Computer Arithmetic (temporary title, work in progress)

Current additions done by Hossam A. H. Fahmy with permission from Michael J. Flynn

Starting material based partly on the book:
Introduction to Arithmetic for Digital Systems Designers
by Shlomo Waser and Michael J. Flynn
Originally published by Holt, Rinehard & Winston,
New York, 1982 (Out of print)

Contents

List of Figures

List of Tables

Chapter 1

Numeric Data Representation

Arithmetic is the science of handling numbers and operating on them. This book is about the arithmetic done on computers. To fulfill its purpose, there is a need to describe the computer representations of the different numbers that humans use and the implementation of the basic mathematical operations such as addition, subtraction, multiplication and division. These operations can be implemented in software or in hardware. The focus of this volume is to introduce the hardware aspects of computer arithmetic. We sprinkled the text freely with examples of different levels of complexity as well as exercices. The exercises are there to be attempted before turning to the solutions at the end of the book. The solutions often expand the concepts further but will not benefit much unless you try to work through the exercises first. At the end of each chapter, there are further problems that are left for the student to solve.

After finishing the book, the reader should be familiar with the fundamentals of the field and able to design simple logic circuits to perform the basic operations. The text often refers to further readings for advanced material. We believe that such a presentation helps introduce new designers to the advanced parts of the field. This presentation style also does not get into too many details and gives a general background for those in other specialities such as computer architecture, VLSI design, and numerical analysis who might be interested to strengthen their knowledge of computer arithemtic.

In fact, if one contemplates the design of large digital integrated circuits one finds that it is mostly composed of four main entities:

Memories are used for temporary storage of results (registers), for reducing the time delay of retrieving the information (caches), or as the main store of information (main memories).

Control logic blocks handle the flow of information and assure that the circuit performs what is desired by the user.

Datapath blocks are the real engine that performs the work. These are mainly circuits performing either some arithmetic or logic operation on the data.

Communications between all the elements is via wires usually arranged in the form of buses.

In our following discussions, we mainly focus on the datapath and see how it interacts with the three other elements present in digital circuits.

A good optimization of the arithmetic blocks results in an improved datapath which directly leads to a better overall design. Such an optimization might be to improve the speed of operation, to lower the power consumption, to lower the cost of the circuit (usually related to the number of gates used or the area on the chip), or to improve any other desired factor. As the different possibilities for implementing the arithmetic operations are explained, we will see that the designer has a large array of techniques to use in order to fulfill the desired outcome. A skilled designer chooses the best technique for the problem at hand. We hope to help future designers make these informed choices by presenting some simple tools to measure the different factors for the options that they evaluate.

1.1 Infinite aspirations and finite resources

Using computers to perform arithemtic introduces some constraints to what can be done. The main one is the limit on the number of digits used. This limitation translates into the representation of only a finite set of numbers. All other numbers from the set of real numbers are not representable. Some of the effects of this finitude are clear. Definitely any irrational number with an infinite number of digits after the fractional point is not representable. The same case applies for rational numbers whose representation as a fractional number is beyond the number of digits available. For example, if we assume a decimal number system with five digits after the fractional point then the number $1234567/500000 = 2.469134$ is not represented exactly. Increasing the number of digits used to six may help to include an accurate representation for that rational number, however, the numbers $1/7$, $\sqrt{2}$, e and π are still not represented.

This finitude also means that there is an upper bound on the numbers that are representable. If an arithmetic operation has a result beyond this upper limit a condition called overflow occurs and either the hardware or the software running on top of it must handle the situation differently to get a meaningful result. Similarly, a lower bound on the minimum absolute value of a fraction exist and a condition called underflow occurs if an arithmetic operation has a result below this limit.

Said differently, the primary problem in computer arithmetic is the mapping from the infinite number systems of mathematics to the finite representational capability of the machine. Finitude is the principal characteristic of a computer number system. Almost all other considerations are a direct consequence of this finitude.

The common solution to this problem is the use of modular arithmetic. In this scheme, every integer from the infinite number set has one unique representation in a finite system. However, now a problem of multiple interpretations is introduced—that is, in a modulo 8 system, the number 9 is mapped into the number 1. As a result of mapping, the number 1 corresponds in the infinite number system to $1, 9, 17, 25$, etc.

As humans originaly used numbers to count, we start by the natural numbers representing positive integers and see the effect of finitude and modular arithmetic on such numbers. General integer numbers including the representation of negative numbers follow. The presentation of the basic arithmetic operations on integers is given next. Once these fundamentals are laid out, we refer to further readings that is related. The following chapter deals with the representations of real numbers and the operations involving them.

1.2 Natural Numbers, Finitude, and Modular Arithmetic

The historical need for numbers and their first use was for counting. Even nowadays, the child's numerical development starts with counting. The counting function is accomplished by the infinite set of numbers 1, 2, 3, 4, . . . , which are described as natural numbers. These numbers have been used for thousands of years, and yet only in the 19^{th} century were they described precisely by Peano (1858–1932). The following description of Peano's postulates is adapted from Parker [Parker, 1966].

Postulate 1: For every natural number x, there is a unique natural number which we call the successor of x and which is denoted by $s(x)$.

Postulate 2: There is a unique natural number which we call *1*.

Postulate 3: The natural number 1 is not the successor of any natural number.

Postulate 4: If two natural numbers x and y are such that $s(x) = s(y)$, then $x = y$.

Postulate 5: (Principle of Mathematical Induction): Let $\mathcal{M}$ be a subset of the natural numbers with the following properties:

(a) 1 is a member of $\mathcal{M}$;

(b) For any x that belongs to $\mathcal{M}$, $s(x)$ also belongs to $\mathcal{M}$.

Then $\mathcal{M}$ is the set of natural numbers.

Later on, we will show that all other number systems (negative, real, rational) can be described in terms of natural numbers. At this point, our attention is on the problem of mapping from the infinite set to a finite set of numbers.

Garner [Garner, 1965] show that the most important characteristic of machine number systems is finitude. Overflows, underflows, scaling, and complement coding are consequences of this finitude.

On a computer, the infinite set of natural numbers needs to be represented by a finite set of numbers. Arithmetic that takes place within a closed set of numbers is known as *modular arithmetic*. Brennan [Brennan, 1968] provides the following examples of modular arithmetic in everyday life. The clock tells time in terms of the closed set (modules) of 12 hours, and the days of the week all fall within modulo 7. If the sum of any two numbers within such a modulus exceeds the modulus, only the remainder number is considered; e.g., eight hours after seven o'clock, the time is three o'clock, since

$$(8 + 7) \text{ modulo } 12 = \text{remainder of } \frac{15}{12} = 3.$$

Seventeen days after Tuesday, the third day of the week, the day is Friday, the sixth day of the week, since

$$(17+3) \text{ modulo } 7 = \text{remainder of } \frac{20}{7} = 6.$$

In modular arithmetic, the property of congruence (having the same remainder) is of particular importance. By definition [Steinard and Munro, 1971]:

> **If μ is a positive integer, then any two integers N and M are *congruent*, modulo μ, if and only if there exists an integer K such that**
>
> $$N - M = K\mu.$$
>
> **Hence,**
>
> $$N\mathbf{mod}_{\mu} \equiv M\mathbf{mod}_{\mu},$$
>
> **where μ is called the modulus.**

Informally, the modulus is the quantity of numbers within which a computation takes place, i.e. $(0, 1, 2, 3, \ldots, \mu - 1.)$

Example 1.1 If $\mu = 256$, $M = 258$, and $N = 514$ are M and N congruent $\mathbf{mod}_{\mu}$?
Solution: The modulo operation yields

$$514\mathbf{mod_{256}} = 2\mathbf{mod_{256}}$$

and

$$258\mathbf{mod_{256}} = 2\mathbf{mod_{256}},$$

i.e., they are congruent $\mathbf{mod_{256}}$, and

$$514 - 258 = 1 \times 256,$$

i.e., $K = 1$.

1.2.1 Properties

Congruence has the same properties with respect to the operations of addition, subtraction, and multiplication, or any combination.

If $N' = N\mathbf{mod}_{\mu}$ and $M' = M\mathbf{mod}_{\mu}$, then

$$\begin{aligned}
(N+M)\mathbf{mod}_{\mu} &= (N'+M')\mathbf{mod}_{\mu} \\
(N-M)\mathbf{mod}_{\mu} &= (N'-M')\mathbf{mod}_{\mu} \\
(N\times M)\mathbf{mod}_{\mu} &= (N'\times M')\mathbf{mod}_{\mu}
\end{aligned}$$

Example 1.2 If $\mu = 4$, $N = 11$ and $M = 5$ check the three operations.
Solution: Since $(11)\mathbf{mod_4} = 3$ and $(5)\mathbf{mod_4} = 1$, we get

$$\begin{aligned}(3+1)\mathbf{mod_4} &= (11+5)\mathbf{mod_4} \equiv 0,\\ (3-1)\mathbf{mod_4} &= (11-5)\mathbf{mod_4} \equiv 2, \text{and}\\ (3\times 1)\mathbf{mod_4} &= (11\times 5)\mathbf{mod_4} \equiv 3.\end{aligned}$$

$\overset{?}{\Longrightarrow}$ **Exercise 1.1 Can you prove that if $N' = N\mathbf{mod}_\mu$ and $M' = M\mathbf{mod}_\mu$, then $(N[+,-,\times]M)\mathbf{mod}_\mu = (N'[+,-,\times]M')\mathbf{mod}_\mu$ where $[+,-,\times]$ means any of the addition, subtraction, or multiplication operations?**

Negative numbers pose a small difficulty. If N is negative while μ is positive in the operation $N\mathbf{mod}_\mu$ then several conventions apply. Depending on how it is defined,

$$-7\mathbf{mod_3} \equiv -1 \text{ or } +2,$$

since

$$\frac{-7}{3} = -2 \text{ quotient}, \ -1 \text{ remainder}$$

or

$$\frac{-7}{3} = -3 \text{ quotient}, \ +2 \text{ remainder}.$$

For modulus operations, the usual convention is to choose the *least positive residue* (including zero). Unless otherwise specified, we will assume this convention throughout this book, even if we are dividing by a negative number such as $(-7)/(-3) = +2$. That is,

$$\frac{-7}{-3} = +3 \text{ quotient}, \ +2 \text{ remainder}.$$

In terms of conventional division, this is surprising, since one might expect

$$\frac{-7}{-3} = +2 \text{ quotient}, \ -1 \text{ remainder}.$$

We will distinguish between the two division conventions by referring to the former as *modulus division* and the latter as *signed division.* In signed division, the magnitude of the quotient is independent of the signs of the divisor and dividend. This distinction follows the work of Warren and his colleagues [Warren et al., 1979].

$\overset{?}{\Longrightarrow}$ **Exercise 1.2 For the operation of integer division $\pm 11 \div \pm 5$ find the quotient and remainder for each of the four sign combinations**

(a) **for signed division, and**

(b) **for modulus division.**

The division operation is defined as

$$\frac{a}{b} = q + \frac{r}{b},$$

where q is the quotient and r is the remainder. But even the modulus division operation does not extend as simply as the other three operations; for example,

$$\frac{3}{1} \neq \frac{11}{5}\mathbf{mod_4}.$$

Nevertheless, division is a central operation in modular arithmetic. It can be shown that for any modulus division M/μ, there is a unique quotient-remainder pair, and the remainder has one of the μ possible values $0, 1, 2, \ldots, \mu - 1$ which leads to the concept of *residue class*.

A residue class is the set of all integers having the same remainder upon division by the modulus μ. For example, if $\mu = 4$, then the numbers 1, 5, 9, 13 . . . are of the same residue class. Obviously, there are exactly μ residue classes, and each integer belongs to one and only one residue class. Thus, the modulus μ partitions the set of all integers into μ distinct and disjoint subsets called residue classes.

Example 1.3 If $\mu = 4$, find the residue classes.
Solution: In this case, there are four residue classes which partition the integers:

$$\begin{array}{llllllll}
\{\ldots, & -8, & -4, & 0, & 4, & 8, & 12, & \ldots\} \\
\{\ldots, & -7, & -3, & 1, & 5, & 9, & 13, & \ldots\} \\
\{\ldots, & -6, & -2, & 2, & 6, & 10, & 14, & \ldots\} \\
\{\ldots, & -5, & -1, & 3, & 7, & 11, & 15, & \ldots\}
\end{array}.$$

If we are not dealing with individual integers but only with the residue class of which the integer is a member, the problem of working with an infinite set is reduced to one of working with a finite set. This is a basic principle of number representation in computers.

Before we leave these points, let us check that you understand them thoroughly.

$\overset{?}{\Longrightarrow}$ **Exercise 1.3 If $\div_m$ denotes the modular division so that $N \div_m D$ result in q_m and r_m as the quotient and remainder while $\div_s$ (with q_s, r_s) denotes signed division, find q_s and r_s in terms of q_m and r_m.**

$\overset{?}{\Longrightarrow}$ **Exercise 1.4 Another type of division is possible; this is called "floor division." In this operation, the quotient is the greatest integer that is contained by (is less than or equal to) the numerator divided by the denominator (note that minus 3 is greater than minus 4). Find q_f, r_f in terms of q_m, r_m.**

1.2.2 Extending Peano's Numbers

Peano's numbers are the natural integers 1, 2, 3, . . ., but in real life we deal with more numbers. The historic motivation for the extension can be understood by studying some arithmetic operations. The operations of addition and multiplication (on Peano's numbers) result in numbers that are still described by the original Peano's postulates. However, subtraction of two numbers may result in negative numbers or zero. Thus, the extended set of all integers is

$$-\infty, \ldots, -2, -1, 0, 1, 2, \ldots + \infty,$$

and natural integers are a subset of these integers. The operation of division on integers may result in noninteger numbers. By definition, such a number is a rational number, which is represented exactly as a ratio of two integers. However, if the rational number is to be approximated as a single number, an infinite sequence of digits may be required for such a number, for example, $1/3 = 0.33333\ldots$. Between any two rational numbers, however small but finite their difference, lies an infinite number of other rational numbers and infinitely more numbers which cannot be expressed as rationals. We call these latter numbers *real* numbers and they include such constants as π and e. Real numbers can be viewed as all points along the number axis from $-\infty$ to $+\infty$.

Real numbers need to be represented in a machine with the characteristics of finitude. This is accomplished by approximating real numbers and rational numbers by terminating sequences of digits. Thus, all numbers (real, rational, and integers) can be operated on as if they were integers (provided scaling and rounding are done properly). We devote the remainder of this chapter to integers. Other numbers are discussed in subsequent chapters.

1.3 Integer Representation

The data representation to be described here is a weighted positional representation. The development for a weighted system was a particular breakthrough in ancient man's way of counting. While his hearthmate was simmering clams, and children demanding equal portions, to count seventeen shells he may have counted the first ten and marked something in the sand (to indicate 10), then counted the remaining seven shells. If his mark on the sand happened to look like 1, he could easily have generated the familiar (decimal) weighted positional number system.

The decimal system is also called base-10 system and its digits range from 0 to 9, i.e. from 0 to $10-1$. For the decimal system, 10 is called the radix and the digits usually go up to the radix minus one. The same idea applies for other systems. For example, in a binary (base-2) system the digits usually are 0 or 1, in a base-8 system the digits are usually 0 to 7. A number N with n digits ($d_{n-1}, \cdots, d_0$ in the radix β) is written as $d_{n-1}\, d_{n-2}\, d_{n-3} \cdots d_1\, d_0$. The d_0 represents the units or β^0 values, the d_1 represents the β^1 values, the d_2 represents the β^2 values and so on. The total value of N is $\sum_{i=0}^{i=n-1} d_i \beta^i$. Such a system is called a weighted positional number system since each position has a weight and the digits are multiplied by that weight. This system was invented in India and developed by the Muslims who called it hisab al-hind حِسَابُ الهِندِ [ibn Musa Al-Khawarizmi, circa 830 C.E.] or Indian reckoning in English.

That Indo-Arabic system was later introduced to Europe through the Islamic civilization in Spain and replaced the Roman numerals. That is the reason why the numerals 0 to 9 are known in the west as the Arabic numerals. A simple idea links the Roman system to the much older Egyptian system: the units have a symbol used to count them and that symbol is repeated to count for more than one. A group of five units has a different symbol. Ten units have another symbol, fifty units have yet another symbol and so on. This Roman system only survives today for special applications such as numbering the chapters of a book but is not in much use in arithmetic. Another number system that existed in history is the Babylonian system which was a sexadecimal system and it survives today in the way we tell the time by dividing the hour into sixty minutes and the minute into sixty seconds. Chapter 2 discusses the advantages gained from some alternative number systems.

Example 1.4 In the familiar decimal system, the base is $\beta = 10$, and the 4-digit number 1736 is:

$$1736 = 1 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 6.$$

In the binary system, $\beta = 2$, and the 5-digit number 10010 is:

$$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 = 18 \text{ (base 10)}.$$

The leading digit, d_m, is the most significant digit (*MSD*) or the most significant bit (*MSB*) for binary base. Similarly, d_0 is the least significant digit or bit—(*LSD* or *LSB*) respectively.

The preceding positional number system does not include a representation of negative numbers. Two methods are commonly used to represent signed numbers [Garner, 1965]:

Sign plus magnitude: Digits are represented according to the simple positional number system; an additional high-order symbol represents the sign. This code is natural for humans, but unnatural for a modular computer system.

Complement codes: Two types are commonly used; namely, *radix complement* code (**RC**) and *diminished radix* complement code (**DRC**). Complement coding is natural for computers, since no special sign symbology or computation is required. In binary arithmetic (base = 2), the **RC** code is called *two's complement* and the **DRC** is called *ones' complement.*

1.3.1 Complement Coding

Suppose for a moment that we had a modular number system with modulus 2μ. We could designate numbers in the range 0 to $\mu-1$ as positive numbers similar to the case of our previous modulus μ system, and *treat* numbers μ to $2\mu-1$ as negative, since they lie in the same residue class as numbers $-(\mu)$ to -1:

$$\begin{aligned} -1\mathbf{mod_{2\mu}} &= (2\mu-1)\mathbf{mod_{2\mu}};\\ -(\mu)\mathbf{mod_{2\mu}} &= (2\mu-\mu)\mathbf{mod_{2\mu}} = \mu\mathbf{mod_{2\mu}}. \end{aligned}$$

Mapping these negative numbers into large positive residues is called *complement coding.* We deal with $2\mu - x$ rather than $-x$. However, because both representations are congruent, they produce the same modular results.

Of course, "overflows" are a problem. These are results that appear as correct representations $\mathbf{mod}_{\mathbf{2}\mu}$, but are incorrect in our mapped $\mathbf{mod}_{\mu}$ system. If two positive or two negative numbers a and b have sum c, which exceeds $|\mu|$, overflow occurs and this must be detected. The decision to actually use the **RC** or the **DRC** makes the implementation details differ slightly.

1.3.2 Radix Complement Code—Subtraction Using Addition

We start the discussion by supposing that a number N is a positive integer of the form

$$N = d_m \cdot \beta^m + d_{m-1} \cdot \beta^{m-1} + \cdots + d_0.$$

$\overset{?}{\Longrightarrow}$ **Exercise 1.5 If the digits $d_i \in \{0, 1, \cdots, \beta - 1\}$, prove that the maximum value N may assume is $\beta^{m+1} - 1$.**

Now, suppose we wish to represent $-N$, a negative $m + 1$ digit number. We define the radix complement of N as

$$\mathbf{RC}(N) = \beta^{m+1} - N.$$

Clearly, the $\mathbf{RC}(N)$ is a nonnegative integer.

For ease of representation, let us assume that β is even and let $n = m+1$; then $\mathbf{RC}(N) = \beta^n - N$. Suppose P and N are n-digit numbers and we wish to compute $P - N$ using the addition operation. P and N may be either positive or negative numbers, as long as

$$\frac{\beta^n}{2} - 1 \geq P,\ N \geq \frac{-\beta^n}{2}.$$

In fact, $P - N$ is more accurately $(P - N)\mathbf{mod}_{\beta^n}$, and

$$(P - N)\mathbf{mod}_{\beta^n} = (P\mathbf{mod}_{\beta^n} - N\mathbf{mod}_{\beta^n})\mathbf{mod}_{\beta^n}.$$

However, if we replace $-N$ with $\beta^n - N$ the equality is unchanged. That is, by taking

$$(P\mathbf{mod}_{\beta^n} + (\beta^n - N)\mathbf{mod}_{\beta^n})\,\mathbf{mod}_{\beta^n},$$

we get

$$P\mathbf{mod}_{\beta^n} - N\mathbf{mod}_{\beta^n}.$$

The computation of $\beta^n - N$ is relatively straightforward.

$\overset{?}{\Longrightarrow}$ **Exercise 1.6 Prove that $\mathbf{RC}(N) = \beta^n - N$ (which is represented by $\mathbf{RC}(N)_m\mathbf{RC}(N)_{m-1} \cdots \mathbf{RC}(N)_0$) is given by this simple algorithm:**

1. **Scan the digits of N from the least significant side till you reach the first non-zero digit. Assume this non-zero digit is at position $i + 1$.**

2. **The digits of $\mathbf{RC}(N)$ are given by**

$$\mathbf{RC}(N)_j = \begin{cases} 0 & 0 \leq j \leq i \\ \beta - d_{i+1} & j = i + 1 \\ \beta - 1 - d_j & i + 2 \leq j \leq m \end{cases}$$

i.e., the first non-zero digit is subtracted from β and all the other digits at higher positions are subtracted from $\beta - 1$.

For example, in a three-position decimal number system, the radix complement of the positive number 245 is $1000 - 245 = 755$. In this system, $\beta = 10$, $n = 3$ and, according to our definition, 755 represents a negative number since $755 > \beta^n/2$.

This presentation of radix complement illustrates that by properly scaling the represented positive and negative numbers about zero, no special treatment of the sign is required. In fact, the most significant digit indicates the sign of the number. In the base 10 system, the digits 5, 6, 7, 8, 9 (in the most significant position) indicate negative numbers; i.e., three decimal digits represent numbers from $+499$ to -500.

Example 1.5 If $P = +250$ and $N = +245$ compute $P - N$ using the radix complement.
Solution:

$$\begin{array}{rcrl} 250 & \Rightarrow & 250 & \\ \underline{-245} & & \underline{+755} & \\ & & 1005 & \mathbf{mod_{1000}} \equiv 5 \end{array}$$

$\overset{?}{\Longrightarrow}$ **Exercise 1.7 In the specific case of the binary system, a most significant digit of 1 is an indication of negative numbers. A nice property follows for the two's complement binary system, If a binary number N is represented in two's complement form by the bit string $d_m d_{m-1} \cdots d_1 d_0$, then $N = (-1) d_m 2^m + \sum_{i=0}^{m-1} d_i 2^i$. Can you prove it?**

For the familiar case of even radix, a disadvantage of the radix complement code is the asymmetry around zero; that is, the number of negative numbers is greater by one than the number of positive numbers. However, this shortcoming is not a serious one. If the number zero is viewed as a positive number then there are as many positive numbers as there are negative numbers!

Although the operation of finding the radix complement is quite simple as shown in exercise 1.6 it is a sequential operation. We scan the digits in sequence and hence it takes time to perform it. The greatest disadvantage of the two's complement number system is this difficulty in converting from positive to negative numbers, and vice versa. This difficulty is the motivation [Stone, 1975] for developing the diminished radix complement code.

1.3.3 Diminished Radix Complement Code

By definition, the diminished radix complement of the previously defined number N, $\mathbf{DRC}(N)$ is $\beta^n - 1 - N$. In a decimal number system, this code is called *nines' complement*, and in binary system, it is called *ones' complement*.

The computation of the diminished radix complement (**DRC**) is simpler than that of the radix complement. Since, if $N \mathbf{mod}_{\beta^n} = d_{n-1} d_{n-2} \ldots d_0$, then for all $d_i (n - 1 \geq i \geq 0)$

$$\mathbf{DRC}(d)_i = \beta - 1 - d_i.$$

Since $\beta - 1$ is the highest valued symbol in a radix β system, no borrows can occur and the **DRC** digits can be computed independently.

Example 1.6 To verify their independence, calculate the digits representing the diminished radix complement of $N = +245$ starting from the most significant side.
Solution:

$$\begin{array}{rcl} 9-2 & = & 7 \\ 9-4 & = & 5 \\ 9-5 & = & 4 \\ \mathbf{DRC}(245) & = & 754 \end{array}$$

It is easy to verify that doing the digits in any order yields the same result.

This simplicity of the diminished radix complement computation comes at some expense in arithmetic operation as shown in the following example.

Example 1.7 Suppose we have two $\mathbf{mod_{99}}$ numbers P and N, i.e. each having two digits. The following operations are performed by allowing a carry to overflow to the third digit position. The operations are thus $\mathbf{mod_{1000}}$ originally, then we correct the result to $\mathbf{mod_{100}}$ and finally to $\mathbf{mod_{99}}$:

(i) $P = 47$, $N = 24$:

$$\begin{array}{rl} 47 & \\ +24 & \\ \cline{1-1} 071 & \quad 71\mathbf{mod_{100}} \equiv 71\mathbf{mod_{99}} = \text{result.} \end{array}$$

(ii) $P = 47$, $N = 57$:

$$\begin{array}{rl} 47 & \\ +57 & \\ \cline{1-1} 104 & \quad 4\mathbf{mod_{100}} \equiv 5\mathbf{mod_{99}} = \text{result.} \\ +1 & \\ \cline{1-1} 05 & \end{array}$$

(iii) $P = 47$, $N = 52$:

$$\begin{array}{rl} 47 & \\ +52 & \\ \cline{1-1} 099 & \quad 99\mathbf{mod_{100}} \equiv 0\mathbf{mod_{99}} = \text{result.} \end{array}$$

The $\mathbf{mod_{99}}$ result is the same as the $\mathbf{mod_{100}}$ result if the sum is less than 99. If the sum is an exact multiple of 99 the $\mathbf{mod_{99}}$ result is zero. On the other hand, if the sum exceeds 99 the $\mathbf{mod_{99}}$ result is greater than the $\mathbf{mod_{100}}$ result. We add one to the $\mathbf{mod_{100}}$ result in this latter case.
Basically, if the sum is an exact multiple of 99 the final result is zero, otherwise we add the carry into the third digit position to the $\mathbf{mod_{100}}$ result to get the $\mathbf{mod_{99}}$ result.

To state these findings more formally, since the arithmetic logic itself is always $\mathbf{mod}_{\beta^{\mathbf{p}}}$, (where $p \geq n$), we need to define the computation of the sum $S\mathbf{mod}_{\beta^{\mathbf{n}}\mathbf{-1}}$ in terms of $S\mathbf{mod}_{\beta^{\mathbf{n}}}$.

Two functions used throughout this book can help. We use the two symbols: $\lceil x \rceil$ and $\lfloor x \rfloor$, respectively for the ceiling and the floor of the real number x. *The ceiling function* is defined as the smallest integer that properly contains x; e.g., if $x = 1.33$, then $\lceil x \rceil = \lceil 1.33 \rceil = 2$. *The floor function* is defined as the largest integer contained by x, e.g., $\lfloor x \rfloor = \lfloor 1.33 \rfloor = 1$.

If S is initially represented as a $\mathbf{mod}_{\beta^{\mathbf{P}}}$ number, or the result of addition or subtraction of two numbers $\mathbf{mod}_{\beta^{\mathbf{n}}}$, then the conversion to a $\mathbf{mod}_{\beta^{\mathbf{n}}-\mathbf{1}}$ number, S', is

$$\text{If } S < \beta^n - 1 \quad \text{then} \quad S' = S.$$

That is,

$$S\mathbf{mod}_{\beta^{\mathbf{n}}} \equiv S\mathbf{mod}_{\beta^{\mathbf{n}}-\mathbf{1}} = S'.$$

If $S = \beta^n - 1$ or in general

$$S = k(\beta^n - 1),$$

where k is any integer, then

$$S' = 0.$$

Finally, if $\beta^n - 1 < S$, then S' must be increased by 1 (called the end around carry) for each multiple of $\beta^n - 1$ contained in S. Thus,

$$S' = \left(S + \lfloor \frac{S}{\beta^n - 1} \rfloor \right) \mathbf{mod}_{\beta^{\mathbf{n}}}.$$

That is, S' is S plus the largest integer contained by $\frac{S}{\beta^n - 1}$.

Since $\beta^n - 1$ is a represented element in n-digit arithmetic ($\mathbf{mod}_{\beta^{\mathbf{n}}}$ arithmetic), we have two equivalent representations for zero in the $\mathbf{mod}_{(\beta^{\mathbf{n}}-\mathbf{1})}$ case: $\beta^n - 1$ and 0.

The broader issue of $\beta^n - 1$ and β^n modular compatibility will be of interest to us again in Chapter 2. For the moment, we focus on a restricted version of this issue when using of the **DRC** in subtraction. In order to represent negative numbers using the **DRC**, we will partition the range of β^n representation as follows:

$$\underbrace{\begin{array}{llll} \beta^n - 1, & \cdots, & \frac{\beta^n}{2} + 1, & \frac{\beta^n}{2} \\ 0 & & & \textit{most} \\ & & & \textit{negative} \end{array}}_{\text{Negative numbers}} \underbrace{\begin{array}{llll} \frac{\beta^n}{2} - 1, & \frac{\beta^n}{2} - 2, & \cdots, & 0 \\ \textit{most} & & & 0 \\ \textit{positive} & & & \end{array}}_{\text{Positive numbers}}$$

Thus, any m-digit ($m = n - 1$) number P must be in the following range:

$$\frac{\beta^n}{2} - 1 \geq P \geq \frac{-\beta^n}{2} + 1.$$

Note that $\frac{\beta^n}{2}$ is congruent to (lies in the same residue class as) $\frac{-\beta^n}{2} + 1$ modulo $\beta^n - 1$, since

$$\left(\frac{-\beta^n}{2} + 1 \right) \mathbf{mod}_{\beta^{\mathbf{n}}-\mathbf{1}} \equiv \left((\beta^n - 1) - \frac{\beta^n}{2} + 1 \right) \mathbf{mod}_{\beta^{\mathbf{n}}-\mathbf{1}} \equiv \left(\frac{\beta^n}{2} \right) \mathbf{mod}_{\beta^{\mathbf{n}}-\mathbf{1}}.$$

So long as β has 2 as a factor, there will be a unique set of leading digit identifiers for negative numbers. For example, if $\beta = 10$, a negative number will have 5, 6, 7, 8, 9 as a leading digit.

$\overset{?}{\Longrightarrow}$ **Exercise 1.8 Is it really accurate to say "negative numbers" in the previous paragraph?**

Consider the computation $P - N$ using the diminished radix complement (**DRC**) with $\mathbf{mod}_{\beta^\mathbf{n}}$ arithmetic logic to be corrected to $\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}}$. P and N satisfy $\frac{\beta^n}{2} - 1 \geq P, N \geq \frac{-\beta^n}{2} + 1$ and due to the properties of modular arithmetic we have

$$(P - N)\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} \equiv (P\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} - N\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}})\,\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}}.$$

Since $P\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} = P$ and $-N\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} = \beta^n - 1 - N$ then

$$(P - N)\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} = (P + \beta^n - 1 - N)\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}} \equiv (P + \mathbf{DRC}(N))\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}}.$$

However, the basic addition logic is performed $\mathbf{mod}_{\beta^\mathbf{n}}$. We must thus correct the $\mathbf{mod}_{\beta^\mathbf{n}}$ difference, S, to find the $\mathbf{mod}_{\beta^\mathbf{n}-\mathbf{1}}$ difference, S'.

$$S = P + \beta^n - 1 - N.$$

If $S > \beta^n - 1$, then $S' = S + 1$; i.e., $P - N > 0$.
If $S < \beta^n - 1$, then $S' = S$; i.e., $P - N < 0$.
If $S = \beta^n - 1$, then $S' = 0$; i.e., $P = N$,
and *the result is zero* (i.e., one of the two representations).

