# Chapter 5

# Division

Division algorithms can be grouped into two classes, according to their iterative operator. The first class, where subtraction is the iterative operator, contains many familiar algorithms (such as nonrestoring division) which are relatively slow, as their execution time is proportional to the operand (divisor) length. We then examine a higher speed class of algorithm, where multiplication is the iterative operator. Here, the algorithm converges quadratically; its execution time is proportional to $\log_2$ of the divisor length.

## 5.1 Subtractive Algorithms: General Discussion

### 5.1.1 Restoring and Nonrestoring Binary Division

Most existing descriptions of nonrestoring division are from one of two distinct viewpoints. The first is mathematical in nature, and describes the quotient digit selection as being $-1$ or $+1$, but it does not show the translation from the set $\{-1, +1\}$ to the standard binary representation $\{0, 1\}$. The second is found mostly in application notes of semiconductor manufacturers, where the algorithm is given without any explanation of what makes it work. The following section ties together these two viewpoints. We start by reviewing the familiar pencil and paper division, then show the similarities and differences between this and restoring and nonrestoring division. After this, we examine nonrestoring division from the mathematical concepts of the signed digit representation to the problem of conversion to the standard binary representation. This is followed by an example of hardware implementation. Special attention is given to two exceptional conditions: the case of a zero partial remainder, and the case of overflow.

### 5.1.2 Pencil and Paper Division

Let us perform the division 4537/3, using the method we learned in elementary school:

$$
\begin{array}{rr}
 & 1\ \ 5\ \ 1\ \ 2 \\
3)\overline{4\ \ 5\ \ 3\ \ 7} \\
 & 3 \\
 & \overline{1\ \ 5} \\
 & 1\ \ 5 \\
 & \overline{\phantom{0}\ \ \phantom{0}\ \ 3} \\
 & \phantom{0}\ \ \phantom{0}\ \ 3 \\
 & \overline{\phantom{0}\ \ \phantom{0}\ \ 7} \\
 & \phantom{0}\ \ \phantom{0}\ \ 6 \\
 & \overline{\phantom{0}\ \ \phantom{0}\ \ 1}
\end{array}
$$

The forgoing is an acceptable shorthand; for example, in the first step a 3 is shown subtracted from 4, but mathematically the number 3000 is actually subtracted from 4537, yielding a partial remainder of 1537. The above division is now repeated, showing the actual steps more explicitly:

$$
\begin{array}{rl}
 & 1512 \leftarrow \text{Quotient} \\
3)\overline{4537} & \leftarrow \text{Dividend} \\
 & 3000 \leftarrow \text{Divisor}{*}q(\text{MSD}) * 10^3 \\
 & \overline{1537} \leftarrow \text{Partial remainder} \\
 & 1500 \\
 & \overline{0037} \\
 & 0030 \\
 & \overline{0007} \\
 & 0006 \\
 & \overline{0001} \leftarrow \text{Remainder}
\end{array}
$$

Let us represent the remainder as $R$, the divisor as $D$, and the quotient as $Q$. We will indicate the $i$th digit of the quotient as $q_i$, and the value of the partial remainder after subtraction of the $j$th radix power, trial product $(q_j * D * B^j)$ as $R(j)$ i.e., $R(0)$ is the final remainder. Then the process of obtaining the quotient and the final remainder can be shown as follows:

$$
\begin{array}{lll}
4537 - 1 * 3 * 10^3 = 1537 & \text{or} & R(4) - q_3 * D * 10^3 = R(3) \\
1537 - 5 * 3 * 10^2 = 0037 & \text{or} & R(3) - q_2 * D * 10^2 = R(2) \\
0037 - 1 * 3 * 10^1 = 0007 & \text{or} & R(2) - q_1 * D * 10^1 = R(1)\text{'} \\
0007 - 2 * 3 * 10^0 = 0001 & \text{or} & R(1) - q_0 * D * 10^0 = R(0)
\end{array}
$$

or, in general, at any step:

