3

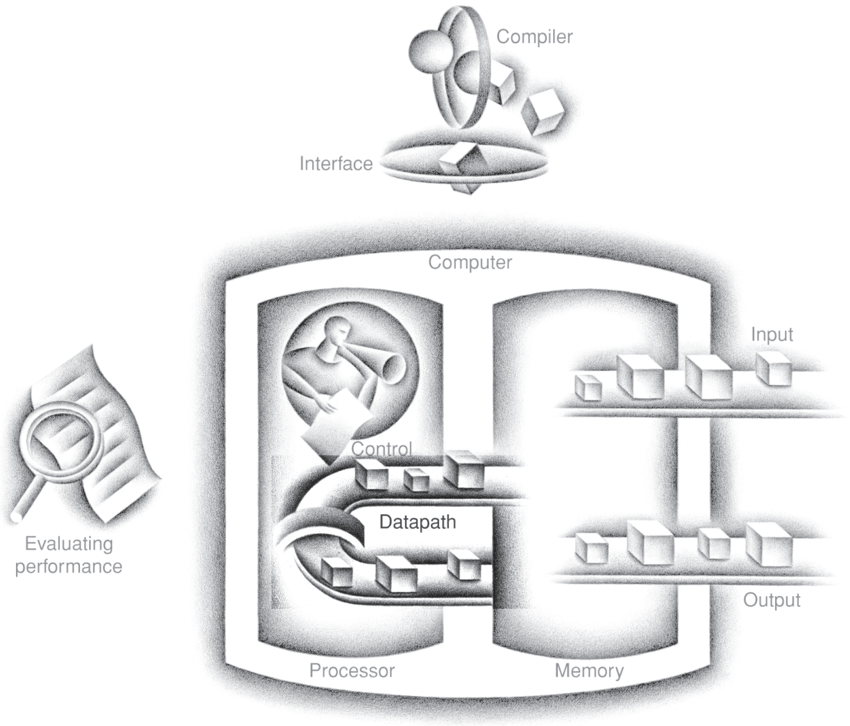
Arithmetic for Computers

Numerical precision is the very soul of science.

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

Computer Organization and Design. DOI: http://dx.doi.org/10.1016/B978-0-12-407726-3.00001-1

The Five Classic Components of a Computer



3.1 Introduction

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

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

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

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., *Book of Top Ten Lists,* 1990

3.2 Addition and Subtraction

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

Binary Addition and Subtraction

EXAMPLE

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

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

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

ANSWER

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

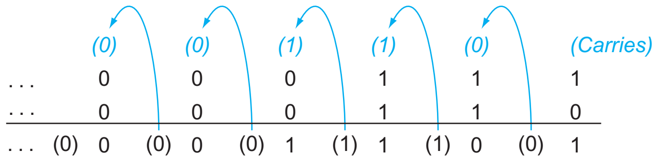


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of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

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

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

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} & = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \end{array}$$

Recall that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, $-10 + 4 = -6$. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

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

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

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

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

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

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu`, no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

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

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

Elaboration: A constant source of confusion for `addiu` is its name and what happens to its immediate field. The u stands for unsigned, which means addition cannot cause an overflow exception. However, the 16-bit immediate field is sign extended to 32 bits, just like `addi`, `slti`, and `sltiu`. Thus, the immediate field is signed, even if the operation is "unsigned."

Hardware/Software Interface

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

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 4.9 covers exceptions in

exception Also called **interrupt** on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

more detail; Chapter 5 describes other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

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

Summary

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

Check Yourself

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

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

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

Elaboration: MIPS can trap on overflow, but unlike many other computers, there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the *Elaboration* on page 89 in Chapter 2 for a description of the xor instruction):

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

For unsigned addition ($t0 = $t1 + $t2), the test is

```
addu $t0, $t1, $t2     # $t0 = sum
nor $t3, $t1, $zero    # $t3 = NOT $t1
                       # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2     # (2^32 - $t1 - 1) < $t2
                       # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

Elaboration: In the preceding text, we said that you copy EPC into a register via mfc0 and then return to the interrupted code via jump register. This directive leads to an interesting question: since you must first transfer EPC to a register to use with jump register, how can jump register return to the interrupted code *and* restore the original values of *all* registers? Either you restore the old registers first, thereby destroying your return address from EPC, which you placed in a register for use in jump register, or you restore all registers but the one with the return address so that you can jump—meaning an exception would result in changing that one register at any time during program execution! Neither option is satisfactory.

To rescue the hardware from this dilemma, MIPS programmers agreed to reserve registers $k0 and $k1 for the operating system; these registers are *not* restored on exceptions. Just as the MIPS compilers avoid using register $at so that the assembler can use it as a temporary register (see *Hardware/Software Interface* in Section 2.10), compilers also abstain from using registers $k0 and $k1 to make them available for the operating system. Exception routines place the return address in one of these registers and then use jump register to restore the instruction address.

Elaboration: The speed of addition is increased by determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry. The most popular is *carry lookahead*, which Section B.6 in **Appendix B** describes.

3.3 Multiplication

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

Anonymous, Elizabethan manuscript, 1570

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

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

```
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 happens to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

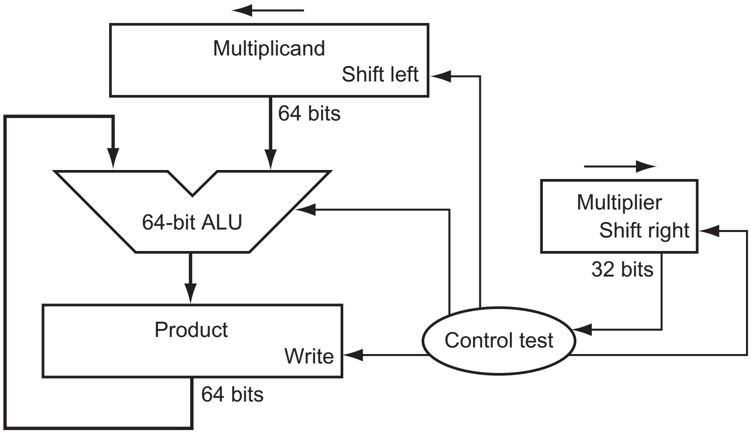


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; Figure 3.3 shows the hardware. We have drawn the hardware so that data flows from top to bottom to resemble more closely the paper-and-pencil method.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 32 steps, a 32-bit multiplicand would move 32 bits to the left. Hence, we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and zero in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

