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

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

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

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

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

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

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

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

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

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

(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.

PARTICIPATION ACTIVITY 3.4.1: Scientific notation.

Indicate if the following numbers are in scientific notation.

1) $3.56_{ten} \times 10^{-6}$
   - ○ Yes
   - ○ No

2) $25.11_{ten} \times 10^{-8}$
   - ○ Yes
   - ○ No

3) $9.8_{ten} \times 10^{12}$
   - ○ Yes
   - ○ No

4) $1.1101_{two} \times 2^{3}$
   - ○ Yes
   - ○ No

5) $100.10_{two} \times 2^{-6}$
   - ○ Yes
   - ○ No

6) Indicate if the following number is in *normalized* scientific notation: $0.0514_{ten} \times 10^{-6}$
   - ○ Yes
   - ○ No

Floating-point representation

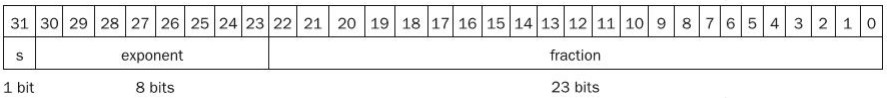
A designer of a floating-point representation must find a compromise between the size of the *fraction* and the size of the *exponent*, because a fixed word size means you must take a bit from one to add a bit to the other. This tradeoff is between precision and range: increasing the

size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from COD Chapter 2 (Instructions: Language of the Computer) reminds us, good design demands good compromise.

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

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

Floating-point numbers are usually a multiple of the size of a word. The representation of a LEGv8 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 COD Chapter 2 (Instructions: Language of the Computer), this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the number.



In general, floating-point numbers are of the form

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

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that LEGv8 does something slightly more sophisticated.)

PARTICIPATION ACTIVITY 3.4.2: Representation of single precision floating-point values.

In the tool below, "significand" is another name for the mantissa.

Enter a decimal value:

Convert

These chosen sizes of exponent and fraction give LEGv8 computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that *overflow* here means that the exponent is too large to be represented in the exponent field.

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

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

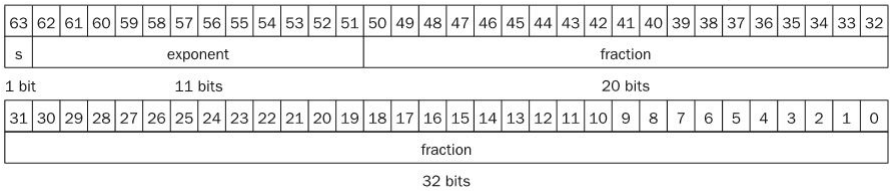
Underflow (floating-point): A situation in which a negative exponent becomes too large to fit in the exponent field.

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

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

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

The representation of a double precision floating-point number takes one LEGv8 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.



LEGv8 double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

PARTICIPATION ACTIVITY 3.4.3: Floating-point representation.

1) A calculation that leads to a number being too large to represent is called _____.

O

- overflow
- underflow
- a fraction

2) Increasing the size of the _____ used to represent a floating-point number impacts the number's precision.
   - fraction
   - exponent

3) A _____ precision floating-point number is represented with one LEGv8 doubleword.
   - single
   - double

Exceptions and interrupts

What should happen on an overflow or underflow to let the user know that a problem occurred? LEGv8 can raise an *exception*, also called an *interrupt* on many computers. 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. (COD Section 4.9 (Exceptions) covers exceptions in more detail; COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) describes other situations where exceptions and interrupts occur.)

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

Interrupt: An exception that comes from outside of the processer. (Some architectures use the term *interrupt* for all exceptions.

IEEE 754 floating-point standard

These formats go beyond LEGv8. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

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

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

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

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

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

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

Figure 3.4.1: IEEE 754 encoding of floating-point numbers (COD Figure 3.13).

A separate sign bit determines the sign. Denormalized numbers are described in the Elaboration towards the end of this section.

