J

Computer Arithmetic

by David Goldberg
Xerox Palo Alto Research Center

The Fast drives out the Slow even if the Fast is wrong.

W. Kahan

Introduction

Although computer arithmetic is sometimes viewed as a specialized part of CPU design, it is a very important part. This was brought home for Intel in 1994 when their Pentium chip was discovered to have a bug in the divide algorithm. This floating-point flaw resulted in a flurry of bad publicity for Intel and also cost them a lot of money. Intel took a $300 million write-off to cover the cost of replacing the buggy chips.

In this appendix, we will study some basic floating-point algorithms, including the division algorithm used on the Pentium. Although a tremendous variety of algorithms have been proposed for use in floating-point accelerators, actual implementations are usually based on refinements and variations of the few basic algorithms presented here. In addition to choosing algorithms for addition, subtraction, multiplication, and division, the computer architect must make other choices. What precisions should be implemented? How should exceptions be handled? This appendix will give you the background for making these and other decisions.

Our discussion of floating point will focus almost exclusively on the IEEE floating-point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating-point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms (but not necessarily sharing the hardware that implements integer instructions). Thus, after our discussion of floating point, we will take a more detailed look at integer algorithms.

Some good references on computer arithmetic, in order from least to most detailed, are Chapter 3 of Patterson and Hennessy [2009]; Chapter 7 of Hamacher, Vranesic, and Zaky [1984]; Gosling [1980]; and Scott [1985].

Basic Techniques of Integer Arithmetic

Readers who have studied computer arithmetic before will find most of this section to be review.

Ripple-Carry Addition

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are *half adders* and *full adders*. The half adder takes two bits a and b as input and produces a sum bit s and a carry bit c_{out} as output. Mathematically, $s = (a + b) \bmod 2$, and $c_{out} = \lfloor (a + b)/2 \rfloor$, where $\lfloor \ \rfloor$ is the floor function. As logic equations, $s = a\overline{b} + \overline{a}b$ and $c_{out} = ab$, where ab means $a \wedge b$ and $a + b$ means $a \vee b$. The half adder is also called a (2,2) adder, since it takes two inputs and produces two outputs. The full adder

is a (3,2) adder and is defined by $s = (a + b + c) \bmod 2$, $c_{\text{out}} = \lfloor (a + b + c)/2 \rfloor$, or the logic equations

J.2.1
$$s = a\bar{b}\,\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

J.2.2
$$c_{\text{out}} = ab + ac + bc$$

The principal problem in constructing an adder for n-bit numbers out of smaller pieces is propagating the carries from one piece to the next. The most obvious way to solve this is with a *ripple-carry adder*, consisting of n full adders, as illustrated in Figure J.1. (In the figures in this appendix, the least-significant bit is always on the right.) The inputs to the adder are $a_{n-1}a_{n-2} \cdots a_0$ and $b_{n-1}b_{n-2} \cdots b_0$, where $a_{n-1}a_{n-2} \cdots a_0$ represents the number $a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_0$. The c_{i+1} output of the ith adder is fed into the c_{i+1} input of the next adder (the $(i + 1)$-th adder) with the lower-order carry-in c_0 set to 0. Since the low-order carry-in is wired to 0, the low-order adder could be a half adder. Later, however, we will see that setting the low-order carry-in bit to 1 is useful for performing subtraction.

In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. However, determining the exact relationship between logic levels and timings is highly technology dependent. Therefore, when comparing adders we will simply compare the number of logic levels in each one. How many levels are there for a ripple-carry adder? It takes two levels to compute c_1 from a_0 and b_0. Then it takes two more levels to compute c_2 from c_1, a_1, b_1, and so on, up to c_n. So, there are a total of $2n$ levels. Typical values of n are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way.

Because the ripple-carry adder is relatively slow compared with the designs discussed in Section J.8, you might wonder why it is used at all. In technologies like CMOS, even though ripple adders take time $O(n)$, the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders.
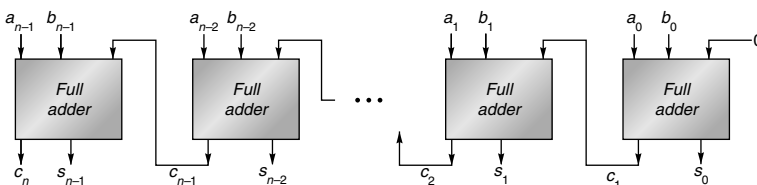

Figure J.1 Ripple-carry adder, consisting of *n* full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

Radix-2 Multiplication and Division

The simplest multiplier computes the product of two unsigned numbers, one bit at a time, as illustrated in Figure J.2(a). The numbers to be multiplied are $a_{n-1}a_{n-2} \cdots a_0$ and $b_{n-1}b_{n-2} \cdots b_0$, and they are placed in registers A and B, respectively. Register P is initially 0. Each multiply step has two parts:

Multiply Step   (*i*)   If the least-significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2} \cdots b_0$, is added to P; otherwise, $00 \cdots 00$ is added to P. The sum is placed back into P.

(*ii*)   Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

(a)

(b)
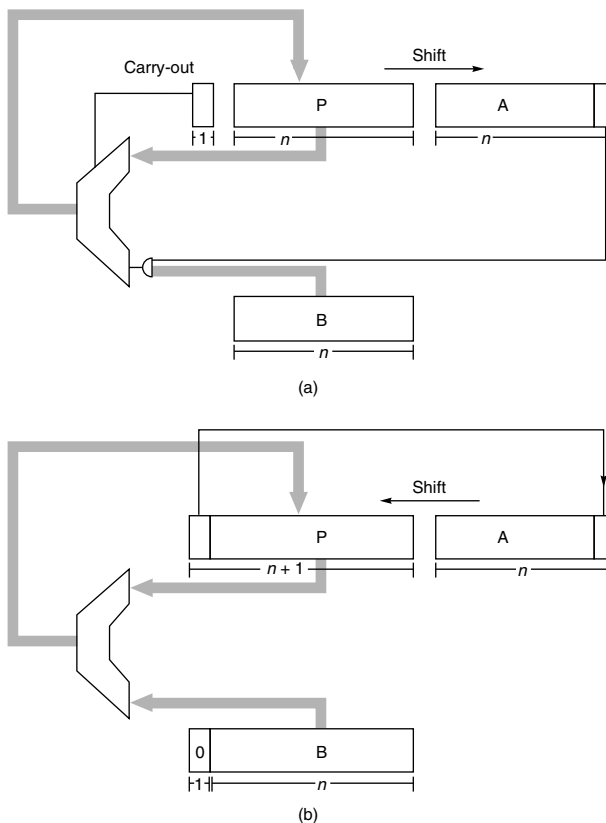
Figure J.2 Block diagram of (a) multiplier and (b) divider for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

After *n* steps, the product appears in registers P and A, with A holding the lower-order bits.

The simplest divider also operates on unsigned numbers and produces the quotient bits one at a time. A hardware divider is shown in Figure J.2(b). To compute *a/b*, put *a* in the A register, *b* in the B register, and 0 in the P register and then perform *n* divide steps. Each divide step consists of four parts:

Divide Step

(*i*)   Shift the register pair (P,A) one bit left.

(*ii*)   Subtract the content of register B (which is $b_{n-1}b_{n-2} \cdots b_0$) from register P, putting the result back into P.

(*iii*) If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

(*iv*)   If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this process *n* times, the A register will contain the quotient, and the P register will contain the remainder. This algorithm is the binary version of the paper-and-pencil method; a numerical example is illustrated in Figure J.3(a).

Notice that the two block diagrams in Figure J.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in Figure J.3(a) is called *restoring,* because if subtraction by *b* yields a negative result, the P register is restored by adding *b* back in. The restoring algorithm has a variant that skips the restoring step and instead works with the resulting negative numbers. Each step of this *nonrestoring* algorithm has three parts:

Nonrestoring Divide Step

If P is negative,

(*i-a*)   Shift the register pair (P,A) one bit left.

(*ii-a*) Add the contents of register B to P.

Else,

(*i-b*)   Shift the register pair (P,A) one bit left.

(*ii-b*) Subtract the contents of register B from P.

(*iii*)   If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this *n* times, the quotient is in A. If P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add *b*), and then it will be the remainder. A numerical example is given in Figure J.3(b). Since steps (*i-a*) and (*i-b*) are the same, you might be tempted to perform this common step first, and then test the sign of P. That doesn't work, since the sign bit can be lost when shifting.

```
         P        A
      00000    1110      Divide 14 = 1110₂ by 3 = 11₂. B always contains 0011₂.
      00001    110       step 1(i): shift.
     −00011               step 1(ii): subtract.
     −00010    1100      step 1(iii): result is negative, set quotient bit to 0.
      00001    1100      step 1(iv): restore.
      00011    100       step 2(i): shift.
     −00011               step 2(ii): subtract.
      00000    1001      step 2(iii): result is nonnegative, set quotient bit to 1.
      00001    001       step 3(i): shift.
     −00011               step 3(ii): subtract.
     −00010    0010      step 3(iii): result is negative, set quotient bit to 0.
      00001    0010      step 3(iv): restore.
      00010    010       step 4(i): shift.
     −00011               step 4(ii): subtract.
     −00001    0100      step 4(iii): result is negative, set quotient bit to 0.
      00010    0100      step 4(iv): restore. The quotient is 0100₂ and the remainder is 00010₂.

                                       (a)


      00000    1110      Divide 14 = 1110₂ by 3 = 11₂. B always contains 0011₂.
      00001    110       step 1(i-b): shift.
     +11101               step 1(ii-b): subtract b (add two's complement).
      11110    1100      step 1(iii): P is negative, so set quotient bit to 0.
      11101    100       step 2(i-a): shift.
     +00011               step 2(ii-a): add b.
      00000    1001      step 2(iii): P is nonnegative, so set quotient bit to 1.
      00001    001       step 3(i-b): shift.
     +11101               step 3(ii-b): subtract b.
      11110    0010      step 3(iii): P is negative, so set quotient bit to 0.
      11100    010       step 4(i-a): shift.
     +00011               step 4(ii-a): add b.
      11111    0100      step 4(iii): P is negative, so set quotient bit to 0.
     +00011               Remainder is negative, so do final restore step.
      00010               The quotient is 0100₂ and the remainder is 00010₂.

                                       (b)
```

Figure J.3 Numerical example of (a) restoring division and (b) nonrestoring division.

The explanation for why the nonrestoring algorithm works is this. Let r_k be the contents of the (P,A) register pair at step k, ignoring the quotient bits (which are simply sharing the unused bits of register A). In Figure J.3(a), initially A contains 14, so $r_0 = 14$. At the end of the first step, $r_1 = 28$, and so on. In the restoring

algorithm, part (i) computes $2r_k$ and then part (ii) $2r_k - 2^n b$ ($2^n b$ since b is sub-tracted from the left half). If $2r_k - 2^n b \geq 0$, both algorithms end the step with identical values in (P,A). If $2r_k - 2^n b < 0$, then the restoring algorithm restores this to $2r_k$, and the next step begins by computing $r_{res} = 2(2r_k) - 2^n b$. In the non-restoring algorithm, $2r_k - 2^n b$ is kept as a negative number, and in the next step $r_{nonres} = 2(2r_k - 2^n b) + 2^n b = 4r_k - 2^n b = r_{res}$. Thus (P,A) has the same bits in both algorithms.

If a and b are unsigned n-bit numbers, hence in the range $0 \leq a,b \leq 2^n - 1$, then the multiplier in Figure J.2 will work if register P is n bits long. However, for division, P must be extended to $n + 1$ bits in order to detect the sign of P. Thus the adder must also have $n + 1$ bits.

Why would anyone implement restoring division, which uses the same hard-ware as nonrestoring division (the control is slightly different) but involves an extra addition? In fact, the usual implementation for restoring division doesn't actually perform an add in step (*iv*). Rather, the sign resulting from the sub-traction is tested at the output of the adder, and only if the sum is nonnegative is it loaded back into the P register.

As a final point, before beginning to divide, the hardware must check to see whether the divisor is 0.

Signed Numbers

There are four methods commonly used to represent signed n-bit numbers: *sign magnitude*, *two's complement*, *one's complement*, and *biased*. In the sign magni-tude system, the high-order bit is the sign bit, and the low-order $n - 1$ bits are the magnitude of the number. In the two's complement system, a number and its neg-ative add up to 2^n. In one's complement, the negative of a number is obtained by complementing each bit (or, alternatively, the number and its negative add up to $2^n - 1$). In each of these three systems, nonnegative numbers are represented in the usual way. In a biased system, nonnegative numbers do not have their usual representation. Instead, all numbers are represented by first adding them to the bias and then encoding this sum as an ordinary unsigned number. Thus, a nega-tive number k can be encoded as long as $k + \text{bias} \geq 0$. A typical value for the bias is 2^{n-1}.

Example   Using 4-bit numbers ($n = 4$), if $k = 3$ (or in binary, $k = 0011_2$), how is $-k$ expressed in each of these formats?

Answer   In signed magnitude, the leftmost bit in $k = \mathbf{0}011_2$ is the sign bit, so flip it to 1: $-k$ is represented by $\mathbf{1}011_2$. In two's complement, $k + 1101_2 = 2^n = 16$. So $-k$ is rep-resented by 1101_2. In one's complement, the bits of $k = 0011_2$ are flipped, so $-k$ is represented by 1100_2. For a biased system, assuming a bias of $2^{n-1} = 8$, k is represented by $k + \text{bias} = 1011_2$, and $-k$ by $-k + \text{bias} = 0101_2$.

The most widely used system for representing integers, two's complement, is the system we will use here. One reason for the popularity of two's complement is that it makes signed addition easy: Simply discard the carry-out from the high-order bit. To add $5 + -2$, for example, add 0101_2 and 1110_2 to obtain 0011_2, resulting in the correct value of 3. A useful formula for the value of a two's complement number $a_{n-1}a_{n-2} \cdots a_1 a_0$ is

J.2.3
$$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_1 2^1 + a_0$$

As an illustration of this formula, the value of 1101_2 as a 4-bit two's complement number is $-1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 1 = -3$, confirming the result of the example above.

Overflow occurs when the result of the operation does not fit in the representation being used. For example, if unsigned numbers are being represented using 4 bits, then $6 = 0110_2$ and $11 = 1011_2$. Their sum (17) overflows because its binary equivalent (10001_2) doesn't fit into 4 bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is a carry-out of the most-significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into the high-order bit is different from the (to be discarded) carry-out of the high-order bit. In the example of $5 + -2$ above, a 1 is carried both into and out of the leftmost bit, avoiding overflow.

Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate 0011_2, complement it to get 1100_2 and then add 1 to get 1101_2. Thus, to implement $a - b$ using an adder, simply feed a and $\overline{b}$ (where $\overline{b}$ is the number obtained by complementing each bit of b) into the adder and set the low-order, carry-in bit to 1. This explains why the rightmost adder in Figure J.1 is a full adder.

Multiplying two's complement numbers is not quite as simple as adding them. The obvious approach is to convert both operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result. Although this is conceptually simple, it requires extra time and hardware. Here is a better approach: Suppose that we are multiplying a times b using the hardware shown in Figure J.2(a). Register A is loaded with the number a; B is loaded with b. Since the content of register B is always b, we will use B and b interchangeably. If B is potentially negative but A is nonnegative, the only change needed to convert the unsigned multiplication algorithm into a two's complement one is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P (rather than the carry-out from the addition). Note that our n-bit-wide adder will now be adding n-bit two's complement numbers between -2^{n-1} and $2^{n-1} - 1$.

Next, suppose a is negative. The method for handling this case is called *Booth recoding*. Booth recoding is a very basic technique in computer arithmetic and will play a key role in Section J.9. The algorithm on page J-4 computes $a \times b$ by examining the bits of a from least significant to most significant. For example, if $a = 7 = 0111_2$, then step (*i*) will successively add B, add B, add B, and add 0. Booth recoding "recodes" the number 7 as $8 - 1 = 1000_2 - 0001_2 = 100\overline{1}$, where

$\overline{1}$ represents -1. This gives an alternative way to compute $a \times b$, namely, successively subtract B, add 0, add 0, and add B. This is more complicated than the unsigned algorithm on page J-4, since it uses both addition and subtraction. The advantage shows up for negative values of a. With the proper recoding, we can treat a as though it were unsigned. For example, take $a = -4 = 1100_2$. Think of 1100_2 as the unsigned number 12, and recode it as $12 = 16 - 4 = 10000_2 - 0100_2 = 10\overline{1}00$. If the multiplication algorithm is only iterated n times ($n = 4$ in this case), the high-order digit is ignored, and we end up subtracting $0100_2 = 4$ times the multiplier—exactly the right answer. This suggests that multiplying using a recoded form of a will work equally well for both positive and negative numbers. And, indeed, to deal with negative values of a, all that is required is to sometimes subtract b from P, instead of adding either b or 0 to P. Here are the precise rules: If the initial content of A is $a_{n-1} \cdots a_0$, then at the ith multiply step the low-order bit of register A is a_i, and step (i) in the multiplication algorithm becomes:

I.   If $a_i = 0$ and $a_{i-1} = 0$, then add 0 to P.

II. If $a_i = 0$ and $a_{i-1} = 1$, then add B to P.

III. If $a_i = 1$ and $a_{i-1} = 0$, then subtract B from P.

IV. If $a_i = 1$ and $a_{i-1} = 1$, then add 0 to P.

For the first step, when $i = 0$, take a_{i-1} to be 0.

Example   When multiplying -6 times -5, what is the sequence of values in the (P,A) register pair?

Answer   See Figure J.4.

```
   P       A
 0000    1010     Put −6 = 1010₂ into A, −5 = 1011₂ into B.

 0000    1010     step 1(i): a₀ = a₋₁ = 0, so from rule I add 0.

 0000    0101     step 1(ii): shift.

+ 0101            step 2(i): a₁ = 1, a₀ = 0. Rule III says subtract b (or add −b = −1011₂ = 0101₂).

 0101    0101

 0010    1010     step 2(ii): shift.

+ 1011            step 3(i): a₂ = 0, a₁ = 1. Rule II says add b (1011).

 1101    1010

 1110    1101     step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)

+ 0101            step 4(i): a₃ = 1, a₂ = 0. Rule III says subtract b.

 0011    1101

 0001    1110     step 4(ii): shift. Final result is 00011110₂ = 30.
```

Figure J.4 Numerical example of Booth recoding. Multiplication of $a = -6$ by $b = -5$ to get 30.

The four prior cases can be restated as saying that in the ith step you should add $(a_{i-1} - a_i)$B to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) + ba_{-1}$$

Using Equation J.2.3 (page J-8) together with $a_{-1} = 0$, the right-hand side is seen to be the value of $b \times a$ as a two's complement number.

The simplest way to implement the rules for Booth recoding is to extend the A register one bit to the right so that this new bit will contain a_{i-1}. Unlike the naive method of inverting any negative operands, this technique doesn't require extra steps or any special casing for negative operands. It has only slightly more control logic. If the multiplier is being shared with a divider, there will already be the capability for subtracting b, rather than adding it. To summarize, a simple method for handling two's complement multiplication is to pay attention to the sign of P when shifting it right, and to save the most recently shifted-out bit of A to use in deciding whether to add or subtract b from P.

Booth recoding is usually the best method for designing multiplication hardware that operates on signed numbers. For hardware that doesn't directly implement it, however, performing Booth recoding in software or microcode is usually too slow because of the conditional tests and branches. If the hardware supports arithmetic shifts (so that negative b is handled correctly), then the following method can be used. Treat the multiplier a as if it were an unsigned number, and perform the first $n - 1$ multiply steps using the algorithm on page J-4. If $a < 0$ (in which case there will be a 1 in the low-order bit of the A register at this point), then subtract b from P; otherwise ($a \geq 0$), neither add nor subtract. In either case, do a final shift (for a total of n shifts). This works because it amounts to multiplying b by $-a_{n-1} 2^{n-1} + \cdots + a_1 2 + a_0$, which is the value of $a_{n-1} \cdots a_0$ as a two's complement number by Equation J.2.3. If the hardware doesn't support arithmetic shift, then converting the operands to be nonnegative is probably the best approach.

Two final remarks: A good way to test a signed-multiply routine is to try $-2^{n-1} \times -2^{n-1}$, since this is the only case that produces a $2n - 1$ bit result. Unlike multiplication, division is usually performed in hardware by converting the operands to be nonnegative and then doing an unsigned divide. Because division is substantially slower (and less frequent) than multiplication, the extra time used to manipulate the signs has less impact than it does on multiplication.

Systems Issues

When designing an instruction set, a number of issues related to integer arithmetic need to be resolved. Several of them are discussed here.

First, what should be done about integer overflow? This situation is complicated by the fact that detecting overflow differs depending on whether the operands

are signed or unsigned integers. Consider signed arithmetic first. There are three approaches: Set a bit on overflow, trap on overflow, or do nothing on overflow. In the last case, software has to check whether or not an overflow occurred. The most convenient solution for the programmer is to have an enable bit. If this bit is turned on, then overflow causes a trap. If it is turned off, then overflow sets a bit (or, alternatively, have two different add instructions). The advantage of this approach is that both trapping and nontrapping operations require only one instruction. Furthermore, as we will see in Section J.7, this is analogous to how the IEEE floating-point standard handles floating-point overflow. Figure J.5 shows how some common machines treat overflow.

