

3

*Numerical precision
is the very soul of
science.*

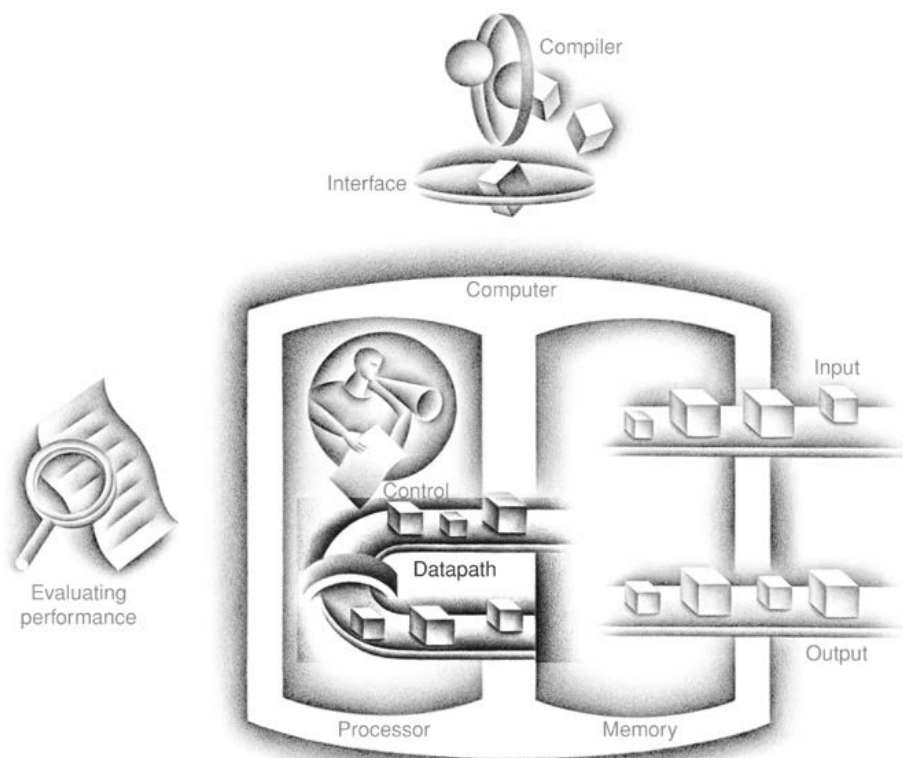
Sir D'arcy Wentworth Thompson,
On Growth and Form, 1917

Arithmetic for Computers

- 3.1 Introduction** 190
- 3.2 Addition and Subtraction** 190
- 3.3 Multiplication** 193
- 3.4 Division** 199
- 3.5 Floating Point** 208
- 3.6 Parallelism and Computer Arithmetic:
Subword Parallelism** 233
- 3.7 Real Stuff: Streaming SIMD Extensions and
Advanced Vector Extensions in x86** 234

3.8	Going Faster: Subword Parallelism and Matrix Multiply	236
3.9	Fallacies and Pitfalls	238
3.10	Concluding Remarks	241
 3.11	Historical Perspective and Further Reading	242
3.12	Self-Study	242
3.13	Exercises	246

The Five Classic Components of a Computer



3.1 Introduction

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

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

The goal of this chapter is to unravel these mysteries—including representation of real numbers, arithmetic algorithms, hardware that follows these algorithms—and the implications of all this for instruction sets. These insights may explain quirks that you have already encountered with computers. Moreover, we show how to use this knowledge to make arithmetic-intensive programs go much faster.

3.2 Addition and Subtraction

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

Subtraction: Addition's Tricky Pal
No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., *Book of Top Ten Lists*, 1990

EXAMPLE

Binary Addition and Subtraction

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

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

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

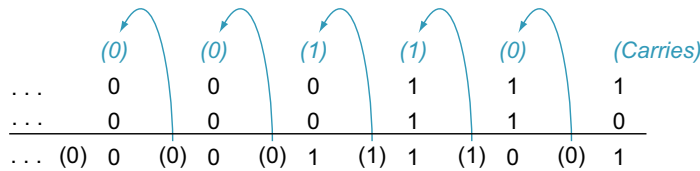


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is $0 + 1 + 1$. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of $1 + 1 + 1$, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is $1 + 0 + 0$, yielding a 1 sum and no carry.

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

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

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

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

ANSWER

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 $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Knowing when an overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? Clearly, adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed.

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.

The lack of a 33rd bit means that when an overflow occurs, the sign bit is set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum means a carry out occurred into the sign bit.

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

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

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

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

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

Hardware/Software Interface

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

Summary

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

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

Check Yourself

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

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

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

3.3 Multiplication

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

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} :

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

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

Anonymous,
Elizabethan manuscript,
1570

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 3.3](#) shows the hardware. We have drawn the hardware so that data flow from top to bottom to resemble more closely the paper-and-pencil method.

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.

