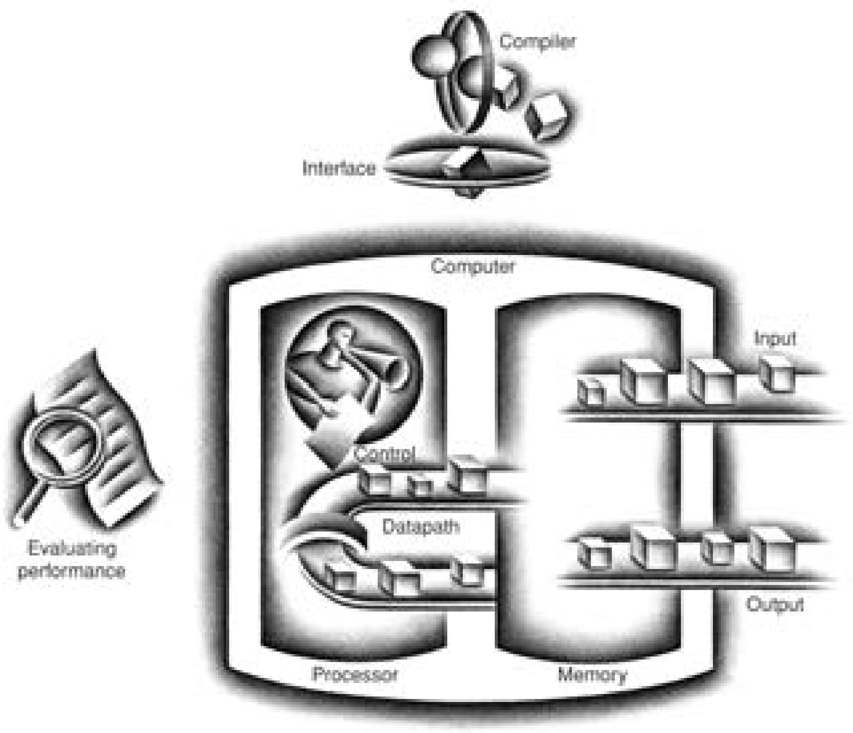
3

Arithmetic for Computers

Numerical precision is the very soul of science.

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

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.

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

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.

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.

$$\begin{array}{lr} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten} \end{array}$$

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

ANSWER

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

```
    0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
–   0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

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

```
    0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+   1111 1111 1111 1111 1111 1111 1111 1010two = -6ten
=   0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

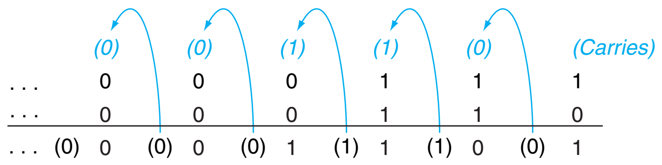


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 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.

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-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 32 bits and the sum is no larger than an operand, the sum must fit in 32 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 $x - y = x + (-y)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Knowing when 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 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when 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 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. This means a borrow occurred from the sign bit. Figure 3.2 shows the combination of operations, operands, and results that indicate an overflow.

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.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The ARM solution is to have two conditional branches that test for overflow: `BVS` (branch if overflow set) and `BVC` (branch if overflow clear. The arithmetic instruction just needs to append `S` to set the condition flags before the branch. Because C ignores overflows, the ARM C compilers would not set the condition flag and branch on overflow. The ARM Fortran compilers, however, would add the test if the operands were signed integers.

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.

Arithmetic Logic Unit (ALU) Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

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

Arithmetic for Multimedia

Since every desktop microprocessor by definition has its own graphical displays, 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 is little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 64-bit adder, a processor could perform simultaneous operations on short vectors of eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned adders was small. These extensions have been called vector or SIMD, for single instruction, multiple data (see Section 2.17 and Chapter 7).

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 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 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. Figure 3.3 shows arithmetic and logical operations found in many multimedia extensions to modern instruction sets.

Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

FIGURE 3.3 Summary of multimedia support for desktop computers.

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 detection of overflow, so today all computers have a way to detect it.

The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half. What ARM instructions would be used?

1. Load with `LDRB`, `LDRH`; arithmetic with `ADD`, `SUB`, `MUL`; then store using `STRB`, `STRH`.
2. Load with `LDRBS`, `LDRHS`; arithmetic with `ADD`, `SUB`, `MUL`; then store using `STRB`, `STRH`.
3. `LDRBS`, `LDRHS`; arithmetic with `ADD`, `SUB`, `MUL`; using `AND` to mask result to 8 or 16 bits after each operation; then store using `STRB`, `STRH`.

Elaboration: The speed of addition is increased by determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the log2 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 C.6 in **Appendix C** on the CD describes.

3.3 Multiplication

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

Anonymous, Elizabethan manuscript, 1570

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

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}:

$$\begin{array}{llr}
\text{Multiplicand} & & 1000_{ten} \\
\text{Multiplier} & \times & 1001_{ten} \\
\hline
 & & 1000\phantom{_{ten}} \\
 & & 0000\phantom{0_{ten}} \\
 & & 0000\phantom{00_{ten}} \\
 & & 1000\phantom{000_{ten}} \\
\hline
\text{Product} & & 1001000_{ten}
\end{array}$$

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

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

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

1. Just place a copy of the multiplicand ($1 \times$ multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 ($0 \times$ 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 3.4 shows the hardware. We have drawn the hardware so that data flows from top to bottom to resemble more closely the paper-and-pencil method.

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

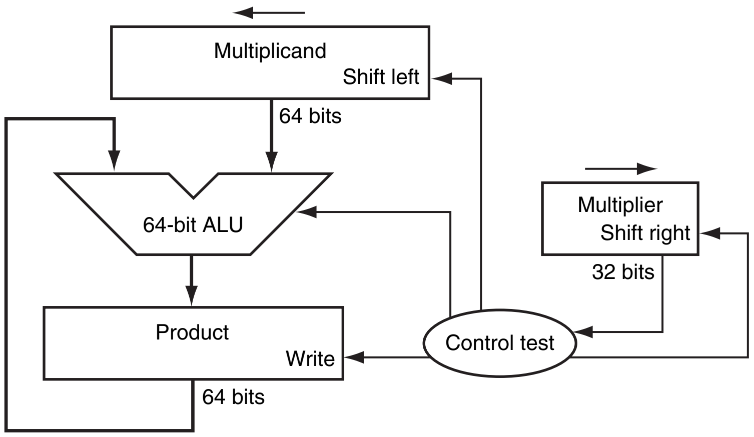


FIGURE 3.4 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix C describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Figure 3.5 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 32 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply

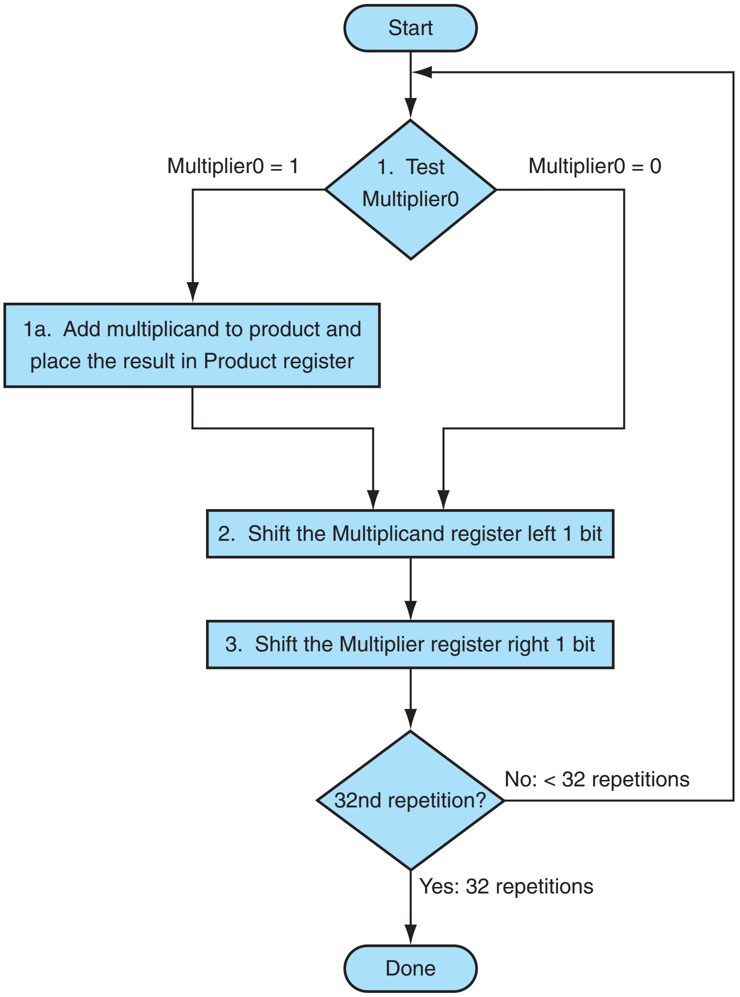

Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

FIGURE 3.5 The first multiplication algorithm, using the hardware shown in Figure 3.4. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Section 1.8) reminds us that even a moderate frequency for a slow operation can limit performance.

This algorithm and hardware are easily refined to take 1 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 3.6 shows the revised hardware.

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.

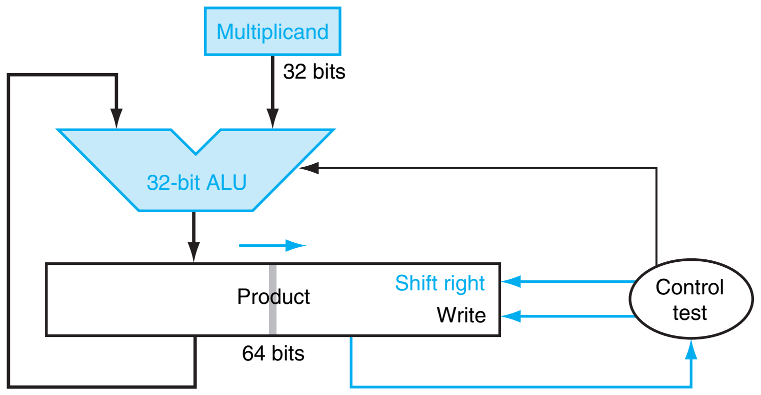


FIGURE 3.6 Refined version of the multiplication hardware. Compare with the first version in Figure 3.4. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.4.)

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 3.7 shows the value of each register for each of the steps labeled according to Figure 3.5, 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.

