
 × 1101    (0)
11111001   1-0
0000111    0-1
111001     1-0
11101011   (-21)
```

(b) (7) × (−3) = (−21)

```
   1001
 × 0011    (0)
00000111   1-0
0000000    1-1
111001     0-1
11101011   (-21)
```

(c) (−7) × (3) = (−21)

```
   1001
 × 1101    (0)
00000111   1-0
1111001    0-1
000111     1-0
00010101   (21)
```

(d) (−7) × (−3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

Why does Booth's algorithm work? Consider first the case of a positive multiplier. In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (e.g., 00011110). As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= \text{M} \times (16 + 8 + 4 + 2) \\ &= \text{M} \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \tag{10.3}$$

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^5 - 2^1) \\ &= \text{M} \times (32 - 2) \\ &= \text{M} \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} \text{M} \times (01111010) &= \text{M} \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= \text{M} \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

To show that the same scheme works for a negative multiplier, we need to observe the following. Let X be a negative number in twos complement notation:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3} \ldots x_1x_0\}$$

Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots (x_1 \times 2^1) + (x_0 \times 2^0) \tag{10.4}$$

The reader can verify this by applying the algorithm to the numbers in Table 10.2.

The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111 \ldots 10x_{k-1}x_{k-2} \ldots x_1x_0\} \tag{10.5}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \tag{10.6}$$

From Equation (10.3), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k-1} = 2^{n-1} - 2^{k-1}$$

Rearranging

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \quad \textbf{(10.7)}$$

Substituting Equation (10.7) into Equation (10.6), we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \quad \textbf{(10.8)}$$

At last we can return to Booth's algorithm. Remembering the representation of X [Equation (10.5)], it is clear that all of the bits from x_0 up to the leftmost 0 are handled properly because they produce all of the terms in Equation (10.8) but (-2^{k+1}) and thus are in the proper form. As the algorithm scans over the leftmost 0 and encounters the next 1 (2^{k+1}), a 1–0 transition occurs and a subtraction takes place (-2^{k+1}). This is the remaining term in Equation (10.8).

As an example, consider the multiplication of some multiplicand by (-6). In twos complement representation, using an 8-bit word, (-6) is represented as 11111010. By Equation (10.4), we know that

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

which the reader can easily verify. Thus,

$$\text{M} \times (11111010) = \text{M} \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Using Equation (10.7),

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^1)$$

which the reader can verify is still $\text{M} \times (-6)$. Finally, following our earlier line of reasoning,

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^2 - 2^1)$$

We can see that Booth's algorithm conforms to this scheme. It performs a subtraction when the first 1 is encountered (10), an addition when (01) is encountered, and finally another subtraction when the first 1 of the next block of 1s is encountered. Thus, Booth's algorithm performs fewer additions and subtractions than a more straightforward algorithm.

Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 10.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder.*

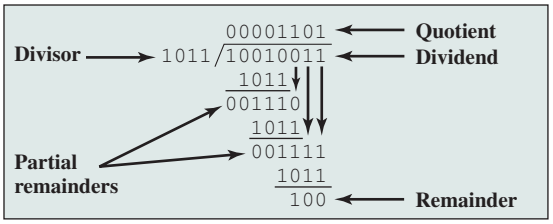


Figure 10.15 Example of Division of Unsigned Binary Integers

From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 10.16 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At

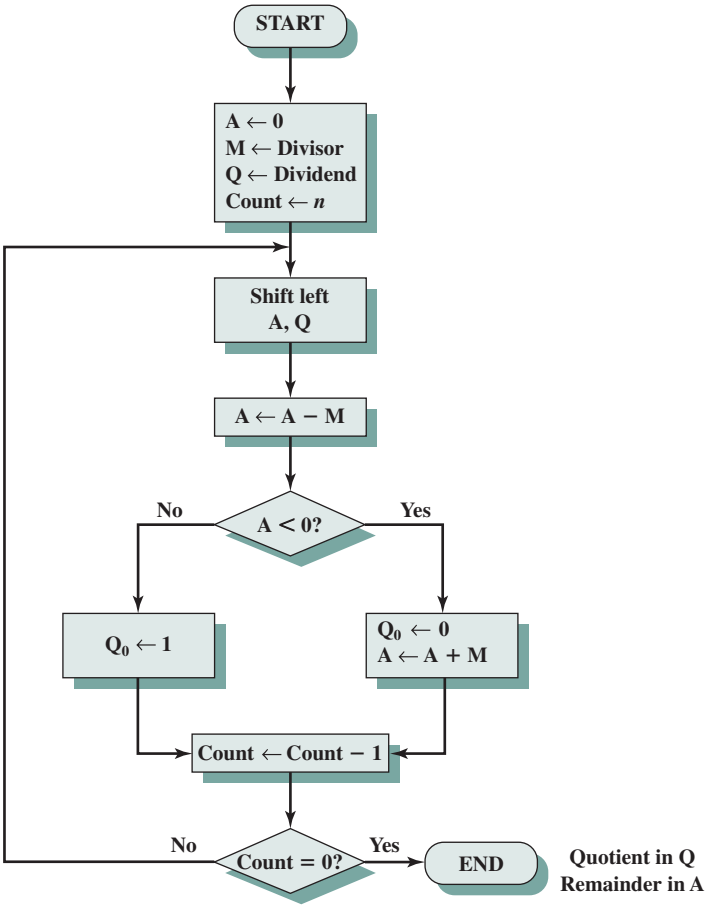


Figure 10.16 Flowchart for Unsigned Binary Division

A	Q	
0000	0111	**Initial value**
0000	1110	**Shift**
1101		**Use twos complement of 0011 for subtraction**
1101		**Subtract**
0000	1110	**Restore, set $Q_0 = 0$**
0001	1100	**Shift**
1101		
1110		**Subtract**
0001	1100	**Restore, set $Q_0 = 0$**
0011	1000	**Shift**
1101		
0000	1001	**Subtract, set $Q_0 = 1$**
0001	0010	**Shift**
1101		
1110		**Subtract**
0001	0010	**Restore, set $Q_0 = 0$**

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.[3] If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. An example of this approach is shown in Figure 10.17.

The algorithm assumes that the divisor V and the dividend D are positive and that $|V| < |D|$. If $|V| = |D|$, then the quotient $Q = 1$ and the remainder $R = 0$. If $|V| > |D|$, then $Q = 0$ and $R = D$. The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a $2n$-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform $A \leftarrow A - M$. This operation subtracts the divisor from the contents of A.
4. **a.** If the result is nonnegative (most significant bit of $A = 0$), then set $Q_0 \leftarrow 1$.
   b. If the result is negative (most significant bit of $A = 1$), then set $Q_0 \leftarrow 0$. and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

[3]This is subtraction of unsigned integers. A result that requires a borrow out of the most significant bit is a negative result.

To deal with negative numbers, we recognize that the remainder is defined by

$$D = Q \times V + R$$

That is, the remainder is the value of R needed for the preceding equation to be valid. Consider the following examples of integer division with all possible combinations of signs of D and V:

$$\begin{array}{llllll} D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\ D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\ D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\ D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1 \end{array}$$

The reader will note from Figure 10.17 that $(-7)/(3)$ and $(7)/(-3)$ produce different remainders. We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable from the signs of D and V. Specifically, $\text{sign}(R) = \text{sign}(D)$ and $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$. Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed. This is the method of choice for the restoring division algorithm [PARH10].

10.4 FLOATING-POINT REPRESENTATION

Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on or near 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

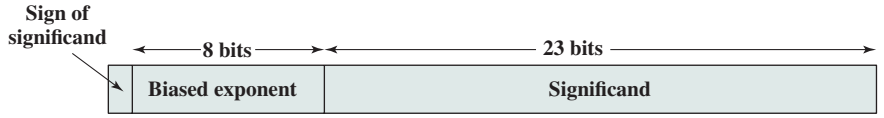
For decimal numbers, we get around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}, What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

$$\pm \mathrm{S} \times \mathrm{B}^{\pm \mathrm{E}}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E



(a) Format

$$
\begin{aligned}
1.1010001 \times 2^{10100} &= 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20} \\
-1.1010001 \times 2^{10100} &= 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20} \\
1.1010001 \times 2^{-10100} &= 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} &= 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}
\end{aligned}
$$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand. That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. Figure 10.18a shows a typical 32-bit floating-point format. The leftmost bit stores the **sign** of the number (0 = positive, 1 = negative). The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field to get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to $+128$. In this example, the base is assumed to be 2.

Table 10.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000. This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

The final portion of the word (23 bits in this case) is the **significand**.[4]

Any floating-point number can be expressed in many ways.

The following are equivalent, where the significand is expressed in binary form:

$$0.110 \times 2^5$$
$$110 \times 2^2$$
$$0.0110 \times 2^6$$

To simplify operations on floating-point numbers, it is typically required that they be normalized. A **normal number** is one in which the most significant digit of the

[4]The term **mantissa**, sometimes used instead of *significand,* is considered obsolete. *Mantissa* also means "the fractional part of a logarithm," so is best avoided in this context.

significand is nonzero. For base 2 representation, a normal number is therefore one in which the most significant bit of the significand is one. As was mentioned, the typical convention is that there is one bit to the left of the radix point. Thus, a normal nonzero number is one in the form

$$\pm 1.\text{bbb} \ldots \text{ b} \times 2^{\pm \text{E}}$$

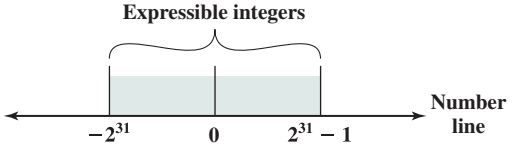
where b is either binary digit (0 or 1). Because the most significant bit is always one, it is unnecessary to store this bit; rather, it is implicit. Thus, the 23-bit field is used to store a 24-bit significand with a value in the half open interval [1, 2). Given a number that is not normal, the number may be normalized by shifting the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

Figure 10.18b gives some examples of numbers stored in this format. For each example, on the left is the binary number; in the center is the corresponding bit pattern; on the right is the decimal value. Note the following features:

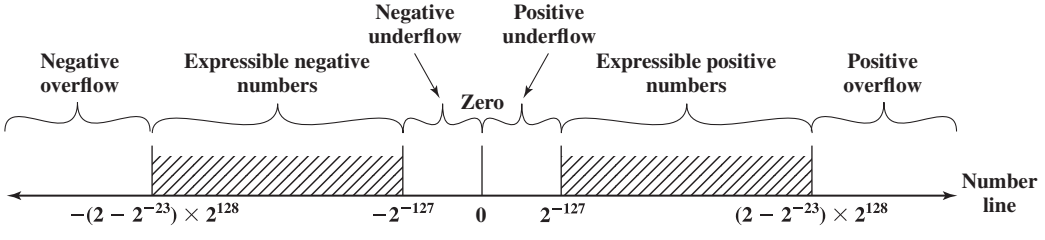
- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

For comparison, Figure 10.19 indicates the range of numbers that can be represented in a 32-bit word. Using twos complement integer representation, all of the integers from -2^{31} to $2^{31} - 1$ can be represented, for a total of 2^{32} different numbers. With the example floating-point format of Figure 10.18, the following ranges of numbers are possible:

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$



(a) Twos complement integers



(b) Floating-point numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$, called **negative overflow**
- Negative numbers greater than 2^{-127}, called **negative underflow**
- Zero
- Positive numbers less than 2^{-127}, called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$, called **positive overflow**

The representation as presented will not accommodate a value of 0. However, as we shall see, actual floating-point representations include a special bit pattern to designate zero. Overflow occurs when an arithmetic operation results in an absolute value greater than can be expressed with an exponent of 128 (e.g., $2^{120} \times 2^{100} = 2^{220}$). Underflow occurs when the fractional magnitude is too small (e.g., $2^{-120} \times 2^{-100} = 2^{-220}$). Underflow is a less serious problem because the result can generally be satisfactorily approximated by 0.

It is important to note that we are not representing more individual values with floating-point notation. The maximum number of different values that can be represented with 32 bits is still 2^{32}. What we have done is to spread those numbers out in two ranges, one positive and one negative. In practice, most floating-point numbers that one would wish to represent are represented only approximately. However, for moderate sized integers, the representation is exact.

Also, note that the numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away, as shown in Figure 10.20. This is one of the trade-offs of floating-point math: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

In the type of format depicted in Figure 10.18, there is a trade-off between range and precision. The example shows 8 bits devoted to the exponent and 23 to the significand. If we increase the number of bits in the exponent, we expand the range of expressible numbers. But because only a fixed number of different values can be expressed, we have reduced the density of those numbers and therefore the precision. The only way to increase both range and precision is to use more bits. Thus, most computers offer, at least, single-precision numbers and doubleprecision numbers. For example, a processor could support a single-precision format of 64 bits, and a double-precision format of 128 bits.

So there is a trade-off between the number of bits in the exponent and the number of bits in the significand. But it is even more complicated than that. The implied base of the exponent need not be 2. The IBM S/390 architecture, for example, uses a base of 16 [ANDE67b]. The format consists of a 7-bit exponent and a 24-bit significand.

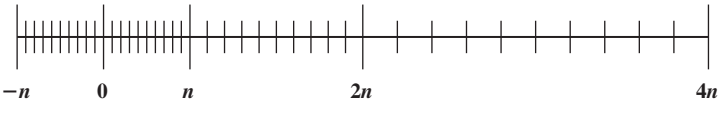


Figure 10.20 Density of Floating-Point Numbers

In the IBM base-16 format,

$$0.11010001 \times 2^{10100} = 0.11010001 \times 16^{101}$$

and the exponent is stored to represent 5 rather than 20.

The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits. But remember, we have not increased the number of different values that can be represented. Thus, for a fixed format, a larger exponent base gives a greater range at the expense of less precision.

IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754, adopted in 1985 and revised in 2008. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors. IEEE 754-2008 covers both binary and decimal floating-point representations. In this chapter, we deal only with binary representations.

IEEE 754-2008 defines the following different types of floating-point formats:

- **Arithmetic format:** All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- **Basic format:** This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- **Interchange format:** A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

The three basic binary formats have bit lengths of 32, 64, and 128 bits, with exponents of 8, 11, and 15 bits, respectively (Figure 10.21). Table 10.3 summarizes the characteristics of the three formats. The two basic decimal formats have bit lengths of 64 and 128 bits. All of the basic formats are also arithmetic format types (can be used for arithmetic operations) and interchange format types (platform independent).

Several other formats are specified in the standard. The binary16 format is only an interchange format and is intended for storage of values when higher precision is not required. The binary{k} format and the decimal{k} format are interchange formats with total length k bits and with defined lengths for the significand and exponent. The format must be a multiple of 32 bits; thus formats are defined for $k = 160, 192$, and so on. These two families of formats are also arithmetic formats.

In addition, the standard defines **extended precision formats**, which extend a supported basic format by providing additional bits in the exponent (extended range) and in the significand (extended precision). The exact format

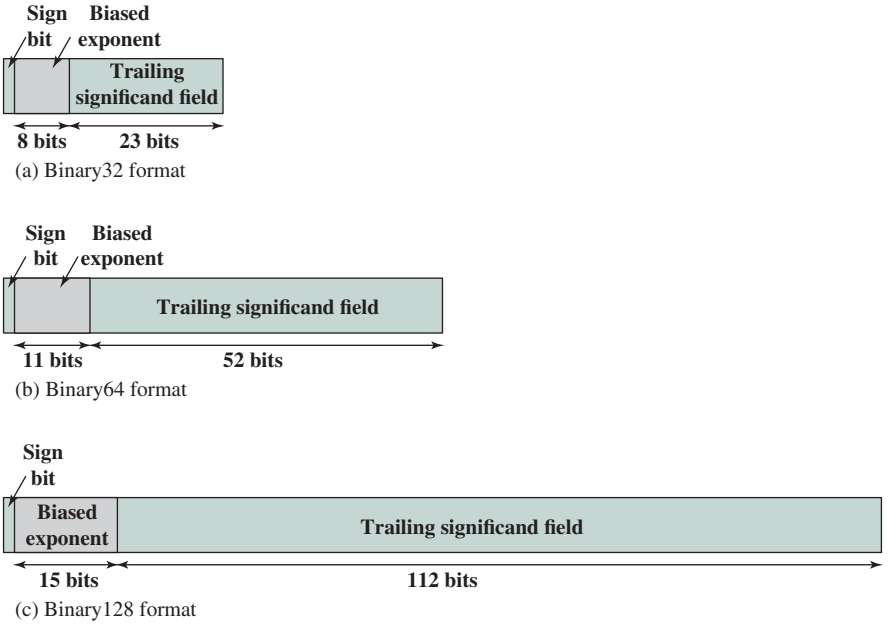


Figure 10.21 IEEE 754 Formats

is implementation dependent, but the standard places certain constraints on the length of the exponent and significand. These formats are arithmetic format types but not interchange format types. The extended formats are to be used for intermediate calculations. With their greater precision, the extended formats lessen the

Table 10.3 IEEE 754 Format Parameters

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Note: * Not including implied bit and not including sign bit.

chance of a final result that has been contaminated by excessive roundoff error; with their greater range, they also lessen the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format. An additional motivation for the extended format is that it affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision.

Finally, IEEE 754-2008 defines an **extendable precision format** as a format with a precision and range that are defined under user control. Again, these formats may be used for intermediate calculations, but the standard places no constraint or format or length.

Table 10.4 shows the relationship between defined formats and format types.

Not all bit patterns in the IEEE formats are interpreted in the usual way; instead, some bit patterns are used to represent special values. Table 10.5 indicates the values assigned to various bit patterns. The exponent values of all zeros (0 bits) and all ones (1 bits) define special values. The following classes of numbers are represented:

- For exponent values in the range of 1 through 254 for 32-bit format, 1 through 2046 for 64-bit format, and 1 through 16382, normal nonzero floating-point numbers are represented. The exponent is biased, so that the range of exponents is −126 through +127 for 32-bit format, and so on. A normal number requires a 1 bit to the left of the binary point; this bit is implied, giving an effective 24-bit, 53-bit, or 113-bit significand. Because one of the bits is implied, the corresponding field in the binary format is referred to as the **trailing significand field**.
- An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented.

Table 10.4 IEEE Formats

Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
extended precision	X		
extendable precision	X		

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers

(a) binary32 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 225	f	$2^{e-127}(1.f)$
negative normal nonzero	1	0 < e < 225	f	$-2^{e-127}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-126}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-126}(0.f)$

(b) binary64 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 2047	f	$2^{e-1023}(1.f)$
negative normal nonzero	1	0 < e < 2047	f	$-2^{e-1023}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-1022}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-1022}(0.f)$

(c) binary128 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2^{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2^{e-16383}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-16383}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-16383}(0.f)$

- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value ∞ and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a subnormal number. In this case, the bit to the left of the binary point is zero and the true exponent is -126 or -1022. The number is positive or negative depending on the sign bit.
- An exponent of all ones together with a nonzero fraction is given the value NaN, which means *Not a Number*, and is used to signal various exception conditions.

The significance of subnormal numbers and NaNs is discussed in Section 10.5.

10.5 FLOATING-POINT ARITHMETIC

Table 10.6 summarizes the basic operations for floating-point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

A floating-point operation may produce one of these conditions:

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we will discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we will explain.