What about unsigned addition? Notice that none of the architectures in Figure J.5 traps on unsigned overflow. The reason for this is that the primary use of unsigned arithmetic is in manipulating addresses. It is convenient to be able to subtract from an unsigned address by adding. For example, when $n = 4$, we can subtract 2 from the unsigned address $10 = 1010_2$ by adding $14 = 1110_2$. This generates an overflow, but we would not want a trap to be generated.

A second issue concerns multiplication. Should the result of multiplying two n-bit numbers be a $2n$-bit result, or should multiplication just return the low-order n bits, signaling overflow if the result doesn't fit in n bits? An argument in favor of an n-bit result is that in virtually all high-level languages, multiplication is an operation in which arguments are integer variables and the result is an integer variable of the same type. Therefore, compilers won't generate code that utilizes a double-precision result. An argument in favor of a $2n$-bit result is that it can be used by an assembly language routine to substantially speed up multiplication of multiple-precision integers (by about a factor of 3).

A third issue concerns machines that want to execute one instruction every cycle. It is rarely practical to perform a multiplication or division in the same amount of time that an addition or register-register move takes. There are three possible approaches to this problem. The first is to have a single-cycle *multiply-step* instruction. This might do one step of the Booth algorithm. The second

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. Add sets V bit.	Yes. Add sets C bit.
IBM 370	If enable is on	No	Yes. Add sets cond code.	Yes. Logical add sets cond code.
Intel 8086	No	No	Yes. Add sets V bit.	Yes. Add sets C bit.
MIPS R3000	Two add instructions; one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	Addcc sets V bit. Add does not.	Addcc sets C bit. Add does not.

Figure J.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

approach is to do integer multiplication in the floating-point unit and have it be part of the floating-point instruction set. (This is what DLX does.) The third approach is to have an autonomous unit in the CPU do the multiplication. In this case, the result either can be guaranteed to be delivered in a fixed number of cycles—and the compiler charged with waiting the proper amount of time—or there can be an interlock. The same comments apply to division as well. As examples, the original SPARC had a multiply-step instruction but no divide-step instruction, while the MIPS R3000 has an autonomous unit that does multiplication and division (newer versions of the SPARC architecture added an integer multiply instruction). The designers of the HP Precision Architecture did an especially thorough job of analyzing the frequency of the operands for multiplication and division, and they based their multiply and divide steps accordingly. (See Magenheimer et al. [1988] for details.)

The final issue involves the computation of integer division and remainder for negative numbers. For example, what is −5 DIV 3 and −5 MOD 3? When computing x DIV y and x MOD y, negative values of x occur frequently enough to be worth some careful consideration. (On the other hand, negative values of y are quite rare.) If there are built-in hardware instructions for these operations, they should correspond to what high-level languages specify. Unfortunately, there is no agreement among existing programming languages. See Figure J.6.

One definition for these expressions stands out as clearly superior, namely, x DIV $y = \lfloor x/y \rfloor$, so that 5 DIV 3 = 1 and −5 DIV 3 = −2. And MOD should satisfy $x = (x$ DIV $y) \times y + x$ MOD y, so that x MOD $y \geq 0$. Thus, 5 MOD 3 = 2, and −5 MOD 3 = 1. Some of the many advantages of this definition are as follows:

1. A calculation to compute an index into a hash table of size N can use MOD N and be guaranteed to produce a valid index in the range from 0 to $N − 1$.

2. In graphics, when converting from one coordinate system to another, there is no "glitch" near 0. For example, to convert from a value x expressed in a system that uses 100 dots per inch to a value y on a bitmapped display with 70 dots per inch, the formula $y = (70 \times x)$ DIV 100 maps one or two x coordinates into each y coordinate. But if DIV were defined as in Pascal to be x/y rounded to 0, then 0 would have three different points (−1, 0, 1) mapped into it.

3. x MOD 2^k is the same as performing a bitwise AND with a mask of k bits, and x DIV 2^k is the same as doing a k-bit arithmetic right shift.

Language	Division	Remainder
FORTRAN	−5/3 = −1	MOD(−5, 3) = −2
Pascal	−5 DIV 3 = −1	−5 MOD 3 = 1
Ada	−5/3 = −1	−5 MOD 3 = 1 −5 REM 3 = −2
C	−5/3 undefined	−5 % 3 undefined
Modula-3	−5 DIV 3 = −2	−5 MOD 3 = 1

Figure J.6 Examples of integer division and integer remainder in various programming languages.

Finally, a potential pitfall worth mentioning concerns multiple-precision addition. Many instruction sets offer a variant of the add instruction that adds three operands: two *n*-bit numbers together with a third single-bit number. This third number is the carry from the previous addition. Since the multiple-precision number will typically be stored in an array, it is important to be able to increment the array pointer without destroying the carry bit.

J.3   Floating Point

Many applications require numbers that aren't integers. There are a number of ways that nonintegers can be represented. One is to use *fixed point*; that is, use integer arithmetic and simply imagine the binary point somewhere other than just to the right of the least-significant digit. Adding two such numbers can be done with an integer add, whereas multiplication requires some extra shifting. Other representations that have been proposed involve storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers (*a,b*) to represent the fraction *a/b*. However, only one noninteger representation has gained widespread use, and that is *floating point.* In this system, a computer word is divided into two parts, an exponent and a significand. As an example, an exponent of −3 and a significand of 1.5 might represent the number 1.5×2^{-3} = 0.1875. The advantages of standardizing a particular representation are obvious. Numerical analysts can build up high-quality software libraries, computer designers can develop techniques for implementing high-performance hardware, and hardware vendors can build standard accelerators. Given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use.

The semantics of floating-point instructions are not as clear-cut as the semantics of the rest of the instruction set, and in the past the behavior of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and overflow. Computer architecture books used to dispense advice on how to deal with all these details, but fortunately this is no longer necessary. That's because the computer industry is rapidly converging on the format specified by IEEE standard 754-1985 (also an international standard, IEC 559). The advantages of using a standard variant of floating point are similar to those for using floating point over other noninteger representations.

IEEE arithmetic differs from many previous arithmetics in the following major ways:

1. When rounding a "halfway" result to the nearest floating-point number, it picks the one that is even.

2. It includes the *special values* NaN, ∞, and −∞.

3. It uses *denormal* numbers to represent the result of computations whose value is less than $1.0 \times 2^{E_{min}}$.

4. It rounds to nearest by default, but it also has three other rounding modes.

5. It has sophisticated facilities for handling exceptions.

To elaborate on (1), note that when operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number. For example, in a floating-point system using base 10 and two significant digits, $6.1 \times 0.5 = 3.05$. This needs to be rounded to two digits. Should it be rounded to 3.0 or 3.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 3.05 rounds to 3.0, not 3.1. The standard actually has four *rounding modes*. The default is *round to nearest*, which rounds ties to an even number as just explained. The other modes are round toward 0, round toward $+\infty$, and round toward $-\infty$.

We will elaborate on the other differences in following sections. For further reading, see IEEE [1985], Cody et al. [1984], and Goldberg [1991].

Special Values and Denormals

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a *NaN* (*N*ot *a N*umber), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful, consider the code for a zero finder that takes a function F as an argument and evaluates F at various points to determine a zero for it. If the zero finder accidentally probes outside the valid values for F, then F may well cause an exception. Writing a zero finder that deals with this case is highly language and operating-system dependent, because it relies on how the operating system reacts to exceptions and how this reaction is mapped back into the programming language. In IEEE arithmetic it is easy to write a zero finder that handles this situation and runs on many different systems. After each evaluation of F, it simply checks to see whether F has returned a NaN; if so, it knows it has probed outside the domain of F.

In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.g., 3 + NaN = NaN). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks. For example, suppose that arccos is computed in terms of arctan, using the formula arccos $x = 2$ arctan$(\sqrt{(1-x)/(1+x)}\,)$. If arctan handles an argument of NaN properly, arccos will automatically do so, too. That's because if x is a NaN, $1+x$, $1-x$, $(1+x)/(1-x)$, and $\sqrt{(1-x)/(1+x)}$ will also be NaNs. No checking for NaNs is required.

While the result of $\sqrt{-1}$ is a NaN, the result of $1/0$ is not a NaN, but $+\infty$, which is another special value. The standard defines arithmetic on infinities (there are both

+∞ and −∞) using rules such as $1/\infty = 0$. The formula arccos $x = 2$ arctan$(\sqrt{(1-x)/(1+x)})$ illustrates how infinity arithmetic can be used. Since arctan x asymptotically approaches $\pi/2$ as x approaches ∞, it is natural to define arctan(∞) $= \pi/2$, in which case arccos(−1) will automatically be computed correctly as 2 arctan(∞) $= \pi$.

The final kind of special values in the standard are *denormal* numbers. In many floating-point systems, if E_{min} is the smallest exponent, a number less than $1.0 \times 2^{E_{min}}$ cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand, numbers less than $1.0 \times 2^{E_{min}}$ are represented using significands less than 1. This is called *gradual underflow*. Thus, as numbers decrease in magnitude below $2^{E_{min}}$, they gradually lose their significance and are only represented by 0 when all their significance has been shifted out. For example, in base 10 with four significant figures, let $x = 1.234 \times 10^{E_{min}}$. Then, $x/10$ will be rounded to $0.123 \times 10^{E_{min}}$, having lost a digit of precision. Similarly $x/100$ rounds to $0.012 \times 10^{E_{min}}$, and $x/1000$ to $0.001 \times 10^{E_{min}}$, while $x/10000$ is finally small enough to be rounded to 0. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as $x = y \Leftrightarrow x - y = 0$. For example, in a flush-to-zero system (again in base 10 with four significant digits), if $x = 1.256 \times 10^{E_{min}}$ and $y = 1.234 \times 10^{E_{min}}$, then $x - y = 0.022 \times 10^{E_{min}}$, which flushes to zero. So even though $x \neq y$, the computed value of $x - y = 0$. This never happens with gradual underflow. In this example, $x - y = 0.022 \times 10^{E_{min}}$ is a denormal number, and so the computation of $x - y$ is exact.

Representation of Floating-Point Numbers

Let us consider how to represent single-precision numbers in IEEE arithmetic. Single-precision numbers are stored in 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method (see the subsection "Signed Numbers," page J-7) with a bias of 127. The term *biased exponent* refers to the unsigned number contained in bits 1 through 8, and *unbiased exponent* (or just exponent) means the actual power to which 2 is to be raised. The fraction represents a number less than 1, but the *significand* of the floating-point number is 1 plus the fraction part. In other words, if e is the biased exponent (value of the exponent field) and f is the value of the fraction field, the number being represented is $1.f \times 2^{e-127}$.

Example   What single-precision number does the following 32-bit word represent?

    1 10000001 01000000000000000000000

Answer   Considered as an unsigned number, the exponent field is 129, making the value of the exponent $129 - 127 = 2$. The fraction part is $.01_2 = .25$, making the significand 1.25. Thus, this bit pattern represents the number $-1.25 \times 2^2 = -5$.

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand $1.f$ does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part is usually *unpacked*, which is to say the implicit 1 is made explicit.

Figure J.7 summarizes the parameters for single (and other) precisions. It shows the exponents for single precision to range from –126 to 127; accordingly, the biased exponents range from 1 to 254. The biased exponents of 0 and 255 are used to represent special values. This is summarized in Figure J.8. When the biased exponent is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are 0, then the number represented is 0. Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to 1. Thus, a special convention such as this is required to represent 0. Denormalized numbers are implemented by having a word with a zero exponent field represent the number $0.f \times 2^{E_{\min}}$.

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that 0 is represented by a word of all 0s. The downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

	Single	Single extended	Double	Double extended
p (bits of precision)	24	≥ 32	53	≥ 64
$E_{\max}$	127	≥ 1023	1023	≥ 16383
$E_{\min}$	–126	≤ -1022	–1022	≤ -16382
Exponent bias	127		1023	

Figure J.7 Format parameters for the IEEE 754 floating-point standard. The first row gives the number of bits in the significand. The blanks are unspecified parameters.

Exponent	Fraction	Represents
$e = E_{\min} - 1$	$f = 0$	± 0
$e = E_{\min} - 1$	$f \neq 0$	$0.f \times 2^{E_{\min}}$
$E_{\min} \leq e \leq E_{\max}$	—	$1.f \times 2^e$
$e = E_{\max} + 1$	$f = 0$	$\pm \infty$
$e = E_{\max} + 1$	$f \neq 0$	NaN

Figure J.8 Representation of special values. When the exponent of a number falls outside the range $E_{\min} \leq e \leq E_{\max}$, then that number has a special interpretation as indicated in the table.

J.4   **Floating-Point Multiplication**

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number x is represented as a significand and an exponent, $x = s \times 2^e$. The formula

$$\left(s_1 \times 2^{e1}\right) \bullet \left(s_2 \times 2^{e2}\right) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating-point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned p-bit numbers (e.g., $p = 24$ for single precision), then the product can have as many as $2p$ bits and must be rounded to a p-bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents.

Example   How does the multiplication of the single-precision numbers

$$1\ 10000010\ 000... = -1 \times 2^3$$

$$0\ 10000011\ 000... = 1 \times 2^4$$

proceed in binary?

Answer   When unpacked, the significands are both 1.0, their product is 1.0, and so the result is of the form:

$$1\ ????????\ 000...$$

To compute the exponent, use the formula:

biased exp $(e_1 + e_2)$ = biased exp(e_1) + biased exp(e_2) − bias

From Figure J.7, the bias is $127 = 01111111_2$, so in two's complement -127 is 10000001_2. Thus, the biased exponent of the product is

$$\begin{array}{r} 10000010 \\ 10000011 \\ +\ 10000001 \\ \hline 10000110 \end{array}$$

Since this is 134 decimal, it represents an exponent of $134 - \text{bias} = 134 - 127 = 7$, as expected.

The interesting part of floating-point multiplication is rounding. Some of the different cases that can occur are illustrated in Figure J.9. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2.
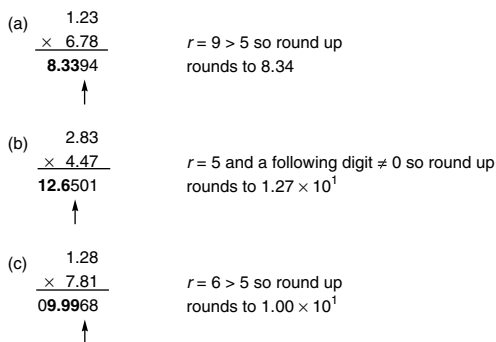
(a)
```
    1.23
  × 6.78
  8.3394
      ↑
```
$r = 9 > 5$ so round up
rounds to 8.34

(b)
```
    2.83
  × 4.47
 12.6501
      ↑
```
$r = 5$ and a following digit $\neq 0$ so round up
rounds to 1.27×10^1

(c)
```
    1.28
  × 7.81
 09.9968
      ↑
```
$r = 6 > 5$ so round up
rounds to 1.00×10^1

Figure J.9 Examples of rounding a multiplication. Using base 10 and $p = 3$, parts (a) and (b) illustrate that the result of a multiplication can have either $2p - 1$ or $2p$ digits; hence, the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

In the figure, $p = 3$, so the final result must be rounded to three significant digits. The three most-significant digits are in boldface. The fourth most-significant digit (marked with an arrow) is the *round* digit, denoted by r.

If the round digit is less than 5, then the bold digits represent the rounded result. If the round digit is greater than 5 (as in part (a)), then 1 must be added to the least-significant bold digit. If the round digit is exactly 5 (as in part (b)), then additional digits must be examined to decide between truncation or incrementing by 1. It is only necessary to know if any digits past 5 are nonzero. In the algorithm below, this will be recorded in a *sticky bit*. Comparing parts (a) and (b) in the figure shows that there are two possible positions for the round digit (relative to the least-significant digit of the product). Case (c) illustrates that, when adding 1 to the least-significant bold digit, there may be a carry-out. When this happens, the final significand must be 10.0.

There is a straightforward method of handling rounding using the multiplier of Figure J.2 (page J-4) together with an extra sticky bit. If p is the number of bits in the significand, then the A, B, and P registers should be p bits wide. Multiply the two significands to obtain a $2p$-bit product in the (P,A) registers (see Figure J.10). During the multiplication, the first $p - 2$ times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let s represent the sticky bit, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A. There are two cases:

1. The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.

2. The high-order bit of P is 1. Set $s := s \vee r$ and $r := g$, and add 1 to the exponent.

Now if $r = 0$, P is the correctly rounded product. If $r = 1$ and $s = 1$, then P + 1 is the product (where by P + 1 we mean adding 1 to the least-significant bit of P).
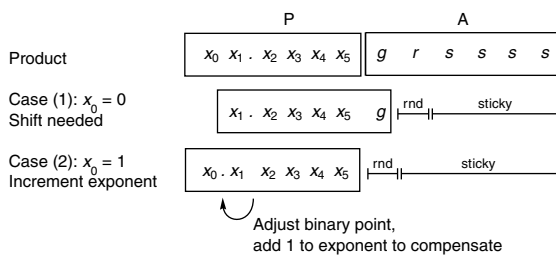
Figure J.10 The two cases of the floating-point multiply algorithm. The top line shows the contents of the P and A registers after multiplying the significands, with $p = 6$. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked s.

If $r = 1$ and $s = 0$, we are in a halfway case and round up according to the least-significant bit of P. As an example, apply the decimal version of these rules to Figure J.9(b). After the multiplication, P = 126 and A = 501, with $g = 5$, $r = 0$ and $s = 1$. Since the high-order digit of P is nonzero, case (2) applies and $r := g$, so that $r = 5$, as the arrow indicates in Figure J.9. Since $r = 5$, we could be in a halfway case, but $s = 1$ indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product.

The precise rules for rounding depend on the rounding mode and are given in Figure J.11. Note that P is nonnegative, that is, it contains the magnitude of the result. A good discussion of more efficient ways to implement rounding is in Santoro, Bewick, and Horowitz [1989].

Example   In binary with $p = 4$, show how the multiplication algorithm computes the product -5×10 in each of the four rounding modes.

Answer   In binary, -5 is $-1.010_2 \times 2^2$ and $10 = 1.010_2 \times 2^3$. Applying the integer multiplication algorithm to the significands gives 01100100_2, so P = 0110_2, A = 0100_2, $g = 0$, $r = 1$, and $s = 0$. The high-order bit of P is 0, so case (1) applies. Thus, P becomes 1100_2, and since the result is negative, Figure J.11 gives:

round to $-\infty$	1101_2	add 1 since $r \vee s = 1/0 =$ TRUE
round to $+\infty$	1100_2	
round to 0	1100_2	
round to nearest	1100_2	no add since $r \wedge p_0 = 1 \wedge 0 =$ FALSE and
		$r \wedge s = 1 \wedge 0 =$ FALSE

The exponent is $2 + 3 = 5$, so the result is $-1.100_2 \times 2^5 = -48$, except when rounding to $-\infty$, in which case it is $-1.101_2 \times 2^5 = -52$.

Rounding mode	Sign of result ≥ 0	Sign of result < 0
$-\infty$		$+1$ if $r \vee s$
$+\infty$	$+1$ if $r \vee s$	
0		
Nearest	$+1$ if $r \wedge p_0$ or $r \wedge s$	$+1$ if $r \wedge p_0$ or $r \wedge s$

Figure J.11 Rules for implementing the IEEE rounding modes. Let S be the magnitude of the preliminary result. Blanks mean that the p most-significant bits of S are the actual result bits. If the condition listed is true, add 1 to the pth most-significant bit of S. The symbols r and s represent the round and sticky bits, while p_0 is the pth most-significant bit of S.

Overflow occurs when the rounded result is too large to be represented. In single precision, this occurs when the result has an exponent of 128 or higher. If e_1 and e_2 are the two biased exponents, then $1 \leq e_i \leq 254$, and the exponent calculation $e_1 + e_2 - 127$ gives numbers between $1 + 1 - 127$ and $254 + 254 - 127$, or between -125 and 381. This range of numbers can be represented using 9 bits. So one way to detect overflow is to perform the exponent calculations in a 9-bit adder (see Exercise J.12). Remember that you must check for overflow *after* rounding—the example in Figure J.9(c) shows that this can make a difference.

Denormals

Checking for underflow is somewhat more complex because of denormals. In single precision, if the result has an exponent less than -126, that does not necessarily indicate underflow, because the result might be a denormal number. For example, the product of (1×2^{-64}) with (1×2^{-65}) is 1×2^{-129}, and -129 is below the legal exponent limit. But this result is a valid denormal number, namely, 0.125×2^{-126}. In general, when the unbiased exponent of a product dips below -126, the resulting product must be shifted right and the exponent incremented until the exponent reaches -126. If this process causes the entire significand to be shifted out, then underflow has occurred. The precise definition of underflow is somewhat subtle—see Section J.7 for details.

When one of the operands of a multiplication is denormal, its significand will have leading zeros, and so the product of the significands will also have leading zeros. If the exponent of the product is less than -126, then the result is denormal, so right-shift and increment the exponent as before. If the exponent is greater than -126, the result may be a normalized number. In this case, *left*-shift the product (while decrementing the exponent) until either it becomes normalized or the exponent drops to -126.

Denormal numbers present a major stumbling block to implementing floating-point multiplication, because they require performing a variable shift in the multiplier, which wouldn't otherwise be needed. Thus, high-performance, floating-point multipliers often do not handle denormalized numbers, but instead trap,

letting software handle them. A few practical codes frequently underflow, even when working properly, and these programs will run quite a bit slower on systems that require denormals to be processed by a trap handler.

