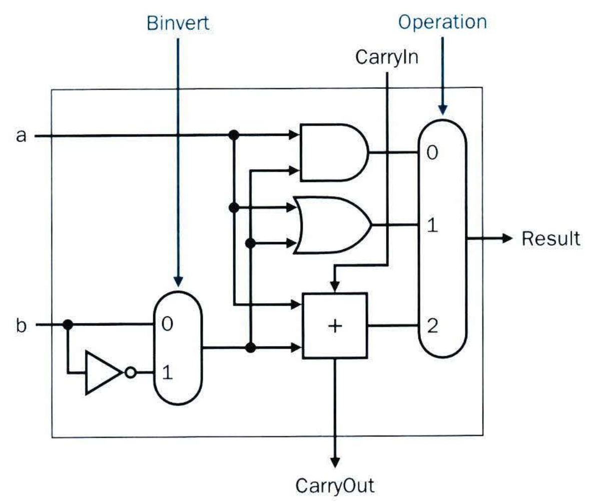


FIGURE 4.16 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\bar{b}$. By selecting b (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

The adder will then calculate a + b + 1. By selecting the inverted version of b, we get exactly what we want:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.

Tailoring the 32-Bit ALU to MIPS

This set of operations—add, subtract, AND, OR—is found in the ALU of almost every computer. If we look at Figure 4.7 on page 228, we see that the operations of most MIPS instructions can be performed by this ALU. But the design of the ALU is incomplete.

One instruction that still needs support is the set on less than instruction (`slt`). Recall that the operation produces 1 if rs < rt, and 0 otherwise. Consequently, `slt` will set all but the least significant bit to 0, with the least significant bit set according to the comparison. For the ALU to perform `slt`, we first need to expand the three-input multiplexor in Figure 4.16 to add an input for the `slt` result. We call that new input *Less,* and use it only for `slt`.

The top drawing of Figure 4.17 shows the new 1-bit ALU with the expanded multiplexor. From the description of `slt` above, we must connect 0 to the Less input for the upper 31 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set *the least significant bit* for set on less than instructions.

What happens if we subtract b from a? If the difference is negative, then a < b since

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$
$$\Rightarrow a < b$$

We want the least significant bit of a set on less than operation to be a 1 if a < b; that is, a 1 if a − b is negative and a 0 if it's positive. This desired result corresponds exactly to the sign-bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.

Unfortunately, the Result output from the most significant ALU bit in the top of Figure 4.17 for the `slt` operation is *not* the output of the adder; the ALU output for the `slt` operation is obviously the input value Less.

Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure 4.17 shows the design, with this new adder output line called *Set,* and used only for `slt`. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.

Alas, the test of less than is a little more complicated than just described because of overflow; Exercise 4.23 on page 326 explores what must be done. Figure 4.18 shows the 32-bit ALU.

Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1. For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called *Bnegate.*

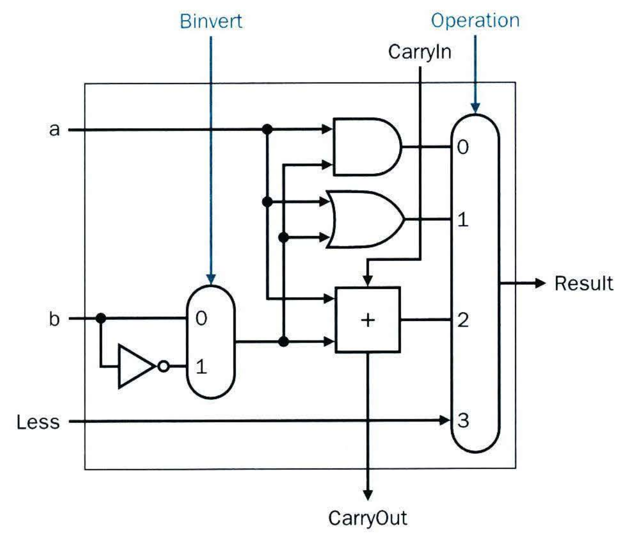
To further tailor the ALU to the MIPS instruction set, we must support conditional branch instructions. These instructions branch either if two registers are equal or if they are unequal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0 since

$$(a - b = 0) \Rightarrow a = b$$

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

Figure 4.19 shows the revised 32-bit ALU. We can think of the combination of the 1-bit Bnegate line and the 2-bit Operation lines as 3-bit control lines for



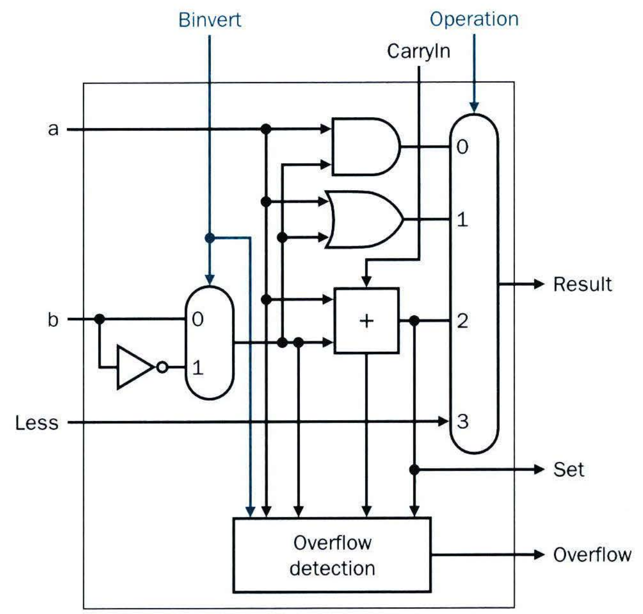


FIGURE 4.17 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{b}$, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure 4.18); the bottom has a direct output from the adder for the less than comparison called Set. (Refer to Exercise 4.42 to see how to calculate overflow with fewer inputs.)

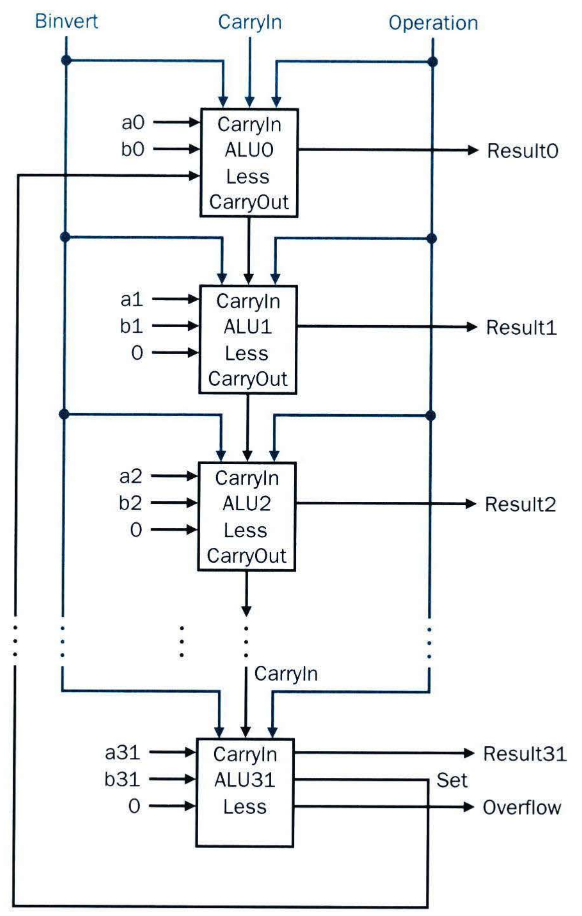


FIGURE 4.18 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure 4.17 and one 1-bit ALU in the bottom of that figure. The Less inputs are connected to 0 except for the least significant bit, and that is connected to the Set output of the most significant bit. If the ALU performs a – b and we select the input 3 in the multiplexor in Figure 4.17, then Result = 0 . . . 001 if a < b, and Result = 0 . . . 000 otherwise.

the ALU, telling it to perform add, subtract, AND, OR, or set on less than. Figure 4.20 shows the ALU control lines and the corresponding ALU operation.

Finally, now that we have seen what is inside a 32-bit ALU, we will use the universal symbol for a complete ALU, as shown in Figure 4.21.

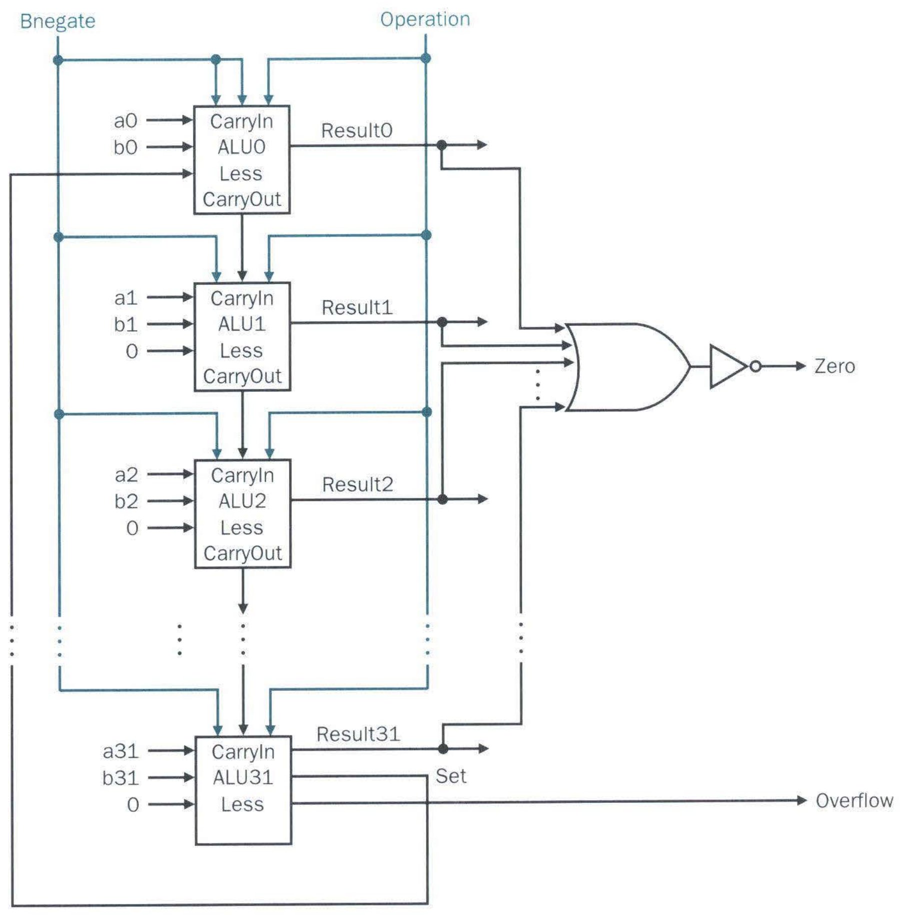


FIGURE 4.19 The final 32-bit ALU. This adds a Zero detector to Figure 4.18.

ALU control lines	Function
000	and
001	or
010	add
110	subtract
111	set on less than

FIGURE 4.20 The values of the three ALU control lines Bnegate and Operation and the corresponding ALU operations.

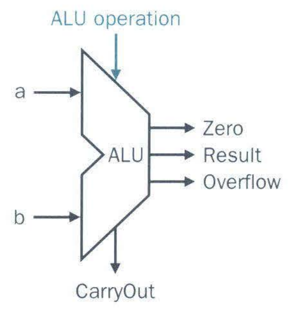


FIGURE 4.21 The symbol commonly used to represent an ALU, as shown in Figure 4.19. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

Carry Lookahead

The next question is, How quickly can this ALU add two 32-bit operands? We can determine the a and b inputs, but the CarryIn input depends on the operation in the adjacent 1-bit adder. If we trace all the way through the chain of dependencies, we connect the most significant bit to the least significant bit, so the most significant bit of the sum must wait for the *sequential* evaluation of all 32 1-bit adders. This sequential chain reaction is too slow to be used in time-critical hardware.

There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry.

A key to understanding fast carry schemes is to remember that, unlike software, hardware executes in parallel whenever inputs change.