Addition and Subtraction

In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros.
2. Align the significands.
3. Add or subtract the significands.
4. Normalize the result.

A typical flowchart is shown in Figure 10.22. A step-by-step narrative highlights the main functions required for floating-point addition and subtraction. We assume a format similar to those of Figure 10.21. For the addition or subtraction operation, the two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit, that bit must be made explicit for the operation.

Phase 1. Zero check: Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2. Significand alignment: The next phase is to manipulate the numbers so that the two exponents are equal.

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4.56 \times 10^0) = 127.56 \times 10^0$$

Alignment may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left. Because either operation may result in the loss of digits, it is the smaller number that is shifted; any digits that are lost are therefore of relatively small significance. The alignment

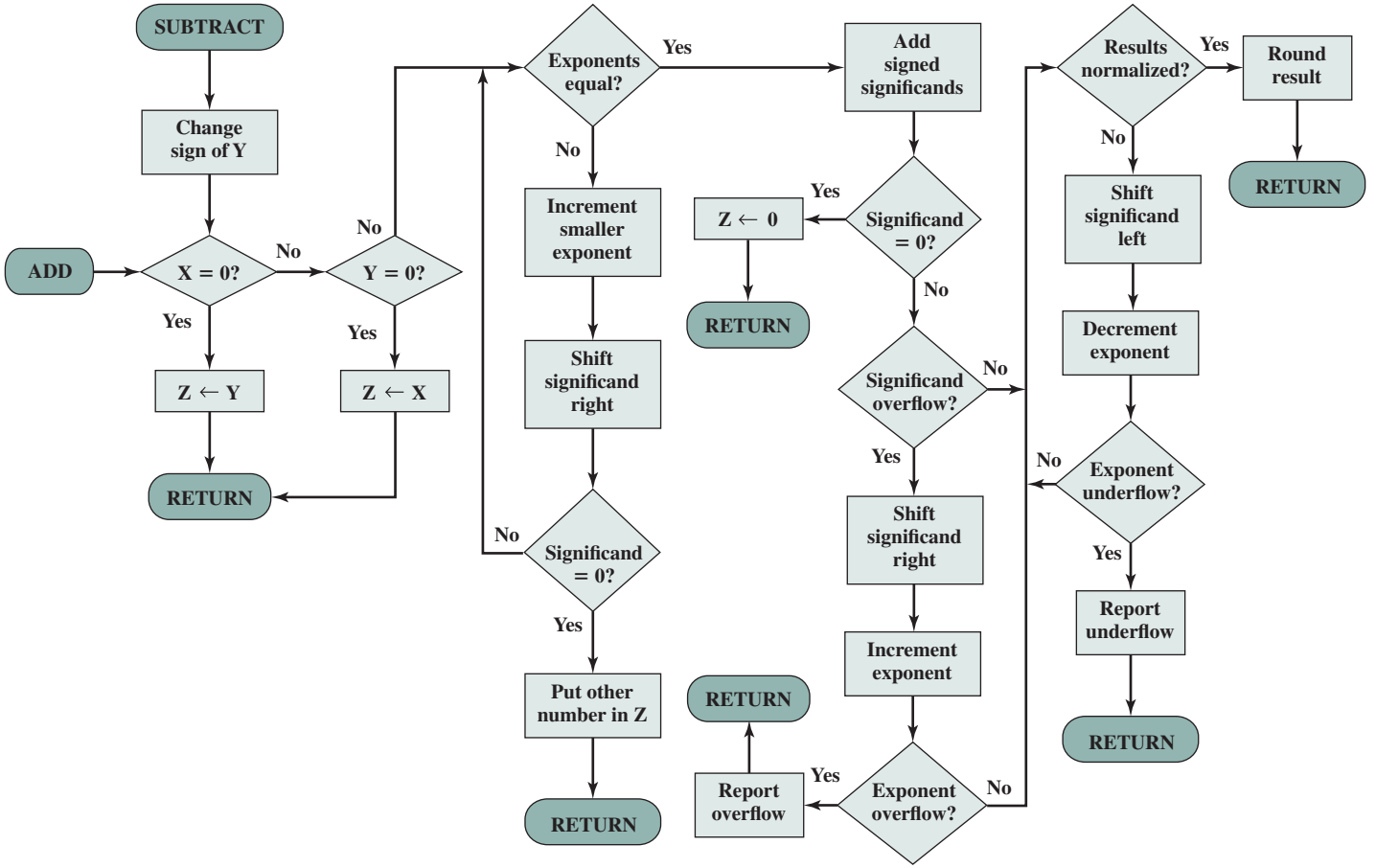


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two exponents are equal. (Note that if the implied base is 16, a shift of 1 digit is a shift of 4 bits.) If this process results in a 0 value for the significand, then the other number is reported as the result. Thus, if two numbers have exponents that differ significantly, the lesser number is lost.

Phase 3. Addition: Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4. Normalization: The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

Multiplication and Division

Floating-point multiplication and division are much simpler processes than addition and subtraction, as the following discussion indicates.

We first consider multiplication, illustrated in Figure 10.23. First, if either operand is 0, 0 is reported as the result. The next step is to add the exponents. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which would be reported, ending the algorithm.

If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed in the same way as for integers. In this case, we are dealing with a sign-magnitude representation, but the details are similar to those for twos complement representation. The product will be double the length of the multiplier and multiplicand. The extra bits will be lost during rounding.

After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

Finally, let us consider the flowchart for division depicted in Figure 10.24. Again, the first step is testing for 0. If the divisor is 0, an error report is issued, or the result is set to infinity, depending on the implementation. A dividend of 0 results in 0. Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back in. Tests are then made for exponent underflow or overflow.

The next step is to divide the significands. This is followed with the usual normalization and rounding.

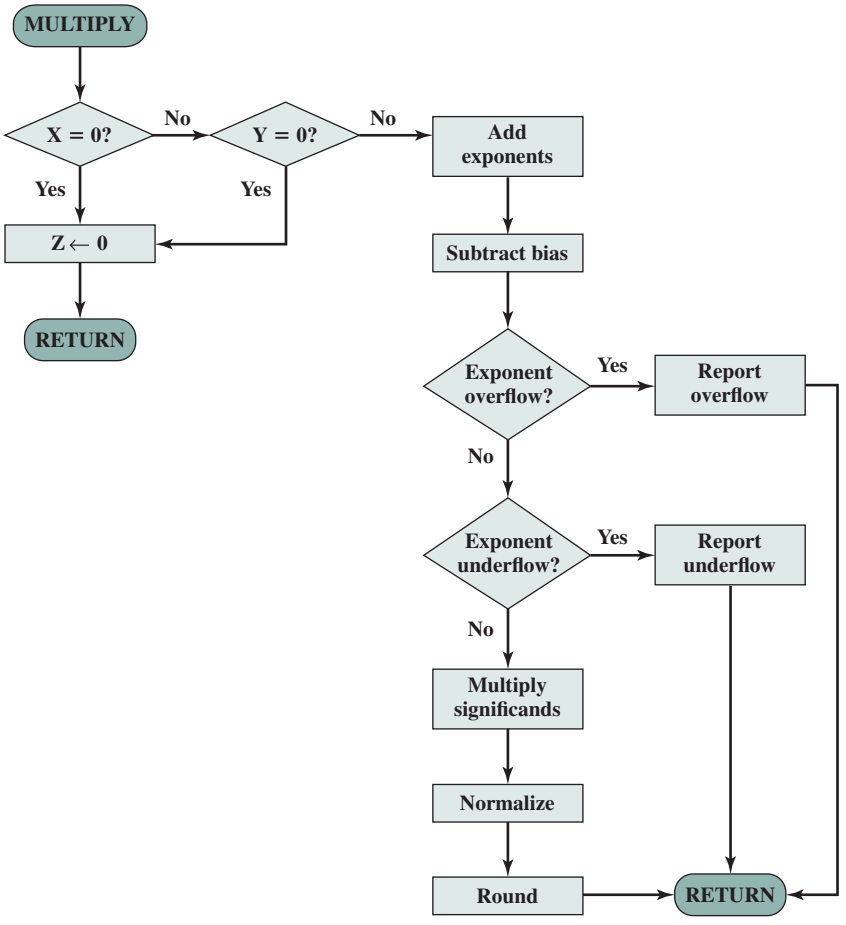


Figure 10.23 Floating-Point Multiplication ($Z \leftarrow X \pm Y$)

Precision Considerations

GUARD BITS We mentioned that, prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers. In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

The reason for the use of guard bits is illustrated in Figure 10.25. Consider numbers in the IEEE format, which has a 24-bit significand, including an implied 1 bit to the left of the binary point. Two numbers that are very close in value are $x = 1.00 \cdots 00 \times 2^1$ and $y = 1.11 \cdots 11 \times 2^0$. If the smaller number is to be subtracted from the larger, it must be shifted right 1 bit to align the exponents. This is shown in Figure 10.25a. In the process, y loses 1 bit of significance; the result is 2^{-22}. The same operation is repeated in

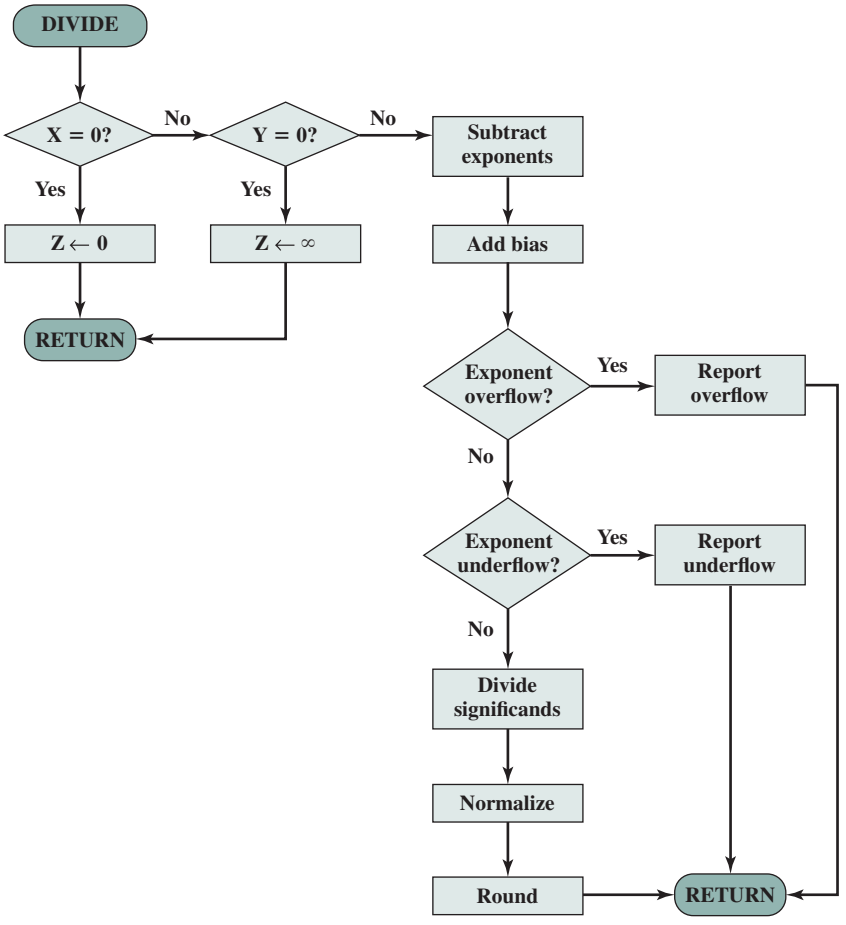


Figure 10.24 Floating-Point Division (Z ← X/Y)

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= 0.111.....11 \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= .0FFFFF \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= 0.111.....11\ 1000 \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= .0FFFFF\ F0 \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits

part (b) with the addition of guard bits. Now the least significant bit is not lost due to alignment, and the result is 2^{-23}, a difference of a factor of 2 from the previous answer. When the radix is 16, the loss of precision can be greater. As Figures 10.25c and (d) show, the difference can be a factor of 16.

ROUNDING Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result. This process is called **rounding**.

A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward $+\infty$:** The result is rounded up toward plus infinity.
- **Round toward $-\infty$:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Let us consider each of these policies in turn. **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.

If the extra bits, beyond the 23 bits that can be stored, are 10010, then the extra bits amount to more than one-half of the last representable bit position. In this case, the correct answer is to add binary 1 to the last representable bit, rounding up to the next representable number. Now consider that the extra bits are 01111. In this case, the extra bits amount to less than one-half of the last representable bit position. The correct answer is simply to drop the extra bits (truncate), which has the effect of rounding down to the next representable number.

The standard also addresses the special case of extra bits of the form 10000.... Here the result is exactly halfway between the two possible representable values. One possible technique here would be to always truncate, as this would be the simplest operation. However, the difficulty with this simple approach is that it introduces a small but cumulative bias into a sequence of computations. What is required is an unbiased method of rounding. One possible approach would be to round up or down on the basis of a random number so that, on average, the result would be unbiased. The argument against this approach is that it does not produce predictable, deterministic results. The approach taken by the IEEE standard is to force the result to be even: If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0.

The next two options, **rounding to plus** and **minus infinity**, are useful in implementing a technique known as interval arithmetic. Interval arithmetic provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result. The two values correspond to the lower and upper endpoints of an interval that contains the true result. The width of the interval, which is the difference between the upper and lower endpoints, indicates the accuracy of the result. If the endpoints of an interval are not representable, then the interval endpoints are rounded down and up, respectively. Although the width of the interval may vary according to implementation, many algorithms have been designed to produce narrow intervals. If the range between the upper and lower bounds is sufficiently narrow, then a sufficiently accurate result has been obtained. If not, at least we know this and can perform additional analysis.

The final technique specified in the standard is **round toward zero**. This is, in fact, simple truncation: The extra bits are ignored. This is certainly the simplest technique. However, the result is that the magnitude of the truncated value is always less than or equal to the more precise original value, introducing a consistent bias toward zero in the operation. This is a serious bias because it affects every operation for which there are nonzero extra bits.

IEEE Standard for Binary Floating-Point Arithmetic

IEEE 754 goes beyond the simple definition of a format to lay down specific practices and procedures so that floating-point arithmetic produces uniform, predictable results independent of the hardware platform. One aspect of this has already been discussed, namely rounding. This subsection looks at three other topics: infinity, NaNs, and subnormal numbers.

INFINITY Infinity arithmetic is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

With the exception of the special cases discussed subsequently, any arithmetic operation involving infinity yields the obvious result.

For example:

$$\begin{array}{ll} 5 + (+\infty) = +\infty & 5 \div (+\infty) = +0 \\ 5 - (+\infty) = -\infty & (+\infty) + (+\infty) = +\infty \\ 5 + (-\infty) = -\infty & (-\infty) + (-\infty) = -\infty \\ 5 - (-\infty) = +\infty & (-\infty) - (+\infty) = -\infty \\ 5 \times (+\infty) = +\infty & (+\infty) - (-\infty) = +\infty \end{array}$$

QUIET AND SIGNALING NANS A NaN is a symbolic entity encoded in floating-point format, of which there are two types: signaling and quiet. A signaling NaN signals an invalid operation exception whenever it appears as an operand. Signaling

Table 10.7 Operations that Produce a Quiet NaN

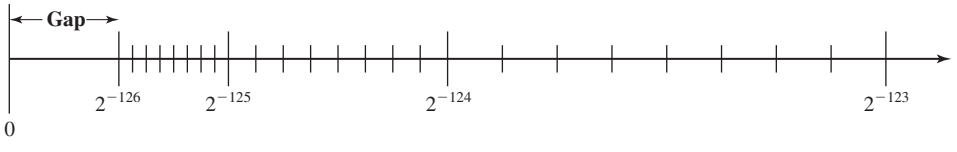
Operation	Quiet NaN Produced By
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$, where $x < 0$

NaNs afford values for uninitialized variables and arithmetic-like enhancements that are not the subject of the standard. A quiet NaN propagates through almost every arithmetic operation without signaling an exception. Table 10.7 indicates operations that will produce a quiet NaN.

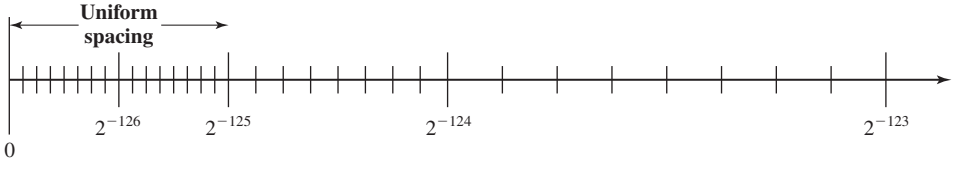
Note that both types of NaNs have the same general format (Table 10.4): an exponent of all ones and a nonzero fraction. The actual bit pattern of the nonzero fraction is implementation dependent; the fraction values can be used to distinguish quiet NaNs from signaling NaNs and to specify particular exception conditions.

SUBNORMAL NUMBERS Subnormal numbers are included in IEEE 754 to handle cases of exponent underflow. When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is subnormalized by right shifting the fraction and incrementing the exponent for each shift until the exponent is within a representable range.

Figure 10.26 illustrates the effect of including subnormal numbers. The representable numbers can be grouped into intervals of the form $[2^n, 2^{n+1}]$. Within



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

Figure 10.26 The Effect of IEEE 754 Subnormal Numbers

each such interval, the exponent portion of the number remains constant while the fraction varies, producing a uniform spacing of representable numbers within the interval. As we get closer to zero, each successive interval is half the width of the preceding interval but contains the same number of representable numbers. Hence the density of representable numbers increases as we approach zero. However, if only normal numbers are used, there is a gap between the smallest normal number and 0. In the case of the 32-bit IEEE 754 format, there are 2^{23} representable numbers in each interval, and the smallest representable positive number is 2^{-126}. With the addition of subnormal numbers, an additional $2^{23} - 1$ numbers are uniformly added between 0 and 2^{-126}.

The use of subnormal numbers is referred to as *gradual underflow* [COON81]. Without subnormal numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills in that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normal numbers.

10.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

arithmetic and logic unit (ALU)	minuend	radix point
arithmetic shift	multiplicand	range extension
base	multiplier	remainder
biased representation	negative overflow	rounding
dividend	negative underflow	sign bit
divisor	normal number	sign-magnitude representation
exponent	ones complement representation	significand
exponent overflow	overflow	significand overflow
exponent underflow	partial product	significand underflow
fixed-point representation	positive overflow	subnormal number
floating-point representation	positive underflow	subtrahend
guard bits	product	twos complement representation
mantissa	quotient	

Review Questions

10.1 Briefly explain the following representations: sign magnitude, twos complement, biased.

10.2 Explain how to determine if a number is negative in the following representations: sign magnitude, twos complement, biased.

10.3 What is the sign-extension rule for twos complement numbers?

10.4 How can you form the negation of an integer in twos complement representation?

10.5 In general terms, when does the twos complement operation on an n-bit integer produce the same integer?

10.6 What is the difference between the twos complement representation of a number and the twos complement of a number?

10.7 If we treat two twos complement numbers as unsigned integers for purposes of addition, the result is correct if interpreted as a twos complement number. This is not true for multiplication. Why?