So far we haven't mentioned how to deal with operands of zero. This can be handled by either testing both operands before beginning the multiplication or testing the product afterward. If you test afterward, be sure to handle the case $0 \times \infty$ properly: This results in NaN, not 0. Once you detect that the result is 0, set the biased exponent to 0. Don't forget about the sign. The sign of a product is the XOR of the signs of the operands, even when the result is 0.

Precision of Multiplication

In the discussion of integer multiplication, we mentioned that designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a "single times single gets double" instruction. The situation is different for floating point. Many languages allow assigning the product of two single-precision variables to a double-precision one, and the construction can also be exploited by numerical algorithms. The best-known case is using iterative refinement to solve linear systems of equations.

J.5   Floating-Point Addition

Typically, a floating-point operation takes two inputs with p bits of precision and returns a p-bit result. The ideal algorithm would compute this by first performing the operation exactly, and then rounding the result to p bits (using the current rounding mode). The multiplication algorithm presented in the previous section follows this strategy. Even though hardware implementing IEEE arithmetic must return the same result as the ideal algorithm, it doesn't need to actually perform the ideal algorithm. For addition, in fact, there are better ways to proceed. To see this, consider some examples.

First, the sum of the binary 6-bit numbers 1.10011_2 and $1.10001_2 \times 2^{-5}$: When the summands are shifted so they have the same exponent, this is

$$
\begin{array}{r}
1.10011 \\
+ \quad .0000110001 \\
\hline
\end{array}
$$

Using a 6-bit adder (and discarding the low-order bits of the second addend) gives

$$
\begin{array}{r}
1.10011 \\
+ \quad .00001 \\
\hline
1.10100 \\
\end{array}
$$

The first discarded bit is 1. This isn't enough to decide whether to round up. The rest of the discarded bits, 0001, need to be examined. Or, actually, we just need to record whether any of these bits are nonzero, storing this fact in a sticky bit just as in the multiplication algorithm. So, for adding two p-bit numbers, a p-bit adder is sufficient, as long as the first discarded bit (round) and the OR of the rest of the bits (sticky) are kept. Then Figure J.11 can be used to determine if a roundup is necessary, just as with multiplication. In the example above, sticky is 1, so a roundup is necessary. The final sum is 1.10101_2.

Here's another example:

$$\begin{array}{r} 1.11011 \\ + \quad .0101001 \\ \hline \end{array}$$

A 6-bit adder gives:

$$\begin{array}{r} 1.11011 \\ + \quad .01010 \\ \hline 10.00101 \end{array}$$

Because of the carry-out on the left, the round bit isn't the first discarded bit; rather, it is the low-order bit of the sum (1). The discarded bits, 01, are OR'ed together to make sticky. Because round and sticky are both 1, the high-order 6 bits of the sum, 10.0010_2, must be rounded up for the final answer of 10.0011_2.

Next, consider subtraction and the following example:

$$\begin{array}{r} 1.00000 \\ - \quad .00000101111 \\ \hline \end{array}$$

The simplest way of computing this is to convert $-.00000101111_2$ to its two's complement form, so the difference becomes a sum:

$$\begin{array}{r} 1.00000 \\ + 1.11111010001 \\ \hline \end{array}$$

Computing this sum in a 6-bit adder gives:

$$\begin{array}{r} 1.00000 \\ + 1.11111 \\ \hline 0.11111 \end{array}$$

Because the top bits canceled, the first discarded bit (the guard bit) is needed to fill in the least-significant bit of the sum, which becomes 0.111110_2, and the second discarded bit becomes the round bit. This is analogous to case (1) in the multiplication algorithm (see page J-19). The round bit of 1 isn't enough to decide whether to round up. Instead, we need to OR all the remaining bits (0001) into a sticky bit. In this case, sticky is 1, so the final result must be rounded up to 0.111111. This example shows that if subtraction causes the most-significant bit to cancel, then one guard bit is needed. It is natural to ask whether two guard bits

are needed for the case when the *two* most-significant bits cancel. The answer is no, because if x and y are so close that the top two bits of $x - y$ cancel, then $x - y$ will be exact, so guard bits aren't needed at all.

To summarize, addition is more complex than multiplication because, depending on the signs of the operands, it may actually be a subtraction. If it is an addition, there can be carry-out on the left, as in the second example. If it is subtraction, there can be cancellation, as in the third example. In each case, the position of the round bit is different. However, we don't need to compute the exact sum and then round. We can infer it from the sum of the high-order p bits together with the round and sticky bits.

The rest of this section is devoted to a detailed discussion of the floating-point addition algorithm. Let a_1 and a_2 be the two numbers to be added. The notations e_i and s_i are used for the exponent and significand of the addends a_i. This means that the floating-point inputs have been unpacked and that s_i has an explicit leading bit. To add a_1 and a_2, perform these eight steps:

1. If $e_1 < e_2$, swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.

2. If the signs of a_1 and a_2 differ, replace s_2 by its two's complement.

3. Place s_2 in a p-bit register and shift it $d = e_1 - e_2$ places to the right (shifting in 1's if s_2 was complemented in the previous step). From the bits shifted out, set g to the most-significant bit, set r to the next most-significant bit, and set sticky to the OR of the rest.

4. Compute a preliminary significand $S = s_1 + s_2$ by adding s_1 to the p-bit register containing s_2. If the signs of a_1 and a_2 are different, the most-significant bit of S is 1, and there was no carry-out, then S is negative. Replace S with its two's complement. This can only happen when $d = 0$.

5. Shift S as follows. If the signs of a_1 and a_2 are the same and there was a carry-out in step 4, shift S right by one, filling in the high-order position with 1 (the carry-out). Otherwise, shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.

6. Adjust r and s. If S was shifted right in step 5, set $r := $ low-order bit of S before shifting and $s := g$ OR r OR s. If there was no shift, set $r := g$, $s := r$ OR s. If there was a single left shift, don't change r and s. If there were two or more left shifts, $r := 0$, $s := 0$. (In the last case, two or more shifts can only happen when a_1 and a_2 have opposite signs and the same exponent, in which case the computation $s_1 + s_2$ in step 4 will be exact.)

7. Round S using Figure J.11; namely, if a table entry is nonempty, add 1 to the low-order bit of S. If rounding causes carry-out, shift S right and adjust the exponent. This is the significand of the result.

8. Compute the sign of the result. If a_1 and a_2 have the same sign, this is the sign of the result. If a_1 and a_2 have different signs, then the sign of the result depends on which of a_1 or a_2 is negative, whether there was a swap in step 1, and whether S was replaced by its two's complement in step 4. See Figure J.12.

swap	compl	sign(a_1)	sign(a_2)	sign(result)
Yes		+	−	−
Yes		−	+	+
No	No	+	−	+
No	No	−	+	−
No	Yes	+	−	−
No	Yes	−	+	+

Figure J.12 Rules for computing the sign of a sum when the addends have different signs. The *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are "don't care."

Example   Use the algorithm to compute the sum $(-1.001_2 \times 2^{-2}) + (-1.111_2 \times 2^0)$.

Answer $s_1 = 1.001$, $e_1 = -2$, $s_2 = 1.111$, $e_2 = 0$

1. $e_1 < e_2$, so swap. $d = 2$. Tentative exp = 0.
2. Signs of both operands negative, don't negate s_2.
3. Shift s_2 (1.001 after swap) right by 2, giving $s_2 = .010$, $g = 0$, $r = 1$, $s = 0$.
4.
```
      1.111
  +    .010
    (1)0.001        S = 0.001, with a carry-out.
```
5. Carry-out, so shift S right, $S = 1.000$, exp = exp + 1, so exp = 1.
6. r = low-order bit of sum = 1, $s = g \vee r \vee s = 0 \vee 1 \vee 0 = 1$.
7. r AND s = TRUE, so Figure J.11 says round up, $S = S + 1$ or $S = 1.001$.
8. Both signs negative, so sign of result is negative. Final answer: $-S \times 2^{\exp} = 1.001_2 \times 2^1$.

Example   Use the algorithm to compute the sum $(-1.010_2) + 1.100_2$.

Answer $s_1 = 1.010$, $e_1 = 0$, $s_2 = 1.100$, $e_2 = 0$

1. No swap, $d = 0$, tentative exp = 0.
2. Signs differ, replace s_2 with 0.100.
3. $d = 0$, so no shift. $r = g = s = 0$.
4.
```
      1.010
  +   0.100
      1.110        Signs are different, most-significant bit is 1, no carry-out, so
                   must two's complement sum, giving S = 0.010.
```

5. Shift left twice, so $S = 1.000$, exp = exp − 2, or exp = −2.

6. Two left shifts, so $r = g = s = 0$.

7. No addition required for rounding.

8. Answer is sign $\times S \times 2^{\text{exp}}$ or sign $\times 1.000 \times 2^{-2}$. Get sign from Figure J.12. Since complement but no swap and sign(a_1) is −, the sign of the sum is +. Thus, the answer = $1.000_2 \times 2^{-2}$.

Speeding Up Addition

Let's estimate how long it takes to perform the algorithm above. Step 2 may require an addition, step 4 requires one or two additions, and step 7 may require an addition. If it takes T time units to perform a p-bit add (where $p = 24$ for single precision, 53 for double), then it appears the algorithm will take at least $4T$ time units. But that is too pessimistic. If step 4 requires two adds, then a_1 and a_2 have the same exponent and different signs, but in that case the difference is exact, so no roundup is required in step 7. Thus, only three additions will ever occur. Similarly, it appears that a variable shift may be required both in step 3 and step 5. But if $|e_1 − e_2| \leq 1$, then step 3 requires a right shift of at most one place, so only step 5 needs a variable shift. And, if $|e_1 − e_2| > 1$, then step 3 needs a variable shift, but step 5 will require a left shift of at most one place. So only a single variable shift will be performed. Still, the algorithm requires three sequential adds, which, in the case of a 53-bit double-precision significand, can be rather time consuming.

A number of techniques can speed up addition. One is to use pipelining. The "Putting It All Together" section gives examples of how some commercial chips pipeline addition. Another method (used on the Intel 860 [Kohn and Fu 1989]) is to perform two additions in parallel. We now explain how this reduces the latency from $3T$ to T.

There are three cases to consider. First, suppose that both operands have the same sign. We want to combine the addition operations from steps 4 and 7. The position of the high-order bit of the sum is not known ahead of time, because the addition in step 4 may or may not cause a carry-out. Both possibilities are accounted for by having two adders. The first adder assumes the add in step 4 will not result in a carry-out. Thus, the values of r and s can be computed before the add is actually done. If r and s indicate that a roundup is necessary, the first adder will compute $S = s_1 + s_2 + 1$, where the notation +1 means adding 1 at the position of the least-significant bit of s_1. This can be done with a regular adder by setting the low-order carry-in bit to 1. If r and s indicate no roundup, the adder computes $S = s_1 + s_2$ as usual. One extra detail: When $r = 1$, $s = 0$, you will also need to know the low-order bit of the sum, which can also be computed in advance very quickly. The second adder covers the possibility that there will be carry-out. The values of r and s and the position where the roundup 1 is added are different from above, but again they can be quickly computed in advance. It is not known whether there will be a carry-out until after the add is actually done, but that doesn't matter. By doing both adds in parallel, one adder is guaranteed to reduce the correct answer.

The next case is when a_1 and a_2 have opposite signs but the same exponent. The sum $a_1 + a_2$ is exact in this case (no roundup is necessary) but the sign isn't known until the add is completed. So don't compute the two's complement (which requires an add) in step 2, but instead compute $\overline{s_1} + s_2 + 1$ and $s_1 + \overline{s_2} + 1$ in parallel. The first sum has the result of simultaneously complementing s_1 and computing the sum, resulting in $s_2 - s_1$. The second sum computes $s_1 - s_2$. One of these will be nonnegative and hence the correct final answer. Once again, all the additions are done in one step using two adders operating in parallel.

The last case, when a_1 and a_2 have opposite signs and different exponents, is more complex. If $|e_1 - e_2| > 1$, the location of the leading bit of the difference is in one of two locations, so there are two cases just as in addition. When $|e_1 - e_2| = 1$, cancellation is possible and the leading bit could be almost anywhere. However, only if the leading bit of the difference is in the same position as the leading bit of s_1 could a roundup be necessary. So one adder assumes a roundup, and the other assumes no roundup. Thus, the addition of step 4 and the rounding of step 7 can be combined. However, there is still the problem of the addition in step 2!

To eliminate this addition, consider the following diagram of step 4:

$$|\!\!-\!\!-\!\!-\, p \,-\!\!-\!\!-\!\!|$$
$$s_1 \quad 1.xxxxxxx$$
$$s_2 - \qquad\qquad 1\underline{yyzzzzz}$$

If the bits marked z are all 0, then the high-order p bits of $S = s_1 - s_2$ can be computed as $s_1 + \overline{s_2} + 1$. If at least one of the z bits is 1, use $s_1 + \overline{s_2}$. So $s_1 - s_2$ can be computed with one addition. However, we still don't know g and r for the two's complement of s_2, which are needed for rounding in step 7.

To compute $s_1 - s_2$ and get the proper g and r bits, combine steps 2 and 4 as follows. Don't complement s_2 in step 2. Extend the adder used for computing S two bits to the right (call the extended sum S'). If the preliminary sticky bit (computed in step 3) is 1, compute $S' = s_1' + \overline{s_2'}$, where s_1' has two 0 bits tacked onto the right, and s_2' has preliminary g and r appended. If the sticky bit is 0, compute $s_1' + \overline{s_2'} + 1$. Now the two low-order bits of S' have the correct values of g and r (the sticky bit was already computed properly in step 3). Finally, this modification can be combined with the modification that combines the addition from steps 4 and 7 to provide the final result in time T, the time for one addition.

A few more details need to be considered, as discussed in Santoro, Bewick, and Horowitz [1989] and Exercise J.17. Although the Santoro paper is aimed at multiplication, much of the discussion applies to addition as well. Also relevant is Exercise J.19, which contains an alternative method for adding signed magnitude numbers.

Denormalized Numbers

Unlike multiplication, for addition very little changes in the preceding description if one of the inputs is a denormal number. There must be a test to see if the exponent field is 0. If it is, then when unpacking the significand there will not be

a leading 1. By setting the biased exponent to 1 when unpacking a denormal, the algorithm works unchanged.

To deal with denormalized outputs, step 5 must be modified slightly. Shift S until it is normalized, or until the exponent becomes E_{min} (that is, the biased exponent becomes 1). If the exponent is E_{min} and, after rounding, the high-order bit of S is 1, then the result is a normalized number and should be packed in the usual way, by omitting the 1. If, on the other hand, the high-order bit is 0, the result is denormal. When the result is unpacked, the exponent field must be set to 0. Section J.7 discusses the exact rules for detecting underflow.

Incidentally, detecting overflow is very easy. It can only happen if step 5 involves a shift right and the biased exponent at that point is bumped up to 255 in single precision (or 2047 for double precision), or if this occurs after rounding.

Division and Remainder

In this section, we'll discuss floating-point division and remainder.

Iterative Division

We earlier discussed an algorithm for integer division. Converting it into a floating-point division algorithm is similar to converting the integer multiplication algorithm into floating point. The formula

$$\left(s_1 \times 2^{e_1}\right) / \left(s_2 \times 2^{e_2}\right) = (s_1 / s_2) \times 2^{e_1 - e_2}$$

shows that if the divider computes s_1/s_2, then the final answer will be this quotient multiplied by $2^{e_1-e_2}$. Referring to Figure J.2(b) (page J-4), the alignment of operands is slightly different from integer division. Load s_2 into B and s_1 into P. The A register is not needed to hold the operands. Then the integer algorithm for division (with the one small change of skipping the very first left shift) can be used, and the result will be of the form $q_0.q_1\cdots$. To round, simply compute two additional quotient bits (guard and round) and use the remainder as the sticky bit. The guard digit is necessary because the first quotient bit might be 0. However, since the numerator and denominator are both normalized, it is not possible for the two most-significant quotient bits to be 0. This algorithm produces one quotient bit in each step.

A different approach to division converges to the quotient at a quadratic rather than a linear rate. An actual machine that uses this algorithm will be discussed in Section J.10. First, we will describe the two main iterative algorithms, and then we will discuss the pros and cons of iteration when compared with the direct algorithms. A general technique for constructing iterative algorithms, called *Newton's iteration*, is shown in Figure J.13. First, cast the problem in the form of finding the zero of a function. Then, starting from a guess for the zero, approximate the function by its tangent at that guess and form a new guess based
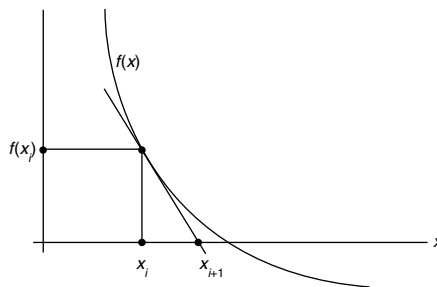
Figure J.13 Newton's iteration for zero finding. If x_i is an estimate for a zero of f, then x_{i+1} is a better estimate. To compute x_{i+1}, find the intersection of the x-axis with the tangent line to f at $f(x_i)$.

on where the tangent has a zero. If x_i is a guess at a zero, then the tangent line has the equation:

$$y - f(x_i) = f'(x_i)(x - x_i)$$

This equation has a zero at

J.6.1
$$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

To recast division as finding the zero of a function, consider $f(x) = x^{-1} - b$. Since the zero of this function is at $1/b$, applying Newton's iteration to it will give an iterative method of computing $1/b$ from b. Using $f'(x) = -1/x^2$, Equation J.6.1 becomes:

J.6.2
$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2\, b = x_i(2 - x_i b)$$

Thus, we could implement computation of a/b using the following method:

1. Scale b to lie in the range $1 \le b < 2$ and get an approximate value of $1/b$ (call it x_0) using a table lookup.

2. Iterate $x_{i+1} = x_i(2 - x_i b)$ until reaching an x_n that is accurate enough.

3. Compute $a x_n$ and reverse the scaling done in step 1.

Here are some more details. How many times will step 2 have to be iterated? To say that x_i is accurate to p bits means that $\left| (x_i - 1/b)/(1/b) \right| = 2^{-p}$, and a simple algebraic manipulation shows that when this is so, then $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Thus, the number of correct bits doubles at each step. Newton's iteration is *self-correcting* in the sense that making an error in x_i doesn't really matter. That is, it treats x_i as a guess at $1/b$ and returns x_{i+1} as an improvement on it (roughly doubling the digits). One thing that would cause x_i to be in error is rounding error.

More importantly, however, in the early iterations we can take advantage of the fact that we don't expect many correct bits by performing the multiplication in reduced precision, thus gaining speed without sacrificing accuracy. Another application of Newton's iteration is discussed in Exercise J.20.

The second iterative division method is sometimes called *Goldschmidt's algorithm*. It is based on the idea that to compute a/b, you should multiply the numerator and denominator by a number r with $rb \approx 1$. In more detail, let $x_0 = a$ and $y_0 = b$. At each step compute $x_{i+1} = r_i x_i$ and $y_{i+1} = r_i y_i$. Then the quotient $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ is constant. If we pick r_i so that $y_i \to 1$, then $x_i \to a/b$, so the x_i converge to the answer we want. This same idea can be used to compute other functions. For example, to compute the square root of a, let $x_0 = a$ and $y_0 = a$, and at each step compute $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Then $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, so if the r_i are chosen to drive $x_i \to 1$, then $y_i \to \sqrt{a}$. This technique is used to compute square roots on the TI 8847.

Returning to Goldschmidt's division algorithm, set $x_0 = a$ and $y_0 = b$, and write $b = 1 - \delta$, where $|\delta| < 1$. If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$. We next pick $r_1 = 1 + \delta^2$, so that $y_2 = r_1 y_1 = 1 - \delta^4$, and so on. Since $|\delta| < 1$, $y_i \to 1$. With this choice of r_i, the x_i will be computed as $x_{i+1} = r_i x_i = (1 + \delta^{2^i})x_i = (1 + (1 - b)^{2^i})x_i$, or

J.6.3
$$x_{i+1} = a \left[1 + (1 - b)\right] \left[1 + (1 - b)^2\right] \left[1 + (1 - b)^4\right] \cdots \left[1 + (1 - b)^{2^i}\right]$$

There appear to be two problems with this algorithm. First, convergence is slow when b is not near 1 (that is, δ is not near 0), and, second, the formula isn't self-correcting—since the quotient is being computed as a product of independent terms, an error in one of them won't get corrected. To deal with slow convergence, if you want to compute a/b, look up an approximate inverse to b (call it b'), and run the algorithm on ab'/bb'. This will converge rapidly since $bb' \approx 1$.

To deal with the self-correction problem, the computation should be run with a few bits of extra precision to compensate for rounding errors. However, Goldschmidt's algorithm does have a weak form of self-correction, in that the precise value of the r_i does not matter. Thus, in the first few iterations, when the full precision of $1 - \delta^{2^i}$ is not needed you can choose r_i to be a truncation of $1 + \delta^{2^i}$, which may make these iterations run faster without affecting the speed of convergence. If r_i is truncated, then y_i is no longer exactly $1 - \delta^{2^i}$. Thus, Equation J.6.3 can no longer be used, but it is easy to organize the computation so that it does not depend on the precise value of r_i. With these changes, Goldschmidt's algorithm is as follows (the notes in brackets show the connection with our earlier formulas).