$$
\boxed{R(i) = R(i+1) - q_i * D * 10^i},
$$

$$
\text{where} \quad i = n-1, n-2, \ldots, 1, 0.
$$

How did we determine at every step the value $q_i$? We did it by a mental trial and error; for example, for $q_3$, we may have guessed 2, which would have given $q_3 \cdot D \cdot 10^3 = 2 \cdot 3 \cdot 1000 = 6000$, but that is larger than the dividend; so we mentally realized that $q_3 = 1$, and so on. Now a machine would have to go explicitly through the above steps; that is, it would have to subtract until the partial remainder became negative, which means it was subtracted one time too many, and it would have to be restored to a positive partial remainder. This brings us to restoring division—algorithms, which restore the partial remainder to a positive condition before beginning the next quotient digit iteration.

**Restoring Division**

The following equations illustrate the restoring process for the previous decimal example:

$$
\begin{array}{rll}
4537 - 3 * 10^3 = +1537 & & q_3 = 1 \\
1537 - 3 * 10^3 = -1463 & & q_3 = 2 \\
-1463 + 3 * 10^3 = +1537 & \text{restore} & \boxed{q_3 = 1} \\
+1537 - 3 * 10^2 = +1237 & & q_2 = 1 \\
+1237 - 3 * 10^2 = +937 & & q_2 = 2 \\
+\ 937 - 3 * 10^2 = +637 & & q_2 = 3 \\
+\ 637 - 3 * 10^2 = +337 & & q_2 = 4 \\
+\ 337 - 3 * 10^2 = +37 & & q_2 = 5 \\
+\ 37 - 3 * 10^2 = -263 & & q_2 = 6 \\
-\ 263 + 3 * 10^2 = +37 & \text{restore} & \boxed{q_2 = 5} \\
+\ 37 - 3 * 10^1 = +7 & & q_1 = 1 \\
+\ 7 - 3 * 10^1 = -23 & & q_1 = 2 \\
-\ 23 + 3 * 10^1 = +7 & \text{restore} & \boxed{q_1 = 1} \\
+\ 7 - 3 * 10^0 = +4 & & q_0 = 1 \\
+\ 4 - 3 * 10^0 = +1 & & q_0 = 2 \\
+\ 1 - 3 * 10^0 = -2 & & q_0 = 3 \\
-\ 2 + 3 * 10^0 = +1 & \text{restore} & \boxed{q_0 = 2}
\end{array}
$$

For binary representation, the restoring division is simply a process of quotient digit selection from the set $\{0, 1\}$. The selection is performed according to the following recursive relation:

$$R(i+1) - q_i * d * 2^i = R(i).$$

We start by assuming $q_i = 1$; therefore, subtraction is performed:

$$R(i+1) - D * 2^i = R(i).$$

Consider the following two cases (for simplicity, assume that dividend and divisor are positive numbers):

**Case 1**: If $R(i) \geq 0$, then the assumption was correct, and $q_i = 1$.

**Case 2**: If $R(i) < 0$, then the assumption was wrong, $q_i = 0$, and restoration is necessary.

Let us illustrate the restoring division process for a binary division of 29/3:

$$
\begin{array}{rll}
29 - 3 * 2^4 = -19 & & q_4 = 1 \\
-19 + 3 * 2^4 = +29 & \text{restore} & \boxed{q_4 = 0} \\
29 - 3 * 2^3 = +5 & & \boxed{q_3 = 1} \\
+5 - 3 * 2^2 = -7 & & q_2 = 1 \\
-7 + 3 * 2^2 = +5 & \text{restore} & \boxed{q_2 = 0} \\
+5 - 3 * 2^1 = -1 & & q_1 = 1 \\
-1 + 3 * 2^1 = +5 & \text{restore} & \boxed{q_1 = 0} \\
+5 - 3 * 2^0 = +2 & & \boxed{q_0 = 1}
\end{array}
$$