$\overset{?}{\Longrightarrow}$ **Exercise 1.9 If $P = +250$ and $N = +245$ compute $P - N$ using the diminished radix complement.**

In summary, in the decimal system $-43 \Rightarrow 99 - 43 = 56$, and in the binary system $-3 \Rightarrow 111 - 011 = 100$. These examples illustrate the advantage of the diminished radix complement code—the ease of initial conversion from positive to negative numbers; the conversion is done by taking the complement of each digit. Of course, in the binary system, the complement is the simple Boolean *NOT* operation.

A disadvantage of the system is illustrated by taking the complement of zero; for example, in a 3-digit decimal system, the complement of zero $= 999 - 000 = 999$. Thus, the number zero has two representations: 000 and 999. (Note: the complement of the new zero is $999 - 999 = 000$.)

Another disadvantage is that the arithmetic logic may require correction of results (end-around carry)—see Chapter 3.

It is important to remember that the same bit pattern means different things when interpreted differently. Table 1.1 illustrates this fact for all the combinations of four bits and six different coding schemes. For the sign magnitude representation, the *MSB* is assumed to represent a negative sign if it is equal to one. The excess code is yet another way of representing negative numbers. In excess code, the unsigned value of a bit pattern represents the required number plus a known excess value (sometimes called bias). In the table, the bias equals eight in the first case, seven in the second, and three in the third. Although four bits provide 16 distinct representations, the use of sign-magnitude or ones complement leads to only 15 distinct numbers since there are two equivalent representations of zero in each of those two codes. The two's complement and the excess coding allow the representation of 16 different numbers. However, the range of representable numbers may be changed according to the implicitly assumed bias. Other ways of encoding numbers are possible and we will see more of these as we progress in the book.

Table 1.1: Some binary coding schemes

Pattern	Unsigned	S-M	1s	2's	excess-8	excess-7	excess-3
1111	15	−7	−0	−1	7	8	12
1110	14	−6	−1	−2	6	7	11
1101	13	−5	−2	−3	5	6	10
1100	12	−4	−3	−4	4	5	9
1011	11	−3	−4	−5	3	4	8
1010	10	−2	−5	−6	2	3	7
1001	9	−1	−6	−7	1	2	6
1000	8	−0	−7	−8	0	1	5
0111	7	7	7	7	−1	0	4
0110	6	6	6	6	−2	−1	3
0101	5	5	5	5	−3	−2	2
0100	4	4	4	4	−4	−3	1
0011	3	3	3	3	−5	−4	0
0010	2	2	2	2	−6	−5	−1
0001	1	1	1	1	−7	−6	−2
0000	0	0	0	0	−8	−7	−3

In complement coding, the bit pattern range is divided in two halves with the upper half (i.e. where the $MSB = 1$) representing negative values. In excess codes, on the other hand, the bit patterns that look smaller (i.e. their unsigned value is smaller) are in fact less than those that look larger.

Example 1.8 The mix between the various coding schemes is a common bug for beginners in programing. For example, the following C code shows what we might get if we 'look' at the same bit pattern as unsigned versus if we look at it as signed within the context of a program using 32 bits to represent integers.

```
#include<stdio.h>

int main(void)
        {
        int x=2000000000;
        int y=2000000000;

        printf("x = %d,    y = %d\n",x,y);
        printf("(unsigned) x+y = %u\n",x+y);
        printf("(   signed) x+y = %d\n",x+y);
        }
```

Once we compile and run this code the result is:

```
x = 2000000000,    y = 2000000000
(unsigned) x+y = 4000000000
(   signed) x+y = -294967296
```

The sum of `x` and `y` in example 1.8 has $MSB = 1$ which indicates a negative number if the programmer is not careful to ask for an unsigned interpretation. In this example, two numbers in the 'lower half' of the range of two's complement representation were summed together. Their sum is in fact a number beyond the lower half and lies within the upper half of the range. If we interpret that sum as a two's complement representation we get the strange result. This is an instance of 'overflow' as we will see in the following section.

1.4 Implementation of Integer Operations

For each integer data representation, five operations will be analyzed: addition, subtraction, shifting, multiplication, and division. Most of the discussion assumes binary arithmetic (radix 2).

Addition and subtraction are treated together, since the subtraction is the same as addition of two numbers of opposite signs. Thus, subtraction is performed by adding the negative of the subtrahend to the minuend. Therefore, the first thing to be addressed is the negation operation in each data representation.

1.4.1 Negation

In a ones' complement system, negation is a simple Boolean *NOT* operation. Negation in a two's complement (**TC**) system can be viewed as

$$\mathbf{TC}(N) = 2^n - N = (2^n - 1 - N) + (1),$$

where n is the number of digits in the representation. It may look awkward in the equation, but in practice this form is easier to implement, since the first term is the simple ones' complement (i.e., *NOT* operation) and the second term calls for adding one to the least significant bit (*LSB*).

Although we have just described radix complement and diminished radix complement in general terms, it is instructive to re-iterate some of the issues for the special case of a binary radix. For this purpose, we follow the discussion of ones' and two's complement operations provided by Stone [Stone, 1975].

1.4.2 Two's Complement Addition

Two's complement addition is performed as if the two numbers were unsigned numbers; that is, no correction is required. However, it is necessary to determine when an overflow occurs. For two summands P and N, there are four cases to consider:

Case	P	N	Comments
1	Positive	Positive	
2	Negative	Negative	
3	Positive	Negative	$\|P\| < \|N\|$
4	Positive	Negative	$\|P\| > \|N\|$

For positive numbers, the sign bit (the *MSB*) is zero, and for negative numbers, the sign bit is one. The sign bit is added just like all the other bits. Thus, the sign bit of the final result is made up of the sum of the summands' sign bits plus the carry into the sign bit.

In the first case, the sum of the sign bits is zero ($0 + 0 = 0$), and if no carry is generated by the remaining lower order bits, the resultant sign bit is zero. No overflow occurs under this condition. On the other hand, if a carry is generated by the remaining lower order bits, the binary representation of the result does not fit in the number of bits allocated to the summands and the resultant sign bit becomes one. That is, adding two positive summands generates a result surpassing the boundary which separates the negative and positive numbers. The result is falsely interpreted as being negative. An overflow must be signaled under this condition.

The rest of the cases are analyzed in a similar fashion and summarized in the following table:

Case	P	N	Sum of Signs	Carry-in to Sign Bit (C_{n-1})	Carry-out of Sign Bit (C_n)	Overflow	Notes
1a	Pos	Pos	0	0	0	no	
1b	Pos	Pos	0	1	0	yes	
2a	Neg	Neg	0	1	1	no	
2b	Neg	Neg	0	0	1	yes	
3	Pos	Neg	1	0	0	no	$\|P\| < \|N\|$
4	Pos	Neg	1	1	1	no	$\|P\| > \|N\|$

Two observations can be made from the above table:

1. It is impossible to overflow the result when the two summands have different signs (this is quite clear intuitively).
2. The overflow condition can be stated in terms of the carries in and out of the sign bit—that is, overflow occurs when these carries are different.

Using $\oplus$ for the exclusive *OR*, the *XOR*, operation, the Boolean expression for the overflow is thus:

$$\text{OVERFLOW } = C_{n-1} \oplus C_n.$$

1.4.3 Ones Complement Addition

It was mentioned earlier that addition in ones' complement representation requires correction. Another way of looking at the reason for correction is to analyze the four cases as was done for the two's complement addition (for simplicity, the overflow cases are ignored).

Case 1, both P and N are positive: Same as two's complement addition, and no correction is required.

Case 2, both P and N are negative: Remeber that we are using the diminished radix complement ($\mathbf{DRC}(|x|) = 2^n - 1 - |x|$). We want to get $\mathbf{DRC}(|P| + |N|)$ by adding $\mathbf{DRC}(|P|) +$

DRC$(|N|)$. However,

$$\begin{array}{rrcl} & 2^n-1 & - & |P| \\ + & 2^n-1 & - & |N| \\ \hline & 2^{n+1}-2 & - & (|P|+|N|) \end{array}$$

which is not **DRC**$(|P|+|N|)$. The resulting sum is, in fact, larger than what is possible to represent in n bits and a carry-out of the sign bit occurs. In modulo 2^n, the number 2^{n+1} is represented by its congruent 2^n. Thus, the sum is $2^n - 2 - (|P| + |N|)$, whereas we wanted the ones' complement format $2^n - 1 - (|P| + |N|)$. Therefore, 1 must be added to the *LSB* to have the correct result.

Case 3, P is positive, N is negative, and $|P| < |N|$:

$$\begin{array}{rrcl} & & & |P| \\ + & 2^n-1 & - & |N| \\ \hline & 2^n-1 & - & (|N|-|P|) \end{array}$$

This form requires no correction since it gives a result representable in n bits with the correct value.

Case 4, P is positive, N is negative, and $|P| > |N|$:

$$\begin{array}{rrcl} & & & |P| \\ + & 2^n-1 & - & |N| \\ \hline & 2^n-1 & + & (|P|-|N|) \end{array}$$

Since $|P| > |N|$, this result is positive and in the modulo 2^n system a carry-out of the sign bit is generated leaving a result congruent to $-1 + (|P| - |N|)$. Hence, a correction is required.

The implementation of the correction term is relatively easy. In both cases when a correction is necessary there is a carry-out of the sign bit. In the other two cases, this carry-out is zero. Thus, in hardware, the carry-out of the sign bit is added to the *LSB* (if no correction is required, zero is added to the *LSB*). The correction term is the end-around carry, and it causes ones' complement addition to be slower than two's complement addition.

Overflow detection in one's complement addition is the same as in two's complement addition; that is, OVERFLOW $= C_{n-1} \oplus C_n$.

1.4.4 Computing Through the Overflows

This subject is covered in detail by Garner [Garner, 1978]. Here, we just state the main property. In complement-coded arithmetic, it is possible to perform a chain of additions, subtractions, multiplications, or any combination that will generate a final correct (representable) result, even though some of the intermediate results have overflowed.

Example 1.9 In 4-bit two's complement representation, where the range of representable numbers is -8 to $+7$, consider the following operation:

$$+5+4-6=+3. \qquad \begin{array}{ll} 0101 & +5 \\ 0100 & +4 \\ \hline 1001 & \text{Overflow} \\ 1010 & -6 \\ \hline 0011 & +3 \text{ (correct)} \end{array}$$

The important condition allowing us to neglect the intermediate overflows is that the final result is bound to a representable value. In fact, the computation through the overflow follows from the properties of modular arithmetic:

$$\begin{aligned} &(A\,[+,-,\times]\,B\,[+,-,\times]\,C\,[+,-,\times]\,D)\mathbf{mod}_\mu \\ &\quad = ((A\,[+,-,\times]\,B\,[+,-,\times]\,C)\mathbf{mod}_\mu\,[+,-,\times]\,D\mathbf{mod}_\mu)\,\mathbf{mod}_\mu \\ &\quad = (((A\,[+,-,\times]\,B)\mathbf{mod}_\mu\,[+,-,\times]\,C\mathbf{mod}_\mu)\,\mathbf{mod}_\mu\,[+,-,\times]\,D\mathbf{mod}_\mu)\,\mathbf{mod}_\mu \end{aligned}$$

If the final result is representable then the intermediate results may be computed $\mathbf{mod}_\mu$ without affecting the correctness of the operations.

1.4.5 Arithmetic Shifts

The arithmetic shifts are discussed as an introduction to the multiplication and division operations. An arithmetic left shift is equivalent to multiplying by the radix (assuming the shifted result does not overflow), and an arithmetic right shift is equivalent to dividing by the radix. In binary, shifting p places is equivalent to multiplying or dividing by 2^p. In left shifts (multiplying), zeros are shifted into the least significant bits, and in right shifts, the sign bit is shifted into the most significant bit (since the quotient will have the same sign as the dividend).

Example 1.10 Perform both a left and a right shift by three bits on the binary number $0001\,0110 = 22$ and check the results.
Solution: After a left shift by two we get $1011\,0000$. If the numbers are unsigned then $1011\,0000 = 176$ which is the expected result ($= 2^3 \times 22$). However, if the numbers are signed numbers in two's complement format then the operation has caused an overflow. For the case of a right shift, we get $0000\,0010 = 2$ which is the integer part of $22/(2^3)$.

The difference between a logical and an arithmetic shift is important to note. In a logical shift, *all bits* of a word are shifted right or left by the indicated amount with zeros filling unreplaced end bits. In an arithmetic shift, the sign bit is fixed and the sign convention must be observed when filling unreplaced end bits. Thus, a right shift (divide) of a number will fix the sign bit and fill the higher order unreplaced bits with either ones or zeros in accordance with the sign bit. With arithmetic left shift, the lower order bits are filled with zeros regardless of the sign bit.

As illustrated by the previous example, so long as a p place left shift does not cause an overflow—i.e., 2^p times the original value is less than or equal to the maximum representable number in the word—arithmetic left shift is the same as logical left shift.

Example 1.11 What is the effect of a three bit arithmetic left and right shift on $1111\,0111 = -9$ and $1110\,0101 = -27$? What if the shifts are logical?
Solution: The results of the four kinds of shifting are

	Arith. Left $\times 2^3$	Arith. Right $\div 2^3$	Logic Left Shift by 3	Logic Right Shift by 3
-9	$1011\,1000 = -72$	$1111\,1110 = -2$	$1011\,1000 = -72$	$0001\,1110 = +30$
-27	$1010\,1000 = -88$	$1111\,1100 = -4$	$0010\,1000 = +40$	$0001\,1100 = +28$

It is quite clear that logical right shifts produce wrong results for two's complement negative numbers. Only the arithmetic right shifts are useful if we are implementing complement coding.
For left shifts, both arithmetic and logical shifts produce the same results for -9 since no overflow occurs. However, the difference is evident in the case of -27.

We notice something from this last example: in two's complement arithmetic right shift, there is an asymmetry between the shifted results of positive and negative numbers:

$$-13 = 1\,0011 \xrightarrow{\text{1 bit right shift}} 1\,1001 = -7;$$
$$+13 = 0\,1101 \xrightarrow{\text{1 bit right shift}} 0\,0110 = +6.$$

This, of course, relates to the asymmetry of the two's complement data representation, where the quantity of negative numbers is larger by one than the quantity of positive numbers.

By contrast, the ones' complement right shift is symmetrical:

$$-13 = 1\,0010 \xrightarrow{\text{1 bit right shift}} 1\,1001 = -6;$$
$$+13 = 0\,1101 \xrightarrow{\text{1 bit right shift}} 0\,0110 = +6.$$

Notice that the asymmetric resultant quotients correspond to modular division—i.e., creating a quotient so that the remainder is always positive. Similarly, symmetric quotients correspond to signed division—the remainder assumes the sign of the dividend.

1.4.6 Multiplication

In unsigned data representation, multiplying two operands, one with n bits and the other with m bits, requires that the result will be $n + m$ bits. If each of the two operands is n bits, then the product has to be $2n$ bits. This, of course, corresponds to the common notion that the multiplication product is a double-length operand.

$\overset{?}{\Longrightarrow}$ **Exercise 1.10 Prove that $2n$ bits are necessary to correctly represent the product P of two unsigned n bits operands.**

In signed numbers, where the *MSB* of each of the operands is a sign bit, the product should require only $2n-1$ bits, since the product has only one sign bit. However, in the two's complement code there is one exceptional case: multiplying -2^n by -2^n results in $+2^{2n}$. But this positive number is not representable in $2n - 1$ bits. This latter case is often treated as an overflow, especially in fractional representation when both operand and results are restricted to the range $[-1, +1[$. Thus, multiplying -1×-1 gives the unrepresentable $+1$.

1.4.7 Division

Division is the most difficult operation of the four basic arithmetic operations. Two properties of the division are the source for this difficulty:

1. Overflow—Even when the dividend is n bits long and the divisor is n bits long, an overflow may occur. A special case is a zero divisor.
2. Inaccurate results—In most cases, dividing two numbers gives a quotient that is an approximation to the actual rational number.

In general, one would like to think of division as the converse operation to multiplication but, by definition:

$$\begin{aligned} \frac{a}{b} &= q + \frac{r}{b}, \\ a &= b \times q + r, \end{aligned}$$

where a is the dividend, b is the divisor, q is the quotient, and r is the remainder. In the subset of cases when $r = 0$, the division is the exact converse of multiplication.

In terms of the natural integers (Peano's numbers), all multiplication results are still integers, but only a small subset of the division results are such numbers. The rest of the results are rational numbers, and to represent them accurately a pair of integers is required.

In terms of machine division, the result must be expressed by one finite number. Going back to the definition of division,

$$\frac{a}{b} = q + \frac{r}{b},$$

we observe that the same equation holds true for any desired finite precision.

Example 1.12 In decimal arithmetic, if $a = 1$, $b = 7$, then $1/7$ is computed as follows:

$$\begin{array}{llll} a/b = q + r/b & \text{or} & a = \ \ b \times q + r & \\ 1/7 = 0.1 + 0.3/7 & \text{or} & 1 = 0.7 + 0.3 & q = 0.1 \\ 1/7 = 0.14 + 0.02/7 & \text{or} & 1 = 0.98 + 0.02 & q = 0.14 \\ 1/7 = 0.142 + 0.006/7 & \text{or} & 1 = 0.994 + 0.006 & q = 0.142 \\ 1/7 = 0.1428 + 0.0004/7 & \text{or} & 1 = 0.9996 + 0.0004 & q = 0.1428 \end{array}$$

The multiplicity of valid results is a difficulty in division. This multiplicity depends on the sign conventions, e.g., signed versus modular division. Recall that $-7 \div_m 3 = -3$ while $-7 \div_s 3 = -2$. Thus, if the hardware provides modular division using two's complement code and one wishes a signed division, a negative quotient requires a correction by adding one to the least significant bit.

Multiplication can be thought of as successive additions, and division is similarly successive subtractions. However, in multiplication it is known how many times to add, in division the quotient digits are not known in advance. It is not absolutely certain how many times it will be necessary to subtract the divisor from a given order of the dividend.

Example 1.13 If we divide 01111 by 00100 through successive subtractions we get:

Iteration	Remainder	Is the remainder negative?
1	$01111 - 00100 = 01011$	no
2	$01011 - 00100 = 00111$	no
3	$00111 - 00100 = 00011$	no
4	$00011 - 00100 = 11111$	yes $\Rightarrow$ Stop

which means that the result is 3 in decimal or 00011 in binary.

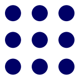
As the example shows, in these algorithms, which are trial and error processes, it is not known that the divisor has been subtracted a sufficient number of times until it has been subtracted once too often. In implementing a simple subtractive division algorithm, the lack of knowledge regarding the number of subtractions to perform in the division becomes evident. Several subtractive techniques exist with varying complexities and time delays.

The difficulties encountered in performing division as a trial and error shift and subtract process are eliminated when we choose a different a approach to the implementation. The division of a/b is equivalent to the multiplication of a by the reciprocal of b, $(1/b)$. Thus, the problem is reduced to the computation of a reciprocal, which is discussed in chapter 6 on division algorithms.

1.5 Going far and beyond

After you have learned about the basic fundamentals with integers, it is now time to stretch that framework. To start, try to relax a bit and solve the following exercise.

$\overset{?}{\Longrightarrow}$ **Exercise 1.11 Given the following nine dots, connect them using only four straight lines *without lifting the pen off the paper.***



(Hint: remember to go far and beyond!)

1.5.1 Fractions

In our discussion of the weighted positional number system so far we used the formula $N = \sum_{i=0}^{i=n-1} d_i\beta^i$ for integers. Obviously, there is a need to extend integers to represent fractions as well. Such an extension is quite simple if we write $N = \sum_{i=l}^{i=n-1} d_i\beta^i$ where $l \leq 0$.

Example 1.14 Using $N = \sum_{i=l}^{i=n-1} d_i\beta^i$ with $l = -3, n = 6, \beta = 2$, and $d_i \in \{0, 1\}$, represent $\frac{240}{16}$. What if $l = -4$?
Solution: The infinite precision result of $\frac{241}{16} = 1111.0001$ which is represented as 001111000 when $l = -3$. The fractional point is implicit. This representation has six $(= n)$ integer bits and three $(= l)$ fractional bits. The precision is not enough to represent the result accurately.
When $l = -4$, the system has enough precision and the representation is 0011110001.

Table 1.2: A 4 bits negabinary system.

-8	+4	-2	+1	Value	-8	+4	-2	+1	Value
0	0	0	0	0	1	0	0	0	-8
0	0	0	1	+1	1	0	0	1	-7
0	0	1	0	-2	1	0	1	0	-10
0	0	1	1	-1	1	0	1	1	-9
0	1	0	0	+4	1	1	0	0	-4
0	1	0	1	+5	1	1	0	1	-3
0	1	1	0	+2	1	1	1	0	-6
0	1	1	1	+3	1	1	1	1	-5

Such a representation has a fixed number of fractional and integer bits. Hence, we call the numbers represented in this manner *fixed point* numbers. The example hinted to a limitation of fixed point representation, it is not possible to represent both very small and very large numbers using a fixed point system unless a very large number of bits exist at both sides of the implicit fractional point. Such a large number of bits is not easy in practical implementation. It is this specific need to represent a wide range of numbers that gave rise to another representation: *floating point* numbers which we discuss in detail in chapter 5.

1.5.2 Is the radix a natural number?

Let us concentrate once more on $N = \sum_{i=l}^{i=n-1} d_i \beta^i$ to eliminate two more implicit assumptions that were made earlier. These are that β is a positive integer and that $0 \leq d_i < \beta$. Neither of these two conditions is necessary and, in fact, systems have been proposed and built using bases and digits that violate those assumptions. We introduce the basic concepts here and develop them in later chapters.

The negabinary system [Oberman, 1979] uses $\beta = -2$ and represents both positive and negative numbers without a need for complement coding as shown in Table 1.2 for a four bits system. This 4 bits system represents numbers in the range $\{-10, -9, \cdots, 0, 1, \cdots, 5\}$. It has a unique representation for zero. However, the system is not balanced in the sense that the number of negative numbers is twice that of positive numbers. Hence, in practical application, the numbers $\{-10, -9, -8, -7, -6\}$ cannot be used because their complements with the same modulus are missing. In fact, complementation in this system is more complicated that the case of two's and ones' complement. If the number of bits is odd, the system will represent more positive numbers than negative numbers.

For a large number of applications, there is a need to support complex numbers. Usually, the support is done in software by grouping a pair of numbers where one represents the real part while the other represents the imaginary part. The hardware in such cases simply supports the real numbers only and the software layer gives the applications the illusions of complex number support.

Despite being slightly complicated, complex number support directly in the hardware is also possible. Because such direct support is quite rare in practice, we will continue to assume that β is real in the remainder of our discussion. However, to illustrate the possibility and as a challenge, try to solve the following exercise completely before looking at the solution.

$\overset{?}{\Longrightarrow}$ **Exercise 1.12 Your new company wants to build some hardware to directly support complex numbers. The arithmetic group decided to use a weighted positional number system with the radix $\beta = -1 + j$ (where $j = \sqrt{-1}$) and the digit set $d_i \in \{0, 1\}$ for any bit location i.**

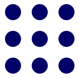
(a) **What does the bit patterns $0\,1011\,0011$ and $1\,1101\,0001$ represent in this system?**

(b) **Find a procedure to determine if a bit pattern represents a real number (i.e. that the imaginary part is zero).**

(c) **Show that the system is capable of representing any positive or negative integer number.**

You might think that after dealing with complex numbers, we have gone quite *far and beyond* the original system of integers that we defined earlier. Well, we did but not maybe far enough in all directions! Try to solve the following challenge.

$\overset{?}{\Longrightarrow}$ **Exercise 1.13 In exercise 1.11 you managed to connect the nine dots using four lines. Now try to connect them using three lines only without lifting the pen off the paper.**



1.5.3 Redundant representations

Now that we know that β is not necessarily a positive integer, the condition on the digits being $0 \leq d_i < \beta$ is easy to dismiss. Although the use of non-integer, non-positive β is theoretically possible it is infrequent in practice. On the other hand, the use of a digit set that breaks the condition $0 \leq d_i < \beta$ is quite frequent. In fact, the case when the number of digits exceeds the radix of the system is of great importance for high speed arithmetic since it provides a redundant system. Almost, all the high speed multipliers and dividers in general purpose processors in the world use redundant representations internally. Hence the study of these redundant representations has a high practical importance. Redundant systems have the possibility to perform addition without carry propagation and hence achieve higher speeds.

Example 1.15 Given $\beta = 10$ and $0 \leq d_i \leq 19$, perform the addition of 569, 783, and 245.

Solution: Due to the larger set of digits available in this system, we get

$$\begin{array}{cccc} & 5 & 6 & 9 \\ + & 7 & 8 & 3 \\ + & 2 & 4 & 5 \\ \hline & 14 & 18 & 17 \end{array}$$

which is the final result without any carry propagation! It is also important to note that it is possible to perform the addition at all the digit positions *in parallel* and not necessarily starting at the *LSD*.

An attentive reader might note that the numbers in the previous example were chosen so that the sum at any position does not exceed the largest available digit. What if we get a larger sum? Another question might be *"Is the lower bound on d_i always 0?"*

Both questions lead us to consider a more general case for a system where d_i belongs to a digit set $\mathcal{D} = \{\alpha, \alpha+1, ..., \gamma\}$. If 0 does not belong to $\mathcal{D}$, the system is not capable of representing the value of absolute zero and special measures must be taken for that. Hence, for most practical systems, $\alpha \leq 0 \leq \gamma$. For the cases described so far, we have had $\alpha = 0$ and $\gamma = |\beta| - 1$. We define a redundancy index ρ as the number of excess digits available in the system beyond the base:

$$\rho = \text{Number of digits in } \mathcal{D} - \text{Radix of the system.}$$

When $\mathcal{D}$ has all the numbers starting from α and going up to γ,

$$\rho = (\gamma - \alpha + 1) - \beta.$$

If $\rho > 0$ the system is redundant. We adopt the notation of using an over bar such as $\bar{3}$ to indicate that the value of a digit is negative.

Example 1.16 To illustrate some points, let us assume a weighted positional signed digit system with base β where the digits d_i are such that $-\beta < d_i < \beta$. Since signed digits are used, the numbers, in general, have multiple representations. Prove that there exists a special number that has only a unique representation.

Solution: The special number is zero. Any non-zero number X represented by $x_{n-1}x_{n-2}\cdots x_0$ where $-\beta < x_i < \beta$ has at least two representations. To prove this redundancy, we select any two consecutive digits $x_j x_{j-1}$ satisfying the following conditions and recode them according to the given rules:

Old values	New values
$x_{j-1} > 0$ and $x_j < (\beta - 1)$	$(x_j + 1)(x_{j-1} - \beta)$
$x_{j-1} < 0$ and $x_j > -(\beta - 1)$	$(x_j - 1)(x_{j-1} + \beta)$

Such consecutive digits must exist in any number with the exception of

1. $X = (\beta - 1)(\beta - 1)(\beta - 1)\cdots$, and
2. $X = \overline{(\beta - 1)}\ \overline{(\beta - 1)}\ \overline{(\beta - 1)}\ \cdots$.

These two cases are recoded as

$$\begin{array}{ccccc|ccccc} & \beta-1 & \beta-1 & \cdots & \beta-1 & & \overline{\beta-1} & \overline{\beta-1} & \cdots & \overline{\beta-1} \\ 1 & 0 & 0 & \cdots & -1 & -1 & 0 & 0 & \cdots & 1 \end{array}$$

To complete the proof, we must consider the case of $X = 0$ and assume that it has a representation where some of the digits are non-zero. If x_j is the most significant non-zero digit in this representation then $\sum_{i=0}^{i=j-1} \beta^i \times x_i$ should equal $-x_j \times \beta^j$ to yield $X = 0$. Since $\mid x_i \mid < \beta$ then $\mid \sum_{i=0}^{i=j-1} \beta^i \times x_i \mid \leq \sum_{i=0}^{i=j-1} \beta^i \times (\beta - 1) < \beta^j$ which means that it is impossible to cancel x_j by $\sum_{i=0}^{i=j-1} \beta^i \times x_i$. The only representation for $X = 0$ is thus a string of all zero digits.

A system is called maximally redundant [Parhami, 1990] if $\rho = \beta - 1$ and $|\alpha| = \gamma = \beta - 1$. The previous example uses such a maximally redundant system. In fact, a range of systems is defined based on the different values of the parameters. Of particular interest is the minimally redundant symmetric systems where $\rho = 1$ and $2|\alpha| = 2\gamma = \beta$ with $\beta \geq 4$. We extensively employ those minimally redundant symmetric systems in parallel multiplication (chapter 4) when a technique called Booth recoding is used. The systems with $\rho \geq \beta$ are called over-redundant.

$\overset{?}{\Longrightarrow}$ **Exercise 1.14 When do redundant systems have non unique representations of zero?**

In the case of $\beta \geq 2$ and $\rho \geq 2$, addition with a guaranteed limited carry propagation is possible if we implement the following rules[Parhami, 1990]:

1. At each position i, form the primary sum $p_i = x_i + y_i$ of the two operands x and y.
2. If $p_i \geq \gamma$ generate a carry $c_{i+1} = 1$. If $p_i \leq \alpha$ generate a carry $c_{i+1} = -1$. Otherwise, $c_{i+1} = 0$.
3. The intermediate sum at position i is $w_i = p_i - \beta c_{i+1}$.
4. The final sum at position i is $s_i = w_i + c_i$.

Example 1.17 Using $\beta = 10$ and $d_i \in \{-9, \ldots, 9\}$, apply the previous rules to $202 + 189$ and $212 + 189$.
Solution: Obviously, the results are 391 and 401 but let us see the detailed operations:

$$\begin{array}{rrrcrrr}
2 & 0 & 2 & & 2 & 1 & 2\\
+1 & 8 & 9 & & +1 & 8 & 9\\ \cline{1-3}\cline{5-7}
3 & 8 & 11 & |p_i| \geq |\gamma|? & 3 & 9 & 11\\
 & & & & & & \\
0 & 1 & & c_i & 1 & 1 & \\
3 & 8 & 1 & w_i & 3 & \bar{1} & 1\\ \cline{1-3}\cline{5-7}
3 & 9 & 1 & s_i & 4 & 0 & 1
\end{array}$$

It is quite important to note two things:

1. It is possible to do these computations from the *MSD* to the *LSD* or in any other order. If the hardware is there, all of the digit positions can be done in parallel. There is no carry propagation from the *LSD* to the *MSD*.
2. Since we recode the primary sum if it is equal to $|\gamma|$ (as in the example to the right), we are sure that any carry of value ±1 from the next lower order position is completely absorbed and never propagates to higher order digit positions.

We will further explore the use of redundancy for various reasons in practical arithmetic circuits in subsequent chapters.