10.8 What are the four essential elements of a number in floating-point notation?

10.9 What is the benefit of using biased representation for the exponent portion of a floating-point number?

10.10 What are the differences among positive overflow, exponent overflow, and significand overflow?

10.11 What are the basic elements of floating-point addition and subtraction?

10.12 Give a reason for the use of guard bits.

10.13 List four alternative methods of rounding the result of a floating-point operation.

Problems

10.1 Represent the following decimal numbers in both binary sign/magnitude and twos complement using 16 bits: + 512; − 29.

10.2 Represent the following twos complement values in decimal: 1101011; 0101101.

10.3 Another representation of binary integers that is sometimes encountered is **ones complement**. Positive integers are represented in the same way as sign magnitude. A negative integer is represented by taking the Boolean complement of each bit of the corresponding positive number.

a. Provide a definition of ones complement numbers using a weighted sum of bits, similar to Equations (10.1) and (10.2).

b. What is the range of numbers that can be represented in ones complement?

c. Define an algorithm for performing addition in ones complement arithmetic.

Note: Ones complement arithmetic disappeared from hardware in the 1960s, but still survives checksum calculations for the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

10.4 Add columns to Table 10.1 for sign magnitude and ones complement.

10.5 Consider the following operation on a binary word. Start with the least significant bit. Copy all bits that are 0 until the first bit is reached and copy that bit, too. Then take the complement of each bit thereafter. What is the result?

10.6 In Section 10.3, the twos complement operation is defined as follows. To find the twos complement of X, take the Boolean complement of each bit of X, and then add 1.

a. Show that the following is an equivalent definition. For an n-bit integer X, the twos complement of X is formed by treating X as an unsigned integer and calculating $(2_n - X)$.

b. Demonstrate that Figure 10.5 can be used to support graphically the claim in part (a), by showing how a clockwise movement is used to achieve subtraction.

10.7 The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Find the tens complement of the decimal number 13,250.

10.8 Calculate (72,530 − 13,250) using tens complement arithmetic. Assume rules similar to those for twos complement arithmetic.

10.9 Consider the twos complement addition of two n-bit numbers:

$$z_{n-1}z_{n-2} \cdots z_0 = x_{n-1}x_{n-2} \cdots x_0 + y_{n-1}y_{n-2} \cdots y_0$$

Assume that bitwise addition is performed with a carry bit c_i generated by the addition of x_i, y_i, and c_{i-1}. Let ν be a binary variable indicating overflow when $\nu = 1$. Fill in the values in the table.

Input	x_{n-1}	0	0	0	0	1	1	1	1
	y_{n-1}	0	0	1	1	0	0	1	1
	c_{n-2}	0	1	0	1	0	1	0	1
Output	z_{n-1}								
	v								

10.10 Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

a. $6 + 13$ **b.** $-6 + 13$ **c.** $6 - 13$ **d.** $-6 - 13$

10.11 Find the following differences using twos complement arithmetic:

a. $\begin{array}{r} 111000 \\ -110011 \end{array}$ **b.** $\begin{array}{r} 11001100 \\ -101110 \end{array}$ **c.** $\begin{array}{r} 111100001111 \\ -110011110011 \end{array}$ **d.** $\begin{array}{r} 11000011 \\ -11101000 \end{array}$

10.12 Is the following a valid alternative definition of overflow in twos complement arithmetic?

If the exclusive-OR of the carry bits into and out of the leftmost column is 1, then there is an overflow condition. Otherwise, there is not.

10.13 Compare Figures 10.9 and 10.12. Why is the C bit not used in the latter?

10.14 Given $x = 0101$ and $y = 1010$ in twos complement notation (i.e., $x = 5, y = -6$), compute the product $p = x \times y$ with Booth's algorithm.

10.15 Use the Booth algorithm to multiply 23 (multiplicand) by 29 (multiplier), where each number is represented using 6 bits.

10.16 Prove that the multiplication of two n-digit numbers in base B gives a product of no more than $2n$ digits.

10.17 Verify the validity of the unsigned binary division algorithm of Figure 10.16 by showing the steps involved in calculating the division depicted in Figure 10.15. Use a presentation similar to that of Figure 10.17.

10.18 The twos complement integer division algorithm described in Section 10.3 is known as the restoring method because the value in the A register must be restored following unsuccessful subtraction. A slightly more complex approach, known as nonrestoring, avoids the unnecessary subtraction and addition. Propose an algorithm for this latter approach.

10.19 Under computer integer arithmetic, the quotient J/K of two integers J and K is less than or equal to the usual quotient. True or false?

10.20 Divide -145 by 13 in binary twos complement notation, using 12-bit words. Use the algorithm described in Section 10.3.

10.21 **a.** Consider a fixed-point representation using decimal digits, in which the implied radix point can be in any position (to the right of the least significant digit, to the right of the most significant digit, and so on). How many decimal digits are needed to represent the approximations of both Planck's constant (6.63×10^{-27}) and Avogadro's number (6.02×10^{23})? The implied radix point must be in the same position for both numbers.

b. Now consider a decimal floating-point format with the exponent stored in a biased representation with a bias of 50. A normalized representation is assumed. How many decimal digits are needed to represent these constants in this floating-point format?

10.22 Assume that the exponent e is constrained to lie in the range $0 \leq e \leq X$, with a bias of q, that the base is b, and that the significand is p digits in length.

a. What are the largest and smallest positive values that can be written?

b. What are the largest and smallest positive values that can be written as normalized floating-point numbers?

10.23 Express the following numbers in IEEE 32-bit floating-point format:

a. −5 **b.** −6 **c.** −1.5 **d.** 384 **e.** 1/16 **f.** −1/32

10.24 The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

a. 1 10000011 11000000000000000000000
b. 0 01111110 10100000000000000000000
c. 0 10000000 00000000000000000000000

10.25 Consider a reduced 7-bit IEEE floating-point format, with 3 bits for the exponent and 3 bits for the significand. List all 127 values.

10.26 Express the following numbers in IBM's 32-bit floating-point format, which uses a 7-bit exponent with an implied base of 16 and an exponent bias of 64 (40 hexadecimal). A normalized floating-point number requires that the leftmost hexadecimal digit be nonzero; the implied radix point is to the left of that digit.

a. 1.0	**c.** 1/64	**e.** − 15.0	**g.** 7.2×10^{75}
b. 0.5	**d.** 0.0	**f.** 5.4×10^{-79}	**h.** 65,535

10.27 Let 5BCA0000 be a floating-point number in IBM format, expressed in hexadecimal. What is the decimal value of the number?

10.28 What would be the bias value for

a. A base-2 exponent (B = 2) in a 6-bit field?
b. A base-8 exponent (B = 8) in a 7-bit field?

10.29 Draw a number line similar to that in Figure 10.19b for the floating-point format of Figure 10.21b.

10.30 Consider a floating-point format with 8 bits for the biased exponent and 23 bits for the significand. Show the bit pattern for the following numbers in this format:

a. −720 **b.** 0.645

10.31 The text mentions that a 32-bit format can represent a maximum of 2^{32} different numbers. How many different numbers can be represented in the IEEE 32-bit format? Explain.

10.32 Any floating-point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A' is the stored value approximating the real value A, then the relative error, r, is expressed as

$$r = \frac{A - A'}{A}$$

Represent the decimal quantity + 0.4 in the following floating-point format: base = 2; exponent: biased, 4 bits; significand, 7 bits. What is the relative error?

10.33 If $A = 1.427$, find the relative error if A is truncated to 1.42 and if it is rounded to 1.43.

10.34 When people speak about inaccuracy in floating-point arithmetic, they often ascribe errors to cancellation that occurs during the subtraction of nearly equal quantities. But when X and Y are approximately equal, the difference $X - Y$ is obtained exactly, with no error. What do these people really mean?

10.35 Numerical values A and B are stored in the computer as approximations A' and B' Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors in the factors.

10.36 One of the most serious errors in computer calculations occurs when two nearly equal numbers are subtracted. Consider $A = 0.22288$ and $B = 0.22211$. The computer truncates all values to four decimal digits. Thus $A' = 0.2228$ and $B' = 0.2221$.

a. What are the relative errors for A' and B'?
b. What is the relative error for $C' = A' - B'$?

10.37 To get some feel for the effects of denormalization and gradual underflow, consider a decimal system that provides 6 decimal digits for the significand and for which the smallest normalized number is 10^{-99}. A normalized number has one nonzero decimal digit to the left of the decimal point. Perform the following calculations and denormalize the results. Comment on the results.

a. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-43})$
b. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-60})$
c. $(5.67834 \times 10^{-97}) - (5.67812 \times 10^{-97})$

10.38 Show how the following floating-point additions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $5.566 \times 10^{2} + 7.777 \times 10^{2}$
b. $3.344 \times 10^{1} + 8.877 \times 10^{-2}$

10.39 Show how the following floating-point subtractions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $7.744 \times 10^{-3} - 6.666 \times 10^{-3}$ **b.** $8.844 \times 10^{-3} - 2.233 \times 10^{-1}$

10.40 Show how the following floating-point calculations are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $(2.255 \times 10^{1}) \times (1.234 \times 10^{0})$ **b.** $(8.833 \times 10^{2}) \div (5.555 \times 10^{4})$

chapter

9

ARITHMETIC

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Adder and subtractor circuits
- High-speed adders based on carry-lookahead logic circuits
- The Booth algorithm for multiplication of signed numbers
- High-speed multipliers based on carry-save addition
- Logic circuits for division
- Arithmetic operations on floating-point numbers conforming to the IEEE standard

Addition and subtraction of two numbers are basic operations at the machine-instruction level in all computers. These operations, as well as other arithmetic and logic operations, are implemented in the arithmetic and logic unit (ALU) of the processor. In this chapter, we present the logic circuits used to implement arithmetic operations. The time needed to perform addition or subtraction affects the processor's performance. Multiply and divide operations, which require more complex circuitry than either addition or subtraction operations, also affect performance. We present some of the techniques used in modern computers to perform arithmetic operations at high speed. Operations on floating-point numbers are also described.

In Section 1.4 of Chapter 1, we described the representation of signed binary numbers, and showed that 2's-complement is the best representation from the standpoint of performing addition and subtraction operations. The examples in Figure 1.6 show that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bits. In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement. The first two sections of this chapter present logic circuits for addition and subtraction.

9.1 Addition and Subtraction of Signed Numbers

Figure 9.1 shows the truth table for the sum and carry-out functions for adding equally weighted bits x_i and y_i in two numbers X and Y. The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use c_i to represent the carry-in to stage i, which is the same as the carry-out from stage $(i-1)$.

The logic expression for s_i in Figure 9.1 can be implemented with a 3-input XOR gate, used in Figure 9.2*a* as part of the logic required for a single stage of binary addition. The carry-out function, c_{i+1}, is implemented with an AND-OR circuit, as shown. A convenient symbol for the complete circuit for a single stage of addition, called a *full adder* (FA), is also shown in the figure.

A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, as shown in Figure 9.2*b*. 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 in Figure 9.2*c*.

9.1.1 Addition/Subtraction Logic Unit

The n-bit adder in Figure 9.2*b* 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 was discussed in Section 1.4. It occurs when the signs of the two

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

Example:

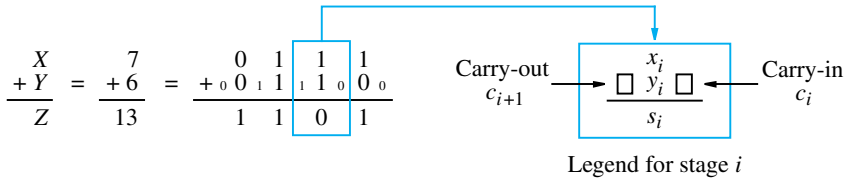


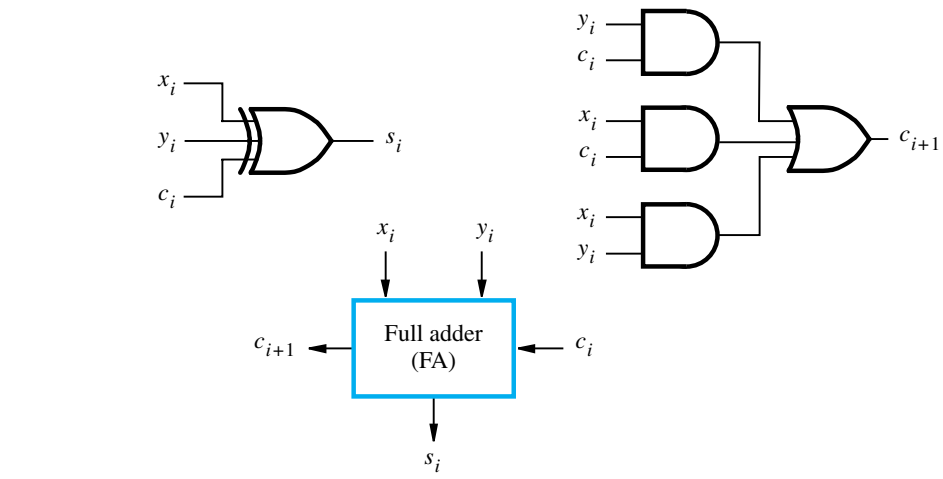
Figure 9.1 Logic specification for a stage of binary addition.

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

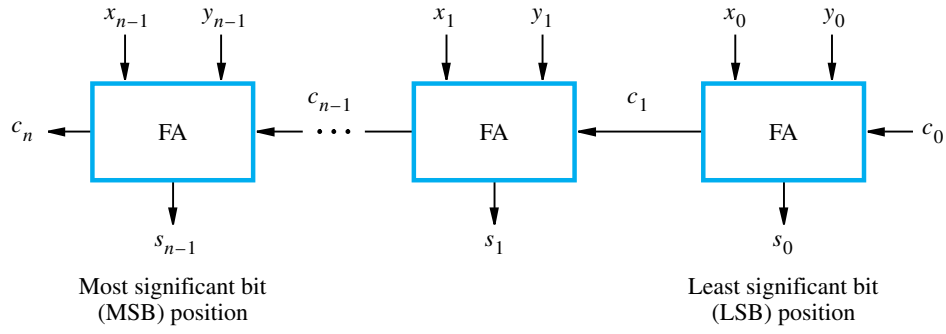
$$\text{Overflow} = x_{n-1} y_{n-1} \overline{s}_{n-1} + \overline{x}_{n-1} \overline{y}_{n-1} s_{n-1}$$

It can also be shown that overflow occurs when the carry bits c_n and c_{n-1} are different. (See Problem 9.5.) Therefore, a simpler circuit for detecting overflow can be obtained by implementing the expression $c_n \oplus c_{n-1}$ with an XOR gate.

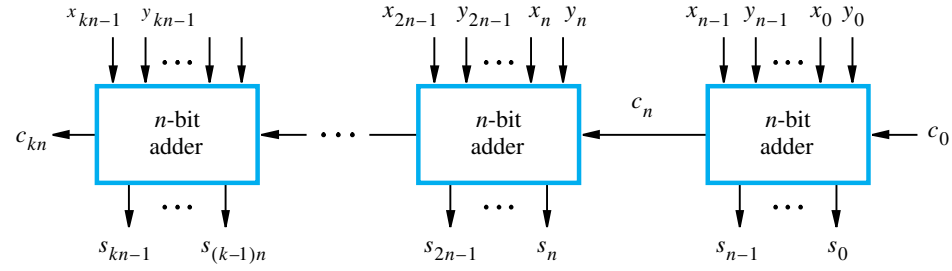
In order to perform the subtraction operation $X - Y$ on 2's-complement numbers X and Y, we form the 2's-complement of Y and add it to X. The logic circuit shown in Figure 9.3 can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, c_0, of 0. When the Add/Sub control line is set to 1, the Y number is 1's-complemented (that is, bit-complemented) by the XOR gates and c_0 is set to 1 to complete the 2's-complementation of Y. Recall that 2's-complementing a negative number is done in exactly the same manner as for a positive number. An XOR gate can be added to Figure 9.3 to detect the overflow condition $c_n \oplus c_{n-1}$.



(a) Logic for a single stage



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



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

Figure 9.2 Logic for addition of binary numbers.

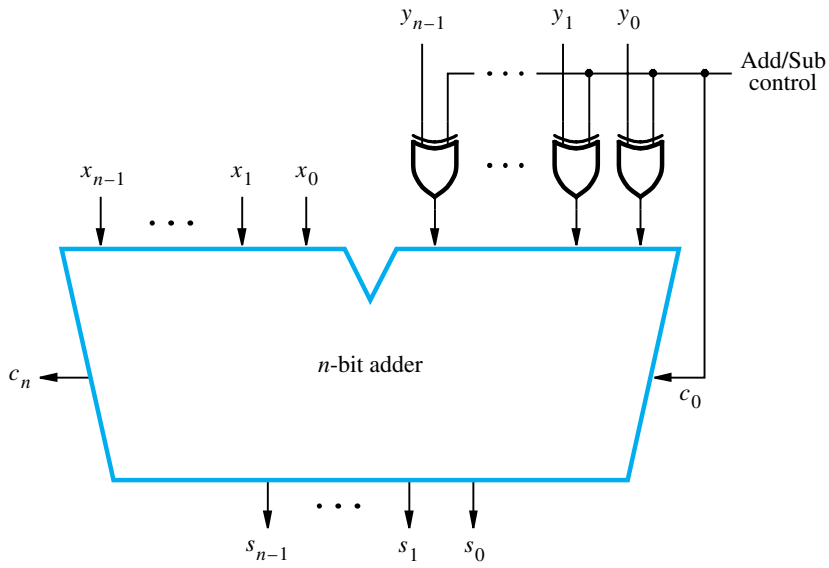


Figure 9.3 Binary addition/subtraction logic circuit.

9.2 DESIGN OF FAST ADDERS

If an n-bit ripple-carry adder is used in the addition/subtraction circuit of Figure 9.3, it may have too much delay in developing its outputs, s_0 through s_{n-1} and c_n. Whether or not the delay incurred is acceptable can be decided only in the context of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology used in fabricating the network and on the number of gates in the paths from inputs to outputs. 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.

Using the implementation indicated in Figure 9.2*a*, c_{n-1} is available in $2(n-1)$ gate delays, and s_{n-1} is correct one XOR gate delay later. The final carry-out, c_n, is available after $2n$ gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure 9.3, 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.

Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology. The second approach is to use a logic gate network called a carry-lookahead network, which is described in the next section.

9.2.1 CARRY-LOOKAHEAD ADDITION

A fast adder circuit must speed up the generation of the carry signals. The logic expressions for s_i (sum) and c_{i+1} (carry-out) of stage i (see Figure 9.1) are

$$s_i = x_i \oplus y_i \oplus c_i$$

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

$$c_{i+1} = G_i + P_i c_i$$

where

$$G_i = x_i y_i \qquad \text{and} \qquad P_i = x_i + y_i$$

The expressions G_i and P_i are called the *generate* and *propagate* functions for stage 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 in Figure 9.4*a* 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

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0 \qquad (9.1)$$

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.