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

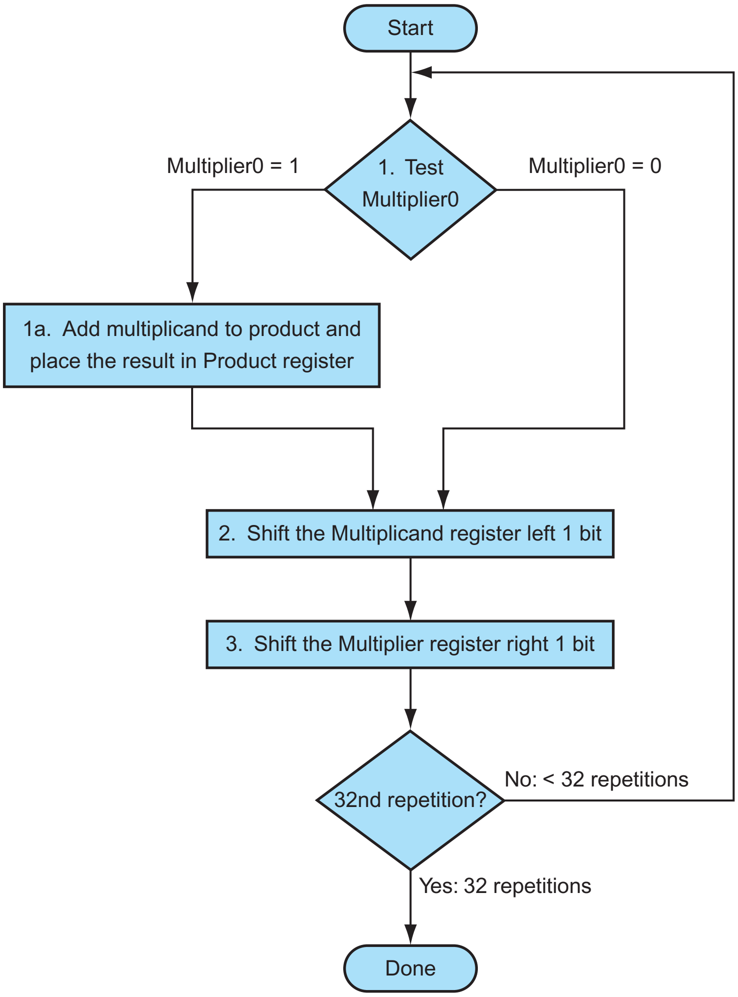

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

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

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

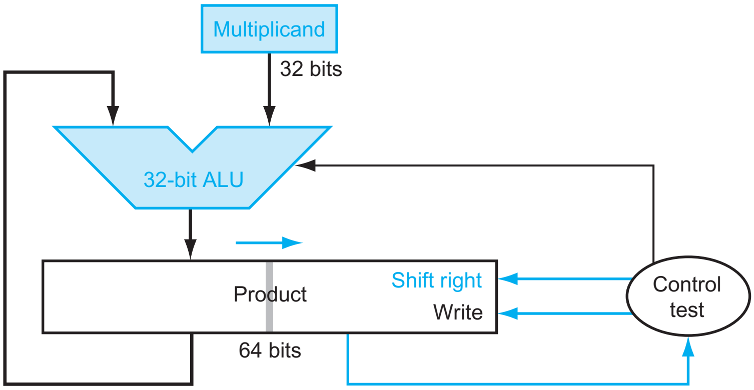


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Hardware/ Software Interface

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

EXAMPLE

A Multiply Algorithm

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

ANSWER

Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, with the final value of $0000\ 0110_{two}$ 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 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Signed Multiplication

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

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

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high. An alternative way to organize these 32 additions is in a parallel tree, as Figure 3.7 shows. Instead of waiting for 32 add times, we wait just the $\log_2$ (32) or five 32-bit add times.

MOORE'S LAW

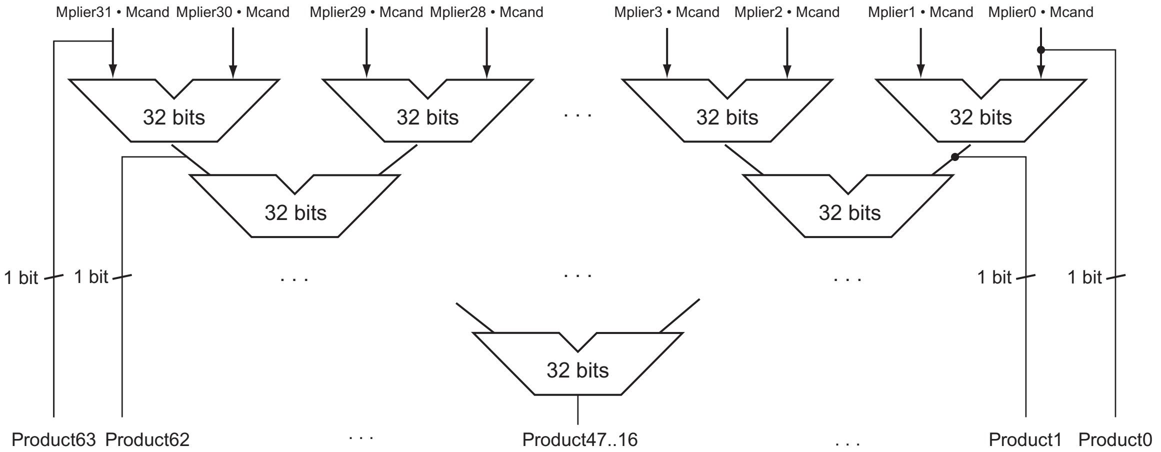


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.

PIPELINING

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

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.

PARALLELISM

Summary

Multiplication hardware simply shifts and add, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

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. There is no overflow if Hi is 0 for `multu` or 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.

3.4 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 $1{,}001{,}010_{\text{ten}}$ by 1000_{ten}:

$$
\begin{array}{rrl}
 & 1001_{\text{ten}} & \text{Quotient} \\
\text{Divisor } 1000_{\text{ten}} & \overline{\big)1001010_{\text{ten}}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{\text{ten}} & \text{Remainder}
\end{array}
$$

Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

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

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

dividend A number being divided.

divisor A number that the dividend is divided by.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

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

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

A Division Algorithm and Hardware

Figure 3.8 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

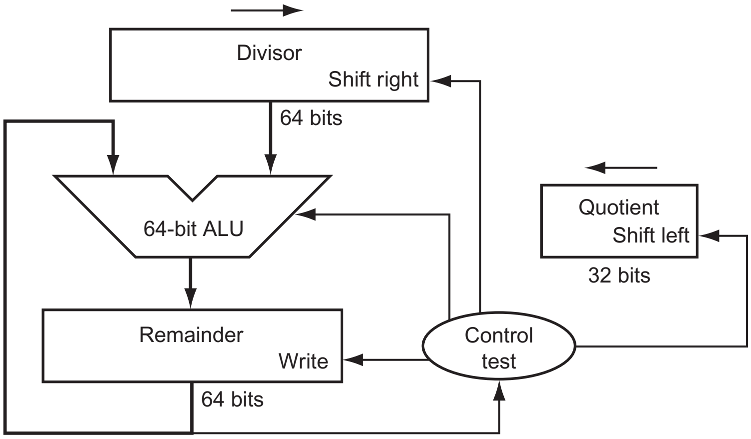


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized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

Figure 3.9 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed 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.

EXAMPLE

A Divide Algorithm

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

ANSWER

Figure 3.10 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply 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.