1.5.4 Mixed radix systems

A last implicit assumption that we should throw away before leaving this chapter is the formula $N = \sum_{i=l}^{i=n-1} d_i \beta^i$ itself! In the way humans count time, we have 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 7 days in a week. That system does not have a

Table 1.3: Some decimal coding schemes

Pattern	8421	5421	4221	5211	6331	5221	5321	4421	2421	4311
1111	*i*	*i*	9	9	*i*	*i*	*i*	*i*	9	9
1110	*i*	*i*	8	8	*i*	9	*i*	*i*	8	8
1101	*i*	*i*	7	8	*i*	8	9	9	7	8
1100	*i*	9	6	7	9	7	8	8	6	7
1011	*i*	8	7	7	*i*	8	8	7	5	6
1010	*i*	7	6	6	9	7	7	6	4	5
1001	9	6	5	6	7	6	6	5	3	5
1000	8	5	4	5	6	5	5	4	2	4
0111	7	7	5	4	7	5	6	7	7	5
0110	6	6	4	3	6	4	5	6	6	4
0101	5	5	3	3	4	3	4	5	5	4
0100	4	4	2	2	3	2	3	4	4	3
0011	3	3	3	2	4	3	3	3	3	2
0010	2	2	2	1	3	2	2	2	2	1
0001	1	1	1	1	1	1	1	1	1	1
0000	0	0	0	0	0	0	0	0	0	0

fixed β and does not fit the formula $N = \sum_{i=l}^{i=n-1} d_i \beta^i$. It is an example of mixed radix systems where each position has a specific weight but the weights are not necessarily multiples of a single number. The elapsed time in 2 weeks, 3 days, 2 hours, 23 minutes, and 17 seconds is

Time	2 weeks	3 days	2 hours	23 minutes	17 seconds
Weights	$7 \times 24 \times 60 \times 60$	$24 \times 60 \times 60$	60×60	60	1
Value	$2 \times 7 \times 24 \times 60 \times 60 +$	$3 \times 24 \times 60 \times 60 +$	$2 \times 60 \times 60 +$	$23 \times 60 +$	$17 \times 1 =$ 1 477 397s.

In mixed radix systems, it is important to clearly specify the possible set of digit values. In the case of time, the digit values for seconds and minutes is $\in \{0, \ldots, 59\}$ while for hours it is $\in \{0, \ldots, 23\}$ or $\{1, \ldots, 12\}$.

Once we allow mixed radix systems, some interesting encodings become feasible. For example, to represent a single decimal digit using four bits we may use the conventional Binary Coded Decimal (BCD) which has the weights 8 4 2 1 or any other weights such as those presented in Table 1.3. Some bit patterns are invalid for certain codes, in those cases, they are marked with an *i* in the table.

We see that, with the exception of 8 4 2 1, all the presented codes have redundant representations: the same number is represented by multiple bit patterns. However, this form of redundancy is different from what we have just studied in section 1.5.3! Here the bits take only one of two values either 0 or 1 and do not exceed any 'radix', the redundancy comes from the positional weights.

Some combination of choices lead to incomplete codes such as the 6 3 3 1 code where there is no way to represent 2 nor 5.

Among the codes of Table 1.3, the 4 2 2 1, 5 2 1 1, 2 4 2 1, and 4 3 1 1 use all the possible sixteen combinations and do not have any invalid bit combinations. The 4 2 2 1, 5 2 1 1, and

2 4 2 1 have another interesting feature that does not exist in 4 3 1 1: the nines complement of a digit is equivalent to the ones complement of its binary representation.

Designers use the properties of these various coding schemes (and others) to their advantage in many ways when building binary circuits for decimal numbers as we will see. In fact, most digital circuits are binary. Multi-valued logic is a mature field from the theoretical point of view. However, the design of circuits implementing multi-valued logic is a much harder task that does not scale easily to large systems. Hence, multi-valued logic has a very small market share. When designers need to implement a system with $\beta \neq 2$, they usually resort to codes similar to the ones we presented.

To illustrate their use, we briefly present the addition of two numbers using BCD digits. We assume in this problem that the input digits to the adder are each 4 bits with the normal BCD coding, i.e. the digit has the same value as the corresponding conventional binary encoding.

It is important to remember that any addition result exceeding the value of nine in a digit position must produce a carry to the next higher digit position. In a regular four bits binary adder, a carry is produced when the sum exceeds sixteen. A regular binary adder produces the primary sum bits p_3, p_2, p_1, p_0, and the carry c_4

$$\begin{array}{ccccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline c_4 & p_3 & p_2 & p_1 & p_0 \end{array}$$

For example,

$$\begin{array}{rr} & 5 \\ + & 3 \\ \cline{2-2} 0 & 8 \end{array} \qquad \begin{array}{rr} & 0101 \\ + & 0011 \\ \cline{2-2} 0 & 1000 \end{array}$$

For a BCD adder, we must indicate if the sum exceeds nine and produce the correct results in BCD. The sum exceeds nine when there is a carry out of the 4 bit binary adder or if the bits of the resulting digit are of the form: 101x or 11xx as in

$$\begin{array}{rr} & 8 \\ + & 9 \\ \cline{2-2} 1 & 7 \end{array} \quad \begin{array}{rr} & 1000 \\ + & 1001 \\ \cline{2-2} 1 & 0001 \end{array} \rightarrow \; 1\,0111 \qquad \text{and} \qquad \begin{array}{rr} & 5 \\ + & 6 \\ \cline{2-2} 1 & 1 \end{array} \quad \begin{array}{rr} & 0101 \\ + & 0110 \\ \cline{2-2} 0 & 1011 \end{array} \rightarrow \; 1\,0001$$

then the decimal carry out signal is

$$c_{out} = c_4 + p_3(p_2 + p_1)$$

If a carry is produced, we must correct the value of the resulting digit. This correction compensates for the six values that are not used in the case of BCD but that exist in the case of binary. Hence, we add 0110 to the primary sum.

Another way of looking at this correction would be to subtract ten from the primary sum since we are now generating a carry. The subtraction of ten is done by the addition of its two's complement, i.e. by adding 10110 to the *whole* digit including c_4. We then take the least significant 4 bits and any carry produced is conveyed to the next higher digit position.

Whichever way, we correct the least significant bits by adding 0110. We will see more circuits for decimal adders using different coding schemes in chapter 3.

1.6 Further readings

Several good sources exist on computer arithmetic. The reader is encouraged to check Scott [1985] for a good introduction on the history of numbers. It also contains a chapter on nonconventional number systems as well as decimal arithmetic.

Oberman [1979] presents the negabinary system and shows how to implement several circuits for negabinary as well as for other signed digits systems.

Parhami [2000] presents a good exposition of redundant numbers.

1.7 Summary

In arithmetic, the representation of integers is a key problem. Machines, by their nature, have a finite set of symbols or codes upon which they can operate, as contrasted with the infinity that they are supposed to represent. This finitude defines a modular form of arithmetic widely used in computing systems. The familiar modular system, a single binary base, lends itself readily towards complement coding schemes which serve to scale negative numbers into the positive integer domain.

Complement coding reduces the subtraction operation to an addition of the negative of the subtrahend. Hence, the negation of a number is a topic of study in arithmetic systems using complement codes. Within integer numbers, arithmetic shifting is equivalent to multiplication and division depending on the shifting direction.

The basic ideas behind the weighted positional number system can be extended by using different bases and digit sets.

$\overset{?}{\Longrightarrow}$ **Exercise 1.15 The great challenge after reading this chapter is to go back to the nine dots of exercise 1.11 and connect them all using only one straight line. Once you solve this one, go to relax before starting your next exploration with the following chapter.**

1.8 Problems

Problem 1.1 Suppose in an 8 bit base 2 system $A = 01010101$ and $B = 10100110$, find $A + B$ and $A - B$ if

1. A and B are in 2's complement form
2. A and B are in 1's complement form
3. A and B are in sign magnitude form

Problem 1.2 Find an algorithm for computing $X\mathbf{mod_M}$ for known M, using only the addition, subtraction, multiplication, and comparison operations. You should not make any assumptions as to the relative size of X and M in considering this problem.

Problem 1.3 Find an algorithm for multiplication with negative numbers using an unsigned multiplication operation. Show either that nothing need be done, or describe in detail what must be done to the product of the two numbers, one or both of which may be in complement form.

1. For radix complement.
2. For diminished radix complement.

Problem 1.4 A number system has $\beta = 10$ as a radix and the digits are chosen from the set $\mathcal{D}\{0, 1, 20, 21, 40, 41, 60, 61, 80, 81\}$.

1. Is this system redundant?
2. Represent the integers 0 through 99 in this system. (You will need at most three digit positions to represent any of these integers.)

Problem 1.5 In a signed digit system with five digits $\beta = 10$ and the digit set is $\{\bar{7}, \bar{6}, \cdots, 6, 7\}$

1. Convert each of the following conventional decimal numbers into the signed digit representation: 9898, 2921, 5770, -0045.
2. In SD find the sum of the four numbers.
3. Convert the result back from SD to conventional form to confirm the correctness of the calculation.

Problem 1.6 In order to subtract an unsigned positive binary number B from another unsigned positive binary number A (each n bits long), a designer complements each bit of B and uses a normal binary adder to add A to the complemented version of B with an additional carry-in of one.

1. Prove that the absence of a carry out signal (i.e. carry out equal to zero) indicates a negative result.

2. In what format (unsigned, ones complement, two's complement) is the result?

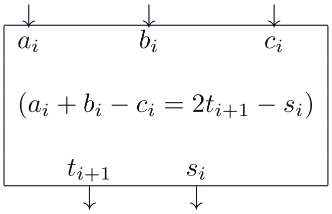
Problem 1.7 In a certain communication standard, the only available digits in a binary system are -1 and 1.

1. Is this a redundant or non-redundant number system? Why?
2. Prove that the least significant bit, *LSB*, of the sum of an even number of digits (such as in $1 + \bar{1} + \bar{1} + 1 + 1 + 1 = 10$) is always equal to zero.
3. Prove that the sum of a number N (even or odd) of digits in this system can be determined by counting only either the positive or the negative digits but not necessarily both.
4. Can the sum be represented in the same system (same digits and radix) as the original input digits? If yes how and if no why?

Problem 1.8 A full adder circuit takes two inputs a_i and b_i as well as a carry-in signal c_i to produce two outputs: the carry-out signal t_{i+1} and the sum bit s_i. The mathematical relation between these quantities is $a_i + b_i + c_i = 2t_{i+1} + s_i$ (the $+$ sign stands for addition here) which yields the logical equations $t_{i+1} = a_i b_i \vee a_i c_i \vee b_i c_i$ (the $\vee$ sign stands for a logical OR here) and $s_i = a_i \oplus b_i \oplus c_i$.

1. We want to design a similar circuit that has the mathematical relation $a_i + b_i - c_i = 2t_{i+1} - s_i$ (the $+$ sign stands for addition and the $-$ sign stands for subtraction). Write the corresponding truth table and logical equations to get the output bits.

   Assume that you have two unsigned numbers X and Y represented in regular binary by their bits ($X = x_{n-1} \ldots x_i x_{i-1} \ldots\ x_1 x_0$ and $Y = y_{n-1}\ \ldots\ y_i y_{i-1}\ \ldots\ y_1 y_0$). Use a number of blocks from the circuit that you have just designed

   to get a subtractor giving the result $R = X - Y$. State how many cells you are using. Remember to set any unused inputs and to indicate how many bits are in the result R.
2. In the two's complement format a number W of n bits has the value $W = -w_{n-1}2^{n-1} + \sum_{i=0}^{n-2} w_i 2^i$. What is the corresponding equation giving the value of R? Why?

Problem 1.9 A row of full adders is used to sum three binary vectors represented in two's complement form. If two vectors are non-negative (positive or zero) and the third vector is strictly negative, prove that the result is two vectors having opposite signs.

Problem 1.10 One way to represent decimal digits from 0 to 9 on computers is to use the excess-3 encoding. In this code, each digit is represented by a four bits binary number equal to that digit plus three. The four bit combinations corresponding to a digit in the range from 0 to 9 are called valid. For example, 0111 represents 4 while 1100 represents 9 and both of these four bit combinations is thus valid. The combinations that do not correspond to a digit in that range are called invalid.

1. Give a table with all the possible combinations of four bits indicating the corresponding digit if it is valid or labeling it as invalid.
2. How do we get the nines complement of a number in excess-3?
3. Design a circuit that takes two inputs A and B each consisting of four bits representing a decimal digit in excess-3 encoding as well as a carry-in signal c (a single bit that is either 1 or 0) to produce two outputs: the carry-out signal t (a single bit) and the sum digit S (four bits in excess-3 encoding). The mathematical relation between these quantities is $A + B + c = 10t + S$ (the + sign stands for addition here). You can use full adders and any other simple logic gates (AND, OR, NOT, NAND, NOR, XOR, and XNOR) as your building blocks. If you need, you can also use multiplexers. Remember to check that your circuit produces the correct result even when the carry-out signal is set to one.

Problem 1.11 Fast parallel decimal multipliers Vazquez et al. [2007] present another skillful use of the codes presented in Table 1.3. The redundant codes 4 2 2 1 and 5 2 1 1 are used together in the same design. The assumption that we should stick to a single number system in a design is yet another assumption that we should shed away in this chapter.

1. A number is encoded in 4 2 2 1 and we want to recode it to 5 2 1 1. Is there a unique way of making the transformation? Why?
2. Give the logical equations of a block that takes a number in 4 2 2 1 and recodes it in 5 2 1 1.
3. Prove that if a multi-digit number X where each digit is encoded in 5 2 1 1 and we form another number Y by shifting X one bit position to the left then interpret Y as being encoded in 4 2 2 1 then $Y = 2X$. (Y is obviously one digit larger than X. Assume that zero is shifted into the *LSB* and that three extra zeros are padded to fill the *MSD* of Y.)

Problem 1.12 In a certain design three types of bits are used. Posibits (p) are regular conventional bits where the mathematical value $m_p \in \{0, 1\}$. The mathematical value of negabits (n) on the other hand are $m_n \in \{-1, 0\}$ and that of unibits (u) are $m_u \in \{-1, 1\}$. The bits are represented by the logical values $\ell_p, \ell_n, \ell_u \in \{0, 1\}$ to correspond with their mathematical values as follows.

Bit type	p		n		u	
Logial value ℓ	0	1	0	1	0	1
Mathematical value m	0	1	-1	0	-1	1

1. It is obvious that $m_p = \ell_p$. Give the mathematical relation between m_n and ℓ_n as well as between m_u and ℓ_u.
2. If we use a single type of bits in a number (i.e. all $nn \ldots nn$ or all $uu \ldots uu$ for example) where the radix is $\beta = 2$ will that be a redundant number system? Clearly indicate the case for all three types of bits.

3. The regular full adder for posibits takes three bits a, b, and c and produces two bits their sum s and the carry transfered to the higher weight t such that $2t + s = a + b + c$. Prove that the same full adder structure produces the correct sum and transfer bits if the inputs are three negabits (the t and s in that case are negabits as well). Repeat for three unibits.

4. We form a radix-16 digit using a combination of those different bits arranged in powers of 2 as

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ n & p & p & p \\ & & & u \end{array}$$

   where a posibit and a unibit have the same mathematical weight of 2^0 and a negabit has the weight 2^3. What is the range of values that such a digit can take given the mathematical values of the different bits and their weights? Is the system using such radix-16 digits a redundant or non-redundant number system?

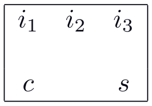
Problem 1.13 In this problem we will try to add more than just two numbers.

1. If we add seven input bits $i_1, i_2, \cdots, i_7$ (all of the same mathematical weight), how many output bits do we get?

2. Assume that you have a full adder circuit taking three inputs i_1, i_2, and i_3 and producing two outputs s and c such that

$$\begin{aligned} s &= i_1 \oplus i_2 \oplus i_3 \\ c &= i_1 i_2 + i_1 i_3 + i_2 i_3 \end{aligned}$$

   Draw a diagram connecting a number of full adders only (no other logic) to add seven inputs. Please use the following symbol for the full adder:



3. If the time taken by each full adder is 2 gate delays, what is the delay of your design?

Problem 1.14 For your new decimal multiplier that uses BCD digits (bit weights 8 4 2 1), you are tasked to design a direct generator for the $2X$ and $5X$ multiples such that the output uses the same digit encoding (8 4 2 1).

1. Give the truth table of the result (representation of the resulting digit value and the carry into the higher position value) from the different bit combinations of a single digit for each of $2X$ and $5X$.

2. Deduce the logic equations for the output bits in each of $2X$ and $5X$.

3. Estimate the number of gate delays required to implement each multiple if X has 16 digits.

Problem 1.15 Assume that you want to design registers to hold the numbers in your new computer. The cost C of the register is proportional to the number of digits n and the base of the system β such that $C = k \times n \times \beta$ where k is a constant. You want the number in a register to represent up to M different values.

1. Assuming that there is no constraints on β find the radix value that minimizes the cost of a register.
2. If β is constrained to positive integer values only what is the best choice(s) for it? Why?
3. Comment on the most practical (given the constraints of real circuits) choices.

Problem 1.16 Given two numbers in sign magnitude notation

$$A = a_{n-1}a_{n-2}\cdots a_0 \quad (1.1)$$
$$B = b_{n-1}b_{n-2}\cdots b_0 \quad (1.2)$$

in radix β and the conversions

$$C = a_{n-1}a^*_{n-2}\cdots a^*_0 \quad (1.3)$$
$$D = b_{n-1}b^*_{n-2}\cdots b^*_0 \quad (1.4)$$

where for all $0 \le i \le n-2$

$$x^*_i = \begin{cases} x_i & \text{if } x_{n-1} = 0 \\ \beta - 1 - x_i & \text{if } x_{n-1} = \beta - 1 \end{cases} \quad (1.5)$$

1. In what representation are the two numbers C and D?
2. The following sequence of operations is done

$$\begin{array}{rr} & a_{n-1}a^*_{n-2}\cdots a^*_0 \\ + & b_{n-1}b^*_{n-2}\cdots b^*_0 \\ \hline & e_n e_{n-1}e_{n-2}\cdots e_0 \end{array}$$

then we add the carry out

$$\begin{array}{rr} & e_{n-1}e_{n-2}\cdots e_0 \\ + & e_n \\ \hline & f_{n-1}f_{n-2}\cdots f_0 \end{array}$$

and finally we convert to

$$G = f_{n-1}f^*_{n-2}\cdots f^*_0 \quad (1.6)$$

where, as above,

$$f^*_i = \begin{cases} f_i & \text{if } f_{n-1} = 0 \\ \beta - 1 - f_i & \text{if } f_{n-1} = \beta - 1 \end{cases} \quad (1.7)$$

In what representation is G?

3. Prove that $G = A + B$.

Problem 1.17 A friend suggests the use of the biquinary encoding for decimal digits in your new design. In this encoding, each decimal digit has two parts— the binary part with two bits and the quinary part with five bits. The two bits in the binary part signify whether the decimal digit is from 0 to 4 or whether it is from 5 to 9. The bits of this binary part may be considered as having the weights of 5 and 0. Those two bits of the binary part can never be both 0 or both 1. The five bits of the quinary part have the weights 4, 3, 2, 1, and 0. Similarly, only one bit of those five may be 1 and they cannot be all zeros.

1. Complete the following table with the codes corresponding to the decimal digits.

Digit	50	43210
0	01	00001
1		
2		
3		
4		
5		
6		
7	10	00100
8		
9		

2. Design an adder that takes two biquinary encoded digits and produces their sum encoded in biquinary and a possible carry to the next higher position. Assume that the bits of the two digits are $x_h x_\ell x_4 x_3 x_2 x_1 x_0$ and $y_h y_\ell y_4 y_3 y_2 y_1 y_0$ (x_h has weight 5, x_ℓ has weight 0, the other bits have the weight indicated by their index).

3. Your friend claims that the property of having only one bit high in the quinary part and only one bit high in the binary part helps to easily detect any errors in the calculations using this encoding. Provide a logic circuit to check that a biquinary digit has one and only one bit high in the quinary part as well as one and only one bit high in the binary part.

Problem 1.18 Consider a signed digit redundant number $X = d_{n-1} \cdots d_1 d_0 d_{-1} \cdots d_{-m} = \sum_{i=-m}^{i=n-1} d_i \times \beta^i$ with a positive integer radix β and digits $d_i \in \{-\alpha, -\alpha+1, \ldots, 0, \ldots, \alpha-1, \alpha\}$ where $\left\lfloor \frac{\beta}{2} \right\rfloor + 1 \leq \alpha \leq \beta - 1$. Prove that

1. the sign of X is the sign of the most significant non-zero digit

2. one representation of $-X$ is $s_{n-1} \cdots s_1 s_0 s_{-1} \cdots s_{-m}$ where $s_i = -d_i$

Chapter 2

Are there any limits?

In the early days of automated calculators, the parts involved were mechanical. The physical limitations imposed on such systems are quite different from those of computers based on transistors. However, the basic concepts of number representations presented earlier as well as the algorithms for adding, multiplying, and dividing presented later are still applicable. As the technology used changes, the designer must re-evaluate the choices made while using the older technology and see if the trade-offs still carry to the new generation of devices.

In this chapter, we are concerned about the limits imposed by the physical implementation of high performance arithmetic circuits. Usually a designer needs to know three basic parameters: the time, the gate count, and the power a system takes to fulfil the arithmetic operation. We present a few simple tools for the evaluation of those parameters. We emphasize simple tools because the designer needs to have a general overview of the possible alternatives before starting the detailed design. If the tools are not simple and fast yet maintaining a good degree of accuracy the designer will not use them. As the design nears completion, more sophisticated tools providing a higher accuracy ought to be used.

The time indicates how quickly the operation is executed and hence it has an effect on the overall digital system. One of the basic operations that exist in all synchronous microprocessors for example is the incrementation of the program counter. This incrementation occurs every clock cycle and it is impossible for the whole system to run faster if this operation is slow.

The number of logic gates indicates how large the circuit is. This translates to a certain chip area in VLSI designs which in turn translates into cost. Larger circuits are usually more costly to implement. Depending on the regularity of the design, large circuits might also be more complex. Complexity leads to a longer design time, lengthy testing procedures, and troubles in maintaining or upgrading the design later.

The power consumption represents the running cost to operate the circuit. If a circuit uses a part of the consumed power for the operation and dissipates the remaining part in the form of heat to its environment then this heat must be removed. The cooling mechanism design and its operation depends on the amount of heat and adds to the running cost of the overall system. The power consumption of a system running on batteries obviously affects the batteries life time.

The three parameters are linked. A design might use a large number of gates in parallel, hence has a large area and consume considerable power, to achieve a faster operation. Another design reuses the same piece of hardware to save on the number of gates but performs the computation serially in a longer time. A third design spreads the operation in time by clocking the circuit at a lower frequency but uses less power. Such a system might consume half of the power and take double the time of another design thus maintaining the same total energy used for the operation. From the perspective of the energy source (say the battery), the two designs appear to be equivalent. This is not necessarily true in all systems. The two designs do consume the same energy but one of them at a higher rate. If the energy source is not able to supply the higher rate then the higher power design is not feasible. In yet another case, the design might be able to compute faster and use a lower energy at the expense of a complicated scheme of clocking the circuit.

We see from these simple examples that a designer must really have a sense of the specific requirements of the overall system and how the arithmetic blocks fit in that system. Depending on the requirements, the figure of merit for a design might be the time delay (T), the gate count or area (A), the power (P), the energy (power multiplied by time), or in general a formula such as

$$\text{merit} = T^a A^b P^c$$

where the exponents a, b, and c are parameters defined according to the requirements.

Whether this or a more sophisticated formula is used, a designer must evaluate the time, the size, and the power consumption of the proposed design to compare it to other alternatives.

2.1 The logic level and the technology level

In most arithmetic systems, the speed is limited by

1. the extent to which decisions of low order numeric significance affect results of higher significance and
2. the nature of the building block that makes logic decisions.

The first problem is best illustrated by the addition operation where it is possible for a low order carry to change the most significant bits of the sum.

Example 2.1 For the sum

$$\begin{array}{r} 0101101 \\ +0010011 \\ \hline 1000000 \end{array}$$

a carry generated at the *LSB* changes all the sum bits up to the *MSB*.

The essence of this problem is the issue of sequential processing. Do we have to do things in this specific sequence or is there another way to enhance the speed by doing things in parallel for example? This exchange is, in fact, trading-off the size of the circuit to gain speed. Alternatively, a designer may choose a redundant representation and have a carry-free addition for as long as there is no need to convert to regular binary. The conversion has the time delay of the carry propagation.

Another instance of the sequentiality problem is clear in the case of floating point addition and subtraction where several steps are taken in sequence: equalization of the exponent, alignment shift, addition, detection of the leading zeros, normalization, and rounding. As we progress in the book, we will see that it is possible to reduce the number of sequential steps in floating point operations and to improve the carry propagation speed in the integer addition. However, there are bounds or limits to fast arithmetic enhancements. We explore the limits in this chapter and use them later as benchmarks for comparison purposes.

The second problem regarding the nature of the building blocks is technology dependent. Each logic device has an inherent switching speed that depends on the physics involved in the switching mechanism. With vacuum tubes in the early electronic computers, the average switching speed was quite different from the time when the integrated circuits technology used Emitter Coupled Logic (ECL) transistors or later when Complementary Metal Oxide Semiconductor (CMOS) transistors became the norm.

The technology limits the speed in other ways as well beyond the switching speed. Depending on the output signal strength of a device, we decide the maximum number of logic gates, the *fanout*, that can be driven directly by this signal. If we use an electric voltage to indicate the logic level and the inputs to the gates are not drawing in much current, it is easier to allow a larger fanout. On the other hand, if the signal indicating the logic level is an electric current or charge value, it is not as easy to share it among the inputs to subsequent gates and a special circuitry is sometimes necessary. In either case, some required logic functions might exceed the fanout limit and the signal must be buffered, i.e. we use additional gates to strengthen the signal and preserve the required speed. Such special arrangements for fanout represent one facet of the trade-off between the speed and number of gates that is technology dependent. The case of doing the floating point addition with more parallelism is a trade-off that is independent of the technology.

Fundamentally, there is no minimum amount of energy required to process the information nor to communicate it [Landauer, 1996] if the process is conducted sufficiently slowly. However, most computers attempt to process their information quickly and dissipate a considerable amount of energy. Computers are bound by the maximum allowable amount of heat that the packages of the circuits are able to dissipate.

By understanding the technology constraints, a designer is able to build efficient basic blocks.

We begin by examining ways of representing numbers, especially insofar as they can reduce the sequential effect of carries on digits of higher significance. Carry independent arithmetic is possible within some limits using redundant representations or using residue arithmetic. This residue arithmetic representation is a way of approaching a famous bound on the speed at which addition and multiplication are performed.

This bound, called Winograd's bound, determines a minimum time for arithmetic operations and is an important basis for determining the comparative value of the various implementation algorithms discussed in subsequent chapters.

For certain operations, it is possible to use a memory storage, especially a Read Only Memory (ROM), as a table to "look-up" a result or partial result. Since very dense ROM technology is now available, the last section of this chapter develops a performance model of ROM access. Unlike Winograd's work, this is not a strict bound, but rather an approximation to the retrieval time.

2.2 The Residue Number System

2.2.1 Representation

The number systems considered so far in this book are linear, positional, and weighted, in which all positions derive their weight from the same radix (base). In the binary number systems, the weights of the positions are 2^0, 2^1, 2^2, etc. In the decimal number system, the weights are $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, etc.

The residue number system [Garner, 1959, Gschwind, 1967] usually uses positional bases that are relatively prime to each other, i.e. their greatest common divisor is one. For example, the two sets $(2, 3, 5)$ and $(4, 5, 7, 9)$ satisfy this condition.

Any number is represented by its residues (least positive remainders) after dividing the number by the base. For instance, if the number 8 is divided by the base 5, the residue is 3. Hence, to convert a conventionally weighted number (X) to the residue system, we simply take the residue of X with respect to each of the positional moduli.

Example 2.2 To convert the decimal number 29 to a residue number with the bases $5, 3, 2$, we compute:

$$\begin{aligned} R_5 &= 29\mathbf{mod_5} = 4 \\ R_3 &= 29\mathbf{mod_3} = 2 \\ R_2 &= 29\mathbf{mod_2} = 1 \end{aligned}$$

and say that the decimal number 29 is represented by $[4, 2, 1]$.

Example 2.3 In a residue system with the bases $5, 3, 2$ how many unique representations exist? Develop a table giving all those representations and the corresponding number.
Solution: The number of unique representations is $2 \times 3 \times 5 = 30$. The following table lists the numbers 0 to 29 and their residues to bases 5, 3, and 2.

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

Because the bases $5, 3, 2$ are relatively prime, the residues in this example uniquely identify a number. The configuration $[2, 1, 1]$ represents the decimal number 7 just as uniquely as binary 111.

The main advantage of the residue number system is the absence of carries between columns in addition and in multiplication. This advantage is due to the properties of modular arithmetic: if $N' = N\mathbf{mod}_\mu$ and $M' = M\mathbf{mod}_\mu$, then

$$\begin{aligned}
(N+M)\mathbf{mod}_\mu &= (N'+M')\mathbf{mod}_\mu \\
(N-M)\mathbf{mod}_\mu &= (N'-M')\mathbf{mod}_\mu \\
(N\times M)\mathbf{mod}_\mu &= (N'\times M')\mathbf{mod}_\mu
\end{aligned}$$

Arithmetic is closed (done completely) within each residue position. Since the speed is determined by the largest modulus position, it is possible to perform addition and multiplication on long numbers that have many digits at the same speed as on short numbers.[1] Recall that in the conventional linear weighted number system, an operation on many digits is slower due to the carry propagation.

2.2.2 Operations in the Residue Number System

Addition and multiplication are easily carried within each base with no carries between the columns.

Example 2.4 In the $5,3,2$ residue system, perform $9+16$, $8+19$, and 7×4
Solution: We start by converting each number to its representation we then perform the operation and check the result using the table of example 2.3.

$$\begin{array}{rcc}
9 & \rightarrow & [4,0,1] \\
+16 & \rightarrow & [1,1,0] \\ \hline
\text{decimal} & & \text{residue} \\
 & & 5,3,2
\end{array}
\qquad\qquad
\begin{array}{rcc}
8 & \rightarrow & [3,2,0] \\
+19 & \rightarrow & [4,1,1] \\ \hline
\text{decimal} & & \text{residue} \\
 & & 5,3,2
\end{array}$$

Note that each column is added modulo its base, disregarding any interposition carries. As for the multiplication:

$$\begin{array}{rcr}
7 & \rightarrow & [2,1,1] \\
\times 4 & \rightarrow & \times[4,1,0] \\ \hline
28 & & [3,1,0]
\end{array}$$

Again, each column is multiplied modulo its base, disregarding any interposition carries; for example, $(2\times 4)\mathbf{mod_5} = 8\mathbf{mod_5} = 3$.

The uniqueness of representation property is the result of the famous Chinese Remainder Theorem.

Theorem 1 (Chinese Remainder) *Given a set of relatively prime moduli (m_1, m_2, …, m_i, …, m_n), then for any $X < M$, the set of residues $\{X\mathbf{mod_{m_i}} | 1 \le i \le n\}$ is unique, where*

$$M = \prod_{i=1}^{n} m_i.$$

The proof is straightforward:

[1] The redundant representations that we introduced in the first chapter achieve the same goal using a different way.

Suppose there were two numbers Y and Z that have identical residue representations; i.e., for each i, $y_i = z_i$, where

$$\begin{aligned} y_i &= Y\mathbf{mod_{m_i}} \\ z_i &= Z\mathbf{mod_{m_i}}. \end{aligned}$$

Then $Y - Z$ is a multiple of m_i, and $Y - Z$ is a multiple of the least common multiple of m_i. But since the m_i are relatively prime, their least common multiple is M. Thus, $Y - Z$ is a multiple of M, and Y and Z cannot both be less than M [Stone, 1973].