Fast Carry Using "Infinite" Hardware

Appendix B mentions that any equation can be represented in two levels of logic. Since the only external inputs are the two operands and the CarryIn to the least significant bit of the adder, in theory we could calculate the CarryIn values to all the remaining bits of the adder in just two levels of logic.

For example, the CarryIn for bit 2 of the adder is exactly the CarryOut of bit 1, so the formula is

$$\text{CarryIn2} = (b1 \cdot \text{CarryIn1}) + (a1 \cdot \text{CarryIn1}) + (a1 \cdot b1)$$

Similarly, CarryIn1 is defined as

$$\text{CarryIn1} = (b0 \cdot \text{CarryIn0}) + (a0 \cdot \text{CarryIn0}) + (a0 \cdot b0)$$

Using the shorter and more traditional abbreviation of *ci* for CarryIn*i*, we can rewrite the formulas as

$$c2 = (b1 \cdot c1) + (a1 \cdot c1) + (a1 \cdot b1)$$
$$c1 = (b0 \cdot c0) + (a0 \cdot c0) + (a0 \cdot b0)$$

Substituting the definition of c1 for the first equation results in this formula:

$$c2 = (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot c0) + (a1 \cdot b0 \cdot c0) + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot c0) + (b1 \cdot b0 \cdot c0) + (a1 \cdot b1)$$

You can imagine how the equation expands as we get to higher bits in the adder; it grows exponentially with the number of bits. This complexity is reflected in the cost of the hardware for fast carry, making this simple scheme prohibitively expensive for wide adders.

Fast Carry Using the First Level of Abstraction: Propagate and Generate

Most fast carry schemes limit the complexity of the equations to simplify the hardware, while still making substantial speed improvements over ripple carry. One such scheme is a *carry-lookahead adder*. In Chapter 1, we said computer systems cope with complexity by using levels of abstraction. A carry-lookahead adder relies on levels of abstraction in its implementation.

Let's factor our original equation as a first step:

$$ci{+}1 = (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi)$$
$$= (ai \cdot bi) + (ai + bi) \cdot ci$$

If we were to rewrite the equation for c2 using this formula, we would see some repeated patterns:

$$c2 = (a1 \cdot b1) + (a1 + b1) \cdot ((a0 \cdot b0) + (a0 + b0) \cdot c0)$$

Note the repeated appearance of $(ai \cdot bi)$ and $(ai + bi)$ in the formula above. These two important factors are traditionally called *generate* (*gi*) and *propagate* (*pi*):

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

Using them to define *ci+1*, we get

$$ci{+}1 = gi + pi \cdot ci$$

To see where the signals get their names, suppose *gi* is 1. Then

$$ci{+}1 = gi + pi \cdot ci = 1 + pi \cdot ci = 1$$

That is, the adder *generates* a CarryOut (*ci+1*) independent of the value of CarryIn (*ci*). Now suppose that *gi* is 0 and *pi* is 1. Then

$$ci{+}1 = gi + pi \cdot ci = 0 + 1 \cdot ci = ci$$

That is, the adder *propagates* CarryIn to a CarryOut. Putting the two together, CarryIn*i+1* is a 1 if either *gi* is 1 or both *pi* is 1 and CarryIn*i* is 1.

As an analogy, imagine a row of dominoes set on edge. The end domino can be tipped over by pushing one far away provided there are no gaps between the two. Similarly, a carry out can be made true by a generate far away provided all the propagates between them are true.

Relying on the definitions of propagate and generate as our first level of abstraction, we can express the CarryIn signals more economically. Let's show it for 4 bits:

$$c1 = g0 + (p0 \cdot c0)$$
$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$
$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$
$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

These equations just represent common sense: CarryIn*i* is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. Figure 4.22 uses plumbing to try to explain carry lookahead.

Even this simplified form leads to large equations and, hence, considerable logic even for a 16-bit adder. Let's try moving to two levels of abstraction.

Fast Carry Using the Second Level of Abstraction

First we consider this 4-bit adder with its carry-lookahead logic as a single building block. If we connect them in ripple carry fashion to form a 16-bit adder, the add will be faster than the original with a little more hardware.

To go faster, we'll need carry lookahead at a higher level. To perform carry lookahead for 4-bit adders, we need propagate and generate signals at this higher level. Here they are for the four 4-bit adder blocks:

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0$$
$$P1 = p7 \cdot p6 \cdot p5 \cdot p4$$
$$P2 = p11 \cdot p10 \cdot p9 \cdot p8$$
$$P3 = p15 \cdot p14 \cdot p13 \cdot p12$$

That is, the "super" propagate signal for the 4-bit abstraction (P*i*) is true only if each of the bits in the group will propagate a carry.

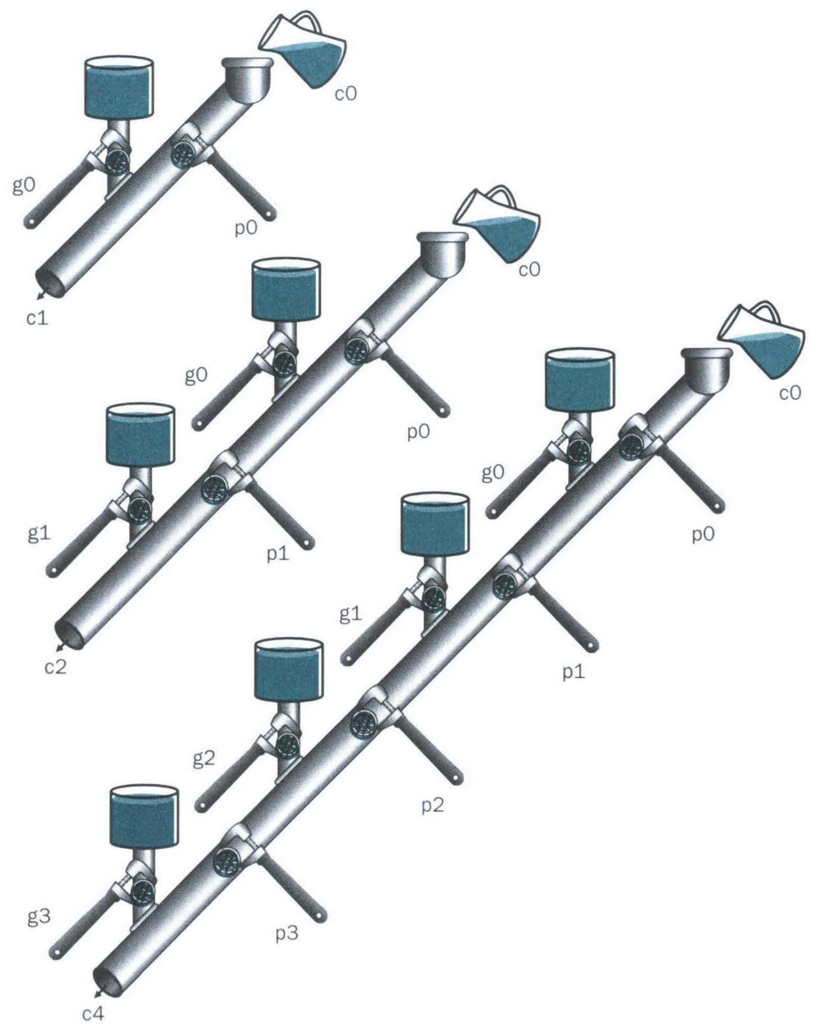


FIGURE 4.22 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water, pipes, and valves. The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe (*ci+1*) will be full if either the nearest generate value (*gi*) is turned on or if the *i* propagate value (*pi*) is on and there is water further upstream, either from an earlier generate, or propagate with water behind it. CarryIn (c0) can result in a carry out without the help of any generates, but with the help of *all* propagates.

For the "super" generate signal (G*i*), we care only if there is a carry out of the most significant bit of the 4-bit group. This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

$$G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$
$$G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$
$$G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$
$$G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

Figure 4.23 updates our plumbing analogy to show P0 and G0.

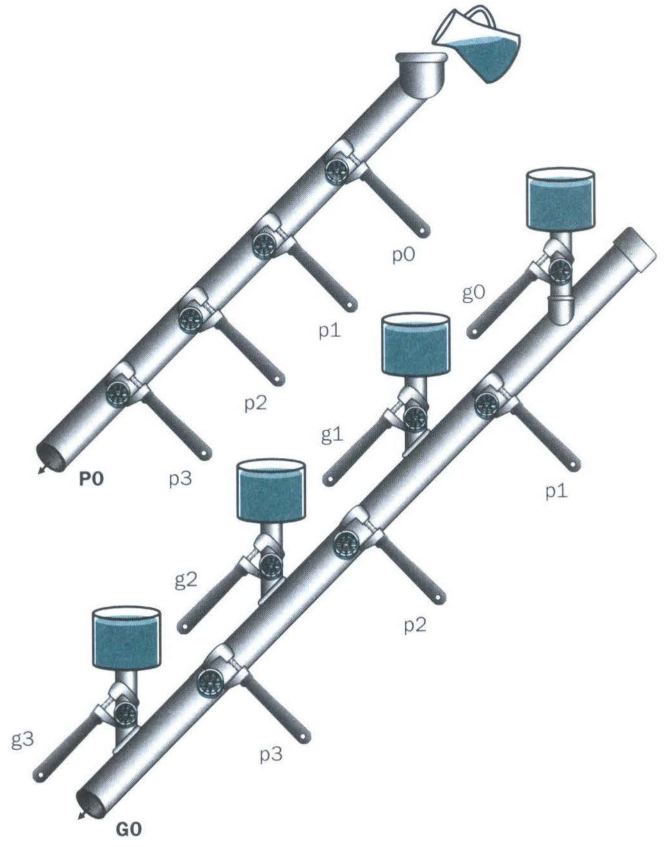


FIGURE 4.23 A plumbing analogy for the next-level carry-lookahead signals P0 and G0. P0 is open only if all four propagates (*pi*) are open, while water flows in G0 only if at least one generate (*gi*) is open and all the propagates downstream from that generate are open.

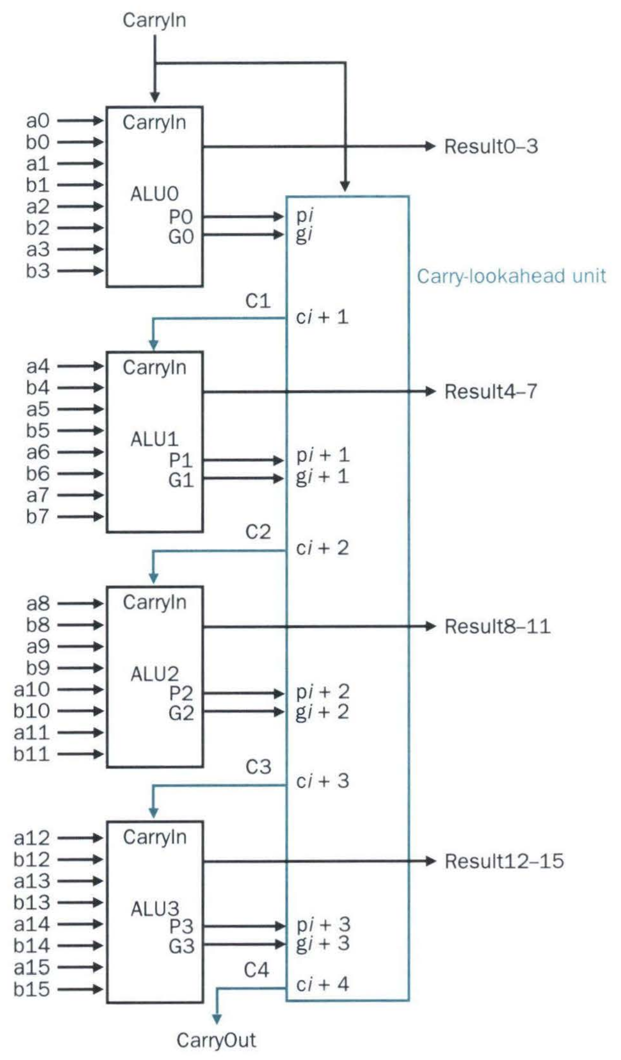


FIGURE 4.24 Four 4-bit ALUs using carry lookahead to form a 16-bit adder. Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder (C1, C2, C3, C4 in Figure 4.24) are very similar to the carry out equations for each bit of the 4-bit adder (c1, c2, c3, c4) on page 243:

$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

Figure 4.24 shows 4-bit adders connected with such a carry lookahead unit. Exercises 4.44 through 4.48 explore the speed differences between these carry schemes, different notations for multibit propagate and generate signals, and the design of a 64-bit adder.