Signed Multiplication

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

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

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001$\textcircled{1}$	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$\textcircled{1}$	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$\textcircled{0}$	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$\textcircled{0}$	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 3.7 Multiply example using algorithm in Figure 3.5. The bit examined to determine the next step is circled in color.

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 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

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 32 high. An alternative way to organize these 32 additions is in a parallel tree, as Figure 3.8 shows. Instead of waiting for 32 add times, we wait just the $\log_2$ (32) or five 32-bit add times. Figure 3.8 shows how this is a faster way to connect them.

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

Multiply in ARM

ARM provides a multiply instruction (`MUL`) that puts the lower 32 bits of the product into the destination register. Since it doesn't offer the upper 32 bits, there is no difference between signed and unsigned multiplication.

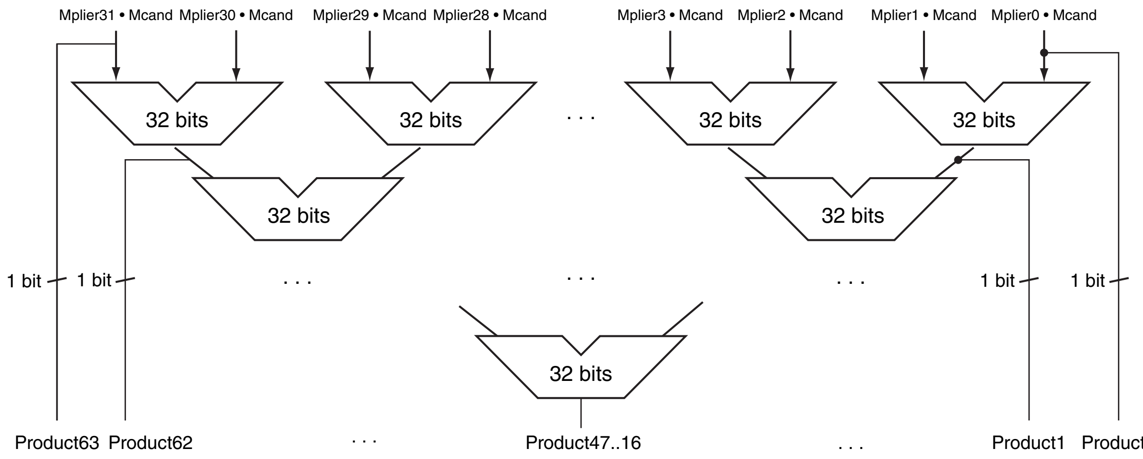


FIGURE 3.8 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.

Summary

Multiplication hardware is simply shifts and add, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2.

Elaboration: The ARM multiply instruction ignores overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits.

Divide et impera.

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

3.4 Division

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

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

$$\begin{array}{lrl}
 & 1001_{ten} & \text{Quotient} \\ \cline{2-2}
\text{Divisor } 1000_{ten} & 1001010_{ten} & \text{Dividend} \\
 & -1000\phantom{000_{ten}} & \\ \cline{2-2}
 & 10\phantom{0_{ten}} & \\
 & 101\phantom{_{ten}} & \\
 & 1010\phantom{_{ten}} & \\
 & -1000\phantom{_{ten}} & \\ \cline{2-2}
 & 10_{ten} & \text{Remainder}
\end{array}$$

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.

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:

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

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

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

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

A Division Algorithm and Hardware

Figure 3.9 shows hardware to mimic our grammar school algorithm. We start with the 32-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 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

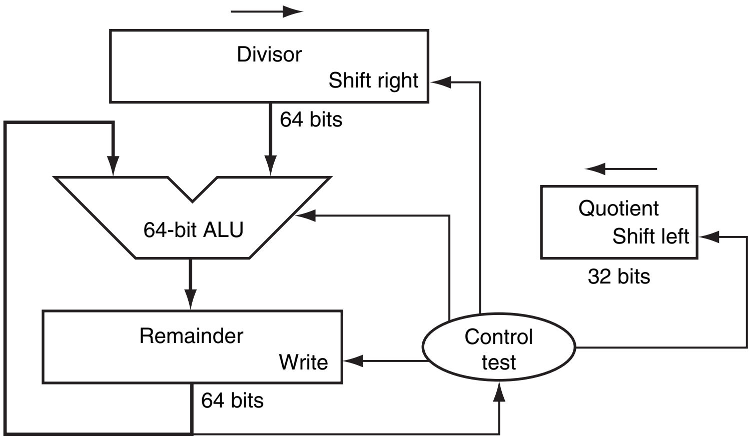


FIGURE 3.9 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized 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 3.10 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

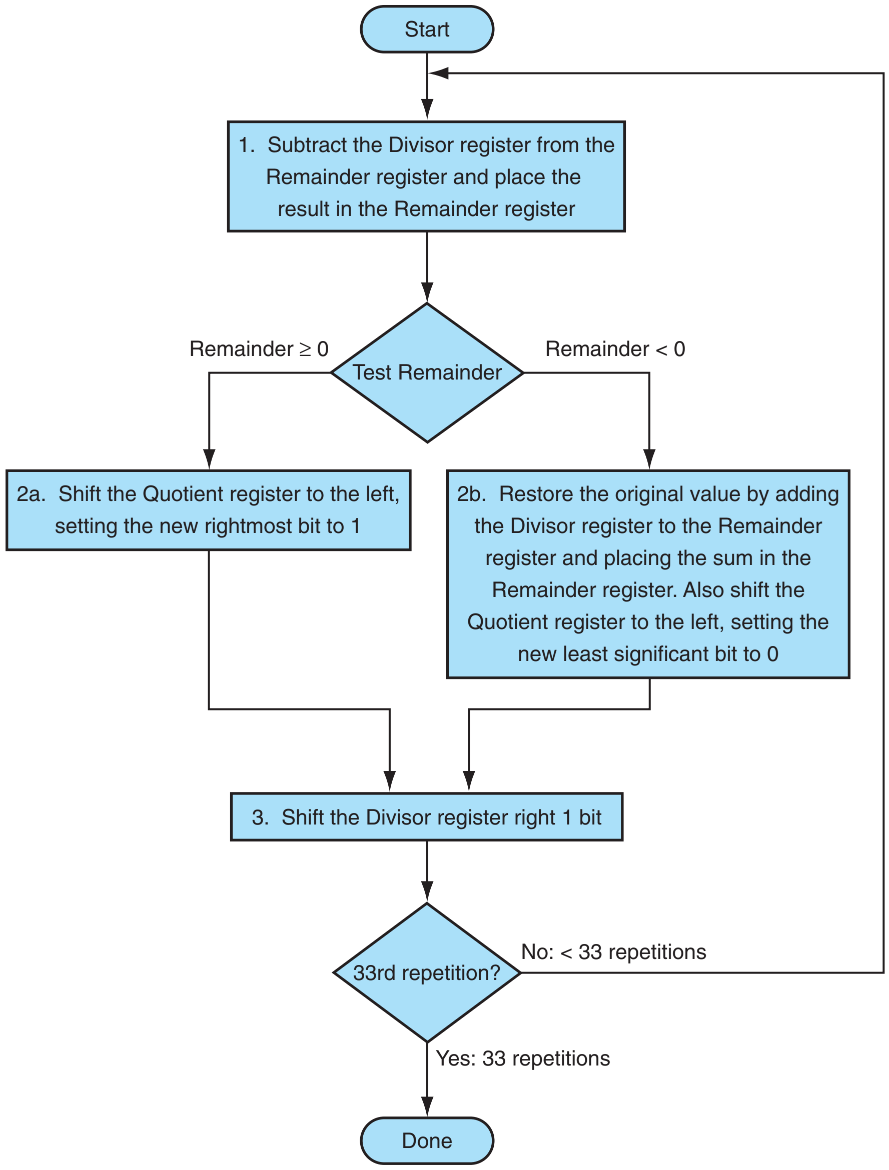

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

FIGURE 3.10 A division algorithm, using the hardware in Figure 3.9. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

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

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 3.12 shows the revised hardware.

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.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 $\Longrightarrow$ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 $\Longrightarrow$ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 $\Longrightarrow$ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem $\geq$ 0 $\Longrightarrow$ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem $\geq$ 0 $\Longrightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.11 Division example using the algorithm in Figure 3.10. The bit examined to determine the next step is circled in color.

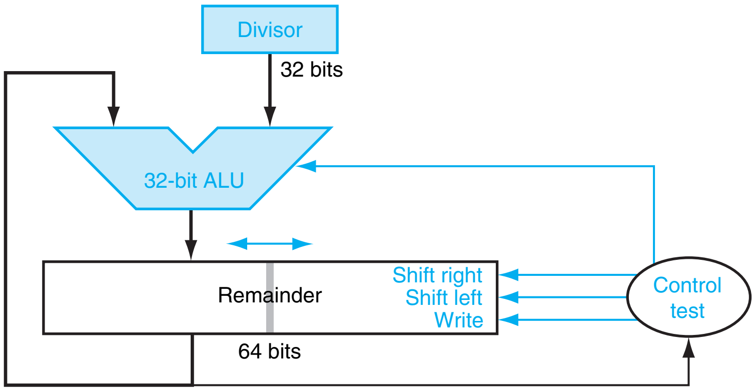


FIGURE 3.12 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.9, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.6, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

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:

Dividend = Quotient × Divisor + 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 ÷ +2: Quotient = +3, Remainder = +1

Checking the results:

7 = 3 × 2 + (+1) = 6 + 1

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

–7 ÷ +2: Quotient = –3

Rewriting our basic formula to calculate the remainder:

Remainder = (Dividend – Quotient × Divisor) = –7 – (–3 × +2) = –7–(–6) = –1

So,

–7 ÷ +2: Quotient = –3, Remainder = –1

Checking the results again:

–7 = –3 × 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 the same 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

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 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to guess 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 guesses. 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 263 in Section 3.8 shows what can happen if the table is incorrect.

Divide in ARM (or lack there of)

While there are many versions of ARM, the classic ARM instruction set had no divide instruction. That tradition continues with the ARMv7A, although the ARMv7R and AMv7M include signed integer divide (SDIV) and unsigned integer divide (UDIV) instructions.

Summary

The common hardware support for multiply and divide allows ARM to provide a single pair of 32-bit registers that are used both for multiply and divide.

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 $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock cycle per step, is explored further in the exercises; the algorithm here is called *restoring* division. A third algorithm that doesn't save the result of the subtract if its negative is called a *nonperforming* division algorithm. It averages one-third fewer arithmetic operations.