1. Scale a and b so that $1 \le b < 2$.

2. Look up an approximation to $1/b$ (call it b') in a table.

3. Set $x_0 = ab'$ and $y_0 = bb'$.

4. Iterate until x_i is close enough to a/b:

Loop

$$r \approx 2 - y \qquad \text{[if } y_i = 1 + \delta_i, \text{ then } r \approx 1 - \delta_i]$$

$$y = y \times r \qquad [y_{i+1} = y_i \times r \approx 1 - \delta_i^2]$$

$$x_{i+1} = x_i \times r \quad [x_{i+1} = x_i \times r]$$

End loop

The two iteration methods are related. Suppose in Newton's method that we unroll the iteration and compute each term x_{i+1} directly in terms of b, instead of recursively in terms of x_i. By carrying out this calculation (see Exercise J.22), we discover that

$$x_{i+1} = x_0(2 - x_0 b) \, [(1 + (x_0 b - 1)^2] \, [1 + (x_0 b - 1)^4] \, ... \, [1 + (x_0 b - 1)^{2^i}]$$

This formula is very similar to Equation J.6.3. In fact, they are identical if a and b in J.6.3 are replaced with ax_0, bx_0, and $a = 1$. Thus, if the iterations were done to infinite precision, the two methods would yield exactly the same sequence x_i.

The advantage of iteration is that it doesn't require special divide hardware. Instead, it can use the multiplier (which, however, requires extra control). Further, on each step, it delivers twice as many digits as in the previous step—unlike ordinary division, which produces a fixed number of digits at every step.

There are two disadvantages with inverting by iteration. The first is that the IEEE standard requires division to be correctly rounded, but iteration only delivers a result that is close to the correctly rounded answer. In the case of Newton's iteration, which computes $1/b$ instead of a/b directly, there is an additional problem. Even if $1/b$ were correctly rounded, there is no guarantee that a/b will be. An example in decimal with $p = 2$ is $a = 13$, $b = 51$. Then $a/b = .2549. . .$, which rounds to .25. But $1/b = .0196. . .$, which rounds to .020, and then $a \times .020 = .26$, which is off by 1. The second disadvantage is that iteration does not give a remainder. This is especially troublesome if the floating-point divide hardware is being used to perform integer division, since a remainder operation is present in almost every high-level language.

Traditional folklore has held that the way to get a correctly rounded result from iteration is to compute $1/b$ to slightly more than $2p$ bits, compute a/b to slightly more than $2p$ bits, and then round to p bits. However, there is a faster way, which apparently was first implemented on the TI 8847. In this method, a/b is computed to about 6 extra bits of precision, giving a preliminary quotient q. By comparing qb with a (again with only 6 extra bits), it is possible to quickly decide whether q is correctly rounded or whether it needs to be bumped up or down by 1 in the least-significant place. This algorithm is explored further in Exercise J.21.

One factor to take into account when deciding on division algorithms is the relative speed of division and multiplication. Since division is more complex than multiplication, it will run more slowly. A common rule of thumb is that division algorithms should try to achieve a speed that is about one-third that of

multiplication. One argument in favor of this rule is that there are real pro-
grams (such as some versions of spice) where the ratio of division to multi-
plication is 1:3. Another place where a factor of 3 arises is in the standard
iterative method for computing square root. This method involves one division
per iteration, but it can be replaced by one using three multiplications. This is
discussed in Exercise J.20.

Floating-Point Remainder

For nonnegative integers, integer division and remainder satisfy:

$$a = (a \text{ DIV } b)b + a \text{ REM } b, \; 0 \le a \text{ REM } b < b$$

A floating-point remainder x REM y can be similarly defined as $x = \text{INT}(x/y)y + x$
REM y. How should x/y be converted to an integer? The IEEE remainder function
uses the round-to-even rule. That is, pick $n = \text{INT } (x/y)$ so that $|x/y - n| \le 1/2$.
If two different n satisfy this relation, pick the even one. Then REM is defined
to be $x - yn$. Unlike integers where $0 \le a \text{ REM } b < b$, for floating-point numbers
$|x \text{ REM } y| \le y/2$. Although this defines REM precisely, it is not a practical opera-
tional definition, because n can be huge. In single precision, n could be as large as
$2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

There is a natural way to compute REM if a direct division algorithm is used.
Proceed as if you were computing x/y. If $x = s_1 2^{e_1}$ and $y = s_2 2^{e_2}$ and the divider is
as in Figure J.2(b) (page J-4), then load s_1 into P and s_2 into B. After $e_1 - e_2$
division steps, the P register will hold a number r of the form $x - yn$ satisfying
$0 \le r < y$. Since the IEEE remainder satisfies $|\text{REM}| \le y/2$, REM is equal to either
r or $r - y$. It is only necessary to keep track of the last quotient bit produced,
which is needed to resolve halfway cases. Unfortunately, $e_1 - e_2$ can be a lot of
steps, and floating-point units typically have a maximum amount of time they are
allowed to spend on one instruction. Thus, it is usually not possible to implement
REM directly. None of the chips discussed in Section J.10 implements REM, but
they could by providing a remainder-step instruction—this is what is done on the
Intel 8087 family. A remainder step takes as arguments two numbers x and $y,$ and
performs divide steps until either the remainder is in P or n steps have been per-
formed, where n is a small number, such as the number of steps required for divi-
sion in the highest-supported precision. Then REM can be implemented as a
software routine that calls the REM step instruction $\lfloor (e_1 - e_2)/n \rfloor$ times, initially
using x as the numerator but then replacing it with the remainder from the previ-
ous REM step.

REM can be used for computing trigonometric functions. To simplify things,
imagine that we are working in base 10 with five significant figures, and consider
computing sin x. Suppose that $x = 7$. Then we can reduce by $\pi = 3.1416$ and com-
pute sin(7) = sin(7 − 2 × 3.1416) = sin(0.7168) instead. But, suppose we want to
compute sin(2.0×10^5). Then $2 \times 10^5/3.1416 = 63661.8$, which in our five-place
system comes out to be 63662. Since multiplying 3.1416 times 63662 gives
200000.5392, which rounds to 2.0000×10^5, argument reduction reduces 2×10^5
to 0, which is not even close to being correct. The problem is that our five-place

system does not have the precision to do correct argument reduction. Suppose we had the REM operator. Then we could compute 2×10^5 REM 3.1416 and get $-.53920$. However, this is still not correct because we used 3.1416, which is an approximation for π. The value of 2×10^5 REM π is $-.071513$.

Traditionally, there have been two approaches to computing periodic functions with large arguments. The first is to return an error for their value when x is large. The second is to store π to a very large number of places and do exact argument reduction. The REM operator is not much help in either of these situations. There is a third approach that has been used in some math libraries, such as the Berkeley UNIX 4.3bsd release. In these libraries, π is computed to the nearest floating-point number. Let's call this machine π, and denote it by π'. Then, when computing sin x, reduce x using x REM π'. As we saw in the above example, x REM π' is quite different from x REM π when x is large, so that computing sin$x$ as $\sin(x$ REM $\pi')$ will not give the exact value of sin x. However, computing trigonometric functions in this fashion has the property that all familiar identities (such as $\sin^2 x + \cos^2 x = 1$) are true to within a few rounding errors. Thus, using REM together with machine π provides a simple method of computing trigonometric functions that is accurate for small arguments and still may be useful for large arguments.

When REM is used for argument reduction, it is very handy if it also returns the low-order bits of n (where x REM $y = x - ny$). This is because a practical implementation of trigonometric functions will reduce by something smaller than 2π. For example, it might use $\pi/2$, exploiting identities such as $\sin(x - \pi/2)$ $= -\cos x$, $\sin(x - \pi) = -\sin x$. Then the low bits of n are needed to choose the correct identity.

J.7   More on Floating-Point Arithmetic

Before leaving the subject of floating-point arithmetic, we present a few additional topics.

Fused Multiply-Add

Probably the most common use of floating-point units is performing matrix operations, and the most frequent matrix operation is multiplying a matrix times a matrix (or vector), which boils down to computing an inner product, $x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n$. Computing this requires a series of multiply-add combinations.

Motivated by this, the IBM RS/6000 introduced a single instruction that computes $ab + c$, the *fused multiply-add*. Although this requires being able to read three operands in a single instruction, it has the potential for improving the performance of computing inner products.

The fused multiply-add computes $ab + c$ exactly and then rounds. Although rounding only once increases the accuracy of inner products somewhat, that is not its primary motivation. There are two main advantages of rounding once.

First, as we saw in the previous sections, rounding is expensive to implement because it may require an addition. By rounding only once, an addition operation has been eliminated. Second, the extra accuracy of fused multiply-add can be used to compute correctly rounded division and square root when these are not available directly in hardware. Fused multiply-add can also be used to implement efficient floating-point multiple-precision packages.

The implementation of correctly rounded division using fused multiply-add has many details, but the main idea is simple. Consider again the example from Section J.6 (page J-30), which was computing a/b with $a = 13$, $b = 51$. Then $1/b$ rounds to $b' = .020$, and ab' rounds to $q' = .26$, which is not the correctly rounded quotient. Applying fused multiply-add twice will correctly adjust the result, via the formulas

$$r = a - bq'$$
$$q'' = q' + rb'$$

Computing to two-digit accuracy, $bq' = 51 \times .26$ rounds to 13, and so $r = a - bq'$ would be 0, giving no adjustment. But using fused multiply-add gives $r = a - bq' = 13 - (51 \times .26) = -.26$, and then $q'' = q' + rb' = .26 - .0052 = .2548$, which rounds to the correct quotient, .25. More details can be found in the papers by Montoye, Hokenek, and Runyon [1990] and Markstein [1990].

Precisions

The standard specifies four precisions: *single*, *single extended*, *double*, and *double extended*. The properties of these precisions are summarized in Figure J.7 (page J-16). Implementations are not required to have all four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented. Thus, the computer designer usually only has to decide whether to support double extended and, if so, how many bits it should have.

The Motorola 68882 and Intel 387 coprocessors implement extended precision using the smallest allowable size of 80 bits (64 bits of significand). However, many of the more recently designed, high-performance floating-point chips do not implement 80-bit extended precision. One reason is that the 80-bit width of extended precision is awkward for 64-bit buses and registers. Some new architectures, such as SPARC V8 and PA-RISC, specify a 128-bit extended (or *quad*) precision. They have established a *de facto* convention for quad that has 15 bits of exponent and 113 bits of significand.

Although most high-level languages do not provide access to extended precision, it is very useful to writers of mathematical software. As an example, consider writing a library routine to compute the length of a vector (x,y) in the plane, namely, $\sqrt{x^2 + y^2}$. If x is larger than $2^{E_{max}/2}$, then computing this in the obvious way will overflow. This means that either the allowable exponent

range for this subroutine will be cut in half or a more complex algorithm using scaling will have to be employed. But, if extended precision is available, then the simple algorithm will work. Computing the length of a vector is a simple task, and it is not difficult to come up with an algorithm that doesn't overflow. However, there are more complex problems for which extended precision means the difference between a simple, fast algorithm and a much more complex one. One of the best examples of this is binary-to-decimal conversion. An efficient algorithm for binary-to-decimal conversion that makes essential use of extended precision is very readably presented in Coonen [1984]. This algorithm is also briefly sketched in Goldberg [1991]. Computing accurate values for transcendental functions is another example of a problem that is made much easier if extended precision is present.

One very important fact about precision concerns *double rounding*. To illustrate in decimals, suppose that we want to compute 1.9×0.66 and that single precision is two digits, while extended precision is three digits. The exact result of the product is 1.254. Rounded to extended precision, the result is 1.25. When further rounded to single precision, we get 1.2. However, the result of 1.9×0.66 correctly rounded to single precision is 1.3. Thus, rounding twice may not produce the same result as rounding once. Suppose you want to build hardware that only does double-precision arithmetic. Can you simulate single precision by computing first in double precision and then rounding to single? The above example suggests that you can't. However, double rounding is not always dangerous. In fact, the following rule is true (this is not easy to prove, but see Exercise J.25).

> *If* x *and* y *have* p-*bit significands, and* x + y *is computed exactly and then rounded to* q *places, a second rounding to* p *places will not change the answer if* q ≥ 2p + 2. *This is true not only for addition, but also for multiplication, division, and square root.*

In our example above, $q = 3$ and $p = 2$, so $q \check{S} 2p + 2$ is not true. On the other hand, for IEEE arithmetic, double precision has $q = 53$ and $p = 24$, so $q = 53 \check{S} 2p + 2 = 50$. Thus, single precision can be implemented by computing in double precision—that is, computing the answer exactly and then rounding to double—and then rounding to single precision.

Exceptions

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact, and invalid. By default, when these exceptions occur, they merely set a flag and the computation continues. The flags are *sticky*, meaning that once set they remain set until explicitly cleared. The standard strongly encourages implementations to provide a trap-enable bit for each exception. When an exception with an enabled trap handler occurs, a user trap handler is called, and the value of the associated exception flag is undefined. In Section J.3 we mentioned that $\sqrt{-3}$ has the value NaN and 1/0 is ∞. These are examples of operations that raise an exception. By default, computing $\sqrt{-3}$ sets the invalid flag and returns the value NaN. Similarly 1/0 sets the divide-by-zero flag and returns ∞.

The underflow, overflow, and divide-by-zero exceptions are found in most other systems. The *invalid exception* is for the result of operations such as $\sqrt{-1}$, 0/0, or $\infty - \infty$, which don't have any natural value as a floating-point number or as $\pm\infty$. The *inexact exception* is peculiar to IEEE arithmetic and occurs either when the result of an operation must be rounded or when it overflows. In fact, since 1/0 and an operation that overflows both deliver ∞, the exception flags must be consulted to distinguish between them. The inexact exception is an unusual "exception," in that it is not really an exceptional condition because it occurs so frequently. Thus, enabling a trap handler for the inexact exception will most likely have a severe impact on performance. Enabling a trap handler doesn't affect whether an operation is exceptional except in the case of underflow. This is discussed below.

The IEEE standard assumes that when a trap occurs, it is possible to identify the operation that trapped and its operands. On machines with pipelining or multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support may be necessary to identify exactly which operation trapped.

Another problem is illustrated by the following program fragment.

```
r1 = r2/r3
r2 = r4 + r5
```

These two instructions might well be executed in parallel. If the divide traps, its argument r2 could already have been overwritten by the addition, especially since addition is almost always faster than division. Computer systems that support trapping in the IEEE standard must provide some way to save the value of r2, either in hardware or by having the compiler avoid such a situation in the first place. This kind of problem is not peculiar to floating point. In the sequence

```
r1 = 0(r2)
r2 = r3
```

it would be efficient to execute r2 = r3 while waiting for memory. But, if accessing 0(r2) causes a page fault, r2 might no longer be available for restarting the instruction r1 = 0(r2).

One approach to this problem, used in the MIPS R3010, is to identify instructions that may cause an exception early in the instruction cycle. For example, an addition can overflow only if one of the operands has an exponent of E_{max}, and so on. This early check is conservative: It might flag an operation that doesn't actually cause an exception. However, if such false positives are rare, then this technique will have excellent performance. When an instruction is tagged as being possibly exceptional, special code in a trap handler can compute it without destroying any state. Remember that all these problems occur only when trap handlers are enabled. Otherwise, setting the exception flags during normal processing is straightforward.

Underflow

We have alluded several times to the fact that detection of underflow is more complex than for the other exceptions. The IEEE standard specifies that if an underflow trap handler is enabled, the system must trap if the result is denormal. On the other hand, if trap handlers are disabled, then the underflow flag is set only if there is a loss of accuracy—that is, if the result must be rounded. The rationale is, if no accuracy is lost on an underflow, there is no point in setting a warning flag. But if a trap handler is enabled, the user might be trying to simulate flush-to-zero and should therefore be notified whenever a result dips below $1.0 \times 2^{E_{\min}}$.

So if there is no trap handler, the underflow exception is signaled only when the result is denormal and inexact, but the definitions of *denormal* and *inexact* are both subject to multiple interpretations. Normally, inexact means there was a result that couldn't be represented exactly and had to be rounded. Consider the example (in a base 2 floating-point system with 3-bit significands) of $(1.11_2 \times 2^{-2})$ $\times (1.11_2 \times 2^{E_{\min}}) = 0.110001_2 \times 2^{E_{\min}}$, with round to nearest in effect. The delivered result is $0.11_2 \times 2^{E_{\min}}$, which had to be rounded, causing inexact to be signaled. But is it correct to also signal underflow? Gradual underflow loses significance because the exponent range is bounded. If the exponent range were unbounded, the delivered result would be $1.10_2 \times 2^{E_{\min}-1}$, exactly the same answer obtained with gradual underflow. The fact that denormalized numbers have fewer bits in their significand than normalized numbers therefore doesn't make any difference in this case. The commentary to the standard [Cody et al. 1984] encourages this as the criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same fraction size, but with a very large exponent range. However, owing to the difficulty of implementing this scheme, the standard allows setting the underflow flag whenever the result is denormal and different from the infinitely precise result.

There are two possible definitions of what it means for a result to be denormal. Consider the example of $1.10_2 \times 2^{-1}$ multiplied by $1.01_2 \times 2^{E_{\min}}$. The exact product is $0.1111 \times 2^{E_{\min}}$. The rounded result is the normal number $1.00_2 \times 2^{E_{\min}}$. Should underflow be signaled? Signaling underflow means that you are using the *before rounding* rule, because the result was denormal before rounding. Not signaling underflow means that you are using the *after rounding* rule, because the result is normalized after rounding. The IEEE standard provides for choosing either rule; however, the one chosen must be used consistently for all operations.

To illustrate these rules, consider floating-point addition. When the result of an addition (or subtraction) is denormal, it is always exact. Thus, the underflow flag never needs to be set for addition. That's because if traps are not enabled then no exception is raised. And if traps are enabled, the value of the underflow flag is undefined, so again it doesn't need to be set.

One final subtlety should be mentioned concerning underflow. When there is no underflow trap handler, the result of an operation on p-bit numbers that causes

an underflow is a denormal number with $p - 1$ or fewer bits of precision. When traps are enabled, the trap handler is provided with the result of the operation rounded to p bits and with the exponent wrapped around. Now there is a potential double-rounding problem. If the trap handler wants to return the denormal result, it can't just round its argument, because that might lead to a double-rounding error. Thus, the trap handler must be passed at least one extra bit of information if it is to be able to deliver the correctly rounded result.

J.8 Speeding Up Integer Addition

The previous section showed that many steps go into implementing floating-point operations; however, each floating-point operation eventually reduces to an integer operation. Thus, increasing the speed of integer operations will also lead to faster floating point.

Integer addition is the simplest operation and the most important. Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to calculate addresses. Despite the simplicity of addition, there isn't a single best way to perform high-speed addition. We will discuss three techniques that are in current use: carry-lookahead, carry-skip, and carry-select.

Carry-Lookahead

An n-bit adder is just a combinational circuit. It can therefore be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. How do you figure out what this circuit looks like? From Equation J.2.1 (page J-3) the formula for the ith sum can be written as:

J.8.1
$$s_i = a_i \overline{b_i} \, \overline{c_i} + \overline{a_i} b_i \overline{c_i} + \overline{a_i} \, \overline{b_i} c_i + a_i b_i c_i$$

where c_i is both the carry-in to the ith adder and the carry-out from the $(i-1)$-st adder.

The problem with this formula is that, although we know the values of a_i and b_i—they are inputs to the circuit—we don't know c_i. So our goal is to write c_i in terms of a_i and b_i. To accomplish this, we first rewrite Equation J.2.2 (page J-3) as:

J.8.2
$$c_i = g_{i-1} + p_{i-1} c_{i-1}, \quad g_{i-1} = a_{i-1} b_{i-1}, \quad p_{i-1} = a_{i-1} + b_{i-1}$$

Here is the reason for the symbols p and g: If g_{i-1} is true, then c_i is certainly true, so a carry is *generated*. Thus, g is for generate. If p_{i-1} is true, then if c_{i-1} is true, it is *propagated* to c_i. Start with Equation J.8.1 and use Equation J.8.2 to replace c_i with $g_{i-1} + p_{i-1} c_{i-1}$. Then, use Equation J.8.2 with $i-1$ in place of i to replace c_{i-1} with c_{i-2}, and so on. This gives the result:

J.8.3
$$c_i = g_{i-1} + p_{i-1} g_{i-2} + p_{i-1} p_{i-2} g_{i-3} + \cdots + p_{i-1} p_{i-2} \cdots p_1 g_0 + p_{i-1} p_{i-2} \cdots p_1 p_0 c_0$$
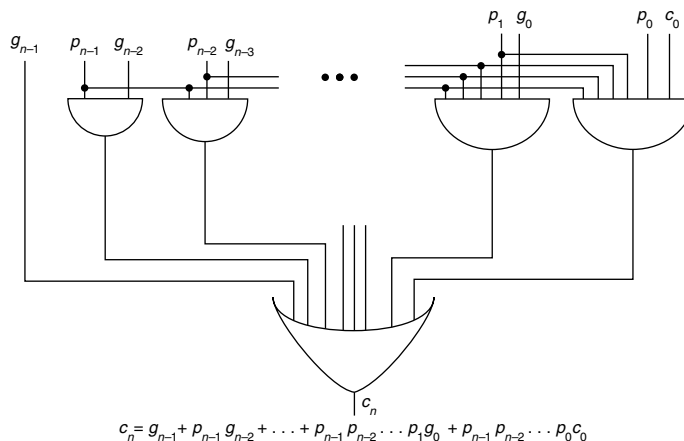
Figure J.14 Pure carry-lookahead circuit for computing the carry-out c_n of an n-bit adder.

An adder that computes carries using Equation J.8.3 is called a *carry-lookahead adder*, or CLA. A CLA requires one logic level to form p and g, two levels to form the carries, and two for the sum, for a grand total of five logic levels. This is a vast improvement over the $2n$ levels required for the ripple-carry adder.