Both Levels of the Propagate and Generate

Example

Determine the *gi*, *pi*, *Pi*, and *Gi* values of these two 16-bit numbers:

```
a:      0001 1010 0011 0011two
b:      1110 0101 1110 1011two
```

Also, what is CarryOut15 (C4)?

Answer

Aligning the bits makes it easy to see the values of generate gi $(ai \cdot bi)$ and propagate pi $(ai + bi)$:

```
a:      0001 1010 0011 0011
b:      1110 0101 1110 1011
gi:     0000 0000 0010 0011
pi:     1111 1111 1111 1011
```

where the bits are numbered 15 to 0 from left to right. Next, the "super" propagates (P3, P2, P1, P0) are simply the AND of the lower-level propagates:

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

The "super" generates are more complex, so use the following equations:

$$\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)\\
&= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0\\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1\\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0\\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0
\end{aligned}$$

Finally, CarryOut15 is

$$\begin{aligned}
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0)\\
&\qquad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)\\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0)\\
&= 0 + 0 + 1 + 0 + 0 = 1
\end{aligned}$$

Hence there *is* a carry out when adding these two 16-bit numbers.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking a shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.

To appreciate the importance of carry lookahead, we need to calculate the relative performance between it and ripple carry adders.

Speed of Ripple Carry versus Carry Lookahead

Example

One simple way to model time for logic is to assume each AND or OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the longest path through a piece of logic. Compare the number of *gate delays* for the critical paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

Answer

Figure 4.13 on page 233 shows that the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is $16 \times 2 = 32$.

For carry lookahead, the carry out of the most significant bit is just C4, defined in the example. It takes two levels of logic to specify C4 in terms of P*i* and G*i* (the OR of several AND terms). P*i* is specified in one level of logic (AND) using p*i*, and G*i* is specified in two levels using p*i* and g*i*, so the worst case for this next level of abstraction is two levels of logic. p*i* and g*i* are each one level of logic, defined in terms of a*i* and b*i*. If we assume one gate delay for each level of logic in these equations, the worst case is $2 + 2 + 1 = 5$ gate delays.

Hence for 16-bit addition a carry-lookahead adder is six times faster, using this simple estimate of hardware speed.

Summary

The primary point of this section is that the traditional ALU can be constructed from a multiplexor and a few gates that are replicated 32 times. To make it more useful to the MIPS architecture, we expand the traditional ALU with hardware to test if the result is 0, detect overflow, and perform the basic operation for set on less than.

Carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, generate and propagate. The former creates a carry regardless of the carry input, and the other passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity.

Elaboration: We have now accounted for all but one of the arithmetic and logical operations for the core MIPS instruction set: the ALU in Figure 4.21 omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or right shift by 1 bit. But hardware designers have created a circuit called a *barrel shifter*, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers, so shifting is normally done outside the ALU.

Elaboration: The logic equation for the Sum output of the full adder on page 234 can be expressed more simply by using a more powerful gate than AND and OR. An *exclusive OR* gate is true if the two operands disagree; that is,

$$x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0$$

In some technologies, exclusive OR is more efficient than two levels of AND and OR gates. Using the symbol $\oplus$ to represent exclusive OR, here is the new equation:

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

Also, we have drawn the ALU the traditional way, using gates. Computers are designed today in CMOS transistors, which are basically switches. CMOS ALU and barrel shifters take advantage of these switches and have many fewer multiplexors than shown in our designs, but the design principles are similar.

4.6 Multiplication

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

Anonymous, Elizabethan manuscript, 1570

With the construction of the ALU and explanation of addition, subtraction, and shifts, we are ready to build the more vexing operation of multiply.

But first let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. Multiplying 1000_{ten} by 1001_{ten}:

```
Multiplicand          1000ten
Multiplier         x  1001ten
                      ----
                      1000
                     0000
                    0000
                   1000
                   -------
Product            1001000ten
```

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

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

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

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above happened to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through three generations. The rest of this section presents successive refinements of the hardware and the algorithm until we have a version used in some computers. For now, let's assume that we are multiplying only positive numbers.

First Version of the Multiplication Algorithm and Hardware

The initial design mimics the algorithm we learned in grammar school; the hardware is shown in Figure 4.25. We have drawn the hardware so that data flows from top to bottom to more closely resemble the paper-and-pencil method.

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

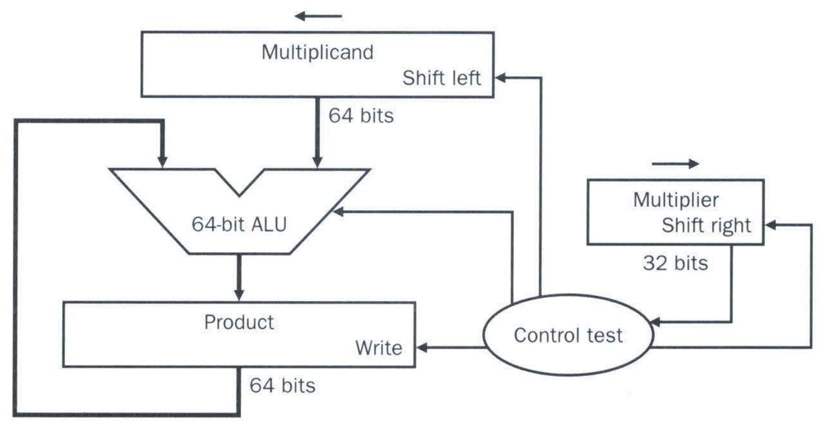


FIGURE 4.25 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register, and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

32-bit multiplicand would move 32 bits to the left. Hence we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and 0 in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

Figure 4.26 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is

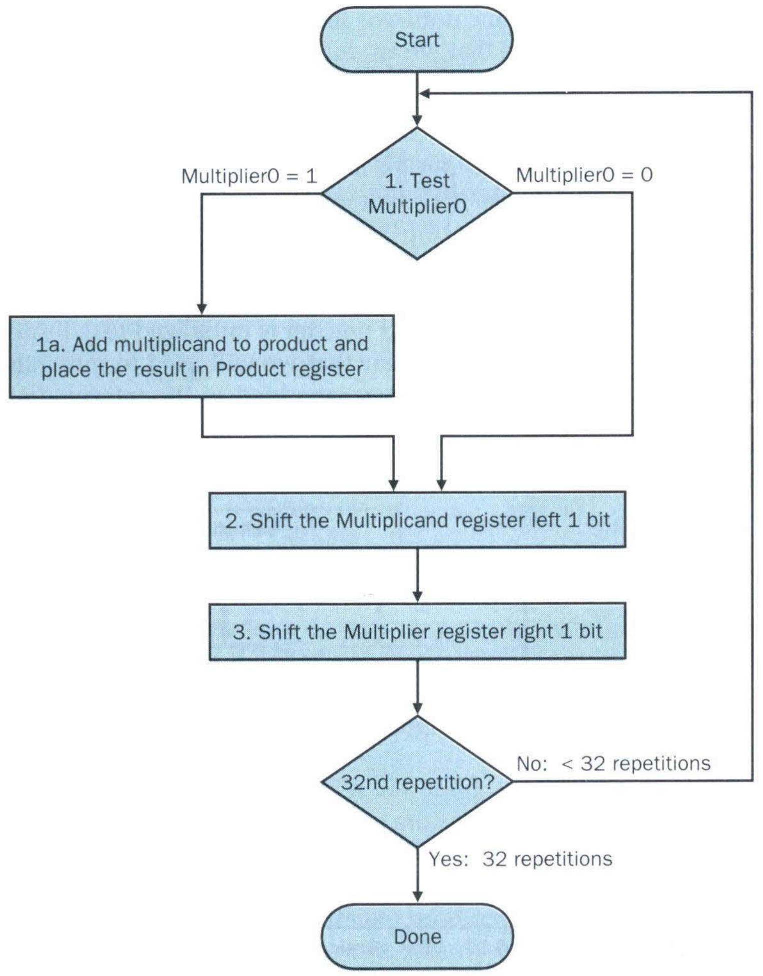

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

FIGURE 4.26 The first multiplication algorithm, using the hardware shown in Figure 4.25. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product.

First Multiply Algorithm

Example

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

Answer

Figure 4.27 shows the value of each register for each of the steps labeled according to Figure 4.26, with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 4.27 Multiply example using first algorithm in Figure 4.26. The bit examined to determine the next step is circled in color.

If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Chapter 2, page 75) reminds us that even a moderate frequency for a slow operation can limit performance.

INTEL - 1012

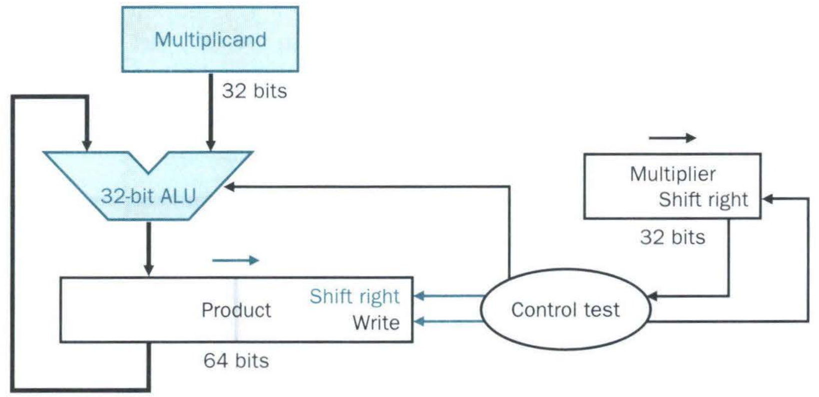


FIGURE 4.28 Second version of the multiplication hardware. Compare with the first version in Figure 4.25. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. These changes are highlighted in color.

Second Version of the Multiplication Algorithm and Hardware

Computer pioneers recognized that half of the bits of the multiplicand in the first algorithm were always 0, so only half could contain useful bit values. A full 64-bit ALU thus seemed wasteful and slow since half of the adder bits were adding 0 to the intermediate sum.

The original algorithm shifts the multiplicand left with 0s inserted in the new positions, so the multiplicand cannot affect the least significant bits of the product after they settle down. Instead of shifting the multiplicand left, they wondered, what if we shift the *product right*? Now the multiplicand would be fixed relative to the product, and since we are adding only 32 bits, the adder need be only 32 bits wide. Figure 4.28 shows how this change halves the widths of both the ALU and the multiplicand.

Figure 4.29 shows the multiply algorithm inspired by this observation. This algorithm starts with the 32-bit Multiplicand and 32-bit Multiplier registers set to their named values and the 64-bit Product register set to 0. This algorithm only forms a 32-bit sum, so only the left half of the 64-bit Product register is changed by the addition.

