a. The higher radix makes the guessing of the correct quotient digit more difficult; we certainly do not want to try subtracting $2^k d, 2(2^k d), 3(2^k d)$, etc., and noting the sign of the partial remainder in each case, until the correct quotient digit has been determined—this would nullify all the speed gain (in radix 4, two trial subtractions of d and $2d$ would be needed, thus making each cycle almost twice as long with one adder).

b. Unlike multiplication, where all the partial products can be computed initially and then subjected to parallel processing by multiple carry-save adders, the values to be subtracted from (added to) z in division are determined sequentially, one per cycle. Furthermore, the determination of the quotient digits depends on the magnitude and/or sign of the partial remainder; information that is not readily available from the stored-carry representation.

Thus before discussing high-radix division in depth, we try to solve the more pressing problem of using carry-save techniques to speed up the iterations in binary division. Once we have learned how to use a carry-save representation for the partial remainder, we will revisit the problem of high-radix division. The reason we attach greater importance to the use of carry-save partial remainders than to high-radix division is that in going from radix 2 to radix 4, say, the division is at best speeded up by a factor of 2. The use of carry-save partial remainders, on the other hand, can lead to a larger performance improvement via replacing the delay of a carry-propagate adder by the delay of a single full adder.

The key to being able to keep the partial remainder in carry-save form is introducing redundancy in the representation of the quotient. With a nonredundant quotient, there is no room for error. If the binary quotient is $(0110\cdots)_{\text{two}}$, say, subsequent recovery from an incorrect guess setting the MSB of q to 1 will be impossible. However, if we allow the digit set $[-1, 1]$ for the radix-2 quotient, the partial quotient $(1\cdots)_{\text{two}}$ can be modified to $(1\ \bar{1}\cdots)_{\text{two}}$ in the next cycle if we discover that 1 was too large a guess for the MSB. The aforementioned margin for error allows us to guess the next quotient digit based on the approximate magnitude of the partial remainder. The greater the margin for error, the less precision (fewer bits of the carry-save partial remainder) we need in determining the quotient digits.

14.2 RADIX-2 SRT DIVISION

Let us reconsider the radix-2 nonrestoring division algorithm for fractional operands characterized by the recurrence

$$s^{(j)} = 2s^{(j-1)} - q_{-j}d \quad \text{with} \quad s^{(0)} = z \quad \text{and} \quad s^{(k)} = 2^k s$$

with $q_{-j} \in \{-1, 1\}$. Note that the same algorithm can be applied to integer operands if d is viewed as standing for $2^k d$.

The quotient is obtained with the digit set $\{-1, 1\}$ and is then converted (on the fly) to the standard digit set $\{0, 1\}$. Figure 14.3 plots the new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$. For $2s^{(j-1)} \geq 0$, we subtract the divisor d from $2s^{(j-1)}$ to obtain $s^{(j)}$, while for $2s^{(j-1)} < 0$, we add d to obtain $s^{(j)}$. These actions are represented by the two oblique lines in Fig. 14.3. The heavy dot in Fig. 14.3 indicates the action taken for $2s^{(j-1)} = 0$.

Nonrestoring division with shifting over 0s is a method that avoids addition or subtraction when the partial remainder is "small." More specifically, when $2s^{(j-1)}$ is in the range $[-d, d)$,

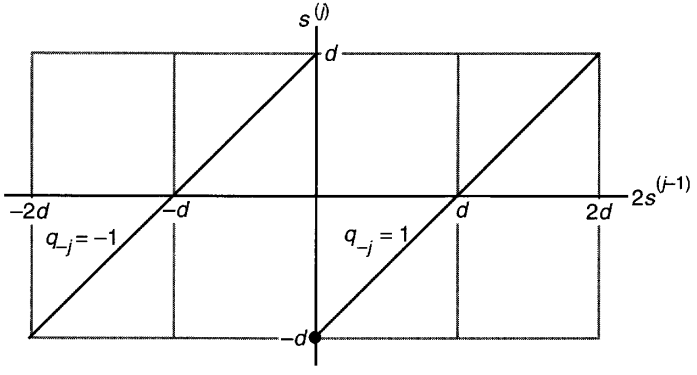


Fig. 14.3 The new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$, in radix-2 nonrestoring division.

we know that the addition/subtraction prescribed by the algorithm will change its sign. Thus, we can choose $q_{-j} = 0$ and only shift the partial remainder. This will not cause a problem because the shifted partial remainder will still be in the valid range $[-2d, 2d)$ for the next step. With this method, the quotient is obtained using the digit set $\{-1, 0, 1\}$, corresponding to "add," "no operation," and "subtract," respectively. Figure 14.4 plots the new partial remainder $s^{(j)}$ as a function of the shifted old partial remainder $2s^{(j-1)}$ for such a modified nonrestoring division algorithm that selects $q_{-j} = 0$ for $-d \leq 2s^{(j-1)} < d$.

Since, with the preceding method, some iterations are reduced to just shifting, one might think that the average division speed will improve in an asynchronous design in which the adder can be selectively bypassed. But how can you tell if the shifted partial remainder is in $[-d, d)$? The answer is that you can't, unless you perform trial subtractions. But the trial subtractions would take more time than they save! An ingenious solution to this problem was independently suggested by Sweeney, Robertson, and Tocher. The resulting algorithm is known as SRT division in their honor.

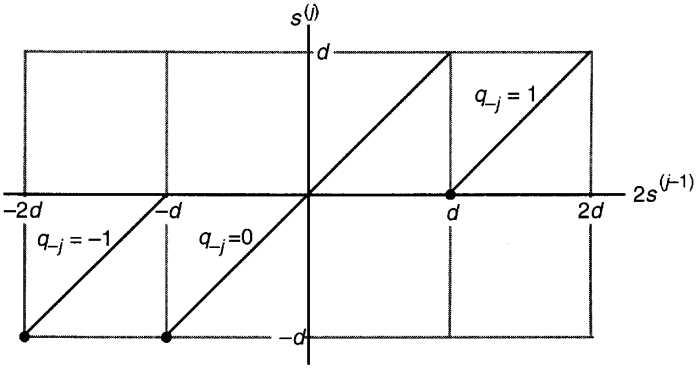


Fig. 14.4 The new partial remainder, $s^{(j)}$, as a function of the shifted old partial remainder, $2s^{(j-1)}$, with q_{-j} in $\{-1, 0, 1\}$.

Let us assume $d \geq 1/2$ (positive bit-normalized divisor) and restrict the partial remainder to the range $[-1/2, 1/2)$ rather than $[-d, d)$. Initially this latter condition might not hold, so we may have to shift the dividend z (which is assumed to be in the range $-d \leq z < d$ if overflow is to be avoided) to the right by one bit. To compensate for this initial right shift, we double the quotient and remainder obtained after $k + 1$ cycles.

Once the initial partial remainder $s^{(0)}$ is adjusted to be in the range $[-1/2, 1/2)$, all subsequent partial remainders can be kept in that range, as evident from the solid rectangle in Fig. 14.5.

The quotient digit selection rule associated with Fig. 14.5 to guarantee that $s^{(j)}$ remains in the range $[-1/2, 1/2)$ is:

```
if 2s^(j-1) < -1/2
then q_-j = -1
else  if 2s^(j-1) ≥ 1/2
      then q_-j = 1
      else q_-j = 0
      endif
endif
```

Two comparisons are still needed to select the appropriate quotient digit, but the comparisons are with the constants $-1/2$ and $1/2$ rather than with $-d$ and d. Comparison with $1/2$ or $-1/2$ is quite simple. When the partial remainder $s^{(j-1)}$ is in $[-1/2, 1/2)$, the shifted partial remainder $2s^{(j-1)}$ will be in $[-1, 1)$, thus requiring 1 bit before the radix point (the sign bit) for its 2's-complement representation.

$$2s^{(j-1)} \geq +1/2 = (0.1)_{2\text{'s-compl}} \quad \text{implies} \quad 2s^{(j-1)} = (0.1u_{-2}u_{-3}\cdots)_{2\text{'s-compl}}$$
$$2s^{(j-1)} < -1/2 = (1.1)_{2\text{'s-compl}} \quad \text{implies} \quad 2s^{(j-1)} = (1.0u_{-2}u_{-3}\cdots)_{2\text{'s-compl}}$$

We see that the condition $2s^{(j-1)} \geq 1/2$ is given by the logical AND term $\overline{u}_0 u_{-1}$ and that of $2s^{(j-1)} < -1/2$ by $u_0\overline{u}_{-1}$. Thus, the required comparisons are preformed by two 2-input AND gates. What could be simpler?

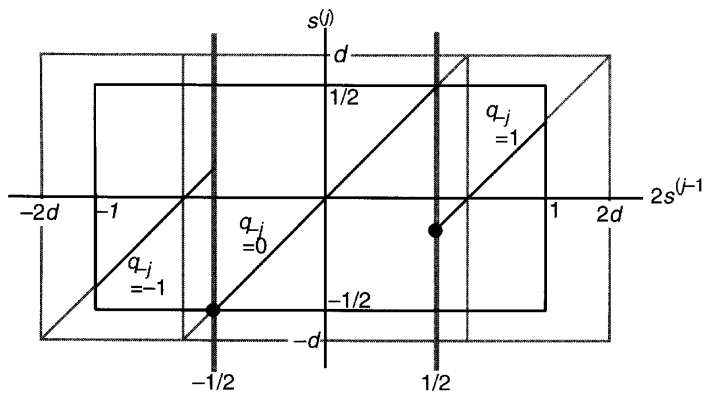


Fig. 14.5 The relationship between new and old partial remainders in radix-2 SRT division.

The nonrestoring divider of Fig. 13.10 is also valid for the SRT algorithm (only the control state machine will change). Everything in the data path portion, including the quotient digit selection logic and the conversion process, remains the same. What the SRT algorithm does is similar to Booth's recoding: it changes an addition (subtraction) followed by a sequence of subtractions (additions) to a number of no-ops followed by a single addition (subtraction); that is, it takes advantage of the equality $\pm(2^j - 2^{j-1} - 2^{j-2} - \cdots - 2^i) = \pm 2^i$.

Figure 14.6 shows an example division performed with the SRT algorithm. The rules for the final correction, if required, are exactly the same as for nonrestoring division, but the quotient conversion algorithm given in Section 13.4 in inapplicable here in view of the presence of 0s in the quotient. One can use an on-the-fly conversion algorithm to convert the BSD quotient to binary [Erce87]. Alternatively, one can have two quotient registers into which the positive and negative digits of q are shifted. The binary version of q, before correction, can then be obtained by a subtraction after all digits have been shifted in.

To further speed up the division process, we can skip over any number of identical leading bits in $s^{(j-1)}$ by shifting. A combinational logic circuit can detect the number of identical leading bits, resulting in significant speedup if a variable shifter is available. Here are two examples:

$s^{(j-1)} = 0.0000110\cdots$ Shift left by 4 bits and subtract
$s^{(j-1)} = 1.1110100\cdots$ Shift left by 3 bits and add

z	.0 1 0 0 0 1 0 1	In [–1/2, 1/2), so OK
d	.1 0 1 0	In [1/2, 1), so OK
$-d$	1.0 1 1 0	
$s^{(0)}$	0.0 1 0 0 0 1 0 1	
$2s^{(0)}$	0.1 0 0 0 1 0 1	≥ 1/2, so set $q_{-1} = 1$
$+(-d)$	1.0 1 1 0	and subtract
$s^{(1)}$	1.1 1 1 0 1 0 1	
$2s^{(1)}$	1.1 1 0 1 0 1	In [–1/2, 1/2), so set $q_{-2} = 0$
$s^{(2)} = 2s^{(1)}$	1.1 1 0 1 0 1	
$2s^{(2)}$	1.1 0 1 0 1	< –1/2, so set $q_{-3} = \bar{1}$
$+d$	0.1 0 1 0	and add
$s^{(3)}$	0.0 1 0 0 1	
$2s^{(3)}$	0.1 0 0 1	≥ 1/2, so set $q_{-4} = 1$
$+(-d)$	1.0 1 1 0	and subtract
$s^{(4)} = 2s^{(3)}$	1.1 1 1 1	Negative,
$+d$	0.1 0 1 0	so add to correct
$s^{(4)}$	0.1 0 0 1	
s	0.0 0 0 0 1 0 0 1	
q	0.1 0 $\bar{1}$ 1	Uncorrected BSD quotient
q	0.0 1 1 0	Convert and subtract *ulp*

Fig. 14.6 Example of unsigned radix-2 SRT division.

When we shift the partial remainder to the left by h bits, the quotient is extended by $h - 1$ zeros and one nonzero digit in $\{-1, 1\}$. In the first example above, the digits 0 0 0 1 must be appended to q, while in the second example, the quotient is extended using the digits 0 0 $\bar{1}$.

Through statistical analysis, the average skipping distance in variable-shift SRT division has been determined to be 2.67 bits. This means that on the average, one add/subtract is performed per 2.67 bits, compared to one per bit in simple nonrestoring division. The result above assumes random bit values in the numbers. However, numbers encountered in practice are not uniformly distributed. This leads to a slight increase in the average shift distance.

Speedup of division by means of standard or variable-shift SRT algorithm is no longer applied in practice. One reason is that modern digital systems are predominantly synchronous. Another, equally important, reason is that in fast dividers, we do not really perform a carry-propagate addition in every cycle. Rather, we keep the partial remainder in stored-carry form, which needs only a carry-save addition in each cycle (see Section 14.3). Now, carry-save addition is so fast that skipping it does not buy us anything; in fact the logic needed to decide whether to skip will have delay comparable to the carry-save addition itself.

14.3 USING CARRY-SAVE ADDERS

Let us set aside SRT division and go back to the radix-2 division scheme with the partial remainders in $[-d, d)$, as represented by Fig. 14.4. However, instead of forcing the selection of $q_{-j} = 0$ whenever $2s^{(j-1)}$ falls in the range $[-d, d)$, we allow the choice of either valid digit in the two overlap areas where the quotient digit can be -1 or 0 and 0 or $+1$ (see Fig. 14.7).

Now, if we want to choose the quotient digits based on comparing the shifted partial remainder to constants, the two constants can fall anywhere in the overlap regions. In particular, we can use the thresholds $-1/2$ and 0 for our decision, choosing $q_{-j} = -1$, 0, or 1 when $2s^{(j-1)}$ falls in the intervals $[-2d, -1/2)$, $[1/2, 0)$, or $[0, 2d)$, respectively. The advantages of these particular comparison constants will become clear shortly.

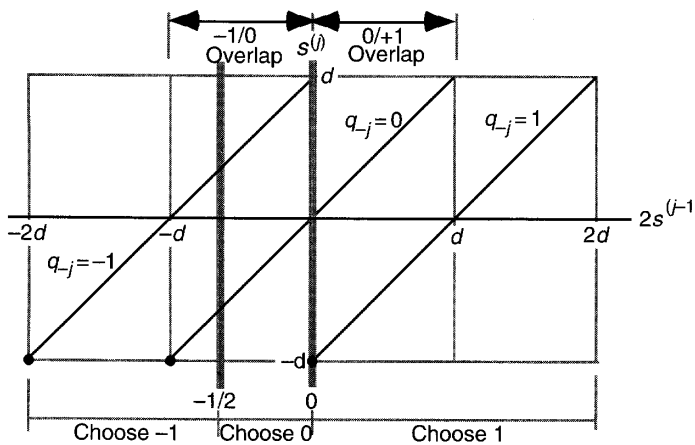


Fig. 14.7 Constant thresholds used for quotient digit selection in radix-2 division with q_{k-j} in $\{-1, 0, 1\}$.

Suppose that the partial remainder is kept in stored-carry form: that is, as two numbers whose sum is equal to the true partial remainder. To perform exact magnitude comparison with such carry-save numbers would require full carry propagation since, in the worst case, the least significant bit values can affect the most significant end of the sum. However, the overlaps in valid ranges of $2s^{(j-1)}$ for selecting $q_{-j} = -1$, 0, or 1 in Fig. 14.7 allow us to perform approximate comparisons without risk of choosing a wrong quotient digit.

Let $u = (u_1u_0.u_{-1}u_{-2}\cdots)_{2\text{'s-compl}}$ and $v = (v_1v_0.v_{-1}v_{-2}\cdots)_{2\text{'s-compl}}$ be the sum and carry components of the stored-carry representation of $2s^{(j-1)}$. Like $2s^{(j-1)}$ itself, each of these components is a 2's-complement number in the range $[-2d, 2d)$. Then the following quotient digit selection algorithm can be devised based on Fig. 14.7:

```
t = u[-2,1] + v[-2,1]   {Add the most significant 4 bits of u and v}
if t < -1/2
then q_-j = -1
else  if t ≥ 0
      then q_-j = 1
      else q_-j = 0
      endif
endif
```

The 4-bit number $t = (t_1t_0.t_{-1}t_{-2})_{2\text{'s-compl}}$ obtained by adding the most significant 4 bits of u and v [i.e., $(u_1u_0.u_{-1}u_{-2})_{2\text{'s-compl}}$ and $(v_1v_0.v_{-1}v_{-2})_{2\text{'s-compl}}$] can be compared to the constants $-1/2$ and 0 based only on the three bit values t_1, t_0, and t_{-1}. If $t < -1/2$, the true value of $2s^{(j-1)}$ is guaranteed to be less than 0, since the error in truncating each component was less than 1/4. Similarly, if $t < 0$, we are guaranteed to have $2s^{(j-1)} < 1/2 \leq d$. Note that when we truncate a 2's-complement number, we always reduce its value independent of the number's sign. This is true because the discarded bits are positively weighted.

The preceding division algorithm requires the use of a 4-bit fast adder to propagate the carries in the high-order 4 bits of the stored-carry shifted partial remainder. Then, the high-order 3 bits of the 4-bit result can be supplied to a logic circuit or an eight-entry table to obtain the next quotient digit. Figure 14.8 is a block diagram for the resulting divider. The 4-bit fast adder to compute t and the subsequent logic circuit or table to obtain q_{-j} are lumped together into the box labeled "Select q_{-j}." Each cycle for this divider entails quotient digit selection, as discussed above, plus only a few logic gate levels of delay through the multiplexer and CSA.

Even though a 4-bit adder is quite simple and fast, we can obtain even better performance by using a 256×2 table in which the 2-bit encoding of the quotient digit is stored for all possible combinations of $4 + 4$ bits from the two components u and v of the shifted partial remainder. Equivalently, an eight-input programmable logic array (PLA) can be used to derive the two output bits using two-level AND-OR logic. This does not affect the block diagram of Fig. 14.8, since only the internal design of the "Select q_{-j}" box will change. The delay per iteration now consists of a table lookup (PLA) plus a few logic levels.

Can we use stored-carry partial remainders with SRT division? Unless we modify the algorithm in some way, the answer is "no." Figure 14.9, derived from Fig. 14.5 by extending the lines corresponding to $q_{-j} = -1$ and $q_{-j} = 1$ inside the solid rectangle, tells us why. The width of each overlap region in Fig. 14.9 is $1 - d$. Thus, the overlaps can become arbitrarily small as d approaches 1, leaving no margin for error and making approximate comparisons impossible.

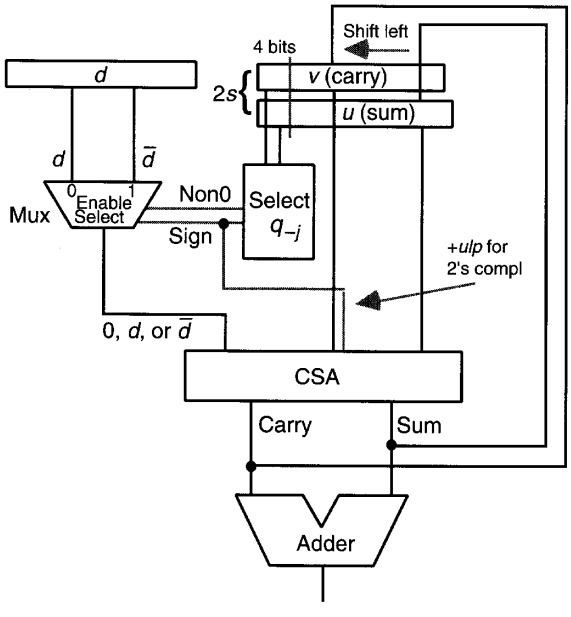


Fig. 14.8 Block diagram of a radix-2 divider with partial remainder in stored-carry form.

14.4 CHOOSING THE QUOTIENT DIGITS

We can use a p-d plot (shifted partial remainder vs. divisor) as a graphical tool for understanding the quotient digit selection process and deriving the needed precision (number of bits to look at) for various division algorithms. Figure 14.10 shows the p-d plot for the radix-2 division, with quotient digits in $[-1, 1]$, depicted in Fig. 14.7. The area between lines $p = -d$ and $p = d$

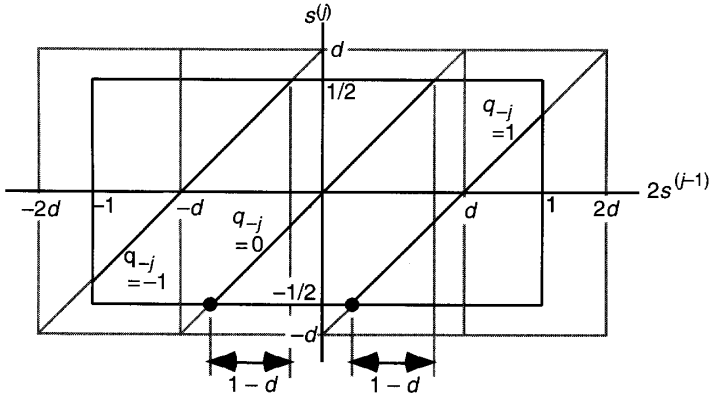


Fig. 14.9 Overlap regions in radix-2 SRT division.

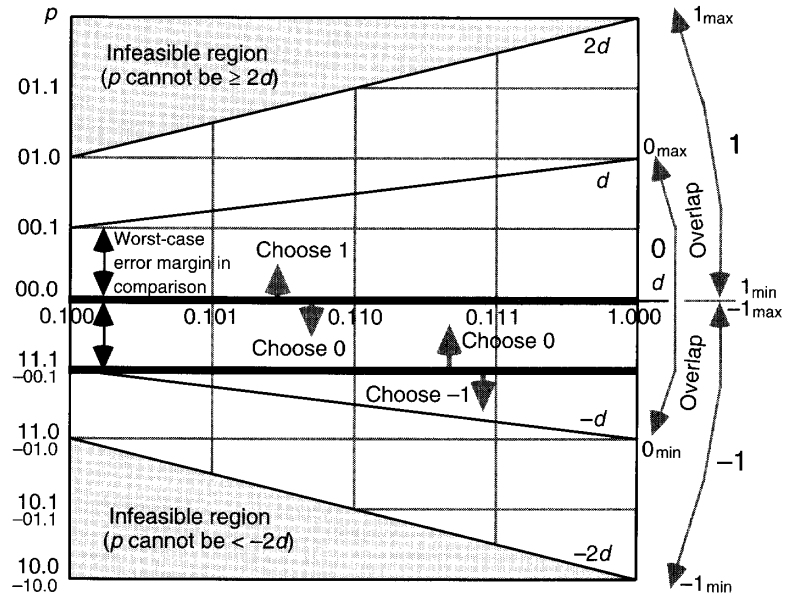


Fig. 14.10 A p-d plot for radix-2 division with $d \in [1/2, 1)$, partial remainder in $[-d, d)$, and quotient digits in $[-1, 1]$.

is the region in which 0 is a valid choice for the quotient digit q_{-j}. Similar observations apply to -1 and 1, whose associated areas overlap with that of $q_{-j} = 0$.

In the overlap regions between $p = 0$ and $p = \pm d$, two valid choices for the quotient digit exist. As noted earlier, placing the decision lines at $p = 0$ and $p = -1/2$ would allow us to choose the quotient digit by inspecting the sign, one integer, and two fractional bits in the sum and carry parts of p. This is because the error margins of 1/2 in the partial remainder depicted in Fig. 14.10 allow us to allocate an error margin of 1/4 in each of its two components. We use an approximate shifted partial remainder $t = (t_1 t_0.t_{-1}t_{-2})_{2\text{'s-compl}}$, obtained by adding 4 bits of the sum and carry components, to select the quotient digit value of 1 when $t_1 = 0$ and -1 when $t_1 = 1$ and t_0 and t_{-1} are not both 1s. Thus the logic equations for the "Non0" and "Sign" signals in Fig. 14.8 become:

$$\text{Non0} = \bar{t}_1 + \bar{t}_0 + \bar{t}_{-1} = \overline{t_1 t_0 t_{-1}}$$
$$\text{Sign} = t_1(\bar{t}_0 + \bar{t}_{-1})$$

Because decision boundaries in the p-d plot of Fig. 14.10 are horizontal lines, the value of d does not affect the choice of q_{-j}. We will see later that using horizontal decision lines is not always possible in high-radix division. In such cases, we embed staircaselike boundaries in the overlap regions that allow us to choose the quotient digit value by inspecting a few bits of both p and d.

Note that the decision process for quotient digit selection is asymmetric about the d axis. This is due to the asymmetric effect of truncation on positive and negative values represented in 2's-complement format.

In our discussions thus far, we have assumed that the divisor d is positive. For a 2's-complement divisor, the p-d plot must be extended to the left to cover negative divisors. If Fig. 14.10 is thus extended for negative values of d, the two straight lines can still be used as decision boundaries, as the value of d is immaterial. However, for staircaselike boundaries just alluded to, the asymmetry observed about the d axis is also present about the p axis. Thus, all four quadrants of the p-d plot must be used to derive the rules for quotient digit selection. Very often, though, we draw only one quadrant of the p-d plot, corresponding to positive values for d and p, with the understanding that the reader can fill in the details for the other three quadrants if necessary.

14.5 RADIX-4 SRT DIVISION

We are now ready to present our first high-radix division algorithm with the partial remainder kept in stored-carry form. We begin by looking at radix-4 division with quotient digit set $[-3, 3]$. Figure 14.11 shows the relationship of new and shifted old partial remainders along with the overlapping regions within which various quotient digit values can be selected.

The p-d plot corresponding to the division algorithm above is shown in Fig. 14.12. For the sake of simplicity, the decision boundaries (heaviest lines) are drawn with the assumption that the exact partial remainder is used in the comparisons. In this example, we see, for the first time, a decision boundary that is not a straight horizontal line. What this means is that the choice between $q_{-j} = 3$ or $q_{-j} = 2$ depends not only on the value of p but also on one bit, d_{-2}, of d (to tell us whether d is in [1/2, 3/4) or in [3/4, 1). If p is only known to us approximately, the selection boundaries must be redrawn to allow for correct selection with the worst-case error in p. More on this later.

When the quotient digit value of ± 3 is selected, one needs to add/subtract the multiple $3d$ of the divisor to/from the partial remainder. One possibility is to precompute and store $3d$ in a register at the outset. Recall that we faced the same problem of needing the multiple $3a$ in radix-4 multiplication. This reminds us of Booth's recoding and the possibility of restricting the quotient digits to $[-2, 2]$, since this restriction would facilitate quotient digit selection (fewer comparisons) and the subsequent multiple generation.

Figure 14.13 shows that we can indeed do this if the partial remainder range is suitably restricted. To find the allowed range, let the restricted range be $[-hd, hd)$ for some $h < 1$. Then, $4s^{(j-1)}$ will be in the range $[-4hd, 4hd)$. We should be able to bring the worst-case values to within the original range by adding $\pm 2d$ to it. Thus, we must have $4hd - 2d \leq hd$ or $h \leq 2/3$. Let us choose $h = 2/3$. As in SRT division, since z may not be in this range, an initial shift and final adjustment of the results may be needed.

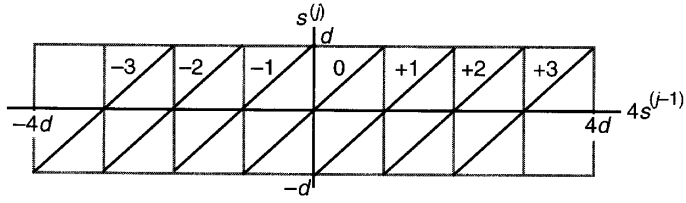


Fig. 14.11 New versus shifted old partial remainder in radix-4 division with q_{-j} in $[-3, 3]$.

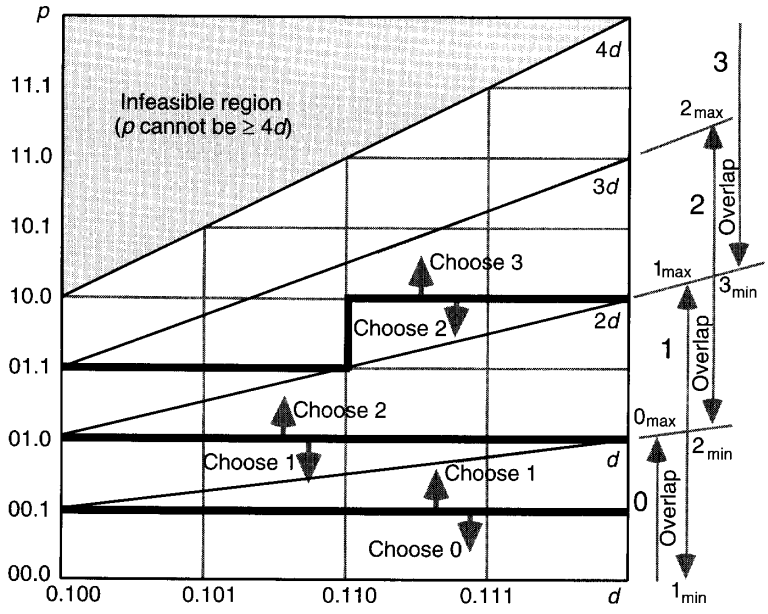


Fig. 14.12 A p-d plot for radix-4 SRT division with quotient digit set $[-3, 3]$.

The p-d plot corresponding to the preceding division scheme is given in Fig. 14.14. Upon comparing Figs. 14.14 and 14.12, we see that restricting the digit set to $[-2, 2]$ has made the overlap regions narrower, forcing us to examine p and d with greater accuracy to correctly choose the quotient digit. On the positive side, we have gotten rid of the $3d$ multiple, which would be hard to generate. Based on staircaselike boundaries in the p-d plot of Fig. 14.14, we see that 4 bits of p (plus its sign) and 4 bits of d must be inspected (d_{-1} also provides the sign information).

The block diagram of a radix-4 divider based on the preceding algorithm is quite similar to the radix-2 divider in Fig. 14.8 except for the following changes:

Four bits of d are also input to the quotient digit selection box.

We need a four-input multiplexer, with "enable" and two select control lines, the inputs to which are d and $2d$, as well as their complements. Alternatively, a two-input multiplexer

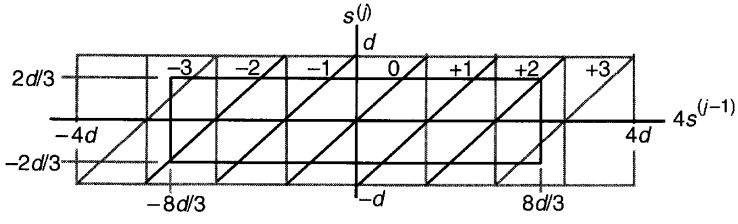


Fig. 14.13 New versus shifted old partial remainder in radix-4 division with q_{-j} in $[-2, 2]$.

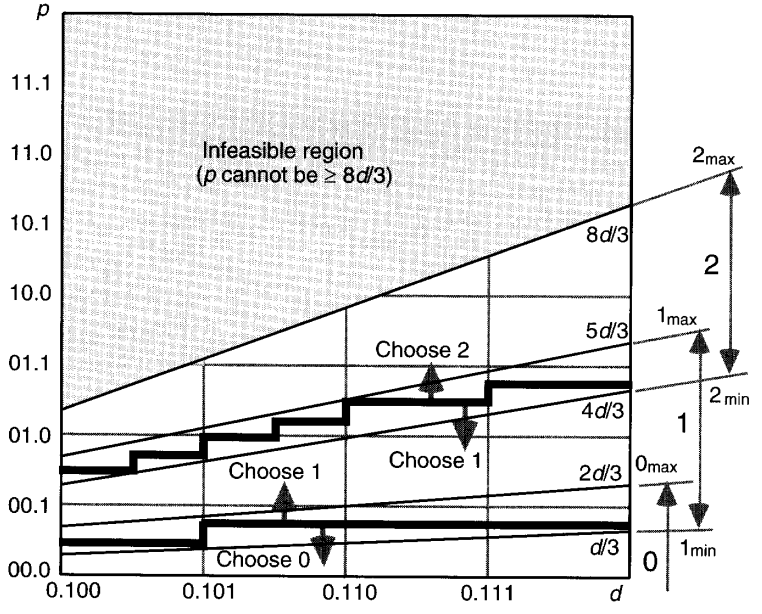


Fig. 14.14 A p-d plot for radix-4 SRT division with quotient digit set $[-2, 2]$.

with "enable" line can be used to choose between 0, d, and $2d$, followed by a selective complementer to produce $-d$ or $-2d$ if needed.

The final conversion of the quotient from radix-4 signed-digit form, with the digit set $[-2, 2]$, to 2's-complement form, is more involved.

Radix-4 SRT division is the division algorithm used in the Intel Pentium processor. The quotient selection box in Pentium's hardware is implemented by a programmable logic array. According to Intel's explanation of the division bug in early Pentium chips, after the p-d plot was numerically generated, a script was written to download the entries into a hardware PLA. An error in this script resulted in the inadvertent removal of a few table entries from the lookup table. These missing entries, when hit, would result in the digit 0, instead of +2, being read out from the PLA [Gepp95].

Unfortunately for Intel, these entries are consulted very rarely, and thus the problem was not caught during the testing of the chip. Fuller explanations of the mathematics behind the Intel Pentium division flaw, and why it was very subtle and difficult to detect, are offered in [Coe95] and [Edel97].

14.6 GENERAL HIGH-RADIX DIVIDERS

Now that we know how to construct a fast radix-4 divider, it is quite easy to generalize the idea to higher radices. For example, a radix-8 divider can be built by restricting the partial remainder in the range $[-4d/7, 4d/7)$ and using the minimal quotient digit set $[-4, 4]$. The required

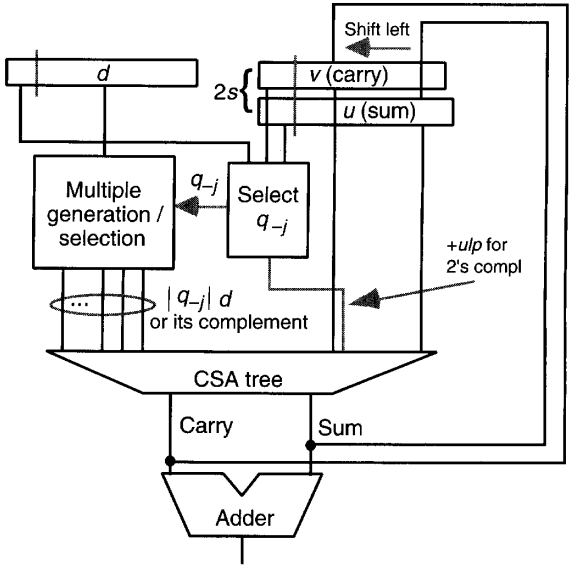


Fig. 14.15 Block diagram of radix-r divider with partial remainder in stored-carry form.

$3d$ multiple can either be precomputed and stored in a register or dynamically produced by selectively supplying $2d$ and d as inputs to a CSA tree that receives the two numbers representing the partial remainder as its other two inputs. Determining the required precision in inspecting the partial remainder and the divisor to select the next quotient digit is left as an exercise.

Digit sets with greater redundancy, such as $[-7, 7]$ in radix 8, are possible and lead to wider overlap regions and, thus, lower precision in the comparisons needed for selecting the quotient digit. However, they also lead to more comparisons and the need to generate other difficult multiples (e.g., ± 5 and ± 7) of the divisor.

The block diagram of a radix-r hardware divider is shown in Fig. 14.15. Note that this radix-r divider is similar to the radix-2 divider in Fig. 14.8, except that its more general multiple generation/selection circuit may produce the required multiple as a set of numbers, and several bits of d are also examined by the quotient digit selection logic. For further details and design issues for high-radix dividers, see Sections 15.1 and 15.2.

PROBLEMS

14.1 Nonrestoring unsigned integer division Given the binary dividend $z = 0110\ 1101\ 1110\ 0111$ and the divisor $d = 1010\ 0111$, perform the unsigned radix-2 division z/d to determine the 8-bit quotient q and 16-bit remainder s, selecting the quotient digits according to

a. Fig. 14.3

b. Fig. 14.4

c. Fig. 14.5

d. Fig. 14.10

14.2 Nonrestoring signed integer division Given the binary 2's-complement operands $z = 1.1010\ 0010\ 11$ and $d = 0.10110$, perform the signed radix-2 division z/d to

determine the 2's-complement quotient $q = q_0.q_{-1}q_{-2}q_{-3}q_{-4}q_{-5}$ and remainder $1.11111s_{-6}s_{-7}s_{-8}s_{-9}s_{-10}$, selecting the quotient digits according to:

a. Fig. 14.3

b. Fig. 14.4

c. Fig. 14.5

d. Fig. 14.10

14.3 Carry-save and high-radix division Perform the division z/d, with $z = 1.1010\ 0010\ 11$ and $d = 0.10110$, using:

a. Radix-2 division, with the partial remainder kept in carry-save form (Fig. 14.7).

b. The radix-4 division scheme depicted in Fig. 14.12.

c. The radix-4 division scheme depicted in Fig. 14.14.

14.4 Robertson diagram for division A Robertson diagram for division is constructed as follows. We take the $s^{(j)}$-versus-$2s^{(j-1)}$ plot of the division algorithm, exemplified by Figs. 14.3–14.5, and mark off the dividend $z = s^{(0)}$ on the vertical axis. We then draw a curved arrow from this point to the point representing $2s^{(0)}$ on the horizontal axis, a vertical arrow from there to the diagonal line representing the quotient digit value, followed by a horizontal arrow to the $s^{(1)}$ point on the vertical axis. If we continue in this manner, the arrows will trace a path showing the variations in the partial remainders and the accompanying quotient digits selected. Construct Robertson diagrams corresponding to the following divisions using the nonrestoring algorithm.

a. $z = +.1001$ and $d = +.0101$

b. $z = +.1001$ and $d = -.0101$

c. $z = -.1001$ and $d = +.0101$

d. $z = -.1001$ and $d = -.0101$

14.5 Restoring binary division

a. Construct a diagram similar to Figs. 14.3–14.5 for restoring division.

b. Draw a Robertson diagram (see Problem 14.4) for the unsigned binary division .101001/.110.

14.6 Radix-4 SRT division

a. Complete Fig. 14.14 by drawing all four quadrants on graph paper.

b. Use rectangular tiles to tile the diagram of part a with dimensions determined by smallest step size in each direction. On each tile, write the quotient digit value(s).

c. If the quotient digit is to be selected by a PLA, rather than a ROM table, adjacent tiles of part b that have identical labels can be merged into a single product term. Combine the tiles to minimize the number of product terms required.

14.7 Radix-4 SRT division Present a complete logic design for the quotient digit selection box of Fig. 14.8, trying to maximize the speed.

14.8 Radix-8 SRT division

a. Draw a *p-d* plot, similar to Fig. 14.14, for radix-8 division using the quotient digit set $[-4, 4]$.

b. Estimate the size of the ROM table needed for quotient digit selection with and without a small fast adder to add a few bits of the stored-carry partial remainder.

14.9 Pentium's division flaw The Intel Pentium division flaw was due to five incorrect entries in the quotient digit lookup table for its radix-4 SRT division algorithm with carry-save partial remainder and quotient digits in $[-2, 2]$. The bad entries should have contained ± 2 but instead contained 0. Because of redundancy, it is conceivable that on later iterations, the algorithm could recover from a bad quotient digit. Show that recovery is impossible for the Pentium flaw.

14.10 Division with shifting over 0s and 1s

a. Assuming uniform distribution of 0 and 1 digits in the dividend, divisor, and all intermediate partial remainders, determine the expected shift amount if division is performed by shifting over 0s and 1s, as discussed at the end of Section 14.2.

b. Arbitrarily long shifts require the use of a complex shifter. What would be the expected shift amount in part a if the maximum shift is limited to 4 bits?

c. Repeat part b with maximum shift limited to 8 bits and discuss whether increasing the maximum shift to 8 bits would be cost-effective.

d. Explain the difference between the result of part a and the 2.67-bit average shift mentioned near the end of Section 14.2.

14.11 Conversion of redundant quotients A redundant radix-r quotient resulting from high-radix division needs to be converted to standard representation at the end of the division process.

a. Show how to convert the BSD quotient of SRT division to 2's-complement.

b. To avoid a long conversion delay on the critical path of the divider, one can use on-the-fly conversion [Erce87]. Show that by keeping two standard binary versions of the quotient and updating them appropriately as each quotient digit is chosen in $[-1, 1]$, one can obtain the final 2's-complement quotient by simple selection from one of the two registers.

c. Repeat part a for radix-4 SRT algorithm with the digit set $[-2, 2]$.

d. Repeat part b for radix-4 SRT algorithm with the digit set $[-2, 2]$.

14.12 Radix-3 division

a. Develop an algorithm for unsigned radix-3 division with standard operands (i.e., digit set $[0, 2]$) and the quotient obtained with the redundant digit set $[-2, 2]$.

b. Repeat part a when the inputs are signed radix-3 numbers using the symmetric digit set $[-1, 1]$.

14.13 SRT division with $2d$ and $d/2$ multiples The following method has been suggested to increase the average shift amount, and thus the speed, of SRT division. Suppose we shift over 0s in a positive partial remainder. In the next step, corresponding to

a 1 digit in the partial remainder, we choose the quotient digit 1 and subtract d. If the partial remainder is much larger than the divisor, the 1 in the quotient will be followed by other 1s, as in $\cdots$ 0000111 $\cdots$, necessitating several subtractions. In this case, we can subtract $2d$ instead of d, which is akin to going back and "correcting" the previous 0 digit in the quotient to 1 and setting the current digit to 0 in order to produce a small negative partial remainder and thus a larger shift. On the other hand, if the partial remainder is much smaller than the divisor, the 1 in the quotient will be followed by -1s, and thus one or more additions. In this case, it is advantageous to subtract $d/2$ rather than d, which corresponds to picking the current and next quotient digits to be 01.

a. Construct an 8×8 table in which, for the various combination of values in the upper 4 bits of d and s, you indicate whether $d/2$, d, or $2d$ should be subtracted. Assume that d is of the form .1xxx and s is positive.

b. Extend the table in part a to negative partial remainders.

c. Use the table of part b to perform the example division z/d with 2's-complement operands $z = 1.1010\ 0010\ 11$ and $d = 0.10110$.

14.14 Radix-2 division with over-redundant quotient Consider radix-2 division with the "over-redundant" [Srin97] quotient digit set $[-2, 2]$.

a. Draw a p-d plot for this radix-2 division.

b. Show that inspecting the sign and two digits of the partial remainder (three if in carry-save form) is sufficient for determining the next quotient digit.

c. Devise a method for converting the over-redundant quotient to binary signed-digit using the digit set $[-1, 1]$ as the first step of converting it to standard binary. *Hint:* When a quotient digit is ± 2, the next digit must be 0 or of the opposite sign. Rewrite a digit ± 2 as ± 1, with a right-moving "carry" of ± 2.

14.15 Decimal division The quotient digit set $[-\alpha, \alpha]$ can be used to perform radix-10 division.

a. Determine the minimally redundant quotient digit set if the next quotient digit is to be determined based on 1 decimal digit each from the partial remainder and divisor.

b. Present a design for the decimal divider, including its quotient digit selection box.

c. Assume that the decimal partial remainder is kept in carry-save form (i.e., using the digit set [0, 10]). How does this change affect the quotient digit selection logic?

REFERENCES

[Atki68] Atkins, D. E., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Trans. Computers*, Vol. 17, No. 10, pp. 925–934, 1968.

[Coe95] Coe, T., and P. T. P. Tang, "It Takes Six Ones to Reach a Flaw," *Proc. 12th Symp. Computer Arithmetic*, July 1995, pp. 140–146.

[Edel97] Edelman, A., "The Mathematics of the Pentium Division Bug," *SIAM Rev.*, Vol. 39, No. 1, pp. 54–67, March 1997.

[Erce87] Ercegovac, M. D., and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," *IEEE Trans. Computers*, Vol. 36, No. 7, pp. 895–897, 1987.

[Frei61] Freiman, C. V., "Statistical Analysis of Certain Binary Division Algorithms," *Proc. IRE*, Vol. 49, No. 1, pp. 91–103, 1961.

[Gepp95] Geppert, L., "Biology 101 on the Internet: Dissecting the Pentium Bug," *IEEE Spectrum*, pp. 16–17, 1995.

[Robe58] Robertson, J. E., "A New Class of Digital Division Methods," *IRE Trans. Electronic Computers*, Vol. 7, pp. 218–222, September 1958.

[Srin97] Srinivas, H. R., K. K. Parhi, and L. A. Montalvo, "Radix 2 Division with Over-Redundant Quotient Selection," *IEEE Trans. Computers*, Vol. 46, No. 1, pp. 85–92, 1997.

[Tayl85] Taylor, G. S., "Radix-16 SRT Dividers with Overlapped Quotient Selection Stages," *Proc. 7th Symp. Computer Arithmetic*, pp. 64–71, 1985.

[Toch58] Tocher, K. D., "Techniques of Multiplication and Division for Automatic Binary Computers," *Quarterly J. Mechanics and Applied Mathematics*, Vol. 11, Pt. 3, pp. 364–384, 1958.

Chapter 15 VARIATIONS IN DIVIDERS

In this chapter, we cover some practical aspects in implementing high-radix dividers. We also deal with prescaling methods, modular dividers, and array dividers. Chapter 12, entitled "Variations in Multipliers," covered the special case of squaring. It may appear, therefore, that a discussion of square-rooting belongs in this chapter. However, square-rooting, though quite similar to division, is not its special case. We will deal with square-rooting methods in Chapter 21. Chapter topics include:

15.1 QUOTIENT DIGIT SELECTION REVISITED

In the first two sections of this chapter, we elaborate on the quotient digit selection process and the practical use of *p-d* plots for high-radix division.

The dotted portion of Fig. 15.1 defines radix-r SRT division where the partial remainder s is in $[-d, d)$, the shifted partial remainder is in $[-rd, rd)$, and quotient digits are in $[-(r-1), r-1]$. Radix-4 division with the quotient digit set $[-3, 3]$, discussed in Section 14.5, is an example of this general scheme.

Consider now radix-r division with the symmetric quotient digit set $[-\alpha, \alpha]$, where $\alpha < r-1$. Because of the restriction on quotient digit values, we need to restrict the partial remainder range, say to $[-hd, hd)$, to ensure that a valid quotient digit value always exists. From the solid rectangle in Fig. 15.1, we can easily derive the condition $rhd - \alpha d \leq hd$ or, equivalently, $h \leq \alpha/(r-1)$. To minimize the restriction on range, we usually choose:

$$h = \frac{\alpha}{r-1}$$

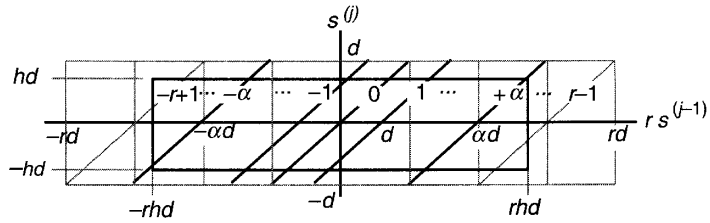


Fig. 15.1 The relationship between new and shifted old partial remainders in radix-r division with quotient digits in $[-\alpha, +\alpha]$.

As a special case, $r = 4$ and $\alpha = 2$ lead to $h = 2/3$ and the range $[-2d/3, 2d/3)$ for the partial remainder (see Fig. 14.13). Note that since $\alpha \geq r/2$, we have $h > 1/2$. Thus, a 1-bit right shift is always enough to ensure that $s^{(0)}$ is brought to within the required range at the outset.

The p-d plot is a very general and useful tool. Even though thus far we have assumed that d is in the range [1/2, 1), this does not have to hold, and we can easily draw a p-d plot in which d ranges from any $d^{\min}$ to any $d^{\max}$ (e.g., from 1 to 2 for IEEE floating-point significands, introduced in Chapter 17). Figure 15.2 shows a portion of a p-d plot with this more general view of d.

With reference to the partial p-d plot depicted in Fig. 15.2, let us assume that inspecting 4 bits of p and 3 bits of d places us at point A. Because of truncation, the point representing the actual values of p and d can be anywhere inside the rectangle attached to point A. As long as the entire area of this "uncertainty rectangle" falls within the region associated with β or $\beta + 1$, there is no problem. So, at point A, we can confidently choose $q_{-j} = \beta + 1$ despite the uncertainty.

Now consider point B in Fig. 15.2 and assume that 3 bits of p and 4 bits of d are inspected. The new uncertainty rectangle drawn next to point B is twice as tall and half as wide and contains points for which each of the values β or $\beta + 1$ is the only correct choice. In this case, the ambiguity cannot be resolved and a choice for q_{-j} that is valid within the entire rectangle does not exist.

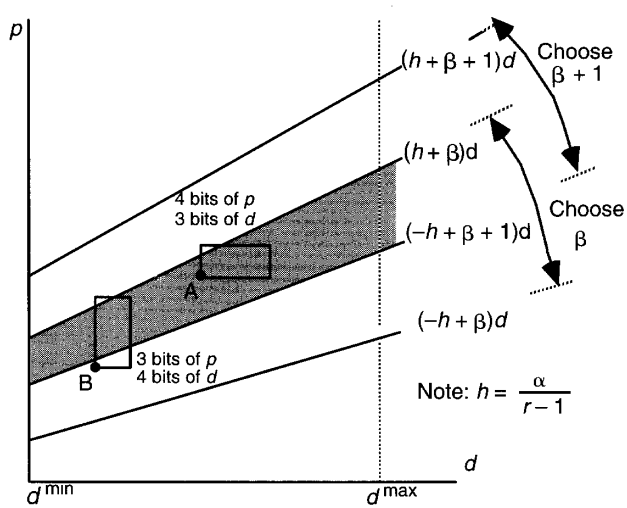


Fig. 15.2 A part of p-d plot showing the overlap region for choosing the quotient digit value β or $\beta + 1$ in radix-r division with quotient digit set $[-\alpha, \alpha]$.

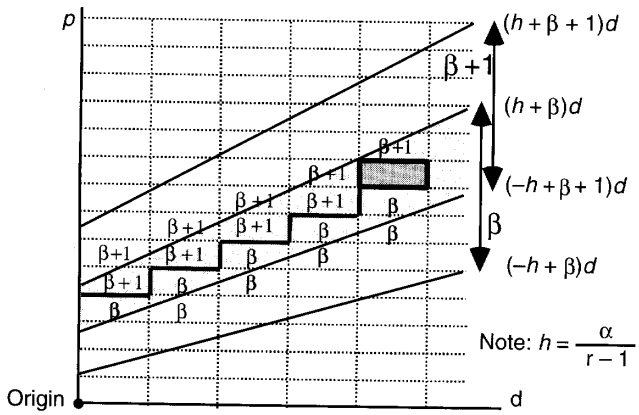


Fig. 15.3 A part of *p-d* plot showing an overlap region and its staircaselike selection boundary.

In practice, we want to make the uncertainty rectangle as large as possible, to minimize the number of bits in p and d needed for choosing the quotient digits. To determine whether uncertainty rectangles of a given size (say the one shown at point A in Fig. 15.2) are admissible, we tile the entire *p-d* plot with the given rectangle beginning at the origin (see Fig. 15.3). Next we verify that no tile intersects both boundaries of an overlap region (touching one boundary, while intersecting another one, is allowed). This condition is equivalent to being able to embed a staircaselike path, following the tile boundaries, in each overlap region (Fig. 15.3).

If the tiling is successful, we complete the process by associating a quotient digit value with each tile. This value is the table entry corresponding to the lower left corner point of the tile. When there is a choice, as is the case for the dark tile in Fig. 15.3, we use implementation- and technology-dependent criteria to pick one value over the other. More on this later.

In the preceding discussion, the partial remainder was assumed to be in standard binary form. If p is in carry-save form, then to get l bits of accuracy for p, we need to inspect $l + 1$ bits in each of its two components. Hence to simplify the selection logic (or size of the lookup table), we try to maximize the height of the uncertainty rectangle. For example, if both rectangles shown in Fig. 15.2 represented viable choices for the precision required of p and d, then the one associated with point B would be preferable (the quotient digit is selected based on $3+3+4 = 10$ bits, rather than $4 + 4 + 3 = 11$ bits, of information).

15.2 USING *p-d* PLOTS IN PRACTICE

Based on the preceding discussion, the goal of the designer of a high-radix divider is to find the coarsest possible grid (the dotted lines in Fig. 15.3) such that staircaselike boundaries, entirely contained within each of the overlap areas, can be built. Unfortunately, there is no closed-form formula for the required precisions, given the parameters r and α and the range of d. Thus, the process involves some trial and error, with the following analytical results used to limit the search space.

Consider the staircase embedded in the narrowest overlap area corresponding to the overlap between the digit values α and $\alpha - 1$. The minimum horizontal and vertical distances between the lines $(-h + \alpha)d$ and $(h + \alpha - 1)d$ place upper bounds on the dimensions of uncertainty rectangles (why?). From Fig. 15.4, these bounds, Δd and Δp, can be found:

$$\Delta d = d^{\min}\frac{2h-1}{-h+\alpha}$$
$$\Delta p = d^{\min}(2h-1)$$

For example, in radix-4 division with the divisor range $[1/2, 1)$ and the quotient digit set $[-2, 2]$, we have $\alpha = 2$, $d^{\min} = 1/2$, and $h = \alpha/(r-1) = 2/3$. Therefore:

$$\Delta d = (1/2)\frac{4/3-1}{-2/3+2} = 1/8$$
$$\Delta p = (1/2)(4/3-1) = 1/6$$

Since $1/8 = 2^{-3}$ and $2^{-3} \leq 1/6 < 2^{-2}$, at least 3 bits of d (2, excluding its leading 1) and 3 bits of p must be inspected. These are lower bounds, and they may turn out to be inadequate. However, they help us limit the search to larger values only. Constructing a detailed p-d plot on graph paper for the preceding example shows that in fact 3 bits of p and 4 (3) bits of d are required. If p is kept in carry-save form, then 4 bits of each component must be inspected (or first added in a small fast adder to give the high-order 3 bits).

The entire process discussed thus far, from determining lower bounds on the precisions required to finding the actual precisions along with table contents or PLA structure, can be easily automated. However, the Intel Pentium bug teaches us that the results of such an automated design process must be rigorously verified.

So far, our p-d plots have been mostly limited to the upper right quadrant of the plane (nonnegative p and d). Note that even if we divide unsigned numbers, p can become negative in the course of division. So, we must consider at least one other quadrant of the p-d plot. We emphasize that the asymmetric effect of truncation of positive and negative values in 2's-complement format prevents us from using the same table entries, but with opposite signs, for the lower right quadrant.

To justify the preceding observation, consider point A, with coordinates d and p, along with its mirror image B, having coordinates d and $-p$ (Fig. 15.5). We see, from Fig. 15.5, that the quotient digit value associated with point B is not the negative of that for point A. So the table size must be expanded to include both (all four) quadrants of the p-d plot. To account for the sign information, one bit must be added to the number of bits inspected in both d and p.

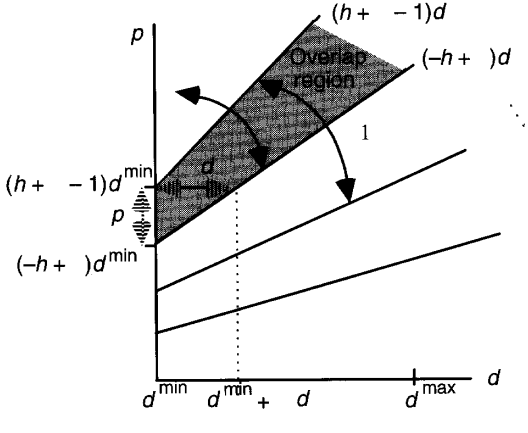


Fig. 15.4 Establishing upper bounds on the dimensions of uncertainty rectangles.

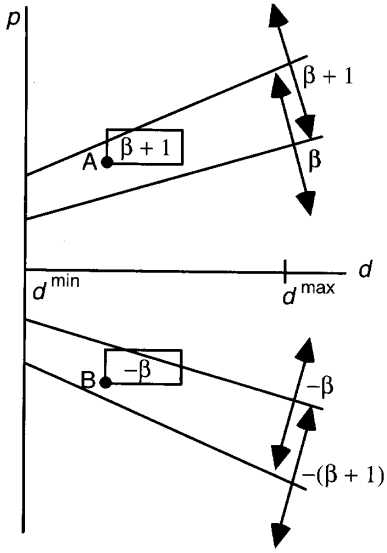


Fig. 15.5 The asymmetry of quotient digit selection process.

Occasionally, we have a choice of two different quotient digit values for a given tile in the *p-d* plot (dark tiles in Figs. 15.3 and 15.6). In a full table-lookup implementation of the quotient digit selection box, the choice has no implication for cost or delay. With a PLA implementation, however, such entries constitute partial don't-cares and can lead to logic simplification. The extent of simplification is of course dependent on the encoding used for the quotient digit.

In practice, one might select a lower precision that is "almost" good enough in the sense that only a few uncertainty rectangles are not totally contained within the region of a single quotient digit. These exceptions are then handled by including more inputs in their corresponding product terms. For the portion of the *p-d* plot shown in Fig. 15.6, the required precision can be reduced by one bit for each component (combining four small tiles into a larger tile), except for the four small tiles marked with asterisks.

For instance, if 3 bits of p in carry-save form ($u_{-1}, u_{-2}, u_{-3}, v_{-1}, v_{-2}, v_{-3}$) and 2 bits of d (d_{-2}, d_{-3}) are adequate in most cases, with d_{-4} also needed occasionally, the logical expression for each of the PLA outputs will consist of the sum of product terms involving eight variables in true or complement form. The ninth variable is needed in only a few of the product terms; thus its effect on the complexity of the required PLA is small.

15.3 DIVISION WITH PRESCALING

By inspecting Fig. 15.6 (or any of the other *p-d* plots that we have encountered thus far), one may observe that the overlap regions are wider toward the high end of the divisor range. Thus, if we can restrict the magnitude of the divisor to an interval close to $d^{\max}$(say $1 - \varepsilon < d < 1 + \delta$, when $d^{\max} = 1$), the selection of quotient digits may become simpler; that is, it may be based on inspecting fewer bits of p and d or perhaps even made independent of d altogether.

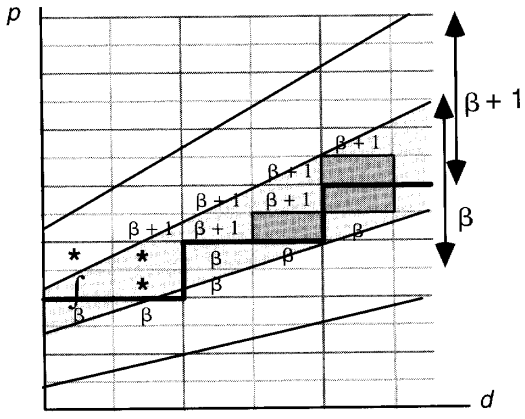


Fig. 15.6 Example of p-d plot allowing larger uncertainty rectangles, if the four cases marked with asterisks are handled as exceptions.

The preceding goal can be accomplished by performing the division $(zm)/(dm)$, instead of z/d, for a suitably chosen scale factor m ($m > 1$). Multiplying both the dividend and the divisor by a factor m to put the divisor in the restricted range $(1 - \varepsilon, 1 + \delta)$ is called "prescaling."

Of course for an arbitrary scaling factor, two multiplications would be required to find the scaled dividend and divisor. The trick is to accomplish the scaling through addition. A reasonable restriction, to keep the time and hardware overhead of prescaling to a minimum, is to require that only one pass through the hardware circuit that performs the division iterations be used for scaling each operand. In this way, we essentially use two additional cycles in the division process (one for scaling each operand). Since simpler quotient selection logic makes each iteration simpler and thus faster, a net gain in speed may result despite the extra cycles.

For example, in radix-8 division of 60-bit fractions, the number of iterations required is increased by 10% (from 20 to 22). A reduction of 20%, say, in the delay of each iteration would lead to a net gain of 12% in division time.

A main issue in the design of division algorithms with prescaling is the choice of the scaling factors. Consider the high-radix divider shown in Fig. 14.15: except that the partial remainder is kept as a single number rather than in stored-carry form. In the new arrangement, the carry-propagate adder is used in each cycle, with its output loaded into the partial remainder register. If the multiple generation/selection circuit provides h inputs to the CSA tree, then each division cycle essentially consists of an $(h + 1)$-operand addition. Let the scaling factor m be represented in radix 4 as $m = (m_0.m_{-1}m_{-2}\cdots m_{-h})_{\text{four}}$ using the digit set $[-1, 2]$; in fact, m_0 can be further restricted to $[1, 2]$. Then, the scaled divisor $m \times d$ can be computed by the $(h + 1)$-operand summation

$$m_0 d + 4^{-1}m_{-1}d + 4^{-2}m_{-2}d + \cdots + 4^{-h}m_{-h}d$$

Each of the $h + 1$ terms is easily obtained from d by shifting. The m_j values can be read out from a table based on a few most significant bits of d.

Consider an example with $h = 3$. If we inspect only 4 bits of d (beyond the mandatory 1) and they happen to be 0110, then $d = (0.10110\cdots)_{\text{two}}$ is in the range [11/16, 23/32). To put the

scaled divisor as close to 1 as possible, we can pick the scale factor to be $m = (1.\ 2^{-1}\ 2^{-1})_{\text{four}} = 91/64$. The scaled divisor will thus be in $[1001/1024, 2093/2048)$ or $[1 - 23/1024, 1 + 45/2048)$. For more detail and implementation considerations, see [Erce94].

15.4 MODULAR DIVIDERS AND REDUCERS

Given a dividend z and divisor d, with $d \geq 0$, a modular divider computes

$$q = \lfloor z/d \rfloor \quad \text{and} \quad s = z \bmod d = \langle z \rangle_d$$

Note that the quotient q is, by definition, an integer, but the inputs z and d do not have to be integers. For example, we have:

$$\lfloor -3.76/1.23 \rfloor = -4 \quad \text{and} \quad \langle -3.76 \rangle_{1.23} = 1.16$$

When z is positive, modular division is the same as ordinary integer division. Even when z and d are fixed-point numbers with fractional parts, we can apply an integer division algorithm to find q and s (how?). For a negative dividend z, however, ordinary division yields a negative remainder s, whereas the remainder (residue) in modular division is always positive. Thus, in this case, we must follow the division iterations with a correction step (adding d to the remainder and subtracting 1 from the integer quotient) whenever the final remainder is negative.

Often the aim of modular division is determining only the quotient q, or only the remainder s, with no need to obtain the other result. When only q is needed, we still have to perform a normal division; the remainder is obtained as a by-product of computing q. However, the computation of $\langle z \rangle_d$, which is referred to as modular reduction, might be faster or need less work than a full-blown division.

We have already discussed modular reduction for a constant divisor d in connection with obtaining the RNS representation of binary or decimal numbers (Section 4.3). Consider now the computation of $\langle z \rangle_d$ for arbitrary $2k$-bit dividend z and k-bit divisor d (both unsigned integers). The $2k$-bit dividend z can be decomposed into k-bit parts z_H and z_L, leading to:

$$\langle z \rangle_d = \langle z_H 2^k + z_L \rangle_d = \langle z_H (2^k - 1) + z_H + z_L \rangle_d$$

Thus, modular reduction can be converted to mod-d multiplication of z_H by $2^k - 1$ (see Section 12.4) and a couple of modular additions. This might be an attractive option if a fast modular multiplier is already available. One of the two additive terms, z_H or z_L, can be accommodated by using it as the initial value of the cumulative partial product. Both additive terms can be accommodated initially if the modular multiplier uses a stored-carry cumulative partial product.

If d is bit-normalized (its MSB is 1), then:

$$\langle 2^k \rangle_d = 2^k - d = \text{ 2's-complement of } d$$

Thus, in this case, $\langle z \rangle_d$ can be computed by mod-d multiplication of z_H and $2^k - d$, with the cumulative partial product initialized to z_L.

Of course, the preceding methods are relevant only if we do not have, or need, a fast hardware divider.

15.5 ARRAY DIVIDERS

Cells and structure very similar to those of array multipliers, discussed in Section 11.5, can be used to build an array divider. Figure 15.7 shows a restoring array divider built of controlled subtractor cells. Each cell has a full subtractor (FS) and a two-input multiplexer. When the control input broadcast to the multiplexers in a row of cells is 0, the cells' vertical inputs (bits of the partial remainder) are passed down unchanged. Otherwise, the diagonal input (divisor) is subtracted from the partial remainder and the difference is passed down. Note that the layout of the cells in Fig. 15.7 resembles the layout of dots in the dot notation view of division, exemplified by Fig. 13.1.

Effectively, each row of cells performs a trial subtraction, with the sign of the result determining the next quotient digit as well as whether the original partial remainder or the trial difference is to be forwarded to the next row. For practical hardware implementation, a faster cell can be built by merging the function of the multiplexer with that of the full subtractor.

The similarity of the array divider of Fig. 15.7 to an array multiplier is somewhat deceiving. The same number of cells is involved in both designs, and the cells have comparable complexities. However, the critical path in a $k \times k$ array multiplier contains $O(k)$ cells, whereas in Fig. 15.7 the critical path passes through all k^2 cells. This is because the borrow signal ripples in each row. Thus, an array divider is quite slow, and, given its high cost, not very cost-effective.

If many divisions are to be performed, pipelining can be applied to improve the throughput of the array divider. For example, if latches are inserted on the output lines for each row of cells in Fig. 15.7, the input data rate will be dictated by the delay associated with borrow propagation in a single row. Thus, with pipelining, the array divider of Fig. 15.7 becomes much more cost-effective, though it will still be slower than its pipelined array multiplier counterpart.

Figure 15.8 depicts a nonrestoring array divider. The cells have roughly the same complexity as the controlled subtractor cells of Fig. 15.7, but more of them are used to handle the extra sign

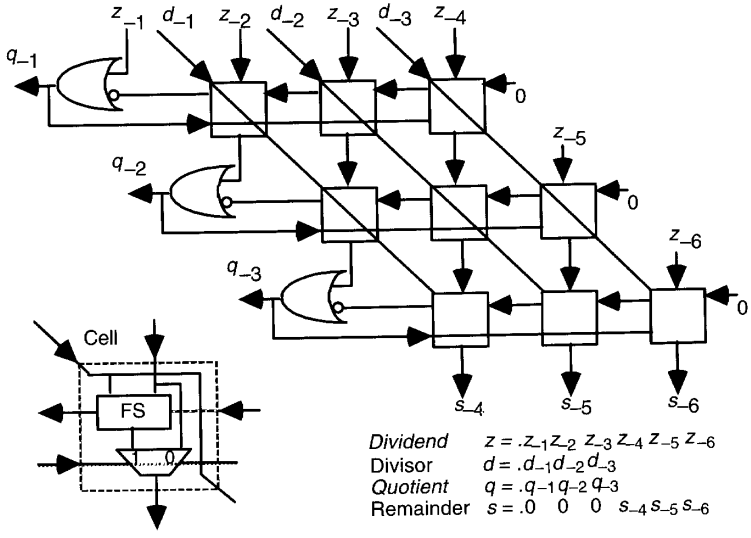


Fig. 15.7 Restoring array divider composed of controlled subtractor cells.

position and the final correction of the partial remainder (last row of cells). The XOR gate in the cells of Fig. 15.7 acts as a selective complementer that passes the divisor or its complement to the full adder, thus leading to addition or subtraction being performed, depending on the sign of the previous partial remainder. The delay is still $O(k^2)$, and considerations for pipelining remain the same as for the restoring design.

Several techniques are available for reducing the delay of an array divider, but in each case additional complexity is introduced into the design. Therefore, none of these methods has found significant practical applications.

To obviate the need for carry/borrow propagation in each row, the partial remainder can be passed between rows in carry-save form. However, we still need to know the carry-out or borrow-out resulting from each row in order to determine the action to be performed in the following row (subtract vs. do not subtract in Fig. 15.7 or subtract vs. add in Fig. 15.8). This can be accomplished by using a carry- (borrow-) lookahead circuit laid out between successive rows of the array divider. However, in view of their need for long wires, the tree-structured lookahead circuits add considerably to the layout area and nullify some of the speed advantages of the original regular layout with localized connections.

Alternatively, a radix-2 or high-radix SRT algorithm can be used to estimate the quotient digit from a redundant digit set, using only a few of the most significant bits of the partial remainder and divisor. This latter approach may simplify the logic to be inserted between rows, but necessitates a more complex conversion of the redundant quotient to standard binary. Even though the wires required for this scheme are shorter than those for a lookahead circuit, they tend to make the layout irregular and thus less efficient.

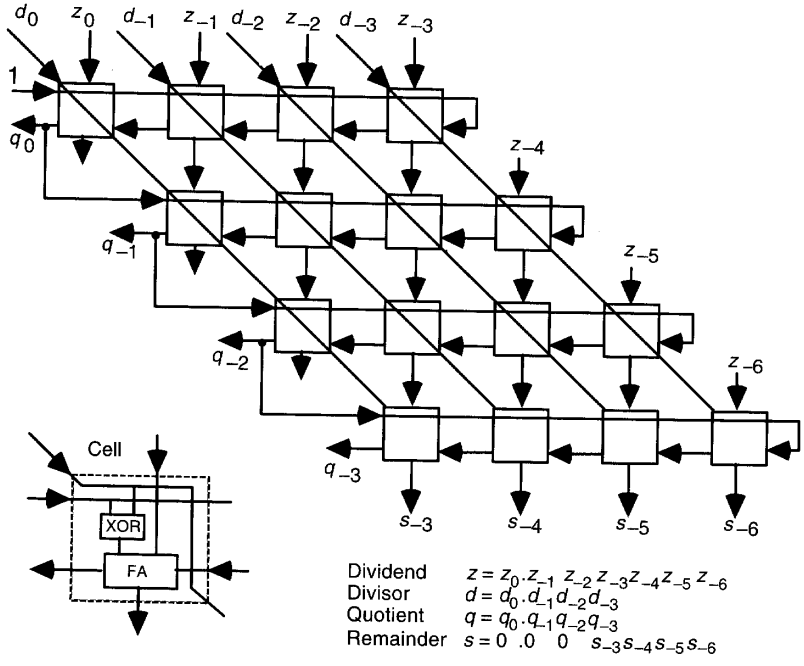


Fig. 15.8 Nonrestoring array divider built of controlled add/subtract cells.

15.6 COMBINED MULTIPLY/DIVIDE UNITS

Except for the quotient digit selection logic in dividers, which has no counterpart in multipliers, the required hardware elements for multipliers and dividers are quite similar. This similarity, which extends from basic radix-2 units, through high-radix designs, to array implementations, stems from the fact that both multiplication and division are essentially multioperand addition problems.

It is thus quite natural to combine multiplication and division capabilities into a single unit. Often, a capability for square-rooting is also included in the unit, since it too requires the same hardware elements (see Chapter 21). Such combined designs are desirable when the volume of numerical computations in expected applications does not warrant the inclusion of separate dedicated multiply and divide units. Even in a high-performance CPU optimized for applications with heavy use of multiplications and divisions, the use of two combined multiply/divide units, say, provides more opportunities for concurrent execution than separate multiply and divide units.

Figure 15.9 shows a radix-2 multiply/divide unit obtained by merging the multiplier of Fig. 9.4 with the nonrestoring divider of Fig. 13.10. The reader should be able to understand all elements in Fig. 15.9 by referring to the aforementioned figures and their accompanying descriptions. Note that the multiplier (quotient) register has been merged with the partial product (remainder) register, with their shifting boundary shown by a dotted line. Another difference is that the extra flip-flop in Fig. 13.10, used to hold the MSB of $2s^{(j-1)}$ has been incorporated into the multiply/divide control unit logic.

A similar merging of high-radix multipliers and dividers leads to combined high-radix multiply/divide units. For example, a radix-4 multiplier with Booth's recoding (Fig. 10.9) can be merged with a radix-4 SRT divider based on the quotient digit set $[-2, 2]$ (Fig. 14.8, modified for radix-4 division, as suggested near the end of Section 14.5) to yield a radix-4 multiply/divide unit. Since the recoded multiplier and the redundant quotient use the same digit set $[-2, 2]$, much of the multiple selection circuitry for the multiplicand and divisor can be shared. Supplying the block diagram and design details is left as an exercise.

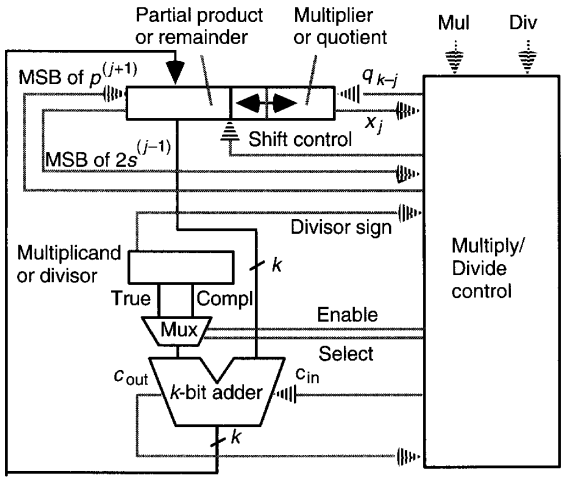


Fig. 15.9 Sequential radix-2 multiply/divide unit.

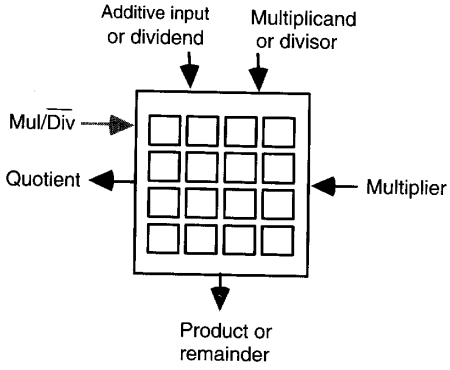


Fig. 15.10 I/O specification of a universal circuit that can act as an array multiplier or array divider.

Merging of partial- or full-tree multipliers with very-high-radix dividers is also possible. One way is to use the multioperand addition capability of the multiplier's partial or full tree to generate a reasonably accurate estimate for the divisor reciprocal $1/d$. This initial step is then followed by a small number of multiplications to produce the quotient q. Division algorithms based on multiplication are discussed in depth in Chapter 16.

Because of the similarity of a nonrestoring array divider (Fig. 15.8) to an array multiplier (Fig. 11.13), it is possible to design a universal circuit that can act as an array multiplier or divider depending on the value of a control input. Figure 15.10 shows a high-level view of such a circuit that also accepts an additive input for multiplication. The cells now become more complex than their array multiplier or divider counterparts, but the universality of the design obviates the need for separate circuits for multiplication and division. In an early universal pipelined array design of this type [Kama74], squaring and square-rooting were also included among the functions that could be performed. The array consisted of identical computational cells, plus special control cells in a column on its left edge.

PROBLEMS

15.1 Decimal division Consider radix-10 division using the quotient digit set $[-6, 6]$.

a. Construct the upper right quadrant of the p-d plot and determine the number of decimal digits that need to be examined in p and d for selecting the quotient digit.

b. Can the quotient digit selection logic or ROM be simplified if we are not restricted to inspect whole decimal digits (e.g., we can, if necessary, inspect the most significant 2 bits in the binary encoding of a decimal digit)?

c. Present a hardware design for the decimal divider assuming that the multiples $2d$, $3d$, $4d$, $5d$, and $6d$ are precomputed through five additions and stored in registers.

15.2 Quotient digit selection logic Formulate a lower bound on the size of the lookup table for quotient digit selection as a function of Δd and Δp, introduced in Section 15.2. State all your assumptions. Does your lower bound apply to the number of product terms in a PLA implementation?

15.3 Radix-8 division

a. Draw the complete p-d plot (both quadrants) for radix-8 division, with quotient digits in $[-4, 4]$ and the divisor in the range $[1, 2)$, on graph paper.

b. Using Δd and Δp, as discussed in Section 15.2, determine lower bounds on the precisions required of d and p in order to correctly select the quotient digit.

c. Assuming that p is in stored-carry form, determine the needed precision for d and p to minimize the number of input bits to the quotient digit selection logic or table.

d. Can you reduce the precisions obtained in part c for common cases by allowing a few special cases with higher precision?

15.4 Theory of high-radix division Prove or disprove the following assertions.

a. Once lower bounds on the number of bits of precision in p and d have been obtained through the analysis presented in Section 15.2 (i.e., from Δd and Δp), the use of one extra bit of precision for each is always adequate.

b. It is always possible to trade off one extra bit of precision in d for one less bit of precision in p in quotient digit selection.

15.5 Bit-serial division Prove that bit-serial division is infeasible for standard binary numbers, regardless of whether the inputs are supplied LSB-first or MSB-first. We are, of course, excluding any scheme in which all input bits are shifted in serially before division begins.

15.6 High-radix division with over-redundant quotient Study the effect of changing the radix from r to $r/2$, while keeping the same digit set as in radix r, on the overlap regions in Fig. 15.4 and the precision required of p and d in selecting the quotient digit. Relate your discussion to radix-2 division with over-redundant quotient introduced in Problem 14.14.

15.7 Significand divider with no remainder In dividing the significands of two floating-point numbers, both the dividend and divisor are k bits wide and computing the remainder is not needed. Discuss if and how this can lead to simplified hardware for the significand divider. Note that the divider can have various designs (restoring or nonrestoring binary, high-radix, array, etc.).

15.8 One's-complement binary dividers

a. Draw the block diagram of a restoring signed divider for 1's-complement numbers. Discuss any complication due to the use of 1's-complement operands and differences with a 2's-complement divider.

b. Repeat part a for a nonrestoring 1's-complement binary divider.

15.9 RNS dividers Sketch the design of an RNS divider that uses approximate magnitude comparison between RNS partial remainder and divisor, as discussed in Section 4.4, to produce a BSD quotient. Include on-the-fly conversion hardware to generate an RNS quotient from the BSD quotient and an analysis of the precision required in the comparisons.

15.10 Division with prescaling Suppose that prescaling is used to limit the range of the divisor d to (0.9, 1.1).

a. Construct a p-d plot similar to that in Fig. 14.14 for radix-4 division with the digit set $[-2, 2]$.

b. Derive the required precision in p and d for quotient digit selection.

c. Compare the results of part b to those obtained from Fig. 14.14 and discuss.

15.11 Division with prescaling Discuss whether it is possible to apply prescaling to a divider that keeps its partial remainder in stored-carry form.

15.12 Restoring array divider For the restoring array divider of Fig. 15.7:

a. Explain the function of the OR gates at the left edge of the array.

b. Can the OR gates be replaced by controlled subtractor cells in the interest of uniformity? How or why not?

c. Verify that the array divider works correctly by tracing through the signal values for the division .011111/.110.

d. Explain how the array can be modified to perform signed division.

15.13 Nonrestoring array divider For the nonrestoring array divider of Fig. 15.8:

a. Explain the wraparound links for the four cells located at the right edge of the array.

b. Explain the dangling or unused outputs in three of the four cells located at the left edge of the array.

d. Verify that the array divider works correctly by tracing through the signal values for the division 0.011111/0.110.

e. Present modifications in the design such that partial remainders are passed downward in carry-save form and lookahead circuits are used between rows to derive the carry-out q_{-i}.

f. Estimate the improvement in speed as a result of the modifications presented in part e and discuss the cost-effectiveness of the new design.

g. Show how the array can be used for signed division. *Hint:* Modify the input at the upper left corner, which is now connected to the constant 1.

h. Test your proposed solution to part g by tracing the division 1.10001/0.110.

i. Show how the array can be modified to perform modular division, as discussed in Section 15.4.

15.14 BSD array divider We would like to construct an array divider for binary signed-digit (BSD) numbers using the digit set $[-1, 1]$, encoded as 10, 00, and 01, for -1, 0, and 1, respectively.

a. Present the design of a controlled subtractor cell for BSD numbers.

b. Show how the structure of a nonrestoring array divider must be modified to deal with BSD numbers.

c. Compare the resulting design with a nonrestoring array divider with respect to speed and cost.

15.15 Combined multiply/divide units

a. Draw a complete block diagram for a radix-4 multiply/divide unit, as discussed in Section 15.6.

b. Supply the detailed design of the array multiplier/divider shown in Fig. 15.10, assuming unsigned inputs.

c. Discuss modifications required to the design of part b for 2's-complement inputs.

15.16 Divider with a multiplicative input Consider the design of a unit to compute $y = az/d$, where y, a, z, and d are k-bit fractions. A radix-4 algorithm is to be used for computing $q = z/d$. As digits of $q = z/d$ in $[-2, 2]$ are obtained, they are multiplied by a and the product aq is accumulated using radix-4 multiplication with left shifts.

a. Present a block diagram for the design of this divider with multiplicative input.

b. Evaluate the speed advantage of the unit compared to cascaded multiply and divide units.

c. Evaluate the speed penalty of the unit when used to perform simple multiplication or division.