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)

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

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

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

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

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

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

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

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

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

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

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

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

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

to as large as

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

Let's demonstrate.

PARTICIPATION ACTIVITY 3.4.4: @Example of floating-point representation.

Start 2x speed

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

$-\frac{3}{4}$ or $-\frac{3}{2^2}$ Rewrite as a fraction

$-\frac{11_{two}}{2^2}$ or -0.11_{two} Rewrite as a binary fraction

$-0.11_{two} \times 2^0$ Rewrite as a normalized scientific notation

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

Single precision binary representation:

$-1 \times 1.1_{two} \times 2^{-1}$ $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$

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

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

Double precision binary representation:

$-1 \times 1.1_{two} \times 2^{-1}$ $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 1023)}$

$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022 - 10}$

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
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

1 bit 11 bits 20 bits

32 bits

PARTICIPATION ACTIVITY 3.4.5: Single precision floating-point representation.

Show the IEEE 754 binary representation of the number $+0.375_{ten}$ in single precision:
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$

$1.1_{two} \times 2^{-2}$ .1000 0000 0000 0000 0000 0000 or 0.011_{two} 0 125

$(-1)^0 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000) \times 2^{(125 - 127)}$ or

Rewrite as a fraction

Rewrite as a binary number

Rewrite as normalized scientific notation

S = ?

Exponent = ?

Fraction = ?

IEEE 754 binary single precision representation

Reset

PARTICIPATION ACTIVITY 3.4.6: Double precision floating-point representation.

Show the IEEE 754 binary representation of the number -0.9375$_{ten}$ in double precision:
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 1023)}$

$(-1)^1 \times (1 + .1110\ 0000\ \ldots\ 0000) \times 2^{(1022 - 1023)}$ or 0.1111_{two}

.1110 0000 … 0000 1022 or $1.111_{two} \times 2^{-1}$ 1

Rewrite as a fraction

Rewrite as a binary number

Rewrite as normalized scientific notation

S = ?

Exponent = ?

Fraction = ?

IEEE 754 binary double precision representation

Reset

Now let's try going the other direction.

PARTICIPATION ACTIVITY 3.4.7: Example of converting binary to decimal floating point.

Start ☐ 2x speed

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	0	0	0

1 bit 8 bits 23 bits

sign 1

exponent field $1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_{two} = 129_{ten}$

fraction field $0\ 1\ 0\ \ldots_{two} = 1 \times 2^{-2} = \frac{1}{4} = 0.25_{ten}$

Single precision binary representation:

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)} = (-1)^1 \times (1 + 0.25) \times 2^{(129 - 127)}$

$= -1 \times 1.25 \times 2^2$

$= -1.25 \times 4$

$= -5.0$

PARTICIPATION ACTIVITY 3.4.8: Converting a single precision binary floating-point representation to decimal.

Convert the single precision binary floating-point representation to decimal.

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	0	0	0	0	0	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

1 bit 8 bit 23 bit

1) The sign bit is ____.

Check Show answer

2) Exponent field is ____ ten.

Check Show answer

3) The fraction field is ____ ten.

Check Show answer

4) $(-1)^0 \times (? + 0.625) \times 2^{(131 - 127)}$

Check Show answer

5) What decimal number is represented by

the above single precision binary floating-point number?

$(-1)^0 \times (1 + 0.625) \times 2^{(131 - 127)}$
$= 1 \times (1 + 0.625) \times 2^4$
$= ?$

Check Show answer

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 animations.

Elaboration

Following IEEE guidelines, the IEEE 754 committee was reformed 20 years after the standard to see what changes, if any, should be made. The revised standard IEEE 754-2008 includes nearly all the IEEE 754-1985 and adds a 16-bit format ("half precision") and a 128-bit format ("quadruple precision"). No hardware has yet been built that supports quadruple precision, but it will surely come. The revised standard also adds decimal floating point arithmetic, which IBM mainframes have implemented.