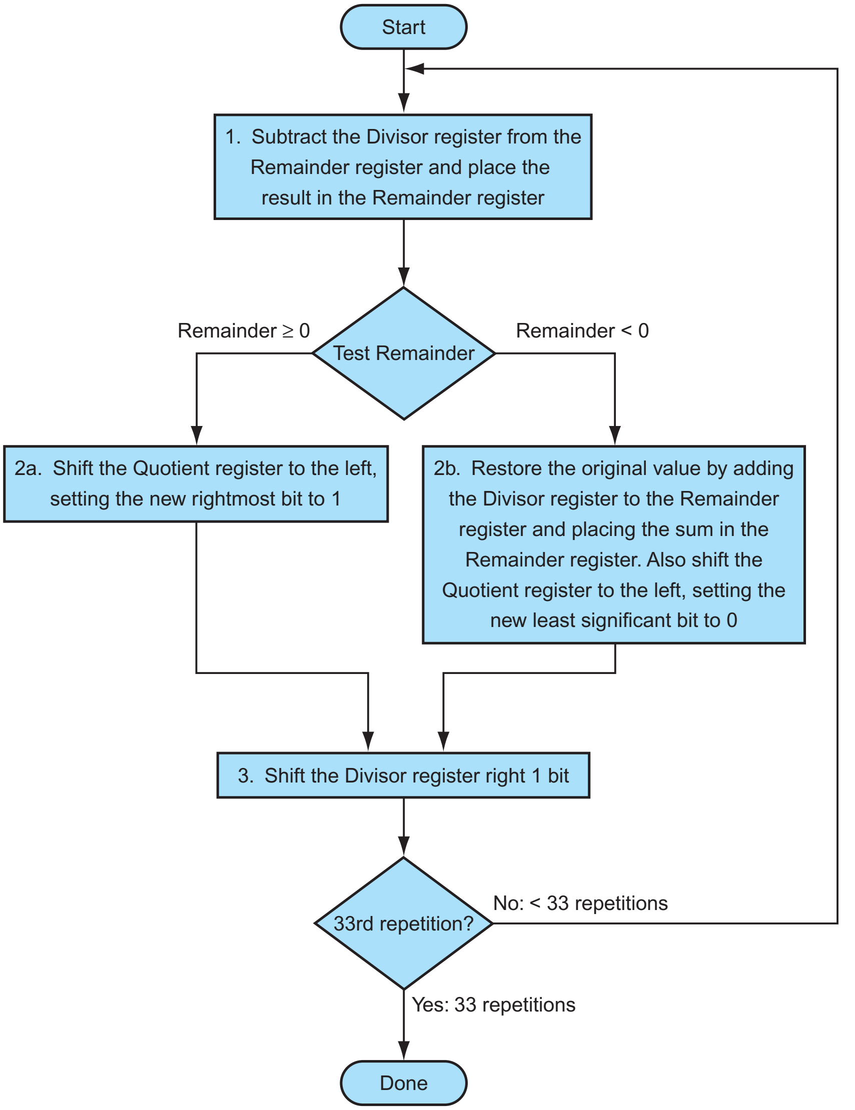

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to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

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

This algorithm and hardware can be refined to be faster and cheaper. The speed-up comes from shifting the operands and the quotient simultaneously with the subtraction. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.11 shows the revised hardware.

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

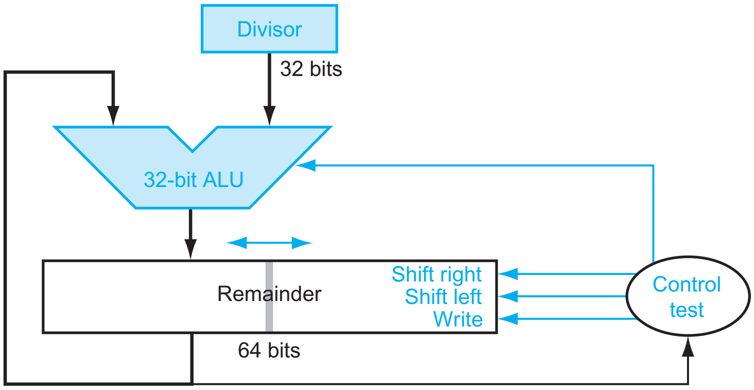


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.5, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

Signed Division

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

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

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

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

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

Checking the results:

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

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:

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

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.

Faster Division

MOORE'S LAW

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

PREDICTION

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

The accuracy of this fast method depends on having proper values in the lookup table. The fallacy on page 231 in Section 3.9 shows what can happen if the table is incorrect.

Divide in MIPS

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

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. We accelerate division by predicting multliple quotient bits and then correcting mispredictions later, Figure 3.12 summarizes the enhancements to the MIPS architecture for the last two sections.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo

	divide	div $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Lo = quotient, Hi = remainder

	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Unsigned quotient and remainder

	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store conditional word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half atomic swap
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	AND	AND $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	OR	OR $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	NOR	NOR $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	AND immediate	ANDi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	OR immediate	ORi $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative

	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1 = 0
				Compare less than; two's complement

	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1=0
				Compare < constant; two's complement

	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1=0
				Compare less than; natural numbers

	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1 = 0
				Compare < constant; natural numbers

Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.12 MIPS core architecture. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the Hi and Lo registers to support multiply and divide. MIPS machine language is listed in the MIPS Reference Data Card at the front of this book.

Hardware/Software Interface

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

Elaboration: An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step, since $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock cycle per step, is explored further in the exercises; the algorithm above is called *restoring* division. A third algorithm that doesn't save the result of the subtract if it's negative is called a *nonperforming* division algorithm. It averages one-third fewer arithmetic operations.

3.5 Floating Point

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

American proverb

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

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

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

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

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

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

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

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

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

Floating-Point Representation

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

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

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

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

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>s</td><td colspan="8">exponent</td><td colspan="23">fraction</td></tr>
<tr><td>1 bit</td><td colspan="8">8 bits</td><td colspan="23">23 bits</td></tr>
</table>

In general, floating-point numbers are of the form

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

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

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

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

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

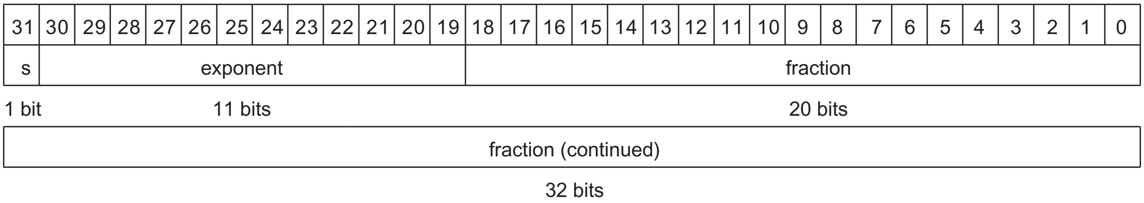
Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event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

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

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

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

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



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

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

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

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

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

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

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

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

Figure 3.13 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

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

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

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

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	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 $00 \ldots 00_{\text{two}}$ and the most positive as $11 \ldots 11_{\text{two}}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

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

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

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

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

to as large as

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

Let's demonstrate.

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

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

It is also represented by the binary fraction

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

In scientific notation, the value is

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

and in normalized scientific notation, it is

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

The general representation for a single precision number is

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

Subtracting the bias 127 from the exponent of $-1.1_{\text{two}} \times 2^{-1}$ yields

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

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

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

1 bit 8 bits 23 bits

The double precision representation is

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

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

1 bit 11 bits 20 bits

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

32 bits

Now let's try going the other direction.

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

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

ANSWER

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

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

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

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

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

Floating-Point Addition

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

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

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

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

$$0.016 \times 10^1$$

Step 2. Next comes the addition of the significands:

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

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

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

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

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

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

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

is rounded to four digits in the significand to

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

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

Figure 3.14 shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the *Elaboration* on page 222). For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is −126.

EXAMPLE

Binary Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.14.