Speed gets you nowhere if you're headed the wrong way.

American proverb

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:

$3.14159265\ldots_{\text{ten}}$ (pi)

$2.71828\ldots_{\text{ten}}$ (e)

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^{9}$ (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, $1.0_{\text{ten}} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{\text{ten}} \times 10^{-8}$ and $10.0_{\text{ten}} \times 10^{-10}$ 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:

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

To keep a binary number in 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

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

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

(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.)

A standard scientific notation for reals in 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 the unnecessary leading 0s are replaced by real digits to the right of the binary point.

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 trade-off 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.

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 ARM 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. This representation is called *sign and magnitude*, since the sign is a separate bit from the rest of the 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
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

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 ARM does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give ARM computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{\text{ten}} \times 10^{-38}$ and numbers almost as large as $2.0_{\text{ten}} \times 10^{38}$ 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.

Hardware/Software Interface

One of the optional modes of the IEEE floating point standard is to cause an exception when underflow or overflow occurs. The programmer or the programming environment must then decide what to do.

An **exception**, also called an **interrupt** on many computers, 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; Chapters 5 and 6 describe other situations where exceptions and interrupts occur.)

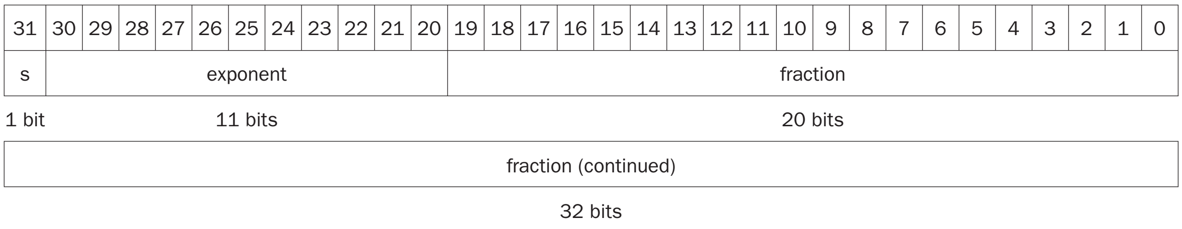
exception Also called **interrupt**. An unscheduled event that disrupts program execution

double precision A floating-point value represented in two 32-bit words.

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

One way to reduce 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.

The representation of a double precision floating-point number takes two ARM words, 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.



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

These formats go beyond ARM. 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 significand, 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 $00 \ldots 00_{two}$ 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 $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

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

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 257.

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 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
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	.	.	.

(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	.	.	.

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

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

to as large as

$\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_{two} \times 2^{+127}$.

Let's show the representation.

Floating-Point Representation

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

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	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

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

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	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Now let's try going the other direction.

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

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	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

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 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: 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. Recent IBM mainframes support IEEE 754 as well as the hex format.

Floating-Point Addition

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

We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

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

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

$$0.016_{\text{ten}} \times 10^{1}$$

Step 2. Next comes the addition of the significands:

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

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

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

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

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 can 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

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

is rounded to four digits in the significand to

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

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.

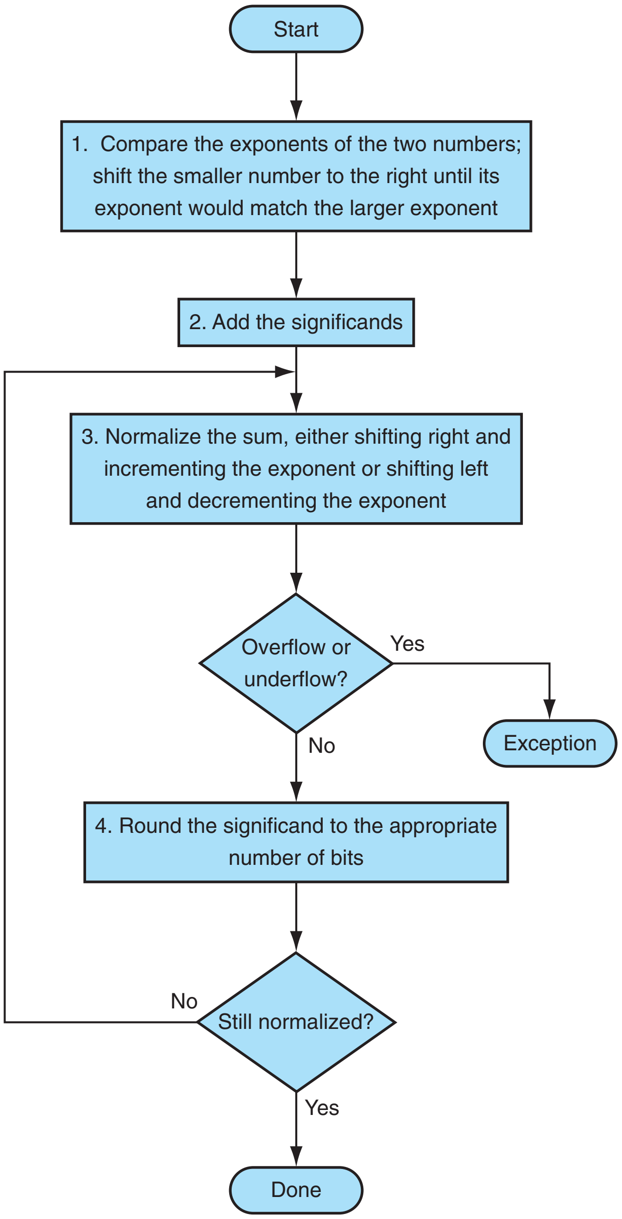

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
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.

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 257). Thus, for single precision, the maximum exponent is 127, and the minimum exponent is −126. The limits for double precision are 1023 and −1022.

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.

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:

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

Now we follow the algorithm:

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

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

Step 2. Add the significands:

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

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

$$\begin{aligned}
0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\
&= 1.000_{\text{two}} \times 2^{-4}
\end{aligned}$$

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:

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

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.

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.

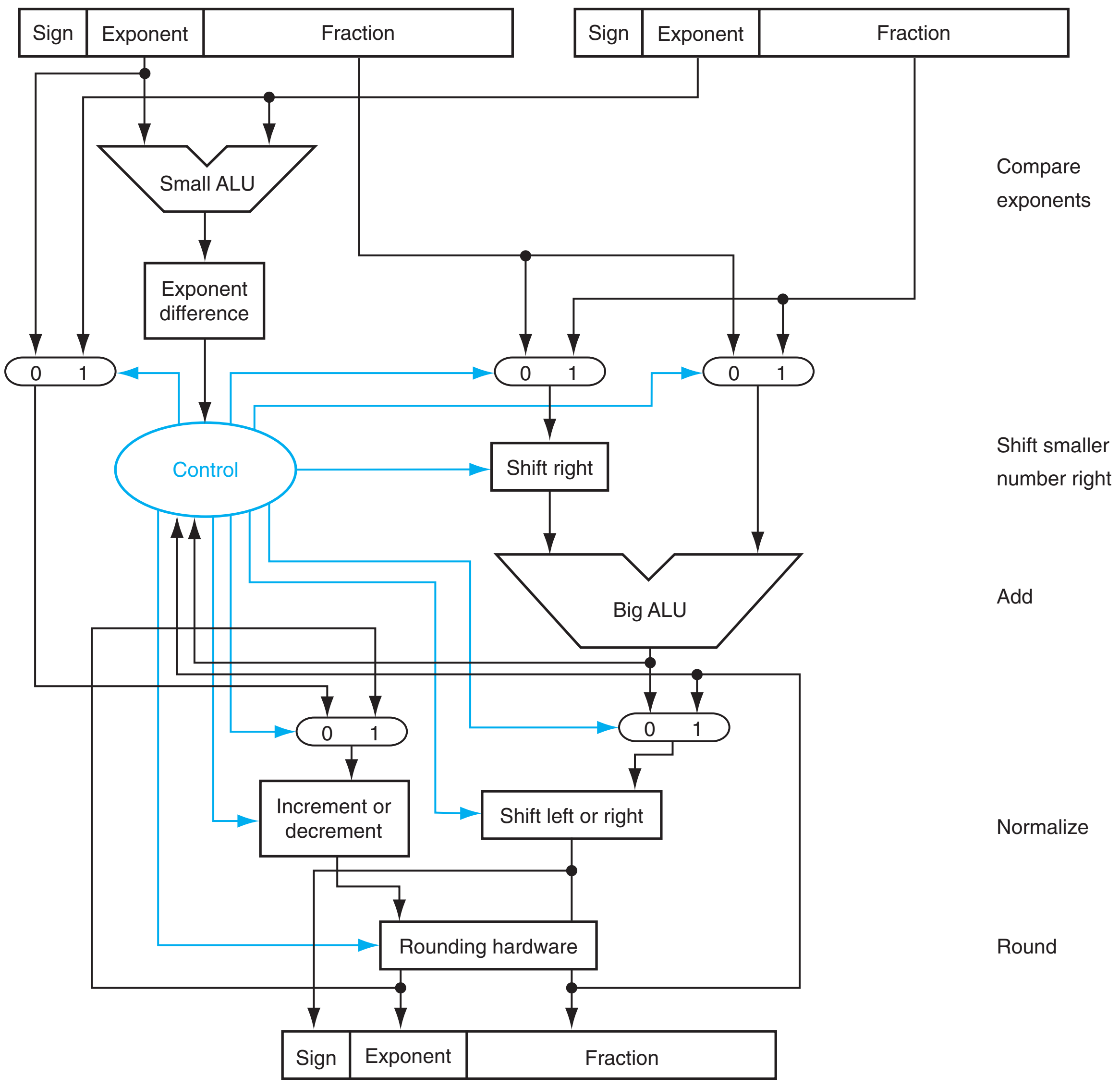


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference 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 final result.

Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times\ \ 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{ten}} \end{array}$$

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:

$$10.212000_{\text{ten}}$$

Assuming that 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:

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

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

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

is rounded to four digits in the significand to

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

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

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

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

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

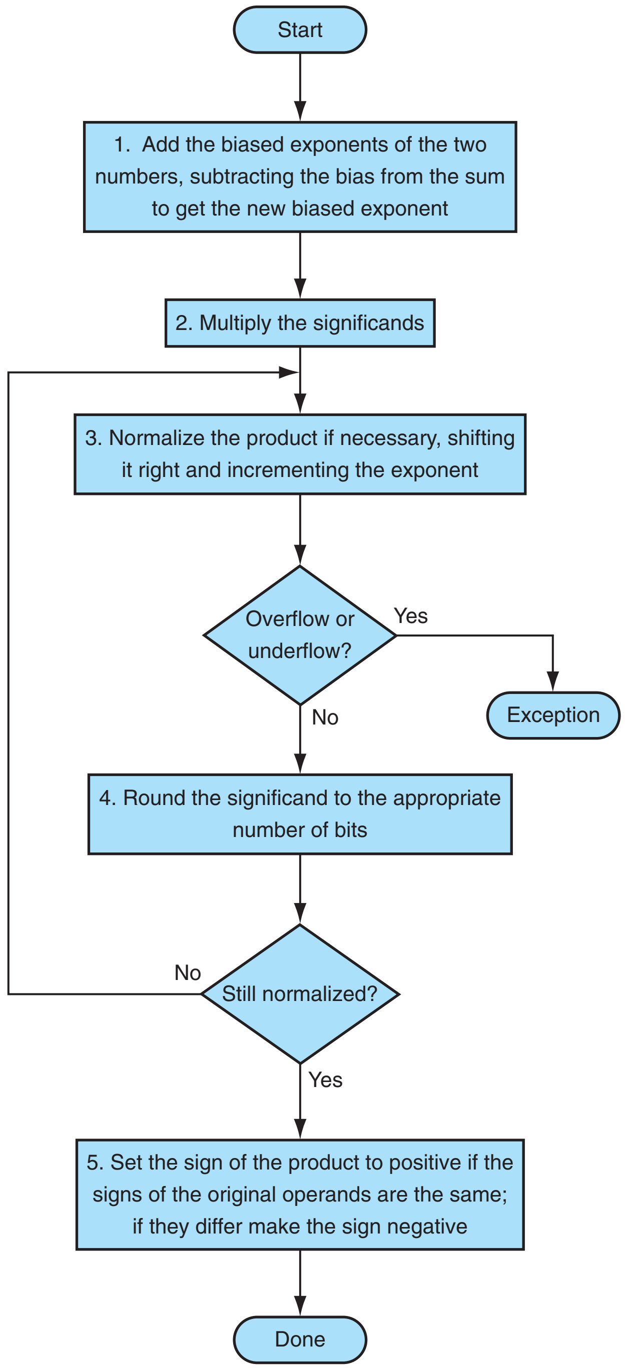

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to 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.

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).

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 $1.000_{\text{two}} \times 2^{-1}$ by $-1.110_{\text{two}} \times 2^{-2}$.

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:

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

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:

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

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

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

Converting to decimal to check our results:

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

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

Floating-Point Instructions in ARM

ARM supports the IEEE 754 single precision and double precision formats with these instructions with the optional VFP set of instructions: These include

- Floating-point *addition, single* (`FADDS`) and *addition, double* (`FADDD`)
- Floating-point *subtraction, single* (`FSUBS`) and *subtraction, double* (`FSUBD`)
- Floating-point *multiplication, single* (`FMULS`) and *multiplication, double* (`FMULD`)
- Floating-point *division, single* (`FDIVS`) and *division, double* (`FDIVD`)
- Floating-point *comparison, single* (`FCMPS`) and *comparison, double* (`FCMPD`)

Floating-point comparison set floating point condition flags. To be able to branch on them, the programmer must first transfer them to the integer condition flags, which is accomplished with the `FMSTAT` instruction. As you might expect, BEQ, BNE, BGT, and BGE test as their namesakes suggest for floating point comparisons. However, because of the way the floating point conditions are mapped to integer condition flags, you test for less than with BMI and less than or equal with BLS.

The ARM designers decided to add 32 separate floating-point registers—called `s0`, `s1`, `s2`, . . . , `s31`—used for single precision. Hence, they included separate loads and stores for floating-point registers: `FLDS` and `FSTS`. The base registers for floating-point data transfers remain integer registers. The ARM code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
FLDS     s4,[sp,#x]  ; Load 32-bit F.P. number into s4
FLDS     s6,[sp,#y]  ; Load 32-bit F.P. number into s6
FADDS    s2,s4,s6    ; s2 = s4 + s6 single precision
FSTS     s2,[sp,#z]  ; Store 32-bit F.P. number from s2
```

FP data transfers have a limited number of addressing modes. The most useful are immediate offset, immediate offset pre-indexed, and immediate offset post-indexed. Note that the offsets are multiplied by 4 to extend the range of the offset, similar to what ARM does for branches. (see Chapter 2).

A double precision register is really just an even-odd pair of single precision registers, using the names `d0`, `d1`, `d2`, . . . , `d15`. Thus, the pair of single precision registers `s16` and `s17` also form the double precision register named `d8`.

Figure 3.17 summarizes the floating-point portion of the ARM architecture revealed in this chapter, with the additions to support floating point shown in color.

ARM floating-point operands

Name	Example	Comments
32 floating-point registers	`s0, s1, s2, . . . , s31` or `d0, d1, d2, . . ., d15`	ARM single-precision floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. ARM uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

ARM floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`FADDS s2,s4,s6`	`s2 = s4 + s6`	FP add (single precision)
	FP subtract single	`FSUBS s2,s4,s6`	`s2 = s4 - s6`	FP sub (single precision)
	FP multiply single	`FMULS s2,s4,s6`	`s2 = s4 × s6`	FP multiply (single precision)
	FP divide single	`FDIVS s2,s4,s6`	`s2 = s4 / s6`	FP divide (single precision)
	FP add double	`FADDD d2,d4,d6`	`d2 = d4 + d6`	FP add (double precision)
	FP subtract double	`FSUBD d2,d4,d6`	`d2 = d4 - d6`	FP sub (double precision)
	FP multiply double	`FMULD d2,d4,d6`	`d2 = d4 × d6`	FP multiply (double precision)
	FP divide double	`FDIVD d2,d4,d6`	`d2 = d4 / d6`	FP divide (double precision)
Data transfer	FP load, single prec.	`FLDS s1,[r1,#100]`	`s1 = Memory[r1 + 400]`	32-bit data to FP register
	FP store, single prec.	`FSTS s1,[r1,#100]`	Memory[`r1` + 400] = `s1`	32-bit data to memory
	FP load, double prec.	`FLDD d1,[r1,#100]`	`d1 = Memory[r1 + 400]`	64-bit data to FP register
	FP store, double prec.	`FSTD d1,[r1,#100]`	Memory[`r1` + 400] = `d1`	64-bit data to memory
Compare	FP compare single	`FCMPS s2,s4`	if (`s2 - s4`)	FP compare less than single precision
	FP compare double	`FCMPD d2,d4`	if (`d2 - d4`)	FP compare less than double precision
	FP Move Status (for conditional branch)	`FMSTAT`	cond. flags = FP cond. flags	Copy FP condition flags to integer condition flags

FIGURE 3.17 ARM floating-point architecture revealed thus far.

Elaboration: VPF version 3 has 32 double-precision floating-point registers.

Hardware/ Software Interface

One issue that architects face in supporting floating-point arithmetic is whether to use 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 separate 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.

Elaboration: The V in VFP stands for vector, as the floating point coprocessor actually supports short vector instructions. Section 7.6 of Chapter 7 describes Vector instruction set architecture. The maximum length of a vectors is either 8 single precision registers or 4 double precision registers. Vector length is determined by a LEN field in the floating point status and control register of ARM (FPSCR). If its set to 0, VFP acts like a scalar floating point instruction. The vector loads and stores can do unit stride accesses. If the LEN and STRIDE fields of the FPSCR are set properly, they can also support a stride of 2. The other pieces commonly found in vector architectures are missing, such as scatter-gather data transfers and conditional execution of vector elements.

To be able to perform the common scalar -vector operations, where an operator using a single scalar variable in combination with each element of the vector, VFP divides the FP registers into those that can make up the vector registers and those that can only be scalar registers. For single precision, registers s0 to s7 are scalar and three sets of registers can be used as vectors; 8 to 15, 16 to 23, and 24 to 31. We use the corresponding registers in double precision to the same end. Scalar are s0 to s3, and the three sets of vectorizable registers are 4 to 7, 8 to 11, and 12 to 15. In both cases, if the destination register is in the first bank, then the whole operation is considered scalar.

Compiling a Floating-Point C Program into ARM Assembly Code

EXAMPLE

Let's convert a temperature in Fahrenheit to Celsius:

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

Assume that the floating-point argument `fahr` is passed in `s12` and the result should go in `s0`. (Unlike integer registers, floating-point register 0 can contain a number.) What is the ARM assembly code?

ANSWER

We assume that the compiler places the three floating-point constants in memory within easy reach of the register r12. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:

      FLDS s16,[r12,  const5] ; s16 = 5.0 (5.0 in memory)
      FLDS s18,[r12,  const9] ; s18 = 9.0 (9.0 in memory)
```

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

```
        FDIVS s16, s16, s18 ; s16 = 5.0 / 9.0
```

(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` (`s12`):

```
        FLDS s18,[r12,const32]; s18 = 32.0
        FSUBS s18, s12, s18 ; s18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `s0` as the return result, and then return

```
        FMULS s0,  s16, s18    ; s0 = (5/9)*(fahr - 32.0)
        MOV pc, lr             ; return
```

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

EXAMPLE

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into ARM

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of X = X + Y * Z. Let's assume X, Y, and Z are all square matrices with 32 elements in each dimension.