Elaboration

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

Floating-point addition

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

Step 1

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

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

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

$$0.016 \times 10^1$$

Step 2

Next comes the addition of the significands:

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

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

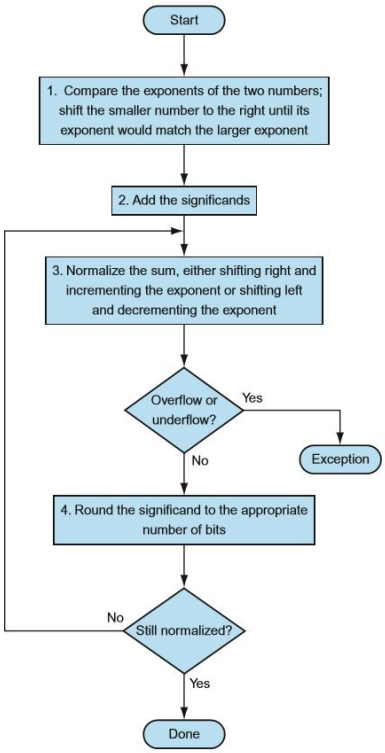
$$1.002_{ten} \times 10^2$$

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.

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

Figure 3.4.2: Floating-point addition (COD Figure 3.14).

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.



PARTICIPATION ACTIVITY 3.4.9: Example of binary floating-point addition.

Start 2x speed

Add the numbers 0.5_{ten} and -0.4375_{ten} in binary using the floating-point addition algorithm.

$$0.5_{ten} = \frac{1}{2}_{ten} = \frac{1}{2^1}_{ten} = 0.1_{two} = 0.1_{two} \times 2^0 = 1.000_{two} \times 2^{-1}$$

$$-0.4375_{ten} = \frac{-7}{16}_{ten} = \frac{-7}{2^4}_{ten} = -0.0111_{two} = -0.0111_{two} \times 2^0 = -1.110_{two} \times 2^{-2}$$

Follow the algorithm:

$1.000_{two} \times 2^{-1}$ $\quad -0.111_{two} \times 2^{-1}$ 1. Rewrite to match exponents

$1.000_{two} \times 2^{-1}$
$+ \quad -0.111_{two} \times 2^{-1}$
$0.001_{two} \times 2^{-1}$ 2. Add significands

$1.000_{two} \times 2^{-4}$ 3. Normalize sum
No overflow/underflow

4. Round sum
Already 4 bits

Check:

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

PARTICIPATION ACTIVITY 3.4.10: Binary floating-point addition.

Add the following numbers using the floating-point addition algorithm. Assume 4 bits of precision.

1) $1.010 \times 2^{-3} + 0.011 \times 2^{-3} = ?$
   - 1.101
   - 1.101×2^{-6}
   - 1.101×2^{-3}

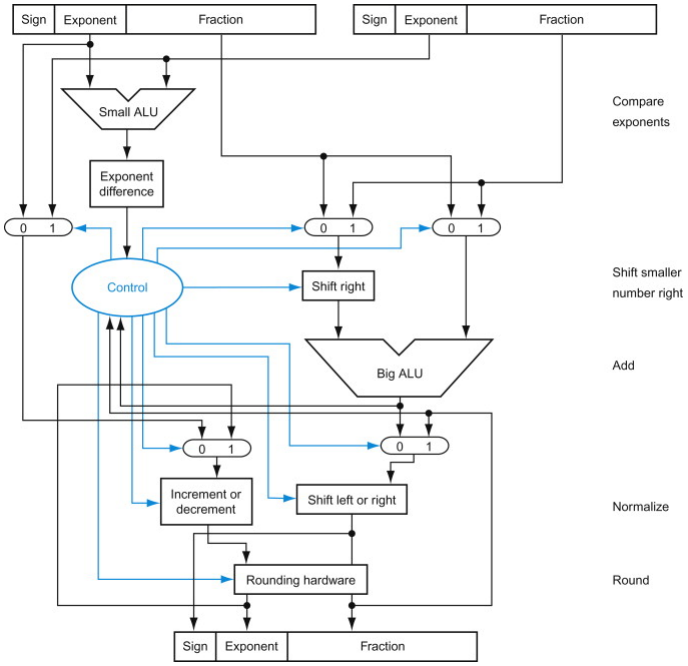
2) $1.001 \times 2^{-4} + 1.000 \times 2^{-6} = ?$
   - 10.001×2^{-4}
   - 1.011×2^{-4}