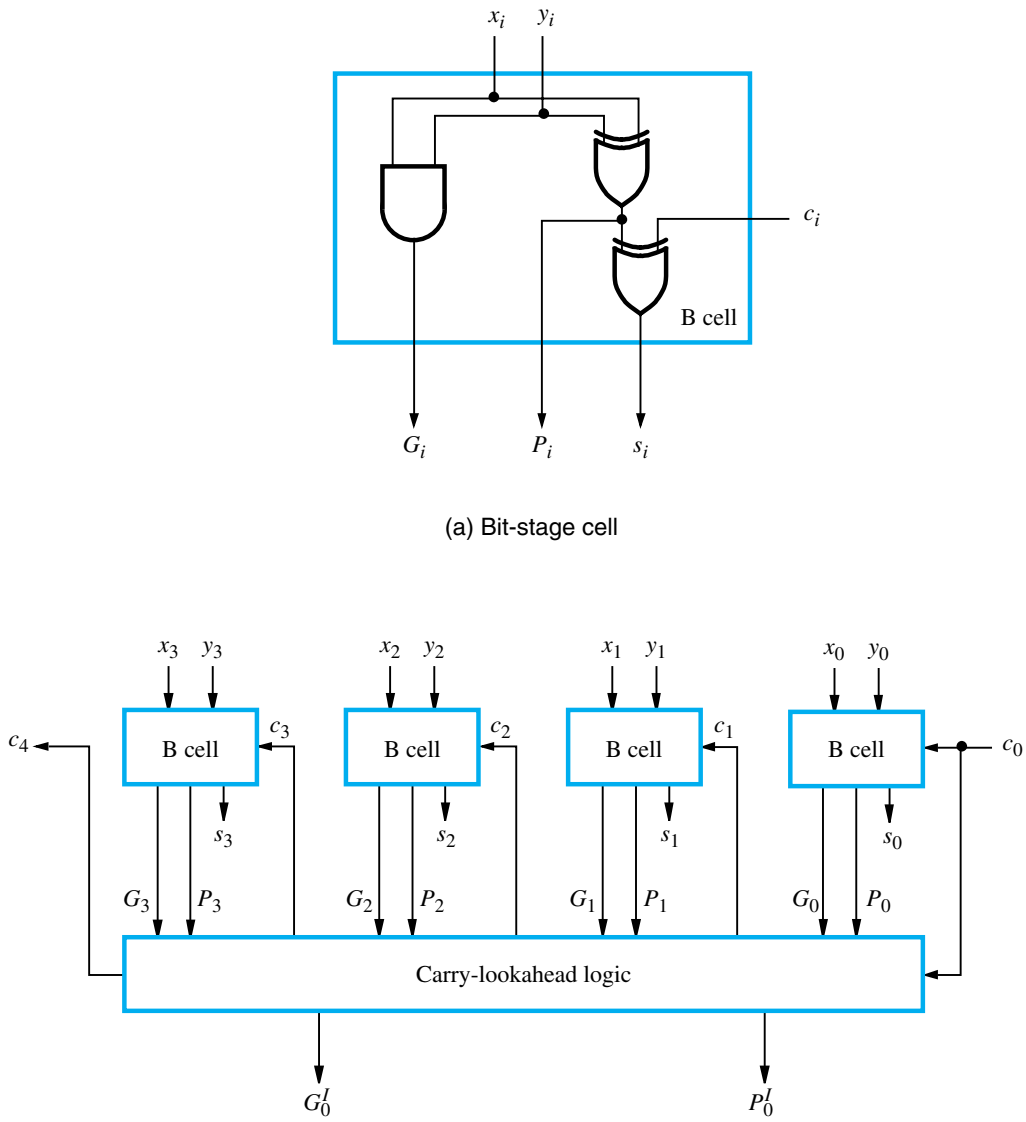


Figure 9.4 A 4-bit carry-lookahead adder.

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

$$c_1 = G_0 + P_0c_0$$
$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$
$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$
$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

The complete 4-bit adder is shown in Figure 9.4*b*. The carries are produced in the block labeled carry-lookahead logic. An adder implemented in this form is called a *carry-lookahead adder*. Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, a 4-bit ripple-carry adder requires 7 gate delays for s_3 and 8 gate delays for c_4.

If we try to extend the carry-lookahead adder design of Figure 9.4*b* for longer operands, we encounter the problem of gate fan-in constraints. From Expression 9.1, we see that the last AND gate and the OR gate require a fan-in of $i + 2$ in generating c_{i+1}. A fan-in of 5 is required for c_4 in the 4-bit adder. This is about the limit for practical gates. So the adder design shown in Figure 9.4*b* cannot be extended easily for longer operands. However, it is possible to build longer adders by cascading a number of 4-bit adders, as shown in Figure 9.2*c*.

Eight, 4-bit, carry-lookahead adders can be connected as in Figure 9.2*c* to form a 32-bit adder. The delays in generating sum bits s_{31}, s_{30}, s_{29}, s_{28}, and carry bit c_{32} in the high-order 4-bit adder in this cascade are calculated as follows. The carry-out c_4 from the low-order adder is available 3 gate delays after the input operands X, Y, and c_0 are applied to the 32-bit adder. Then, c_8 is available at the output of the second adder after a further 2 gate delays, c_{12} is available after a further 2 gate delays, and so on. Finally, c_{28}, the carry-in to the high-order 4-bit adder, is available after a total of $(6 \times 2) + 3 = 15$ gate delays. Then, c_{32} and all carries inside the high-order adder are available after a further 2 gate delays, and all 4 sum bits are available after 1 more gate delay, for a total of 18 gate delays. This should be compared to total delays of 63 and 64 for s_{31} and c_{32} if a ripple-carry adder is used.

In the next section, we show how it is possible to improve upon the cascade structure just discussed, leading to further reduction in adder delay. The key idea is to generate the carries c_4, c_8, ... in parallel, similar to the way that c_1, c_2, c_3, and c_4, are generated in parallel in the 4-bit carry-lookahead adder.

Higher-Level Generate and Propagate Functions

In the 32-bit adder just discussed, the carries $c_4, c_8, c_{12}, \ldots$ ripple through the 4-bit adder blocks with two gate delays per block, analogous to the way that individual carries ripple through each bit stage in a ripple-carry adder. It is possible to use the lookahead approach to develop the carries c_4, c_8, $c_{12}, \ldots$ in parallel by using higher-level block generate and propagate functions.

Figure 9.5 shows a 16-bit adder built from four 4-bit adder blocks. These blocks provide new output functions defined as G_k^I and P_k^I, where $k = 0$ for the first 4-bit block, $k = 1$ for the second 4-bit block, and so on, as shown in Figures 9.4*b* and 9.5. In the first block,

$$P_0^I = P_3P_2P_1P_0$$

and

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

The first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry. The second-level G_k^I and P_k^I functions determine whether block k generates or propagates a carry. With these new functions available, it is not necessary to wait for carries to ripple through the 4-bit blocks. Carry c_{16} is formed by one of the carry-lookahead

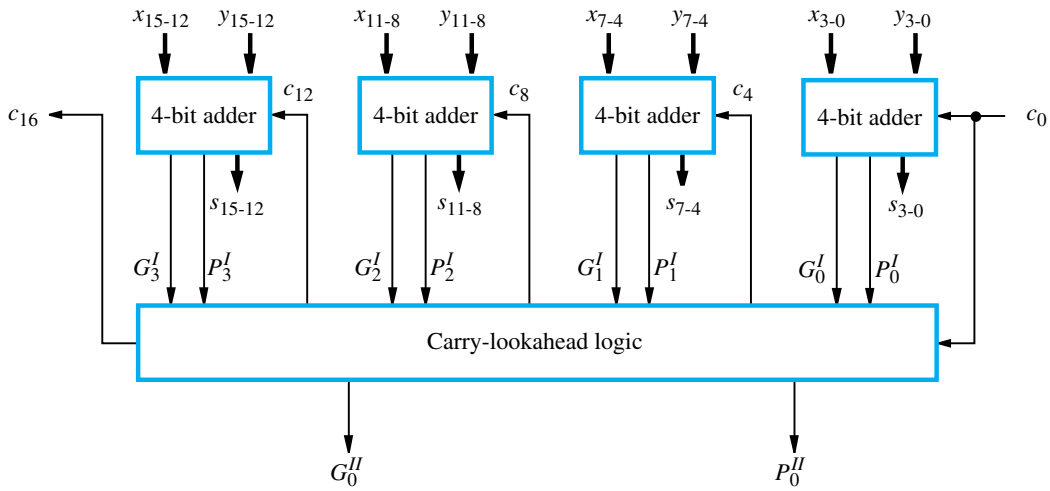


Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

circuits in Figure 9.5 as

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

The input carries to the 4-bit blocks are formed in parallel by similar shorter expressions. Expressions for c_{16}, c_{12}, c_8, and c_4, are identical in form to the expressions for c_4, c_3, c_2, and c_1, respectively, implemented in the carry-lookahead circuits in Figure 9.4*b*. Only the variable names are different. Therefore, the structure of the carry-lookahead circuits in Figure 9.5 is identical to the carry-lookahead circuits in Figure 9.4*b*. However, the carries c_4, c_8, c_{12}, and c_{16}, generated internally by the 4-bit adder blocks, are not needed in Figure 9.5 because they are generated by the higher-level carry-lookahead circuits.

Now, consider the delay in producing outputs from the 16-bit carry-lookahead adder. The delay in developing the carries produced by the carry-lookahead circuits is two gate delays more than the delay needed to develop the G_k^I and P_k^I functions. The latter require two gate delays and one gate delay, respectively, after the generation of G_i and P_i. Therefore, all carries produced by the carry-lookahead circuits are available 5 gate delays after X, Y, and c_0 are applied as inputs. The carry c_{15} is generated inside the high-order 4-bit block in Figure 9.5 in two gate delays after c_{12}, followed by s_{15} in one further gate delay. Therefore, s_{15} is available after 8 gate delays. If a 16-bit adder is built by cascading 4-bit carry-lookahead adder blocks, the delays in developing c_{16} and s_{15} are 9 and 10 gate delays, respectively, as compared to 5 and 8 gate delays for the configuration in Figure 9.5.

Two 16-bit adder blocks can be cascaded to implement a 32-bit adder. In this configuration, the output c_{16} from the low-order block is the carry input to the high-order block. The delay is much lower than the delay through the 32-bit adder that we discussed earlier, which was built by cascading eight 4-bit adders. In that configuration, recall that s_{31} is available after 18 gate delays and c_{32} is available after 17 gate delays. The delay analysis

for the cascade of two 16-bit adders is as follows. The carry c_{16} out of the low-order block is available after 5 gate delays, as calculated above. Then, both c_{28} and c_{32} are available in the high-order block after a further 2 gate delays, and c_{31} is available 2 gate delays after c_{28}. Therefore, c_{31} is available after a total of 9 gate delays, and s_{31} is available in 10 gate delays. Recapitulating, s_{31} and c_{32} are available after 10 and 7 gate delays, respectively, compared to 18 and 17 gate delays for the same outputs if the 32-bit adder is built from a cascade of eight 4-bit adders.

The same reasoning used in developing second-level G_k^I and P_k^I functions from first-level G_i and P_i functions can be used to develop third-level G_k^{II} and P_k^{II} functions from G_k^I and P_k^I functions. Two such third-level functions are shown as outputs from the carry-lookahead logic in Figure 9.5. A 64-bit adder can be built from four of the 16-bit adders shown in Figure 9.5, along with additional carry-lookahead logic circuits that produce carries c_{16}, c_{32}, c_{48}, and c_{64}. Delay through this adder can be shown to be 12 gate delays for s_{63} and 7 gate delays for c_{64}, using an extension of the reasoning used above for the 16-bit adder. (See Problem 9.7.)

9.3 MULTIPLICATION OF UNSIGNED NUMBERS

The usual algorithm for multiplying integers by hand is illustrated in Figure 9.6*a* for the binary system. The product of two, unsigned, n-digit numbers can be accommodated in $2n$ digits, so the product of the two 4-bit numbers in this example is accommodated in 8 bits, as shown. In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position. If the multiplier bit is 0, then 0s are entered, as in the third row of the example. The product is computed one bit at a time by adding the bit columns from right to left and propagating carry values between columns.

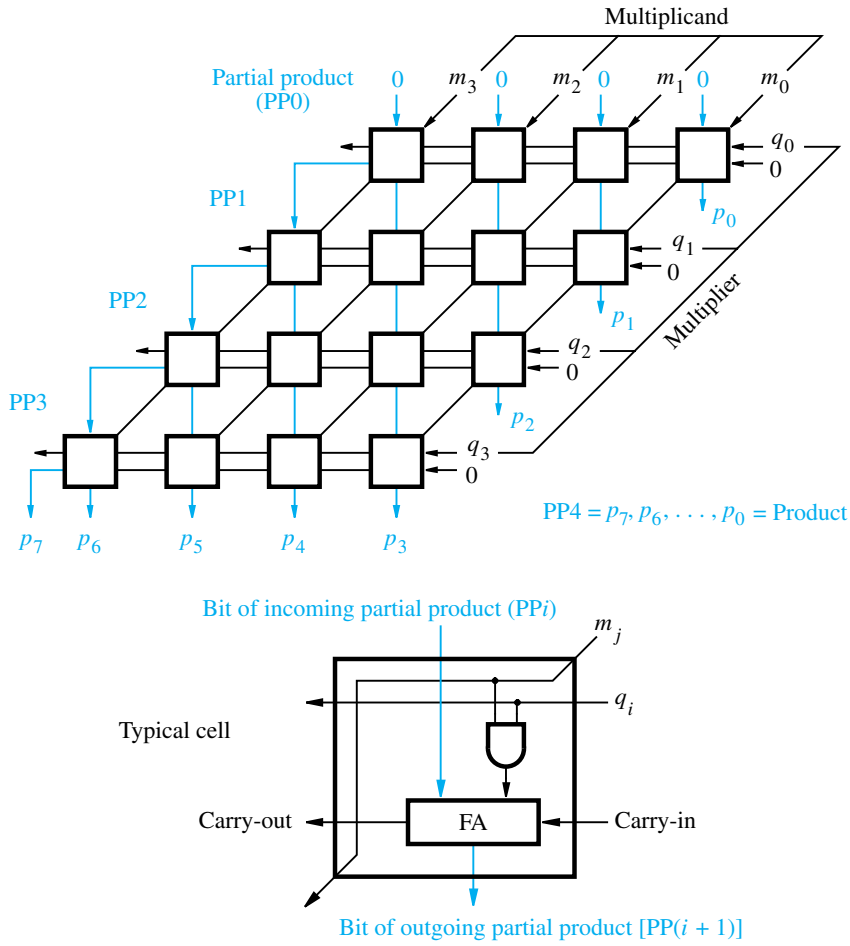
9.3.1 ARRAY MULTIPLIER

Binary multiplication of unsigned operands can be implemented in a combinational, two-dimensional, logic array, as shown in Figure 9.6*b* for the 4-bit operand case. 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, PPi, to generate the outgoing partial product, PP$(i + 1)$, if $q_i = 1$. If $q_i = 0$, PPi is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.

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

```
                1  1  0  1      (13) Multiplicand M
              × 1  0  1  1      (11) Multiplier Q
             ---------------
                1  1  0  1
             1  1  0  1
          0  0  0  0
       1  1  0  1
    ---------------------------
    1  0  0  0  1  1  1  1      (143) Product P
```

(a) Manual multiplication algorithm



(b) Array implementation

Figure 9.6 Array multiplication of unsigned binary operands.

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. (See Problem 9.8.) In the first row of the array, no full adders are needed, because the incoming partial product PP0 is zero. This has been taken into account in developing the delay expression.

9.3.2 SEQUENTIAL CIRCUIT MULTIPLIER

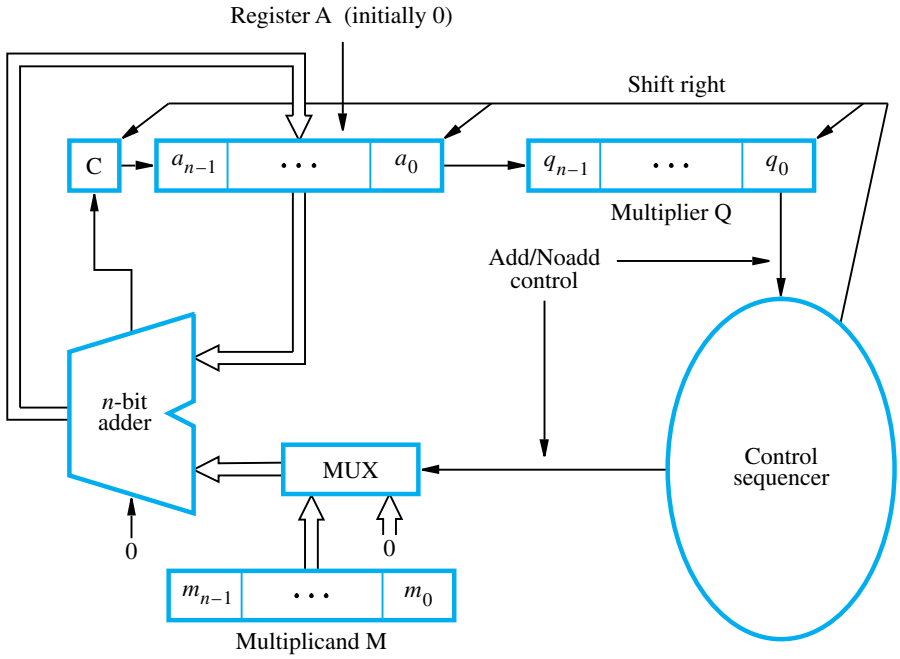
The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication of two n-bit numbers can also be performed in a sequential circuit that uses a single n-bit adder.

The block diagram in Figure 9.7*a* shows the hardware arrangement for sequential multiplication. 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 in Figure 9.6*b*. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PPi 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 PPi 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, PP0, 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. The multiplication example of Figure 9.6*a* is shown in Figure 9.7*b* as it would be performed by this hardware arrangement.

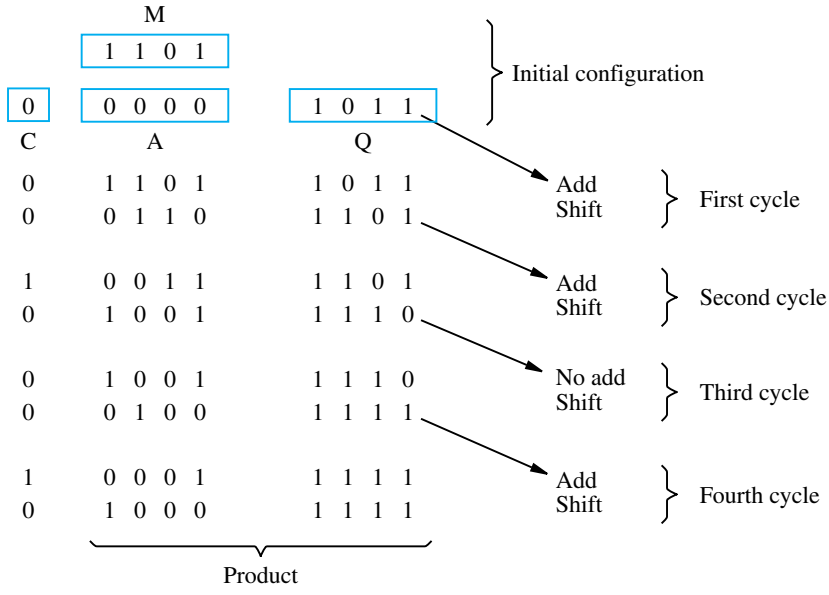
9.4 MULTIPLICATION OF SIGNED NUMBERS

We now discuss multiplication of 2's-complement operands, generating a double-length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.