Unfortunately, as is evident from Equation J.8.3 or from Figure J.14, a carry-lookahead adder on n bits requires a fan-in of $n + 1$ at the OR gate as well as at the rightmost AND gate. Also, the p_{n-1} signal must drive n AND gates. In addition, the rather irregular structure and many long wires of Figure J.14 make it impractical to build a full carry-lookahead adder when n is large.

However, we can use the carry-lookahead idea to build an adder that has about $\log_2 n$ logic levels (substantially fewer than the $2n$ required by a ripple-carry adder) and yet has a simple, regular structure. The idea is to build up the p's and g's in steps. We have already seen that

$$c_1 = g_0 + c_0 p_0$$

This says there is a carry-out of the 0th position (c_1) either if there is a carry generated in the 0th position or if there is a carry into the 0th position and the carry propagates. Similarly,

$$c_2 = G_{01} + P_{01} c_0$$

G_{01} means there is a carry generated out of the block consisting of the first two bits. P_{01} means that a carry propagates through this block. P and G have the following logic equations:

$$G_{01} = g_1 + p_1 g_0$$

$$P_{01} = p_1 p_0$$

More generally, for any j with $i < j$, $j + 1 < k$, we have the recursive relations:

J.8.4
$$c_{k+1} = G_{ik} + P_{ik} c_i$$

J.8.5
$$G_{ik} = G_{j+1,k} + P_{j+1,k} G_{ij}$$

J.8.6
$$P_{ik} = P_{ij} P_{j+1,k}$$

Equation J.8.5 says that a carry is generated out of the block consisting of bits i through k inclusive if it is generated in the high-order part of the block $(j + 1, k)$ or if it is generated in the low-order part of the block (i,j) and then propagated through the high part. These equations will also hold for $i \le j < k$ if we set $G_{ii} = g_i$ and $P_{ii} = p_i$.

Example   Express P_{03} and G_{03} in terms of p's and g's.

Answer   Using Equation J.8.6, $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$. Since $P_{ii} = p_i$, $P_{03} = p_0 p_1 p_2 p_3$. For G_{03}, Equation J.8.5 says $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$.

With these preliminaries out of the way, we can now show the design of a practical CLA. The adder consists of two parts. The first part computes various values of P and G from p_i and g_i, using Equations J.8.5 and J.8.6; the second part uses these P and G values to compute all the carries via Equation J.8.4. The first part of the design is shown in Figure J.15. At the top of the diagram, input numbers $a_7 \ldots a_0$ and $b_7 \ldots b_0$ are converted to p's and g's using cells of type 1. Then various P's and G's are generated by combining cells of type 2 in a binary tree structure. The second part of the design is shown in Figure J.16. By feeding c_0 in at the bottom of this tree, all the carry bits come out at the top. Each cell must know a pair of (P,G) values in order to do the conversion, and the value it needs is written inside the cells. Now compare Figures J.15 and J.16. There is a one-to-one correspondence between cells, and the value of (P,G) needed by the carry-generating cells is exactly the value known by the corresponding (P,G)-generating cells. The combined cell is shown in Figure J.17. The numbers to be added flow into the top and downward through the tree, combining with c_0 at the bottom and flowing back up the tree to form the carries. Note that one thing is missing from Figure J.17: a small piece of extra logic to compute c_8 for the carry-out of the adder.

The bits in a CLA must pass through about $\log_2 n$ logic levels, compared with $2n$ for a ripple-carry adder. This is a substantial speed improvement, especially for a large n. Whereas the ripple-carry adder had n cells, however, the CLA has
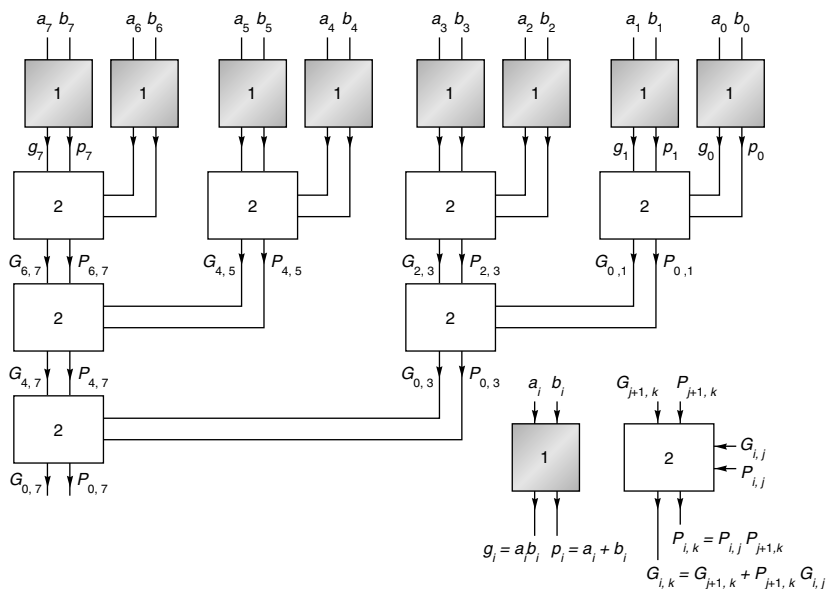
Figure J.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of P and G are computed.
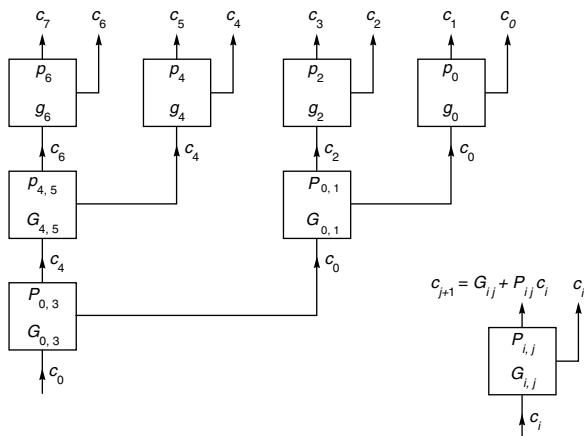

Figure J.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with P and G to form the carries.

$2n$ cells, although in our layout they will take $n \log n$ space. The point is that a small investment in size pays off in a dramatic improvement in speed.

A number of technology-dependent modifications can improve CLAs. For example, if each node of the tree has three inputs instead of two, then the height
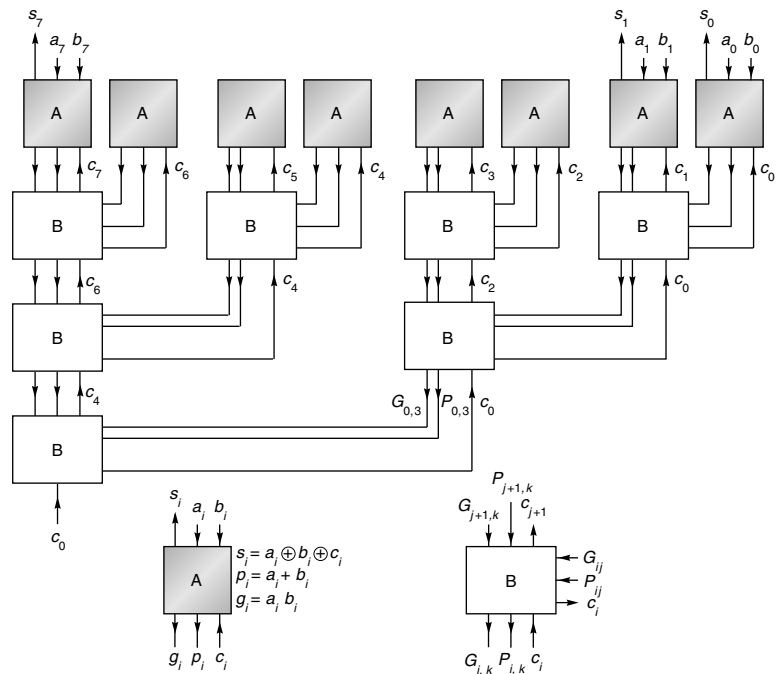
Figure J.17 Complete carry-lookahead tree adder. This is the combination of Figures J.15 and J.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

of the tree will decrease from $\log_2 n$ to $\log_3 n$. Of course, the cells will be more complex and thus might operate more slowly, negating the advantage of the decreased height. For technologies where rippling works well, a hybrid design might be better. This is illustrated in Figure J.19. Carries ripple between adders at the top level, while the "B" boxes are the same as those in Figure J.17. This design will be faster if the time to ripple between four adders is faster than the time it takes to traverse a level of "B" boxes. (To make the pattern more clear, Figure J.19 shows a 16-bit adder, so the 8-bit adder of Figure J.17 corresponds to the right half of Figure J.19.)

Carry-Skip Adders

A *carry-skip adder* sits midway between a ripple-carry adder and a carry-lookahead adder, both in terms of speed and cost. (A carry-skip adder is not called a CSA, as that name is reserved for carry-save adders.) The motivation for this adder comes from examining the equations for P and G. For example,

$$P_{03} = p_0 p_1 p_2 p_3$$

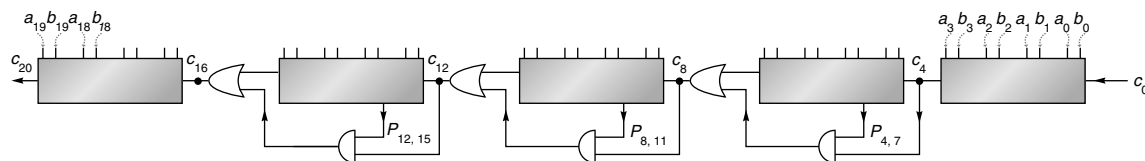$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Figure J.18 Carry-skip adder. This is a 20-bit carry-skip adder ($n = 20$) with each block 4 bits wide ($k = 4$).
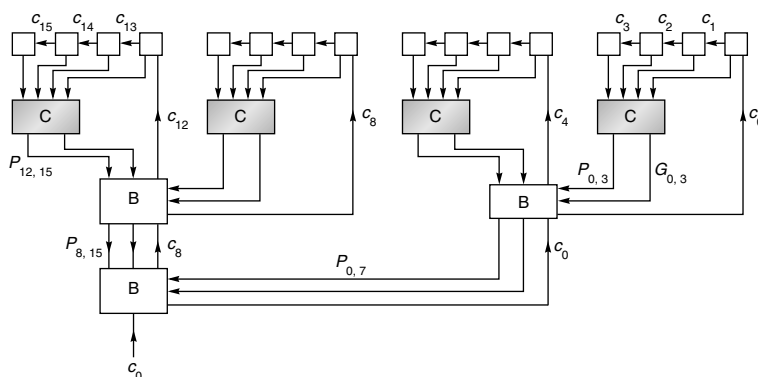

Figure J.19 Combination of CLA and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

Computing P is much simpler than computing G, and a carry-skip adder only computes the P's. Such an adder is illustrated in Figure J.18. Carries begin rippling simultaneously through each block. If any block generates a carry, then the carry-out of a block will be true, even though the carry-in to the block may not be correct yet. If at the start of each add operation the carry-in to each block is 0, then no spurious carry-outs will be generated. Thus, the carry-out of each block can be thought of as if it were the G signal. Once the carry-out from the least-significant block is generated, it not only feeds into the next block but is also fed through the AND gate with the P signal from that next block. If the carry-out and P signals are both true, then the carry *skips* the second block and is ready to feed into the third block, and so on. The carry-skip adder is only practical if the carry-in signals can be easily cleared at the start of each operation—for example, by precharging in CMOS.

To analyze the speed of a carry-skip adder, let's assume that it takes 1 time unit for a signal to pass through two logic levels. Then it will take k time units for a carry to ripple across a block of size k, and it will take 1 time unit for a carry to skip a block. The longest signal path in the carry-skip adder starts with a carry being generated at the 0th position. If the adder is n bits wide, then it takes k time units to ripple through the first block, $n/k - 2$ time units to skip blocks, and k more to ripple through the last block. To be specific: if we have a 20-bit adder

broken into groups of 4 bits, it will take $4 + (20/4 - 2) + 4 = 11$ time units to perform an add. Some experimentation reveals that there are more efficient ways to divide 20 bits into blocks. For example, consider five blocks with the least-significant 2 bits in the first block, the next 5 bits in the second block, followed by blocks of size 6, 5, and 2. Then the add time is reduced to 9 time units. This illustrates an important general principle. For a carry-skip adder, making the interior blocks larger will speed up the adder. In fact, the same idea of varying the block sizes can sometimes speed up other adder designs as well. Because of the large amount of rippling, a carry-skip adder is most appropriate for technologies where rippling is fast.

Carry-Select Adder

A *carry-select adder* works on the following principle: Two additions are performed in parallel, one assuming the carry-in is 0 and the other assuming the carry-in is 1. When the carry-in is finally known, the correct sum (which has been precomputed) is simply selected. An example of such a design is shown in Figure J.20. An 8-bit adder is divided into two halves, and the carry-out from the lower half is used to select the sum bits from the upper half. If each block is computing its sum using rippling (a linear time algorithm), then the design in Figure J.20 is twice as fast at 50% more cost. However, note that the c_4 signal must drive many muxes, which may be very slow in some technologies. Instead of dividing the adder into halves, it could be divided into quarters for a still further speedup. This is illustrated in Figure J.21. If it takes k time units for a block to add k-bit numbers, and if it takes 1 time unit to compute the mux input from the two carry-out signals, then for optimal operation each block should be 1 bit wider than the next, as shown in Figure J.21. Therefore, as in the carry-skip adder, the best design involves variable-size blocks.

As a summary of this section, the asymptotic time and space requirements for the different adders are given in Figure J.22. (The times for carry-skip and
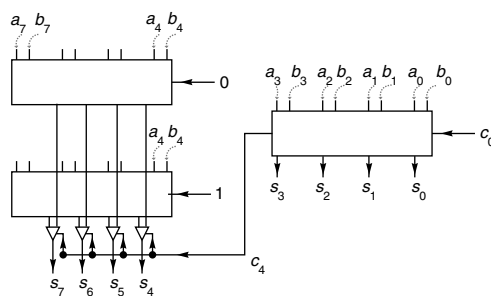

Figure J.20 Simple carry-select adder. At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.
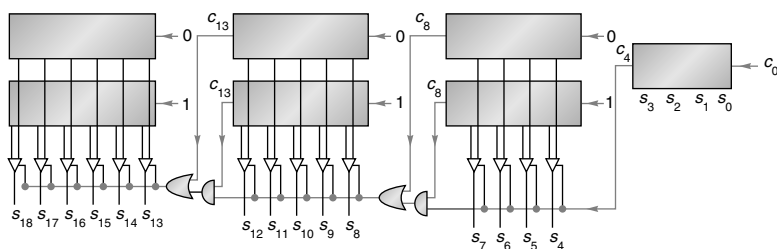
Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

carry-select come from a careful choice of block size. See Exercise J.26 for the carry-skip adder.) These different adders shouldn't be thought of as disjoint choices, but rather as building blocks to be used in constructing an adder. The utility of these different building blocks is highly dependent on the technology used. For example, the carry-select adder works well when a signal can drive many muxes, and the carry-skip adder is attractive in technologies where signals can be cleared at the start of each operation. Knowing the asymptotic behavior of adders is useful in understanding them, but relying too much on that behavior is a pitfall. The reason is that asymptotic behavior is only important as n grows very large. But n for an adder is the bits of precision, and double precision today is the same as it was 20 years ago—about 53 bits. Although it is true that as computers get faster, computations get longer—and thus have more rounding error, which in turn requires more precision—this effect grows very slowly with time.

J.9 Speeding Up Integer Multiplication and Division

The multiplication and division algorithms presented in Section J.2 are fairly slow, producing 1 bit per cycle (although that cycle might be a fraction of the CPU instruction cycle time). In this section, we discuss various techniques for higher-performance multiplication and division, including the division algorithm used in the Pentium chip.

Shifting over Zeros

Although the technique of shifting over zeros is not currently used much, it is instructive to consider. It is distinguished by the fact that its execution time is operand dependent. Its lack of use is primarily attributable to its failure to offer enough speedup over bit-at-a-time algorithms. In addition, pipelining, synchronization with the CPU, and good compiler optimization are difficult with algorithms that run in variable time. In multiplication, the idea behind shifting over zeros is to add logic that detects when the low-order bit of the A register is 0 (see Figure J.2(a) on page J-4) and, if so, skips the addition step and proceeds directly to the shift step—hence the term *shifting over zeros*.

What about shifting for division? In nonrestoring division, an ALU operation (either an addition or subtraction) is performed at every step. There appears to be no opportunity for skipping an operation. But think about division this way: To compute a/b, subtract multiples of b from a, and then report how many subtractions were done. At each stage of the subtraction process the remainder must fit into the P register of Figure J.2(b) (page J-4). In the case when the remainder is a small positive number, you normally subtract b; but suppose instead you only shifted the remainder and subtracted b the next time. As long as the remainder was sufficiently small (its high-order bit 0), after shifting it still would fit into the P register, and no information would be lost. However, this method does require changing the way we keep track of the number of times b has been subtracted from a. This idea usually goes under the name of *SRT division*, for Sweeney, Robertson, and Tocher, who independently proposed algorithms of this nature. The main extra complication of SRT division is that the quotient bits cannot be determined immediately from the sign of P at each step, as they can be in ordinary nonrestoring division.

More precisely, to divide a by b where a and b are n-bit numbers, load a and b into the A and B registers, respectively, of Figure J.2 (page J-4).

SRT Division

1. If B has k leading zeros when expressed using n bits, shift all the registers left k bits.

2. For $i = 0, n - 1$,

   (a) If the top three bits of P are equal, set $q_i = 0$ and shift (P,A) one bit left.

   (b) If the top three bits of P are not all equal and P is negative, set $q_i = -1$ (also written as $\bar{1}$), shift (P,A) one bit left, and add B.

   (c) Otherwise set $q_i = 1$, shift (P,A) one bit left, and subtract B.

   End loop

3. If the final remainder is negative, correct the remainder by adding B, and correct the quotient by subtracting 1 from q_0. Finally, the remainder must be shifted k bits right, where k is the initial shift.

```
    P       A
  00000    1000    Divide 8 = 1000 by 3 = 0011. B contains 0011.
  00010    0000    Step 1: B had two leading 0s, so shift left by 2. B now contains 1100.
                   Step 2.1: Top three bits are equal. This is case (a), so
  00100    0000       set q₀ = 0 and shift.
                   Step 2.2: Top three bits not equal and P ≥ 0 is case (c), so
  01000    0001       set q₁ = 1 and shift.
+ 10100                Subtract B.
  11100    0001    Step 2.3: Top bits equal is case (a), so
  11000    0010       set q₂ = 0 and shift.
                   Step 2.4: Top three bits unequal is case (b), so
  10000    0101̄      set q₃ = −1 and shift.
+ 01100                Add B.
  11100            Step 3. Remainder is negative so restore it and subtract 1 from q.
+ 01100
  01000            Must undo the shift in step 1, so right-shift by 2 to get true remainder.
                      Remainder = 10, quotient = 0101̄ − 1 = 0010.
```

Figure J.23 **SRT division of $1000_2/0011_2$.** The quotient bits are shown in bold, using the notation $\bar{1}$ for −1.

A numerical example is given in Figure J.23. Although we are discussing integer division, it helps in explaining the algorithm to imagine the binary point just left of the most-significant bit. This changes Figure J.23 from $01000_2/0011_2$ to $0.1000_2/.0011_2$. Since the binary point is changed in both the numerator and denominator, the quotient is not affected. The (P,A) register pair holds the remainder and is a two's complement number. For example, if P contains 11110_2 and A = 0, then the remainder is $1.1110_2 = -1/8$. If r is the value of the remainder, then $-1 \leq r < 1$.

Given these preliminaries, we can now analyze the SRT division algorithm. The first step of the algorithm shifts b so that $b \geq 1/2$. The rule for which ALU operation to perform is this: If $-1/4 \leq r < 1/4$ (true whenever the top three bits of P are equal), then compute $2r$ by shifting (P,A) left one bit; if $r < 0$ (and hence $r < -1/4$, since otherwise it would have been eliminated by the first condition), then compute $2r + b$ by shifting and then adding; if $r \geq 1/4$ and subtract b from $2r$. Using $b \geq 1/2$, it is easy to check that these rules keep $-1/2 \leq r < 1/2$. For nonrestoring division, we only have $|r| \leq b$, and we need P to be $n + 1$ bits wide. But, for SRT division, the bound on r is tighter, namely, $-1/2 \leq r < 1/2$. Thus, we can save a bit by eliminating the high-order bit of P (and b and the adder). In particular, the test for equality of the top three bits of P becomes a test on just two bits.

The algorithm might change slightly in an implementation of SRT division. After each ALU operation, the P register can be shifted as many places as necessary to make either $r \geq 1/4$ or $r < -1/4$. By shifting k places, k quotient bits are set equal to zero all at once. For this reason SRT division is sometimes described as one that keeps the remainder normalized to $|r| \geq 1/4$.