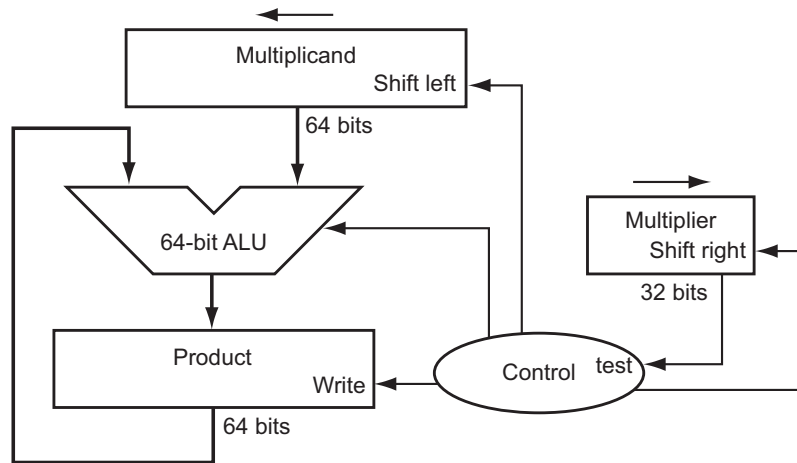


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted 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 3.4 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took one clock cycle, this algorithm would require almost 200 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take several clock cycles without significantly affecting performance. However, Amdahl's Law (see Section 1.10) reminds us that even a moderate frequency for a slow operation can limit performance.

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

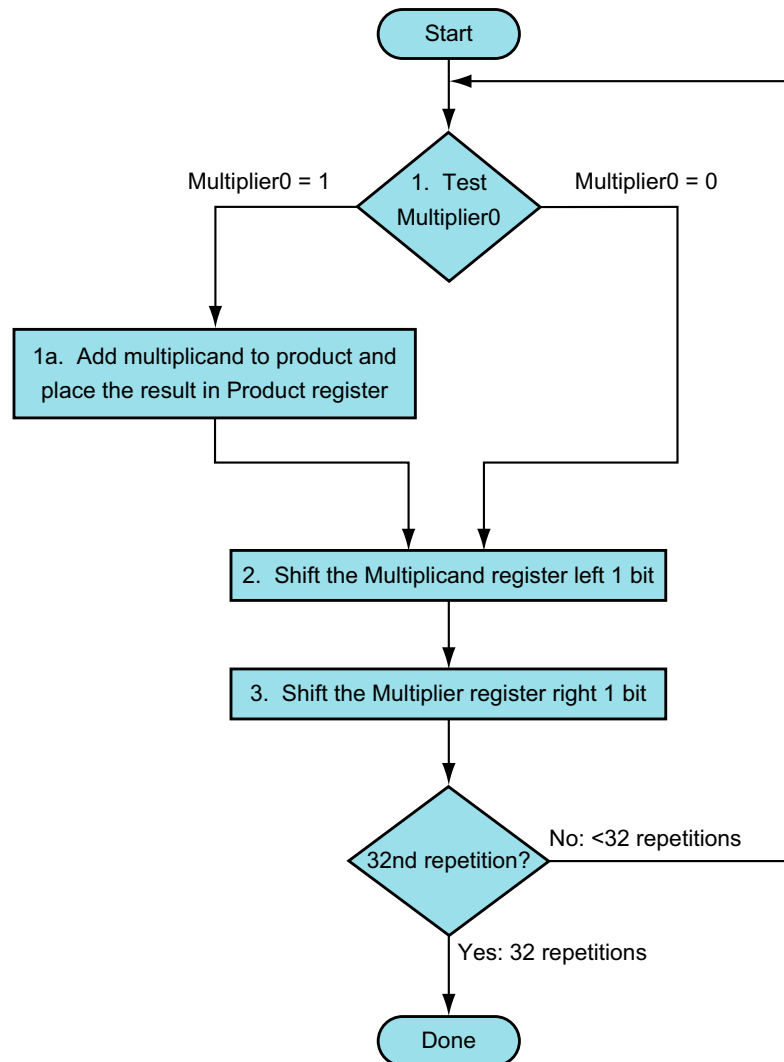


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

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.

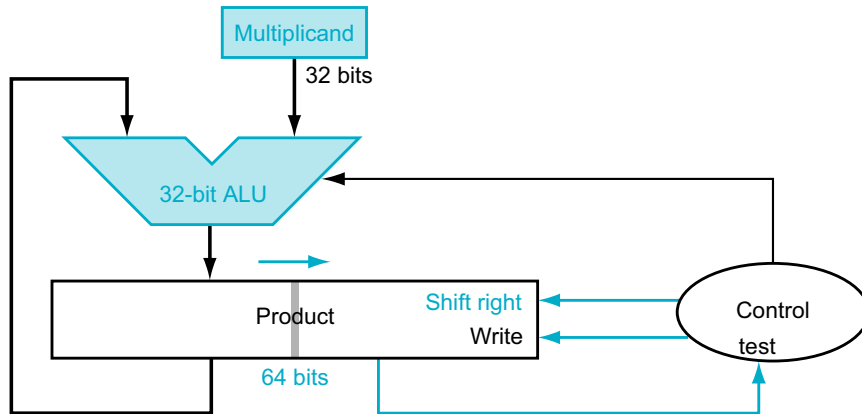


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 32 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 65 bits to hold the carry-out of the adder. These changes are highlighted in color.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001 ^①	0000 0010	0000 0000
1	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000 ^①	0000 0100	0000 0010
2	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000 ^①	0000 1000	0000 0110
3	1: $0 \Rightarrow$ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000 ^①	0001 0000	0000 0110
4	1: $0 \Rightarrow$ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

EXAMPLE

A Multiply Algorithm

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

ANSWER

Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, with the final value of $0000\ 0110_{\text{two}}$ 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 their original signs. The algorithms should next be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

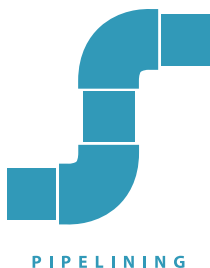
It turns out that the last algorithm will work for signed numbers, if we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 32 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes,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

Faster Multiplication

Moore's Law provided so much more in resources that hardware designers could 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 64 high. An alternative way to organize these 32 additions is in a parallel tree, as [Figure 3.7](#) shows. Instead of waiting for 32 add times, we wait just the $\log_2(32)$ or five 32-bit add times.