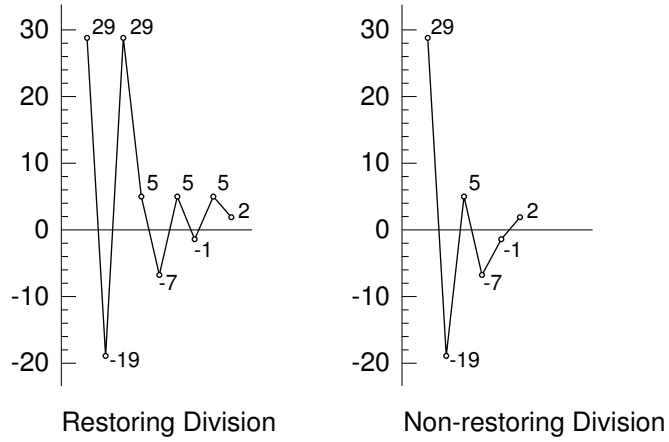


Figure 5.1: Graphical illustration of partial remainder computations in restoring and nonrestoring division.

The left side of Figure 5.1 graphically illustrates the preceding division process.

Using the following terminology,

$$\begin{aligned} Y &= \text{Dividend} \\ Q &= \text{Quotient (all quotient bits)} \\ (0) &= \text{Final Remainder} \\ D &= \text{Divisor} \end{aligned}$$

we have the following relationships:

$$\begin{aligned} Y &= Q * D + R(0) \\ Q &= q_4 * 2^4 + q_3 * 2^3 + q_2 * 2^2 + q_1 * 2^1 + q_0 * 2^0, \end{aligned}$$

and in the above example:

$$\begin{aligned} Y &= 29 \quad \text{and } D = 3 \\ Q &= 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 9 \\ 29 &= 9 * 3 + 2. \end{aligned}$$

It is obvious that, for $n$ bits, we may need as many as $2n$ cycles to select all the quotient digits; that is, there are $n$ cycles for the trial subtractions, and there may be an additional $n$ cycles for the restoration. However, these restoration cycles can be eliminated by a more powerful class of division algorithm: *nonrestoring division.*

## 5.2 Multiplicative Algorithms

Algorithms of this second class obtain a reciprocal of the divisor, and then multiply the result by the dividend. Thus, the main difficulty is the evaluation of a reciprocal. Flynn [12] points

out that there are two main ways of iteration to find the reciprocal. One is the series expansion, and the other is the Newton–Raphson iteration.

### 5.2.1 Division by Series Expansion

The series expansion is based on the Maclaurin series (a special case of the familiar Taylor series). Let $b$, the divisor, equal $1 + x$.

$$g(X) = \frac{1}{b} = \frac{1}{1+X} = 1 - X + X^2 - X^3 + X^4 - \dots$$

Since $X = b - 1$, the above can be factored ($0.5 \leq b < 1.0$):

$$\frac{1}{b} = (1-X)(1+X^2)(1+X^4)(1+X^8)(1+X^{16})\dots$$

The two's complement of $1 + X^n$ is $1 - X^n$, since:

$$2 - (1 + X^n) = 1 - X^n.$$

Conversely, the two's complement of $1 - X^n$ is $1 + X^n$. This algorithm was implemented in the IBM 360/91 [1], where division to 32-bit precision was evaluated as follows:

1. $(1-X)(1+X^2)(1+X^4)$ is found from a ROM look-up table.
2. $1 - X^8 = \left[(1-X)(1+X^2)(1+X^4)\right](1+X)$.
3. $1 + X^8$ is the two's complement of $1 - X^8$.
4. $1 - X^{16}$ is computed by multiplication $(1+X^8)(1-X^8)$.
5. $1 + X^{16}$ is the two's complement of $1 - X^{16}$.
6. $1 - X^{32}$ is the product of $(1+X^{16})(1-X^{16})$.
7. $1 + X^{32}$ is the two's complement of $(1 - X^{32})$.

In the ROM table lookup, the first $i$ bits of the $b$ are used as an address of the approximate quotient. Since $b$ is bit-normalized ($0.5 \leq b < 1$), then $|X| \leq 0.5$ and $|X^{32}| \leq 2^{-32}$; i.e., 32-bit precision is obtained in Step 7.