Subtraction

Since $(a\mathbf{mod_m}) - (b\mathbf{mod_m}) = (a - b)\mathbf{mod_m}$, the subtraction operation poses no problem in residue arithmetic, but the representation of negative numbers requires the use of complement coding.

Following our earlier discussion on complementation, we create a signed residue system by dividing the range and using the numbers below $M/2$ to represent positive numbers while those greater than or equal to $M/2$ represent negative numbers.

So, a negative number $-M/2 \le -Y < 0$ is represented by $X = M - Y$. Said differently, a number $X \ge M/2$ is treated as $-Y = X - M$,

$$(X - M)\mathbf{mod_M} = X\mathbf{mod_M},$$

and the complement of $X\mathbf{mod_M}$ (Y) is:

$$X^c = (M - X)\mathbf{mod_M}.$$

$\overset{?}{\Longrightarrow}$ **Exercise 2.1 In residue representation $X = [x_i]$, where $x_i = X\mathbf{mod_{m_i}}$, call the complement of X, X^c and that of x_i, $x_i^c = (m_i - x_i)\mathbf{mod_{m_i}}$. Prove that $X^c = [x_i^c]$.**

Example 2.5 In the 5,3,2 residue system, $M = 30$, integer representations 0 through 14 are positive, and 15 through 29 are negative (i.e., represent numbers -15 through -1). Calculate $(8)^c$ and $(9)^c$ as well as $8 - 9$.
Solution: The representations of 8 and 9 are

$$\begin{aligned} 8 &= [3, 2, 0], \\ 9 &= [4, 0, 1] \end{aligned}$$

So, $(8)^c = [2, 1, 0]$ i.e. $5 - 3, 3 - 2$, and $(2 - 0)\mathbf{mod_2}$ while $(9)^c = [1, 0, 1]$.

$$\begin{array}{rcccr} 8 & = & 8 & = & [3, 2, 0] \\ -9 & = & (9)^c & = & +[1, 0, 1] \\ \hline -1 & & & & [4, 2, 1] \quad = 29\text{or } \ -1 \end{array}$$

$\overset{?}{\Longrightarrow}$ **Exercise 2.2 What is the range of signed integers that can be represented in the $32, 31, 15$ residue number system? Show how to perform $(123 - 283)$ in this residue system.**

2.2.3 Selection of the Moduli

Usually, a designer of a residue system chooses the moduli so that

1. they are relatively prime to satisfy the conditions of the Chinese Remainder Theorem and provide unique representations, and
2. they minimise the largest modulus in order to get the highest speed. (They minimise the time needed for the carry propagation in the residues of the largest modulus.)

$\overset{?}{\Longrightarrow}$ **Exercise 2.3 Assume that a designer chooses as the bases $4, 3, 2$, list all the *possible* representations and the corresponding numbers. Are there some impossible combination of residues? Why?**

Beyond those main criteria, certain moduli are more attractive than others for two reasons:

1. They are efficient in their binary representation; that is, n binary bits represent approximately 2^n distinct residues. By way of contrast, the $5, 3, 2$ bases require three bits to represent the residue of base 5, two bits for the residue of base 3, and one bit for the residue of base 2 giving a total of six bits. That residue system represents 30 different numbers while a regular binary system with six bits represents $2^6 = 64$ numbers.
2. They provide straightforward computational operations using binary adder logic. From an implementation point of view, the easiest is to build adders that are modulo some power of two. However, we are not allowed to have more than one base as a power of two in order to have unique representations. Some odd moduli are easier than others, we should choose the easy ones!

Merrill [Merrill, 1964] suggested moduli of the form 2^{k_1}, $2^{k_1} - 1$, $2^{k_2} - 1$, $\ldots 2^{k_n} - 1$ ($k_1, k_2, \ldots, k_n$ are integers) as meeting the above criteria.

Note that not all numbers of the form $2^k - 1$ are relatively prime. In fact, if k is even:

$$2^k - 1 = (2^{k/2} - 1)(2^{k/2} + 1).$$

If k is an odd composite, $2^k - 1$ is also factorable. For $k = ab$, the factors of $2^k - 1$ are $(2^a - 1)$ and $\left(2^{a(b-1)} + 2^{a(b-2)} + \cdots + 2^{a(0)}\right)$, whose product is $\left(2^{ab} - 1\right) = (2^k - 1)$. For $k = p$, with p a prime, the resulting numbers may or may not be prime. These are the famous Merseene's numbers [Newman, 1956]:

$$M_p = 2^p - 1 \qquad (p \text{ a prime}).$$

Merseene asserted in 1644 that M_p is prime for:

$$p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257,$$

Table 2.1: A Partial List of Moduli of the Form 2^k and $2^k - 1$ and Their Prime Factors

Moduli	Prime Factors
3	—
7	—
15	3,5
31	—
63	3,7
127	—
255	3,5
511	7,73
1023	3,11,31
2047	23,89
4095	3,5,7,13
8191	—
2^k $(k = 1, 2, 3, 4 \ldots)$	2

and composite for all other $p < 258$. The conjecture stood for about 200 years. In 1883, Pervushin proved that M_{61} is prime. It is only in 1947 that the whole range stated by Mersenne's (p¡258) had been completely checked and it was determined that the correct list is

$$p = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107 and 127.$$

Table 2.1 lists factors for numbers of the form $2^k - 1$. Note that any 2^n will be relatively prime to any $2^k - 1$. The table is from Merrill [Merrill, 1964].

Since the addition time is limited in the residue system to the time for addition in the largest modulus, we should select moduli as close as possible to limit the size of the largest modulus. Merrill suggests the largest be of the form 2^k and the second largest of the form $2^k - 1$, k the same. The remaining moduli should avoid common factors. He cites some examples of interest:

Bits to represent	Moduli set
17	32, 31, 15, 7
25	128, 127, 63, 31
28	256, 255, 127, 31

If the Merrill moduli are chosen relatively prime, it is possible to represent "almost" as many objects as the pure binary representation. For example, in the 17-bit case, instead of 2^{17} code points, we have

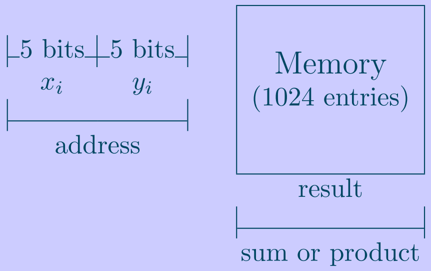
$$2^5(2^5 - 1)(2^4 - 1)(2^3 - 1) = 2^{17} - \mathcal{O}(2^{14}).$$

where $\mathcal{O}(2^{14})$ indicates a term on the order of 2^{14}. Thus, we have lost less than 1 bit of representational capability (a loss of 1 bit would correspond to an $\mathcal{O}(2^{16})$ loss).

2.2.4 Operations with General Moduli

With the increasing availability of memory cells in current integrated circuit technology, the restriction to moduli of the forms 2^k or $2^k - 1$ is less important. Thus, it is possible to perform addition, subtraction, and multiplication by table look-up. In the most straightforward implementation, separate tables are kept for each modulus, and the arguments x_i and y_i (both $\mathbf{mod}_{\mathbf{m_i}}$) are concatenated to form an address in the table that contains the proper sum or product.

Example 2.6 A table of 1024 or 2^{10} entries is used for moduli up to 32, or 2^5; i.e., if x_i and y_i are 5-bit arguments, then their concatenated 10-bit value forms an address into a table of results.



In this case, addition, subtraction, and multiplication are accomplished in one access time to the table. Note that since access time is a function of table size and since the table size grows at 2^{2n} (number of bits to represent a number), residue arithmetic has a considerable advantage over conventional representations in its use of table look-up techniques.

2.2.5 Conversion To and From Residue Representation

Conversion from an ordinary weighted number representation into a residue representation is conceptually simple—but implementations tend to be somewhat less obvious.

Conceptually, we divide the number to be converted by each of the respective moduli, and the remainders form the residues. The hardware decomposes an integer A with the value $A = \sum_{i=0}^{n} a_i \beta^i$ (where β is the radix and a_i the value at the i^{th} position) with respect to radix position, or pairs of positions, simply by the ordered configuration of the digits. In the usual case, the radix and the modular base are relatively prime and for a single position conversion we have:

$$x_{ji} = (a_i \beta^i)\mathbf{mod}_{\mathbf{m_j}},$$

where x_{ji} is the i^{th} component of the m_j residue of A, and then x_j (the residue of $A\mathbf{mod}_{\mathbf{m_j}}$) is

$$x_j = \left(\sum_i x_{ji}\right)\mathbf{mod}_{\mathbf{m_j}}.$$

$\overset{?}{\Longrightarrow}$ **Exercise 2.4 If the radix β and m_j are *not* relatively prime, can you still use the equations just mentioned? Are there any special cases with easier expressions?**

The process of conversion is easy to implement. Since $x_{ji} = (a_i\mathbf{mod}_{\mathbf{m_j}}\beta^i\mathbf{mod}_{\mathbf{m_j}})\mathbf{mod}_{\mathbf{m_j}}$, the $\beta^i\mathbf{mod}_{\mathbf{m_j}}$ term is precomputed and included in a table that maps a_i into x_{ji}. Thus, x_{ji} is derived from a_i in a single table look-up.

Example 2.7 Compute the residue mod 7 of the radix 10 integer 1826.
Solution: We begin by decomposing the number

$$\begin{aligned} 1826 &= 1 \times 1000 + 8 \times 100 + 2 \times 10 + 6 \\ &= a_3 \times 10^3 + a_2 \times 10^2 + a_1 \times 10 + a_0 \end{aligned}$$

and note that

$$\begin{aligned} 10\mathbf{mod_7} &= 3 \\ 100\mathbf{mod_7} &= (10\mathbf{mod_7} \times 10\mathbf{mod_7})\mathbf{mod_7} = 2 \\ 1000\mathbf{mod_7} &= (100\mathbf{mod_7} \times 10\mathbf{mod_7})\mathbf{mod_7} = 6. \end{aligned}$$

Thus, we have the following table and get

$$1826\mathbf{mod_7} = (6+2+6+6)\mathbf{mod_7} = 6.$$

a_3	x_{j3}	a_2	x_{j2}	a_1	x_{j1}	a_0	x_{j0}
0	0	0	0	0	0	0	0
1	6	1	2	1	3	1	1
2	5	2	4	2	6	2	2
3	4	3	6	3	2	3	3
4	3	4	1	4	5	4	4
5	2	5	3	5	1	5	5
6	1	6	5	6	4	6	6
7	0	7	0	7	0	7	0
8	6	8	2	8	3	8	1
9	5	9	4	9	6	9	2

It is possible to use larger tables where multiple digit positions are grouped together.

Example 2.8 Compute $1826\mathbf{mod_7}$ again but using two digits at a time
Solution: We have $1826 = 18 \times 100 + 26 = a_2 \times 10^2 + a_0$

and we use the longer corresponding table to get

$$1826\mathbf{mod_7} = (1+5)\mathbf{mod_7} = 6.$$

a_2	x_{j2}	a_0	x_{j0}
0	0	0	0
1	2	1	1
2	4	2	2
3	6	3	3
⋮	⋮	⋮	⋮
18	1	18	4
⋮	⋮	⋮	⋮
26	3	26	5
⋮	⋮	⋮	⋮

Although larger tables have a longer access time, they reduce the number of additions required and thus may improve the speed of conversion.

There is an important special case of conversion into a residue system: converting a $\mathbf{mod_{2^n}}$ number into a residue representation $\mathbf{mod_{2^k}}$ or $\mathbf{mod_{2^k-1}}$. This case is important because of the previously mentioned coding efficiency with these moduli, and because $\mathbf{mod_{2^n}}$ numbers arise from arithmetic operations using conventional binary type logic. To simplify the following discussion, let us present a simple exercise.

$\overset{?}{\Longrightarrow}$ **Exercise 2.5 Prove that for the number $X = \sum x_i \beta^i$ the residue $X\mathbf{mod}_{\beta-1} = (\sum(x_i\mathbf{mod}_{\beta-1}))\,\mathbf{mod}_{\beta-1}$.**

The conversion process from a binary representation (actually, a residue $\mathbf{mod_{2^n}}$) to a residue of either 2^k or $2^k - 1\,(n > k)$ starts by partitioning the n bits into m digits of size k bits; that is, $m = \lceil \frac{n}{k} \rceil$. Then a binary number $X\mathbf{mod_{2^n}}$ with the value

$$X_{base2} = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0,$$

where x_i has value 0 or 1, is rewritten as:

$$X_{base2^k} = X_{m-1}\left(2^k\right)^{m-1} + X_{m-2}\left(2^k\right)^{m-2} + \cdots + X_0,$$

where X_i has values $\{0, 1, \ldots 2^k - 1\}$. This is a simple regrouping of digits. For example, consider a binary 24-bit number arranged in eight 3-bit groups.

$$X_{base2} = 101\ 011\ 111\ 010\ 110\ 011\ 110\ 000.$$

This is rewritten in octal ($k = 3$) as $\lceil \frac{n}{k} \rceil = \lceil \frac{24}{3} \rceil$ digits:

$$X_{base2^3} = 5\ 3\ 7\ 2\ 6\ 3\ 6\ 0.$$

The residue $X\mathbf{mod_{2^k}} = X_0$ (the least significant k bits), since all other digits in the representation are multiplied by 2^k raised to some power which yields 0 as a residue (recall exercise 2.4).

The residue of $X_{base2^k}\mathbf{mod_{2^k-1}}$ is based on the result of exercise 2.5. We compute that residue directly from the $\mathbf{mod_{2^k}}$ representation. If X is a base 2^k number with m digits $(X_{m-1} \ldots X_0)$, and X_i is its i^{th} digit:

$$X\mathbf{mod_{2^k-1}} = \left(\sum_{i=0}^{m-1} X_i\left(2^k\right)^i \mathbf{mod_{2^k-1}}\right) \mathbf{mod_{2^k-1}}.$$

For $X_0\mathbf{mod_{2^k-1}}$, the residue is the value X_0 for all digit values except $X_0 = 2^k - 1$, where the residue is 0. Similarly, for $(X_i(2^k)^i)\mathbf{mod_{2^k-1}}$, the residue is X_i, where $X_i \neq 2^k - 1$ and the residue $= 0$ if $X_i = 2^k - 1$. This is the familiar process of "casting-out" $(\beta - 1)$. In the previous example (X in octal),

$$X = 5\,3\,7\,2\,6\,3\,6\,0$$

and

$$x = X\mathbf{mod_7} = (5 + 3 + 0 + 2 + 6 + 3 + 6 + 0)\mathbf{mod_7}.$$

Now, the sum of two digits $\mathbf{mod_7}$ is computed easily from a $\mathbf{mod_8}$ adder by recognising three cases:

1. $a + b < 2^k - 1$, that is, $a + b < 7$; then $(a + b)\mathbf{mod_7} = (a + b)\mathbf{mod_8} = a + b$.
2. $(a + b) = 2^k - 1$, $a + b = 7$; then $(a + b)\mathbf{mod_7} = 0$—that is, cast out 7's.
3. $a + b > 2^k - 1$, that is, $a + b > 7$; then a carryout occurs in a $\mathbf{mod_8}$ adder since the largest representable number $\mathbf{mod_8}$ is 7. This end-around carry must be added to the $\mathbf{mod_8}$ sum. This is the same idea of **DRC** calculations using **RC** hardware presented earlier. If $a + b > 7$, then $((a + b)\mathbf{mod_8} + 1)\mathbf{mod_8} = (a + b)\mathbf{mod_7}$, i.e., we use end-around carry.

In our example, $X\mathbf{mod_7} = (5+3+0+2+6+3+6+0)\mathbf{mod_7}$.

$$\begin{array}{lcccc}
 & \underbrace{5+3} & \underbrace{0+2} & \underbrace{6+3} & \underbrace{6+0} \\
\text{octal} & 10 & 2 & 11 & 6 \\
\text{mod 7} & 1+0=1 & 2 & 1+1=2 & 6 \\
 & \multicolumn{2}{c}{\underbrace{\hspace{3cm}}} & \multicolumn{2}{c}{\underbrace{\hspace{3cm}}} \\
\text{octal} & \multicolumn{2}{c}{1+2=3} & \multicolumn{2}{c}{2+6=10} \\
\text{mod 7} & \multicolumn{2}{c}{3} & \multicolumn{2}{c}{1+0=1} \\
 & \multicolumn{4}{c}{\underbrace{\hspace{6cm}}} \\
\text{octal} & \multicolumn{4}{c}{3+1=4} \\
\text{mod 7} & \multicolumn{4}{c}{4} \\
\text{and } X\mathbf{mod_7}=4 & & & &
\end{array}$$

Conversion from residue representation is conceptually more difficult; however, the implementation is also straightforward [Stone, 1973].

First, the integer that corresponds to the residue representation that has a "1" in the j^{th} residue position and zero for all other residues is designated the *weight* of the j^{th} residue, w_j. The ordering of the residues (the "j"s) is not important; it can be taken as their order in the physical layout of the datapath in the circuit or in any other order. According to the Chinese remainder theorem, only one integer (mod the product of relatively prime moduli) has a residue representation of 0, 0, 1, 0, ..., 0. That is, it has a zero residue for all positions $\neq j$ and a residue $= 1$ at j.

Now the problem is to scale the weighted sum of the residues up to the integer representation modulo M, the product of the relatively prime moduli. By construction of the weights, $w_j\mathbf{mod_{m_k}} = 0 \forall k \neq j$, i.e. w_j is a multiple of all $m_k (k \neq j)$. Hence, the product

$$\sum_k (x_j w_j)\mathbf{mod_{m_k}} = x_j$$

and $(x_j w_j)\mathbf{mod_{m_j}} = X\mathbf{mod_{m_j}}$ for all j. Thus, to recover the integer X from its residue representation, we sum the weighted residue modulo M:

$$X\mathbf{mod_M} = \left(\sum (x_j w_j)\right)\mathbf{mod_M}.$$

Example 2.9 Suppose we wish to encode integers with the relatively prime moduli 4 and 5. The product (M) is 20. Thus, we encode integers 0 through 19 in residue representation and find the weights (values of X for which the representations are [1,0] and [0,1]) as:

$$\begin{aligned} w_1 &= [1,0] = 5 \\ w_2 &= [0,1] = 16. \end{aligned}$$

Suppose we encode two integers, 5 and 13, in this representation:

$$\begin{aligned} 5 &= [1,0] \\ 13 &= [1,3] \end{aligned}$$

If we sum them we get:

$$\begin{array}{r} [1,0] \\ +[1,3] \\ \hline [2,3] \end{array}$$

To convert this to integer representation:

$$\begin{aligned} (x_1 w_1 + x_2 w_2)\mathbf{mod_{20}} &= X \\ (2 \times 5 + 3 \times 16)\mathbf{mod_{20}} &= 18. \end{aligned}$$

	Residues	
X	$X\mathbf{mod_4}$	$X\mathbf{mod_5}$
0	0	0
1	1	1
2	2	2
3	3	3
4	0	4
5	1	0
6	2	1
7	3	2
8	0	3
9	1	4
10	2	0
11	3	1
12	0	2
13	1	3
14	2	4
15	3	0
16	0	1
17	1	2
18	2	3
19	3	4

2.2.6 Uses of the Residue Number System

In the past, the importance of the residue system was in its theoretic significance rather than in its fast arithmetic capability. While multiplication is straightforward, division is not, and comparisons are quite complex. The conversion from and to binary is also a lengthy operation. In summary, the difficulties in using a residue number system are:

1. the long conversion times,
2. the complexity of number comparisons,
3. the difficulty of overflow detection (whether we are dealing with only positive numbers or both positive and negative numbers), and
4. the indirect division process.

These reasons have limited the applicability of residue arithmetic. With the availability of powerful arithmetic technology, this may change for suitable algorithms and applications. In any event, it remains an important theoretic system, as we shall see when determining the computational time bounds for arithmetic operations.

Another important application of residue arithmetic is error checking. If, in an n-bit binary system:

$$\begin{array}{r} a\mathbf{mod_{2^n}} \\ +b\mathbf{mod_{2^n}} \\ \hline c\mathbf{mod_{2^n}} \end{array}$$

then it also follows that:

$$\begin{array}{r} a\mathbf{mod_{2^k-1}} \\ +b\mathbf{mod_{2^k-1}} \\ \hline c\mathbf{mod_{2^k-1}} \end{array}$$

Since 2^n and $2^k - 1$ are relatively prime, it is possible to use a small k-bit adder ($n \gg k$) to check the operation of the n-bit adder. In practice, $k = 2, 3, 4$ is most commonly used. The larger k's are more expensive, but since they provide more unique representations, they afford a more comprehensive check of the arithmetic. Watson and Hastings [Watson and Hastings, 1966] as well as Rao [Rao, 1974] provide more information on using residue arithmetic in error checking.

$\overset{?}{\Longrightarrow}$ **Exercise 2.6 It has been observed that one can check a list of sums by comparing the single digit sum of each of the digits, e.g:**

$$\begin{array}{rrcr} & 374 & 3+7+4=14; & 1+4=5 \\ & 281 & 2+8+1=11; & 1+1=2 \\ & 523 & 5+2+3=10; & 1+0=1 \\ \cline{2-2}\cline{4-4} & 1178 & & 5+2+1 \\ 1+1+7+8=17; & 1+7=8 & \longleftarrow \ \textit{check} \ \longrightarrow & =8 \end{array}$$

Does this always work? If yes, prove it. If no, show a counterexample and develop a scheme that works.

2.3 The limits of fast arithmetic

2.3.1 Background

This section presents the theoretical bounds on speed of arithmetic operations in order to compare the state of the art in arithmetic algorithms against these bounds. Said differently, those bounds serve as an ideal case to measure the practical results, and provide a clear understanding of how much more speed improvement can be obtained.

2.3.2 Levels of evaluation

The execution speed of an arithmetic operation is a function of two factors. One is the circuit technology, and the other is the algorithm used. It can be confusing to discuss both factors simultaneously; e.g., a ripple carry adder implemented in ECL technology may be faster than a carry-look-ahead adder implemented in CMOS. In this section, we are interested only in the algorithm and not in the technology; therefore, the speed of the algorithms will be expressed in terms of gate delays. Using this approach, the carry-look-ahead adder is faster than the ripple carry adder. Simplistically translating gate delays for a given technology to actual speed is done by multiplying the gate delays by the gate speed.

Modeling at the logic gate level as described above does not capture all the details of real circuits. Design evaluation is an iterative process with several levels of complexity. At each level different ideas are compared and the most promising are tried at the following level of complexity. The possible levels are:

1. Modeling at the logic level just as described above. This does not provide very accurate estimates and can be used for rough comparisons.
2. Implementing the design in transistors and simulating. This level forces the designer to think about sizing the transistors and to buffer any gates that are driving a large fan-out. This level gives a much more accurate estimation of the time delay but it still does not include the long wire delays.
3. For more accuracy, a layout of the full circuit can be done to extract the details about the wires. An extracted circuit (or at least its critical path) can then be simulated to give a more accurate time delay estimate. Area and power consumption estimation are also possible at this level.
4. The design is actually fabricated and the chip is tested. This is the ultimate test for a proposed design with a specific technology process and fabrication facilities.
5. To really show the merit of a proposed idea, the whole design can be simulated over a variety of scalable physical design rule sets and one or more are fabricated and tested.

The model proposed in this chapter is used primarily to evaluate the general trade-offs in the designs assuming that they use the same technology for the fabrication of real circuits.

One might ask:

> Then why is modeling needed? Couldn't we just fabricate the new design and test it to see how it compares to other designs?

Such a fabricated circuit is really the ultimate test to check the veracity of any claims made about a design. However, this is a very costly thing to do every time a designer considers a new idea. Arithmetic units are used in general purpose processors, dedicated hardware for graphics and in digital signal processors. For a certain application and design specifications (speed, area and power consumption), it may be necessary to compare several architectures. The designer cannot fabricate all of these to compare them. A simple model is needed to help in targeting a design for use at a different operand width (for example single and double precision), with a different set of hardware building blocks, or with a different radix and number system. In all such cases, there is no need to remodel. These variables should be parameters in the model.

2.3.3 The (r, d) Circuit Model

Winograd [Winograd, 1965, 1967] presented much of the original work to determine a minimum bound on arithmetic speed. In his model, the speed (in gate delays) of any logic and arithmetic operation is a function of three items:

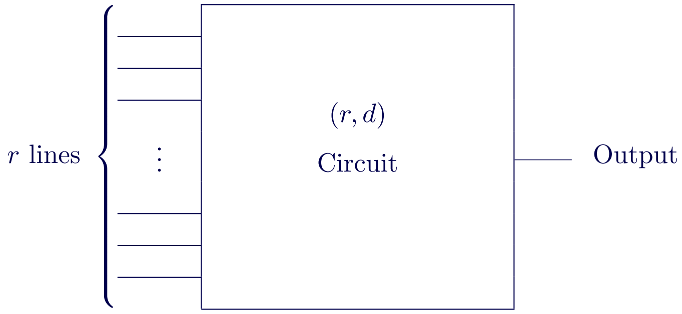


Figure 2.1: The (r, d) circuit.

1. Number of digits (n) in each operand.
2. The maximum fan-in in the circuit (r) which is the largest number of logic inputs or arguments for a logic element.
3. The number of truth values in the logic system (d). Remember that in general, we can implement our arithmetic circuits using multi-valued logic elements and not necessarily binary.

Definition: An (r, d) circuit (Fig. 2.1) is a d-valued logic circuit in which each element has fan-in of at most r, and can compute any r-argument d-valued logic function in unit time.

In any practical technology, logic path delay depends upon many factors: the number of gates (circuits) that must be serially encountered before a decision can be made, the logic capability of each circuit, cumulative distance among all such serial members of a logic path, the electrical signal propagation time of the medium per unit distance, etc. In many high-speed logic implementations, the majority of total logic path delay is frequently attributable to delay external to logic gates, especially the wire delays. Thus, a comprehensive model of performance has to include technology, distance, geography, and layout, as well as the electrical and logical capabilities of a gate. Clearly, the inclusion of all these variables makes a general model of arithmetic performance infeasible. Winograd's (r, d) model of a logic gate is idealised in many ways:

1. There is zero propagation delay between logic blocks.
2. The output of any logic block may go to any number of other logic blocks without affecting the delay; i.e., the model is *fan-out* independent. The fan-out of a gate refers to its ability to drive from output to input a number of other similar gates. Practically speaking, any gate has a maximum limit on the number of circuits it may communicate with based on electrical considerations. Also, as additional loads are added to a circuit, its delay is adversely affected.
3. The (r, d) circuit can perform any logical decision in a unit delay—more comments on this below.
4. Finally, the delay in, and indeed the feasibility of, implementations are frequently affected by mechanical considerations such as the ability to connect a particular circuit module

to another, or the number of connectors through which such an electrical path might be established. These, of course, are ignored in the (r, d) model.

Despite these limitations, the (r, d) model serves as a useful first approximation in the analysis of the time delay of arithmetic algorithms in most technologies. The effects of propagation delay, fan-out, etc., are merely averaged out over all blocks to give an initial estimate as to the delay in a particular logic path. Thus, in a particular technology, the basic delay within a block may be something; but the effective delay, including average path lengths, line loading effects, fan-out, etc., might be three or four times the basic delay. Still, the number of blocks encountered between functional input and final result is an important and primary determinant (again, for most technologies) in determining speed.

The (r, d) model is a fan-in limited model, the number of inputs to a logical gate is limited at r inputs, each gate has one output, and all gates take a unit delay time (given valid inputs) to establish an output. The model allows for multi-valued logic, where d is the number of values in the logic system. The model further assumes that any logic decision capable of being performed within an r inputs d-valued truth system is available in this unit time. This is an important premise. For example, in a 2-input binary logic system ($r = 2$, $d = 2$) there are 16 distinct logic functions (*AND*, *OR*, *NOT*, *NOR*, *NAND*, etc.).

$\overset{?}{\Longrightarrow}$ **Exercise 2.7 Prove that, in general, there are d^{d^r} distinct logic functions in the (r, d) system.**

In any practical logic system, only a small subset of these functions are available. These are chosen in such a way as to be *functionally complete*, i.e., able to generate any of the other logic expressions in the system. However, the functionally complete set in general will not perform a required arbitrary logic function in unit delay, e.g. *NOR*s implementing *XOR* may require two unit delays. Thus, the (r, d) circuit is a lower bound on a practical realization. What we will discover in later chapters is that familiar logic subsets (e.g., *NAND*) can by themselves come quite close to the performance predicted by the (r, d) model.

2.3.4 First Approximation to the Lower Bound

Spira [Spira, 1973] summarizes the lower bounds for the computation time of different arithmetic and logic functions. He shows that if a d-valued output is a function of all n arguments (d-valued inputs), then t, the number of (r, d) delays, is:

$$t \geq \lceil \log_r n \rceil$$

in units of (r, d) circuit delay.

Example 2.10 As shown in Fig. 2.2, for the case of $n = 10$, $r = 4$, and $d = 2$ we get

$$\lceil \log_r n \rceil = \lceil \log_4 10 \rceil = \lceil 1.65 \rceil = 2.$$

Proof: Spira's bound is proved by induction and follows from the definition of the (r, d) circuit. The (r, d) circuit has a single output and r inputs; thus, a single level ($t = 1$) has r inputs. Let f_t designate a circuit with n inputs and t units of delay.

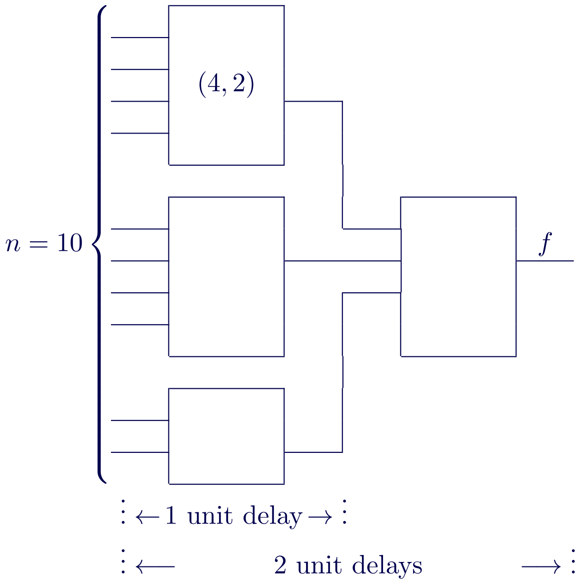


Figure 2.2: Time delays in a circuit with 10 inputs and $(r, d) = (4, 2)$.

Consider the case of unit delay, i.e., $t = 1$. Since the fan-in in a unit block is r, if the number of inputs $n \leq r$ we use one gate to define the function f. In this case, the bound is correct since

$$1 \geq \lceil \log_r n \rceil.$$

Now suppose Spira's bound is correct for delays in a circuit with a time delay equal to $t - 1$ (f_{t-1}). Let us find the resulting delay in the network (Fig. 2.3) for f_t that has n inputs. The last (r, d) gate in this network has r inputs. at least one of those inputs is an output of another subcircuit f_{t-1} which depends on at least $\lceil n/r \rceil$ inputs at time $t - 1$. We are given that the bound is correct for f_{t-1} as a function of $\lceil n/r \rceil$ inputs. Hence,

$$\begin{aligned} t - 1 &\geq \left\lceil \log_r \left\lceil \frac{n}{r} \right\rceil \right\rceil \\ &\geq \left\lceil \log_r \frac{n}{r} \right\rceil . \end{aligned}$$

However,

$$\begin{aligned} \left\lceil \log_r \frac{n}{r} \right\rceil &= \lceil \log_r(n) - log_r(r) \rceil \\ &= \lceil \log_r(n) \rceil - 1. \end{aligned}$$

Hence,

$$t \geq \lceil \log_r(n) \rceil.$$

which proves the bound.