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



Multiply in RISC-V

To produce a properly signed or unsigned 64-bit product, RISC-V has four instructions: *multiply* (`mul`), *multiply high* (`mulh`), *multiply high unsigned* (`mulhu`), and *multiply high signed-unsigned* (`mulhsu`). To get the integer 32-bit product, the programmer uses `mul`. To get the upper 32 bits of the 64-bit product, the programmer uses (`mulh`) if both operands are signed, (`mulhu`) if both operands are unsigned, or (`mulhsu`) if one operand is signed and the other is unsigned.



Summary

Multiplication hardware simply shifts and adds, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions

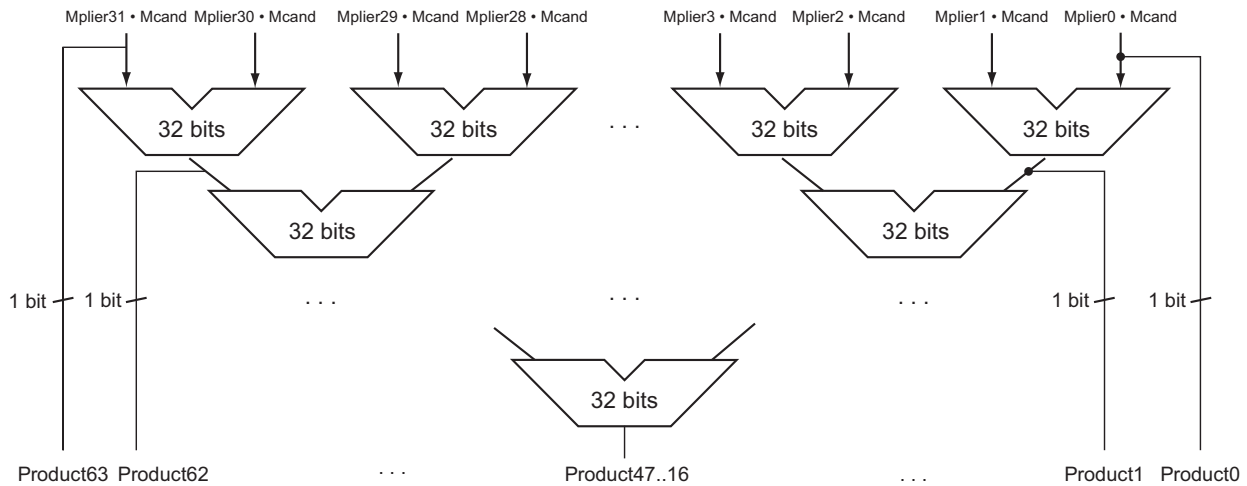


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.

for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

Software can use the multiply-high instructions to check for overflow from 32-bit multiplication. There is no overflow for 32-bit unsigned multiplication if `mulhu`'s result is zero. There is no overflow for 32-bit signed multiplication if all of the bits in `mulh`'s result are copies of the sign bit of `mul`'s result.

Hardware/ Software Interface

3.4 Division

The reciprocal operation of multiply is divide, an operation that is even less frequent and even quirrier. 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 division algorithm from grammar school. For

Divide et impera.

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

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.

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

Divisor 1000_{ten}

1001_{ten}

1001010_{ten}

-1000

10

101

1010

-1000

10_{ten}

Quotient

Dividend

Remainder

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. Occasionally, programs use the divide instruction simply to get the remainder, ignoring the quotient.

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

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

A Division Algorithm and Hardware

Figure 3.8 shows hardware to mimic our grammar school algorithm. We start with the 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 shows three steps of the first division algorithm. Unlike a human, the computer isn’t smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed comparison. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted

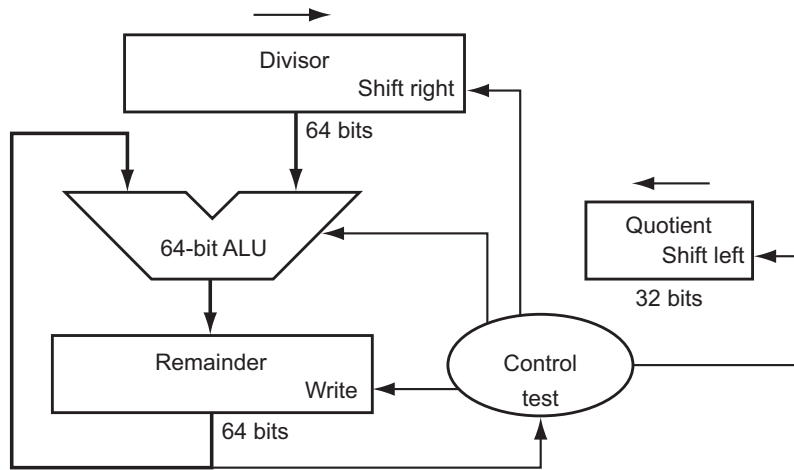


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

right, and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations complete.