The careful reader of the preceding steps will be puzzled by a seeming sleight-of-hand. Since all divisors of the form $b_0 \dots b_i xxx \dots$ have same leading digits, they will map into the same table entry *regardless* of the value of $0.00 \dots 0xxxx \dots$. How, then, does the algorithm use the different trailing digits to form the proper quotient?

$$\frac{1}{b} = \frac{1}{1+X} = \underbrace{(1-X)(1+X^2)(1+X^4)}\dots$$

table entry—approximate quotient.

If we wish the quotient of $1/b$,
Suppose we look up the product of the indicated three terms. Since our lookup cannot be exact,

we have actually found

$$(1-X)(1+X^2)(1+X^4)+\epsilon_0.$$

Let us make the table sufficiently large so that

$$|\epsilon_0| \leq 2^{-9}.$$

Now, in order to find $1+X^8$, multiply the above by $b$ (i.e., the entire number $.b_0 \ldots b_8 xxxx \ldots$). Then, since $b = 1+X$:

$$\overbrace{(1+X)}^{b}\overbrace{(1-X)(1+X^2)(1+X^4)}^{\text{Table entry}} = 1-X^8$$

$$\underbrace{\qquad}_{(1-X^2)}$$

$$\underbrace{\qquad\qquad}_{(1-X^4)}$$

$$\underbrace{\qquad\qquad\qquad}_{(1-X^8)}$$

Thus, by multiplying the table entry by $b$, we have found

$$(1-X^8)+b\epsilon_0.$$

Upon complementation, we get:

$$1+X^8-b\epsilon_0,$$

and multiplying, we get:

$$1-X^{16}+2X^8\epsilon_0 b-(b\epsilon_0)^2.$$

Since $X = b-1$, the new error is actually

$$\epsilon_1 = 2b(b-1)^8\epsilon_0-(b\epsilon_0)^2,$$

whose max value over the range

$$\frac{1}{2} \leq b < 1$$

occurs at

$$b = \frac{1}{2};$$

thus,

$$\epsilon_1 < 2^{-8}\epsilon_0 - 2^{-2}\epsilon_0^2 = \epsilon_0(2^{-8}-2^{-2}\epsilon_0).$$

If, in the original table, $\epsilon_0$ was selected such that

$$|\epsilon_0| \leq 2^{-9},$$

then

$$|\epsilon_1| < 2^{-17}.$$

Thus, the error is decreasing at a rate equal to the increasing accuracy of the quotient.

The ROM table for quotient approximations warrants some further discussion, as the table structure is somewhat deceptive. One might think, for example, that a table accurate to $2^{-8}$ would be a simple structure $2^8 \times 8$, but division is not a linear function with the same range and domain. Thus, the width of an output is determined by the value of the quotient; when $1/2 \leq b < 1$, the quotient is $2 \geq q > 1$. The table entry should be 10 bits: $xx.xxxxxxxx$ in the example. Actually, by recognizing the case $b = 1/2$ and avoiding the table for this case, $q$ will always start $1.xx\ldots x$, the "1" can be omitted, and we again have 8 bits per entry.

The size of the table is determined by the required accuracy. Suppose we can tolerate error no greater than $\epsilon_0$. Then

$$\left| \frac{1}{b} - \frac{1}{b - 2^{-n}} \right| \leq \epsilon_0.$$

That is, when truncating $b$ at the $n$th bit, the quotient approximation must not differ from the true quotient by more than $\epsilon_0$.

$$\left| \frac{b - 2^{-n} - b}{b^2 - b2^{-n}} \right| \leq \epsilon_0.$$

$$2^{-n} \leq b^2\epsilon_0 - b2^{-n}\epsilon_0.$$

Since $b^2\epsilon_0 \gg b2^{-n}\epsilon_0$ $(1/2 \leq b < 1)$, we rewrite as

$$2^{-n} \leq b^2\epsilon_0.$$

Thus, if $|\epsilon_0|$ were to be $2^{-9}$,

$$2^{-n} \leq 2^{-9} \cdot 2^{-2},$$

$$n = 11 \text{ bits}.$$

Now again by recognizing the case $b = 1/2$ and that the leading bit of $b = 0.1x$, we can reduce the table size; i.e., $n = 10$ bits.