15.17 Division with quotient digit prediction In a divider, whether using a carry-propagate or a carry-save adder in each cycle, the quotient digit selection logic is on the critical path that determines the cycle time. Since the delay for quotient digit selection can be significant for higher radices, one idea is to select the following cycle's quotient digit q_{-j-1} as the current cycle's quotient digit q_{-j} is used to produce the new partial remainder $s^{(j)}$. The trick is to overcome the dependence of q_{-j-1} on $s^{(j)}$ by generating an approximation to $s^{(j)}$ that is then used to predict q_{-j-1} in time for the start of the next cycle. Discuss the issues involved in the design of dividers with quotient digit prediction. Include in your discussion the two cases of carry-propagate and carry-save division cycles [Erce94].

REFERENCES

[Agra79] Agrawal, D. P., "High-Speed Arithmetic Arrays," *IEEE Trans. Computers*, Vol. 28, No. 3, pp. 215–224, 1979.

[Atki68] Atkins, D. E., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Trans. Computers*, Vol. 17, No. 10, pp. 925–934, 1968.

[Capp73] Cappa, M., and V. C. Hamacher, "An Augmented Iterative Array for High-Speed Binary Division," *IEEE Trans. Computers*, Vol. 22, pp. 172–175, February 1973.

[Erce94] Ercegovac, M. D., and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.

[Kama74] Kamal, A. K., et al., "A Generalized Pipeline Array," *IEEE Trans. Computers*, Vol. 23, No. 5, pp. 533–536, 1974.

[Lo86] Lo, H.-Y., "An Improvement of Nonrestoring Array Divider with Carry-Save and Carry-Lookahead Techniques," in *VLSI '85*, E. Horbst, (ed.), Elsevier, 1986, pp. 249–257.

[Ober97] Oberman, S. F., and M. J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Computers*, Vol. 46, No. 8, pp. 833–854, 1997.

[Robe58] Robertson, J. E., "A New Class of Digital Division Methods," *IRE Trans. Electronic Computers*, Vol. 7, pp. 218–222, September 1958.

[Schw93] Schwarz, E. M., and M. J. Flynn, "Parallel High-Radix Nonrestoring Division," *IEEE Trans. Computers*, Vol. 42, No. 10, pp. 1234–1246, 1993.

[Stef72] Stefanelli, R., "A Suggestion for a High-Speed Parallel Divider," *IEEE Trans. Computers*, Vol. 21, No. 1, pp. 42–55, 1972.

[Tayl85] Taylor, G. S., "Radix-16 SRT Dividers with Overlapped Quotient Selection Stages," *Proc. 7th Symp. Computer Arithmetic*, pp. 64–71, 1985.

[Zura87] Zurawski, J. H. P., and J. B. Gosling, "Design of a High-Speed Square Root, Multiply, and Divide Unit," *IEEE Trans. Computers*, Vol. 36, No. 1, pp. 13–23, 1987.

Chapter 16 DIVISION BY CONVERGENCE

Digit-recurrence division schemes discussed in Chapters 13-15 can be viewed as manipulation of s (initially z) and q (initially 0) in k cycles such that s tends to 0 as q converges to the quotient. One digit of convergence is obtained per cycle. In this chapter, we will see that through the use of multiplication as the basic step, instead of addition, convergence of q to its final value can occur in O(log k) rather than O(k) cycles, albeit with each cycle being more complex than in digit-recurrence division.

16.1 GENERAL CONVERGENCE METHODS

Convergence computation methods are characterized by two or three recurrence equations that are used to iteratively adjust/update the values of the variables u and v (and w). The two- and three-variable versions of such convergence methods are written as follows:

$$u^{(i+1)} = f(u^{(i)}, v^{(i)}) \qquad u^{(i+1)} = f(u^{(i)}, v^{(i)}, w^{(i)})$$
$$v^{(i+1)} = g(u^{(i)}, v^{(i)}) \qquad v^{(i+1)} = g(u^{(i)}, v^{(i)}, w^{(i)})$$
$$w^{(i+1)} = h(u^{(i)}, v^{(i)}, w^{(i)})$$

The functions f and g (and h) specify the computations to be performed in each updating cycle. Beginning with the initial values $u^{(0)}$ and $v^{(0)}$ (and $w^{(0)}$), we go through a number of iterations, each time computing $u^{(i+1)}$ and $v^{(i+1)}$ (and $w^{(i+1)}$) based on $u^{(i)}$ and $v^{(i)}$ (and $w^{(i)}$). We direct the iterations such that one value, say u, converges to some constant. The value of v (and/or w) then converges to the desired function(s).

The complexity of this method obviously depends on two factors:

ease of evaluating f and g (and h)

rate of convergence (or number of iterations needed)

Many specific instances of the preceding general method are available and can be used to compute a variety of useful functions. A number of examples are discussed in this chapter and in Chapters 21–23.

Digit-recurrence division methods, discussed in Chapters 13–15, can in fact be formulated as convergence computations. Given the fractional dividend z and divisor d, the quotient q and remainder s can be computed by a recurrence scheme of the general form

$$\begin{aligned} s^{(j)} &= s^{(j-1)} - \gamma^{(j)} d \qquad \text{Set } s^{(0)} = z\text{; make } s \text{ converge to 0} \\ q^{(j)} &= q^{(j-1)} + \gamma^{(j)} \qquad \text{Set } q^{(0)} = 0\text{; obtain } q \approx q^{(k)} \end{aligned}$$

where the $\gamma^{(j)}$ can be any sequence of values that make the residual (partial remainder) s converge to 0. The invariant of the iterative computation above is

$$s^{(j)} + q^{(j)} d = z$$

which leads to $q^{(k)} \approx z/d$ when $s^{(k)} \approx 0$.

In digit-recurrence division with fractional operands, $\gamma^{(j)}$ is taken to be $q_{-j} r^{-j}$ (i.e., the contribution of the jth digit of the quotient q to its value). We can rewrite the preceding recurrences by dealing with $r^j s^{(j)}$ and $r^j q^{(j)}$ as the scaled residual and quotient, respectively:

$$\begin{aligned} s^{(j)} &= r s^{(j-1)} - q_{-j} d \qquad \text{Set } s^{(0)} = z\text{; keep } s \text{ bounded} \\ q^{(j)} &= r q^{(j-1)} + q_{-j} \qquad \text{Set } q^{(0)} = 0\text{; obtain } q \approx q^{(k)} r^{-k} \end{aligned}$$

The original residual s can be made to converge to 0 by keeping the magnitude of the scaled residual in check. For example, if the scaled residual $s^{(j)}$ is in $[-d, d)$, the unscaled residual would be in $[-d2^{-j}, d2^{-j})$; thus convergence of s to 0 is readily accomplished.

The many digit-recurrence division schemes considered in Chapters 13–15 simply correspond to variations in the radix r, the scaled residual bound, and quotient digit selection rule. The functions f and g of digit-recurrence division are quite simple. The function f, for updating the scaled residual, is computed by shifting and (multioperand) addition. The function g, for updating the scaled quotient, corresponds to the insertion of the next quotient digit into a register via a one-digit left shift.

Even though high-radix schemes can reduce the number of iterations in digit-recurrence division, we still need $O(k)$ iterations with any small fixed radix $r = 2^b$. The rest of this chapter deals with division by other convergence methods that require far fewer [i.e. $O(\log k)$] iterations. Note that as we go to digit-recurrence division schemes entailing very high radices, quotient digit selection and the computation of the subtractive term $q_{-j} d$ become more difficult. Computation of $q_{-j} d$ involves a multiplication in which one of the operands is much narrower than the other one. So, in a sense, high-radix digit-recurrence division also involves multiplication.

16.2 DIVISION BY REPEATED MULTIPLICATIONS

To compute the ratio $q = z/d$, one can repeatedly multiply z and d by a sequence of m multipliers $x^{(0)}, x^{(1)}, \cdots, x^{(m-1)}$:

$$q = \frac{z}{d} = \frac{zx^{(0)}x^{(1)} \cdots x^{(m-1)}}{dx^{(0)}x^{(1)} \cdots x^{(m-1)}}$$

If this is done in such a way that the denominator $dx^{(0)}x^{(1)} \cdots x^{(m-1)}$ converges to 1, the numerator $zx^{(0)}x^{(1)} \cdots x^{(m-1)}$ will converge to q. This process does not yield a remainder, but the remainder s (if needed) can be computed, via an additional multiplication and a subtraction, using $s = z - qd$.

To perform division based on the preceding idea, we face three questions:

1. How should we select the multipliers $x^{(i)}$ such that the denominator does in fact converge to 1?
2. Given a selection rule for the multipliers $x^{(i)}$ how many iterations (pairs of multiplications) are needed?
3. How are the required computation steps implemented in hardware?

In what follows, we will answer these three questions in turn. But first, let us formulate this process as a convergence computation.

Assume a bit-normalized fractional divisor d in $[1/2, 1)$. If this condition is not satisfied initially, it can be made to hold by appropriately shifting both z and d. The corresponding convergence computation is formulated as follows:

$$\begin{aligned} d^{(i+1)} &= d^{(i)}x^{(i)} && \text{Set } d^{(0)} = d\text{; make } d^{(m)} \text{ converge to 1} \\ z^{(i+1)} &= z^{(i)}x^{(i)} && \text{Set } z^{(0)} = z\text{; obtain } z/d = q \approx z^{(m)} \end{aligned}$$

We now answer the first question posed above by selecting:

$$x^{(i)} = 2 - d^{(i)}$$

This choice transforms the recurrence equations into:

$$\begin{aligned} d^{(i+1)} &= d^{(i)}(2 - d^{(i)}) && \text{Set } d^{(0)} = d\text{; iterate until } d^{(m)} \approx 1 \\ z^{(i+1)} &= z^{(i)}(2 - d^{(i)}) && \text{Set } z^{(0)} = z\text{; obtain } z/d = q \approx z^{(m)} \end{aligned}$$

Thus, computing the functions f and g consists of determining the 2's-complement of $d^{(i)}$ and two multiplications by the result $2 - d^{(i)}$.

Now on to the second question: How quickly does $d^{(i)}$ converge to 1? In other words, how many multiplications are required to perform division? Noting that

$$d^{(i+1)} = d^{(i)}(2 - d^{(i)}) = 1 - (1 - d^{(i)})^2$$

we conclude that:

$$1 - d^{(i+1)} = (1 - d^{(i)})^2$$

Thus, if $d^{(i)}$ is already close to 1 (i.e., $1 - d^{(i)} \leq \varepsilon$), $d^{(i+1)}$ will be even closer to 1 (i.e., $1 - d^{(i+1)} \leq \varepsilon^2$). This property is known as *quadratic convergence* and leads to a logarithmic number m of iterations to complete the process. To see why, note that because d is in [1/2, 1), we begin with $1 - d^{(0)} \leq 2^{-1}$. Then, in successive iterations, we have $1 - d^{(1)} \leq 2^{-2}$, $1 - d^{(2)} \leq 2^{-4}, \cdots, 1 - d^{(m)} \leq 2^{-2^m}$. If the machine word is k bits wide, we can get no closer to 1 than $1 - 2^{-k}$. Thus, the iterations can stop when 2^m equals or exceeds k. This gives us the required number of iterations:

$$m = \lceil \log_2 k \rceil$$

Table 16.1 shows the progress of computation, and the pattern of convergence, in the four cycles required with 16-bit operands. For a 16-by-16 division, the preceding convergence method requires 7 multiplications (two per cycle, except in the last cycle, where only $z^{(4)}$ is computed); with 64-bit operands, we need 11 multiplications and 6 complementation steps. In general, for k-bit operands, we need

$$2m - 1 \text{ multiplications} \qquad \text{and} \qquad m \text{ 2's-complementations}$$

where $m = \lceil \log_2 k \rceil$.

Figure 16.1 shows a graphical representation of the convergence process in division by repeated multiplications. Clearly, convergence of $d^{(i)}$ to 1 and $z^{(i)}$ to q occurs from below; that is, in all intermediate steps, $d^{(i)} < 1$ and $z^{(i)} < q$. After the required number m of iterations, $d^{(m)}$ equals $1 - ulp$, which is the closest it can get to 1. At this point, $z^{(m)}$ is the required quotient q.

Answering the third, and final, question regarding hardware implementation is postponed until after the discussion of a related algorithm in Section 16.3.

Let us now say a few words about computation errors. Note that even if machine arithmetic is completely error-free, $z(m)$ can be off from q by up to ulp (when $z = d$, both $d^{(i)}$ and $z^{(i)}$ converge to $1 - ulp$). The maximum error in this case can be reduced to $ulp/2$ by simply adding ulp to any quotient with $q_{-1} = 1$.

The following approximate analysis captures the effect of errors in machine arithmetic. We present a more detailed discussion of computation errors in Chapter 19 in connection with real-number arithmetic.

TABLE 16.1
Quadratic convergence in computing z/d by repeated multiplications, where $1/2 \leq d = 1 - y < 1$

i	$d^{(i)} = d^{(i-1)}\, x^{(i-1)}$, with $d^{(0)} = d$	$x^{(i)} = 2 - d^{(i)}$
0	$1 - y = (.1\text{xxx xxxx xxxx xxxx})_{\text{two}} \geq 1/2$	$1 + y$
1	$1 - y^2 = (.11\text{xx xxxx xxxx xxxx})_{\text{two}} \geq 3/4$	$1 + y^2$
2	$1 - y^4 = (.1111\ \text{xxxx xxxx xxxx})_{\text{two}} \geq 15/16$	$1 + y^4$
3	$1 - y^8 = (.1111\ 1111\ \text{xxxx xxxx})_{\text{two}} \geq 255/256$	$1 + y^8$
4	$1 - y^{16} = (.1111\ 1111\ 1111\ 1111)_{\text{two}} = 1 - ulp$	

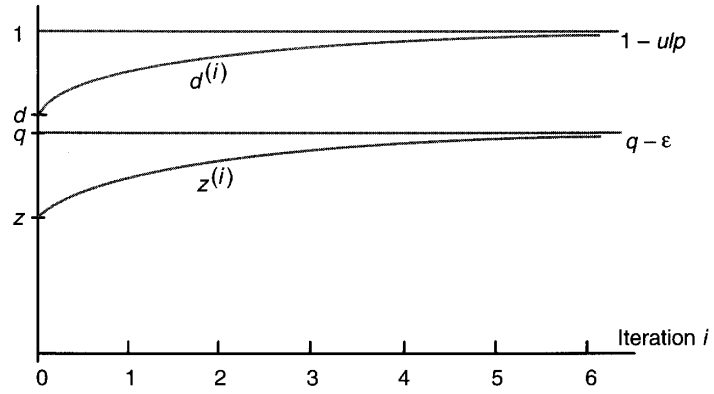


Fig. 16.1 Graphical representation of convergence in division by repeated multiplications.

Suppose that $k \times k$ multiplication is performed by simply truncating the exact $2k$-bit product to k bits, thus introducing a negative error that is upper-bounded by *ulp*. Note that computing $2 - d^{(i)}$ can be error-free, provided we can represent, and compute with, numbers that are in [0, 2), or else we scale down such numbers by shifting them to the right and keeping an extra bit or two of precision beyond position $-k$. We can also ignore any error in computing $d^{(i+1)}$, since such errors affect both recurrence equations and thus do not change the ratio z/d.

The worst-case error of *ulp*, introduced by the multiplication used to compute z in each iteration, leads to an accumulated error that is bounded by m *ulp* after m iterations. If we want to keep this error bound below 2^{-k}, we must perform all intermediate computations with at least $\log_2 m$ extra bits of precision. Since in practice m is quite small (say, $m \leq 5$), this requirement can be easily satisfied.

16.3 DIVISION BY RECIPROCATION

Another way to compute $q = z/d$ is to first find $1/d$ and then multiply the result by z. If several divisions by the same divisor d need to be performed, this method is particularly efficient, since once $1/d$ is found for the first division, each subsequent division involves just one additional multiplication.

The method we use for computing $1/d$ is based on Newton–Raphson iteration to determine a root of $f(x) = 0$. We start with some initial estimate $x^{(0)}$ for the root and then iteratively refine the estimate using the recurrence

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

where $f'(x)$ is the derivative of $f(x)$. Figure 16.2 provides a graphical representation of the refinement process. Let $\tan \alpha^{(i)}$ be the slope of the tangent to $f(x)$ at $x = x^{(i)}$. Then, referring to Fig. 16.2, the preceding iterative process is easily justified by noting that:

$$\tan \alpha^{(i)} = f'(x^{(i)}) = \frac{f(x^{(i)})}{x^{(i)} - x^{(i+1)}}$$

To apply the Newton–Raphson method to reciprocation, we use $f(x) = 1/x - d$ which has a root at $x = 1/d$. Then, $f'(x) = -1/x^2$, leading to the recurrence:

$$x^{(i+1)} = x^{(i)}(2 - x^{(i)}d) \quad \text{See below for the initial value } x^{(0)}$$

Computationally, two multiplications and a 2's-complementation step are required per iteration.

Let $\delta^{(i)} = 1/d - x^{(i)}$ be the error at the ith iteration. Then:

$$\begin{aligned} \delta^{(i+1)} &= 1/d - x^{(i+1)} = 1/d - x^{(i)}(2 - x^{(i)}d) \\ &= d(1/d - x^{(i)})^2 = d(\delta^{(i)})^2 \end{aligned}$$

Since $d < 1$, we have $\delta^{(i+1)} < (\delta^{(i)})^2$, proving quadratic convergence. If the initial value $x^{(0)}$ is chosen such that $0 < x^{(0)} < 2/d$, leading to $|\delta^{(0)}| < 1/d$, convergence is guaranteed.

At this point, we are interested only in simple schemes for selecting $x^{(0)}$, with more elaborate, and correspondingly more accurate, methods to be discussed later. For d in [1/2, 1), picking

$$x^{(0)} = 1.5$$

is quite simple and adequate, since it limits $|\delta^{(0)}|$ to the maximum of 0.5. A better approximation, with a maximum error of about 0.1, is

$$x^{(0)} = 4(\sqrt{3} - 1) - 2d = 2.9282 - 2d$$

which can be obtained easily and quickly from d by shifting and adding.

The effect of inexact multiplications on the final error $\delta^{(m)} = 1/d - x^{(m)}$ can be determined by an analysis similar to that offered at the end of Section 16.2. Here, each iteration involves two back-to-back multiplications, thus leading to the bound $2m$ *ulp* for the accumulated error and the requirement for an additional bit of precision in the intermediate computations.

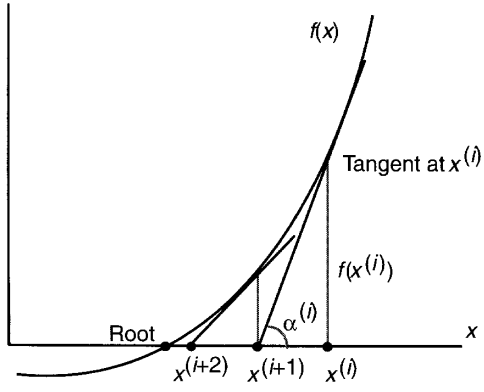


Fig. 16.2 Convergence to a root of $f(x) = 0$ in the Newton–Raphson method.

16.4 SPEEDUP OF CONVERGENCE DIVISION

Thus far, we have shown that division can be performed via $2\lceil \log_2 k \rceil - 1$ multiplications. This is not yet very impressive, since with 64-bit numbers and a 5-ns multiplier, division would need at least 55 ns. Three types of speedup are possible in division by repeated multiplications or by reciprocation:

reducing the number of multiplications
using narrower multiplications
performing the multiplications faster

Note that convergence is slow in the beginning. For example, in division by repeated multiplications, it takes six multiplications to get 8 bits of convergence and another five to go from 8 bits to 64 bits. The role of the first four multiplications is to provide a number $x^{(2)} = 2 - dx^{(0)}x^{(1)}$ such that when $x^{(2)}$ is multiplied by $z^{(2)}$ and $d^{(2)} = dx^{(0)}x^{(1)}$, we have 8 bits of convergence in the latter.

$$\begin{aligned} d &= (0.1\text{xxx xxxx} \cdots)_{\text{two}} \\ dx^{(0)} &= (0.11\text{xx xxxx} \cdots)_{\text{two}} \\ dx^{(0)}x^{(1)} &= (0.1111\ \text{xxxx} \cdots)_{\text{two}} \\ dx^{(0)}x^{(1)}x^{(2)} &= (0.1111\ 1111 \cdots)_{\text{two}} \end{aligned}$$

Since $x^{(0)}x^{(1)}x^{(2)}$ is essentially an approximation to $1/d$, these four initial multiplications can be replaced by a table-lookup step that directly supplies $x^{(0+)}$, an approximation to $x^{(0)}x^{(1)}x^{(2)}$ obtained based on a few high-order bits of d, provided the same convergence is achieved. Similarly, in division by reciprocation, a better starting approximation can be obtained via table lookup.

The remaining question is: How many bits of d must be inspected to achieve w bits of convergence after the first iteration? This is important because it dictates the size of the lookup table. In fact, we will see that $x^{(0+)}$ need not be a full-width number. If $x^{(0+)}$ is 8 bits rather than 64 bits wide, say, the lookup table will be eight times smaller and the first iteration can become much faster, since it involves multiplying an 8-bit multiplier by two 64-bit multiplicands.

We will prove, in Section 16.6, that a $2^w \times w$ lookup table is necessary and sufficient for achieving w bits of convergence after the first pair of multiplications. Here, we make a useful observation. For division by repeated multiplications, we saw that convergence to 1 and q occurred from below (Fig. 16.1). This does not have to be the case. If at some point in our iterations, $d^{(i)}$ overshoots 1 (e.g., becomes $1+\varepsilon$), the next multiplicative factor $2 - d^{(i)} = 1 - \varepsilon$ will lead to a value smaller than 1, but still closer to 1, for $d^{(i+1)}$ (Fig. 16.3).

So, in fact, what is important is that $|d^{(i)} - 1|$ decrease quadratically. It does not matter if $x^{(0+)}$ obtained from the table causes $dx^{(0+)}$ to become greater than 1; all we need to guarantee that $1 - 2^{-16} \leq dx^{(0+)}x^{(3)} < 1$ is to have $1 - 2^{-8} \leq dx^{(0+)} \leq 1 + 2^{-8}$. This added flexibility helps us in reducing the table size (both the number of words and the width).

We noted earlier that the first pair of multiplications following the table-lookup involve a narrow multiplier and may thus be faster than a full-width multiplication. The same applies to subsequent multiplications if the multiplier is suitably truncated. The result is that convergence occurs from above or below (Fig. 16.4).

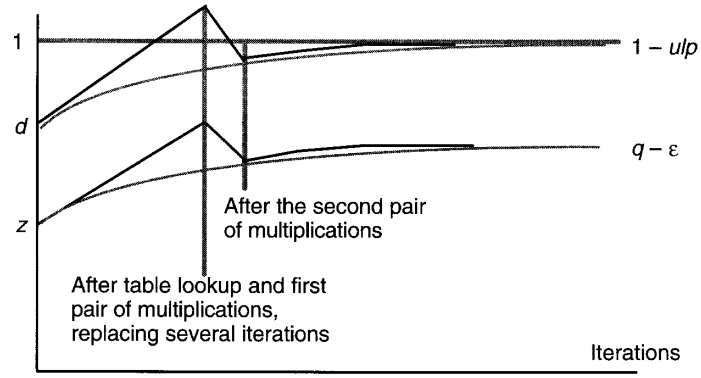


Fig. 16.3 Convergence in division by repeated multiplications with initial table lookup.

Here is an analysis for the effect of truncating the multiplicative factors to speed up the multiplications. We begin by noting that:

$$dx^{(0)}x^{(1)}\cdots x^{(i)} = 1 - y^{(i)}$$
$$x^{(i+1)} = 2 - (1 - y^{(i)}) = 1 + y^{(i)}$$

Assume that we truncate $1 - y^{(i)}$ to an a-bit fraction, thus obtaining $(1 - y^{(i)})_{\mathrm{T}}$ with an error of $\alpha < 2^{-a}$. With this truncated multiplicative factor, we get

$$(x^{(i+1)})_{\mathrm{T}} = 2 - (1 - y^{(i)})_{\mathrm{T}} \quad \text{where} \quad 0 \leq (x^{(i+1)})_{\mathrm{T}} - x^{(i+1)} < 2^{-a}$$

Thus:

$$\begin{aligned} dx^{(0)}x^{(1)}\cdots x^{(i)}(x^{(i+1)})_{\mathrm{T}} &= (1 - y^{(i)})(1 + y^{(i)} + \alpha) = 1 - (y^{(i)})^2 + \alpha(1 - y^{(i)}) \\ &= dx^{(0)}x^{(1)}\cdots x^{(i)}x^{(i+1)} + \alpha(1 - y^{(i)}) \end{aligned}$$

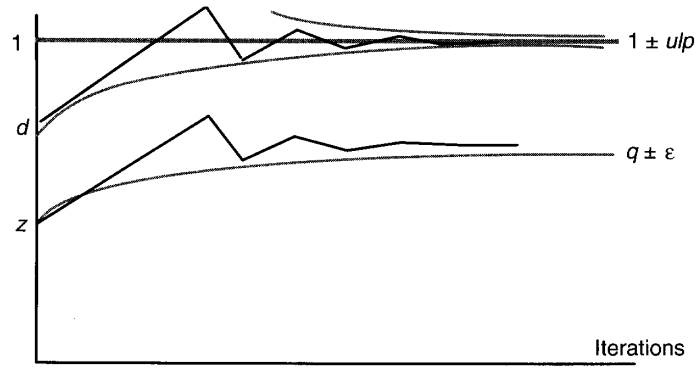


Fig. 16.4 Convergence in division by repeated multiplications with initial table lookup and the use of truncated multiplicative factors.

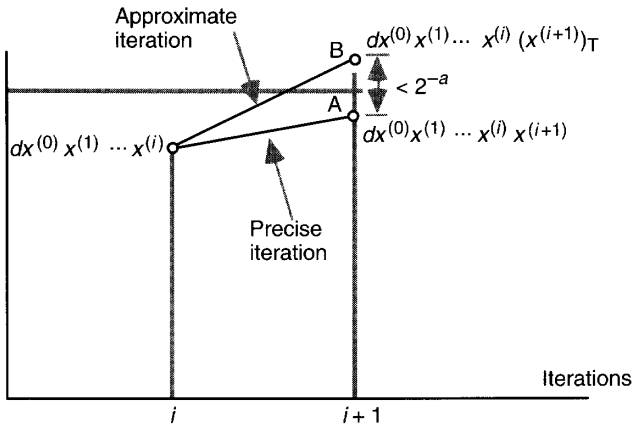


Fig. 16.5 One step in convergence division with truncated multiplicative factors.

Since $(1 - y^{(i)})$ is less than 1, the last term above is less than α and we have:

$$0 \leq \alpha(1 - y^{(i)}) < 2^{-a}$$

Hence, if we are aiming to go from l bits to $2l$ bits of convergence, we can truncate the next multiplicative factor to $2l$ bits. To justify this claim, consider Fig. 16.5. Point A, which is the result of precise iteration, is no more than 2^{-2l} below 1. Thus, with $a = 2l$, point B, arrived at by the approximate iteration, will be no more than 2^{-2l} above 1.

Now, putting things together for an example 64-bit multiplication, we need a table of size $256 \times 8 = 2$K bits for the lookup step. Then we need pairs of multiplications, with the multiplier being 9 bits, 17 bits, and 33 bits wide. The final step involves a single 64×64 multiplication.

16.5 HARDWARE IMPLEMENTATION

The hardware implementation of basic schemes for division by repeated multiplications or by reciprocation is straightforward. Both methods need two multiplications per iteration and both can use an initial table lookup step and truncation of the intermediate results to reduce the number of iterations and to speed up the multiplications.

If the hardware multiplier used is based on a digit-recurrence (binary or high-radix) algorithm, then narrower operands translate directly into fewer steps and correspondingly higher speed. For the 64-bit example at the end of Section 16.4, the total number of bit-level iterations to perform the seven multiplications required would be $2(9 + 17 + 33) + 64 = 182$. This is roughly equivalent to the number of bit-level iterations in three full 64×64 multiplications.

Convergence division methods are more likely to be implemented when a fast parallel (tree) multiplier is available. In the case of a full-tree multiplier, the narrower multiplicative factors may not offer any speed advantage. However, if a partial CSA tree, of the type depicted in Fig. 11.9 is used, a narrower multiplier leads to higher speed. For example, if the tree can handle $h = 9$ new inputs at once, the first pair of multiplications in our 64-bit example would require just one pass through the tree, the second pair would need two passes each (one pass if Booth's recoding is applied), and so on.

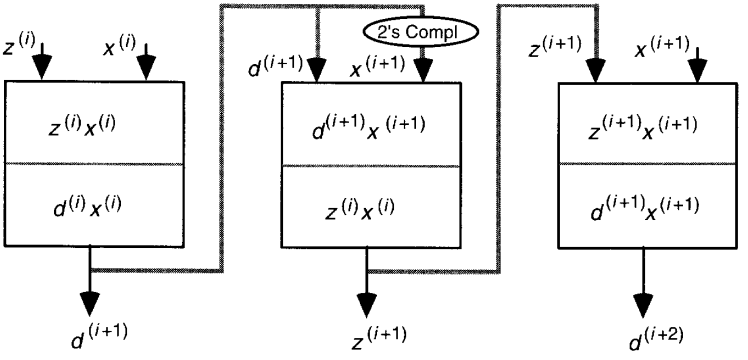


Fig. 16.6 Two multiplications fully overlapped in a two-stage pipelined multiplier.

Finally, since two independent multiplications by the same multiplier are performed in each step of division by repeated multiplications, the two can be pipelined (in both the full-tree and partial-tree implementations), thus requiring less time than two back-to-back multiplications. In such a case, the multiplication for $d^{(i)}$ is scheduled first, to get the result needed for the next iteration quickly and to keep the pipeline as full as possible. This is best understood for a multiplier that is implemented as a two-stage pipeline (Fig. 16.6). As the computation of $z^{(i)}x^{(i)}$ moves from the top to the bottom pipeline stage, the next iteration begins by computing the top stage of $d^{(i+1)}x^{(i+1)}$. We thus see that with a pipelined multiplier, the two multiplications needed in each iterations can be fully overlapped.

The pipelining scheme shown in Fig. 16.6 is not applicable to convergence division through divisor reciprocation, since in the recurrence $x^{(i+1)} = x^{(i)}(2 - x^{(i)}d)$, the second multiplication by $x^{(i)}$ needs the result of the first one. The most promising speedup method in this case relies on deriving a better starting approximation to $1/d$. For example, if the starting approximation is obtained with an error bound of 2^{-16}, then only three multiplications would be needed for a 32-bit quotient and five for a 64-bit result. But 16 bits of precision in the starting approximation would imply a large lookup table. The required lookup table can be made smaller, or totally eliminated, by a variety of methods:

1. Store the reciprocal values for fewer points and use linear (one multiply-add operation) or higher-order interpolation to compute the starting approximation (see Section 24.4).
2. Formulate the starting approximation as a multioperand addition problem and use one pass through the multiplier's CSA tree, suitably augmented, to compute it [Schw96].

With all the speedup methods discussed so far, the total division time can often be reduced to that of two to five ordinary multiplications. This has made convergence division a common choice for high-performance CPUs.

16.6 ANALYSIS OF LOOKUP TABLE SIZE

The required table size, for radix 2 with the goal of w bits of convergence after the first iteration (i.e., $1 - 2^{-w} \leq dx^{(0+)} \leq 1 + 2^{-w}$), is given in the following theorem.

THEOREM 16.1 To get $w \geq 5$ bits of convergence in the first iteration of division by repeated multiplications, w bits of d (beyond the mandatory 1) must be inspected. The factor $x^{(0+)}$ read out from the table is of the form $(1.\ \text{xxxx} \cdots \text{xxxx})_{\text{two}}$, with w bits after the radix point [Parh87].

Based on the Theorem 16.1, the required table size is $2^w \times w$ and the first pair of multiplications involve a $(w+1)$-bit multiplier $x^{(0+)}$.

A proof sketch for Theorem 16.1 begins as follows. A general analysis for an arbitrary radix r as well as a complete derivation of special cases that allow smaller tables (in number of words and/or width) can be found elsewhere [Parh87]. These special cases ($r = 3$ and $w = 1$, or $r = 2$ with $w \leq 4$) almost never arise in practice, and we can safely ignore them here.

Recall that our objective is to have $1 - 2^{-w} \leq dx^{(0+)} \leq 1 + 2^{-w}$. Let

$$d = (0.\underbrace{1d_{-2}d_{-3}\cdots d_{-(w+1)}}_{w \text{ bits to be inspected}}d_{-(w+2)}\cdots d_{-l})_{\text{two}}$$

Theorem 16.1 postulates the existence of $x^{(0+)} = (1.x^+_{-1}x^+_{-2}\cdots x^+_{-w})_{\text{two}}$ satisfying the objective inequality. Let $u = (1d_{-2}d_{-3}\cdots d_{-(w+1)})_{\text{two}}$, satisfying $2^w \leq u < 2^{w+1}$, be the integer composed of the first $w+1$ bits of d. We have:

$$2^{-(w+1)}u \leq d < 2^{-(w+1)}(u+1)$$

Similarly, let $v = (1x^+_{-1}x^+_2\cdots x^+_{-w})_{\text{two}}$ be obtained from $x^{(0+)}$ by removing its radix point (multiplying it by 2^w). From the preceding inequalities for d and because the objective inequality can be rewritten as $2^w - 1 \leq dv \leq 2^w + 1$, we derive the following sufficient conditions:

$$2^w - 1 \leq 2^{-(w+1)}uv \qquad \text{and} \qquad 2^{-(w+1)}(u+1)v \leq 2^w + 1$$

These conditions lead to the following restrictions on v:

$$\frac{2^{w+1}(2^w-1)}{u} \leq v \leq \frac{2^{w+1}(2^w+1)}{u+1}$$

The existence of $x^{(0+)}$, as postulated, is thus contingent upon the preceding inequalities yielding an integer solution for v. This latter condition is equivalent to:

$$\left\lceil \frac{2^{w+1}(2^w-1)}{u} \right\rceil \leq \left\lfloor \frac{2^{w+1}(2^w+1)}{u+1} \right\rfloor$$

Showing that this last inequality always holds is left as an exercise and completes the "sufficiency" part of the proof. The "necessity" part—namely, that at least w bits of d must be inspected and that $x^{(0+)}$ must have at least w bits after the radix point—is also left as an exercise.

Thus, to achieve 8 bits of convergence after the initial pair of multiplications, we need to look at 8 bits of d (beyond the mandatory 1) and read out an 8-bit fractional part f for $x^{(0+)} = 1 + .f$. Table 16.2 shows two sample entries in the required lookup table. The first entry in this table

TABLE 16.2
Sample entries in the lookup table replacing the first four multiplications in division by repeated multiplications

Address	$d = 0.1$ xxxx xxxx	$x^{(0+)} = 1.$xxxx xxxx
55	0011 0111	1010 0101
64	0100 0000	1001 1001

has been determined as follows. Since d begins with the bit pattern 0.1001 1011 1, its value is in the range

$$311/512 \leq d < 312/512$$

Given the requirement for 8 bits of convergence after the first pair of multiplications, the table entry f must be chosen such that

$$311/512(1 + .f) \geq 1 - 2^{-8}$$
$$312/512(1 + .f) \leq 1 + 2^{-8}$$

From the preceding restrictions, we conclude that $199/311 \leq .f \leq 101/156$, or for the integer $f = 256 \times .f$, $163.81 \leq f \leq 165.74$. Hence, the table entry f can be either of the integers $164 = (1010\ 0100)_{\text{two}}$ or $165 = (1010\ 0101)_{\text{two}}$.

PROBLEMS

16.1 Division by repeated multiplications

a. Perform the division z/d, with unsigned fractional dividend $z = (.0101\ 0110)_{\text{two}}$ and divisor $d = (.1011\ 1001)_{\text{two}}$, through repeated multiplications.

b. Construct a table that provides the initial factor leading to 4 bits of convergence after the first multiplication. Note that $w = 4$ is a special case that leads to a smaller table compared to the one suggested by Theorem 16.1.

c. Perform the division of part a using the table of part b at the outset.

16.2 Division by repeated multiplications

a. Perform the division z/d, with unsigned fractional dividend $z = (.4321)_{\text{ten}}$ and divisor $d = (.4456)_{\text{ten}}$, through repeated multiplications.

b. Suggest a simple final correction to improve the accuracy of the result in part a.

c. Construct a table that provides the initial multiplicative factor leading to 1 decimal digit of convergence after the first multiplication.

d. Perform the division of part a using the table of part b at the outset.

16.3 Iterative reciprocation Using Newton–Raphson iterations and decimal arithmetic with six digits of precision after the radix point throughout:

a. Compute the reciprocal of $d = (.823\ 456)_{\text{ten}}$.

b. Compute the reciprocal of $d = (.512\ 345)_{\text{ten}}$.

c. Construct a segment of the initial lookup table with 10 two-digit entries (corresponding to $d = .50, .51, \cdots, .59$, with an entry ij representing $1.ij$) to provide the best possible initial approximation to $1/d$.

d. Repeat part b, this time using the table of part c at the outset.

16.4 Iterative reciprocation

a. Compute the reciprocal of $d = (.318\ 310)_{\text{ten}} \approx 1/\pi$ using $x^{(i+1)} = x^{(i)}(2 - x^{(i)}d)$ and arithmetic with six digits after the decimal point throughout. Keep track of the difference between $x^{(i)}$ and π to determine the number of iterations needed.

b. Repeat part a, using the expansion $1/d = 1/(1-y) \approx (1+y)(1+y^2)(1+y^4)\cdots$, where $y = 1 - d$, instead of the Newton–Raphson iteration. Each term $1 + y^{2^{i+1}}$ is computed by squaring y^{2^i} and adding 1.

c. Compare the methods of parts a and b and discuss.

16.5 Division by reciprocation

a. Perform the division z/d, with unsigned fractional dividend $z = (.0101\ 0110)_{\text{two}}$ and divisor $d = (.1010\ 1100)_{\text{two}}$, through reciprocation.

b. Construct a table of approximate reciprocals providing 4 bits of convergence (i.e., the product of the approximate reciprocal and d should have four leading 0s or 1s).

c. Perform the division of part a using the table of part b at the outset.

d. Based on the example of part b, formulate and prove a theorem, similar to Theorem 16.1, for the initial reciprocal approximation.

16.6 Division by reciprocation

An alternative Newton–Raphson iterative method for computing the reciprocal of d uses $f(x) = (x - 1 + 1/d)/(x - 1)$, which has a root at the complement of $1/d$.

a. Find the alternative iteration formula.

b. Compute the error term and prove quadratic convergence.

c. Use this alternative method to compute the reciprocal of $d = (.823\ 456)_{\text{ten}}$.

d. Use this alternative method to compute the reciprocal of $d = (.512\ 345)_{\text{ten}}$.

e. Comment on this new algorithm compared to the original one.

16.7 Division by reciprocation

a. Derive the maximum error for the starting approximation $x^{(0)} = 4(\sqrt{3} - 1) - 2d$ in division by reciprocation.

b. Find the best linear approximation involving a multiply-add operation and compare its worst-case error to the error of part a.

16.8 Table lookup for convergence division

a. Complete the "sufficiency" proof of Theorem 16.1 by showing that the inequality $\lceil 2^{w+1}(2^w - 1)/u \rceil \leq \lfloor 2^{w+1}(2^w + 1)/(u + 1) \rfloor$ always holds. *Hint:* Let q and s

($s \leq u$) be the quotient and remainder of dividing $2^{w+1}(2^w + 1)$ by $u + 1$. The right-hand side of the inequality is thus q. Try simplifying the left-hand side.

b. Construct the "necessity" part of the proof of Theorem 16.1 by showing that $x^{(0+)}$ satisfying $1 - 2^{-w} \leq dx^{(0+)} \leq 1 + 2^{-w}$ cannot have fewer than w bits after the radix point and cannot be obtained by inspecting fewer than w bits of d.

16.9 Convergence division with truncated multipliers

a. Prove that in division through repeated multiplications, a truncated denominator $d^{(i)}$, with a identical leading bits and b extra bits ($b \leq a$), will lead to a new denominator $d^{(i+1)}$ with at least $a + b$ identical leading bits.

b. Briefly discuss the implications of the result of part a for an arithmetic unit that uses an initial table lookup to obtain 8 bits of convergence and can perform 18×64 multiplications about 2.5 times as fast as full 64×64 multiplications.

16.10 Cubic convergence method Consider the following iterative formula for finding an approximate root of a nonlinear function f: $x^{(i+1)} = x^{(i)} - f(x^{(i)}) - f(x^{(i)})f''(x^{(i)})/(2f'(x^{(i)}))$.

a. Show that this iterative scheme exhibits cubic convergence.

b. Discuss the practical use of this method for function evaluation.

16.11 Cubic convergence method Consider the following iterative formula for finding an approximate root of a nonlinear function f: $x^{(i+1)} = x^{(i)} - 2f(x^{(i)})f'(x^{(i)})/[2(f'(x^{(i)}))^2 - f(x^{(i)})f''(x^{(i)})]$.

a. Show that this iterative scheme exhibits cubic convergence.

b. Try out the iterative formula for a nonlinear function of your choosing.

c. Discuss the practicality of the formula for function evaluation in digital computers.

16.12 Table lookup for convergence division Justify the second entry in Table 16.2 in the same manner as was done for the first entry in Section 16.6. Then, supply the entries for the addresses 5, 158, and 236.

16.13 Mystery convergence method The following two iterative formulas are applied to a bit-normalized binary fraction z in [1/2, 1): $u^{(i+1)} = u^{(i)}(x^{(i)})^2$ with $u^{(0)} = z$ and $v^{(i+1)} = v^{(i)}x^{(i)}$ with $v^{(0)} = z$.

a. Determine the function $v = g(z)$ that is computed if $x^{(i)} = 1 + (1 - u^{(i)})/2$.

b. Discuss the number of iterations that are needed and the operations that are executed in each iteration.

c. Suggest how the multiplicative term $x^{(i)}$ might be calculated.

d. Estimate the error in the final result.

e. Suggest ways to speed up the calculation.

f. Calculate the 8-bit result $v = g(z)$ using the procedure above and compare it to the correct result, given $z = (.1110\ 0001)_{\text{two}}$.

16.14 Table lookup for reciprocal approximation Inspecting w bits of the divisor in the initial table lookup for division by reciprocation divides the divisor range into 2^w equal-width intervals $[a^{(i)}, b^{(i)})$.

a. Show that a table entry equal to the average of $1/a^{(i)}$ and $1/b^{(i)}$ minimizes the worst-case error.

b. Show that a table entry equal to $2/(a^{(i)} + b^{(i)})$, that is, the reciprocal of the midpoint of the interval, minimizes the average-case error, assuming uniform distribution of divisor values.

16.15 Table lookup for reciprocal approximation Inspecting w bits of the divisor in the initial table lookup for division by reciprocation divides the divisor range into 2^w equal-width intervals. Prove that rounding the reciprocals of the midpoints of these intervals provides minimal worst-case relative errors in a w-bits-in, $(w + b)$-bits-out table [DasS94].

16.16 Division by convergence Consider the recurrences $s^{(j)} = rs^{(j-1)} - q_{-j}d$ and $q^{(j)} = rq^{(j-1)} + q_{-j}$, discussed in Section 16.1. We can take a somewhat more general view of these recurrences by rewriting q_{-j} as γ_j, an estimate for the rest of the quotient rather than its next digit. The estimate is obtained by table lookup based on a few high-order bits in $rs^{(j-1)}$. With this more general view, the second recurrence must be evaluated through addition rather than by concatenation (shifting the next digit into a register). Evaluate the suitability of this method for division via repeated multiplications [Wong92].

REFERENCES

[Ande67] Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Research and Development*, Vol. 11, No. 1, pp. 34–53, 1967.

[DasS94] DasSarma, D., and D. W. Matula, "Measuring the Accuracy of ROM Reciprocal Tables," *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 932–940, 1994.

[Ferr67] Ferrari, D., "A Division Method Using a Parallel Multiplier," *IEEE Trans. Electronic Computers*, Vol. 16, pp. 224–226, April 1967

[Flyn70] Flynn, M. J., "On Division by Functional Iteration," *IEEE Trans. Computers*, Vol. 19, pp. 702–706, August 1970.

[Kris70] Krishnamurthy, E. V., "On Optimal Iterative Schemes for High Speed Division," *IEEE Trans. Computers*, Vol. 19, No. 3, pp. 227–231, 1970.

[Ober97] Oberman, S. F., and M. J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Computers*, Vol. 46, No. 8, pp. 833–854, 1997.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Parh87] Parhami, B., "On the Complexity of Table Look-Up for Iterative Division," *IEEE Trans. Computers*, Vol. 36, No. 10, pp. 1233–1236, 1987.

[Schw96] Schwarz, E. M., and M. J. Flynn, "Hardware Starting Approximation Method and Its Application to the Square Root Operation," *IEEE Trans. Computers*, Vol. 45, No. 12, pp. 1356–1369, 1996.

[Wong92] Wong, D., and M. Flynn, "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 981–995, 1992.

PART V REAL ARITHMETIC

In many scientific and engineering computations, numbers in a wide range, from very small to extremely large, are processed. Fixed-point number representations and arithmetic are ill-suited to such applications. For example, a fixed-point decimal number system capable of representing both 10^{-20} and 10^{20} would require at least 40 decimal digits and even then, would not offer much precision with numbers close to 10^{-20}. Thus, we need special number representations that possess both a wide range and acceptable precision. Floating-point numbers constitute the primary mode of real arithmetic in most digital systems. In this part, we discuss key topics in floating-point number representation, arithmetic, and computational errors. Additionally, we cover alternative representations, such as logarithmic and rational number systems, that can offer certain advantages in range and/or accuracy. This part is composed of the following four chapters:

Chapter 17 FLOATING-POINT REPRESENTATIONS

In Chapters 1–3, we dealt with various methods for representing fixed-point numbers. Such representations suffer from limited range and/or precision, in the sense that they can provide high precision only by sacrificing the dynamic range, and vice versa. By contrast, a floating-point number system offers both a wide dynamic range for accommodating extremely large numbers (e.g., astronomical distances) and high precision for very small numbers (e.g., atomic distances). Chapter topics include:

17.1 FLOATING-POINT NUMBERS

Clearly, no finite representation method is capable of representing all real numbers, even within a small range. Thus, most real values will have to be represented in an approximate manner. Various methods of representation can be used:

Fixed-point number systems: offer limited range and/or precision. Computations must be "scaled" to ensure that values remain representable and that they do not lose too much precision.

Rational number systems: approximate a real value by the ratio of two integers. Lead to difficult arithmetic operations (see Section 20.2).

Floating-point number systems: the most common approach; discussed in Chapters 17–20.

Logarithmic number systems: represent numbers by their signs and logarithms. Attractive for applications needing low precision and wide dynamic range. Can be viewed as a limiting special case of floating-point representation (see Section 17.6).

Fixed-point representation leads to equal spacing in the set of representable numbers. Thus the maximum absolute error is the same throughout (*ulp* with truncation and *ulp*/2 with rounding). The problem with fixed-point representation is illustrated by the following examples:

$$x = (0000\ 0000.\ 0000\ 1001)_{\text{two}} \quad \text{Small number}$$
$$y = (1001\ 0000.\ 0000\ 0000)_{\text{two}} \quad \text{Large number}$$

The relative representation error due to truncation or rounding is quite significant for x while it is much less severe for y. On the other hand, both x^2 and y^2 are unrepresentable, because their computations lead to underflow (number too small) and overflow (too large), respectively.

The other three representation methods listed above lead to denser codes for smaller values and sparser codes for larger values. However, the code assignment patterns are different, leading to different ranges and error characteristics. For the same range of representable values, these representations tend to be better than fixed-point systems in terms of average relative representation error, even though the absolute representation error increases as the values get larger.

The numbers x and y in the preceding examples can be represented as $(1.001)_{\text{two}} \times 2^{-5}$ and $(1.001)_{\text{two}} \times 2^{+7}$, respectively. The exponent -5 or $+7$ essentially indicates the direction and amount by which the radix-point must be moved to produce the corresponding fixed-point representation shown above. Hence the designation "floating-point numbers."

A floating-point number has four components: the sign, the significand s, the exponent base b, and the exponent e. The exponent base b is usually implied (not explicitly represented) and is usually a power of 2, except, of course, for decimal arithmetic, where it is 10. Together, these four components represent the number:

$$x = \pm s \times b^e \quad \text{or} \quad \pm \text{significand} \times \text{base}^{\text{exponent}}$$

A typical floating-point representation format is shown in Fig. 17.1. A key point to observe is that two signs are involved in a floating-point number.

1. The significand or number sign, which indicates a positive or negative floating-point number and is usually represented by a separate sign bit (signed-magnitude convention).
2. The exponent sign which, roughly speaking, indicates a large or small number and is usually embedded in the biased exponent (Section 2.2). When the bias is a power of 2 (e.g., 128 with an 8-bit exponent), the exponent sign is the complement of its most significant bit.

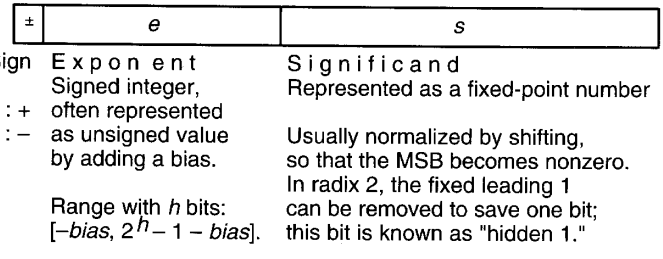


Fig. 17.1 Typical floating-point number format.

The use of a biased exponent format has virtually no effect on the speed or cost of exponent arithmetic (addition/subtraction), given the small number of bits involved. It does, however, facilitate zero detection (zero can be represented with the smallest biased exponent of 0 and an all-zero significand) and magnitude comparison (we can compare normalized floating-point numbers as if they were integers).

The range of values in a floating-point number representation format is composed of the intervals $[-max, -min]$ and $[min, max]$, where:

$$max = \text{largest significand} \times b^{\text{largest exponent}}$$
$$min = \text{smallest significand} \times b^{\text{smallest exponent}}$$

Figure 17.2 shows the number distribution pattern and the various subranges in floating-point representations. In particular, it includes the three special or singular values $-\infty$, 0, and $+\infty$ (0 is special because it cannot be represented with a normalized significand) and depicts the meanings of overflow and underflow. Overflow occurs when a result is less than $-max$ or greater than max. Underflow, on the other hand, occurs for results in the range $(-min, 0)$ or $(0, min)$.

Within the preceding framework, many alternative floating-point representation formats can be devised. In fact, before the IEEE standard format (see Section 17.2) was adopted, numerous competing, and incompatible, floating-point formats existed in digital computers. Even now that the IEEE standard format is dominant, on certain (rare) occasions in the design of special-purpose systems, the designer might choose a different format for performance or cost reasons.

The equation $x = \pm s \times b^e$ for the value of a floating-point number suggests that the range $[-max, max]$ increases if we choose a larger exponent base b. A larger b also simplifies arithmetic operations on the exponents, since for a given range, smaller exponents must be dealt with. However, if the significand is to be kept in normalized form, effective precision decreases for larger b. In the past, machines with $b = 2$, 8, 16, or 256 were built. But the modern trend is to use $b = 2$ to maximize the precision with normalized significands.

The exponent sign is almost always encoded in a biased format, for reasons given earlier in this section. As for the sign of a floating-point number, alternatives to the currently dominant signed-magnitude format include the use of 1's- or 2's-complement representation. Several variations have been tried in the past, including the complementation of the significand part only and the complementation of the entire number (including the exponent part) when the number to be represented is negative.

Once we have fixed b and assigned one bit to the number sign, the next question is the allocation of the remaining bits to the exponent and significand parts. Devoting more bits to the exponent part widens the number representation range but reduces the precision. So, the designer's choice is dictated by the range and precision requirements of the application(s) at hand.

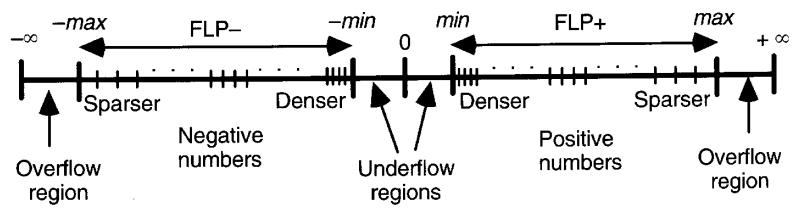


Fig. 17.2 Subranges and special values in floating-point number representations.

The final question, given the allocation of a total of m bits for the binary fixed-point significand s, is the choice of k, the number of whole bits to the left of the radix point in s. Again, many variations appeared in the past. The choice $k = 0$ leads to a fractional significand in the range [0, 1), sometimes referred to as the *mantissa*. At the other extreme, choosing $k = m$ leads to an integer significand that increases both *max* and *min* (see Fig. 17.2), thus narrowing the overflow region and widening the underflow region. The same effect can be achieved by choosing an off-center bias for the exponent.

The only other common choice for the number of whole bits in the significand of a floating-point number, and the one used in the IEEE standard, is $k = 1$, leading to significands in the range [1, 2). With normalized binary significands, this single whole bit, which is always 1, can be dropped and the significand represented by its fractional part alone.

Virtually all digital computers have separate formats for integers and floating-point numbers, even though, in principle, k-digit integers can be represented in a floating-point format that has a k-digit significand. One reason is that integer arithmetic is both simpler and faster; thus there is no point in subjecting integers to unnecessary complications. Another reason is that with a separate integer format, that has no exponent part, larger numbers can be represented exactly.

If one chooses to have a common format for integers and floating-point numbers, it is a good idea to include an "inexact flag" in the representation. For numbers that have exact representations in the floating-point format, the inexact flag may be set to 0. When the result of a computation with exact operands is too small or too large to be represented exactly, the inexact flag of the result can be set to 1. Note that dealing with this inexact flag is another source of complexity.

17.2 THE ANSI/IEEE FLOATING-POINT STANDARD

In the early days of digital computers, it was quite common for machines from various vendors to have different word widths and unique floating-point formats. Word widths were standardized at powers of 2 early on, with nonconforming word widths such as 24, 36, 48, and 60 bits all but disappearing. However, even after 32- and 64-bit words became the norm, different floating-point formats persisted. A main objective in developing a standard floating-point representation is to make numerical programs predictable and completely portable, in the sense of producing the same results when run on different machines.

The two representation formats in IEEE standard for binary floating-point numbers, formally known as "ANSI/IEEE Std 754-1985," are depicted in Fig. 17.3. The short, or single-precision, format is 32 bits wide, whereas the long, or double-precision, version requires 64 bits. The two formats have 8- and 11-bit exponent fields and use exponent biases of 127 and 1023, respectively. The significand is in the range [1, 2), with its single whole bit, which is always 1, removed and only the fractional part shown. The notation "23 + 1" or "52 + 1" for the width of the significand is meant to explicate the role of the hidden bit, which does contribute to the precision without taking up space.

Table 17.1 summarizes the most important features of the IEEE standard floating-point representation format.

Since 0 cannot be represented with a normalized significand, a special code must be assigned to it. In the IEEE standard format, zero has the all-0s representation, with positive or negative sign. Special codes are also needed for representing $\pm\infty$ and NaN (not-a-number). The NaN special value is useful for representing undefined results such as 0/0. When one of these special

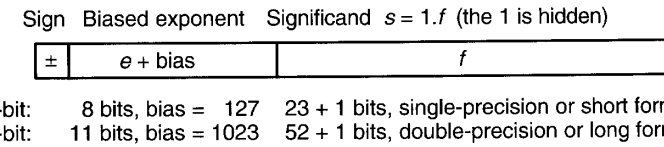


Fig. 17.3 The ANSI/IEEE standard floating-point number representation formats.

values appears as an operand in an arithmetic operation, the result of the operation is specified according to defined rules that are part of the standard. For example:

$$\text{Ordinary number} \div (+\infty) = \pm 0$$
$$(+\infty) \times \text{Ordinary number} = \pm\infty$$
$$\text{NaN} + \text{Ordinary number} = \text{NaN}$$

The special codes thus allow exceptions to be propagated to the end of a computation rather than bringing it to a halt. More on this later.

Denormals, or denormalized values, are defined as numbers without a hidden 1 and with the smallest possible exponent. They are provided to make the effect of underflow less abrupt. In other words, certain small values that are not representable as normalized numbers, hence must be rounded to 0 if encountered in the course of computations, can be represented more precisely as denormals. For example, $(0.0001)_{\text{two}} \times 2^{-126}$ is a denormal that does not have a normalized representation in the IEEE single/short format. This "graceful underflow" provision, which can lead to cost and speed overhead in hardware, is optional; implementations of the standard do not have to support denormals. Figure 17.4 shows the role of denormals in providing representation points in the otherwise empty interval $(0, min)$.

TABLE 17.1
Some features of the ANSI/IEEE standard floating-point number representation formats

Feature	Single/Short	Double/Long
Word width, bits	32	64
Significand bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + bias = 0, f = 0$	$e + bias = 0, f = 0$
Denormal	$e + bias = 0, f \neq 0$ represents $\pm 0.f \times 2^{-126}$	$e + bias = 0, f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm\infty$)	$e + bias = 255, f = 0$	$e + bias = 2047, f = 0$
Not-a-number (NaN)	$e + bias = 255, f \neq 0$	$e + bias = 2047, f \neq 0$
Ordinary number	$e + bias \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	$e + bias \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
min	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.2 \times 10^{-308}$
max	$\approx 2^{128} \approx 3.4 \times 10^{38}$	$\approx 2^{1024} \approx 1.8 \times 10^{308}$

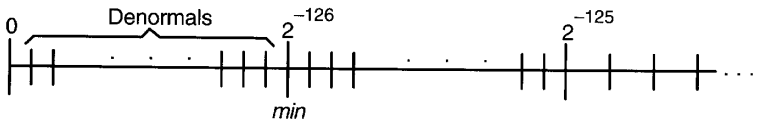


Fig. 17.4 Denormals in the IEEE single-precision format.

The IEEE floating-point standard also defines the four basic arithmetic operations (add, subtract, multiply, divide), as well as square-root, with regard to the expected precision in their results. Basically, the results of these operations must match the results that would be obtained if all intermediate computations were carried out with infinite precision. Thus, it is up to the designers of floating-point hardware units adhering to the IEEE standard to carry sufficient precision in intermediate results to satisfy this requirement.

Finally, the IEEE standard defines extended formats that allow implementations to carry higher precisions internally to reduce the effect of accumulated errors. Two extended formats are defined:

Single-extended: $\geq$ 11 bits for exponent, $\geq$ 32 bits for significand
(Bias unspecified, but exponent range must include [−1022, 1023].)

Double-extended: $\geq$ 15 bits for exponent, $\geq$ 64 bits for significand
(Bias unspecified, but exponent range must include [−16382, 16383].)

The use of an extended format does not, in and of itself, guarantee that the precision requirements of floating-point operations will be satisfied. Rather, extended formats are useful for controlling error propagation in a sequence of arithmetic operations. For example, when adding a list of floating-point numbers, a more precise result is obtained if positive and negative values are added separately, with the two subtotals combined in a final addition (we discuss computation errors in Chapter 19). Now if the list of numbers has thousands of elements, it is quite possible that computing one or both subtotals will lead to overflow. If an extended format is used (single-extended with single-precision operands, double-extended for double-precision operands), overflow becomes much less likely.

17.3 BASIC FLOATING-POINT ALGORITHMS

Basic arithmetic on floating-point numbers is conceptually simple. However, care must be taken in hardware implementations for ensuring correctness and avoiding undue loss of precision; in addition, it must be possible to handle any exceptions.

Addition and subtraction are the most difficult of the elementary operations for floating-point operands. Here, we deal only with addition, since subtraction can be converted to addition by flipping the sign of the subtrahend. Consider the addition:

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = \pm s \times b^{e}$$

Assuming $e1 \geq e2$, we begin by aligning the two operands through right-shifting of the significand $s2$ of the number with the smaller exponent:

$$\pm s2 \times b^{e2} = \frac{\pm s2}{b^{e1-e2}} \times b^{e1}$$

If the exponent base b and the number representation radix r are the same, we simply shift $s2$ to the right by $e1 - e2$ digits. When $b = r^a$ the shift amount, which is computed through direct subtraction of the biased exponents, is multiplied by a. In either case, this step is referred to as *alignment shift*, or *preshift* (in contrast to *normalization shift* or *postshift*, which is needed when the resulting significand s is unnormalized). We then perform the addition as follows:

$$\begin{aligned}(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) &= (\pm s1 \times b^{e1}) + \left(\frac{\pm s2}{b^{e1-e2}} \times b^{e1}\right) \\ &= \left(\pm s1 \pm \frac{s2}{b^{e1-e2}}\right) \times b^{e1} = \pm s \times b^{e}\end{aligned}$$

When the operand signs are alike, a single-digit normalizing shift is always enough. For example, with the IEEE format, we have $1 \leq s < 4$, which may have to be reduced by a factor of 2 through a single-bit right shift (and adding 1 to the exponent to compensate). However, when the operands have different signs, the resulting significand may be very close to 0 and left shifting by many positions may be needed for normalization. Overflow/underflow can occur during the addition step as well as due to normalization.

Floating-point multiplication is simpler than floating-point addition; it is performed by multiplying the significands and adding the exponents:

$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = \pm(s1 \times s2) \times b^{e1+e2}$$

Postshifting may be needed, since the product $s1 \times s2$ of the two significands can be unnormalized. For example, with the IEEE format, we have $1 \leq s1 \times s2 < 4$, leading to the possible need for a single-bit right shift. Also, the computed exponent needs adjustment if the exponents are biased or if a normalization shift is performed. Overflow/underflow is possible during multiplication if $e1$ and $e2$ have like signs; overflow is also possible due to normalization.

Similarly, floating-point division is performed by dividing the significands and subtracting the exponents:

$$\frac{\pm s1 \times b^{e1}}{\pm s2 \times b^{e2}} = \pm\frac{s1}{s2} \times b^{e1-e2}$$

Here, problems to be dealt with are similar to those of multiplication. The ratio $s1/s2$ of the significands may have to be normalized. With the IEEE format, we have $1/2 < s1/s2 < 2$ and a single-bit left shift is always adequate. The computed exponent needs adjustment if the exponents are biased or if a normalizing shift is performed. Overflow/underflow is possible during division if $e1$ and $e2$ have unlike signs; underflow due to normalization is also possible.

To extract the square root of a positive floating-point number, we first make its exponent even. This may require subtracting 1 from the exponent and multiplying the significand by b. We then use the following:

$$\sqrt{s \times b^{e}} = \sqrt{s} \times b^{e/2}$$

In the case of IEEE floating-point numbers, the adjusted significand will be in the range $1 \leq s < 4$, which leads directly to a normalized significand for the result. Square-rooting never produces overflow or underflow.

In the preceding discussion, we ignored the need for rounding. The product $s1 \times s2$ of two significands, for example, may have more digits than can be accommodated. When such a value is rounded so that it is representable with the available number of digits, the result may have to be normalized and the exponent adjusted again. Thus, though the event is quite unlikely, rounding can potentially lead to overflow or underflow as well.

17.4 CONVERSIONS AND EXCEPTIONS

An important requirement for the utility of a floating-point system is the ability to convert decimal or binary numbers from/to the format for input/output purposes. Also, at times we need to convert numbers from one floating-point format to another (say from double- to single-precision, or from single-precision to extended-single). These conversions, and their error characteristics, are also spelled out as part of the ANSI/IEEE standard.

Whenever a number with higher precision is to be converted to a format offering lower precision (e.g., double-precision or extended-single to single-precision), rounding is required as part of the conversion process. The same applies to conversions between integer and floating-point formats. Because of their importance, rounding methods, are discussed separately in Section 17.5. Here, we just mention that the ANSI/IEEE standard includes four rounding modes:

Round to nearest even.

Round toward zero (inward).

Round toward $+\infty$ (upward).

Round toward $-\infty$ (downward).

The first of these is the default rounding mode. The latter two rounding modes are optional and find applications in performing interval arithmetic (see Section 19.5).

Another important requirement for any number representation system is defining the order of values in comparisons that yield true/false results. Such comparisons are needed for conditional computations such as "if $x > y$ then $\cdots$". The ANSI/IEEE standard defines comparison results in a manner that is consistent with mathematical laws and intuition. Clearly comparisons of ordered values (ordinary floating-point numbers, ± 0, and $\pm\infty$) should yield the expected results (e.g., $-\infty < +0$ should yield "true"). The two representations of 0 are considered to be the same number, so $+0 > -0$ yields "false." It is somewhat less clear what the results of comparisons such as NaN $\neq$ NaN (true) or NaN $\leq +\infty$ (false) should be. The general rule is that NaN is considered unordered with everything, including itself. Thus, comparisons such as NaN $\leq +\infty$ also produce an "invalid operation" exception.

When the values being compared have different formats (e.g., single vs. single-extended or single vs. double), the result of comparison is defined based on infinitely precise versions of the two numbers being compared.

Besides the exception signaled when certain comparisons between unordered values are performed, the ANSI/IEEE standard also defines exceptions associated with divide by zero, overflow, underflow, inexact result, and invalid operation. The first three conditions are obvious. The "inexact exception" is signaled when the rounded result of an operation or conversion is not exactly representable. The "invalid operation" exception occurs in the following situations, among others:

Addition:	$(+\infty) + (-\infty)$
Multiplication:	$0 \times \infty$
Division:	$0/0$ or ∞/∞
Square-root:	Operand < 0

For a more complete description, refer to the ANSI/IEEE standard document [IEEE85].

17.5 ROUNDING SCHEMES

Rounding is needed to convert higher-precision values, or intermediate computation results with additional digits, to lower-precision formats for storage and/or output. In the discussion that follows, we assume that an unsigned number with integer and fractional digits is to be rounded to an integer.

$$x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l} \xrightarrow{\text{round}} y_{k-1}y_{k-2}\cdots y_1y_0.$$

The simplest rounding method is truncation or chopping, which is accomplished by dropping the extra bits:

$$x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l} \xrightarrow{\text{chop}} x_{k-1}x_{k-2}\cdots x_1x_0.$$

The effect of chopping is different for signed-magnitude and 2's-complement numbers. Figure 17.5 shows the effect of chopping on a signed-magnitude number. The magnitude of the result $y = \text{chop}(x)$ is always smaller than the magnitude of x. Thus, this is sometimes referred to as "round toward 0." Figure 17.6 shows that chopping a 2's-complement number always reduces its value. This is known as "downward-directed rounding" or "rounding toward $-\infty$".

With the "round to nearest" (rtn) scheme, depicted in Fig. 17.7 for signed-magnitude numbers, a fractional part of less than 1/2 is dropped, while a fractional part of 1/2 or more (.1xxx $\cdots$ in binary) leads to rounding to the next higher integer. The only difference when

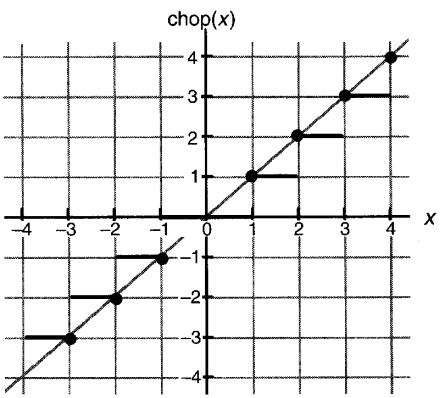


Fig. 17.5 Truncation or chopping of a signed-magnitude number (same as round toward 0).

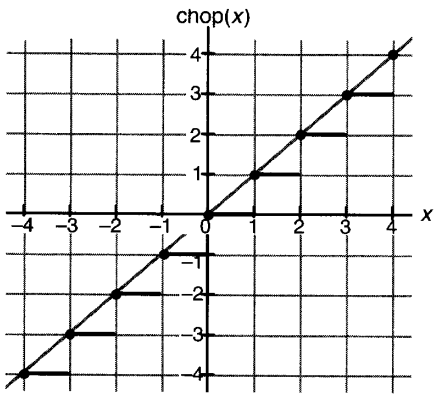


Fig. 17.6 Truncation or chopping of a 2's-complement number (same as downward-directed rounding, or rounding toward $-\infty$).

this rule is applied to 2's-complement numbers is that in Fig. 17.7, the heavy dots for negative values of x move to the left end of the respective heavy lines. Thus, a slight upward bias is created. Such a bias exists for signed-magnitude numbers as well if we consider only positive or negative values.

To understand the effect of this slight bias on computations, assume that a number $(x_{k-1} \cdots x_1 x_0 . x_{-1} x_{-2})_{\text{two}}$ is to be rounded to an integer $y_{k-1} \cdots y_1 y_0$. The four possible cases, and their representation errors are:

$x_{-1}x_{-2} = 00$	Round down	error $= 0$
$x_{-1}x_{-2} = 01$	Round down	error $= -0.25$
$x_{-1}x_{-2} = 10$	Round up	error $= 0.5$
$x_{-1}x_{-2} = 11$	Round up	error $= 0.25$

If these four cases occur with equal probability, the average error is 0.125. The resulting bias may create problems owing to error accumulation. In practice, the situation may be somewhat

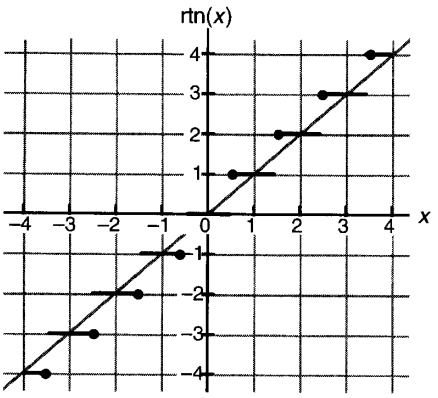


Fig. 17.7 Rounding of a signed-magnitude value to the nearest number.

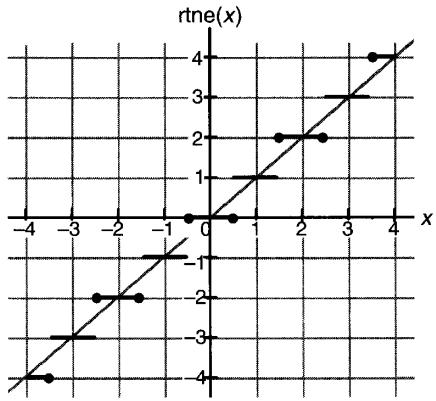


Fig. 17.8 Rounding to the nearest even number.

worse in that for certain calculations, the probability of getting a midpoint value can be much higher than 2^{-l}.

One way to deal with the preceding problem is to always round to an even (or odd) integer, thus causing the "midpoint" values ($x_{-1}x_{-2} = 10$ in our example) to be rounded up or down with equal probabilities. Rounding to the nearest even (rather than odd) value has the additional benefit that it leads to "rounder" values and, thus, lesser errors downstream in the computation. Figure 17.8 shows the effect of the "round to nearest even" (rtne) scheme on signed-magnitude numbers. The diagram for 2's-complement numbers is the same (since, e.g., -1.5 will be rounded to -2 in either case). Round-to-nearest-even is the default rounding scheme of the IEEE floating-point standard.

Another scheme, known as R* rounding, is similar to the preceding methods except that for midpoint values (e.g., when $x_{-1}x_{-2} = 10$), the fractional part is chopped and the least significant bit of the rounded result is forced to 1. Thus, in midpoint cases, we round up if the least significant bit happens to be 0 and round down when it is 1. This is clearly the same as the "round to nearest odd" scheme. Figure 17.9 contains a graphical representation of R* rounding.

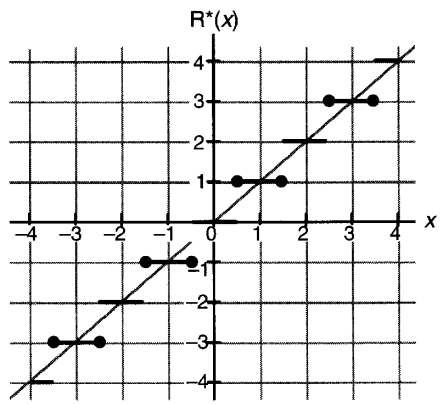


Fig. 17.9 R* rounding or rounding to the nearest odd number.

In all the rounding schemes discussed thus far, full carry-propagation over the k integer positions is needed in the worst case. This imposes an undesirable overhead on floating-point arithmetic operations, especially since the final rounding is always on the critical path. The next two methods, which eliminate this overhead, are not used in practice because they are accompanied by other problems.

Jamming, or von Neumann rounding, is simply truncation with the least significant bit forced to 1. As shown in Fig. 17.10, this method combines the simplicity of chopping with the symmetrical error characteristics of ordinary rounding (not rounding to nearest even). However, its worst-case error is twice as large as that of rounding to the nearest integer.

ROM rounding is based on directly reading a few of the least significant bits of the rounded result from a table, using the affected bits, plus the most significant (leftmost) dropped bit, as the address. For example, if the 4 bits $y_3y_2y_1y_0$ of the rounded result are to be determined, a 32 × 4 ROM table can be used that takes $x_3x_2x_1x_0x_{-1}$ as the address and supplies 4 bits of data:

$$x_{k-1}\cdots x_4x_3x_2x_1x_0.x_{-1}\cdots x_{-l} \xrightarrow{32\times4\text{–ROM–round}} x_{k-1}\cdots x_4y_3y_2y_1y_0.$$

ROM address: $x_3x_2x_1x_0x_{-1}$; ROM data: $y_3y_2y_1y_0$

Thus, in the preceding example, the fractional bits of x are dropped, the 4 bits read out from the table replace the 4 least significant integral bits of x, and the higher-order bits of x do not change. The ROM output bits $y_3y_2y_1y_0$ are related to the address bits $x_3x_2x_1x_0x_{-1}$ as follows:

$$(y_3y_2y_1y_0)_{\text{two}} = (x_3x_2x_1x_0)_{\text{two}} \qquad \text{when } x_{-1} = 0 \text{ or } x_3 = x_2 = x_1 = x_0 = 1$$
$$(y_3y_2y_1y_0)_{\text{two}} = (x_3x_2x_1x_0)_{\text{two}} + 1 \qquad \text{otherwise}$$

Thus, the rounding result is the same as that of the round to nearest scheme in 15 of the 16 possible cases, but a larger error is introduced when $x_3 = x_2 = x_1 = x_0 = 1$. Figure 17.11 depicts the results of ROM rounding for a smaller 8 × 2 table.

Finally, we sometimes need to force computational errors to be in a certain known direction. For example, if we are computing an upper bound for some quantity, larger results are acceptable, since the derived upper bound will still be valid, but results that are smaller than correct values

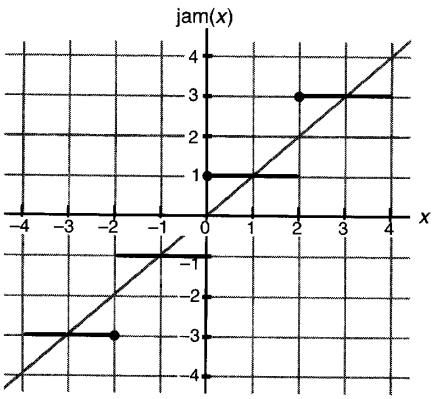


Fig. 17.10 Jamming or von Neumann rounding.

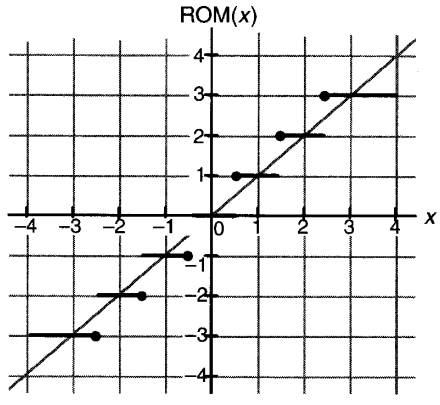


Fig. 17.11 ROM rounding with an 8 × 2 table.

could invalidate the upper bound. This leads to the definition of upward-directed rounding (round toward $+\infty$) and downward-directed rounding (round toward $-\infty$) schemes depicted in Figs. 17.12 and 17.6, respectively. Upward- and downward-directed rounding schemes are optional features of the IEEE floating-point standard.

17.6 LOGARITHMIC NUMBER SYSTEMS

Fixed-point representations can be viewed as extreme special cases of floating-point numbers with the exponent equal to 0, thus making the exponent field unnecessary. The other extreme of removing the significand field, and assuming that the significand is always 1, is known as logarithmic number representation. With the IEEE floating-point standard terminology, the significand of a logarithmic number system consists only of the hidden 1 and has no fractional part.

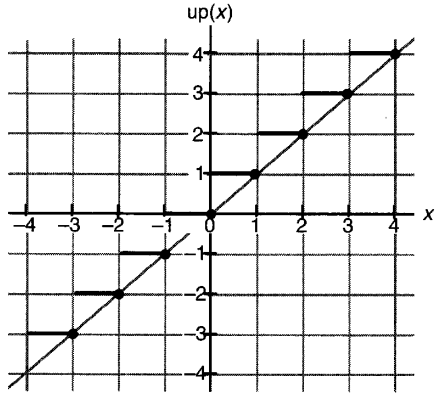


Fig. 17.12 Upward-directed rounding, or rounding toward $+\infty$ (see Fig. 17.6 for downward-directed rounding, or rounding toward $-\infty$).

The components of a logarithmic number are its sign, exponent base b (not explicitly shown), and exponent e, together representing the number $x = \pm b^e$. Since the relationship between x and e can be written as

$$e = \log_b |x|$$

we often refer to b as the logarithm base, rather than the exponent base, and to the number system as the sign-and-logarithm representation. Of course, if e were an integer, as is the case in floating-point representations, only powers of b would be representable. So we allow e to have a fractional part (Fig. 17.13). Since numbers between 0 and 1 have negative logarithms, the logarithm must be viewed as a signed number or all numbers scaled up by a constant factor (the logarithm part biased by the logarithm of that constant) if numbers less than 1 are to be representable. The base b of the logarithm is usually taken to be 2.

In what follows, we will assume that the logarithm part is a 2's-complement number. A number x is thus represented by a pair:

$$(Sx, Lx) = (\text{sign}(x), \log_2 |x|)$$

■ **Example 17.1** Consider a 12-bit, base-2, logarithmic number system in which the 2's-complement logarithm field contains 5 whole and 6 fractional bits. The exponent range is thus $[-16, 16 - 2^{-6}]$, leading to a number representation range of approximately $[-2^{16}, 2^{16}]$, with $min = 2^{-16}$. The bit pattern

1 1 0 1 1 0 0 0 1 0 1 1

Sign Radix point

represents the number $-2^{-9.828125} \approx -(0.0011)_{\text{ten}}$.

Multiplication and division of logarithmic numbers are quite simple, and this constitutes the main advantage of logarithmic representations. To multiply, we XOR the signs and add the logarithms:

$$(\pm 2^{e1}) \times (\pm^{e2}) = \pm 2^{e1+e2}$$

To divide, we XOR the signs and subtract the logarithms:

$$\frac{\pm 2^{e1}}{\pm^{e2}} = \pm 2^{e1-e2}$$

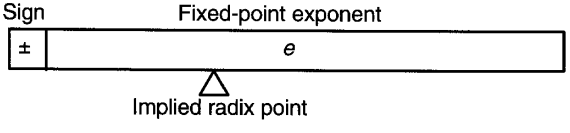


Fig. 17.13 Logarithmic number representation with sign and fixed-point exponent.

Addition/subtraction of logarithmic numbers is equivalent to solving the following problem: given $\log x$ and $\log y$, find $\log(x \pm y)$. This is somewhat more difficult than multiplication or division. Straightforward table lookup requires a table of size $2^{2k} \times k$ with k-bit representations (including the sign bit), so it is impractical unless the word width k is fairly small (say, 8–12 bits). A more practical hardware realization scheme is presented in Section 18.6.

Number conversions from binary to logarithmic, and from logarithmic to binary, representation involve computing the logarithm and inverse logarithm (exponential) functions. These are covered in Chapters 22 and 23, which deal with methods of function evaluation.

PROBLEMS

17.1 Unnormalized floating-point numbers In an unnormalized floating-point representation format, a significand of 0 with any exponent can be used to represent 0, since $0 \times 2^e = 0$. Argue that even in this case, it is beneficial to represent 0 with the smallest possible exponent. *Hint:* Consider floating-point addition.

17.2 Spacing of floating-point numbers

a. In Fig. 17.4, three of the vertical tick marks have been labeled with the numbers 0, 2^{-126}, and 2^{-125}. Supply the labels for the remaining 13 tick marks shown.

b. Draw a similar diagram for the double-precision format and label its tick marks.

17.3 Floating-point puzzle You are given a bit string $x_{k-1}x_{k-2}\cdots x_1x_0$ and told that it is a floating-point number. You can make no assumption about the format except that it consists of a sign bit, an exponent field, and a significand field with their usual meanings (i.e., you cannot assume that the sign is the leftmost bit, that 1 means negative, or that the exponent is to the left of the significand). Your goal is to decode the format and find the number being represented by asking a *minimal* number of questions in the worst case. Questions must be about the format, not the number itself, and must be posed so that they can be answered yes/no or with an integer (e.g., How many bits are there in the exponent field?). Present your strategy in the form of a decision tree.