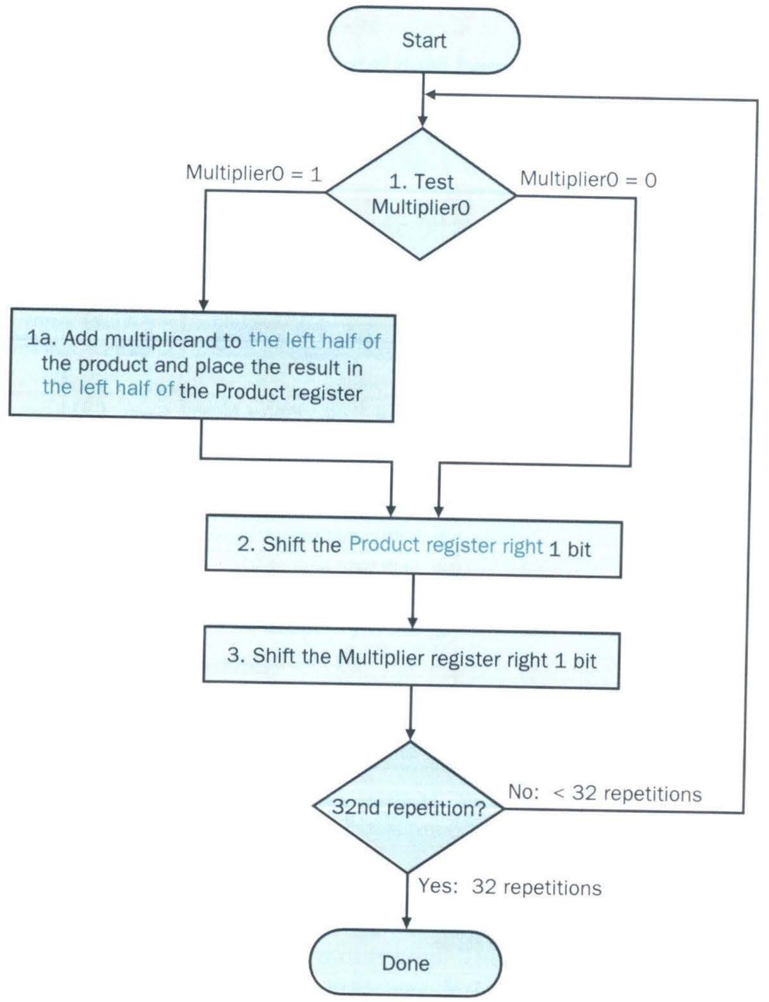

Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: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

FIGURE 4.29 The second multiplication algorithm, using the hardware in Figure 4.28. In this version, the Product register is shifted right instead of shifting the multiplicand. Color type shows the changes from Figure 4.26.

Second Multiply Algorithm

Example

Multiply $0010_{two} \times 0011_{two}$ using the algorithm in Figure 4.29.

Answer

Figure 4.30 shows the revised 4-bit example, again giving a product of $0000\ 0110_{two}$.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001①	0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	000①	0010	0001 0000
2	1a: 1 => Prod = Prod + Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	000⓪	0010	0001 1000
3	1: 0 => no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	000⓪	0010	0000 1100
4	1: 0 => no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

FIGURE 4.30 Multiply example using second algorithm in Figure 4.29. The bit examined to determine the next step is circled in color.

Final Version of the Multiplication Algorithm and Hardware

The final observation of the frugal computer pioneers was that the Product register had wasted space that matched exactly the size of the multiplier: As the wasted space in the product disappears, so do the bits of the multiplier. In response, the third version of the multiplication algorithm combines the rightmost half of the product with the multiplier. Figure 4.31 shows the hardware. The least significant bit of the 64-bit Product register (Product0) now is the bit to be tested.

The algorithm starts by assigning the multiplier to the right half of the Product register, placing 0 in the upper half. Figure 4.32 shows the new steps.

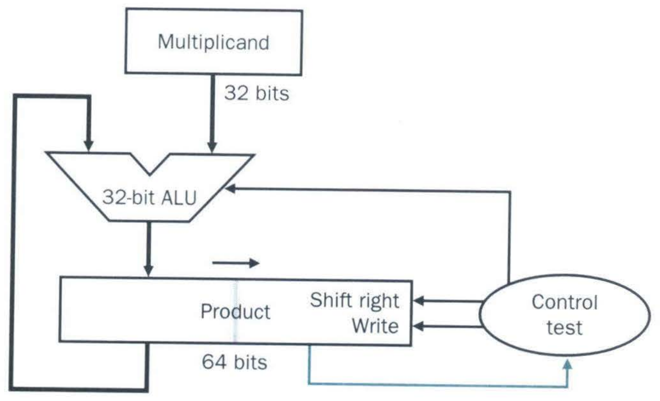


FIGURE 4.31 Third version of the multiplication hardware. Comparing with the second version in Figure 4.28 on page 254, the separate Multiplier register has disappeared. The multiplier is placed instead in the right half of the Product register.

Third Multiply Algorithm

Example

Multiply $0010_{two} \times 0011_{two}$ using the algorithm in Figure 4.32.

Answer

Figure 4.33 shows the revised 4-bit example for the final algorithm.

Signed Multiplication

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

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

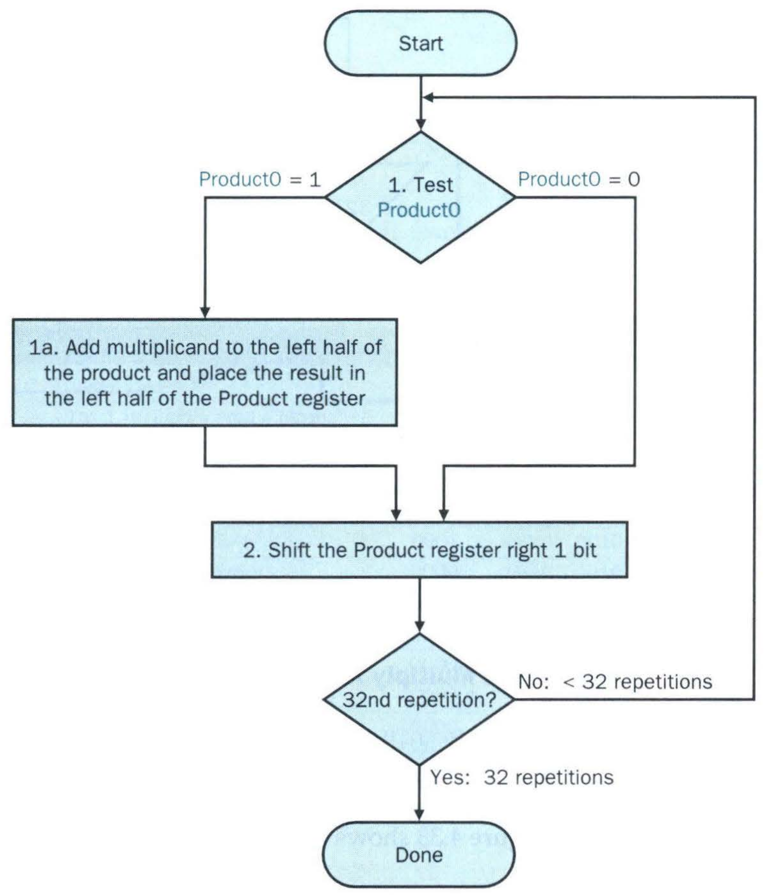

Start
1. Test Product0
Product0 = 1
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

FIGURE 4.32 The third multiplication algorithm. It needs only two steps because the Product and Multiplier registers have been combined. Color type shows changes from Figure 4.29.

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

FIGURE 4.33 Multiply example using third algorithm in Figure 4.32. The bit examined to determine the next step is circled in color.

Booth's Algorithm

A more elegant approach to multiplying signed numbers than above is called *Booth's algorithm.* It starts with the observation that with the ability to both add and subtract there are multiple ways to compute a product. Suppose we want to multiply 2_{ten} by 6_{ten}, or 0010_{two} by 0110_{two}:

```
          0010two
    x     0110two
    ---------------
    +        0000   shift (0 in multiplier)
    +       0010    add   (1 in multiplier)
    +      0010     add   (1 in multiplier)
    +     0000      shift (0 in multiplier)
    ---------------
        00001100two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$$6_{ten} = -2_{ten} + 8_{ten}$$

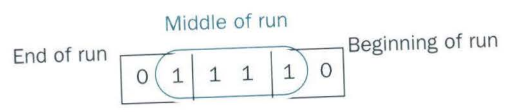
or

$$0110_{two} = -0010_{two} + 1000_{two}$$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

```
          0010two
    x     0110two
    ---------------
    +        0000 shift (0 in multiplier)
    -       0010  sub (first 1 in multiplier)
    +      0000   shift (middle of string of 1s)
    +     0010    add (prior step had last 1)
    ---------------
        00001100two
```

Booth invented this approach in a quest for speed because in machines of his era shifting was faster than addition. Indeed, for some patterns his algorithm would be faster; it's our good fortune that it handles signed numbers as well, and we'll prove this later. The key to Booth's insight is in his classifying groups of bits into the beginning, the middle, or the end of a run of 1s:



Of course, a string of 0s already avoids arithmetic, so we can leave these alone.

If we are limited to looking at just 2 bits, we can then try to match the situation in the preceding drawing, according to the value of these 2 bits:

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	$0000111\mathbf{10}00_{two}$
1	1	Middle of a run of 1s	$00001\mathbf{11}1000_{two}$
0	1	End of a run of 1s	$000\mathbf{01}111000_{two}$
0	0	Middle of a run of 0s	$0\mathbf{00}01111000_{two}$

Booth's algorithm changes the first step of the algorithm in Figure 4.32—looking at 1 bit of the multiplier and then deciding whether to add the multiplicand—to looking at 2 bits of the multiplier. The new first step, then, has four cases, depending on the values of the 2 bits. Let's assume that the pair of bits examined consists of the current bit and the bit to the right—which was the current bit in the previous step. The second step is still to shift the product right. The new algorithm is then the following:

1. Depending on the current and previous bits, do one of the following:

   00: Middle of a string of 0s, so no arithmetic operation.

   01: End of a string of 1s, so add the multiplicand to the left half of the product.

   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.

   11: Middle of a string of 1s, so no arithmetic operation.

2. As in the previous algorithm, shift the Product register right 1 bit.

Now we are ready to begin the operation, shown in Figure 4.34. It starts with a 0 for the mythical bit to the right of the rightmost bit for the first stage. Figure 4.34 compares the two algorithms, with Booth's on the right. Note that

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod – Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

FIGURE 4.34 Comparing algorithm in Figure 4.32 and Booth's algorithm for positive numbers. The bit(s) examined to determine the next step is circled in color.

Booth's operation is now identified according to the values in the 2 bits. By the fourth step, the two algorithms have the same values in the Product register.

The one other requirement is that shifting the product right must preserve the sign of the intermediate result, since we are dealing with signed numbers. The solution is to extend the sign when the product is shifted to the right. Thus, step 2 of the second iteration turns $1110\ 0011\ 0_{two}$ into $1111\ 0001\ 1_{two}$ instead of $0111\ 0001\ 1_{two}$. This shift is called an *arithmetic right shift* to differentiate it from a logical right shift.

Example

Booth's Algorithm

Let's try Booth's algorithm with negative numbers: $2_{ten} \times -3_{ten} = -6_{ten}$, or $0010_{two} \times 1101_{two} = 1111\ 1010_{two}$.

Answer

Figure 4.35 shows the steps.

Our example multiplies one bit at a time, but it is possible to generalize Booth's algorithm to generate multiple bits for faster multiplies (see Exercise 4.53).

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⟹ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⟹ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⟹ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⟹ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

FIGURE 4.35 Booth's algorithm with negative multiplier example. The bits examined to determine the next step are circled in color.

Hardware Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts, adds, and subtracts. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2, so almost every compiler will substitute a left shift for a multiply by a power of 2.

Multiply by 2^i via Shift

Example

Let's multiply 5_{ten} by 2_{ten} using a left shift by 1.

Answer

Given that

$$101_{two} = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)_{ten} = 4 + 0 + 1_{ten} = 5_{ten}$$

if we shift left 1 bit, we get

$$1010_{two} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)_{ten}$$
$$= 8 + 0 + 2 + 0_{ten} = 10_{ten}$$

and

$$5 \times 2^1{}_{ten} = 10_{ten}$$

Hence the MIPS `sll` instruction can be used for multiplies by powers of 2.

Now that we have seen Booth's algorithm work, we are ready to see *why* it works for two's complement signed integers. Let a be the multiplier and b be the multiplicand and we'll use a_i to refer to bit i of a. Recasting Booth's algorithm in terms of the bit values of the multiplier yields this table:

a_i	a_{i-1}	Operation
0	0	Do nothing
0	1	Add b
1	0	Subtract b
1	1	Do nothing

Instead of representing Booth's algorithm in tabular form, we can represent it as the expression

$$(a_{i-1} - a_i)$$

where the value of the expression means the following actions:

$$\begin{array}{rl} 0: & \text{do nothing} \\ +1: & \text{add } b \\ -1: & \text{subtract } b \end{array}$$

Since we know that shifting of the multiplicand left with respect to the Product register can be considered multiplying by a power of 2, Booth's algorithm can be written as the sum

$$\begin{array}{rl} & (a_{-1} - a_0) \times b \times 2^0 \\ + & (a_0 - a_1) \times b \times 2^1 \\ + & (a_1 - a_2) \times b \times 2^2 \\ \ldots & \ldots \\ + & (a_{29} - a_{30}) \times b \times 2^{30} \\ + & (a_{30} - a_{31}) \times b \times 2^{31} \end{array}$$

We can simplify this sum by noting that

$$-a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = (2a_i - a_i) \times 2^i = a_i \times 2^i$$

recalling that $a_{-1} = 0$ and by factoring out b from each term:

$$b \times ((a_{31} \times -2^{31}) + (a_{30} \times 2^{30}) + (a_{29} \times 2^{29}) + \ldots + (a_1 \times 2^1) + (a_0 \times 2^0))$$

The long formula in parentheses to the right of the first multiply operation is simply the two's complement representation of a (see page 213.) Thus the sum is further simplified to

$$b \times a$$

Hence Booth's algorithm does in fact perform two's complement multiplication of a and b.

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Hardware Software Interface

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. To avoid overflow, Hi must be 0 for `multu` or must be the replicated sign of Lo for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer Hi to a general-purpose register to test for overflow.

Summary

Multiplication is accomplished by simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of two. Signed multiplication is more challenging, with Booth's algorithm rising to the challenge with essentially a clever factorization of the two's complement number representation of the multiplier.

Elaboration: The original reason for Booth's algorithm was speed because early machines could shift faster than they could add. The hope was that this encoding scheme would increase the number of shifts. This algorithm is sensitive to particular bit patterns, however, and may actually increase the number of adds or subtracts. For example, bit patterns that alternate 0 and 1, called *isolated 1s*, will cause the hardware to add or subtract at each step. Looking at more bits to carefully avoid isolated 1s can reduce the number of adds in the worst case. Greater advantage comes from performing multiple bits per step, which we explore in Exercise 4.53.

Even faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder. When adding such a large column of numbers, a *carry save adder* is useful (see Exercises 4.49 to 4.52).

Elaboration: The replacement of a multiply by a shift, as in the example on page 262, is an instance of a general compiler optimization strategy called *strength reduction.*

4.7 Division

Divide et impera.

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

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

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