Notice that the value of the quotient bit computed in a given step is based on which operation is performed in that step (which in turn depends on the result of the operation from the previous step). This is in contrast to nonrestoring division,

where the quotient bit computed in the ith step depends on the result of the operation in the same step. This difference is reflected in the fact that when the final remainder is negative, the last quotient bit must be adjusted in SRT division, but not in nonrestoring division. However, the key fact about the quotient bits in SRT division is that they can include $\overline{1}$. Although Figure J.23 shows the quotient bits being stored in the low-order bits of A, an actual implementation can't do this because you can't fit the three values -1, 0, 1 into one bit. Furthermore, the quotient must be converted to ordinary two's complement in a full adder. A common way to do this is to accumulate the positive quotient bits in one register and the negative quotient bits in another, and then subtract the two registers after all the bits are known. Because there is more than one way to write a number in terms of the digits -1, 0, 1, SRT division is said to use a *redundant* quotient representation.

The differences between SRT division and ordinary nonrestoring division can be summarized as follows:

1. ALU decision rule—In nonrestoring division, it is determined by the sign of P; in SRT, it is determined by the two most-significant bits of P.

2. Final quotient—In nonrestoring division, it is immediate from the successive signs of P; in SRT, there are three quotient digits $(1, 0, \overline{1})$, and the final quotient must be computed in a full n-bit adder.

3. Speed—SRT division will be faster on operands that produce zero quotient bits.

The simple version of the SRT division algorithm given above does not offer enough of a speedup to be practical in most cases. However, later on in this section we will study variants of SRT division that are quite practical.

Speeding Up Multiplication with a Single Adder

As mentioned before, shifting-over-zero techniques are not used much in current hardware. We now discuss some methods that are in widespread use. Methods that increase the speed of multiplication can be divided into two classes: those that use a single adder and those that use multiple adders. Let's first discuss techniques that use a single adder.

In the discussion of addition we noted that, because of carry propagation, it is not practical to perform addition with two levels of logic. Using the cells of Figure J.17, adding two 64-bit numbers will require a trip through seven cells to compute the P's and G's and seven more to compute the carry bits, which will require at least 28 logic levels. In the simple multiplier of Figure J.2 on page J-4, each multiplication step passes through this adder. The amount of computation in each step can be dramatically reduced by using *carry-save adders* (CSAs). A carry-save adder is simply a collection of n independent full adders. A multiplier using such an adder is illustrated in Figure J.24. Each circle marked "+" is a single-bit full adder, and each box represents one bit of a register. Each addition operation results in a pair of bits, stored in the sum and carry parts of P. Since each
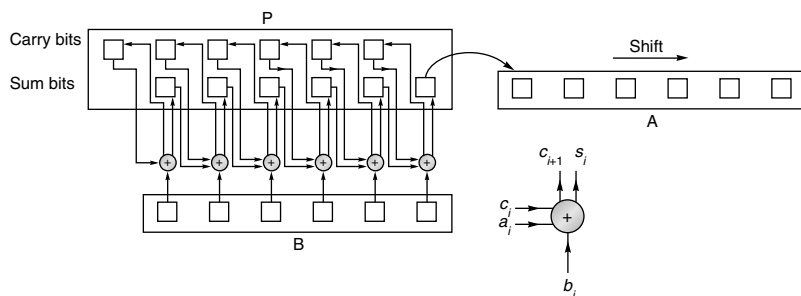
Figure J.24 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

add is independent, only two logic levels are involved in the add—a vast improvement over 28.

To operate the multiplier in Figure J.24, load the sum and carry bits of P with zero and perform the first ALU operation. (If Booth recoding is used, it might be a subtraction rather than an addition.) Then shift the low-order sum bit of P into A, as well as shifting A itself. The $n - 1$ high-order bits of P don't need to be shifted because on the next cycle the sum bits are fed into the next lower-order adder. Each addition step is substantially increased in speed, since each add cell is working independently of the others, and no carry is propagated.

There are two drawbacks to carry-save adders. First, they require more hardware because there must be a copy of register P to hold the carry outputs of the adder. Second, after the last step, the high-order word of the result must be fed into an ordinary adder to combine the sum and carry parts. One way to accomplish this is by feeding the output of P into the adder used to perform the addition operation. Multiplying with a carry-save adder is sometimes called *redundant multiplication* because P is represented using two registers. Since there are many ways to represent P as the sum of two registers, this representation is redundant. The term *carry-propagate adder* (CPA) is used to denote an adder that is not a CSA. A propagate adder may propagate its carries using ripples, carry-lookahead, or some other method.

Another way to speed up multiplication without using extra adders is to examine k low-order bits of A at each step, rather than just one bit. This is often called *higher-radix multiplication*. As an example, suppose that $k = 2$. If the pair of bits is 00, add 0 to P; if it is 01, add B. If it is 10, simply shift b one bit left before adding it to P. Unfortunately, if the pair is 11, it appears we would have to compute $b + 2b$. But this can be avoided by using a higher-radix version of Booth recoding. Imagine A as a base 4 number: When the digit 3 appears, change it to $\overline{1}$ and add 1 to the next higher digit to compensate. An extra benefit of using this

scheme is that just like ordinary Booth recoding, it works for negative as well as positive integers (Section J.2).

The precise rules for radix-4 Booth recoding are given in Figure J.25. At the ith multiply step, the two low-order bits of the A register contain a_{2i} and a_{2i+1}. These two bits, together with the bit just shifted out (a_{2i-1}), are used to select the multiple of b that must be added to the P register. A numerical example is given in Figure J.26. Another name for this multiplication technique is *overlapping triplets,* since it looks at 3 bits to determine what multiple of b to use, whereas ordinary Booth recoding looks at 2 bits.

Besides having more complex control logic, overlapping triplets also requires that the P register be 1 bit wider to accommodate the possibility of $2b$ or $-2b$ being added to it. It is possible to use a radix-8 (or even higher) version of Booth recoding. In that case, however, it would be necessary to use the multiple 3B as a potential summand. Radix-8 multipliers normally compute 3B once and for all at the beginning of a multiplication operation.

| Low-order bits of A		Last bit shifted out	
$2i + 1$	$2i$	$2i - 1$	Multiple
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

Figure J.25 Multiples of b to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is $-b$, obtained from the second-to-last row of the table.

```
  P       A    L
00000   1001         Multiply –7 = 1001 times –5 = 1011. B contains 1011.
+ 11011              Low-order bits of A are 0, 1; L = 0, so add B.
11011   1001
11110   1110  0      Shift right by two bits, shifting in 1s on the left.
+ 01010              Low-order bits of A are 1, 0; L = 0, so add –2b.
01000   1110  0
00010   0011  1      Shift right by two bits.
                     Product is 35 = 0100011.
```

Figure J.26 Multiplication of –7 times –5 using radix-4 Booth recoding. The column labeled L contains the last bit shifted out the right end of A.

Faster Multiplication with Many Adders

If the space for many adders is available, then multiplication speed can be improved. Figure J.27 shows a simple array multiplier for multiplying two 5-bit numbers, using three CSAs and one propagate adder. Part (a) is a block diagram of the kind we will use throughout this section. Parts (b) and (c) show the adder in more detail. All the inputs to the adder are shown in (b); the actual adders with
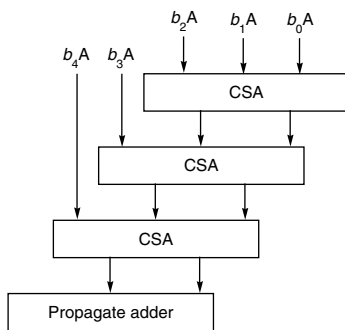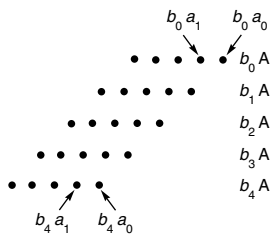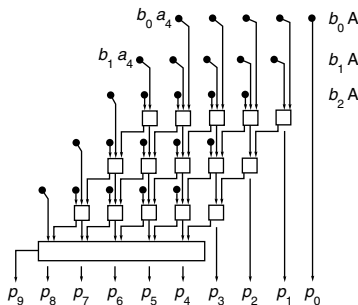
Figure J.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

their interconnections are shown in (c). Each row of adders in (c) corresponds to a box in (a). The picture is "twisted" so that bits of the same significance are in the same column. In an actual implementation, the array would most likely be laid out as a square instead.

The array multiplier in Figure J.27 performs the same number of additions as the design in Figure J.24, so its latency is not dramatically different from that of a single carry-save adder. However, with the hardware in Figure J.27, multiplication can be pipelined, increasing the total throughput. On the other hand, although this level of pipelining is sometimes used in array processors, it is not used in any of the single-chip, floating-point accelerators discussed in Section J.10. Pipelining is discussed in general in Appendix C and by Kogge [1981] in the context of multipliers.

Sometimes the space budgeted on a chip for arithmetic may not hold an array large enough to multiply two double-precision numbers. In this case, a popular design is to use a two-pass arrangement such as the one shown in Figure J.28. The first pass through the array "retires" 5 bits of B. Then the result of this first pass is fed back into the top to be combined with the next three summands. The result of this second pass is then fed into a CPA. This design, however, loses the ability to be pipelined.

If arrays require as many addition steps as the much cheaper arrangements in Figures J.2 and J.24, why are they so popular? First of all, using an array has a smaller latency than using a single adder—because the array is a combinational circuit, the signals flow through it directly without being clocked. Although the two-pass adder of Figure J.28 would normally still use a clock, the cycle time for passing through k arrays can be less than k times the clock that would be needed for designs like the ones in Figures J.2 or J.24. Second, the array is amenable to
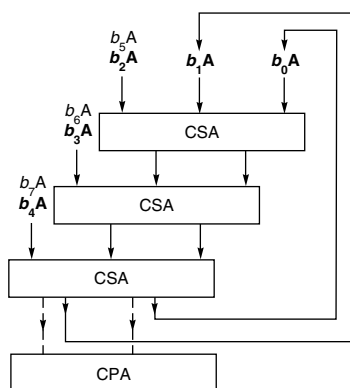

Figure J.28 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure J.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.
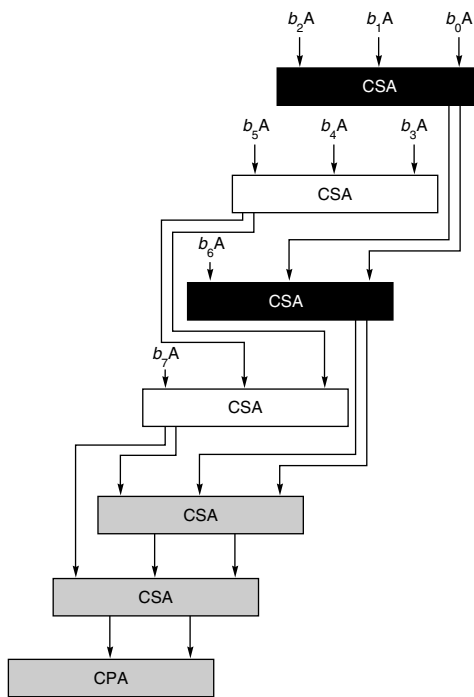
Figure J.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

various schemes for further speedup. One of them is shown in Figure J.29. The idea of this design is that two adds proceed in parallel or, to put it another way, each stream passes through only half the adders. Thus, it runs at almost twice the speed of the multiplier in Figure J.27. This *even/odd* multiplier is popular in VLSI because of its regular structure. Arrays can also be speeded up using asynchronous logic. One of the reasons why the multiplier of Figure J.2 (page J-4) needs a clock is to keep the output of the adder from feeding back into the input of the adder before the output has fully stabilized. Thus, if the array in Figure J.28 is long enough so that no signal can propagate from the top through the bottom in the time it takes for the first adder to stabilize, it may be possible to avoid clocks altogether. Williams et al. [1987] discussed a design using this idea, although it is for dividers instead of multipliers.

The techniques of the previous paragraph still have a multiply time of $0(n)$, but the time can be reduced to log n using a tree. The simplest tree would combine pairs of summands $b_0A \cdots b_{n-1}A$, cutting the number of summands from n to $n/2$. Then these $n/2$ numbers would be added in pairs again, reducing to $n/4$, and so on, and resulting in a single sum after log n steps. However, this simple binary tree idea doesn't map into full (3,2) adders, which reduce three inputs to two rather than reducing two inputs to one. A tree that does use full adders, known as
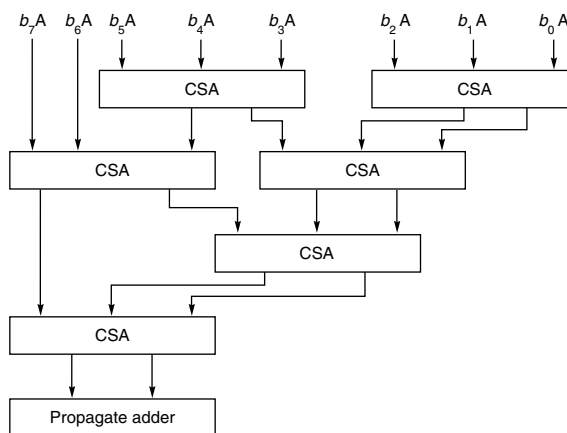
Figure J.30 Wallace tree multiplier. An example of a multiply tree that computes a product in $0(\log n)$ steps.

a *Wallace tree*, is shown in Figure J.30. When computer arithmetic units were built out of MSI parts, a Wallace tree was the design of choice for high-speed multipliers. There is, however, a problem with implementing it in VLSI. If you try to fill in all the adders and paths for the Wallace tree of Figure J.30, you will discover that it does not have the nice, regular structure of Figure J.27. This is why VLSI designers have often chosen to use other log n designs such as the *binary tree multiplier*, which is discussed next.

The problem with adding summands in a binary tree is coming up with a (2,1) adder that combines two digits and produces a single-sum digit. Because of carries, this isn't possible using binary notation, but it can be done with some other representation. We will use the *signed-digit representation* 1, $\overline{1}$, and 0, which we used previously to understand Booth's algorithm. This representation has two costs. First, it takes 2 bits to represent each signed digit. Second, the algorithm for adding two signed-digit numbers a_i and b_i is complex and requires examining $a_i a_{i-1} a_{i-2}$ and $b_i b_{i-1} b_{i-2}$. Although this means you must look 2 bits back, in binary addition you might have to look an arbitrary number of bits back because of carries.

We can describe the algorithm for adding two signed-digit numbers as follows. First, compute sum and carry bits s_i and c_{i+1} using Figure J.31. Then compute the final sum as $s_i + c_i$. The tables are set up so that this final sum does not generate a carry.

Example What is the sum of the signed-digit numbers $1\overline{1}0_2$ and 001_2?

Answer The two low-order bits sum to $0 + 1 = 1\overline{1}$, the next pair sums to $\overline{1} + 0 = 0\overline{1}$, and the high-order pair sums to $1 + 0 = 01$, so the sum is $1\overline{1} + 0\overline{1}0 + 0100 = 10\overline{1}_2$.

1	1	$\bar{1}$	0	1 x	$\bar{1}$ x
+1	+$\bar{1}$	+$\bar{1}$	+0	+0 y	+0 y
$\overline{1\,0}$	$\overline{0\,0}$	$\overline{1\,0}$	$\overline{0\,0}$	$1\,\bar{1}$ if $x \geq 0$ and $y \geq 0$	$0\,\bar{1}$ if $x \geq 0$ and $y \geq 0$
				0 1 otherwise	$\bar{1}$ 1 otherwise

Figure J.31 Signed-digit addition table. The leftmost sum shows that when computing $1 + 1$, the sum bit is 0 and the carry bit is 1.

This, then, defines a (2,1) adder. With this in hand, we can use a straightforward binary tree to perform multiplication. In the first step it adds $b_0A + b_1A$ in parallel with $b_2A + b_3A, \ldots, b_{n-2}A + b_{n-1}A$. The next step adds the results of these sums in pairs, and so on. Although the final sum must be run through a carry-propagate adder to convert it from signed-digit form to two's complement, this final add step is necessary in any multiplier using CSAs.

To summarize, both Wallace trees and signed-digit trees are log n multipliers. The Wallace tree uses fewer gates but is harder to lay out. The signed-digit tree has a more regular structure, but requires 2 bits to represent each digit and has more complicated add logic. As with adders, it is possible to combine different multiply techniques. For example, Booth recoding and arrays can be combined. In Figure J.27 instead of having each input be b_iA, we could have it be $b_ib_{i-1}A$. To avoid having to compute the multiple $3b$, we can use Booth recoding.

Faster Division with One Adder

The two techniques we discussed for speeding up multiplication with a single adder were carry-save adders and higher-radix multiplication. However, there is a difficulty when trying to utilize these approaches to speed up nonrestoring division. If the adder in Figure J.2(b) on page J-4 is replaced with a carry-save adder, then P will be replaced with two registers, one for the sum bits and one for the carry bits (compare with the multiplier in Figure J.24). At the end of each cycle, the sign of P is uncertain (since P is the unevaluated sum of the two registers), yet it is the sign of P that is used to compute the quotient digit and decide the next ALU operation. When a higher radix is used, the problem is deciding what value to subtract from P. In the paper-and-pencil method, you have to guess the quotient digit. In binary division, there are only two possibilities. We were able to finesse the problem by initially guessing one and then adjusting the guess based on the sign of P. This doesn't work in higher radices because there are more than two possible quotient digits, rendering quotient selection potentially quite complicated: You would have to compute all the multiples of b and compare them to P.

Both the carry-save technique and higher-radix division can be made to work if we use a redundant quotient representation. Recall from our discussion of SRT division (page J-45) that by allowing the quotient digits to be −1, 0, or 1, there is often a choice of which one to pick. The idea in the previous algorithm was to choose 0 whenever possible, because that meant an ALU operation could be
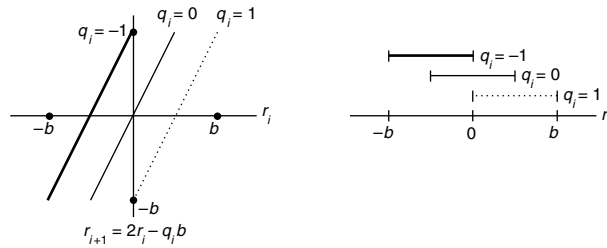
Figure J.32 Quotient selection for radix-2 division. The *x*-axis represents the *i*th remainder, which is the quantity in the (P,A) register pair. The *y*-axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

skipped. In carry-save division, the idea is that, because the remainder (which is the value of the (P,A) register pair) is not known exactly (being stored in carry-save form), the exact quotient digit is also not known. But, thanks to the redundant representation, the remainder doesn't have to be known precisely in order to pick a quotient digit. This is illustrated in Figure J.32, where the *x*-axis represents r_i, the remainder after *i* steps. The line labeled $q_i = 1$ shows the value that r_{i+1} would be if we chose $q_i = 1$, and similarly for the lines $q_i = 0$ and $q_i = -1$. We can choose any value for q_i, as long as $r_{i+1} = 2r_i - q_i b$ satisfies $|r_{i+1}| \leq b$. The allowable ranges are shown in the right half of Figure J.32. This shows that you don't need to know the precise value of r_i in order to choose a quotient digit q_i. You only need to know that r lies in an interval small enough to fit entirely within one of the overlapping bars shown in the right half of Figure J.32.

This is the basis for using carry-save adders. Look at the high-order bits of the carry-save adder and sum them in a propagate adder. Then use this approximation of r (together with the divisor, b) to compute q_i, usually by means of a lookup table. The same technique works for higher-radix division (whether or not a carry-save adder is used). The high-order bits P can be used to index a table that gives one of the allowable quotient digits.

The design challenge when building a high-speed SRT divider is figuring out how many bits of P and B need to be examined. For example, suppose that we take a radix of 4, use quotient digits of 2, 1, 0, $\overline{1}$, $\overline{2}$, but have a propagate adder. How many bits of P and B need to be examined? Deciding this involves two steps. For ordinary radix-2 nonrestoring division, because at each stage $|r| \leq b$, the P buffer won't overflow. But, for radix 4, $r_{i+1} = 4r_i - q_i b$ is computed at each stage, and if r_i is near b, then $4r_i$ will be near $4b$, and even the largest quotient digit will not bring r back to the range $|r_{i+1}| \leq b$. In other words, the remainder might grow without bound. However, restricting $|r_i| \leq 2b/3$ makes it easy to check that r_i will stay bounded.

After figuring out the bound that r_i must satisfy, we can draw the diagram in Figure J.33, which is analogous to Figure J.32. For example, the diagram shows
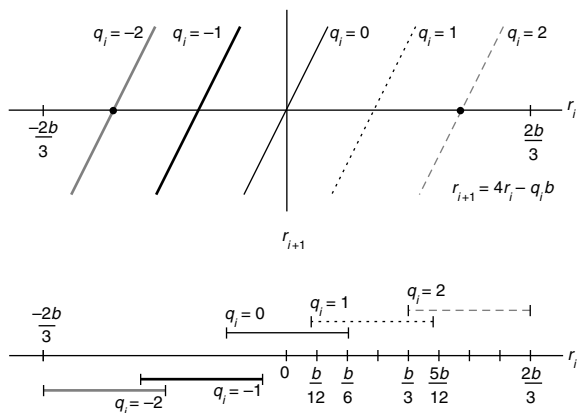
Figure J.33 Quotient selection for radix-4 division with quotient digits –2, –1, 0, 1, 2.

that if r_i is between $(1/12)b$ and $(5/12)b$, we can pick $q = 1$, and so on. Or, to put it another way, if r/b is between $1/12$ and $5/12$, we can pick $q = 1$. Suppose the divider examines 5 bits of P (including the sign bit) and 4 bits of b (ignoring the sign, since it is always nonnegative). The interesting case is when the high bits of P are $00011xxx\cdots$, while the high bits of b are $1001xxx\cdots$. Imagine the binary point at the left end of each register. Since we truncated, r (the value of P concatenated with A) could have a value from 0.0011_2 to 0.0100_2, and b could have a value from $.1001_2$ to $.1010_2$. Thus, r/b could be as small as $0.0011_2/.1010_2$ or as large as $0.0100_2/.1001_2$, but $0.0011_2/.1010_2 = 3/10 < 1/3$ would require a quotient bit of 1, while $0.0100_2/.1001_2 = 4/9 > 5/12$ would require a quotient bit of 2. In other words, 5 bits of P and 4 bits of b aren't enough to pick a quotient bit. It turns out that 6 bits of P and 4 bits of b are enough. This can be verified by writing a simple program that checks all the cases. The output of such a program is shown in Figure J.34.

