Summary

The common hardware support for multiply and divide allows MIPS to provide a single pair of 32-bit registers that are used both for multiply and divide. Figure 3.14 summarizes the additions to the MIPS architecture for the last two sections.

Hardware Software Interface

MIPS divide instructions ignore overflow, so software must determine if 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. MIPS software must check the divisor to discover division by 0 as well as overflow.

Elaboration: An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step since $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock per step, is explored further in Exercise 3.29; the algorithm here is called *restoring* division.

3.6 Floating Point

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

American proverb

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

$3.14159265\ldots_{ten}$ (π)

$2.71828\ldots_{ten}$ (e)

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

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

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

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

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

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	lhu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.14 MIPS architecture revealed thus far. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the hi and lo registers to support multiply and divide. Color indicates the portions revealed since Figure 3.4 on page 175. MIPS machine language is listed in the MIPS summary reference card at the front of this book.

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

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

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

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

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

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

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

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

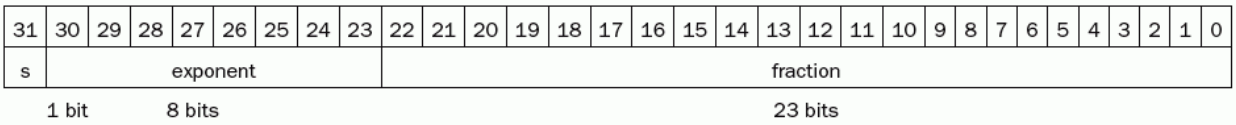
Floating-Point Representation

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

fraction The value, generally between 0 and 1, placed in the fraction field.

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

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.



In general, floating-point numbers are generally 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 MIPS does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

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

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

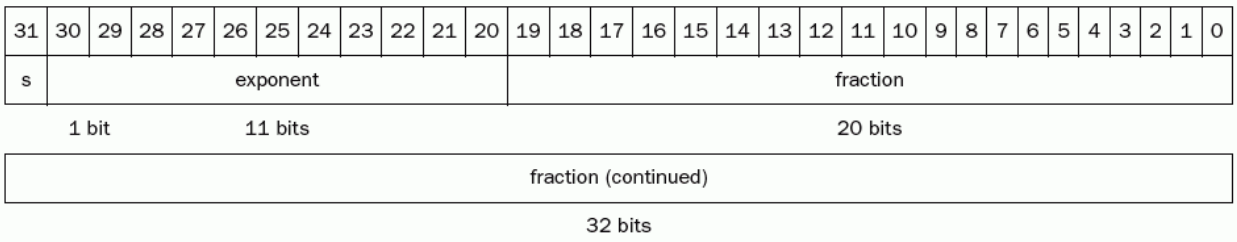
underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

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

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

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

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



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

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

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

Thus $00\ldots00_{two}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

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

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

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

Figure 3.15 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when it is 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	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

FIGURE 3.15 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the elaboration on page 217.

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

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

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

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

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

The desirable notation must therefore represent the most negative exponent as $00\ldots00_{two}$ and the most positive as $11\ldots11_{two}$. 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 –1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. 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 exponent bias for double precision is 1023.

Thus IEEE 754 notation can be processed by integer compares to accelerate sorting of floating-point numbers. Let's show the representation.

Floating-Point Representation

EXAMPLE

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

ANSWER

The number -0.75_{ten} is also

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

It is also represented by the binary fraction

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

In scientific notation, the value is

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

and in normalized scientific notation, it is

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

The general representation for a single precision number is

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

When we subtract the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$, the result is

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

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

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

The double precision representation is

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

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.

Converting Binary to Decimal Floating Point

EXAMPLE

What decimal number is represented by this single precision float?

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

ANSWER

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

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

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

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

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

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

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

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really:

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

Step 2. Next comes the addition of the significands:

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

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

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

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

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows

shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

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

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

is rounded to four digits in the significand to

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

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

Figure 3.16 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 zero bits in the exponent is reserved and used for the floating-point representation of zero. Also, the pattern of all one bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the Elaboration on page 217). Thus, for single precision, the maximum exponent is 127, and the minimum exponent is –126. The limits for double precision are 1023 and –1022.

EXAMPLE

Decimal Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.16.

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:

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

Now we follow the algorithm:

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

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

Step 2. Add the significands:

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

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

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

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:

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

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

This sum is then

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

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 3.17 sketches the basic organization of hardware for floating-point addition.