```
  void mm (double x[][], double y[][], double z[][])
  {
          int 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)
              x[i][j] = x[i][j] + y[i][k] * z[k][j];
    }
```

The array starting addresses are parameters, so they are in r0, r1, and r2. Assume that the integer variables are in r3, r4, and r5, respectively. What is the ARM assembly code for the body of the procedure?

ANSWER

Note that x[i][j] is used in the innermost loop above. Since the loop index is k, the index does not affect x[i][j], so we can avoid loading and storing x[i][j] each iteration. Instead, the compiler loads x[i][j] into a register outside the loop, accumulates the sum of the products of y[i][k] and z[k][j] in that same register, and then stores the sum into x[i][j] upon termination of the innermost loop.

As we did in Chapter 2, let's rename the registers to make it easier to read and write the code:

```
x               RN 0    ; 1st argument address of x
y               RN 1    ; 2nd argument address of y
z               RN 2    ; 3rd argument address of z
i               RN 3    ; local variable i
j               RN 4    ; local variable j
k               RN 5    ; local variable k
xijAddr         RN 6    ; address of x[i][j]
tempAddr        RN 12   ; address of y[i][j] or z[i][j]
```

Since r4, r5, and r6 are not preserved across the procedure call, we must save them:

```
mm:
SUB   sp,sp,#12      ;  make room on stack for 3 registers
STR   r4, [sp, #8]   ;  save r4 on stack
STR   r5, [sp, #4]   ;  save r5 on stack
STR   r6,  [sp, #0] ;  save r6 on stack
```

The body of the procedure starts with initializing the three *for* loop variables:

```
         MOV   i,  0   ;  i = 0; initialize 1st for loop
 L1:     MOV   j,  0   ;  j = 0; restart 2nd for loop
 L2:     MOV   k,  0   ;  k = 0; restart 3rd for loop
```

To calculate the address of x[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-dimension 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 as part of adding the second index to select the jth element of the desired row:

```
ADD  xijAddr, j, i, LSL #5 ; xijAddr = 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 3 as part of adding this sum to the base address of x, giving the address of x[i][j]:

```
ADD  xijAddr, x, xijAddr, LSL #3 ; xijAddr = byte
 ;                                address of x[i][j]
```

We then load the double precision number x[i][j] into s4:

```
FLDD   s4, [xijAddr,#0]   ; s4 = 8 bytes of x[i][j]
```

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

```
L3:   ADD tempAddr, j , k, LSL #5 ; tempAddr = k *
       ;    size(row) + j
      ADD tempAddr, z, tempAddr, LSL #3
       ;   tempAddr=byte address of z[k][j]
      FLDD s16, [tempAddr,#0]   ; s16 = 8 bytes of
        z[k][j]
```

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

```
ADD tempAddr, k, i, LSL #5      ; tempAddr = i *
 ;   size(row) + k
ADD tempAddr, y, tempAddr, LSL #3      ; tempAddr=byte
 ;   address of y[i][k]
FLDD s18, [tempAddr,#0]   ; s18 = 8 bytes of y[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of y and z located in registers s18 and s16, and then accumulate the sum in s4.

```
FMULD s16, s18, s16   ; s16 = y[i][k] * z[k][j]
FADDD s4, s4, s16     ; s4 = x[i][j]+ y[i][k] * z[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 s4 into x[i][j].

```
ADD     k, k, #1              ; k = k + 1
CMP     k, #32
BLT     L3                    ; if (k < 32) go to L3
FSTD    s4, [xijAddr,#0]      ; x[i][j] = s4
```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is less than 32 and exiting if the index is 32.

```
ADD  j, j, #1             ; j = j + 1
CMP  j, #32
BLT  L2 ; if (j < 32) go to L2
ADD  i, i, #1             ; i = i + 1
CMP  i, #32
BLT  L1 ; if (i < 32) go to L1
```

The end of the procedure restores r4, r5, and r6 from the stack, and then returns to the callee.

Elaboration: The array layout discussed in the example, called *row-major order,* is used by C and many other programming languages. 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 *coprocessors,* Today a coprocessor function may mean its optional for some members of the family, in that lower cost chips might leave out coprocessors so that the chip can be smaller.

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 find the reciprocal $1/x$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solves 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 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 access. 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, 0 and 1, 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 intermediate 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.

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

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

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 is 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 need 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: PowerPC, SPARC64, and AMD SSE5 architectures 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 standard (see **Section 3.10** on the CD).

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, 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 word. Programmers must remember these limits and write programs accordingly.

C type	Java type	Data transfers	Operations

`int`	`int`	`LDR, STR`	`ADD, SUB, MUL,`
			`AND, ORR, MVN, MOV, CMP`

`unsigned int`	—	`LDR, STR`	`ADD, SUB, MUL,`
			`AND, ORR, MVN, MOV, CMP`

`char`	—	`LDRSB, STRB`	`ADD, SUB, MUL,`
			`AND, ORR, MVN, MOV, CMP`

—	`char`	`LDRSH, STRB`	`ADD, SUB, MUL,`
			`AND, ORR, MVN, MOV, CMP`

`float`	`float`	`FLDS, FSTS`	`FADDS, FSUBS, FMULS, FDIVS, FCMPS`
`double`	`double`	`FLDD, FSTD`	`FADDD, FSUBD, FMULD, FDIVD, FCMPD`

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 ARM 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

Suppose there was a 16-bit IEEE 754 floating-point format with five exponent bits. What would be the likely range of numbers it could represent?

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

Elaboration: To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. Hence, the full ARM instruction set has many flavors of compares to support NaNs. (Java does not support unordered compares.)

In an attempt to squeeze every last 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 significand. 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

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

but the smallest single precision denormalized number is

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

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: Associativity

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 1 processor or 1000 processors. This assumption holds for two's complement integers, even if the computation overflows. Another way to say this is that integer addition is associative.

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. That is, floating-point addition is not associative.

Testing Associativity of Floating-Point Addition

EXAMPLE

See if $x + (y + z) = (x + y) + z$. For example, suppose $x = -1.5_{\text{ten}} \times 10^{38}$, $y = 1.5_{\text{ten}} \times 10^{38}$, and $z = 1.0$, and that these are all single precision numbers.

ANSWER

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, as we shall see:

$$\begin{aligned} x + (y + z) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\ &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\ &= 0.0 \end{aligned}$$

$$\begin{aligned} (x + y) + z &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\ &= (0.0_{\text{ten}}) + 1.0 \\ &= 1.0 \end{aligned}$$

Therefore $x + (y + z) \neq (x + y) + z$, so floating-point addition is not associative.

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{\text{ten}} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{\text{ten}} \times 10^{38} + 1.0$ is still $1.5_{\text{ten}} \times 10^{38}$. That is why the sum of x, y, and z is 0.0 or 1.0, depending on the order of the floating-point additions, and hence floating-point add is *not* associative.

A more vexing version of this pitfall 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. The unaware parallel programmer may be flummoxed by his or her program getting slightly different answers each time it is run for the same identical code and the same identical input, as the varying number of processors from each run would cause the floating-point sums to be calculated in different orders.

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 same exact 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.

Elaboration: A subtle version of the associativity issue occurs when two processors perform a redundant computation that is executed in different order so they get slightly different answers, although both answers are considered accurate. The bug occurs if a conditional branch compares to a floating-point number and the two processors take different branches when common sense reasoning suggests they should take the same branch.

3.7 Real Stuff: Floating Point in the x86

The main differences between ARM x86 arithmetic instructions are found in floating-point instructions. The x86 floating-point architecture is different between ARM and from all other computers in the world.

The x86 Floating-Point Architecture

The Intel 8087 floating-point coprocessor was announced in 1980. This architecture extended the 8086 with about 60 floating-point instructions.

Intel provided a stack architecture with its floating-point instructions: *loads* push numbers onto the stack, *operations* find operands in the two top elements of the stacks, and *stores* can pop elements off the stack. Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers on-chip below the top of the stack. Thus, a complete stack instruction set is supplemented by a limited set of register-memory instructions.

This hybrid is still a restricted register-memory model, however, since loads always move data to the top of the stack while incrementing the top-of-stack pointer, and stores can only move the top of stack to memory. Intel uses the notation `ST` to indicate the top of stack, and `ST(i)` to represent the *i*th register below the top of stack.

Another novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Unlike the maximum of 64 bits on ARM, the x86 floating-point operands on the stack are 80 bits wide. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. This *double extended precision* is not supported by programming languages, although it has been useful to programmers of mathematical software.

Memory data can be 32-bit (single precision) or 64-bit (double precision) floating-point numbers. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to floating point, and vice versa, for integer loads and stores.

The x86 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer processor so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure 3.18 shows some of the 60 floating-point operations. Note that we get even more combinations when we include the operand modes for these operations. Figure 3.19 shows the many options for floating-point add.

Data transfer	Arithmetic	Compare	Transcendental
`F{I}LD mem/ST(i)`	`F{I}ADD{P} mem/ST(i)`	`F{I}COM{P}`	`FPATAN`
`F{I}ST{P} mem/ST(i)`	`F{I}SUB{R}{P} mem/ST(i)`	`F{I}UCOM{P}{P}`	`F2XM1`
`FLDPI`	`F{I}MUL{P} mem/ST(i)`	`FSTSW AX/mem`	`FCOS`
`FLD1`	`F{I}DIV{R}{P} mem/ST(i)`		`FPTAN`
`FLDZ`	`FSQRT`		`FPREM`
	`FABS`		`FSIN`
	`FRNDINT`		`FYL2X`

FIGURE 3.18 The floating-point instructions of the x86. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the order of the operands in this operation. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations in the first column push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the `FSTSW` instruction, either into the `AX` register or into memory, followed by an `SAHF` instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations. Not all combinations suggested by the notation are provided. Hence, `F{I}SUB{R}{P}` operations represent these instructions found in the x86: `FSUB`, `FISUB`, `FSUBR`, `FISUBR`, `FSUBP`, `FSUBRP`. For the integer subtract instructions, there is no pop (`FISUBP`) or reverse pop (`FISUBRP`).

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is *i*th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces *i*th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.

FIGURE 3.19 The variations of operands for floating-point add in the x86.

The floating-point instructions are encoded using the ESC opcode of the 8086 and the postbyte address specifier (see Figure 2.45). The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit floating point or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

In the past, floating-point performance of the x86 family lagged far behind other computers. As a result, Intel created a more traditional floating-point architecture as part of SSE2.

The Intel Streaming SIMD Extension 2 (SSE2) Floating-Point Architecture

Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands, giving the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE2 registers as floating-point registers like those found in other computers. AMD expanded the number to 16 registers as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.20 summarizes the SSE and SSE2 instructions.

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 operate simultaneously on four singles (PS) or two doubles (PD). This architecture more than doubles performance over the stack architecture.

Data transfer	Arithmetic	Compare
`MOV{A/U}{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 {H/L} {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 3.20 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. We use the curly brackets {} to show optional 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; {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.

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.

Bertrand Russell, *Recent Words on the Principles of Mathematics,* 1901

3.8 Fallacies and Pitfalls

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 *i*th bit, represents the number

$$\ldots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0)$$

Shifting the bits of *x* right by *n* bits would seem to be the same as dividing by 2^n. 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

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}$

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

$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1{,}073{,}741{,}822_{\text{ten}}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. Indeed, ARM has another optional shift for the second operand of data processing instructions that performs an arithmetic right shift: ASR. A 2-bit arithmetic shift right of -5_{ten} produces

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

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

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

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.21). The following is the inside story behind the headlines.