ANSWER

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

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

Now we follow the algorithm:

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

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

Step 2. Add the significands:

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

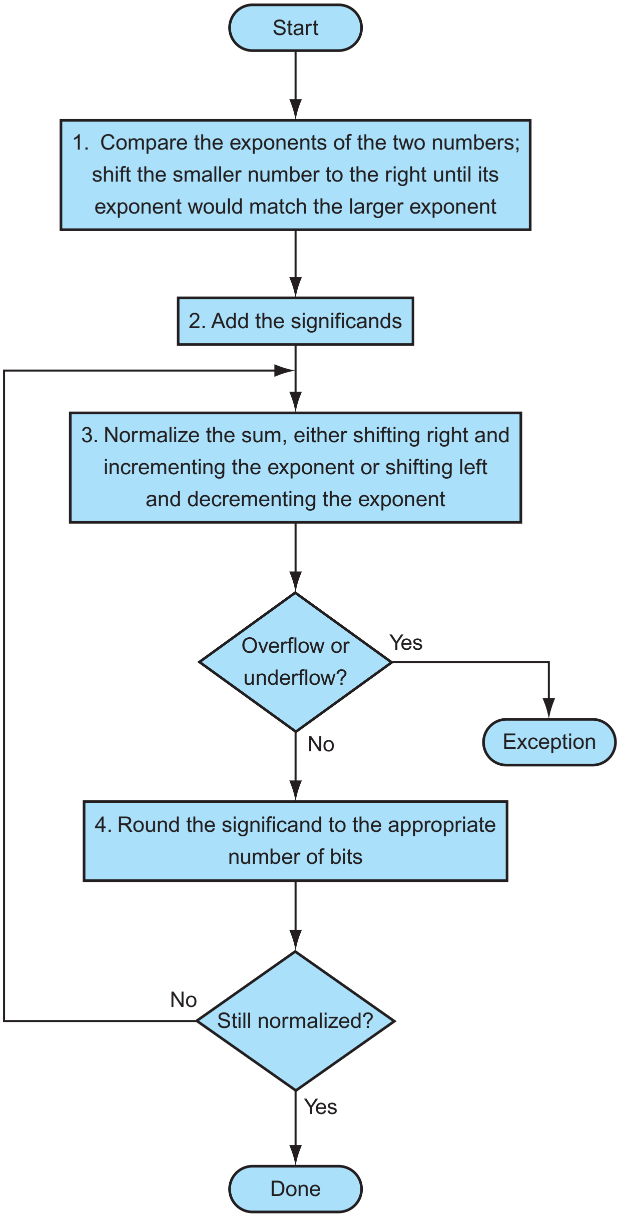

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?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

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

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

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

Since $127 \geq +4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

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

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

This sum is then

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

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure 3.15 sketches the basic organization of hardware for floating-point addition.

Floating-Point Multiplication

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

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

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

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

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

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

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

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

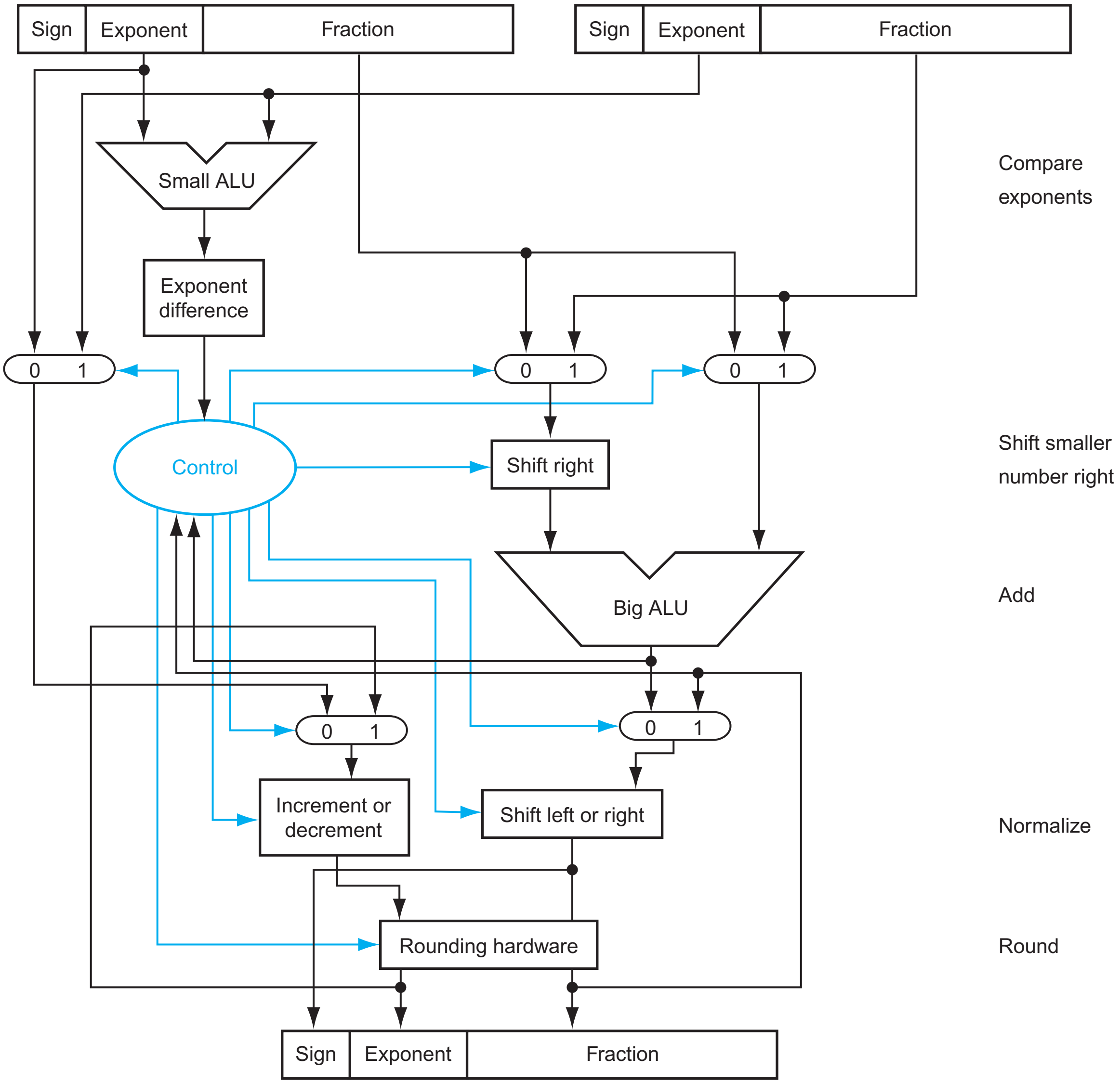


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference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

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

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

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

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

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

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

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

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

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

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

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

is rounded to four digits in the significand to

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

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

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

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

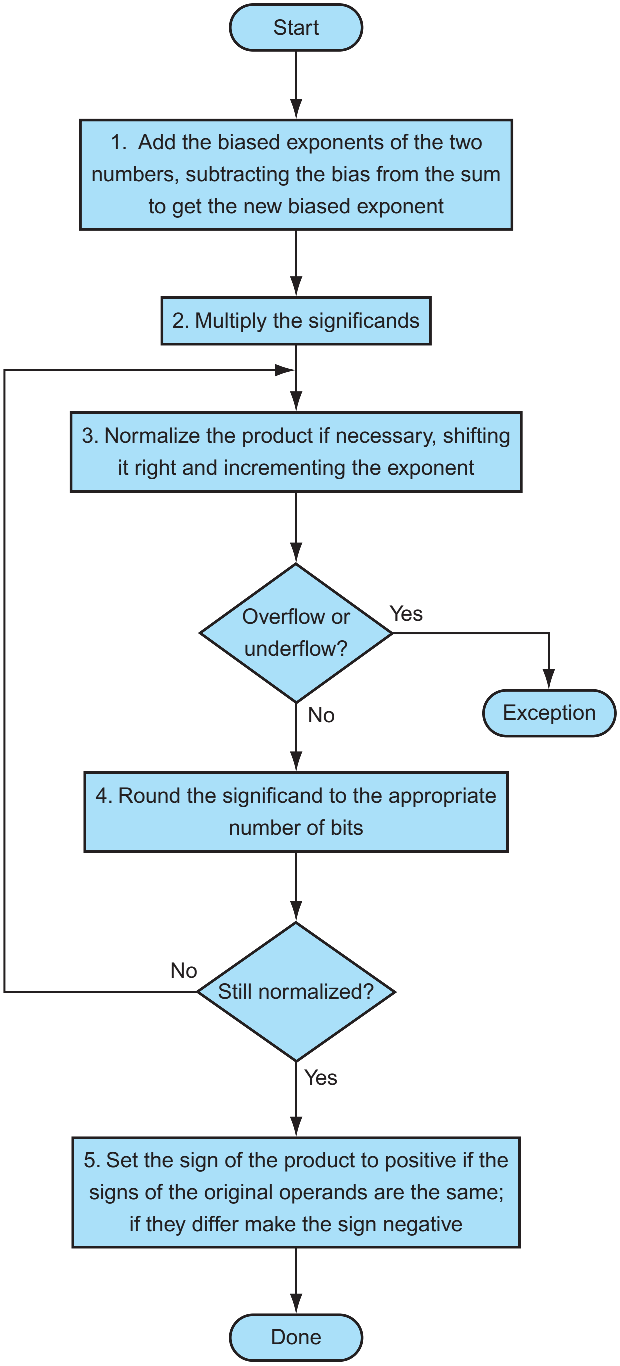

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to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