Example   Using 8-bit registers, compute 149/5 using radix-4 SRT division.

Answer   Follow the SRT algorithm on page J-45, but replace the quotient selection rule in step 2 with one that uses Figure J.34. See Figure J.35.

The Pentium uses a radix-4 SRT division algorithm like the one just presented, except that it uses a carry-save adder. Exercises J.34(c) and J.35 explore this in detail. Although these are simple cases, all SRT analyses proceed in the same way. First compute the range of r_i, then plot r_i against r_{i+1} to find the quotient ranges, and finally write a program to compute how many bits are necessary. (It is sometimes also possible to compute the required number of bits analytically.) Various details need to be considered in building a practical SRT divider.

b	Range of P		q	b	Range of P		q
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

Figure J.34 Quotient digits for radix-4 SRT division with a propagate adder. The top row says that if the high-order 4 bits of b are $1000_2 = 8$, and if the top 6 bits of P are between $110100_2 = -12$ and $111001_2 = -7$, then -2 is a valid quotient digit.

```
      P            A
000000000   10010101    Divide 149 by 5. B contains 00000101.
000010010   10100000    Step 1:     B had 5 leading 0s, so shift left by 5. B now
                                     contains 10100000, so use b = 10 section of table.
                        Step 2.1:   Top 6 bits of P are 2, so
                                    shift left by 2. From table, can pick q to be
001001010   1000000                 0 or 1. Choose q₀ = 0.
                        Step 2.2:   Top 6 bits of P are 9, so
100101010    000002                 shift left 2. q₁ = 2.
+ 011000000                         Subtract 2b.
111101010    000002     Step 2.3:   Top bits = −3, so
110101000     00020                 shift left 2. Can pick 0 or −1 for q, pick q₂ = 0.
                        Step 2.4:   Top bits = −11, so
010100000      0202                 shift left 2. q₃ = −2.
+ 101000000                         Add 2b.
  111100000             Step 3:     Remainder is negative, so restore
+ 010100000                         by adding b and subtract 1 from q.
  010000000             Answer:     q = 0202̄ − 1 = 29.
                                    To get remainder, undo shift in step 1 so
                                    remainder = 010000000 >> 5 = 4.
```

Figure J.35 Example of radix-4 SRT division. Division of 149 by 5.

For example, the quotient lookup table has a fairly regular structure, which means it is usually cheaper to encode it as a PLA rather than in ROM. For more details about SRT division, see Burgess and Williams [1995].

J.10   Putting It All Together

In this section, we will compare the Weitek 3364, the MIPS R3010, and the Texas Instruments 8847 (see Figures J.36 and J.37). In many ways, these are ideal chips to compare. They each implement the IEEE standard for addition, subtraction, multiplication, and division on a single chip. All were introduced in 1988 and run with a cycle time of about 40 nanoseconds. However, as we will see, they use quite different algorithms. The Weitek chip is well described in Birman et al. [1990], the MIPS chip is described in less detail in Rowen, Johnson, and Ries [1988], and details of the TI chip can be found in Darley et al. [1989].

These three chips have a number of things in common. They perform addition and multiplication in parallel, and they implement neither extended precision nor a remainder step operation. (Recall from Section J.6 that it is easy to implement the IEEE remainder function in software if a remainder step instruction is available.) The designers of these chips probably decided not to provide extended precision because the most influential users are those who run portable codes, which can't rely on extended precision. However, as we have seen, extended precision can make for faster and simpler math libraries.

In the summary of the three chips given in Figure J.36, note that a higher transistor count generally leads to smaller cycle counts. Comparing the cycles/op numbers needs to be done carefully, because the figures for the MIPS chip are those for a complete system (R3000/3010 pair), while the Weitek and TI numbers are for stand-alone chips and are usually larger when used in a complete system.

Features	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/square root	–	30	14

Figure J.36 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.
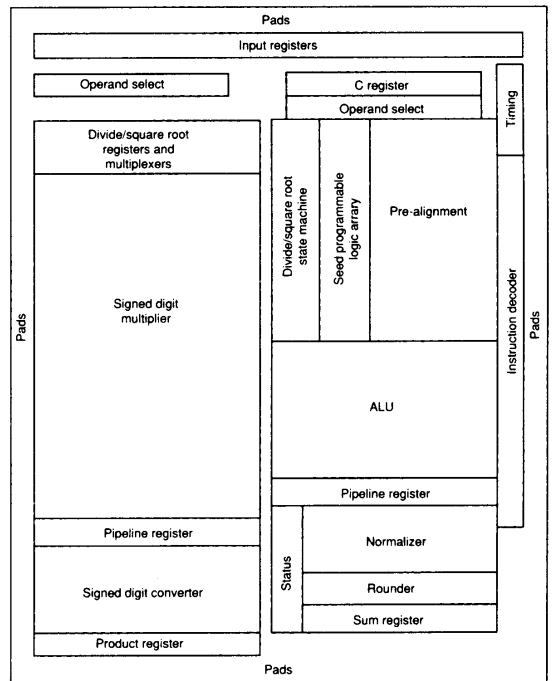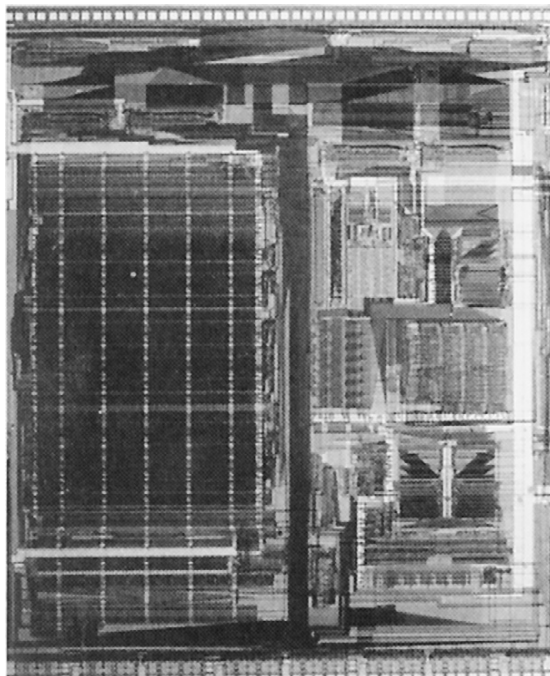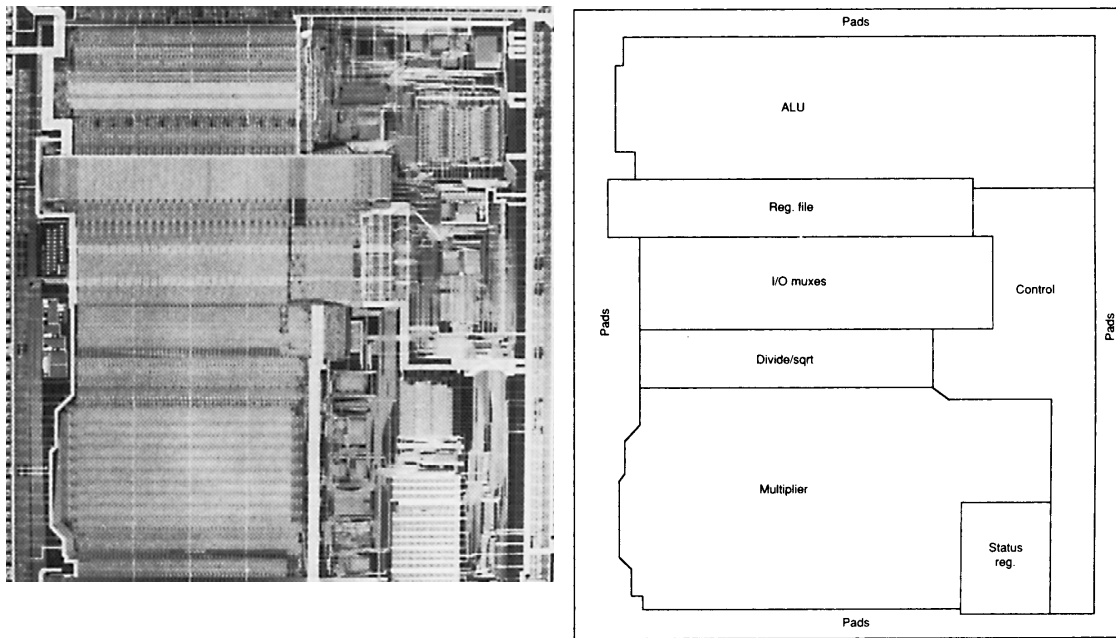
TI 8847
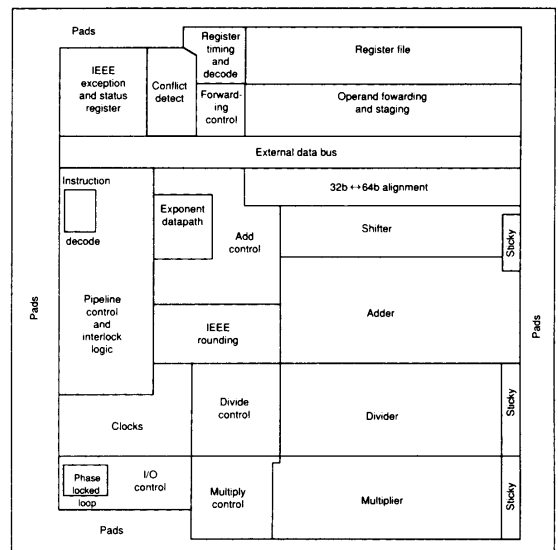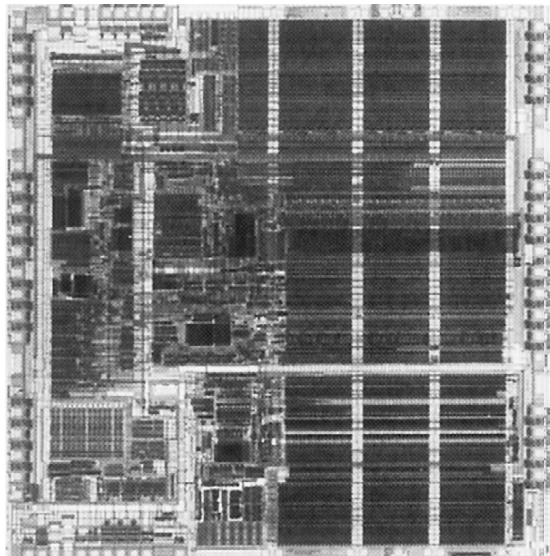
MIPS R3010

Figure J.37 Chip layout for the TI 8847, MIPS R3010, and Weitek 3364. In the left-hand columns are the photomicrographs; the right-hand columns show the corresponding floor plans.

Weitek 3364

Figure J.37 (*Continued.*)

The MIPS chip has the fewest transistors of the three. This is reflected in the fact that it is the only chip of the three that does not have any pipelining or hardware square root. Further, the multiplication and addition operations are not completely independent because they share the carry-propagate adder that performs the final rounding (as well as the rounding logic).

Addition on the R3010 uses a mixture of ripple, CLA, and carry-select. A carry-select adder is used in the fashion of Figure J.20 (page J-43). Within each half, carries are propagated using a hybrid ripple-CLA scheme of the type indicated in Figure J.19 (page J-42). However, this is further tuned by varying the size of each block, rather than having each fixed at 4 bits (as they are in Figure J.19). The multiplier is midway between the designs of Figures J.2 (page J-4) and J.27 (page J-50). It has an array just large enough so that output can be fed back into the input without having to be clocked. Also, it uses radix-4 Booth recoding and the even/odd technique of Figure J.29 (page J-52). The R3010 can do a divide and multiply in parallel (like the Weitek chip but unlike the TI chip). The divider is a radix-4 SRT method with quotient digits -2, -1, 0, 1, and 2, and is similar to that described in Taylor [1985]. Double-precision division is about four times slower than multiplication. The R3010 shows that for chips using an $0(n)$

multiplier, an SRT divider can operate fast enough to keep a reasonable ratio between multiply and divide.

The Weitek 3364 has independent add, multiply, and divide units. It also uses radix-4 SRT division. However, the add and multiply operations on the Weitek chip are pipelined. The three addition stages are (1) exponent compare, (2) add followed by shift (or *vice versa*), and (3) final rounding. Stages (1) and (3) take only a half-cycle, allowing the whole operation to be done in two cycles, even though there are three pipeline stages. The multiplier uses an array of the style of Figure J.28 but uses radix-8 Booth recoding, which means it must compute 3 times the multiplier. The three multiplier pipeline stages are (1) compute $3b$, (2) pass through array, and (3) final carry-propagation add and round. Single precision passes through the array once, double precision twice. Like addition, the latency is two cycles.

The Weitek chip uses an interesting addition algorithm. It is a variant on the carry-skip adder pictured in Figure J.18 (page J-42). However, P_{ij}, which is the logical AND of many terms, is computed by rippling, performing one AND per ripple. Thus, while the carries propagate left within a block, the value of P_{ij} is propagating right within the next block, and the block sizes are chosen so that both waves complete at the same time. Unlike the MIPS chip, the 3364 has hardware square root, which shares the divide hardware. The ratio of double-precision multiply to divide is 2:17. The large disparity between multiply and divide is due to the fact that multiplication uses radix-8 Booth recoding, while division uses a radix-4 method. In the MIPS R3010, multiplication and division use the same radix.

The notable feature of the TI 8847 is that it does division by iteration (using the Goldschmidt algorithm discussed in Section J.6). This improves the speed of division (the ratio of multiply to divide is 3:11), but means that multiplication and division cannot be done in parallel as on the other two chips. Addition has a two-stage pipeline. Exponent compare, fraction shift, and fraction addition are done in the first stage, normalization and rounding in the second stage. Multiplication uses a binary tree of signed-digit adders and has a three-stage pipeline. The first stage passes through the array, retiring half the bits; the second stage passes through the array a second time; and the third stage converts from signed-digit form to two's complement. Since there is only one array, a new multiply operation can only be initiated in every other cycle. However, by slowing down the clock, two passes through the array can be made in a single cycle. In this case, a new multiplication can be initiated in each cycle. The 8847 adder uses a carry-select algorithm rather than carry-lookahead. As mentioned in Section J.6, the TI carries 60 bits of precision in order to do correctly rounded division.

These three chips illustrate the different trade-offs made by designers with similar constraints. One of the most interesting things about these chips is the diversity of their algorithms. Each uses a different add algorithm, as well as a different multiply algorithm. In fact, Booth recoding is the only technique that is universally used by all the chips.

J.11 Fallacies and Pitfalls

Fallacy *Underflows rarely occur in actual floating-point application code.*

Although most codes rarely underflow, there are actual codes that underflow frequently. SDRWAVE [Kahaner 1988], which solves a one-dimensional wave equation, is one such example. This program underflows quite frequently, even when functioning properly. Measurements on one machine show that adding hardware support for gradual underflow would cause SDRWAVE to run about 50% faster.

Fallacy *Conversions between integer and floating point are rare.*

In fact, in spice they are as frequent as divides. The assumption that conversions are rare leads to a mistake in the SPARC version 8 instruction set, which does not provide an instruction to move from integer registers to floating-point registers.

Pitfall *Don't increase the speed of a floating-point unit without increasing its memory bandwidth.*

A typical use of a floating-point unit is to add two vectors to produce a third vector. If these vectors consist of double-precision numbers, then each floating-point add will use three operands of 64 bits each, or 24 bytes of memory. The memory bandwidth requirements are even greater if the floating-point unit can perform addition and multiplication in parallel (as most do).

Pitfall *−x is not the same as 0 − x.*

This is a fine point in the IEEE standard that has tripped up some designers. Because floating-point numbers use the sign magnitude system, there are two zeros, +0 and −0. The standard says that $0 - 0 = +0$, whereas $-(0) = -0$. Thus, $-x$ is not the same as $0 - x$ when $x = 0$.

J.12 Historical Perspective and References

The earliest computers used fixed point rather than floating point. In "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Burks, Goldstine, and von Neumann [1946] put it like this:

> There appear to be two major purposes in a "floating" decimal point system both of which arise from the fact that the number of digits in a word is a constant fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplicative constants which serve to keep numbers within the limits of the machine.

There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity that could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.

This enables us to see things from the perspective of early computer designers, who believed that saving computer time and memory were more important than saving programmer time.

The original papers introducing the Wallace tree, Booth recoding, SRT division, overlapped triplets, and so on are reprinted in Swartzlander [1990]. A good explanation of an early machine (the IBM 360/91) that used a pipelined Wallace tree, Booth recoding, and iterative division is in Anderson et al. [1967]. A discussion of the average time for single-bit SRT division is in Freiman [1961]; this is one of the few interesting historical papers that does not appear in Swartzlander.

The standard book of Mead and Conway [1980] discouraged the use of CLAs as not being cost effective in VLSI. The important paper by Brent and Kung [1982] helped combat that view. An example of a detailed layout for CLAs can be found in Ngai and Irwin [1985] or in Weste and Eshraghian [1993], and a more theoretical treatment is given by Leighton [1992]. Takagi, Yasuura, and Yajima [1985] provide a detailed description of a signed-digit tree multiplier.

Before the ascendancy of IEEE arithmetic, many different floating-point formats were in use. Three important ones were used by the IBM 370, the DEC VAX, and the Cray. Here is a brief summary of these older formats. The VAX format is closest to the IEEE standard. Its single-precision format (F format) is like IEEE single precision in that it has a hidden bit, 8 bits of exponent, and 23 bits of fraction. However, it does not have a sticky bit, which causes it to round halfway cases up instead of to even. The VAX has a slightly different exponent range from IEEE single: E_{min} is -128 rather than -126 as in IEEE, and E_{max} is 126 instead of 127. The main differences between VAX and IEEE are the lack of special values and gradual underflow. The VAX has a reserved operand, but it works like a signaling NaN: It traps whenever it is referenced. Originally, the VAX's double precision (D format) also had 8 bits of exponent. However, as this is too small for many applications, a G format was added; like the IEEE standard, this format has 11 bits of exponent. The VAX also has an H format, which is 128 bits long.

The IBM 370 floating-point format uses base 16 rather than base 2. This means it cannot use a hidden bit. In single precision, it has 7 bits of exponent and 24 bits (6 hex digits) of fraction. Thus, the largest representable number is $16^{27} = 2^4 \times 2^7 = 2^{29}$, compared with 2^{28} for IEEE. However, a number that is normalized in the hexadecimal sense only needs to have a nonzero leading digit. When interpreted in binary, the three most-significant bits could be zero. Thus, there are potentially fewer than 24 bits of significance. The reason for using the

higher base was to minimize the amount of shifting required when adding floating-point numbers. However, this is less significant in current machines, where the floating-point add time is usually fixed independently of the operands. Another difference between 370 arithmetic and IEEE arithmetic is that the 370 has neither a round digit nor a sticky digit, which effectively means that it truncates rather than rounds. Thus, in many computations, the result will systematically be too small. Unlike the VAX and IEEE arithmetic, every bit pattern is a valid number. Thus, library routines must establish conventions for what to return in case of errors. In the IBM FORTRAN library, for example, $\sqrt{-4}$ returns 2!

Arithmetic on Cray computers is interesting because it is driven by a motivation for the highest possible floating-point performance. It has a 15-bit exponent field and a 48-bit fraction field. Addition on Cray computers does not have a guard digit, and multiplication is even less accurate than addition. Thinking of multiplication as a sum of p numbers, each $2p$ bits long, Cray computers drop the low-order bits of each summand. Thus, analyzing the exact error characteristics of the multiply operation is not easy. Reciprocals are computed using iteration, and division of a by b is done by multiplying a times $1/b$. The errors in multiplication and reciprocation combine to make the last three bits of a divide operation unreliable. At least Cray computers serve to keep numerical analysts on their toes!

The IEEE standardization process began in 1977, inspired mainly by W. Kahan and based partly on Kahan's work with the IBM 7094 at the University of Toronto [Kahan 1968]. The standardization process was a lengthy affair, with gradual underflow causing the most controversy. (According to Cleve Moler, visitors to the United States were advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.) The standard was finally approved in 1985. The Intel 8087 was the first major commercial IEEE implementation and appeared in 1981, before the standard was finalized. It contains features that were eliminated in the final standard, such as projective bits. According to Kahan, the length of double-extended precision was based on what could be implemented in the 8087. Although the IEEE standard was not based on any existing floating-point system, most of its features were present in some other system. For example, the CDC 6600 reserved special bit patterns for INDEFINITE and INFINITY, while the idea of denormal numbers appears in Goldberg [1967] as well as in Kahan [1968]. Kahan was awarded the 1989 Turing prize in recognition of his work on floating point.