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

BUSINESS
Intel to work on software patch for Pentium bug
Grove says I sorry about Pentium bu
INFO WORLD
Bug Dodge Booed
Flawed Chip Bruises Intel
Investors react, stock plunges
BUSINESS
Intel's Pentium Problem Pers
Pentium woes continue
Faulty FPU flubs math in certain equations
IBM to stop shipping Pentium PCs

FIGURE 3.21 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.

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 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.

The following is a timeline of the Pentium bug morality play.

- *July 1994:* Intel discovers the bug in the Pentium. The actual cost to fix the bug was several hundred thousand dollars. Following normal bug fix procedures, it will take months to make the change, reverify, and put the corrected chip into production. Intel planned to put good chips into production in January 1995, estimating that 3 to 5 million Pentiums would be produced with the bug.
- *September 1994:* A math professor at Lynchburg College in Virginia, Thomas Nicely, discovers the bug. After calling Intel technical support and getting no official reaction, he posts his discovery on the Internet. It quickly gained a following, and some pointed out that even small errors become big when multiplied by big numbers: the fraction of people with a rare disease times the population of Europe, for example, might lead to the wrong estimate of the number of sick people.
- *November 7, 1994: Electronic Engineering Times* puts the story on its front page, which is soon picked up by other newspapers.
- *November 22, 1994:* Intel issues a press release, calling it a "glitch." The Pentium "can make errors in the ninth digit. . . . Even most engineers and financial analysts require accuracy only to the fourth or fifth decimal point. Spreadsheet and word processor users need not worry. . . . There are maybe several dozen people that this would affect. So far, we've only heard from one. . . . [Only] theoretical mathematicians (with Pentium computers purchased before the summer) should be concerned." What irked many was that customers were told to describe their application to Intel, and then *Intel* would decide whether or not their application merited a new Pentium without the divide bug.
- *December 5, 1994:* Intel claims the flaw happens once in 27,000 years for the typical spreadsheet user. Intel assumes a user does 1000 divides per day and multiplies the error rate assuming floating-point numbers are random, which is one in 9 billion, and then gets 9 million days, or 27,000 years. Things begin to calm down, despite Intel neglecting to explain why a typical customer would access floating-point numbers randomly.
- *December 12, 1994:* IBM Research Division disputes Intel's calculation of the rate of errors (you can access this article by visiting *www.mkp.com/books_catalog/cod/links.htm*). IBM claims that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. IBM assumes 5000 divides per second, for 15 minutes, yielding 4.2 million divides per day, and does not assume

random distribution of numbers, instead calculating the chances as one in 100 million. As a result, IBM immediately stops shipment of all IBM personal computers based on the Pentium. Things heat up again for Intel.

- *December 21, 1994:* Intel releases the following, signed by Intel's president, chief executive officer, chief operating officer, and chairman of the board:

> "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?

In April 1997, another floating-point bug was revealed in the Pentium Pro and Pentium II microprocessors. When the floating-point-to-integer store instructions (`fist`, `fistp`) encounter a negative floating-point number that is too large to fit in a 16- or 32-bit word after being converted to integer, they set the wrong bit in the FPO status word (precision exception instead of invalid operation exception). To Intel's credit, this time they publicly acknowledged the bug and offered a software patch to get around it—quite a different reaction from what they did in 1994.

3.9 Concluding Remarks

A side effect of the stored-program computer is that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, instruction, and so on. It is the instruction that operates on the word that determines its meaning.

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. Chapter 4 discusses 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 are part of the inspiration for the field of numerical analysis. The recent 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.

Over the years, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic and IEEE 754 binary floating-point arithmetic are found in the vast majority of computers sold today. For example, every desktop computer sold since this book was first printed follows these conventions.

With the explanation of computer arithmetic in this chapter comes a description of much more of the ARM instruction set. Figure 3.22 lists the ARM instructions

ARM core instructions	Name	Format	ARM arithmetic core	Name	Format
add	ADD	DP	multiply	MUL	DP
subtract	SUB	DP	floating-point add single	FADDS	R
move	MOV	DP	floating-point add double	FADDD	R
AND	AND	DP	floating-point subtract single	FSUBS	R
OR	ORR	DP	floating-point subtract double	FSUBD	R
NOT	MVN	DP	floating-point multiply single	FMULS	R
logical shift left	LSL	DP	floating-point multiply double	FMULD	R
NOT	MVN	DP	floating-point divide single	FDIVS	R
logical shift right	LSR	DP	floating-point divide double	FDIVD	R
load register	LDR	DT	load word to floating-point single	FLDS	I
store register	STR	DT	store word to floating-point single	FSTS	I
load register halfword	LDRH	DT	load word to floating-point double	FLDD	I
store register halfword	STRH	DT	store word to floating-point double	FSTD	I
load register byte	LDRB	DT	floating-point compare single	FCMPS	R
store register byte	STRB	DT	floating-point compare double	FCMPD	R
swap (*atomic update*)	SWP	DT	FP Move Status (for conditional branch)	FMSTAT	
branch on x	BEQ	BR			
(x = eq, ne, lt, le, gt, ge)					
compare	CMP	DP			
branch (always)	B	BR			
branch and link	BL	BR			

FIGURE 3.22 The ARM instruction set. This book concentrates on the instructions like those in the left column.

covered in this chapter and Chapter 2. We call the set of instructions on the left-hand side of the figure the *ARM core*. The instructions on the right we call the *ARM arithmetic core*. For the application version of the ARM Instruction set, floating-point instructions are becoming standard. On the left of Figure 3.23 are the instructions the ARM processor executes that are not found in Figure 3.22. Figure 3.24 gives the popularity of the ARM instructions for SPEC2006 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 0.3% of the instructions executed.

Remaining ARMv3-v6	Name	Remaining ARM VFP	
exclusive or (*Rn* ⊕ *Rm*)	EOR	FP move*(S or D)*	FCPY*F*
bit clear (*Rn* & ~ *Rm*)	BIC	FP convert integer to FP *(S or D)*	F*S*ITO*F*
arithmetic shift right (operation)	ASR	FP convert FP *(S or D)* to integer	FTO*S*I*F*
rotate right (operation)	ROR	FP square root *(S or D)*	FSQRT*F*
count leading zeros	CLZ	FP absolute value *(S or D)*	FABS*F*
reverse subtract	RSB	FP negate *(S or D)*	FNEG*F*
add with carry	ADC	FP convert *(S or D)*	FCVT*FF*
sutract with carry	SBC	FP compare w. exceptions *(S or D)*	FCMPE*F*
reverse sutract with carry	RSC	FP compare to zero w. exceptions *(S or D)*	FCMPEZ*F*
load register signed byte	LDRSB	move from SP FP to integer	FMRS
load register signed halfword	LDRSH	move to SP FP from integer	FMSR
swap byte (*atomic update*)	SWPB	move from High half DP FP to integer	FMRDH
load multiple	LDM	move to High half DP FP from integer	FMDHR
store multiple	STM	move from Low half DP FP to integer	FMRDL
compare nagative (*Rn* + *Rm*)	CMN	move to Low half DP FP from integer	FMDLR
test equal (*Rn* ⊕ *Rm*)	TEQ	load multiple FP	LDM*F*
test equal (*Rn* & *Rm*)	TST	store multiple FP	STM*F*
multiply and add	MLA	multiply and add	FMAC*F*
long multiply - 64 bit (S or *Uns.)*	*S*MULL	multiply and subtract	FMSC*F*
long multiply and add (S or *Uns.)*	*S*MLAL	negated multiply and add	FNMAC*F*
load byte with user priviledge mode	LDRBT	negated multiply and subtract	FNMSC*F*
load word with user priviledge mode	LDRT		
store byte with user priviledge mode	STRBT		
store word with user priviledge mode	STRT		
coprocessor data operation	CDP		
load coprocessor register	LDC		
store coprocessor register	STC		
move coprocessor register to regiister	MRC		
move register to coprocessor regiister	MCR		
move status register to regiister	MRS		
move register to status regiister	MSR		
breakpoint *(cause exception)*	BKPT		
software inter. *(cause exception)*	SWI		

FIGURE 3.23 Remaining ARM instructions. *F* means single (S) or double (D) precision floating-point instructions, and *S* means signed (S) and unsigned (U) versions. The underscore represents the letter to include to represent that datatype.

Core ARM	Name	Integer	Fl. pt.	Arithmetic core + ARMv4	Name	Integer	Fl. pt.
add	ADD	14.2%	10.7%	FP add double	add.d	0.0%	10.6%
subtract	SUB	2.2%	0.6%	FP subtract double	sub.d	0.0%	4.9%
and	AND	0.9%	0.3%	FP multiply double	mul.d	0.0%	15.0%
or	ORR	5.0%	1.4%	FP divide double	div.d	0.0%	0.2%
not	MVN	0.4%	0.2%	FP add single	add.s	0.0%	1.5%
move	MOV	10.4%	3.4%	FP subtract single	sub.s	0.0%	1.8%
load register	LDR	18.6%	5.8%	FP multiply single	mul.s	0.0%	2.4%
store register	STR	7.6%	2.0%	FP divide single	div.s	0.0%	0.2%
load register byte	LDRB	3.7%	0.1%	load word to FP double	l.d	0.0%	17.5%
store register byte	STRB	0.6%	0.0%	store word to FP double	s.d	0.0%	4.9%
conditional branch	Bcc	17.0%	4.0%	load word to FP single	l.s	0.0%	4.2%
branch and link	BL	0.7%	0.2%	store word to FP single	s.s	0.0%	1.1%
compare	CMP	17.5%	3.5%	floating-point compare double	c.x.d	0.0%	0.6%
				multiply	mul	0.0%	0.2%
				load half	lhu	1.3%	0.0%
				store half	sh	0.1%	0.0%

FIGURE 3.24 The frequency of the ARM instructions for SPEC2006 integer and floating point. All instructions that accounted for at least 1% of the instructions are included in the table. Extrapolated from measurements of MIPS programs.

Note that although programmers and compiler writers have a rich menu of options, ARM core instructions dominate integer SPEC2006 execution, and the integer core plus arithmetic core dominate SPEC2006 floating point, as the table below shows.

Instruction subset	Integer	Fl. pt.
ARM core	98%	31%
ARM arithmetic core	2%	66%
Remaining ARM	0%	3%

For the rest of the book, we concentrate on the core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. We use the MIPS instruction set core, although there are a few differences between it and ARM, as it is a little simpler. However, the same techniques apply to ARM. As you can see, the core includes the most popular instructions; be assured that understanding a computer that runs the core will give you sufficient background to understand even more ambitious computers.

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

3.10 Historical Perspective and Further Reading

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 Section 3.10.