Now we can derive the lower bound for addition in the residue number system.

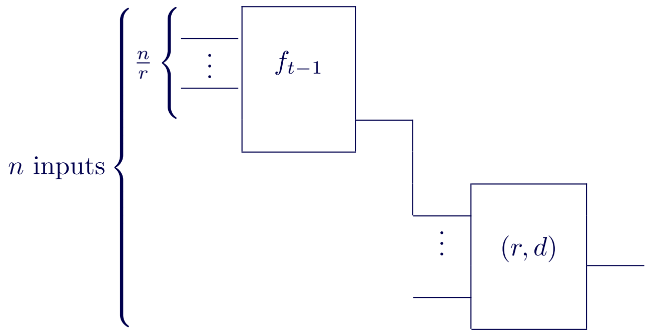


Figure 2.3: The (r, d) network.

2.3.5 Spira/Winograd bound applied to residue arithmetic

In the addition operation, as we have already noted, it is possible for a low order carry in certain configurations to affect the most significant output line. The most significant output line then depends upon all input lines for both operands. According to Spira's bound

$$t \geq \lceil \log_r (2 \times \text{number of inputs per operand}) \rceil .$$

Since residue arithmetic is carry independent between the various moduli m_i, we only need to concern ourselves with the carry and propagation delay for the largest of the moduli. We denote by $\alpha(N)$ the number of distinct values that the largest modulus represents. Hence, $\log_d \alpha(N)$ is the number of d-valued lines required to represent a number for this modulus. Thus, an addition network for this modulus has $2\lceil \log_d \alpha(N) \rceil$ input lines and the time for addition using (r, d) circuits in the residue system is at least:

$$t \geq \lceil \log_r (2\lceil \log_d \alpha(N) \rceil) \rceil ,$$

where $\alpha(N)$ is the number of elements representable by the largest of the relatively prime moduli.

Winograd's theorem is actually more general than the above. That theorem shows that the bound is valid not only for the residue arithmetic but for any arithmetic representation obeying group theoretic properties. In the general case of modular addition, the $\alpha(N)$ function needs more clarification. In modular arithmetic, we are operating with single arguments $\mathbf{mod}_{(\mathbf{A^n})}$. If A is prime, then $\alpha(N)$ is simply A^n. On the other hand, if A is composite (i.e., not a prime), then $A = A_1 A_2 \cdots A_m$ and arithmetic is decomposed into simultaneous operations $\mathbf{mod}_{\mathbf{A_1^n}}$, $\mathbf{mod}_{\mathbf{A_2^n}}$, ..., $\mathbf{mod}_{\mathbf{A_m^n}}$. In this case, $\alpha(N)$ is A_i^n, where A_i is the largest element composing A.

For example, in decimal arithmetic, $A = 10^n = 2^n \times 5^n$ and independent pair arithmetic (residue arithmetic) can be defined for A_2^n and A_5^n, limiting the carry computation to the largest modules; in this case $\alpha(10^n) = 5^n$.

Frequently, we are not interested in a bound for a particular modular system (say A^n), but in a tight lower bound for a residue system that has *at least* the capacity of A^n. We designate such a system $(> A^n)$, since the product of its relatively prime moduli must exceed A^n.

Example 2.11

1. Modular representation:

$$\begin{aligned} \text{prime base} \qquad & \alpha(2^{12}) = 2^{12}, \\ \text{composite base} \qquad & \alpha(10^{12}) = 5^{12}; \end{aligned}$$

   Note: a composite base has multiple factors ($\neq 1$); e.g., $10 = 5 \times 2$ is a composite base, while 2 is not composite.

2. Residue representation: Using the set $\{2^5, 2^5 - 1, 2^4 - 1, 2^3 - 1\}$

$$\alpha(> 2^{16}) = 2^5.$$

Note that 2^5 in the previous example is not necessarily the *minimum* $\alpha(> 2^{16})$. In fact, the minimum $\alpha(> 2^k) = p_n$, where p_n is the n^{th} prime in the product function defined as the smallest product of consecutive primes p_i, or *powers of primes*, that equals or exceeds 2^k:

$$\prod_{i=1}^{n} p_i \geq 2^k.$$

The selection of moduli to minimise the α function is best illustrated by an example.

Example 2.12 Suppose we wish to design a residue system that has $M \geq 2^{47}$, i.e., at least 2^{47} unique representations. We wish to minimise the largest factor of M, $\alpha(M)$, in order to assure fast arithmetic. If we simply selected the product of the primes, we get:

$$2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 \times 29 \times 31 \times 37 \times 41 > 2^{47};$$

that is, the $\alpha(> 2^{47})$ for this selection is 41.
We can improve the α function by using powers of the lower order primes. Thus:

$$2^5 \times 3^3 \times 5^2 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 \times 29 \times 31 > 2^{47}.$$

Here, $\alpha(> 2^{47})$ is $2^5 = 32$.

Thus, finding the minimum α function requires that before increasing the product (in the development of M) by the next larger prime, p_n, we check if there is any lower order prime, p_i, which when raised to its next integer power will lie between p_{n-1} and p_n. That is, for each $i < n - 1$ and x the next integer power of p_i we check if

$$p_{n-1} < p_i^x < p_n.$$

We use all such qualified p_i^x terms before introducing p_n into the product.

2.3.6 Winograd's Lower Bound on Multiplication

Typically, one thinks of multiplication as successive additions and shifts so that a multiplication by an n-inputs operand takes n addition times.

However, Winograd surprises us by saying that multiplication is not necessarily slower than addition! And, if this were not enough, multiplication can be even slightly faster than addition [Brennan, 1968, Winograd, 1967].

Since multiplication is also a group operation involving two n-digit d-valued numbers (whose output is dependent on all inputs), the Spira bound applies.

$$t \geq \lceil \log_r 2n \rceil,$$

where $2n =$ the total number of d-valued input lines.

To see that multiplication can be performed at the same speed as addition, one need only consider multiplication by addition of the log representation of numbers: if $a \times b = c$, then $\log a + \log b = \log c$. This is known as the *Logarithmic Number System* or LNS for short.

Notice that in a log representation, fewer significant product bits are required than in the familiar linear weighted system. For example, $\log_2 16 = 4.0$ requires 4 bits (3, plus one after the binary point) instead of 5 bits, as $16 = 10000$ requires. Of course, log representations require subtraction (i.e., negative log) for numbers less than 1.0, and zero is a special case.

Since division in this representation is simply subtraction, the bound applies equally to multiplication and division. Also, for numbers represented with a composite modular base (i.e., A^n, where $A^n = A_1 \times A_2 \times \ldots \times A_n$), a set of log representations can be used. This coding of each base A number as an n-tuple $\{\log Ai;\ i = 1 \text{ to } n\}$ minimises the length of the carry path by reducing the number of d-valued input lines required to represent a number.

As an analog to residue representation, numbers are represented as composite powers of primes, and then multiplication is simply the addition of corresponding powers.

Example 2.13

12×20

$$\begin{aligned} 12 &= 2^2 \times 3^1 \times 5^0 \\ 20 &= 2^2 \times 3^0 \times 5^1 \\ \text{product } 240 &= 2^4 \times 3^1 \times 5^1 \end{aligned}$$

$12 \div 20$

$$\begin{aligned} 12 &= 2^2 \times 3^1 \times 5^0 \\ 20 &= 2^2 \times 3^0 \times 5^1 \\ 12/20 &= 2^0 \times 3^1 \times 5^{-1} = 3/5 \end{aligned}$$

Winograd formalises this by defining $\beta(N)$ akin to the $\alpha(N)$ of addition and shows that for multiplication:

$$t \geq \lceil \log_r (2 \lceil \log_d \beta(N) \rceil) \rceil$$

where

$$\beta(N) < \alpha(N)$$

The exact definition of $\beta(N)$ is more complex than $\alpha(N)$. Three cases are recognised:

Case 1: Binary radix ($N = 2^n$); $n \geq 3$

$$\beta(2^n) = 2^{n-2}$$

for Binary radix ($N = 2^n$); $n < 3$,

$$\begin{aligned} \beta(4) &= 2 \\ \beta(2) &= 1 \end{aligned}$$

Case 2: Prime radix ($N = p^n$); p a prime > 2

$$\begin{aligned} \beta(p^n) &= \textbf{max}\left(p^{n-1}, \alpha(p-1)\right) \\ \text{e.g., } \beta(59) &= \alpha(58) = \alpha(29 \times 2) = 29 \end{aligned}$$

Case 3: Composite powers of primes ($N = p_1^{n_1} \times p_2^{n_2} \times \cdots p_m^{n_m}$)

$$\beta(N) = \textbf{max}\left(\beta(p_1^{n_1}), \ldots, \beta(p_i^{n_i}) \ldots\right).$$

Example 2.14

1. $N = 2^{10} \Rightarrow \beta(2^{10}) = 2^8$.
2. $N = 5^{10} \Rightarrow \beta(5^{10}) = 5^9$.
3. $N = 10^{10}$

$$\begin{aligned} 10^{10} = 5^{10} \times 2^{10} &= \beta(5^{10}, 2^{10}) \\ &= \textbf{max}\left(\beta(5^{10}), \beta(2^{10})\right) \\ &= \textbf{max}(5^9, 2^8) \\ &= 5^9 \end{aligned}$$

In order to reach the lower bounds of addition or multiplication, it is necessary to use data representations that are nonstandard. By optimising the representation for fast addition or multiplication, a variety of other operations will occur much slower. In particular, performing comparisons or calculating overflow are much more difficult and require additional hardware using this nonstandard representation. Winograd showed that both these functions require at least $\lceil \log_r(2\lceil \log_2 N \rceil) \rceil$ time units to compute [Winograd, 1967]. In conventional binary notation, both of these functions can be easily implemented by making minor modifications to the adder. Hence, the type of data representation used must be decided from a broader perspective, and not based merely on the addition or multiplication speed.

2.4 Modeling the speed of memories

As an alternative to computing sums or products each time the arguments are available, it is possible to simply store all the results in a table. We then use the arguments to look up (address) the answer, as shown in example 2.6.

Does such a scheme lead to even faster arithmetic, i.e., better than the (r, d) bound? The answer is probably not, since the table size grows rapidly as the number of argument digits, n, increases.

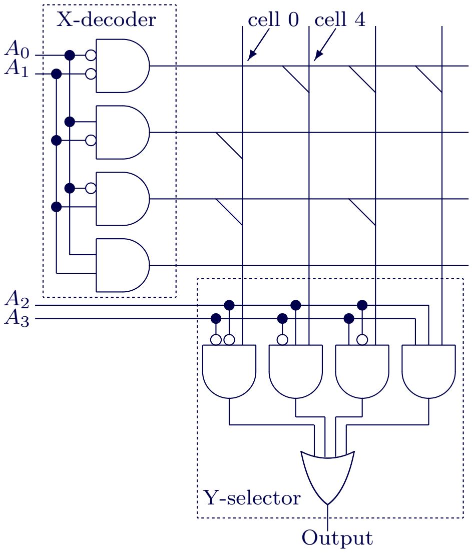


Figure 2.4: A simple memory model.

For β-based arithmetic, there are β^{2n} required entries. Access delay is obviously a function of the table size.

Modeling this delay is not the same as finding a lower time bound, however. In ROMs as well as many storage technologies, the access delay is a function of many physical parameters. What we present here is a simple model of access delay as an approximation to the access time.

We start by the simple model of Fig. 2.4 for a 16×1 ROM. This ROM is made of 16 cells which store information by having optional connections at each row and column intersection. For example, in Fig. 2.4, if the lines are normally pulled low then cell $0(A_3A_2A_1A_0 = 0000)$ stores a zero and cell 4 stores a one. The delay of the ROM is a combination of the X decoder, the memory cells, and the Y selector. In a real memory the effective load seen by the individual memory cell when it attempts to change the value on the wire affects its delay. To produce a simple yet useful model we just assume that the cell has one gate delay. Hence, the delay of the memory in the figure is made of four gates for fan-in $r \leq 4$.

In general, a memory with L address lines has half of the address lines $(L/2)$ decoded in the X dimension, and according to Spira's bound the associated delay is $\lceil \log_r(\frac{L}{2}) \rceil$.

In the Y-selector delay, the fan-in to each gate is composed of the $L/2$ address lines plus a single input from the ROM array. These gates must, in turn, be multiplexed to arrive at a final result. As there are $2^{\frac{L}{2}}$ array outputs, there are $\lceil \log_r 2^{\frac{L}{2}} \rceil$ stages of delay, again by the Spira argument. Hence, the table look-up has the following delays:

$$\begin{aligned} X\text{-decoder} &= \left\lceil \log_r \left(\frac{L}{2} \right) \right\rceil \\ \text{Memory cell} &= 1 \end{aligned}$$

$$Y\text{-selector} \quad = \quad \left\lceil \log_r \left(\frac{L}{2}+1\right)\right\rceil + \left\lceil \log_r 2^{\frac{L}{2}} \right\rceil$$

Actually, since only the input arriving from the memory cell to the Y-selector is critical, it is possible to have an improved configuration. Each input to the Y-selector from the memory cells is brought down to a two input *AND* gate. The other input to this *AND* gate is the decoded Y-selection. Now the Y-decoder delay is increased by one gate delay[2] but this is not worse than the X-decoder plus the memory cell delay. Thus the total memory access time is that of the X-decoder, followed by the access to the cell, then by the unoverlapped part of the Y-selection. The unoverlapped part consists of the two input *AND* gate and a $2^{\frac{L}{2}}$ inputs *OR* gate with a delay of:

$$\text{Unoverlapped Y-selector delay} = 1 + \left\lceil \log_r 2^{\frac{L}{2}} \right\rceil,$$

giving as a total:

$$\text{ROM delay} = 2 + \left\lceil \log_r \frac{L}{2} \right\rceil + \left\lceil \log_r 2^{\frac{L}{2}} \right\rceil.$$

Example 2.15 For a $1K$ word ROM, $L = 10$. If we assume $r = 5$, what is its access time?
Solution: The time delays of the different parts are

$$\begin{aligned} X\text{-decode} &= 1 \\ \text{cells} &= 1 \\ Y\text{-selector} &= 1+3=4 \\ \text{total} &= 6 \text{ gate delays} \end{aligned}$$

When the ROM is used as a binary operator on n-bit numbers, the preceding formula is expressed as a function of n, where $n = \frac{L}{2}$:

$$\text{ROM delay} = 2 + \lceil \log_r n \rceil + \lceil \log_r 2^n \rceil.$$

In many ways, this memory delay points out the weakness of the (r, d) circuit model. In practical use of VLSI memory implementations, the delay equation above is conservative when normalized to the gate delay in the same technology. The (r, d) model gives no "credit" to the ROM for its density, regular implementation structure, limited fan-out requirements, etc.

2.5 Modeling the multiplexers and shifters

Arithmetic circuits, especially for floating point numbers, usually contain a number of multiplexers and combinational shifters. A designer needs simple model for those two elements as well. However, before we introduce new models, let us think for a while on the issues raised

[2] In the special case where $\frac{L}{2} < r$ it is simpler to integrate the Y-selector and *AND* gate to have a gate delay of 1.

so far by the (r, d) gate. In section 2.3.3, we alluded to the fan-out problem that the simple (r, d) model of Winograd ignores and said that the effective delay (that includes the path length, loading, and fan-out) may be three or four times the basic delay given by the model. We will attempt to clarify why this difference between the basic delay and the effective one is so in modern technologies.

Designers of CMOS circuits sometimes present the time delay of a block in what is termed "fanout of 4" delays: the delay of an inverter driving a load that is four times its own size. This is commonly abbreviated as *FO4* for the "fanout of 4" inverter. A chain of inverters properly scaled so that each one is four times the size of the preceding one is used to estimate the time delay of an *FO4* inverter. The pull up and pull down times of the middle inverters is then measured and averaged to the unit called *FO4* delay.

Using units of *FO4* delays makes our modeling independent of the technology scaling to a large degree since this elementary gate scales almost linearly with the technology McFarland [1997]. Such units also make the model take into effect the time delay associated with the small local wires inside the *FO4* inverter as well as those connecting it to neighbouring gates. However, our model does not include any assumptions about long wires across the chip and the time delay associated with them. Hence, obviously, it does not give an accurate estimate of the absolute delay of a logic unit. However, the model is still useful to compare different architectures to estimate their relative speeds.

In section 2.3.3 we also noted that the (r, d) model assumes that any logical function is performed in the same time delay. In real circuits this equal delay cannot be exact. However, in our modeling we will not differentiate between the time delay of the different types of gates. Designers usually change the sizes of the transistors in attempt to equalize the time taken by all gates on the critical path. The general rule that designers apply is: "keep it close to a *FO4* delay." Hence, in the subsequent we use the term *FO4* delay as equivalent to the gate delay used earlier.

More elaborate models for time delays in CMOS circuits exist. The logical effort model [Sutherland and Sproull, 1991] which includes the fanout as well as a correction for the type of gate used is an important example. Using logical effort, a designer might resize the transistors inside the logic gates to achieve the best performance. A model such as logical effort requires a knowledge of the exact connections between the gates as well as the internal design of the gates. Our target here is to present a model at a slightly higher level where the designer does not know yet the exact topology of the circuit nor the specific gates used. The target of our model is for the preliminary work of evaluation between different algorithms and architectures. Our model is also useful when the information is only available in the form of block diagrams as is often the case with published material from other research institutions or industrial companies. It is possible with our simple model to get a first approximation of the speed of such published designs and compare them to any new ideas the designer is contemplating.

Having said all that, we are still able to include the effect of fanout in some cases without a detailed knowledge of the circuit topology. Let us see how this is done with the multiplexers and shifters.

A single m-to-1 multiplexer is considered to take only one *FO4* delay from its inputs to the output assuming it is realized using CMOS pass gates. This assumption for the multiplexer is valid up to a loading limit. Small m is the usual case in VLSI design since multiplexers rarely exceed say a 5-to-1 multiplexer. By simulation, we find that the 2-to-1, 3-to-1 and 4-to-1

Table 2.2: Time delay of various components in terms of number of *FO4* delays. r is the maximum fan-in of a gate and n is the number of inputs.

Part	Delay
Multiplexer, input to output	1
Multiplexer, select to output	$\lceil \log_4(n) \rceil + 1$
Shifter	$\lceil \log_2(n) \rceil$
Memory	$2 + \lceil \log_r \frac{n}{2} \rceil + \lceil \log_r 2^{\frac{n}{2}} \rceil$
Spira's bound (no design details)	$\lceil \log_r(n) \rceil$

multiplexers exhibit a time delay from the inputs to the output within the range of one *FO4* delay. When the input lines are held constant and the select lines change, the delay from the select lines to the output is between one and two *FO4* delays. Hence, for a single multiplexer the delay from the select lines to the output is bounded by 2 *FO4* delays.

A series of m to 1 multiplexers connected to form a larger n-bit multiplexer heavily loads its select lines. Hence there is even a larger delay from the select lines to the output in this case. To keep up a balanced design with a fanout of four rule, each four multiplexers should have a buffer and form a group together. Four such groups need yet another buffer and form a super group and so on. The delay of the selection is then $\lceil \log_4(n) \rceil + 1$. This last formula is applicable even in the case of a single multiplexer since it yields 2 as given above.

Combinational shifters are either done by a successive use of multiplexers or as a barrel shifter realized in CMOS pass transistors. In either case, the delay of an n-way shifter from its inputs to its outputs takes $\lceil \log_2(n) \rceil$ *FO4* delays. The select lines are heavily loaded as in the case of multiplexers. However, if the same idea of grouping four basic cells is used then the delay from the select lines is the same as for the multiplexers. This is smaller than the delay from the inputs to the outputs in the shifter. Hence the input to output delay dominates and is the only one used.

In the following chapters, we will discuss some basic cells used to build adders and multipliers and model their time delays with the simple ideas presented in this chapter.

A designer is able to use these ideas for other pieces of combinational logic where a specific design is reported in the published papers. If the detailed design is not known, and the logic has n inputs then Spira's bound of $\lceil \log_r(n) \rceil$ *FO4* delays is a safe estimate. The different parts presented thus far are summarised in Table 2.2. Note that for the memory in this table the symbol n represents the total number of address lines.

2.6 Additional Readings

The two classic works in the development of residue arithmetic are by Garner [Garner, 1959] and Szabo and Tanaka [Szabo and Tanaka, 1967]. They both are recommended to the serious student.

A readable, complete proof of Winograd's addition bound is found in Stone [Stone, 1973], a book that is also valuable for its introduction to residue arithmetic.

For those interested in modeling the time delays of circuits, the logical effort method is explained in details in a book [Sutherland et al., 1999] by the same name.

2.7 Summary

A designer must explore the different possibilities to improve the speed, area, and power consumption of a circuit. This exploration is greatly simplified by good models.

Alternate representation techniques exist using multiple moduli. These are called residue systems, with the principle advantage of allowing the designer to use small independent operands for arithmetic. Thus, a multitude of these smaller arithmetic operations are performed simultaneously with a potential speed advantage. As we will see in later chapters, the speed advantage is usually limited to about a factor of 2 to 1 over more conventional representations and techniques. Thus, the difficulty in performing operations such as comparison and overflow detection limits the general purpose applicability of the residue representation approach. Of course, where special purpose applications involve only the basic add–multiply, serious consideration could be given to this approach.

Winograd's bound, while limited in applicability by the (r, d) model, is an important and fundamental limitation to arithmetic speed.

2.8 Problems

Problem 2.1 Using residues of the form 2^k and $2^k - 1$, create an efficient residue system to include the range ± 32. Develop all tables and then perform the operation $-3 \times 2 + 7$.

Problem 2.2 The residue system is used to span the range of 0 to 10 000. What is the best set that includes the smallest maximum modulus (i.e., $\alpha(N)$)?

1. If any integer modulus is permitted.
2. If moduli only of the form 2^k or $2^k - 1$ are allowed.

Problem 2.3 Repeat the above problem, if the range is to be $\pm 8\,192$.

Problem 2.4 Analyse the use of an excess code as a method of representing both positive and negative numbers in a residue system.

Problem 2.5 Suppose that two m bit numbers, A and B, are added and we need to check if the sum S is correct. An even parity check on the sum is proposed for error detection. Let us denote the parity of the sum as P_S while those of the numbers as P_A and P_B.

$$\begin{array}{ccc} & \boxed{A} & \boxed{P_A} \\ + & \boxed{B} & \boxed{P_B} \\ \hline & \boxed{S} & \boxed{P_S} \end{array}$$

1. Show that a scheme where the parity of the sum P_S is compared to the parity of $(P_A + P_B)$ (i.e. $P(P_A + P_B) \overset{?}{=} P_S$) is not adequate to detect the errors.
2. Describe an n-bit check (i.e., P_A, P_B, and P_S, each n bits but not necessarily representing a parity function) so that it is possible to detect arithmetic errors with any of the functions $+, -, \times$, i.e. $P(P_A\{+, -, \times\}P_B) = P_S$
3. Find the probability of an undetected error in the new system, where this probability is defined as:
$$\frac{\text{Number of valid representations}}{\text{Total number of representations}}.$$
4. Devise an alternative scheme that provides a complete check on the sum using parity. This system may use logic on the individual bit sum and carry signals to complete the check.

Problem 2.6 In example 2.12 we found the optimum decomposition of prime factors for $M \geq 2^{47}$. Find the next seven factors (either a new prime or a power of prime) following the "32" term to form a new system going up to $M' > M$. What is the new M' (approximately) and the new $\alpha(M')$?

Problem 2.7 If $r = 4$, $d = 2$, and M and M' as defined in Problem 2.6, find:

1. the lower bound on addition,
2. the lower bound on multiplication, and
3. the number of equivalent gate delays using a ROM implementation of addition or multiplication.

Problem 2.8 It is desired to perform the computation $z = \frac{1}{x} + y$ in hardware as fast as possible. Given that x and y are fractions $(0.5 \leq x, y < 1)$ represented in 8 bits, evaluate the number of gate delays $(r = 4)$ if

1. a single table look-up is used to evaluate z,
2. a table look-up is used to find $\frac{1}{x}$, then the result is added to y using an adder with gate delays $= 4\lceil \log_r 2n \rceil$ where n is the number of bits in one operand of the adder.

If x and y are n-bit numbers, for what values of n is the single look-up table better than the combination of a table and an adder? (Ignore ceiling function effects in your evaluation.)

Problem 2.9 One of your friends claims that he uses a "cast-out 8's" check to check decimal addition. His explanation is:

> Find a check digit for each operand by summing the digits of a number. If the result contains multiple digits, sum them until they are reduced to a single digit. If anywhere along the way we encounter an '8,' discard the '8' and *subtract 1*! If we encounter a '9,' ignore it (i.e., treat it as '0'). The sum of the check digits then equals the check digit of the sum as in this example:

$$\begin{array}{rrrrrrcrlcr}
 & 3 & 4 & 8 & 3 & 1 & = & 3+4-1+3+1 & =10=1+0 & = & 1\\
+ & 8 & 8 & 7 & 2 & 1 & = & -1-1+7+2+1 & =8 & = & -1\\
\cline{1-6}\cline{11-11}
1 & 2 & 3 & 5 & 5 & 2 & & & & & 0\\
\multicolumn{6}{r}{1+2+3+5+5+2} & = & 18 & =1-1 & = & 0
\end{array}$$

Does this always work? Prove or show a counterexample.

Problem 2.10 Yet *another* sum checking scheme has been proposed! The details are:

> Find the check digits by adding *pairs* of digits together, reducing to a final pair. Then subtract the leading digit from the unit digit. If the result is negative but not equal to -1, recomplement (i.e., add 10) and then add "1." If -1, leave it alone. Always reduce to a single digit, either -1, 0, or a positive digit as in this example:

$$\begin{array}{rrrlrlrl}
03 & 48 & 31 & =03+48+31= & 82 & =-6= & +5 & \\
+08 & 87 & 21 & =08+87+21= & 116 & =17= & +6 & \\
\cline{1-3}\cline{7-7}
 & & & & & & 11 & =0\\
12 & 35 & 52 & & & & & \\
\multicolumn{3}{r}{12+35+52} & & =99 & & & =0
\end{array}$$

How about this one, will it always work? Prove, or show a counterexample.

Problem 2.11 Several authors suggested the use of the residue number system in low power applications. An embedded system has an RNS adder for the calculation of the hours, minutes and seconds. If any integer modulus is permitted, choose the best set to represent each of the three fields (hours, minutes, and seconds). Please state the reasons for your choice.

Show how the following operation is performed:

	hours	minutes	seconds
	13	10	55
+	10	12	04

Can you think of a problem that exists due to the use of RNS in such a system? You do not need to solve it, just state it. (Hint: add big numbers.)

Why do you think those authors suggested RNS for low power applications?

Problem 2.12 A designer of a residue number system wants to use $\mu = 2^k + 2^{k-1}$ as a modulus where k is a positive nonzero integer.

1. For an unsigned binary number Y represented in $k+1$ bits as $y_k y_{k-1} y_{k-2} \dots y_0$, what is the result of $Y\mathbf{mod}_\mu$? Sketch the design of a circuit perfoming this modulo operation.
2. Prove that $2^{i(k+1)}\mathbf{mod}_\mu = (2^{k-1})^i\mathbf{mod}_\mu$ for any positive integer i.
3. If an unsigned integer number X is represented by n bits with n an exact multiple of $k+1$, use the fact that $2^{i(k+1)}\mathbf{mod}_\mu = (2^{k-1})^i\mathbf{mod}_\mu$ to find $X\mathbf{mod}_\mu$ as a function of the bits of X.

Problem 2.13 A designer of a residue number system wants to use $2^k + 1$ as a modulus where k is a positive nonzero integer.

1. Prove that $2^{ik}\mathbf{mod}_{\mathbf{2^k+1}} = (-1)^i\mathbf{mod}_{\mathbf{2^k+1}}$ for any positive integer i.
2. If an unsigned integer number X is represented by n bits with n an exact multiple of k, use the fact that $2^{ik}\mathbf{mod}_{\mathbf{2^k+1}} = (-1)^i\mathbf{mod}_{\mathbf{2^k+1}}$ to find $X\mathbf{mod}_{\mathbf{2^k+1}}$ as a function of the bits of X.
3. What are the modifications required if n is not an exact multiple of k?
4. What are the changes to your previous answers (if any) when X is a signed number represented in two's complement notation?
5. Using a number of regular binary adders each taking two k bits operands, how do you implement $X\mathbf{mod}_{\mathbf{2^k+1}}$? Hint: analyze the cases of an addition result equal to 2^k or 2^k+1 carefully.

Chapter 3

Addition and Subtraction (Incomplete chapter)

3.1 Fixed Point Algorithms

3.1.1 Historical Review

The first electronic computers used ripple-carry addition. For this scheme, the sum at the i^{th} bit is:

$$S_i = A_i \oplus B_i \oplus C_i,$$

where S is the sum bit, A_i and B_i are the i^{th} bits of each operand, and C_i is the carry into the i^{th} stage. The carry to the next stage $(i+1)$ is:

$$C_{i+1} = A_i B_i + C_i(A_i + B_i)$$

Thus, to add two n-bit operands takes at the most $n-1$ carry delays and one sum delay; but on the average the carry propagation is about $\log_2 n$ delays (see Problem **??** at the end of this chapter). In the late fifties and early sixties, most of the time required for addition was attributable to carry propagation. This observation resulted in many papers describing faster ways of propagating the carry. In reviewing these papers, some confusion may result unless one keeps in mind that there are two different approaches to speeding up addition. The first approach is *variable time addition* (asynchronous), where the objective is to detect the completion of the addition as soon as possible. The second approach is *fixed time addition* (synchronous), where the objective is to propagate the carry as fast as possible to the last stage for all operand values. Today the second approach is preferred, as most computers are synchronous, and that is the only approach we describe here. However, a good discussion of the variable time adder is given by Weigel Wiegel [1966] in his report "Methods of Binary Additions," which also provides one of the best overall summaries of various hardware implementations of binary adders.

Conventional fixed-time adders can be roughly categorized into two classes of algorithms: conditional sum and carry-look-ahead. Conditional sum was invented by Sklansky Sklansky [1960], and has been considered by Winograd Winograd [1967] to be the fastest addition algorithm, but it never has become a standard integrated circuit building block. In fact, Winograd showed that with (r, d) circuits, the lower bound on addition is achievable with the conditional sum algorithm. The carry-look-ahead method was first described by Weinberger and Smith in 1956 Weinberger and Smith [1956], and it has been implemented in standard ICs that have been used to build many different computer systems. A third algorithm described in this chapter, *canonic addition*, is a generalization of the carry-look-ahead algorithm that is faster than either conditional sum or carry-look-ahead. Canonic addition has implementation limitations, especially for long word length operands. A fourth algorithm, the Ling adder Ling [1981], uses the ability of certain circuits to perform the OR function by simply wiring together gate outputs. Ling adders provide a very fast sum, performing close to Winograd's bound, since the (r, d) circuit premise is no longer valid.

3.1.2 Conditional Sum

The principle in conditional sum is to generate, for each digit position, a sum digit and a carry digit assuming that there is a carry into that position, and another sum and carry digit assuming there is no carry input. Then pairs of conditional sums and carries are combined according to whether there is (and is not) a carry into that pair of digits. This process continues until the true sum results. Figure 3.1 illustrates this process for a decimal example.

In order to show the hardware implementation of this algorithm, the equations for a 4-bit slice can be derived.