17.4 Floating-point representations Consider the IEEE 32-bit standard floating-point format.

a. Ignoring $\pm\infty$, denormals, etc., how many distinct real numbers are representable?

b. What is the smallest number of bits needed to represent this many distinct values? What is the encoding or representation efficiency of this format?

c. Discuss the consequences (in terms of range and precision) of shortening the exponent field by 2 bits, adding 2 bits to the significand field, and using the exponent base of 16 instead of 2.

17.5 Fixed- and floating-point representations Find the largest value of n for which $n!$ can be represented exactly in the following two formats. Explain the results.

a. 32-bit, 2's-complement integer format.

b. 32-bit IEEE standard floating-point format.

17.6 Fixed- versus floating-point systems Digital signal processor chips are special-purpose processors that have been tailored to the need of signal processing applications. They come in both fixed-point and floating-point versions. Discuss the issues involved in choosing a fixed- versus floating-point DSP chip for such applications [Inac96].

17.7 Floating-point arithmetic operations Represent each of the following floating-point operands in 32-bit IEEE standard format. Then perform the specified operations, normalizing the results if necessary.

a. $(+41 \times 2^{+0}) \times (+0.875 \times 2^{-16})$

b. $(-4.5 \times 2^{-1}) \div (+0.0625 \times 2^{+12})$

c. $\sqrt{+1.125 \times 2^{+11}}$

d. $(+1.25 \times 2^{-10}) + (+0.5 \times 2^{+11})$

e. $(-1.5 \times 2^{-11}) + (+0.625 \times 2^{-10})$

17.8 Floating-point exceptions Give examples of IEEE 32-bit standard floating-point numbers x and y such that they produce overflow in the rounding stage of computing $x + y$. Repeat for computing the product $x \times y$. Then show that rounding overflow is impossible in the normalization phase of floating-point division.

17.9 Conversion of floating-point numbers The conversion problem for floating-point numbers involves changing representations from radix r with exponent base b to radix R with exponent base B.

a. Describe the conversion process for the special case of $r = b$ and $R = B$.

b. Apply the method of part a to convert $(0.2313\ 0130)_{\text{four}} \times 4^{(-0211)_{\text{four}}}$ from $r = 4$ to $R = 10$.

c. Describe a shortcut method for the conversion when $r = \beta^g$ and $R = \beta^G$ for some β.

d. Apply the shortcut method of part c to convert the radix-4 floating-point number of part b to radix $R = 8$.

17.10 Denormalized floating-point numbers The ANSI/IEEE floating-point standard allows denormalized numbers to be used when the results obtained are too small for normalized representation.

a. Can floating-point numbers be compared as integers even when denormals are considered?

b. Is it possible for an operation involving one or two denormals to yield a normalized result?

c. Prove or disprove: the sum of two denormals is always exactly representable.

17.11 Errors in floating-point representations Only some real numbers are exactly representable in the ANSI/IEEE standard floating-point format (or any finite number representation method for that matter).

a. Plot the absolute representation error of the IEEE single format for a number x in $[1, 16)$, as a function of x, using logarithmic scales for both x and the error value.

b. Repeat part a for the relative representation error in $[1, 16)$.

c. What are the worst-case relative and absolute representation errors in $[1,16)$?

d. Does the relative (absolute) error get better or worse for numbers greater than 16? What about for numbers less than 1?

17.12 Round-to-nearest-even The following example shows the advantage of rounding to nearest even over ordinary rounding. All numbers are decimal. Consider the floating-point numbers $u = .100 \times 10^0$ and $v = -.555 \times 10^{-1}$. Let $u^{(0)} = u$ and use the recurrence $u^{(i+1)} = (u^{(i)} -_{\text{fp}} v) +_{\text{fp}} v$ to compute $u^{(1)}, u^{(2)}, \cdots$. With ordinary rounding, we get the sequence of values.101,.102, $\cdots$, an occurrence known as *drift* [Knut81, p. 222]. Verify that drift does not occur in the preceding example if round to nearest even is used. Then prove the general result $(((u +_{\text{fp}} v) -_{\text{fp}} v) +_{\text{fp}} v) -_{\text{fp}} v = (u +_{\text{fp}} v) -_{\text{fp}} v$ when floating-point operations are exactly rounded using the round-to-nearest-even rule.

17.13 ROM rounding

a. In ROM rounding, only the most significant one of the bits to be dropped is used as part of the ROM address. Is there any benefit to using the other dropped bits as part of the address?

b. Discuss the feasibility of compensating for the downward bias of ROM rounding (because of using truncation in the one special case) through the introduction of upward bias in some cases.

17.14 Logarithmic number systems Consider a 16-bit sign-and-logarithm number system, using $k = 6$ whole and $l = 9$ fractional bits for the logarithm. Assume that the logarithm base is 2 and that 2's-complement representation is used for negative logarithms.

a. Find the smallest and largest positive numbers that can be represented.

b. Calculate the maximum relative representation error.

c. Find the representations of $x = 2.5$ and $y = 3.7$ in this number system.

d. Perform the operations $x \times y$, x/y, $1/x$, x^2, and $\sqrt{x}$, in this number system.

e. Find the representations of $x + y$, $x - y$, and x^y, using a calculator where needed.

f. Repeat part b, this time assuming that the logarithm base is 10.

17.15 Logarithmic number systems Compare a sign-and-logarithm number system with 8 whole bits, 23 fractional bits, and a bias of 127, to the 32-bit IEEE standard floating-point format with regard to range and precision. Devise methods for converting numbers between the two formats.

17.16 Semilogarithmic number systems Consider a floating-point system in which the exponent is a multiple of 2^{-h} (i.e., it is a fixed-point number with h fractional bits) and the k-bit significand is in $[1, 1 + 2^{-h})$ with $h + 1$ hidden bits $1.00 \cdots 0$. The extremes of $h = 0$ and $h = k$ in such a semilogarithmic number system [Mull98] correspond to floating-point and logarithmic number systems.

a. What are possible advantages of such a number system?

b. Describe basic arithmetic algorithms for semilogarithmic numbers.

c. Develop algorithms for conversion of such numbers to/from floating-point.

d. Compare a semilogarithmic number system to floating-point and logarithmic number systems with regard to representation error.

REFERENCES

[Camp62] Campbell, S. G., "Floating-Point Operation," in *Planning a Computer System: Project Stretch*, W. Buchholz, (ed.), McGraw-Hill, 1992, pp. 92–121.

[Holm97] Holmes, W. N., "Composite Arithmetic: Proposal for a New Standard," *IEEE Computer*, Vol. 30, No. 3, pp. 65–73, 1997.

[IEEE85] *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985), IEEE Press, 1985.

[Inac96] Inacio, C., and D. Ombres, "The DSP Decision: Fixed Point or Floating?" *IEEE Spectrum*, Vol. 33, No. 9, pp. 72–74, 1996.

[Knut81] Knuth, D. E., *The Art of Computer Programming*, 2nd ed., *Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

[Kuck77] Kuck, D. J., D. S. Parker, and A. H. Sameh, "Analysis of Rounding Methods in Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 26, No. 7, pp. 643–650, 1977.

[Mull98] Muller, J.-M., A. Scherbyna, and A. Tisserand, "Semi-Logarithmic Number Systems," *IEEE Trans. Computers*, Vol. 47, No. 2, pp. 145–151, 1998.

[Swar75] Swartzlander, E. E., and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Trans. Computers*, Vol. 24, No. 12, pp. 1238–1242, 1975.

[Yohe73] Yohe, J. M., "Roundings in Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 22, No. 6, pp. 577–586, 1973.

[Yoko92] Yokoo, H., "Overflow/Underflow-Free Floating-Point Number Representations with Self-Delimiting Variable-Length Exponent Field," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 1033–1039, 1992.

Chapter 18 FLOATING-POINT OPERATIONS

In this chapter, we examine hardware implementation issues for the four basic floating-point arithmetic operations of addition, subtraction, multiplication, and division. Consideration of square-rooting is postponed to Section 21.6. The bulk of our discussions concern the handling of exponents, alignment of significands, and normalization and rounding of the results. Arithmetic operations on significands, which are fixed-point numbers, have already been covered. Chapter topics include:

18.1 FLOATING-POINT ADDERS/SUBTRACTORS

A floating-point adder/subtractor consists of a fixed-point adder for the aligned significands, plus support circuitry to deal with the signs, exponents, alignment preshift, normalization postshift, and special values (0, $\pm\infty$, etc.). Figure 18.1 is the block diagram of a floating-point adder. The major components of this adder are described in Sections 18.1–18.3. Floating-point multipliers and dividers, which are relatively simpler, are covered in Sections 18.4 and 18.5, respectively.

As shown in Fig. 18.1, the two operands entering the floating-point adder are first unpacked. Unpacking involves:

Separating the sign, exponent, and significand for each operand and reinstating the hidden 1.

Converting the operands to the internal format, if different (e.g., single-extended or double-extended).

Testing for special operands and exceptions (e.g., recognizing NaN inputs and bypassing the adder).

The difference of the two exponents is used to determine the amount of alignment right shift and the operand to which it should be applied. To economize on hardware, preshifting capability is often provided for only one of the two operands, with the operands swapped if the other one needs to be shifted. Since the computed sum or difference may have to be shifted to the left in the post normalization step, several bits of the right-shifted operand, which normally would be discarded as they moved off the right end, may be kept for the addition. Thus, the significand adder is typically wider than the significands of the input numbers. More on this in Section 18.3.

Similarly, complementation logic may be provided for only one of the two operands (typically the one that is not preshifted, to shorten the critical path). If both operands have the same sign, the common sign can be ignored in the addition process and later attached to the result. If $-x$ is the negative operand and complementation logic is provided only for y, which is positive, y is complemented and the negative sign of $-x$ ignored, leading to the result $x - y$ instead of $-x + y$. This negation is taken into account by the sign logic in determining the correct sign of the result.

Selective complementation, and the determination of the sign of the result, are also affected by the $+/-$ control input of the floating-point adder/subtractor, which specifies the operation to be performed.

With IEEE standard floating-point format, the sum/difference of the aligned significands has a magnitude in the range $[0, 4)$. If the result is in $[2, 4)$, then it is too large and must be normalized by shifting it one bit to the right and incrementing the tentative exponent to compensate for the shift. If the result is in $[0, 1)$, it is too small. In this case, a multibit left shift may be required, along with a compensatory reduction of the exponent.

Note that a positive (negative) 2's-complement number $(x_1 x_0 . x_{-1} x_{-2} \cdots)_{2\text{'s-compl}}$ whose magnitude is less than 1 will begin with two or more 0s (1s). Hence, the amount of left shift needed is determined by a special circuit known as *leading zeros/ones counter*. It is also possible, with a somewhat more complex circuit, to *predict* the number of leading zeros/ones in parallel with the addition process rather than detecting them after the addition result becomes known. This removes the leading zeros/ones detector from the critical path and improves the overall speed. Details are given in Section 18.2.

Rounding the result may necessitate another normalizing shift and exponent adjustment. To improve the speed, adjusted exponent values can be precomputed and the proper value selected once the normalization results become known. To obtain a properly rounded floating-point sum or difference, a binary floating-point adder must maintain at least three extra bits beyond the *ulp*; these are called *guard bit*, *round bit*, and *sticky bit*. The roles of these bits, along with the hardware implementation of rounding, are discussed in Sections 18.3.

The significand adder is almost always a fast logarithmic time 2's-complement adder, usually with carry-lookahead design. Two's-complement addition is the preferred choice because with 1's-complement addition, the end-around carry can cause speed degradation in fast adders. Two's-complementation does not cause any difficulty because at most one of the operands is complemented and the addition of *ulp* can be performed by setting the carry-in of the significand adder to 1 (see problem 18.4). When the resulting significand is negative, it must be complemented to form the signed-magnitude output. As usual, this is done by 1's-complementation and addition of *ulp*. The latter addition can be merged with the addition of *ulp*, which may be needed for rounding. Thus, 0, *ulp* or 2*ulp* will be added to the true or complemented output of the significand adder during the rounding process.

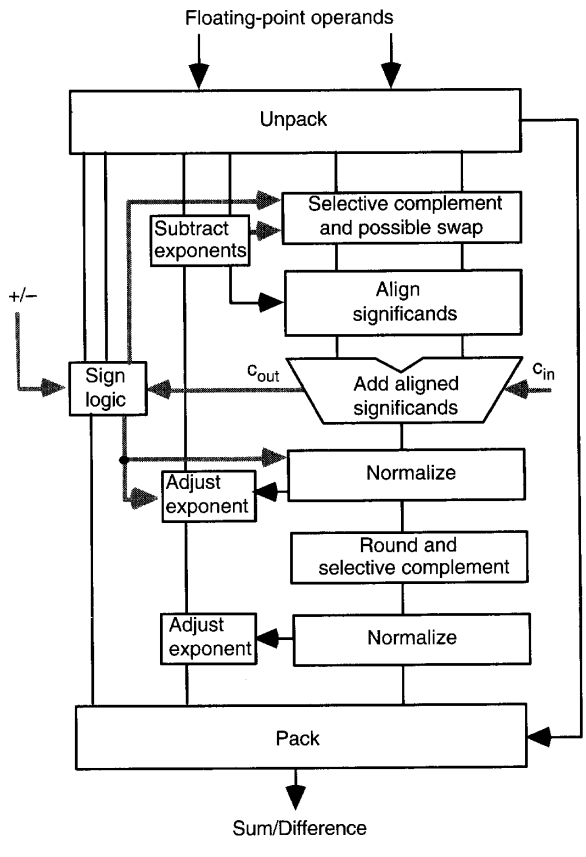


Fig. 18.1 Block diagram of a floating-point adder/subtractor.

If an equally fast adder can be designed for 1's-complement numbers, then 1's-complementation becomes the preferred choice, especially when results are to be rounded by chopping.

Finally, packing the result involves:

Combining the sign, exponent, and significand for the result and removing the hidden 1.

Testing for special operands and exceptions (e.g., zero result, overflow, or underflow).

Note that unlike the unpacking step, conversion between the internal and external formats is not included in the packing process. This is because converting a wider significand to a narrower one requires rounding and is best accomplished in the rounding stage, which produces the result with the desired output precision.

Floating-point adders found in various processors may differ in details from the generic design depicted in Fig. 18.1. However, the basic principles are the same, and the differences in implementation relate to clever schemes for speeding up the various subcomputations or for economizing on hardware cost. Some of these techniques are covered in Sections 18.2 and 18.3.

18.2 PRE- AND POSTSHIFTING

The preshifter always shifts to the right by an amount equal to the difference of the two exponents. Note that with the IEEE single-precision floating-point format, the difference of the two exponents can be as large as $127 - (-126) = 253$. However, even with extra bits of precision maintained during addition, the operands and results are much narrower. This allows us to simplify and speed up the exponent subtractor and preshift logic in Fig. 18.1.

For example, if the adder is 32 bits wide, then any preshift of 32 bits or more will result in the preshifted input becoming 0. Thus, only the least significant 5 bits of the exponent difference needs to be computed, with the preshifted input forced to 0 when the difference is 32 or more.

Let us continue with the assumption that right shifts of 0 to 31 bits must be implemented. In principle, this can be done by a set of 32-to-1 multiplexers, as shown in Fig. 18.2. The multiplexer producing the bit y_i of the shifted operand selects one of the bits x_i through x_{i+31} of the (sign-extended) 32-bit input that is being aligned based on the 5-bit shift amount. Such a design, however, would lead to fan-in and fan-out problems, especially for the sign bit, which will have to feed multiple inputs of several multiplexers.

As usual, a multistage design can be used to mitigate the fan-in and fan-out problems. Figure 18.3 shows a portion of a combinational shifter that can preshift an input operand x by any amount from 0 to 15 bits. Each circular node is a 2-to-1 multiplexer, with its output fanned out to two nodes in the level below. The four levels, from top to bottom, correspond to shifting by 1, 2, 4, and 8 bits, respectively.

In practice, designs that fall between the two extremes shown in Figs. 18.2 and 18.3 are used. For example, preshifts of up to 31 bits might be implemented in two stages, one performing any shift from 0 to 7 bits and the other performing shifts of 0, 8, 16, and 24 bits. The first stage is then controlled by the three least significant bits, and the second stage by the two most significant bits, of the binary shift amount.

Note that the difference $e1 - e2$ of the two (biased) exponents may be negative. The sign of the difference indicates which operand is to be preshifted, while the magnitude provides the shift amount. One way to obtain the shift amount in case of a negative difference is to complement it. However, this introduces additional delay due to carry propagation. A second way is to use a ROM table or PLA that receives the signed difference as input and produces the shift amount as output. A third way is to compute both $e1 - e2$ and $e2 - e1$, choosing the positive value as the shift amount. Given that only a few bits of the difference need to be computed, duplicating the exponent subtractor does not have significant cost implications.

The postshifter is similar to the preshifter, with one difference: it should be able to perform either a right shift of 0–1 bit or a left shift of 0–31 bits, say. One hardware implementation option is to use two separate shifters for right- and left-shifting. Another option is to combine the two functions into one multistage combinational shifter. Supplying the details in the latter case is left as an exercise.

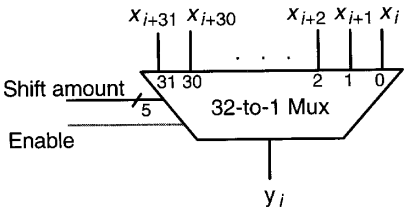


Fig. 18.2 One bit slice of a single-stage preshifter.

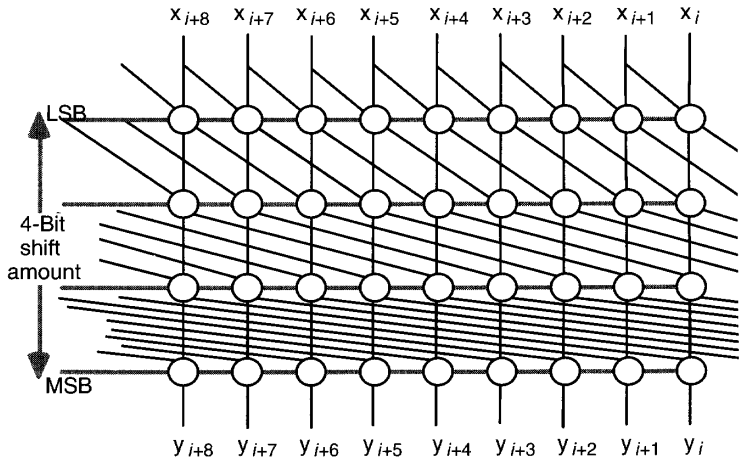


Fig. 18.3 Four-stage combinational shifter for preshifting an operand by 0 to 15 bits.

For IEEE floating-point operands, the need for right-shifting by 1 bit during normalization is indicated by the magnitude of the adder output equaling or exceeding 2. The adder output is a 2's-complement number in the range $(-4, 4)$, represented as $z = (c_{\text{out}} z_1 z_0 . z_{-1} z_{-2} \cdots)_{2\text{'s-compl}}$. The condition for right-shifting is thus easily determined as $c_{\text{out}} \neq z_1$. Assuming that right-shifting is not needed for normalization, we must have $c_{\text{out}} = z_1$, with the left-shift amount then determined by the number of consecutive bits in z that are identical to z_1. So, if $z_1 = 0$ (1), we need to detect the number of consecutive 0s (1s) in z, beginning with z_0. As mentioned in Section 18.1, this is done either by applying a leading zeros/ones counter to the adder output or by predicting the number of leading zeros/ones concurrently with the addition process (to shorten the critical path). The two schemes are depicted in Fig. 18.4.

Leading zeros/ones counting is quite simple and is thus left as an exercise. Predicting the number of leading zeros/ones can be accomplished as follows. Note that when the inputs to a floating-point adder are normalized, normalization left shift is needed only when the operands, and thus the inputs to the significand adder, have unlike signs. Leading zeros/ones prediction for unnormalized inputs is somewhat more involved, but not more difficult conceptually.

Let the inputs to the significand adder be 2's-complement positive and negative values $(0x_0 . x_{-1} x_{-2} \cdots)_{2\text{'s-compl}}$ and $(1y_0 . y_{-1} y_{-2} \cdots)_{2\text{'s-compl}}$. Let there be exactly i consecutive positions, beginning with position 0, that propagate the carry during addition. Borrowing the carry "generate," "propagate," and "annihilate" notation from our discussions of adders, we have the following:

$$\begin{aligned} p_0 &= p_{-1} = p_{-2} = \cdots = p_{-i+1} = 1 \\ p_{-i} &= 0 \quad \text{(i.e., } g_{-i} = 1 \text{ or } a_{-i} = 1\text{)} \end{aligned}$$

In case $g_{-i} = 1$, let j be the smallest index such that:

$$\begin{aligned} g_{-i} &= a_{-i-1} = a_{-i-2} = \cdots = a_{-j+1} = 1 \\ a_{-j} &= 0 \quad \text{(i.e., } g_{-j} = 1 \text{ or } p_{-j} = 1\text{)} \end{aligned}$$

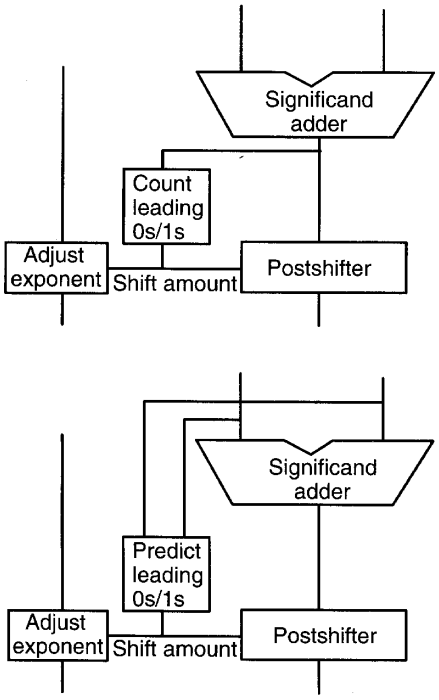


Fig. 18.4 Leading zeros/ones counting versus prediction.

Then, we will have j or $j-1$ leading 0s depending on whether the carry leaving position j is 0 or 1, respectively.

In case $a_{-i} = 1$, let j be the smallest index such that:

$$a_{-i} = g_{-i-1} = g_{-i-2} = \cdots = g_{-j+1} = 1$$
$$g_{-j} = 0 \quad \text{(i.e., } p_{-j} = 1 \text{ or } a_{-j} = 1\text{)}$$

Then, we will have $j-1$ or j leading 1s, depending on whether the carry-out of position j is 0 or 1, respectively.

Note that the g, p, a, and carry signals needed for leading zeros/ones prediction can be extracted from the significand adder to save on hardware. Based on the preceding discussion, given the required signals, the circuit needed to predict the number of leading zeros/ones can be designed with two stages. The first stage, which is similar to a carry-lookahead circuit, produces a 1 in the jth position and 0s in all positions to its left (this can be formulated as a parallel prefix computation, since we are essentially interested in detecting one of the four patterns $pp\cdots ppgaa\cdots aag$, $pp\cdots ppgaa\cdots aap$, $pp\cdots ppagg\cdots gga$, or $pp\cdots ppagg\cdots ggp$). The second stage is an encoder or priority encoder (depending on the design of the first stage) that yields the index of the leading 1.

Finally, in the preceding discussion, we assumed separate hardware for pre- and postshifting. This is a desirable choice for higher-speed or pipelined operation. If the two shifters are to be combined for economy, the unit must be capable of shifting both to the right and to the left

by an arbitrary amount. Modifying the design of Fig. 18.3 to derive a bidirectional shifter is straightforward.

18.3 ROUNDING AND EXCEPTIONS

If an alignment preshift is performed, the bits that are shifted out should not all be discarded, since they can potentially affect the rounding of the result. Recall that proper floating-point addition/subtraction requires that the result match what would be obtained if the computation were performed with infinite precision and the result rounded. In may thus appear that we have to keep all bits that are shifted out in case left-shifting is later needed for normalization. Keeping all the bits that are shifted out effectively doubles the width of the significand adder.

We know from earlier discussions that the significand adder must be widened by one bit at the left to accommodate the sign bit of its 2's-complement inputs. It turns out that widening the adder by 3 bits at the right is adequate for obtaining properly rounded results. Calling the three extra bits at the right G, R, and S, for reasons to become apparent shortly, the output of the significand adder can be represented as follows:

$$\text{Adder output} = (c_{\text{out}} z_1 z_0 . z_{-1} z_{-2} \cdots z_{-l} GRS)_{2\text{'s-compl}}$$

In the preceding equation, z_1 is the sign indicator, c_{out} represents significand overflow, and the extra bits at the right are:

G: Guard bit

R: Round bit

S: Sticky bit

We next explain the roles of the G, R, and S bits and why they are adequate for proper rounding. The explanation is in terms of the IEEE floating-point format, but it is valid in general.

When an alignment right-shift of 1 bit is performed, G will hold the bit that is shifted out and no precision is lost (so, G "guards" against loss of precision). For alignment right shifts of 2 bits or more, the shifted significand will have a magnitude in [0, 1/2). Since the magnitude of the unshifted significand is in [1, 2), the difference of the aligned significands will have a magnitude in [1/2, 2). Thus, in this latter case, the normalization left shift will be by at most one bit, and G is still adequate to protect us against loss of precision.

In case a normalization left shift actually takes place, the "round bit" is needed for determining whether to round the resulting significand down ($R = 0$, discarded part $< ulp/2$) or up ($R = 1$, discarded part $\geq ulp/2$). All that remains is to establish whether the discarded part is exactly equal to $ulp/2$. This information is needed in some rounding schemes, and providing it is the role of the "sticky bit," which is set to the logical OR of all the bits that are shifted through it. Thus, following an alignment right shift of 7 bits, say, the sticky bit will be set to the logical OR of the 5 bits that move past G and R. This logical ORing operation can be accommodated in the design of the preshifter (how?).

The effect of 1-bit normalization shifts on the rightmost few bits of the significand adder output is as follows

Before postshifting (z)	$\cdots$	z_{-l+1}	z_{-l}	G	R	S
1-bit normalizing right-shift	$\cdots$	z_{-l+2}	z_{-l+1}	z_{-l}	G	$R \vee S$
1-bit normalizing left-shift	$\cdots$	z_{-l}	G	R	S	0
After normalization (Z)	$\cdots$	Z_{-l+1}	Z_{-l}	Z_{-l-1}	Z_{-l-2}	Z_{-l-3}

where the Z_h are the final digit values in the various positions, after any normalizing shift has been applied. Note that during a normalization right shift, the new value of the sticky bit is set to the logical OR of its old value and the value of R. Given a positive normalized result Z, we can round it to nearest even by simply dropping the extra 3 bits and:

Doing nothing	if $Z_{-l-1} = 0$ or $Z_{-l} = Z_{-l-2} = Z_{-l-3} = 0$
Adding $ulp = 2^{-l}$	otherwise

Note than no rounding is necessary in case of a multibit normalizing left shift, since full precision is preserved in this case. Other rounding modes can be implemented similarly.

Overflow and underflow exceptions are easily detected by the exponent adjustment blocks in Fig. 18.1. Overflow can occur only when we have a normalizing right shift, while underflow is possible only with normalizing left shifts. Exceptions involving NaNs and invalid operations are handled by the unpacking and packing blocks in Fig. 18.1. One remaining issue is the detection of a zero result and encoding it as the all-zeros word. Note that detection of a zero result is essentially a by-product of the leading zeros/ones detection discussed earlier. Determining when the "inexact" exception must be signaled is left as an exercise.

18.4 FLOATING-POINT MULTIPLIERS

A floating-point multiplier consists of a fixed-point multiplier for the significands, plus peripheral and support circuitry to deal with the exponents and special values (0, $\pm\infty$, etc.). Figure 18.5 depicts a generic block diagram for a floating-point multiplier. The role of unpacking is exactly as discussed for floating-point adders at the beginning of Section 18.1. Similarly, the final packing of the result is done as for floating-point adders. The sign of the product is obtained by XORing the signs of the two operands.

A tentative exponent is computed by adding the two biased exponents and subtracting the bias from the sum. With the ANSI/IEEE short floating-point format, subtracting the bias of 127 can be easily accomplished by providing a carry-in of 1 into the exponent adder and subtracting 128 from the sum. This latter subtraction amounts to simply flipping the most significant bit of the result.

The significand multiplier is the slowest and most complex part of the unit shown in Fig. 18.5. With the IEEE floating-point format, the product of the two unsigned significands, each in the range [1, 2), will be in the range [1, 4). Thus, the result may have to be normalized by shifting it one position to the right and incrementing the tentative exponent. Rounding the result may necessitate another normalizing shift and exponent adjustment. When each significand has a hidden 1 and l fractional bits, the significand multiplier is an unsigned $(l + 1) \times (l + 1)$

multiplier that would normally yield a $(2l + 2)$-bit product. Since this full product must be rounded to $l + 1$ bits at the output, it may be possible to discard the extra bits gradually as they are produced, rather than in a single step at the end. All that is needed is to keep an extra round bit and a sticky bit to be able to round the final result properly. Keeping a guard bit is not needed here (why?).

To improve the speed, the incremented exponent can be precomputed and the proper value selected once it is known whether a normalization postshift is required. Since multiplying the significands is the most complex part of floating-point multiplication, there is ample time for such computations. Also, rounding need not be a separate step at the end. With proper design, it may be possible to incorporate the bulk of the rounding process in the multiplication hardware.

To see how, note that most multipliers produce the least significant half of the product earlier than the rest of the bits. So, the bits that will be used for rounding are produced early in the multiplication cycle. However, the need for normalization right shift becomes known at or near the end. Since there are only two possibilities (no postshift or a right shift of 1 bit), we can devise a stepwise rounding scheme by developing two versions of the rounded product and selecting the correct version in the final step.

Because floating-point multiplication consists of several sequential stages or subcomputations, it is quite simple and natural to pipeline it for increased throughput. Pipeline latches can be inserted across the natural block boundaries in Fig. 18.5 as well as within the significand multiplier if the latter is of the full-tree or array variety. Chapter 25 presents a detailed discussion of pipelining considerations and design methods.

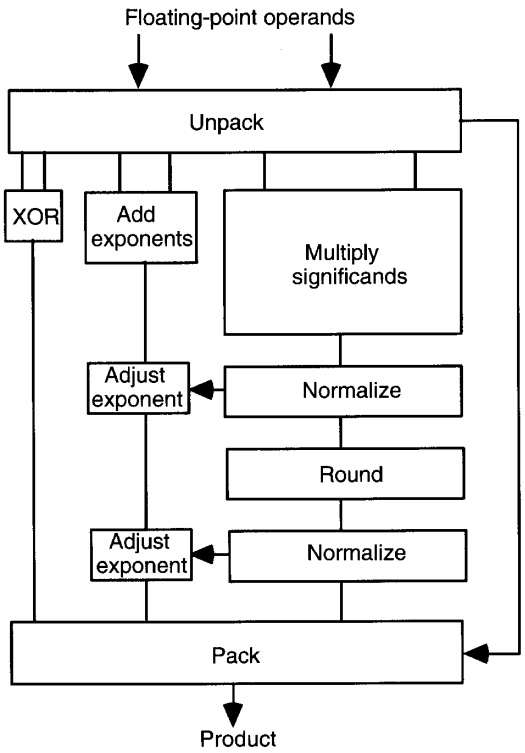


Fig. 18.5 Block diagram of a floating-point multiplier.

18.5 FLOATING-POINT DIVIDERS

A floating-point divider has the same overall structure as a floating-point multiplier. Figure 18.6 is a generic block diagram for a floating-point divider. The two operands of floating-point division are unpacked, the resulting components pass through several computation steps, and the final result is packed into the appropriate format for output. Unpacking and packing have the same roles here as those discussed for floating-point adders in Section 18.1 (the divide-by-0 exception is detected during unpacking). The sign of the quotient is obtained by XORing the operand signs.

A tentative exponent is computed by subtracting the divisor's biased exponent from the dividend's biased exponent and adding the bias to the difference. With the ANSI/IEEE short floating-point format, the bias of 127 must be added to the difference of the two exponents. Since adding 128 is simpler than adding 127, we can compute the difference less one by holding c_{in} to 0 in a 2's-complement subtraction (normally, in 2's-complement subtraction, $c_{in} = 1$) and then flipping the most significant bit of the result.

The significand divider is the slowest and the most complex part of the unit shown in Fig. 18.6. With ANSI/IEEE floating-point format, the ratio of two significands in [1, 2) is in the range (1/2, 2). Thus, the result may have to be normalized by shifting it one position to the left and decrementing the tentative exponent. Rounding the result may necessitate another normalizing shift and exponent adjustment.

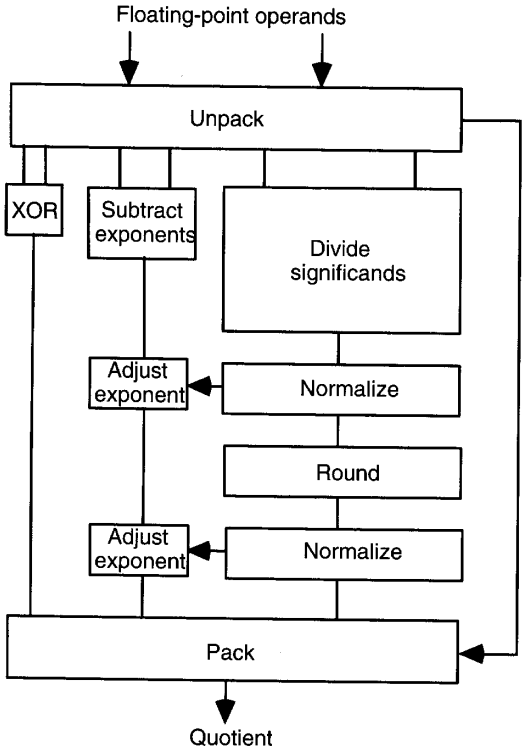


Fig. 18.6 Block diagram of a floating-point divider.

As in the case of multiplication, speed can be gained by precomputing the adjusted exponent and selecting the proper value when the need for normalization becomes known. Since dividing the significands is the most complex part of floating-point division, there is ample time for such computations. Considerations for pipelining of the computations are also quite similar to those of floating-point multiplication (see Section 18.4).

One main difference between floating-point division and multiplication is in rounding. Since the significand divider's output may have to be left-shifted by 1 bit for normalization, the quotient must be developed with two extra bits that serve as the guard and round bits (see the discussion of rounding for floating-point addition in Section 18.3). In division schemes that produce a remainder, the final remainder is used to derive the value of the sticky bit (how?). Then, the rounding process discussed at the end of Section 18.3 is applied. Convergence division creates some difficulty for rounding in view of the absence of a remainder.

As was the case for fixed-point multipliers and dividers, floating-point multipliers and dividers can share much hardware. In particular, when the significand division is performed by one of the convergence methods discussed in Chapter 16, little additional hardware is required to convert a floating-point multiplier into a floating-point multiply/divide unit.

18.6 LOGARITHMIC ARITHMETIC UNIT

As discussed in Section 17.6, representing numbers by their signs and base-b logarithms offers the advantage of simple multiplication and division, inasmuch as these operations are converted to addition and subtraction of the logarithms, respectively. In this section, we demonstrate the algorithms and hardware needed for adding and subtracting logarithmic numbers and present the design of a complete logarithmic arithmetic unit.

We noted, in Section 17.6, that addition and subtraction of logarithmic numbers can, in principle, be performed by table lookup. One method of reducing the size of the required table is via converting the two-operand (binary) operation of interest to a single-operand (unary) operation that needs a smaller table. Consider the add/subtract operation

$$(Sx, Lx) \pm (Sy, Ly) = (Sz, Lz)$$

for logarithmic operands and assume $x > y > 0$ (other cases are similar). Then:

$$\begin{aligned} Lz = \log z &= \log(x \pm y) = \log(x(1 \pm y/x)) \\ &= \log x + \log(1 \pm y/x) \end{aligned}$$

Note that $\log x$ is known and $\log(y/x)$ is easily computed as $\Delta = -(\log x - \log y)$. Given Δ, the term

$$\log(1 \pm y/x) = \log(1 \pm \log^{-1} \Delta)$$

is easily obtained by table lookup (two tables, φ^+ and φ^-, are needed). Hence, addition and subtraction of logarithmic numbers can be based on the following computations:

$$\begin{aligned} \log(x + y) &= \log x + \varphi^+(\Delta) \\ \log(x - y) &= \log x + \varphi^-(\Delta) \end{aligned}$$

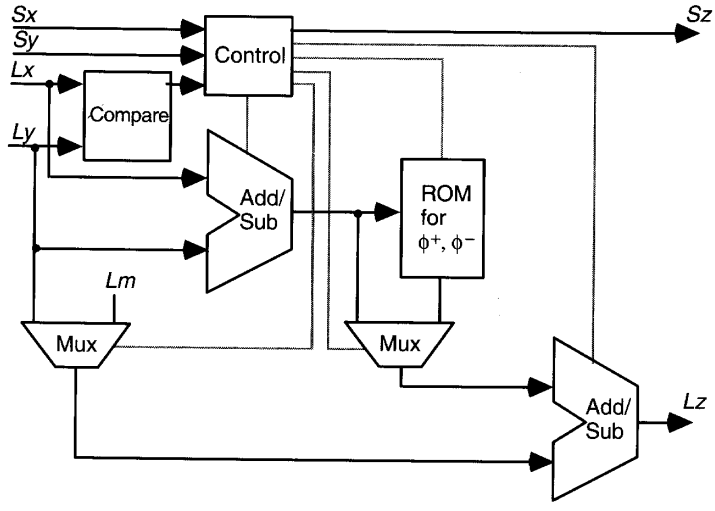


Fig. 18.7 Arithmetic unit for a logarithmic number system.

Figure 18.7 depicts a complete arithmetic unit for logarithmic numbers. For addition and subtraction, Lx and Ly are compared to determine which one is larger. This information is used by the control box for properly interpreting the result of the subtraction $Lx - Ly$. The reader should be able to supply the details.

The design of Fig. 18.7 assumes the use of scaling of all values by a multiplicative factor m so that numbers between 0 and 1 are also represented with unsigned logarithms. Because of this scaling, the logarithm of the scale factor m (or the bias Lm) must be subtracted in multiplication and added in division.

PROBLEMS

18.1 Exponent arithmetic in floating-point adder

a. Design the "Subtract exponents" block of the floating-point adder in Fig. 18.1 for the IEEE standard 64-bit floating-point format. Assume that a 6-bit difference, plus a "force to zero" output, is to be provided.

b. Repeat part a, this time assuming that the output difference is to be forced to 63 if the real difference exceeds 63.

c. Compare the designs of parts a and b and discuss.

18.2 Sign logic in floating-point adder Consider the "Sign logic" block in the floating-point adder of Fig. 18.1.

a. Explain the role of the output from this block that is fed to the "Normalize" and "Adjust exponent" blocks.

b. Supply a complete logic design for this block, assuming the use of a 2's-complement significand adder.

18.3 Alignment preshifter Design an alignment preshifter for IEEE single-precision floating-point numbers that produces a shifted output with guard, round, and sticky bits.

18.4 Precision in floating-point adders Referring to the discussion at the beginning of Section 18.3, why would the width of the significand adder double if we were to keep all the bits that are shifted out during the alignment preshift? In other words, doesn't the presence of 0s in those extra positions of the unshifted operand mean that the addition width will not change? Of course, the same question applies when we keep only three extra bits of precision. Do we really have to extend the adder width by 3 bits? *Hint:* The answer depends on which operand is complemented.

18.5 Leading zeros/ones counter

a. Design a ripple-type leading zeros/ones counter for the normalization stage of floating-point addition and derive its worst-case delay. Is this a viable design?

b. Show that the problem of leading zeros/ones detection can be converted to parallel prefix logical AND.

c. Using the result of part b, design a logarithmic time, leading zeros/ones counter.

18.6 Leading zeros/ones counter

a. Use a PLA to design an 8-input leading zeros/ones counter with the following specifications: eight data inputs, two control inputs, three address (index) outputs, and one "all-zeros/ones" output. One of the control inputs specifies whether leading 0s or leading 1s should be counted. The other control input turns the tristate drivers of the address outputs on or off, thus allowing the address outputs of several modules to be tied together. The tristate drivers are also turned off when the "all-zeros/ones" output is asserted.

b. Show how two leading zeros/ones counters of the type described in part a can be cascaded to form a 16-bit leading zeros/ones counter.

c. Can the cascading scheme of part b be extended to wider inputs (say 24 or 32 bits)?

18.7 Leading zeros/ones prediction Extend the results concerning leading zeros/ones prediction, presented at the end of Section 18.2, to unnormalized inputs. *Hint:* Consider three separate cases of positive inputs, negative inputs, and inputs with unlike signs.

18.8 Rounding in floating-point operations

a. Extend the round-to-nearest-even procedure for a positive value, given near the end of Section 18.3, to a 2's-complement result Z.

b. Occasionally, when performing double-precision arithmetic, we would like to be able to specify that the result be rounded as if it were a single-precision number, with the single-rounded result then output in double-precision format. Why might such an option be useful, and how can it be implemented?

c. Show how the guard, round, and sticky bits can be used when an "inexact" exception is to be indicated following the rounding process.

18.9 Rounding in floating-point operations Given that an intermediate 2's-complement result for a floating-point operation with guard, round, and sticky bits is at hand, describe how each of the following rounding schemes can be implemented:

a. Round toward 0.

b. Round toward $+\infty$.

c. Round toward $-\infty$.

d. R* rounding (see Fig. 17.9).

18.10 Floating-point multipliers In multiplying the significands of two floating-point numbers, the lower half of the fractional part is not needed, except to properly round the upper half. Discuss whether, and if so, how, this can lead to simplified hardware for the significand multiplier. Note that the significand multiplier can have various designs (tree, array, built of AMMs, etc.).

18.11 Floating-point multiply-add unit In many computation-intensive applications, a significant fraction of floating-point multiplications are immediately followed by a floating-point addition. This justifies additional investment in hardware to build a floating-point multiply-add unit.

a. Sketch the design of such a unit. Then, enumerate, and discuss, the main sources of speedup over cascaded multiply and add operations.

b. Extend your discussion to a multiply-add unit that is optimized for inner-product computations. The unit allows several products to be computed in sequence, while maintaining a running sum of greater precision. This approach allows the rounding step to be postponed to the very end of the inner product computation.

18.12 Rounding in floating-point division

a. Explain how the sticky bit needed for properly rounding the quotient of floating-point division is derived from the final remainder.

b. Explain how a properly rounded result might be derived with convergence division.

18.13 On-the-fly rounding in division To avoid a carry-propagate addition in rounding the quotient of floating-point division, one can combine the rounding process with the on-the-fly conversion of the quotient digits from redundant to conventional binary format [Erce92]. Outline the algorithm and hardware requirements for such an on-the-fly rounding scheme.

18.14 Floating-point operations on denormals Based on what you have learned about floating-point add/subtract, multiply, and divide units in this chapter, briefly discuss design complications if denormalized numbers of the IEEE floating-point format were to be accepted as inputs and produced as output.

18.15 Logarithmic arithmetic Consider a 16-bit sign-and-logarithm number system, using $k = 6$ whole and $l = 9$ fractional bits for the logarithm. Assume that the logarithm base is 2 and that 2's-complement representation is used for negative logarithms.

a. Find the representations of $x = 2.5$ and $y = 3.7$ in this number system.

b. What is the required ROM size for the arithmetic unit of Fig. 18.7?

c. Do the operations $x + y$ and $x - y$, supplying the needed table entries φ^+ and φ^-.

18.16 Flexible floating-point processor Consider a 64-bit floating-point number representation format where the sign bit is followed by a 5-bit "exponent width" field. This field

specifies the exponent field as being 0–31 bits wide, the remaining 27–58 bits being a fractional significand with no hidden 1. Do not worry about special values such as $\pm\infty$ or NaN.

a. Enumerate the advantages and possible drawbacks of this format.

b. Outline the design of a floating-point adder to add two numbers in this format.

c. Draw a block diagram of a multiplier for flexible floating-point numbers.

d. Briefly discuss any complication in the design of a divider for flexible floating-point numbers.

18.17 Double rounding Consider the multiplication of two-digit, single-precision decimal values .34 and .78, yielding .2652. If we round this exact result to an internal three-digit, extended-precision format, we get .265, which when subsequently rounded to single precision by means of round-to-nearest-even, yields .26. However, if the exact result were directly rounded to single precision, it would yield .27.

a. Can double rounding lead to a similar problem if we always round up the halfway cases instead of applying round-to-nearest-even?

b. Prove that for floating-point operands x and y with p-bit significands, if $x + y$ is rounded to p' bits of precision ($p' \geq 2p + 2$), a second rounding to p bits of precision will yield the same result as direct rounding of the exact sum to p bits.

c. Show that the claim of part b also holds for multiplication, division, and square-rooting.

d. Discuss the implications of the preceding results for converting the results of double-precision IEEE floating-point arithmetic to single precision.

18.18 Rounding in ternary arithmetic If we had ternary as opposed to binary computers, radix-3 arithmetic would be in common use today. Discuss the effects of this change on rounding in floating-point arithmetic.

REFERENCES

[Ande67] Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Research and Development*, Vol. 11, No. 1, pp. 34–53, 1967.

[Bose87] Bose, B. K., L. Pei, G. S. Taylor, and D. A. Patterson, "Fast Multiply and Divide for a VLSI Floating-Point Unit," *Proc. 8th Symp. Computer Arithmetic*, pp. 87–94, 1987.

[Coon80] Coonen, J. T., "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *IEEE Computer*, Vol. 13, pp. 69–79, January 1980.

[Davi74] Davis, R. L., "Uniform Shift Networks," *IEEE Computer*, Vol. 7, pp. 60–71, September 1974.

[Erce92] Ercegovac, M. D., and T. Lang, "On-the-Fly Rounding," *IEEE Trans. Computers*, Vol. 41, No. 12, pp. 1497–1503, 1992.

[Gosl71] Gosling, J. B., "Design of Large High-Speed Floating-Point Arithmetic Units," *Proc. IEE*, Vol. 118, pp. 493–498.

[Mont90] Montoye, R. K., E. Hokonek, and S. L. Runyan, "Design of the Floating-Point Execution Unit in the IBM RISC System/6000," *IBM J. Research and Development*, Vol. 34, No. 1, pp. 59–70, 1990.

[Ober97] Oberman, S. F., and M. J. Flynn, "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Computers*, Vol. 46, No. 2, pp. 154–161, 1997.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Wase82] Waser, S., and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

Chapter 19 ERRORS AND ERROR CONTROL

Machine arithmetic is inexact in two ways. First, many numbers of interest, such as $\sqrt{2}$ or π, do not have exact representations. Second, floating-point operations, even when performed on exactly representable numbers, may lead to errors in the results. It is essential for arithemetic designers and serious computer users to understand the nature and extent of such errors, as well as how they can lead to results that are counterintuitive and, occasionally, totally invalid. Chapter topics include:

19.1 SOURCES OF COMPUTATIONAL ERRORS

Integer arithmetic is exact and all integer results can be trusted to be correct as long as overflow does not occur (assuming, of course, that the hardware was designed and built correctly and has not since failed; flaw- and fault-induced errors are dealt with in Chapter 27). Floating-point arithmetic, on the other hand, only approximates exact computations with real numbers. There are two sources of errors: (1) representation errors and (2) arithmetic errors.

Representation errors occur because many real numbers do not have exact machine representations. Examples include 1/3, $\sqrt{2}$, and π. Arithmetic errors, on the other hand, occur because some results are inherently inexact or need more bits for exact representation than are available. For example, a given exact operand may not have a finitely representable square root and multiplication produces a double-width result that must be rounded to single-width format.

Thus, familiarity with representation and arithmetic errors, as well as their propagation and accumulation in the course of computations, is important for the design of arithmetic algorithms and their realizations in hardware, firmware, or software. Example 19.1 illustrates the effect of representation and computation errors in floating-point arithmetic.

Example 19.1 Consider the decimal computation 1/99 –1/100, using a decimal floating-point format with a four-digit significand in [1, 10) and a single-digit signed exponent. Given that both 99 and 100 have exact representations in the given format, the floating-point divider will compute 1/99 and 1/100 accurately to within the machine precision:

$$x = 1/99 \approx 1.010 \times 10^{-2} \qquad \text{error} \approx 10^{-6} \text{ or } 0.01\%$$
$$y = 1/100 = 1.000 \times 10^{-2} \qquad \text{error} = 0$$

The precise result is 1/9900, with its floating-point representation 1.010×10^{-4} containing an approximate error of 10^{-8} or 0.01%. However, the floating-point subtraction $z = x -_{\text{fp}} y$ yields the result

$$z = 1.010 \times 10^{-2} - 1.000 \times 10^{-2} = 1.000 \times 10^{-4}$$

which has a much larger error of around 10^{-6} or 1%.

A floating-point number representation system may be characterized by a radix r (which we assume to be the same as the exponent base b), a precision p in terms of radix-r digits, and an approximation or "rounding" scheme A. We symbolize such a floating-point system as

$$\text{FLP}(r, p, \text{A})$$

where $A \in$ {chop, round, rtne, chop(g), . . . }; "rtne" stands for "round to nearest even" and chop(g) for a chopping method with g guard digits kept in all intermediate steps. Rounding schemes were discussed in Section 17.5.

Let $x = r^e s$ be an unsigned real number, normalized such that $1/r \leq s < 1$, and x_{fp} be its representation in FLP(r, p, A). Then

$$x_{\text{fp}} = r^e s_{\text{fp}} = (1 + \eta)x$$

where

$$\eta = \frac{x_{\text{fp}} - x}{x} = \frac{s_{\text{fp}} - s}{s}$$

is the relative representation error. One can establish bounds on the value of η:

A = chop	$-ulp < s_{\text{fp}} - s \leq 0$	$r \times ulp < \eta \leq 0$
A = round	$ulp/2 < s_{\text{fp}} - s \leq ulp/2$	$\lvert\eta\rvert \leq r \times ulp/2$

where $ulp = r^{-p}$. We note that the worst-case relative representation error increases linearly with r; the larger the value of r, the larger the worst-case relative error η and the greater its variations. As an example, for FLP($r = 16$, $p = 6$, chop), we have $|\eta| \leq 16^{-5} = 2^{-20}$. Such a floating-point system uses a 24-bit fractional significand. To achieve the same bound for $|\eta|$ in FLP($r = 2$, p, chop), we need $p = 21$.

Arithmetic in FLP(r, p, A) assumes that an infinite precision result is obtained and then chopped, rounded, . . . , to the available precision. Some real machines approximate this process

by keeping $g > 0$ guard digits, thus doing arithmetic in FLP(r, p, chop(g)). In either case, the result of a floating-point arithmetic operation is obtained with a relative error that is bounded by some constant η, which depends on the parameters r and p and the approximation scheme A. Consider multiplication, division, addition, and subtraction of the positive operands

$$x_{\text{fp}} = (1 + \sigma)x \text{ and } y_{\text{fp}} = (1 + \tau)y$$

with relative representation errors σ and τ, respectively, in FLP(r, p, A). Note that the relative errors σ and τ can be positive or negative.

For the multiplication operation $x \times y$, we can write

$$\begin{aligned} x_{\text{fp}} \times_{\text{fp}} y_{\text{fp}} &= (1 + \eta)x_{\text{fp}}y_{\text{fp}} = (1 + \eta)(1 + \sigma)(1 + \tau)xy \\ &= (1 + \eta + \sigma + \tau + \eta\sigma + \eta\tau + \sigma\tau + \eta\sigma\tau)xy \\ &\approx (1 + \eta + \sigma + \tau)xy \end{aligned}$$

where the last expression is obtained by ignoring second- and third-order error terms. We see that in multiplication, relative errors add up in the worst case.

Similarly, for the division operation x/y, we have:

$$\begin{aligned} x_{\text{fp}} /_{\text{fp}} y_{\text{fp}} &= \frac{(1 + \eta)x_{\text{fp}}}{y_{\text{fp}}} = \frac{(1 + \eta)(1 + \sigma)x}{(1 + \tau)y} \\ &= (1 + \eta)(1 + \sigma)(1 - \tau)(1 + \tau^2)(1 + \tau^4)(\cdots)\frac{x}{y} \\ &\approx (1 + \eta + \sigma - \tau)\frac{x}{y} \end{aligned}$$

So, relative errors add up in division just as they do in multiplication.

Now, let's consider the addition operation $x + y$:

$$\begin{aligned} x_{\text{fp}} +_{\text{fp}} y_{\text{fp}} &= (1 + \eta)(x_{\text{fp}} + y_{\text{fp}}) = (1 + \eta)(x + \sigma x + y + \tau y) \\ &= \left[(1 + \eta)\left(1 + \frac{\sigma x + \tau y}{x + y}\right)\right](x + y) \end{aligned}$$

Since $|\sigma x + \tau y| \leq max(|\sigma|, |\tau|)(x + y)$, the magnitude of the worst-case relative error in the computed sum is upper-bounded by $|\eta| + max(|\sigma|, |\tau|)$.

Finally, for the subtraction operation $x - y$, we have:

$$\begin{aligned} x_{\text{fp}} -_{\text{fp}} y_{\text{fp}} &= (1 + \eta)(x_{\text{fp}} - y_{\text{fp}}) = (1 + \eta)(x + \sigma x - y - \tau y) \\ &= \left[(1 + \eta)\left(1 + \frac{\sigma x - \tau y}{x - y}\right)\right](x - y) \end{aligned}$$

Unfortunately, $(\sigma x - \tau y)/(x - y)$ can be very large if x and y are both large but $x - y$ is relatively small (recall that τ can be negative). The arithmetic error η is also unbounded for subtraction without guard digits, as we will see shortly. Thus, unlike the three preceding operations, no bound can be placed on the relative error when numbers with like signs are being subtracted (or numbers with different signs are added). This situation is known as cancellation or loss of significance.

The part of the problem that is due to η being large can be fixed by using guard digits, as suggested by the following result.

THEOREM 19.1 In FLP(r, p, chop(g)) with $g \geq 1$ and $-x < y < 0 < x$, we have:

$$x +_{\text{fp}} y = (1 + \eta)(x + y) \text{ with } -r^{-p+1} < \eta < r^{-p-g+2}$$

COROLLARY: In FLP(r, p, chop(1))

$$x +_{\text{fp}} y = (1 + \eta)(x + y) \text{ with } |\eta| < r^{-p+1}$$

So, a single guard digit is sufficient to make the relative arithmetic error in floating-point addition or subtraction comparable to the representation error with truncation.

Example 19.2 Consider a decimal floating-point number system ($r = 10$) with $p = 6$ and no guard digit. The exact operands x and y are shown below along with their floating-point representations in the given system:

$$\begin{aligned} x &= 0.100\,000\,000 \times 10^3 & x_{\text{fp}} &= .100\,000 \times 10^3 \\ y &= -0.999\,999\,456 \times 10^2 & y_{\text{fp}} &= -.999\,999 \times 10^2 \end{aligned}$$

Then, $x + y = 0.544 \times 10^{-4}$ and $x_{\text{fp}} + y_{\text{fp}} = 10^{-4}$, but:

$$x_{\text{fp}} +_{\text{fp}} y_{\text{fp}} = .100\,000 \times 10^3 -_{\text{fp}} .099\,999 \times 10^3 = .100\,000 \times 10^{-2}$$

The relative error of the result is thus $[10^{-3} - (0.544 \times 10^{-4})]/(0.544 \times 10^{-4}) \approx 17.38$; that is, the result is 1738% larger than the correct sum! With 1 guard digit, we get:

$$x_{\text{fp}} +_{\text{fp}} y_{\text{fp}} = .100\,000\,0 \times 10^3 -_{\text{fp}} .099\,999\,9 \times 10^3 = .100\,000 \times 10^{-3}$$

The result still has a large relative error of 80.5% compared to the exact sum $x + y$; but the error is 0% with respect to the correct sum of x_{fp} and y_{fp} (i.e., what we were given to work with).

19.2 INVALIDATED LAWS OF ALGEBRA

Many laws of algebra do not hold for floating-point arithmetic (some don't even hold approximately). Such areas of inapplicability can be a source of confusion and incompatibility. For example, take the associative law of addition:

$$a + (b + c) = (a + b) + c$$

If the associative law of addition does not hold, as we will see shortly, then an optimizing compiler that changes the order of operations in an attempt to reduce the delays resulting from data dependencies may inadvertently change the result of the computation.

The following example shows that the associative law of addition does not hold for floating-point computations, even in an approximate sense:

$$a = 0.123\,41 \times 10^5 \quad b = -0.123\,40 \times 10^5 \quad c = 0.143\,21 \times 10^1$$

$$\begin{aligned} a +_{\text{fp}} (b +_{\text{fp}} c) &= (0.123\,41 \times 10^5) +_{\text{fp}} \left[(-0.123\,40 \times 10^5) +_{\text{fp}} (0.143\,21 \times 10^1)\right] \\ &= (0.123\,41 \times 10^5) -_{\text{fp}} (0.123\,39 \times 10^5) = 0.200\,00 \times 10^1 \\ (a +_{\text{fp}} b) +_{\text{fp}} c &= \left[(0.123\,41 \times 10^5) -_{\text{fp}} (0.123\,40 \times 10^5)\right] +_{\text{fp}} (0.143\,21 \times 10^1) \\ &= (0.100\,00 \times 10^1) +_{\text{fp}} (0.143\,21 \times 10^1) = 0.243\,21 \times 10^1 \end{aligned}$$

The two results $0.200\,00 \times 10^1$ and $0.243\,21 \times 10^1$ differ by about 20%. So the associative law of addition does not hold.

One way of dealing with the preceding problem is to use unnormalized arithmetic. With unnormalized arithmetic, intermediate results are kept in their original form (except as needed to avoid overflow). So normalizing left shifts are not performed. Let us redo the two computations using unnormalized arithmetic:

$$\begin{aligned} a +_{\text{fp}} (b +_{\text{fp}} c) &= (0.123\,41 \times 10^5) +_{\text{fp}} \left[(-0.123\,40 \times 10^5) +_{\text{fp}} (0.143\,21 \times 10^1)\right] \\ &= (0.123\,41 \times 10^5) -_{\text{fp}} (0.123\,39 \times 10^5) = 0.000\,02 \times 10^5 \\ (a +_{\text{fp}} b) +_{\text{fp}} c &= \left[(0.123\,41 \times 10^5) -_{\text{fp}} (0.123\,40 \times 10^5)\right] +_{\text{fp}} (0.143\,21 \times 10^1) \\ &= (0.000\,01 \times 10^5) +_{\text{fp}} (0.143\,21 \times 10^1) = 0.000\,02 \times 10^5 \end{aligned}$$

Not only are the two results the same but they carry with them a kind of warning about the extent of potential error in the result. In other words, here we know that our result is correct to only one significant digit, whereas the earlier result (0.24321×10^1) conveys five digits of accuracy without actually possessing it. Of course the results will not be identical in all cases (i.e., the associative law still does not hold), but the user is warned about potential loss of significance.

The preceding example, with normalized arithmetic and two guard digits, becomes:

$$\begin{aligned} a +_{\text{fp}} (b +_{\text{fp}} c) &= (0.123\,41 \times 10^5) +_{\text{fp}} \left[(-0.123\,40 \times 10^5) +_{\text{fp}} (0.143\,21 \times 10^1)\right] \\ &= (0.123\,41 \times 10^5) -_{\text{fp}} (0.123\,385\,7 \times 10^5) = 0.243\,00 \times 10^1 \\ (a +_{\text{fp}} b) +_{\text{fp}} c &= \left[(0.123\,41 \times 10^5) -_{\text{fp}} 0.123\,40 \times 10^5)\right] +_{\text{fp}} (0.143\,21 \times 10^1) \\ &= (0.100\,00 \times 10^1) +_{\text{fp}} (0.143\,21 \times 10^1) = 0.243\,21 \times 10^1 \end{aligned}$$

The difference has now been reduced to about 0.1%; the error is much better but still too high to be acceptable in practice.

Using more guard digits will improve the situation but the laws of algebra still cannot be assumed to hold in floating-point arithmetic. Here are some other laws of algebra that do not hold in floating-point arithmetic:

Associative law of multiplication	$a \times (b \times c) = (a \times b) \times c$
Cancellation law (for $a > 0$)	$a \times b = a \times c$ implies $b = c$
Distributive law	$a \times (b + c) = (a \times b) + (a \times c)$
Multiplication canceling division	$a \times (b/a) = b$

Before the ANSI/IEEE floating-point standard became available and widely adopted, the preceding problem was exacerbated by different ranges and precisions in the floating-point representation formats of various computers. Now, with standard representation, one of the sources of difficulties has been removed, but the fundamental problems persist.

Because laws of algebra do not hold for floating-point computations, it is desirable to determine, if possible, which of several algebraically equivalent computations yields the most accurate result. Even though no general procedure exists for selecting the best alternative, numerous empirical and theoretical results have been developed over the years that help us in organizing or rearranging the computation steps to improve the accuracy of the results. We present two examples that are indicative of the methods used. Additional examples can be found in the problems at the end of the chapter.

■ **Example 19.3** The formula $x = -b \pm d$, with $d = \sqrt{b^2 - c}$, yields the two roots of the quadratic equation $x^2 + 2bx + c = 0$. The formula can be rewritten as $x = -c/(b \pm d)$. When $b^2 >> c$, the value of d is close to $|b|$. Thus, if $b > 0$, the first formula results in cancellation or loss of significance in computing the first root $(-b + d)$, whereas no such cancellation occurs with the second formula. The second root $(-b - d)$, however, is more accurately computed based on the first formula. The roles of the two formulas are reversed for $b < 0$.

■ **Example 19.4** The area of a triangle with sides of length a, b, and c is given by the formula $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a + b + c)/2$. For ease of discussion, let $a \geq b \geq c$. When the triangle is very flat, such that $a \approx b + c$, we have $s \approx a$ and the term $s - a$ in the preceding formula causes precision loss. The following version of the formula, attributed to W. Kahan [Gold91], returns accurate results, even for flat triangles:

$$A = \frac{\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}}{4}$$

19.3 WORST-CASE ERROR ACCUMULATION

In a sequence of computations, arithmetic or round-off errors may accumulate. The larger the number of cascaded computation steps (that depend on results from earlier steps), the greater the chance for, and the magnitude of, accumulated errors. With rounding, errors of opposite signs

tend to cancel each other out in the long run, thus leading to smaller average error in the final result. Yet one cannot count on such cancellations.

For example, in computing the inner product

$$z = \sum_{i=0}^{1023} x^{(i)} y^{(i)}$$

if each multiply-add step introduces an absolute error of $ulp/2 + ulp/2 = ulp$, the total absolute error will be 1024 *ulp* in the worst case. This is equivalent to losing 10 bits of precision. As for the relative error, the situation may be worse. This is because in computing the sum of signed values, cancellations, or loss of precision, can occur in one or more intermediate steps.

The kind of worst-case analysis carried out for the preceding example is very rough, and its results are expressed in terms of the number of *significant digits* in the computation results. When cascading of computations lead to the worst-case accumulation of an absolute error of m *ulp*, the effect is equivalent to losing $\log_2 m$ bits of precision.

For our inner-product example, if we begin with 24 bits of precision, say, the result is only guaranteed to have $24 - 10 = 14$ significant digits. For more complicated computations, the worth of such a worst-case estimate decreases (the analysis might indicate that the result has no significant digit remaining).

An obvious cure for our inner-product example is to keep the double-width products in their entirety and add them to compute a double-width result, which is then rounded to single-width at the very last step. Now, the multiplications do not introduce any round-off error and each addition introduces a worst-case absolute error of $ulp^2/2$. Thus, the total error is bounded by $1024 \times ulp^2/2$ (or $n \times ulp^2/2$ when n product terms are involved). Therefore, provided overflow is not a problem, a highly accurate result is obtained. In fact, if n is smaller than $r^p = 1/ulp$, the result can be guaranteed accurate to within *ulp* (error of $n \times ulp^2/2 < ulp/2$ as described above, plus *ulp*/2 for the final rounding). This is as good as one would get with infinitely precise computation and final truncation.

The preceding discussion explains the need for performing the intermediate computations with a higher precision than is required in the final result. Carrying more precision in intermediate results is in fact very common in practice; even inexpensive calculators use several "guard digits" to protect against serious error accumulation (see Section 1.2). The IEEE floating-point standard defines extended formats associated with single- and double-precision numbers (see Section 17.2) for precisely this reason. Virtually all digital signal processors, which are essentially microprocessor chips designed with the goal of efficiently performing the computations commonly required in signal processing applications, have the built-in capability to compute inner products with very high precision (see Section 28.4).

Clearly, reducing the number of cascaded arithmetic operations counteracts the effect of error accumulation. So, using computationally more efficient algorithms has the double benefit of reducing both execution time and accumulated errors. However, in some cases, simplifying the arithmetic leads to problems elsewhere. A good example is found in numerical computations whose inherent accuracy is a function of a step size or grid resolution (numerical integration is a case in point). Since a smaller step size or finer grid leads to more computation steps, and thus greater accumulation of round-off errors, there may be an optimal choice that yields the best result with regard to the worst-case total error.

Since summation of a large number of terms is a frequent cause of error accumulation in software floating-point computations, Kahan's summation algorithm or formula is worth mentioning here. To compute $s = \sum_{i=0}^{n-1} x^{(i)}$, proceed as follows (justifying this algorithm is left as an exercise):

```
s ← x^(0)
c ← 0                          {c is a correction term}
for i = 1 to n − 1 do
   y ← x^(i) − c               {subtract correction term}
   z ← s + y
   c ← (z − s) − y             {find next correction term}
   s ← z
endfor
```

19.4 ERROR DISTRIBUTION AND EXPECTED ERRORS

Analyzing worst-case errors and their accumulation (as was done in Section 19.3) is an overly pessimistic approach, but it is necessary if guarantees are to be provided for the precision of the results. From a practical standpoint, however, the distribution of errors and their expected values may be more important. In this section, we review some results concerning average representation errors with chopping and rounding.

Denoting the magnitude of the worst-case or maximum relative representation error by MRRE, we recall that in Section 19.1 we established:

$$\text{MRRE(FLP}(r, p, \text{chop})) = r^{-p+1}$$

$$\text{MRRE(FLP}(r, p, \text{round})) = \frac{r^{-p+1}}{2}$$

In the analysis of the magnitude of average relative representation error (ARRE), we limit our attention to positive significands and begin by defining:

$$\text{ARRE(FLP}(r, p, \text{A})) = \int_{1/r}^{1} \frac{|x_{\text{fp}} - x|}{x} \frac{dx}{x \ln r}$$

where "ln" stands for the natural logarithm ($\log_e$) and $|x_{\text{fp}} - x|/x$ is the magnitude of the relative representation error for x. Multiplying this relative error by the probability density function $1/(x \ln r)$ is a consequence of the logarithmic law for the distribution of normalized significands [Tsao74]. Recall that a density function must be integrated to obtain the cumulative distribution function, prob($\varepsilon \leq z$), and that the area underneath it is 1.

Figure 19.1 plots the probability density function $1/(x \ln r)$ for $r = 2$. The density function $1/(x \ln r)$ essentially tells us that the probability of having a significand value in the range $[x, x+dx]$ is $dx/(x \ln r)$, thus leading to the integral above for the average relative representation error. Note that smaller significand values are more probable than larger values.

For a first-cut approximate analysis, we can take $|x_{\text{fp}} - x|$ to be equal to $r^{-p}/2$ for FLP(r, p, chop) and $r^{-p}/4$ for FLP(r, p, round): that is, half of the respective maximum absolute errors. Then the definite integral defining ARRE can be evaluated to yield the expected errors in the two cases:

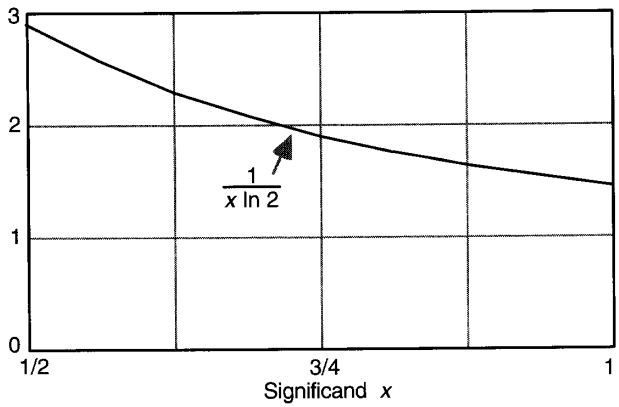


Fig. 19.1 Probability density function for the distribution of normalized significands in FLP($r = 2, p$, A).

$$\text{ARRE(FLP}(r, p, \text{chop})) \approx \int_{1/r}^{1} \frac{r^{-p}}{2x} \frac{dx}{x \ln r} = \frac{(r-1)r^{-p}}{2 \ln r}$$

$$\text{ARRE(FLP}(r, p, \text{round})) \approx \frac{(r-1)r^{-p}}{4 \ln r}$$

More detailed analyses can be carried out to derive probability density functions for the relative error $|x_{\text{fp}} - x|/x$ with various rounding schemes, which are then integrated to provide accurate estimates for the expected errors.

One such study [Tsao74] has yielded the following probability density functions for the relative error ε being equal to z with chopping and rounding:

$$\text{pdf}_{\text{chop}}(z) = \begin{cases} \dfrac{r^{p-1}(r-1)}{\ln r} & \text{for } 0 \le z < r^{-p} \\ \dfrac{1/z - r^{p-1}}{\ln r} & \text{for } r^{-p} \le z < r^{-p+1} \end{cases}$$

$$\text{pdf}_{\text{round}}(z) = \begin{cases} \dfrac{r^{p-1}(r-1)}{\ln r} & \text{for } |z| \le \dfrac{r^{-p}}{2} \\ \dfrac{1/(2z) - r^{p-1}}{\ln r} & \text{for } \dfrac{r^{-p}}{2} \le |z| < \dfrac{r^{-p+1}}{2} \end{cases}$$

Note the uniform distribution of the relative error at the low end and the reciprocal distribution for larger values of the relative error z. From the preceding probability density functions, the expected error can be easily derived:

$$\text{ARRE(FLP}(r, p, \text{chop})) = \int_{0}^{r^{-p+1}} [\text{pdf}_{\text{chop}}(z)]z \, dz = \frac{(r-1)r^{-p}}{2 \ln r}$$

$$\text{ARRE(FLP}(r, p, \text{round})) = \int_{-r^{p+1}/2}^{r^{-p+1}/2} [\text{pdf}_{\text{round}}(z)]z \, dz = \frac{(r-1)r^{-p}}{4 \ln r} \left(1 + \frac{1}{r}\right)$$

We thus see that the more rigorous analysis yields the same result as the approximate analysis in the case of chopping and a somewhat larger average error for rounding. In particular, for $r = 2$, the expected error of rounding is 3/4 (not 1/2, as the worst-case values and the approximate analysis indicate) that of chopping. These results are in good agreement with experimental results.

19.5 FORWARD ERROR ANALYSIS

Consider the simple computation $y = ax + b$ and its floating-point version:

$$y_{fp} = (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp}$$

Assuming that $y_{fp} = (1 + \eta)y$ and given the relative errors in the input operands a_{fp}, b_{fp}, and x_{fp} can we establish any useful bound on the magnitude of the relative error η in the computation result? The answer is that we cannot establish a bound on η in general, but we may be able to do it with specific constraints on the input operand ranges. The reason for the impossibility of error-bounding in general is that if the two numbers $a_{fp} \times_{fp} x_{fp}$ and b_{fp} are comparable in magnitude but different in sign, loss of significance may occur in the final addition, making the result quite sensitive to even small errors in the inputs. Example 19.2 of Section 19.1 illustrates this point.

Estimating or bounding η, the relative error in the computation result, is known as "forward error analysis": that is, finding out how far y_{fp} can be from $ax + b$, or at least from $a_{fp}x_{fp} + b_{fp}$, in the worst case. In the remainder of this section, we briefly review four methods for forward error analysis.

a. Automatic error analysis

For an arithmetic-intensive computation whose accuracy is suspect, one might run selected test cases with higher precision and observe the differences between the new, more precise, results and the original ones. If the computation under study is single precision, for example, one might use double-precision arithmetic, or execute on a multiprecision software package in lieu of double precision. If test cases are selected carefully and the differences resulting from automatic error analysis turn out to be insignificant, the computation is probably safe, although nothing can be guaranteed.

b. Significance arithmetic

Roughly speaking, *significance arithmetic* is the same as unnormalized floating-point arithmetic, although there are some fine distinctions [Ashe59], [Metr63]. By not normalizing the intermediate computation results, except as needed to correct a significand spill, we at least get a warning when precision is lost. For example, the result of the unnormalized decimal addition

$$(.1234 \times 10^5) +_{fp} (.0000 \times 10^{10}) = .0000 \times 10^{10}$$

tells us that precision has been lost. Had we normalized the second intermediate result to true zero, we would have arrived at the misleading answer $.1234 \times 10^5$. The former answer gives us a much better feel for the potential errors.

Note that if 0.0000×10^{10} is a rounded intermediate decimal result, its infinitely precise version can be any value in $[-0.5 \times 10^6, 0.5 \times 10^6]$. Thus, the true magnitude of the second operand can be several times larger than that of the first operand. Normalization would hide this information.

c. Noisy-mode computation

In noisy-mode computation, (pseudo)random digits, rather than 0s, are inserted during left shifts that are performed for normalization of floating-point results. Noisy-mode computation can be

either performed with special hardware support or programmed; in the latter case, significant software overhead is involved.

If several runs of the computation in noisy mode produce comparable results, loss of significance is probably not serious enough to cause problems. This is true because in various runs, different digits will be inserted during each normalization postshift. Getting comparable results from these runs is an indication that the computation is more or less insensitive to the random digits, and thus to the original digits that were lost as a result of cancellation or alignment right shifts.

d. Interval arithmetic

One can represent real values by intervals: an interval $[x_{lo}, x_{hi}]$ representing the real value x means that $x_{lo} \leq x \leq x_{hi}$. So, x_{lo} and x_{hi} are lower and upper bounds on the true value of x. To find $z = x/y$, say, we compute

$$[z_{lo}, z_{hi}] = [x_{lo} /_{\nabla fp} y_{hi}, x_{hi} /_{\Delta fp} y_{lo}] \quad \text{assuming } x_{lo}, x_{hi}, y_{lo}, y_{hi} > 0$$

with downward-directed rounding used in the first division ($/_{\nabla fp}$), and upward-directed rounding in the second one ($/_{\Delta fp}$), to ensure that the interval $[z_{lo}, z_{hi}]$ truly bounds the value of z.

Interval arithmetic [Moor66], [Alef83] is one the earliest methods for the automatic tracking of computational errors. It is quite intuitive, efficient, and theoretically appealing. Unfortunately, however, the intervals obtained in the course of long computations tend to widen until, after many steps, they become so wide as to be virtually worthless. Note that the span, $z_{hi} - z_{lo}$, of an interval is an indicator of the precision in the final result. So, an interval such as $[.8365 \times 10^{-3}, .2093 \times 10^{-2}]$ tells us little about the correct result.

It is sometimes possible to reformulate a computation to make the resulting output intervals narrower. Multiple computations also may help. If, using two different computation schemes (e.g., different formulas, as in Examples 19.3 and 19.4 at the end of Section 19.2) and find the intervals containing the result to be $[u_{lo}, u_{hi}]$ and $[v_{lo}, v_{hi}]$, we can use the potentially narrower interval

$$[w_{lo}, w_{hi}] = [max(u_{lo}, v_{lo}), min(u_{hi}, v_{hi})]$$

for continuing the computation or for output. We revisit interval arithmetic in Section 20.5 in connection with certifiable arithmetic computations.

19.6 BACKWARD ERROR ANALYSIS

In the absence of a general formula to bound the relative error $\eta = (y_{fp} - y)/y$ of the computation $y_{fp} = (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp}$, alternative methods of error analysis may be sought. Backward error analysis replaces the original question

> How much does the result y_{fp} deviate from the correct result y?

with another question:

> What changes in the inputs would produce the same deviation in the result?

In other words, if the exact identity $y_{fp} = a_{alt}x_{alt} + b_{alt}$ holds for alternate input parameter values a_{alt}, b_{alt}, and x_{alt}, we want to find out how far a_{alt}, b_{alt}, and x_{alt} can be from a_{fp}, b_{fp}, and x_{fp}. Thus, computation errors are, in effect, converted or compared to additional input errors.

We can easily accomplish this goal for our example computation $y = (a \times x) + b$:

$$\begin{aligned}
y_{fp} &= (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp} \\
&= (1+\mu)\big[(a_{fp} \times_{fp} x_{fp}) + b_{fp}\big] \qquad \text{with } |\mu| < r^{-p+1} = r \times ulp \\
&= (1+\mu)\big[(1+\nu)a_{fp}x_{fp} + b_{fp}\big] \qquad \text{with} |\nu| < r^{-p+1} = r \times ulp \\
&= (1+\mu)a_{fp}\,(1+\nu)x_{fp} + (1+\mu)b_{fp} \\
&= (1+\mu)(1+\sigma)a\,(1+\nu)(1+\delta)x + (1+\mu)(1+\gamma)b \\
&\approx (1+\sigma+\mu)a\,(1+\delta+\nu)x + (1+\gamma+\mu)b
\end{aligned}$$

So the approximate solution of the original problem is viewed as the exact solution of a problem close to the original one (i.e., with each input having an additional relative error of μ or ν). According to the preceding analysis, we can assure the user that the effect of arithmetic errors on the result y_{fp} is no more severe than that of $r \times ulp$ additional error in each of the inputs a, b, and x. If the inputs are not precise to this level anyway, then arithmetic errors should not be a concern.

More generally, we do the computation $y_{fp} = f_{fp}(x_{fp}^{(1)}, x_{fp}^{(2)}, \cdots, x_{fp}^{(n)})$, where the subscripts "fp" indicate approximate operands and computation. Instead of trying to characterize the difference between y (the exact result) and y_{fp} (the result obtained), we try to characterize the difference between $x_{fp}^{(i)}$ and $x_{alt}^{(i)}$ such that the identity $y_{fp} = f(x_{alt}^{(1)}, x_{alt}^{(2)}, \cdots, x_{alt}^{(n)})$ holds exactly, with f being the exact computation. When it is applicable, this method is very powerful and useful.

PROBLEMS

19.1 Representation errors In Section 19.1, the maximum relative representation error was related to *ulp* using the assumption $1/r \leq s < 1$. Repeat the analysis, this time assuming $1 \leq s < r$ (as in IEEE floating-point standard format, e.g.). Explain your results.

19.2 Variations in rounding

a. Show that in FLP(r, p, A) with even r, choosing round-to-nearest-even for $r/2$ odd, and round-to-nearest-odd for $r/2$ even, can reduce the errors. *Hint:* Successively round the decimal fraction 4.4445, each time removing one digit [Knut81].

b. What about FLP(r, p, A) with an odd radix r?

19.3 Addition errors with guard digits

a. Prove Theorem 19.1, given near the end of Section 19.1.

b. Is the error derived in Example 19.1 of Section 19.1 consistent with Theorem 19.1?

c. Redo the computation of Example 19.2 in Section 19.1 with two guard digits.

d. Is it beneficial to have more than one guard digit as far as the worst-case error in floating-point addition is concerned?

19.4 Errors with guard digits

a. Show that in FLP(r, p, chop) with no guard digit, the relative error in addition or subtraction of exactly represented numbers can be as large as $r - 1$.

b. Show that if $x - y$ is computed with one guard digit and $y/2 \leq x \leq 2y$, the result is exact.

c. Modify Example 19.2 of Section 19.1 such that the relative arithmetic error is as close as possible to the bound given in the corollary to Theorem 19.1.

19.5 Optimal exponent base in a floating-point system Consider two floating-point systems, FLP($r = 2^a$, p, A) and FLP($r = 2^b$, q, A), comparable ranges, and the same total number w of bits.

a. Derive a relationship between a, b, p, and q. *Hint:* Assume that x and y bits are used for the exponent parts and use the identity $x + ap = y + bq = w - 1$.

b. Using the relationship of part a, show that FLP($r = 2$, p, A) provides the lowest worst-case relative representation error among all floating-point systems with comparable ranges and power-of-2 radices.

19.6 Laws of algebra In Section 19.2, examples were given to show that the associative law of addition may be violated in floating-point arithmetic. Provide examples that show the violation of the other laws of algebra listed in Section 19.2.

19.7 Laws of algebra for inequalities

a. Show that with floating-point arithmetic, if $a < b$, then $a +_{\text{fp}} c \leq b +_{\text{fp}} c$ holds for all c; that is, adding the same value to both sides of a strict inequality cannot affect its direction but may change the strict "$<$" relationship to "$\leq$."

b. Show that if $a < b$ and $c < d$, then $a +_{\text{fp}} c \leq b +_{\text{fp}} d$.

c. Show that if $c > 0$ and $a < b$, then $a \times_{\text{fp}} c \leq b \times_{\text{fp}} c$.

19.8 Equivalent computations Evaluating expressions of the form $(1+g)^n$, where $g << 1$, is quite common in financial calculations. For example, g might be the daily interest rate ($0.06/365 \approx 0.000\ 164\ 383\ 6$ with a 6% annual rate) for a savings account that compounds interest daily. In calculating $1 +_{\text{fp}} g$, many bits of g are lost as a result of the alignment right shift. This error is then amplified when the result is raised to a large power n. The preceding expression can be rewritten as $e^{n \ln(1+g)}$. Even if an accurate natural logarithm function LN is available such that LN(x) is within *ulp*/2 of $\ln x$, our problem is still not quite solved since LN($1 +_{\text{fp}} g$) may not be close to $\ln(1 + g)$. Show that, for $g << 1$, computing $\ln(1 + g)$ as g when $1 +_{\text{fp}} g = g$ and as $[g \times_{\text{fp}} \text{LN}(1 +_{\text{fp}} g)]/_{\text{fp}} [(1 +_{\text{fp}} g) -_{\text{fp}} 1]$ when $1 +_{\text{fp}} g \neq 1$ provides good relative error.