3.11 Exercises

Contributed by Matthew Farrens, UC Davis

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

Exercise 3.1

The book shows how to add and subtract binary and decimal numbers. However, other numbering systems were also very popular when dealing with computers. Octal (base 8) numbering system was one of these. The following table shows pairs of octal numbers.

	A	B
a.	5323	2275
b.	0147	3457

3.1.1 [5] <3.2> What is the sum of A and B if they represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

3.1.2 [5] <3.2> What is the sum of A and B if they represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

3.1.3 [10] <3.2> Convert A into a decimal number, assuming it is unsigned. Repeat assuming it stored in sign-magnitude format. Show your work.

The following table also shows pairs of octal numbers.

	A	B
a.	2762	2032
b.	2646	1066

3.1.4 [5] <3.2> What is A – B if they represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

3.1.5 [5] <3.2> What is A – B if they represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

3.1.6 [10] <3.2> Convert A into a binary number. What makes base 8 (octal) an attractive numbering system for representing values in computers?

Exercise 3.2

Hexadecimal (base 16) is also a commonly used numbering system for representing values in computers. In fact, it has become much more popular than octal. The following table shows pairs of hexadecimal numbers.

	A	B
a.	0D34	DD17
b.	BA1D	3617

3.2.1 [5] <3.2> What is the sum of A and B if they represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

3.2.2 [5] <3.2> What is the sum of A and B if they represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

3.2.3 [10] <3.2> Convert A into a decimal number, assuming it is unsigned. Repeat assuming it stored in sign-magnitude format. Show your work.

The following table also shows pairs of hexadecimal numbers.

	A	B
a.	BA7C	241A
b.	AADF	47BE

3.2.4 [5] <3.2> What is A – B if they represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

3.2.5 [5] <3.2> What is A – B if they represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

3.2.6 [10] <3.2> Convert A into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

Exercise 3.3

Overflow occurs when a result is too large to be represented accurately given a finite word size. Underflow occurs when a number is too small to be represented correctly—a negative result when doing unsigned arithmetic, for example. (The case when a positive result is generated by the addition of two negative integers is also referred to as underflow by many, but in this textbook, that is considered an overflow). The following table shows pairs of decimal numbers.

	A	B
a.	69	90
b.	102	44

3.3.1 [5] <3.2> Assume A and B are unsigned 8-bit decimal integers. Calculate A – B. Is there overflow, underflow, or neither?

3.3.2 [5] <3.2> Assume A and B are signed 8-bit decimal integers stored in sign-magnitude format. Calculate A + B. Is there overflow, underflow, or neither?

3.3.3 [5] <3.2> Assume A and B are signed 8-bit decimal integers stored in sign-magnitude format. Calculate A – B. Is there overflow, underflow, or neither?

The following table also shows pairs of decimal numbers.

	A	B
a.	200	103
b.	247	237

3.3.4 [10] <3.2> Assume A and B are signed 8-bit decimal integers stored in two's-complement format. Calculate A + B using saturating arithmetic. The result should be written in decimal. Show your work.

3.3.5 [10] <3.2> Assume A and B are signed 8-bit decimal integers stored in two's-complement format. Calculate A – B using saturating arithmetic. The result should be written in decimal. Show your work.

3.3.6 [10] <3.2> Assume A and B are unsigned 8-bit integers. Calculate A + B using saturating arithmetic. The result should be written in decimal. Show your work.

Exercise 3.4

Let's look in more detail at multiplication. We will use the numbers in the following table.

	A	B
a.	50	23
b.	66	04

3.4.1 [20] <3.3> Using a table similar to that shown in Figure 3.7, calculate the product of the octal unsigned 6-bit integers A and B using the hardware described in Figure 3.4. You should show the contents of each register on each step.

3.4.2 [20] <3.3> Using a table similar to that shown in Figure 3.7, calculate the product of the hexadecimal unsigned 8-bit integers A and B using the hardware described in Figure 3.6. You should show the contents of each register on each step.

3.4.3 [60] <3.3> Write an ARM assembly language program to calculate the product of unsigned integers A and B, using the approach described in Figure 3.4.

The following table shows pairs of octal numbers.

	A	B
a.	54	67
b.	30	07

3.4.4 [30] <3.3> When multiplying signed numbers, one way to get the correct answer is to convert the multiplier and multiplicand to positive numbers, save the original signs, and then adjust the final value accordingly. Using a table similar to that shown in Figure 3.7, calculate the product of A and B using the hardware described in Figure 3.4. You should show the contents of each register on each step, and include the step necessary to produce the correctly signed result. Assume A and B are stored in 6-bit sign-magnitude format.

3.4.5 [30] <3.3> When shifting a register one bit to the right, there are several ways to decide what the new entering bit should be. It can always be a 0, or always a 1, or the incoming bit could be the one that is being pushed out of the right side (turning a shift into a rotate), or the value that is already in the leftmost bit can simply be retained (called an arithmetic shift right, because it preserves the sign of the number that is being shifted.) Using a table similar to that shown in Figure 3.7, calculate the product of the 6-bit two's-complement numbers A and B using the hardware described in Figure 3.6. The right shifts should be done using an arithmetic shift right. Note that the algorithm described in the text will need to be modified slightly to make this work—in particular, things must be done differently if the multiplier is negative. You can find details by searching the Web. Show the contents of each register on each step.

3.4.6 [60] <3.3> Write an ARM assembly language program to calculate the product of the signed integers A and B. State if you are using the approach given in 3.4.4 or 3.4.5.

Exercise 3.5

For many reasons, we would like to design multipliers that require less time. Many different approaches have been taken to accomplish this goal. In the following table, A represents the bit width of an integer, and B represents the number of time units (tu) taken to perform a step of an operation.

	A	B
a.	4	3 tu
b.	32	7 tu

3.5.1 [10] <3.3> Calculate the time necessary to perform a multiply using the approach given in Figures 3.4 and 3.5 if an integer is A bits wide and each step of the operation takes B 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.

3.5.2 [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 A bits wide and an adder takes B time units.

3.5.3 [20] <3.3> Calculate the time necessary to perform a multiply using the approach given in Figure 3.8, if an integer is A bits wide and an adder takes B time units.

Exercise 3.6

In this exercise we will look at a couple of other ways to improve the performance of multiplication, based primarily on doing more shifts and fewer arithmetic operations. The following table shows pairs of hexadecimal numbers.

	A	B
a.	24	c9
b.	41	18

3.6.1 [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 A · B using shifts and adds/subtracts. Assume that A and B are 8-bit unsigned integers.

3.6.2 [20] <3.3> Show the best way to calculate A · B using shift and adds, if A and B are 8-bit signed integers stored in sign-magnitude format.

3.6.3 [60] <3.3> Write an ARM assembly language program that performs a multiplication on signed integers using shift and adds, as described in 3.6.1.

The following table shows further pairs of hexadecimal numbers.

	A	B
a.	42	36
b.	9F	8E

3.6.4 [30] <3.3> Booth's algorithm is another approach to reducing the number of arithmetic operations necessary to perform a multiplication. This algorithm has been around for years, and details about how it works are available on the Web. Basically, it assumes that a shift takes less time than an add or subtract, and uses this fact to reduce the number of arithmetic operations necessary to perform a multiply. It works by identifying runs of 1s and 0s, and performing shifts during the runs. Find a description of the algorithm and explain in detail how it works.

3.6.5 [30] <3.3> Show the step-by-step result of multiplying A and B, using Booth's algorithm. Assume A and B are 8-bit two's-complement integers, stored in hexadecimal format.

3.6.6 [60] <3.3> Write an ARM assembly language program to perform the multiplication of A and B using Booth's algorithm.

Exercise 3.7

Let's look in more detail at division. We will use the octal numbers in the following table.

	A	B
a.	50	23
b.	25	44

3.7.1 [20] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using the hardware described in Figure 3.9. You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers.

3.7.2 [30] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using the hardware described in Figure 3.12. 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 3.10. 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 3.12 implies the remainder register can be shifted either direction).

3.7.3 [60] <3.4> Write an ARM assembly language program to calculate A divided by B, using the approach described in Figure 3.9. Assume A and B are unsigned 6-bit integers.

The following table shows further pairs of octal numbers.

	A	B
a.	55	24
b.	36	51

3.7.4 [30] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using the hardware described in Figure 3.9. You should show the contents of each register on each step. Assume A and B are 6-bit signed integers in sign-magnitude format. Be sure to include how you are calculating the signs of the quotient and remainder.

3.7.5 [30] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using the hardware described in Figure 3.12. You should show the contents of each register on each step. Assume A and B are 6-bit signed integers in sign-magnitude format. Be sure to include how you are calculating the signs of the quotient and remainder.

3.7.6 [60] <3.4> Write a ARM assembly language program to calculate A divided by B, using the approach described in Figure 3.12. Assume A and B are signed integers.

Exercise 3.8

Figure 3.10 describes a restoring division algorithm, because when subtracting the divisor from the remainder produces a negative result, the divisor is added back to the remainder (thus restoring the value). However, there are other algorithms that have been developed that eliminate the extra addition. Many references to these algorithms are easily found on the Web. We will explore these algorithms using the pairs of octal numbers in the following table.

	A	B
a.	75	12
b.	52	37

3.8.1 [30] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using nonrestoring division. You should show the contents of each register on each step. Assume A and B are 6-bit unsigned integers.

3.8.2 [60] <3.4> Write an ARM assembly language program to calculate A divided by B using nonrestoring division. Assume A and B are 6-bit signed (two's-complement) integers.

3.8.3 [60] <3.4> How does the performance of restoring and non-restoring division compare? Demonstrate by showing the number of steps necessary to calculate A divided by B using each method. Assume A and B are 6-bit signed (sign-magnitude) integers. Writing a program to perform the restoring and nonrestoring divisions is acceptable.

The following table shows further pairs of octal numbers.

	A	B
a.	17	14
b.	70	23

3.8.4 [30] <3.4> Using a table similar to that shown in Figure 3.11, calculate A divided by B using nonperforming division. You should show the contents of each register on each step. Assume A and B are 6-bit unsigned integers.

3.8.5 [60] <3.4> Write a ARM assembly language program to calculate A divided by B using nonperforming division. Assume A and B are 6-bit two's complement signed integers.

3.8.6 [60] <3.4> How does the performance of non-restoring and nonperforming division compare? Demonstrate by showing the number of steps necessary to calculate A divided by B using each method. Assume A and B are signed 6-bit integers, stored in sign-magnitude format. Writing a program to perform the nonperforming and non-restoring divisions is acceptable.