```
                   1001ten      Quotient
Divisor 1000ten   |1001010ten   Dividend
                  -1000
                      10
                      101
                      1010
                     -1000
                        10ten   Remainder
```

The two operands (*dividend* and *divisor*) and the result (*quotient*) of divide are accompanied by a second result called the *remainder*. Here is another way to express the relationship between the components:

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

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

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

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

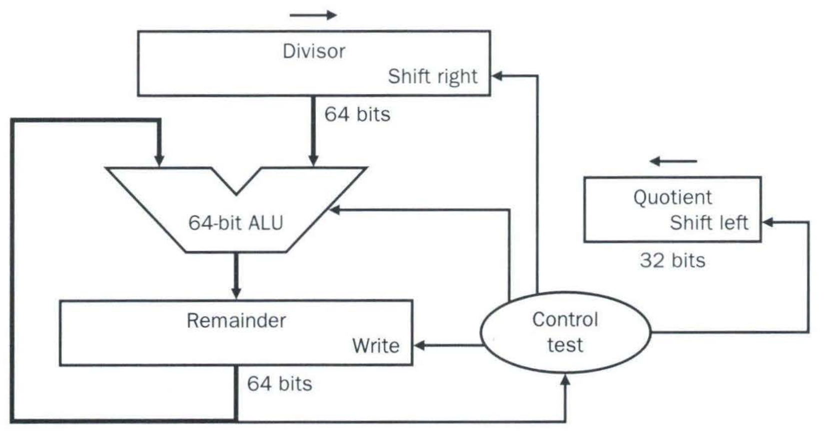


FIGURE 4.36 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

each of the three evolutionary versions explicit with drawings and examples, as we did for multiply, we will save space by giving a sketch for the two intermediate steps and then give the final algorithm in detail.

First Version of the Division Algorithm and Hardware

Figure 4.36 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 4.37 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

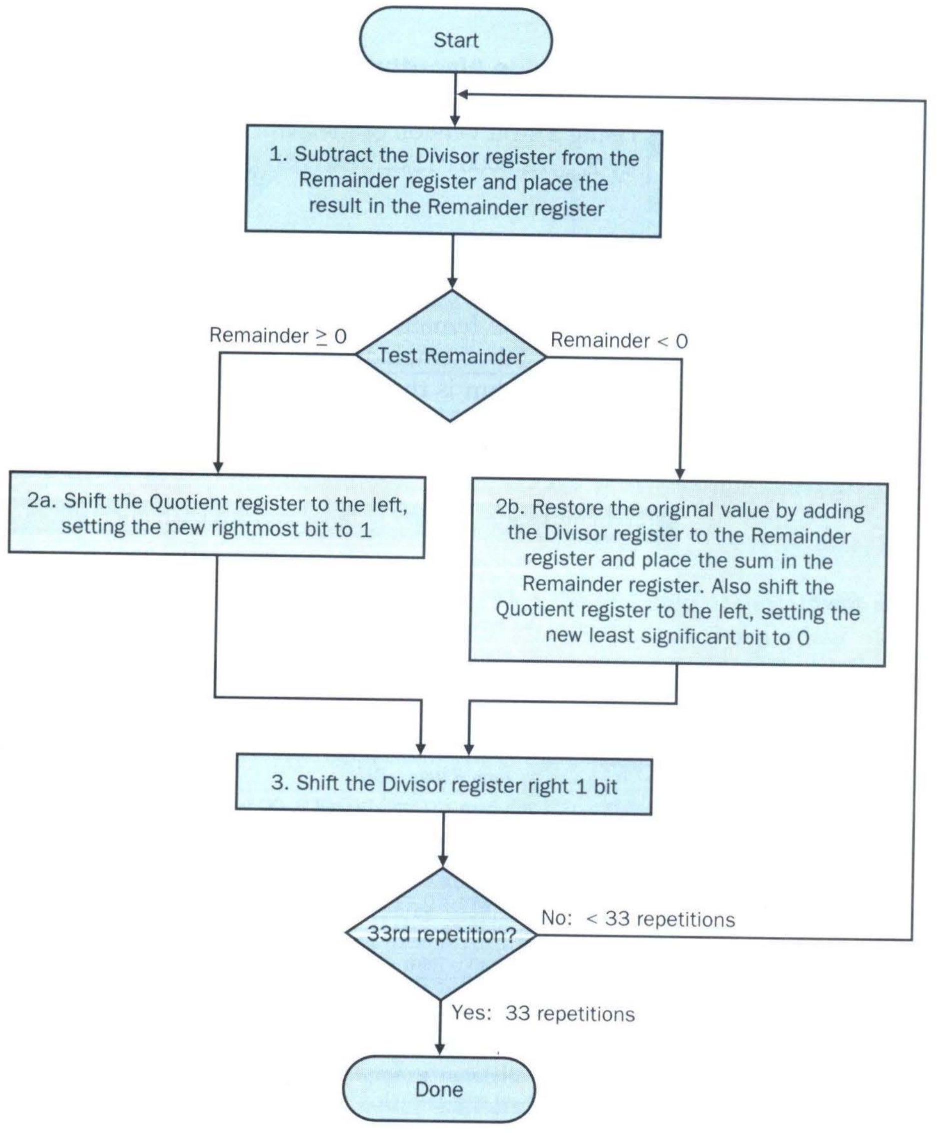

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to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

FIGURE 4.37 The first division algorithm, using the hardware in Figure 4.36. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times; the reason for the apparent extra step will become clear in the next version of the algorithm.

First Divide Algorithm

Example

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

Answer

Figure 4.38 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

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

FIGURE 4.38 Division example using first algorithm in Figure 4.37. The bit examined to determine the next step is circled in color.

Second Version of the Division Algorithm and Hardware

Once again the frugal computer pioneers recognized that, at most, half of the divisor has useful information, and so both the divisor and ALU could potentially be cut in half. Shifting the remainder to the left instead of shifting the divisor to the right produces the same alignment and accomplishes the goal of simplifying the hardware necessary for the ALU and the divisor. Figure 4.39 shows the simplified hardware for the second version of the algorithm.

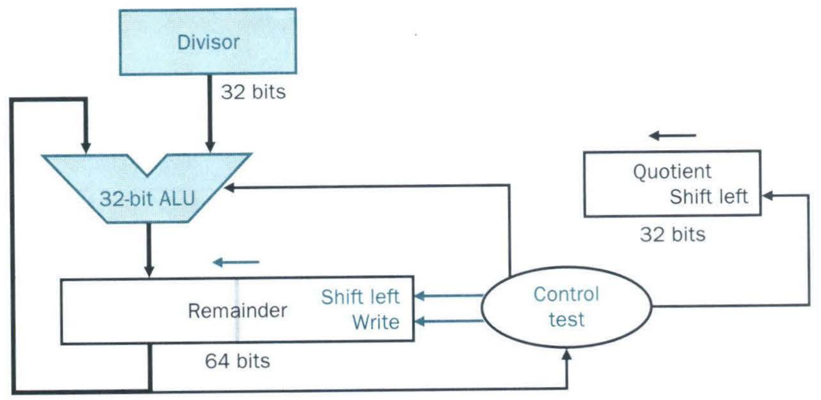


FIGURE 4.39 Second version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 4.36, the ALU and Divisor registers are halved and the remainder is shifted left. These changes are highlighted.

Another change comes from noticing that the first step of the current algorithm cannot produce a 1 in the quotient bit; if it did, then the quotient would be too large for the register. By switching the order of the operations to shift and then subtract, one iteration of the algorithm can be removed. When the algorithm terminates, the remainder will be found in the left half of the Remainder register.

Final Version of Division Algorithm and Hardware

With the same insight and motivation as in the third version of the multiplication algorithm, computer pioneers saw that the Quotient register could be eliminated by shifting the bits of the quotient into the Remainder instead of shifting in 0s as in the preceding algorithm. Figure 4.40 shows the third version of the algorithm.

We start the algorithm by shifting the Remainder left as before. Thereafter, the loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half (see Figure 4.41). The consequence of combining the two registers and the new order of the operations in the loop is that the remainder will be shifted left one time too many. Thus the final correction step must shift back only the remainder in the left half of the register.

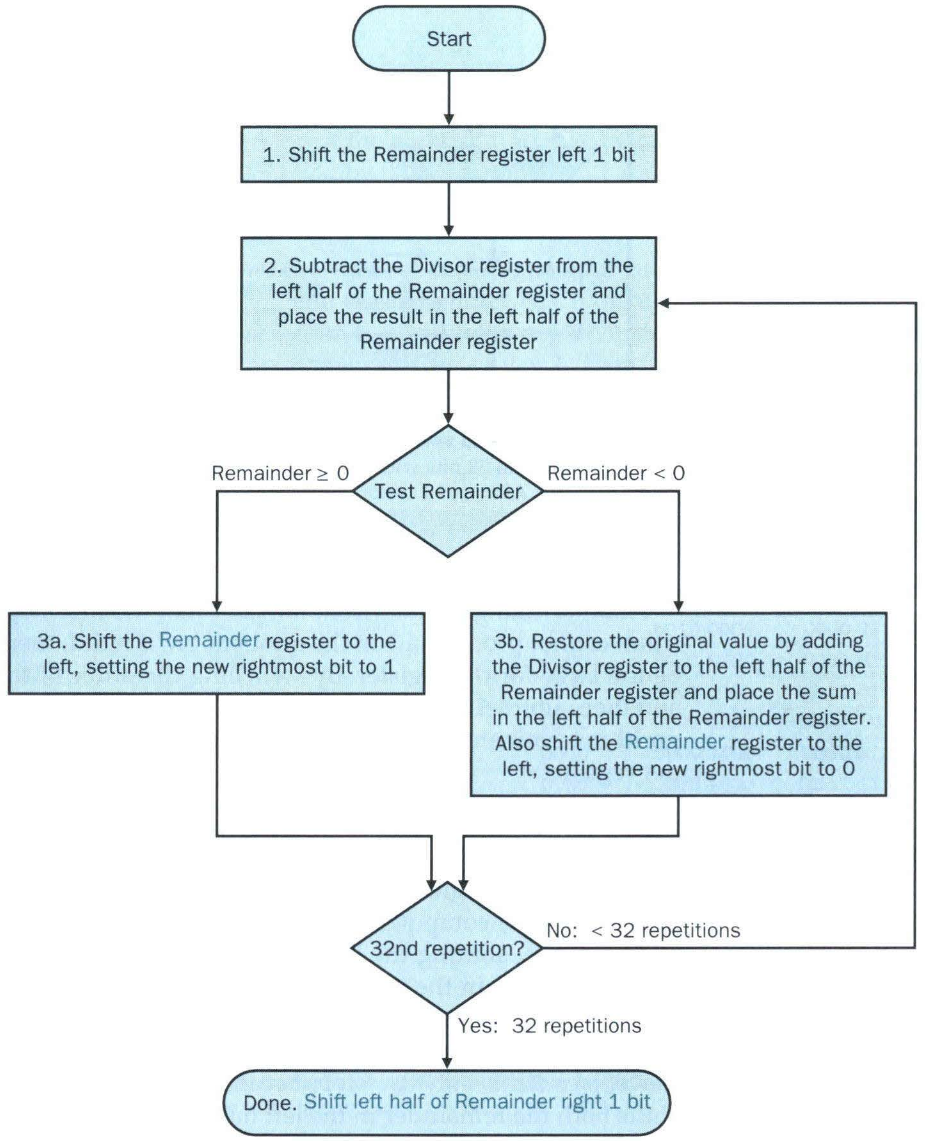

Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

FIGURE 4.40 The third division algorithm has just two steps. The Remainder register shifts left.

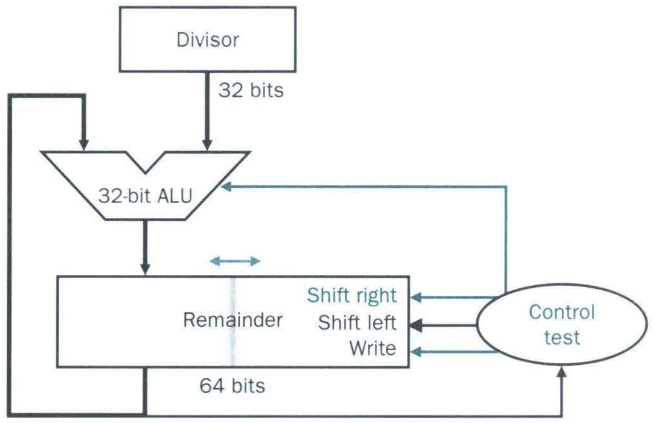


FIGURE 4.41 Third version of the division hardware. This version combines the Quotient register with the right half of the Remainder register.

Third Divide Algorithm

Example

Use the third version of the algorithm to divide $0000\ 0111_{two}$ by 0010_{two}.

Answer

Figure 4.42 shows how the quotient is created in the bottom of the Remainder register and how both are shifted left in a single operation.

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⟹ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⟹ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⟹ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⟹ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

FIGURE 4.42 Division example using third algorithm in Figure 4.40. The bit examined to determine the next step is circled in color.

INTEL - 1012

Signed Division

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

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

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

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

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

Checking the results:

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

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

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

Rewriting our basic formula to calculate the remainder:

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

So,

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

Checking the results again:

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

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

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

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

We calculate the other combinations by following the same rule:

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

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

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

Divide in MIPS

You may have already observed that the same hardware can be used for both multiply and divide. The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts. For example, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide. As we might expect from the algorithm above, Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.

To handle both signed integers and unsigned integers, MIPS has two instructions: *divide* (`div`) and *divide unsigned* (`divu`). The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

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 machines distinguish these two anomalous events. MIPS software must check the divisor to discover division by 0 as well as overflow.

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 4.43 summarizes the additions to the MIPS architecture for the last two sections.

Elaboration: The reason for needing an extra iteration for the first algorithm and the early shift in the second and third algorithms involves the placement of the dividend in the Remainder register. We expect to have a 32-bit quotient and a 32-bit divisor, but each is really a 31-bit integer plus a sign bit. The product would be 31+31, or 62 bits plus a single sign bit; the hardware can then support only a 63-bit dividend. Given that registers are normally powers of 2, this means we must place the 63-bit dividend properly in the 64-bit Remainder register. If we place the 63 bits to the right, we need to run the algorithm for an extra step to get to that last bit. A better solution is to shift early, thereby saving a step of the algorithm.

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 4.54; the algorithm here is called *restoring* division.

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7`, `$t0–$t9`, `$gp`, `$fp`, `$zero`, `$sp`, `$ra`, `$at`, `Hi`, `Lo`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants. `Hi` and `Lo` contain the results of multiply and divide.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

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`	Used to copy Exception PC plus other special registers
	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
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; logical AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; logical OR
	and immediate	`andi $s1,$s2,100`	`$s1` = `$s2` & 100	Logical AND reg, constant
	or immediate	`ori $s1,$s2,100`	`$s1` = `$s2` | 100	Logical OR reg, 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
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 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
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 4.43 MIPS architecture revealed thus far. Color indicates the portions revealed since Figure 4.7 on page 228. MIPS machine language is listed on the back endpapers of this book. *(page 274)*

4.8 Floating Point

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

American proverb

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

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

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

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

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

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

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

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

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we'll 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

The designer of a floating-point representation must find a compromise between the size of the significand 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 accuracy and range: Increasing the size of the significand enhances the accuracy of the significand, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 3 reminds us, good design demands good compromises.

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 *significand* is the 23-bit number in the fraction. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.

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