The subscripts N and E are used to indicate no carry input and carry input (exists), respectively, to the 4-bit slice.

At each bit position the following relations hold:

$$\left.\begin{array}{rcl} S_{\mathrm{N}i} & = & A_i \oplus B_i \\ C_{\mathrm{N}(i+1)} & = & A_i B_i \end{array}\right\} \qquad \text{when } C_i = 0,$$

$$\left.\begin{array}{rcl} S_{\mathrm{E}i} & = & \overline{S}_{\mathrm{N}i} \\ C_{\mathrm{E}(i+1)} & = & A_i + B_i \end{array}\right\} \qquad \text{when } C_i = 1.$$

The following is a shorthand notation (which also assumes each operation takes a unit gate delay):

$$\begin{array}{rcl} G_i & = & A_i B_i \\ P_i & = & A_i + B_i \\ T_i & = & A_i \oplus B_i \quad (T_i, \text{ toggle bit}) \end{array}$$

For the 4-bit slice $i = 0, 1, 2, 3$.

$$\begin{array}{rcl} S_{\mathrm{N}0} & = & A_0 \oplus B_0 \\ S_{\mathrm{E}0} & = & \overline{S}_{\mathrm{N}0} \\ S_{\mathrm{N}1} & = & A_1 \oplus B_1 \oplus G_0 \end{array}$$

<table>
<tr><td>$i \rightarrow$</td><td colspan="2">15</td><td colspan="2">14</td><td colspan="2">13</td><td colspan="2">12</td><td colspan="2">11</td><td colspan="2">10</td><td colspan="2">9</td><td colspan="2">8</td><td></td></tr>
<tr><td>X_i</td><td colspan="2">2</td><td colspan="2">6</td><td colspan="2">7</td><td colspan="2">7</td><td colspan="2">4</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">0</td><td></td></tr>
<tr><td>Y_i</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">0</td><td colspan="2">4</td><td colspan="2">9</td><td colspan="2">7</td><td colspan="2">9</td><td colspan="2">4</td><td></td></tr>
<tr><td></td><td>08</td><td>07</td><td>13</td><td>12</td><td>08</td><td>07</td><td>12</td><td>11</td><td>14</td><td>13</td><td>09</td><td>08</td><td>10</td><td>09</td><td>05</td><td>04</td><td>t_0</td></tr>
<tr><td></td><td colspan="2">083</td><td colspan="2">082</td><td colspan="2">082</td><td colspan="2">081</td><td colspan="2">139</td><td colspan="2">138</td><td colspan="2">095</td><td colspan="2">094</td><td>t_1</td></tr>
<tr><td></td><td colspan="4">08282</td><td colspan="4">08281</td><td colspan="4">13895</td><td colspan="4">13894</td><td>t_2</td></tr>
<tr><td></td><td colspan="8">082823895</td><td colspan="8">082823894</td><td>t_3</td></tr>
<tr><td></td><td colspan="16"></td><td>t_4</td></tr>
</table>

<table>
<tr><td>$i \rightarrow$</td><td colspan="2">7</td><td colspan="2">6</td><td colspan="2">5</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">2</td><td colspan="2">1</td><td colspan="2">0</td><td></td></tr>
<tr><td>X_i</td><td colspan="2">2</td><td colspan="2">6</td><td colspan="2">9</td><td colspan="2">2</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">8</td><td></td></tr>
<tr><td>Y_i</td><td colspan="2">1</td><td colspan="2">5</td><td colspan="2">1</td><td colspan="2">7</td><td colspan="2">1</td><td colspan="2">6</td><td colspan="2">4</td><td colspan="2">5</td><td></td></tr>
<tr><td></td><td>04</td><td>03</td><td>12</td><td>11</td><td>11</td><td>10</td><td>10</td><td>09</td><td>06</td><td>05</td><td>10</td><td>09</td><td>10</td><td>09</td><td></td><td>13</td><td>t_0</td></tr>
<tr><td></td><td colspan="2">042</td><td colspan="2">041</td><td colspan="2">110</td><td colspan="2">109</td><td colspan="2">060</td><td colspan="2">059</td><td colspan="2"></td><td colspan="2">103</td><td>t_1</td></tr>
<tr><td></td><td colspan="4">04210</td><td colspan="4">04209</td><td colspan="4"></td><td colspan="4">06003</td><td>t_2</td></tr>
<tr><td></td><td colspan="8"></td><td colspan="8">042096003</td><td>t_3</td></tr>
<tr><td></td><td colspan="16">08282389442096003</td><td>t_4</td></tr>
</table>

Selector bit = The most significant digit of each number.

The addition performed is:

$$\begin{array}{cccccccccccccccc}
2 & 6 & 7 & 7 & 4 & 1 & 0 & 0 & 2 & 6 & 9 & 2 & 4 & 3 & 5 & 8 \\
5 & 6 & 0 & 4 & 9 & 7 & 9 & 4 & 1 & 5 & 1 & 7 & 1 & 6 & 4 & 5 \\
\hline
8 & 2 & 8 & 2 & 3 & 8 & 9 & 4 & 4 & 2 & 0 & 9 & 6 & 0 & 0 & 3
\end{array}$$

At any digit position, two numbers are shown at t_0. The right number assumes no carry input, and the number on the left assumes that there is a carry input. During t_1, pairs of digits are combined, and now with each pair of digits two numbers are shown. On the right, no carry-in, and on the left, a carry-in is assumed. This process continues until the true sum results (t_4).

Figure 3.1: Example of the conditional sum mechanism.

$$
\begin{aligned}
S_{\mathrm{E}1} &= A_1 \oplus B_1 \oplus P_0 \\
S_{\mathrm{N}2} &= A_2 \oplus B_2 \oplus (G_1 + T_1 G_0) \\
S_{\mathrm{E}2} &= A_2 \oplus B_2 \oplus (G_1 + T_1 P_0) \\
S_{\mathrm{N}3} &= A_3 \oplus B_3 \oplus (G_2 + T_2 G_1 + T_2 T_1 G_0) \\
S_{\mathrm{E}3} &= A_3 \oplus B_3 \oplus (G_2 + T_2 G_1 + T_2 T_1 P_0) \\
C_{\mathrm{N}4} &= G_3 + T_3 G_2 + T_3 T_2 G_1 + T_3 T_2 T_1 G_0 \\
C_{\mathrm{E}4} &= G_3 + T_3 G_2 + T_3 T_2 G_1 + T_3 T_2 T_1 P_0
\end{aligned}
$$

Of course, terms such as $G_1 + T_1 G_0$ could also be written in the more familiar form $G_1 + P_1 G_0$, which is logically equivalent. Replacing T_i with P_i may simplify the implementation.

Thus, the 4-bit sums are generated, and the true sum is selected according to the lower order carry-in, i.e:

$$
\begin{array}{ccccc}
S_0 & = & S_{\mathrm{E}0} C_0 & + & S_{\mathrm{N}0} \overline{C}_0 \\
\vdots & & \vdots & & \vdots \\
S_3 & = & S_{\mathrm{E}3} C_0 & + & S_{\mathrm{N}3} \overline{C}_0
\end{array}
$$

Figure 3.2 shows the logic diagram of a 4-bit slice (conditional sum) adder.

In general, the true carry into a group is formed from the carries of the previous groups. In order to speed up the propagation of the carry to the last stage, look-ahead techniques can be derived assuming a 4-bit adder as a basic block. The carry-out of bit $i (C_i)$ is valid whenever a carry-out is developed within the 4-bit group $(C_{\mathrm{N}i})$, or whenever there is a conditional carry-out $(C_{\mathrm{E}i})$ for the group *and* there was a valid carry-in (C_{i-4}). Using this, we have:

$$
\begin{aligned}
C_4 &= C_{\mathrm{N}4} + C_{\mathrm{E}4} C_0 \\
C_8 &= C_{\mathrm{N}8} + C_{\mathrm{E}8} C_4 \\
C_8 &= C_{\mathrm{N}8} + C_{\mathrm{E}8} C_{\mathrm{N}4} + C_{\mathrm{E}8} C_{\mathrm{E}4} C_0 \\
C_{12} &= C_{\mathrm{N}12} + C_{\mathrm{E}12} C_8 \\
C_{12} &= C_{\mathrm{N}12} + C_{\mathrm{E}12} C_{\mathrm{N}8} + C_{\mathrm{E}12} C_{\mathrm{E}8} C_{\mathrm{N}4} + C_{\mathrm{E}12} C_{\mathrm{E}8} C_{\mathrm{E}4} C_0 \\
C_{16} &= C_{\mathrm{N}16} + C_{\mathrm{E}16} C_{12} \\
C_{16} &= C_{\mathrm{N}16} + C_{\mathrm{E}16} C_{\mathrm{N}12} + C_{\mathrm{E}16} C_{\mathrm{E}12} C_{\mathrm{N}8} + C_{\mathrm{E}16} C_{\mathrm{E}12} C_{\mathrm{E}8} C_{\mathrm{N}4} + \\
&\quad C_{\mathrm{E}16} C_{\mathrm{E}12} C_{\mathrm{E}8} C_{\mathrm{E}4} C_0
\end{aligned}
$$

Note that a fan-in of 5 is needed in the preceding equations to propagate the carry across 16 bits in two gate delays. Thus, 16-bit addition can be completed in seven gate delays: three to generate conditional carry, two to propagate the carry, and two to select the correct sum bit. This delay can be generalized for n bits and r fan-in (for $r \geq 4$ and $n \geq r$) as:

$$
t = 5 + 2 \left\lceil \log_{r-1}(\lceil n/r \rceil - 1) \right\rceil \tag{3.1}
$$

The factor 5 is determined by the longest C_4 path in Figure 3.2. The n bits of each operand are broken into $\lceil \frac{n}{r} \rceil$ groups, as shown in the equations for $\mathrm{C}_{\mathrm{N}4}$ and $\mathrm{C}_{\mathrm{E}4}$, but since the lowest order group (C_4) is already known, only $\lceil \frac{n}{r} \rceil - 1$ groups must be resolved. Finally, sum resolution can

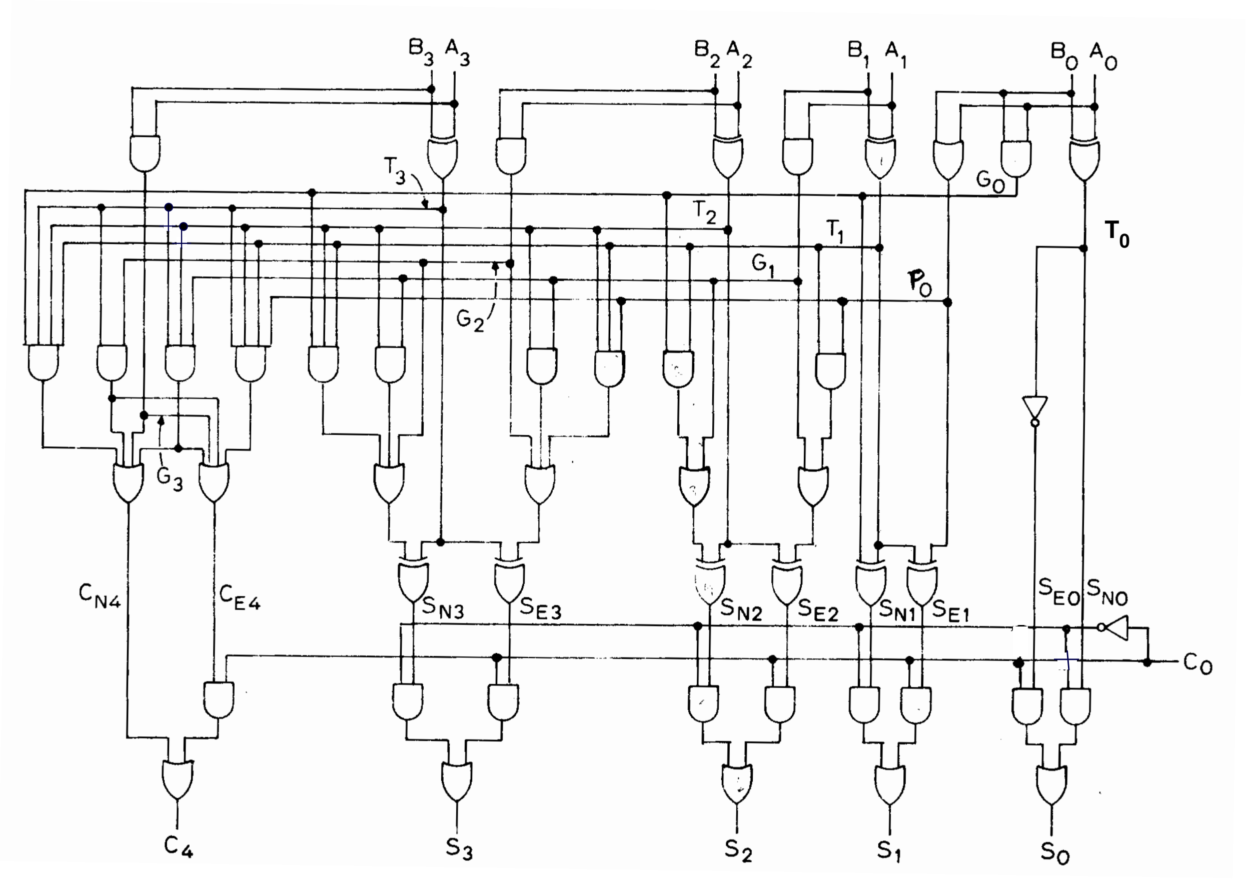


Figure 3.2: 4-bit conditional sum adder slice with carry-look-ahead (gate count= 45).

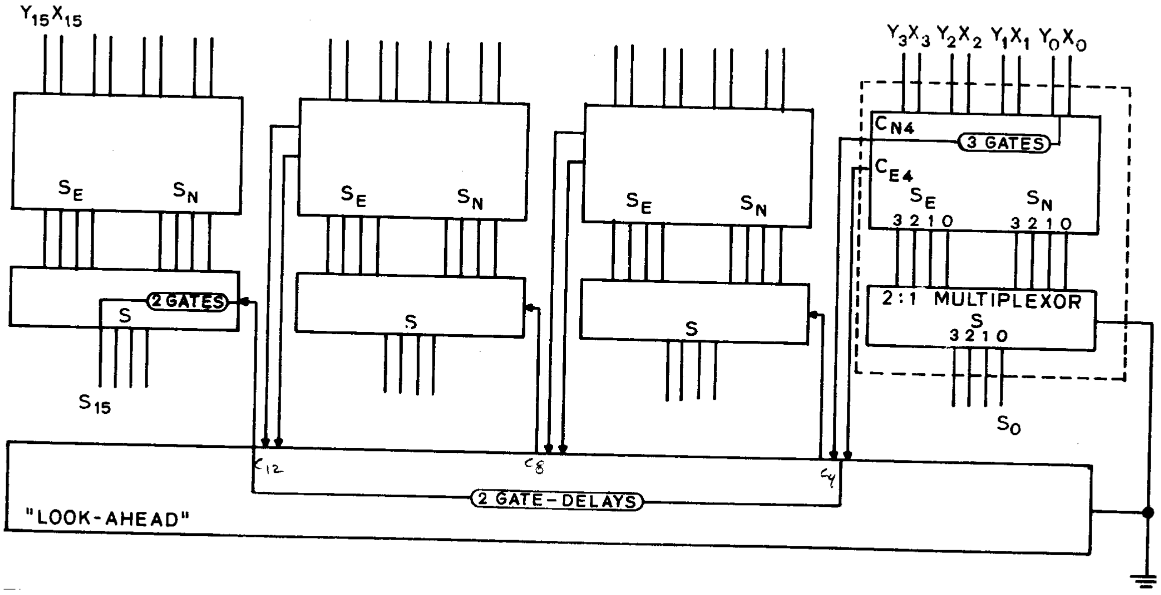


Figure 3.3: 16-bit conditional sum adder. The dotted line encloses a 4-bit slice with internal look ahead. The rectangular box (on the bottom) accepts conditional carries and generates fast true carries between slices. The worst case path delay is seven gates.

be performed on $r-1$ groups per AND–OR gate pair (see preceding equation for C_{16}), with two delays for each pair.

If $r \gg 1$, then $r \simeq r-1$, and if $r \ll n$, then

$$t \simeq 3 + 2\lceil \log_r n \rceil$$

The delay equation (3.1) is correct for $r \geq 4$. For $r = 3$ or $r = 2$ and $n \leq r$, then $t = 7$. The cases $r = 3$ and $r = 2$ where $n > r$ are left as an exercise.

3.1.3 Carry-Look-Ahead Addition

In the last decade, the carry-look-ahead has become the most popular method of addition, due to a simplicity and modularity that make it particularly adaptable to integrated circuit implementation. To see this modularity, we derive the equations for a 4-bit slice.

The sum equations for each bit position are:

$$\left.\begin{array}{rcl} S_0 & = & A_0 \oplus B_0 \oplus C_0 \\ S_1 & = & A_1 \oplus B_1 \oplus C_1 \\ S_2 & = & A_2 \oplus B_2 \oplus C_2 \\ S_3 & = & A_3 \oplus B_3 \oplus C_3 \end{array}\right\} \qquad \begin{array}{l} \text{in general:} \\ S_i = A_i \oplus B_i \oplus C_i \end{array}$$

The carry equations are as follows:

$$\left.\begin{array}{lcl} C_1 & = & A_0B_0 + C_0(A_0 + B_0) \\ C_2 & = & A_1B_1 + C_1(A_1 + B_1) \\ C_3 & = & A_2B_2 + C_2(A_2 + B_2) \\ C_4 & = & A_3B_3 + C_3(A_3 + B_3) \end{array}\right\} \qquad \begin{array}{l} \text{in general:} \\ C_{i+1} = A_iB_i + C_i(A_i + B_i) \end{array}$$

The general equations for the carry can be verbalized as follows: there is a carry into the $(i+1)^{th}$ stage if a carry is generated locally at the i^{th} stage, or if a carry is propagated through the i^{th} stage from the $(i-1)^{th}$ stage. Carry is generated locally if both A_i and B_i are ones, and it is expressed by the generate equation $G_i = A_iB_i$. A carry is propagated only if either A_i or B_i is one, and the equation for the propagate term is $P_i = A_i + B_i$.
We now proceed to derive the carry equations, and show that they are functions only of the previous generate and propagate terms:

$$\begin{array}{lcl} C_1 & = & G_0 + P_0C_0 \\ C_2 & = & G_1 + P_1C_1 \end{array}$$

Substitute C_1 into the C_2 equation (in general, substitute C_i in the C_{i+1} equation):

$$\begin{array}{lcl} C_2 & = & G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 & = & G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 & = & G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{array}$$

We can now generalize the carry-look-ahead equation:

$$C_{i+1} \quad = \quad G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \cdots + P_iP_{i-1}\ldots P_0C_0$$

This equation implies that a carry to any bit position could be computed in two gate delays, if it were not limited by fan-in and modularity; but the fan-in is a serious limitation, since for an n-bit look ahead the required fan-in is n, and modularity requires a somewhat regular implementation structure so that similar parts can be used to build adders of differing operand sizes. This modularity requirement is what distinguishes the CLA algorithm from the canonic algorithm discussed in the next section.

The solution of the fan-in and modularity problems is to have several levels of carry-look-ahead. This concept is illustrated by rewriting the equation for C_4 (assuming fan-in of 4, or 5 if a C_0 term is required):

$$C_4 = \underbrace{G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0}_{\text{Group generate} = G_0'} + \underbrace{P_3P_2P_1P_0}_{\text{Group propagate} = P_0'}\, C_0$$

$$C_4 = G_0' + P_0'C_0$$

Notice the similarity of C_4 in the last equation to C_1. Similarly, the equations for C_5 and C_6 resemble those for C_2 and C_3.

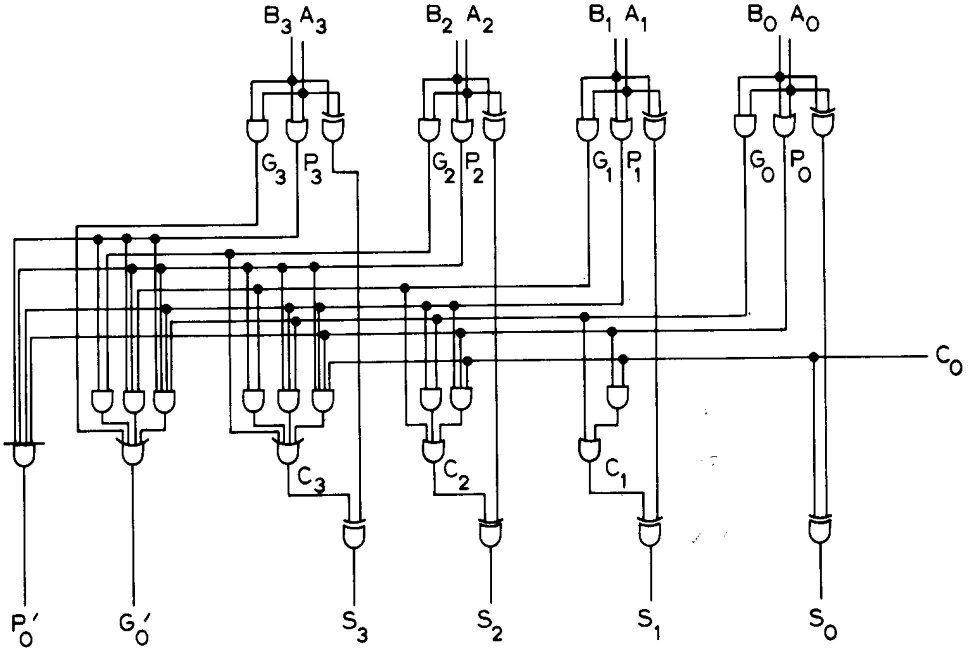


Figure 3.4: 4-bit adder slice with internal carry-look-ahead (gate count = 30).

The CLA level consists of the logic to form fan-in limited generate and progate terms. It requires two gate delays. With a fan-in of 4, two levels of carry-look-ahead (CLA) are sufficient for 16 bit additions. Similarly, CLA of between 17 and 64 bits requires a third level. In general, CLA across 17 to 64 bits requires a second level of carry generator. In general, the number of CLA levels is:

$$\lceil \log_r n \rceil$$

where r is the fan-in, and n is the number of bits to be added.

We now describe the hardware implementation of a carry-look-ahead addition. It is assumed that the fan-in is 4; consequently, the building blocks are 4-bit slices. Two building blocks are necessary. The first one is a 4-bit adder with internal carry-look-ahead across 4 bits, and the second one is 4 group carry generator. Figure 3.4 shows the gate level implementation of the 4-bit CLA adder, according to the equations for S_0 through S_3 and C_1 through C_3.

Figure 3.5 is the gate implementation of the four group CLA generator. The equations for this generator are as follows, where G'_0 and P'_0 are the (0–3) *group* generate and propagate terms (to distinguish them from G_0 and P_0, which are *bit* generate and propagate terms):

$$\begin{aligned}
C_4 &= G'_0 + P'_0 C_0 \\
C_8 &= G'_1 + P'_1 G'_0 + P'_1 P'_0 C_0 \\
C_{12} &= G'_2 + P'_2 G'_1 + P'_2 P'_1 G'_0 + P'_2 P'_1 P'_0 C_0
\end{aligned}$$

and the third level generate (G'') and propagate (P'') terms are:

$$G'' = G'_3 + P'_3 G'_2 + P'_3 P'_2 G'_1 + P'_3 P'_2 P'_1 G'_0$$

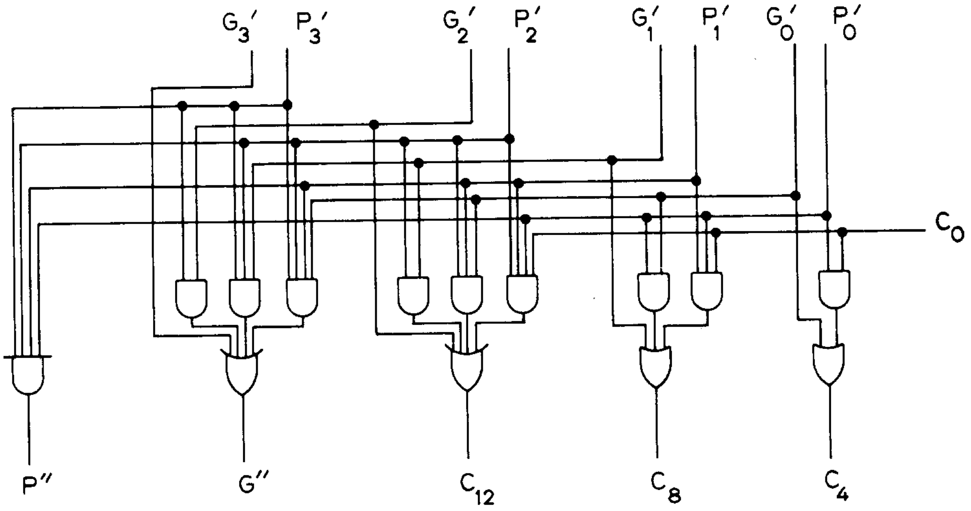


Figure 3.5: Four group carry-look-ahead generator (gate count = 14).

$$P'' = P_3'P_2'P_1'P_0'$$

The G'' and P'' are more completely labeled G_0'' and P_0''. The corresponding third level carrys are:

$$\begin{aligned}
C_{16} &= G_0'' + P_0''C_0 \quad (C_0 \text{ can be end around carry}) \\
C_{32} &= G_1'' + P_1''G_0'' + P_1''P_0''C_0 \\
C_{48} &= G_2'' + P_2''G_1'' + P_2''P_1''G_0'' + P_2''P_1''P_0''C_0 \\
C_{64} &= G_3'' + P_3''G_2'' + P_3''P_2''G_1'' + P_3''P_2''P_1''G_0'' + P_3''P_2''P_1''C_0
\end{aligned}$$

The implementation of a 64-bit addition from these building blocks is shown in Figure 3.6. From this figure, we can derive the general equation for worst case path delay (in gates) as a function of fan-in and number of bits.

The longest path in the 64-bit addition consists of the following delays:

• Initial generate term	per bit	1 gate delay
• Generate term	across 4 bits	2 gate delays
• Generate term	across 16 bits	2 gate delays
• C_{48} generation		2 gate delays
• C_{60} generation		2 gate delays
• C_{63} generation		2 gate delays
• S_{63} generation		1 gate delays
	Total =	12 gate delays

In general, for n-bit addition limited by fan-in of r:

• Generate term	per bit	1 gate delay
• Generate C_n	2×(2(number of CLA levels)−1) gate delays	
• Generate S_n		1 gate delay

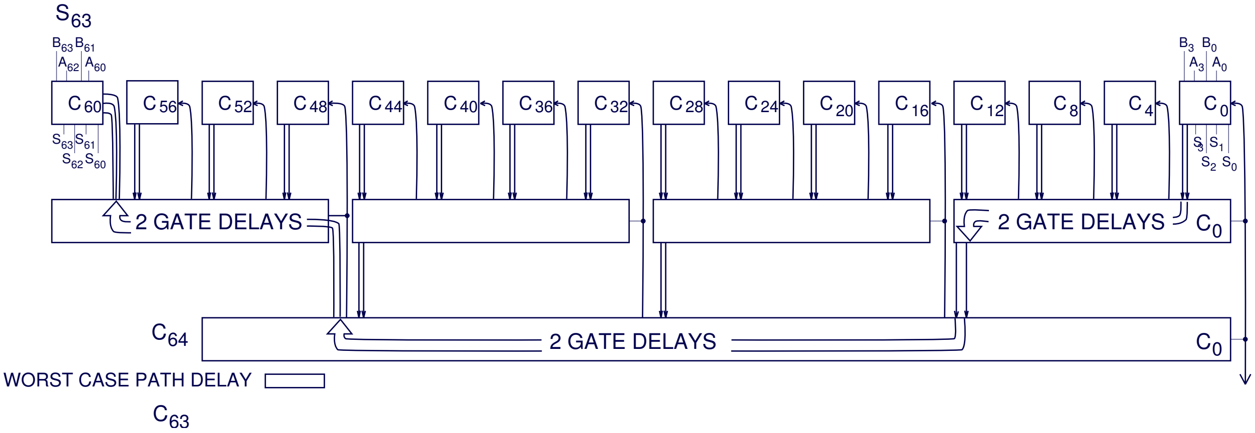


Figure 3.6: 64-bit addition using full carry-look-ahead. The first row is made of a 4-bit adder slice with internal carry-look-ahead (see Figure 3.4). The rest are look ahead carry generators (see Figure 3.5). The worst case path delay is 12 gates (the delay path is strictly for addition).

$$\text{Total CLA gate delays} = 2 + 4 \text{ (number of CLA levels)} - 2$$

$$\text{Total CLA gate delays} = 4 \text{ (number of CLA levels)}.$$

The number of CLA levels is $\lceil \log_r n \rceil$.

$$\boxed{\text{CLA gate delays} = 4\lceil \log_r n \rceil}$$

Before we conclude the discussion on carry-look-ahead, it is interesting to survey the actual integrated circuit implementations of the adder-slice and the carry-look-ahead generator. The TTL 74181 Texas Instruments, Inc. [1976] is a 4-bit slice that can perform addition, subtraction, and several Boolean operations such as AND, OR, XOR, etc. Therefore, it is called an ALU (Arithmetic Logic Unit) slice. The slice depicted in Figure 3.4 is a subset of the 74181. The 74182 Texas Instruments, Inc. [1976] is a four-group carry-look-ahead generator that is very similar in implementation to Figure 3.5. The only difference is in the opposite polarity of the carries, due to an additional buffer on the input carry. (Inspection of Figure 3.6 shows that the Generate and Propagate signals drive only one package, regardless of the number of levels, whereas the carries' driving requirement increases directly with the number of levels.) For more details on integrated circuit implementation of adders, see Waser Waser [1978].

3.1.4 Canonic Addition: Very Fast Addition and Incrementation

So far, we have examined the delay in practical implementation algorithms—conditional sum and CLA—as well as reviewing Winograd's theoretic delay limit. Now Winograd Winograd [1965] shows that his bound of binary addition is achievable using (r, d) circuits with a conditional sum algorithm. The question remaining is what is the fastest known binary addition algorithm using conventional AND–OR circuits (fan-in limited without use of a wired OR).