Exercise 3.9

Division is so time-consuming and difficult that the CRAY T3E Fortran Optimization guide states, “The best strategy for division is to avoid it whenever possible.” This exercise looks at the following different strategies for performing divisions.

a.	restoration division
b.	SRT division

3.9.1 [30] <3.4> Describe the algorithm in detail.

3.9.2 [60] <3.4> Use a flow chart (or a high-level code snippet) to describe how the algorithm works.

3.9.3 [60] <3.4> Write a ARM assembly language program to perform a division using the algorithm.

Exercise 3.10

In a Von Neumann architecture, groups of bits have no intrinsic meanings by themselves. What a bit pattern represents depends entirely on how it is used. The following table shows bit patterns expressed in hexademical notation.

a.	0x24A60004
b.	0xAFBF0000

3.10.1 [5] <3.5> What decimal number does the bit pattern represent if it is a two’s-complement integer? An unsigned integer?

3.10.2 [10] <3.5> If this bit pattern is placed into the Instruction Register, what ARM instruction will be executed?

3.10.3 [10] <3.5> What decimal number does the bit pattern represent if it is a floating point number? Use the IEEE 754 standard.

The following table shows decimal numbers.

a.	–1609.5
b.	–938.8125

3.10.4 [10] <3.5> Write down the binary representation of the decimal number, assuming the IEEE 754 single precision format.

3.10.5 [10] <3.5> Write down the binary representation of the decimal number, assuming the IEEE 754 double precision format.

3.10.6 [10] <3.5> Write down the binary representation of the decimal number assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

Exercise 3.11

In the IEEE 754 floating point standard the exponent is stored in "bias" (also known as "Excess-N") format. This approach was selected because we want an all zero pattern to be as close to zero as possible. Because of the use of a hidden 1, if we were to represent the exponent in two's-complement format an all-zero pattern would actually be the number 1! (Remember, anything raised to the zeroth power is 1, so $1.0^0 = 1$.) There are many other aspects of the IEEE 754 standard that exist in order to help hardware floating point units work more quickly. However, in many older machines floating point calculations were handled in software, and therefore other formats were used. The following table shows decimal numbers.

a.	5.00736125×10^5
b.	$-2.691650390625 \times 10^{-2}$

3.11.1 [20] <3.5> Write down the binary bit pattern assuming a format similar to that employed by the DEC PDP-8 (left most 12 bits are the exponent stored as a two's-complement number, and the rightmost 24 bits are the mantissa 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.

3.11.2 [20] <3.5> NVIDIA has a "half" format, which is similar to IEEE 754 except that it is only 16 bits wide. The left-most bit is still the sign bit, the exponent is 5 bits wide and stored in excess-16 format, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern assuming this format. Comment on how the range and accuracy of this 16-bit pattern compares to the single precision IEEE 754 standard.

3.11.3 [20] <3.5> The Hewlett-Packard 2114, 2115, and 2116 used a format with the left-most 16 bits being the mantissa stored in two's-complement format, followed by another 16-bit field which had the leftmost 8 bits an extension of the mantissa (making the mantissa 24 bits long), and the right-most 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 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.

The following table shows pairs of decimal numbers.

	A	B
a.	-1278×10^{3}	-3.90625×10^{-1}
b.	2.3109375×10^{1}	$6.391601562 \times 10^{-1}$

3.11.4 [20] <3.5> Calculate the sum of A and B by hand, assuming A and B are stored in the 16-bit NVIDIA format described in 3.11.2 (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.

3.11.5 [60] <3.5> Write an ARM assembly language program to calculate the sum of A and B, assuming they are stored in the 16-bit NVIDIA format described in 3.11.2 (and also described in the text). Assume 1 guard, 1 round and 1 sticky bit, and round to the nearest even.

3.11.6 [60] <3.5> Write an ARM assembly language program to calculate the sum of A and B, assuming they are stored using the format described in 3.11.1. Now modify the program to calculate the sum assuming the format described in 3.11.3. Which format is easier for a programmer to deal with? How do they each compare to the IEEE 754 format? (Do not worry about sticky bits for this question.)

Exercise 3.12

Floating-point multiplication is even more complicated and challenging than floating-point addition, and both pale in comparison to floating-point division.

	A	B
a.	5.66015625×10^{0}	8.59375×10^{0}
b.	6.18×10^{2}	5.796875×10^{1}

3.12.1 [30] <3.5> Calculate the product of A and B by hand, assuming A and B are stored in the 16-bit NVIDIA format described in 3.11.2 (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; 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 3.4 through 3.6. Indicate if there is overflow or underflow. Write your answer as a 16-bit pattern, 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?

3.12.2 [60] <3.5> Write an ARM assembly language program to calculate the product of A and B, assuming they are stored in IEEE 754 format. Indicate if there is overflow or underflow. (Remember, IEEE 754 assumes 1 guard, 1 round and 1 sticky bit, and rounds to the nearest even.)

3.12.3 [60] <3.5> Write an ARM assembly language program to calculate the product of A and B, assuming they are stored using the format described in 3.11.1. Now modify the program to calculate the sum assuming the format described in 3.11.3. Which format is easier for a programmer to deal with? How do they each compare to the IEEE 754 format? (Do not worry about sticky bits for this question.)

The following table shows further pairs of decimal numbers.

	A	B
a.	3.264×10^3	6.52×10^2
b.	-2.27734375×10^0	1.154375×10^2

3.12.4 [30] <3.5> Calculate by hand A divided by B. Show all the steps necessary to achieve your answer. Assume there is a guard, round, and sticky bit, and use them if necessary. Write the final answer in both 16-bit floating-point format and in decimal and compare the decimal result to that which you get if you use a calculator.

The Livermore Loops are a set of floating-point intensive kernels taken from scientific programs run at Lawrence Livermore Laboratory. The following table identifies individual kernels from the set.

a.	Livermore Loop 1
b.	Livermore Loop 7

3.12.5 [60] <3.5> Write the loop in ARM assembly language.

3.12.6 [60] <3.5> Describe in detail one technique for performing floating-point division in a digital computer. Be sure to include references to the sources you used.

Exercise 3.13

Operations performed on fixed-point integers behave the way one expects—the commutative, associative, and distributive laws all hold. This is not always the case when working with floating-point numbers, however. Let's first look at the associative law. The following table shows sets of decimal numbers.

	A	B	C
a.	-1.6360×10^{4}	1.6360×10^{4}	1.0×10^{0}
b.	2.865625×10^{1}	4.140625×10^{-1}	1.2140625×10^{1}

3.13.1 [20] <3.2, 3.5, 3.6> Calculate (A + B) + C by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (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 both in 16-bit floating point format and in decimal.

3.13.2 [20] <3.2, 3.5, 3.6> Calculate A + (B + C) by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (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 both in 16-bit floating point format and in decimal.

3.13.3 [10] <3.2, 3.5, 3.6> Based on your answers to 3.13.1 and 3.13.2, does (A + B) + C = A + (B + C)?

The following table shows further sets of decimal numbers.

	A	B	C
a.	4.8828125×10^{-4}	1.768×10^{3}	2.50125×10^{2}
b.	4.721875×10^{1}	2.809375×10^{1}	3.575×10^{1}

3.13.4 [30] <3.3, 3.5, 3.6> Calculate (A · B) · C by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (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 both in 16-bit floating point format and in decimal.

3.13.5 [30] <3.3, 3.5, 3.6> Calculate A · (B · C) by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (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 both in 16-bit floating point format and in decimal.

3.13.6 [10] <3.3, 3.5, 3.6> Based on your answers to 3.13.4 and 3.13.5, does (A · B) · C = A · (B · C)?

Exercise 3.14

The associative law is not the only one that does not always hold in dealing with floating point numbers. There are other oddities that occur as well. The following table shows sets of decimal numbers.

	A	B	C
a.	1.5234375×10^{-1}	2.0703125×10^{-1}	9.96875×10^{1}
b.	-2.7890625×10^{1}	-8.088×10^{3}	1.0216×10^{4}

3.14.1 [30] <3.2, 3.3, 3.5, 3.6> Calculate $A \cdot (B + C)$ by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (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 both in 16-bit floating point format and in decimal.

3.14.2 [30] <3.2, 3.3, 3.5, 3.6> Calculate $(A \times B) + (A \times C)$ by hand, assuming A, B, and C are stored in the 16-bit NVIDIA format described in 3.11.2 (and also described in the text). Assume one guard, one round bit and one sticky bit, and round to the nearest even. Show all the steps, and write your answer both in 16-bit floating-point format and in decimal.

3.14.3 [10] <3.2, 3.3, 3.5, 3.6> Based on your answers to 3.14.1 and 3.14.2, does $(A \cdot B) + (A \cdot C) = A \cdot (B + C)$?

The following table shows pairs, each consisting of a fraction and an integer.

	A	B
a.	1/3	3
b.	–1/7	7

3.14.4 [10] <3.5> Using the IEEE 754 floating point format, write down the bit pattern that would represent A. Can you represent A exactly?

3.14.5 [10] <3.2, 3.3, 3.5, 3.6> What do you get if you add A to itself B times? What is $A \times B$? Are they the same? What should they be?

3.14.6 [60] <3.2, 3.3, 3.4, 3.5, 3.6> What do you get if you take the square root of B and then multiply that value by itself? What should you get? Do for both single and double precision floating point numbers. (Write a program to do these calculations).

Exercise 3.15

Binary numbers are used in the mantissa field, but they do not have to be. IBM used base 16 numbers, for example, in some of their floating point formats. There are other approaches that are possible as well, each with their own particular advantages and disadvantages. The following table shows fractions to be represented in various floating point formats.

a.	1/2
b.	1/9

3.15.1 [10] <3.5, 3.6> Write down the bit pattern in the mantissa assuming a floating point format that uses binary numbers in the mantissa (essentially what you have been doing in this chapter). Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.15.2 [10] <3.5, 3.6> Write down the bit pattern in the mantissa assuming a floating-point format that uses Binary Coded Decimal (base 10) numbers in the mantissa instead of base 2. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.15.3 [10] <3.5, 3.6> Write down the bit pattern assuming that we are using base 15 numbers in the mantissa 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?

3.15.4 [20] <3.5, 3.6> Write down the bit pattern assuming that we are using base 30 numbers in the mantissa 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? Do you see any advantage to using this approach?

Answers to Check Yourself

§3.2, page 219: 3.
§3.5, page 257: 3.