First, 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. Figure 9.8 shows an example in which a 5-bit signed operand, -13, is the multiplicand. It is multiplied by $+11$ to get



(a) Register configuration



(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

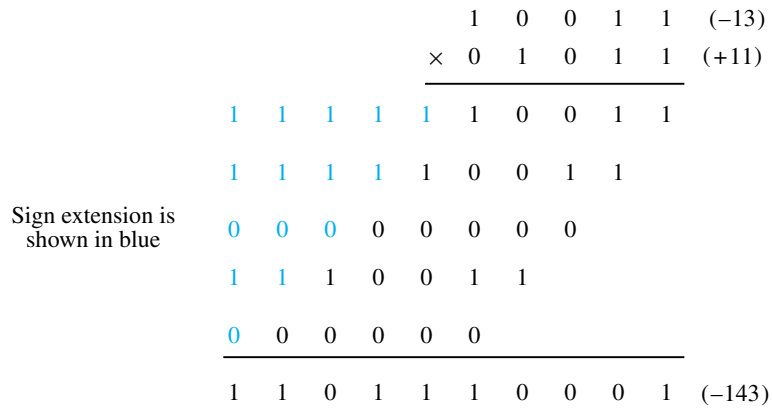


Figure 9.8 Sign extension of negative multiplicand.

the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.

For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm, is described next.

9.4.1 THE BOOTH ALGORITHM

The Booth algorithm [1] generates a $2n$-bit product and treats both positive and negative 2's-complement n-bit operands uniformly. To understand the basis of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers:

$$\begin{array}{rll} & 0100000 & (32) \\ - & 0000010 & (2) \\ \hline & 0011110 & (30) \end{array}$$

This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as $0\ +1\ 0\ 0\ 0\ -1\ 0$.

In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from

1 to 0, as the multiplier is scanned from right to left. Figure 9.9 illustrates the normal and the Booth algorithms for the example just discussed. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. Figure 9.10 shows another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 9.11.

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement system.

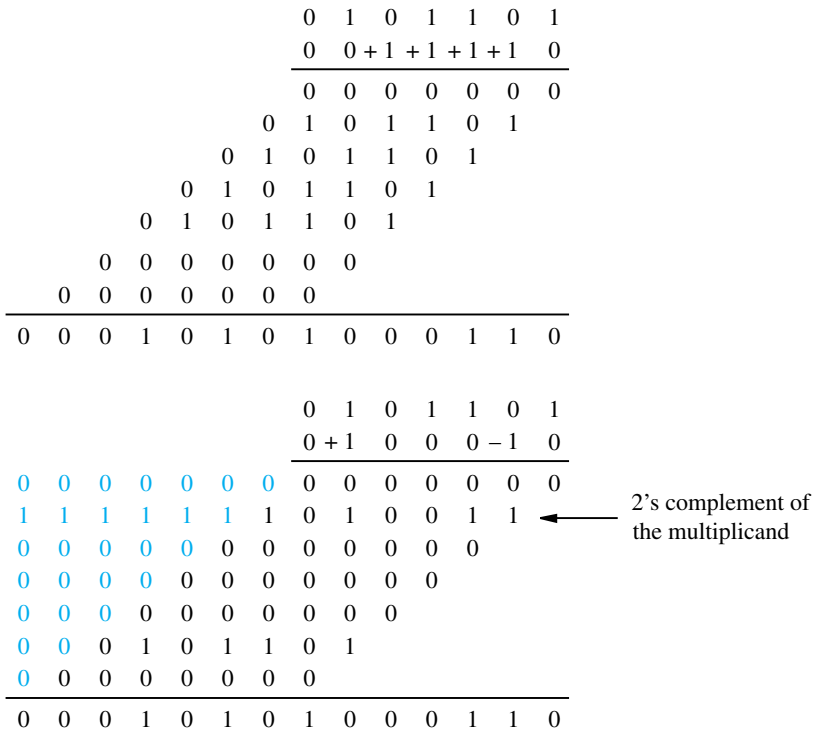


Figure 9.9 Normal and Booth multiplication schemes.

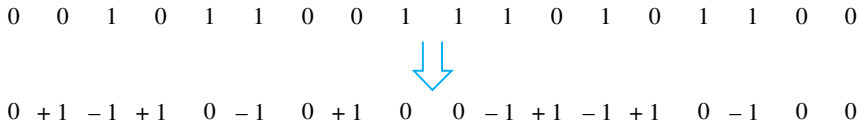


Figure 9.10 Booth recoding of a multiplier.

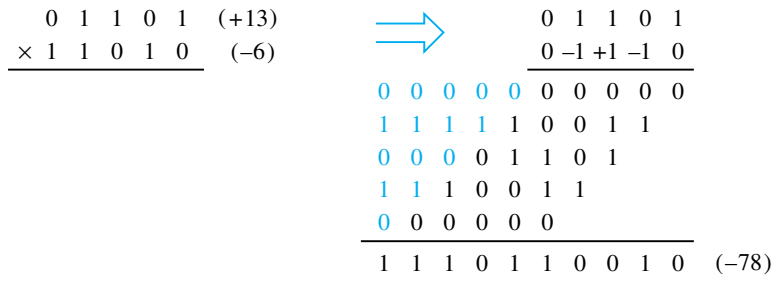


Figure 9.11 Booth multiplication with a negative multiplier.

Suppose that the leftmost 0 of a negative number, X, is at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

Then the value of X is given by

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

The correctness of this expression for $V(X)$ is shown by observing that if X is formed as the sum of two numbers, as follows,

$$\begin{array}{rr} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline X = & 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

then the upper number is the 2's-complement representation of -2^{k+1}. The recoded multiplier now consists of the part corresponding to the lower number, with -1 added in position $k+1$. For example, the multiplier 110110 is recoded as $0\ -1\ +1\ 0\ -1\ 0$.

The Booth technique for recoding multipliers is summarized in Figure 9.12. The transformation $011\ldots110 \Rightarrow +1\ 0\ 0\ldots0\ -1\ 0$ is called *skipping over 1s*. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous blocks. Only a few versions of the shifted multiplicand (the summands) need to be added to generate the product, thus speeding up the multiplication operation. However, in the worst case—that of alternating 1s and 0s in the multiplier—each bit of the multiplier selects a summand. In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Figure 9.13.

The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

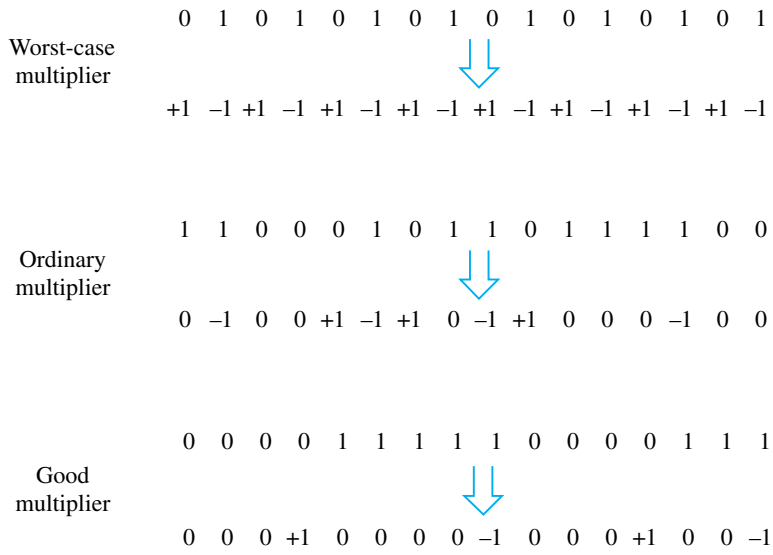


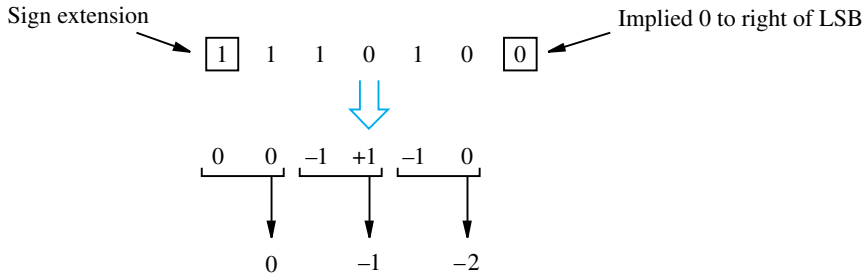
Figure 9.13 Booth recoded multipliers.

9.5 FAST MULTIPLICATION

We now describe two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n-bit operands. The second technique leads to adding the summands in parallel.

9.5.1 BIT-PAIR RECODING OF MULTIPLIERS

A technique called *bit-pair recoding* of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following. The pair $(+1\ -1)$ is equivalent to the pair $(0\ +1)$. That is, instead of adding -1 times the multiplicand M at shift position i to $+1 \times$ M at position $i + 1$, the same result is obtained by adding $+1 \times$ M at position i. Other examples are: $(+1\ 0)$ is equivalent to $(0\ +2)$, $(-1\ +1)$ is equivalent to $(0\ -1)$, and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits. Figure 9.14*a* shows an example of bit-pair recoding of the multiplier in Figure 9.11, and Figure 9.14*b*



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

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times$ M
0	0	1	$+1 \times$ M
0	1	0	$+1 \times$ M
0	1	1	$+2 \times$ M
1	0	0	$-2 \times$ M
1	0	1	$-1 \times$ M
1	1	0	$-1 \times$ M
1	1	1	$0 \times$ M

(b) Table of multiplicand selection decisions

Figure 9.14 Multiplier bit-pair recoding.

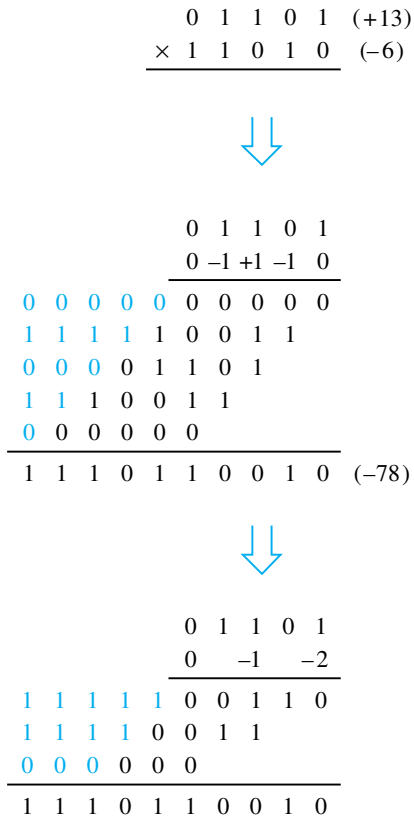


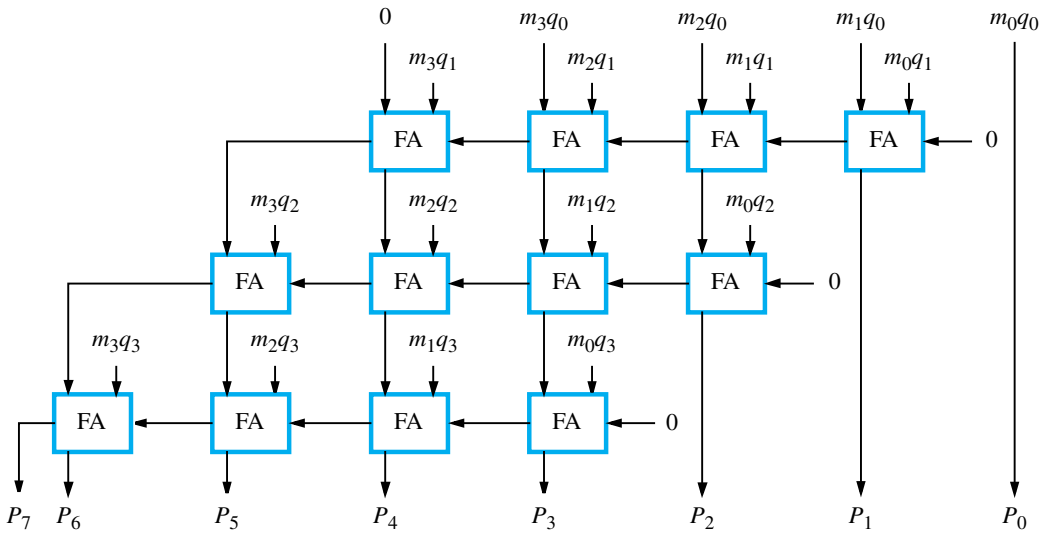
Figure 9.15 Multiplication requiring only $n/2$ summands.

shows a table of the multiplicand selection decisions for all possibilities. The multiplication operation in Figure 9.11 is shown in Figure 9.15 as it would be computed using bit-pair recoding of the multiplier.

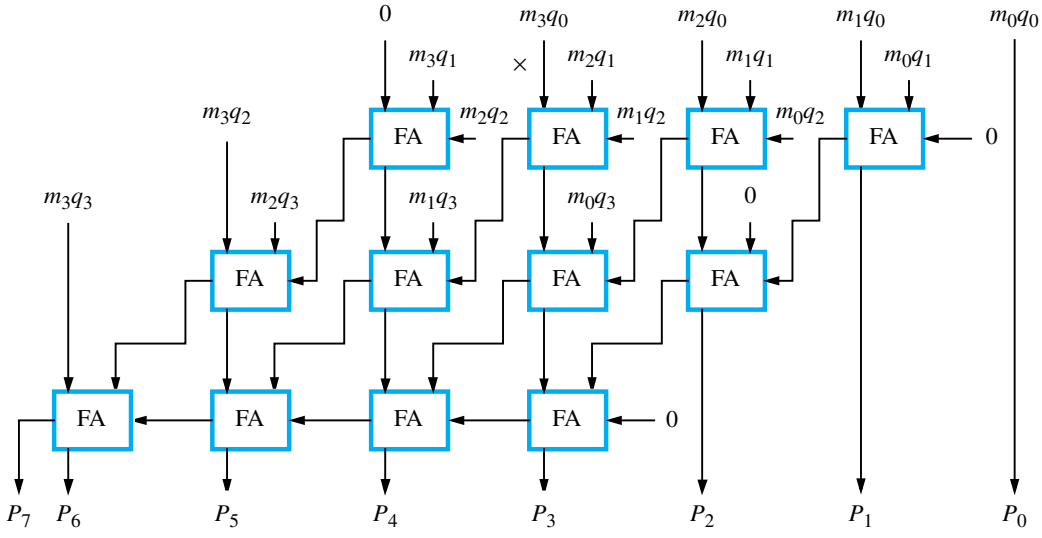
9.5.2 CARRY-SAVE ADDITION OF SUMMANDS

Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process. Consider the 4 × 4 multiplication array shown in Figure 9.16*a*. This structure is in the form of the array shown in Figure 9.6, in which the first row consists of just the AND gates that produce the four inputs m_3q_0, m_2q_0, m_1q_0, and m_0q_0.

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 9.16*b*. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce



(a) Ripple-carry array



(b) Carry-save array

Figure 9.16 Ripple-carry and carry-save arrays for a 4 × 4 multiplier.

the third summand bits m_2q_2, m_1q_2, and m_0q_2. Now, two inputs of each of three full adders in the second row are fed by the sum and carry outputs from the first row. The third input is used to introduce the bits m_2q_3, m_1q_3, and m_0q_3 of the fourth summand. The high-order bits m_3q_2 and m_3q_3 of the third and fourth summands are introduced into the remaining free full-adder inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row, which is a ripple-carry adder, to produce the final product bits.

The delay through the carry-save array is somewhat less than the delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay. The amount of reduction in delay is considered in Problem 9.15.

9.5.3 SUMMAND ADDITION TREE USING 3-2 REDUCERS

A more significant reduction in delay can be achieved when dealing with longer operands than those considered in Figure 9.16. We can group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay. Here, we will refer to a full-adder circuit as simply an adder. Next, we group all the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more adder delay. We continue with this process until there are only two vectors remaining. The adder at each bit position of the three summands is called a *3-2 reducer*, and the logic circuit structure that reduces a number of summands to two is called a *CSA tree*, as described by Wallace [2]. The final two S and C vectors can be added in a carry-lookahead adder to produce the desired product.

Consider the example shown in Figure 9.17. It involves adding the six shifted versions of the multiplicand for the case of multiplying two, 6-bit, unsigned numbers, where all six

						1	0	1	1	0	1	(45)	M
					×	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1		A
					1	0	1	1	0	1			B
				1	0	1	1	0	1				C
			1	0	1	1	0	1					D
		1	0	1	1	0	1						E
	1	0	1	1	0	1							F
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

bits of the multiplier are equal to 1. The six summands, $A, B, \ldots, F$ are added by carry-save addition in Figure 9.18. The blue boxes in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 9.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 9.19. This figure shows that the final two vectors S_4 and C_4 are available in three adder delays

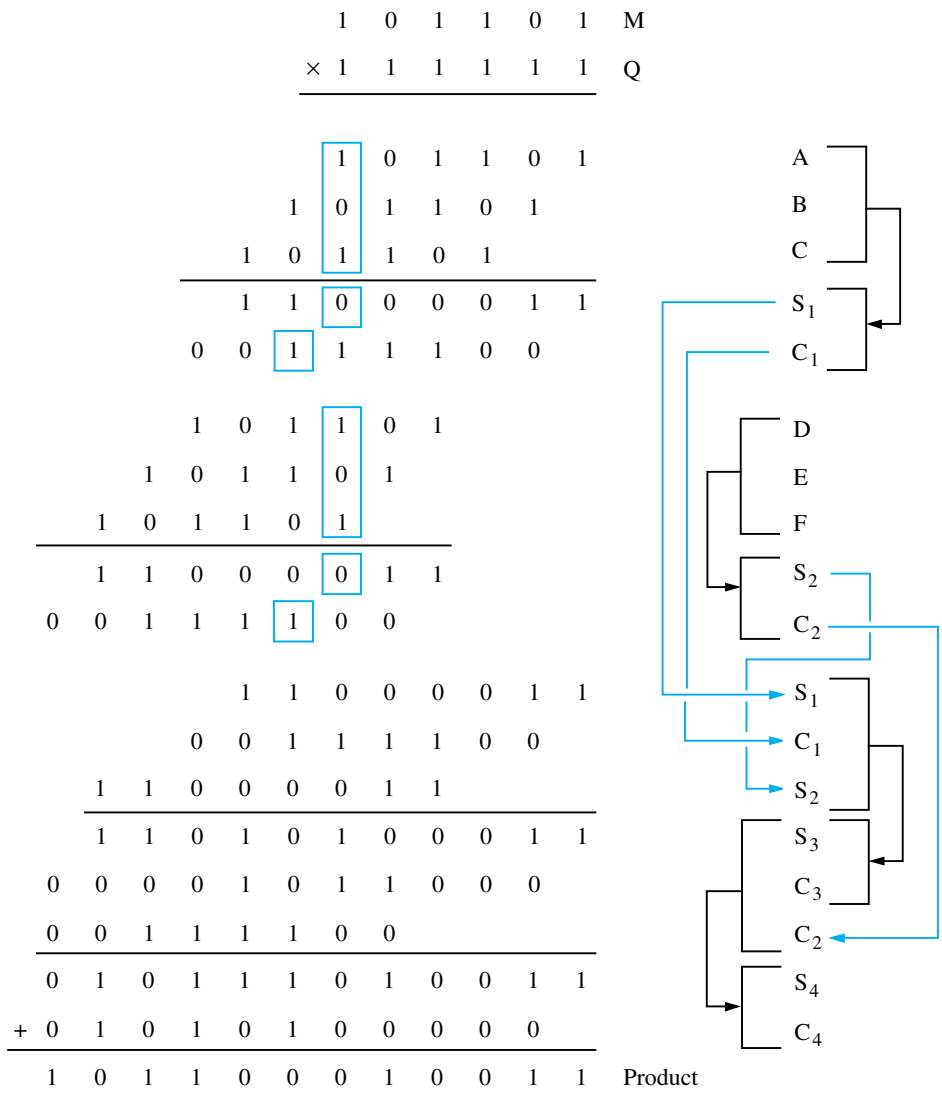


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

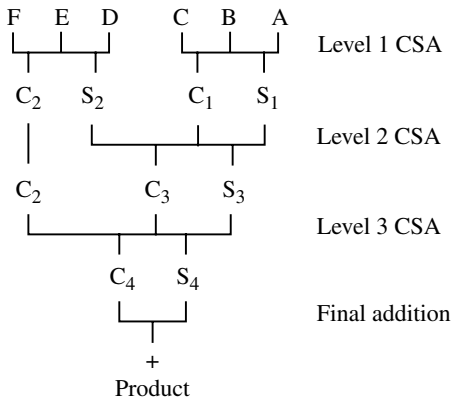


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

after the six input summands are applied to level 1. The final regular addition operation on S_4 and C_4, which produces the product, can be done with a carry-lookahead adder.