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

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

EXAMPLE

Binary Floating-Point Multiplication

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

ANSWER

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

Step 1. Adding the exponents without bias:

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

or, using the biased representation:

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

Step 2. Multiplying the significands:

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

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

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

Step 4. Rounding the product makes no change:

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

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

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

Converting to decimal to check our results:

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

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

Floating-Point Instructions in 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 which are used for addresses 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     $f4,c($sp)  # Load 32-bit F.P. number into F4
lwc1     $f6,a($sp)  # Load 32-bit F.P. number into F6
add.s    $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1     $f2,b($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. Thus, the pair of single precision registers `$f2` and `$f3` also form the double precision register named `$f2`.

Figure 3.17 summarizes the floating-point portion of the MIPS architecture revealed in this chapter, with the additions to support floating point shown in color. Similar to Figure 2.19 in Chapter 2, Figure 3.18 shows the encoding of these instructions.

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 word addresses 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 3.17 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, for more detail. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

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 3.18 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`). Instructions in color are described in Chapter 2 or this chapter, with Appendix A covering all instructions. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

Hardware/Software Interface

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

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

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 C = C + A * B. It is commonly called DGEMM, for Double precision, General Matrix Multiply. We'll see versions of DGEMM again in Section 3.8 and subsequently in Chapters 4, 5, and 6. Let's assume C, A, and B are all square matrices with 32 elements in each dimension.

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

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

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

Now we add the second index to select the jth 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 c, giving the address of c[i][j], and then load the double precision number c[i][j] into $f4:

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

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

```
 L3: sll $t0, $s2, 5     # $t0 = k * 2^5 (size of row of b)
     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 b[k][j]
     l.d $f16, 0($t0)   # $f16 = 8 bytes of b[k][j]
```

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

```
sll    $t0, $s0, 5    # $t0 = i * 2^5 (size of row of a)
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 a[i][k]
l.d    $f18, 0($t0)   # $f18 = 8 bytes of a[i][k]
```

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

```
mul.d $f16, $f18, $f16 # $f16 = a[i][k] * b[k][j]
add.d $f4, $f4, $f16   # f4 = c[i][j] + a[i][k] * b[k][j]
```

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

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

Figure 3.22 below shows the x86 assembly language code for a slightly different version of DGEMM in Figure 3.21.

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

Elaboration: Only 16 of the 32 MIPS floating-point registers could originally be used for double precision operations: $f0, $f2, $f4, …, $f30. Double precision is computed using pairs of these single precision registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added l.d and s.d to the instruction set. MIPS-32 also added "paired single" versions of all floating-point instructions, where a single instruction results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers (see Section 3.6). For example, add.ps $f0, $f2, $f4 is equivalent to add.s $f0, $f2, $f4 followed by add.s $f1, $f3, $f5.

Elaboration: Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence, the floating-point unit, including the floating-point registers, was optionally available as a second chip. Such optional accelerator chips are called *coprocessors,* 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 5.) 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: As mentioned in Section 3.4, accelerating division is more challenging than multiplication. In addition to SRT, another technique to leverage a fast multiplier is *Newton's iteration*, where division is recast as finding the zero of a function to find the reciprocal $1/c$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solved this problem by calculating an extra-precise reciprocal.

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

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

Accurate Arithmetic

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

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point. The best we can do is getting the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

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

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

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

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

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

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

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

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

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

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

IEEE 754 has four rounding modes: always round up (toward +∞), always round down (toward −∞), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The U.S. *Internal Revenue Service* (IRS) always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one;

if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

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

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

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

Elaboration: PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures provide a single instruction that does a multiply and add on three registers: $a = a + (b \times c)$. Obviously, this instruction allows potentially higher floating-point performance for this common operation. Equally important is that instead of performing two roundings—after the multiply and then after the add—which would happen with separate instructions, the multiply add instruction can perform a single rounding after the add. A single rounding step increases the precision of multiply add. Such operations with a single rounding are called **fused multiply add**. It was added to the IEEE 754-2008 standard (see 🌐 **Section 3.11**).

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

Summary

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

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

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

The BIG Picture

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

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

C type	Java type	Data transfers	Operations
`int`	`int`	`lw, sw, lui`	`addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti`
`unsigned int`	—	`lw, sw, lui`	`addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu`
`char`	—	`lb, sb, lui`	`add, addi, sub, mult, div AND, ANDi, OR, ORi, NOR, slt, slti`
—	`char`	`lh, sh, lui`	`addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu`
`float`	`float`	`lwc1, swc1`	`add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s`
`double`	`double`	`l.d, s.d`	`add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d`

Hardware/Software Interface

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

Check Yourself

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

1. $1.0000\ 00 \times 2^{0}$ to $1.1111\ 1111\ 11 \times 2^{31}$, 0
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}$, ± 0, $\pm\infty$, NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}$, ± 0, $\pm\infty$, NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \times 2^{14}$, ± 0, $\pm\infty$, NaN

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

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

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

but the smallest single precision denormalized number is

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

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

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

3.6 Parallelism and Computer Arithmetic: Subword Parallelism

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

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

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

PARALLELISM

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used either with ARMv7 or ARMv8. It added 256 bytes of new registers for NEON that can be viewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supports all the subword data types you can imagine *except* 64-bit floating point numbers:

- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
- 32-bit floating point numbers

Figure 3.19 gives a summary of the basic NEON instructions.

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3.4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3.4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

Elaboration: In addition to signed and unsigned integers, ARM includes "fixed-point" format of four sizes called I8, I16, I32, and I64, of which 16, 8, 4, and 2 fit in a 128-bit register, respectively. A portion of the fixed point is for the fraction (to the right of the binary point) and the rest of the data is the integer portion (to the left of the binary point). The location of the binary point is up to the software. Many ARM processors do not have floating point hardware and thus floating point operations must be performed by library routines. Fixed point arithmetic can be significantly faster than software floating point routines, but more work for the programmer.

3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) and SSE (*Streaming SIMD Extension*) instructions for the x86 included similar operations to those found in ARM NEON. Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.20 summarizes the SSE and SSE2 instructions.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles (PS) or two doubles (PD).