### 5.2.2 The Newton–Raphson Division

The Newton–Raphson iteration is based on the following procedure to solve the equation $f(X) = 0$ [43]:

- Make a rough graph $y = f(X)$.
- Estimate the root where the $f(X)$ crosses the X axis.
- This estimate is the first approximation; call it $X_1$.
- The next approximation, $X_2$, is the place where the tangent to $f(X)$ at $(X_1, f(X_1))$ crosses the X axis.

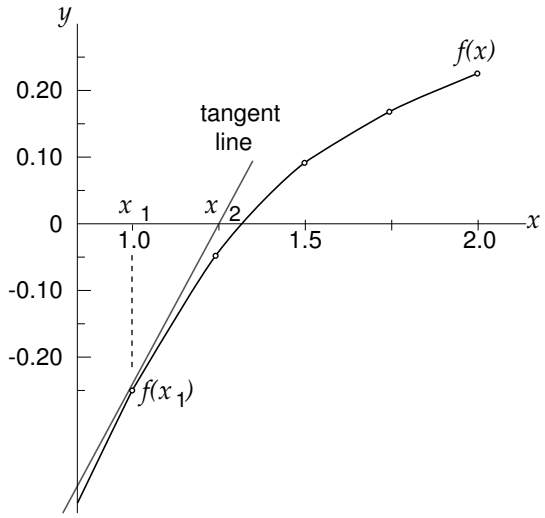


Figure 5.2: Plot of the curve $f(X) = 0.75 - \frac{1}{X}$ and its tangent at $f(X_1)$, where $X_1 = 1$ (first guess). $f'(x_1) = \frac{\delta y}{\delta x}$.

- From Figure 5.2, the equation of this tangent line is:

$$y - f(X_1) = f'(X_1)(X - X_1).$$

- The tangent line crosses the X axis at $X = X_2$ and $y = 0$.

$$0 - f(X_1) = f'(X_1)(X_2 - X_1),$$

$$X_2 = X_1 - \frac{f(X_1)}{f'(X_1)}.$$

- More generally,

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}.$$

- *Note*: the resulting subscripted values $(X_i)$ are successive approximations to the quotient; they should not be confused with the unsubscripted $X$ used in the preceding section on binomial expansion where $X$ is always equal to $b - 1$.

The preceding formula is a recursive iteration that can be used to solve many equations. In our specific case, we are interested in computing the reciprocal of $b$. Thus, the equation $f(X) = \frac{1}{X} - b = 0$ can be solved using the above recursion. Note that if:

$$f(X) = \frac{1}{X} - b,$$

then

$$f'(X) = -\left(\frac{1}{X}\right)^2,$$

and at $X = X_n$

$$f'(X_n) = -\left(\frac{1}{X_n}\right)^2.$$

After substitution, the following recursive solution for reciprocal is obtained:

$$X_{n+1} = X_n(2 - bX_n),$$

where $X_0 = 1$.

The following decimal example illustrates the simplicity and the quadratic convergence of this scheme:

**Example 5.1**

Find $\frac{1}{b}$, where $b = 0.75$ (error $= \epsilon$).

$$\begin{array}{lll} X_0 = & = 1 & \epsilon_1 = 0.333334 \\ X_1 = 1(2 - 0.75) & = 1.25 & \epsilon_2 = 0.083334 \\ X_2 = 1.25\,(2 - (1.25 \times 0.75)) & = 1.328125 & \epsilon_3 = 0.005208 \\ X_3 = X_2\,(2 - (1.328125 \times 0.75)) & = 1.333313 & \epsilon_4 = 0.000021 \end{array}$$