The multiplier delay is lower when using the tree structure illustrated in Figure 9.19 than when using the array structure illustrated in Figure 9.16*b*. When the number of summands is large, the reduction in delay is significant. For example, the addition of 32 summands following the pattern shown in Figure 9.19 requires only 8 levels of 3-2 reduction before the final Add operation. In general, it can be shown that approximately $1.7log_2k - 1.7$ levels of 3-2 reduction are needed to reduce k summands to 2 vectors, which, when added, produce the desired product. (See Example 9.3. in Section 9.10.)

We should note that negative summands are involved when signed-number multiplication and Booth recoding of multipliers is used. This requires sign extension of the summands before they are entered into the reduction tree. Also, the number of summands that need to be added is reduced if bit-pair recoding of the multiplier is done.

The 3-2 reducer is not the only logic circuit that can be used in building reduction trees. It is also possible to use 4-2 reducers and 7-3 reducers. The first of these possibilities is described in the next subsection, and the second is explored in Problem 9.17.

9.5.4 SUMMAND ADDITION TREE USING 4-2 REDUCERS

The interconnection pattern between levels in a CSA tree that uses 3-2 reducers is irregular, as can be seen in Figure 9.19. A more regularly structured tree can be obtained by using 4-2 reducers [3], especially for the case in which the number of summands to be reduced is a power of 2. This is the usual case for the multiplication operation in the ALU of a processor. For example, if 32 summands are reduced to 2 using 4-2 reducers at each reduction level, then only four levels are needed. The tree has a regular structure, with 16, 8, 4, and 2 summands at the outputs of the four levels. If 3-2 reducers are used, eight levels

are required, and the wiring connections between levels are quite irregular. Regular tree structures facilitate logic circuit and wiring layout for VLSI circuit implementation.

Let us consider the design of a 4-2 reducer as developed in reference [3]. The addition of four equally-weighted bits, w, x, y, and z, from four summands, produces a value in the range 0 to 4. Such a value cannot be represented by a sum bit, s, and a single carry bit, c. However, a second carry bit, c_{out}, with the same weight as c, can be used along with s and c, to represent any value in the range 0 to 5. This is sufficient for our purposes here.

We do not want to send three output bits down to the next reduction level. That would implement a 4-3 reducer, which provides less reduction than a 3-2 reducer. The solution is to send c_{out} laterally to the 4-2 reducer in the next higher-weighted bit position on the same reduction level. Thus, each 4-2 reducer must have a fifth input, c_{in}, which is the c_{out} output from the 4-2 reducer in the next lower-weighted bit position on the same reduction level.

A final requirement on the design of the 4-2 reducer is that the value of c_{out} cannot depend on the value of c_{in}. This is a key requirement. Without it, carries would ripple laterally along a reduction level, defeating the purpose of parallel reduction of summands with short fixed delay. A 4-2 reducer block is shown in Figure 9.20.

In summary, the specification for a 4-2 reducer is as follows:

- The three outputs, s, c, and c_{out}, represent the arithmetic sum of the five inputs, that is

$$w + x + y + z + c_{in} = s + 2(c + c_{out})$$

  where all operators here are arithmetic.

- Output s is the usual sum variable; that is, s is the XOR function of the five input variables.

- The lateral carry, c_{out}, must be independent of c_{in}. It is a function of only the four input variables w, x, y, and z.

There are different possibilities for specifying the two carry outputs in a way that meets the given conditions. We present one that is easy to describe. First, assign the lateral carry

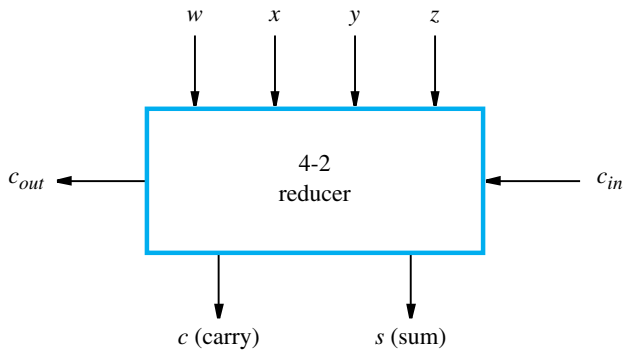


Figure 9.20 A 4-2 reducer block.

				$c_{in} = 0$		$c_{in} = 1$		
w	x	y	z	c	s	c	s	c_{out}
0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0	0
0	0	1	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	1	1
0	1	0	1	0	0	0	1	1
0	1	1	0	0	0	0	1	1
1	0	0	1	0	0	0	1	1
1	0	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
0	1	1	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1
1	1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1	1

Figure 9.21 A 4-2 reducer truth table.

output, c_{out}, to be 1 when two or more of the input variables w, x, y, and z, are equal to 1. Then, the other carry output, c, is determined so as to satisfy the arithmetic condition. A complete truth table satisfying these conditions is given in Figure 9.21. The table is shown in a form that is different from the usual form used in Appendix A. The four inputs w, x, y, and z, are not listed in binary numerical order. They are listed in groups corresponding to the number of inputs that have the value 1. This makes it easy to see how the outputs are specified to meet the given conditions. A logic gate network can be derived from the table.

9.5.5 SUMMARY OF FAST MULTIPLICATION

We now summarize the techniques for high-speed multiplication. Bit-pair recoding of the multiplier, derived from the Booth algorithm, can be used to initially reduce the number of summands by a factor of two. The resulting summands can then be reduced to two in a reduction tree with a relatively small number of reduction levels. The final product

can be generated by an addition operation that uses a carry-lookahead adder. All three of these techniques—bit-pair recoding of the multiplier, parallel reduction of summands, and carry-lookahead addition—have been used in various combinations by the designers of high-performance processors to reduce the time needed to perform multiplication.

9.6 INTEGER DIVISION

In Section 9.3, we discussed the multiplication of unsigned numbers by relating the way the multiplication operation is done manually to the way it is done in a logic circuit. We use the same approach here in discussing integer division. We discuss unsigned-number division in detail, and then make some general comments on the signed-number case.

Figure 9.22 shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that $27 - 26 = 1$ is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

A circuit that implements division by this 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.

Restoring Division

Figure 9.23 shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure 9.7. An n-bit positive divisor is loaded into register M and an n-bit positive dividend

```
      21                    10101
13 ) 274          1101 ) 100010010
     26                     1101
     14                    10000
     13                     1101
      1                     1110
                            1101
                               1
```

Figure 9.22 Longhand division examples.

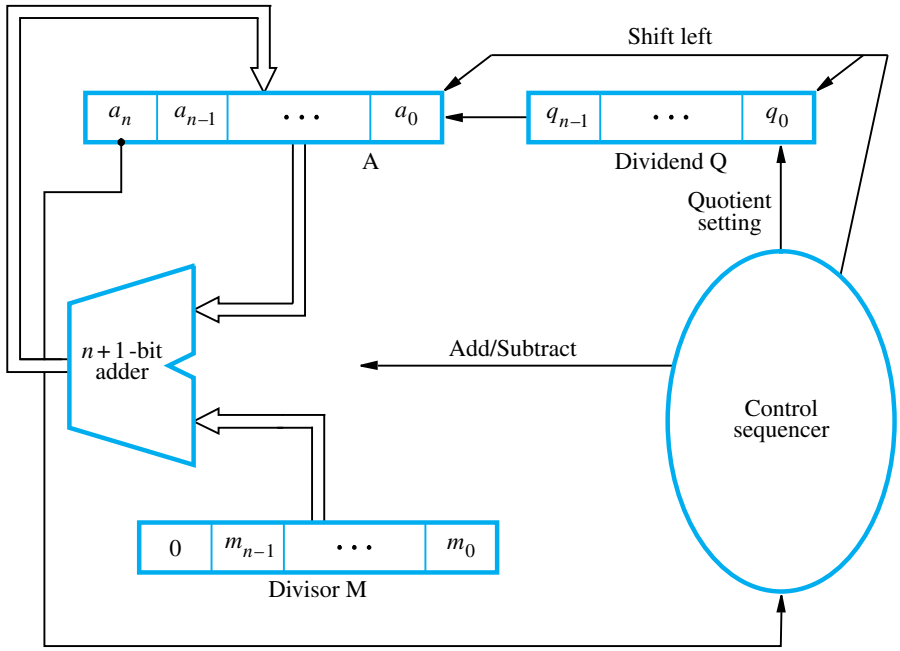


Figure 9.23 Circuit arrangement for binary division.

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.

Figure 9.24 shows a 4-bit example as it would be processed by the circuit in Figure 9.23.

Non-Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform $2A - M$. If A is negative, we restore it by performing $A + M$, and then we shift it left and subtract M. This is equivalent to performing $2A + M$. The q_0 bit is appropriately

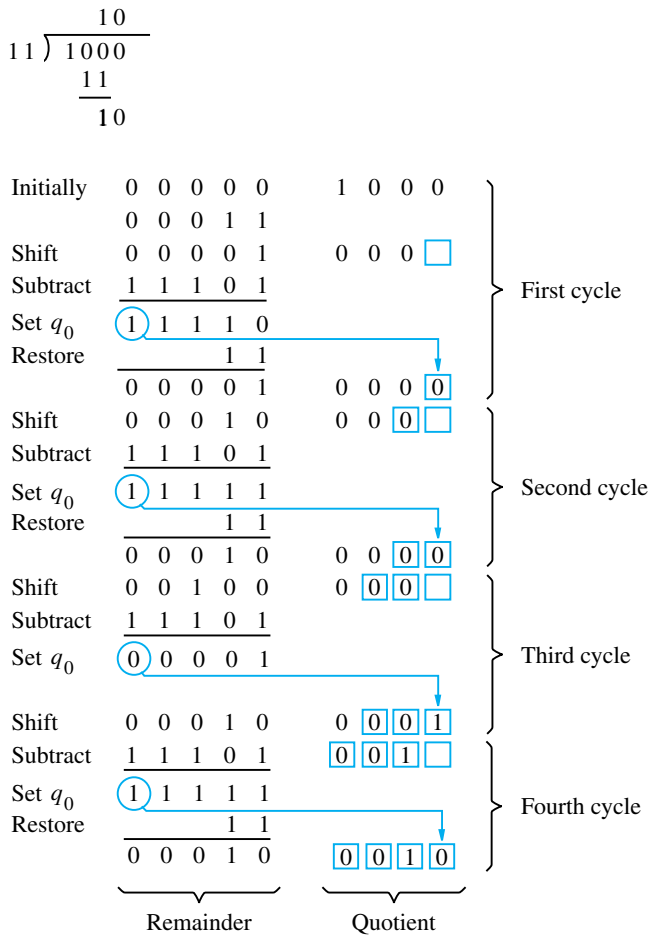


Figure 9.24 A restoring division example.

set to 0 or 1 after the correct operation has been performed. We can summarize this in the following 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. The logic circuitry in Figure 9.23 can also be used to perform this algorithm, except that

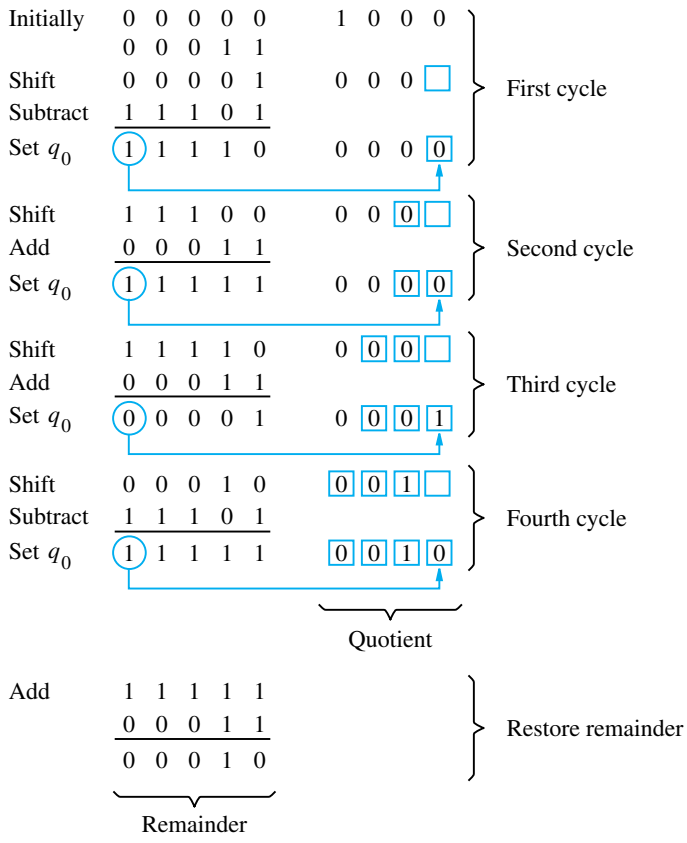


Figure 9.25 A non-restoring division example.

the Restore operations are no longer needed. One Add or Subtract operation is performed in each of the n cycles of stage 1, plus a possible final addition in Stage 2. Figure 9.25 shows how the division example in Figure 9.24 is executed by the non-restoring division algorithm.

There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to change them into positive values. After using one of the algorithms just discussed, the signs of the quotient and the remainder are adjusted as necessary.

9.7 FLOATING-POINT NUMBERS AND OPERATIONS

Chapter 1 provided the motivation for using floating-point numbers and indicated how they can be represented in a 32-bit binary format. In this chapter, we provide more detail on representation formats and arithmetic operations on floating-point numbers. The descriptions

provided here are based on the 2008 version of IEEE (Institute of Electrical and Electronics Engineers) Standard 754, labeled 754-2008 [4].

Recall from Chapter 1 that a binary floating-point number can be represented by

- A sign for the number
- Some significant bits
- A signed scale factor exponent for an implied base of 2

The basic IEEE format is a 32-bit representation, shown in Figure 9.26*a*. The leftmost bit represents the sign, S, for the number. The next 8 bits, E', represent the signed exponent of the scale factor (with an implied base of 2), and the remaining 23 bits, M, are the

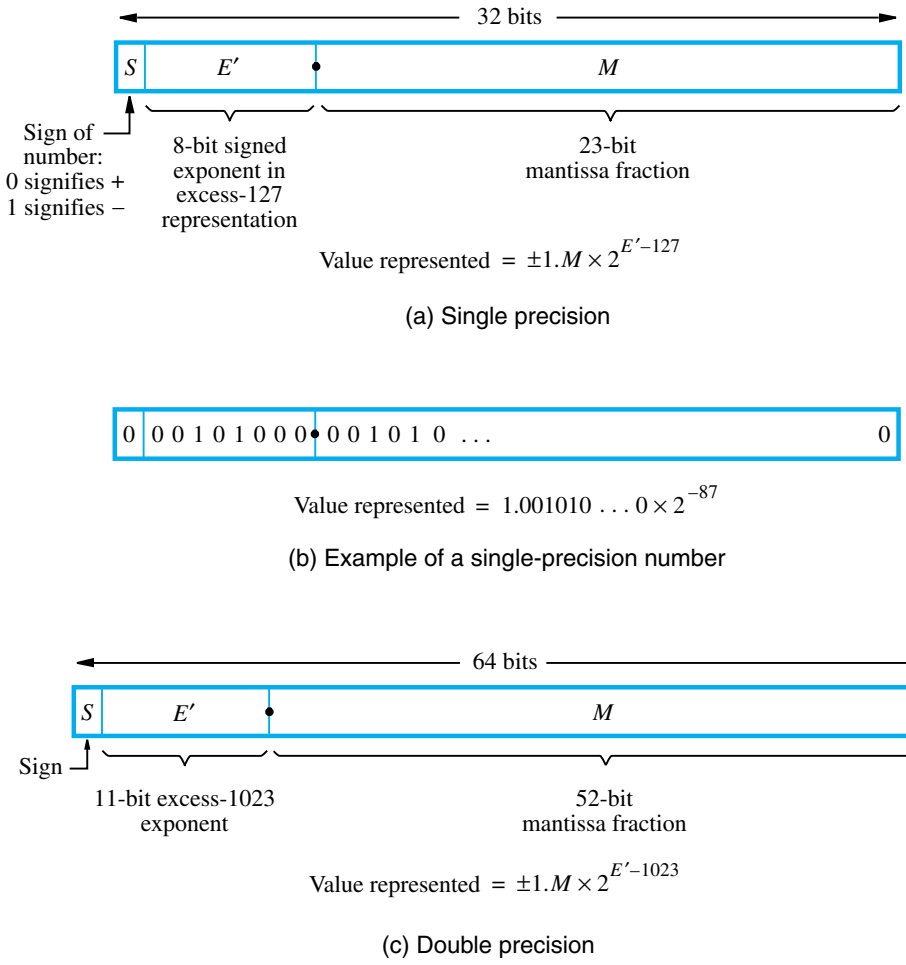


Figure 9.26 IEEE standard floating-point formats.

fractional part of the significant bits. The full 24-bit string, B, of significant bits, called the *mantissa*, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa

$$B = 1.M = 1.b_{-1}b_{-2}\ldots b_{-23}$$

has the value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

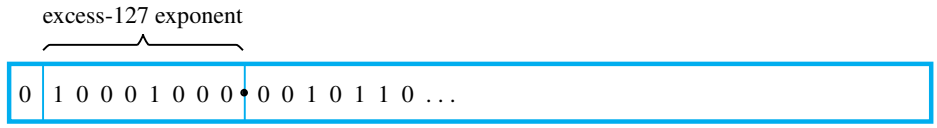
By convention, when the binary point is placed to the right of the first significant bit, the number is said to be *normalized*. Note that the base, 2, of the scale factor and the leading 1 of the mantissa are both fixed. They do not need to appear explicitly in the representation.