Before developing such fast adders, called canonic adders, consider the problem of incrementation—simply adding one to X, an n-bit binary number. Winograd's approach would yield a bound on an increment of:

$$\text{Increment } (r,d) \text{ delays} = \lceil \log_r(n+1) \rceil.$$

Such a bound is largely realizable by *AND* circuits, since the longest path delay (the highest order sum bit, S_n) depends simply on the configuration of the old value of X. Thus, if we designate I as the increment function:

$$\begin{array}{cccccc} & X_{n-1} & X_{n-2} & \ldots & X_0 \\ + & & & & I \\ \hline C_n & S_{n-1} & S_{n-2} & \ldots & S_0 \end{array}.$$

Then C_n, the overflow carry, is determined by:

$$C_n = \prod_{i=0}^{n-1} X_i \cdot I \quad \text{(i.e., the AND of all elements in X)},$$

and intermediate carries, C_j:

$$C_j = \prod_{i=0}^{j-1} X_i \cdot I.$$

C_n is implementable as a fan-in limited tree of AND circuits in:

$$C_n \text{ gate delays } = \lceil \log_r(n+1) \rceil.$$

Each output S_j bit in the increment would have an AND tree:

$$\begin{aligned} S_0 &= X_0 \oplus I \\ S_j &= X_j \oplus C_j \end{aligned}$$

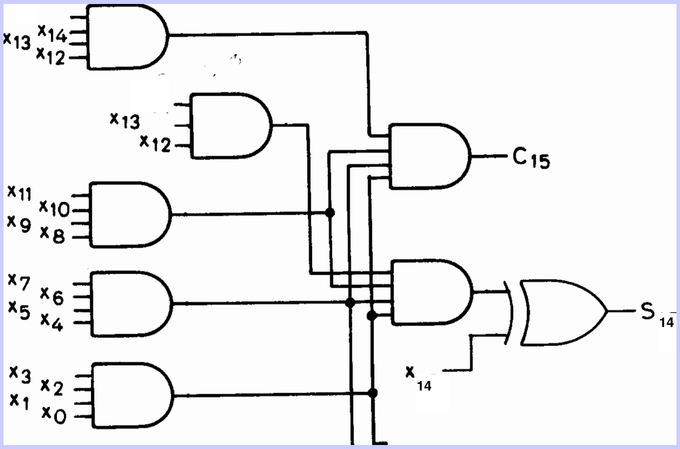
Thus, the delay in realizing S_{n-1} (the n^{th} sum bit) is:

$$\text{Increment gate delays } = \lceil \log_r n \rceil + 1,$$

that is, the gate delays in C$_{n-1}$ plus the final exclusive OR.

Example 3.1

A 15-bit incrementer (bits 0–14) might have the following high order configuration for S_{14} and C_{15}.



The amount of hardware required to implement this approach is not as significant as it first appears. The carry-out circuitry requires:

$$\left\lceil \frac{n}{r} \right\rceil + \left\lceil \frac{n}{r} \cdot \frac{1}{r} \right\rceil + \ldots \text{ gates,}$$

or approximately

$$\left\lceil \frac{n}{r} \right\rceil \left(1 + \frac{1}{r} + \frac{1}{r^2} \cdots \right),$$

where the series consists of only a few terms, as it terminates for the lowest k that satisfies:

$$\left\lfloor \frac{n}{r^k} \right\rfloor = 1.$$

The familiar geometric series $(1 + \frac{1}{r} + \frac{1}{r^2} + \cdots)$ can conservatively be replaced by its infinite sum $\frac{r}{r-1}$. Thus:

$$\text{number of increment gates in } C_n \leq \left\lceil \frac{n}{r} \right\rceil \left(\frac{r}{r-1} \right)$$

and summing over the carry terms and adding the n exclusive ORs for the sums,

$$\text{total increment gates } \leq \sum_{i=1}^{n} \left\lceil \frac{i}{r} \right\rceil \left(\frac{r}{r-1} \right) + n$$

or, ignoring effects of the ceiling,

$$\text{total increment gates } \simeq \frac{n(n+1)}{2(r-1)} + n.$$

Now, most of these gates can be shared by lower order sum terms (fan-out permitting). Thus, for lower order terms (e.g., S_{n-2}):

$$S_{n-2} = (X_{n-2} \cdot X_{n-3} \cdots X_{n-2-r}) \cdot \text{(existing terms from } C_{n-2-r}).$$

Thus, only two additional circuits per lower order sum bit are required. The total number of increment gates is then approximately:

$$\text{increment gates } \simeq \left\lceil \frac{n}{r} \right\rceil + 2(n-1) + n \simeq \left\lceil \frac{n}{r} \right\rceil + 3n,$$

e.g., for $r = 4$ and $n = 32$ the total number of gates is 104 gates.

The same technique can be applied to the problem of n-bit binary addition. Here, in order to add two n-bit binary numbers:

$$\begin{array}{ccccc} & X_{n-1} & X_{n-2} & \ldots & X_0 \\ + & Y_{n-1} & Y_{n-2} & \ldots & Y_0 \\ \hline & S_{n-1} & S_{n-2} & \ldots & S_0 \end{array}$$

We have:

$$\begin{array}{cclccl} C_n & = & G_{n-1} & \textit{designatedas} & C_n & = C_n^{n-1} \\ & & +P_{n-1} \cdot G_{n-2} & & & +C_n^{n-2} \\ & & +P_{n-1} \cdot P_{n-2} \cdot G_{n-3} & & & +C_n^{n-3} \\ & & + & & & + \qquad , \\ & & \vdots & & & \vdots \\ & & +\prod_{i=1}^{n-1} P_i \cdot G_0 & & & +C_n^0 \end{array}$$

and for each sum bit S_j $(n-1 \geq j > 0)$,

$$\begin{array}{rl} & S_j = X_j \oplus Y_j \oplus C_j \\ \textit{and} & S_0 = X_0 \oplus Y_0 \end{array}$$

In the above, $G_i = X_i \cdot Y_i$, $P_i = X_i + Y_i$ and C_n^i designates the term that generates a carry-out of bit i and propagates it to a bit n. This is simply an AND–OR expansion of the required carry—hence the term "canonic addition."

The C_n term consists of an n-way OR, the longest of whose input paths is an n-way AND which generates in bit 0 and propagates elsewhere. Note that since $G_i = X_i \cdot Y_i$, a separate level to form G_i is not required, but each P_i requires an OR level.

Thus, the number of gate delays is $\lceil \log_r n \rceil$ for the AND tree and a similar number for the OR tree, plus one for the initial P_i:

$$\text{Gate delays in } C_n = 2\lceil \log_r n \rceil + 1.$$

The formation of the highest order sum (S_{n-1}) requires the formation of C_{n-1} and a final exclusive OR. Thus,

$$\text{Gate delays in } \mathrm{S}_n = 2\lceil \log_r (n-1) \rceil + 2.$$

Actually, the delay bound can be slightly improved in a number of cases by arranging the inputs to the OR tree so that short paths such as G_{n-1} or $P_{n-1} \cdot G_{n-1}$ are assigned to higher nodal inputs, while long paths such as $\prod_{i=k}^{n-1} P_k \cdot G_k$ $(k = 1, 2, 3 \ldots)$ are assigned to a lower node.

This prioritization of the inputs to the OR tree provides a benefit in a number of cases where the number of inputs n exceeds an integer tree boundary by a limited amount. The AND terms use from one gate delay to $\lceil \log_r n \rceil$ gate delays. If we can site the slow AND terms in fast positions in the OR tree (and there are enough of them!), we can save a gate delay ($\delta = 1$). For example, if $r = 4$ and $n = 7$, we would have

$$C_7 = G_6 + P_6 \cdot G_5 + P_6P_5G_4 + P_6P_5P_4G_3 + P_6P_5P_4P_3G_2 + P_6P_5P_4P_3P_2G_1 + P_6P_5P_4P_3P_2P_1G_0$$

The first four AND terms are generated in one gate delay, while the remaining three terms require two delays ($r = 4$). However, the OR tree consists of seven input terms—four at the second level and three at the root. Thus, the slow AND terms can be accommodated in the three fast (first-level) sites in the OR tree, saving a gate delay.

More generally, the number of long path terms in the AND tree is

$$n - r^{\lceil \log_r n \rceil - 1}.$$

The OR (with $\lceil \log_r n \rceil$ levels) has

$$r^{\lceil \log_r n \rceil - 1},$$

total preferred sites of which

$$\left\lceil \frac{n - r^{\lceil \log_r n \rceil - 1}}{r - 1} \right\rceil$$

have been used for the highest level inputs. Thus,

$$r^{\lceil \log_r n \rceil - 1} - \left\lceil \frac{n - r^{\lceil \log_r n \rceil - 1}}{r - 1} \right\rceil$$

are the number of preferred sites available in the *OR* tree. Now, if the available site equals or exceeds the number of long *AND* paths, we have saved one gate delay:

$$n - r^{\lceil \log_r n \rceil - 1} \leq r^{\lceil \log_r n \rceil - 1} - \lceil \log_r (n - r^{\lceil \log_r n \rceil - 1} \rceil$$

$$n \leq 2r^{\lceil \log_r n \rceil - 1} - \lceil \log_r (n - r^{\lceil \log_r n \rceil - 1}) \rceil$$

Thus, the exact delay is:

$$\boxed{\text{Canonic addition gate delays} = 2\lceil \log_r (n-1) \rceil + 2 - \delta}$$

where δ is the Kronecker δ and is equal to 1 whenever $\lceil \log_r n \rceil > 1$ and the above integer boundary condition is satisfied.

Consider the example $r = 4$ and $n = 20$.

Now $\lceil \log_4 20 \rceil = 3$

and $r^{\lceil \log_r n \rceil - 1} = r^{3-1} = 16$

and $\log_4 (20 - 4^2) = 1$

Since $n = 20 \leq 32 - 1$

then $\delta = 1$

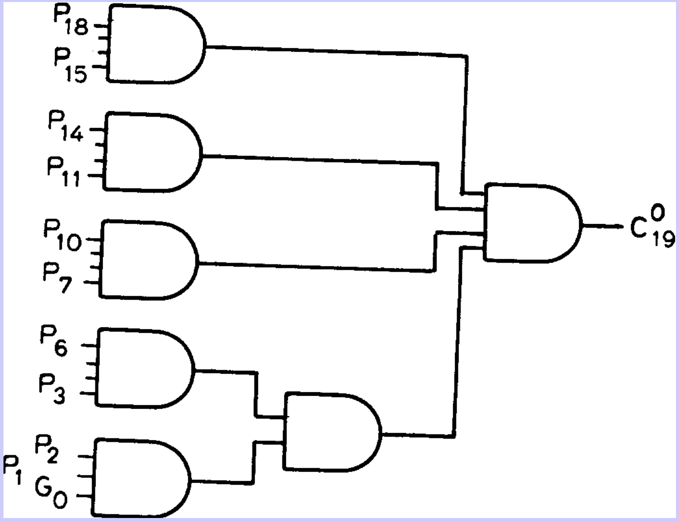
$$\begin{aligned} Gatedelays &= 2\lceil \log_4 19 \rceil + 2 - 1 \\ &= 7 \end{aligned}$$

Whereas

$$\begin{aligned} \text{Winograd's bound} = \lceil \log_4 2 \cdot 20 \rceil &= \lceil \log_4 40 \rceil, \\ &= 3 \end{aligned}$$

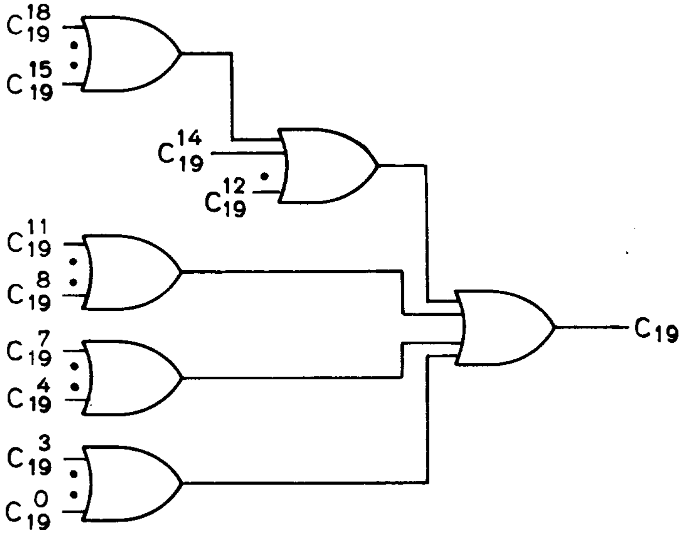
Example 3.2

The AND tree for the *generate* in bit 0 and propagate to bit 19 (C_{19}^{0}) is:



Terms C_{19}^{1}, C_{19}^{2} and C_{19}^{3} (two stages of delay) will have similar structures (i.e., three stages of delay), however, lower stages C_{19}^{i} for i between 4 and 14 have two stages of delay, while C_{19}^{15} through G_{19} have one stage.

Thus, in the OR tree we must insure that terms C_{19}^{0} through C_{19}^{3} are preferentially situated ($\delta = 1$).



The amount of hardware required is not determinable in a straightforward way, especially for the AND networks. For the OR networks, we have:

$$\begin{array}{c} 4bitsat6gates \\ 4bitsat5gates \\ 8bitsat4gates \\ 4bitsat1gate \end{array},$$

or 80 gates total. To this must be added 20 × 2 gates for initial propagate and generate terms. The AND gates required for bit 19 include the six gates in the AND tree used to form C_{19}^0 plus the AND circuits required to form all other C_{19}^i (i from 1 to 18), terms. Since many of the AND network terms have been formed in C_{19}^0, only two additional gates are required for each i in C_{19}^i; one to create an initial term and one to collect all terms. Actually, we ignore a number of cases where only one additional gate is required. Then the C_{19} AND network consists of $6 + 2 \cdot 18 = 42$ gates. So far, we have ignored fan-out limitations, and it is worth noting that many terms are heavily used—up to 20 times. However, careful design using consolidated terms (gates) where appropriate can keep the fan-out down to about 10—probably a practical maximum. Thus, fan-out limits the use of C_{19} terms in C_{18}, etc. But the size of the AND trees decreases for intermediate bits C_j; e.g., for C_9 about 13 gates are required. As a conservative estimate, assume that (1/2)(42) gates are required as an average for the AND networks. The total number of gates is then:

AND networks:	(1/2)(42)(20)	= 420	
OR networks:		= 80	
initial terms:	2 × 20	= 40	
Exclusive ORs:	2 × 20	= 40	
total:		= 580	gates

While 580 gates (closer to 450 with a more detailed count) is high compared to a 20-bit CLA addition, the biggest drawbacks to canonic addition are fan-out and topology, not cost. The high average gate fan-out coupled with relatively congested layout problems leads to an almost three-dimensional communication structure within the AND trees. Both serve to significantly increase average gate delay. Still, canonic addition is an interesting algorithm with practical significance in those cases where at least one operand is limited in size.

3.1.5 Ling Adders

Adders can be developed to recognize the ability of certain circuit families to perform special logic functions very rapidly. The classic case of this is the ability to "DOT" gates together. Here, the output of AND gates (usually) can simply be wired together, giving an OR function. This wired OR or DOT OR has no additional gate delay (although a small additional loading delay is experienced per wired output, due to a line capacitance). Another circuit family feature of interest is complementary outputs: each gate has both the expected (true) output and another complemented output. The widely used current switching (sometimes called emitter coupled or current mode) circuit family incorporates both features. Of course, using the DOT feature may invalidate the premise of the (r, d) circuit model that all logic decisions have unit delay with fan-in r. Ling Ling [1981] has carefully developed adder structures to capitalize on the DOT OR ability of these circuits. By encoding pairs of digit positions $(A_i, B_i, A_{i-1}, B_{i-1})$, Ling redefines our notion of sum and carry. To somewhat oversimplify Ling's approach, we attribute the local (lower neighbor) carry enable terms (P_{i-1}) to the definition of the sum (S_i), leaving a reduced synthetic carry (designated H_{i+1}) for non-local carry propagation $(P_i = A_i + B_i)$. Ling finds that the sum (S_i) at bit i can be written as:

$$\begin{array}{rcl} S_i & = & (H_{i+1} \oplus P_i) + G_i \cdot H_i \cdot P_{i-1} \\ & = & (G_i + P_{i-1} \cdot H_i) \oplus P_i + G_i \cdot H_i \cdot P_{i-1} \end{array},$$

where H_i is defined by the recursion

$$H_{i+1} = G_i + H_i \cdot P_{i-1}.$$

While the combinatorics of the derivation are formidable, the validity of the above can also be seen from the following table. Note that the recursion is equivalent to:

$$H_{i+1} = G_i + C_i$$

compared with

$$C_{i+1} = G_i + P_i C_i.$$

Now

$$\begin{array}{rcl} P_i \cdot H_{i+1} & = & P_i G_i + P_i C_i \\ & = & G_i + P_i C_i \\ & = & C_{i+1} \end{array}$$

or

$$C_i = P_{i-1} \cdot H_i,$$

and

$$H_{i+1} = G_i + C_i = G_i + P_{i-1} H_i.$$

Also, since

$$S_i = A_i \oplus B_i \oplus C_i,$$

then

$$S_i = A_i \oplus B_i \oplus P_{i-1}H_i.$$

Function	Inputs			Outputs	
$f(n)$	A_i	B_i	H_i	S_i	H_{i+1}
0	0	0	0	0	0
1	0	0	1	P_{i-1}	P_{i-1}
2	0	1	0	1	0
3	0	1	1	$\overline{P}_{i-1}$	P_{i-1}
4	1	0	0	1	0
5	1	0	1	$\overline{P}_{i-1}$	P_{i-1}
6	1	1	0	0	1
7	1	1	1	P_{i-1}	1

H_i is conditioned by P_{i-1} in determining the equivalent of C_i. If a term in the table has $H_i = 0$, the equivalent C_i must be zero and the S_i determination can be directly made (as in the cases $f(0), f(2), f(4), f(6)$). Now whenever $H_i = 1$ determines the sum outcome, the P_{i-1} dependency must be introduced. For $f(1)$ and $f(7)$ the $S_i = 1$ if $P_{i-1} = 1$; for $f(3)$ and $f(5)$, the $S_i = 0$ if $P_{i-1} = 1$ (i.e., $f(3)$ and $f(5)$ are conditioned by $\overline{P}_{i-1}$). A direct expansion of the minterms of

$$S_i = (G_i + P_{i-1} \cdot H_i) \oplus P_i + G_i \cdot H_i \cdot P_{i-1}$$

produces the S_i output in the above table:

$$S_i = P_{i-1} \cdot (f(1) + f(7)) + \overline{P}_{i-1} \cdot (f(3) + f(5)) + f(2) + f(4).$$

The synthetic carry H_{i+1} has similar dependency on P_{i-1}; for $f(3)$ and $f(5)$, $H_{i+1} = 1$ occurs if $P_{i-1} = 1$. For $f(6)$ and $f(7)$ $H_{i+1} = 1$ regardless of the $H_i \cdot P_{i-1}$, since $G_i = 1$. The $f(1)$ term is an interesting "don't care" term introduced to simplify the H_{i+1} structure. This $f(4)$ term in H_i does not affect S_i, since S_i depends on H_i. Now S_{i+1} cannot be affected by $H_{i+1}\,(f(1))$, since $P_i\,(f(1)) = 0$. Similarly H_{i+2} also contains the term $H_{i+1}P_i$, which for $f(1)$ is zero by action of P_i.

To understand the advantage of the Ling adder, consider the conventional C_4 (carry-out of bit 3), as contrasted with H_4:

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0,$$

$$H_4 = G_3 + P_2G_2 + P_2P_1G_1 + P_2P_1P_0G_0.$$

Without the DOT function C_4 is implementable ($r = 4$) in three gate delays (two shown, plus one for either P or G). C_4 can be expanded in terms of the input arguments:

$$\begin{aligned} C_4 &= A_3B_3 + (A_3 + B_3)A_2B_2 + (A_3 + B_3)(A_2 + B_2)A_1B_1 \\ &\quad +(A_3 + B_3)(A_2 + B_2)(A_1 + B_1)A_0B_0 \end{aligned}$$

$$\begin{aligned} C_4 \quad = \quad & A_3B_3 + A_3A_2B_2 + B_3A_2B_2 + A_3A_2A_1B_1 \\ & +A_3B_2A_1B_1 + B_3A_2A_1B_1 + B_3B_2A_1B_1 \\ & +A_3A_2A_1A_0B_0 + A_3A_2B_1A_0B_0 + A_3B_2A_1A_0B_0 \\ & +A_3B_2B_1A_0B_0 + B_3A_2A_1A_0B_0 + B_3A_2B_1A_0B_0 \\ & +B_3B_2A_1A_0B_0 + B_3B_2B_1A_0B_0 \end{aligned}$$

If we designate s as the maximum number of lines that can be dotted, then we see that to perform C_4 in one dotted gate delay requires $r = 5$ and $s = 15$.

Now consider the expansion of H_4:

$$\begin{aligned} H_4 \quad = \quad & A_3B_3 + (A_2 + B_2)A_2B_2 + (A_2 + B_2)(A_1 + B_1)A_1B_1 \\ & +(A_2 + B_2)(A_1 + B_1)(A_0 + B_0)A_0B_0 \end{aligned}$$

$$\begin{aligned} H_4 \quad = \quad & A_3B_3 + A_2B_2 + A_2A_1B_1 + B_2A_1B_1 \\ & +\mathrm{A_2A_1A_0B_0} + \mathrm{A_2B_1A_0B_0} \\ & +\mathrm{B_2A_1A_0B_0} + \mathrm{B_2B_1A_0B_0} \end{aligned}$$

Thus, the Ling structure provides one dotted gate delay with $r = 4$ and $s = 8$.

Higher order H look-ahead can be derived in a similar fashion by defining a fan-in limited I term as the conjunction of P_is; e.g.,

$$I_7 = P_6P_5P_4P_3.$$

Rather than dot ORing the summands to form the P_i term, the bipolar nature of the ECL circuit can be used to form the OR in one gate delay:

$$P_i = A_i + B_i$$

and the P_i terms can be dot ANDed to form the I terms. Thus, I_7, I_{11}, and I_{15} can be found with one gate delay.

Suppose we designate the pseudo-carryout of each four bit group as H'_{16}, H'_{12}, H'_8, H'_4, and the group carry-generate as G_4, G_8, G_{12}. Then

$$\begin{aligned} H_4 &= H'_4 \\ H_8 &= H'_8 + I_7H'_4 \\ H_{12} &= H'_{12} + I_{11}H'_8 + I_{11}I_7H'_4 \\ H_{16} &= H'_{16} + I_{15}H'_{12} + I_{15}I_{11}H'_8 + I_{15}I_{11}I_7H'_4. \end{aligned}$$

Of course,

$$\begin{aligned} C_{16} &= P_{15}H_{16} \\ &= P_{15}(H'_{16} + I_{15}H'_{12} + I_{15}I_{11}H'_8 + I_{15}I_{11}I_7H'_4), \end{aligned}$$

	Fixed Radix (Binary)				
	Winograd's lower bound	Conditional sum	Carry-look-ahead	Canonic	Ling
Formula	$\lceil \log_r 2n \rceil$	$5 + 2\left\lceil \log_{r-1}\left(\lceil \frac{n}{r} \rceil - 1\right)\right\rceil$	$4\lceil \log_r n \rceil$	$2\lceil \log_r (n-1) \rceil + 2 - \delta$	$\lceil \log_r \frac{n}{2} \rceil + 1$
gate delays $n = 64$ bits $r =$ fan-in $= 5$	4	9	12	8	4^*

	Variable Radix (Residue)	
	Winograd's lower bound	ROM look-up table
Formula	$\lceil \log_r 2\lceil \log_d \alpha(n) \rceil \rceil$	$2 + \lceil \log_r m \rceil + \lceil \log_r 2^m \rceil$
gate delays n = 64 bits $r =$ fan-in $= 5$	$d = 2,$	$\alpha(> 2^n) = 59, \quad m = \lceil \log_d \alpha(> 2^n) \rceil = 6$
	2	7

* The Ling adder requires dot OR of 16 terms and assumes no additional delay for such dotting.

Table 3.1: Comparison of addition speed (in gate delay) of the various hardware realizations and the lower bounds of Winograd.

since terms such as

$$I_7 H_4' = P_6 P_5 P_4 P_3 H_4 = P_6 P_5 P_4 C_4.$$

Thus,

$$I_{15} I_{11} I_7 H_4' = \prod_{i=4}^{14} P_i \cdot C_4, \qquad G_4' = C_4.$$

Similarly,

$$\begin{aligned} I_{15} I_{11} H_8' &= I_{15} P_{10} P_9 P_8 P_7 H_8' \\ &= I_{15} P_{10} P_9 P_8 G_8' \\ &= \prod_{i=8}^{14} P_i \cdot G_8'. \end{aligned}$$

Ling suggests that the conditional sum algorithm be used in forming the final result. Thus, S_{31} through S_{16} is found for both $C_{16} = 0$ and $C_{16} = 1$; these results are gated with the appropriate true value of C_{16} and dot ORed in one gate delay. The "extra" delay forming C_{16} from $P_{15} H_{16}$ adds no delay, since P_{15} is ANDed with the sum selector MUX function as below:

$$\begin{aligned} S &= S_E C_{16} + S_N \overline{C_{16}} \\ S &= S_E P_{15} H_{16} + S_N (\overline{P_{15}} + \overline{H_{16}}) \\ &= S_E P_{15} H_{16} + S_N \overline{P_{15}} + S_N \overline{H_{16}}, \end{aligned}$$

where S_E and S_N represent the higher order 16-bit sum with and without carry-in. (ECL circuits have complementary outputs; $\overline{H_{16}}$ is always available.) Thus, the Ling adder can realize a sum delay in:

$$\boxed{\text{Ling gate delays} = \lceil \log_r \tfrac{n}{2} \rceil + 1} \qquad ,$$

so long as the gates can be dotted with capability $2^{r-1} \leq s$.

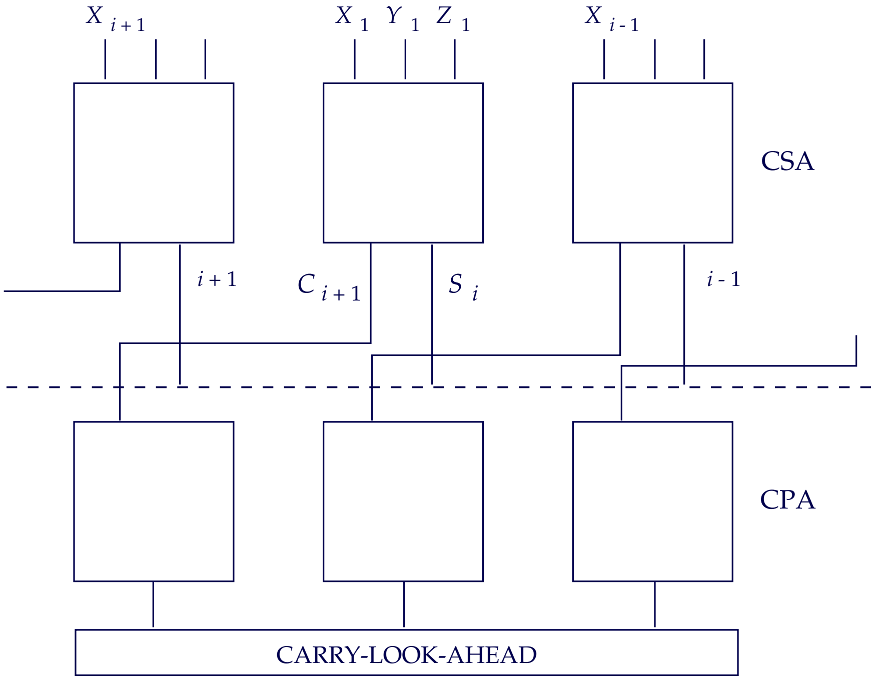


Figure 3.7: Addition of three n-bit numbers.

3.1.6 Simultaneous Addition of Multiple Operands: Carry-Save Adders.

Frequently, more than two operands (positive or negative) are to be summed in the minimum time. In fact, this is the basic requirement of multiplication. Clearly, one can do better than simply summing a pair and then adding each additional operand to the previous sum. Consider the following decimal example:

Carry-	176	
Saving	324	
Addition	$\underline{558}$	
Carry-	948	Column sum
Propagating	$\underline{11\ }$	Column carry
Addition	1058	Total

Regardless of the number of entries to be summed, summation can proceed simultaneously on all columns generating a pair of numbers: column sum and column carry. These numbers must be added with carry propagation. Thus, it should be possible to reduce the addition of any number of operands to a *carry-propagating addition* of only two: Column sum and Column carry. Of course, the generation of these two column operands may take some time, but this should be significantly less than the serial operand by operand propagating addition.

Consider the addition of three n-bit binary numbers. We refer to the structure that sums a column as a *carry-save adder* (CSA). That is, the CSA will take 3 bits of the same significance and produce the sum (same significance) and the carry (1 bit higher significance). Note that this is exactly what a 1-bit position of a *binary full adder* does; but the input connections are different between CSA and binary full adder. Suppose we wish to add X, Y, and Z. Let X_i, Y_i, and Z_i represent the i^{th}-bit position.

We thus have the desired structure: the binary-full adder. However, instead of chaining the carry-out signal from a lower order position to the carry-in input, the third operand is introduced to the "carry-in" and the output produces two operands which now must be summed by a propagating adder. Binary full adders when used in this way are called carry-save adders (CSA). Thus, to add three numbers we require only two additional gate delays (the CSA delay) in excess of the carry propagate adder delay.

The same technique can be extended to more than three operand addition by cascading CSAs.

Suppose we wish to add W, X, Y, Z; the i^{th}-bit position might be implemented as in Figure 3.8. High-speed multiplication depends on rapid addition of multiples of the multiplicand and, as we shall see in the next chapter, uses a generalization of the carry-save adder technique.

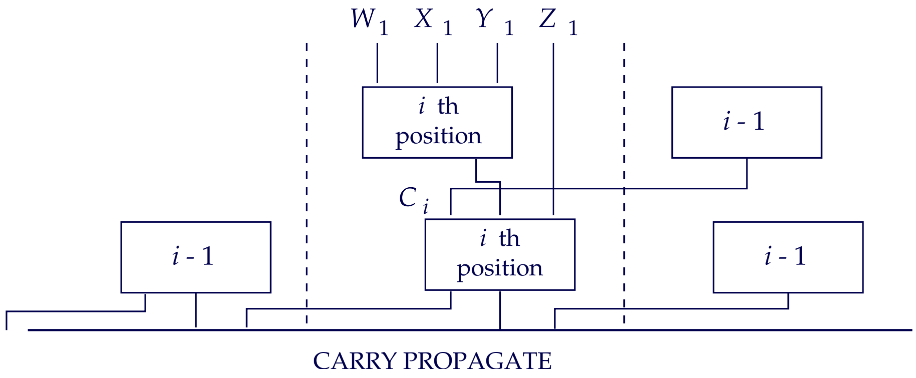


Figure 3.8: Addition of four n-bit numbers.

3.2 Problems

Problem 3.1 What is the gate delay of a 24-bit adder for the following implementations ($r = 4$)?

1. CLA.

2. Conditional sum.

3. Canonic adder.

Problem 3.2 In this problem we attempt to estimate the delays of various adders.

1. Suppose $r = 4$ and the maximum dot-OR capability is also 4; for a 64-bit addition the Ling adder will require how many unit delays, while a canonic adder requires how many unit delays?

2. If six 32-bit operands are to be added simultaneously, how many unit delays are required in CSAs before two operands can be added in a CPA?

3. In a certain machine, the execution time for floating point addition is greater than that for floating point multiplication. Explain (i.e., state the delay conditions which lead to this situation).

Problem 3.3 The System 370 effective address computation involves the addition of three unsigned numbers, two of 24 bits and one of 12 low order bits.

1. Design a fast adder for this address computation (an overflow is an invalid address).

2. Extend your adder to accommodate a fast comparison of the effective address with a 24^{bit} upper bound address.

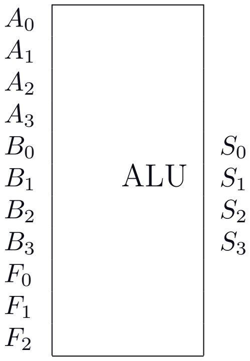
Problem 3.4 Design a circuit that can be connected to a 4-bit ALU to detect 2's complement arithmetic overflow. The ALU takes two 4-bit input operands and provides a 4-bit output result defined by three function bits. The ALU can perform eight functions, defined by $F_2F_1F_0$. The object is to make the circuit as fast as possible. Use as many gates as you like.