19.9 Equivalent computations Assume that x and y are numbers in FLP(r, p, chop(g)), $g \geq 1$.

a. Show that the midpoint of the interval $[x, y]$, obtained from $(x +_{\text{fp}} y)/_{\text{fp}} 2$ may not be within the interval but that $x +_{\text{fp}} ((y -_{\text{fp}} x)/_{\text{fp}} 2)$ always is.

b. Show that the relative error in the floating-point calculation $(x \times_{\text{fp}} x) -_{\text{fp}} (y \times_{\text{fp}} y)$ can be quite large but that $(x -_{\text{fp}} y) \times_{\text{fp}} (x +_{\text{fp}} y)$ yields good relative error.

c. Assume that the library program SQRT has good relative error. Show that calculating $1 -_{fp} \text{SQRT}(1 -_{fp} x)$ may lead to bad worst-case relative error but that $x /_{fp} [1 + \text{SQRT}(1 -_{fp} x)]$ is safe.

19.10 Errors in radix conversion

a. Show that when a binary single-precision IEEE floating-point number is converted to the closest eight-digit decimal number, the original binary number may not be uniquely recoverable from the resulting decimal version.

b. Would nine decimal digits be adequate to remedy the problem stated in part a? Fully justify your answer.

19.11 Kahan's summation algorithm

a. Apply Kahan's summation algorithm, presented at the end of Section 19.3, to the example computations in Section 19.2 showing that the associative law of addition does not hold in floating-point arithmetic. Explain the results obtained.

b. Provide an intuitive justification for the use of the correction term c in Kahan's summation algorithm.

19.12 Distribution of significand values

a. Verify that Fig. 19.1 does in fact represent a probability density function.

b. Find the average value of a normalized binary significand x based on Fig. 19.1 and comment on the result.

19.13 Error distribution and expected errors

a. Verify that $\text{pdf}_{\text{chop}}(z)$ and $\text{pdf}_{\text{round}}(z)$, introduced near the end of Section 19.4, do in fact represent probability density functions.

b. Verify that the probability density functions of part a lead to the ARRE values derived near the end of Section 19.4.

c. Provide an intuitive explanation for the expected error in rounding being somewhat more than half that of truncation.

19.14 Noisy-mode computation Perform the computation $(a +_{fp} b) +_{fp} c$, where $a = .123\,41 \times 10^5$, $b = -.123\,40 \times 10^5$, and $c = .143\,21 \times 10^1$ four times in noisy mode, using pseudorandom digits during normalization left shifts. Compare and discuss the results.

19.15 Interval arithmetic You are given the decimal floating-point numbers $x = .100 \times 10^0$ and $y = -.555 \times 10^{-1}$.

a. Use interval arithmetic to compute the mean of x and y via the arithmetic expression $(x +_{fp} y) /_{fp} 2$.

b. Repeat part a, this time using the arithmetic expression $x +_{fp} [(y -_{fp} x) /_{fp} 2]$.

c. Combine the results of parts a and b into a more precise resulting interval. Discuss the result.

d. Repeat parts a, b, and c with the equivalent computations $(x \times_{fp} x) -_{fp} (y \times_{fp} y)$ and $(x -_{fp} y) \times_{fp} (x +_{fp} y)$.

e. Repeat parts a, b, and c with the equivalent computations $1 -_{fp} \text{SQRT}(1 -_{fp} x)$ and $x/_{fp} [1 + \text{SQRT}(1 -_{fp} x)]$, assuming that the library program SQRT provides precisely rounded results.

19.16 Backward error analysis An $(n-1)$th-degree polynomial in x, with the coefficient of the ith-degree term denoted as $c^{(i)}$, is evaluated with at least one guard digit by using Horner's rule (i.e., n computation steps, each involving a floating-point multiplication by x followed by a floating-point addition). Using backward error analysis, show that this procedure, has allowed us to compute a polynomial with coefficients $(1 + \eta^{(i)})c^{(i)}$, and find a bound for $\eta^{(i)}$. Then, show that if $c^{(i)} \geq 0$ for all i and $x > 0$, a useful bound can be placed on the relative error of the final result.

19.17 Computational errors

a. Armed with what you have learned from this chapter, reexamine the sources of computation errors in Problem 1.1 of Chapter 1. Describe your findings using the terminology introduced in this chapter.

b. Repeat part a for Problem 1.2.

c. Repeat part a for Problem 1.3.

REFERENCES

[Alef83] Alefeld, G., and J. Herzberger, *An Introduction to Interval Computations*, Academic Press, 1983.

[Ashe59] Ashenhurst, R.L., and N. Metropolis, "Unnormalized Floating-Point Arithmetic," *J. ACM*, Vol. 6, pp. 415–428, March 1959.

[Cody73] Cody, W.J., "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 22, No. 6, pp. 598–601, 1973.

[Gold91] Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 5–48, March 1991.

[Knut81] Knuth, D.E., *The Art of Computer Programming*, 2nd ed., *Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

[Kuck77] Kuck, D.J., D.S. Parker, and A.H. Sameh, "Analysis of Rounding Methods in Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 26, No. 7, pp. 643–650, 1977.

[McKe67] McKeenan, W. M., "Representation Error for Real Numbers in Binary Computer Arithmetic," *IEEE Trans. Computers*, Vol. 16, pp. 682–683, 1967.

[Metr63] Metropolis, N., and R.L. Ashenhurst, "Basic Operations in an Unnormalized Arithmetic System," *IEEE Trans. Electronic Computers*, Vol. 12, pp. 896–904, 1963.

[Moor66] Moore, R., *Interval Analysis*, Prentice-Hall, 1966.

[Ster74] Sterbenz, P.H., *Floating-Point Computation*, Prentice-Hall, 1974.

[Tsao74] Tsao, N., "On the Distribution of Significant Digits and Roundoff Errors," *Commun. ACM*, Vol. 17, No. 5, pp. 269–271, 1974.

Chapter 20 PRECISE AND CERTIFIABLE ARITHMETIC

In certain application contexts, where wrong answers might jeopardize operational safety, or even endanger human lives, all system functions must be certifiable. In the case of arithmetic, this means either doing exact calculations or the ability to put strict upper bounds on the errors (fail-safe mode) and/or on the probability of intolerable errors (probabilistic certification). In this chapter, we review methods for performing arithmetic operations with greater precision and/or with guaranteed error bounds. Chapter topics include:

20.1 HIGH PRECISION AND CERTIFIABILITY

Numerical computations performed with short or long floating-point formats are remarkably accurate in most cases. Errors resulting from the finiteness of representation and imprecise calculations (e.g., approximation or convergence schemes) are by now reasonably well understood and can be kept under control by algorithmic methods. In some situations, however, ordinary floating-point arithmetic is inadequate, either because it is not precise enough or because of our inability to establish useful bounds on the errors. In such cases, the results may well possess adequate precision but there is a "credibility-gap problem . . . [as] we don't know how much of the computer's answers to believe" [Knut81].

We will discuss three distinct approaches for coping with the aforementioned credibility gap:

1. Obtaining completely trustworthy results by performing arithmetic calculations exactly (Section 20.2). Of course, if this approach were always possible and cost-effective, we wouldn't need any of the following alternatives.
2. Making the arithmetic highly precise, in order to raise our confidence in the validity of the results. This pragmatic goal can be accomplished by multiprecision calculations (Section

20.3) or via a more flexible variable-precision arithmetic system (Section 20.4). The two approaches correspond to static and dynamic precision enhancement, respectively. Both methods make irrelevant results less likely but provide no guarantee, except in a probabilistic sense.

3. Performing ordinary or high-precision calculations, while keeping track of potential error accumulation (Section 20.5). Then, based on the worst-case suspected error in the result, we can either certify the result as carrying adequate precision or produce a warning that would prevent incorrect conclusions or actions that might have catastrophic consequences (fail-safe operation).

After studying the preceding approaches, we devote Section 20.6 to techniques that render precise and/or certifiable arithmetic more efficient.

Besides problems with precision, the finite range of machine arithmetic can also become problematic. Thus provisions for exact or highly precise arithmetic are often accompanied by methods for extending the range. A common way is via number representation systems in which the range can grow dynamically. Usually, numbers are represented in a single word. However, one or two bits are assigned special meanings and allow the number to extend into subsequent words. The price we pay for this flexibility is loss of the aforementioned bit(s) and more complex arithmetic algorithms, including the overhead of the special checks needed to establish whether the range must be extended.

Of course certifiability in computer arithmetic is concerned not only with precision, or lack thereof, but also spans algorithm and hardware verification as well as fault detection and tolerance. Modern digital systems tend to be extremely complex. Thus, unless full attention is paid to correctness issues during the design, there is little hope of catching all problems afterward. The already difficult verification process is exacerbated by complex interrelationships between advanced design features such as parallelism, pipelining, and power-saving mechanisms. The Pentium floating-point division flaw aptly illustrates this point. As for fault-induced errors, we deal with them in Chapter 27.

20.2 EXACT ARITHMETIC

The ultimate in error control is exact (error-free) arithmetic. This ideal has been pursued by many arithmetic designers and researchers, leading to proposals for using continued fractions, rational numbers, and p-adic representations, among others. In this section, we introduce a few of the proposed methods and briefly discuss their implementation aspects, advantages, and drawbacks.

a. Continued Fractions

Any unsigned rational number $x = p/q$ has a unique continued-fraction expansion

$$x = \frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \cfrac{1}{a_{m-1} + \cfrac{1}{a_m}}}}}$$

with $a_0 \geq 0$, $a_m \geq 2$, and $a_i \geq 1$ for $1 \leq i \leq m-1$. For example, 277/642 has the following continued-fraction representation:

$$\frac{277}{642} = 0 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{3 + 1/3}}}}} = [0/2/3/6/1/3/3]$$

Representation of $-277/642$ is obtained by simply attaching a sign bit or negating all the digits in the representation of 277/642.

Note that the continued-fraction representation of x is obtained by writing $x = s^{(0)}$ as $\lfloor s^{(0)} \rfloor + 1/s^{(1)}$, and then repeating the process for representing each $s^{(i)}$ in turn (i.e., $s^{(1)} = \lfloor s^{(1)} \rfloor + 1/s^{(2)}, \ldots$). Thus, for $s^{(0)} = 277/642$, we get $s^{(1)} = 642/277$, $s^{(2)} = 277/88$, $s^{(3)} = 88/13$, $s^{(4)} = 13/10$, $s^{(5)} = 10/3$, and $s^{(6)} = 3$.

Approximations for finite representation can be obtained by limiting the number of "digits" in the continued-fraction representation. For example, the following are successively better approximations to the exact value $x = [0/2/3/6/1/3/3] = 277/642$:

[0]	= 0
[0/2]	= 1/2
[0/2/3]	= 3/7
[0/2/3/6]	= 19/44
[0/2/3/6/1]	= 22/51
[0/2/3/6/1/3]	= 85/197

Vuillemin [Vuil90] has suggested that continued fractions be used in the following way for performing exact arithmetic. Each potentially infinite, continued fraction is represented by a finite number of digits, plus a *continuation*, which is, in effect, a procedure for obtaining the next digit as well as a new continuation. Notationally, we can write the digits as before (i.e., separated by /), following them with a semicolon and a description of the continuation.

When the representation is periodic, the continuation can simply be specified by a sequence of one or more digits. This is what we do in decimal arithmetic when we write 8/3 as $(2.66;6)_{\text{ten}}$ and 1/7 as $(0.1;428571)_{\text{ten}}$. When additional digits can be derived as a simple function of an index $i \geq 0$, the relevant expression is given. Here are some examples:

$(1+\sqrt{5})/2$	$= [1/1/1/1/\cdots] = [;\, 1]$
$\sqrt{2}$	$= [1/2/2/2/\cdots] = [1;\, 2]$
e	$= [2/1/2/1/1/4/1/1/6/1/\cdots] = [2;\, 1/2i + 2/1]$
∞	$= [1/0/1/0/1/0\cdots] = [;\, 1/0] = [;\, 2/0] = \cdots$
aN	$= [0/0/0/0/\cdots] = [;\, 0]$ {any number}

Unfortunately, arithmetic operations on continued fractions are quite complicated. So, we will not pursue this representation further.

b. Fixed-Slash Number Systems

In a fixed-slash number system, a rational number is represented as the ratio of a pair of integers p and q, each with a fixed range. Representation of numbers as finite-precision rationals is related to the continued-fraction expansion discussed earlier in the sense that when a number is not exactly representable, the best continued-fraction approximation that fits is used as its "rounded" version. For example, suppose we want to represent the rational number 277/642 in a $2+2$ decimal fixed-slash number system (2 digits each for the numerator and the denominator). From the continued-fraction representation given earlier, we find the best approximation to be 22/51, which has a relative error slightly exceeding 2%.

A possible fixed-slash format for representing rational numbers consists of a sign bit, followed by an "inexact" flag, a k-bit numerator, and an m-bit denominator, for a total of $k+m+2$ bits (Fig. 20.1). The inexact flag is useful for denoting a value that has been rounded off because the precise result did not fit within the available format. Note that integers are a subclass of representable numbers (with $q = 1$). The representation of a rational number is normalized if $\gcd(p, q) = 1$. Special values can also be represented by appropriate conventions. Here is one way to do it:

Rational number	if $p > 0, q > 0$
± 0	if $p = 0, q$ odd
$\pm\infty$	if p odd, $q = 0$
NaN (not a number)	otherwise

When a number is not representable exactly, it is rounded to the closest representable value. On overflow (underflow), the number is rounded to $\pm\infty(\pm 0)$ and the inexact bit is set.

The following mathematical result, due to Dirichlet, shows that the space waste due to multiple representations such as $3/5 = 6/10 = 9/15 = \cdots$ is no more than one bit:

$$\lim_{n\to\infty} \frac{|\{p/q | 1 \le p, q \le n, \gcd(p, q) = 1\}|}{n^2} = \frac{6}{\pi^2} \approx 0.608$$

This result essentially says that for n sufficiently large, two randomly selected numbers in $[1, n]$ are relatively prime with probability greater than 0.6. Thus, more than half of the codes represent unique numbers and the waste is less than 1 bit.

Note that the additive (multiplicative) inverse of a number is easily obtained with fixed-slash representation by simply flipping the sign bit (switching p and q). Adding two fixed-slash numbers requires three integer multiplications and one addition, while multiplying them involves two multiplications. Subtraction (division) can be done as addition (multiplication) by first forming the additive (multiplicative) inverse of the subtrahend (divisor).

The results of these operations are exact, unless the numerator or denominator becomes too large. In such a case, we can avoid overflow through *normalization* if p and q have a common factor. The overhead implied by computing $\gcd(p, q)$ is often unacceptably high. Additionally, once the capacity of the number system for exact representation of the result has been exceeded,

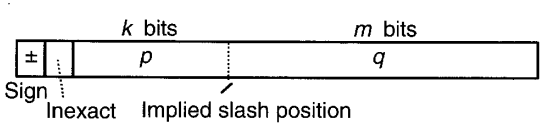


Fig. 20.1 Example fixed-slash number representation format.

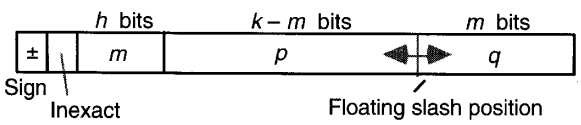


Fig. 20.2 Example floating-slash representation format.

the process of rounding the result to the nearest representable rational number is fairly complex. For these reasons, fixed-slash representations have not found widespread use.

c. Floating-Slash Number Systems

In a fixed-slash number system, a fixed number of bits is allocated to each of the numerator and denominator parts. These bits sometimes go to waste, as evident in the case of $q = 1$ for representing integers. A floating-slash format for representing rational numbers consists of a sign bit, followed by an "inexact" flag, an h-bit field (m) specifying the explicit slash position, and a k-bit field containing a $(k - m)$-bit numerator and the least significant m bits of an $(m + 1)$-bit denominator with a hidden MSB of 1. We obtain integers for $m = 0$. The set of numbers represented in such a floating-slash number system (Fig. 20.2) is:

$$\{\pm p/q | p, q \geq 1, \gcd(p, q) = 1, \lfloor \log_2 p \rfloor + \lfloor \log_2 q \rfloor \leq k - 2\}$$

Special codes for ± 0, $\pm\infty$, and NaN are also needed, as in fixed-slash representations. For the sake of simplicity, one can replace the preceding condition $\lfloor \log_2 p \rfloor + \lfloor \log_2 q \rfloor \leq k - 2$ with the approximate condition $pq \leq 2^k$. Again the following mathematical result, due to Dirichlet, shows that the space waste is no more than one bit:

$$\lim_{n \to \infty} \frac{|\{\pm p/q | pq \leq n, \gcd(p, q) = 1\}|}{|\{\pm p/q |\ pq \leq n, p, q \geq 1\}|} = \frac{6}{\pi^2} \approx 0.608$$

Floating-slash format removes some of the problems of fixed-slash representations, but arithmetic operations are complicated even further; hence, applications are limited.

20.3 MULTIPRECISION ARITHMETIC

One could in principle build a highly precise arithmetic unit, say operating on 1024-bit floating-point numbers instead of the standard 32- or 64-bit varieties. There are several obvious problems with this approach, including high cost, waste of time and hardware for computations that do not need such a high precision, and inability to adapt to special situations that call for even higher precision. Thus, floating-point hardware is provided for more commonly used 32- and 64-bit numbers.

When the range or precision of the number representation scheme supported by the hardware is inadequate for a given application, we are forced to represent numbers as multiword data structures and to perform arithmetic operations by means of software routines that manipulate these structures. Examples in the case of integer arithmetic can be found in cryptography, where large integers are used as keys for the encoding/decoding processes, and in mathematical research, where properties of large primes are investigated. Extended-precision floating-point numbers may be encountered in some scientific calculations, where highly precise results are

required, or in error analysis efforts, where the numerical stability of algorithms must be verified by computing certain test cases with much higher precision.

Multiprecision arithmetic refers to the representation of numbers in multiple machine words. The number of words used to represent each integer or real number is chosen a priori; if the number of words can change dynamically, we have variable-precision arithmetic (see Section 20.4). In the case of integer values, the use of multiple words per number extends the range; for floating-point numbers, either the range or the precision parameter or both might be extended, depending on need. All these approaches are referred to as "multiprecision arithmetic," even though, strictly speaking, the term makes no sense for integers.

Multiprecision integer arithmetic is conceptually quite simple. An integer can be represented by a list of smaller integers, each of which fits within a single machine word (Fig. 20.3). These extended-precision integers are then viewed as radix-2^k numbers, where k is the word width. As an example, with 32-bit machine words, one can represent a quadruple-precision 2's-complement integer x by using the four unsigned words $x^{(3)}, x^{(2)}, x^{(1)}, x^{(0)}$, such that:

$$x = -x_{31}^{(3)}\, 2^{127} + 2^{96} \sum_{j=0}^{30} x_j^{(3)}\, 2^j + 2^{64} x^{(2)} + 2^{32} x^{(1)} + x^{(0)}$$

The radix in this example is 2^{32}. With this representation, radix-2^k digit-serial arithmetic algorithms can be applied to the multiprecision numbers in a straightforward manner to simulate 128-bit, 2's-complement arithmetic. To perform the addition $z = x + y$, for example, we begin by performing $z^{(0)} = x^{(0)} + y^{(0)}$, which leads to the carry-out $c^{(1)}$ being saved in the carry flag. Next, we perform the addition $z^{(1)} = x^{(1)} + y^{(1)} + c^{(1)}$. Virtually all processors provide a special instruction for adding with carry-in. The process can thus be repeated in a loop, with special overflow detection rules applied after the last iteration.

Multiplication can be performed by either implementing a shift/add algorithm directly or by using the machine's multiply instruction, if available. For further details, see [Knut81, Section 4.3, on multiple-precision arithmetic, pp. 250–301].

Performing complicated arithmetic computations on multiprecision numbers can be quite slow. For this reason, people sometimes prefer to perform such computations on highly parallel computers, thus speeding up the computation by concurrent operations on various words of the multiword numbers. Since each word of the resulting multiword numbers in general depends on all words of the operands, proper data distribution and occasional rearrangement may be required to minimize the communication overhead that otherwise might nullify much of the speed gain due to concurrency. Many standard parallel algorithms can be used directly in such arithmetic computations. For example, parallel prefix can be used for carry prediction (lookahead) and FFT for multiplication [Parh98]. Whether one uses a sequential or parallel computer for multiprecision arithmetic, the selection of the optimal algorithm depends strongly on the available hardware features and the width of numbers to be processed [Zura93].

Multiprecision floating-point arithmetic can be similarly programmed. When precision is to be extended but a wider range is not needed, a standard floating-point number can be used

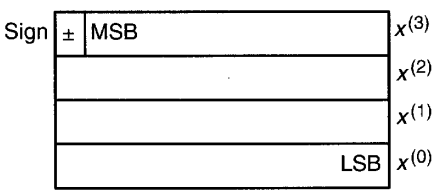


Fig. 20.3 Example quadruple-precision integer format.

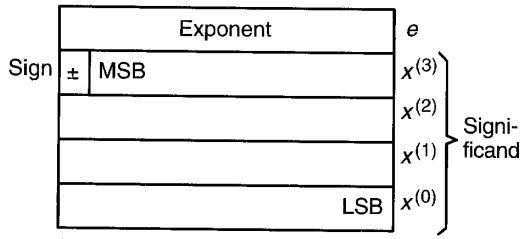


Fig. 20.4 Example quadruple-precision floating-point format.

to represent the sign, exponent, and part of the significand, with the remaining bits of the high-precision significand extending into one or more additional words. However, given that modern computers have plenty of register and storage space available, it is perhaps better to use a separate word for storing the exponent and one or more words for the extended-precision significand, thus eliminating the overhead for repeated packing and unpacking. The significand can be represented as an integer using the format of Fig. 20.3. The separate exponent, which is a 32-bit biased number, say, provides a very wide range that is adequate for all practical purposes. Figure 20.4 depicts the resulting format.

Arithmetic operations are performed by programming the required steps for the floating-point algorithms of Section 17.3, with details in Chapter 18. To perform addition, for example, the significand of the operand with the smaller exponent is shifted to the right by an amount equal to the difference of the two exponents, the aligned significands are added, and the resulting sum is normalized (Fig. 20.5). Floating-point multiplication and division are similarly performed.

As for rounding of the results, two approaches are possible. One is to simply chop any bit that is shifted out past the right end of the numbers, hoping that the extended precision will be adequate to compensate for any extra error. An alternative is to derive guard, round, and sticky bits from the bits that are shifted out (see Fig. 20.5) in the manner outlined in Section 18.3.

20.4 VARIABLE-PRECISION ARITHMETIC

As mentioned in Section 20.3, multiprecision arithmetic suffers both from inefficiency in the common case (i.e., when high precision is not needed) and from the inability to adapt to situations that might require even higher precision. Alternatively, a variable-precision floating-point capability can be implemented to operate on data of various widths under program control. Variable precision is useful not only for situations calling for high precision; it may be beneficial, as well, for improving performance when lower precision would do.

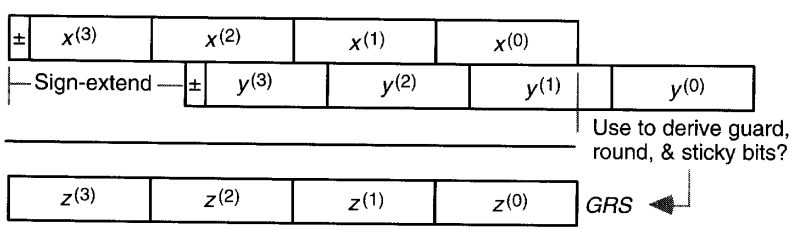


Fig. 20.5 Quadruple-precision significands aligned for the floating-point addition $z = x +_{\text{fp}} y$.

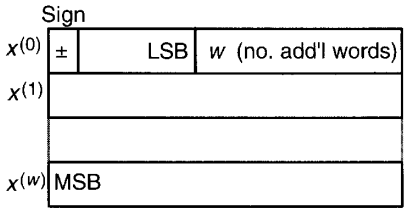


Fig. 20.6 Example variable-precision integer format.

Dispensing precision on demand in different stages of computations, or even at the level of individual arithmetic operations, has been an elusive goal in the field of computer arithmetic, except where bit- or digit-serial arithmetic is involved. For our discussion here, we consider variable precision with machine-word granularity. This is quite similar to multiprecision arithmetic, as discussed in Section 20.3, except that a "width" field must be added to all numbers that specifies how many words are used to represent the number. Also, if the operand widths are to be modifiable at run time, dynamic storage allocation and facilities for reclaiming space (garbage collection) are required.

To represent variable-precision (really variable-range) integers, we might use 1 or 2 bytes in the first 32-bit word to hold the width information, 1 bit for the sign, and the remaining part to hold the low-order 15 or 23 bits of the number. If the number is wider, additional words will be tacked on as needed to hold the higher-order bits (Fig. 20.6). Note that this convention, known as "little-endian," is opposite that of Fig. 20.4, which is referred to as "big-endian." Storing the low-order bits first leads to a slight simplification in variable-precision addition, since indexing for both operands and the result starts at 0.

Again to avoid packing and unpacking of values and to remove the need for special handling of the first chunk of the number, one might assign the number's width information to an entire word, which can then be directly loaded into a counter or register for processing.

A corresponding variable-precision floating-point format can be similarly devised. Figure 20.7 depicts one alternative. Here, the first word contains the number's sign, its width w, the exponent e, and designations for special operands. The significand then follows in w subsequent words. Again, we might want to put the exponent in a separate word, both to reduce the need for packing and unpacking and to provide greatly extended range.

From an implementation standpoint, addition becomes much simpler if the exponent base is taken to be 2^k instead of 2, since the former case would lead to shift amounts that are multiples of k bits (bit-level operations are avoided). This will, of course, have implications in terms of the available precision (see Section 17.1). The effect of shifting can then be taken into account by indexing rather than actual data movement. For example, if the alignment shift amount applied to the v-word operand y before adding it the u-word operand x to obtain the u-word sum z is h words, then referring to Fig. 20.8 and defining $g = v + h - u$, we can write the main part of the floating-point addition algorithm as the following three loops:

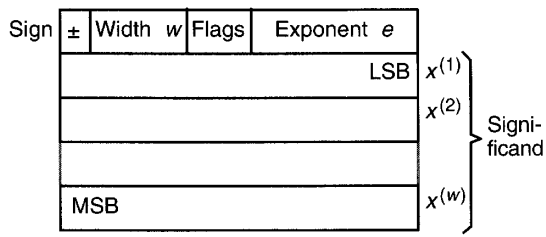


Fig. 20.7 Example variable-precision floating-point format.

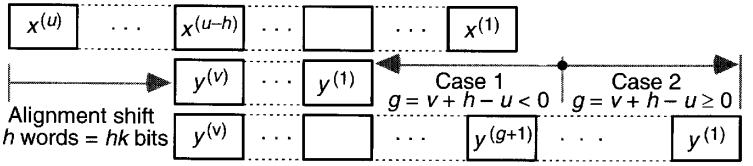


Fig. 20.8 Variable-precision floating-point addition.

```
for i = 1 to −g do                                  {empty loop if g ≥ 0}
  c, z^(i) ← x^(i) + c
endfor
for i = max(1, −g + 1) to u − h do                  {empty loop if u ≤ h}
  c, z^(i) ← x^(i) + y^(g+i) + c
endfor
for i = max(1, u − h + 1) to u do                   {empty loop if h = 0}
  c, z^(i) ← x^(i) + c − signbit(y)                 {must sign-extend y}
endfor
```

In the complete algorithm, the loops must be preceded by various checks and initializations and followed by any normalization and rounding required.

20.5 ERROR BOUNDING VIA INTERVAL ARITHMETIC

Interval arithmetic was introduced at the end of Section 19.5 as an error analysis method. When computation with intervals yields a result $z = [z_{lo}, z_{hi}]$, the width of the interval $w = z_{hi} - z_{lo} \geq 0$ can be interpreted as the extent of uncertainty, and the midpoint $(z_{lo} + z_{hi})/2$ of the interval can be used as an approximate value for z with a worst-case error of about $w/2$. Even when a result interval is too wide to be practically useful, at least a fail-safe mode of operation can be ascertained.

The interval $[a, a]$ represents the real number a, while $[a, b]$, with $a > b$, can be viewed as representing the empty interval ϕ. Intervals can be combined and compared in a natural way. For example:

$$[x_{lo}, x_{hi}] \cap [y_{lo}, y_{hi}] = \big[max(x_{lo}, y_{lo}), min(x_{hi}, y_{hi})\big]$$

$$[x_{lo}, x_{hi}] \cup [y_{lo}, y_{hi}] = \big[min(x_{lo}, y_{lo}), max(x_{hi}, y_{hi})\big]$$

$$[x_{lo}, x_{hi}] \supseteq [y_{lo}, y_{hi}] \text{ iff } x_{lo} \leq y_{lo} \text{ and } x_{hi} \geq y_{hi}$$

$$[x_{lo}, x_{hi}] = [y_{lo}, y_{hi}] \text{ iff } x_{lo} = y_{lo} \text{ and } x_{hi} = y_{hi}$$

$$[x_{lo}, x_{hi}] < [y_{lo}, y_{hi}] \text{ iff } x_{hi} < y_{lo}$$

Interval arithmetic operations are quite intuitive and efficient. For example, the additive inverse $-x$ of an interval $x = [x_{lo}, x_{hi}]$ is derived as follows:

$$-[x_{lo}, x_{hi}] = [-x_{hi}, -x_{lo}]$$

The multiplicative inverse of an interval $x = [x_{lo}, x_{hi}]$ is derived as:

$$\frac{1}{[x_{lo}, x_{hi}]} = \left[\frac{1}{x_{hi}}, \frac{1}{x_{lo}}\right] \quad \text{provided } 0 \notin [x_{lo}, x_{hi}]$$

When $0 \in [x_{lo}, x_{hi}]$—that is, when x_{lo} and x_{hi} have unlike signs or are both 0s—the multiplicative inverse is undefined (alternatively, it can be said to be $[-\infty, +\infty]$). Note that with machine arithmetic, $1/x_{hi}$ must be computed with downward-directed rounding and $1/x_{lo}$ with upward-directed rounding.

In what follows, we assume that proper rounding is performed in each case and deal only with exact intervals for simplicity. Here are the four basic arithmetic operations on intervals:

$$\begin{aligned}
[x_{lo}, x_{hi}] + [y_{lo}, y_{hi}] &= [x_{lo} + y_{lo}, x_{hi} + y_{hi}] \\
[x_{lo}, x_{hi}] - [y_{lo}, y_{hi}] &= [x_{lo} - y_{hi}, x_{hi} - y_{lo}] \\
[x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}] &= [min(x_{lo}y_{lo}, x_{lo}y_{hi}, x_{hi}y_{lo}, x_{hi}y_{hi}), \\
&\qquad max(x_{lo}y_{lo}, x_{lo}y_{hi}, x_{hi}y_{lo}, x_{hi}y_{hi})] \\
[x_{lo}, x_{hi}] / [y_{lo}, y_{hi}] &= [x_{lo}, x_{hi}] \times [1/y_{hi}, 1/y_{lo}]
\end{aligned}$$

Several interesting properties of intervals and interval arithmetic are explored in the end-of-chapter problems. In particular, we will see that multiplication is not as inefficient as the preceding definition might suggest.

From the viewpoint of arithmetic calculations, a very important property of interval arithmetic is stated in the following theorem.

THEOREM 20.1 If $f(x^{(1)}, x^{(2)}, \cdots, x^{(n)})$ is a rational expression in the interval variables $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$, that is, f is a finite combination of $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$ and a finite number of constant intervals by means of interval arithmetic operations, then $x^{(i)} \supset y^{(i)}, i = 1, 2, \cdots, n$, implies:

$$f(x^{(1)}, x^{(2)}, \cdots, x^{(n)}) \supset f^{(1)}, y^{(2)}, \cdots, y^{(n)})$$

Thus, arbitrarily narrow result intervals can be obtained by simply performing arithmetic with sufficiently high precision. In particular, we can show that with reasonable assumptions about machine arithmetic, the following theorem holds.

THEOREM 20.2 Consider the execution of an algorithm on real numbers by means of machine interval arithmetic with precision p in radix r [i.e., in FLP(r, p, $\nabla|\Delta$)]. If the same algorithm is executed using the precision q, with $q > p$, the bounds for both absolute error and relative error are reduced by the factor r^{q-p}.

Note that the absolute or relative error itself may not be reduced by the same factor; the guaranteed reduction applies only to the upper bound.

Based on Theorem 20.2, one can devise a practical strategy for obtaining results with a desired bound on the absolute or relative error. For example, let w_{max} be the maximum width of a result interval when interval arithmetic is performed with p radix-r digits of precision and assume that the required bound on the absolute error is ε. If $w_{max} \leq \varepsilon$, then we are done. Otherwise, interval calculations with the higher precision

$$q = p + \lceil \log_r w_{max} - \log_r \varepsilon \rceil$$

is guaranteed to yield the desired accuracy.

20.6 ADAPTIVE AND LAZY ARITHMETIC

In some applications, arithmetic algorithms and/or hardware structures must adapt to changing conditions or requirements. For example, not all computations require the same precision, and using a 64-bit multiplier to multiply 8-bit numbers would be a waste of hardware resources, and perhaps even time. In this section, we briefly discuss some ideas for building adaptable arithmetic systems. An aspect of adaptability is fault tolerance, namely, the capacity for continued operation, perhaps at lower performance, acquired by reconfiguring around faulty elements. This latter type of adaptability is the subject of Chapter 27.

One way to provide adaptivity is via built-in multiprecision arithmetic capability. For example, facilities may be provided to allow the dynamic switching of a computation from single- to multiprecision according to the precision requirements for the results. Variable-precision capability can extend the preceding two-way adaptive scheme to an incremental or multiway scheme.

Interestingly, the opposite of multiprecision arithmetic, which we may call fractional precision arithmetic, is also of some interest. Whereas modern high-performance microprocessors have arithmetic capability for 32- or 64-bit numbers, many arithmetic-intensive applications, such as voice compression or image processing for multimedia, may deal with 8- or 16-bit data representing color or other audiovisual elements. Recent microprocessor designs have recognized the need for efficient handling of such fractional precision numbers through special hardware extensions. For example, Intel's MMX (multimedia extension) for the Pentium processor [Pele97] uses the microprocessor's eight floating-point registers to store 64-bit packed integer data (8×8, 4×16, 2×32, in signed/unsigned versions). Special add, multiply, multiply-add, and parallel compare instructions are made available that operate on these packed MMX data types.

An alternative approach to adaptive arithmetic is via multiple number representation formats that are distinguished by tagging. For example, in a simple two-way adaptive scheme, primary and secondary representation modes may be associated with each number type; the primary mode is more precise but offers limited range, while the secondary mode offers a wider range with less precision. Computation is then switched between the two representations based on need. In this way, overflow can be avoided or postponed. One proposal along these lines [Holm97] uses four-way tagging to distinguish between primary and secondary formats for exact and inexact values.

Lazy evaluation is a powerful paradigm that has been and is being used in many different contexts. For example, in evaluating composite conditionals such as

if *cond1* and *cond2* then *action*

the evaluation of *cond2* may be totally skipped if *cond1* evaluates to "false". More generally, lazy evaluation means postponing all computations or actions until they become irrelevant or unavoidable. In the context of computer hardware architecture, the opposite of lazy evaluation (viz., speculative or aggressive execution) has been applied extensively; however, lazy evaluation is found only in certain special-purpose systems with data- or demand-driven designs.

In the absence of hardware support for lazy arithmetic, all known implementations of this method rely on software. Schwarz [Schw89] describes a C++ library for arbitrary precision arithmetic that is based on representing results by a data value corresponding to the known bits and an expression that can be manipulated to obtain more bits when needed. A lazy rational arithmetic system [Mich97] uses a triple $\langle x_{lo}, x_{xct}, x_{hi} \rangle$ to represent each number, where x_{xct} is an exact rational value, or a pointer to a procedure for obtaining it, and $[x_{lo}, x_{hi}]$ represents an interval bounded by the floating-point values x_{lo} and x_{hi}. Computation normally proceeds with floating-point values using the rules of interval arithmetic. When this primary mode of computation runs into precision problems, and only then, exact computation is invoked.

Lazy arithmetic, as suggested above, comes with nontrivial representational and computational overheads. Thus far, the viability of lazy arithmetic, and its cost–performance implications, have been investigated only for certain geometric computations. Even within this limited application domain, some problems remain to be resolved [Mich97].

It is noteworthy that redundant number representations offer some advantages for lazy arithmetic. Since arithmetic on redundant numbers can be performed by means of MSD-first algorithms, it is possible to produce a small number of digits of the result by using correspondingly less computational effort. When precision problems are encountered, one can backtrack and obtain more digits of the results as needed.

PROBLEMS

20.1 Computing the *i*th Fibonacci number The sequence of Fibonacci numbers Fib(i), $i = 1, 2, 3, \cdots$, is defined recursively as Fib(1) = Fib(2) = 1 and Fib(i) = Fib($i - 1$) + Fib($i-2$) for $i \geq 3$. One can show that Fib(i) $= (x^i - y^i)/\sqrt{5}$, where $x = (1+\sqrt{5})/2$ and $y = (1 - \sqrt{5})/2$.

a. Devise an exact representation for numbers of the form $a + b\sqrt{5}$, where a and b are rational numbers.

b. Develop algorithms for addition, subtraction, multiplication, division, and exponentiation for the numbers in part a.

c. Use your representation and arithmetic algorithms to compute Fib(10) and Fib(64).

20.2 Converging interval representation The golden ratio $\phi = (1 + \sqrt{5})/2$ can be represented increasingly accurately by a sequence of intervals $x^{(j)} = [\text{Fib}(2j+2)/\text{Fib}(2j+1), \text{Fib}(2j+1)/\text{Fib}(2j)]$ that get narrower as j increases. In the preceding description, Fib(i) is the ith Fibonacci number recursively defined as Fib(1) = Fib(2) = 1 and Fib(i) = Fib($i - 1$) + Fib($i - 2$) for $i \geq 3$.

a. Using exact rational arithmetic, obtain the first eight intervals in the sequence defined.

b. Repeat part a, this time using decimal arithmetic with six fractional digits. From the last result, find an approximation to ϕ with an associated error bound.

20.3 Approximating π with exact arithmetic Using exact rational arithmetic, find an interval that is guaranteed to contain the exact value of π based on the identity

$\pi/4 = \tan^{-1}(1/2) + \tan^{-1}(1/5) + \tan^{-1}(1/8)$ and the inequalities $x - x^3/3 + x^5/5 - x^7/7 < \tan^{-1} x < x - x^3/3 + x^5/5$.

20.4 Fixed-slash number systems

a. Discuss the factors that might affect the choice of the widths k and m in the fixed-slash format of Fig. 20.1. In what respects is $k = m$ a good choice?

b. Compute the number of different values that can be represented in a 15-bit signed, fixed-slash number system with 7-bit numerator and denominator parts, plus a sign bit (no inexact bit), and discuss its representation efficiency relative to a 15-bit, signed-magnitude, fixed-point binary system.

20.5 Floating-slash number systems For the floating-slash number system shown in Fig. 20.2:

a. Obtain the parameters *max* and *min* (i.e., the largest representable magnitude and the smallest nonzero magnitude) as functions of h and k.

b. Calculate the maximum relative representation error for numbers in [*min*, *max*].

c. Obtain a lower bound on the total number of different values that can be represented as a function of h and k.

20.6 Continued-fraction number representation In continued-fraction number representation, it is possible to use rounding, instead of the floor function, namely, $a_i = \text{round}(s^{(i)})$ rather than $a_i = \lfloor s^{(i)} \rfloor$, to obtain more accurate encodings with a given number of digits. Obtain 10-digit continued-fraction representations of $\sqrt{2}$, e, and π with the "rounding" rule and compare the results to the "floor" versions with respect to accuracy.

20.7 Exact representation of certain rationals Consider rational numbers of the form $\pm 2^a 3^b 5^c$, represented in 16 bits by devoting 1 bit to the sign and 5 bits each to the 2's-complement representation of a, b, and c.

a. Obtain the parameters *max* and *min* (i.e., the largest representable magnitude and the smallest nonzero magnitude).

b. Calculate the maximum relative representation error for numbers in [*min*, *max*].

c. Find the number of different values represented and the representational efficiency of this number system.

d. Briefly discuss the feasibility of exact arithmetic operations on such numbers.

20.8 Multiprecision arithmetic

a. Provide the structure of an assembly-language program (similar to Fig. 9.3) to perform quadruple-precision integer arithmetic based on the format of Fig. 20.3

b. Repeat part a for floating-point arithmetic based on the format of Fig. 20.4.

20.9 Variable-precision arithmetic

a. Show that the three "for" loops in the program fragment given near the end of Section 20.4 do indeed process all the words of x and y properly.

b. Justify the inclusion of the term $-\text{signbit}(y)$ to effect sign extension for y.

c. Modify the three loops for the case of a sum z that is to be of a specified width w, rather than of the same width u as the operand with the larger exponent.

20.10 Interval arithmetic Answer the following questions for interval arithmetic.

a. Would interval arithmetic be of any use if machine arithmetic were exact? Discuss.

b. How is the requirement $q = p + \lceil \log_r w_{max} - \log_r \varepsilon \rceil$ for extra bits of precision, given near the end of Section 20.5, derived from Theorem 20.2?

20.11 Archimedes' interval method To compute the number π, Archimedes used a sequence of increasing lower bounds, derived from the perimeters of inscribed polygons in a circle with unit diameter, and a sequence of decreasing upper bounds, based on circumscribing polygons.

a. Use the method of Archimedes, with a pair of hexagons and exact calculations, to derive an interval that is guaranteed to contain π.

b. Repeat part a, this time performing the arithmetic with four fractional decimal digits and proper rounding.

c. Repeat part a with a pair of octagons.

d. Repeat part b with a pair of octagons.

20.12 Distance between intervals The distance between two intervals $x = [x_{lo}, x_{hi}]$ and $y = [y_{lo}, y_{hi}]$ can be defined as $\delta(x, y) = max(|x_{lo} - y_{lo}|, |x_{hi} - y_{hi}|)$.

a. Show that δ is a metric in that it satisfies the three conditions $\delta(x, y) \geq 0$, $\delta(x, y) = 0$ if and only if $x = y$, and $\delta(x, y) + \delta(y, z) \geq \delta(x, z)$ (the triangle inequality).

b. Defining the absolute value $|x|$ of an interval x as $|[x_{lo}, x_{hi}]| = max(|x_{lo}|, |x_{hi}|)$, prove that $\delta[(x + y), (x + z)] = \delta(y, z)$ and $\delta(xy, xz) \leq |x|\delta(y, z)$.

20.13 Laws of algebra for intervals

a. Show that the commutative laws of addition and multiplication hold for interval arithmetic; namely, $x + y = y + x$ and $xy = yx$ for intervals x and y.

b. Show that the associative laws of addition and multiplication hold for interval arithmetic; namely, $x + (y + z) = (x + y) + z$ and $x(yz) = (xy)z$.

c. Show that the distributive law $x(y + z) = xy + xz$ does not always hold.

d. Show that subdistributivity holds; namely, $x(y + z)$ is contained in $xy + xz$.

20.14 Interval arithmetic operations

a. Show that by testing the signs of x_{lo}, x_{hi}, y_{lo}, and y_{hi}, the formula for interval multiplication given in Section 20.5 can be broken down into nine cases, only one of which requires more than two multiplications.

b. Discuss the square-rooting operation for intervals.

20.15 Multidimensional intervals A rectangle with sides parallel to the coordinate axes on the two-dimensional plane can be viewed as a two-dimensional interval. Relate two-dimensional intervals to arithmetic on complex numbers and derive the rules for complex interval arithmetic.

20.16 Lazy arithmetic with intervals Consider a lazy arithmetic system with interval arithmetic and exact rational arithmetic as its primary and secondary (fallback) computation modes, respectively. Define rules for comparing numbers in the primary mode such that each comparison has three possible outcomes: "true," "false," and "unknown" (with the last outcome triggering exact computation to remove the ambiguity).

20.17 Fixed-point iteration A *fixed point* of the function $f(x)$ is a value x_{fxpt} such that $x_{\text{fxpt}} = f(x_{\text{fxpt}})$. Geometrically, the fixed point x_{fxpt} corresponds to an intersection of the curve $y = f(x)$ with the line $y = x$. A fixed point of $f(x)$ can sometimes be obtained using the iterative formula $x^{(i+1)} = f(x^{(i)})$, with a suitably chosen initial value $x^{(0)}$.

a. The function $f(x) = 1 + x - x^2/a$ has two fixed points at $x = \pm\sqrt{a}$. Assuming $a = 2$ and $x^{(0)} = 3/2$, use exact rational arithmetic to find $x^{(4)}$.

b. Repeat part a using a calculator.

c. Repeat part a using interval arithmetic; round calculations to six fractional digits.

d. Compare the results of parts a, b, and c. Discuss.

REFERENCES

[Alef83] Alefeld, G., and J. Herzberger, *An Introduction to Interval Computations*, Academic Press, 1983.

[Greg81] Gregory, R.T., "Error-Free Computation with Rational Numbers," *BIT*, Vol. 21, pp. 194–202, 1981.

[Holm97] Holmes, W.N., "Composite Arithmetic: Proposal for a New Standard," *IEEE Computer*, Vol. 30, No. 3, pp. 65–73, 1997.

[Knut81] Knuth, D.E., *The Art of Computer Programming*, 2nd ed., *Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

[Matu85] Matula, D.W., and P. Kornerup, "Finite Precision Rational Arithmetic: Slash Number Systems," *IEEE Trans. Computers*, Vol. 34, No. 1, pp. 3–18, 1985.

[Mich97] Michelucci, D., and J.-M. Moreau, "Lazy Arithmetic," *IEEE Trans. Computers*, Vol. 46, No. 9, pp. 961–975, 1997.

[Moor66] Moore, R., *Interval Analysis*, Prentice-Hall, 1966.

[Parh98] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Press, 1999.

[Pele97] Peleg, A., S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs," *Commun. ACM*, Vol. 40, No. 1, pp. 25–38, 1997.

[Schw89] Schwarz, J., "Implementing Infinite Precision Arithmetic," *Proc. 9th Symp. Computer Arithmetic*, 1989, pp. 10–17.

[Vuil90] Vuillemin, J., "Exact Real Computer Arithmetic with Continued Fractions," *IEEE Trans. Computers*, Vol. 39, No. 8, pp. 1087–1105, 1990.

[Zura93] Zuras, D., "On Squaring and Multiplying Large Integers," *Proc. 11th Symp. Computer Arithmetic*, June 1993, pp. 260–271.

PART VI FUNCTION EVALUATION

One way of computing functions such as $\sqrt{x}$, sin x, tanh x, ln x, and e^x is to evaluate their series expansions by means of addition, multiplication, and division operations. Another is through convergence computations of the type used for evaluating the functions z/d and $1/d$ in Chapter 16. In this part, we introduce several methods for evaluating elementary and other functions. We begin by examining the important operation of extracting the square root of a number, covering both digit-recurrence and convergence square-rooting methods. We then devote two chapters to CORDIC algorithms, other convergence methods, approximations, and merged arithmetic. We conclude by discussing versatile, and highly flexible, table-lookup schemes, which are assuming increasingly important roles as advances in VLSI technology lead to ever cheaper and denser memories. This part is composed of the following four chapters:

Chapter 21 SQUARE-ROOTING METHODS

The function $\sqrt{z}$ is the most important elementary function. Since square-rooting is widely used in many applications, and hardware realization of square-rooting has quite a lot in common with division, the IEEE floating-point standard specifies square-rooting as a basic arithmetic operation alongside the usual four basic operations. This chapter is devoted to square-rooting methods, beginning with the pencil-and-paper algorithm and proceeding through shift/subtract, high-radix, and convergence versions. Chapter topics include:

21.1 THE PENCIL-AND-PAPER ALGORITHM

Unlike multiplication and division, for which the pencil-and-paper algorithms are widely taught and used, square-rooting by hand appears to have fallen prey to the five-dollar calculator. Since shift/subtract methods for computing $\sqrt{z}$, are derived directly from the ancient manual algorithm, we begin by describing the pencil-and-paper algorithm for square-rooting.

Our discussion of integer square-rooting algorithms uses the following notation:

z	Radicand	$z_{2k-1}z_{2k-2}\cdots z_1z_0$	
q	Square root	$q_{k-1}q_{k-2}\cdots q_1q_0$	
s	Remainder $(z-q^2)$	$s_ks_{k-1}s_{k-2}\cdots s_1s_0$	$(k+1$ digits$)$

The expression $z - q^2$ for the remainder s is derived from the basic square-rooting equation $z = q^2 + s$. For integer values, the remainder satisfies $s \leq 2q$, leading to the requirement for

$k + 1$ digits in the representation of s with a $2k$-digit radicand z and a k-digit root q. The reason for the requirement $s \leq 2q$ is that for $s \geq 2q + 1$, we have $z = q^2 + s \geq (q + 1)^2$ so q cannot be the correct square-root of z.

Consider the decimal square-rooting example depicted in Fig. 21.1. In this example, the five digits of the decimal number $(9\ 52\ 41)_{\text{ten}}$ are broken into groups of two digits starting at the right end. The number k of groups indicates the number of digits in the square root ($k = 3$ in this example).

The leftmost two-digit group (09) in the example of Fig. 21.1 indicates that the first root digit is 3. We subtract the square of 3 (really, the square of 300) from the 0th partial remainder z to find the 1st partial remainder 52. Next, we double the partial root 3 to get 6 and look for a digit q_1 such that $(6q_1)_{\text{ten}} \times q_1$ does not exceed the current partial remainder 52. Even 1 is too large for q_1, so $q_1 = 0$ is chosen. In the final iteration, we double the partial root 30 to get 60 and look for a digit q_0 such that $(60q_0)_{\text{ten}} \times q_0$ does not exceed the partial remainder 5241. This condition leads to the choice $q_0 = 8$, giving the results $q = (308)_{\text{ten}}$ for the root and $s = (377)_{\text{ten}}$ for the remainder.

The key to understanding the preceding algorithm is the process by which the next root digit is selected. If the partial root thus far is $q^{(i)}$, then attaching the next digit q_{k-i-1} to it will change its value to $10q^{(i)} + q_{k-i-1}$. The square of this latter number is $100(q^{(i)})^2 + 20q^{(i)}q_{k-i-1} + q_{k-i-1}^2$. Since the term $100(q^{(i)})^2 = (10q^{(i)})^2$ has been subtracted from the partial remainder in earlier steps, we need to subtract the last two terms, or $(10(2q^{(i)}) + q_{k-i-1}) \times q_{k-i-1}$, to obtain the new partial remainder. This is the reason for doubling the partial root and looking for a digit q_{k-i-1} to attach to the right end of the result, yielding $10(2q^{(i)}) + q_{k-i-1}$, such that this latter value times q_{k-i-1} does not exceed the partial remainder.

Figure 21.2 shows a binary example for the pencil-and-paper square-rooting algorithm. The root digits are in {0, 1}. In trying to determine the next root digit q_{k-i-1}, we note that the square of $2q^{(i)} + q_{k-i-1}$ is $4(q^{(i)})^2 + 4q^{(i)}q_{k-i-1} + q_{k-i-1}^2$. So, q_{k-i-1} must be selected such that $(4q^{(i)} + q_{k-i-1}) \times q_{k-i-1}$ does not exceed the partial remainder. For $q_{k-i-1} = 1$, this latter expression becomes $4q^{(i)} + 1$ (i.e., $q^{(i)}$ with 01 appended to its right end). Therefore, to determine whether the next root digit should be 1, we need to perform the trial subtraction of $q^{(i)}01$ from the partial remainder; q_{k-i-1} is 1 if the trial subtraction yields a positive result.

From the example in Fig. 21.2, we can abstract the dot notation representation of binary square-rooting (see Fig. 21.3). The radicand z and the root q are shown at the top. Each of the following four rows of dots corresponds to the product of the next root digit q_{k-i-1} and a number

$q_2 : q_1 : q_0$	q		$q^{(0)} = 0$
$\sqrt{9:52:41} =$	z	$q_2 = 3$	$q^{(1)} = 3$
9			
052	$6q_1 \times q_1 \leq 52$	$q_1 = 0$	$q^{(2)} = 30$
00			
5241	$60q_0 \times q_0 \leq 5241$	$q_0 = 8$	$q^{(3)} = 308$
4864			
0377	$s = (377)_{\text{ten}}$		$q = (308)_{\text{ten}}$

Fig. 21.1 Using the pencil-and-paper algorithm to extract the square root of a decimal integer.

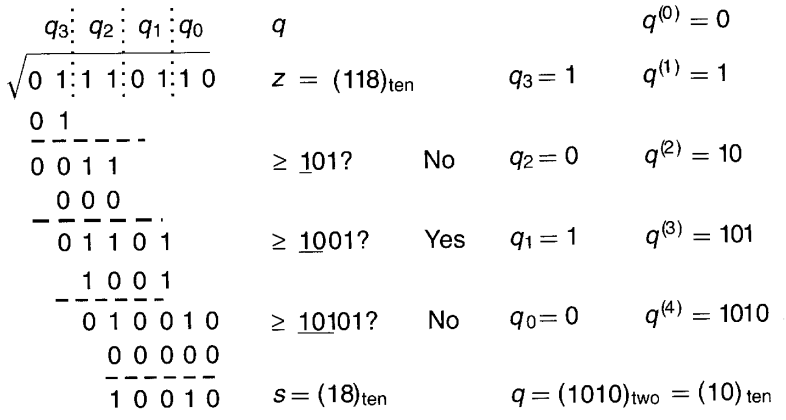


Fig. 21.2 Extracting the square root of a binary integer using the pencil-and-paper algorithm.

obtained by appending $0q_{k-i-1}$ to the right end of the partial root $q^{(i)}$. Thus, since the root digits are in {0, 1}, the problem of binary square-rooting reduces to subtracting a set of numbers, each being 0 or a shifted version of $(q^{(i)}01)_{\text{two}}$, from the radicand z.

The preceding discussion and Fig. 21.3 also apply to nonbinary square-rooting, except that with $r > 2$, both the selection of the next root digit q_{k-i-1} and the computation of the term $(2rq^{(i)} + q_{k-i-1}) \times q_{k-i-1}$ become more difficult. The rest of the process, however, remains substantially the same.

21.2 RESTORING SHIFT/SUBTRACT ALGORITHM

Like division, square-rooting can be formulated as a sequence of shift and subtract operations. The formulation is somewhat cleaner if we think in terms of fractional operands rather than integers. In fact, since in practice square-rooting is applied to floating-point numbers, we formulate our shift/subtract algorithms for a radicand in the range $1 \leq z < 4$ corresponding to the significand of a floating-point number in the IEEE standard format. Because the exponent must be halved in floating-point square-rooting, we decrement an odd exponent by 1 to make it even and shift the significand to the left by 1 bit; this accounts for the extended range assumed for z. The notation for our algorithm is thus as follows:

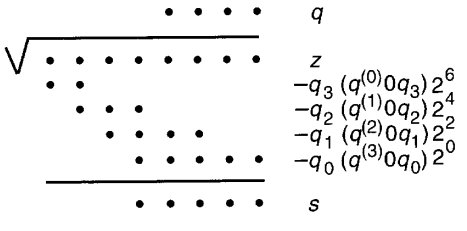


Fig. 21.3 Binary square-rooting in dot notation.

z	Radicand	$z_1 z_0 . z_{-1} z_{-2} \cdots z_{-l}$	$(1 \leq z < 4)$
q	Square-root	$1 . q_{-1} q_{-2} \cdots q_{-l}$	$(1 \leq q < 2)$
s	Scaled remainder	$s_1 s_0 . s_{-1} s_{-2} \cdots s_{-l}$	$(0 \leq s < 4)$

With these assumptions, binary square-rooting is defined by the recurrence

$$s^{(j)} = 2s^{(j-1)} - q_{-j}(2q^{(-j)} + 2^{-j}q_{-j}) \quad \text{with } s^{(0)} = z - 1, q^{(0)} = 1, s^{(l)} = s$$

where, for a binary quotient with digits in {0, 1}, the term subtracted from the shifted partial remainder $2s^{(j-1)}$ is $2q^{(j-1)} + 2^{-j}$ or 0. Here, $q^{(j)}$ stands for the root up to its $(-j)$th digit; thus $q = q^{(l)}$ is the desired square root.

Here is a general proof of the preceding square-rooting recurrence. First, we note that, by definition:

$$q^{(j)} = q^{(j-1)} + 2^{-j}q_{-j}$$

During square-rooting iterations, we strive to maintain the invariant:

$$s^{(j)} = z - [q^{(j)}]^2$$

In particular, $q^{(0)} = 1$ and $s^{(0)} = z - 1$. From the preceding invariant, we derive the requirement:

$$\begin{aligned} s^{(j-1)} - s^{(j)} &= [q^{(j)}]^2 - [q^{(j-1)}]^2 = [q^{(j-1)} + 2^{-j}q_{-j}]^2 - [q^{(j-1)}]^2 \\ &= 2^{-j}q_{-j}[2q^{(j-1)} + 2^{-j}q_{-j}] \end{aligned}$$

Multiplying both sides by 2^j and rearranging the terms, we get:

$$2^j s^{(j)} = 2(2^{j-1} s^{(j-1)}) - q_{-j}[2q^{(j-1)} + 2^{-j}q_{-j}]$$

Redefining the jth partial remainder to be $2^j s^{(j)}$ yields the desired recurrence. Note that after l iterations, the partial remainder $s^{(l)}$, which is in [0, 4), represents the scaled remainder $s = 2^l(z - q^2)$.

To choose the next square-root digit q_{-j} from the set {0, 1}, we perform a trial subtraction of

$$2q^{(j-1)} + 2^{-j} = (1 q_{-1}^{(j-1)} . q_{-2}^{(j-1)} \cdots q_{-j+1}^{(j-1)} 0\ 1)_{\text{two}}$$

from the shifted partial remainder $2s^{(j-1)}$. If the difference is negative, the shifted partial remainder is not modified and $q_{-j} = 0$. Otherwise, the difference becomes the new partial remainder and $q_{-j} = 1$.

The preceding algorithm, which is similar to restoring division, is quite naturally called "restoring square-rooting." An example of binary restoring square-rooting using the preceding recurrence is shown in Fig. 21.4, where we have provided three whole digits, plus the required six fractional digits, for representing the partial remainders. Two whole digits are required given that the partial remainders, as well as the radicand z, are in [0, 4). The third whole digit is needed to accommodate the extra bit that results from shifting the partial remainder $s^{(j-1)}$ to the left to form $2s^{(j-1)}$. This bit also acts as the sign bit for the trial difference.

The hardware realization of restoring square-rooting is quite similar to restoring division. Figure 21.5 shows the required components and their connections, assuming that they will be used only for square-rooting. In practice, square-rooting hardware may be shared with division (and perhaps even multiplication). To allow such sharing of hardware, some changes are needed

z	0 1 . 1 1 0 1 1 0		(118/64)
$s^{(0)} = z - 1$	0 0 0 . 1 1 0 1 1 0	$q_0 = 1$	$q^{(0)} = 1.$
$2s^{(0)}$	0 0 1 . 1 0 1 1 0 0		
$-[2\times(1.)+2^{-1}]$	1 0 . 1		
$s^{(1)}$	1 1 1 . 0 0 1 1 0 0	$q_{-1} = 0$	$q^{(1)} = 1.0$
$s^{(1)} = 2s^{(0)}$	0 0 1 . 1 0 1 1 0 0	Restore	
$2s^{(1)}$	0 1 1 . 0 1 1 0 0 0		
$-[2\times(1.0)+2^{-2}]$	1 0 . 0 1		
$s^{(2)}$	0 0 1 . 0 0 1 0 0 0	$q_{-2} = 1$	$q^{(2)} = 1.01$
$2s^{(2)}$	0 1 0 . 0 1 0 0 0 0		
$-[2\times(1.01)+2^{-3}]$	1 0 . 1 0 1		
$s^{(3)}$	1 1 1 . 1 0 1 0 0 0	$q_{-3} = 0$	$q^{(3)} = 1.010$
$s^{(3)} = 2s^{(2)}$	0 1 0 . 0 1 0 0 0 0	Restore	
$2s^{(3)}$	1 0 0 . 1 0 0 0 0 0		
$-[2\times(1.010)+2^{-4}]$	1 0 . 1 0 0 1		
$s^{(4)}$	0 0 1 . 1 1 1 1 0 0	$q_{-4} = 1$	$q^{(4)} = 1.0101$
$2s^{(4)}$	0 1 1 . 1 1 1 0 0 0		
$-[2\times(1.0101)+2^{-5}]$	1 0 . 1 0 1 0 1		
$s^{(5)}$	0 0 1 . 0 0 1 1 1 0	$q_{-5} = 1$	$q^{(5)} = 1.01011$
$2s^{(5)}$	0 1 0 . 0 1 1 1 0 0		
$-[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1		
$s^{(6)}$	1 1 1 . 1 0 1 1 1 1	$q_{-6} = 0$	$q^{(6)} = 1.010110$
$s^{(6)} = 2s^{(5)}$	0 1 0 . 0 1 1 1 0 0	Restore	(156/64)
s (true remainder)	0 . 0 0 0 0 1 0 0 1 1 1 0 0		(156/64²)
q	1 . 0 1 0 1 1 0		(86/64)

Fig. 21.4 Example of sequential binary square-rooting by means of the restoring algorithm.

to maximize common parts. Any component or extension that is specific to one of the operations may then be incorporated into the unit's control logic. It is instructive to compare the design in Fig. 21.5 to that of restoring binary divider in Fig. 13.5.

In fractional square-rooting, the remainder is usually of no interest. To properly round the square root, we can produce an extra digit q_{-l-1} and use its value to decide whether to truncate ($q_{-l-1} = 0$) or to round up ($q_{-l-1} = 1$). The midway case, (i.e., $q_{-l-1} = 1$ with only 0s to its right), is impossible (why?), so we don't even have to test the remainder for 0.

For the Example of Fig. 21.4, an extra iteration produces $q_{-7} = 1$. So the root must be rounded up to $q = (1.010111)_{\text{two}} = 87/64$. To check that the rounded-up value is closer to the actual root than the truncated version, we note that:

$$118/64 = (87/64)^2 - 17/64^2$$

Thus, the rounded-up value yields a remainder with a smaller magnitude.

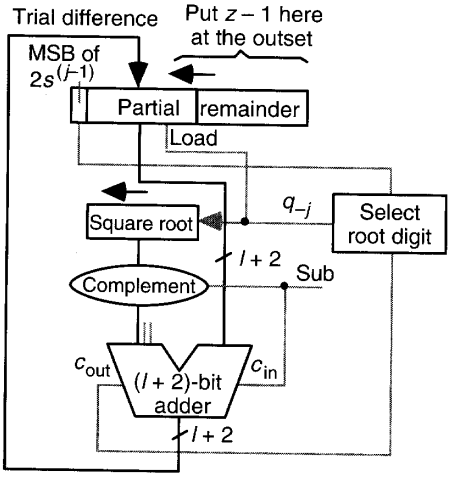


Fig. 21.5 Sequential shift/subtract restoring square-rooter.

21.3 BINARY NONRESTORING ALGORITHM

In a manner similar to binary division, one can formulate a binary nonrestoring square-rooting algorithm. Figure 21.6 shows the square-rooting example of Fig. 21.4 performed with the nonrestoring algorithm. As was the case for nonrestoring division, the square root must be corrected by subtracting *ulp* from it if the final remainder becomes negative. Remainder correction, however, is usually not needed, as discussed at the end of Section 21.2.

Performing an extra iteration in the binary square-rooting example of Fig. 21.6 yields $q_{-7} = \bar{1}$ and $q = (1.1\bar{1}1\bar{1}11\bar{1})_{\text{two}} = (1.0101101)_{\text{two}}$. This indicates that the root must be rounded up to $q = (1.010111)_{\text{two}}$.

In nonrestoring square-rooting, root digits are chosen from the set $\{\bar{1}, 1\}$ and the resulting BSD root is converted, on the fly, to binary format. The case $q_{-j} = 1$, corresponding to a nonnegative partial remainder, is handled as in the restoring algorithm; that is, it leads to the subtraction of

$$q_{-j}[2q^{(j-1)} + 2^{-j}q_{-j}] = 2q^{(j-1)} + 2^{-j}$$

from the partial remainder. For $q_{-j} = \bar{1}$, we must subtract

$$q_{-j}[2q^{(j-1)} + 2^{-j}q_{-j}] = -[2q^{(j-1)} - 2^{-j}]$$

which is equivalent to adding $2q^{(j-1)} - 2^{-j}$ (see Fig. 21.6).

From the standpoint of hardware implementation, computing the term $2q^{(j-1)} - 2^{-j}$ is problematic. Recall that $2q^{(j-1)} + 2^{-j} = 2[q^{(j-1)} + 2^{-j-1}]$ is formed by simply appending 01 to the right end of $q^{(j-1)}$ and shifting.

The following scheme allows us to form $2q^{(j-1)} - 2^{-j}$ just as easily. Suppose that we keep $q^{(j-1)}$ and $q^{(j-1)} - 2^{-j+1}$ in registers Q (partial root) and Q* (diminished partial root), respectively. Then:

z	0 1 . 1 1 0 1 1 0		(118/64)
$s^{(0)} = z - 1$	0 0 0 . 1 1 0 1 1 0	$q_0 = 1$	$q^{(0)} = 1.$
$2s^{(0)}$	0 0 1 . 1 0 1 1 0 0	$q_{-1} = 1$	$q^{(1)} = 1.1$
$-[2\times(1.)+2^{-1}]$	1 0 . 1		
$s^{(1)}$	1 1 1 . 0 0 1 1 0 0	$q_{-2} = -1$	$q^{(2)} = 1.01$
$2s^{(1)}$	1 1 0 . 0 1 1 0 0 0		
$+[2\times(1.1)-2^{-2}]$	1 0 . 1 1		
$s^{(2)}$	0 0 1 . 0 0 1 0 0 0	$q_{-3} = 1$	$q^{(3)} = 1.011$
$2s^{(2)}$	0 1 0 . 0 1 0 0 0 0		
$-[2\times(1.01)+2^{-3}]$	1 0 . 1 0 1		
$s^{(3)}$	1 1 1 . 1 0 1 0 0 0	$q_{-4} = -1$	$q^{(4)} = 1.0101$
$2s^{(3)}$	1 1 1 . 0 1 0 0 0 0		
$+[2\times(1.011)-2^{-4}]$	1 0 . 1 0 1 1		
$s^{(4)}$	0 0 1 . 1 1 1 1 0 0	$q_{-5} = 1$	$q^{(5)} = 1.01011$
$2s^{(4)}$	0 1 1 . 1 1 1 0 0 0		
$-[2\times(1.0101)+2^{-5}]$	1 0 . 1 0 1 0 1		
$s^{(5)}$	0 0 1 . 0 0 1 1 1 0	$q_{-6} = 1$	$q^{(6)} = 1.010111$
$2s^{(5)}$	0 1 0 . 0 1 1 1 0 0		
$-[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1		
$s^{(6)}$	1 1 1 . 1 0 1 1 1 1	Negative;	(−17/64)
$+[2\times(1.01011)+2^{-6}]$	1 0 . 1 0 1 1 0 1	Correct	
$s^{(6)}$ (corrected)	0 1 0 . 0 1 1 1 0 0		(156/64)
s (true remainder)	0 . 0 0 0 0 0 1 0 0 1 1 1 1 0 0		(156/64²)
q (signed-digit)	1 . 1 -1 1 -1 1 1		(87/64)
q (binary)	1 . 0 1 0 1 1 1		(87/64)
q (corrected binary)	1 . 0 1 0 1 1 0		(86/64)

Fig. 21.6 Example of sequential binary square-rooting by means of the nonrestoring algorithm.

$$q_{-j} = 1 \quad \text{Subtract} \quad 2q^{(j-1)} + 2^{-j} \quad \text{formed by shifting Q 01}$$
$$q_{-j} = -1 \quad \text{Add} \quad 2q^{(j-1)} - 2^{-j} \quad \text{formed by shifting Q*11}$$

The updating rules for Q and Q* registers are also easily derived:

$$q_{-j} = 1 \quad \Rightarrow \quad \text{Q} := \text{Q}\,1 \quad \text{Q}^* := \text{Q}\,0$$
$$q_{-j} = -1 \quad \Rightarrow \quad \text{Q} := \text{Q}^*1 \quad \text{Q}^* := \text{Q}^*0$$

The preceding can be easily extended to a square-rooting algorithm in which leading 0s or 1s in the partial remainder are detected and skipped (shifted over) while producing 0s as root digits.

The resulting algorithm is quite similar to SRT division and needs the following additional updating rule for Q and Q* registers:

$$q_{-j} = 0 \quad \Rightarrow \quad \mathrm{Q} = \mathrm{Q0} \quad \mathrm{Q}^* = \mathrm{Q}^*1$$

As in the carry-save version of SRT division with quotient digit set $[-1, 1]$, discussed in Section 14.3, we can keep the partial remainder in stored-carry form and choose the next root digit by inspecting a few most significant bits of the sum and carry components. The preceding modifications in the algorithm, and the corresponding hardware realizations, are left to the reader.

21.4 HIGH-RADIX SQUARE-ROOTING

Square-rooting can be performed in higher radices using techniques that are quite similar to those of high-radix division. The basic recurrence for fractional radix-r square-rooting is:

$$s^{(j)} = rs^{(j-1)} - q_{-j}(2q^{(j-1)} + r^{-j}q_{-j})$$

As in the case of radix-2 nonrestoring algorithm in Section 21.3, we can use two registers Q and Q* to hold $q^{(j-1)}$ and $q^{(j-1)} - r^{-j+1}$, respectively, suitably updating them in each step.

For example, with $r = 4$ and the root digit set $[-2, 2]$, Q* will hold $q^{(j-1)} - 4^{-j+1} = q^{(j-1)} - 2^{-2j+2}$. Then, it is easy to see that one of the following values must be subtracted from or added to the shifted partial remainder $rs^{(j-1)}$:

$q_{-j} = 2$	Subtract	$4q^{(j-1)} + 2^{-2j+2}$	formed by double-shifting	Q 010
$q_{-j} = 1$	Subtract	$2q^{(j-1)} + 2^{-2j}$	formed by shifting	Q 001
$q_{-j} = \bar{1}$	Add	$2q^{(j-1)} - 2^{-2j}$	formed by shifting	Q* 111
$q_{-j} = \bar{2}$	Add	$4q^{(j-1)} - 2^{-2j+2}$	formed by double-shifting	Q* 110

For ANSI/IEEE standard floating-point numbers, a radicand in the range $[1, 4)$ yields a root in $[1, 2)$. As a radix-4 number with the digit set $[-2, 2]$, the root will have a single whole digit. This is more than adequate to represent the root that is in $[1, 2)$. In fact, the first root digit can be restricted to $[0, 2]$, though not to $[0, 1]$, which at first thought might appear to be adequate (why not?).

The updating rules for Q and Q* registers are again easily derived:

$$\begin{aligned}
q_{-j} &= 2 \quad \Rightarrow \quad \mathrm{Q} := \mathrm{Q}\ 10 \quad \mathrm{Q}^* := \mathrm{Q}\ 01 \\
q_{-j} &= 1 \quad \Rightarrow \quad \mathrm{Q} := \mathrm{Q}\ 01 \quad \mathrm{Q}^* := \mathrm{Q}\ 00 \\
q_{-j} &= 0 \quad \Rightarrow \quad \mathrm{Q} := \mathrm{Q}\ 00 \quad \mathrm{Q}^* := \mathrm{Q}^*\ 11 \\
q_{-j} &= \bar{1} \quad \Rightarrow \quad \mathrm{Q} := \mathrm{Q}^*\ 11 \quad \mathrm{Q}^* := \mathrm{Q}^*\ 10 \\
q_{-j} &= \bar{2} \quad \Rightarrow \quad \mathrm{Q} := \mathrm{Q}^*\ 10 \quad \mathrm{Q}^* := \mathrm{Q}^*\ 01
\end{aligned}$$

In this way, the root is obtained in standard binary form without a need for a final conversion step (conversion takes place on the fly).

As in division, root digit selection can be based on examining a few bits of the partial remainder and of the partial root. Since only a few high-order bits are needed to estimate the next root digit, s can be kept in carry-save form to speed up the iterations. One extra bit of each component of s (sum and carry) must then be examined for root digit estimation.

In fact, with proper care, the same lookup table can be used for quotient digit selection in division and root digit selection in square-rooting. To see how, let us compare the recurrences for radix-4 division and square-rooting:

$$\begin{aligned} &\text{Division:} && s^{(j)} = 4s^{(j-1)} - q_{-j}\, d \\ &\text{Square-rooting:} && s^{(j)} = 4s^{(j-1)} - q_{-j}(2q^{(j-1)} + 4^{-j}q_{-j}) \end{aligned}$$

To keep the magnitudes of the partial remainders for division and square-rooting comparable, thus allowing the use of the same tables, we can perform radix-4 square-rooting using the digit set $\{-1, -1/2, 0, 1/2, 1\}$. A radix-4 number with the latter digit set can be converted to a radix-4 number with the digit set $[-2, 2]$, or directly to binary, with no extra computation (how?). For details of the resulting square-rooting scheme, see [Omon94, pp. 387–389].

21.5 SQUARE-ROOTING BY CONVERGENCE

In Section 16.3, we used the Newton–Raphson method for computing the reciprocal of the divisor d, thus allowing division to be performed by means of multiplications with more rapid convergence. To use the Newton–Raphson method for computing $\sqrt{z}$, we choose $f(x) = x^2 - z$ which has a root at $x = \sqrt{z}$. Recall that the Newton–Raphson iteration is:

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

Thus, the function $f(x) = x^2 - z$ leads to the following convergence scheme for square-rooting:

$$x^{(i+1)} = 0.5(x^{(i)} + z/x^{(i)})$$

Each iteration involves a division, an addition, and a single-bit shift. As was the case for reciprocation, it is easy to prove quadratic convergence of x to $\sqrt{z}$. Let $\delta_i = \sqrt{z} - x^{(i)}$. Then:

$$\begin{aligned} \delta_{i+1} &= \sqrt{z} - x^{(i+1)} = \sqrt{z} - \frac{x^{(i)} + z/x^{(i)}}{2} \\ &= \frac{-(\sqrt{z} - x^{(i)})^2}{2x^{(i)}} = \frac{-\delta_i^2}{2x^{(i)}} \end{aligned}$$

Since δ_{i+1} is always negative, the recurrence converges to $\sqrt{z}$ from above. Let z be in the range $1 \leq z < 4$ (as in square-rooting with IEEE floating-point format). Then, beginning with the initial estimate $x^{(0)} = 2$, the value of $x^{(i)}$ will always remain in the range $1 \leq x^{(i)} < 2$. This means that $|\delta_{i+1}| \leq 0.5\delta_i^2$.

An initial table-lookup step can be used to obtain a better starting estimate for $\sqrt{z}$. For example, if the initial estimate is accurate to within 2^{-8}, then three iterations would be sufficient to increase the accuracy of the root to 64 bits.

Example 21.1 Suppose we want to compute the square root of $z = (2.4)_{\text{ten}}$ and the initial table lookup provides the starting value $x^{(0)} = 1.5$, accurate to 10^{-1}. Then, we will go through the following steps to find the result to eight decimal positions (accurate to 10^{-8}):

$$
\begin{aligned}
&x^{(0)} \text{ (read out from table)} = 1.5 && \text{Accurate to } 10^{-1}\\
&x^{(1)} = 0.5(x^{(0)} + 2.4/x^{(0)}) = 1.550\,000\,000 && \text{Accurate to } 10^{-2}\\
&x^{(2)} = 0.5(x^{(1)} + 2.4/x^{(1)}) = 1.549\,193\,548 && \text{Accurate to } 10^{-4}\\
&x^{(3)} = 0.5(x^{(2)} + 2.4/x^{(2)}) = 1.549\,193\,338 && \text{Accurate to } 10^{-8}
\end{aligned}
$$

Instead of referring to a table to get an estimate of $\sqrt{z}$, one can use an approximating function that is easy to compute. In the case of fractional square-rooting, that is, with z in [0.5, 1), the approximation $(1 + z)/2$ provides a good starting value without requiring any arithmetic. The error is 0 at $z = 1$ and reaches its maximum value of $0.75 - \sqrt{0.5} \approx 0.0429$, or about 6.07%, at $z = 0.5$.

For integer operands, a starting approximation with the same maximum error of 6.07% can be found as follows [Hash90]. Assume that the most significant 1 in the binary representation of an integer-valued radicand z is in position $2m - 1$ (if the most significant 1 is not in an odd position, simply double z and multiply the resulting square root by $1/\sqrt{2} \approx 0.707\,107$). Then, we have $z = 2^{2m-1} + z^{\text{rest}}$, with $0 \le z^{\text{rest}} < 2^{2m-1}$. We claim that the starting approximation

$$x^{(0)} = 2^{m-1} + 2^{-(m+1)}z = (3 \times 2^{m-2}) + 2^{-(m+1)}z^{\text{rest}}$$

which can be obtained from z by counting the leading zeros and shifting, has a maximum relative error of 6.07%. The difference between $(x^{(0)})^2$ and z is:

$$
\begin{aligned}
\Delta &= (x^{(0)})^2 - z\\
&= (9 \times 2^{2m-4}) + \frac{3z^{\text{rest}}}{4} + 2^{-2(m+1)}(z^{\text{rest}})^2 - (2^{2m-1} + z^{\text{rest}})\\
&= 2^{2m-4} - \frac{z^{\text{rest}}}{4} + 2^{-2(m+1)}(z^{\text{rest}})^2\\
&= 2^{2m-4} - \frac{z^{\text{rest}}(1 - 2^{-2m}\, z^{\text{rest}})}{4}
\end{aligned}
$$

Since the derivative of Δ with respect to z^{rest} is uniformly negative, we only need to check the two extremes to find the worst-case error. At the upper extreme (i.e., for $z^{\text{rest}} \approx 2^{2m-1}$), we have $\Delta \approx 0$. At the lower extreme of $z^{\text{rest}} = 0$, we find $\Delta = 2^{2m-4}$. For this latter case, $x^{(0)}/\sqrt{z} = 3/\sqrt{8} \approx 1.0607$.

Schwarz and Flynn [Schw96] propose a general hardware approximation method and illustrate its applicability to the square-root function. Their method consists of generating a number of Boolean terms (bits or "dots") such that when these terms are added by the same hardware that is used for multiplication, the result is a good starting approximation for the desired function. In the case of square-rooting, they show that adding about 1000 gates of complexity to a 53-bit multiplier allows for the generation of a 16-bit approximation to the square root, which can then be refined in only two iterations.

The preceding convergence method involves a division in each iteration. Since division is a relatively slow operation, especially if a dedicated hardware divider is not available,

division-free variants of the method have been suggested. One such variant relies on the availability of a circuit or table to compute the approximate reciprocal of a number. We can rewrite the square-root recurrence as follows:

$$x^{(i+1)} = x^{(i)} + 0.5(1/x^{(i)})(z - (x^{(i)})^2)$$

Let $\gamma(x^{(i)})$ be an approximation to $1/x^{(i)}$ obtained by a simple circuit or read out from a table. Then, each iteration requires a table lookup, a one-bit shift, two multiplications, and two additions. If multiplication is much more than twice as fast as division, this variant may be more efficient. However, note that because of the approximation used in lieu of the exact value of the reciprocal $1/x^{(i)}$, the convergence rate will be less than quadratic and a larger number of iterations will be needed in general.

Since we know that the reciprocal function can also be computed by Newton–Raphson iteration, one can use the preceding recurrence, but with the reciprocal itself computed iteratively, effectively interlacing the two iterative computations. Using the function $f(y) = 1/y - x$ to compute the reciprocal of x, we find the following combination of recurrences:

$$\begin{aligned} x^{(i+1)} &= 0.5(x^{(i)} + zy^{(i)}) \\ y^{(i+1)} &= y^{(i)}(2 - x^{(i)}y^{(i)}) \end{aligned}$$

The two multiplications, of z and $x^{(i)}$ by $y^{(i)}$ can be pipelined for improved speed, as discussed in Section 16.5 for convergence division. The convergence rate of this algorithm is less than quadratic but better than linear.