In general, floating-point numbers are of the form

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

F involves the value in the significand field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon.

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

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. This situation occurs when the negative exponent is too large to fit in the exponent field. To distinguish it from overflow, people call this event *underflow.*

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

The representation of a double precision floating-point number takes two 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 *significand* is the 52-bit number in the fraction.

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

significand (continued)
32 bits

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 accuracy because of the large 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 significand is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1+52). Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

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

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

where the bits of the significand represent the fraction 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 significand 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 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 test of less than, greater than, or equal to 0 to be performed quickly.

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. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

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

IEEE 754 uses a bias of 127 for single precision, so –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

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

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

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

and so when we add the bias 127 to 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

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

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 word?

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 significand field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

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

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 4.12.

Floating-Point Addition

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

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

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

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

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

Step 2. Next comes the addition of the significands:

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

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

Step 3. This sum is not in normalized scientific notation, so we need to correct it. Again, there are multiple representations of this number; we pick the normalized form:

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

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

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

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

is rounded to four digits in the significand to

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

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

Figure 4.44 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 300). Thus, for single precision, the maximum exponent is 127 and the minimum exponent is –126. The limits for double precision are 1023 and –1022.

For simplicity, we assume truncation in step 4, one of four rounding options in IEEE 754 floating point. The accuracy of floating-point calculations depends a great deal on the accuracy of rounding, so although it is easy to follow, truncation leads away from accuracy.

Decimal Floating-Point Addition

Example

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 4.44.

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 machines dedicate hardware to run floating-point operations as fast as possible. Figure 4.45 sketches the basic organization of hardware for floating-point addition.

Floating-Point Multiplication

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

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

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

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:

New exponent = (10 + 127) + (–5 +127) = (5 + 2 × 127) = 259

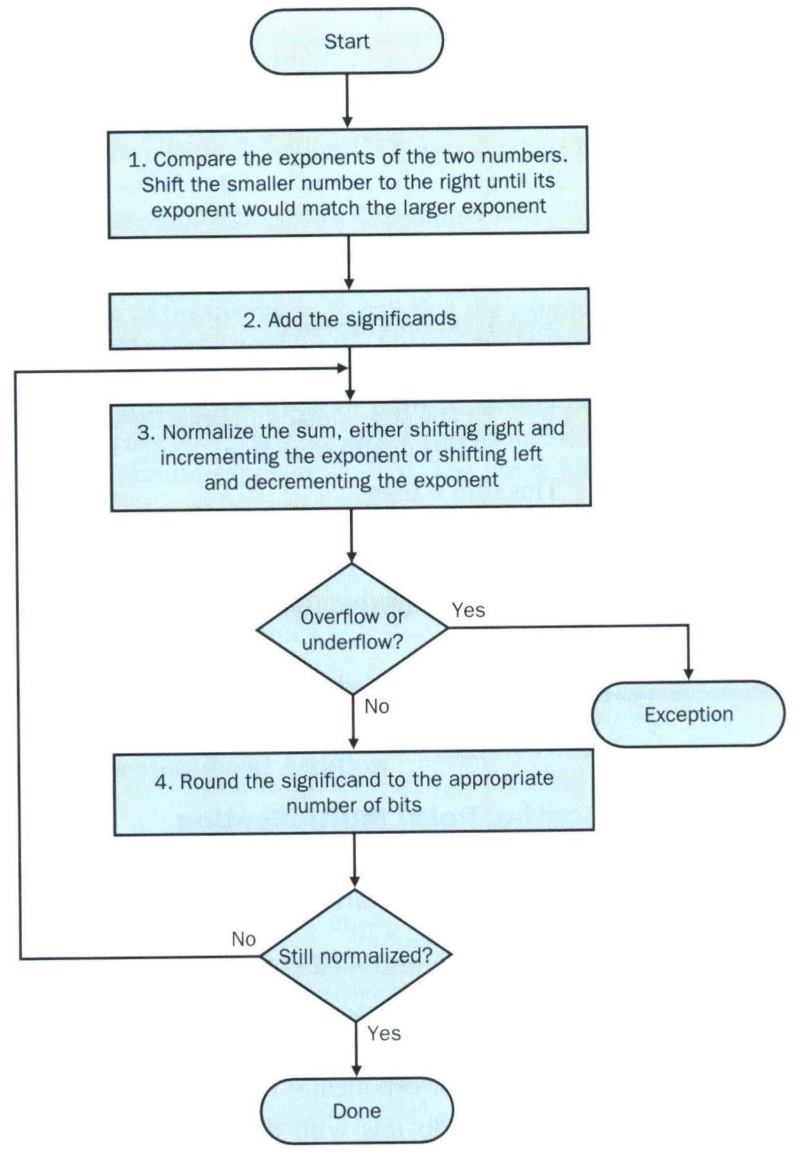

Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?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

FIGURE 4.44 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.

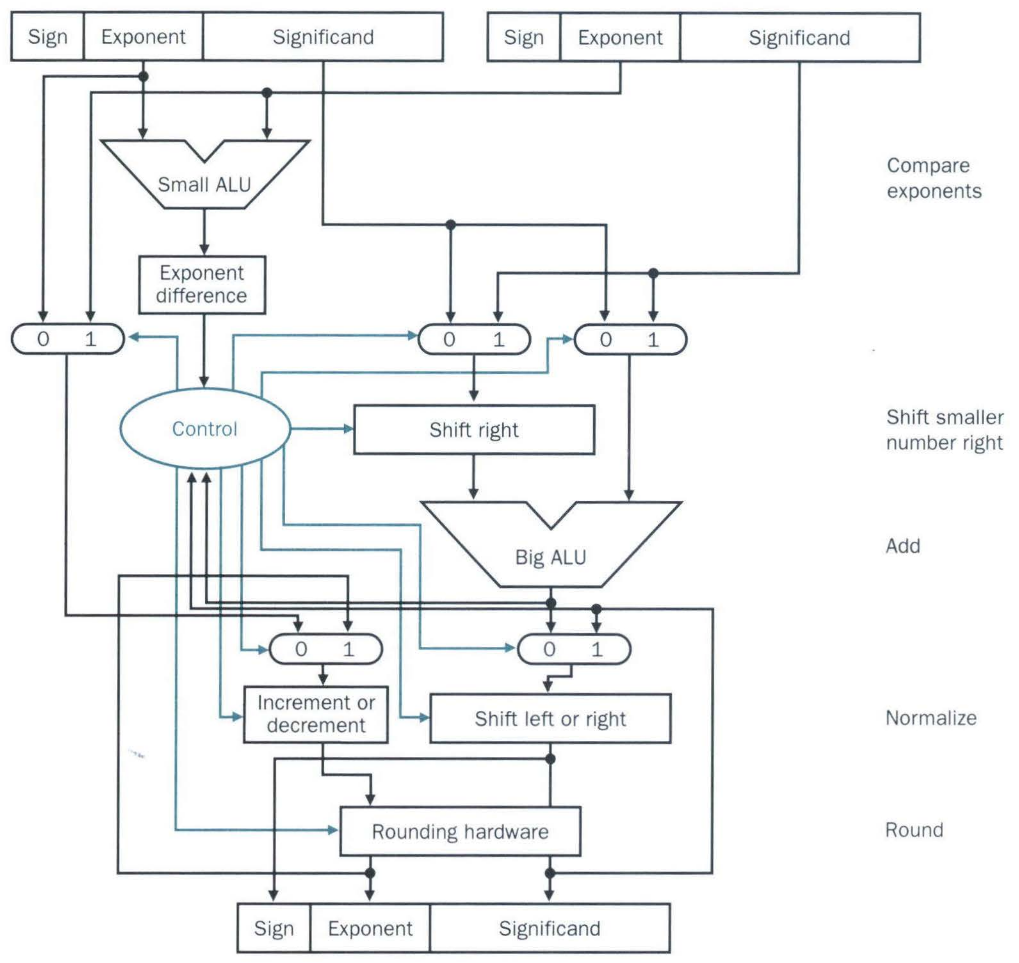


FIGURE 4.45 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 4.44 correspond to each block, from top to bottom. First the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference 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.

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

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 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 correct it. Again, there are multiple representations of this number, so we must pick the normalized form:

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

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 4.46 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 4.46.

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:

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

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:

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

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

Floating-Point Instructions in MIPS

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

- Floating-point *addition, single* (`add.s`) and *addition, double* (`add.d`)
- Floating-point *subtraction, single* (`sub.s`) and *subtraction, double* (`sub.d`)
- Floating-point *multiplication, single* (`mul.s`) and *multiplication, double* (`mul.d`)
- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where `x` may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

The MIPS designers decided to add separate floating-point registers—called `$f0`, `$f1`, `$f2`, . . .—used either for single precision or double precision. Hence they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1    $f4,x($sp)    # Load 32-bit F.P. number into F4
lwc1    $f6,y($sp)    # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6   # F2 = F4 + F6 single precision
swc1    $f2,z($sp)    # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

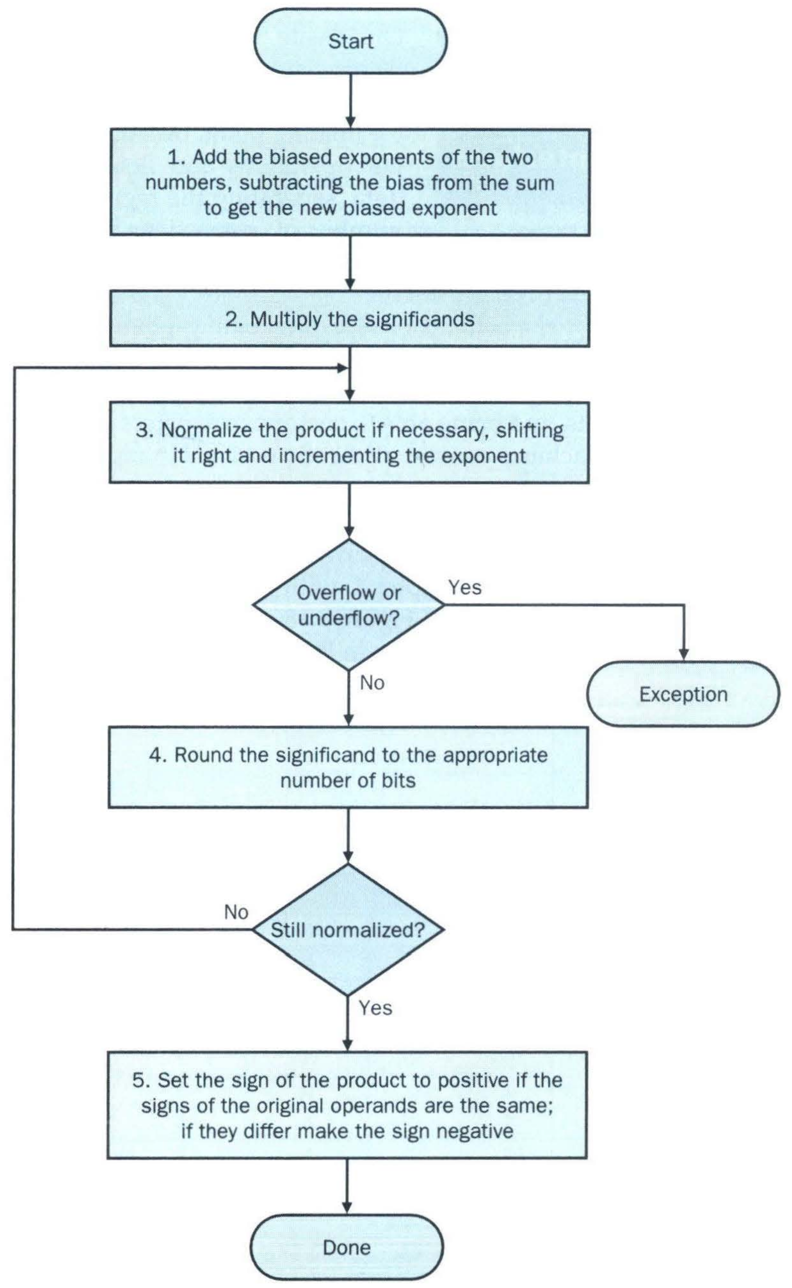

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
No
Exception
4. Round the significand to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

FIGURE 4.46 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.

Hardware Software Interface

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

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

Figure 4.47 summarizes the floating-point portion of the MIPS architecture revealed in Chapter 4, with the additions to support floating point shown in color. Similar to Figure 3.18 on page 153 in Chapter 3, we show the encoding of these instructions in Figure 4.48.

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2` = `$f4` + `$f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2` = `$f4` – `$f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2` = `$f4` × `$f6`	FP. multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2` = `$f4` / `$f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2` = `$f4` + `$f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2` = `$f4` – `$f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2` = `$f4` × `$f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2` = `$f4` / `$f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1` = Memory[`$s2` + 100]	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[`$s2` + 100] = `$f1`	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2` < `$f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2` < `$f4`) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
`add.s`	R	17	16	6	4	2	0	`add.s $f2,$f4,$f6`
`sub.s`	R	17	16	6	4	2	1	`sub.s $f2,$f4,$f6`
`mul.s`	R	17	16	6	4	2	2	`mul.s $f2,$f4,$f6`
`div.s`	R	17	16	6	4	2	3	`div.s $f2,$f4,$f6`
`add.d`	R	17	17	6	4	2	0	`add.d $f2,$f4,$f6`
`sub.d`	R	17	17	6	4	2	1	`sub.d $f2,$f4,$f6`
`mul.d`	R	17	17	6	4	2	2	`mul.d $f2,$f4,$f6`
`div.d`	R	17	17	6	4	2	3	`div.d $f2,$f4,$f6`
`lwc1`	I	49	20	2		100		`lwc1 $f2,100($s4)`
`swc1`	I	57	20	2		100		`swc1 $f2,100($s4)`
`bc1t`	I	17	8	1		25		`bc1t 25`
`bc1f`	I	17	8	0		25		`bc1f 25`
`c.lt.s`	R	17	16	4	2	0	60	`c.lt.s $f2,$f4`
`c.lt.d`	R	17	17	4	2	0	60	`c.lt.d $f2,$f4`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

FIGURE 4.47 MIPS floating-point architecture revealed thus far. See Appendix A, section A.10, on page A-49, for more detail.

op(31:26):

28–26 / 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):

23–21 / 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FlPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):

2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

FIGURE 4.48 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure lw is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op = 010001_{two}) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction) is 000001_{two} and the op field (bits 31–26) is 010001_{two}. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (f = s so rs = 10000) or double precision (f = d so rs = 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 = 1 =>bc1.t) or false (bit 16 = 0 =>bc1.f). Rfmt and TLB instruction encodings are found in Figure 3.18 on page 153. Instructions in color are described in Chapters 3 or 4, with Appendix A covering all instructions.

Example

Compiling a Floating-Point C Program into MIPS 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 $f12 and the result should go in $f0. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

Answer

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