Instead of the actual signed exponent, E, the value stored in the exponent field is an unsigned integer $E' = E + 127$. This is called the *excess*-127 format. Thus, E' is in the range $0 \leq E' \leq 255$. The end values of this range, 0 and 255, are used to represent special values, as described later. Therefore, the range of E' for normal values is $1 \leq E' \leq 254$. This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$. The use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers. (See Problem 9.23.)

The 32-bit standard representation in Figure 9.26*a* is called a *single-precision* representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. An example of a single-precision floating-point number is shown in Figure 9.26*b*.

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a *double-precision* format, as shown in Figure 9.26*c*. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E' has the range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations leading to a desired result. For example, the dot product of two vectors of numbers involves accumulating a sum of products. The input vector components are given in a standard precision, either single or double, and the final answer (the dot product) is truncated to the same precision. All intermediate calculations should be done using extended precision to limit accumulation of errors. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. This is because they are usually evaluated by adding up a number of terms in a series representation. In addition to requiring the four basic arithmetic operations, the standard requires three additional operations to be provided: remainder, square root, and conversion between binary and decimal representations.



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^{9}$$

(a) Unnormalized value

$$\text{Value represented} = +1.0110\ldots \times 2^{6}$$

(b) Normalized version

Figure 9.27 Floating-point normalization in IEEE single-precision format.

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent. Figure 9.27 shows an unnormalized value, $0.0010110\ldots \times 2^9$, and its normalized version, $1.0110\ldots \times 2^6$. Since the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than $+127$. In the first case, we say that *underflow* has occurred, and in the second case, we say that *overflow* has occurred.

Special Values

The end values 0 and 255 of the excess-127 exponent E' are used to represent special values. When $E' = 0$ and the mantissa fraction M is zero, the value 0 is represented. When $E' = 255$ and $M = 0$, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still used in these representations, so there are representations for ± 0 and $\pm\infty$.

When $E' = 0$ and $M \neq 0$, *denormal* numbers are represented. Their value is $\pm 0.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for *gradual underflow*, providing an extension of the range of normal representable numbers. This is useful in dealing with very small numbers, which may be needed in certain situations. When $E' = 255$ and $M \neq 0$, the value

represented is called *Not a Number* (NaN). A NaN represents the result of performing an invalid operation such as $0/0$ or $\sqrt{-1}$.

Exceptions

In conforming to the IEEE Standard, a processor must set *exception* flags if any of the following conditions arise when performing operations: underflow, overflow, divide by zero, inexact, invalid. We have already mentioned the first three. *Inexact* is the name for a result that requires rounding in order to be represented in one of the normal formats. An *invalid* exception occurs if operations such as $0/0$ or $\sqrt{-1}$ are attempted. When an exception occurs, the result is set to one of the special values.

If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

9.7.1 ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules given below apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed. Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections.

When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ. Consider a decimal example in which we wish to add 2.9400×10^2 to 4.3100×10^4. We rewrite 2.9400×10^2 as 0.0294×10^4 and then perform addition of the mantissas to get 4.3394×10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

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

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

Multiply Rule

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

Divide Rule

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

9.7.2 GUARD BITS AND TRUNCATION

Let us consider some important aspects of implementing the steps in the preceding algorithms. Although the mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1, it is important to retain extra bits, often called *guard* bits, during the intermediate steps. This yields maximum accuracy in the final results.

Removing guard bits in generating a final result requires that the extended mantissa be *truncated* to create a 24-bit number that approximates the longer version. This operation also arises in other situations, for instance, in converting from decimal to binary numbers. We should mention that the general term rounding is also used for the truncation operation, but a more restrictive definition of rounding is used here as one of the forms of truncation.

There are several ways to truncate. The simplest way is to remove the guard bits and make no changes in the retained bits. This is called *chopping*. Suppose we want to truncate a fraction from six to three bits by this method. All fractions in the range $0.b_{-1}b_{-2}b_{-3}000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$. The error in the 3-bit result ranges from 0 to 0.000111. In other words, the error in chopping ranges from 0 to almost 1 in the least significant position of the retained bits. In our example, this is the b_{-3} position. The result of chopping is a *biased* approximation because the error range is not symmetrical about 0.

The next simplest method of truncation is *von Neumann rounding*. If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits. However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. In our 6-bit to 3-bit truncation example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.b_{-1}b_{-2}1$. The error in this truncation method ranges between -1 and $+1$ in the LSB position of the retained bits. Although the range of error is larger with this technique than it is with chopping, the maximum magnitude is the same, and the approximation is *unbiased* because the error range is symmetrical about 0.

Unbiased approximations are advantageous if many operands and operations are involved in generating a result, because positive errors tend to offset negative errors as the computation proceeds. Statistically, we can expect the results of a complex computation to be more accurate.

The third truncation method is a *rounding* procedure. Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. The procedure is as follows: A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. Thus, $0.b_{-1}b_{-2}b_{-3}1\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3} + 0.001$, and $0.b_{-1}b_{-2}b_{-3}0\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3}$. This provides the desired approximation, except for the case in which the bits to be removed are $10\ldots0$. This is a tie situation; the longer value is halfway between the two closest truncated representations. To break the tie in an unbiased way, one possibility is to choose the retained

bits to be the nearest even number. In terms of our 6-bit example, the value $0.b_{-1}b_{-2}0100$ is truncated to the value $0.b_{-1}b_{-2}0$, and $0.b_{-1}b_{-2}1100$ is truncated to $0.b_{-1}b_{-2}1 + 0.001$. The descriptive phrase "round to the nearest number or nearest even number in case of a tie" is sometimes used to refer to this truncation technique. The error range is approximately $-\frac{1}{2}$ to $+\frac{1}{2}$ in the LSB position of the retained bits. Clearly, this is the best method. However, it is also the most difficult to implement because it requires an addition operation and a possible renormalization. This rounding technique is the default mode for truncation specified in the IEEE floating-point standard. The standard also specifies other truncation methods, referring to all of them as rounding modes.

This discussion of errors that are introduced when guard bits are removed by truncation has treated the case of a single truncation operation. When a long series of calculations involving floating-point numbers is performed, the analysis that determines error ranges or bounds for the final results can be a complicated study. We do not discuss this aspect of numerical computation further, except to make a few comments on the way that guard bits and rounding are handled in the IEEE floating-point standard.

According to the standard, results of single operations must be computed to be accurate within half a unit in the LSB position. This means that rounding must be used as the truncation method. Implementing rounding requires only three guard bits to be carried along during the intermediate steps in performing an operation. The first two of these bits are the two most significant bits of the section of the mantissa to be removed. The third bit is the logical OR of all bits beyond these first two bits in the full representation of the mantissa. This bit is relatively easy to maintain during the intermediate steps of the operations to be performed. It should be initialized to 0. If a 1 is shifted out through this position while aligning mantissas, the bit becomes 1 and retains that value; hence, it is usually called the *sticky bit*.

9.7.3 IMPLEMENTING FLOATING-POINT OPERATIONS

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

An example of the implementation of floating-point operations is shown in Figure 9.28. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 9.26*a*. Following the Add/Subtract rule given in Section 9.7.1, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_A - E'_B$, or n, is sent to the SHIFTER unit. If n is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result. We do not explore this in detail.

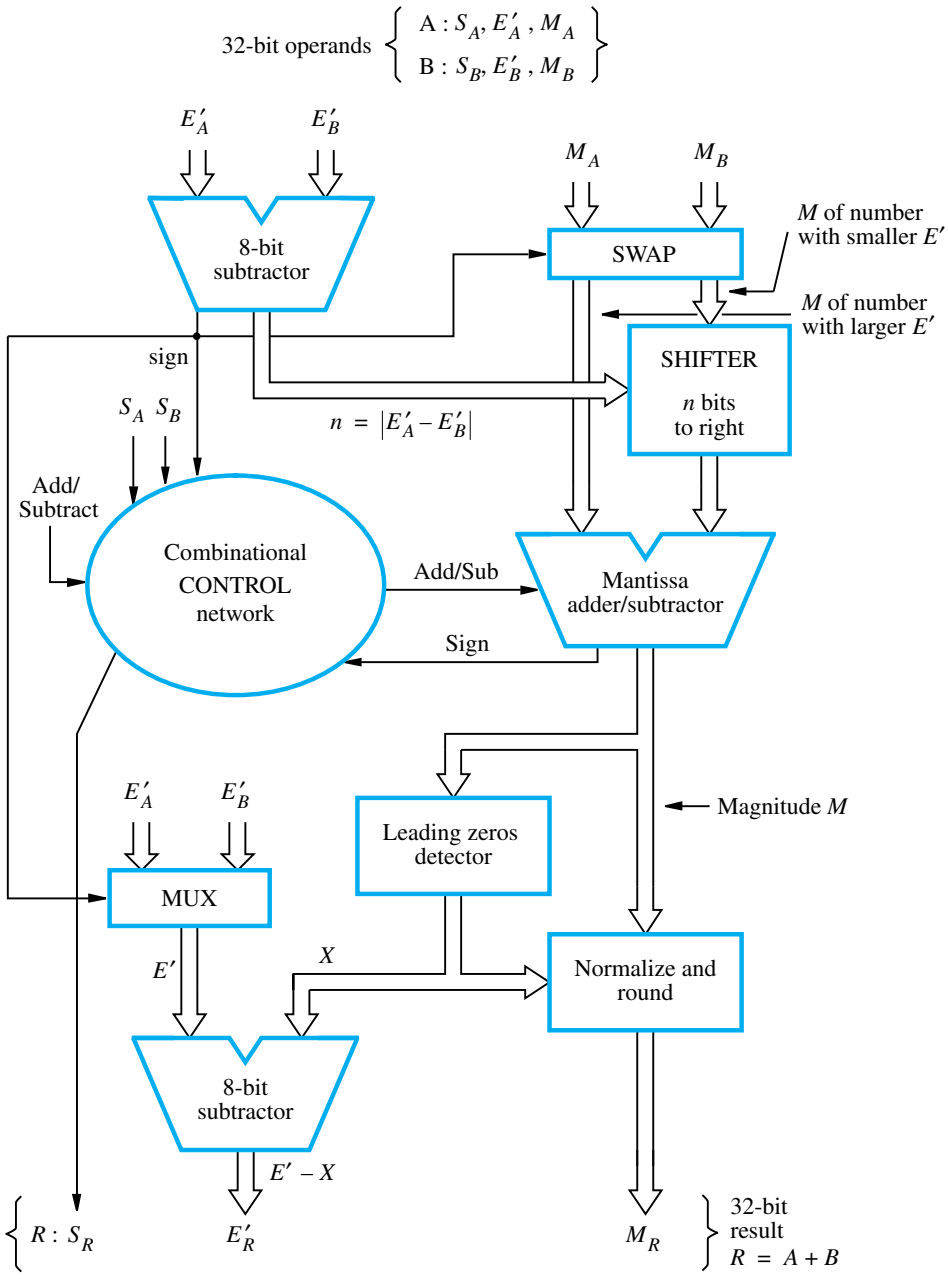


Figure 9.28 Floating-point addition-subtraction unit.

The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network in the upper right corner of Figure 9.28. If the sign is 0, then $E'_A \geq E'_B$ and the mantissas M_A and M_B are sent straight through the SWAP network. This results in M_B being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, M_A, is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E', is tentatively determined as E'_A if $E'_A \geq E'_B$, or E'_B if $E'_A < E'_B$, based on the sign of the difference resulting from comparing exponents in step 1.

Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A and S_B) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, S_R. For example, if A is negative ($S_A = 1$), B is positive ($S_B = 0$), and the operation is $A - B$, then the mantissas are added and the sign of the result is negative ($S_R = 1$). On the other hand, if A and B are both positive and the operation is $A - B$, then the mantissas are subtracted. The sign of the result, S_R, now depends on the mantissa subtraction operation. For instance, if $E'_A > E'_B$, then $M = M_A -$ (shifted M_B) and the resulting number is positive. But if $E'_B > E'_A$, then $M = M_B -$ (shifted M_A) and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When $E'_A = E'_B$ and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be able to construct the complete truth table for the CONTROL network (see Problem 9.26).

Step 4 of the Add/Subtract rule consists of normalizing the result of step 3 by shifting M to the right or to the left, as appropriate. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M. The normalized value is rounded to generate the 24-bit mantissa, M_R, of the result. The value X is also subtracted from the tentative result exponent E' to generate the true result exponent, E'_R. Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form $1.xx\ldots$ were added. The vector M would then have the form $1x.xx\ldots$.

We have not given any details on the guard bits that must be carried along with intermediate mantissa values. In the IEEE standard, only a few bits are needed, as discussed earlier, to generate the 24-bit normalized mantissa of the result.

Let us consider the actual hardware that is needed to implement the blocks in Figure 9.28. The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign-and-magnitude form, we must modify some of our earlier discussions. A combination of 1's-complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation. The operations can be implemented with shift registers. However, they can also be built as combinational logic units for high-performance.

9.8 DECIMAL-TO-BINARY CONVERSION

In Chapter 1 and in this chapter, examples that involve decimal numbers have used small values. Conversion from decimal to binary representation has been easy to do based on the binary bit-position weights 1, 2, 4, 8, 16, ... However, it is useful to have a general method for converting decimal numbers to binary representation.

The fixed-point, unsigned, binary number

$$B = b_{n-1}b_{n-2}\ldots b_0.b_{-1}b_{-2}\ldots b_{-m}$$

has an n-bit integer part and an m-bit fraction part. Its value, $V(B)$, is given by

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

To convert a fixed-point decimal number into binary, the integer and fraction parts are handled separately. Conversion of the integer part starts by dividing it by 2. The remainder, which is either 0 or 1, is the least significant bit, b_0, of the integer part of B. The quotient is again divided by 2. The remainder is the next bit, b_1, of B. This process is repeated up to and including the step in which the quotient becomes 0.

Conversion of the fraction part starts by multiplying it by 2. The part of the product to the left of the decimal point, which is either 0 or 1, is bit b_{-1} of the fraction part of B. The fraction part of the product is again multiplied by 2, generating the next bit, b_{-2} of the fraction part of B. The process is repeated until the fraction part of the product becomes 0 or until the required accuracy is obtained.

Figure 9.29 shows an example of conversion from the decimal number 927.45 to binary. Note that conversion of the integer part is always exact and terminates when the quotient becomes 0. But an exact binary fraction may not exist for a given decimal fraction. For example, the decimal fraction 0.45 used in Figure 9.29 does not have an exact binary equivalent. This is obvious from the pattern developing in the figure. In such cases, the binary fraction is generated to some desired level of accuracy. Of course, some decimal fractions have an exact binary representation. For example, the decimal fraction 0.25 has a binary equivalent of 0.01.

9.9 CONCLUDING REMARKS

Computer arithmetic poses several interesting logic design problems. This chapter discussed some of the techniques that have proven useful in designing binary arithmetic units. The carry-lookahead technique is one of the major ideas in high-performance adder design. In the design of fast multipliers, bit-pair recoding of the multiplier, derived from the Booth algorithm, reduces the number of summands that must be added to generate the product. The parallel addition of summands using carry-save reduction trees substantially reduces

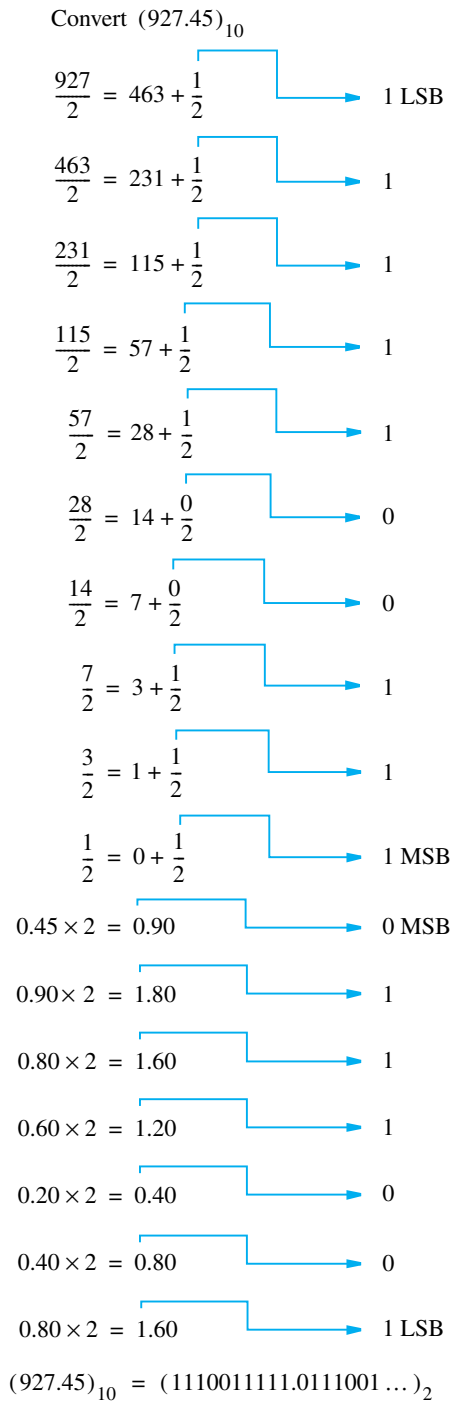


Figure 9.29 Conversion from decimal to binary.

the time needed to add the summands. The important IEEE floating-point number representation standard was described, and rules for performing the four standard operations were given.

9.10 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 9.1 **Problem:** How many logic gates are needed to build the 4-bit carry-lookahead adder shown in Figure 9.4?

Solution: Each B cell requires 3 gates as shown in Figure 9.4*a*. Hence, 12 gates are needed for all four B cells.

The carries c_1, c_2, c_3, and c_4, produced by the carry-lookahead logic, require 2, 3, 4, and 5 gates, respectively, according to the four logic expressions in Section 9.2.1. The carry-lookahead logic also produces G_0^I, using 4 gates, and P_0^I, using 1 gate, as also shown in Section 9.2.1. Hence, a total of 19 gates are needed to implement the carry-lookahead logic.

The complete 4-bit adder requires $12 + 19 = 31$ gates, with a maximum fan-in of 5.

Example 9.2 **Problem:** Assuming 6-bit 2's-complement number representation, multiply the multiplicand A = 110101 by the multiplier B = 011011 using both the normal Booth algorithm and the bit-pair recoding Booth algorithm, following the pattern used in Figure 9.15.

Solution: The multiplications are performed as follows:

(*a*) Normal Booth algorithm

						1	1	0	1	0	1
					×	+1	0	−1	+1	0	−1
0	0	0	0	0	0	0	0	1	0	1	1
										0	
1	1	1	1	1	1	0	1	0	1		
0	0	0	0	0	1	0	1	1			
							0				
1	1	1	0	1	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

(*b*) Bit-pair recoding Booth algorithm

						1	1	0	1	0	1
					×		+2		−1		−1
0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	1	0	1	1		
1	1	1	0	1	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

Example 9.3

Problem: How many levels of 4-2 reducers are needed to reduce k summands to 2 in a reduction tree? How many levels are needed if 3-2 reducers are used?