Example 21.2 Suppose we want to compute the square root of $z = (1.4)_{\text{ten}}$. Beginning with $x^{(0)} = y^{(0)} = 1.0$, we find the following results:

$$\begin{aligned} x^{(0)} &= & 1.0 \\ y^{(0)} &= & 1.0 \\ x^{(1)} &= 0.5(x^{(0)} + 1.4y^{(0)}) &= 1.200\,000\,000 \\ y^{(1)} &= y^{(0)}(2 - x^{(0)}y^{(0)}) &= 1.000\,000\,000 \\ x^{(2)} &= 0.5(x^{(1)} + 1.4y^{(1)}) &= 1.300\,000\,000 \\ y^{(2)} &= y^{(1)}(2 - x^{(1)}y^{(1)}) &= 0.800\,000\,000 \\ x^{(3)} &= 0.5(x^{(2)} + 1.4y^{(2)}) &= 1.210\,000\,000 \\ y^{(3)} &= y^{(2)}(2 - x^{(2)}y^{(2)}) &= 0.768\,000\,000 \\ x^{(4)} &= 0.5(x^{(3)} + 1.4y^{(3)}) &= 1.142\,600\,000 \\ y^{(4)} &= y^{(3)}(2 - x^{(3)}y^{(3)}) &= 0.822\,312\,960 \\ x^{(5)} &= 0.5(x^{(4)} + 1.4y^{(4)}) &= 1.146\,919\,072 \\ y^{(5)} &= y^{(4)}(2 - x^{(4)}y^{(4)}) &= 0.872\,001\,394 \\ x^{(6)} &= 0.5(x^{(5)} + 1.4y^{(5)}) &= 1.183\,860\,512 \approx \sqrt{1.4} \end{aligned}$$

A final variant, that has found wider application in high-performance processors, is based on computing the reciprocal of $\sqrt{z}$ and then multiplying the result by z to obtain $\sqrt{z}$. We can use the function $f(x) = 1/x^2 - z$ that has a root at $x = 1/\sqrt{z}$ for this purpose. Since $f'(x) = -2/x^3$, we get the recurrence:

$$x^{(i+1)} = 0.5x^{(i)}(3 - z(x^{(i)})^2)$$

Each iteration now requires three multiplications and one addition, but quadratic convergence leads to only a few iterations with a suitably accurate initial estimate.

The Cray-2 supercomputer uses this last method [Cray89]. An initial estimate $x^{(0)}$ for $1/\sqrt{z}$ is plugged into the equation to obtain a more accurate estimate $x^{(1)}$. In this first iteration, $1.5x^{(0)}$ and $0.5(x^{(0)})^3$ are read out from a table to reduce the number of operations to only one multiplication and one addition. Since $x^{(1)}$ is accurate to within half the machine precision, a second iteration to find $x^{(2)}$, followed by a multiplication by z, completes the process.

■ **Example 21.3** Suppose we want to obtain the square root of $z = (.5678)_{\text{ten}}$ and the initial table lookup provides the starting value $x^{(0)} = 1.3$ for $1/\sqrt{z}$. We can then to find a fairly accurate result by performing only two iterations, plus a final multiplicatin by z.

$$\begin{aligned}
x^{(0)} \text{ (read out from table)} &= 1.3 \\
x^{(1)} = 0.5x^{(0)}(3 - 0.5678(x^{(0)})^2) &= 1.326\,271\,700 \\
x^{(2)} = 0.5x^{(1)}(3 - 0.5678(x^{(1)})^2) &= 1.327\,095\,128 \\
\sqrt{z} \approx z \times x^{(2)} &= 0.753\,524\,613
\end{aligned}$$

21.6 PARALLEL HARDWARE SQUARE-ROOTERS

As stated in Section 21.2 in connection with the restoring square-rooter depicted in Fig. 21.5, and again at the end of Section 21.4, the hardware realization of digit-recurrence square-rooting algorithms (binary or high-radix) is quite similar to that of digit-recurrence division. Thus, it is feasible to modify divide or multiply/divide units (Fig. 15.9) to also compute the square-root function. An extensive discussion of design issues is available elsewhere [Zura87]. Similar observations apply to convergence methods that perform various combinations of multiplications, additions, and shifting in each iteration.

It is also possible to derive a restoring or nonrestoring array square-rooter directly from the dot notation representation of Fig. 21.3 in a manner similar to the derivation of the array dividers of Section 15.5 from the dot notation representation of division in Figure 13.1. Fig. 21.7 depicts a possible design for an 8-bit fractional square-rooter based on the nonrestoring algorithm. The design uses controlled add/subtract cells to perform the required subtraction/addition prescribed by the nonrestoring square-rooting algorithm depending on the sign of the preceding partial remainder.

The reader should be able to understand the operation of the array square-rooter of Fig. 21.7 based on our discussion of nonrestoring square-rooting in Section 21.3, and by comparison to the nonrestoring array divider in Fig. 15.8. The design of a restoring array square-rooter is left as an exercise.

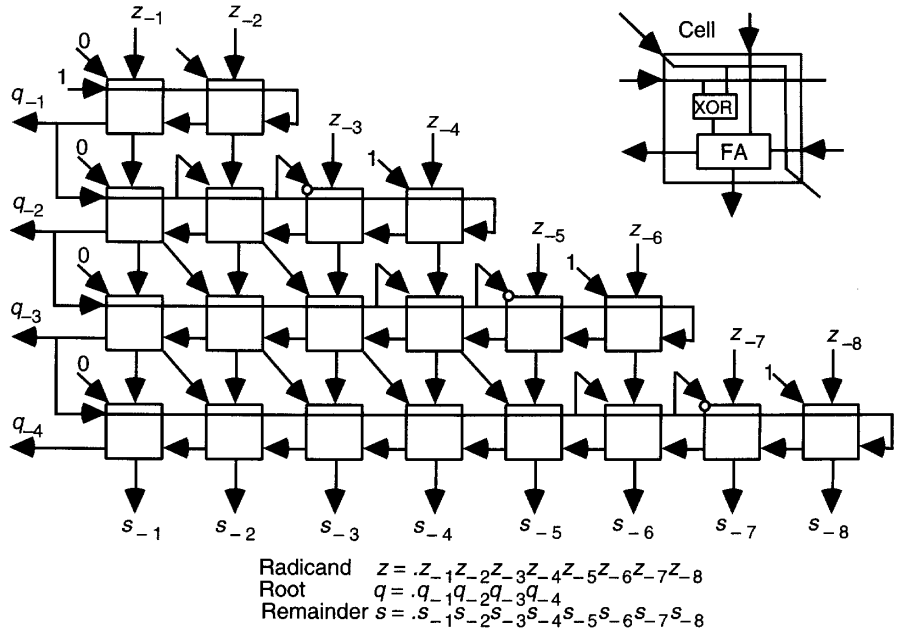


Fig. 21.7 Nonrestoring array square-rooter built of controlled add/subtract cells.

PROBLEMS

21.1 Decimal square-rooting Using the pencil-and-paper square-rooting algorithm:

a. Compute the four-digit integer square root of the decimal number $(12\ 34\ 56\ 78)_{ten}$.

b. Compute the square root of the decimal fraction $(.4321)_{ten}$ with four fractional digits.

c. Repeat part b, this time obtaining the result rounded to 4 fractional digits.

21.2 Integer square-rooting Compute the 8-bit square root of the unsigned radicand $z = (1011\ 0001\ 0111\ 1010)_{two}$.

a. Use the restoring radix-2 algorithm.

b. Use the nonrestoring radix-2 algorithm.

c. Convert the number to radix 4 and compute the square root in radix 4, using the pencil-and-paper method.

21.3 Fractional square-rooting Compute the 8-bit square root of the unsigned fractional radicand $z = (.0111\ 1100)_{two}$.

a. Use the restoring radix-2 algorithm, with the result rounded to 8 bits.

b. Repeat part a with the nonrestoring radix-2 algorithm.

c. Convert the number to radix 4 and compute the square root in radix 4, using the pencil-and-paper method.

21.4 Programmed square-rooting Write an assembly-language program similar to the division program in Fig. 13.4 for computing the square root of a $2k$-bit binary integer using the restoring shift/subtract algorithm.

21.5 Combinational square-rooter A fully combinational multiplier circuit computing $p = ax$ can be used as a squarer by connecting both its inputs to x, leading to the output $p = x^2$. A fully combinational divider circuit computes $q = z/d$. If we feed back the quotient output q to the divisor input d, can we expect to get $q = \sqrt{z}$ at the output? Discuss.

21.6 Restoring square-rooter For the restoring square-rooter in Fig. 21.5:

a. Explain the initial placement of $z - 1$ in the partial remainder register.

b. Explain the two unlabeled input bits on the left side of the adder (dotted lines).

c. Explain the alignment of the two inputs to the adder (i.e., Which bits of the partial remainder register are added to the complement of the partial square root?).

d. Provide a complete logic design for the root digit selection block.

21.7 Nonrestoring square-rooter Consider the hardware implementation of a nonrestoring square-rooter.

a. Draw a block diagram similar to Fig. 21.5 for the hardware, assuming that the partial remainder is kept is standard binary form.

b. Repeat part a for a nonrestoring square-rooter that keeps the partial remainder in stored-carry form.

c. Provide a complete logic design for the root digit selection block in part b.

21.8 High-radix integer square-rooting Compute the 8-bit square root of the following 16-bit unsigned binary numbers using the radix-4 square-rooting algorithm of Section 21.4. Do not worry about the process of selecting a root digit in $[-2, 2]$; that is, use a trial-and-error approach.

a. 0011 0001 0111 1010

b. 0111 0001 0111 1010

c. 1011 0001 0111 1010

21.9 High-radix fractional square-rooting Given the radicand $z = 0.0111\ 1100$, compute the square root $q = 0.q_{-1}q_{-2} \cdots q_{-8}$ and remainder $s = 0.0000\ 000r_{-8} \cdots r_{-16}$ using:

a. The radix-2 restoring algorithm.

b. The radix-4 algorithm with root digit set $[-2, 2]$. *Hint:* Preshifting is required to make the root representable with the given digit set.

21.10 High-radix square-rooting Consider the radix-4 square-rooting algorithm discussed in Section 21.4

a. Develop a p–q plot (similar to the p–d plot in high-radix division) for this algorithm and discuss the root digit selection process.

b. Draw a block diagram of the hardware required to execute the radix-4 square-rooting algorithm. In particular, show the complete logical design of the elements needed to update the registers Q and Q*.

c. Derive the add/subtract rules and the updating process for Q and Q* in radix-8 square-rooting with the root digit set $[-4, 4]$.

d. Briefly discuss the cost-effectiveness of the radix-8 square-rooter of part c compared to the simpler radix-4 implementation.

21.11 Rounding of the square root

a. Prove that rounding of a fractional square root (see the end of Section 21.2) can be done by generating an extra digit of the result and that the equivalent of the "sticky bit" is not required (i.e., the midway case never arises).

b. Show that as an alternative to the extra iteration, the rounding decision can be based on whether $s^{(l)} \leq q$ (truncate) or $s^{(l)} > q$ (round up).

21.12 Approximating the square-root function Show that $(1+z)/2$ is a good approximation to $\sqrt{z}$ in the extended range $0.5 \leq z < 2$ (in Section 21.5, we dealt with the range $0.5 \leq z < 1$). Demonstrate that the approximation is still easy to obtain and analyze its worst-case error. Is the extended range of any help in computing the square root of a floating-point number?

21.13 Approximating the square-root function

a. Formulate and prove a theorem similar to Theorem 16.1 (concerning the initial multiplicative factor in convergence division) that relates the accuracy of the square root approximation to the required table size.

b. Identify any special case that might allow smaller tables.

21.14 Convergence square-rooting Discuss the practicality of the following method for convergence square-rooting. Initially, the square root of a radicand in $[1, 4)$ is known to be in $[1, 2)$. The interval holding the square root is iteratively refined by a binary search process: the midpoint $m = (l + u)/2$ of the current interval $[l, u)$ is squared and the result compared to the radicand to decide if the search must be restricted to $[l, m)$ or to $[m, u)$ in the next iteration.

21.15 Convergence square-rooting

a. Derive a convergence scheme for square-rooting using the Newton–Raphson method and the function $f(x) = z/x^2 - 1$.

b. Show that with $x^{(0)} = y^{(0)} = 1$, the pair of iterative formulas $x^{(i+1)} = x^{(i)} + y^{(i)}z$ and $y^{(i+1)} = x^{(i)} + y^{(i)}$ converges to $x^{(m)}/y^{(m)} = \sqrt{z}$.

21.16 Square-rooting by convergence Consider square-rooting by convergence when the radicand z is in the range $[1, 4)$, intermediate computations are to be performed with 60 bits of precision after the radix point, and a lookup table is used to provide an initial estimate for the square root that is accurate to within $\pm 2^{-8}$. Identify the best approach by determining the required table size and analyzing the convergence methods described in Section 21.5. Assume that hardware add, multiply, and divide times are 1, 3, and 8 units, respectively, and that shifting and control overheads can be ignored.

21.17 Array square-rooter

a. In the nonrestoring square-rooter of Fig. 21.7, explain the roles of all inputs connected to a constant 0 or 1, the connections from horizontally broadcast signals to the diagonal inputs of some cells, and the wraparound connections of the cells located at the right edge.

b. Present the design of a restoring array square-rooter for radix-2 radicands.

c. Compare the design of part b to the nonrestoring square-rooter of Fig. 21.7 with regard to speed and cost.

d. Design a 4-bit array squarer with a cell layout similar to that in Fig. 21.7, so that the operand enters from the left side and the square emerges from the bottom.

e. Based on the design of part d, build an array that can compute the square or square-root function depending on the status of a control signal.

REFERENCES

[Agra79] Agrawal, D.P., "High-Speed Arithmetic Arrays," *IEEE Trans. Computers*, Vol. 28, No. 3, pp. 215–224, 1979.

[Cimi90] Ciminiera, L., and P. Montuschi, "Higher Radix Square Rooting," *IEEE Trans. Computers*, Vol. 39, No. 10, pp. 1220–1231, 1990.

[Cray89] Cray Research, "Cray-2 Computer System Functional Description Manual," Cray Research, Chippewa Falls, WI, 1989.

[Erce94] Ercegovac, M.D., and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.

[Hash90] Hashemian, R., "Square Rooting Algorithms for Integer and Floating-Point Numbers," *IEEE Trans. Computers*, Vol. 39, No. 8, pp. 1025–1029, 1990.

[Maje85] Majerski, S., "Square-Root Algorithms for High-Speed Digital Circuits," *IEEE Trans. Computers*, Vol. 34, No. 8, pp. 1016–1024, 1985.

[Maji71] Majithia, J.C., "Cellular Array for Extraction of Squares and Square Roots of Binary Numbers," *IEEE Trans. Computers*, Vol. 20, No. 12, pp. 1617–1618, 1971.

[Mont90] Montuschi, P., and M. Mezzalama, "Survey of Square-Rooting Algorithms," *Proc. IEE: Pt. E*, Vol. 137, pp. 31–40, 1990.

[Omon94] Omondi, A.R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Schw96] Schwarz, E.M., and M.J. Flynn, "Hardware Starting Approximation Method and Its Application to the Square Root Operation," *IEEE Trans. Computers*, Vol. 45, No. 12, pp. 1356–1369, 1996.

[Zura87] Zurawski, J.H.P., and J.B. Gosling, "Design of a High-Speed Square Root, Multiply, and Divide Unit," *IEEE Trans. Computers*, Vol. 36, No. 1, pp. 13–23, 1987.

Chapter 22 THE CORDIC ALGORITHMS

In this chapter, we learn an elegant convergence method for evaluating trigonometric and many other functions of interest. We will see that, somewhat surprisingly, all these functions can be evaluated with delays and hardware costs that are only slightly higher than those of division or square-rooting. The simple form of CORDIC is based on the observation that if a unit-length vector with end point at $(x, y) = (1, 0)$ is rotated by an angle z, its new end point will be at $(x, y) = (\cos z, \sin z)$. Thus, $\cos z$ and $\sin z$ can be computed by finding the coordinates of the new end point of the vector after rotation by z. Chapter topics include:

22.1 ROTATIONS AND PSEUDOROTATIONS

Consider the vector $OE^{(i)}$ in Fig. 22.1, having one end point at the origin O and the other at $E^{(i)}$ with coordinates $(x^{(i)}, y^{(i)})$. If $OE^{(i)}$ is rotated about the origin by an angle $\alpha^{(i)}$, as shown in Fig. 22.1, the new end point $E^{(i+1)}$ will have coordinates $(x^{(i+1)}, y^{(i+1)})$ satisfying:

$$
\begin{aligned}
x^{(i+1)} &= x^{(i)} \cos \alpha^{(i)} - y^{(i)} \sin \alpha^{(i)} \\
&= \frac{x^{(i)} - y^{(i)} \tan \alpha^{(i)}}{(1 + \tan^2 \alpha^{(i)})^{1/2}} \\
y^{(i+1)} &= y^{(i)} \cos \alpha^{(i)} + x^{(i)} \sin \alpha^{(i)} \qquad \text{[Real rotation]} \\
&= \frac{y^{(i)} + x^{(i)} \tan \alpha^{(i)}}{(1 + \tan^2 \alpha^{(i)})^{1/2}}
\end{aligned}
$$

$$z^{(i+1)} = z^{(i)} - \alpha_i$$

where the variable z allows us to keep track of the total rotation over several steps. More specifically, $z^{(i)}$ can be viewed as the residual rotation still to be performed; thus $z^{(i+1)}$ is the updated version of $z^{(i)}$ after rotation by $\alpha^{(i)}$. If $z^{(0)}$ is the initial rotation goal and if the $\alpha^{(i)}$ angles are selected at each step such that $z^{(m)}$ tends to 0, the end point $\mathrm{E}^{(m)}$ with coordinates $(x^{(m)}, y^{(m)})$ will be the end point of the vector after it has been rotated by the angle $z^{(0)}$.

In the CORDIC computation method, which derives its name from the **co**ordinate **ro**tations **di**gital **c**omputer designed in the late 1950s, rotation steps are replaced by pseudorotations as depicted in Fig. 22.1. Whereas a real rotation does not change the length $R^{(i)}$ of the vector, a pseudorotation step increases its length to:

$$R^{(i+1)} = R^{(i)}(1 + \tan^2 \alpha^{(i)})^{1/2}$$

The coordinates of the new end point $\mathrm{E}'^{(i+1)}$ after pseudorotation are derived by multiplying the coordinates of $\mathrm{E}^{(i+1)}$ by the expansion factor $(1 + \tan^2 \alpha^{(i)})^{1/2}$. The pseudorotation by the angle $\alpha^{(i)}$ is thus characterized by the equations:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - y^{(i)} \tan \alpha^{(i)} \\ y^{(i+1)} &= y^{(i)} + x^{(i)} \tan \alpha^{(i)} \qquad \text{[Pseudorotation]} \\ z^{(i+1)} &= z^{(i)} - \alpha^{(i)} \end{aligned}$$

Assuming $x^{(0)} = x$, $y^{(0)} = y$, and $z^{(0)} = z$, after m real rotations by the angles $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$, we have:

$$\begin{aligned} x^{(m)} &= x \cos\left(\sum \alpha^{(i)}\right) - y \sin\left(\sum \alpha^{(i)}\right) \\ y^{(m)} &= y \cos\left(\sum \alpha^{(i)}\right) + x \sin\left(\sum \alpha^{(i)}\right) \\ z^{(m)} &= z - \left(\sum \alpha^{(i)}\right) \end{aligned}$$

After m pseudorotations by the angles $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(m)}$, with $x^{(0)} = x$, $y^{(0)} = y$, and $z^{(0)} = z$, we have:

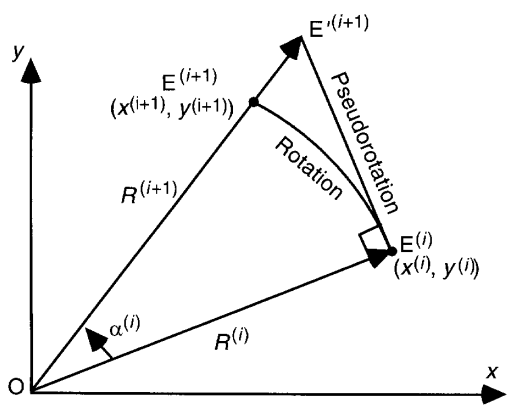


Fig. 22.1 A pseudorotation step in CORDIC.

$$
\begin{aligned}
x^{(m)} &= \left(x \cos\left(\sum \alpha^{(i)}\right) - y \sin\left(\sum \alpha^{(i)}\right)\right) \prod (1 + \tan^2 \alpha^{(i)})^{1/2} \\
&= K\left(x \cos\left(\sum \alpha^{(i)}\right) - y \sin\left(\sum \alpha^{(i)}\right)\right) \\
x^{(m)} &= \left(y \cos\left(\sum \alpha^{(i)}\right) + x \sin\left(\sum \alpha^{(i)}\right)\right) \prod (1 + \tan^2 \alpha^{(i)})^{1/2} \qquad [*] \\
&= K\left(y \cos\left(\sum \alpha^{(i)}\right) + x \sin\left(\sum \alpha^{(i)}\right)\right) \\
z^{(m)} &= z - \left(\sum \alpha^{(i)}\right)
\end{aligned}
$$

The expansion factor $K = \prod(1 + \tan^2 \alpha^{(i)})^{1/2}$ depends on the rotation angles $\alpha^{(1)}$, $\alpha^{(2)}$, ..., $\alpha^{(m)}$. However, if we always rotate by the same angles, with positive or negative signs, then K is a constant that can be precomputed. In this case, using the simpler pseudorotations instead of true rotations has the effect of expanding the vector coordinates and length by a known constant.

22.2 BASIC CORDIC ITERATIONS

To simplify each pseudorotation, pick $\alpha^{(i)}$ such that $\tan \alpha^{(i)} = d_i 2^{-i}$, $d_i \in \{-1, 1\}$. Then:

$$
\begin{aligned}
x^{(i+1)} &= x^{(i)} - d_i y^{(i)} 2^{-i} \\
y^{(i+1)} &= y^{(i)} + d_i x^{(i)} 2^{-i} \qquad \text{[CORDIC iteration]} \\
z^{(i+1)} &= z^{(i)} - d_i \tan^{-1} 2^{-i}
\end{aligned}
$$

The computation of $x^{(i+1)}$ or $y^{(i+1)}$ requires an i-bit right shift and an add/subtract. If the function $\tan^{-1} 2^{-i}$ is precomputed and stored in a table (see Table 22.1) for different values of i, a single add/subtract suffices to compute $z^{(i+1)}$. Each CORDIC iteration thus involves two shifts, a table lookup, and three additions.

If we always pseudorotate by the same set of angles (with + or − signs), then the expansion factor K is a constant that can be precomputed. For example, to pseudorotate by 30 degrees, we can pseudorotate by the following sequence of angles that add up to $\approx 30°$.

$$
\begin{aligned}
30.0 &\approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6 + 1.8 - 0.9 + 0.4 - 0.2 + 0.1 \\
&= 30.1
\end{aligned}
$$

In effect, what actually happens in CORDIC is that z is initialized to 30° and then, in each step, the sign of the next rotation angle is selected to try to change the sign of z; that is, we choose $d_i = \text{sign}(z^{(i)})$, where the sign function is defined to be −1 or 1 depending on whether the argument is negative or nonnegative. This is reminiscent of nonrestoring division.

Table 22.2 shows the process of selecting the signs of the rotation angles for a desired rotation of +30°. Figure 22.2 depicts the first few steps in the process of forcing z to 0.

TABLE 22.1
Approximate value of the function $e^{(i)} = \tan^{-1} 2^{-i}$, in degrees, for $0 \leq i \leq 9$

i	$e^{(i)}$
0	45.0
1	26.6
2	14.0
3	7.1
4	3.6
5	1.8
6	0.9
7	0.4
8	0.2
9	0.1

In CORDIC terminology, the preceding selection rule for d_i, which makes z converge to 0, is known as "rotation mode." We rewrite the CORDIC iterations as follows, where $e^{(i)} = \tan^{-1} 2^{-i}$:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - d_i(2^{-i}y^{(i)}) \\ y^{(i+1)} &= y^{(i)} + d_i(2^{-i}x^{(i)}) \\ z^{(i+1)} &= z^{(i)} - d_i e^{(i)} \end{aligned}$$

After m iterations in rotation mode, when $z^{(m)}$ is sufficiently close to 0, we have $\sum \alpha^{(i)} = z$, and the CORDIC equations [*] become:

TABLE 22.2
Choosing the signs of the rotation angles to force z to 0

i	$z^{(i)}$		$\alpha^{(i)}$		$z^{(i+1)}$
0	+30.0	−	45.0	=	−15.0
1	−15.0	+	26.6	=	+11.6
2	+11.6	−	14.0	=	−2.4
3	−2.4	+	7.1	=	+4.7
4	+4.7	−	3.6	=	+1.1
5	+1.1	−	1.8	=	−0.7
6	−0.7	+	0.9	=	+0.2
7	+0.2	−	0.4	=	−0.2
8	−0.2	+	0.2	=	+0.0
9	+0.0	−	0.1	=	−0.1

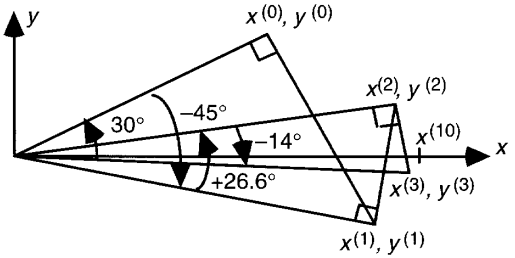


Fig. 22.2 The first three of 10 pseudorotations leading from $(x^{(0)}, y^{(0)})$ to $(x^{(10)}, 0)$ in rotating by $+30°$.

$$
\begin{aligned}
x^{(m)} &= K(x \cos z - y \sin z) \\
y^{(m)} &= K(y \cos z + x \sin z) \qquad \text{[Rotation mode]} \\
z^{(m)} &= 0 \\
\text{Rule:} \quad & \text{Choose } d_i \in \{-1, 1\} \text{ such that } z \to 0.
\end{aligned}
$$

The constant K in the preceding equations is $K = 1.646\ 760\ 258\ 121 \cdots$. Thus, to compute $\cos z$ and $\sin z$, one can start with $x = 1/K = 0.607\ 252\ 935 \cdots$ and $y = 0$. Then, as $z^{(m)}$ tends to 0 with CORDIC iterations in rotation mode, $x^{(m)}$ and $y^{(m)}$ converge to $\cos z$ and $\sin z$, respectively. Once $\sin z$ and $\cos z$ are known, $\tan z$ can be obtained through division if necessary.

For k bits of precision in the resulting trigonometric functions, k CORDIC iterations are needed. The reason is that for large i, we have $\tan^{-1} 2^{-i} \approx 2^{-i}$. Hence, for $i > k$, the change in z will be less than *ulp*.

In rotation mode, convergence of z to 0 is possible because each angle in Table 22.1 is more than half the previous angle or, equivalently, each angle is less than the sum of all the angles following it. The domain of convergence is $-99.7° \leq z \leq 99.7°$, where $99.7°$ is the sum of all the angles in Table 22.1. Fortunately, this range includes angles from $-90°$ to $+90°$, or $[-\pi/2, \pi/2]$ in radians. For outside the preceding range, we can use trigonometric identities to convert the problem to one that is within the domain of convergence:

$$
\begin{aligned}
\cos(z \pm 2j\pi) &= \cos z & \sin(z \pm 2j\pi) &= \sin z \\
\cos(z - \pi) &= -\cos z & \sin(z - \pi) &= -\sin z
\end{aligned}
$$

Note that these transformations become particularly convenient if angles are represented and manipulated in multiples of π radians, so that $z = 0.2$ really means $z = 0.2\pi$ radian or $36°$. The domain of convergence then includes $[-1/2, 1/2]$, with numbers outside this domain converted to numbers within the domain quite easily.

In a second way of utilizing CORDIC iterations, known as "vectoring mode," we make y tend to zero by choosing $d_i = -\text{sign}(x^{(i)} y^{(i)})$. After m iterations in vectoring mode we have $\tan(\sum \alpha^{(i)}) = -y/x$. This means that:

$$
\begin{aligned}
x^{(m)} &= K\left[x \cos\left(\sum \alpha^{(i)}\right) - y \sin\left(\sum \alpha^{(i)}\right)\right] \\
&= \frac{K\left(x - y \tan\left(\sum \alpha^{(i)}\right)\right)}{\left[1 + \tan^2\left(\sum \alpha^{(i)}\right)\right]^{1/2}}
\end{aligned}
$$

$$= \frac{K(x + y^2/x)}{(1 + y^2/x^2)^{1/2}}$$
$$= K(x^2 + y^2)^{1/2}$$

The CORDIC equations [*] thus become:

$$x^{(m)} = K(x^2 + y^2)^{1/2}$$
$$y^{(m)} = 0 \qquad \text{[Vectoring mode]}$$
$$z^{(m)} = z + \tan^{-1}(y/x)$$

Rule: Choose $d_i \in \{-1, 1\}$ such that $y \to 0$.

One can compute $\tan^{-1} y$ in vectoring mode by starting with $x = 1$ and $z = 0$. This computation always converges. However, one can take advantage of the identity

$$\tan^{-1}(1/y) = \pi/2 - \tan^{-1} y$$

to limit the range of fixed-point numbers that are encountered. We will see later, in Section 22.5, that the CORDIC method also allows the computation of other inverse trigonometric functions.

22.3 CORDIC HARDWARE

A straightforward hardware implementation for CORDIC arithmetic is shown in Fig. 22.3. It requires three registers for x, y, and z, a lookup table to store the values of $e^{(i)} = \tan^{-1} 2^{-i}$, and two shifters to supply the terms $2^{-i}x$ and $2^{-i}y$ to the adder/subtractor units. The d_i factor (-1 or 1) is accommodated by selecting the (shifted) operand or its complement.

Of course, a single adder and one shifter can be shared by the three computations if a reduction in speed by a factor of about 3 is acceptable. In the extreme, CORDIC iterations can be implemented in firmware (microprogram) or even software using the ALU and general-purpose registers of a standard microprocessor. In this case, the lookup table supplying the terms $e^{(i)}$ can be stored in the control ROM or in main memory.

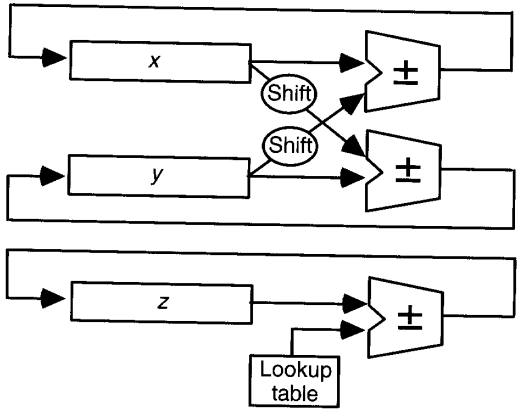


Fig. 22.3 Hardware elements needed for the CORDIC method.

Where high speed is not required and minimizing the hardware cost is important (as in calculators), the adders in Fig. 22.3 can be bit-serial. Then with k-bit operands, $O(k^2)$ clock cycles would be required to complete the k CORDIC iterations. This is acceptable for handheld calculators, since even a delay of tens of thousands of clock cycles constitutes a small fraction of a second and thus is hardly noticeable to a human user. Intermediate between the fully parallel and fully bit-serial realizations are a wide array of digit-serial (say decimal or radix-16) implementations that provide trade-offs of speed versus cost.

22.4 GENERALIZED CORDIC

The basic CORDIC method of Section 22.2 can be generalized to provide a more powerful tool for function evaluation. Generalized CORDIC is defined as follows:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - \mu d_i y^{(i)} 2^{-i} \\ y^{(i+1)} &= y^{(i)} + d_i x^{(i)} 2^{-i} \\ z^{(i+1)} &= z^{(i)} - d_i e^{(i)} \end{aligned} \qquad \text{[Generalized CORDIC iteration]}$$

Note that the only difference with basic CORDIC is the introduction of the parameter μ in the equation for x and redefinition of $e^{(i)}$. The parameter μ can assume one of three values:

$\mu = 1$	Circular rotations (basic CORDIC)	$e = \tan^{-1} 2^{-i}$
$\mu = 0$	Linear rotations	$e^{(i)} = 2^{-i}$
$\mu = -1$	Hyperbolic rotations	$e^{(i)} = \tanh^{-1} 2^{-i}$

Figure 22.4 illustrates the three types of rotation in generalized CORDIC.

For the circular case with $\mu = 1$, we introduced pseudorotations that led to expansion of the vector length by a factor $(1 + \tan^2 \alpha^{(i)})^{1/2} = 1/\cos \alpha^{(i)}$ in each step, and by $K = 1.646\,760\,258\,121 \cdots$ overall, where the vector length is the familiar $R^{(i)} = \sqrt{x^2 + y^2}$. With reference to Fig. 22.4, the rotation angle AOB can be defined in terms of the area of the sector AOB as follows:

$$\text{angle AOB} = \frac{2(\text{area AOB})}{(\text{OU})^2}$$

The following equations, repeated here for ready comparison, characterize the results of circular CORDIC rotations:

$$\begin{aligned} x^{(m)} &= K(x \cos z - y \sin z) \\ y^{(m)} &= K(y \cos z + x \sin z) \\ z^{(m)} &= 0 \\ \text{Rule:} \quad & \text{Choose } d_i \in \{-1, 1\} \text{ such that } z \to 0. \end{aligned} \qquad \text{[Circular rotation mode]}$$

$$
\begin{aligned}
x^{(m)} &= K(x^2 + y^2)^{1/2} \\
y^{(m)} &= 0 \qquad\qquad\qquad\qquad\qquad\qquad \text{[Circular vectoring mode]} \\
z^{(m)} &= z + \tan^{-1}(y/x) \\
\text{Rule:} &\quad \text{Choose } d_i \in \{-1, 1\} \text{ such that } y \to 0.
\end{aligned}
$$

In linear rotations corresponding to $\mu = 0$, the end point of the vector is kept on the line $x = x^{(0)}$ and the vector "length" is defined by $R^{(i)} = x^{(i)}$. Hence, the length of the vector is always its true length OV and the scaling factor is 1 (our pseudorotations are true linear rotations in this case). The following equations characterize the results of linear CORDIC rotations:

$$
\begin{aligned}
x^{(m)} &= x \\
y^{(m)} &= y + xz \qquad\qquad\qquad\qquad\qquad\qquad \text{[Linear rotation mode]} \\
z^{(m)} &= 0 \\
\text{Rule:} &\quad \text{Choose } d_i \in \{-1, 1\} \text{ such that } z \to 0.
\end{aligned}
$$

$$
\begin{aligned}
x^{(m)} &= x \\
y^{(m)} &= 0 \qquad\qquad\qquad\qquad\qquad\qquad \text{[Linear vectoring mode]} \\
z^{(m)} &= z + y/x \\
\text{Rule:} &\quad \text{Choose } d_i \in \{-1, 1\} \text{ such that } y \to 0.
\end{aligned}
$$

Hence, linear CORDIC rotations can be used to perform multiplication (rotation mode, $y = 0$), multiply-add (rotation mode), division (vectoring mode, $z = 0$), or divide-add (vectoring mode).

In hyperbolic rotations corresponding to $\mu = -1$, the rotation "angle" EOF can be defined in terms of the area of the hyperbolic sector EOF as follows:

$$\text{angle EOF} = \frac{2(\text{area EOF})}{(\text{OW})^2}$$

The vector "length" is defined as $R^{(i)} = \sqrt{x^2 - y^2}$, with the length expansion due to pseudorotation being $(1 - \tanh^2 \alpha^{(i)})^{1/2} = 1/\cosh\alpha^{(i)}$. Because $\cosh \alpha^{(i)} > 1$, the vector length actually shrinks, leading to an overall shrinkage factor $K' = 0.828\ 159\ 360\ 960\ 2 \cdots$ after all the iterations. The following equations characterize the results of hyperbolic CORDIC rotations:

$$
\begin{aligned}
x^{(m)} &= K'(x \cosh z + y \sinh z) \\
y^{(m)} &= K(y \cosh z + x \sinh z) \qquad\qquad \text{[Hyperbolic rotation mode]} \\
z^{(m)} &= 0 \\
\text{Rule:} &\quad d_i \in \{-1, 1\} \text{ such that } z \to 0.
\end{aligned}
$$

$$
\begin{aligned}
x^{(m)} &= K(x^2 - y^2)^{1/2} \\
y^{(m)} &= 0 \qquad\qquad\qquad\qquad\qquad\qquad \text{[Hyperbolic vectoring mode]} \\
z^{(m)} &= z + \tanh^{-1}(y/x) \\
\text{Rule:} &\quad d_i \in \{-1, 1\} \text{ such that } y \to 0.
\end{aligned}
$$

Hence, hyperbolic CORDIC rotations can be used to compute the hyperbolic sine and cosine functions (rotation mode, $x = 1/K'$, $y = 0$) or the $\tanh^{-1}$ function (vectoring mode, $x = 1$, $z = 0$). Other functions can be computed indirectly, as we shall see shortly.

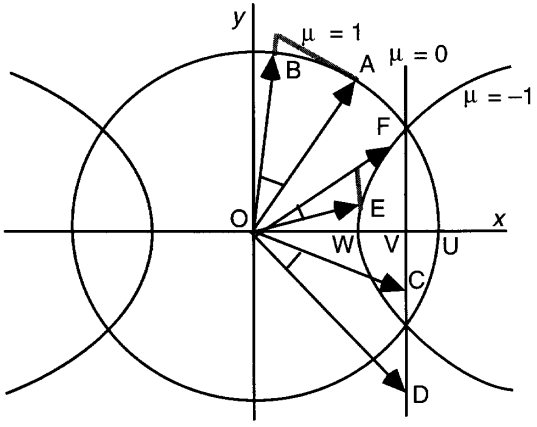


Fig. 22.4 Circular, linear, and hyperbolic CORDIC.

Convergence of circular CORDIC iterations was discussed in Section 22.2. Linear CORDIC iterations trivially converge for suitably restricted values of z (rotation mode) or y (vectoring mode). For hyperbolic CORDIC iterations, ensuring convergence is a bit more tricky, since whereas $\tan^{-1}(2^{-(i+1)}) \geq 0.5\tan^{-1}(2^{-i})$, the corresponding relation for tanh, namely, $\tanh^{-1}(2^{-(i+1)}) \geq 0.5\tanh^{-1}(2^{-i})$, does not hold in general.

A relatively simple cure is to repeat steps $i = 4, 13, 40, 121, \ldots, j, 3j+1, \ldots$ to ensure convergence (each term is 1 more than 3 times the preceding term). In other words, the iterations corresponding to the foregoing values of i are executed twice. The effect of these repetitions on performance is minimal because in practice we always stop for $m < 121$. These repeated steps have already been taken into account in computing the shrinkage constant K' given earlier. With these provisions, convergence in computing hyperbolic sine and cosine functions is guaranteed for $|z| < 1.13$ and in the case of the $\tanh^{-1}$ function, for $|y| < 0.81$.

The preceding convergence domains are more than adequate to compute the cosh, sinh, and $\tanh^{-1}$ functions over the entire range of arguments using the following identities that hold for $|z| < \ln 2 \approx 0.69$:

$$\cosh(q \ln 2 + z) = 2^{q-1}[\cosh z + \sinh z + 2^{-2q}(\cosh z - \sinh z)]$$

$$\sinh(q \ln 2 + z) = 2^{q-1}[\cosh z + \sinh z - 2^{-2q}(\cosh z - \sinh z)]$$

$$\tanh^{-1}(1 - 2^{-e}s) = \tanh^{-1}\left(\frac{2 - s - 2^{-e}s}{2 + s - 2^{-e}s}\right) + \frac{e \ln 2}{2}$$

22.5 USING THE CORDIC METHOD

We have already seen that the generalized CORDIC method can directly compute sin, cos, $\tan^{-1}$, sinh, cosh, $\tanh^{-1}$, as well as multiplication and division functions. To use CORDIC iterations for computing these functions, it is necessary to check that the arguments are within the domain of convergence and to convert the problem, if necessary, to one for which the iterations are guaranteed to converge.

Of course, somewhat more complex functions such as $\tan^{-1}(y/x)$, $y + xz$, $(x^2 + y^2)^{1/2}$, $(x^2 - y^2)^{1/2}$, and $e^z = \sinh z + \cosh z$, can also be directly computed with suitable initializations. We will see shortly that some special cases of the above, such as $(1 + w^2)^{1/2}$ and $(1 - w^2)^{1/2}$, are quite useful in computing other functions.

Many other functions are computable by suitable pre- or postprocessing steps or by multiple passes through the CORDIC hardware. Figure 22.5 provides a summary of CORDIC for ease of reference and also contains formulas for computing some of these other functions. For example, the tan function can be computed by first computing sin and cos and then performing a division, perhaps through another set of (linear) CORDIC iterations. Similarly, the tanh function can be computed through dividing sinh by cosh.

Computing the natural logarithm function, ln w, involves precomputing $y = w - 1$ and $x = w + 1$ via two additions and then using the identity:

$$\ln w = 2 \tanh^{-1} \left| \frac{w-1}{w+1} \right|$$

Logarithms in other bases (such as 2 or 10) can be obtained from the natural logarithm through multiplication by constant factors. Thus, all such logarithms can be computed quite easily by suitably modifying the constant 2 in the preceding equation.

Exponentiation can be done through CORDIC iterations by noting that

$$w^t = e^{t \ln w}$$

with the natural logarithm, multiplication, and the exponential function all computable through CORDIC iterations.

The following procedures for computing the functions $\sin^{-1}$, $\cos^{-1}$, $\sinh^{-1}$, $\cosh^{-1}$, and for square-rooting are also listed in Fig. 22.5:

$$\begin{aligned}
\cos^{-1} w &= \tan^{-1}(y/w) && \text{for } y = \sqrt{1 - w^2} \\
\sin^{-1} w &= \tan^{-1}(w/x) && \text{for } x = \sqrt{1 - w^2} \\
\cosh^{-1} w &= \ln(w + x) && \text{for } x = \sqrt{1 - w^2} \\
\sinh^{-1} w &= \ln(w + x) && \text{for } x = \sqrt{1 + w^2} \\
\sqrt{w} &= \sqrt{x^2 - y^2} && \text{for } x = w + 1/4 \text{ and } y = w - 1/4
\end{aligned}$$

Modified forms of CORDIC have been suggested for computing still other functions or for computing some of the aforementioned functions more efficiently. Some of these are explored in the end-of-chapter problems.

From the preceding discussion, we see that a CORDIC computation unit can evaluate virtually all functions of common interest and is, in a sense, a universally efficient hardware implementation for evaluating these functions.

The number or iterations in CORDIC is fixed, to ensure that K and K' remain constants. In other words, if at some point during the computation in rotation (vectoring) mode z (y) becomes 0, we cannot stop the computation, except of course for the linear version with $\mu = 0$. Thus, it appears that we always need k iterations for k digits of precision. Recall that basic sequential multiplication and division algorithms, discussed in Chapters 11 and 16, also involve k shift/add iterations. Each iteration of CORDIC requires three shift/adds. Nevertheless, it is quite remarkable that a large number of useful, and seemingly complicated, functions can be

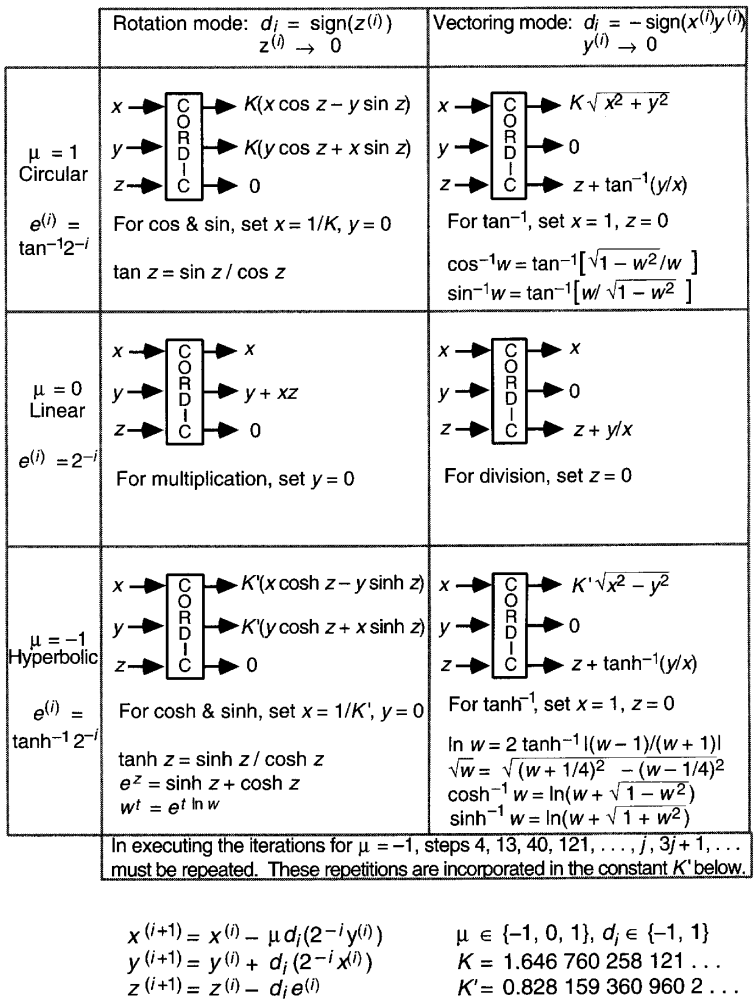


Fig. 22.5 Summary of generalized CORDIC algorithms.

computed through CORDIC with a latency that is essentially comparable to that of sequential multiplication or division.

Note that it is possible to terminate the CORDIC algorithm with $\mu \neq 0$ before k iterations, or to skip some rotations, by keeping track of the expansion factor via the recurrence:

$$(K^{(i+1)})^2 = (K^{(i)})^2(1 \pm 2^{-2i})$$

Thus, by using an additional shift/add in each iteration to update the square of the expansion factor, we can free ourselves from the requirement that every rotation angle be used once and only once (or exactly twice in some iterations of the hyperbolic pseudorotations). At the end, after m iterations, we may have to divide the results by the square root of the $(K^{(m)})^2$ value thus obtained. Given the additional variable to be updated and the final adjustment steps involving

square-rooting and division, these modifications are usually not worthwhile and *constant-factor* CORDIC is almost always preferred to the *variable-factor* version above.

Several speedup methods have been suggested to reduce the number of iterations in constant-factor CORDIC to less than k. One idea for circular CORDIC (in rotation mode) is to do $k/2$ iterations as usual and then combine the remaining $k/2$ iterations into a single step, involving multiplication, by means of the following:

$$\begin{aligned} x^{(k/2+1)} &= x^{(k/2)} - y^{(k/2)} z^{(k/2)} \\ y^{(k/2+1)} &= y^{(k/2)} + x^{(k/2)} z^{(k/2)} \\ z^{(k/2+1)} &= z^{(k/2)} - z^{(k/2)} = 0 \end{aligned}$$

This is possible because for very small values of z, we have $\tan^{-1} z \approx z \approx \tan z$. The expansion factor K presents no problem because for $e^{(i)} < 2^{-k/2}$, the contribution of the ignored terms that would have been multiplied by K is provably less than *ulp*. In other words, the same expansion factor K can be used with $k/2$ or more iterations.

Like high-radix multiplication and division algorithms, CORDIC can be extended to higher radices. For example, in a radix-4 CORDIC algorithm, d_i assumes values in $\{-2, -1, 1, 2\}$ (perhaps with 0 also included in the set) rather than in $\{-1, 1\}$. The hardware required for the radix-4 version of CORDIC is quite similar to Fig. 22.3, except that 2-to-1 multiplexers are inserted after the shifters and the lookup table to allow the operand or twice the operand to be supplied to the corresponding adder/subtractor. The contents of the lookup table will of course be different for the radix-4 version. The number of iterations in radix-4 CORDIC will be half that of the radix-2 algorithm.

Such high-radix algorithms are best understood in terms of additive and multiplicative normalization methods discussed in Chapter 23.

22.6 AN ALGEBRAIC FORMULATION

Let us accept that the following iterations, with initial values $u^{(0)} = u$ and $v^{(0)} = v$, lead to the computation of the exponential function $v^{(m)} = ve^u$ when $u^{(m)}$ is made to converge to 0 (we will prove this in Section 23.3).

$$\begin{aligned} u^{(i+1)} &= u^{(i)} - \ln\, c^{(i)} \\ v^{(i+1)} &= v^{(i)} c^{(i)} \end{aligned}$$

Since $\cos\, z + j\, \sin\, z = e^{jz}$, where $j = \sqrt{-1}$, we can compute both $\cos z$ and $\sin z$ by means of the iterations above if we start with $v^{(0)} = 1$ and $u^{(0)} = jz$ and use complex arithmetic. Consider now the identity

$$a + jb = \sqrt{a^2 + b^2}\, e^{j\theta} = \sqrt{a^2 + b^2}\, (\cos\, \theta + j\, \sin\, \theta)$$

where $\theta = \tan^{-1}(b/a)$ and suppose that we choose

$$c^{(i)} = \frac{1 + j\, d_i 2^{-i}}{\sqrt{1 + 2^{-2i}}}$$

with $d_i \in \{-1, 1\}$. Defining $g^{(i)} = \tan^{-1}(d_i 2^{-i})$, the complex number $c^{(i)}$ can be written in the form:

$$c^{(i)} = \frac{\sqrt{1+2^{-2i}}(\cos\ g^{(i)} + j\ \sin\ g^{(i)})}{\sqrt{1+2^{-2i}}} = \exp(jg^{(i)})$$

This leads to:

$$\ln\ c(i) = jg^{(i)} = j\ \tan^{-1}(d_i 2^{-i})$$

To make the multiplication needed for computing $v^{(i+1)}$ simpler, we can replace our second recurrence by:

$$v^{(i+1)} = v^{(i)} c^{(i)} \sqrt{1+2^{-2i}} = v^{(i)}(1 + jd_i 2^{-i})$$

The effect of multiplying the right-hand side by $\sqrt{1+2^{-2i}}$ will change $v^{(m)} = v^{(0)} e^{jz}$ to:

$$v^{(m)} = v^{(0)} e^{jz} \prod_{i=1}^{m-1} \sqrt{1+2^{-2i}}$$

Thus, we can still get $v^{(m)} = e^{jz}$ by setting $v^{(0)} = 1/(\prod_{i=1}^{m-1} \sqrt{1+2^{-2i}})$ instead of $v^{(0)} = 1$. Note that in the terminology of circular CORDIC, the term $\prod_{i=1}^{m-1} \sqrt{1+2^{-2i}}$ is the expansion factor K and the complex multiplication

$$v^{(i+1)} = v^{(i)}(1 + j\, d_i 2^{-i}) = (x^{(i)} + jy^{(i)})(1 + j\, d_i 2^{-i})$$

is performed by computing the real and imaginary parts separately:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - d_i y^{(i)} 2^{-i} \\ y^{(i+1)} &= y^{(i)} + d_i x^{(i)} 2^{-i} \end{aligned}$$

Note also that since the variable u is initialized to the imaginary number jz and then only imaginary values $jg^{(i)}$ are subtracted from it until it converges to 0, we can ignore the factor j and use real computation on the real variable $z^{(i)} = -ju^{(i)}$, which is initialized to $z^{(0)} = z$, instead. This completes our algebraic derivation of the circular CORDIC method.

PROBLEMS

22.1 Circular CORDIC arithmetic example

a. Use the CORDIC method to compute sin 45° and cos 45°. Perform all arithmetic in decimal with at least six significant digits and show all intermediate steps. Note the absolute and relative errors by comparing the results to exact values.

b. Since sin 45° = cos 45°, explain any difference in the accuracy of the two results.

c. Repeat part a for $\tan^{-1}$ 1.

22.2 Circular CORDIC arithmetic example

a. Use the CORDIC method to compute sin 30° and cos 30°. Perform all arithmetic in decimal with at least six significant digits and show all intermediate steps. Note the absolute and relative errors by comparing the results to exact values.

b. Calculate tan 30° from the results of part a and discuss its error.

c. Repeat part a for $\tan^{-1} 0.41421$.

22.3 Generalized CORDIC arithmetic example Use (generalized) CORDIC iterations, along with appropriate pre- and postprocessing steps, to compute the following. Use decimal arithmetic with at least six digits.

a. sinh 1 and cosh 1

b. $e^{0.5}$

c. $\tanh^{-1} 0.9$

d. $\sqrt{2}$

e. ln 2

f. $2^{1/3}$

22.4 Generalized CORDIC arithmetic in binary Use generalized CORDIC iterations, along with appropriate pre- and postprocessing steps, to compute the following. Use binary arithmetic with 8 bits after the radix point in all computations.

a. ln(1.1011 0001)

b. exp(.1011 0001)

c. $\sqrt{.1011\ 0001}$

d. $\sqrt[3]{.1011\ 0001}$

22.5 Multiplication/Division via CORDIC The generalized CORDIC iterations with $\mu = 0$ leave x unchanged and modify y and z as follows: $y^{(i+1)} = y^{(i)} \pm 2^{-i}x^{(i)}$, $z^{(i+1)} = z^{(i)} - (\pm 2^{-i})$.

a. Show how these iterations can be used to do multiplication and compare the procedure to basic (one-bit-at-a-time) sequential multiplication in terms of speed and implementation cost.

b. Repeat part a for division.

22.6 CORDIC preprocessing Assume that angles are represented and manipulated in multiples of π radians, as suggested near the end of Section 22.2.

a. Given an angle z' in fixed-point format, with k whole and l fractional digits, the computation of $\sin z'$ can be converted to the computation of $\pm \sin z$ or $\pm \cos z$, where z is in $[-1/2, 1/2]$. Show the details of the conversion process leading from z' to z.

b. Repeat part a for $\cos z$.

c. Repeat part a when the input z is in 32-bit IEEE standard floating-point format.

22.7 Composite CORDIC algorithms Determine which of the functions listed in Section 22.5 requires the largest number of CORDIC iterations if it is to be evaluated solely by a CORDIC computation unit and no other hardware element.

22.8 Truncated CORDIC iterations Verify that the difference between the CORDIC scale factors for m and $m/2$ iterations [i.e., $K = K^{(m)} = \prod_{i=0}^{m}(1 + 2^{-2i})^{1/2}$ and $K^{(m/2)} =$

$\prod_{i=0}^{m/2}(1 + 2^{-2i})^{1/2}]$ is less than 2^{-m}, thus justifying the truncated version of CORDIC discussed in Section 22.5.

22.9 Scaling in CORDIC If in some step of the (generalized) CORDIC algorithm we multiply both x and y by a common factor, the algorithm will still converge but the result(s) would be larger than original values by the same factor. Such *scaling* steps can be inserted at will, provided the product of all scaling factors is maintained and used at the end to adjust the final results. In the special case that the product of all scaling factors is a power of 2, the final adjustment consists of a shifting operation. How can one use scaling steps to make $(K^{(m)})^2$, normally in $[1, K^2]$ for variable-factor CORDIC, converge to 4?

22.10 Circular CORDIC constant Show that the circular CORDIC constant K need not be recomputed for each word length k and that it can be derived by simply truncating a highly precise version to k bits. In other words, the first k bits of $K^{(k)}$ will not change if we compute it by multiplying more than k "expansion" terms to obtain $K^{(m)}$ for some $m > k$ [Vach87].

22.11 Composite CORDIC algorithms

a. What would the final results be if the three output lines from the CORDIC computation box at the top left corner of Fig. 22.5 were directly connected to the three input lines of the box to its right?

b. Repeat part a for the two linear CORDIC boxes of Fig. 22.5.

c. Repeat part a for the two hyperbolic CORDIC boxes of Fig. 22.5.

22.12 Convergence of hyperbolic CORDIC To ensure the convergence of the hyperbolic version of CORDIC, certain steps must be performed twice. Consider the analogy of having to pay someone a sum z of money using bills and coins in the following denominations: \$50, \$20, \$10, \$5, \$2, \$1, \$0.50, \$0.25, \$0.10, \$0.05, and \$0.01. The sum must be paid to within \$0.01 (i.e., an error of \$0.01 in either direction is acceptable). Every denomination must be used. For example, a \$5 bill must be used, either in the form of payment or by way of refund.

a. Prove or disprove that the goal can always be accomplished for $z \leq \$100$ by giving or receiving each denomination exactly once and a few of them exactly twice.

b. Add a minimum number of new denominations to the given list so that convergence is guaranteed with each denomination used exactly once.

22.13 Algebraic formulation of CORDIC An algebraic formulation of circular CORDIC iterations was presented in Section 22.6. Construct a similar formulation for the hyperbolic version of CORDIC.

22.14 Computing tan and cot via CORDIC The function tan z or cot z, for $0 \leq z < \pi/4$, can be computed by first using circular CORDIC iterations to find sin z and cos z and then performing a division. However, if we do not need sin z or cos z and are interested only in tan z or cot z, we can use variable-factor CORDIC with no need to keep track of the expansion factor [Omon94].

a. Use this method to compute tan 30°.

b. Use this method to compute cot 15°.

c. Estimate the worst-case absolute error in $\tan z$ if we stop after k iterations.

d. Estimate the worst-case error in $\cot z$ if we stop after k iterations, and show that it can be quite large for $z \approx 0$.

22.15 Redundant CORDIC algorithms The values of x, y, and z in CORDIC computations can be represented in redundant form to speed up each iteration through carry-free addition. A problem that must be overcome is that the sign of a redundant value cannot be determined without full carry-propagation in the worst case. It has been suggested [Taka91] that an estimate of the sign be obtained by looking at a few bits of the redundant form, with the scale factor kept constant by (1) performing two rotations for every angle (possibly in opposite directions), and (2) inserting corrective iterations in some steps, the frequency of which is dependent on the accuracy of the sign estimation.

a. Study the two methods and describe their implementation requirements.

b. Compare the two methods with respect to speed and implementation cost.

22.16 High-radix CORDIC algorithms Study the issues involved in high-radix CORDIC algorithms and the differences between such algorithms with variable scale factor, constant scale factor, and constant scale factor that is forced to be a power of 2 [Lee92].

22.17 Direct CORDIC method for inverse sine and cosine The CORDIC equations [*] become $x^{(m)} = K \cos \theta$, $y^{(m)} = K \sin \theta$, and $z^{(m)} = -\theta$, where $\theta = -\sum \alpha^{(i)}$, if we start with $x = 1$, $y = 0$, and $z = 0$. To compute $\cos^{-1} u$, we pick the rotation directions (the digits d_i in $\{-1, 1\}$) such that x converges to Ku. Then, z will converge to $-\cos^{-1} u$. One way to make x converge to Ku is to compare $x^{(i)}$ to $K^{(i)}u$ at each step. If $x^{(i)} \geq K^{(i)}u$, we subtract from it; otherwise we add to it. The problem with this approach is that $K^{(i)}$ cannot be easily computed. However, if we perform each CORDIC pseudorotation exactly twice, the factor K will be replaced by K^2. Now, x must be compared to $(K^{(i)})^2 u$, a value that can be easily calculated in each step by using the recurrence $t^{(i+1)} = t^{(i)} + 2^{-2i} t^{(i)}$, with $t^{(0)} = u$ [Maze93].

a. Supply the details of the algorithm for computing $\cos^{-1} u$, including the selection rule for d_i.

b. Repeat part a for $\sin^{-1} u$.

c. How do the methods of parts a and b compare to the methods shown in Fig. 22.5 for computing the $\sin^{-1}$ and $\cos^{-1}$ functions?

d. Show that the iterations above can also lead to the computation of $\sqrt{1 - u^2}$.

e. Show how a similar modification to generalized CORDIC iterations can be used for computing the $\sinh^{-1}$, $\cosh^{-1}$, and $\sqrt{1 + u^2}$ functions.

f. Show that the use of double iterations extends the domain of convergence and that it leads to the need for extra iterations (how many?).

REFERENCES

[Dupr93] Duprat, J., and J.-M. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," *IEEE Trans. Computers*, Vol. 42, No. 2, pp. 168–178, 1993.

[Lee92] Lee, J.-A., and T. Lang, "Constant-Factor Redundant CORDIC for Angle Calculation and Rotation," *IEEE Trans. Computers*, Vol. 41, No. 8, pp. 1016–1025, 1992.

[Maze93] Mazenc, C., X. Merrheim, and J.-M. Muller, "Computing Functions $\cos^{-1}$ and $\sin^{-1}$ Using CORDIC," *IEEE Trans. Computers*, Vol. 42, No. 1, pp. 118–122, 1993.

[Omon94] Omondi, A.R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice Hall, 1994.

[Taka91] Takagi, N., T. Asada, and S. Yajima, "Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computations," *IEEE Trans. Computers*, Vol. 40, No. 9, pp. 989–995, 1991.

[Vach87] Vachss, R., "The CORDIC Magnification Function," *IEEE Micro*, Vol. 7, No. 5, pp. 83–84, October 1987.

[Vold59] Volder, J.E., "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers*, Vol. 8, pp. 330–334, September 1959.

[Walt71] Walther, J.S., "A Unified Algorithm for Elementary Functions," *Proc. Spring Joint Computer Conf.*, 1971, pp. 379–385.

[Phat98] Phatak, D.S., "Double Step Branching CORDIC: A New Algorithm for Fast Sine and Cosine Generation," *IEEE Trans. Computers*, Vol. 47, pp. 587–603, May 1998.

Chapter 23 VARIATIONS IN FUNCTION EVALUATION

The CORDIC method of Chapter 22 can be used to compute virtually all elementary functions of common interest. Now we turn to other schemes for evaluating some of the same functions. These alternate schemes may have advantages with certain implementation methods or technologies or may provide higher performance, given the availability of particular arithmetic operations as building blocks. In addition, we introduce the notion of merged arithmetic, a technique that allows us to optimize arithmetic computations at the level of bit manipulations as opposed to the word-level arithmetic found in CORDIC and other iterative methods. Chapter topics include:

23.1 ADDITIVE/MULTIPLICATIVE NORMALIZATION

We begin by introducing some terminology that is commonly used for characterizing iterative function evaluation methods. Recall from Section 16.1 that a general convergence method is characterized by two or three recurrences of the form:

$$u^{(i+1)} = f(u^{(i)}, v^{(i)}) \qquad u^{(i+1)} = f(u^{(i)}, v^{(i)}, w^{(i)})$$
$$v^{(i+1)} = g(u^{(i)}, v^{(i)}) \qquad v^{(i+1)} = g(u^{(i)}, v^{(i)}, w^{(i)})$$
$$w^{(i+1)} = h(u^{(i)}, v^{(i)}, w^{(i)})$$

Beginning with the initial values $u^{(0)}$, $v^{(0)}$, and perhaps $w^{(0)}$, we iterate such that one value, say u, converges to a constant; v and/or w then converge to the desired function(s). The iterations

are performed a preset number of times based on the required precision, or a stopping rule may be applied to determine when the precision of the result is adequate.

Making u converge to a constant is sometimes referred to as "normalization." If u is normalized by adding a term to it in each iteration, the convergence method is said to be based on *additive normalization*. If a single multiplication is needed per iteration to normalize u, then we have a *multiplicative normalization* method. These two special classes of convergence methods are important in view of the availability of cost-effective fast adders and multipliers.

Of course, since multipliers are slower and more costly than adders, we try to avoid multiplicative normalization when additive normalization will do. However, multiplicative methods often offer faster convergence, thus making up for the slower steps by requiring fewer of them. Furthermore, when the multiplicative terms are of the form 1 ± 2^a, multiplication reduces to shift and add/subtract

$$u(1 \pm 2^a) = u \pm 2^a u$$

thus making multiplicative convergence just as fast as the additive schemes. Hence, both additive and multiplicative convergence are useful in practice.

The CORDIC computation algorithms of Chapter 22 use additive normalization. The rate of convergence for CORDIC is roughly one bit or digit per iteration. Thus, CORDIC is quite similar to digit-recurrence algorithms for division and square-rooting in terms of computation speed. Convergence division and reciprocation, discussed in Chapter 16, offer examples of multiplicative normalization. The rate of convergence is much higher for this class (e.g., quadratic). Trade-offs are often possible between the complexity of each iteration and the number of iterations. Redundant and high-radix CORDIC algorithms, mentioned in Section 22.5, provide good examples of such trade-offs.

In the next three sections, we examine convergence methods based on additive or multiplicative normalization for logarithm evaluation, exponentiation, and square-rooting. Similar convergence methods exist for evaluating many other functions of interest (e.g., reciprocals, cube roots, and trigonometric functions, both circular and hyperbolic).

23.2 COMPUTING LOGARITHMS

The logarithm function and its inverse (exponentiation) are important for many applications and, thus, various methods have been suggested for their evaluation. For example, these functions are needed for converting numbers to and from logarithmic number systems (Section 17.6). We begin by discussing a method for computing $\ln x$. The following equations define a convergence method based on multiplicative normalization in which multiplications are done by shift/add:

$$x^{(i+1)} = x^{(i)}c^{(i)} = x^{(i)}(1 + d_i 2^{-i}) \quad d_i \in \{-1, 0, 1\}$$
$$y^{(i+1)} = y^{(i)} - \ln c^{(i)} = y^{(i)} - \ln(1 + d_i 2^{-i})$$

where $\ln(1 + d_i 2^{-i})$ is read out from a table. Beginning with $x^{(0)} = x$ and $y^{(0)} = y$ and choosing the d_i digits such that $x^{(m)}$ converges to 1, we have, after m steps:

$$x^{(m)} = x \prod c^{(i)} \approx 1 \quad \Rightarrow \quad \prod c^{(i)} \approx 1/x$$
$$y^{(m)} = y - \sum \ln c^{(i)} = y - \ln \prod c^{(i)} \approx y + \ln x$$

So starting with $y = 0$ leads to the computation of $\ln x$. The domain of convergence for this algorithm is easily obtained:

$$\frac{1}{\prod(1+2^{-i})} \leq x \leq \frac{1}{\prod(1-2^{-i})} \quad \text{or} \quad 0.21 \leq x \leq 3.45$$

We need k iterations to obtain $\ln x$ with k bits of precision. The reason is that for large i, we have $\ln(1 \pm 2^{-i}) \approx \pm 2^{-i}$. Thus, the kth iteration changes the value of y by at most *ulp* and subsequent iterations have even smaller effects.

Clearly, the preceding method can be used directly for x in $[1, 2)$. Any value x outside $[1, 2)$ can be written as $x = 2^q s$, with $1 \leq s < 2$. Then:

$$\begin{aligned} \ln x &= \ln(2^q s) = q \ \ln 2 + \ln s \\ &= 0.693\ 147\ 180\ q + \ln s \end{aligned}$$

The logarithm function in other bases can be computed just as easily. For example, base-2 logarithms are computed as follows:

$$\begin{aligned} \log_2 x &= \log_2(2^q s) = q + \log_2 s \\ &= q + \log_2 e \times \ln s = q + 1.442\ 695\ 041\ \ln s \end{aligned}$$

A radix-4 version of this algorithm can be easily developed. For this purpose, we begin with general, radix-r version of the preceding recurrences for x and y

$$\begin{aligned} x^{(i+1)} &= x^{(i)} b^{(i)} = x^{(i)}(1 + d_i r^{-i}) \qquad d_i \in [-a, a] \\ y^{(i+1)} &= y^{(i)} - \ln b^{(i)} = y^{(i)} - \ln(1 + d_i r^{-i}) \end{aligned}$$

where $\ln(1 + d_i r^{-i})$ is read out from a table.

In practice, it is easier to deal with scaled values $u^{(i)} = r^i(x^{(i)} - 1)$. This scaled value must then be made to converge to 0, using comparisons of the magnitude of $u^{(i)}$ with a few constants to determine the next choice for d_i. The scaled versions of the radix-r recurrences are:

$$\begin{aligned} u^{(i+1)} &= r(u^{(i)} + d_i + d_i u^{(i)} r^{-i}) \quad d_i \in [-a, a] \\ y^{(i+1)} &= y^{(i)} - \ln(1 + d_i r^{-i}) \end{aligned}$$

The following selection rules apply to $d_i \in [-2, 2]$ for the radix-4 version of this algorithm

$$d_i = \begin{cases} 2 & \text{if } u \leq -13/8 \\ 1 & \text{if } -13/8 < u \leq -5/8 \\ 0 & \text{if } -5/8 < u < 5/8 \\ -1 & \text{if } 5/8 \leq u < 13/8 \\ -2 & \text{if } u \geq 13/8 \end{cases}$$

provided u and y are initialized to $4(\delta x - 1)$ and $-\ln \delta$, respectively, with $\delta = 2$ if $1/2 \leq x < 5/8$ and $\delta = 1$ if $5/8 \leq x < 1$. For justification of the preceding rules, see [Omon94 pp. 410–412].

We next describe a clever method [Lo87] that requires the availability of a fast multiplier (actually a fast squarer would do). To compute base-2 logarithms, let $y = \log_2 x$ be a fractional number represented in binary as $(.y_{-1}y_{-2}\cdots y_{-l})_{\text{two}}$. Hence:

$$x = 2^y = 2^{(.y_{-1}y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}}$$
$$x^2 = 2^{2y} = 2^{(y_{-1}.y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}} \quad \Rightarrow \quad y_{-1} = 1 \text{ iff } x^2 \geq 2$$

Thus, computing x^2 and comparing the result to 2 allows us to determine the most significant bit y_{-1} of y. If $y_{-1} = 1$, then dividing both sides of the preceding equation by 2 yields:

$$\frac{x^2}{2} = \frac{2^{(1.y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}}}{2} = 2^{(.y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}}$$

Subsequent bits of y can be determined in a similar way. The complete procedure for computing $\log_2 x$ for $1 \leq x < 2$ is thus:

```
for i = 1 to l do
        x := x^2
        if x ≥ 2
        then y_{-i} = 1; x := x/2
        else y_{-i} = 0
        endif
endfor
```

A hardware realization for the preceding algorithm is shown in Fig. 23.1.
Generalization to base-b logarithms is straightforward if one notes that $y = \log_b x$ implies:

$$x = b^y = b^{(.y_{-1}y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}}$$
$$x^2 = b^{2y} = b^{(y_{-1}.y_{-2}y_{-3}\cdots y_{-l})_{\text{two}}} \quad \Rightarrow \quad y_{-1} = 1 \text{ iff } x^2 \geq b$$

Hence, the comparison with 2 in the base-2 version is replaced by a comparison with b for computing base-b logarithms. If $y_{-1} = 1$, then dividing both sides of the preceding equation by b allows us to iterate as before. However, since both comparison to b and division by b are in general more complicated, the method is of direct interest only for bases that are powers of 2. Note that logarithms in other bases are easily computed by scaling base-2 logarithms.

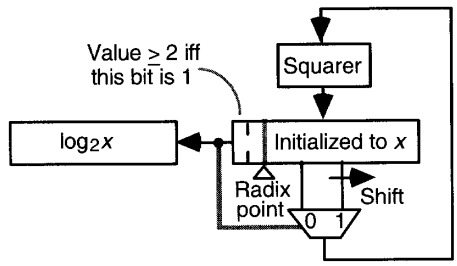


Fig. 23.1 Hardware elements needed for computing $\log_2 x$.

23.3 EXPONENTIATION

We begin by presenting a convergence method based on additive normalization for computing the exponential function e^x:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - \ln\, c^{(i)} = x^{(i)} - \ln(1 + d_i 2^{-i}) \\ y^{(i+1)} &= y^{(i)} c^{(i)} = y^{(i)}(1 + d_i 2^{-i}) \quad d_i \in \{-1, 0, 1\} \end{aligned}$$

As before, $\ln(1 + d_i 2^{-i})$ is read out from a table. If we choose the d_i digits such that x converges to 0, we have after m steps:

$$\begin{aligned} x^{(m)} &= x - \sum \ln c^{(i)} \approx 0 \quad \Rightarrow \quad \sum \ln c^{(i)} \approx x \\ y^{(m)} &= y \prod c^{(i)} = y\, e^{\ln \Pi c^{(i)}} = y\, e^{\Sigma \ln c^{(i)}} \approx y\, e^x \end{aligned}$$

The domain of convergence for this algorithm is easily obtained:

$$\sum \ln(1 - 2^{-i}) \le x \le \sum \ln(1 + 2^{-i}) \quad \text{or} \quad -1.24 \le x \le 1.56$$

The algorithm requires k iterations to provide the result with k bits of precision. This is true because in the kth iteration, $\ln(1 \pm 2^{-k}) \approx \pm 2^{-k}$ is subtracted from x. The effect of all subsequent changes would be less than *ulp*. Half the k iterations can be eliminated by noting that for $\varepsilon^2 < ulp$, we have:

$$\ln(1 + \varepsilon) = \varepsilon - \varepsilon^2/2 + \varepsilon^3/3 - \cdots \approx \varepsilon$$

So when $x^{(j)} = 0.00\cdots 00\text{xx}\cdots\text{xx}$, with $k/2$ leading zeros, we have $\ln(1 + x^{(j)}) \approx x^{(j)}$, allowing us to perform the computation step

$$\begin{aligned} x^{(j+1)} &= x^{(j)} - x^{(j)} = 0 \\ y^{(j+1)} &= y^{(j)}(1 + x^{(j)}) \end{aligned}$$

to terminate the algorithm. This termination process replaces the remaining iterations with a single (true) multiplication.

Clearly, the preceding method can be used directly for x in $(-1, 1)$. Any value x outside $(-1, 1)$ can be written as $2^q s$, for $-1 < s < 1$ and some integer q. Then, the following equality, where squaring or square-rooting is done $|q|$ times, will hold:

$$\begin{aligned} e^x = (e^s)^{2^q} &= ((\cdots(e^s)\cdots)^2)^2 \quad \text{if } q \ge 0 \\ &= \sqrt{\sqrt{\cdots\sqrt{e^s}}} \quad \text{if } q < 0 \end{aligned}$$

A more efficient method is as follows. Rewrite x as $x\ (\log_2 e)(\ln 2)$ and let $x(\log_2 e) = h + f$, with h an integer and f a fraction. Then:

$$\begin{aligned} e^x &= e^{(x \log_2 e) \ln 2} = e^{(h+f)\ln 2} \\ &= e^{h \ln 2} e^{f \ln 2} = 2^h\, e^{f \ln 2} \end{aligned}$$

Hence, one can premultiply x by $\log_2 e = 1.442\ 695\ 041\cdots$ to obtain h and f, multiply f by $\ln 2 = 0.693\ 147\ 180\cdots$ to get $u = f \ln 2$, and then compute $2^h e^u$ by using the exponential algorithm followed by shifts (or exponent adjustment).

A radix-4 version of the algorithm for computing e^x can be easily developed. Again, begin with the general radix-r version of the recurrences for x and y:

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - \ln c^{(i)} = x^{(i)} - \ln(1 + d_i r^{-i}) \\ y^{(i+1)} &= y^{(i)} c^{(i)} = y^{(i)}(1 + d_i r^{-i}) \qquad d_i \in [-a, a] \end{aligned}$$

where $\ln(1 + d_i r^{-i})$ is read out from a table. As for the radix-4 natural logarithm function, we convert the two recurrences to include scaled values $u^{(i)} = r^i x^{(i)}$, comparing the magnitude of $u^{(i)}$ with a few constants to determine the next choice for d_i. Scaled versions of the radix-r recurrences for the exponential function are:

$$\begin{aligned} u^{(i+1)} &= r(u^{(i)} - r^i \ln(1 + d_i r^{-i})) \\ y^{(i+1)} &= y^{(i)} + d_i r^{-i} y^{(i)} \qquad d_i \in [-a, a] \end{aligned}$$

Assuming $d_i \in [-2, 2]$, selection rules for the radix-4 version of this algorithm are:

$$d_i = \begin{cases} 2 & \text{if } u \le -11/8 \\ 1 & \text{if } -11/8 < u \le -3/8 \\ 0 & \text{if } -3/8 < u < 3/8 \\ -1 & \text{if } 3/8 \le u < 11/8 \\ -2 & \text{if } u \ge 11/8 \end{cases}$$

provided u and y are initialized to $4(x - \delta)$ and e^δ, respectively, with $\delta = -1/2$ if $x < -1/4$, $\delta = 0$ if $-1/4 \le x < 1/4$, and $\delta = 1/2$ if $x \ge 1/4$. For justification of the preceding rules, see [Omon94, pp. 413–415].

The general exponentiation function x^y can be computed by noting that:

$$x^y = (e^{\ln x})^y = e^{y \ln x}$$

Thus, general exponentiation can be performed by combining the logarithm and exponential functions, separated by a single multiplication.

When y is a positive integer, exponentiation can be done by repeated multiplication. In particular, when y is a constant, the methods used are reminiscent of multiplication by constants as discussed in Section 9.5. This method will lead to better accuracy, since in the preceding approach, the errors in evaluating the logarithm and exponential functions add up.

As an example, we can compute x^{25} using the identity

$$x^{25} = ((((x)^2 x)^2)^2)^2 x$$

which implies four squarings and two multiplications. Noting that

$$25 = (1\ 1\ 0\ 0\ 1)_{\text{two}}$$

leads us to a general procedure. To raise x to the power y, where y is a positive integer, initialize the partial result to 1. Scan the binary representation of y starting with its most significant bit. If the current bit is 1, multiply the partial result by x; if the current bit is 0, do not change the partial result. In either case, square the partial result before the next step (if any).

Methods similar to those used to obtain more efficient routines for multiplication by certain constants are applicable here. For example, to compute x^{15}, the preceding method involves three squarings and three multiplications (four if the redundant multiplication by 1 is not avoided):

$$x^{15} = ((((x)^2)x)^2x)^2x$$

Applying Booth's recoding $15 = (1\ 1\ 1\ 1)_{\text{two}} = (1\ 0\ 0\ 0\ \bar{1})_{\text{two}}$ leads to the computation of x^{15} using three squarings and one division. Taking advantage of the factorization $15 = 3 \times 5$ leads to three squarings and two multiplications, provided the value of x^3 can be stored in a temporary register:

$$w = x^3 = (x)^2x \quad \text{and} \quad x^{15} = \big(((w)^2)^2\big)w$$

For $y = dq + s$, we can write:

$$w = x^y = x^s(x^d)^q$$

Thus, if we compute x^d in an extra register z and initialize w to x^s, the problem is converted to computing z^q. Details of this divide-and-conquer scheme are given elsewhere [Walt98].

23.4 DIVISION AND SQUARE-ROOTING, AGAIN

In Chapter 16, we examined a convergence method based on multiplicative normalization for computing the quotient $q = z/d$. The digit-recurrence division schemes of Chapters 13–15, are essentially additive normalization methods, where the partial remainder s is made to converge to 0 as q converges z/d. CORDIC division also falls in the additive normalization category. At this point, it is instructive to examine a broader formulation of division via additive normalization.

Let z and d be the dividend and divisor, respectively. Then, the following recurrences compute the quotient $q = z/d$ and the remainder s:

$$\begin{aligned} s^{(i+1)} &= s^{(i)} - \gamma^{(i)} \times d && \text{Set } s^{(0)} = z \text{ and make } s^{(m)} \text{ converge to 0} \\ q^{(i+1)} &= q^{(i)} + \gamma^{(i)} && \text{Set } q^{(0)} = 0 \text{ and find } q = q^{(m)} \end{aligned}$$

The preceding formulation is quite general and can be tailored to form a wide array of useful, and not so useful, division schemes. For example, given integer operands z and d, we can choose $\gamma^{(i)}$ to be $+1$ or -1, depending on whether z and d have identical or opposing signs. The resulting algorithm, which is often assigned as an exercise to help novice programmers master the notion of loop, is too slow for general use. However, if z is in a very limited range, say $0 \leq z < 2d$ as in addition modulo d, this is the algorithm of choice.

Since $s^{(i)}$ becomes successively smaller as it converges to 0, a scaled version of the recurrences, where $s^{(i)}$ now stands for $s^{(i)}r^i$ and $q^{(i)}$ for $q^{(i)}r^i$, is often used. Assuming fractional dividend z and divisor $d (0 \leq z, d < 1)$ we have:

$$\begin{aligned} s^{(i+1)} &= rs^{(i)} - \gamma^{(i)} \times d && \text{Set } s^{(0)} = z \text{ and keep } s^{(i)} \text{ bounded} \\ q^{(i+1)} &= rq^{(i)} + \gamma^{(i)} && \text{Set } q^{(0)} = 0 \text{ and find } q^* = q^{(m)}r^{-m} \end{aligned}$$

Note, in particular, that in this general version of the division recurrence based on additive normalization, the term $\gamma^{(i)}$ does not have to be a quotient "digit"; rather, it can be any estimate for

$$r(r^{i-m}q - q^{(i)}) = r(r^i q^* - q^{(i)})$$

where $r^{-m}q$ is the true quotient q^*. If $\gamma^{(i)}$ is indeed the quotient digit q_{-i-1}, then the addition required to compute $rq^{(i)} + \gamma^{(i)}$ is simplified (it turns into concatenation). See [Erce94] for a thorough treatment of digit-recurrence algorithms for division and square-rooting.

As in the case of division, we have already seen three approaches to square-rooting. One approach, based on digit-recurrence (division-like) algorithms, was discussed in Section 21.2 (radix 2, restoring), Section 21.3 (radix 2, nonrestoring), and Section 21.4 (high radix). The second approach using convergence methods, including those based on Newton–Raphson iteration, was covered in Section 21.5. The third approach, based on CORDIC, was introduced in Section 22.5. Here, we will see still other convergence algorithms for square-rooting based on additive and multiplicative normalization.