```
f2c:
     lwc1  $f16,const5($gp)    # $f16 = 5.0 (5.0 in memory)
     lwc1  $f18,const9($gp)    # $f18 = 9.0 (9.0 in memory)
```

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

```
     div.s  $f16, $f16, $f18   # $f16 = 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 ($f12):

```
     lwc1   $f18, const32($gp)# $f18 = 32.0
     sub.s  $f18, $f12, $f18   # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in $f0 as the return result, and then return:

```
     mul.s  $f0,  $f16, $f18   # $f0 = (5/9)*(fahr - 32.0)
     jr     $ra                # return
```

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

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

Example

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

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

      for (i = 0; i! = 32; i = i + 1)
          for (j = 0; j! = 32; j = j + 1)
             for (k = 0; k! = 32; k = k + 1)
                 x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

The array starting addresses are parameters, so they are in `$a0`, `$a1`, and `$a2`. Assume that the integer variables are in `$s0`, `$s1`, and `$s2`, respectively. What is the MIPS assembly code for the body of the procedure?

Answer

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

We keep the code simpler by using the assembly language pseudoinstructions `li` (which loads a constant into a register), and `l.d` and `s.d` (which the assembler turns into a pair of data transfer instructions, `lwc1` or `swc1`, to a pair of floating-point registers).

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

```
mm:...
      li     $t1, 32    # $t1 = 32 (row size/loop end)
      li     $s0, 0     # i = 0; initialize 1st for loop
L1:   li     $s1, 0     # j = 0; restart 2nd for loop
L2:   li     $s2, 0     # k = 0; restart 3rd for loop
```

To calculate the address of `x[i][j]`, we need to know how a 32 × 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the `i` "single-dimensional arrays," or rows, to get the one we want. Thus we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead (see page 262):

```
sll   $t2, $s0, 5     # $t2 = i * 2^5 (size of row of x)
```

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

```
addu  $t2, $t2, $s1   # $t2 = i * size(row) + j
```

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

```
sll    $t2, $t2, 3    # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of `x`, giving the address of `x[i][j]`, and then load the double precision number `x[i][j]` into `$f4`:

```
addu  $t2, $a0, $t2   # $t2 = byte address of x[i][j]
l.d   $f4, 0($t2)     # $f4 = 8 bytes of x[i][j]
```

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

```
L3:  sll  $t0, $s2, 5   # $t0 = k * 2^5 (size of row of z)
     addu $t0, $t0, $s1 # $t0 = k * size(row) + j
     sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
     addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
     l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
```

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

```
sll   $t2, $s0, 5     # $t0 = i * 2^5 (size of row of y)
addu  $t0, $t0, $s2   # $t0 = i * size(row) + k
sll   $t0, $t0, 3     # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0   # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)    # $f18 = 8 bytes of y[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `y` and `z` located in registers `$f18` and `$f16`, and then accumulate the sum in `$f4`.