3) $1.000 \times 2^{3} + 0.011 \times 2^{5} = ?$
   - 1.010×2^{4}
   - 0.101×2^{5}
   - 10.001×2^{5}

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

Figure 3.4.3: Block diagram of an arithmetic unit dedicated to floating-point addition (COD Figure 3.15).

The steps of COD Figure 3.14 (Floating-point addition) 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.



PARTICIPATION ACTIVITY 3.4.11: Floating-point addition hardware.

1) The small ALU is used to add the significands.
   - True
   - False

2) The "Increment or decrement" and "Shift left or right" hardware normalize the sum.
   - True
   - False

3) Rounding may require the result to be normalized again.
   - True
   - False

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:

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

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

$$10.212000_{ten}$$

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

Step 3

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

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

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

Step 4

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

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

is rounded to four digits in the significand to

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

Step 5

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

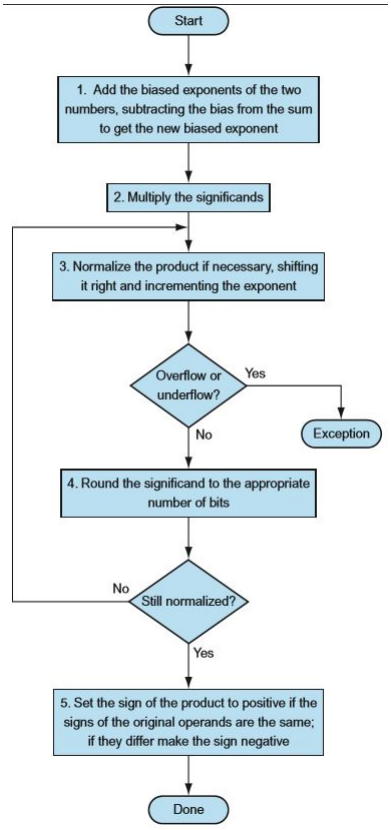
$$+1.021_{ten} \times 10^6$$

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

Once again, as the figure below 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).

Figure 3.4.4: Floating-point multiplication (COD Figure 3.16).

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.



PARTICIPATION ACTIVITY 3.4.12: Example of binary floating-point multiplication.

Start 2x speed

Multiply the numbers 0.5_{ten} and -0.4375_{ten} using the floating-point multiplication algorithm.

$0.5_{ten} = 1.000_{two} \times 2^{-1}$

$-0.4375_{ten} = -1.110_{two} \times 2^{-2}$

Follow the algorithm:

1. Add the exponents

New exponent (no bias) = $-1 + (-2)$
$= -3$

New exponent (biased) = $(-1 + 127) + (-2 + 127) - 127$
$= -3 + 127$
$= 124$

2. Multiply significands

1.000_{two}
x 1.110_{two}
0000
1000
1000
\+ 1000
1110000_{two}

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

3. Normalize sum
Normalized, no overflow/underflow

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

4. Round sum
Already 4 bits

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

5. Set sign
Signs differ, set to negative

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

Check:

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

PARTICIPATION ACTIVITY 3.4.13: Multiplication of binary floating-point numbers.

Multiply -14_{ten} and -0.25_{ten}, or $-1.110 \times 2^3 \times -1.000 \times 2^{-2}$. Assume 4 bits of precision.