The quadratic convergence of this scheme is proved below. That is, $e_{i+1} \leq (e_i)^2$:

$$\begin{aligned} X_{i+1} &= X_i(2 - bX_i) \\ \text{to find} \quad \epsilon_i &= \frac{1}{b} - X_i \\ \epsilon_{i+1} &= \frac{1}{b} - X_{i+1} \\ \epsilon_{i+1} &= \frac{1}{b} - [X_i(2 - bX_i)] = \frac{1 - 2bX_i + (bX_i)^2}{b} \\ \text{but} \quad (\epsilon_i)^2 &= \frac{(1 - bX_i)^2}{b^2} = \frac{1 - 2bX_i + (bX_i)^2}{b^2}. \end{aligned}$$

Substituting for $X_i$,

$$\begin{aligned} \epsilon_{i+1} &= \frac{1 - 2b(\frac{1 - b\epsilon_i}{b}) + (1 - b\epsilon_i)^2}{b} \\ \epsilon_{i+1} &= 1 - 2 + 2b\epsilon_i + 1 - 2b\epsilon_i + b^2\epsilon_i/b \\ \epsilon_{i+1} &= b{\epsilon_i}^2. \end{aligned}$$

(Recall that $b < 1$). $\diamondsuit$

The division execution time, using the Newton–Raphson approximation, can be reduced by using a ROM look–up table. For example, computing the reciprocal of a 32–bit number can start by using $1024 \times 8$ ROM to provide the 8 most significant bits; the next iteration provides 16 bits, and the third iteration produces a 32–bit quotient. The Newton–Raphson seems similar in many

ways to the previously discussed binomial approximation. In fact, for the Newton–Raphson iteration:

$$X_{i+1} = X_i(2 - bX_i).$$

If $X_0 = 1$, then

$$\begin{aligned} X_1 &= (2-b) \\ X_2 &= (2-b)(2-2b+b^2) \\ &= (2-b)(1+(b-1)^2) \\ &\vdots \\ X_i &= (2-b)(1+(b-1)^2)(1+(b-1)^4)\ldots(1+(b-1)^{2i}) \end{aligned}$$

which is exactly the binomial series when $X = b - 1$.

Thus, the Newton–Raphson iteration on $f(X) = \frac{1}{X} - b$ and the binomial expansion of $\frac{1}{b} = \frac{1}{1+X}$ are different ways of viewing the same algorithm [12].

## 5.3 Additional Readings

Session 14 of the 1980 WESCON included several good papers on the theme of "Hardware Alternative for Floating Point Processing."

Undheim [44] describes the floating point processor of the NORD-500 computer, which is made by NORSK-DATA in Norway. The design techniques are very similar to the ones described in this book, where a combinatorial approach is used to obtain maximum performance. The entire floating point processor is made of 579 ICs and it performs floating point multiplication (64 bits) in 480ns.

Birkner [2] describes the architecture of a high-speed matrix processor which uses a subset of the proposed IEEE (short) floating point format for data representation. The paper describes some of the tradeoff used in selecting the above format, and it also discuss the detailed implementation of the processor using LSI devices.

Cheng [5] and McMinn [26] describe single chip implementation of the proposed IEEE floating point format. Cheng describes the AMD 9512, and McMinn the Inter 8087.

Much early literature was concerned with higher radix subtractive division. Robertson [31] was a leader in the development of such algorithms. Both Hwang [23] and Spaniol [33] contain reviews of this literature.

Flynn [12] provides a review of multiplicative division algorithms.

## 5.4 Exercises

1. Using restoring two's complement division, perform $\frac{a}{b}$ where $a = 0.110011001100$ and $b = 0.100111$. Show each iteration.

2. Repeat the above using nonrestoring two's complement division.

3. Using the Newton–Raphson iteration, compute $1/b$ where $b = .9$; $b = .6$; $b = .52$.

4. Construct a look-up table for two decimal digits (20 entries only; i.e., divisors from .60 to .79). Use this table to find $b = 0.666$.

5. An alternate Newton–Raphson iteration uses $f(x) = \frac{X-1+1/b}{X-1}$ (converges quadratically toward the complement of the reciprocal), which has a root at the complement of the quotient.

   (a) Find the iteration.

   (b) Compute the error term.

   (c) Use this to find $1/b$ when $b = .9$ and $b = .6$.