EXAMPLE

A Divide Algorithm

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

ANSWER

Figure 3.10 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten} . Notice that the test in step 2 of whether the remainder is positive or negative simply checks whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

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

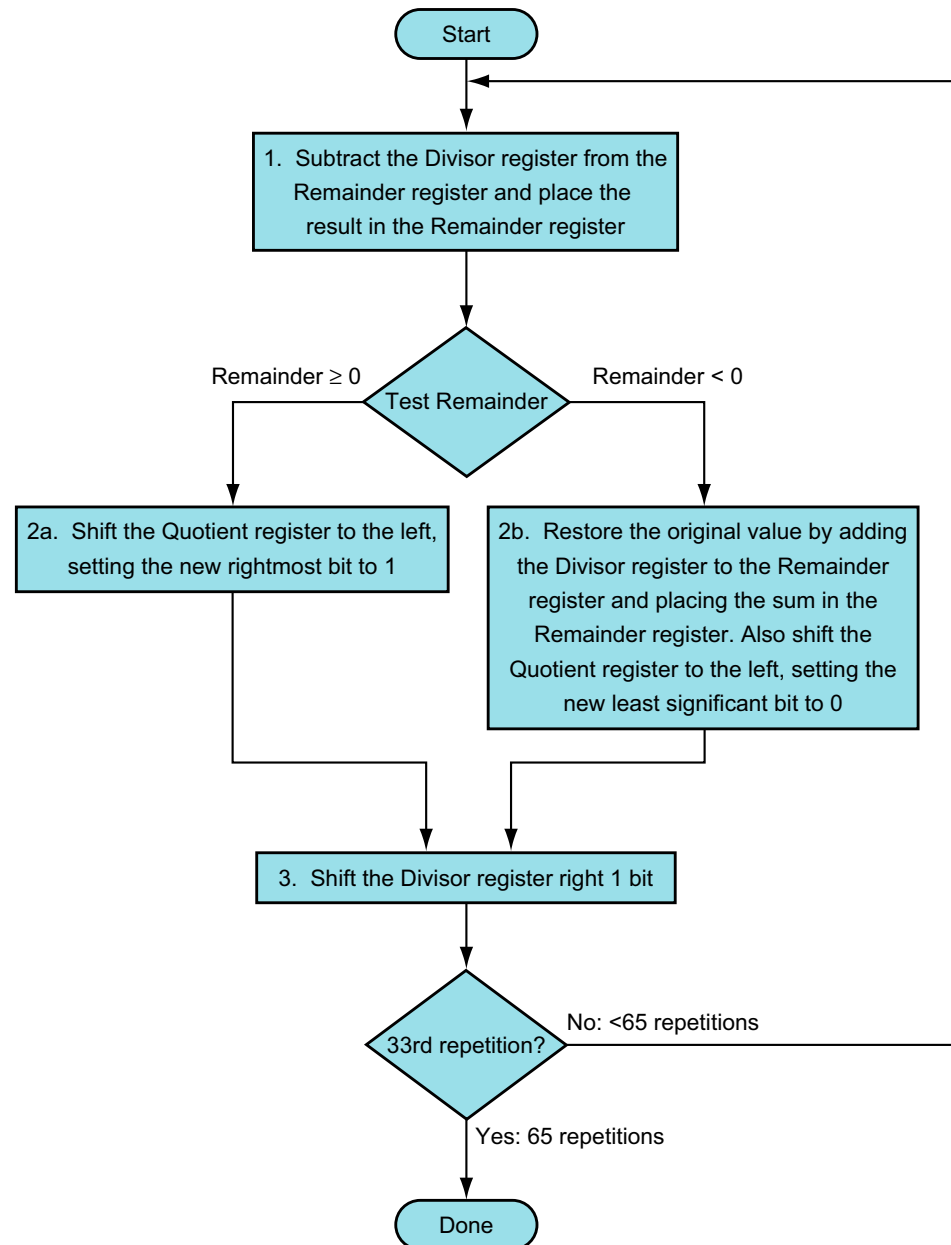


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

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 $\Rightarrow$ +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 $\Rightarrow$ +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 $\Rightarrow$ +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	①000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ 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	①000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

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

where there are unused portions of registers and adders. Figure 3.11 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.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

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

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

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

Checking the results:

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

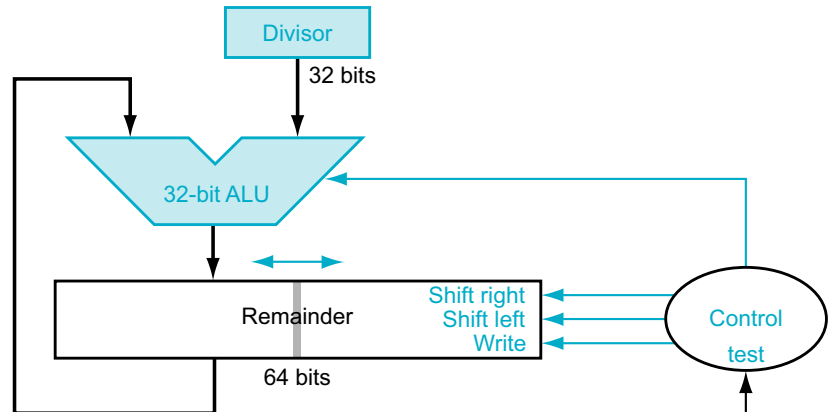


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. As in Figure 3.5, the Remainder register has grown to 65 bits to make sure the carry out of the adder is not lost.

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

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

Rewriting our basic formula to calculate the remainder:

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

So,

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

Checking the results again:

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

The reason the answer isn't a quotient of -4 and a remainder of $+1$, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

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

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

We calculate the other combinations by following the same rule:

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

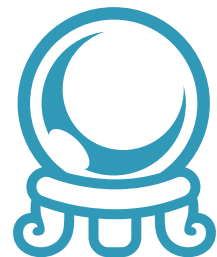
Thus, the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division

Moore's Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

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

The accuracy of this fast method depends on having proper values in the lookup table. The *Fallacy* on page 239 in [Section 3.8](#) shows what can happen if the table is incorrect.