```
mul.d $f16, $f18, $f16# $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # f4 = x[i][j] + y[i][k] * z[k][j]
```

The final block increments the index k and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in $f4 into x[i][j].

```
addiu  $s2, $s2, 1       # $k k + 1
bne    $s2, $t1, L3      # if (k != 32) go to L3
s.d    $f4, 0($t2)       # x[i][j] = $f4
```

Similarly, these final four 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.

```
addiu  $s1, $s1, 1       # $j = j + 1
bne    $s1, $t1, L2      # if (j != 32) go to L2
addiu  $s0, $s0, 1       # $i = i + 1
bne    $s0, $t1, L1      # if (i != 32) go to L1
...
```

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

Only 16 of the 32 MIPS floating-point registers can be used for single precision operations: $f0, $f2, $f4, . . . , $f30. Double precision is computed using pairs of these registers. The odd-numbered floating-point registers are used only to load and store the right half of 64-bit floating-point numbers. A later version of the MIPS instruction set, MIPS II, added l.d and s.d to the hardware instruction set. An even later version, MIPS IV, added indexed addressing for floating-point data transfers, removing the need for the fourth instruction of the five-instruction load sequences above.

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, were optionally available as a second chip. Such optional accelerator chips are called *coprocessors*, and explain the acronym for floating-point loads in MIPS: lwc1 means load word to coprocessor 1, the floating-point unit. (Coprocessor 0 deals with virtual memory, described in Chapter 7.) Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and hence the term "coprocessor" joins "accumulator" and "core memory" as quaint terms that date the speaker.

Elaboration: Although there are many ways to throw hardware at floating-point multiply to make it go fast, floating-point division is considerably more challenging to make fast and accurate. Slow divides in early computers led to removal of divides from many algorithms, but parallel computers have inspired rediscovery of divide-intensive algorithms that work better on these machines. Hence we may need faster divides.

One technique to leverage a fast multiplier is *Newton's iteration*, where division is recast as finding the zero of a function to find the reciprocal $1/x$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solves this problem by calculating an extra-precise reciprocal, and IBM relies on fused multiply-add to solve it (see section 4.9).

The *SRT division* technique instead tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder, relying on subsequent steps to correct wrong guesses. A Cyrix chip uses this technique to generate 16 bits per step!

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 get 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 2 extra bits on the right during intermediate calculations, called *guard* and *round*, respectively. Let's do a decimal example to illustrate the value of these extra digits.

Rounding with Guard Digits

Example

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

Answer

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

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

INTEL - 1012

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

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

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

Elaboration: Although the example above really needed just one extra bit, multiply can need two. A binary product may have one leading 0 bit, hence the normalizing step must shift the product 1 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.

The goal of the extra rounding bits is to allow the machine to get the same results as if the intermediate results were calculated to infinite precision and then rounded. Thus 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.

Summary

The Big Picture below reinforces the stored-program concept from Chapter 3; 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

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{bias})}$$

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

The Big Picture

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

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

Hardware Software Interface

In the last chapter we presented the storage classes of the programming language C (see the Hardware Software Interface section on page 140). The following table shows some of the C data types together with the MIPS data transfer instructions and instructions that operate on those types that appear in Chapters 3 and 4.

C type	Data transfers	Operations
`int`	`lw, sw, lui`	`addu, addiu, subu, mult, div, and, andi, or, ori, slt, slti`
`unsigned int`	`lw, sw, lui`	`addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu`
`char`	`lb, sb, lui`	`addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu`
`bit field`	`lw, sw, lui`	`and, andi, or, ori, sll, srl`
`float`	`lwc1, swc1`	`add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s`
`double`	`lwc1, swc1`	`add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d`

Elaboration: The IEEE 754 floating-point standard is filled with little widgets to help the programmer try to maintain accuracy. We'll cover a few here, but take a look at the references at the end of section 4.12 to learn more.

There are four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The Internal Revenue Service 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, giving the rounding mode its name. This mode is the most commonly used.

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. To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. Hence the full MIPS instruction set has many flavors of compares to support NaNs.

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

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

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. Also, if programmers do not expect denorms, their programs may be surprised.

Here are the encodings of IEEE 754 floating-point numbers, with the sign bit determining the sign:

Single precision		Double precision		Object represented
Exponent	Significand	Exponent	Significand	
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)

4.9 Real Stuff: Floating Point in the PowerPC and 80x86

Both the PowerPC and 80x86 have regular multiply and divide instructions that operate entirely on registers, unlike the reliance on Hi and Lo in MIPS. (In fact, later versions of the MIPS instruction set have added similar instructions.)

The main differences are found in floating-point instructions. PowerPC is like MIPS except for one novel instruction and twice as many registers: PowerPC offers 32 single precision and 32 double precision floating-point registers. The 80x86 floating-point architecture, on the other hand, is completely different from all other computers in the world.

The Multiply-Add Instruction of the PowerPC

The matrix multiply on page 294 relied on a multiply operation and an add operation, which is typical of many matrix and vector operations. Hence the PowerPC has a "fused" multiply-add instruction: a single instruction reads three operands, multiplies two operands and adds the third to the product, and writes the sum in the result operand. Hence the two MIPS floating-point instructions in the matrix multiply example would be replaced by one in PowerPC. This instruction can increase peak floating-point performance.

Fused multiply-add also performs the two operations and *then* rounds, unlike separate multiply and add instructions, which would round after each operation. The instructions also calculate extra bits for intermediate results to improve accuracy. Besides being potentially faster, the extra accuracy of fused multiply-add can also be helpful for calculating divide and square root, and in software libraries that calculate at higher precision than 64 bits. In fact, PowerPC hardware uses fused multiply-add hardware to calculate divide, and accurate division was the motivation for skipping the round between the two operations.

The 80x86 Floating-Point Architecture

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

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

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

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

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

The 80x86 floating-point operations can be divided into four major classes:

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

Figure 4.49 shows some of the 60 floating-point operations. We use the curly brackets `{}` to show optional variations of the basic operations: `{I}` means there is an integer version of the instruction, `{P}` means this variation will pop one operand off the stack after the operation, and `{R}` means reverse the order of the operands in this operation.

Not all combinations suggested by the notation are provided. Hence

```
F{I}SUB{R}{P}
```

represents these instructions found in the 80x86:

```
FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP
```

For the integer subtract instructions, there is no pop (`FISUBP`) or reverse pop (`FISUBRP`).

Note that we get even more combinations when including the operand modes for these operations. Figure 4.50 shows the many options for floating-point add, even ignoring the integer and pop versions of the instruction.

The floating-point instructions are encoded using the `ESC` opcode of the 8086 and the postbyte address specifier (see Figure 3.35 on page 185). The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit floating point or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

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

FIGURE 4.49 The floating-point instructions of the 80x86. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations in the first column push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the `FSTSW` instruction, either into the `AX` register or into memory, followed by an `SAHF` instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations.

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

FIGURE 4.50 The variations of operands for floating-point add in the 80x86.

Floating-point performance of the 80x86 family has traditionally lagged far behind other computers. It is hard to tell whether it is simply a lack of attention by Intel engineers, a disinterest by customers of PCs, if the fault lies with its architecture, or most likely some combination. We can say that many new architectures have been announced since 1980, and none have followed in Intel's footsteps.

4.10 Fallacies and Pitfalls

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

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

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

Fallacy: Floating-point addition is associative; that is, $x + (y + z) = (x + y) + z$.

Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number. For example, suppose $x = -1.5_{ten} \times 10^{38}$, $y = 1.5_{ten} \times 10^{38}$, and $z = 1.0$, and that these are all single precision numbers. Then

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

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

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

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

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

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

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

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

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

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

A solution would be to have an arithmetic right shift (see page 261) that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

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

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

The PowerPC, however, does have a fast shift instruction (*shift right algebraic*) that in conjunction with a special add (add with carry) gives the same answer as dividing by a power of 2.

Pitfall: The MIPS instruction add immediate unsigned `addiu` *sign-extends its 16-bit immediate field.*

Despite its name, `addiu` is used to add constants to signed integers when we don't care about overflow. MIPS has no subtract immediate instruction and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.

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

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 4.51). The following is the inside story behind the headlines.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a look-

A Closer Loo
Warning: Intel Inside
BUSINESS
MARKETS • HIGH TECH • ECONOMY
TALKING BUSINESS
Intel to work on software patch for Pentium bug
Quick fix:
PERSONAL TECHNOLOGY
Grove says I sorry about Pentium bu
INFO WORLD
The Voice of Personal Computing in the Enterprise
Photo finish
Bug Dodge Booed
Intel Knocked For Response On Flaw
Flawed Chip Bruises Intel
Investors react, stock plunges
INTEL'S MISTAKE
Chip Shot
Computer Giants' War Over Flaw in Pentium Jolts the PC Industry
San Francisco Chronicle
BUSINESS
Some Scientists Are Angry Over Flaw In Pentium Chip, and Intel's Response
Intel's Pentium Problem Persi
Pentium woes continue
Faulty FPU flubs math in certain equations
Multithreading gets lost on P100 systems
IBM to stop shipping Pentium PCs
TECHNOLOGY
Intel's Grove Airs Apology for Pentium Over the Internet

FIGURE 4.51 A sampling of newspaper and magazine articles from November 1994, including the *New York Times, San Jose Mercury News, San Francisco Chronicle,* and *Infoworld.* The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

up table containing –2, –1, 0, +1, or +2. The guess is multiplied by the divisor and subtracted from the remainder to generate a new remainder. Like nonrestoring division (see Exercise 4.54), if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

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

The following is a time line of the Pentium bug morality play:

- *July 1994:* Intel discovers the bug in the Pentium. The actual cost to fix the bug was several hundred thousand dollars. Following normal bug fix procedures, it will take months to make the change, reverify, and put the corrected chip into production. Intel planned to put good chips into production in January 1995, estimating that 3 to 5 million Pentiums would be produced with the bug.

- *September 1994:* A math professor at Lynchburg College in Virginia, Thomas Nicely, discovers the bug. After calling Intel technical support and getting no official reaction, he posts his discovery on the Internet. It quickly gained a following, and some pointed out that even small errors become big when multiplying by big numbers: the fraction of people with a rare disease times the population of Europe, for example, might lead to the wrong estimate of the number of sick people.
- *November 7, 1994: Electronic Engineering Times* puts the story on its front page, which is soon picked up by other newspapers.
- *November 22, 1994:* Intel issues a press release, calling it a "glitch." The Pentium "can make errors in the ninth digit. . . . Even most engineers and financial analysts require accuracy only to the fourth or fifth decimal point. Spreadsheet and word processor users need not worry. . . . There are maybe several dozen people that this would affect. So far, we've only heard from one. . . . [Only] theoretical mathematicians (with Pentium machines purchased before the summer) should be concerned." What irked many was that customers were told to describe their application to Intel, and then *Intel* would decide whether or not their application merited a new Pentium without the divide bug.
- *December 5, 1994:* Intel claims the flaw happens once in 27,000 years for the typical spreadsheet user. Intel assumes a user does 1000 divides per day and multiplies the error rate assuming floating-point numbers are random, which is one in 9 billion, and then gets 9 million days, or 27,000 years. Things begin to calm down, despite Intel neglecting to explain why a typical customer would access floating-point numbers randomly.
- *December 12, 1994:* IBM Research Division disputes Intel's calculation of the rate of errors (you can access this article by visiting *www.mkp.com/books_catalog/cod/links.htm*). IBM claims that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. IBM assumes 5000 divides per second, 15 minutes, yielding 4.2 million divides per day, and does not assume random distribution of numbers, instead calculating the chances as one in 100 million. As a result, IBM immediately stops shipment of all IBM personal computers based on the Pentium. Things heat up again for Intel.

WEB ENHANCED

- *December 21, 1994:* Intel releases the following, signed by Intel's president, chief executive officer, chief operating officer, and chairman of the board: "We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has