An algorithm based on multiplicative normalization can be developed by noting that if z is multiplied by a sequence of values $(c^{(i)})^2$, chosen such that the product converges to 1, then z multiplied by the $c^{(i)}$ values converges to $\sqrt{z}$, since:

$$z\prod(c^{(i)})^2 \approx 1 \quad \Rightarrow \quad \prod c^{(i)} \approx 1/\sqrt{z} \quad \Rightarrow \quad z\prod c^{(i)} \approx \sqrt{z}$$

So, one can initialize $x^{(0)}$ and $y^{(0)}$ to z and use the following iterations:

$$\begin{aligned} x^{(i+1)} &= x^{(i)}(1 + d_i 2^{-i})^2 = x^{(i)}(1 + 2d_i 2^{-i} + d_i^2\, 2^{-2i}) \\ y^{(i+1)} &= y^{(i)}(1 + d_i 2^{-i}) \end{aligned}$$

Devising rules for selecting d_i from the set $\{-1, 0, 1\}$ completes the algorithm. Basically, $d_i = 1$ is selected for $x^{(i)} < 1 - \varepsilon$ and $d_i = -1$ is selected for $x^{(i)} > 1 + \varepsilon$, where $\varepsilon = \alpha 2^{-i}$ is suitably picked to guarantee convergence. To avoid different comparison constants in different steps, $x^{(i)}$ is replaced by its scaled form $u^{(i)} = 2^i(x^{(i)} - 1)$, leading to the iterations:

$$\begin{aligned} u^{(i+1)} &= 2(u^{(i)} + 2d_i) + 2^{-i+1}(2d_i u^{(i)} + d_i^2) + 2^{-2i+1} d_i^2 u^{(i)} \\ y^{(i+1)} &= y^{(i)}(1 + d_i 2^{-i}) \end{aligned}$$

Then, selection of d_i in each step will be based on uniform comparisons with $\pm\alpha$. The radix-4 version of this square-rooting algorithm, with d_i in $[-2, 2]$, or equivalently in $\{-1, -1/2, 0, 1/2, 1\}$, has also been proposed and analyzed. The radix-4 algorithm requires comparison constants $\pm\alpha$ and $\pm\beta$. For details of the radix-2 and radix-4 algorithms, including the choice of the comparison constants, the reader is referred to [Omon94, pp. 380–385].

Similarly, an algorithm based on additive normalization uses the property that if a sequence of values $c^{(i)}$ can be obtained with $z - (\sum c^{(i)})^2$ converging to 0, then $\sqrt{z}$ is approximated by $\sum c^{(i)}$. Letting $c^{(i)} = -d_i 2^{-i}$ with d_i in $\{-1, 0, 1\}$, we derive:

$$\begin{aligned} x^{(i+1)} &= z - (y^{(i+1)})^2 = z - (y^{(i)} + c^{(i)})^2 \\ &= x^{(i)} + 2d_i y^{(i)} 2^{-i} - d_i^2 2^{-2i} \\ y^{(i+1)} &= y^{(i)} + c^{(i)} = y^{(i)} - d_i 2^{-i} \end{aligned}$$

Initial values for this algorithm are $x^{(0)} = z$ and $y^{(0)} = 0$. The choice of the d_i digit in $\{-1, 0, 1\}$ must ensure that $|x|$ is reduced in every step. Comparison with the constants $\pm\alpha 2^{-i}$ is one way to ensure convergence. As usual, to make the comparison constants the same for all steps, we rewrite $x^{(i)}$ as $2^{-i}u^{(i)}$, leading to:

$$u^{(i+1)} = 2(u^{(i)} + 2d_i y^{(i)} - d_i^2 2^{-i})$$
$$y^{(i+1)} = y^{(i)} - d_i 2^{-i}$$

Selection of the digit d_i in each step is then based on uniform comparison with $\pm\alpha$. Again, speed can be gained by using the radix-4 version of this algorithm, with d_i in $[-2, 2]$, or equivalently in $\{-1, -1/2, 0, 1/2, 1\}$. For details of both the radix-2 and the radix-4 algorithms, including a discussion of their convergence and choice of the required comparison constants, see [Omon94, pp. 385–389].

23.5 USE OF APPROXIMATING FUNCTIONS

The problem of evaluating a given function f can be converted to that of evaluating a different function g that approximates f, perhaps with a small number of pre- and postprocessing operations to bring the operands within appropriate ranges for g, to scale the results, or to minimize the effects of computational errors.

Since polynomial evaluation involves only additions and multiplications, the use of approximating polynomials can lead to efficient computations when a fast multiplier is available. Polynomial approximations can be obtained based on various schemes (e.g., Taylor–Maclaurin series expansion).

The Taylor series expansion of $f(x)$ about $x = a$ is

$$f(x) = \sum_{j=0}^{\infty} f^{(j)}(a) \frac{(x-a)^j}{j!}$$

The error that results from omitting all terms of degree greater than m is:

$$f^{(m+1)}(a + \mu(x-a)) \frac{(x-a)^{m+1}}{(m+1)!} \qquad 0 < \mu < 1$$

Setting $a = 0$ yields the Maclaurin–series expansion

$$f(x) = \sum_{j=0}^{\infty} f^{(j)}(0) \frac{x^j}{j!}$$

and its corresponding error bound:

$$f^{(m+1)}(\mu x) \frac{x^{m+1}}{(m+1)!} \qquad 0 < \mu < 1$$

Table 23.1 shows approximating polynomials, obtained from Taylor–Maclaurin series expansions, for some functions of interest. Others can be easily derived or looked up in standard mathematical handbooks.

The particular polynomial chosen affects the number of terms to be included for a given precision and thus the computational complexity. For example, if $\ln x$ is to be computed where x is fairly close to 1, the polynomial given in Table 23.1 in terms of $y = 1 - x$, which is the Maclaurin series expansion of $\ln(1 - y)$, converges rapidly and constitutes a good approximating function for $\ln x$. However, if $x \approx 2$, say, we have $y \approx -1$. A very large number of terms must be included to get $\ln x$ with about 32 bits of precision. In this latter case, the expansion in terms of $z = (x - 1)/(x + 1)$, which is derived from the Maclaurin series for $\ln[(1 + z)/(1 - z)]$, is much more efficient, since $z = (x - 1)/(x + 1) \approx 1/3$.

Evaluating an mth-degree polynomial may appear to be quite difficult. However, we can use Horner's method

$$\begin{aligned} f(y) &= c^{(m)}y^m + c^{(m-1)}y^{m-1} + \cdots + c^{(1)}y + c^{(0)} \\ &= ((c^{(m)}y + c^{(m-1)})y + \cdots + c^{(1)})y + c^{(0)} \end{aligned}$$

to efficiently evaluate an mth-degree polynomial by means of m multiply-add steps. The coefficients $c^{(i)}$ for some of the approximating polynomials in Table 23.1 are relatively simple functions of i that can be stored in tables or computed on the fly [e.g., $1/(2i + 1)$ for $\ln x$ or $\tanh^{-1} x$]. For other polynomials, the coefficients are more complicated but can be incrementally evaluated based on previously computed values: for example, $c^{(i)} = c^{(i-1)}/[2i(2i + 1)]$ for $\sin x$ or $\sinh x$.

A divide-and-conquer strategy, similar to that used for synthesizing larger multipliers from smaller ones (see Section 12.1), can be used for general function evaluation. Let x in $[0, 4)$ be the $(l + 2)$-bit significand of a floating-point number or its shifted version. Divide x into two chunks x_H and x_L (the high and low parts):

TABLE 23.1
Polynomial approximations for some useful functions

Function	Polynomial approximation	Conditions
$1/x$	$1 + y + y^2 + y^3 + \cdots + y^i + \cdots$	$0 < x < 2$ and $y = 1 - x$
$\sqrt{x}$	$1 - \frac{1}{2}y - \frac{1}{2\times4}y^2 - \frac{1\times3}{2\times4\times6}y^3 - \cdots - \frac{1\times3\times5\times\cdots\times(2i-3)}{2\times4\times6\times\cdots\times2i}y^i - \cdots$	$y = 1 - x$
e^x	$1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \cdots + \frac{1}{i!}x^i + \cdots$	
$\ln x$	$-y - \frac{1}{2}y^2 - \frac{1}{3}y^3 - \frac{1}{4}y^4 - \cdots - \frac{1}{i}y^i - \cdots$	$0 < x \leq 2$ and $y = 1 - x$
$\ln x$	$2\left(z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \cdots + \frac{1}{2i+1}z^{2i+1} + \cdots\right)$	$x > 0$ and $z = (x - 1)/(x + 1)$
$\sin x$	$x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots + (-1)^i\frac{1}{(2i+1)!}x^{2i+1} + \cdots$	
$\cos x$	$1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots + (-1)^i\frac{1}{(2i)!}x^{2i} + \cdots$	
$\tan^{-1} x$	$x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots + (-1)^i\frac{1}{2i+1}x^{2i+1} + \cdots$	$-1 < x < 1$
$\sinh x$	$x + \frac{1}{3!}x^3 + \frac{1}{5!}x^5 + \frac{1}{7!}x^7 + \cdots + \frac{1}{(2i+1)!}x^{2i+1} + \cdots$	
$\cosh x$	$1 + \frac{1}{2!}x^2 + \frac{1}{4!}x^4 + \frac{1}{6!}x^6 + \cdots + \frac{1}{(2i)!}x^{2i} + \cdots$	
$\tanh^{-1} x$	$x + \frac{1}{3}x^3 + \frac{1}{5}x^5 + \frac{1}{7}x^7 + \cdots + \frac{1}{2i+1}x^{2i+1} + \cdots$	$-1 < x < 1$

$$x = x_H + 2^{-t}x_L \qquad \underset{t+2 \text{ bits}}{0 \le x_H < 4} \qquad \underset{l-t \text{ bits}}{0 \le x_L < 1}$$

The Taylor series expansion of $f(x)$ about $x = x_H$ is

$$f(x) = \sum_{j=0}^{\infty} f^{(j)}(x_H) \frac{(2^{-t}x_L)^j}{j!}$$

where $f^{(j)}(x)$ is the jth derivative of $f(x)$, with the 0th derivative being $f(x)$ itself. If one takes just the first two terms, a linear approximation is obtained

$$f(x) \approx f(x_H) + 2^{-t}x_L f'(x_H)$$

In practice, only a few terms are needed, since as j becomes large, $2^{-jt}/j!$ rapidly diminishes in magnitude. If t is not too large, the evaluation of f and/or f' (as well as subsequent derivatives of f, if needed) can be done by table lookup. Examples of such table-based methods are presented in Chapter 24.

Functions can be approximated in many other ways (e.g., by the ratio of two polynomials with suitably chosen coefficients). For example, it has been suggested that good results can be obtained for many elementary functions if we approximate them using the ratio of two fifth-degree polynomials [Kore90]:

$$f(x) \approx \frac{a^{(5)}x^5 + a^{(4)}x^4 + a^{(3)}x^3 + a^{(2)}x^2 + a^{(1)}x + a^{(0)}}{b^{(5)}x^5 + b^{(4)}x^4 + b^{(3)}x^3 + b^{(2)}x^2 + b^{(1)}x + b^{(0)}}$$

When Horner's method for evaluating the numerator and the denominator is used, such a "rational approximation" needs 10 multiplications, 10 additions, and 1 division.

23.6 MERGED ARITHMETIC

The methods we have discussed thus far are based on building-block operations such as addition, multiplication, and shifting. When very high performance is needed, it is sometimes desirable, or even necessary, to build hardware structures to compute the function of interest directly without breaking it down into conventional operations. This "merged arithmetic" approach [Swar80] always leads to higher speed and often implies lower component count and power consumption as well. The drawback of starting from scratch is that designing, implementing, and testing of the corresponding algorithms and hardware structures may become difficult and thus more costly.

We have already seen several examples of merged arithmetic in the construction of additive multiply modules of Section 12.2 and combined multiply-add units of Section 12.6. In particular, Figs. 12.4 and 12.19 show how the required composite operations are synthesized at the bit level rather than through the use of standard word-level arithmetic building blocks.

Here, we illustrate the power of merged arithmetic through an additional example. Suppose that the inner product of two three-element vectors must be computed and the result added to an initial value. The computation, written as

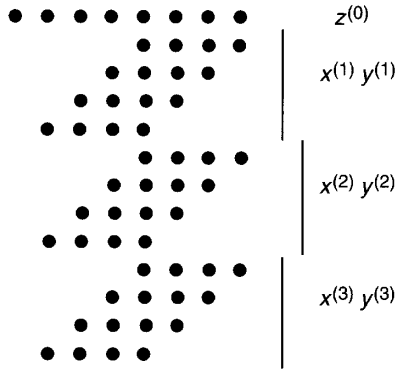


Fig. 23.2 Merged arithmetic computation of an inner product followed by accumulation.

	1	4	7	10	13	10	7	4	16 FAs
	2	4	6	8	8	6	4	2	10 FAs + 1 HA
	3	4	4	6	6	3	3	1	9 FAs
1	2	3	4	4	3	2	1	1	4 FAs + 1 HA
1	3	2	3	3	2	1	1	1	3 FAs + 2 HAs
2	2	2	2	2	1	1	1	1	5-bit CPA

Fig. 23.3 Tabular representation of the dot matrix for inner-product computation and its reduction.

$$z = z^{(0)} + x^{(1)}y^{(1)} + x^{(2)}y^{(2)} + x^{(3)}y^{(3)}$$

involves three multiplications and three additions if broken down into conventional word-level operations. However, one can also compute the result directly as a function of the seven operands ($8k$ Boolean variables for k-bit vector elements and a $2k$-bit $z^{(0)}$), provided the partial results $x^{(1)}y^{(1)}$, $x^{(2)}y^{(2)}$, and $x^{(3)}y^{(3)}$ are not needed for other purposes.

Figure 23.2 shows the computation in dot notation if $x^{(i)}$ and $y^{(i)}$ are 4-bit unsigned numbers and $z^{(0)}$ is an 8-bit unsigned number. This matrix of partial products, or dots, can be reduced using the methods discussed for the design of tree multipliers (e.g., by using the Wallace or the Dadda method). Figure 23.3 is a tabular representation of the reduction process for our example. The numbers in the first row are obtained by counting the number of dots in each column of Fig. 23.2. Subsequent rows are obtained by Wallace's reduction method.

The critical path of the resulting merged arithmetic circuit goes through one 2-input AND gate, 5 full adders, and a 5-bit carry-propagate adder: the cost is 48 AND gates, 46 FAs, 4 HAs, and a 5-bit adder—considerably less than the corresponding parameters if three separate 4×4 multipliers were implemented and their results added to the 8-bit input $z^{(0)}$.

PROBLEMS

23.1 Alternate view of convergence algorithms Given a function $z = f(x)$, a convergence algorithm for evaluating $c = f(a)$ can be constructed based on the following observations. Suppose we introduce an additional variable y and a convergence function $F(x, y)$ with the following three properties: (1) there is a known initiation value $y = b$ such that $F(a, b) = f(a)$; (2) a given pair of values $(x^{(i)}, y^{(i)})$ can be conveniently transformed to

the new pair $(x^{(i+1)}, y^{(i+1)})$ such that $F(x^{(i)}, y^{(i)}) = F(x^{(i+1)}, y^{(i+1)})$; that is, the value of F is invariant under the transformation; and (3) there exists a constant d, such that $F(d, y) = y$ for all y. Thus, if we make x converge to d, y will converge to $c = f(a)$, given the invariance of $F(x, y)$ under the transformation [Chen72].

a. Provide a geometric interpretation of the process above in the three-dimensional xyz space. *Hint:* Use the $x = a$, $y = b$, and $z = c$ planes.

b. Show that the convergence function $F(x, y) = y/\sqrt{x}$ can be used to compute $f(x) = \sqrt{x}$ and derive the needed transformations $x^{(i+1)} = \varphi(x^{(i)}, y^{(i)})$ and $y^{(i+1)} = \psi(x^{(i)}, y^{(i)})$.

c. Repeat part b for $F(x, y) = y + \ln x$ and $f(x) = \ln x$.

d. Repeat part b for $F(x, y) = ye^x$ and $f(x) = e^x$.

e. Derive $F(x, y)$ and its associated transformation rules for computing the reciprocal function $f(x) = 1/x$.

23.2 Computing natural logarithms

a. Compute ln 2 with 8 bits of precision using the radix-2 convergence algorithm based on multiplicative normalization given at the beginning of Section 23.2.

b. Repeat part a using a radix-4 version of the algorithm.

c. Repeat part a using the method based on squaring discussed near the end of Section 23.2. *Hint:* $\ln 2 = 1/\log_2 e$.

d. Compare the results of parts a–c and discuss.

23.3 Computing base-2 logarithms Compute the base-2 logarithm of $x = (1.0110\ 1101)_{\text{two}}$ with 8 bits of precision using:

a. Radix-2 convergence algorithm based on multiplicative normalization given at the beginning of Section 23.2.

b. Radix-4 version of the algorithm of part a.

c. The method based on squaring discussed near the end of Section 23.2.

23.4 Computing base-2 logarithms Here is an alternate method for computing $\log_2 x$ [Kost91]. A temporary variable y is initialized to x. For decreasing values of an index i, each time y is compared to 2^{2^i}. If y is greater than 2^{2^i}, the next digit of the logarithm is 1, and y is multiplied by 2^{-2^i}. Otherwise, the next digit is 0 and nothing is done.

a. Show that the algorithm is correct as described.

b. Use the algorithm to compute the base-2 logarithm of $x = (1.0110\ 1101)_{\text{two}}$.

c. Compare this new algorithm to radix-2 and radix-4 convergence methods, and to the method based on squaring (Section 23.2), with respect to speed and cost.

d. Can you generalize the algorithm to base-2^a logarithms? What about generalization to an arbitrary base b?

23.5 Computing the exponential function Compute $e^{0.5}$ with 8 bits of precision using:

a. Radix-2 convergence algorithm based on additive normalization given at the beginning of Section 23.3.

b. Radix-4 version of the algorithm of part a.

c. A convergence algorithm for square-rooting that you choose at will.

d. Compare the results of parts a–c and discuss.

23.6 Exponentiation Assuming that shift-and-add takes 1 time unit, multiplication 3 time units, and division 8 time units:

a. Devise an efficient algorithm for computing x^{30} using the method discussed near the end of Section 23.3.

b. Use the algorithm of part a to compute 0.99^{30}, with all intermediate values and results carrying eight fractional digits in radix 10.

c. Use the convergence algorithm of Section 23.3 to compute 0.99^{30}.

d. Compare the accuracy of the results and the computational complexity for the algorithms of parts b and c. Discuss.

23.7 Modular exponentiation Modular exponentiation—namely, the computation of $x^y \bmod m$, where x, y, and m are k-bit integers, k is potentially very large, and m is a prime number—plays an important role in some public-key cryptography.

a. Show how $x^y \bmod m$ can be computed using k-bit arithmetic operations.

b. Show how the algorithm can be speeded up if Booth's recoding is used on y.

c. Can radix-4 modified Booth's recoding of the exponent lead to further speedup?

23.8 Logarithmic multiplication/division Discuss the feasibility of performing multiplication or division by computing the natural logarithms of the operands, performing an add/subtract operation, and finally computing the exponential function.

23.9 Convergence division and reciprocation

a. Consider the problem of computing $q = z/d$, where $1 \leq z, d < 2$ and $1/2 < q < 2$, using a strategy similar to the binary search algorithm. The midpoint of [0.5, 2] (viz., 1.25) is taken as an initial estimate for q. Multiplication and comparison then allow us to refine the interval containing q to [0.5,1.25] or [1.25,2]. This refinement process continues until the interval is as narrow as the desired precision for q. Compare the preceding convergence method to other convergence division algorithms and discuss.

b. Devise an algorithm similar to that in part a for computing $1/d$ that uses interpolation for identifying the next point, instead of always taking the midpoint of the interval.

23.10 Computing the generalized square-root function Show that the following convergence computation scheme can lead to the computation of the generalized square-root function $\sqrt{x + y^2}$, provided $d_i = \text{sign}(x^{(i)} y^{(i)})$.

$$x^{(i+1)} = x^{(i)} - 2d_i 2^{-i} y^{(i)} - d_i^2 2^{-2i}$$

$$y^{(i+1)} = y^{(i)} + d_i 2^{-i}$$

23.11 Convergence algorithm for square-rooting In discussing the radix-4 convergence algorithm for square-rooting near the end of Section 23.4, we stated that the root digit set can be $[-2, 2]$ or $\{-1, -1/2, 0, 1/2, 1\}$. Discuss possible advantages of the latter digit set over the former and devise an algorithm for converting such a radix-4 number to standard binary.

23.12 Approximating functions

a. The polynomial approximation for $\tan^{-1} x$ given in Section 23.5 (Table 23.1) is valid only for $x^2 < 1$. Show how this approximation can be used within an algorithm to evaluate $\tan^{-1} x$ for all x. *Hint:* For $x^2 > 1$, $y = 1/x$ satisfies $y^2 < 1$.

b. When $|x|$ is close to 1, the preceding approximation converges slowly. How can one speed up the computation via the application of suitable pre- and postprocessing steps? *Hint:* $\tan(2x) = 2 \tan x/(1 - \tan^2 x)$.

c. Repeat part b for the function $\tanh^{-1} x$.

23.13 Approximating functions Derive approximating functions for $\sin^{-1} x$, $\cos^{-1} x$, $\sinh^{-1} x$, $\cosh^{-1} x$ based on Taylor–Maclaurin series expansions and compare the effort required for their evaluation with those based on indirect methods such as $\sin^{-1} x = \tan^{-1}(x/\sqrt{1 - x^2})$.

23.14 Approximating functions For each of the functions $f(x)$ below, use the approximating polynomial given in Table 23.1 and a convergence computation method of your choice to compute $f(0.75)$ to four decimal digits of precision. Compare the computational efforts expended and the results obtained. Discuss.

a. $1/x$

b. $\sqrt{x}$

c. e^x

d. $\ln x$

e. $\sin x$

f. $\tan^{-1} x$

g. $\sinh x$

23.15 Merged arithmetic operations Consider the computation $s = vw + xy + z$, where v, w, x, and y are k-bit integers and z is a $2k$-bit integer (all numbers are in 2's-complement format).

a. Prove that s can be represented correctly using $2k + 1$ bits.

b. Assuming $k = 4$, draw the partial products matrix for the entire computation in dot notation; 16 dots for each of the two multiplications and 8 dots for z, plus additional dots as required to take care of signed multiplication using the (modified) Baugh–Wooley method of Fig. 11.8d.

c. Use Wallace's method to reduce the matrix of dots in part b to only two rows.

d. Use Dadda's method to reduce the matrix of dots in part b to only two rows.

e. Derive the lengths of the final carry-propagate adders required in parts c and d.

f. Compare the design of part c, with regard to delay and cost, to a design based on two 4×4 multipliers (separately designed using the Baugh–Wooley and Wallace methods), a single level of carry-save addition, and a final fast adder.

g. Repeat part f, replacing Wallace's method with Dadda's method.

h. Summarize the delay–cost comparisons of parts f and g in a table and discuss.

i. Simplify the circuit of part d if it is to perform the computation $s = v^2 + x^2 + z$.

23.16 Merged arithmetic/logic operations Arithmetic operations can sometimes be merged with nonarithmetic functions to derive speed benefits. One example is merging the addition required for computing a cache memory address with the address decoding function in the cache [Lync98].

a. Consider a small example of two 4-bit unsigned values added to find a 4-bit memory address and design the merged adder/decoder circuit.

b. Compare the delay and cost of the design in part a to the respective parameters of a design with separate adder and decoder. Discuss.

REFERENCES

[Chen72] Chen, T.C., "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots," *IBM J. Research and Development*, Vol. 16, pp. 380–388, 1972.

[Erce73] Ercegovac, M.D., "Radix-16 Evaluation of Certain Elementary Functions," *IEEE Trans. Computers*, Vol. 22, No. 6, pp. 561–566, 1973.

[Erce94] Ercegovac, M.D., and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.

[Kore90] Koren, I., and O. Zinaty, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Trans. Computers*, Vol. 39, No. 8, pp. 1030–1037, 1990.

[Kost91] Kostopoulos, D.K., "An Algorithm for the Computation of Binary Logarithms," *IEEE Trans. Computers*, Vol. 40, No. 11, pp. 1267–1270, 1991.

[Lo87] Lo, H.-Y., and J.-L. Chen, "A Hardwired Generalized Algorithm for Generating the Logarithm Base-k by Iteration," *IEEE Trans. Computers*, Vol. 36, No. 11, pp. 1363–1367, 1987.

[Lync98] Lynch, W.L., G. Lauterbach, and J.I. Chamdani, "Low Load Latency Through Sum-Addressed Memory," *Proc. Int. Symp. Computer Architecture*, 1998, pp. 369–379.

[Omon94] Omondi, A.R., *Computer Arithmetic Systems: Algorithms, Architecture, and Implementations*, Prentice-Hall, 1994.

[Swar80] Swartzlander, E.E., Jr., "Merged Arithmetic," *IEEE Trans. Computers*, Vol. 29, No. 10, pp. 946–950, 1980.

[Tang91] Tang, P.K.P., "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th Symp. Computer Arithmetic*, 1991, pp. 232–236.

[Walt98] Walter, C. D., "Exponentiation Using Division Chains," *IEEE Trans. Computers*, Vol. 47, No. 7, pp. 757–765, 1998.

Chapter 24 ARITHMETIC BY TABLE LOOKUP

In earlier chapters we saw how table lookup can be used as an aid in arithmetic computations. Examples include quotient digit selection in high-radix division, speedup of iterative division or reciprocation through an initial table-lookup step, and using tables to store constants of interest in CORDIC. In this chapter, we deal with the use of table lookup as a primary computational mechanism rather than in a supporting role.

24.1 DIRECT AND INDIRECT TABLE LOOKUP

Computation by table lookup is attractive because memory is much denser than random logic in VLSI realizations. Multimegabit lookup tables are already practical in some applications; even larger tables should become practical in the near future as memory densities continue to improve. The use of tables reduces the costs of hardware development (design, validation, testing), provides more flexibility for last-minute design changes, and reduces the number of different building blocks or modules required for arithmetic system design.

Tables stored in read-only memories (especially if individual entries or blocks of data are encoded in error-detecting or error-correcting codes) are more robust than combinational logic circuits, thus leading to improved reliability. With read/write memory and reconfigurable peripheral logic, the same building block can be used for evaluating many different functions by simply loading appropriate values in the table(s). This feature facilitates maintenance and repair.

Given an m-variable function $f(x_{m-1}, x_{m-2}, \cdots, x_1, x_0)$, the *direct table-lookup* evaluation of f requires the construction of a $2^u \times v$ table that holds for each combination of input values (needing a total of u bits to represent), the desired v-bit result. The u-bit string obtained from concatenating the input values is then used as an address into the table, with the v-bit value read out from the table directly forwarded to the output. Such an arrangement is quite flexible

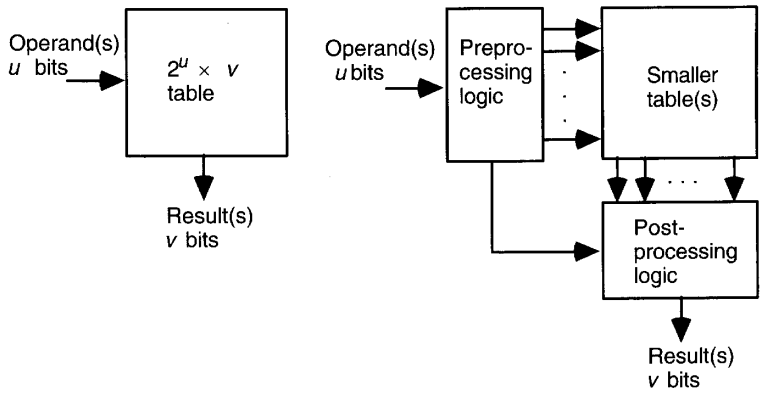


Fig. 24.1 Direct table lookup versus table-lookup with pre- and post-processing.

but unfortunately not very practical in most cases. For unary (single-variable) functions such as $1/x$, $\ln x$, or x^2, the table size remains manageable when the input operand is up to 12–16 bits; table size of 4K–16K words. Binary functions, such as xy, $x \bmod y$, or x^y, can be realized with table lookup only if the operands are very short (8 bits or less, say). For $m > 2$, the exponential growth of the table size becomes totally intolerable.

One solution to the exponential growth of the table size is to apply preprocessing steps to the operands and postprocessing steps to the value(s) read out from the table(s), leading to *indirect table lookup*. If both the pre- and postprocessing elements are simple and fast, this hybrid scheme (Fig. 24.1) may be more cost-effective than either the pure table-lookup approach or the pure logic circuit implementation based on the algorithms discussed in earlier chapters. In a multitable scheme, the tables can be physically separate (with identical or different contents) or realized by multiple accesses to the same table. We explore some such hybrid schemes in the rest of this chapter.

As stated earlier, in contrast to the applications discussed already, in which small tables were used for quotient digit selection, initial approximations, or storage of a few precomputed constants, our focus in this chapter is on the use of tables as primary computational mechanisms.

In reality, the boundary between the two uses of tables (in supporting or primary role) is quite fuzzy. We can visualize the pure logic and pure tabular approaches as extreme points in a continuum of hybrid solutions. In earlier discussions, we started with the goal of designing logic circuits for particular arithmetic computations and ended up using tables to facilitate or speed up certain computational steps. Here, we begin with the goal of a tabular implementation and finish by using peripheral logic circuits to reduce the table size, thus making the approach practical. Some of the intermediate solutions can be derived starting at either end point.

24.2 BINARY-TO-UNARY REDUCTION

One approach to reducing the table size is to evaluate a desired binary function by means of an auxiliary unary function. The unary function requires a smaller table (2^k vs. 2^{2k} entries, say), but its output obviously is not what we are after. However, pre- and postprocessing steps allow us to use the unary function table to compute our binary function. In this section, we review two well-known examples of this method.

We discussed an example of this approach in connection with logarithmic number systems in Section 18.6 To add the sign-and-logarithm numbers (Sx, Lx) and (Sy, Ly), representing $\pm x$ and $\pm y$ with $x \geq y \geq 0$, we need to compute the sign Sz of the result $\pm z$ and its logarithm $Lz = \log z = \log(x \pm y)$. The base of the logarithm is immaterial for this discussion, so we leave it unspecified. The computation of Lz can be transformed to finding the sum of Lx and a unary function of $\Delta = Ly - Lx$ using the following equality

$$\begin{aligned} Lz &= \log(x \pm y) = \log[x(1 \pm y/x)] \\ &= \log x + \log(1 \pm y/x) \\ &= Lx + \log(1 \pm \log^{-1} \Delta) \end{aligned}$$

where $\log^{-1} \Delta$ denotes the inverse logarithm function; that is, b^{Δ} if the base of the logarithm is b.

The required preprocessing steps involve identifying the input $\pm x$ with the larger logarithm (and thus the larger magnitude), determining the sign Sz of the result, and computing $\Delta = Ly - Lx$. Postprocessing consists of adding Lx to the value read out from the table. If the preprocessing, table access, and postprocessing steps are done by distinct hardware elements, a pipelined implementation may be possible for which the cycle time is dictated by the table access time. So, with many additions performed in sequence, the preceding scheme can be as fast as a pure tabular realization and thus considerably more cost-effective.

Our second example concerns multiplication by table lookup. Again, direct table lookup is infeasible in most practical cases. The following identity allows us to convert the problem to the evaluation of a unary function (in this case, squaring):

$$xy = \frac{1}{4}[(x + y)^2 - (x - y)^2]$$

The preprocessing steps consist of computing $x + y$ and $x - y$. Then, after two table lookups yielding $(x + y)^2$ and $(x - y)^2$, a subtraction and a 2-bit shift complete the computation. Again, pipelining can be used to reduce the time overhead of the peripheral logic. Several optimizations are possible for the preceding hybrid solution. For example, if a lower speed is acceptable, one squaring table can be used and consulted twice for finding $(x + y)^2$ and $(x - y)^2$. This would allow us to share the adder/subtractor hardware as well.

In either case, the following observation leads to hardware simplifications. Let x and y be k-bit 2's-complement integers (the same considerations apply to any fixed-point format). Then, $x + y$ and $x - y$ are $(k + 1)$-bit values, and a straightforward application of the preceding method would need one or two tables of size $2^{k+1} \times 2k$ (sign bit is not needed for table entries, since they are all positive). Closer scrutiny, however, reveals that $x + y$ and $x - y$ are both even or odd. Thus, the least significant two bits of $(x + y)^2$ and $(x - y)^2$ are identical (both are 00 or 01). Hence, these two bits always cancel each other out, with the resulting 0s shifted out in the final division by 4, and need not be stored in the tables. This feature reduces the required table size to $2^{k+1} \times (2k - 2)$ and eliminates the 2-bit shift.

The aforementioned reduction in table size is relatively insignificant, but it is achieved at no cost (in fact it improves the speed by eliminating the final shift step). A more significant factor-of-2 reduction in table size can be achieved with some peripheral overhead. Let ε denote the least significant bit of $x + y$ and $x - y$, where $\varepsilon \in \{0, 1\}$. Then:

$$\frac{x+y}{2} = \left\lfloor \frac{x+y}{2} \right\rfloor + \frac{\varepsilon}{2}$$
$$\frac{x-y}{2} = \left\lfloor \frac{x-y}{2} \right\rfloor + \frac{\varepsilon}{2}$$

Then, we can write:

$$\frac{1}{4}[(x+y)^2 - (x-y)^2] = \left(\left\lfloor \frac{x+y}{2} \right\rfloor + \frac{\varepsilon}{2} \right)^2 - \left(\left\lfloor \frac{x-y}{2} \right\rfloor + \frac{\varepsilon}{2} \right)^2$$
$$= \left\lfloor \frac{x+y}{2} \right\rfloor^2 - \left\lfloor \frac{x-y}{2} \right\rfloor^2 + \varepsilon y$$

Based on the preceding equality, upon computing $x + y$ and $x - y$, we can drop the least significant bit of each result, consult squaring tables of size $2^k \times (2k - 1)$, and then perform a three-operand addition, with the third operand being 0 or y depending on the dropped bit ε being 0 or 1. The postprocessing hardware then requires a carry-save adder (to reduce the three values to two) followed by a carry-propagate adder.

To use a single adder and one squaring table to evaluate the preceding three-operand sum, we simply initialize the result to εy and then overlap the first addition $\lfloor (x + y)/2 \rfloor^2 + \varepsilon y$ with the second table access, thus essentially hiding the delay of the extra addition resulting from the introduction of the new εy term.

The preceding is an excellent example of the trade-offs that frequently exist between table size and cost/delay of the required peripheral logic circuits in hybrid implementations using a mix of lookup tables and custom logic.

When the product xy is to be rounded to a k-bit number (as for fractional operands), the entries of the squaring table(s) can be shortened to k bits (again no sign is needed). The extra bit guarantees that the total error remains below *ulp*.

An additional optimization may be applicable to some unary function tables. Assume that a v-bit result is to be computed based on a k-bit operand. Let w bits of the result ($w < v$) depend only on l bits of the operand ($l < k$). Then a split-table approach can be used, with one table of size $2^l w$ providing w bits of the result and another of size $2^k(v - w)$ supplying the remaining $v - w$ bits. The total table size is reduced to $2^k v - (2^k - 2^l)w$, with the fraction of table size saved being:

$$\frac{(2^k - 2^l)w}{2^k v} = \frac{(1 - 2^{k-l})w}{v}$$

Application of this last optimization to squaring leads to additional savings in the table size for multiplication via squaring [Vinn95].

24.3 TABLES IN BIT-SERIAL ARITHMETIC

The many advantages of bit-serial arithmetic were discussed in Section 12.3 in connection with bit-serial multipliers. Here, we discuss two examples of tabular implementation of bit-serial arithmetic that are used for entirely different reasons.

The first example is found in the processors of a massively parallel computer: the Connection Machine CM-2 of Thinking Machines Corporation. Even though CM-2 is no longer in production, its approach to bit-serial computation is quite interesting and potentially useful. CM-2 can have up to 64K processors, each one so simple that 16 processors fit on single IC chip. The processors are bit-serial because otherwise their parallel I/O and memory access requirements could not be satisfied within the pin limitations of a single chip. The design philosophy of CM-2 is that using a large number of slow, inexpensive processors is a cost-effective alternative to a small number of very fast, expensive processors. This is sometimes referred to as the "army of ants" approach to high-performance computing.

The ALU in a CM-2 processor receives three single-bit inputs and produces two single-bit outputs. For addition (e.g.), the three inputs can be the operand bits and the incoming carry, with the two outputs corresponding to the sum bit and the outgoing carry. To provide complete flexibility in programming other computations, CM-2 designers decided that the user should be able to specify each output of the ALU to be any arbitrary logic function of the three input bits. There are $2^{2^3} = 256$ such logic functions, leading to the requirement for an 8-bit op code. The remaining problem is how to encode the 256 functions within an 8-bit op code. The answer is strikingly simple: each of the 256 functions is completely characterized by its 8-bit truth table. So we can simply use the truth table for each function as the op code. Figure 24.2 shows the resulting ALU, which is nothing but two 8-to-1 multiplexers!

In the CM-2 ALU, two of the bit streams, say a and b, come from a 64K-bit memory and are read out in consecutive clock cycles. The third input, c, comes from a 4-bit "flags" register. Thus $16+16+2$ bits are required to specify the addresses of these operands. The f output is stored as a flag bit (2-bit address) and the g output replaces the a memory operand in a third clock cycle. Three more bits are used to specify a flag bit and a value (0 or 1) to conditionalize the operation, thus allowing some processors to selectively ignore the common instruction broadcast to all processors, but this aspect of the processor's design is not relevant to our discussion here.

To perform integer addition with the CM-2 ALU shown in Fig. 24.2, the a and b operands will correspond to the two numbers to be added, and c will be a flag bit that is used to hold the carry from one bit position into the next. The f function op code will be "00010111" (majority or $ab+bc+ca$) and the g function op code will be "01010101" (three-input XOR). A k-bit addition requires $3k$ clock cycles and is thus quite slow. But up to 64K additions can be performed in parallel. As for floating-point arithmetic, bit-serial computation (which was used in CM-1) is too slow. So, designers of CM-2 provided floating-point accelerator chips that are shared by 32 processors.

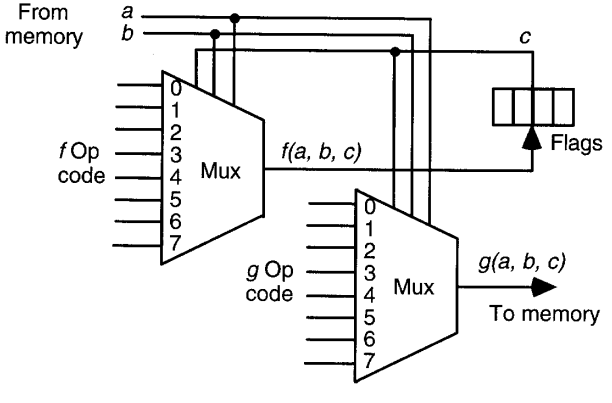


Fig. 24.2 Bit-serial ALU with two tables implemented as multiplexers.

Programming bit-serial arithmetic operations is a tedious and error-prone task. However, it is an easy matter to build useful "macros" that are made available to machine-language programmers of CM-2 and other bit-serial machines. These programmers then do not need to worry about coding the details of bit-serial arithmetic for such routine computations as integer addition, integer multiplication, or their floating-point counterparts. The use of bit-level instructions will then be required only for special operations or for hand-optimization of critical operations in the inner loops of computation-intensive algorithms.

Our second example concerns the implementation of a digital filter, but the method is applicable to computing any linear function of several variables. Consider a second-order digital filter characterized by the equation

$$y^{(i)} = a^{(0)}x^{(i)} + a^{(1)}x^{(i-1)} + a^{(2)}x^{(i-2)} - b^{(1)}y^{(i-1)} - b^{(2)}y^{(i-2)}$$

where the $a^{(j)}$s and $b^{(j)}$s are constants, $x^{(i)}$ is the filter input at time step i, and $y^{(i)}$ is the filter output at time step i. Such a filter is useful in itself and may also be a component in a more complex filter.

Expanding the equation for $y^{(i)}$ in terms of the individual bits of the 2's-complement operands $x = (x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two}}$ and $y = (y_0.y_{-1}y_{-2}\cdots y_{-l})_{\text{two}}$, we get:

$$\begin{aligned} y^{(i)} = {} & a^{(0)}\left(-x_0^{(i)} + \sum_{j=-l}^{-1} 2^j x_j^{(i)}\right) + a^{(1)}\left(-x_0^{(i-1)} + \sum_{j=-l}^{-1} 2^j x_j^{(i-1)}\right) \\ & + a^{(2)}\left(-x_0^{(i-2)} + \sum_{j=-l}^{-1} 2^j x_j^{(i-2)}\right) - b^{(1)}\left(-y_0^{(i-1)} + \sum_{j=-l}^{-1} 2^j x_j^{(i-1)}\right) \\ & - b^{(2)}\left(-y_0^{(i-2)} + \sum_{j=-l}^{-1} 2^j x_j^{i-2)}\right) \end{aligned}$$

Define $f(s,t,u,v,w) = a^{(0)}s + a^{(1)}t + a^{(2)}u - b^{(1)}v - b^{(2)}w$, where s, t, u, v, and w are single-bit variables. If the coefficients are m-bit constants, then each of the 32 possible values for f is representable in $m+3$ bits, as it is the sum of five m-bit operands. These 32 values can be precomputed and stored in a $32 \times (m+3)$-bit table.

Using the function f, we can rewrite the expression for $y^{(i)}$ as follows:

$$\begin{aligned} y^{(i)} = {} & \sum_{j=-l}^{-1} 2^j f\left(x_j^{(i)}, x_j^{(i-1)}, x_j^{(i-2)}, y_j^{(i-1)}, y_j^{(i-2)}\right) \\ & - f\left(x_0^{(i)}, x_0^{(i-1)}, x_0^{(i-2)}, y_0^{(i-1)}, y_0^{(i-2)}\right) \end{aligned}$$

Figure 24.3 shows a hardware unit for computing this last expression with bit-serial input and output. The value of $y^{(i)}$ is accumulated in the s register as $y^{(i-1)}$ is output from the output shift register. At the end of the cycle, the result in the s register is loaded into the output shift register, s is reset to 0, and a new accumulation cycle begins. The output bit $y_j^{(i-1)}$ is supplied to the ROM as an address bit. A second shift register at the output side supplies the corresponding bit $y_j^{(i-2)}$ of the preceding output. At the input side, $x^{(i)}$ is processed on the fly, and two shift registers are used to supply the corresponding bits of the two preceding inputs, $x^{(i-1)}$ and $x^{(i-2)}$, to the 32-entry table.

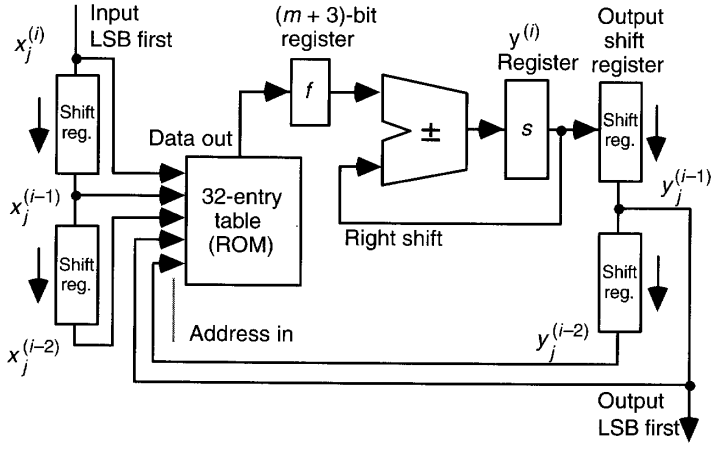


Fig. 24.3 Bit-serial tabular realization of a second-order filter.

Structures similar to that shown in Fig. 24.3 are useful for computing many other functions. For example, if $(x + y + z) \bmod m$ is to be computed for integer-valued operands x, y, z and a modulus m, then the residues of 2^i, 2×2^i, and 3×2^i can be stored in a table for different values of i. The bits x_i, y_i, z_i, and the index i are used to derive an address from which the value of $2^i(x_i + y_i + z_i) \bmod m$ is read out and added to the modulo-m running total.

24.4 INTERPOLATING MEMORY

If the value of a function $f(x)$ is known for $x = x_{\text{lo}}$ and $x = x_{\text{hi}}$, where $x_{\text{lo}} < x_{\text{hi}}$, the function's value for x in the interval $[x_{\text{lo}}, x_{\text{hi}}]$ can be computed from $f(x_{\text{lo}})$ and $f(x_{\text{hi}})$ by interpolation. The simplest method is linear interpolation where $f(x)$ for x in $[x_{\text{lo}}, x_{\text{hi}}]$ is computed as follows:

$$f(x) = f(x_{\text{lo}}) + \frac{(x - x_{\text{lo}})[f(x_{\text{hi}}) - f(x_{\text{lo}})]}{x_{\text{hi}} - x_{\text{lo}}}$$

On the surface, evaluating this expression requires four additions, one multiplication, and one division. However, by choosing the end points x_{lo} and x_{hi} to be consecutive multiples of a power of 2, the division and two of the additions can be reduced to trivial operations.

For example, suppose that $\log_2 x$ is to be evaluated for x in $[1, 2)$. Since $f(x_{\text{lo}}) = \log_2 1 = 0$ and $f(x_{\text{hi}}) = \log_2 2 = 1$, the linear interpolation formula becomes:

$$\log_2 x \approx x - 1 = \text{the fractional part of } x$$

The error in this extremely simple approximation is $\varepsilon = \log_2 x - x + 1$, which assumes its maximum absolute value of 0.086 071 for $x = \log_2 e = 1.442\,695$ and maximum relative value of 0.061 476 for $x = e/2 = 1.359\,141$. Errors this large are obviously unacceptable for useful computations, but before proceeding to make the approach more practical, let us note an improvement in the preceding linear interpolation scheme.

Instead of approximating the function $f(x)$ with a straight line between the two end points of $f(x)$ at x_{lo} and x_{hi}, one can use another straight line that minimizes the absolute or relative error in the worst case. Figure 24.4 depicts this strategy, along with the hardware structure needed for its realization. We now have errors at the two end points as well as elsewhere within the interval (x_{lo}, x_{hi}), but the maximum error has been reduced.

Applying the preceding strategy to computing $\log_2 x$ for x in [1, 2), we can easily derive the following straight-line approximation $a + b(x - 1) = a + b\Delta x$ for minimizing the absolute error (to 0.043 036 for $x = 1.0$, 1.442 695, or 2.0):

$$\log_2 x \approx \frac{\ln 2 - \ln(\ln 2) - 1}{2 \ln 2} + (x - 1) = 0.043\,036 + \Delta x$$

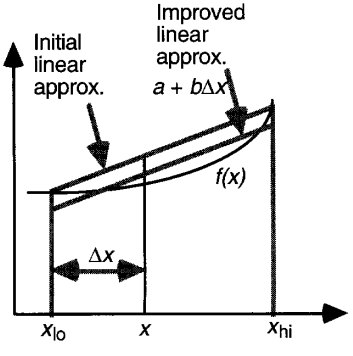
This is better than our first try (half the error), but still too coarse an approximation to be useful. The derivation of a straight line that minimizes the relative error in the worst case is similar but does not lead to closed-form results for a and b.

It appears that a single straight line won't do for the entire interval of interest and we need to apply the interpolation method in narrower intervals to obtain acceptable results. This observation leads to an "interpolating memory" [Noet89] that begins with table lookup to retrieve the coefficients $a^{(i)}$ and $b^{(i)}$ of the approximating straight line $a^{(i)} + b^{(i)}\Delta x$, given the index i of the subinterval containing x, and then uses one multiplication and one addition to complete the computation (Fig. 24.5). Note that since Δx begins with two 0s, it would be more efficient to use $4\Delta x$, which is representable with two fewer bits. The table entries $b^{(i)}$ must then be divided by 4 to keep the products the same.

Clearly, second-degree or higher-order interpolation can be used, an approach that involves more computation but yields correspondingly better approximations. For example, with second-degree interpolation, the coefficients $a^{(i)}$, $b^{(i)}$, and $c^{(i)}$ are read out from tables and the expression $a^{(i)} + b^{(i)}\Delta x + c^{(i)}\ \Delta x^2$ is evaluated using three multipliers and a three-operand adder. The multiplication (squaring) to obtain Δx^2 can be overlapped with table access to obtain better performance. Third- or higher-degree interpolation is also possible but often less cost-effective than simpler linear or quadratic schemes using narrower intervals.

If the number of subintervals is 2^h then the subinterval containing x can be determined by looking at the h most significant bits of x, with the offset Δx simply derived from the remaining bits of x. Since it is more efficient to deal with $2^h\Delta x$, which has h fewer bits than Δx, the tables must contain $a^{(i)}$, $b^{(i)}/2^h$, $c^{(i)}/2^{2h}$, etc.

Let us now apply the method of Fig. 24.5 with four subintervals to compute $\log_2 x$ for x in [1, 2). The four subintervals are [1.00, 1.25), [1.25, 1.50), [1.50, 1.75), and [1.75, 2.00).



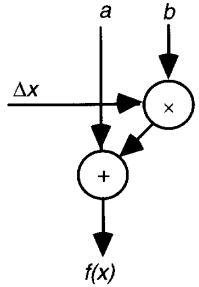


Fig. 24.4 Linear interpolation for computing $f(x)$ and its hardware realization.

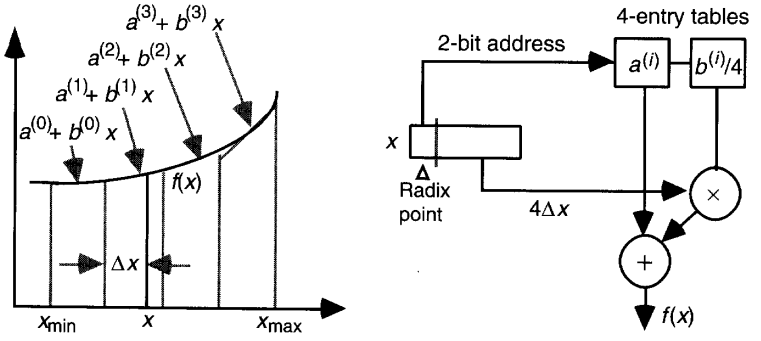


Fig. 24.5 Linear interpolation for computing $f(x)$ using four subintervals.

Table 24.1 lists the parameters of the best linear approximation, along with its worst-case error, for each subinterval.

We see from Table 24.1 that the maximum error is now much less than for simple linear interpolation. We can improve the quality of approximation even further by using more intervals (larger tables) or superlinear interpolation (more tables and peripheral arithmetic computations). The optimal choice will be different for each problem and must be determined by careful analysis based on a reasonably realistic cost model.

24.5 TRADE-OFFS IN COST, SPEED, AND ACCURACY

As noted in Section 24.4, trade-offs exist between table size and complexity/delay of peripheral circuits for a given precision. Generally, the higher the order of interpolation (more peripheral circuits or longer delay with hardware sharing), the smaller the number of subintervals needed to guarantee a given precision for the results (smaller tables). However, it is seldom cost-effective to go beyond second-degree interpolation.

As an example of such trade-offs, Fig. 24.6 shows the maximum absolute error in an interpolating memory unit computing $\log_2 x$ for various numbers h of address bits using mth-degree interpolation, with $m = 1, 2$, or 3. With these parameters, the total number of table entries is $(m + 1)2^h$.

Figure 24.6 can be used in two ways to implement an appropriate interpolating memory unit for evaluating $\log_2 x$. First, if the table size is limited by component availability or chip area to a

TABLE 24.1
Approximating $\log_2 x$ for x in [1, 2) using linear interpolation within 4 subintervals

i	x_{lo}	x_{hi}	$a^{(i)}$	$b^{(i)}/4$	Maximum error
0	1.00	1.25	0.004 487	0.321 928	±0.004 487
1	1.25	1.50	0.324 924	0.263 034	±0.002 996
2	1.50	1.75	0.587 105	0.222 392	±0.002 142
3	1.75	2.00	0.808 962	0.192 645	±0.001 607

total of 256 words, say, then 7 address bits can be used with linear, and 6 bits with either second- or third-degree interpolation. This leads to worst-case absolute errors of about 10^{-5}, 10^{-7}, and 10^{-10}, respectively. Of course if the table size is limited by chip area, then it is unlikely that the second- or third-order schemes can be implemented, since they require multiple adders and multipliers. So, we have an accuracy/speed trade-off to consider.

If a maximum tolerable error of 10^{-6}, say, is given, then Fig. 24.6 tells us that we can use linear interpolation with 9 address bits (two 512-entry tables), second-degree interpolation with 5 address bits (three 32-entry tables), or third-degree interpolation with 3 address bits (four 8-entry tables). Since 32-entry tables are already small enough, little is gained from using third-degree interpolation, which requires significantly more complex and slower peripheral logic.

Except for slight upward or downward shifting of the curves, the shapes of error curves for other functions of interest are quite similar to the ones for $\log_2 x$ shown in Fig. 24.6. In most cases, the number of address bits required for a given precision is within ± 1 of that needed for the $\log_2$ functions. This makes it practical to build a general-purpose interpolating memory unit that can be customized for various functions of interest by plugging in ROMs with appropriate contents or by dynamically loading its RAM tables.

24.6 PIECEWISE LOOKUP TABLES

Several practical methods for function evaluation are based on table lookup using fragments of the operands. These methods essentially fall between the two extremes of direct table lookup and the bit-serial methods discussed in Section 24.3. Here, we review two such methods as representative examples.

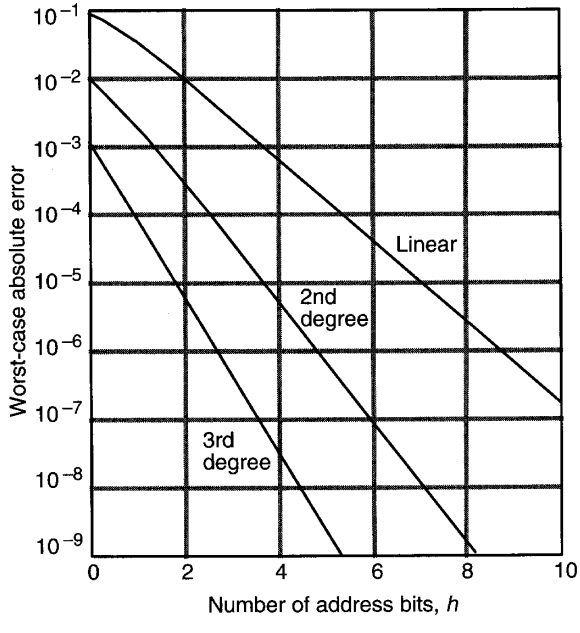


Fig. 24.6 Maximum absolute error in computing $\log_2 x$ as a function of number h of address bits for the tables with linear, quadratic (second-degree), and cubic (third-degree) interpolations [Noet89].

The first method deals with evaluating elementary functions in single-precision IEEE floating-point format. We ignore the sign and exponent in this brief discussion. For details of how the exponent affects the evaluation process, see [Wong95].

Let us divide the 26-bit significand x (with 2 whole and 24 fractional bits) into four sections:

$$x = t + \lambda u + \lambda^2 v + \lambda^3 w = t + 2^{-6}u + 2^{-12}v + 2^{-18}w$$

Each of the components u, v, and w is a 6-bit fraction in [0, 1) and t, with up to 8 bits depending on the function being evaluated, is in [0, 4). The Taylor polynomial for $f(x)$ is:

$$f(x) = \sum_{i=0}^{\infty} f^{(i)}(t + \lambda u)\frac{(\lambda^2 v + \lambda^3 w)^i}{i!}$$

The value of $f(x)$ can be approximated by ignoring terms smaller than $\lambda^5 = 2^{-30}$. Using the Taylor polynomial, one can obtain the following approximation to $f(x)$ which is accurate to $O(\lambda^5)$:

$$f(x) \approx f(t + \lambda u) + \frac{\lambda}{2}[f(t + \lambda u + \lambda v) - f(t + \lambda u - \lambda v)]$$
$$+ \frac{\lambda^2}{2}[f(t + \lambda u + \lambda w) - f(t + \lambda u - \lambda w)] + \lambda^4\left[\frac{v^2}{2}f^{(2)}(t) - \frac{v^3}{6}f^{(3)}(t)\right]$$

The tedious analysis needed to derive the preceding formula, and its associated error bound, is not presented here. With this method, computing $f(x)$ reduces to:

1. Deriving the four 14-bit values $t + \lambda u + \lambda v$, $t + \lambda u - \lambda v$, $t + \lambda u + \lambda w$, and $t + \lambda u - \lambda w$ using four additions ($t + \lambda u$ needs no computation).
2. Reading the five values of f from a single table or from parallel tables (for higher speed).
3. Reading the value of the last term $\lambda^4[(v^2/2)f^{(2)}(t) - (v^3/6)f^{(3)}(t)]$, which is a function of t and v, from a different table.
4. Performing a six-operand addition.

Analytical evaluation has shown that the error in the preceding computation is guaranteed to be less than the upper bound $ulp/2 = 2^{-24}$. In fact, exhaustive search with all possible 24-bit operands has revealed that the results are accurate to anywhere from 27.3 to 33.3 bits for elementary functions of interest [Wong95].

Our second example of piecewise lookup tables is for modular reduction, that is, finding the d-bit residue modulo p of a given b-bit number z in the range [0, m), where $b = \lceil \log_2 m \rceil$ and $d = \lceil \log_2 p \rceil$. Dividing z into two segments with $b - g$ and g bits, we write:

$$z = 2^g \lfloor z/2^g \rfloor + z \bmod 2^g = 2^g z_{[b-1,g]} + z_{[g-1,0]}$$

For $g \geq d$, the preceding equation leads to a two-table method. The most significant $b - g$ bits, $z_{[b-1,g]}$, index a table with $v_H = \lceil m/2^g \rceil$ words to obtain a d-bit residue. The least significant g bits of z, namely, $z_{[g-1,0]}$, index a v_L-word table ($v_L = 2^g$) to obtain another d-bit residue. These residues are then added and the final d-bit residue is obtained by the standard method of trial subtraction followed by selection, as shown in Fig. 24.7. The total table size, in bits, is

$$B_{\text{divide}} = d(v_H + v_L) = d(\lceil m/2^g \rceil + 2^g)$$

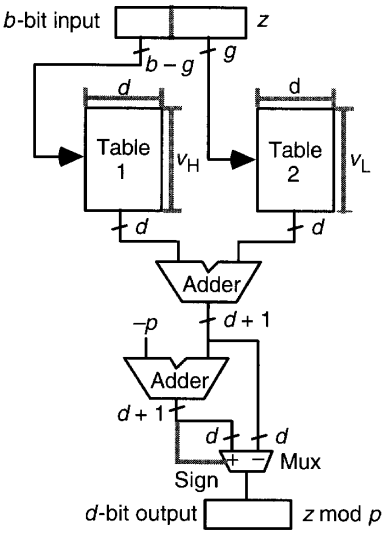


Fig. 24.7 Two-table modular reduction scheme based on the divide-and-conquer approach.

which is minimized if we choose $g = \lfloor \lceil \log_2 m \rceil / 2 \rfloor = \lfloor b/2 \rfloor$. Note that the lower adder and the multiplexer can be replaced by a $2^{d+1} \times d$ table. Alternatively, both adders and the multiplexer in Fig. 24.7 can be replaced by a $2^{2d} \times d$ table.

For example, with $p = 13$, $m = 2^{16}$, $d = 4$, and $b = 16$, the aforementioned optimization leads to tables of total size of 2048 bits—a factor of 128 improvement over direct table lookup.

An alternate two-phase (successive refinement) approach is depicted in Fig. 24.8. First, several high-order bits of z in $[0, m)$ are used to determine what negative multiple of p should be added to z to yield a d^*-bit result z^* in the range $[0, m^*)$, where $p < m^* < m$, $z \bmod p = z^* \bmod p$, and $d^* = \lceil \log_2 m^* \rceil$. Then, the simpler computation $z^* \bmod p$ is performed by direct table lookup.

The most significant $b - h$ bits of z, namely, $z_{[b-1,h]}$, are used to access a v-word table ($v = \lceil m/2^h \rceil$) to obtain a d^*-bit value. This value is the least significant d^* bits of a negative multiple of p such that when it is added to z, the result z^* is guaranteed to satisfy $0 \leq z^* \leq m^*$. A second m^*-word table is used to obtain the d-bit final result $z^* \bmod p$. The total table size, in bits, is:

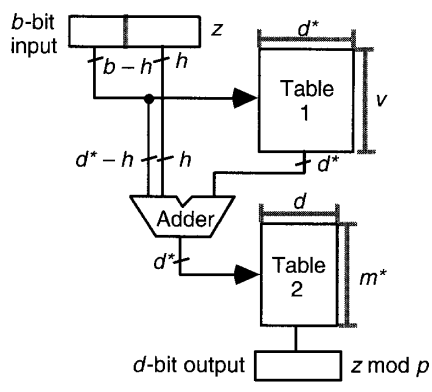


Fig. 24.8 Two-table modular reduction based on successive refinement.

$$B_{\text{refine}} = d^*v + dm^* = d^*\lceil m/2^h \rceil + dm^*$$

In the special case of $m^* < 2p$, the second table can be eliminated and replaced by a subtractor and a multiplexer if desired, thus leading to a single-table scheme.

We see that the total table size is dependent on the parameter m^*. One can prove that the total table size B_{refine} is minimized if d^* is chosen to minimize the objective function $f(d^*) = d^*\lceil m/2^{d^*-1} \rceil + (d \times 2^{d^*-1})$ and m^* is chosen to be $m^* = 2^{d^*-1} + p$. For our earlier example with $p = 13$, $m = 2^{16}$, $d = 4$, $b = 16$, the optimal values for d^* and m^* are 9 and 269, respectively, leading to a total table size of 3380 bits. The resulting tables in this case are larger than for the divide-and-conquer scheme in Fig. 24.7, but the simplicity of the peripheral circuitry (only a single adder besides the tables) can make up for the larger tables.

Modular reduction finds applications in converting numbers from binary or decimal representation to RNS [Parh93a], [Parh94] and in certain error-coding schemes that are based on residues. Details of the preceding methods, including proofs of the results used here, can be found elsewhere [Parh94a], [Parh97].

PROBLEMS

24.1 Squaring by table lookup Show that if the integers x and y are identical in their least significant h bits, their squares will be identical in $h + 1$ bits. Use this result to propose a split-table method (as discussed at the end of Section 24.2) for squaring and estimate the extent of savings in the total table size [Vinn95].

24.2 Squaring by table lookup Consider the following scheme for squaring a k-bit integer x by using much smaller squaring tables. Divide x into two equal-width parts x_H and x_L. Then use the identity $(2^{k/2}x_H + x_L)^2 = 2^k x_H^2 + 2^{k/2+1}x_H x_L + x_L^2$ and perform the multiplication $x_H x_L$ through squaring. Supply the details of the preceding table-lookup scheme for squaring and discuss its speed and cost compared to other methods based on table lookup.

24.3 Squaring by table lookup In Section 24.2 we saw that the table size for squaring can be reduced by a factor of about 2 if the least significant bit ε of $x + y$ and $x - y$ is handled in a specific way. Consider γ and δ, the second LSB of $x + y$ and $x - y$, respectively. Would more complex pre- and postprocessing steps allow us to ignore these bits in table lookup, thus reducing the table size by another factor of 2? Investigate this question, and comment on the cost-effectiveness of the resulting scheme.

24.4 Binary-to-unary reduction method

a. Use the binary-to-unary reduction approach of Section 24.2 to devise a method for computing $x\, e^y$ via table lookup with pre- and/or postprocessing elements.

b. Repeat part a for the function x^y.

24.5 Bit-serial second-order filter Consider the bit-serial second-order filter shown in Fig. 24.3.

a. Show the modifications required in the design to allow radix-4 (2-bits-at-a-time) operation.

b. Show the modifications required in the design to allow the partially accumulated result, now held in register s, to be kept in carry-save form, so that the main adder is replaced by a faster carry-save adder.

c. Compare the suggested modifications of parts a and b with respect to improved speed and added cost.

24.6 Bit-serial arithmetic with table lookup Show how the second-order filter computation depicted in Fig. 24.3 can be programmed on the CM-2 arithmetic unit shown in Fig. 24.2. Assume that the filter coefficients are known at compile time and that all numbers are to be represented as 2's-complement fixed-point numbers with 1 whole (sign) bit and an l-bit fractional part.

24.7 Programmable second-order filter A *programmable filter* is one for which the coefficients $a^{(i)}$ and $b^{(i)}$ can change.

a. How should the filter design in Fig. 24.3 be modified if the coefficients are to be dynamically selectable from among eight sets of values that are known at design time?

b. How should the design be modified if the coefficients are to be dynamically adjustable at run time?

24.8 Function evaluation by table lookup Base-2 logarithm of 16-bit unsigned fractions is to be computed at the input interface of a logarithmic number system processor in which the logarithm is represented as a 12-bit, fixed-point, 2's-complement number with 5 whole (including the sign position) and 7 fractional bits. Using a single table of size $2^{16} \times 12$ bits is impractical. Suggest a method that can use smaller tables (say, up to 10K bits in all) and is also quite fast compared to convergence schemes. Analyze your method with respect to representation error and hardware requirements.

24.9 Interpolating memory for computing $\sin x$ Let angles be represented as 8-bit unsigned fractions x in units of π radians; for example, $(.1000\ 0000)_{\text{two}}$ represents the angle $\pi/2$. Consider the following "interpolating memory" scheme for computing $\sin x$. Two four-word memories are used to store 10-bit, 2's-complement fractions $a^{(i)}$ and $b^{(i)}/4$, $0 \leq i \leq 3$. The function $\sin x$ is then computed by using the linear interpolation formula $\sin x \approx a^{(i)} + b^{(i)}\Delta x$, where $i = (x_{-1}x_{-2})_{\text{two}}$ is the interval index and $4\ \Delta x = (0.x_{-3}x_{-4}x_{-5}x_{-6}x_{-7}x_{-8})_{\text{two}}$ is the scaled offset.

a. Determine the contents of the two tables to minimize the maximum absolute error in computing $\sin x$ for $0 \leq x \leq 1$.

b. Compute the maximum absolute and relative errors implied by your tables.

c. Compare these errors and the implementation cost of your scheme to those of a straight table-lookup scheme, where x is used to access a 256×8 table, and discuss.

24.10 Interpolating memory

a. Construct a table similar to Table 24.1 corresponding to the tabular evaluation of the function e^x for x in $[1, 2)$. Compare the absolute and relative errors for this function to those in Table 24.1 and discuss.

b. Repeat part a for the function $1/x$, with x in $[1, 2)$.

c. Repeat part a for the function $\sqrt{x}$, where x in $[1, 4)$.

24.11 Accuracy of interpolating memory

a. Extend the linear interpolation part of Fig. 24.6 for h up to 16 bits. Show your analysis in full and present the resulting data in tabular as well as graphic form.

b. Repeat part a for linear interpolation applied to the function $\sin x$.

c. Repeat part a for linear interpolation applied to the function e^x.

d. Discuss and compare the observed trends in parts a, b, and c.

24.12 Piecewise table lookup For the piecewise table-lookup method of function evaluation, presented at the beginning of Section 24.6, discuss how the exponent and sign are handled [Wong95].

24.13 Modular reduction with a single table In the description of Fig. 24.7, it was mentioned that for $g \geq d$, two tables are required. For $g < d$, Table 2 of Fig. 24.7 can be eliminated. Derive conditions under which such a single-table realization leads to a smaller total table size.

24.14 Modular reduction by two-step refinement In the two-table modular reduction method shown in Fig. 24.8, it is possible to modify the contents of Table 1 (without increasing its size) in such a way that the d^*-bit adder can be replaced by an h-bit adder plus some extra logic. Show how this can be accomplished and discuss the speed and cost implications of the modified design.

24.15 Modular reduction using tables only Consider tabular reduction by multilevel table lookup using no component other than tables. Figures 24.7 and 24.8 can both be converted to such pure tabular realizations by replacing the adders with tables. Note that other simplifications might occur once the adders have been removed.

a. Derive the total table size for the pure tabular version of Fig. 24.7.

b. Derive the total table size for the pure tabular version of Fig. 24.8.

c. Compare the results of parts a and b and discuss.

24.16 Multilevel modular reduction

a. Generalize the two-level table-lookup scheme of Fig. 24.7 to more than two tables in level 1 followed by a single table, and no other component, in level 2. Discuss how the optimal number of tables in level 1 can be determined.

b. Show how the scheme of part a can be extended to three or more levels.

c. Is the scheme of Fig. 24.8 generalizable to more than two levels?

24.17 Reduced tables for RNS multiplication

a. By relating the mod-p product of $p - x$ and $p - y$ to $xy \bmod p$, show that the size of a mod-p multiplication table can be reduced by a factor of about 4 [Parh93b].

b. Show that an additional twofold reduction in table size is possible because of the commutativity of modular multiplication, namely, $xy \bmod p = yx \bmod p$. Explain how the reduced table is addressed.

REFERENCES

[Ferg91] Ferguson, W.E., Jr., and T. Brightman, "Accurate and Monotone Approximations of Some Transcendental Functions," *Proc. 10th Symp. Computer Arithmetic*, pp. 237–244, 1991.

[Ling90] Ling, H., "An Approach to Implementing Multiplication with Small Tables," *IEEE Trans. Computers*, Vol. 39, No. 5, pp. 717–718, 1990.

[Noet89] Noetzel, A.S., "An Interpolating Memory Unit for Function Evaluation: Analysis and Design," *IEEE Trans. Computers*, Vol. 38, No. 3, pp. 377–384, 1989.

[Parh93a] Parhami, B., "Optimal Table-Lookup Schemes for Binary-to-Residue and Residue-to-Binary Conversions," *Proc. 27th Asilomar Conf. Signals, Systems, and Computers*, Vol. 1, pp. 812–816, November 1993.

[Parh93b] Parhami, B., and H.-F. Lai, "Alternate Memory Compression Schemes for Modular Multiplication," *IEEE Trans. Signal Processing*, Vol. 41, pp. 1378–1385, March 1993.

[Parh94a] Parhami, B., "Analysis of Tabular Methods for Modular Reduction," *Proc. 28th Asilomar Conf. Signals, Systems, and Computers*, October/November 1994, pp. 526–530.

[Parh94b] Parhami, B., and C.Y. Hung, "Optimal Table Lookup Schemes for VLSI Implementation of Input/Output Conversions and Other Residue Number Operations," *VLSI Signal Processing VII* (Proceedings of an IEEE workshop), October 1994, pp. 470–481.

[Parh97] Parhami, B., "Modular Reduction by Multi-Level Table Lookup," *Proc. 40th Midwest Symp. Circuits and Systems*, August 1997, Vol. 1, pp. 381–384.

[Tang91] Tang, P.T.P., "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. Symp. Computer Arithmetic*, 1991, pp. 232–236.

[Vinn95] Vinnakota, B., "Implementing Multiplication with Split Read-Only Memory," *IEEE Trans. Computers*, Vol. 44, No. 11, pp. 1352-1356, 1995.

[Wong95] Wong, W.F., and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers*, Vol. 44, No. 3, pp. 453–457, 1995.

PART VII IMPLEMENTATION TOPICS

We have thus far ignored several important topics that bear on the usefulness and overall quality of computer arithmetic units. In some contexts—say, when we want the hardware to support two floating-point arithmetic operations per cycle on the average and do not mind that the result of each operation becomes available after many cycles—throughput might be more important than latency. Pipelining is the mechanism used to achieve high throughput while keeping the cost and size of the circuits in check. In other contexts, the size or power requirements of the arithmetic circuits are of primary concern. Finally, in critical applications, or in harsh operating environments, tolerance to permanent and transient hardware faults might be required. These topics, along with historical perspectives, case studies, and a look at the impact of emerging technologies, form the following four chapters of this part.

Chapter 25 HIGH-THROUGHPUT ARITHMETIC

With very few exceptions, our discussions to this point have focused on methods of speeding up arithmetic computations by reducing the input-to-output latency, defined as the time interval between the application of inputs and the availability of outputs. When two equal-cost implementations were possible, we always chose the one offering a smaller latency. Once we look beyond individual operations, however, latency ceases to be the only indicator of performance. In pipelined mode of operation, arithmetic operations may have higher latencies owing to pipelining overhead. However, one hardware unit can perform multiple overlapped operations at once. This *concurrency* often more than makes up for the higher latency. Chapter topics include:

25.1 PIPELINING OF ARITHMETIC FUNCTIONS

The key figure of merit for a pipelined implementation is its computational *throughput*, defined as the number of operations that can be performed per unit time. The inverse of throughput, the *pipelining period*, is the time interval between the application of successive input data sets for proper overlapped computation. Of course, latency is still important for two reasons:

1. There may be an occasional need to perform single operations that are not immediately followed by others of the same type.
2. Data dependencies or conditional execution (*pipeline hazards*) may force us to insert *bubbles* into the pipeline or to *drain* it altogether.

However, in pipelined arithmetic, latency assumes a secondary role. We will see later in this chapter that at times, a pipelined implementation may improve the latency of a multistep arithmetic computation while also reducing its hardware cost. In such a case, pipelining is obviously the preferred method, offering the best of all worlds.

Figure 25.1 shows the structure of a σ-stage arithmetic pipeline. Before considering a number of practical issues in the design of arithmetic pipelines, it is instructive to study the trade-offs between throughput, latency, and implementation cost.

Consider an arithmetic function unit whose initial cost is g (in number of logic gates, say) and has a latency of t. Our analysis will be based on a number of simplifying assumptions:

1. The pipelining time overhead per stage is τ (latching time delay).
2. The pipelining cost overhead per stage is γ (latching cost).
3. The function can be divided into σ stages of equal latency for any σ.

Then, the latency T, throughput R, and cost G of the pipelined implementation are:

$$\begin{aligned} &\text{Latency} && T = t + \sigma\tau \\ &\text{Throughput} && R = \frac{1}{T/\sigma} = \frac{1}{t/\sigma + \tau} \\ &\text{Cost} && G = g + \sigma\gamma \end{aligned}$$

We see that, theoretically, throughput approaches its maximum possible value of $1/\tau$ when σ becomes very large. In practice, however, it does not pay to reduce t/σ below a certain threshold; typically four logic gate levels. Even then, one seldom divides the logic into four-level slices blindly; rather, one looks for natural boundaries at which interstage signals (and thus latching costs) will be minimized, even though this may lead to additional stage delay. But let us assume, for the sake of simplifying our analysis, that pipeline stage delay is uniformly equal to four gate delays (4δ). Then, $\sigma = t/(4\delta)$ and:

$$\begin{aligned} &\text{Latency} && T = t\left(1 + \frac{\tau}{4\delta}\right) \\ &\text{Throughput} && R = \frac{1}{T/\sigma} = \frac{1}{4\delta + \tau} \\ &\text{Cost} && G = g\left(1 + \frac{t\gamma}{4g\delta}\right) \end{aligned}$$

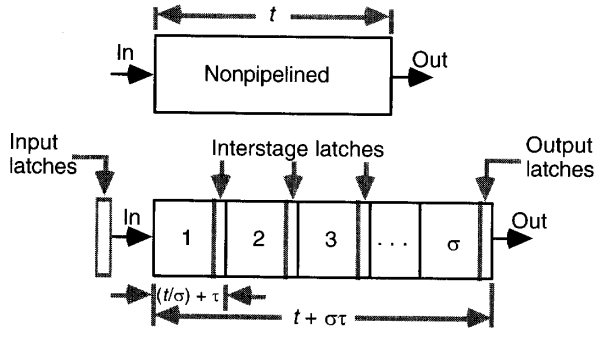


Fig. 25.1 An arithmetic function unit and its σ-stage pipelined version.

The preceding equalities give us an idea of the overhead in latency, $\tau/(4\delta)$, and implementation cost, $t\gamma/(4g\delta)$, to maximize the computational throughput within practical limits.

If throughput is not the single most important factor, one might try to maximize a composite figure of merit. For example, throughput per unit cost may be taken as representing cost-effectiveness:

$$E = \frac{R}{G} = \frac{\sigma}{(t + \sigma\tau)(g + \sigma\gamma)}$$

To maximize E, we compute $dE/d\sigma$:

$$\frac{dE}{d\sigma} = \frac{tg - \sigma^2\tau\gamma}{(t + \sigma\tau)^2(g + \sigma\gamma)^2}$$

Equating $dE/d\sigma$ with 0 yields:

$$\sigma^{\text{opt}} = \left(\frac{t\,g}{\tau\,\gamma}\right)^{1/2}$$

Our simplified analysis thus suggests that the optimal number of pipeline stages for maximal cost-effectiveness is directly related to the latency and cost of the original function and inversely related to pipelining delay and cost overheads: it pays to have many pipeline stages if the function to be implemented is very slow or highly complex, but few pipeline stages are in order if the time and/or cost overhead of pipelining is too high. All in all, not a surprising result!

As an example, with $t = 40\delta$, $g = 500$ gates, $\tau = 4\delta$, and $\gamma = 50$ gates, we obtain $\sigma^{\text{opt}} = 10$ stages. The result of pipelining is that both cost and latency increase by a factor of 2 and throughput improves by a factor of 5. Of course when pipeline hazards are factored in, the optimal number of stages will be much smaller.

25.2 CLOCK RATE AND THROUGHPUT

Consider a σ-stage pipeline and let the worst-case pipeline stage delay be t_{stage}. Suppose one set of inputs is applied to the pipeline at time t_1. At time $t_1 + t_{\text{stage}} + \tau$, the results of this set are safely stored in output latches for the stage. Applying the next set of inputs at time t_2 satisfying $t_2 \geq t_1 + t_{\text{stage}} + \tau$ is enough to ensure proper pipeline operation. With the preceding condition, one set of inputs can be applied to the pipeline every $t_{\text{stage}} + \tau$ time units:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{\text{stage}} + \tau$$

Pipeline throughput is simply the inverse of the clock period:

$$\text{Throughput} = \frac{1}{\text{clock period}} \leq \frac{1}{t_{\text{stage}} + \tau}$$

The preceding analysis assumes that a single clock signal is distributed to all circuit elements and that all latches are clocked at precisely the same time. In reality, we have some uncontrolled or random *clock skew* that may cause the clock signal to arrive at point B before or after its arrival at point A. With proper design of the clock distribution network, we can place an upper bound $\pm\varepsilon$ on the amount of uncontrolled clock skew at the input and output latches of a pipeline stage. Then, the clock period is lower-bounded as follows:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{\text{stage}} + \tau + 2\varepsilon$$

The term 2ε is included because we must assume the worst case when input latches are clocked later and the output latches earlier than planned, reducing the time that is available for stage computation by 2ε. We thus see that uncontrolled clock skew degrades the throughput that would otherwise be achievable.

For a more detailed examination of pipelining, we note that the stage delay t_{stage} is really not a constant but varies from $t_{\min}$ to $t_{\max}$, say; $t_{\min}$ corresponds to fast paths through the logic (fewer gates or faster gates on the path) and $t_{\max}$ to slow paths. Suppose that one set of inputs is applied at time t_1. At time $t_1 + t_{\max} + \tau$, the results of this set are safely stored in output latches for the stage. Assuming that the next set of inputs are applied at time t_2, we must have

$$t_2 + t_{\min} \geq t_1 + t_{\max} + \tau$$

if the signals for the second set of inputs are not to get intermixed with those of the preceding inputs. This places a lower bound on the clock period:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{\max} - t_{\min} + \tau$$

The preceding inequality suggests that we can approach the maximum possible throughput of $1/\tau$ without necessarily requiring very small stage delay. All that is required is to have a very small delay variance $t_{\max} - t_{\min}$.

Using the delay through a pipeline segment as a kind of temporary storage, thus allowing "waves" of unlatched data to travel through the pipeline, is known as *wave pipelining* [Flyn95]. The concept of wave pipelining is depicted in Fig. 25.2, with the wave fronts showing the spatial distribution of fast and slow signals at a given instant. Figure 25.3, an alternate representation of wave pipelining, shows why it is acceptable for the transient regions of consecutive input sets to overlap in time (horizontally) as long as they are separated in space (vertically). Note that conventional pipelining provides separation in both time and space.

The preceding discussion reveals two distinct strategies for increasing the throughput of a pipelined function unit: (1) the traditional method of reducing $t_{\max}$, and (2) the counterintuitive method of increasing $t_{\min}$ so that it is as close to $t_{\max}$ as possible. In the latter method, reducing

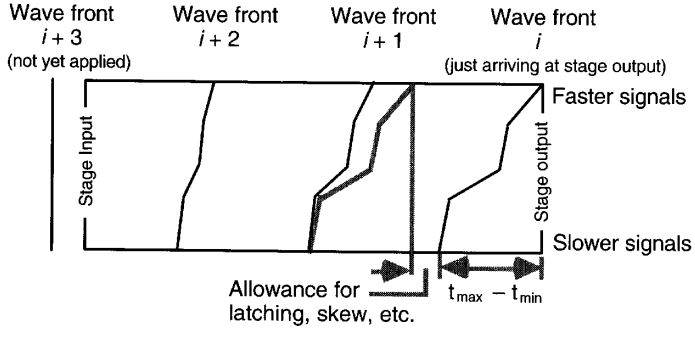


Fig. 25.2 Wave pipelining allows multiple computational wave fronts to coexist in a single pipeline stage.