Data transfer	Arithmetic	Compare
`MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm`	`ADD{SS/PS/SD/PD} xmm,mem/xmm`	`CMP{SS/PS/SD/PD}`
	`SUB{SS/PS/SD/PD} xmm,mem/xmm`	
`MOV {H/L} {PS/PD} xmm, mem/xmm`	`MUL{SS/PS/SD/PD} xmm,mem/xmm`	
	`DIV{SS/PS/SD/PD} xmm,mem/xmm`	
	`SQRT{SS/PS/SD/PD} mem/xmm`	
	`MAX {SS/PS/SD/PD} mem/xmm`	
	`MIN{SS/PS/SD/PD} mem/xmm`	

FIGURE 3.20 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

In 2011 Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions (AVX)*. Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter "v" (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point multiplies

```
addpd  %xmm0, %xmm4
```

It becomes

```
vaddpd  %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies.

Elaboration: AVX also added three address instructions to x86. For example, `vaddpd` can now specify

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

instead of the standard two address version

```
addpd  %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

(Unlike MIPS, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

3.8 Going Faster: Subword Parallelism and Matrix Multiply

To demonstrate the performance impact of subword parallelism, we'll run the same code on the Intel Core i7 first without AVX and then with it. Figure 3.21 shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in Section 3.5, this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled "Going Faster" to demonstrate the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in Chapters 3, 4, 5, and 6 will incrementally improve DGEMM performance using the ideas that each chapter introduces.

Figure 3.22 shows the x86 assembly language output for the inner loop of Figure 3.21. The five floating point-instructions start with a `v` like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include `sd` in the name, which stands for scalar double precision. We'll define the subword parallel instructions shortly.

```
void dgemm (int n, double* A, double* B, double* C)
{
   for (int i = 0; i < n; ++i)
     for (int j = 0; j < n; ++j)
     {
       double cij = C[i+j*n]; /* cij = C[i][j] */
       for( int k = 0; k < n; k++ )
         cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
       C[i+j*n] = cij; /* C[i][j] = cij */
     }
}
```

FIGURE 3.21 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply (GEMM). Because we are passing the matrix dimension as the parameter n, this version of DGEMM uses single dimensional versions of matrices C, A, and B and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

```
vmovsd (%r10),%xmm0           # Load 1 element of C into %xmm0
mov    %rsi,%rcx              # register %rcx = %rsi
xor    %eax,%eax              # register %eax = 0
vmovsd (%rcx),%xmm1           # Load 1 element of B into %xmm1
add    %r9,%rcx               # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add    $0x1,%rax              # register %rax = %rax + 1
cmp    %eax,%edi              # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0      # Add %xmm1, %xmm0
jg     30 <dgemm+0x30>        # jump if %eax > %edi
add    $0x1,%r11d             # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)           # Store %xmm0 into C element
```

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the Elaboration in Section 3.7).

```
 1.  #include <x86intrin.h>
 2.  void dgemm (int n, double* A, double* B, double* C)
 3.  {
 4.    for ( int i = 0; i < n; i+=4 )
 5.      for ( int j = 0; j < n; j++ ) {
 6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
 7.        for( int k = 0; k < n; k++ )
 8.          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
 9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.               _mm256_broadcast_sd(B+k+j*n)));
11.        _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.      }
13.  }
```

FIGURE 3.23 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86. Figure 3.24 shows the assembly language produced by the compiler for the inner loop.

While compiler writers may eventually be able to routinely produce high-quality code that uses the AVX instructions of the x86, for now we must "cheat" by using C intrinsics that more or less tell the compiler exactly how to produce good code. Figure 3.23 shows the enhanced version of Figure 3.21 for which the Gnu C compiler produces AVX code. Figure 3.24 shows annotated x86 code that is the output of compiling using gcc with the –O3 level of optimization.

The declaration on line 6 of Figure 3.23 uses the __m256d data type, which tells the compiler the variable will hold 4 double-precision floating-point values. The intrinsic _mm256_load_pd() also on line 6 uses AVX instructions to load 4 double-precision floating-point numbers in parallel (_pd) from the matrix C into c0. The address calculation C+i+j*n on line 6 represents element C[i+j*n]. Symmetrically, the final step on line 11 uses the intrinsic _mm256_store_pd() to store 4 double-precision floating-point numbers from c0 into the matrix C. As we're going through 4 elements each iteration, the outer *for* loop on line 4 increments i by 4 instead of by 1 as on line 3 of Figure 3.21.

Inside the loops, on line 9 we first load 4 elements of A again using _mm256_load_pd(). To multiply these elements by one element of B, on line 10 we first use the intrinsic _mm256_broadcast_sd(), which makes 4 identical copies of the scalar double precision number—in this case an element of B—in one of the YMM registers. We then use _mm256_mul_pd() on line 9 to multiply the four double-precision results in parallel. Finally, _mm256_add_pd() on line 8 adds the 4 products to the 4 sums in c0.

Figure 3.24 shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with v and

```
 1. vmovapd (%r11),%ymm0              # Load 4 elements of C into %ymm0
 2. mov     %rbx,%rcx                 # register %rcx = %rbx
 3. xor     %eax,%eax                 # register %eax = 0
 4. vbroadcastsd (%rax,%r8,1),%ymm1   # Make 4 copies of B element
 5. add     $0x8,%rax                 # register %rax = %rax + 8
 6. vmulpd (%rcx),%ymm1,%ymm1         # Parallel mul %ymm1,4 A elements
 7. add     %r9,%rcx                  # register %rcx = %rcx + %r9
 8. cmp     %r10,%rax                 # compare %r10 to %rax
 9. vaddpd %ymm1,%ymm0,%ymm0          # Parallel add %ymm1, %ymm0
10. jne     50 <dgemm+0x50>           # jump if not %r10 != %rax
11. add     $0x1,%esi                 # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)              # Store %ymm0 into 4 C elements
```

FIGURE 3.24 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.23. Note the similarities to Figure 3.22, with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

four of the five use pd for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in Figure 3.22 above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (sd) using XMM registers to *parallel double* (pd) with YMM registers. The one exception is line 4 of Figure 3.24. Every element of A must be multiplied by one element of B. One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register, which is just what the instruction vbroadcastsd does.

PARALLELISM

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in Figure 3.21 runs at 1.7 GigaFLOPS (FLoating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in Figure 3.23 performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of 4.0 increase that you might hope for from performing 4 times as many operations at a time by using **subword parallelism**.

Elaboration: As mentioned in the Elaboration in Section 1.6, Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of 3.3/2.6 = 1.27 to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.

3.9 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: 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 *c*, where *xi* means the *i*th bit, represents the number

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

Shifting the bits of *c* 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 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.

Pitfall: Floating-point addition is not associative.

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. 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, let's see if $c + (a + b) = (c + a) + b$. Assume $c = -1.5_{ten} \times 10^{38}$, $a = 1.5_{ten} \times 10^{38}$, and $b = 1.0$, and that these are all single precision numbers.

$$\begin{aligned}
c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\
&= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\
&= 0.0 \\
c + (a + b) &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\
&= (0.0_{\text{ten}}) + 1.0 \\
&= 1.0
\end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{\text{ten}} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{\text{ten}} \times 10^{38} + 1.0$ is still $1.5_{\text{ten}} \times 10^{38}$. That is why the sum of *c, a,* and *b* is 0.0 or 1.0, depending on the order of the floating-point additions, so $c + (a + b) \neq (c + a) + b$. Therefore, floating-point addition is *not* associative.

Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, "Do the two versions get the same answer?" If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you were to add a million numbers together, you would get the same results whether you used 1 processor or 1000 processors. This assumption holds for two's complement integers, since integer addition is associative. Alas, since floating-point addition is not associative, the assumption does not hold.

A more vexing version of this fallacy occurs on a parallel computer where the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer. As the varying number of processors from each run would cause the floating-point sums to be calculated in different orders, getting slightly different answers each time despite running identical code with identical input may flummox unaware parallel programmers.

Given this quandary, programmers who write parallel code with floating-point numbers need to verify whether the results are credible even if they don't give the same exact answer as the sequential code. The field that deals with such issues is called numerical analysis, which is the subject of textbooks in its own right. Such concerns are one reason for the popularity of numerical libraries such as LAPACK and SCALAPAK, which have been validated in both their sequential and parallel forms.