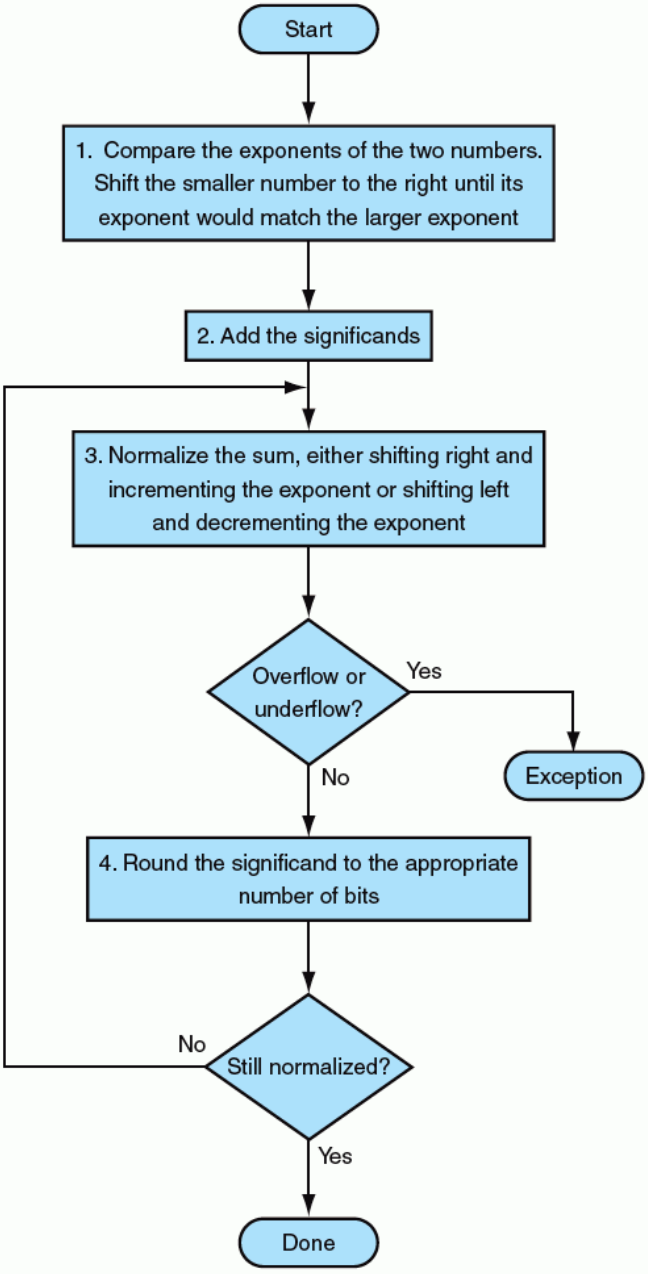


FIGURE 3.16 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

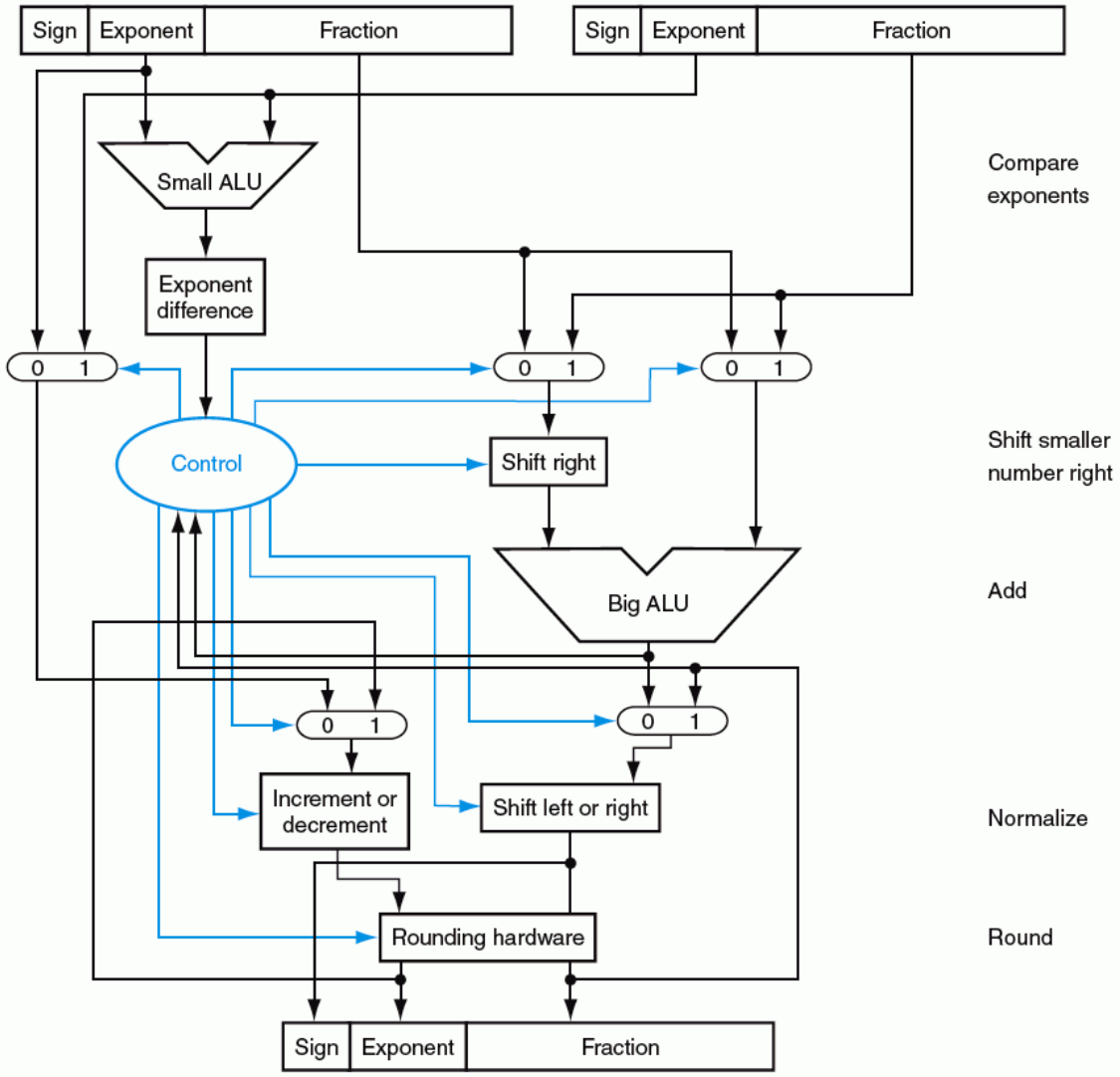


FIGURE 3.17 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.16 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Floating-Point Multiplication

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

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

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

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

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

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

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

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

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

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

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

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

$$10.212000_{ten}$$

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

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

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

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

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

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