   (d) Comment on this algorithm as compared to that described in the text.

6. Another suggested approach uses:

$$f(x) = \exp\left[\frac{-1}{b(1-bx)}\right].$$

   (This recursion is unstable and converges very slowly.) Repeat problem 3 for this function.

7. A hardware cube-root function $a^{1/3}$ is desired based on the Newton–Raphson iteration technique. Using the function

$$f(x) = x^3 - a,$$

   (a) Find the iteration ($x_{i+1} =$ _____).

   (b) Find the first two approximations to $.58^{1/3}$ using the iteration found in (a).

   (c) Show how the convergence (error term) would be found for this iteration (i.e., show $e_{i+1}$ in terms of $e_i$). *Do not simplify!*

8. A new divide algorithm (actually a reciprocal algorithm, $1/b$) has been suggested based on a Newton–Raphson iteration, based on finding the root of:

$$f(x) = b^2 - 1/x^2 = 0.$$

   Will this work? If not, explain why not.

   If so, find the iteration and compare (time required and convergence rate) with other Newton–Raphson based approaches.

9. Two functions have been proposed for use in a Newton–Raphson iteration to find the reciprocal ($1/b$). Answer the following questions for each function:

   (a) Will the iteration converge to $1/b$?

      i. $f(x) = x^2 - 1/b = 0$

      ii. $f(x) = \frac{1}{x^2} - b = 0$

   (b) Find the iteration.

      i. $f(x) = x^2 - 1/b = 0$

      ii. $f(x) = \frac{1}{x^2} - b = 0$

   (c) Is this a practical scheme—will it work in a processor?

i. $f(x) = x^2 - 1/b = 0$
ii. $f(x) = \frac{1}{x^2} - b = 0$

(d) Is it better than the scheme outlined in the chapter?

i. $f(x) = x^2 - 1/b = 0$
ii. $f(x) = \frac{1}{x^2} - b = 0$

# Bibliography


[1] S. F. Anderson, J. G. Earle, R. E. Goldscmidt, and D. M. Powers. The IBM System 360/91 floating-point execution unit. *IBM Journal of Research and Development.*, 11(1), January 1967.

[2] D. A. Birkner. High speed matrix processor using floating point representation. In *Conference Record, WESCON*, 1980. Paper no. 14/3.

[3] A. D. Booth. A signed binary multiplication technique. *Qt. J. Mech. Appl. Math.*, 4, Part 2, 1951.

[4] J. F. Brennan. The fastest time of addition and multiplication. *IBM Research Reports*, 4(1), 1968.

[5] S. Cheng and Rallapalli K. Am 9512: Single chip floating point processor. In *Conference Record, WESCON*, 1980. Paper no. 14/4.

[6] W. J. Cody, Jr. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22, June 1973.

[7] W. J. Cody, Jr. Analysis of proposals for the floating-point standard. *Computer*, March 1981.

[8] J. T. Coonen. Specifications for a proposed standard for floating point arithmetic. Technical Report Memorandum number UCB/ERL M78/72, University of California, January 1979.

[9] J. T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, January 1980.

[10] J. T. Coonen. Underflow and the denormalized numbers. *Computer*, March 1981.

[11] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34, March 1965.

[12] M. J. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8), August 1970.

[13] B. Fraley. Zeros and infinities revisited and gradual underflow, December 1978.

[14] D. D. Gajski. Parallel compressors. *IEEE Transactions on Computers*, C-29(5), May 1980.

[15] H. L. Garner. The residue number system. *IRE Trans. Electronic Computers*, EC-8, June 1959.

[16] H. L. Garner. Number systems and arithmetic. *Advances in Computers*, 6, 1965.

[17] H. L. Garner. Survey of recent contributions to computer arithmetic. *IEEE Transactions on Computers*, December 1976.

[18] H. L. Garner. Theory of computer addition and overflows. *IEEE Transactions on Computers*, April 1978.

[19] M. Ginsberg. Numerical influences on the design of floating point arithmetic for microcomputers. *Proc. 1st Annual Rocky Mountain Symp. Microcomputers*, August 1977.