3.5_{ten} 2^1 3 + (-2) = 1 $1.1100_{two} \times 2^1$ + $1.110000_{two} \times 2^1$ 1.110000

Adding the non-biased exponents of the operands

Multiply the significands:
1.110 × 1.000 = ?

Product = 1.110000 × ?

Normalize the product

Round the product

Set the sign of the product: ? $1.1100_{two} \times 2^{1}$

$-14_{ten} \times -0.25_{ten} = ?$

Reset

Floating-point instructions in ARM

LEGv8 supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point *addition, single* (`FADDS`) and *addition, double* (`FADDD`)
- Floating-point *subtraction, single* (`FSUBS`) and *subtraction, double* (`FSUBD`)
- Floating-point *multiplication, single* (`FMULS`) and *multiplication, double* (`FMULD`)
- Floating-point *division, single* (`FDIVS`) and *division, double* (`FDIVD`)
- Floating-point *comparison, single* (`FCMPS`) and *comparison, double* (`FCMPD`), with the condition codes given slightly different interpretations

Programmers use `B.cond` to branch based on floating-point comparisons.

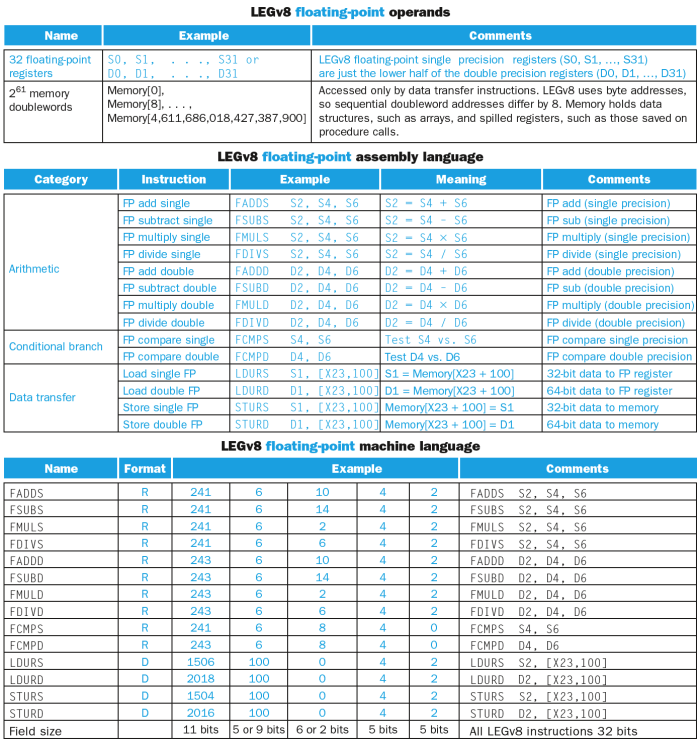
The LEGv8 designers decided to add separate floating-point registers. They are called `S0, S1, S2, ...` for single precision and `D0, D1, D2, ...` for double precision. Hence, they included separate loads and stores for floating-point registers: `LDURS` and `STURS`. The base registers for floating-point data transfers which are used for addresses remain integer registers. The LEGv8 code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
LDURS  S4, [X28,c]     // Load 32-bit F.P. number into S4
LDURS  S6, [X28,a]     // Load 32-bit F.P. number into S6
FADDS  S2, S4, S6      // S2 = S4 + S6 single precision
STURS  S2, [X28,b]     // Store 32-bit F.P. number from S2
```

A single precision register is just the lower half of a double-precision register.

The figure below summarizes the floating-point portion of the LEGv8 architecture revealed in this chapter, with the new pieces to support floating point shown in color.

Figure 3.4.5: LEGv8 floating-point architecture revealed thus far (COD Figure 3.17).



LEGv8 floating-point operands

Name	Example	Comments
32 floating-point registers	S0, S1, . . ., S31 or D0, D1, . . ., D31	LEGv8 floating-point single precision registers (S0, S1, ..., S31) are just the lower half of the double precision registers (D0, D1, ..., D31)
2^{61} memory doublewords	Memory[0], Memory[8], . . . , Memory[4,611,686,018,427,387,900]	Accessed only by data transfer instructions. LEGv8 uses byte addresses, so sequential doubleword addresses differ by 8. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

LEGv8 floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	FADDS S2, S4, S6	S2 = S4 + S6	FP add (single precision)
	FP subtract single	FSUBS S2, S4, S6	S2 = S4 - S6	FP sub (single precision)
	FP multiply single	FMULS S2, S4, S6	S2 = S4 × S6	FP multiply (single precision)