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

Despite its name, add immediate unsigned (`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 3.25). The following is the inside story behind the headlines.

The Pentium used a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing −2, −1, 0, +1, or +2. The guess is multiplied by the divisor and subtracted from the remainder to generate a new remainder. Like nonrestoring division, if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently, there were five elements of the table from the 80486 that Intel engineers thought could never be accessed, and they optimized the logic to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11

A Closer Look
Warning: Intel Inside
BUSINESS
San Jose Mercury News
Intel to work on software patch for Pentium bug
Quick fix:
Grove says he's sorry about Pentium bug
Flawed Chip Bruises Intel
Investors react, stock plunges
INTEL'S MISTAKE
INFO WORLD
The Voice of Personal Computing in the Enterprise
Bug Dodge Booed
Intel Knocked For Response On Flaw
Flawed
PERSONAL TECHNOLOGY
Chip Shot
Computer Giants' War Over Flaw in Pentium Jolts the PC Industry
Who Is Twisting the Truth? Intel Stands by Product As IBM Halts Shipments
I'd Be Totally Confused
Some Scientists Are Angry Over Flaw In Pentium Chip, and Intel's Response
San Francisco Chronicle
BUSINESS
Intel's Pentium Problem Persists
TECHNOLOGY
Intel's Grove Airs Apology for Pentium Over the Internet
But CEO, Defending Policy Not to Replace All Chips, Doesn't Silence Criticism
Pentium woes continue
Faulty FPU flubs math in certain equations
Multithreading gets lost on P100 systems
IBM to stop shipping Pentium PCs

FIGURE 3.25 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.

bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

A math professor at Lynchburg College in Virginia, Thomas Nicely, discovered the bug in September 1994. After calling Intel technical support and getting no official reaction, he posted his discovery on the Internet. This post led to a story in a trade magazine, which in turn caused Intel to issue a press release. It called the bug a glitch that would affect only theoretical mathematicians, with the average spreadsheet user seeing an error every 27,000 years. IBM Research soon counterclaimed that the average spreadsheet user would see an error every 24 days. Intel soon threw in the towel by making the following announcement on December 21:

> *"We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer."*

Analysts estimate that this recall cost Intel $500 million, and Intel engineers did not get a Christmas bonus that year.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

3.10 Concluding Remarks

Over the decades, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic is found in every computer sold today, and if it includes floating point support, it offers the IEEE 754 binary floating-point arithmetic.

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapters 4 and 5 discuss exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation of floating point are part of the inspiration for the field of numerical analysis. The recent switch to **parallelism** shines the searchlight on numerical analysis again, as solutions that were long considered safe on sequential computers must be reconsidered when trying to find the fastest algorithm for parallel computers that still achieves a correct result.

PARALLELISM

Data-level parallelism, specifically subword parallelism, offers a simple path to higher performance for programs that are intensive in arithmetic operations for either integer or floating-point data. We showed that we could speed up matrix multiply nearly fourfold by using instructions that could execute four floating-point operations at a time.

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. Two figures try to make this clear.

Figure 3.26 lists the MIPS instructions covered in this chapter and Chapter 2. We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of Figure 3.27 are the instructions the MIPS processor executes that are not found in Figure 3.26. We call the full set of hardware instructions *MIPS-32*. On the right of Figure 3.27 are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

Figure 3.28 gives the popularity of the MIPS instructions for SPEC CPU2006 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 0.2% of the instructions executed.

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC CPU2006 execution, and the integer core plus arithmetic core dominate SPEC CPU2006 floating point, as the table below shows.

Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As you can see, the MIPS core includes the most popular MIPS instructions; be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers. No matter what the instruction set or its size—MIPS, ARM, x86—never forget that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, string, instruction, and so on. In stored program computers, it is the operation on the bit pattern that determines its meaning.

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 3.26 The MIPS instruction set. This book concentrates on the instructions in the left column. This information is also found in columns 1 and 2 of the MIPS Reference Data Card at the front of this book.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate *(signed or unsigned)*	neg*s*	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw *(signed or uns.)*	mul*s*	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left *(unaligned)*	lwl	I	load address	la	rd,addr
load word right *(unaligned)*	lwr	I	load double	ld	rd,addr
store word left *(unaligned)*	swl	I	store double	sd	rd,addr
store word right *(unaligned)*	swr	I	unaligned load word	ulw	rd,addr
load linked (*atomic update*)	ll	I	unaligned store word	usw	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or *uns.)*	madd*s*	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or *uns.)*	msub*s*	I	branch on compare *(signed or unsigned)*	bx*s*	rs,rt,L
branch on ≥ zero and link	bgezal	I	(x = lt, le, gt, ge)		
branch on < zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare *(signed or unsigned)*	sx*s*	rd,rs,rt
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			load to floating point *(s or d)*	l.*f*	rd,addr
branch compare reg likely	bxl	I	store from floating point *(s or d)*	s.*f*	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
(x = eq, neq, lt, le, gt, ge)					
return from exception	rfe	R			
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move *(s or d)*	mov.*f*	R			
FP move if zero *(s or d)*	movz.*f*	R			
FP move if not zero *(s or d)*	movn.*f*	R			
FP square root *(s or d)*	sqrt.*f*	R			
FP absolute value *(s or d)*	abs.*f*	R			
FP negate *(s or d)*	neg.*f*	R			
FP convert *(w, s, or d)*	cvt.*f*.*f*	R			
FP compare un *(s or d)*	c.xn.*f*	R			

FIGURE 3.27 Remaining MIPS-32 and Pseudo MIPS instruction sets. *f* means single (s) or double (d) precision floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.f/ msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*). The underscore represents the letter to include to represent that datatype.

Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	`add`	0.0%	0.0%	FP add double	`add.d`	0.0%	10.6%
add immediate	`addi`	0.0%	0.0%	FP subtract double	`sub.d`	0.0%	4.9%
add unsigned	`addu`	5.2%	3.5%	FP multiply double	`mul.d`	0.0%	15.0%
add immediate unsigned	`addiu`	9.0%	7.2%	FP divide double	`div.d`	0.0%	0.2%
subtract unsigned	`subu`	2.2%	0.6%	FP add single	`add.s`	0.0%	1.5%
AND	`AND`	0.2%	0.1%	FP subtract single	`sub.s`	0.0%	1.8%
AND immediate	`ANDi`	0.7%	0.2%	FP multiply single	`mul.s`	0.0%	2.4%
OR	`OR`	4.0%	1.2%	FP divide single	`div.s`	0.0%	0.2%
OR immediate	`ORi`	1.0%	0.2%	load word to FP double	`l.d`	0.0%	17.5%
NOR	`NOR`	0.4%	0.2%	store word to FP double	`s.d`	0.0%	4.9%
shift left logical	`sll`	4.4%	1.9%	load word to FP single	`l.s`	0.0%	4.2%
shift right logical	`srl`	1.1%	0.5%	store word to FP single	`s.s`	0.0%	1.1%
load upper immediate	`lui`	3.3%	0.5%	branch on floating-point true	`bc1t`	0.0%	0.2%
load word	`lw`	18.6%	5.8%	branch on floating-point false	`bc1f`	0.0%	0.2%
store word	`sw`	7.6%	2.0%	floating-point compare double	`c.x.d`	0.0%	0.6%
load byte	`lbu`	3.7%	0.1%	multiply	`mul`	0.0%	0.2%
store byte	`sb`	0.6%	0.0%	shift right arithmetic	`sra`	0.5%	0.3%
branch on equal (zero)	`beq`	8.6%	2.2%	load half	`lhu`	1.3%	0.0%
branch on not equal (zero)	`bne`	8.4%	1.4%	store half	`sh`	0.1%	0.0%
jump and link	`jal`	0.7%	0.2%				
jump register	`jr`	1.1%	0.2%				
set less than	`slt`	9.9%	2.3%				
set less than immediate	`slti`	3.1%	0.3%				
set less than unsigned	`sltu`	3.4%	0.8%				
set less than imm. uns.	`sltiu`	1.1%	0.1%				

FIGURE 3.28 The frequency of the MIPS instructions for SPEC CPU2006 integer and floating point. All instructions that accounted for at least 0.2% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here.