PREDICTION

Divide in RISC-V

You may have already observed that the same sequential hardware can be used for both multiply and divide in [Figures 3.5 and 3.11](#). The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts.

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

Summary

The common hardware support for multiply and divide allows RISC-V to provide a single pair of 32-bit registers that are used both for multiply and divide. We accelerate division by predicting multiple quotient bits and then correcting mispredictions later. [Figure 3.12](#) summarizes the enhancements to the RISC-V instruction set for the last two sections.

Hardware/ Software Interface

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

Elaboration: An even faster sequential algorithm than [Figure 3.9](#) does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step, since $(r + d) \times 2 - d = r - 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes one clock cycle per step, is explored further in the exercises; the algorithm in [Figure 3.9](#) is called *restoring* division. A third algorithm that doesn't save the result of the subtract if it's negative is called a *nonperforming* division algorithm. It averages one-third fewer arithmetic operations.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	$x5 = x6 + x7$	Three register operands
	Subtract	sub x5, x6, x7	$x5 = x6 - x7$	Three register operands
	Add immediate	addi x5, x6, 20	$x5 = x6 + 20$	Used to add constants
	Set if less than	slt x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Compare two registers
	Set if less than, unsigned	sltu x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Compare two registers
	Set if less than, immediate	slti x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Comparison with immediate
	Set if less than immediate, unsigned	sltiu x5, x6, x7	$x5 = 1 \text{ if } x5 < x6, \text{ else } 0$	Comparison with immediate
	Multiply	mul x5, x6, x7	$x5 = x6 \times x7$	Lower 32 bits of 64-bit product
	Multiply high	mulh x5, x6, x7	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	$x5 = (x6 \times x7) \gg 32$	Upper 32 bits of 64-bit signed-unsigned product
	Divide	div x5, x6, x7	$x5 = x6 / x7$	Divide signed 32-bit numbers
	Divide unsigned	divu x5, x6, x7	$x5 = x6 / x7$	Divide unsigned 32-bit numbers
	Remainder	rem x5, x6, x7	$x5 = x6 \% x7$	Remainder of signed 32-bit division
	Remainder unsigned	remu x5, x6, x7	$x5 = x6 \% x7$	Remainder of unsigned 32-bit division
Data transfer	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Uns. byte halfword from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits
	Add upper immediate to PC	auipc x5, 0x12345	$x5 = \text{PC} + 0x12345000$	Used for PC-relative data addressing
Logical	And	and x5, x6, x7	$x5 = x6 \& x7$	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x7	$x5 = x6 x7$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x7	$x5 = x6 \wedge x7$	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	$x5 = x6 \& 20$	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	$x5 = x6 20$	Bit-by-bit OR reg. with constant
Shift	Exclusive or immediate	xori x5, x6, 20	$x5 = x6 \wedge 20$	Bit-by-bit XOR reg. with constant
	Shift left logical	sll x5, x6, x7	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srl_i x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if ($x5 == x6$) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if ($x5 != x6$) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if ($x5 < x6$) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if ($x5 \geq x6$) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if ($x5 < x6$) go to PC+100	PC-relative branch if registers less
	Branch if greater/equal, unsigned	bgeu x5, x6, 100	if ($x5 \geq x6$) go to PC+100	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	jal x1, 100	$x1 = \text{PC}+4$; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	$x1 = \text{PC}+4$; go to $x5+100$	Procedure return; indirect call

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

3.5 Floating Point

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

American proverb

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.

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

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

$3.14159265\dots_{\text{ten}}$ (π)

$2.71828\dots_{\text{ten}}$ (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.

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 the normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

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

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

(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 the normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will

always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since real digits to the right of the binary point replace the unnecessary leading 0s.

Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent**, because a fixed word size means you must take a bit from one to give a bit to the other. This tradeoff is between *precision* and *range*: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from [Chapter 2](#) reminds us, good design demands good compromise.

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

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 RISC-V does something slightly more sophisticated.)

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

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.

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

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

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

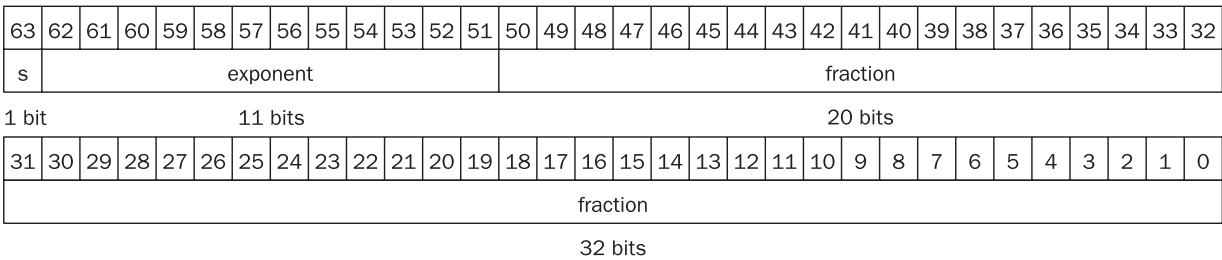
underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

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

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

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

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



RISC-V double precision allows numbers almost as small as $2.0_{\text{ten}} \times 10^{-308}$ 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 fraction.