	FP divide single	FDIVS S2, S4, S6	S2 = S4 / S6	FP divide (single precision)
	FP add double	FADDD D2, D4, D6	D2 = D4 + D6	FP add (double precision)
	FP subtract double	FSUBD D2, D4, D6	D2 = D4 - D6	FP sub (double precision)
	FP multiply double	FMULD D2, D4, D6	D2 = D4 × D6	FP multiply (double precision)
	FP divide double	FDIVD D2, D4, D6	D2 = D4 / D6	FP divide (double precision)
Conditional branch	FP compare single	FCMPS S4, S6	Test S4 vs. S6	FP compare single precision
	FP compare double	FCMPD D4, D6	Test D4 vs. D6	FP compare double precision
Data transfer	Load single FP	LDURS S1, [X23,100]	S1 = Memory[X23 + 100]	32-bit data to FP register
	Load double FP	LDURD D1, [X23,100]	D1 = Memory[X23 + 100]	64-bit data to FP register
	Store single FP	STURS S1, [X23,100]	Memory[X23 + 100] = S1	32-bit data to memory
	Store double FP	STURD D1, [X23,100]	Memory[X23 + 100] = D1	64-bit data to memory

LEGv8 floating-point machine language

Name	Format	Example					Comments
FADDS	R	241	6	10	4	2	FADDS S2, S4, S6
FSUBS	R	241	6	14	4	2	FSUBS S2, S4, S6
FMULS	R	241	6	2	4	2	FMULS S2, S4, S6
FDIVS	R	241	6	6	4	2	FDIVS S2, S4, S6
FADDD	R	243	6	10	4	2	FADDD D2, D4, D6
FSUBD	R	243	6	14	4	2	FSUBD D2, D4, D6
FMULD	R	243	6	2	4	2	FMULD D2, D4, D6
FDIVD	R	243	6	6	4	2	FDIVD D2, D4, D6
FCMPS	R	241	6	8	4	0	FCMPS S4, S6
FCMPD	R	243	6	8	4	0	FCMPD D4, D6
LDURS	D	1506	100	0	4	2	LDURS S2, [X23,100]
LDURD	D	2018	100	0	4	2	LDURD D2, [X23,100]
STURS	D	1504	100	0	4	2	STURS S2, [X23,100]
STURD	D	2016	100	0	4	2	STURD D2, [X23,100]
Field size		11 bits	5 or 9 bits	6 or 2 bits	5 bits	5 bits	All LEGv8 instructions 32 bits

Elaboration

The full ARMv8 instruction set does not use the mnemonics ***FADDS*** *or* ***FADDD****. It just uses* ***FADD****, and lets the assembler pick the proper opcode depending if the registers used are* ***S*** *registers and* ***D*** *registers. We worry that it might be confusing to use the same mnemonic for both opcodes, so for teaching purposes LEGv8 distinguishes the two cases with different mnemonics. LEGv8 follows the same decision for the rest of the floating-point arithmetic and data transfer operations.*

Hardware/Software Interface

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

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

Example 3.4.1: Compiling a floating-point C program into LEGv8 assembly code.

Let's convert a temperature in Fahrenheit to Celsius:

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

Assume that the floating-point argument `fahr` is passed in `S12` and the result should go in `S0`. What is the LEGv8 assembly code?

Answer

We assume that the compiler places the three floating-point constants in memory within easy reach of register `X27`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
    LDURS S16, [X27,const5]   // S16 = 5.0 (5.0 in memory)
    LDURS S18, [X27,const9]   // S18 = 9.0 (9.0 in memory)
```