Solution: Let the number of levels be L.

For 4-2 reducers, we have

$$k(1/2)^L = 2$$

Take logarithms to the base 2 of each side of this equation to derive

$$log_2 k - L = 1$$

or

$$L = log_2 k - 1$$

For 3-2 reducers, we have

$$k(2/3)^L = 2$$

As above, taking logarithms to the base 2, we derive

$$\begin{aligned} log_2 k + L(log_2 2 - log_2 3) &= log_2 2 \\ log_2 k + L(1 - 1.59) &= 1 \\ L &= (1 - log_2 k)/(-0.59) \\ L &= 1.7 log_2 k - 1.7 \end{aligned}$$

These expressions are only approximations unless the number of input summands to each level is a multiple of 4 in the case of 4-2 reduction, or is a multiple of 3 in the case of 3-2 reduction.

Example 9.4

Problem: Convert the decimal fraction 0.1 to a binary fraction. If the conversion is not exact, give the binary fraction approximation to 8 bits after the binary point using each of the three truncation methods discussed in Section 9.7.2.

Solution: Use the conversion method given in Section 9.8. Multiplying the decimal fraction 0.1 by 2 repeatedly, as shown in Figure 9.29, generates the sequence of bits

0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, . . . to the left of the decimal point, which continues indefinitely, repeating the pattern 0, 0, 1, 1. Hence, the conversion is not exact.

- Truncation by chopping gives 0.00011001
- Truncation by von Neumann rounding gives 0.00011001
- Truncation by rounding gives 0.00011010

Example 9.5 **Problem:** Consider the following 12-bit floating-point number representation format that is manageable for working through numerical exercises. The first bit is the sign of the number. The next five bits represent an excess-15 exponent for the scale factor, which has an implied base of 2. The last six bits represent the fractional part of the mantissa, which has an implied 1 to the left of the binary point.

Perform Subtract and Multiply operations on the operands

$A =$	0	10001	011011
$B =$	1	01111	101010

which represent the numbers

$$A = 1.011011 \times 2^2$$

and

$$B = -1.101010 \times 2^0$$

Solution: The required operations are performed as follows:

- Subtraction
  According to the Add/Subtract rule in Section 9.7.1, we perform the following four steps:

  1. Shift the mantissa of B to the right by two bit positions, giving 0.01101010.
  2. Set the exponent of the result to 10001.
  3. Subtract the mantissa of B from the mantissa of A by adding mantissas, because B is negative, giving

$$\begin{array}{rcccccccccc} & 1 & . & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 0 & . & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline & 1 & . & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

  and set the sign of the result to 0 (positive).

  4. The result is in normalized form, but the fractional part of the mantissa needs to be truncated to six bits. If this is done by rounding, the two bits to be removed represent the tie case, so we round to the nearest even number by adding 1, obtaining a result mantissa of 1.110110. The answer is

$A - B =$	0	10001	110110

- Multiplication
  According to the Multiplication rule in Section 9.7.1, we perform the following three steps:

  1. Add the exponents and subtract 15 to obtain 10001 as the exponent of the result.
  2. Multiply mantissas to obtain 10.010110101110 as the mantissa of the result. The sign of the result is set to 1 (negative).
  3. Normalize the resulting mantissa by shifting it to the right by one bit position. Then add 1 to the exponent to obtain 10010 as the exponent of the result. Truncate the mantissa fraction to six bits by rounding to obtain the answer

$A \times B =$

0	10010	001011

PROBLEMS

9.1 **[M]** A *half adder* is a combinational logic circuit that has two inputs, x and y, and two outputs, s and c, that are the sum and carry-out, respectively, resulting from the binary addition of x and y.

(a) Design a half adder as a two-level AND-OR circuit.

(b) Show how to implement a full adder, as shown in Figure 9.2a, by using two half adders and external logic gates, as necessary.

(c) Compare the longest logic delay path through the network derived in part (b) to that of the logic delay of the adder network shown in Figure 9.2a.

9.2 **[M]** The 1's-complement and 2's-complement binary representation methods are special cases of the $(b-1)$'s-complement and b's-complement representation techniques in base b number systems. For example, consider the decimal system. The sign-and-magnitude values +526, −526, +70, and −70 have 4-digit signed-number representations in each of the two complement systems, as shown in Figure P9.1. The 9's-complement is formed by

Representation	Examples			
Sign and magnitude	+526	–526	+70	–70
9's complement	0526	9473	0070	9929
10's complement	0526	9474	0070	9930

Figure P9.1 Signed numbers in base 10 used in Problem 9.2.

taking the complement of each digit position with respect to 9. The 10's-complement is formed by adding 1 to the 9's-complement. In each of the latter two representations, the leftmost digit is zero for a positive number and 9 for a negative number.

Now consider the base-3 (ternary) system, in which the unsigned, 5-digit number $t_4t_3t_2t_1t_0$ has the value $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$, with $0 \leq t_i \leq 2$. Express the ternary sign-and-magnitude numbers $+11011$, -10222, $+2120$, -1212, $+10$, and -201 as 6-digit, signed, ternary numbers in the 3's-complement system.

9.3 **[M]** Represent each of the decimal values 56, -37, 122, and -123 as signed 6-digit numbers in the 3's-complement ternary format, perform addition and subtraction on them in all possible pairwise combinations, and state whether or not arithmetic overflow occurs for each operation performed. (See Problem 9.2 for a definition of the ternary number system, and use a technique analogous to that given in Section 9.8 for decimal-to-ternary integer conversion.)

9.4 **[M]** A modulo 10 adder is needed for adding BCD digits. Modulo 10 addition of two BCD digits, $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$, can be achieved as follows: Add A to B (binary addition). Then, if the result is an illegal code that is greater than or equal to 10_{10}, add 6_{10}. (Ignore overflow from this addition.)

(a) When is the output carry equal to 1?

(b) Show that this algorithm gives correct results for:

(1) $A = 0101$ and $B = 0110$

(2) $A = 0011$ and $B = 0100$

(c) Design a BCD digit adder using a 4-bit binary adder and external logic gates as needed. The inputs are $A_3A_2A_1A_0$, $B_3B_2B_1B_0$, and a carry-in. The outputs are the sum digit $S_3S_2S_1S_0$ and the carry-out. A cascade of such blocks can form a ripple-carry BCD adder.

9.5 **[E]** Show that the logic expression $c_n \oplus c_{n-1}$ is a correct indicator of overflow in the addition of 2's-complement integers by using an appropriate truth table.

9.6 **[E]** Use appropriate parts of the solution in Example 9.1 to calculate how many logic gates are needed to build the 16-bit carry-lookahead adder shown in Figure 9.5.

9.7 **[M]** Carry-lookahead adders and their delay are investigated in this problem.

(a) Design a 64-bit adder that uses four of the 16-bit carry-lookahead adders shown in Figure 9.5 along with additional logic circuits to generate c_{16}, c_{32}, c_{48}, and c_{64}, from c_0 and the G_i^{II} and P_i^{II} variables shown in the figure. What is the relationship of the additional circuits to the carry-lookahead logic circuits in the figure?

(b) Show that the delay through the 64-bit adder is 12 gate delays for s_{63} and 7 gate delays for c_{64}, as claimed at the end of Section 9.2.1.

(c) Compare the gate delays to produce s_{31} and c_{32} in the 64-bit adder of part (a) to the gate delays for the same variables in the 32-bit adder built from a cascade of two 16-bit adders, as discussed in Section 9.2.1.

9.8 **[M]** Show that the worst case delay through an $n \times n$ array of the type shown in Figure 9.6b is $6(n-1) - 1$ gate delays, as claimed in Section 9.3.1.

9.9 **[E]** Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(a) $A = 010111$ and $B = 110110$

(b) $A = 110011$ and $B = 101100$

(c) $A = 001111$ and $B = 001111$

9.10 **[M]** Repeat Problem 9.9 using bit-pair recoding of the multiplier.

9.11 **[M]** Indicate generally how to modify the circuit diagram in Figure 9.7a to implement multiplication of 2's-complement n-bit numbers using the Booth algorithm, by clearly specifying inputs and outputs for the Control sequencer and any other changes needed around the adder and register A.

9.12 **[M]** Extend the Figure 9.14b table to 16 rows, indicating how to recode three multiplier bits: $i + 2$, $i + 1$, and i. Can all required versions of the multiplicand selected at position i be generated by shifting and/or negating the multiplicand M? If not, what versions cannot be generated this way, and for what cases are they required?

9.13 **[M]** If the product of two n-bit numbers in 2's-complement representation can be represented in n bits, the manual multiplication algorithm shown in Figure 9.6a can be used directly, treating the sign bits the same as the other bits. Try this on each of the following pairs of 4-bit signed numbers:

(a) Multiplicand = 1110 and Multiplier = 1101

(b) Multiplicand = 0010 and Multiplier = 1110

Why does this work correctly?

9.14 **[D]** An integer arithmetic unit that can perform addition and multiplication of 16-bit unsigned numbers is to be used to multiply two 32-bit unsigned numbers. All operands, intermediate results, and final results are held in 16-bit registers labeled R_0 through R_{15}. The hardware multiplier multiplies the contents of R_i (multiplicand) by R_j (multiplier) and stores the double-length 32-bit product in registers R_j and R_{j+1}, with the low-order half in R_j. When $j = i - 1$, the product overwrites both operands. The hardware adder adds the contents of R_i and R_j and puts the result in R_j. The input carry to an Add operation is 0, and the input carry to an Add-with-carry operation is the contents of a carry flag C. The output carry from the adder is always stored in C.

Specify the steps of a procedure for multiplying two 32-bit operands in registers R_1, R_0, and R_3, R_2, high-order halves first, leaving the 64-bit product in registers R_{15}, R_{14}, R_{13}, and R_{12}. Any of the registers R_{11} through R_4 may be used for intermediate values, if necessary. Each step in the procedure can be a multiplication, or an addition, or a register transfer operation.

9.15 **[M]** Delay in multiplier arrays is investigated in this problem.

(a) Calculate the delay, in terms of full-adder block delays, in producing product bit p_7 in each of the 4×4 multiplier arrays in Figure 9.16. Ignore the AND gate delay to generate all $m_i q_j$ products at the beginning.

(b) Develop delay expressions for each of the arrays in Figure 9.16 in terms of n for the $n \times n$ case, as an extension of part (a) of the problem. Then use these expressions to calculate delay for the 32×32 case for each array.

9.16 **[M]** Tree depth for carry-save reduction is analyzed in this problem.

(*a*) How many 3-2 reduction levels are needed to reduce 16 summands to 2 using a pattern similar to that shown in Figure 9.19?

(*b*) Repeat part (*a*) for reducing 32 summands to 2 to show that the claim of 8 levels in Section 9.5.3 is correct.

(*c*) Compare the exact answers in parts (*a*) and (*b*) to the results obtained by using the approximation developed in Example 9.3 in Section 9.10.

9.17 **[M]** Tree reduction of summands using 3-2 and 4-2 reducers was described in Sections 9.5.3 and 9.5.4. It is also possible to perform 7-3 reductions on each reduction level. When only three summands remain, a 3-2 reduction is performed, followed by addition of the final two summands.

(*a*) How many 7-3 reduction levels are needed to reduce 32 summands to three? Compare this to the seven levels needed to reduce 32 summands to three when using 3-2 reductions.

(*b*) Example 9.3 in Section 9.10 shows that $log_2 k - 1$ levels of 4-2 reduction are needed to reduce k summands to 2 in a reduction tree. How many levels of 7-3 reduction are needed to reduce k summands to 3?

9.18 **[M]** Show how to implement a 4-2 reducer by using two 3-2 reducers. The truth table for this implementation is different from that shown in Figure 9.21.

9.19 **[E]** Using manual methods, perform the operations $A \times B$ and $A \div B$ on the 5-bit unsigned numbers $A = 10101$ and $B = 00101$.

9.20 **[M]** Show how the multiplication and division operations in Problem 9.19 would be performed by the hardware in Figures 9.7*a* and 9.23, respectively, by constructing charts similar to those in Figures 9.7*b* and 9.25.

9.21 **[D]** In Section 9.7, we used the practical-sized 32-bit IEEE standard format for floating-point numbers. Here, we use a shortened format that retains all the pertinent concepts but is manageable for working through numerical exercises. Consider that floating-point numbers are represented in a 12-bit format as shown in Figure P9.2. The scale factor has an implied base of 2 and a 5-bit, excess-15 exponent, with the two end values of 0 and 31 used to signify exact 0 and infinity, respectively. The 6-bit mantissa is normalized as in the IEEE format, with an implied 1 to the left of the binary point.

(*a*) Represent the numbers $+1.7$, -0.012, $+19$, and $\frac{1}{8}$ in this format.

(*b*) What are the smallest and largest numbers representable in this format?

(*c*) How does the range calculated in part (*b*) compare to the ranges of a 12-bit signed integer and a 12-bit signed fraction?

(*d*) Perform Add, Subtract, Multiply, and Divide operations on the operands

$A =$	0	10000	011011
$B =$	1	01110	101010

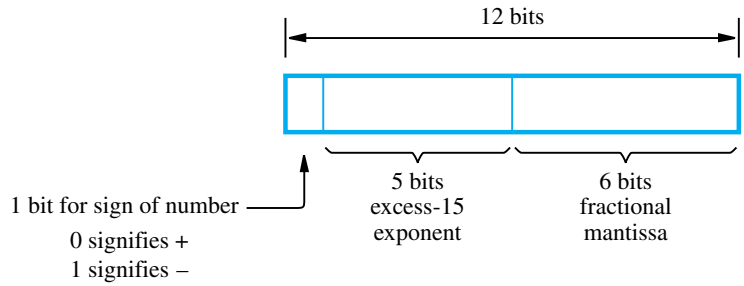


Figure P9.2 Floating-point format used in Problem 9.21.

9.22 **[D]** Consider a 16-bit, floating-point number in a format similar to that discussed in Problem 9.21, with a 6-bit exponent and a 9-bit mantissa fraction. The base of the scale factor is 2 and the exponent is represented in excess-31 format.

(*a*) Add the numbers *A* and *B*, formatted as follows:

$A =$	0	100001	111111110
$B =$	0	011111	001010101

Give the answer in normalized form. Remember that an implicit 1 is to the left of the binary point but is not included in the *A* and *B* formats. Use rounding as the truncation method when producing the final mantissa.

(*b*) Using decimal numbers *w*, *x*, *y*, and *z*, express the magnitude of the largest and smallest (nonzero) values representable in the preceding normalized floating-point format. Use the following form:

$$\text{Largest} = w \times 2^{x}$$
$$\text{Smallest} = y \times 2^{-z}$$

9.23 **[M]** How does the excess-*x* representation for exponents of the scale factor in the floating-point number representation of Figure 9.26*a* facilitate the comparison of the relative sizes of two floating-point numbers? (Hint: Assume that a combinational logic network that compares the relative sizes of two, 32-bit, unsigned integers is available. Use this network, along with external logic gates, as necessary, to design the required network for the comparison of floating-point numbers.)

9.24 **[D]** In Problem 9.21(*a*), conversion of the simple decimal numbers into binary floating-point format is straightforward. However, if the decimal numbers are given in floating-point format, conversion is not straightforward because we cannot separately convert the mantissa and the exponent of the scale factor because $10^{x} = 2^{y}$ does not, in general, allow both *x* and *y* to be integers. Suppose a table of binary, floating-point numbers t_i, such that $t_i = 10^{x_i}$ for x_i in the representable range, is stored in a computer. Give a procedure in general terms for

converting a given decimal floating-point number into binary floating-point format. You may use both the integer and floating-point instructions available in the computer.

9.25 **[D]** Construct an example to show that three guard bits are needed to produce the correct answer when two positive numbers are subtracted.

9.26 **[M]** Derive logic expressions that specify the Add/Sub and S_R outputs of the combinational CONTROL network in Figure 9.28.

9.27 **[M]** If gate fan-in is limited to four, how can the SHIFTER in Figure 9.28 be implemented combinationally?

9.28 **[M]** Sketch a logic-gate network that implements the multiplexer MUX in Figure 9.28.

9.29 **[M]** Relate the structure of the SWAP network in Figure 9.28 to your solution to Problem 9.28.

9.30 **[M]** How can the leading zeros detector in Figure 9.28 be implemented combinationally?

9.31 **[M]** The mantissa adder-subtractor in Figure 9.28 operates on positive, unsigned binary fractions and must produce a sign-and-magnitude result. In the discussion accompanying Figure 9.28, we state that 1's-complement arithmetic is convenient because of the required format for input and output operands. When adding two signed numbers in 1's-complement notation, the carry-out from the sign position must be added to the result to obtain the correct signed answer. This is called *end-around carry correction*. Consider the two examples in Figure P9.3, which illustrate addition using signed, 4-bit encodings of operands and answers in the 1's-complement system.

The 1's-complement arithmetic system is convenient when a sign-and-magnitude result is to be generated because a negative number in 1's-complement notation can be converted to sign-and-magnitude form by complementing the bits to the right of the sign-bit position. Using 2's-complement arithmetic, addition of +1 is needed to convert a negative value into sign-and-magnitude notation. If a carry-lookahead adder is used, it is possible to incorporate the end-around carry operation required by 1's-complement arithmetic into the lookahead logic. With this discussion as a guide, give the complete design of the 1's-complement adder/subtractor required in Figure 9.28.

9.32 **[M]** Signed binary fractions in 2's-complement representation are discussed in Section 1.4.2.

(*a*) Express the decimal values 0.5, −0.123, −0.75, and −0.1 as signed 6-bit fractions. (See Section 9.8 for decimal-to-binary fraction conversion.)

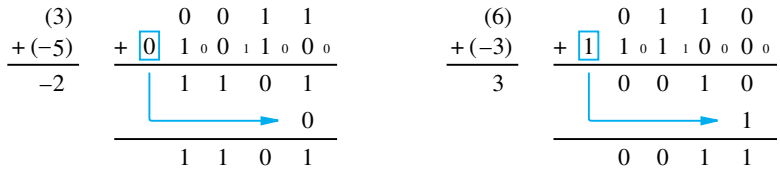


Figure P9.3 1's-complement addition used in Problem 9.31.

(*b*) What is the maximum representation error, *e*, involved in using only 5 significant bits after the binary point?

(*c*) Calculate the number of bits needed after the binary point so that the representation error *e* is less than 0.1, 0.01, or 0.001, respectively.

9.33 **[E]** Which of the four 6-bit answers to Problem 9.32(*a*) are not exact? For each of these cases, give the three 6-bit values that correspond to the three types of truncation defined in Section 9.7.2.

REFERENCES

1. A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 2, part 2, 1951, pp. 236-240.
2. C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, February 1964, pp. 14-17.
3. M. R. Santoro and M. A. Horowitz, "SPIM: A Pipelined 64 × 64-bit Iterative Multiplier," *IEEE Journal of Solid-State Circuits*, vol. 24, No.2, April 1989, pp. 487-493.
4. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, August 2008.