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

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

Exceptions and Interrupts

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

IEEE 754 Floating-Point Standard

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

To pack even more bits into the number, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or

53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00 \dots 00_{\text{two}}$ 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* $s_1, s_2, s_3, \dots$, then the value is

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

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 output 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.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	$\pm$ denormalized number
1–254	Anything	1–2046	Anything	$\pm$ floating-point number
255	0	2047	0	$\pm$ 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 233. This information is also found in Column 4 of the RISC-V Reference Data Card at the front of this book.

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

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

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

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{\text{two}} \times 2^{+1}$ 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	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The desirable notation must therefore represent the most negative exponent as $00 \dots 00_{\text{two}}$ and the most positive as $11 \dots 11_{\text{two}}$. 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 $-1 + 127_{\text{ten}}$, or $126_{\text{ten}} = 0111\ 1110_{\text{two}}$, and $+1$ is represented by $1 + 127$, or $128_{\text{ten}} = 1000\ 0000_{\text{two}}$. 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

$$\pm 1.000000000000000000000000_{\text{two}} \times 2^{-126}$$

to as large as

$$\pm 1.111111111111111111111111_{\text{two}} \times 2^{+127}.$$

Let's demonstrate.

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

EXAMPLE

The number -0.75_{ten} is also

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

It is also represented by the binary fraction

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

In scientific notation, the value is

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

and in normalized scientific notation, it is

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

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

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

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

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

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.

ANSWER

Converting Binary to Decimal Floating Point**EXAMPLE**

What decimal number does this single precision float represent?

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	0	0	0

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,

ANSWER

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

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

Following IEEE guidelines, the IEEE 754 committee was reformed 20 years after the standard to see what changes, if any, should be made. The revised standard IEEE 754-2008 includes nearly all the IEEE 754-1985 and adds a 16-bit format (“half precision”) and a 128-bit format (“quadruple precision”). Half precision has a 1-bit sign, 5-bit exponent (with a bias of 15), and a 10-bit fraction. Quadruple precision has a 1-bit sign, a 15-bit exponent (with a bias of 262143), and a 112-bit fraction. The revised standard also adds decimal floating-point arithmetic.

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

Floating-Point Addition

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

$$0.016 \times 10^1$$

- Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ + \quad 0.016_{\text{ten}} \\ \hline 10.015_{\text{ten}} \end{array}$$

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:

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

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

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

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

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.

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 233). For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is -126 .

EXAMPLE

Binary Floating-Point Addition

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:

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

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:

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

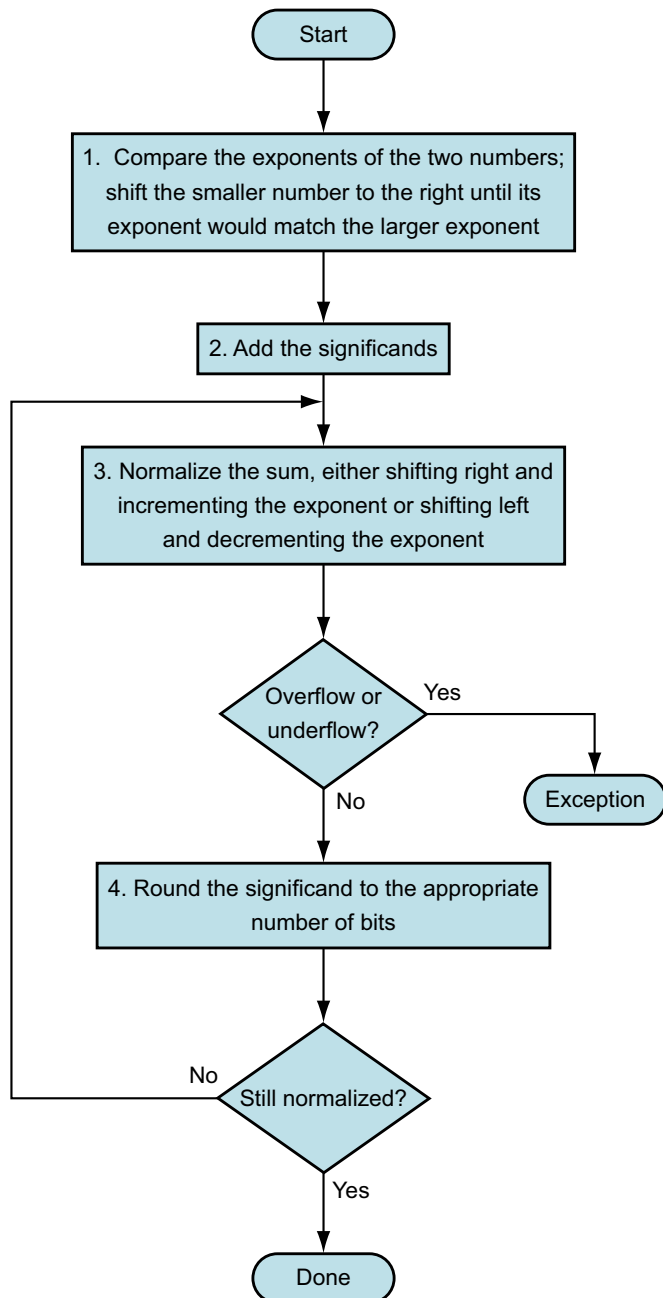


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

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

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

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: $1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$. 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:

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

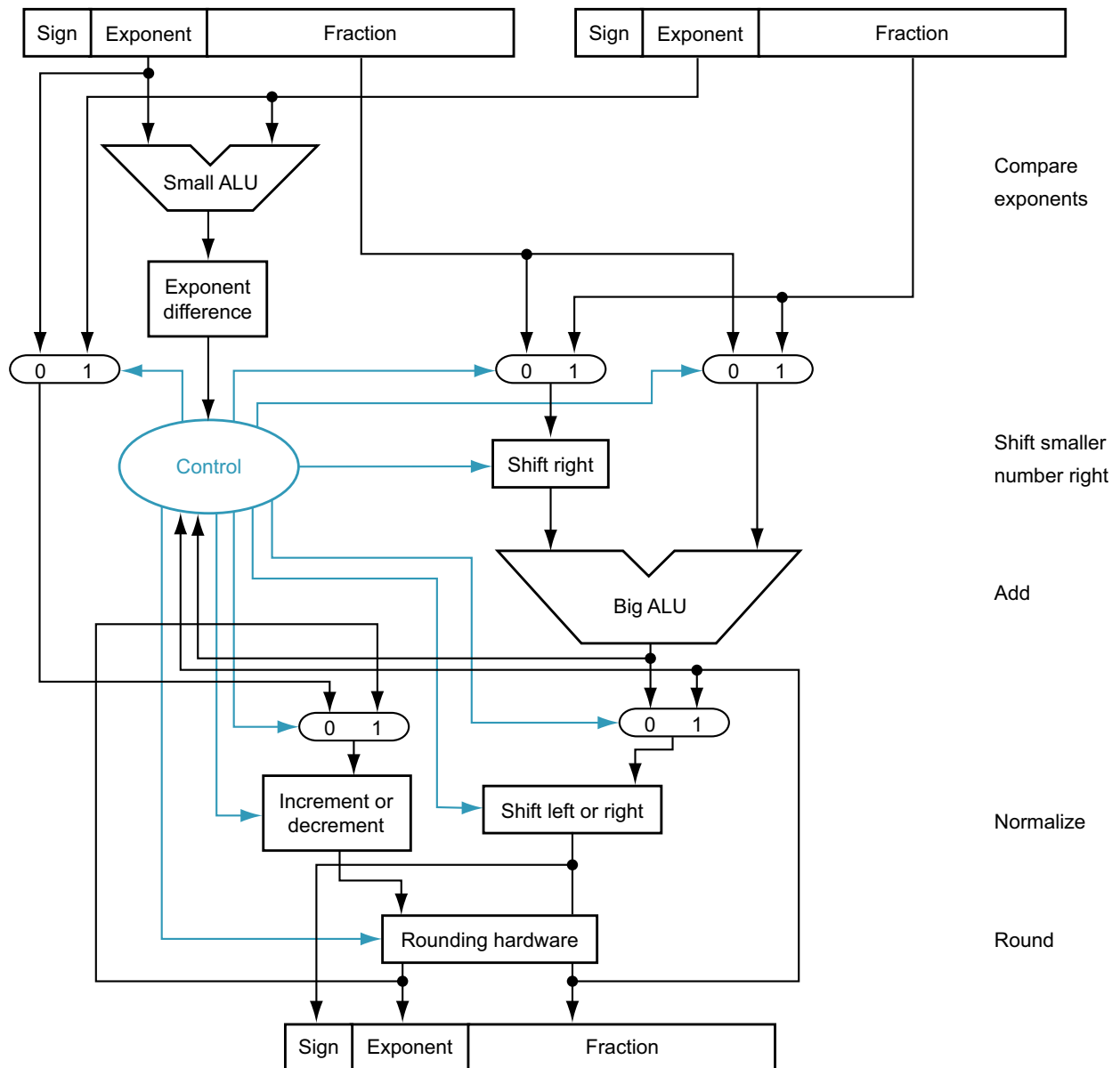


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.

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

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

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

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

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:

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

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

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

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

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 signs of the operands determine the sign of the product.