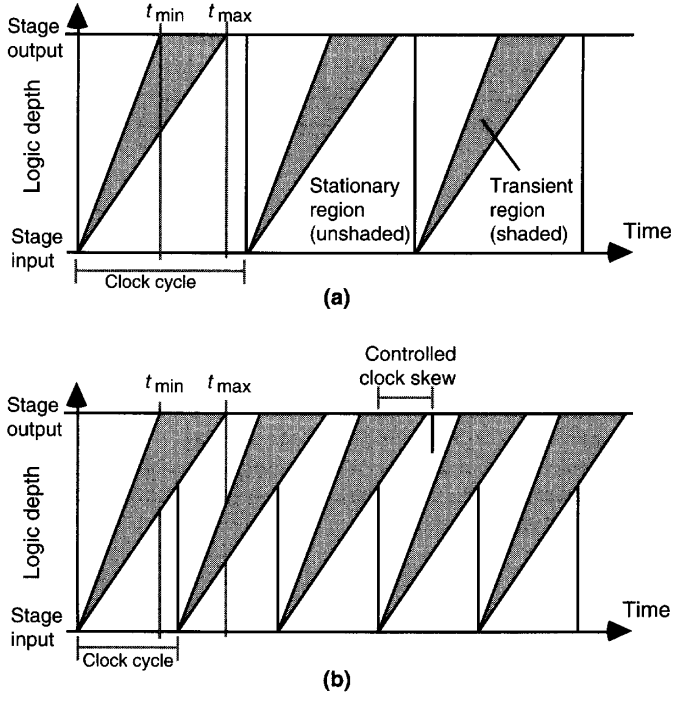


Fig. 25.3 An alternate view of the throughput advantage of wave pipelining (b) over ordinary pipelining (a) using a time–space representation.

t_{max} is beneficial only to the extent that such reduction softens the performance penalty of pipeline hazards.

Suppose, for the moment, that $t_{max} - t_{min} = 0$. Then, the clock period can be taken to be $\Delta t \geq \tau$ and the throughput becomes $1/\Delta t \leq 1/\tau$. Since a new input enters the pipeline stage every Δt time units and the stage latency is $t_{max} + \tau$, the clock application at the output latch must be skewed by $(t_{max} + \tau) \bmod \Delta t$ to ensure proper sampling of the results. For example, if $t_{max} + \tau = 12$ ns and $\Delta t = 5$ ns, then a clock skew of +2 ns is required at the stage output latches relative to the input latches. This *controlled clock skew* is a necessary part of wave pipelining.

More generally, $t_{max} - t_{min}$ is nonzero and perhaps different for the various pipeline stages. Then, the clock period Δt is lower-bounded as follows:

$$\Delta t \geq \max_{1 \leq i \leq \sigma} \left(t_{max}^{(i)} - t_{min}^{(i)} + \tau \right)$$

and the controlled clock skew at the output of stage i will be:

$$S^{(i)} = \sum_{j=1}^{i} \left(t_{max}^{(j)} + \tau \right) \bmod \Delta t$$

We still need to worry about uncontrolled or random clock skew. With the amount of uncontrolled skew upper-bounded by $\pm\varepsilon$, we must have:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{max} - t_{min} + \tau + 4\varepsilon$$

We include the term 4ε because at input, the clocking of the first set of inputs may lag by ε, while that of the second set leads by ε (a net difference of 2ε). In the worst case, the same difference of 2ε may exist at the output, but in the opposite direction. We thus see that uncontrolled clock skew has a larger effect on the performance of wave pipelining than on standard pipelining, especially in relative terms (ε is now a larger fraction of the clock period).

25.3 THE EARLE LATCH

The Earle latch, named after its inventor, J. G. Earle, is a storage element whose output z follows the data input d whenever the clock input C becomes 1. The input data is thus sampled and held in the latch as the clock goes from 1 to 0. Once the input has been sampled, the latch is insensitive to further changes in d as long as the clock C remains at 0. Earle designed the latch of Fig. 25.4 specifically for latching carry-save adders.

Earlier, we derived constraints on the minimum clock period Δt or maximum clock rate $1/\Delta t$. The clock period Δt has two parts: the duration of the clock being high, C_{high}, and duration of the clock being low, C_{low}.

$$\Delta t = C_{high} + C_{low}$$

Now, consider a pipeline stage that is preceded and followed by Earle latches. The duration of the clock being high in each period, C_{high}, must satisfy the inequalities

$$3\delta_{max} - \delta_{min} + S_{max}(C\uparrow, \bar{C}\downarrow) \leq C_{high} \leq 2\delta_{min} + t_{min}$$

where δ_{max} and δ_{min} are maximum and minimum gate delays and $S_{max}(C\uparrow, \bar{C}\downarrow) \geq 0$ is the maximum skew between C going high and $\bar{C}$ going low at the latch input. The right-hand inequality, constraining the maximum width of the clock pulse, simply asserts that the clock must go low before the fastest signals from the next input data set can affect the input z of the Earle latch at the end of the stage. The left-hand inequality asserts that the clock pulse must be wide enough to ensure that valid data is stored in the output latch and to avoid logic hazard, should the 0-to-1 transition of C slightly lead the 1-to-0 transition of $\bar{C}$ at the latch inputs.

The constraints given in the preceding paragraph must be augmented with additional terms to account for clock skew between pipeline segments and to ensure that logic hazards do not lead to the latching of erroneous data. For a more detailed discussion, see [Flyn82, pp. 221–222].

An attractive property of the Earle latch is that it can be merged with the two-level AND-OR logic that precedes it. For example, to latch

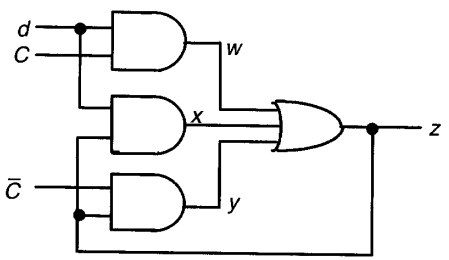


Fig. 25.4 Two-level AND-OR realization of the Earl latch.

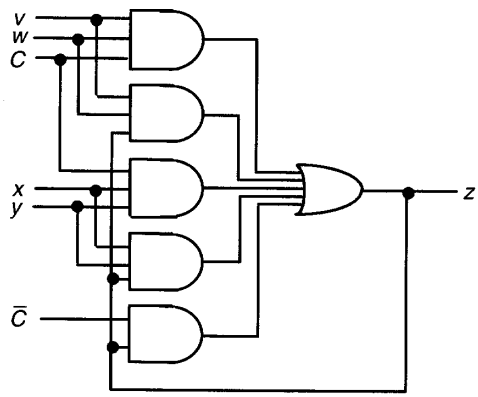


Fig. 25.5 Two-level AND-OR latched realization of the function $z = vw + xy$.

$$d = vw + xy$$

coming from a two-level AND-OR circuit, we substitute for d in the equation for the Earle latch

$$z = dC + dz + \overline{C}z$$

to get the following combined (logic and latch) circuit implementing $z = vw + xy$:

$$\begin{aligned} z &= (vw + xy)C + (vw + xy)z + \overline{C}z \\ &= vwC + xyC + vwz + xyz + \overline{C}z \end{aligned}$$

The resulting two-level AND-OR circuit is shown in Fig. 25.5.

25.4 PARALLEL AND DIGIT-SERIAL PIPELINES

Consider the computation:

$$z = \left[\frac{(a+b)cd}{e-f}\right]^{1/2}$$

To compute z, we need to perform two additions, two multiplications, a division, and a square-root extraction, in the order prescribed by the flow graph shown in Fig. 25.6a. Assuming that multiplication, division, and square-rooting take roughly the same amount of time and that addition is much faster, a timing diagram for the computation can be drawn as shown in Fig. 25.6b. In deriving this timing diagram, it is assumed that enough hardware components are available to do the computation with maximum possible parallelism. This implies the availability of one adder and perhaps a shared multiply/divide/square-root unit.

If the preceding computation is to be performed repeatedly, a pipelined implementation might be contemplated. By using a separate function unit for each node in the flow graph of Fig. 25.6a and inserting latches between consecutive operations, the throughput can be increased

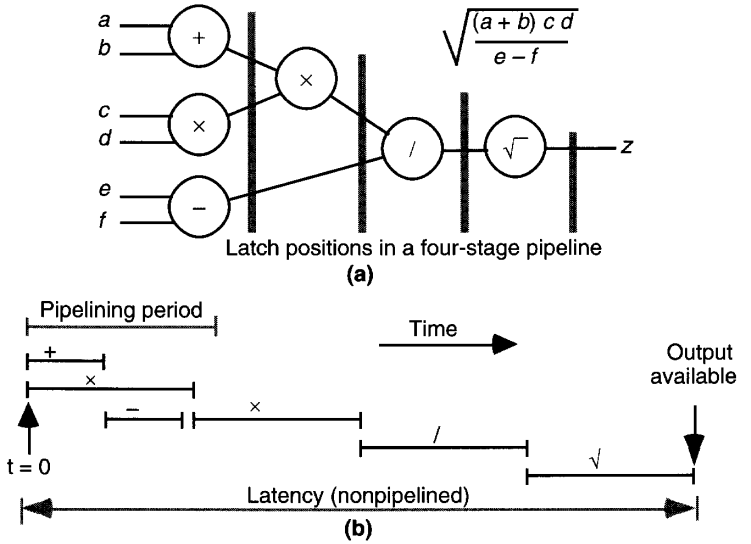


Fig. 25.6 (a) Flow graph representation of an arithmetic expression and (b) timing diagram for its evaluation with digit-parallel computation.

by roughly a factor of 4. However, the requirement for separate multiply, divide, and square-root units would cause the implementation cost to become quite high.

How would one go about doing this computation bit-serially? Bit-serial addition, with the inputs supplied from the least significant end, is easy. We also know how to design an LSB-first, bit-serial multiplier (Section 12.3). With LSB-first, bit-serial computation, as soon as the LSBs of $a + b$ and $c \times d$ are produced, a second bit-serial multiplier can begin the computation of $(a + b) \times (cd)$. This bit-level pipelining is attractive because each additional function unit on the critical path adds very little to the overall latency.

Unfortunately, however, both division and square-rooting are MSB-first operations. So, we cannot begin the division operation in Fig. 25.6 until the results of $(a + b) \times (cd)$ and $e - f$ are available in full. Even then, the division operation cannot be performed in an MSB-first, bit-serial fashion since the MSB of the quotient q in general depends on all the bits of dividend and divisor. To see this, consider the decimal division example 0.1234/0.2469. After inspecting the most significant digits of the two operands, we cannot tell what the MSD of the quotient should be, since

$$\frac{0.1\text{xxx}}{0.2\text{xxx}}$$

can be as large as $0.1999/0.2000 \approx 0.9995$ or as small as $0.1000/0.2999 \approx 0.3334$ (the MSD of the quotient can thus assume any value in [3, 9]). After seeing the second digit of each operand, the ambiguity is still not resolved, since

$$\frac{0.12\text{xx}}{0.24\text{xx}}$$

can be as large as $0.1299/0.2400 \approx 0.5413$ or as small as $0.1200/0.2499 \approx 0.4802$. The next pair of digits further restricts the quotient value to the interval from $0.1239/0.2460 \approx 0.5037$ to

$0.1230/0.2469 \approx 0.4982$ but does not resolve the ambiguity in the MSD of q. Only after seeing all digits of both operands are we able to decide that $q_{-1} = 4$.

To summarize the preceding discussion, with standard number representations, pipelined bit-serial or digit-serial arithmetic is feasible only for computations involving additions and multiplications. These operations are done in LSB-first order, with the output from one block immediately fed to the next block. Division and square-rooting force us to assemble the entire operand(s) and then use one of the algorithms discussed earlier in the book.

If we are allowed to produce the output in a redundant format, quotient/root digits can be produced after only a few bits of each operand have been seen, since the precision required for selecting the next quotient digit is limited. This is essentially because a redundant representation allows us to recover from an underestimated or overestimated quotient or root digit. However, the fundamental difference between LSB-first addition and multiplication and MSB-first division and square-rooting remains and renders a bit-serial approach unattractive.

25.5 ON-LINE OR DIGIT-PIPELINED ARITHMETIC

Redundant number representation can be used to solve the problems discussed at the end of Section 25.4. With redundant numbers, not only can we perform division and square-rooting digit-serially, but we can also convert addition and multiplication to MSD-first operations, thus allowing for smooth flow of data in a pipelined digit-serial fashion.

Figure 25.7 contrasts the timing of the digit-parallel computation scheme (Fig. 25.6) to that of a digit-pipelined scheme. Operations now take somewhat longer to complete (though not much longer, since the larger number of cycles required is partially offset by the higher clock rate allowed for the simpler incremental computation steps). However, the various computation steps are almost completely overlapped, leading to smaller overall latency despite the simpler hardware. The reason for varying operation latencies, defined as the time interval between receiving the ith input digits and producing the ith output digit, will become clear later.

Again, if the computation is to be performed repeatedly, the pattern shown in the digit-pipelined part of Fig. 25.7 can be repeated in time (with a small gap for resetting of the storage elements). Thus, the second computation in Fig. 25.7 can begin as soon as all the digits of the current inputs have been used up.

All that remains is to show that arithmetic operations can be performed in a digit-serial MSD-first fashion, producing the stream of output digits with a small, fixed latency in each case. Binary signed-digit operands, using the digit set $[-1, 1]$ in radix 2, result in the simplest digit-pipelined arithmetic hardware. A higher radix r, with its correspondingly larger digit set, leads to greater circuit complexity, as well as higher pin count, but may improve the performance, given the smaller number of cycles required to supply the inputs. An improvement in performance is uncertain because the more complicated circuit will likely dictate a lower clock rate, thus nullifying some or all of the gain due to reduced cycle count. In practice, $r > 16$ is seldom cost-effective.

Floating-point numbers present additional problems in that the exponents must arrive first and the significands must be processed according to the result of the exponent preprocessing. However, the adjustments needed are straightforward and do not affect the fundamental notions being emphasized here.

Addition is the simplest operation. We already know that in carry-free addition, the $(-i)$th result digit is a function of the $(-i)$th and $(-i-1)$th operand digits. Thus, upon receiving the two

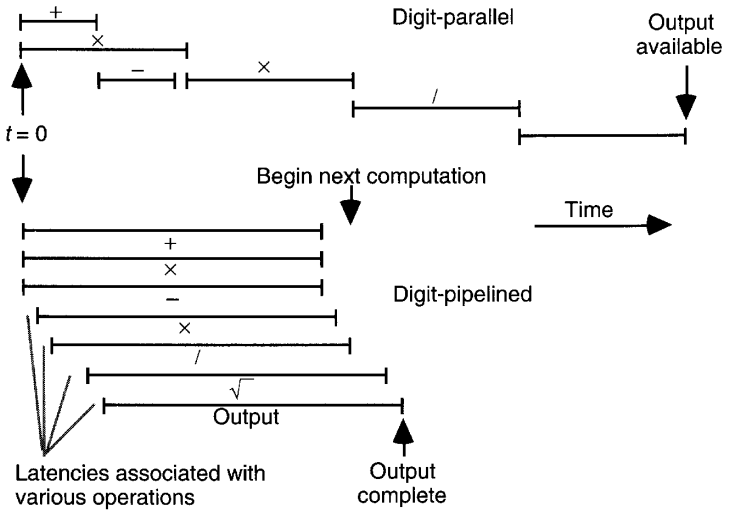


Fig. 25.7 Digit-parallel versus digit-pipelined computation.

most significant digits of the two operands, we have all the information that we need to produce the MSD of the sum/difference.

Figure 25.8 shows a digit-serial MSD-first implementation of carry-free addition. The circuit shown in Fig. 25.8 essentially corresponds to a diagonal slice of Fig. 3.2b and imposes a latency of 1 clock cycle between its input and output.

When carry-free addition is inapplicable (as is the case for binary signed-digit inputs, e.g.), a limited-carry addition algorithm must be implemented. For example, using a diagonal slice of Fig. 3.11a, we obtain the design shown in Fig. 25.9 for digit-pipelined limited-carry addition with a latency of 2 clock cycles.

Multiplication can also be done with a delay of 1 or 2 clock cycles, depending on whether the chosen representation supports carry-free addition. Figure 25.10 depicts the process. In the ith cycle, $i-1$ digits of the operands a and x have already been received and are available in internal registers; call these $a_{[-1,-i+1]}$ and $x_{[-1,-i+1]}$. Also an accumulated partial product $p^{(i-1)}$ (true sum of the processed terms, minus the digits that have already been output) is available. When a_{-i} and x_{-i} are received, the three terms $x_{-i}a_{[-1,-i+1]}$ (two-digit horizontal value in Fig. 25.10),

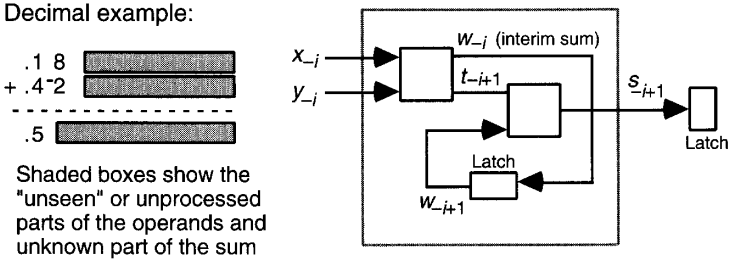


Fig. 25.8 Digit-pipelined MSD-first carry-free addition.

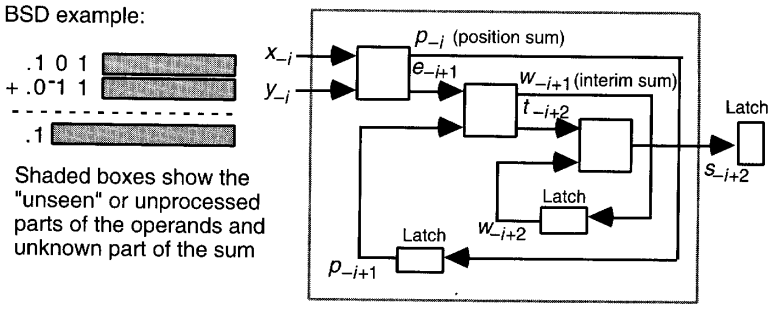


Fig. 25.9 Digit-pipelined MSD-first limited-carry addition.

$a_{-i}x_{[-1,-i+1]}$ (two-digit diagonal value in Fig. 25.10), and $a_{-i}x_{-i}$ (circled term in Fig. 25.10) are computed and combined with the left-shifted $p^{(i-1)}$ to produce an interim partial product by a fast carry-free (limited-carry) addition process. The most significant digit of this result is the next output digit and is thus discarded before the next step. The remaining digits form $p^{(i)}$.

Figure 25.11 depicts a possible hardware realization for digit-pipelined multiplication of BSD fractions. The partial multiplicand $a_{[-1,-i+1]}$ and partial multiplier $x_{[-1,-i+1]}$ are held in registers and the incoming digits a_{-i} and x_{-i} are used to select the appropriate multiples of the two for combining with the product residual $p^{(i-1)}$. This three-operand carry-free addition yields an output digit and a new product residual $p^{(i)}$ to be used for the next step. Note that if the digit-pipelined multiplier is implemented based on Fig. 25.10, then a_{-i} and x_{-i} must be inserted into the appropriate position in their respective registers. Alternatively, each of the digits a_{-i} and x_{-i} may be inserted into the LSD of its respective register, with p_{-i+2} extracted from the appropriate position of the three-operand sum.

Digit-pipelined division is more complicated and involves a delay of 3–4 cycles. Intuitively, the reason for the higher delay in division is seen to lie in the uncertainties in the dividend and divisor, which affect the result in opposite directions. The division example of Table 25.1 shows that with $r = 4$ and digit set $[-2, 2]$, the first quotient digit q_{-1} may remain ambiguous until the fourth digit in the dividend and divisor have appeared. Note that with the given digit set, only fractions in the range $(-2/3, 2/3)$ are representable (we have assumed that overflow is impossible and that the quotient is indeed a fraction).

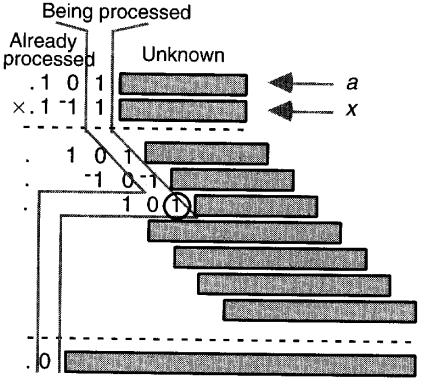


Fig. 25.10 Digit-pipelined MSD-first multiplication process.

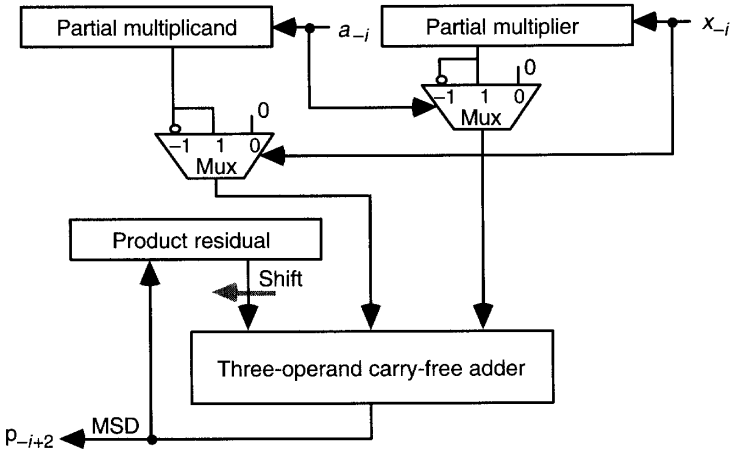


Fig. 25.11 Digit-pipelined MSD-first BSD multiplier.

Note that the example in Table 25.1 shows only that the worst-case delay with this particular representation is at least 3 cycles. One can in fact prove that 3 cycles of delay always is sufficient, provided the number representation system used supports carry-free addition. If limited-carry addition is called for, 4 cycles of delay is necessary and sufficient.

The algorithm for digit-pipelined division and its hardware implementation are similar to those of multiplication. A residual is maintained which is in effect the result of subtracting the product of the known digits of the quotient q and the known digits of the divisor d from the dividend z. With each new digit of q that becomes known, the product of that digit and the partial divisor, as well as the product of the new digit of d and the partial quotient, must be subtracted from the residual. A few bits of the residual, and of the divisor d, may then be used to estimate the next quotient digit.

Square-rooting can be done with a delay of 1–2 cycles, depending on the number representation system used. The first square-rooting example in Table 25.2 shows that, with $r = 10$ and digit set $[-6, 6]$, the first root digit q_{-1} may remain ambiguous until the second digit in the radicand has appeared. The second example, with $r = 2$ and digit set $[-1, 1]$, shows that 2 cycles of delay may be needed in some cases. Again the algorithm and required hardware for digit-pipelined square-rooting are similar to those for digit-pipelined multiplication and division.

TABLE 25.1
Example of digit-pipelined division showing the requirement for 3 cycles of delay before quotient digits can be output (radix = 4, digit set = [−2, 2])

Cycle	Dividend	Divisor	q Range	q_{-1} Range
1	$(.0\cdots)_{four}$	$(.1\cdots)_{four}$	$(-2/3, 2/3)$	$[-2, 2]$
2	$(.0\,0\cdots)_{four}$	$(.1\bar{2}\cdots)_{four}$	$(-2/4, 2/4)$	$[-2, 2]$
3	$(.0\,0\,1\cdots)_{four}$	$(.1\bar{2}\bar{2}\cdots)_{four}$	$(1/16, 5/16)$	$[0, 1]$
4	$(.0\,0\,1\,0\cdots)_{four}$	$(.1\bar{2}\bar{2}\bar{2}\cdots)_{four}$	$(10/64, 14/64)$	1

TABLE 25.2
Examples of digit-pipelined square-root computation showing the requirement for 1–2 cycles of delay before root digits can be output (radix = 10, digit set = [−6, 6], and radix = 2, digit set = [−1, 1])

Cycle	Radicand	q Range	q_{-1} Range
1	$(.3\cdots)_{ten}$	$(\sqrt{7/30}, \sqrt{11/30})$	[5, 6]
2	$(.3\,4\cdots)_{ten}$	$(\sqrt{1/3}, \sqrt{26/75})$	6
1	$(.0\cdots)_{two}$	$(0, \sqrt{1/2})$	[0, 1]
2	$(.0\,1\cdots)_{two}$	$(0, \sqrt{1/2})$	[0, 1]
3	$(.0\,1\,1\cdots)_{two}$	$(1/2, \sqrt{1/2})$	1

25.6 SYSTOLIC ARITHMETIC UNITS

In our discussion of the design of semisystolic and systolic bit-serial unsigned or 2's-complement multipliers (Section 12.3), we noted that the systolic design paradigm allows us to implement certain functions of interest as regular arrays of simple cells (ideally, all identical) with intercell signals carried by short, local wires. To be more precise, we must add to the requirements above the following: no unlatched signal can be allowed to propagate across multiple cells (for otherwise a ripple-carry adder would qualify as a systolic design).

The term "systolic arrays" [Kung82] was coined to characterize cellular circuits in which data elements, entering at the boundaries, advance from cell to cell, are transformed in an incremental fashion, and eventually exit the array, with the lock-step data movement across the array likened to the rhythmic pumping of blood in the veins. As VLSI circuits become faster and denser, we can no longer ignore the contribution of signal propagation delay on long wires to the latency of various computational circuits. In fact, propagation delay, as opposed to switching or gate delays, is now the main source of latency in modern VLSI design. Thus, any high-performance design requires great attention to minimizing wire length, and in the extreme, adherence to systolic design principles.

Fortunately, we already have all the tools needed to design high-performance systolic arithmetic circuits. In what follows, we present two examples.

An array multiplier can be transformed into a bit-parallel systolic multiplier through the application of pipelining methods discussed earlier in this chapter. Referring to the pipelined 5×5 array multiplier in Fig. 11.17, we note that it requires the bits a_i and x_j to be broadcast within the cells of the same column and row, respectively. Now, if a_i is supplied to the cell at the top row and is then passed from cell to cell in the same column on successive clock ticks, the operation of each cell will be delayed by one time step with respect to the cell immediately above it. If the timing of the elements is adjusted, through insertion of latches where needed, such that all other inputs to the cell experience the same added delay, the function realized by the circuit will be unaffected. This type of transformation is known as *systolic retiming*. Of course, additional delays must be inserted on the p outputs if all bits of the product are to become available at once. A similar modification to remove the broadcasting of the x_j signals completes the design.

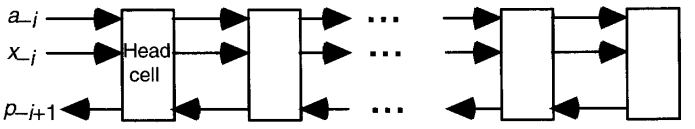


Fig. 25.12 High-level design of a systolic radix-4, digit-pipelined multiplier.

Similarly, a digit-pipelined multiplier can be designed in systolic form to maximize the clock rate and thus the computation speed. Since in the design shown in Fig. 25.11, a_{-i} and x_{-i} are effectively broadcast to a set of 2-to-1 multiplexers, long wires and large fan-outs are involved. Since, however, not all the digits of $x_{-i}a_{[-1,-i+1]}$ and $a_{-i}x_{[-1,-i+1]}$ are needed right away, we can convert the design into a cellular array (Fig. 25.12) in which only the most significant digits of $x_{-i}a_{[-1,-i+1]}$ and $a_{-i}x_{[-1,-i+1]}$ are immediately formed at the head cell, with a_{-i} and x_{-i} passed on to the right on the next clock tick to allow the formation of other digits in subsequent clock cycles and passing of the results to the left when they are needed. Supplying the details of this systolic design is left as an exercise.

PROBLEMS

25.1 Maximizing a pipeline's throughput The assertion in Section 25.1 that the throughput of a pipeline is the inverse of its clock period (which is the sum of the stage delay and latching overhead) is based on the implicit assumption that the pipeline will be utilized continuously for a long period of time. Let ϕ be the probability that a computation is dependent on the preceding computation so that it cannot be initiated until the results of its predecessor have emerged from the pipeline. For each such computation encountered, the pipeline will go unused for $\sigma - 1$ cycles, where σ is the number of stages. Derive the optimal number of pipeline stages to maximize the effective throughput of a pipeline under these conditions.

25.2 Clock rate and pipeline throughput A four-stage pipeline has stage delays of 17, 15, 19, and 14 ns and a fixed per-stage latching overhead of 2 ns. The parameter ϕ, defined as the fraction of operations that cannot enter the pipeline before the preceding operation has been completed, is 0.2.

a. What clock cycle time maximizes throughput if stages cannot be further subdivided? Assume that there is no uncontrolled clock skew.

b. Compare the throughput of part a to the throughput without pipelining.

c. What is the total latency through the pipeline with the cycle time of part a?

d. What clock cycle time maximizes the throughput with arbitrary subdivisions allowed within stages? Latches at the natural boundaries above are not to be removed, but additional latches can be inserted wherever they would be beneficial.

e. What is the total latency through the pipeline with the assumptions of part d?

f. Repeat parts a–e, this time assuming an uncontrolled skew of ± 1 ns in the arrival of each clock pulse.

g. The use of a more elaborate clock distribution network, doubling the clock wiring area (cost) from 20% to 40% of g, can virtually eliminate the uncontrolled clock skew of part f. Would you use the alternate network? Explain.

25.3 Optimal pipelining In the analysis of optimal pipelining in Section 25.1, we assumed that pipelining time and cost overhead per stage are constants. These are simplifying assumptions: in fact, the effects of clock skew intensify for longer, more complex stages and latching overhead increases if the function is sliced indiscriminately at a large number of points. Discuss the optimal number of pipeline stages with each of the following modifications to our original simplifying assumptions.

a. Clock skew increases linearly with stage delay, so that the time or clocking overhead per stage is $\tau + t\alpha/\sigma$.

b. Cost overhead per stage, which grows if the logic function is cut at points other then natural subfunction boundaries, is modeled as a linear function $\gamma + \beta\sigma$ of the number of stages.

c. Both modifications given in parts a and b are in effect.

25.4 Wave pipelining A four-stage pipeline has maximum stage delays of 14, 12, 16, 11 ns, minimum stage delays of 7, 9, 10, 5 ns, and a fixed per-stage overhead of 3 ns. The parameter ϕ defined as the fraction of operations that cannot enter the pipeline before the preceding operation has been completed, is 0.2.

a. With no controlled clock skew allowed, what are the minimum cycle time and the resulting latency?

b. If we allow controlled clock skew, what are the minimum cycle time, clock skews required at the end of each of the four stages, and the overall latency?

c. Repeat parts a and b, this time assuming an uncontrolled skew of ± 1 ns in the arrival of each clock pulse.

25.5 Earle latch logic hazard The Earle latch shown in Fig. 25.4 has a logic hazard.

a. Show the hazard on a Karnaugh map and determine when it leads to failure.

b. Propose a modified latch without a hazard and discuss its practicality.

25.6 Latched full adders

a. Present the complete design of a binary full adder with its sum and carry computations merged with Earle latches.

b. Derive the latching cost overhead with respect to an unlatched FA and an FA followed by separate Earle latches.

25.7 Evaluating a pipelined array multiplier For the pipelined array multiplier design of Fig. 11.17, assume that FA delay is 8 ns and latching overhead is 3 ns.

a. Find the throughput of the design as shown in Fig. 11.17.

b. Modify the design of Fig. 11.17 to have latches following every 2 FAs and repeat part a.

c. Modify the design to have latches following every 3 FAs and repeat part a.

d. Compare the cost-effectiveness of the designs of parts a–c and discuss.

e. The design of Fig. 11.17 can be modified so that the lower part uses HAs instead of FAs. Show how the modification should be done and discuss its implications on optimal pipelining. Assume that HA delay is 4 ns.

25.8 Pipelined ripple-carry adders In designing a deeply pipelined adder, the ripple-carry design provides a good starting point. Study the variations in pipelined ripple-carry adders and their cost–performance implications [Dadd96].

25.9 Optimally pipelined adders In a particular application, 80% of all additions result from operations on long vectors and can thus be performed with full pipeline utilization, leading to a throughput of one addition per clock cycle. The remaining 20% are individual additions for which the total latency of the pipelined adder determines the execution rate. Considering each adder type discussed in Chapters 5–7, derive an optimally pipelined design for the preceding application so that the average addition time is minimized. Is there any adder type that cannot be effectively pipelined? Discuss.

25.10 Pipelined multioperand adders Show that pipelined implementation of a multioperand adder with binary inputs is possible so that the clock period is dictated by the latency of one full-adder [Yeh96].

25.11 Digit-pipelined incrementer/decrementer To compute the expression $(x-1)/(x+1)$ in digit-pipelined fashion, we need to use an incrementer and a decrementer that feed a divider. Assume the use of BSD numbers.

a. Present the design of a combined digit-pipelined incrementer/decrementer unit.

b. Compare your design to a digit-pipelined BSD adder and discuss.

25.12 Digit-pipelined multiplier The multiplier design shown is Fig. 25.11 is incomplete in two respects. First, it does not show how the term $a_{-i}x_{-i}$ is accommodated. Second, it does not specify the alignment of the operands in the three-operand addition or even the width of the adder.

a. Complete the design of Fig. 25.11 by taking care of the problems just identified.

b. Specify additions and modifications to the design for radix-4 multiplication using the digit set $[-2, 2]$.

25.13 Digit-pipelined voting circuits An n-input majority voter produces an output that is equal to a majority of its n inputs, if such a majority exists; otherwise it produces an error signal. A median (mean) voter outputs the median (numerical average) of its n inputs.

a. Show how a three-input digit-serial mean voter can be designed if the inputs are presented in BSD form. What is the latency of your design?

b. Under what conditions can a bit-serial mean voter, with standard binary inputs, be designed and what would be its latency?

c. Discuss whether, and if so, how a digit-serial majority or median voter with BSD inputs can be implemented.

d. Repeat part c with standard binary inputs.

25.14 Systolic digit-pipelined multiplier Design a systolic radix-4 digit-pipelined multiplier structured as in Fig. 25.12 based on the ideas presented in Section 25.6.

25.15 Systolic array multiplier

a. Based on the discussions in Section 25.6, convert the pipelined array multiplier design of Fig. 11.17 into a fully pipelined systolic array multiplier.

b. Repeat part a, this time assuming that propagation across two cells is acceptable.

25.16 Delays in on-line arithmetic That digit-pipelined addition can be performed with one or two cycles of delay between input arrival and output production is a direct result of the theories of carry-free and limited-carry addition developed in Chapter 3.

a. With reference to Fig. 25.10 for digit-pipelined multiplication of BSD numbers, show that two cycles of delay is adequate.

b. Show that digit-pipelined multiplication can be performed with 2–3 cycles of delay.

c. What would be the delay of a digit-pipelined multiply-add unit?

d. Show that digit-pipelined square-rooting can be performed with 1–2 cycles of delay.

e. Show that digit-pipelined division can be performed with 3–4 cycles of delay.

REFERENCES

[Burl98] Burleson, W.P., M. Ciesielski, F. Klass, and W. Liu, "Wave Pipelining: A Tutorial and Research Survey," *IEEE Trans. Very Large Scale Integrated Systems*, Vol. 6, No. 3, pp. 464–474, September 1998.

[Dadd96] Dadda, L., and V. Piuri, "Pipelined Adders," *IEEE Trans. Computers*, Vol. 45, No. 3, pp. 348–356, 1996.

[Davi97] Davidovic, G., J. Ciric, J. Ristic-Djurovic, V. Milutinovic, and M. Flynn, "A Comparative Study of Adders: Wave Pipelining vs. Classical Design," *IEEE Computer Architecture Technical Committee Newsletter*, June 1997, pp. 64–71.

[Erce88] Ercegovac, M.D., and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications," *VLSI Signal Processing III* (Proceedings of an IEEE workshop), 1988, pp. 252–263.

[Flyn82] Flynn, M.J., and S. Waser, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

[Flyn95] Flynn, M.J., *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett, 1995.

[Frie94] Friedman, G., and J.H. Mulligan, Jr., "Pipelining and Clocking of High Performance Synchronous Digital Systems," in *VLSI Signal Processing Technology*, M.A. Bayoumi and E.E. Swartzlander, Jr., (eds.), Kluwer, 1994, pp. 97–133.

[Irwi87] Irwin, M.J., and R.M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial," *IEEE Computer*, Vol. 20, No. 4, pp. 61–73, 1987.

[Kung82] Kung, H.T., "Why Systolic Architectures?" *IEEE Computer*, Vol. 15, No. 1, pp. 37–46, 1982.

[Yeh96] Yeh, C.-H., and B. Parhami, "Efficient Pipelined Multi-Operand Adders with High Throughput and Low Latency: Design and Applications," *Proc. 30th Asilomar Conf. Signals, Systems, and Computers*, November 1996, pp. 894–898.

Chapter 26 LOW-POWER ARITHMETIC

Classical computer arithmetic focuses on latency and hardware complexity as the primary parameters to be optimized or traded off against each other. We saw in Chapter 25 that throughput is also important and may be considered in design trade-offs. Recently, power consumption has emerged as a key factor for two reasons: limited availability of power in small portable or embedded systems and limited capacity to dispose of the heat generated by fast, power-hungry circuits. In this chapter, we review low-power design concepts that pertain to the algorithm or logic design level; as opposed to circuit-level methods, which are outside the scope of this book. Chapter topics include:

26.1 THE NEED FOR LOW-POWER DESIGN

In modern digital systems, factors other then speed and cost are becoming increasingly important. For example, portable or wearable computers are severely constrained in weight, volume, and power consumption. Whereas weight and volume might seem to be strongly correlated with circuit complexity or cost, factors external to the circuits themselves often dominate the system's weight and volume. For example, packaging, power supply, and cooling provisions might exhibit variations over different technologies that dwarf the contribution of the circuit elements to weight and volume. In power consumption, too, logic and arithmetic circuits might be responsible for only a small fraction of the total power. Nevertheless, it is important to minimize power wastage and to apply power saving methods wherever possible.

In portable and wearable electronic devices, power is at a premium. Nickel–cadmium batteries offer around 40–50 watt-hours of energy per kilogram of weight [Raba96], requiring the total power consumption to be limited to 3–5 W to make a day's worth of operation

feasible between recharges, given a practical battery weight of under 1 kg. Power management becomes even more daunting if we focus on personal communication/computation devices with a battery weight of 0.1 kg or less. Newer battery technologies improve the situation only marginally.

This limited power must be budgeted for computation, storage (primary and secondary), video display, and communication, making the share available for computation relatively small. The power consumption of modern microprocessors grows almost linearly with the product of die area and clock frequency and today stands at a few tens of watts in high-performance designs. This is 1–2 orders of magnitude higher than what is required to achieve the aforementioned goal of 3–5 W total power. Roughly speaking, such processors offer 10–20 MFLOPS of performance for each watt of power dissipated.

The preceding discussion leads to the somewhat surprising conclusion that reducing power consumption is also important for high-performance uniprocessor and parallel systems that do not need to be portable or battery-operated. The reason is that higher power dissipation requires the use of more complex cooling techniques, which are costly to build, operate, and maintain. In addition, digital electronic circuits tend to become much less reliable at high operating temperatures; hence we have another incentive for low-power design.

While improvements in technology will steadily increase the battery capacity in portable systems, it is a virtual certainty that increases in die area and clock speed will outpace the improvements in power supplies. Larger circuit area and higher speed are direct results of greater demand for functionality as well as increasing emphasis on computation-intensive applications (e.g., in multimedia), which also require the storage, searching, and analyzing of vast amounts of data.

Thus, low-power design methods, which are quite important now, will likely rise in significance in the coming years as portable digital systems and high-end supercomputers become more prevalent.

Figure 26.1 shows the power consumption trend for each MIPS (million instructions per second) of computational performance in DSP chips [Raba98]. We note that despite higher overall power consumption, there has been a tenfold decrease in power consumption per MIPS every 5 years. This reduction is due to a combination of improved power management methods

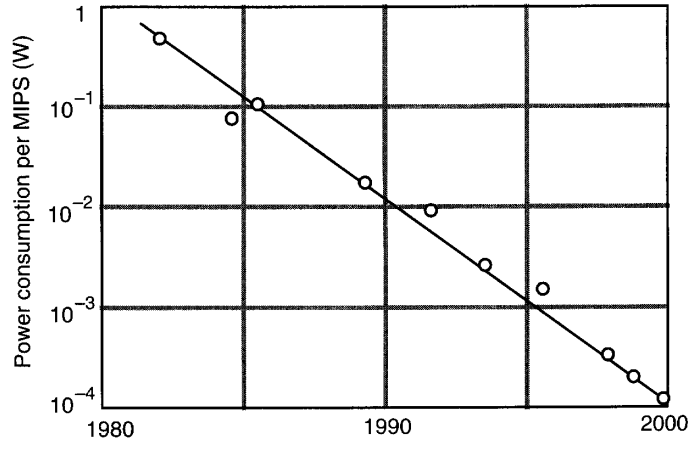


Fig. 26.1 Power consumption trend in DSPs [Raba98].

and lower supply voltages. The 1999–2000 estimates in Fig. 26.1 are for supply voltage of 1–2 V, with 0.5-V DSPs anticipated by the year 2005.

26.2 SOURCES OF POWER CONSUMPTION

To design low-power arithmetic circuits, one must understand the sources of power dissipation and the relationship of power consumption to other important system parameters. Some circuit technologies, such as TTL, are quite unsuitable for low-power designs in view of their relatively high average power consumption. The inherently low-power CMOS technology, on the other hand, can be readily adapted to even more stringent power consumption goals. We will limit our discussion to CMOS, which is currently the predominant implementation technology for both low-cost and high-performance systems.

Besides average power consumption, which is limited by the power budgeted for each subsystem or activity, the peak power consumption is also important in view of its impact on power distribution and signal integrity [Raba96]. Typically, low-power design aims at reducing both the average and peak power.

Power dissipation in CMOS digital circuits is classified as static or dynamic. Static power dissipation occurs, for example, as a result of leakage currents through MOS transistors that form imperfect switches. Excluding certain CMOS families (such as ratioed CMOS logic also known as pseudo-NMOS) that are not used in low-power designs, the aforementioned and other sources of static power dissipation are typically responsible for less than 10% of total power.

Dynamic power dissipation in a CMOS device arises from its transient switching behavior. A small part of such dynamic power dissipation is due to brief short circuits when both the NMOS and PMOS devices between the supply voltage and ground are momentarily turned on. This part of dynamic power dissipation can be kept under control by circuit design techniques and by regulating the signal rise and fall times. This leaves us the dynamic dissipation due to charging and discharging of parasitic capacitance to contend with.

Switching from ground to the supply voltage V, and back to ground, dissipates a power equal to CV^2, where C is the capacitance. Thus, the average power consumption in CMOS can be characterized by the equation

$$P_{\text{avg}} \approx \alpha f C V^2$$

where f is the data rate (clock frequency) and α, known as "activity," is the average number of 0-to-1 transitions per clock cycle.

As a numerical example, consider the power consumption of a 32-bit off-chip bus operating at 5 V and 100 MHz, driving a capacitance of 30 pF per bit. If random values were placed on the bus in every cycle, we would have $\alpha = 0.5$. To account for data correlation and idle bus cycles, let us assume $\alpha = 0.2$. Then:

$$P_{\text{avg}} \approx \alpha f C V^2 = 0.2 \times 10^8 (32 \times 30 \times 10^{-12}) 5^2 = 0.48 \text{ W}$$

Based on the equation for dynamic power dissipation in CMOS digital circuits, once the data rate f has been fixed, there are but three ways to reduce the power requirements:

1. Using a lower supply voltage V.
2. Reducing the parasitic capacitance C.
3. Lowering the switching activity α.

An alternative to all of the above is to avoid power dissipation altogether, perhaps through circuit augmentation and redesign, such that the normally dissipated energy is conserved for later reuse [Atha96]. However, this latter technique, known as adiabatic switching/charging, is still in its infancy and faces many obstacles before practical applications can be planned.

Given that power dissipation increases quadratically with the supply voltage, reduction of V is a highly effective method for low-power design. A great deal of effort has been expended in recent years on the development of low-voltage technologies and design methods. Unfortunately, however, whereas the transition from 5 V to the present 3.3 V was achieved simply and with little degradation in performance, lower supply voltages come with moderate to serious speed penalties and also present problems with regard to compatibility with peripheral off-the-shelf components. Some of the resulting performance degradation can be mitigated by architectural methods such as increased pipeline depth or parallelism, in effect trading silicon area for lower power. Such methods should make supply voltages at or slightly above 1 V feasible in the near future. Beyond that, however, reduction of V becomes even more difficult.

Parasitic capacitance in CMOS can be reduced by using fewer and smaller devices as well as sparser and shorter interconnects. Of course both device-size reduction and interconnect localization have nontrivial performance implications. Smaller devices, with their lower drive currents, tend to be slower. Similarly, high-speed designs often imply a certain amount of nonlocal wires. For example, a ripple-carry adder has a relatively small number of devices and only short local wires, which lead to lower capacitance. However, the resulting capacitance reduction is usually not significant enough for us to altogether avoid the faster carry-lookahead designs with their attendant long, nonlocal interconnects. This interplay between capacitance and speed, combined with the performance effects of lower supply voltage, make the low-power design process a challenging global optimization problem (see Section 26.5).

The preceding points, along with methods for reducing the activity α, as discussed in Section 26.4, lead to several paradigms that are recurring themes in low-power design [Raba96]:

Avoiding waste. Glitching, or signals going through multiple transitions before settling at their final values, clocking modules when they are idle, and use of programmable (rather than dedicated) hardware constitute examples of waste that can be avoided.

Performance vs. power. Slower circuits use less power, so low-power circuits are often designed to barely meet performance requirements.

Area (cost) vs. power. Parallel processing and pipelining, with their attendant area overheads, can be applied to achieve desired performance levels at lower supply voltage and, thus, lower power.

Exploiting locality. Partitioning the design to exploit data locality improves both speed and power consumption.

Minimizing signal transitions. Careful encoding of data and state information, along with optimizations in the order and type of data manipulations, can reduce the average number of signal transitions per clock cycle and thus lead to lower power consumption. This is where number representations and arithmetic algorithms play key roles.

Dynamic adaptation. Changing the operating environment based on the input characteristics, selective precomputation of logic values before they are actually needed, and lazy evaluation (not computing values until absolutely necessary) all affect the power requirements.

These and other methods of saving power are being actively pursued within the research community. The following sections discuss specific examples of these methods in the context of arithmetic circuits.

26.3 REDUCTION OF POWER WASTE

The most obvious method of lowering the power consumption is to reduce the number or complexity of arithmetic operations performed. Two multiplications consume more power than one, and shifting plus addition requires less power than multiplication. Thus, computing from the expression $a(b + c)$ is better than using $ab + ac$. Similarly, $16a - a$ is preferable to $15a$.

Of course, the preceding examples represent optimizations that should be done regardless of whether power consumption is an issue. In other cases, however, operator reduction implies a sacrifice in speed, thus making the trade-off less clear-cut, especially if the lost speed is to be recovered by using a higher clock rate and/or supply voltage.

Multiplication of complex numbers provides a good example. Consider the following complex multiplication:

$$(a + bj)(c + dj) = (ac - bd) + (ad + bc)j$$

which requires four multiplications and two additions if implemented directly. The following equivalent formulation, however, includes only three multiplications, since $c(a + b)$, which appears in both the real and imaginary parts, needs to be computed only once:

$$(a + bj)(c + dj) = [c(a + b) - b(c + d)] + [c(a + b) - a(c - d)]j$$

The resulting circuit will have a critical path that is longer than that of the first design by at least one adder delay. This method becomes more attractive if $c + dj$ is a constant that must be multiplied by a given sequence of complex values $a^{(i)} + b^{(i)}j$. In this case, $c+d$ and $c-d$ are computed only once, leading to three multiplications and three additions per complex step thereafter.

When an arithmetic system consists of several functional units, or subcircuits, some of which remain unused for extended periods, it is advantageous to disable or turn off those units through clock gating (Fig. 26.2). The elimination of unnecessary clock activities inside the gated functional unit saves power, provided the gating signal itself changes at a much lower rate than the clock. Of course, the generation of the gating signals implies some overhead in terms of both cost and power consumption in the control logic. There may also be a speed penalty in view of a slight increase in the critical path for some signals.

A technique related to clock gating is guarded evaluation (Fig. 26.3). If the output of a function unit (FU) is relevant only if a particular select signal is high, that same select signal can be used to control a set of latches (or blocking gates) at the input to the unit. When the select signal is high, the latches become transparent; otherwise, the earlier inputs to the function unit are preserved, to suppress any activity in the unit.

A major source of wasted power in arithmetic and other digital circuits is glitching. Glitching occurs as a result of delay imbalances in signal propagation paths that lead to spurious transitions.

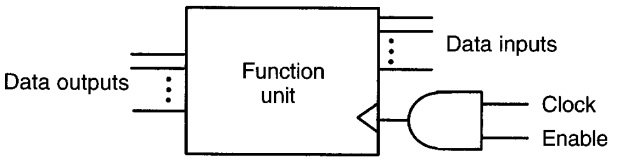


Fig. 26.2 Saving power through clock gating.

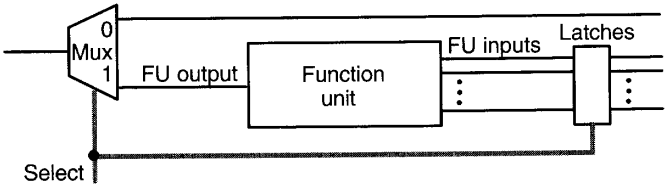


Fig. 26.3 Saving power via guarded evaluation.

Consider, for example, the full-adder cell in position i of a ripple-carry adder (Fig. 26.4). Suppose that c_i, p_i, and s_i are initially set to 0s and that both c_i and p_i are to change to 1 for a new set of inputs. The change in p_i takes effect almost immediately, whereas the 0-to-1 transition of c_i may occur after a long propagation delay. Therefore, s_i becomes 1 and is then switched back to 0. This redundant switching to 1 and then back to 0 wastes power.

Glitching can be eliminated, or substantially reduced, through delay balancing. Consider, for example, the array multiplier of Fig. 26.5. In this multiplier, each cell has four inputs, rather than three for a standard full adder, because one input to the FA is internally computed as the logical AND of the upper-horizontal and vertical inputs. The diagonal output is the sum and the lower-horizontal output is the carry.

Tracing the signal propagation paths in Fig. 26.5, we find that the lower-horizontal carry input and the diagonal sum input into the cell at the intersection of row x_i and column a_j and both experience a critical path delay of $2i + j$ cells, whereas the other input signals arrive with virtually no delay from the primary inputs. This difference can cause significant glitching. To reduce the power waste due to this glitching, one can insert delays along the paths of the vertical and horizontal broadcast inputs, a_i and x_j. Placing 1 and 2 units of delay within each cell on the horizontal and vertical broadcast lines, respectively, balances all the signal paths. Of course, the latency of the array multiplier will increase as a result of this delay balancing.

Similar methods of delay balancing can be applied to fast tree multipliers. However, deriving the delay-balanced design is somewhat harder for the latter in view of their irregular structures leading to signal paths with varying delays. Some delay balancing methods for such multipliers

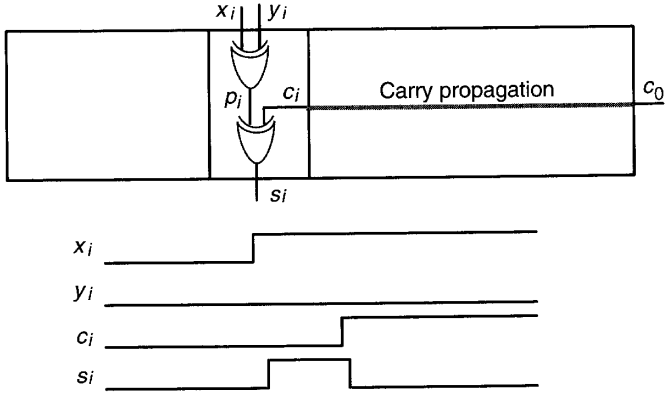


Fig. 26.4 Example of glitching in a ripple-carry adder.

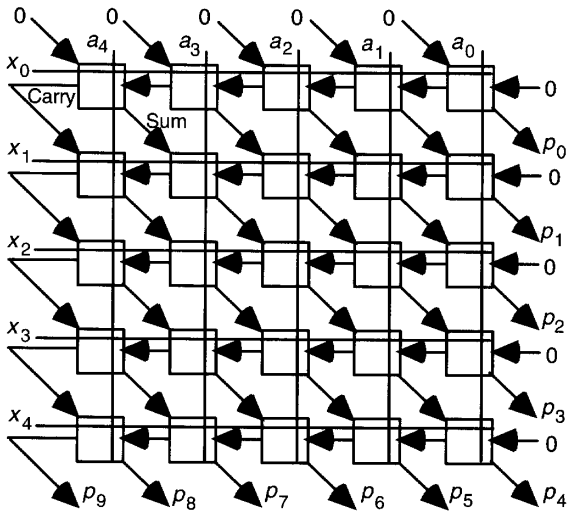


Fig. 26.5 An array multiplier with gated FA cells.

are given in [Saku95], where it is concluded that a power saving of more than 1/3 is feasible. Delay balancing methods for tree multipliers were studied even before their implications for power consumption became important. For example, we saw in Section 11.2, that balanced-tree multipliers were developed to facilitate the synthesis of partial product reduction trees from identical bit slices.

Pipelining also helps with glitch reduction and thus can lead to power savings. In a pipelined implementation, the logic depth within each pipeline segment can be made fairly small, leading to reduced opportunities for glitching. Existence of nodes that are deep, on the other hand, virtually guarantees that glitching will occur, both because of variations in signal path lengths and as a result of the deeper circuit nodes being within the cone of influence of a larger number of primary inputs. The effects of pipelining are further discussed in Sections 26.5 and 26.6.

26.4 REDUCTION OF ACTIVITY

Reduction of the activity α can be accomplished by a variety of methods. An examination of the effects of various information encoding schemes makes a good starting point. Consider, for example, the effect of 2's-complement encoding of numbers versus signed-magnitude encoding during negation or sign change. A signed-magnitude number is negated by simply flipping its sign bit, which involves minimal activity. For a 2's-complement number, on the other hand, many bits will change on the average, thus creating a great deal of activity. This does not mean, however, that signed-magnitude number representation is always better from the standpoint of power consumption. The more complex addition/subtraction process for such numbers may nullify some or all of this gain.

As another example of the effect of information encoding on power consumption, consider the design of a counter. Standard binary encoding of the count implies an average of about two transitions, or bit inversions, per cycle. Counting according to a Gray code, in which the

representation of the next higher or lower number always differs from the current one in exactly one bit, reduces the activity by a factor of 2. This advantage exists in unidirectional counting as well as in up/down counting. One can generalize from this and examine energy-efficient state encoding schemes for sequential machines. If the states of a sequential machine are encoded such that states frequently visited in successive transitions have adjacent codes, the activity will be reduced.

The encoding scheme used might have an effect on power consumption in the implementation of high-radix or redundant arithmetic, as well. Each high-radix or redundant digit is typically encoded in multiple bits. We saw in Section 3.4, for example, that the particular encoding used to represent the BSD digit set $[-1, 1]$ has significant speed and cost implications. Power consumption might also be factored in when selecting the encoding. Very little can be said in general about power-efficient encodings. Distribution and correlation of data have significant effects on the optimal choice.

Generally speaking, shared, as opposed to dedicated, processing elements and data paths tend to increase the activity and should be avoided in low-power design if possible. If a wire or bus carries a positively correlated data stream on successive cycles, then switching activity is likely to be small (e.g., the high-order bits of numbers do not change in every cycle). If the same wire or bus carries elements from two independent data streams on alternate cycles, there will be significant switching activity, as each bit will change with probability 1/2 in every cycle.

Reordering of operations sometimes helps reduce the activity. For example, in adding a list of n numbers, separating them into two groups of positive and negative values, adding each group separately, and then adding the results together is likely to lead to reduced activity. Interestingly, this strategy also minimizes the effect of round-off errors, so it is doubly beneficial.

A method known as precomputation can sometimes help reduce the activity. Suppose we want to evaluate a function f of n variables such that the value of f can often be determined from a small number m of the n variables. Then the scheme depicted in Fig. 26.6 can be used to reduce the switching activity within the main computation circuit. In this scheme, a smaller "prediction" circuit detects the special cases in which the value of f actually depends on the remaining $n - m$ variables, and only then allows these values to be loaded into the input registers. Of course, since the precomputation circuit is added to the critical path, this scheme does involve a speed penalty in addition to the obvious cost overhead.

A variant of the precomputation scheme is to decompose a complicated computation into two or more simpler computations based on the value of one or more input variables. For

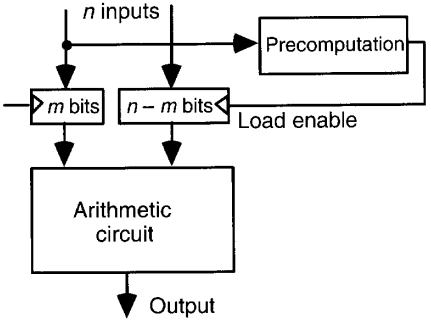


Fig. 26.6 Reduction of activity by precomputation.

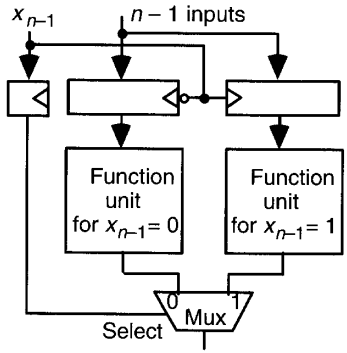


Fig. 26.7 Reduction of activity via Shannon expansion.

example, using the Shannon expansion of a function around the input variable x_{n-1} leads to the implementation shown in Fig. 26.7. Here, the input register is duplicated for $n-1$ of the n variables and the value of x_{n-1} is used to load the input data into one or the other register. The obvious overhead in terms of registers is unavoidable in this scheme. The overhead in the computation portion of the circuit can be minimized by proper selection of the expansion variable(s).

26.5 TRANSFORMATIONS AND TRADE-OFFS

Many power-saving schemes require that some other aspect of the arithmetic circuit, such as its speed or simplicity, be sacrificed. In this section, we look at some trade-offs of this nature.

Replacing the commonly used single-edge-triggered flip-flops (that load data at the rising or falling edge of the clock signal) by double-edge-triggered flip-flops would allow a factor-of-2 reduction in the clock frequency. Since clock distribution constitutes a major source of power consumption in synchronous systems, this transformation can lead to savings in power at the cost of more complex flip-flops. Flip-flops can also be designed to be self-gating, so that if the input of the flip-flop is identical to its output, the switching of its internal clock signal is suppressed to save power. Again, a self-gating flip-flop is more complex that a standard one.

Parallelism and pipelining are complementary methods of increasing the throughput of an arithmetic circuit. A two-way parallel circuit or a two-stage pipelined circuit can potentially increase the throughput by a factor of 2. Both methods can also be used to reduce the power consumption.

Consider an arithmetic circuit, such as a multiplier, that is required to operate at the frequency f; that is, it must perform f operations per second. A standard design, operating at voltage V, is shown in Fig. 26.8a. The power dissipation of this design is proportional to fCV^2, as discussed in Section 26.2, where C is the effective capacitance. If we duplicate the circuit and use each copy to operate on alternating input values, as shown in Fig. 26.8b, then the required operating frequency of each copy becomes $f/2$. This increases the effective capacitance of the overall circuit to $2.2C$, say, but allows the slower copies to use a lower voltage of $0.6V$, say. The net

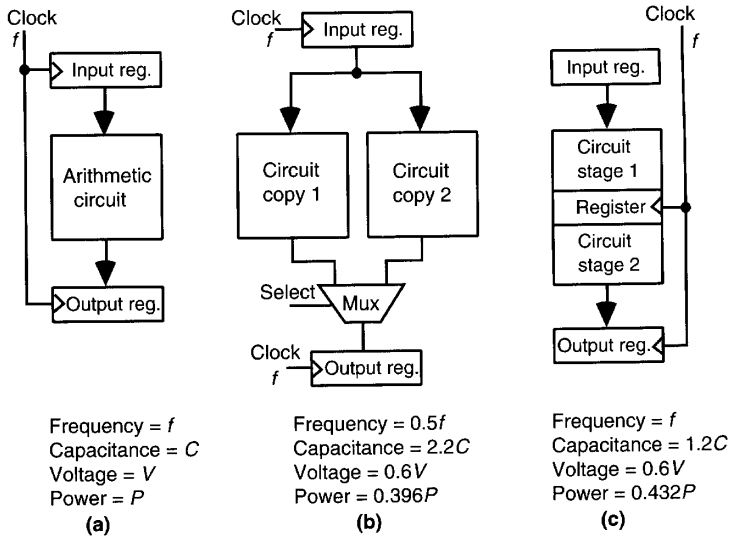


Fig. 26.8 Reduction of power via parallelism or pipelining.

effect is that the power is reduced from P to $(0.5 \times 2.2 \times 0.6^2)P = 0.396P$ while maintaining the original performance.

An alternative power reduction architecture with pipelining is shown in Fig. 26.8c. Here, the computation is sliced into two stages, each only half as deep as the original circuit. Thus, voltage can again be reduced from V to $0.6V$, say. The hardware overhead of pipelining increases the capacitance to $1.2C$, say, while the operating frequency f remains the same. The net effect is that power is reduced from P to $(1 \times 1.2 \times 0.6^2)P = 0.432P$ while maintaining the original performance in terms of throughput.

The possibility of using parallelism or pipelining to save power is not always easily perceived. Consider, for example, the recursive computation

$$y^{(i)} = ax^{(i)} + by^{(i-1)}$$

where the coefficients a and b are constants. For this first-order, infinite impulse response (IIR) filter, the circuit implementation shown in Fig. 26.9 immediately suggests itself. The operating frequency of this circuit is dictated by the latency of a multiply-add operation.

The method that allows us to apply parallelism to this computation is known as loop unrolling. In this method, we essentially compute the two outputs $y^{(i)}$ and $y^{(i+1)}$ simultaneously using the equations:

$$\begin{aligned} y^{(i)} &= ax^{(i)} + by^{(i-1)} \\ y^{(i+1)} &= ax^{(i+1)} + abx^{(i)} + b^2 y^{(i-1)} \end{aligned}$$

The preceding equations lead to the implementation shown in Fig. 26.10 which, just like the parallel scheme of Fig. 26.8, can operate at a lower frequency, and thus at a lower voltage, without affecting the throughput. The new operating frequency will be somewhat lower than $f/2$

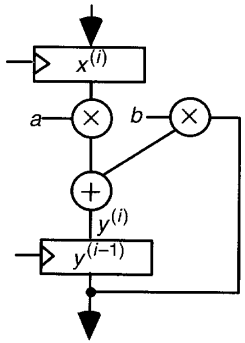


Fig. 26.9 Direct realization of a first-order IIR filter.

because the three-operand adder in Fig. 26.10 is slower than a two-operand adder. However, the difference between the operating frequency and $f/2$ will be negligible if the three-operand adder is implemented by a carry-save adder followed by a standard two-operand adder.

Retiming, or redistribution of delay elements (registers) in a design, is another method that may be used to reduce the power consumption. Note that retiming can also be used for throughput enhancement, as discussed in connection with the design of systolic arithmetic function units in Section 25.6. As an example of power implications of retiming, consider a fourth-order, finite impulse response (FIR) filter characterized by the following equation:

$$y^{(i)} = ax^{(i)} + bx^{(i-1)} + cx^{(i-2)} + dx^{(i-3)}$$

Figure 26.11 shows a straightforward realization of the filter. The frequency at which the filter can operate, and thus the supply voltage, is dictated by the latency of one multiplication and three additions. The number of addition levels can be reduced to two by using a two-level

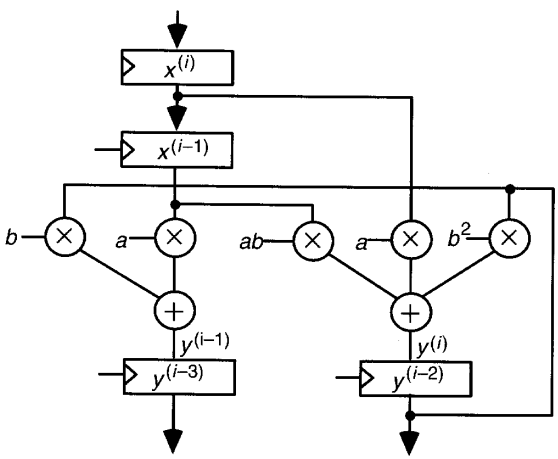


Fig. 26.10 Realization of a first-order IIR filter, unrolled once.

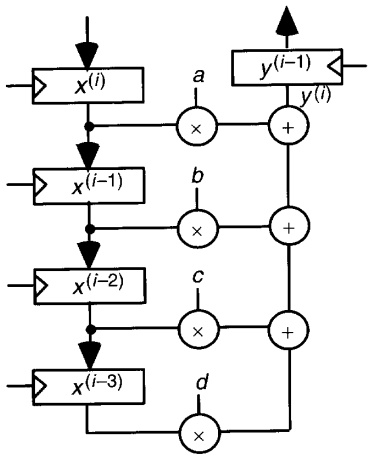


Fig. 26.11 Possible realization of a fourth-order FIR filter.

binary tree of adders, but the resulting design is less regular and more difficult to expand in a modular fashion.

An alternative design, depicted in Fig. 26.12, moves the registers to the right side of the circuit, thereby making the stage latency equal to that of one multiplication and one addition. The registers now hold:

$$
\begin{aligned}
u^{(i-1)} &= dx^{(i-1)} \\
v^{(i-1)} &= cx^{(i-1)} + dx^{(i-2)} \\
w^{(i-1)} &= bx^{(i-1)} + cx^{(i-2)} + dx^{(i-3)} \\
y^{(i-1)} &= ax^{(i-1)} + bx^{(i-2)} + cx^{(i-3)} + dx^{(i-4)}
\end{aligned}
$$

This alternate computation scheme allows a higher operating frequency at a given supply voltage or, alternatively, a lower supply voltage for a desired throughput. The effect of this transformation on the capacitance is difficult to predict and will depend on the detailed design and layout of the arithmetic elements.

26.6 SOME EMERGING METHODS

Asynchronous digital circuits have been studied for many years. Despite advantages in speed, distributed (localized) control, and built-in capability for pipelining, such circuits are not yet widely used. The only exceptions are found in bus handshaking protocols, interrupt handling mechanisms, and the design of certain classes of high-performance, special-purpose systems (wave front arrays). Localized connections and elimination of the clock distribution network may give asynchronous circuits an edge in power consumption. This, along with improvements in the asynchronous circuit design methodologies and reduced overhead may bring such circuits to the forefront in the design of general-purpose digital systems. However, before this happens, design/synthesis tools and testing methods must be improved.

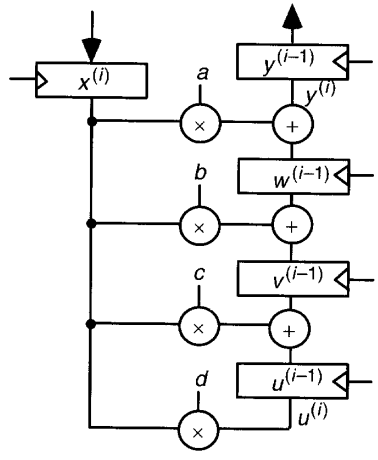


Fig. 26.12 Realization of the retimed fourth-order FIR filter.

In asynchronous circuits, timing information is embedded in, or travels along with, the data signals. Each function unit is activated when its input data becomes available and in turn notifies the next unit in the chain when its results are ready (Fig. 26.13). In the bundled data protocol, a "data ready" or "request" signal is added to a bundle of data lines to inform the receiver, which then uses an "acknowledge" line to release the sending module. In the two-rail protocol, each signal is individually encoded in a self-timed format using two wires. The latter approach essentially doubles the number of wires that go from module to module, but has the advantage of being completely insensitive to delay.

The best form of asynchronous design from the viewpoint of low power uses dual-rail data encoding with transition signaling: two wires are used for each signal, with a transition on one wire indicating that a 0 has arrived and a transition on the other designating the arrival of a 1. Level-sensitive signaling is also possible, but because the signal must return to 0 after each transaction, power consumption is higher.

Wave pipelining, discussed in Section 25.2, affects the power requirements for two reasons. One reason is that the careful balancing of delays within each stage, which is required for maximum performance, also tends to reduce glitching. A second, more important, reason is that in a wave-pipelined system, the same throughput can be achieved at a lower clock frequency. Like asynchronous circuit design, wave pipelining is not yet widely used. However, as problems with this method are better understood and automatic synthesis tools are developed, application of wave pipelining may become commonplace in the design of high-performance digital systems, with or without power considerations.

Clearly, the reduction of dynamic power dissipation in CMOS circuits, which was the focus of our discussions in this chapter, is not the only relevant criterion in dealing with low-power designs. Efforts in this area must deal with a spectrum of methods ranging from the architecture to the individual wires and transistors. Availability of more data on the power requirements of various arithmetic circuits and design styles [Call96] will help in this regard. Similarly, the development of better low-power synthesis and power estimation tools, which will allow the designers to experiment with various designs and fine-tune their parameters, will no doubt lead to greater applicability of these methods.

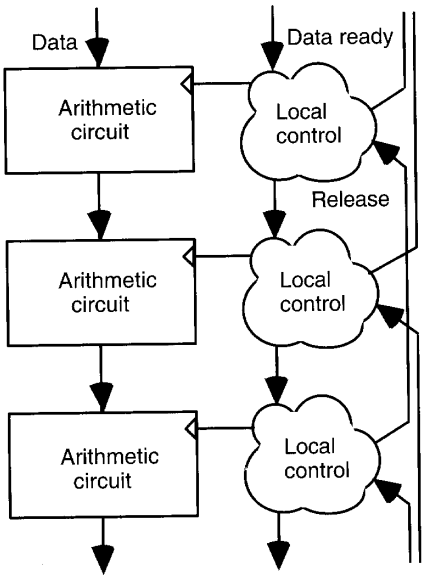


Fig. 26.13 Part of an asynchronous chain of computations.

PROBLEMS

26.1 Clock-related power dissipation Estimate the power dissipation associated with clock distribution in a 250 MHz processor chip operating at 3.3 V if the die dimensions are 1 cm $\times$ 1.3 cm, the length of the 1-μm-wide clock distribution network is roughly four times the die's perimeter, and the parasitic capacitance of the metal layer is 1 nF/mm^2. How will the power dissipation be affected if the chip's technology is scaled down by a factor of 1.4 in all dimensions, assuming that the supply voltage and frequency remain the same? *Hint:* Capacitance of a wire is directly proportional to its area and inversely proportional to the thickness of insulation.

26.2 Power implications of other optimizations Many of the methods considered in earlier chapters for increasing the operation speed or reducing the hardware cost have implications for power consumption. Furthermore, reduction in power consumption is not always in conflict with other optimizations.

a. Provide an example of a speed enhancement method that also reduces power.

b. Describe a speed enhancement method that substantially increases power.

c. Provide an example of a cost-saving method that also leads to reduced power.

d. Describe a cost reduction method that substantially increases power.

26.3 Saving power by operator reduction Consider the complex-number multiplication scheme discussed at the beginning of Section 26.3.

a. Can a similar method be applied to synthesizing a $2k \times 2k$ multiplier from $k \times k$ multipliers? Discuss.

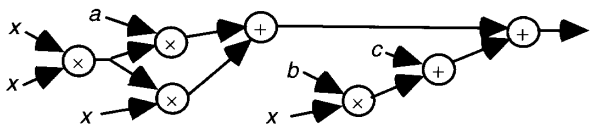
b. What about rotating a series of vectors by the constant angle θ using the familiar transformations $X = x \sin \theta + y \cos \theta$ and $Y = x \cos \theta - y \sin \theta$?

26.4 Saving power by operation reordering

a. The expression $u + 2^{-8}v + 2^{-10}w$, with the fixed-point fractional operands u, v, and w, is to be evaluated using two adders. What is the best order of evaluation from the standpoint of minimizing signal transitions? Does the best order depend on whether the numbers are signed?

b. Generalize the result of part a to the addition of n fractions, where the magnitude of the ith fraction is known to be in $[0, 2^{-m_i})$.

26.5 Saving power by reduction and reordering Rearrange the accompanying computation to reduce the power requirements. If more than one rearrangement is possible, compare them with respect to operation complexity (power), latency, and cost.



26.6 Saving power via delay balancing The array multiplier of Fig. 26.5 is different from the one shown in Fig. 11.13 and in some ways inferior to it. Compare the two designs with respect to worst-case delay and glitching, before and after the application of delay balancing.

26.7 Reduction of activity by bus-invert encoding Bus-invert encoding is a scheme whereby a single wire is added to a bus to designate polarity: a polarity of 0 indicates that the desired data is on the bus, whereas a polarity of 1 means that the complement of the desired data is being transmitted.

a. Draw a complete block diagram of this scheme, including all units needed on the sender and receiver sides.

b. Discuss the power-saving implications of this method. Then, using reasonable assumptions about the data, try to quantify the extent of savings achieved.

26.8 Saving power via precomputation

a. Apply the precomputation scheme of Fig. 26.6 to the design of a 32-bit integer comparator that determines whether $x > y$. Assume 2's-complement inputs and use the sign bit plus 2 magnitude bits for the precomputation. *Hint:* Invert the sign bits and compare as unsigned integers.

b. Repeat part a, this time assuming signed-magnitude inputs.

26.9 Power implications of pipelining

a. Suppose that in the design of Fig. 26.10, the three-operand adder is to be implemented by means of a pair of two-operand adders. The critical path of the circuit

will then become longer than that in the original circuit before unrolling. Show how circuit throughput can be maintained or improved by conversion into a two-stage pipeline.

b. Repeat part a for an implementation of the IIR filter of Fig. 26.9 that uses two steps of unrolling.

26.10 Parallelism and pipelining

a. Choose three convergence computation methods from among those discussed in Chapters 16, 21, and 23. Discuss opportunities that might exist for power savings in these computations through parallelism and/or pipelining.

b. Compare convergence and digit-recurrence methods with regard to their power requirements.

26.11 Arithmetic by table lookup In Chapter 24, we saw that table-lookup methods can be highly cost-effective for certain arithmetic computations.

a. What are the power consumption implications of arithmetic by table lookup?

b. Can you think of any power-saving method for use with tabular implementations?

26.12 A circuit technique for power reduction In CMOS circuit implementation of symmetric functions, such as AND, OR, or XOR, the logically equivalent input nodes may differ in their physical characteristics. For example, the inputs of a four-input AND gate may have different capacitances.

a. How is this observation relevant to the design of low-power arithmetic circuits?

b. Describe an application context for which this property may be used to reduce power. *Hint:* Look at the filter implementations of Section 26.5.

26.13 Power considerations in fast counters Consider the power consumption aspects of the fast counter designs of Section 5.5. Compare the designs with each other and with standard counters and discuss.

26.14 Bit-serial versus parallel arithmetic Study the power efficiency aspects of bit-serial, digit-serial, and bit-parallel arithmetic. What would be a good composite figure of merit incorporating speed, cost, and power?

26.15 Power implications of arithmetic methods Based on what you have learned in this chapter, identify power consumption implications, if any, of the following design choices. Justify your answers.

a. Multiplication with and without Booth's recoding.

b. Floating-point versus logarithmic number representation.

c. Restoring versus nonrestoring division or square-rooting.

26.16 Low-power division Contrast convergence and digit-recurrence division methods from the viewpoint of power consumption, and discuss power reduction strategies that might be applicable in each case. Begin by studying the approach taken in [Nann99].

REFERENCES

[Atha96] Athas, W.C., "Energy-Recovery CMOS," in *Low-Power Design Methodologies*, J.M. Rabaey and M. Pedram (eds.), Kluwer, 1996, pp. 65–100.

[Call96] Callaway, T.K., and E.E. Swartzlander, Jr., "Low Power Arithmetic Components," in *Low-Power Design Methodologies*, J.M. Rabaey and M. Pedram (eds.), Kluwer, 1996, pp. 161–200.

[Chan95] Chandrakasan, A.P., and R.W. Broderson, *Low Power Digital CMOS Design*, Kluwer, 1995.

[Nann99] Nannarelli, A., and T. Lang, "Low-Power Divider," *IEEE Trans. Computers,* Vol. 48, No. 1, pp. 2–14, 1999.

[Parh96] Parhi, K.K., and F. Catthoor, "Design of High-Performance DSP Systems," in *Emerging Technologies: Designing Low-Power Digital Systems*, R.K. Cavin III and W. Liu, eds., IEEE Press, pp. 447–507.

[Raba96] Rabaey, J.M., M. Pedram, and P.E. Landman, "Introduction," in *Low-Power Design Methodologies*, J.M. Rabaey and M. Pedram (eds.), Kluwer, 1996, pp. 1–18.

[Raba98] Rabaey, J.M. (ed.), "VLSI Design and Implementation Fuels the Signal Processing Revolution," *IEEE Signal Processing*, Vol. 15, No. 1, pp. 22–37, 1998.

[Saku95] Sakuta, T., W. Lee, and P. Balsara, "Delay Balanced Multipliers for Low Power/Low Voltage DSP Core," *Digest IEEE Symp. Low-Power Electronics*, 1995, pp. 36–37.

[Yeap98] Yeap, G., *Practical Low Power Digital VLSI Design*, Kluwer, 1998.

Chapter 27 FAULT-TOLERANT ARITHMETIC

Modern digital components are remarkably robust, but with a great many of them put together in a complex arithmetic system, things can and do go wrong. In data communication, a per-bit error probability of around 10^{-10} is considered quite good. However, at a rate of many millions of arithmetic operations per second, such an error probability in computations can lead to several bit-errors per second. While coding techniques are routinely applied to protect against errors in data transmission or storage, the same cannot be said about computations performed in an arithmetic circuit. In this chapter, we examine key methods that can be used to improve the robustness and reliability of arithmetic systems. Chapter topics include:

27.1 FAULTS, ERRORS, AND ERROR CODES

So far, we have assumed that arithmetic and logic elements always behave as expected: an AND gate always outputs the logical AND of its inputs, a table entry maintains its correct initial value, and a wire remains permanently connected. Even though modern integrated circuits are extremely reliable, faults (deviations from specified or correct functional behavior) do occur in the course of lengthy computations, especially in systems that operate under harsh environmental conditions, deal with extreme/unpredictable loads, or are used during long missions. The output of an AND gate may become permanently "stuck on 1," thus yielding an incorrect output when at least one input is 0. Or cross talk or external interference may cause the AND gate to suffer a "transient fault" in which its output becomes incorrect for only a few clock cycles. A table entry may become corrupt as a result of manufacturing imperfections in the memory cells or

logic faults in the read/write circuitry. Because of overheating, a VLSI manufacturing defect, or a combination of both, a wire may break or short-circit to another wire.

Ensuring correct functioning of digital systems in the presence of (permanent and transient) faults is the subject of the *fault-tolerant computing* discipline, also known as *reliable (dependable) computing* [Parh94]. In this chapter, we review some ideas in fault-tolerant computing that are particularly relevant to the computation of arithmetic functions.

Methods of detecting or correcting data errors have their origins in the field of communications. Early communications channels were highly unreliable and extremely noisy. So signals sent from one end were often distorted or changed by the time they reached the receiving end. The remedy, thought up by communications engineers, was to encode the data in redundant formats known as "codes" or "error codes." Examples of coding methods include adding a parity bit (an example of a single-error-detecting or SED code), checksums, and Hamming single-error-correcting, double-error-detecting (SEC/DED) code. Today, error-detecting and error-correcting codes are still used extensively in communications, for even though the reliability of these systems and noise reduction/shielding methods have improved enormously, so have the data rates and data transmission volumes, making the error probability nonnegligible.

Codes originally developed for communications can be used to protect against storage errors. When the early integrated-circuit memories proved to be less reliable than the then-common magnetic core technology, IC designers were quick to incorporate SEC/DED codes into their designs.

The data processing cycle in a system whose storage and memory-to-processor data transfers are protected by an error code can be represented as in Fig. 27.1. In this scheme, which is routinely applied to modern digital systems, the data manipulation part is unprotected. Decoding/encoding is necessary because common codes are not closed under arithmetic operations. For example, the sum of two even-parity numbers does not necessarily have even parity. As another example, when we change an element within a list that is protected by a checksum, we must compute a new checksum that replaces the old one.

One way to protect the arithmetic computation against fault-induced errors is to use duplication with comparison of the two results (for single fault/error detection) or triplication with 2-out-of-3 voting on the three results (for single fault masking or error correction). Figure 27.2 shows possible ways for implementing such duplication and triplication schemes.

In Fig. 27.2a, the decoding logic is duplicated along with the ALU, to ensure that a single fault in the decoder does not go undetected. The encoder, on the other hand, remains a critical element whose failure will lead to undetected errors. However, since the output of the encoder is redundant (coded), it is possible to design the encoding circuitry in a way that ensures the production of a non-codeword at its output if anything goes wrong. Such a design, referred to as *self-checking*, leads to error detection by the checker associated with the memory subsystem or later when the erroneous stored value is used as an input to the ALU. Assuming the use of a self-checking encoder, the duplicated design in Fig. 27.2a can detect any error resulting from a fault that is totally confined within one of the blocks shown in the diagram. This includes the "compare" block whose failure may produce a *false alarm*. An undetected mismatch would require at least two faults in separate blocks.

The design with triplicated ALU in Fig. 27.2b is similar. Here, the voter is a critical element and must be designed with care. Self-checking design cannot be applied to the voter (as used here), since its output is nonredundant. However, by combining the voting and encoding functions, one may be able to design an efficient self-checking voter-encoder. This *three-channel* computation strategy can be generalized to n channels to permit the tolerance of more faults. However, the cost overhead of a higher degree of replication becomes prohibitive.