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, *1992*

3.11 Historical Perspective and Further Reading

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the x86. See the rest of Section 3.11 online.

3.12 Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941

3.1 [5] <§3.2> What is 5ED4 − 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

3.2 [5] <§3.2> What is 5ED4 − 07A4 when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

3.3 [10] <§3.2> Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

3.4 [5] <§3.2> What is 4365 − 3412 when these values represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

3.5 [5] <§3.2> What is 4365 − 3412 when these values represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

3.6 [5] <§3.2> Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate 185 – 122. Is there overflow, underflow, or neither?

3.7 [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 + 122. Is there overflow, underflow, or neither?

3.8 [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 − 122. Is there overflow, underflow, or neither?

3.9 [10] <§3.2> Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151 + 214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.10 [10] <§3.2> Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151 − 214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.11 [10] <§3.2> Assume 151 and 214 are unsigned 8-bit integers. Calculate 151 + 214 using saturating arithmetic. The result should be written in decimal. Show your work.

3.12 [20] <§3.3> Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

3.13 [20] <§3.3> Using a table similar to that shown in Figure 3.6, calculate the product of the hexadecimal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.

3.14 [10] <§3.3> Calculate the time necessary to perform a multiply using the approach given in Figures 3.3 and 3.4 if an integer is 8 bits wide and each step of the operation takes 4 time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

3.15 [10] <§3.3> Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes 4 time units.

3.16 [20] <§3.3> Calculate the time necessary to perform a multiply using the approach given in Figure 3.7 if an integer is 8 bits wide and an adder takes 4 time units.

3.17 [20] <§3.3> As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9 × 6, for example, can be written (2 × 2 × 2 + 1) × 6, we can calculate 9 × 6 by shifting 6 to the left 3 times and then adding 6 to that result. Show the best way to calculate 0×33 × 0×55 using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

3.18 [20] <§3.4> Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.8. You should show the contents of each register on each step. Assume both inputs are unsigned 6-bit integers.

3.19 [30] <§3.4> Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.11. You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers. This algorithm requires a slightly different approach than that shown in Figure 3.9. You will want to think hard about this, do an experiment or two, or else go to the web to figure out how to make this work correctly. (Hint: one possible solution involves using the fact that Figure 3.11 implies the remainder register can be shifted either direction.)

3.20 [5] <§3.5> What decimal number does the bit pattern 0×0C000000 represent if it is a two's complement integer? An unsigned integer?

3.21 [10] <§3.5> If the bit pattern 0×0C000000 is placed into the Instruction Register, what MIPS instruction will be executed?

3.22 [10] <§3.5> What decimal number does the bit pattern 0×0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

3.23 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

3.24 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

3.25 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

3.26 [20] <§3.5> Write down the binary bit pattern to represent -1.5625×10^{-1} assuming a format similar to that employed by the DEC PDP-8 (the leftmost 12 bits are the exponent stored as a two's complement number, and the rightmost 24 bits are the fraction stored as a two's complement number). No hidden 1 is used. Comment on how the range and accuracy of this 36-bit pattern compares to the single and double precision IEEE 754 standards.

3.27 [20] <§3.5> IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

3.28 [20] <§3.5> The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in two's complement format, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the rightmost 8 bits representing the exponent. However, in an interesting twist, the exponent was stored in sign-magnitude format with the sign bit on the far right! Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. No hidden 1 is used. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

3.29 [20] <§3.5> Calculate the sum of 2.6125×10^{1} and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

3.30 [30] <§3.5> Calculate the product of -8.0546875×10^{0} and $-1.79931640625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision format described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps; however, as is done in the example in the text, you can do the multiplication in human-readable format instead of using the techniques described in Exercises 3.12 through 3.14. Indicate if there is overflow or underflow. Write your answer in both the 16-bit floating point format described in Exercise 3.27 and also as a decimal number. How accurate is your result? How does it compare to the number you get if you do the multiplication on a calculator?

3.31 [30] <§3.5> Calculate by hand 8.625×10^1 divided by -4.875×10^0. Show all the steps necessary to achieve your answer. Assume there is a guard, a round bit, and a sticky bit, and use them if necessary. Write the final answer in both the 16-bit floating point format described in Exercise 3.27 and in decimal and compare the decimal result to that which you get if you use a calculator.

3.32 [20] <§3.9> Calculate $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.33 [20] <§3.9> Calculate $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.34 [10] <§3.9> Based on your answers to 3.32 and 3.33, does $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3 = 3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$?

3.35 [30] <§3.9> Calculate $(3.41796875\ 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.36 [30] <§3.9> Calculate $3.41796875\ 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^2)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.37 [10] <§3.9> Based on your answers to 3.35 and 3.36, does $(3.41796875\ 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2 = 3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^2)$?

3.38 [30] <§3.9> Calculate $1.666015625 \times 10^0 \times (1.9760 \times 10^4 + -1.9744 \times 10^4)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.39 [30] <§3.9> Calculate $(1.666015625 \times 10^0 \times 1.9760 \times 10^4) + (1.666015625 \times 10^0 \times -1.9744 \times 10^4)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.40 [10] <§3.9> Based on your answers to 3.38 and 3.39, does (1.666015625 × 10^0 × 1.9760 × 10^4) + (1.666015625 × 10^0 × −1.9744 × 10^4) = 1.666015625 × 10^0 × (1.9760 × 10^4 + −1.9744 × 10^4)?

3.41 [10] <§3.5> Using the IEEE 754 floating point format, write down the bit pattern that would represent −1/4. Can you represent −1/4 exactly?

3.42 [10] <§3.5> What do you get if you add −1/4 to itself 4 times? What is −1/4 × 4? Are they the same? What should they be?

3.43 [10] <§3.5> Write down the bit pattern in the fraction of value 1/3 assuming a floating point format that uses binary numbers in the fraction. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.44 [10] <§3.5> Write down the bit pattern in the fraction assuming a floating point format that uses Binary Coded Decimal (base 10) numbers in the fraction instead of base 2. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.45 [10] <§3.5> Write down the bit pattern assuming that we are using base 15 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 15 numbers would use 0–9 and A–E.) Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.46 [20] <§3.5> Write down the bit pattern assuming that we are using base 30 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 30 numbers would use 0–9 and A–T.) Assume there are 20 bits, and you do not need to normalize. Is this representation exact?

3.47 [45] <§§3.6, 3.7> The following C code implements a four-tap FIR filter on input array sig_in. Assume that all arrays are 16-bit fixed-point values.

```
for (i=3;i<128;i++)
sig_out[i]=sig_in[i-3] * f[0]+sig_in[i-2] * f[1]
  +sig_in[i-1] * f[2]+sig_in[i] * f[3];
```

Assume you are to write an optimized implementation this code in assembly language on a processor that has SIMD instructions and 128-bit registers. Without knowing the details of the instruction set, briefly describe how you would implement this code, maximizing the use of sub-word operations and minimizing the amount of data that is transferred between registers and memory. State all your assumptions about the instructions you use.

Answers to Check Yourself

§3.2, page 182: 2.
§3.5, page 221: 3.