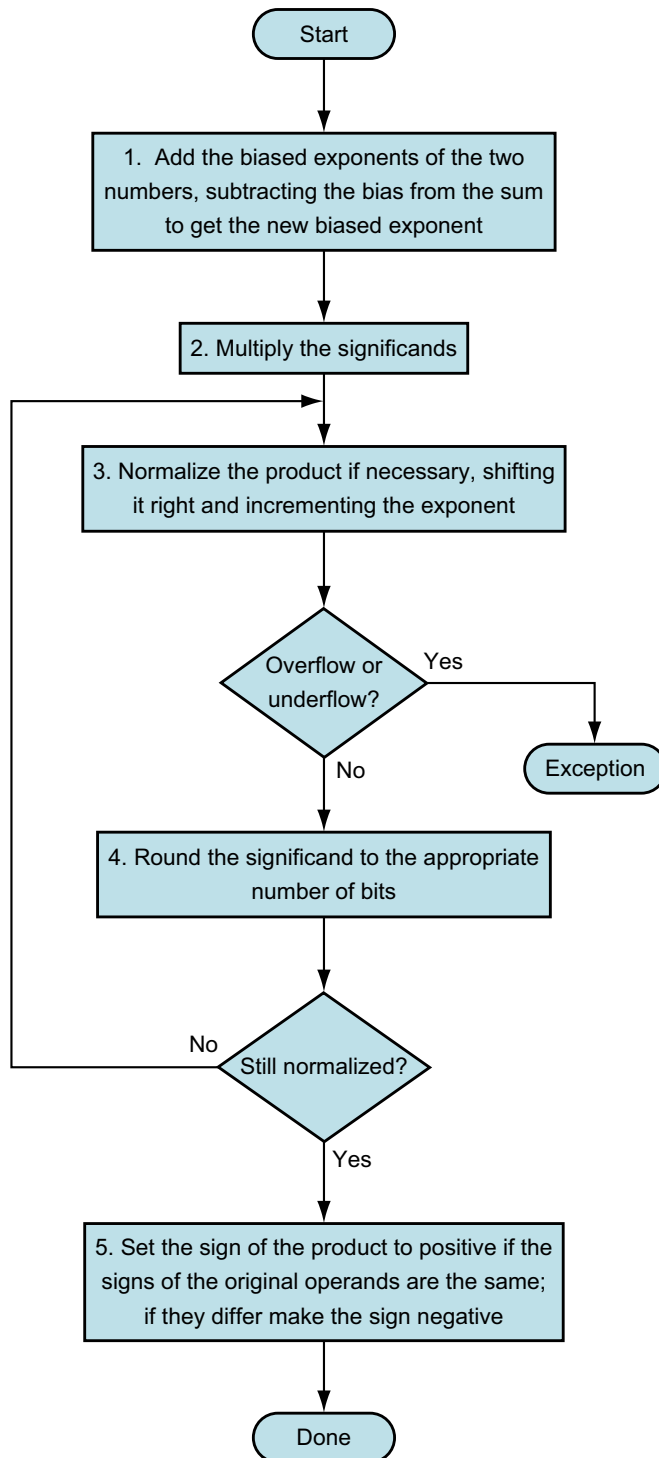


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

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

EXAMPLE

ANSWER

Binary Floating-Point Multiplication

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

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:

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 &= (-1 - 2) + (127 + 127 - 127) \\ &= -3 + 127 = 124 \end{aligned}$$

Step 2. Multiplying the significands:

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

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:

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

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

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

Converting to decimal to check our results:

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

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

Floating-Point Instructions in RISC-V

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

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

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

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

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

RISC-V floating-point operands

32 floating-point registers	f0-f31	An <i>f</i> -register can hold either a single-precision floating-point number or a double-precision floating-point number.
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

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

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

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

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

One issue that architects face in supporting floating-point arithmetic is whether to select the same registers used by the integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a distinct set of data transfer instructions to move data between floating-point registers and memory.

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

Hardware/ Software Interface

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

Let's convert a temperature in Fahrenheit to Celsius:

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

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

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

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

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

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

EXAMPLE

ANSWER

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next, we load the constant 32.0 and then subtract it from `fahr` (`f10`):

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

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

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

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

EXAMPLE

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

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of $C = C + A * B$. This code is a simplified version of the DGEMM program in [Figure 2.43](#) on page 169. Let's assume *C*, *A*, and *B* are all square matrices with 32 elements in each dimension.

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

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

ANSWER

Note that `c[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `c[i][j]`, so we can avoid loading and storing

`c[i][j]` each iteration. Instead, the compiler loads `c[i][j]` into a register outside the loop, accumulates the sum of the products of `a[i][k]` and `b[k][j]` in that same register, and then stores the sum into `c[i][j]` upon termination of the innermost loop.

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

```
mm:...
    addi x28, x0, 32    // x28 = 32 (row size/loop end)
    addi x5, x0, 0     // i = 0; initialize 1st for loop
L1:  addi x6, x0, 0     // j = 0; initialize 2nd for loop
L2:  addi x7, x0, 0     // k = 0; initialize 3rd for loop
```

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

```
slli x30, x5, 5    // x30 = i * 25(size of row of c)
```

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

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

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3 since 8 is a power of 2:

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

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

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

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

```
L3: slli x29, x7, 5    // x29 = k * 25(size of row of b)
    add x29, x29, x6    // x29 = k * size(row) + j
    slli x29, x29, 3    // x29 = byte offset of [k][j]
    add x29, x12, x29    // x29 = byte address of b[k][j]
    fld f1, 0(x29)       // f1 = 8 bytes of b[k][j]
```

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

```
slli x29, x5, 5      // x29 = i * 25(size of row of a)
add  x29, x29, x7     // x29 = i * size(row) + k
slli x29, x29, 3     // x29 = byte offset of [i][k]
add  x29, x11, x29    // x29 = byte address of a[i][k]
fld  f2, 0(x29)      // f2 = a[i][k]
```

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

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

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

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

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

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

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

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

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

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

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

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

Accurate Arithmetic

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

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

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

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

EXAMPLE**ANSWER****Rounding with Guard Digits**

Add $2.56_{\text{ten}} \times 10^0$ to $2.34_{\text{ten}} \times 10^2$, 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.

First we must shift the smaller number to the right to align the exponents, so $2.56_{\text{ten}} \times 10^0$ becomes $0.0256_{\text{ten}} \times 10^2$. 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

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

Thus the sum is $2.3656_{\text{ten}} \times 10^2$. 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 $2.37_{\text{ten}} \times 10^2$.

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

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

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

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.

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

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

IEEE 754 has four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The U.S. *Internal Revenue Service* (IRS) always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way

would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

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

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{\text{ten}} \times 10^{-1}$ to $2.34_{\text{ten}} \times 10^2$ 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 $2.345000 \dots 00$ and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than $2.345000 \dots 00$, we round instead to 2.35.

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.

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

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

Summary

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

$$(-1)^5 \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

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.

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

**The BIG
Picture**

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

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

Hardware/Software Interface

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

Check Yourself

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

- $1.0000\ 00 \times 2^0$ to $1.1111\ 1111\ 11 \times 2^{31}$, 0
- $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}$, ± 0 , $\pm \infty$, NaN
- $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}$, ± 0 , $\pm \infty$, NaN
- $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ 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. RISC-V does not provide instructions for unordered comparisons, but a careful sequence of ordered comparisons has the same effect. (Java does not support unordered compares.)

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

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

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: Subword Parallelism

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

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

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