Since the preceding replication schemes involve significant hardware overheads, one might attempt to apply coding methods for fault detection or fault tolerance within the ALU. The

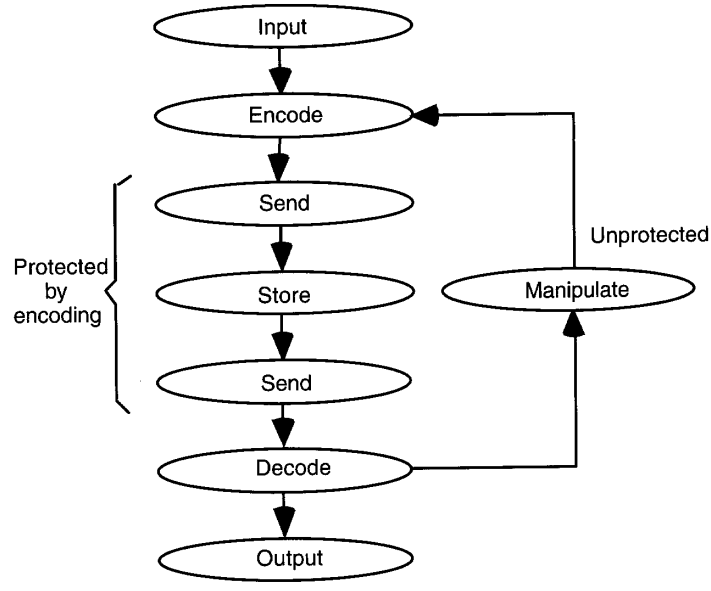


Fig. 27.1 A common way of applying information coding techniques.

first issue we encounter in trying to use this approach is that single, double, burst, and other error types commonly dealt with in communications do not provide useful characterizations for arithmetic. Whereas a spike due to noise may affect a single bit (random error) or a small number of consecutive bits (burst error), a single erroneous carry signal within an adder (caused, e.g., by a faulty gate in the carry logic) may produce an arbitrary number of bit inversions in the output. Figure 27.3 provides an example.

We see in the example of Fig. 27.3 that a single fault in the adder has caused 12 of the sum bits to be inverted. In coding theory parlance, we say that the *Hamming distance* between the correct and incorrect results is 12 or that the error has a *Hamming weight* (number of 1s in the XOR of the two values) of 12.

Error detection and correction capabilities of codes can be related to the minimum Hamming distance between *codewords* as exemplified by the following:

Single-error-detecting (SED)	Min. Hamming distance = 2
Single-error-correcting (SEC)	Min. Hamming distance = 3
SEC/DED	Min. Hamming distance = 4

For example, in the case of SED codes, any single-bit inversion in a codeword is guaranteed not to change it to another codeword, thus leading to error detection. For SEC, a single-bit inversion leads to an invalid word that is closer (in terms of Hamming distance) to the original correct codeword than to any other valid codeword, thus allowing for error correction.

From the addition example in Fig. 27.3, we see that even if some "single-error-detecting code" were closed under addition, it would be incapable of detecting the erroneous result in this case. We note, however, that in our example, the erroneous sum differs from the correct sum by 2^4. Since in computer arithmetic we deal with numbers as opposed to arbitrary bit strings, it is

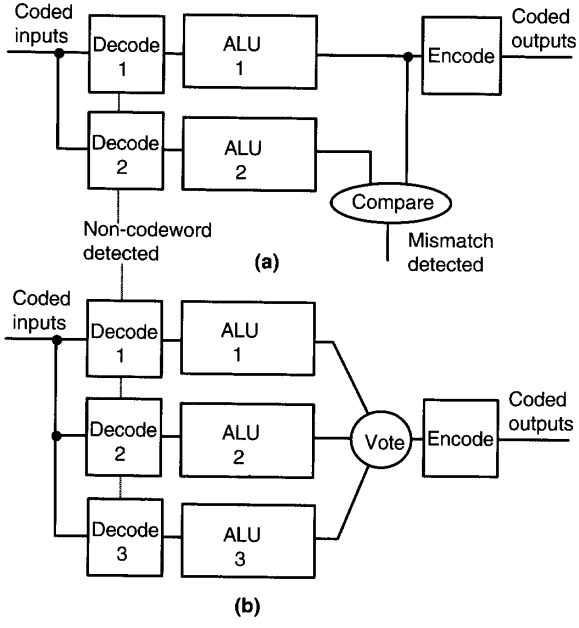


Fig. 27.2 Arithmetic fault detection or fault tolerance (masking) with replicated units.

the numerical difference between the erroneous and correct values that is of interest to us, not the number of bits in which they differ.

Accordingly, we define the *arithmetic weight* of an error as the minimum number of signed powers of 2 that must be added to the correct value to produce the erroneous result (or vice versa). Here are two examples:

Correct result	0111 1111 1111 0100	1101 1111 1111 0100
Erroneous result	1000 0000 0000 0100	0110 0000 0000 0100
Difference (error)	$16 = 2^4$	$-32752 = -2^{15} + 2^4$
Error, in minimum-weight BSD form	0000 0000 0001 0000	$^{-}$1000 0000 0001 0000
Arithmetic weight of the error	1	2
Type of error	Single, positive	Double, negative

Hence, the errors in the preceding examples can be viewed as "single" and "double" errors in the arithmetic sense. Special *arithmetic error codes* have been developed that are capable of detecting or correcting errors that are characterized by their arithmetic, rather than Hamming, weights. We review some such codes in Sections 27.2 and 27.3.

Note that a minimum-weight BSD representation of a k-bit error magnitude has at most $\lceil (k + 1)/2 \rceil$ nonzero digits and can always be written in *canonic BSD* form without any consecutive nonzero digits. The canonic form of a BSD number, which is unique, is intimately related to the notion of arithmetic error weight.

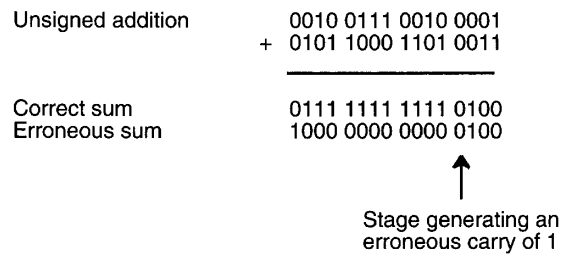


Fig. 27.3 How a single carry error can produce an arbitrary number of bit-errors (inversions) in the sum.

27.2 ARITHMETIC ERROR-DETECTING CODES

Arithmetic error-detecting codes:

1. Are characterized in terms of the arithmetic weights of detectable errors.
2. Allow us to perform arithmetic operations on coded operands directly.

The importance of the first property was discussed at the end of Section 27.1. The second property is crucial because it allows us to protect arithmetic computations against circuit faults with much lower hardware redundancy (overhead) than full duplication or triplication.

In this section, we discuss two classes of arithmetic error-detecting codes: product codes and residue codes. In both cases we will assume unsigned integer operands. Extension of the concepts to signed integers and arbitrary fixed-point numbers is straightforward. Codes for floating-point numbers tend to be more complicated and have received limited attention from arithmetic and fault tolerance researchers.

a. Product codes

In a *product code*, also known as *AN code*, a number N is represented as the product AN, where the check modulus A is a constant. Verifying the validity of an AN-coded operand requires checking its divisibility by A. For odd A, all weight-1 arithmetic errors (including all single-bit errors) are detected. Arithmetic errors of weight 2 and higher may not be detectable. For example, the error $32736 = 2^{15} - 2^5$ is not detectable with $A = 3$, 11, or 31, since the error magnitude is divisible by each of these check moduli.

Encoding/decoding of numbers with product codes requires multiplication/division by A. We will see shortly that performing arithmetic operations with product-coded operands also requires multiplication and division by A. Thus, for these codes to be practically viable, multiplication and division by the check modulus A should be simple. We are thus led to the class of *low-cost product codes* with check moduli of the form $A = 2^a - 1$.

Multiplication by $A = 2^a - 1$ is simple because it requires a shift and a subtract. In particular, if the computation is performed a bits at a time (i.e., digit-serially in radix 2^a), then one needs

only an a-bit adder, an a-bit register to store the previous radix-2^a digit, and a flip-flop for storing the carry. Division by $A = 2^a - 1$ is similarly simple if done a bits at a time. Given $y = (2^a - 1)x$, we find x by computing $2^a x - y$. The first term in this expression is unknown, but we know that it ends in a zeros. This is all that we need to compute the least significant a bits of x based on the knowledge of y. These computed bits of x form the next a bits of $2^a x$, allowing us to find the next a bits of x, etc.

Since $A = 2^a - 1$ is odd, low-cost product codes can detect any weight-1 arithmetic error. Some weight-2 and higher-weight errors may go undetected, but the fraction of such errors becomes smaller with an increase in A. Unidirectional errors, in which all erroneous bits are 0-to-1 or 1-to-0 inversions (but not both), form an important class of errors in VLSI implementations. For unidirectional errors, the error magnitude is the sum of several powers of 2 with the same signs.

THEOREM 27.1 Any unidirectional error with arithmetic weight not exceeding $a - 1$ is detectable by a low-cost product code that uses the check modulus $A = 2^a - 1$.

For example, the low-cost product code with $A = 15$ can detect any weight-2 or weight-3 unidirectional arithmetic error in addition to all weight-1 errors. The following are examples of weight-2 and weight-3 unidirectional errors that are detectable because the resulting error magnitude is not a multiple of 15:

$$\begin{aligned} 8 + 4 &= 12 \\ 128 + 4 &= 132 \\ 16 + 4 + 2 &= 22 \\ 256 + 16 + 2 &= 274 \end{aligned}$$

Product codes are examples of nonseparate, or nonseparable, codes in which the original data and the redundant information for checking are intermixed. In other words, the original number N is not immediately apparent from inspecting its encoded version AN but must be obtained through decoding (in this case, division by the check modulus A).

Arithmetic operations on product-coded operands are quite simple. Addition or subtraction is done directly, since:

$$Ax \pm Ay = A(x \pm y)$$

Direct multiplication results in:

$$Aa \times Ax = A^2 ax$$

So the result must be corrected through division by A. For division, if $z = qd + s$, with q being the quotient and s the remainder, we have:

$$Az = q(Ad) + As$$

So, direct division yields the quotient q along with the remainder As. The remainder is thus obtained in encoded form, but the resulting quotient q must be encoded via multiplication

by A. Because q is obtained in nonredundant form, an error occurring in its computation will go undetected. To keep the data protected against errors in the course of the division process, one can premultiply the dividend Az by A and then divide A^2z by Ad as usual. The problem with this approach is that the division leads to a quotient q^* and remainder s^* satisfying

$$A^2z = q^*(Ad) + s^*$$

which may be different from the expected results Aq and A^2s (the latter needing correction through division by A). Since q^* can be larger than Aq by up to $A - 1$ units, the quotient and remainder obtained from normal division may need correction. However, this again raises the possibility of undetected errors in the handling of the unprotected value q^*, which is not necessarily a multiple of A.

A possible solution to the preceding problem, when one is doing the division a bits at a time for $A = 2^a - 1$, is to adjust the last radix-2^a digit of q^* in such a way that the adjusted quotient q^{**} becomes a multiple of A. This can be done rather easily by keeping a modulo-A checksum of the previous quotient digits. One can prove that suitably choosing the last radix-2^a digit of q^{**} in $[-2^a + 2, 1]$ is sufficient to correct the problem. A subtraction is then needed to convert q^{**} to standard binary representation. Details can be found elsewhere [Aviz73].

Square-rooting leads to a problem similar to that encountered in division. Suppose that we multiply the radicand Az by A and then use a standard square-rooting algorithm to compute:

$$\lfloor\sqrt{A^2x}\rfloor = \lfloor A\sqrt{x}\rfloor$$

Since the preceding result is in general different from the correct result $A\lfloor\sqrt{x}\rfloor$, there is a need for correction. Again, the computed value $\lfloor A\sqrt{x}\rfloor$ can exceed the correct root $A\lfloor\sqrt{x}\rfloor$ by up to $A - 1$ units. So, the same correction procedure suggested for division is applicable here as well.

b. Residue codes

In a *residue code*, an operand N is represented by a pair of numbers $(N, C(N))$, where $C(N) = N \bmod A$ is the check part. The check modulus A is a constant. Residue codes are examples of *separate* or *separable* codes in which the data and check parts are not intermixed, thus making decoding trivial. Encoding a number N requires the computation of $C(N) = N \bmod A$, which is attached to N to form its encoded representation $(N, C(N))$.

As in the case of product codes, we can define the class of *low-cost residue codes*, with $A = 2^a - 1$, for which the encoding computation $N \bmod A$ is simple: it requires that a-bit segments of N be added modulo $2^a - 1$ (using an a-bit adder with end-around carry). This can be done digit-serially by using a single adder or in parallel by using a binary tree of a-bit 1's-complement adders.

Arithmetic operations on residue-coded operands are quite simple, especially if a low-cost check modulus $A = 2^a - 1$ is used. Addition or subtraction is done by operating on the data parts and check parts separately. That is:

$$(x, C(x)) \pm (y, C(y)) = (x \pm y, (C(x) \pm C(y)) \bmod A)$$

Hence, as shown in Fig. 27.4, an arithmetic unit for residue-coded operands has a main adder for adding/subtracting the data parts and a small modulo-A adder to add/subtract the residue checks. To detect faults within the arithmetic unit, the output of this small modular adder (check processor) is compared to the residue of the output from the main adder.

Multiplication of residue-coded operands is equally simple, since:

$$(a, C(a)) \times (x, C(x)) = (a \times x, (C(a) \times C(x)) \bmod A)$$

So, again, the structure shown in Fig. 27.4 is applicable. This method of checking the multiplication operation is essentially what we do when we verify the correctness of our pencil-and-paper multiplication result by casting out nines.

Just as in RNS, division and square-rooting are complicated with residue-coded operands. For these operations, the small residue check processor cannot operate independently from the main processor and must interact with it to compute the check part of the result. Details are beyond the scope of this chapter.

As in product codes, choosing any odd value for A guarantees the detection of all weight-1 arithmetic errors with residue codes. However, residue codes are less capable than product codes for detecting multiple unidirectional errors. For example, we saw earlier that the $15N$ code can detect all weight-2 and weight-3 unidirectional arithmetic errors. The residue code with $A = 15$ cannot detect the weight-2 error resulting from 0-to-1 inversion of the least significant bit of the data as well as the least significant bit of the residue. This error goes undetected because it adds 1 to the data as well as to the residue, making the result a valid codeword.

To correct the preceding problem, *inverse residue codes* have been proposed for which the check part represents $A - (N \bmod A)$ rather than $N \bmod A$. In the special case of $A = 2^a - 1$, the check bits constitute the bitwise complement of $N \bmod A$. Unidirectional errors now affect the data and check parts in opposite directions, making their detection more likely. By noting that attachment of the a-bit inverse residue $C'(N) = A - (N \bmod A)$ to the least significant end of a k-bit number N makes the resulting $(k + a)$-bit number a multiple of $A = 2^a - 1$, the following result is easily proven.

THEOREM 27.2 Any unidirectional error with arithmetic weight not exceeding $a - 1$ is detectable by a low-cost inverse residue code that uses the check modulus $A = 2^a - 1$.

The added cost or overhead of an error-detecting code has two components:

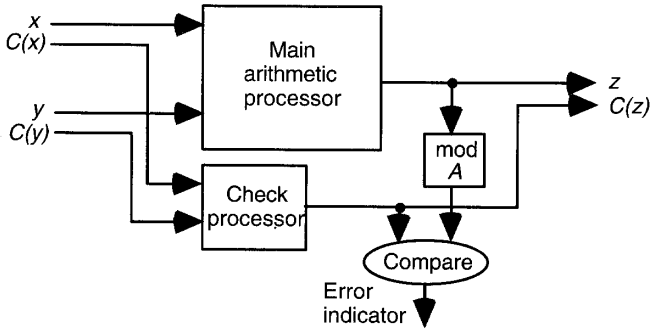


Fig. 27.4 Arithmetic processor with residue checking.

The increased word width for coded operands adds to the cost of registers, memory, and data links.

Checked arithmetic or wider operands make the ALU more complex.

With respect to the first component of cost, product, residue, and inverse residue codes are similar. For example, the low-cost versions of these codes with the check modulus $A = 2^a - 1$ all require a additional bits to represent the coded operands. With regard to arithmetic, residue and inverse residue codes are simpler than product codes for addition and multiplication and more complex for division.

It is interesting to note that the residue-class codes are the only possible separable codes for checking an adder [Pete58]. Also, it has been proven that bitwise logical operations such as AND, OR, and XOR, cannot be checked by any coding scheme with less than 100% redundancy; that is, the best we can do for error detection in logical operations is duplication and comparison [Pete59].

27.3 ARITHMETIC ERROR-CORRECTING CODES

We illustrate the main ideas relating to arithmetic error-correcting codes by way of examples from the class of *biresidue codes*. A biresidue code represents a number N as the triple $(N, C(N), D(N))$, where the check components $C(N) = N \bmod A$ and $D(N) = N \bmod B$ are residues with respect to the check moduli A and B. If the original number requires k bits for its binary representation, its biresidue-coded representation would need $k + \lceil \log_2 A \rceil + \lceil \log_2 B \rceil$ bits.

Encoding for the class of biresidue codes is similar to that of single-residue codes, except that two residues must be computed. Addition and multiplication of biresidue-coded operands can be performed by an arithmetic processor similar to that shown in Fig. 27.4, but with two check processors. Since the two residues can be computed and checked in parallel, no speed is lost.

Consider errors that affect the number N or only one of the residues, say $C(N)$. Such errors can be corrected as follows.

Error in C(N). In this case, $C(N)$ will fail the residue check, while $D(N)$ passes its check; $C(N)$ can then be corrected by recomputing $N \bmod A$.

Error in N. Unless the error magnitude happens to be a multiple of A and/or B (thus being either totally undetectable or else indistinguishable from a residue error), both residue checks will fail, thus pointing to N as the erroneous component. To correct such errors, the differences between $N_{\text{wrong}} \bmod A$ ($N_{\text{wrong}} \bmod B$) and $C(N)$ ($D(N)$) must be noted. The two differences, $[(N_{\text{wrong}} \bmod A) - C(N)] \bmod A$ and $[(N_{\text{wrong}} \bmod B) - D(N)] \bmod B$, constitute an *error syndrome*. The error is then correctable if the syndromes for different errors are distinct.

Consider, as an example, a biresidue code with the low-cost check moduli $A = 7$ and $B = 15$. Table 27.1 shows that any weight-1 arithmetic error E with $|E| \leq 2048$ leads to a unique error syndrome, thus allowing us to correct it by subtracting the associated error value from N_{wrong}. For $|E| \geq 4096$, the syndromes assume the same values as for $E/4096$. Hence, weight-1 error correction is guaranteed only for a 12-bit data part. Since the two residues require a total of 7 bits for their representations, the redundancy for this biresidue code is $7/12 \approx 58\%$.

TABLE 27.1
Error syndrome s for weight-1 arithmetic errors in the (7, 15) biresidue code

Positive Error	Error syndrome		Negative error	Error syndrome	
	mod 7	mod 15		mod 7	mod 15
1	1	1	−1	6	14
2	2	2	−2	5	13
4	4	4	−4	3	11
8	1	8	−8	6	7
16	2	1	−16	5	14
32	4	2	−32	3	13
64	1	4	−64	6	11
128	2	8	−128	5	7
256	4	1	−256	3	14
512	1	2	−512	6	13
1024	2	4	−1024	5	11
2048	4	8	−2048	3	7
4096	1	1	−4096	6	14
8192	2	2	−8192	5	13
16384	4	4	−16384	3	11
32768	1	8	−32768	6	7

A product code with the check modulus $A \times B = 7 \times 15 = 105$ would similarly allow us to correct weight-1 errors via checking the divisibility of the codeword by 7 and 15 and noting the remainders. This is much less efficient, however, since the total word width must be limited to 12 bits for full error coverage. The largest representable number is thus $4095/105 = 39$. This is equivalent to about 5.3 bits of data, leading to a redundancy of 127%.

In general, a biresidue code with relatively prime low-cost check moduli $A = 2^a - 1$ and $B = 2^b - 1$ can support a data part of ab bits for weight-1 error correction with a representational redundancy of $(a+b)/(ab) = 1/a + 1/b$. Thus, with a choice of suitably large values for a and b, the redundancy can be kept low.

Based on our discussion of arithmetic error-detecting and error-correcting codes, we conclude that such codes are effective not only for protecting against fault-induced errors during arithmetic computations but also for dealing with storage and transmission errors. Using a single code throughout the system obviates the need for frequent encoding and decoding, and minimizes the chance of data corruption during the handling of unencoded data.

27.4 SELF-CHECKING FUNCTION UNITS

A self-checking function unit can be designed with or without encoded inputs and outputs. For example, if in Fig. 27.4, x mod A and y mod A are computed internally, as opposed to being supplied as inputs, a self-checking arithmetic unit with unencoded input/output is obtained.

The theory of self-checking logic design is quite well developed and can be used to implement highly reliable, or at least fail-safe, arithmetic units. The idea is to design the required logic circuits in such a way that any fault, from a prescribed set of faults which we wish to protect

against, either does not affect the correctness of the outputs (is *masked*) or else leads to a non-codeword output (is made *observable*). In the latter case, the invalid result is either detected immediately by a code checker attached to the unit's output or else is propagated downstream by the next self-checking module that is required to produce a non-codeword output for any non-codeword input it receives (somewhat similar to computation with NaNs in floating-point arithmetic).

An important issue in the design of such self-checking units is the ability to build self-checking code checkers that are guaranteed not to validate a non-codeword despite internal faults. For example, a self-checking checker for an inverse residue code $(N, C'(N))$ might be designed as follows. First, $N \bmod A$ is computed. If the input is a valid codeword, this computed value must be the bitwise complement of $C'(N)$. We can view the process of verifying that $x_{b-1} \cdots x_1 x_0$ is the bitwise complement of $y_{b-1} \cdots y_1 y_0$ as that of ensuring that the signal pairs (x_i, y_i) are all (1, 0) or (0, 1). This amounts to computing the logical AND of a set of Boolean values that are represented using the following 2-bit encoding:

$$\begin{array}{lll} 1 & \text{encoded as} & (1, 0) \text{ or } (0, 1) \\ 0 & \text{encoded as} & (0, 0) \text{ or } (1, 1) \end{array}$$

Note that the code checker produces two outputs that carry (1, 0) or (0, 1) if the input is correct and (0, 0) or (1, 1) if it is not. It is an easy matter to design the required AND circuit such that no single gate or line fault leads to a (1, 0) or (0, 1) output for a non-codeword input. For example, one can build an AND tree from the two-input AND circuit shown in Fig. 27.5. Note that any code checker that has only one output line cannot be self-checking, since a single stuck-at fault on its output line can produce a misleading result.

Fault detection can also be achieved by *result checking*. This is similar to what, in the field of software fault tolerance, is known as *acceptance testing*. An acceptance test is a (hopefully simple) verification process. For example, the correct functioning of a square-rooter can be verified by squaring each obtained root and comparing the result to the original radicand. If we assume that any error in the squaring process is independent from, and thus unlikely to compensate for, errors in the square-rooting process, a result that passes the verification test is correct with very high probability.

Acceptance tests do not have to be perfect. A test with *imperfect coverage* (e.g., comparing residues) may not detect each fault immediately after it occurs, but over time will signal a

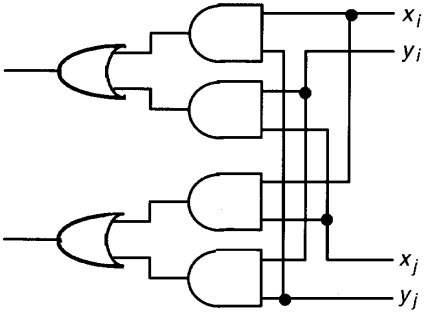


Fig. 27.5 Two-input AND circuit, with 2-bit inputs (x_i, y_i) and (x_j, y_j), for use in a self-checking code checker.

malfunctioning unit with high probability. On the other hand, if we assume that faults are permanent and occur very rarely, then periodic, as opposed to concurrent or on-line, verification might be adequate for fault detection. Such periodic checks might involve computing with several random operands and verifying the correctness of the results to make it less likely for compensating errors to render the fault undetectable [Blum96].

27.5 ALGORITHM-BASED FAULT TOLERANCE

So far, our focus has been on methods that allow us to detect and/or correct errors at the level of individual basic arithmetic operations such as addition and multiplication. An alternative strategy is to accept that arithmetic operations may yield incorrect results and build the mechanisms for detecting or correcting errors at the data structure or application level.

As an example of this approach, consider the multiplication of matrices X and Y yielding the result matrix P. The checksum of a list of numbers (a vector) is simply the algebraic sum of all the numbers modulo some check constant A. For any $m \times n$ matrix M, we define the row-checksum matrix M_r as an $m \times (n+1)$ matrix that is identical to M in its columns 0 through $n-1$ and has as its nth column the respective row checksums. Similarly, the column-checksum matrix M_c is an $(m+1) \times n$ matrix that is identical to M in its rows 0 through $m-1$ and has as its mth row the respective column checksums. The full-checksum matrix M_f is defined as the $(m+1) \times (n+1)$ matrix $(M_r)_c$: that is, the column-checksum matrix of the row-checksum matrix of M. Figure 27.6 shows a 3×3 matrix with its row, column, and full checksum matrices, where the checksums are computed modulo $A = 8$.

The following result allows us to detect and/or correct computation errors in matrix multiplication.

THEOREM 27.3 For matrices X, Y, and P satisfying $P = X \times Y$, we have $P_f = X_c \times Y_r$.

According to Theorem 27.3, we can perform standard matrix multiplication on the encoded matrices X_c and Y_r, and then compare the values in the last column and row of the product matrix to checksums that are computed based on the remaining elements to detect any error that may have occurred. If matrix elements are floating-point numbers, the equalities will hold

$$M = \begin{bmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{bmatrix} \qquad M_r = \begin{bmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{bmatrix}$$

$$M_c = \begin{bmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{bmatrix} \qquad M_f = \begin{bmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{bmatrix}$$

Fig. 27.6 A 3×3 matrix M with its row, column, and full checksum matrices M_r, M_c, and M_f.

approximately, leading to difficulties in selecting a suitable threshold for considering values equal. Some methods to resolve this problem are given in [Dutt96].

The full-checksum matrix M_f is an example of a *robust data structure* for which the following properties of error detection and correction hold.

THEOREM 27.4 In a full-checksum matrix, any single erroneous element can be corrected and any three erroneous elements can be detected.

Thus, for highly localized fault-induced errors (e.g., arising from a very brief transient fault in a hardware multiplier affecting no more than three elements of the product matrix), the preceding scheme allows for error correction or detection. Detection of more extensive errors, though not guaranteed, is quite likely; it would indeed be improbable for several errors to be compensatory in such a way that they escape detection by any of the checksums.

Designing such robust data structures with given capabilities of error detection and/or correction, such that they also lend themselves to direct manipulation by suitably modified arithmetic algorithms, is still an art. However, steady progress is being made in this area.

27.6 FAULT-TOLERANT RNS ARITHMETIC

Redundant encodings can be used with any number representation scheme to detect or correct errors. Residue number systems, in particular, allow very elegant and effective error detection and correction schemes through the use of redundant residues corresponding to extra moduli.

Suppose we choose the set of moduli in an RNS in such a way that one residue is redundant (i.e., if we remove any one modulus, the remaining moduli are adequate for the desired dynamic range). Then, any error that is confined to a single residue will be detectable, since such an error would make the affected residue inconsistent with the others. If this scheme is to work, the redundant modulus obviously must be the largest one (say m). The error detection scheme is thus as follows. Use all other residues to compute the residue of the number mod m. This is done by a process known as *base extension* for which many algorithms exist. Then compare the computed mod-m residue with the mod-m residue in the number representation to detect a possible error.

The beauty of this method is that arithmetic algorithms are totally unaffected; error detection is made possible by simply extending the dynamic range of the RNS. The base extension operation needed for error detection is frequently provided in an RNS processor for other reasons—for example, as a building block for synthesizing different RNS operations. In such a case, no additional hardware, beyond that required to handle the extra residue, is needed for error detection. In fact, it is possible to disable the error-checking capabilities and use the extended dynamic range offered by all the moduli when performing less critical computations.

Providing multiple redundant residues can lead to the detection of more errors and/or correction of certain error classes [Etze80] in a manner similar to the error-correction property of biresidue and multiresidue codes of Section 27.3. Again, the only new elements that are needed are the checking algorithms and the corresponding hardware structures. The arithmetic algorithms do not change.

As an example, consider adding the two redundant moduli 13 and 11 to the RNS with the four moduli 8, 7, 5, 3 (dynamic range = 840). In the resulting 6-modulus redundant RNS, the number 25 is represented as (12, 3, 1, 4, 0, 1). Now suppose that the mod-7 residue is corrupted and the number becomes (12, 3, 1, 6, 0, 1). Using base extension, we compute the two redundant residues from the other four residues; that is, we transform (−, −, 1, 6, 0, 1) to (5, 1, 1, 6, 0, 1). The difference between the first two components of the original corrupted number and the reconstructed number is (+7, +2), which is the error syndrome that points to a particular residue in need of correction. We see that the error correction scheme here is quite similar to that shown in Table 27.1 for a biresidue code.

PROBLEMS

27.1 Voting on integer results One way to design the voter shown in Fig. 27.2 is to use a three-input majority circuit (identical in function to the carry-out of a full adder) and do serial bitwise voting on the outputs of the three ALUs. Assume that the ALU outputs are 8-bit unsigned integers.

a. Show that serial bitwise voting produces the correct voting result, given at most one faulty ALU.

b. What would the output of the bit-serial voter be if its inputs are 15, 19, and 38?

c. Present the design a bit-serial voter that can indicate the absence of majority agreement, should a situation similar to the one in part b arise.

27.2 Approximate voting Suppose that the three-input voter shown in Fig. 27.2 is to interpret its 32-bit unsigned inputs as fractional values that may contain small computational errors (possibly a different amount for each input).

a. Provide a suitable definition of majority agreement in this case.

b. Can a bit-serial voter, producing its output on the fly, be designed in accordance with the definition of part a?

c. Design a bit-serial median voter that outputs the middle value among its three imprecise inputs.

d. Under what conditions is the output of a median voter the same as that of a majority voter?

27.3 Design of comparators For the two-channel redundant arrangement of Fig. 27.2, discuss the design of bit-serial comparators for integer (exact) and fractional (approximate) results.

27.4 Arithmetic weight

a. Prove that any minimal-weight binary signed-digit (BSD) representation of a k-bit binary number has at most $\lceil (k+1)/2 \rceil$ nonzero digits and can always be written in *canonic BSD* form without any consecutive nonzero digits.

b. Show that the arithmetic weight of a binary number x is the same as the Hamming distance between the binary representations of x and $3x$.

27.5 Low-cost product codes

a. Prove Theorem 27.1 characterizing the unidirectional error-detecting power of low-cost product codes.

b. What fraction of random double-bit errors are detectable by a low-cost product code with $A = 2^a - 1$?

c. Can moduli of the form $A = 2^a + 1$ be included in low-cost product codes?

27.6 Low-cost residue codes

a. Prove Theorem 27.2, which characterizes the unidirectional error-detecting power of low-cost inverse residue codes.

b. What fraction of random double-bit errors is detectable by a low-cost residue code with the check modulus $A = 2^a - 1$?

c. Repeat part b for low-cost inverse residue codes.

d. Show how the computation of the modulo-$(2^a - 1)$ residue of a number can be speeded up by using a tree of carry-save adders rather than a tree of a-bit adders with end-around carries.

e. Apply your method of part d to the computation of the mod-15 residue of a 32-bit number and compare the result with respect to speed and cost to the alternative approach.

f. Suggest an efficient method for computing the modulo-17 residue of a 32-bit number and generalize it to the computation of mod-$(2^a + 1)$ residues.

27.7 Division with product-coded operands Show that if q and s are the quotient and remainder in dividing z by d (i.e., $z = qd + s$) and $A = 2^a - 1$, then in dividing $A^2 z$ by Ad, the obtained quotient q^{**} can always be made equal to Aq by choosing the last radix-2^a digit of q^{**} in $[-2^a + 2, 1]$.

27.8 Low-cost biresidue codes

a. Characterize the error correction capability of a (7, 3) low-cost biresidue code.

b. If only error detection is required, how much more effective is the (7, 3) biresidue code compared to a single-residue code with the check modulus 7? Would you say that the additional redundancy due to the second check modulus 3 is worth its cost?

c. Propose a low-cost biresidue code that is capable of correcting all weight-1 arithmetic errors in data elements that are 32 bits wide.

27.9 Self-checking checkers

a. Verify that the AND circuit of Fig. 27.5 is an optimal implementation of the desired functionality. Note that the specification of the design has "coupled don't-cares": that is, one output of the AND circuit can be 0 or 1 provided that the other one is (not) equal to it.

b. Verify that the AND circuit of Fig. 27.5 is self-testing in the sense that both output combinations (0, 1) and (1, 0) appear during normal operation when there is no input error. Note that if a self-checking checker produces only the output (1, 0), say, during normal operation, some output stuck-at faults may go undetected.

c. Use the AND circuit of Fig. 27.5 to construct a self-checking circuit to check the validity of a 10-bit integer that has been encoded in the low-cost product code with the check modulus $A = 3$.

d. Design the OR-circuit and NOT-circuit (inverter) counterparts to the AND circuit of Fig. 27.5. Discuss whether these additional circuits could be useful in practice.

27.10 Self-checking function unit Present the complete design a self-checking additive multiply module (AMM) using the low-cost product code with $A = 3$. The two additive and two multiplicative inputs, originally 4-bit unsigned numbers, are presented in 6-bit encoded form, and the encoded output is 10 bits wide. Analyze the speed and cost overhead of your self-checking design.

27.11 Self-checking arithmetic circuits Consider the design of self-checking arithmetic circuits using two-rail encoding of the signals: 0 represented as (0, 1) and 1 as (1, 0), with (0, 0) and (1, 1) signaling an error.

a. Design a two-rail self-checking full-adder cell. *Hint:* Think of how two-rail AND, OR, and NOT elements might be built.

b. Using the design of an array multiplier as an example, compare the two-rail self-checking design approach to circuit duplication with comparison. Discuss.

27.12 Algorithm-based fault tolerance

a. Verify that the product of the matrices M_c and M_r of Fig. 27.6 yields the full checksum matrix $(M^2)_f$ if the additions corresponding to the checksum elements are performed modulo 8.

b. Prove Theorem 27.3 in general.

c. Construct an example showing that the presence of four erroneous elements in the full checksum matrix M_f can go undetected. Then, prove Theorem 27.4.

27.13 Algorithm-based fault tolerance Formulate an algorithm-based fault tolerance scheme for multiplying a matrix by a vector and discuss its error detection and correction characteristics.

27.14 Redundant RNS representations For the redundant RNS example presented at the end of Section 27.6 (original moduli 8, 7, 5, 3; redundant moduli 13, 11):

a. What is the redundancy with binary-encoded residues? How do you define the redundancy?

b. Construct a syndrome table similar to Table 27.1 for single-residue error correction.

c. Show that all double-residue errors are detectable.

d. Explain whether, and if so, how, one can detect double-residue errors and correct single-residue errors at the same time.

27.15 Redundant RNS representations

a. Prove or disprove: In an RNS having a range approximately equal to that of k-bit numbers, any single-residue error can be detected with $O(\log k)$ bits of redundancy.

b. Repeat part a for single-residue error correction.

27.16 BSD adder with parity checking Show how a binary signed-digit adder can be designed to always produce an output word with even parity. Discuss the fault tolerance capabilities of the resulting adder. *Hint*: If one of the three digit values in $[-1, 1]$ is

assigned two 2-bit codes with odd and even parities, it is possible to encode pairs of output digits so that the resulting 4 bits have even parity [Thor97].

REFERENCES

[Aviz72] Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Trans. Computers*, Vol. 20, No. 11, pp. 1322–1331, 1971.

[Aviz73] Avizienis, A., "Algorithms for Error-Coded Operands," *IEEE Trans. Computers*, Vol. 22, No. 6, pp. 567–572, 1973.

[Blum96] Blum, M., and H. Wasserman, "Reflections on the Pentium Division Bug," *IEEE Trans. Computers*, Vol. 45, No. 4, pp. 385–393, 1996.

[Dutt96] Dutt, S., and F.T. Assaad, "Mantissa-Preserving Operations and Robust Algorithm-Based Fault Tolerance for Matrix Computations," *IEEE Trans. Computers*, Vol. 45, No. 4, pp. 408–424, 1996.

[Etze80] Etzel, M.H., and W.K. Jenkins, "Redundant Residue Number Systems for Error Detection and Correction in Digital Filters," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. 28, No. 5, pp. 538–545, October 1980.

[Huan84] Huang, K.H., and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers*, Vol. 33, No. 6, pp. 518–528, 1984.

[Parh78] Parhami, B., and A. Avizienis, "Detection of Storage Errors in Mass Memories Using Arithmetic Error Codes," *IEEE Trans. Computers*, Vol. 27, pp. 302–308, April 1978.

[Parh94] Parhami, B., "A Multi-Level View of Dependable Computing," *Computers and Electrical Engineering*, Vol. 20, No. 4, pp. 347–368, 1994.

[Pete58] Peterson, W.W., "On Checking an Adder," *IBM J. Research and Development*, Vol. 2, No. 2, pp. 166–168, April 1958.

[Pete59] Peterson, W.W., and M.O. Rabin, "On Codes for Checking Logical Operations," *IBM J. Research and Development*, Vol. 3, No. 2, pp. 163–168, April 1959.

[Rao74] Rao, T.R.N., *Error Codes for Arithmetic Processors*, Academic Press, 1974.

[Thor97] Thornton, M.A., "Signed Binary Addition Circuitry with Inherent Even Parity Output," *IEEE Trans. Computers*, Vol. 46, No. 7, pp. 811–816, 1997.

Chapter 28 PAST, PRESENT, AND FUTURE

In this last chapter, we present a few interesting and diverse case studies that show the applications of some of the algorithms and implementation techniques studied thus far in the context of computational requirements, technological constraints, and overall design goals. We also take a look backward and forward, both to provide some historical perspective and to gauge the current trends and future directions of computer arithmetic. Chapter topics include:

28.1 HISTORICAL PERSPECTIVE

The history of computer arithmetic is intertwined with that of digital computers. Much of this history can be traced through a collection of key papers [Swar90] in the field, some of which are not easily accessible in the original form. Certain ideas used in computer arithmetic have their origins in the age of mechanical calculators. In fact Charles Babbage is said to have been aware of ideas such as carry-skip addition, carry-save addition, and restoring division [Omon94].

In the 1940s, machine arithmetic was a crucial element in efforts to prove the feasibility of computing with stored-program electronic devices. Hardware mechanisms for addition, use of complement representation to facilitate subtraction, and implementation of multiplication and division through shift/add algorithms were developed and fine-tuned early on. A seminal report in the initial development of stored-program electronic digital computers by A. W. Burkes, H. H. Goldstein, and J. von Neumann [Burk46] contained interesting ideas on arithmetic algorithms and their hardware realizations, including choice of number representation radix, distribution

of carry-propagation chains, fast multiplication via carry-save addition, and restoring division. The state of computer arithmetic circa 1950 is evident from an overview paper by R. F. Shaw [Shaw50].

Early stored-program digital computers were primarily number-crunching machines with limited storage and I/O capabilities. Thus, the bulk of design effort was necessarily expended on cost-effective realization of the instruction sequencing and arithmetic/logic functions. The 1950s brought about many important advances in computer arithmetic. With the questions of feasibility already settled, the focus now shifted to algorithmic speedup methods and cost-effective hardware realizations. By the end of the decade, virtually all important fast adder designs had already been published or were in the final phases of development. Similarly, the notions of residue arithmetic, SRT division, and CORDIC algorithms were all proposed and implemented in the 1950s. An overview paper by O.L. MacSorley [MacS61] contains a snapshot of the state of the art circa 1960.

Computer arithmetic advances continued in the 1960s with the introduction of tree multipliers, array multipliers, high-radix dividers, convergence division, redundant signed-digit arithmetic, and implementation of floating-point arithmetic operations in hardware or firmware (in microprogram). A by-product of microprogrammed control, which became prevalent for flexibility and economy of hardware implementations, was that greater arithmetic functionality could be incorporated into even the smallest processors by means of using standardized word widths across a whole range of machines with different computing powers.

Some of the most innovative ideas originated from the design of early supercomputers in the 1960s, when the demand for high performance, along with the still high cost of hardware, led designers to novel solutions that made high-speed machine arithmetic quite cost-effective. Striking examples of design ingenuity can be found in the arithmetic units of the IBM System/360 Model 91 [Ande67] and CDC 6600 [Thor70]. Other digital systems of the pre-IC era no doubt contained interesting design ideas, but the IBM and CDC systems were extensively documented in the open technical literature, making them excellent case studies. It is quite regrettable that today's designs are not described in the technical literature with the same degree of openness and detail. We briefly discuss the design of the floating-point execution unit of IBM System/360 Model 91 in Section 28.2. From this case study, we can deduce that the state of computer arithmetic was quite advanced in the mid-1960s.

As applications of computers expanded in scope and significance, faster algorithms and more compact implementations were sought to keep up with the demand for higher performance and lower cost. The 1970s are distinguished by the advent of microprocessors and vector supercomputers. Early LSI chips were quite limited in the number of transistors or logic gates they could accommodate; thus microprogrammed implementation was a natural choice for single-chip processors, which were not yet expected to offer high performance. At the high end of performance spectrum, pipelining methods were perfected to allow the throughput of arithmetic units to keep up with computational demand in vector supercomputers. In Section 28.3, we study the design of one such vector supercomputer, the Cray X-MP/Model 24.

Widespread application of VLSI circuits in the 1980s triggered a reconsideration of virtually all arithmetic designs in light of interconnection cost and pin limitations. For example, carry-lookahead adders, which appeared to be ill-suited to VLSI implementation, were shown to be efficiently realizable after suitable modifications. Similar ideas were applied to more efficient VLSI implementation of tree and array multipliers. Additionally, bit-serial and on-line arithmetic were advanced to deal with severe pin limitations in VLSI packages. This phase of the development of computer arithmetic was also guided by the demand to perform arithmetic-intensive signal processing functions using low-cost and/or high-performance embedded hardware. Examples of fixed- and floating-point processors for digital signal processing applications are provided in Section 28.4.

During the 1990s, computer arithmetic continued to mature. Despite the lack of any breakthrough design concept, both theoretical development and refinement of the designs continued at a rapid pace. The increasing demand for performance resulted in fine-tuning of arithmetic algorithms to take advantage of particular features of implementation technologies. Thus, we witnessed the emergence of a wide array of hybrid designs that combined features from one or more pure designs into a highly optimized arithmetic structure. Other trends included increasing use of table lookup and tight integration of arithmetic unit and other parts of the processor for maximum performance. As clock speeds reached and surpassed 100, 200, 300, 400, and 500 MHz in rapid succession, everything had to be (deeply) pipelined to ensure the smooth flow of data through the system. A modern example of such methods in the design of Intel's Pentium Pro (P6) microprocessor is discussed in Section 28.5.

28.2 AN EARLY HIGH-PERFORMANCE MACHINE

In this section, we review key design features of the floating-point arithmetic hardware of IBM System/360 Model 91, a supercomputer of the mid-1960s, which brought forth numerous architectural innovations. The technical paper on which this description is based [Ande67] is considered one of the key publications in the history of computer arithmetic. For an insightful retrospective on the Model 91, see [Flyn98].

The IBM System/360 Model 91 had two concurrently operating floating-point execution units (Fig. 28.1), each with a two-stage pipelined adder and a 12×56 pipelined multiplier, to meet the ambitious design goal of executing one floating-point instruction per 20-ns clock cycle on the average. The unit could handle 32-bit or 64-bit floating-point numbers with sign, 7-bit excess-64 base-16 exponent, and 24-bit or 56-bit normalized significand in [1/16, 1). Floating-point operands were supplied to the execution units from a number of buffers or registers. Within the execution units, a number of "reservation stations" (RS), each holding two operands, allowed effective utilization of hardware by ensuring that the next set of operands always was available when an arithmetic circuit was ready to accept it.

The Model 91 floating-point adder consisted of standard blocks such as exponent adder, preshifter, postshifter, and exponent adjuster, in addition to a 56-bit fraction adder. The fraction adder had a three-level carry-lookahead design with 4-bit groups and 8-bit sections. Thus, there were two groups per section and seven sections in the adder. Many clever design methods were used to speed up and simplify the adder. For example, the adder was designed to produce both the true sum and its 2's complement, one of which was then selected as the adder's output. This feature served to reduce the length of the adder's critical path; only the operand that was not preshifted could be complemented. This could force the computation of $y - x$ instead of the desired $x - y$, thus necessitating output complementation. As a result of various optimization and speedup techniques, a floating-add arithmetic operation could be executed in 2 clock cycles (or one add per cycle per floating-point unit with pipelining).

The Model 91 floating-point multiplier could multiply a 56-bit multiplicand by a 12-bit multiplier in one pass through its hardware tree of CSA adders, keeping the partial product in carry-save form, to be subsequently combined with the results from other 12-bit segments of the multiplier. Radix-4 Booth's recoding was used to form six multiples of the multiplicand to be added (thus, actually 13 bits of the multiplier were required in each step in view of the 1-bit overlap). The six multiples were reduced to two

in a three-level CSA tree. Another two CSA levels were used to combine these two values with the shifted carry-save partial product from earlier steps. Pipelining allowed 12 multiplier bits to be processed in each clock cycle. The floating-point multiply took 6 clock cycles, or 120 ns, overall.

Floating-point division was performed by the Newton–Raphson convergence method using the hardware multiplier and a small amount of extra logic. An initial table lookup provided an approximate reciprocal of the divisor that led to 7 bits of convergence with a 12-bit multiplier. Three more steps of such short multiplications (requiring a single pass through the CSA tree) increased the convergence to 14, 23, and 28 bits. A final half-multiply, needing three passes through the CSA tree, completed the process. The pair of multiplications was pipelined in each step, with the result that floating-point divide took only 18 clock cycles. Early versions of the Model 91 floating-point unit sometimes yielded an incorrect least significant bit for the quotient. This problem, which had been due to inadequate analysis of the division convergence process, was corrected in subsequent versions.

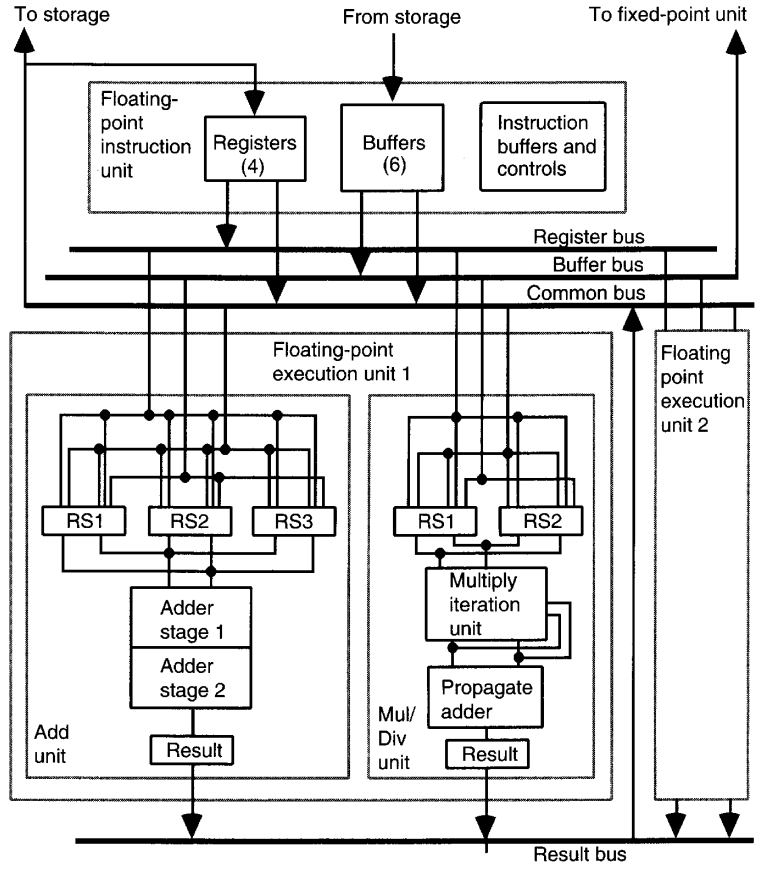


Fig. 28.1 Overall structure of the IBM System/360 Model 91 floating-point execution unit.

28.3 A MODERN VECTOR SUPERCOMPUTER

Modern supercomputers come in two varieties: vector multiprocessors consisting of a small number of powerful vector processors, and parallel computers using a moderate to very large ensemble of simpler processors.

Moderately parallel computers typically use off-the-shelf, high-performance microprocessors as their basic building blocks, while some massively parallel computers are based on very simple custom processors, perhaps with multiple processors on a single microchip. Since we discuss arithmetic in a modern microprocessor in Section 28.5, and since we have already covered an example of arithmetic in the simple bit-serial processors of the CM-2 massively parallel computer (Section 24.3), here we focus on the design of the Cray X-MP/Model 24 processor as an example of the former category [Robb89]. This machine has been superseded by the Y-MP, C-90, and various other Cray supercomputers, but it offers a good example for discussing the principles of high-performance vector processing, with the associated highly pipelined implementation of arithmetic operations and pipeline chaining.

The Cray X-MP/Model 24 consists of two identical CPUs sharing a main memory and an I/O subsystem. Most instructions can begin execution in a single 9.5-ns machine cycle and are capable of producing results on every machine cycle, given suitably long vector computations and appropriate data layout in memory to avoid memory bank conflicts. Each CPU has an address section, a scalar section, and a vector section, each with its own registers and functional units.

The address section is the simplest of the three sections. It uses an integer multiplier and an adder (four- and two-stage pipeline, respectively) for operating on, and computing, 24-bit memory addresses.

The scalar section has functional units for addition (three-stage pipeline), weight/parity/leading-0s determination (three- or four-stage), shifting (two-stage), and logical operations (one-stage). With very few exceptions, all arithmetic and logical operations deal with 64-bit integer or floating-point operands. Floating-point numbers have a sign bit, 15 exponent bits, and 48 significand bits (including an explicit 1 after the radix point).

The vector section is perhaps the most interesting and elaborate part of the processor, and we focus on it in the remainder of this section. Figure 28.2 is a block diagram of the Cray X-MP's vector section. There are eight sets of 64-element vector registers that are used to supply operands to, and accept results from, the functional units. These allow the required vectors or vector segments to be prefetched, and the vector results stored back in memory, concurrently with arithmetic/logic operations on other vectors or vector segments. In fact, intermediate computation results do not need to be stored in a register before further processing. A method known as *pipeline chaining* allows the output of one pipeline (e.g., multiplier) to be forwarded to another (say, adder) if a vector computation such as $(A[i] \times B[i]) + C[i]$ is to be performed.

Vector computations need 3 clock cycles for their *setup*, which includes preparing the appropriate functional units and establishing paths from/to source and destination registers to them. At the end of a vector computation, 3 more clock cycles are needed for *shutdown* before the results in the destination vector register can be used in other operations. This type of pipelining overhead, which becomes insignificant when one is dealing with long vectors, is the main reason for vector machines having a "break-even" vector length (i.e., a length beyond which vector arithmetic is faster than scalar arithmetic performed in a program loop).

Once a vector computation has been set up, a pair of elements enters the first stage of the pipeline on every clock cycle and the partial results for the preceding pairs move one stage forward in the pipeline. Figure 28.2 lists the number σ of pipeline stages for various operations.

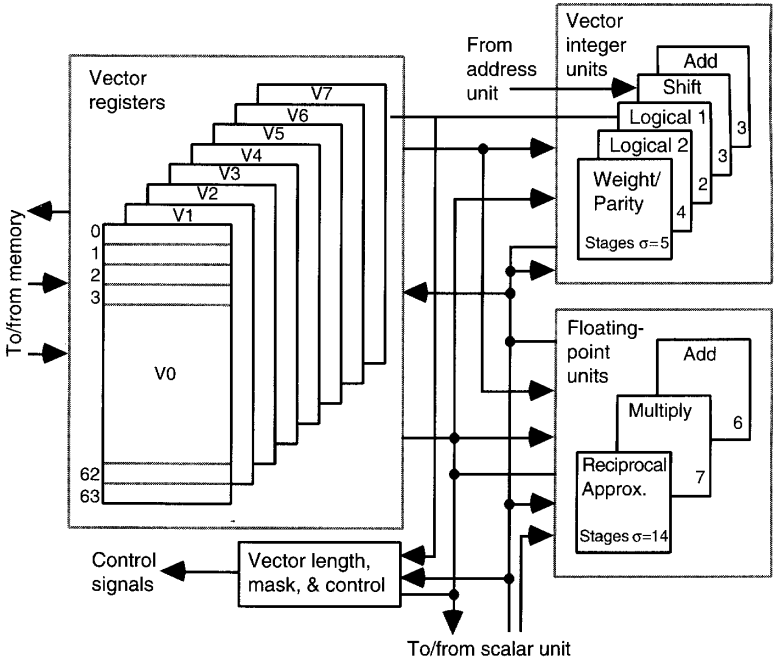


Fig. 28.2 The vector section of one of the processors in the Cray X-MP/Model 24 supercomputer.

The output of a σ-stage pipelined unit becomes available for chaining after $\sigma + 5$ clock cycles. Such a unit needs $\lambda + \sigma + 5$ clock cycles to operate on a λ-element vector. However, the functional unit is freed for the next vector operation after $\lambda + 4$ cycles.

28.4 DIGITAL SIGNAL PROCESSORS

Many digital signal processing (DSP) applications are arithmetic-intensive and cost-sensitive, thus requiring innovative solutions for cost-effective implementation. A digital signal processor (also abbreviated as DSP), can be a special-purpose or a general-purpose unit. Special-purpose DSPs have been designed in a variety of ways, using conventional or unconventional (RNS, logarithmic) number representations. It is impossible to review all these approaches here [Sode86], [Jull94]. We thus focus on the design of typical general-purpose DSP chips.

General-purpose DSPs are available as standard components from several microchip manufacturers. They come in two varieties: fixed point and floating point. Integer DSP chips are simpler and thus both faster and less expensive. They are used whenever the application deals with numerical values in limited and well-defined ranges so that scaling can be done with acceptable overhead (e.g., in simple voice processing). The payoff then is faster processing or higher accuracy. When the range of numerical values is highly variable or unpredictable, or the data rate is too high to allow the use of lengthy scaling computations, built-in floating-point arithmetic capability becomes mandatory (e.g., in multimedia workstations).

Motorola's DSP56002 chip is a 24-bit fixed-point DSP [ElSh96]. It deals with 24-bit and 48-bit signed fractions and internally uses a 56-bit format consisting of 9 whole bits, including the sign, and 47 fractional bits. As shown in Fig. 28.3, there are four 24-bit input registers that can also be used as two 48-bit registers. Similarly, the two 56-bit accumulator registers can be viewed as four 24-bit and two 8-bit registers. Arithmetic/logic operations are performed on up to three operands, with the 56-bit result always stored in an accumulator. Example instructions include the following:

ADD	A, B	$\{A + B \rightarrow B\}$
SUB	X, A	$\{A - X \rightarrow A\}$
MPY	$\pm$X1, X0, B	$\{\pm X1 \times X0 \rightarrow B\}$
MAC	$\pm$ Y1, X1, A	$\{A \pm (Y1 \times X1) \rightarrow A\}$
AND	X1, A	$\{A \text{ AND } X1 \rightarrow A\}$

The ALU can round the least significant half (A0 or B0) into the most significant half (A1 or B1) of each accumulator. So, for example, an MPY or MAC instruction can be executed with or without rounding, leading to a 24- or 48-bit result in an accumulator.

The 56-bit shifter can shift left or right by 1 bit or pass the data through unshifted. The two data shifters, associated with the A and B accumulators, take 56-bit inputs and produce 24-bit

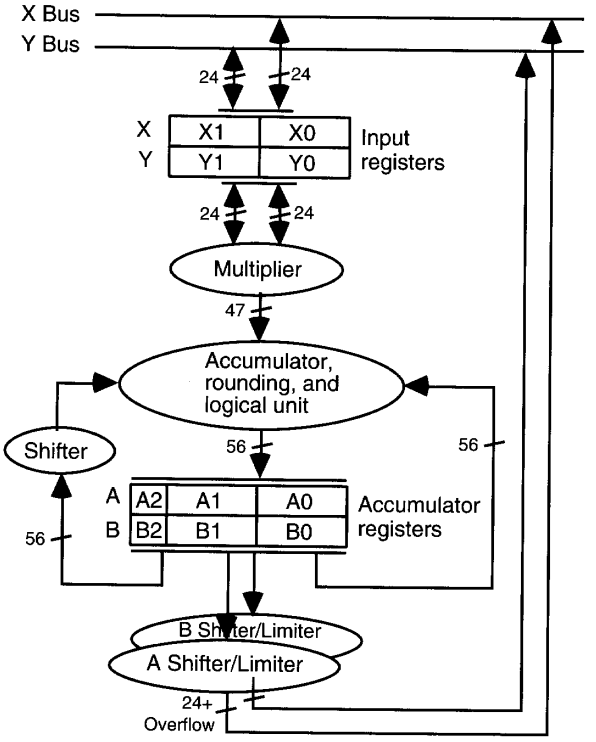


Fig. 28.3 Block diagram of the data ALU in Motorola's DSP56002 (fixed-point) processor.

outputs, each with an "overflow" bit. One-bit left or right shift is possible for scaling purposes. The data limiter causes the largest value of the same sign to be output when the (shifted) 56-bit data is not representable in 24 bits.

There are also a variety of data movement, bit manipulation, and flow control instructions, as in any other processor. Details of the instruction set and programming considerations for Motorola's DSP56002 processor, along with example applications in filter implementation and fast Fourier transform, have been published [ElSh96].

As an example of a floating-point DSP chip, we briefly review Motorola's DSP96002, which has many features of a 32-bit general-purpose processor along with enhancement for DSP applications [Sohi88]. Multiple DSP96002 chips can share a bus and communicate directly with each other in a parallel configuration with very high performance.

DSP96002 implements the IEEE single-precision (32-bit) and single-extended-precision ($1 + 11 + 32 = 44$ bits, no hidden bit) floating-point arithmetic. An internal 96-bit format (sign, 20 bits of special tags, 11-bit exponent, 64-bit significand) is used to minimize error accumulation.

The data ALU (Fig. 28.4), so named to distinguish it from address computation units, supports IEEE floating-point arithmetic in a single instruction cycle or 2 clock cycles. The full instruction actually takes 3 instruction (or 6 clock) cycles to finish but is executed in a three-stage (fetch, decode, execute) pipeline that can accept a new instruction in every cycle.

The floating-point add/subtract unit calculates both the sum and the difference of its two inputs, with one or both results stored in the register file in the same cycle. The add/subtract unit is also used for integer arithmetic, a variety of data type conversions, and multibit shift operations (taking advantage of its barrel shifter). The floating-point multiply unit contains a 32×32 hardware multiplier, thus supporting both 32-bit signed/unsigned integer multiplication and single-extended-precision floating-point multiplication (with 32-bit significands) in one cycle. A full 64-bit product is produced.

Finally, the special function unit implements division, square-rooting, and logical operations. Division and square-rooting require multiple instructions, beginning with a special instruction to generate a reciprocal (root) seed and continuing with a convergence computation.

DSP96002 accepts, and properly handles, denormalized numbers, but requires one additional machine cycle to process each denormalized source operand or denormalized result. A

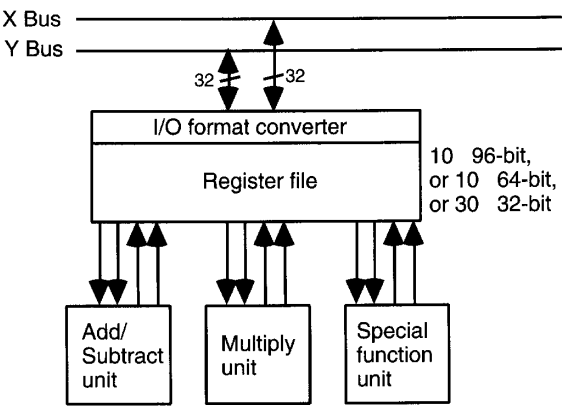


Fig. 28.4 Block diagram of the data ALU in Motorola's DSP96002 (floating-point) processor.

"flush-to-zero" underflow mode can be optionally selected to force denormalized numbers to 0, thus avoiding the possible extra cycles and making the execution timing completely data-independent.

28.5 A WIDELY USED MICROPROCESSOR

Older microprocessors contained an ALU for integer arithmetic within the basic CPU chip and an optional floating-point coprocessor on a separate chip. Recently, increasing VLSI circuit density has led to the trend of integrating both units on a single microchip, while still leaving enough space for large on-chip cache memories for data and instructions.

As an example, we describe a member of the Intel's Pentium family of microprocessors: the Intel Pentium Pro, also known as Intel P6. The primary design goal for the Intel P6 was to achieve the highest possible performance, while keeping the external appearances compatible with the Pentium and using the same mass-production technology [Shan98]. Intel's Pentium II is essentially a Pentium Pro, complemented with a set of multimedia instructions.

The Intel P6 has a 32-bit architecture, internally using a 64-bit data bus, 36-bit addresses, and an 86-bit floating-point format. In the terminology of modern microprocessors, P6 is superscalar and superpipelined: superscalar because it can execute multiple independent instructions concurrently in its many functional units, as opposed to the Cray machine of Section 28.3, which has concurrent execution only for vector operations; superpipelined because its instruction execution pipeline with 14^+ stages is very deep. The design of the Intel P6, which was initially based on a 150- to 200-MHz clock, has 21M transistors, roughly a quarter of which are for the CPU and the rest for the on-chip cache memory. The Intel P6 is also capable of glueless multiprocessing with up to four processors.

Figure 28.5 shows parts of the CPU that are relevant to our discussion. Since high performance in the Intel P6 is gained by out-of-order and speculative instruction execution, a key component in the design is a reservation station that is essentially a hardware-level scheduler of micro-operations. Each instruction is converted to one or more micro-operations, which are then executed in arbitrary order whenever their required operands are available.

The result of a micro-operation is sent to both the reservation station and a special unit called the reorder buffer. This latter unit is responsible for making sure that program execution remains consistent by committing the results of micro-operations to the machine's "retirement" registers only after all pieces of an instruction have terminated and the instruction's "turn" to execute has arrived within the sequential program flow. Thus, if an interrupt occurs, all operations that are in progress can be discarded without causing inconsistency in the machine's state. There is a full crossbar between all five ports of the reservation station so that any returning result can be forwarded directly to any other unit for the next clock cycle.

Fetching, decoding, and setting up the components of an instruction in the reservation station takes 8 clock cycles and is performed as an eight-stage pipelined operation. The retirement process, mentioned above, takes 3 clock cycles and is also pipelined. Sandwiched between the preceding two pipelines is a variable-length pipeline for instruction execution. For this middle part of instruction execution, the reservation station needs 2 cycles to ascertain that the operands are available and to schedule the micro-operation on an appropriate unit. The operation itself takes one cycle for register-to-register integer add and longer for more complex functions. Because of the multiplicity of functional units with different latencies, out-of-order and speculative execution (e.g., branch prediction) are crucial to high performance.

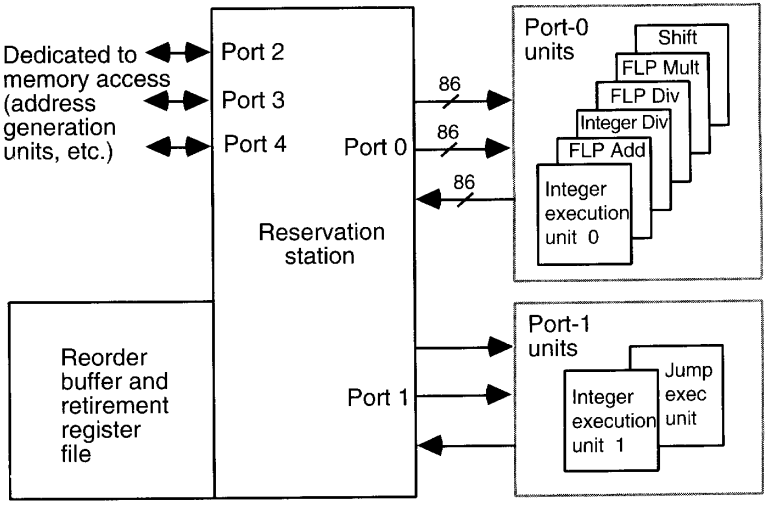


Fig. 28.5 Key parts of the CPU in the Intel Pentium Pro (P6) microprocessor.

In a sense, the deep pipelining of instruction execution in the Intel P6 makes the performance less sensitive to the arithmetic algorithms and circuits. Indeed, the bulk of hardware in the P6 is devoted to the management of pipelining and out-of-order instruction execution rather than to arithmetic circuits.

28.6 TRENDS AND FUTURE OUTLOOK

Arithmetic designs are evolving as a result of changes in the underlying technology. The move from small-scale integration through medium- and large-scale integration to VLSI has gradually shifted the emphasis from reducing the number of gates and gate levels in arithmetic circuits to considering the overall design in terms of both computational elements and interconnections. Increasing densities have also led to concerns about adequate input/output bandwidth, clock and power distribution, heat dissipation, and testability. Design challenges will no doubt continue to emerge as we deal with even newer technologies and application requirements (fully distributed micropipelines, subnanosecond arithmetic, low-power design, the quest for petaFLOPS, etc.).

Today, designs for arithmetic circuits are developed not by analyzing an elegant algorithm and optimizing its various parameters, but rather by getting down to the level of transistors and wires. This explains the proliferation of hybrid designs that use two or more distinct paradigms (e.g., fast adders using Manchester carry chains along with carry-lookahead and carry-select structures) to obtain the best designs for given cost–performance requirements.

Concurrent with developments in the VLSI technology, changing application characteristics have dictated a shift of focus in computer arithmetic from high-speed or high-throughput designs in mainframe computers to low-cost and low-power designs for embedded and mobile applications. These have in turn led to renewed interest in bit- and digit-serial arithmetic as mechanisms to reduce the VLSI area and to improve packageability and testability. High-performance designs requiring lookahead and speculative execution are expensive and often

at odds with the goal of reducing power consumption to extend the battery life and/or simplify heat dissipation. Many challenging problems are being addressed in these areas.

The desirability of synchronous versus asynchronous design is also being reexamined. Thus far, synchronous circuits have prevailed in view of their ease of design, tractability of analysis, and predictability of performance. A secondary, but still important, drawback of asynchronous design is the overhead in time and area for the required handshaking circuits that regulate the flow of data between circuit segments. However, the higher speeds and packaging densities of modern digital circuits are stretching the limits of our ability to distribute the clock signal to all the required points. Also, signal propagation delays over long wires are forcing the designers to modularize the design (e.g., via systolic arrays), thus in some cases introducing an overhead that is comparable to that of handshaking for asynchronous operation. Novel design paradigms and improved tools for the synthesis and analysis of asynchronous systems are slowly changing the balance in favor of the latter [Hauc95]. For example, low-level pipelining methods (micropipelines), perhaps extending all the way down to the logic gate level, are thought to hold promise for the arithmetic circuits of the future.

Fundamentally new technologies and design paradigms may alter the way in which we view or design arithmetic circuits. Just as the availability of cheap, high-density memories brought table-lookup methods to the forefront, certain computational elements being developed in connection with artificial neural networks may revolutionize our approach to arithmetic algorithms. As an example, imagine the deep changes that would ensue if an artificial neuron capable of summing several weighted inputs and comparing the result to a fixed threshold could be built from a few transistors. Such a cell would be considerably more powerful than a switch or standard logic gate, thus leading to new designs for arithmetic functions [Vass96]. As a second example, researchers in the field of optical computing, eager to take full advantage of parallel operations made possible by the absence of pin limitations, have paid significant attention to redundant number representations. Yet another example is found in the field of multivalued logic, which has an inherent bias toward high-radix arithmetic.

On the theoretical front, studies in arithmetic complexity [Pipp87] have been instrumental in broadening our understanding of algorithmic speedup methods. Any n-variable Boolean function that is actually dependent on all n variables (say, the most significant output bit of an $n/2 \times n/2$ unsigned multiplier) requires a gate count or circuit complexity of at least $\Omega(n)$ and a delay or circuit depth of $\Omega(\log n)$. On the other hand, any Boolean function can be realized by a size-$(2^n - 1)$, depth-n, complete binary tree of 2-to-1 multiplexers by using the Shannon expansion

$$f(x_1, x_2, \cdots, x_n) = x_1 \, f(1, x_2, \cdots, x_n) + \overline{x}_1 \, f(0, x_2, \cdots, x_n)$$

for each variable in turn. Key questions in arithmetic complexity thus deal with the determination of where in the wide spectrum of $\Omega(n)$ to $O(2^n)$ circuit complexity, and $\Omega(\log n)$ to $O(n)$ circuit depth, practical implementations of the various arithmetic functions may lie, and what can be achieved in terms of cost (delay) if we restrict the design, say, to having logarithmic delay (linear, or polynomial, cost).

For example, we know in the case of addition/subtraction that the bounds $O(n)$ on cost and $O(\log n)$ on delay are achievable simultaneously by means of certain carry-lookahead adder designs, say. For multiplication, we can achieve $O(\log n)$ delay with $O(n \log n \log \log n)$ cost in theory, though practical designs for small word widths have logarithmic delay with $O(n^2)$ cost. Logarithmic-depth circuits for division are now known, but they are much more complex than logarithmic-depth multipliers. Note that a logarithmic-depth multiplier is capable of performing division in $O(\log^2 n)$ time when a convergence method is used.

Many innovations have appeared in computer arithmetic since the early days of electronic computers [Burk46]. The emergence of new technologies and the unwavering quest for higher

performance are bound to create new challenges in the coming years. These will include completely new challenges, as well as novel or transformed versions of the ones discussed in the preceding paragraphs. Computer arithmetic designers, who helped make digital computers into indispensable tools in the five decades since the introduction of the stored-program concept, will thus have a significant role to play in making them even more useful and ubiquitous as the second half-century of digital computing unfolds.

PROBLEMS

28.1 Historical perspective Using the discussion in Section 28.1 as a basis, and consulting additional references as needed, draw a time line that shows significant events in the development of digital computer arithmetic. On your time line, identify what you consider to be the three most significant ideas or events related to the topics discussed in each of the Parts I to IV of this book. Briefly justify your choices. Include floating-point numbers and arithmetic in your discussion (i.e., floating-point representation in Part I, floating-point addition in Part II, etc.).

28.2 Arithmetic before electronic digital computers

a. Study the implementation of arithmetic operations on mechanical calculators and other machines that preceded electronic computers. Prepare a report (including a time line) discussing the developments of key ideas and various implementations.

b. Repeat part a for electronic analog computers. Compare the ideas and methods to those of digital arithmetic and discuss.

28.3 IBM System/360 Model 91

a. Based on the description in Section 28.2 and what you learned about convergence division in Chapter 16, determine the size of the lookup table providing the initial approximation to the divisor reciprocal in the IBM System/360 Model 91.

b. Estimate, using back-of-the-envelope calculations, the MFLOPS computational power of the IBM System/360 Model 91. Assume complete overlap between instruction preparation and execution. Use an instruction mix of 60% add, 30% multiply, and 10% divide.

c. Study the integer arithmetic capabilities of the IBM System/360 Model 91.

28.4 The CDC 6600 computer Prepare a description of the arithmetic capabilities of CDC 6600 in a manner similar to the discussion of the IBM System/360 Model 91 in Section 28.2. Stress similarities and key differences between the two systems.

28.5 Cray X-MP/Model 24 A polynomial $f(x)$ of degree $n - 1$ (n coefficients, stored in a vector register) is to be evaluated using Horner's rule for n different values of x (available in a second vector register). The n results are to be left in a third vector register. Estimate the number of cycles needed for this computation on the CRAY X-MP/Model 24 with pipeline chaining. What is the machine's MFLOPS rating for this computation?

28.6 Floating-point representation formats The IBM System 360 Model 91 did not use the IEEE standard floating-point format because its design preceded the standard. Until recently, Cray machines did not use the standard either, mainly for performance and program compatibility reasons. Compare these two nonstandard floating-point formats

to the IEEE standard format and discuss difficulties that might arise in porting programs among the three floating-point implementations.

28.7 Digital filtering on a fixed-point DSP

a. A median filter operates on a black-and-white digital image and replaces each pixel value (representing the gray level) with the median of nine values in the pixel itself and in the eight horizontally, vertically, and diagonally adjacent pixels. Estimate the number of cycles for median filtering of a 1024×1024 image using the Motorola DSP56002 fixed-point signal processor, assuming that control is completely overlapped with computation.

b. Repeat part a for a mean filter.

28.8 Polynomial evaluation on a floating-point DSP A degree-$(n-1)$ polynomial $f(x)$ is to be evaluated using Horner's rule for n values of x. Using reasonable assumptions as needed, estimate the execution time of this problem on the Motorola DSP96002 floating-point signal processor. Discuss the cost-effectiveness of this solution compared to a vector supercomputer applied to the same problem.

28.9 A high-performance DSP Recent DSP products announced by Texas Instruments and other suppliers have much greater computational capabilities than those studied in Section 28.4. Pick one such system and describe its arithmetic capabilities and performance relative to the corresponding DSP chip (fixed- or floating-point) described in Section 28.4.

28.10 Higher than peak performance The peak MFLOPS performance of a processor is usually determined based on the speed of floating-point addition. For example, if one floating-point addition can be initiated in every 5-ns clock cycle, the peak performance is considered to be 200 MFLOPS.

a. Show that the Motorola DSP96002 floating-point signal processor can exceed its peak performance for certain problems.

b. Show that a similar effect is possible when arithmetic is performed bit-serially.

28.11 CISC versus RISC microprocessors The Intel Pentium Pro (P6) microprocessor is an example of the class of complex instruction set computers (CISCs). Most modern microprocessors belong to the complementary class of reduced instruction set computers (RISCs). Choose one example of this latter class and contrast it to the Intel P6 with regard to the implementation of arithmetic functions The MIPS R10000 is a particularly good example and has been described in some detail in [Yeag96].

28.12 The Alpha microprocessor The Alpha microprocessor of Digital Equipment Corporation (now part of Compaq) is among the fastest processors available today. Study arithmetic in Alpha and compare it to the Intel P6.

28.13 Role of arithmetic in microprocessor performance Pick a microprocessor with which you are most familiar and/or have ready access to the relevant technical information. Estimate the percentage of instruction cycle time taken up by arithmetic operations. Include in this figure arithmetic operations performed for address calculations and other

bookkeeping tasks. When arithmetic is fully overlapped with nonarithmetic functions, divide the time equally between the two.

28.14 Multiprecision arithmetic on microprocessors We would like to design a set of routines for operating on multiprecision unsigned integers that are represented by variable-length vectors. The 0th element of the vector is the length of the number in k-bit words (e.g., 3 means that the number is $3k$ bits long and is represented in three k-bit chunks following the 0th vector element, MSB first).

a. Express the length of the numbers resulting from addition, multiplication, and division of two numbers, having the length field values of m and n, respectively.

b. Design an algorithm for performing multiprecision add from the most significant end. One way to do this is to store temporary sum digits and then go back and correct them if a carry is produced that affects them. Write the algorithm in such a way that only final sum digit values are written. *Hint:* The value of a digit can be finalized when the next position sum is not $2^k - 1$. So, you need only keep a count of how many such positions appear in a row.

c. Compare the performance of two microprocessors of your choosing in running the multiprecision addition algorithm of part b.

d. It is sometimes necessary to multiply or divide a multiprecision number by a regular (single-precision) number. Provide complete algorithms for this purpose.

e. Repeat part c for the computations defined in part d.

28.15 Synchronous versus asynchronous design Study synchronous and asynchronous adder designs with regard to speed, hardware implementation cost, and power requirement [Kinn96].

28.16 Neuronlike hardware elements Consider the availability of a very simple neuronlike element with three binary inputs and one binary output. During the manufacturing of the element, each input can be given an arbitrary integer weight in [1, 3] and the element can be given an arbitrary threshold in [1, 9]. The output will be 1 if the weighted sum of the inputs equals or exceeds the threshold. Synthesize a single-bit full adder using these elements.

REFERENCES

[Ande67] Anderson, S.F., J.G. Earle, R.E. Goldschmidt, and D.M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Research and Development*, Vol. 11, No. 1, pp. 34–53, 1967.

[Burk46] Burkes, A.W., H.H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Study Report, Princeton, NJ, 1946.

[ElSh96] El-Sharkawy, M., *Digital Signal Processing Applications with Motorola's DSP56002 Processor*, Prentice-Hall, 1996.

[Flyn98] Flynn, M. J., "Computer Engineering 30 Years After the IBM Model 91," *IEEE Computer*, Vol. 31, No. 4, pp. 27–31, 1998.

[Hauc95] Hauck, S., "Asynchronous Design Methodologies," *Proc. IEEE*, Vol. 83, No. 1, pp. 67–93, 1995.

[Jull94] Jullien, G.A., "High Performance Arithmetic for DSP Systems," in *VLSI Signal Processing Technology*, ed. by M.A. Bayoumi and E.E. Swartzlander, Jr. (eds.), Kluwer, 1994, pp. 59–96.

[Kinn96] Kinniment, D.J., "An Evaluation of Asynchronous Addition," *IEEE Trans. Very Large Scale Integration Systems*, Vol. 4, No. 1, pp. 137–140, March 1996.

[Lind96] Linder, D.H., and J.C. Harden, "Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry," *IEEE Trans. Computers*, Vol. 45, No. 9, pp. 1031–1044, 1996.

[MacS61] MacSorley, O.L., "High-Speed Arithmetic in Binary Computers," *IRE Proc.*, Vol. 49, pp. 67–91, 1961. Reprinted in [Swar90], Vol. 1, pp. 14–38.

[Omon94] Omondi, A.R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Pipp87] Pippenger, N., "The Complexity of Computations by Networks," *IBM J. Research and Development*, Vol. 31, No. 2, pp. 235–243, March 1987.

[Robb89] Robbins, K.A., and S. Robbins, *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing*, Springer-Verlag, 1989.

[Shan98] Shanley, T., *Pentium Pro and Pentium II System Architecture*, 2nd ed., MindShare, 1998.

[Shaw50] Shaw, R.F., "Arithmetic Operations in a Binary Computer," *Rev. Scientific Instruments*, Vol. 21, pp. 687–693, 1950. Reprinted in [Swar90], Vol. 1, pp. 7–13.

[Sode86] Soderstrand, M.A., W.K. Jenkins, G.A. Jullien, and F.J. Taylor (eds.), *Residue Number System Arithmetic*, IEEE Press, 1986.

[Sohi88] Sohie, G.R.L., and K.L. Kloker, "A Digital Signal Processor with IEEE Floating-Point Arithmetic," *IEEE Micro*, Vol. 8, No. 6, pp. 49–67, December 1988.

[Swar90] Swartzlander, E.E., Jr., *Computer Arithmetic*, Vols. 1 and 2, IEEE Computer Society Press, 1990.

[Thor70] Thornton, J.E., *Design of a Computer: The Control Data 6600*, Scott, Foresman, & Co., 1970.

[Vass96] Vassiliadis, S., S. Cotofana, and K. Bertels, "2-1 Addition and Related Arithmetic Operations with Threshold Logic," *IEEE Trans. Computers*, Vol. 45, No. 9, pp. 1062–1067, 1996.

[Yeag96] Yeager, K.C., "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Vol. 16, No. 2, pp. 28–40, April 1996.

INDEX

Ideal for graduate and senior undergraduate level courses in computer arithmetic and advanced digital design, *Computer Arithmetic: Algorithms and Hardware Designs* provides a balanced, comprehensive treatment of computer arithmetic, covering topics in arithmetic unit design and circuit implementation that complement the architectural and algorithmic speedup techniques used in high-performance computer architecture and parallel processing. Using a unified and consistent framework, the text begins with number representation and proceeds through basic arithmetic operations, floating-point arithmetic, and function evaluation methods. Later chapters cover broad design and implementation topics—including techniques for high-throughput, low-power, and fault-tolerant arithmetic—and also feature brief case studies.

An indispensable resource for instruction, professional development, and research in digital computer arithmetic, *Computer Arithmetic: Algorithms and Hardware Designs* combines broad coverage of the underlying theories of computer arithmetic with numerous examples of practical designs, worked-out examples, and a large collection of meaningful problems.

Features

- Divided into 28 lecture-size chapters
- Emphasizes both the underlying theories of computer arithmetic and actual hardware designs
- Carefully links computer arithmetic to other subfields of computer engineering
- Includes over 450 end-of-chapter problems ranging in complexity from simple exercises to mini-projects
- Incorporates many examples of practical designs
- Uses consistent standardized notation throughout
- Instructor's manual includes solutions to text problems, additional exercises, test questions, and enlarged versions of figures and charts

About the Author

Behrooz Parhami is Professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. His research deals with parallel architectures and algorithms, computer arithmetic, and reliable computing. His technical publications include over 170 papers in journals and international conferences, the textbook *Introduction to Parallel Processing: Algorithms and Architectures* (1999), and an English/Farsi glossary of computing terms. Dr. Parhami is a Fellow of both the IEEE and the British Computer Society, a member of the Association for Computing Machinery, and a Distinguished Member of the Informatics Society of Iran.