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

```
   FDIVS S16, S16, S18   // S16 = 5.0 / 9.0
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next, we load the constant 32.0 and then subtract it from `fahr (S12)`:

```
    LDURS S18, [X27,const32]  // S18 = 32.0
    FSUBS S18, S12, S18       // S18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `S0` as the return result, and then return

```
    FMULS S0, S16, S18      // S0 = (5/9)*(fahr - 32.0)
    BR LR                   // return
```

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

Example 3.4.2: Compiling floating-point C procedure with two-dimensional matrices into LEGv8.

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of C = C + A * B. It is commonly called *DGEMM*, for Double precision, General Matrix Multiply. We'll see versions of DGEMM again in COD Section 3.9 (Going faster: Subword parallelism and matrix multiply) and subsequently in COD Chapters 4 (The Processor), 5 (Large and Fast: Exploiting Memory Hierarchy), and 6 (Parallel Processor from Client to Cloud). 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 `X0, X1,` and `X2`. Assume that the integer variables are in `X19, X20,` and `X21`, respectively. What is the LEGv8 assembly code for the body of the procedure?

Answer

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

We keep the code simpler by using the assembly language pseudoinstruction `LDI`, which loads a constant into a register.

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

```
mm:...
    LDI X10, 32            // X10 = 32 (row size/loop end)
    LDI X19, 0             // i = 0; initialize 1st for loop
L1: LDI X20, 0             // j = 0; restart 2nd for loop
L2: LDI X21, 0             // k = 0; restart 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-dimension arrays, each with 32 elements. So the first step is to skip over the `i` "single-dimensional arrays," or rows, to get the one we want. Thus, we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
LSL X11, X19, 5           // X11 = i * 2^5 (size of row of c)
```

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

```
ADD  X11, X11, X20        // X11 = i * size(row) + j
```

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

```
LSL X11, X11, 3           // X11 = 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 `D4`:

```
ADD    X11, X0, X11          // X11 = byte address of c[i][j]
LDURD  D4, [X11,#0]          // D4 = 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: LSL    X9,  X21, 5      // X9 = k * 2^5 (size of row of b)
    ADD    X9,  X9, X20     // X9 = k * size(row) + j
    LSL    X9,  X9, 3       // X9 = byte offset of [k][j]
    ADD    X9,  X2, X9      // X9 = byte address of b[k][j]
    LDURD  D16, [X9,#0]     // D16 = 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]`.

```
LSL    X9,  X19, 5          // X9 = i * 2^5 (size of row of a)
ADD    X9,  X9, X21         // X9 = i * size(row) + k
LSL    X9,  X9, 3           // X9 = byte offset of [i][k]
ADD    X9,  X1, X9          // X9 = byte address of a[i][k]
LDURD  D18, [X9,#0]         // D18 = 8 bytes of 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 `D18` and `D16`, and then accumulate the sum in `D4`.

```
FMULD  D16, D18, D16    // D16 = a[i][k] * b[k][j]
FADDD  D4,  D4,  D16    // D4 = 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 `D4` into `c[i][j]`.

```
ADDI   X21, X21, 1      // $k = k + 1
CMP    X21, X10         // test k vs. 32
B.LT   L3               // if (k < 32) go to L3
STURD  D4, [X11,0]      // = D4
```