[20] H. W. Gschwind. *Design of Digital Computers.* Springer-Verlag, New York, 1967.

[21] B. Hasitume. Floating point arithmetic. *Byte*, November 1977.

[22] D. Hough. Applications of the proposed IEEE-754 standard for floating-point arithmetic. *Computer*, March 1981.

[23] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley and Sons, New York, 1978.

[24] H. Ling. High-speed binary adder. *IBM Journal of Research and Development*, 25(2 and 3), May 1981.

[25] J. D. Marasa and D. W. Matula. A simulative study of correlated error propagation in various finite-precision arithmetic. *IEEE Transactions on Computers*, C-22, June 1973.

[26] C. McMinn. The Intel 8087: A numeric data processor. In *Conference Record, WESCON*, 1980. Paper no. 14/5.

[27] R. D. Merrill, Jr. Improving digital computer performance using residue number theory. *IEEE Transactions on Electronic Computers*, EC-13(2):93–101, April 1964.

[28] J. R. Newman. *The World of Mathematics.* Simon and Schuster, New York, 1956.

[29] F. D. Parker. *The Structure of Number Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[30] M. Payne and W. Strecker. Draft proposal for floating point standard, December 1978.

[31] J. E. Robertson. A new class of digital division methods. *IRE Trans. Electronic Computers*, EC-5:65–73, June 1956.

[32] J. Sklansky. Conditional-sum addition logic. *Trans. IRE*, EC-9(2), June 1960.

[33] O. Spaniol. *Computer Arithmetic: Logic and Design.* John Wiley and Sons, New York, 1981.

[34] P. M. Spira. Computation times of arithmetic and boolean functions in $(d, r)$ circuits. *IEEE Transactions on Computers*, C-22, June 1973.

[35] M. L. Steinard and W. D. Munro. *Introduction to Machine Arithmetic.* Addison-Wesley, Reading, MA, 1971.

[36] W. J. Stenzel *et al.* A compact high-speed multiplication scheme. *IEEE Transactions on Computers*, C-26(10), October 1977.

[37] P. H. Sterbenz. *Floating Point Computation.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[38] D. Stevenson. A proposed standard for binary floating-point arithmetic. *Computer*, pages 51–62, March 1981.

[39] H. S. Stone. *Discrete Mathematical Structures.* Science Research Associates, Chicago, 1973.

[40] H. S. Stone. *Introduction to Computer Architecture.* Science Research Associates, Chicago, 1975.

[41] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* McGraw-Hill, New York,, 1967.

[42] Texas Instruments, Inc. TTL Data Book, 1976.

[43] G. B. Thomas. *Calculus and Analytic Geometry.* Addison–Wesley, Reading, MA, 1962.

[44] T. Undheim. Combinatorial floating-point processors as an integral part of the computer. In *Conference Record, WESCON*, 1980. paper no. 14/1.

[45] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, EC-13, February 1964.

[46] H. S. Warren, Jr., A. S. Fox, and P. W. Markstein. Modulus division on a two's complement machine. *IBM Research Report No. RC7712*, June 1979.

[47] S. Waser. State of the art in high-speed arithmetic ics. *Computer Des.*, July 1978.

[48] R. W. Watson and C. W. Hastings. Self-checked computation using residue arithmetic. *Proceedings of the IEEE*, 54(12), December 1966.

[49] A. Weinberger and J. L. Smith. A one-microsecond adder using one-megacycle circuitry. *IRE Trans. Electronic Computers*, EC-5:65–73, June 1956.

[50] R. E. Wiegel. Methods of binary additions. Technical Report Report number 195, Department of Computer Science, University of Illinois, Urbana, February 1966.

[51] S. Winograd. On the time required to perform addition. *Journal ACM*, 12(2), 1965.

[52] S. Winograd. On the time required to perform multiplication. *Journal ACM*, 14(4), 1967.

[53] J. M. Yohe. Roundings in floating-point arithmetic. *IEEE Transactions on Computers*, C-22, June 1973.