is rounded to four digits in the significand to

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

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

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

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

Once again, as Figure 3.18 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).

Decimal Floating-Point Multiplication

EXAMPLE

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.18.

ANSWER

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

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

or, using the biased representation:

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

Step 2. Multiplying the significands:

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

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

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

Step 4. Rounding the product makes no change:

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

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

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

Converting to decimal to check our results:

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

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

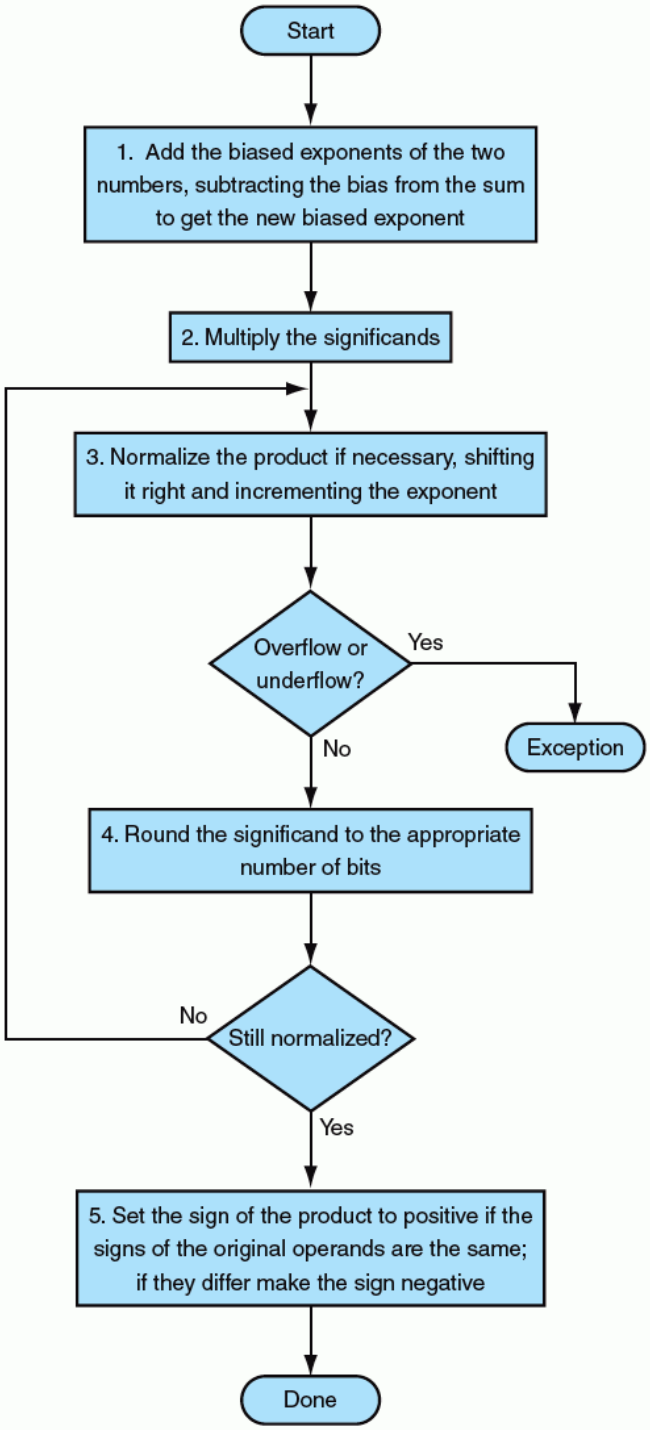


FIGURE 3.18 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

(page 205)