Although floating point rarely attracts the interest of the general press, newspapers were filled with stories about floating-point division in November 1994. A bug in the division algorithm used on all of Intel's Pentium chips had just come to light. It was discovered by Thomas Nicely, a math professor at Lynchburg College in Virginia. Nicely found the bug when doing calculations involving reciprocals of prime numbers. News of Nicely's discovery first appeared in the press on the front page of the November 7 issue of *Electronic Engineering Times.* Intel's immediate response was to stonewall, asserting that the bug would only affect theoretical mathematicians. Intel told the press, "This doesn't even qualify as an errata . . . even if you're an engineer, you're not going to see this."

Under more pressure, Intel issued a white paper, dated November 30, explaining why they didn't think the bug was significant. One of their arguments was based on the fact that if you pick two floating-point numbers at random and divide one into the other, the chance that the resulting quotient will be in error is about 1 in 9 billion. However, Intel neglected to explain why they thought that the typical customer accessed floating-point numbers randomly.

Pressure continued to mount on Intel. One sore point was that Intel had known about the bug before Nicely discovered it, but had decided not to make it public. Finally, on December 20, Intel announced that they would unconditionally replace any Pentium chip that used the faulty algorithm and that they would take an unspecified charge against earnings, which turned out to be $300 million.

The Pentium uses a simple version of SRT division as discussed in Section J.9. The bug was introduced when they converted the quotient lookup table to a PLA. Evidently there were a few elements of the table containing the quotient digit 2 that Intel thought would never be accessed, and they optimized the PLA design using this assumption. The resulting PLA returned 0 rather than 2 in these situations. However, those entries were really accessed, and this caused the division bug. Even though the effect of the faulty PLA was to cause 5 out of 2048 table entries to be wrong, the Pentium only computes an incorrect quotient 1 out of 9 billion times on random inputs. This is explored in Exercise J.34.

References

Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers [1967]. "The IBM System/360 Model 91: Floating-point execution unit," *IBM J. Research and Development* 11, 34–53. Reprinted in Swartzlander [1990].
Good description of an early high-performance floating-point unit that used a pipelined Wallace tree multiplier and iterative division.

Bell, C. G., and A. Newell [1971]. *Computer Structures: Readings and Examples*, McGraw-Hill, New York.

Birman, M., A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes [1990]. "Developing the WRL3170/3171 SPARC floating-point coprocessors," *IEEE Micro* 10:1, 55–64.
These chips have the same floating-point core as the Weitek 3364, and this paper has a fairly detailed description of that floating-point design.

Brent, R. P., and H. T. Kung [1982]. "A regular layout for parallel adders," *IEEE Trans. on Computers* C-31, 260–264.
This is the paper that popularized CLAs in VLSI.

Burgess, N., and T. Williams [1995]. "Choices of operand truncation in the SRT division algorithm," *IEEE Trans. on Computers* 44:7.
Analyzes how many bits of divisor and remainder need to be examined in SRT division.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, 1987, 97–146.

Cody, W. J., J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson [1984]. "A proposed radix- and word-length-independent standard for floating-point arithmetic," *IEEE Micro* 4:4, 86–100.
Contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754. However, be aware that there are some differences between this draft and the final standard.

Coonen, J. [1984]. "Contributions to a proposed standard for binary floating point arithmetic," Ph.D. thesis, University of California–Berkeley.
The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversion.

Darley, H. M. et al. [1989]. "Floating point/integer processor with divide and square root functions," U.S. Patent 4,878,190, October 31, 1989.
Pretty readable as patents go. Gives a high-level view of the TI 8847 chip, but doesn't have all the details of the division algorithm.

Demmel, J. W., and X. Li [1994]. "Faster numerical algorithms via exception handling," *IEEE Trans. on Computers* 43:8, 983–992.
A good discussion of how the features unique to IEEE floating point can improve the performance of an important software library.

Freiman, C. V. [1961]. "Statistical analysis of certain binary division algorithms," *Proc. IRE* 49:1, 91–103.
Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *Computing Surveys* 23:1, 5–48.
Contains an in-depth tutorial on the IEEE standard from the software point of view.

Goldberg, I. B. [1967]. "27 bits are not enough for 8-digit accuracy," *Comm. ACM* 10:2, 105–106.
This paper proposes using hidden bits and gradual underflow.

Gosling, J. B. [1980]. *Design of Arithmetic Units for Digital Computers*, Springer-Verlag, New York.
A concise, well-written book, although it focuses on MSI designs.

Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky [1984]. *Computer Organization*, 2nd ed., McGraw-Hill, New York.
Introductory computer architecture book with a good chapter on computer arithmetic.

Hwang, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York.
This book contains the widest range of topics of the computer arithmetic books.

IEEE [1985]. "IEEE standard for binary floating-point arithmetic," *SIGPLAN Notices* 22:2, 9–25.
IEEE 754 is reprinted here.

Kahan, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159.
This system had many features that were incorporated into the IEEE floating-point standard.

Kahaner, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News* (November).
The benchmark presented in this article turns out to cause many underflows.

Knuth, D. [1981]. *The Art of Computer Programming*, Vol. II, 2nd ed., Addison-Wesley, Reading, Mass.
Has a section on the distribution of floating-point numbers.

Kogge, P. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.
Has a brief discussion of pipelined multipliers.

Kohn, L., and S.-W. Fu [1989]. "A 1,000,000 transistor microprocessor," *IEEE Int'l. Solid-State Circuits Conf. Digest of Technical Papers*, 54–55.
There are several articles about the i860, but this one contains the most details about its floating-point algorithms.

Koren, I. [1989]. *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, N.J.

Leighton, F. T. [1992]. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann, San Francisco.
This is an excellent book, with emphasis on the complexity analysis of algorithms. Section 1.2.1 has a nice discussion of carry-lookahead addition on a tree.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP Precision architecture," *IEEE Trans. on Computers* 37:8, 980–990.
Gives rationale for the integer- and divide-step instructions in the Precision architecture.

Markstein, P. W. [1990]. "Computation of elementary functions on the IBM RISC System/6000 processor," *IBM J. of Research and Development* 34:1, 111–119.
Explains how to use fused muliply-add to compute correctly rounded division and square root.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

Montoye, R. K., E. Hokenek, and S. L. Runyon [1990]. "Design of the IBM RISC System/6000 floating-point execution," *IBM J. of Research and Development* 34:1, 59–70.
Describes one implementation of fused multiply-add.

Ngai, T.-F., and M. J. Irwin [1985]. "Regular, area-time efficient carry-lookahead adders," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 9–15.
Describes a CLA like that of Figure J.17, where the bits flow up and then come back down.

Patterson, D. A., and J. L. Hennessy [2009]. *Computer Organization and Design: The Hardware/Software Interface*, 4th Edition, Morgan Kaufmann, San Francisco.
Chapter 3 is a gentler introduction to the first third of this appendix.

Peng, V., S. Samudrala, and M. Gavrielov [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units," *Proc. Eighth IEEE Symposium on Computer Arithmetic*, 95–102.
Highly recommended survey of different techniques actually used in VLSI designs.

Rowen, C., M. Johnson, and P. Ries [1988]. "The MIPS R3010 floating-point coprocessor," *IEEE Micro,* 53–62 (June).

Santoro, M. R., G. Bewick, and M. A. Horowitz [1989]. "Rounding algorithms for IEEE multipliers," *Proc. Ninth IEEE Symposium on Computer Arithmetic*, 176–183.
A very readable discussion of how to efficiently implement rounding for floating-point multiplication.

Scott, N. R. [1985]. *Computer Number Systems and Arithmetic*, Prentice Hall, Englewood Cliffs, N.J.

Swartzlander, E., ed. [1990]. *Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif.
A collection of historical papers in two volumes.

Takagi, N., H. Yasuura, and S. Yajima [1985]."High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. on Computers* C-34:9, 789–796.
A discussion of the binary tree signed multiplier that was the basis for the design used in the TI 8847.

Taylor, G. S. [1981]. "Compatible hardware for division and square root," *Proc. Fifth IEEE Symposium on Computer Arithmetic*, May 18–19, 1981, Ann Arbor, Mich., 127–134.
Good discussion of a radix-4 SRT division algorithm.

Taylor, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symposium on Computer Arithmetic*, June 4–6, 1985, Urbana, Ill., 64–71.
Describes a very sophisticated high-radix division algorithm.

Weste, N., and K. Eshraghian [1993]. *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed., Addison-Wesley, Reading, Mass.
This textbook has a section on the layouts of various kinds of adders.

Williams, T. E., M. Horowitz, R. L. Alverson, and T. S. Yang [1987]. "A self-timed chip for division," *Advanced Research in VLSI, Proc. 1987 Stanford Conf.*, MIT Press, Cambridge, Mass.
Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.

Exercises

J.1   [12] <J.2> Using *n* bits, what is the largest and smallest integer that can be represented in the two's complement system?

J.2   [20/25] <J.2> In the subsection "Signed Numbers" (page J-7), it was stated that two's complement overflows when the carry into the high-order bit position is different from the carry-out from that position.

   a. [20] <J.2> Give examples of pairs of integers for all four combinations of carry-in and carry-out. Verify the rule stated above.

   b. [25] <J.2> Explain why the rule is always true.

J.3   [12] <J.2> Using 4-bit binary numbers, multiply -8×-8 using Booth recoding.

J.4   [15] <J.2> Equations J.2.1 and J.2.2 are for adding two *n*-bit numbers. Derive similar equations for subtraction, where there will be a borrow instead of a carry.

J.5   [25] <J.2> On a machine that doesn't detect integer overflow in hardware, show how you would detect overflow on a signed addition operation in software.

J.6   [15/15/20] <J.3> Represent the following numbers as single-precision and double-precision IEEE floating-point numbers:

   a. [15] <J.3> 10.

   b. [15] <J.3> 10.5.

   c. [20] <J.3> 0.1.

J.7 [12/12/12/12/12] <J.3> Below is a list of floating-point numbers. In single precision, write down each number in binary, in decimal, and give its representation in IEEE arithmetic.

a. [12] <J.3> The largest number less than 1.

b. [12] <J.3> The largest number.

c. [12] <J.3> The smallest positive normalized number.

d. [12] <J.3> The largest denormal number.

e. [12] <J.3> The smallest positive number.

J.8 [15] <J.3> Is the ordering of nonnegative floating-point numbers the same as integers when denormalized numbers are also considered?

J.9 [20] <J.3> Write a program that prints out the bit patterns used to represent floating-point numbers on your favorite computer. What bit pattern is used for NaN?

J.10 [15] <J.4> Using $p = 4$, show how the binary floating-point multiply algorithm computes the product of 1.875×1.875.

J.11 [12/10] <J.4> Concerning the addition of exponents in floating-point multiply:

a. [12] <J.4> What would the hardware that implements the addition of exponents look like?

b. [10] <J.4> If the bias in single precision were 129 instead of 127, would addition be harder or easier to implement?

J.12 [15/12] <J.4> In the discussion of overflow detection for floating-point multiplication, it was stated that (for single precision) you can detect an overflowed exponent by performing exponent addition in a 9-bit adder.

a. [15] <J.4> Give the exact rule for detecting overflow.

b. [12] <J.4> Would overflow detection be any easier if you used a 10-bit adder instead?

J.13 [15/10] <J.4> Floating-point multiplication:

a. [15] <J.4> Construct two single-precision floating-point numbers whose product doesn't overflow until the final rounding step.

b. [10] <J.4> Is there any rounding mode where this phenomenon cannot occur?

J.14 [15] <J.4> Give an example of a product with a denormal operand but a normalized output. How large was the final shifting step? What is the maximum possible shift that can occur when the inputs are double-precision numbers?

J.15 [15] <J.5> Use the floating-point addition algorithm on page J-23 to compute $1.010_2 - .1001_2$ (in 4-bit precision).

J.16 [10/15/20/20/20] <J.5> In certain situations, you can be sure that $a + b$ is exactly representable as a floating-point number, that is, no rounding is necessary.

    a. [10] <J.5> If a, b have the same exponent and different signs, explain why $a + b$ is exact. This was used in the subsection "Speeding Up Addition" on page J-25.

    b. [15] <J.5> Give an example where the exponents differ by 1, a and b have different signs, and $a + b$ is not exact.

    c. [20] <J.5> If $a \geq b \geq 0$, and the top two bits of a cancel when computing $a - b$, explain why the result is exact (this fact is mentioned on page J-22).

    d. [20] <J.5> If $a \geq b \geq 0$, and the exponents differ by 1, show that $a - b$ is exact unless the high order bit of $a - b$ is in the same position as that of a (mentioned in "Speeding Up Addition," page J-25).

    e. [20] <J.5> If the result of $a - b$ or $a + b$ is denormal, show that the result is exact (mentioned in the subsection "Underflow," on page J-36).

J.17 [15/20] <J.5> Fast floating-point addition (using parallel adders) for $p = 5$.

    a. [15] <J.5> Step through the fast addition algorithm for $a + b$, where $a = 1.0111_2$ and $b = .11011_2$.

    b. [20] <J.5> Suppose the rounding mode is toward $+\infty$. What complication arises in the above example for the adder that assumes a carry-out? Suggest a solution.

J.18 [12] <J.4, J.5> How would you use two parallel adders to avoid the final round-up addition in floating-point multiplication?

J.19 [30/10] <J.5> This problem presents a way to reduce the number of addition steps in floating-point addition from three to two using only a single adder.

    a. [30] <J.5> Let A and B be integers of opposite signs, with a and b their magnitudes. Show that the following rules for manipulating the unsigned numbers a and b gives $A + B$.

      1. Complement one of the operands.

      2. Use end-around carry to add the complemented operand and the other (uncomplemented) one.

      3. If there was a carry-out, the sign of the result is the sign associated with the uncomplemented operand.

      4. Otherwise, if there was no carry-out, complement the result, and give it the sign of the complemented operand.

    b. [10] <J.5> Use the above to show how steps 2 and 4 in the floating-point addition algorithm on page J-23 can be performed using only a single addition.

J.20 [20/15/20/15/20/15] <J.6> Iterative square root.

    a. [20] <J.6> Use Newton's method to derive an iterative algorithm for square root. The formula will involve a division.

    b. [15] <J.6> What is the fastest way you can think of to divide a floating-point number by 2?

c. [20] <J.6> If division is slow, then the iterative square root routine will also be slow. Use Newton's method on $f(x) = 1/x^2 - a$ to derive a method that doesn't use any divisions.

d. [15] <J.6> Assume that the ratio division by 2 : floating-point add : floating-point multiply is 1:2:4. What ratios of multiplication time to divide time makes each iteration step in the method of part (c) faster than each iteration in the method of part (a)?

e. [20] <J.6> When using the method of part (a), how many bits need to be in the initial guess in order to get double-precision accuracy after three iterations? (You may ignore rounding error.)

f. [15] <J.6> Suppose that when spice runs on the TI 8847, it spends 16.7% of its time in the square root routine (this percentage has been measured on other machines). Using the values in Figure J.36 and assuming three iterations, how much slower would spice run if square root were implemented in software using the method of part(a)?

J.21 [10/20/15/15/15] <J.6> Correctly rounded iterative division. Let a and b be floating-point numbers with p-bit significands ($p = 53$ in double precision). Let q be the exact quotient $q = a/b$, $1 \le q < 2$. Suppose that $\overline{q}$ is the result of an iteration process, that $\overline{q}$ has a few extra bits of precision, and that $0 < q - \overline{q} < 2^{-p}$. For the following, it is important that $\overline{q} < q$, even when q can be exactly represented as a floating-point number.

a. [10] <J.6> If x is a floating-point number, and $1 \le x < 2$, what is the next representable number after x?

b. [20] <J.6> Show how to compute q' from $\overline{q}$, where q' has $p + 1$ bits of precision and $\left| q - q' \right| < 2^{-p}$.

c. [15] <J.6> Assuming round to nearest, show that the correctly rounded quotient is either q', $q' - 2^{-p}$, or $q' + 2^{-p}$.

d. [15] <J.6> Give rules for computing the correctly rounded quotient from q' based on the low-order bit of q' and the sign of $a - bq'$.

e. [15] <J.6> Solve part (c) for the other three rounding modes.

J.22 [15] <J.6> Verify the formula on page J-30. (*Hint*: If $x_n = x_0(2 - x_0 b) \times \prod_{i=1,n} [1 + (1 - x_0 b)^{2^i}]$, then $2 - x_n b = 2 - x_0 b(2 - x_0 b) \prod[1 + (1 - x_0 b)^{2^i}] = 2 - [1 - (1 - x_0 b)^2] \prod[1 + (1 - x_0 b)^{2^i}]$.)

J.23 [15] <J.7> Our example that showed that double rounding can give a different answer from rounding once used the round-to-even rule. If halfway cases are always rounded up, is double rounding still dangerous?

J.24 [10/10/20/20] <J.7> Some of the cases of the italicized statement in the "Precisions" subsection (page J-33) aren't hard to demonstrate.

a. [10] <J.7> What form must a binary number have if rounding to q bits followed by rounding to p bits gives a different answer than rounding directly to p bits?

b. [10] <J.7> Show that for multiplication of p-bit numbers, rounding to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \geq 2p$.

c. [20] <J.7> If a and b are p-bit numbers with the same sign, show that rounding $a + b$ to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \geq 2p + 1$.

d. [20] <J.7> Do part (c) when a and b have opposite signs.

J.25   [Discussion] <J.7> In the MIPS approach to exception handling, you need a test for determining whether two floating-point operands could cause an exception. This should be fast and also not have too many false positives. Can you come up with a practical test? The performance cost of your design will depend on the distribution of floating-point numbers. This is discussed in Knuth [1981] and the Hamming paper in Swartzlander [1990].

J.26   [12/12/10] <J.8> Carry-skip adders.

a. [12] <J.8> Assuming that time is proportional to logic levels, how long does it take an n-bit adder divided into (fixed) blocks of length k bits to perform an addition?

b. [12] <J.8> What value of k gives the fastest adder?

c. [10] <J.8> Explain why the carry-skip adder takes time $0(\sqrt{n})$.

J.27   [10/15/20] <J.8> Complete the details of the block diagrams for the following adders.

a. [10] <J.8> In Figure J.15, show how to implement the "1" and "2" boxes in terms of AND and OR gates.

b. [15] <J.8> In Figure J.19, what signals need to flow from the adder cells in the top row into the "C" cells? Write the logic equations for the "C" box.

c. [20] <J.8> Show how to extend the block diagram in J.17 so it will produce the carry-out bit c_8.

J.28   [15] <J.9> For ordinary Booth recoding, the multiple of b used in the ith step is simply $a_{i-1} - a_i$. Can you find a similar formula for radix-4 Booth recoding (overlapped triplets)?

J.29   [20] <J.9> Expand Figure J.29 in the fashion of J.27, showing the individual adders.

J.30   [25] <J.9> Write out the analog of Figure J.25 for radix-8 Booth recoding.

J.31   [18] <J.9> Suppose that $a_{n-1} \ldots a_1 a_0$ and $b_{n-1} \ldots b_1 b_0$ are being added in a signed-digit adder as illustrated in the example on page J-53. Write a formula for the ith bit of the sum, s_i, in terms of a_i, a_{i-1}, a_{i-2}, b_i, b_{i-1}, and b_{i-2}.

J.32   [15] <J.9> The text discussed radix-4 SRT division with quotient digits of $-2, -1, 0, 1, 2$. Suppose that 3 and -3 are also allowed as quotient digits. What relation replaces $|r_i| \leq 2b/3$?

J.33 [25/20/30] <J.9> Concerning the SRT division table, Figure J.34:

    a. [25] <J.9> Write a program to generate the results of Figure J.34.

    b. [20] <J.9> Note that Figure J.34 has a certain symmetry with respect to positive and negative values of P. Can you find a way to exploit the symmetry and only store the values for positive P?

    c. [30] <J.9> Suppose a carry-save adder is used instead of a propagate adder. The input to the quotient lookup table will be k bits of divisor and l bits of remainder, where the remainder bits are computed by summing the top l bits of the sum and carry registers. What are k and l? Write a program to generate the analog of Figure J.34.

J.34 [12/12/12] <J.9, J.12> The first several million Pentium chips produced had a flaw that caused division to sometimes return the wrong result. The Pentium uses a radix-4 SRT algorithm similar to the one illustrated in the example on page J-56 (but with the remainder stored in carry-save format; see Exercise J.33(c)). According to Intel, the bug was due to five incorrect entries in the quotient lookup table.

    a. [12] <J.9, J.12> The bad entries should have had a quotient of plus or minus 2, but instead had a quotient of 0. Because of redundancy, it's conceivable that the algorithm could "recover" from a bad quotient digit on later iterations. Show that this is not possible for the Pentium flaw.

    b. [12] <J.9, J.12> Since the operation is a floating-point divide rather than an integer divide, the SRT division algorithm on page J-45 must be modified in two ways. First, step 1 is no longer needed, since the divisor is already normalized. Second, the very first remainder may not satisfy the proper bound ($|r| \leq 2b/3$ for Pentium; see page J-55). Show that skipping the very first left shift in step 2(a) of the SRT algorithm will solve this problem.

    c. [12] <J.9, J.12> If the faulty table entries were indexed by a remainder that could occur at the very first divide step (when the remainder is the divisor), random testing would quickly reveal the bug. This didn't happen. What does that tell you about the remainder values that index the faulty entries?

J.35 [12] <J.6, J.9> The discussion of the remainder-step instruction assumed that division was done using a bit-at-a-time algorithm. What would have to change if division were implemented using a higher-radix method?

J.36 [25] <J.9> In the array of Figure J.28, the fact that an array can be pipelined is not exploited. Can you come up with a design that feeds the output of the bottom CSA into the bottom CSAs instead of the top one, and that will run faster than the arrangement of Figure J.28?