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  X20, X20, #1       // $j = j + 1
CMP   X20, X10           // test j vs. 32
B.LT  L2                 // if (j < 32) go to L2
ADDI  X19, X19, #1       // $i = i + 1
CMP   X19, X10           // test i vs. 32
B.LT  L1                 // if (i < 32) go to L1
...
```

COD Figure 3.23 (The x86 assembly language for the body of the nested loops …) shows the x86 assembly language code for a slightly different version of DGEMM in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …).

Elaboration

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

Elaboration

Another reason for separate integers and floating-point registers is that microprocessors in the

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

Elaboration

As mentioned in COD Section 3.4 (Division), 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 COD Chapter 2 (Instructions: Language of the Computer), 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, 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point. The best we can do is getting the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly, if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps two extra bits on the right during 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 3.4.3: Rounding with guard digits.

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

Answer

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

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

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

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

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

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

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is

called the number of *units in the last place*, or *ulp*. If a number were off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

Units in the last place (***ulp***): The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

PARTICIPATION ACTIVITY 3.4.14: Rounding floating-point numbers.

1) A floating-point value represented in IEEE 754 is typically an approximation.
   - ○ True
   - ○ False

2) Intermediate calculations of floating-point numbers append two extra bits to improve rounding accuracy.
   - ○ True
   - ○ False

3) *ulp* is a measure of accuracy in floating point numbers.
   - ○ True
   - ○ False

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

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

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \times 10^{-1}$ to $2.34_{ten} \times 10^{2}$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set, since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000 ... 00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000 ... 00, we round instead to 2.35.

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

MIPS-64, PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures provide a single instruction that does a multiply and add on three registers: a = a + (b × 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 COD Section 3.12 (Historical perspective and further reading)). Thus, the full ARMv8 instruction offers fused multiply-adds as well (see COD Section 3.8 (Real stuff: The rest of the ARMv8 arithmetic instructions)).

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 COD Chapter 2 (Instructions: Language of the Computer); the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

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

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

The Big Picture

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

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

Figure 3.4.6: C and Java data types, the ARM data transfer instructions, and instructions that operate on those types.

C type	Java type	Data transfers	Operations
long long int	int	LDUR, STUR, MOVZ, MOVK	ADD, SUB, ADDI, SUBI, ADDS, SUBS, ADDIS, SUBIS, MUL, SMULH, SDIV, AND, ANDI, ORR, ORRI, EOR, EORI
unsigned long long int	–	LDUR, STUR, MOVZ, MOVK	ADD, SUB, ADDI, SUBI, ADDS, SUBS, ADDIS, SUBIS, MUL, UMULH, UDIV, AND, ANDI, ORR, ORRI, EOR, EORI
char	–	LDURB, STURB, MOVZ, MOVK	ADD, SUB, ADDI, SUBI, ADDS, SUBS, ADDIS, SUBIS, MUL, SMULH, SDIV, AND, ANDI, ORR, ORRI, EOR, EORI
–	char	LDURH, STURH, MOVZ, MOVK	ADD, SUB, ADDI, SUBI, ADDS, SUBS, ADDIS, SUBIS, MUL, UMULH, UDIV, AND, ANDI, ORR, ORRI, EOR, EORI
float	float	LDURS, STURS	FADDS, FSUBS, FMULS, FDIVS, FCMPS
double	double	LDURD, STURD	FADDD, FSUBD, FMULD, FDIVD, FCMPD

Hardware/Software Interface

In the last chapter, we presented the storage classes of the programming language C (see the Hardware/Software Interface section in COD Section 2.7 (Instructions for making decisions)). 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 COD Chapter 2 (Instructions: Language of the Computer) and this chapter. Note that Java omits unsigned integers.

PARTICIPATION ACTIVITY 3.4.15: Check yourself: Half precision floating-point format.

1) 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 the half precision format 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, ±∞, NaN
   - ○ $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}$, ±0, ±∞, NaN
   - ○ $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \times 2^{14}$, ±0, ±∞, NaN

Elaboration

To accommodate comparisons that may include NaNs, the standard includes ordered and unordered as options for compares. Hence, the full ARMv8 instruction set has many flavors of compares to support NaNs. (Java does not support unordered compares.)

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

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

but the smallest single precision denormalized number is

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

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

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