For gate timing, use the following:

NAND, NOR, NOT:	5 units
OR, AND:	7 units
XOR:	10 units

Note: Assume delay through ALU $\gg$ single gate delay.

F_2	F_1	F_0	Function
0	0	0	0
0	0	1	$B - A$
0	1	0	$A - B$
0	1	1	$A + B$
1	0	0	$A \oplus B$
1	0	1	A OR B
1	1	0	A AND B
1	1	1	-1



Problem 3.5 In graphics, if the colors representing two pixels are added the operation should never overflow. Rather, if the sum exceeds the maximum, it saturates at the maximum value. So, for example, in such an eight bit adder the result of $254 + 3$ is 255 and not 1 as in conventional adders. Similarly, when the colors are subtracted the result is never negative but saturates at zero. Given two eight bit unsigned integers (i.e. from 0 to 255), design a saturating adder/subtractor using ripple carry and give an estimate of its time delay. (Hint: you may need the results of problem 1.6.)

Problem 3.6 The specific instructions serving multimedia applications in recent processors put new requirements on arithmetic blocks. The new system in your company requires a dynamically partitioned 64 bit adder. Such an adder can operate as a single 64 bit adder, two parallel 32 bit adders, four parallel 16 bit adders, or eight parallel 8 bit adders.

Assume that an 8 bit saturating adder/subtractor such as the one described in problem 3.5 is available as a building block for you. The control lines specifying the operation are labeled a_{64}, a_{32}, a_{16}, and a_8 for the above mentioned cases respectively. The control lines are active high and only one of them is selected at a time. Show how to connect the basic components with any additional logic gates needed to form the partitioned adder. If you need, you can change the design of the previous problem to suit the current one.

Problem 3.7 Provide the Verilog code to simulate a regular 8 bit adder then extend it to the saturating 8 bit adder/subtractor of problem 3.5 and test both *exhaustively*. You should make sure that you test the input and output carry signals as well.

1. Integrate your code to simulate the full partitioned 64 bit adder (problem 3.6) assuming that the carries will ripple between the 8 bit adder blocks. Make sure that you test for correct functionality at all the different modes of operation: 1×64, 2×32, 4×16, and 8×8.

   How many test vectors are needed to test the whole adder exhaustively? If the verification of each test vector takes only $1ns = 10^{-9}$ seconds, what is the time required for an exhaustive test?

2. Instead of rippling the carries within and between the blocks, select a faster scheme from what we studied to redesign the individual blocks and the full 64 bit adder. Provide your design on paper and give an estimate of its gate delays as well as the number of gates used?

   Code your design and simulate it. Verify the time delay and gate count that you estimated on paper. Does it match? If not, why?

Problem 3.8 A friend of yours invented a new way to produce the carry signals in an adder adding two numbers $A = \sum_{i=0}^{n-1} a_i 2^i$ and $B = \sum_{i=0}^{n-1} b_i 2^i$. Instead of the traditional equation

$$c_{i+1} = g_i + p_i c_i \tag{3.2}$$

where $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$ the invention uses a multiplexer to produce c_{i+1} based on p_i as

$$c_{i+1} = g'_i \bar{p}_i + p_i c_i \tag{3.3}$$

with $g'_i = b_i$ and $p_i = a_i \oplus b_i$.

1. Verify that your friend's claim produces the correct carries.
2. Derive a scheme similar to carry-lookahead based on your friend's claim. Group each four bits together and show a block diagram of how a 16 bit adder may use multi-levels of lookahead.
3. Is your friend's scheme better than the traditional scheme? Why or why not?

Problem 3.9 Design a fast two-operands parallel adder where each operand is 16 BCD digits (64 bits) using any of the techniques studied for binary adders. Give an estimate of its time delay.

Problem 3.10 Code your design for the 16 digits BCD adder of problem 3.9 in Verilog and simulate it. Verify the time delay and gate count that you estimated on paper. Does it match? If not, why?

Problem 3.11 The decrementation process is widely needed in digital circuits. It is not necessary to use a complete adder for its implementation. In fact, it is enough to complement from the least significant bit up to and including the least significant 1 bit as in the following example.

$$\begin{array}{rccccccc}
 & & & \downarrow & & & & \\
48 = & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
47 = & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
\cline{4-8}
 & & & \multicolumn{5}{c}{\text{complemented}}
\end{array}$$

1. Derive the necessary logic equations to implement this function on the four bits number $a_3a_2a_1a_0$ and draw the corresponding gate diagram.
2. Derive a scheme similar to carry-lookahead to implement the decrementation on numbers larger than four bits. Draw the diagram for a 16 bits decrementer using your scheme and any number of the four bits decrementer as sub-blocks.

Problem 3.12 The use of modular arithmetic is essential in security algorithms. A simple method to implement $(x+y)\mathbf{mod}_\mu$ where x and y are represented in n bits each is to calculate $z_1 = (x+y)\mathbf{mod_{2^n}}$ using a normal adder. The carry out of that adder is c_1. Then a second n-bit adder adds z_1 and $2^n - \mu$ giving z_2 and a carry out of c_2. Finally, a multiplexer gives the result as z_2 if either c_1 or c_2 is 1, otherwise the result is z_1.

1. Prove that this algorithm gives the correct value of $(x+y)\mathbf{mod}_\mu$.

2. Draw the block diagram of a circuit implementing a modulo subtractor $(x-y)\mathbf{mod}_{\mu}$ using a similar technique.

3. If the adders use ripple carry addition, estimate the delay of both the modulo adder and the modulo subtractor.

Chapter 4

Go forth and multiply (Incomplete chapter)

Usually in both integer and floating point calculations, the multiplication is the second most frequent arithmetic operation after addition. It is quite important to understand it and to perform it efficiently.

For integers, the multiplication is defined by

$$P = \underbrace{X + X + \cdots + X}_{Y \text{ times}}$$

where X is the *multiplicand*, Y is the *multiplier*, and P is the *product*. As we have discussed in chapter 5, in floating point multiplication we multiply the significands as if they were fixed point integers and add the exponents. So a floating point multiplication reduces to an integer multiplication. For multiplication, it is easier to deal with integers in signed and magnitude form. The magnitudes are multiplied as unsigned numbers and the sign of the product is decided separately.

In this chapter, we will explore the different implementation possibilities given the constraints on the time and resources (area and power) allowed for this operation. We will start by speaking about unsigned integer multiplication and then show how to deal with signed integers (including those in two's complement notation). A discussion of the details of floating point multiplication then follows.

4.1 Simple multiplication methods

Conceptually, the simplest method for multiplication is just to apply the definition.

Algorithm 4.1 Loop on Y

```
product = 0; while(Y>0){product = product + X; Y=Y-1; }
```

This is almost the first method a child learns in order to multiply. A software implementation of multiplication with this method uses the equivalent instructions to those in the algorithm. A hardware implementation uses three registers: one to hold the values of X, the second holds the current value of the product, and the third holds the current value of Y. Those registers are the memory elements needed. The combinational elements needed are:

1. an adder to perform `product = product+X`,
2. a decrementing circuit to get `Y=Y-1`; this may reuse the previous adder if the area is constrained or work in parallel with the adder,
3. a comparator to detect `Y=0`; a simple method is to group all the bits of Y in a tree implementing their *NOR* function.

By analyzing this first algorithm, we notice that it is sequential in nature. The new values of the product and Y depend on the previous values. The comparison with zero is the signal to continue the loop or exit from it. If we use a separate circuit to decrement Y, then each cycle in the loop has the time delay of an adder and a comparator. When both X and Y are n bits long, their product is $2n$ bits long which means that the adder has a time delay of $\mathcal{O}(\log 2n)$ or larger depending on the type of adder used. The *NOR* tree of the comparator has a time delay of $\mathcal{O}(\log n)$. The loop continues as long as Y is not zero which means that the total number of cycles is not known a priori. The total time for this multiplication hardware will depend on the specific values of the inputs. If the following processing on the product is capable of starting once the multiplication is completed then this *variable latency* of the multiplier will not constitute a problem. Otherwise, the unit waiting for the product of the multiplier must wait for the worst case condition which is when $Y = 2^n - 1$. If we assume that the adder and the *NOR* tree work in parallel such that their delays do not add, in the worst case the total time delay is

$$t = \mathcal{O}\left((2^n - 1) \times (\log_r(2n))\right).$$

This simple analysis indicates that this first algorithm

1. is slow and
2. has a variable latency depending on the values of the inputs.

On the positive side, this "loop on Y" method is quite simple to implement and uses very little hardware. The hardware requirements decrease if the adder is reused to decrement Y but this doubles the number of cycles needed to finish the computation. If the original value of Y is needed then another register must be used to hold it.

$\overset{?}{\Longrightarrow}$ **Exercise 4.1 A possible alternative method is to use**

```
product = 0; count = 0; while(count<Y){product = product + X;
                             count=count+1; }
```

Will it work? Is this better than algorithm 4.1?

To improve the multiplication time, the *add and shift* method examines the digits of Y. The add and shift algorithm is the basis for all of the subsequent implementations. It is a simplified version of the way one multiplies in the *pencil and paper* method. Humans multiply each digit of the multiplier by the multiplicand and write down on paper (i.e. save for later use) the resulting *partial product*. When we generate a new partial product we write it down shifted relative to the previous partial product by one location to the left. At the end, we add all the partial products.

Example 4.1 Using the pencil and paper method, multiply 6 by 5 in their binary representations
Solution: Obvisouly the answer is 30 but we should see how the detailed work is done.

Multiplicand	110	(6)
Multiplier	$\times$101	$\times$ (5)
	110	(6×2^0)
Partial products	000	(0×2^1)
	110	(6×2^2)
Final product	11110	(30)

In binary, the digit value is either 1 or 0 hence the multiplication by this digit yields a partial product equal to the multiplicand or to a zero respectively.

$\overset{?}{\Longrightarrow}$ **Exercise 4.2 What kind of logic gates are needed to generate a partial product (PP) in binary multiplication?**

As we will see in this chapter, once the digit values go beyond zero and one, the generation of PPs is more difficult.

In the simplified add and shift version, once we generate one partial product we add it directly to the sum of the previous partial products. To maintain the shift of one, we either shift the new partial product by one location to the left or shift the previous sum by one location to the right. Both possibilities are equivalent from a mathematical point of view. The one that is easier to implement is favored. In both cases, the number of bits in the sum of partial products grows as the algorithm proceeds. The register holding the value of that sum must be wide enough for the final $2n$ bits result. At the start however, only n bits are needed to hold the sum value. Without a "growing register" we must have wide register throughout the whole process.

Fig. 4.1 shows a clever idea where we actually provide such a "growing register" for the sum. Each cycle we check the *LSB* of Y and then shift Y to the right to discard that bit and bring the next one in its place. This process frees the *MSB* of Y which we can use to shift in the *LSB* of the sum. Hence, a $2n$ bits wide register is used to initially store Y in its lower n bits and zeros in the upper n bits. After checking the *LSB* of Y, we either add X or 0 to the upper half of the register. We store the result of the addition to the upper half. Then, we shift the *whole* register to the right bringing in the next significant bit of Y to be checked. In each cycle, the barrier between the current sum of products (P) and Y moves to the right by one location and that is why we show it as a dotted line.

Algorithm 4.2 Add and shift

```
1. If the LSB of $Y$ ($y_0$) is 1 add $X$. Otherwise, add zero.

2. Shift both the product and $Y$ to the right one bit.

3. Repeat for the $n$ bits of $Y$.
```

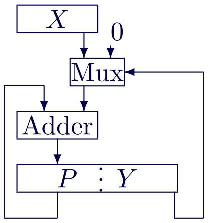


Figure 4.1: A simple implementation of the add and shift multiplication.

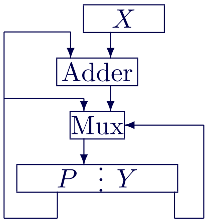


Figure 4.2: A variation of the add and shift multiplication.

In contrast to algorithm 4.1, the add and shift method of algorithm 4.2 has a fixed number of cycles. Regardless of the value of Y, the multiplication has n steps. Fig. 4.1 contains a multiplexer to choose the addition of X or a value of zero. As discussed in chapter 2, the time delay of such a multiplexer from the select to the output is $\mathcal{O}\left(\log_4(n)\right)$. The adder used here is only n bits wide since X is added to the most significant n bits of P. A conservative estimate of the total time delay for algorithm 4.2 is

$$t = \mathcal{O}\left(n \times \left(\log_r(n) + \log_4 n\right)\right).$$

Compared to algorithm 4.1, algorithm 4.2 has a shorter execution time and uses less area since it has a smaller adder. Algorithm 4.2 seems to be a better choice overall. Indeed it is. However, a careful analysis is always due whenever a designer finds a “better” solution.

$\overset{?}{\Longrightarrow}$ **Exercise 4.3 In algorithm 4.1, there was a need to use a comparator to compare Y to zero. Fig. 4.1 does not contain a comparator. Do we still need one?**

The main advantage of the add and shift method is the execution of the multiplication in a much shorter time and with a fixed number of cycles. Fig. 4.2 shows a variation of the add and shift method. This variation exchanges the location of the multiplexer and the adder. With this variation, it is perhaps clearer that when the *LSB* of Y is zero we are doing “nothing”. In Fig. 4.1, when the *LSB* of Y is zero we add zero to P while in the variation of Fig. 4.2 we skip the addition and just put P back into the register. Hence we get a slightly modified algorithm.

Algorithm 4.3 (Add or skip) and shift

```
1. If the LSB of $Y$ ($y_0$) is 1 add $X$. Otherwise, skip the addition.
2. Shift both the product and $Y$ to the right one bit.
3. Repeat for the $n$ bits of $Y$.
```

$\overset{?}{\Longrightarrow}$ **Exercise 4.4 In algorithm 4.2 (Fig. 4.1), while doing "nothing" we are still consuming power by using the adder and the multiplexer. Does the implementation of algorithm 4.3 as in Fig. 4.2 reduce this power consumption?**

It is instructive to think about the effect of algorithm 4.3 on the total time of execution. On some cycles, both the adder and the multiplexer are used while on others only the multiplexer. Can we then have a "faster" cycle in those latter cases to improve the total time? The answer lies in the kind of clocking scheme used in the design. If it is a synchronous design then the clock period is fixed and all the cycles last the same time. In an asynchronous design where a unit acts once it gets a signal from the preceding unit, faster cycles are a possibility. In such an asynchronous design, the average time of execution is thus lower in the case of skipping the addition. The worst case (when all the bits of Y are equal to 1) leads to the same total time of execution for both the synchronous and asynchronous multiplier designs. This worst case is what matters if the multiplier is used in a larger synchronous system. Hence, to really profit from variable latency algorithms, the surrounding system must tolerate this variance. A designer should have a look at the "bigger picture" before spending time in optimizing the design beyond what is useful.

$\overset{?}{\Longrightarrow}$ **Exercise 4.5 If it proves to be fruitful to pursue such a design where some cycles are faster, how do we calculate the average execution time?**

It seems that if Y has more zeros, the multiplier skips more additions and might be made faster and use less power. Booth Booth [1951] proposed an algorithm based on the fact that

$$\begin{array}{ll} \text{a string of ones} & \cdots 011 \cdots 110 \cdots \\ \text{is equal to} & \cdots 100 \cdots 0\bar{1}0 \cdots . \end{array}$$

Hence, instead of adding repeatedly the multiplier adds only twice one for the number and the other for its complement at the correctly shifted positions. The recoding is simple:

1. On a transition from 0 to 1, put $\bar{1}$ at the location of the 1.
2. On a transition from 1 to 0, put 1 instead of the 0.
3. Put zeros at all the remaining locations. (i.e. skip groups of zeros and groups of ones.)

It is possible in an implementation to actually perform this step of recoding followed by another step using a modified add and shift algorithm. Otherwise, it is possible to combine both steps as in the following algorithm. We start first by assuming that Y is an unsigned integer. The case of a signed integer will be treated in the following discussion.

Algorithm 4.4 Simple Booth for an unsigned Y

```
Initially, assume $y_{-1} = 0$.
If $((y_0 = 1)\&(y_{-1} = 0))$ add the two's complement of $X$.
else if $((y_0 = 0)\&(y_{-1} = 1))$ add $X$.
else do not add anything (do nothing or skip).
Shift both the product and $Y$ to the right one bit letting the
current value of $y_0$ go into $y_{-1}$.
Repeat for the $n$ bits of $Y$.
If $(y_{-1} = 1)$ add $X$.
```

The last step actually checks on the *MSB* of the original Y. If that *MSB* is 1 it is the end of a string of ones and we must add X.

In the early days of computers when the availability of a hardware multiplier was rare, programmers used this simple Booth algorithm to implement the multiplication. Two points are worth mentioning given this background:

- The Booth algorithm has a variable latency but, on average, its use reduces the time delay and hence it was attractive to software implementations.
- The worst case is $(01010101 \cdots = 1\bar{1}1\bar{1}1\bar{1}1\bar{1}) \Rightarrow \mathcal{O}(n)$ delay. In this case, the original bits have more zeros than the recoded bits. A modification to the algorithm that prevents the recoding of 'isolated' occurances of a one or a zero avoids this worst case.

$\overset{?}{\Longrightarrow}$ **Exercise 4.6 Modify the simple Booth algorithm to prevent the recoding of isolated ones or zeros as mentioned above.**

In this chapter, the hardware implementations of parallel multipliers are described.

Figure 4.3a illustrates the concept for multiplication of two 8-bit operands, and Figure 4.3b introduces a convenient dot representation of the same multiplication. In this chapter, we will describe the three major categories of parallel multiplier implementation:

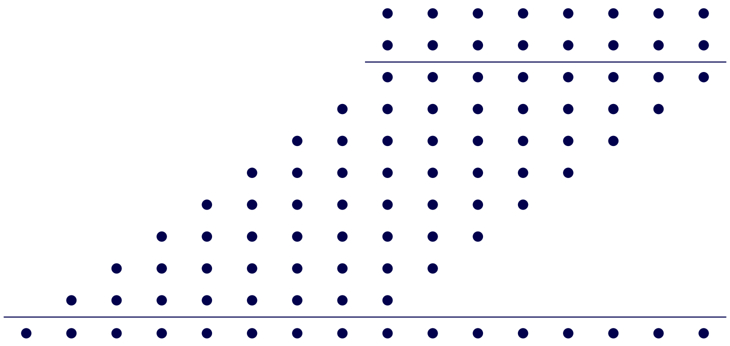
- Simultaneous generation of partial products and simultaneous reduction.
- Simultaneous generation of partial products and iterative reduction.
- Iterative arrays of cells.

4.2 Simultaneous Matrix Generation and Reduction

This scheme is made of two distinct steps. In the first step, the partial products are generated simultaneously, and in the second step, the resultant matrix is reduced to the final product.

$$
\begin{array}{rrrrrrrrrrrrrrrrl}
 & & & & & & & & X7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & \leftarrow \text{MULTIPLICAND} \\
 & & & & & & & & Y7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0 & \leftarrow \text{MULTIPLIER} \\
\cline{9-16}
 & & & & & & & & A7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 & \leftarrow \text{A PARTIAL PRODUCT} \\
 & & & & & & & B7 & B_6 & B_5 & B_4 & B_3 & B_2 & B_1 & B_0 & & \\
 & & & & & & C7 & C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 & & & \\
 & & & & & D7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 & & & & \\
 & & & & E7 & E_6 & E_5 & E_4 & E_3 & E_2 & E_1 & E_0 & & & & & \\
 & & & F7 & F_6 & F_5 & F_4 & F_3 & F_2 & F_1 & F_0 & & & & & & \\
 & & G7 & G_6 & G_5 & G_4 & G_3 & G_2 & G_1 & G_0 & & & & & & & \\
 & H7 & H_6 & H_5 & H_4 & H_3 & H_2 & H_1 & H_0 & & & & & & & & \\
\hline
S15 & S_14 & S_13 & S_12 & S_11 & S_10 & S_9 & S_8 & S_7 & S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0 & \leftarrow \text{FINAL PRODUCT}
\end{array}
$$

(a)



(b)

Figure 4.3: (a) Multiplying two 8-bit operands results in eight partial products which are added to form a 16-bit final product. (b) Dot representation of the same multiplication.

Since the algorithms for each step are mostly independent of each other, we will describe them separately.

4.2.1 Partial Products Generation: Booth's Algorithm

The simplest way to generate partial products is to use *AND* gates as 1×1 multipliers. For example, in Figure 4.3a:

$$A_0 = Y_0X_0, A_1 = Y_0X_1, B_0 = Y_1X_0,$$

and so on. In this manner, an n-bit multiplier generates n partial products. However, it is possible to use encoding techniques that reduce the number of partial products. The modified Booth's algorithm is such an encoding technique, which reduces the number of partial products by half.

The original Booth's algorithm Booth [1951] allows the multiplication operation to skip over any contiguous string of all 1's and all 0's in the multiplier, rather than form a partial product for each bit. Skipping a string of 0's is straightforward, but in skipping over a string of 1's the following property is put to use: a string of 1's can be evaluated by subtracting the weight of the rightmost 1 from the modulus. A string of n 1's is the same as 1 followed by n 0's less 1. For example, the value of the binary string 11100 computes to $2^5 - 2^2 = 28$ (i.e., $100,000 - 100$).

A modified version of Booth's algorithm is more commonly used. The difference between the Booth's and the modified Booth's algorithm is as follows: the latter always generates $n/2$ independent partial products, whereas the former generates a varying (at most $n/2+1$) number of partial products, depending on the bit pattern of the multiplier. Of course, parallel hardware implementation lends itself only to the fixed independent number of partial products. The modified multiplier encoding scheme encodes 2-bit groups and produces five partial products from an 8-bit (unsigned numbers) multiplier, the fifth partial product being a consequence of using two's complement representation of the partial products. (Only four partial products are generated if only two's complement input representation is used, as the most significant input bit represents the sign.)

Each multiplier is divided into substrings of 3 bits, with adjacent groups sharing a common bit. Booth's algorithm can be used with either unsigned or two's complement numbers (the most significant bit of which has a weight of -2^n), and requires that the multiplier be padded with a 0 to the right to form four complete groups of 3 bits each. To work with unsigned numbers, the n-bit multiplier must also be padded with one or two zeros in the multipliers to the left. Table 4.1, from Anderson Anderson et al. [1967], is the encoding table of the eight permutations of the three multiplier bits.

In using Table 4.1, the multiplier is partitioned into 3-bit groups with one bit shared between groups. If this shared bit is a 1, subtraction is indicated, since we prepare for a string of 1's. Consider the case of unsigned (i.e., positive) numbers; let X represent the multiplicand (all bits) and $Y = Y_{n-1}, Y_{n-2}, \cdots, Y_1, Y_0$ an integer multiplier—the binary point following Y_0. (The placement of the point is arbitrary, but all actions are taken with respect to it.) The lowest order action is derived from multiplier bits $Y_1Y_0.0$—the *LSB* has been padded with a zero. Only four actions are possible: $Y_1Y_0.0$ may be either 00.0, 01.0, 10.0, or 11.0. The first two cases are straightforward; for 00.0, the partial product is 0; for 01.0, the partial product is $+X$. The other two cases are perceived as the beginning of a string of 1's. Thus, we subtract $2X$ (i.e., add

Bit			Operation	
2^1	2^0	2^{-1}		
Y_{i+1}	Y_i	Y_{i-1}		
0	0	0	Add zero (no string)	$+0$
0	0	1	Add multiplicand (end of string)	$+X$
0	1	0	Add multiplicand (a string)	$+X$
0	1	1	Add twice the multiplicand (end of string)	$+2X$
1	0	0	Subtract twice the multiplicand (beginning of string)	$-2X$
1	0	1	Subtract the multiplicand ($-2X$ and $+X$)	−X
1	1	0	Subtract the multiplicand (beginning of string)	$-X$
1	1	1	Subtract zero (center of string)	-0

Table 4.1: Encoding 2 multiplier bits by inspecting 3 bits, in the modified Booth's algorithm.

$-2X$) for the case 10.0, and subtract X for the case 11.0. Higher order actions must recognize that this subtraction has occurred. The next higher action is found from multiplier bits $Y_3Y_2Y_1$ (remember, Y_1 is the shared bit). Its action on the multiplicand has 4 times the significance of $Y_1Y_0.0$. Thus, it uses the table as $Y_3Y_2Y_1$, but resulting actions are shifted by 2 (multiplied by 4). Thus, suppose the multiplier was 0010.0; the first action (10.0) would detect the start of a string of 1's and subtract $2X$, while the second action (00.1) would detect the end of a string of 1's and add X. But the second action has a scale or significance point 2 bits higher than the first action (4 times more significant). Thus, $4 \times X - 2X = 2X$, the value of the multiplier, (0010.0). This may seem to the reader to be a lot of work to simply find $2X$, and, indeed, in this case two actions were required rather than one. By inspection of the table, however, only one action (addition or subtraction) is required for each *two* multiplier bits. Thus, use of the algorithm insures that for an n-bit multiplier only $n/2$ actions will be required for any multiplier bit pattern.

For the highest order action with an unsigned multiplier, the action must be derived with a leading or padded zero. For an odd number of multiplier bits, the last action will be defined by $0Y_{n-1}.Y_{n-2}$. For an even number of multiplier bits, $\frac{n}{2}+1$ actions are required, the last action being defined by $00.Y_{n-1}$.

Multipliers in two's complement form may be used directly in the algorithm. In this case, the highest order action is determined by $Y_{n-1}Y_{n-2}Y_{n-3}$ (no padding) for an even number of multiplier bits, and $Y_{n-1}Y_{n-1}.Y_{n-2}$, a sign extended group, for odd sized multipliers; e.g., suppose $Y = -1$. In two's complement, $Y = 1111\cdots 11$. The lowest order action (11.0) is $-X$; all other actions (11.1) are -0, producing the desired result ($-X$).

In implementing the actions, the $2X$ term is simply a 1-bit left shift of X. Thus, multiplicands must be arranged to be gated directly with respect to a scale point or shifted one bit. Subtraction is implemented by gating the complement of X (i.e., the 1's complement) and then adding a 1 with respect to the scale point. In implementing this, the Y_{i+1} can be used as a subtractor indicator that will be added to the *LSB* of the partial product. If bit $Y_{i+1} = 0$, no subtraction is called for, and adding 0 changes nothing. On the other hand, if bit $Y_{i+1} = 1$, then subtraction is called for and the proper two's complement is performed by adding 1 to the *LSB*. Of course, in the two's complement, the sign bit must be extended to the full width of the final result, as shown by the repetitive terms in Figure 4.4.

In Figure 4.4, if X and Y are 8-bit unsigned numbers, then A_8–A_0 are determined by the $Y_1Y_0.0$

A_9	A_9	A_9	A_9	A_9	A_9	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
B_9	B_9	B_9	B_9	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0		Y_1
C_9	C_9	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0		Y_3		
D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		Y_5				
E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0		Y_7						
S_{15}	S_{14}	S_{13}	S_{12}	S_{11}	S_{10}	S_9	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Figure 4.4: Generation of five partial products in 8 × 8 multiplication, using modified Booth's algorithm (only four partial products are generated if the representation is restricted to two's complement).

2^2	2^1	2^0	2^{-1}	Operation
Y_{i+2}	Y_{i+1}	Y_i	Y_{i-1}	
0	0	0	0	$+0$
0	0	0	1	$+X$
0	0	1	0	$+X$
0	0	1	1	$+2X$
0	1	0	0	$+2X$
0	1	0	1	$+3X$
0	1	1	0	$+3X$
0	1	1	1	$+4X$
1	0	0	0	$-4X$
1	0	0	1	$-3X$
1	0	1	0	$-3X$
1	0	1	1	$-2X$
1	1	0	0	$-2X$
1	1	0	1	$-X$
1	1	1	0	$-X$
1	1	1	1	-0

Table 4.2: Extension of the modified Booth's algorithm to 3-bit multiplier group encoding. This requires generation of $\pm 3X$, which is not as trivial as the operations in the 2-bit multiplier encoding. (Compare with Table 4.1.)

action on X. Since $2X$ is a possible action, A_8 may be affected and A_9 is the sign and extension. If a $\pm X$ action is determined, then A_8 is the sign and $A_8 = A_9$. The fifth action is determined by $00.Y_7$—always a positive, unshifted action ($+X$ or $+0$). If X and Y are 8-bit two's complement numbers, then $A_8 = A_9$ are the sign and extension. Also, no fifth action ($E_7 - E_0$) is required.

The A_9, B_9, C_9, D_9 terms appear to significantly increase the hardware required for partial product addition. There are 16 such terms. While the full addition of these terms results in the correct product formations, $S_{15} - S_0$, simpler implementations are possible. By recognizing the $A_9, B_9, \cdots$ terms are sign identifiers and generating the sign logic separately, the additional summing hardware for such terms can be eliminated.

The additional gate delay in implementing the modified Booth's algorithm consists of four gates: two gates for decoding the 3-bit multiplier and two gates for selecting X or $2X$.

An extension of the modified Booth's algorithm involves an encoding of three bits at a time while examining four multiplier bits. This scheme would generate only $n/3$ partial products.

However, the encoding truth table in Table 4.2 requires the generation of a three timesmultiplicand term, which is not as trivial as generating twice the multiplicand. Thus, most hardware implementations use only the 2-bit encoding.

Suppose the multiplicand (X) is to be multiplied by an unsigned Y:

$$0011101011$$

That is, decimal 235.

We use modified Booth's algorithm (Table 4.1) and assume that this multiplier is a binary integer with point indicated by (.). Now the multiplier must be decomposed into overlapping 3-bit segments and actions determined for each segment. Note that the first segment has an implied "0" to the right of the binary point. Thus, we can label each segment as follows:

$$\begin{array}{ccccccccccc} & (5) & & & & (3) & & & & (1) & \\ \overline{0} & \overline{0} & \underline{\overline{1}} & \underline{1} & \underline{\overline{1}} & \overline{0} & \underline{\overline{1}} & \underline{0} & \underline{\overline{1}} & \overline{1} & \overline{.0} \\ & & & (4) & & & & (2) & & & \end{array}$$

While segment (1) is referenced to the original binary point, segment (2) is *four times more significant*. Thus, any segment (2) action on X (the multiplicand) must be scaled by a factor of four. Similarly, segment (3) is four times more significant than 2, and 16 times more significant than 1.

Now, by using the table and scaling as appropriate, we get the following actions:

Segment number	Bits	Action	Scale factor	Result
(1)	110	$-X$	1	$-X$
(2)	101	$-X$	4	$-4X$
(3)	101	$-X$	16	$-16X$
(4)	111	0	64	0
(5)	001	$+X$	256	$+256X$
		Total action		$235X$

Note that the table of actions can be simplified for the first segment (Y_{i-1} always 0) and the last segment (depending on whether there is an even or odd number of bits in the multiplier).

Also note that the actions specified in the table are independent of one another. Thus, the five result actions in the example may be summed simultaneously using the carry-save addition techniques discussed in the last chapter.

4.2.2 Using ROMs to Generate Partial Products

Another way to generate partial products is to use ROMs. For example, the 8×8 multiplication of Figure 4.3 can be implemented using four 256×8 ROMs, where each ROM performs 4×4 multiplication, as shown in Figure 4.5.

In Figure 4.5, each 4-bit value of each element of the pair $(Y_{\mathrm{A}}, X_{\mathrm{A}})$ $(Y_{\mathrm{B}},\ X_{\mathrm{A}})$ $(Y_{\mathrm{A}}, X_{\mathrm{B}})$ and $(Y_{\mathrm{B}}, X_{\mathrm{B}})$ is concatenated to form an 8-bit address into the 256 entry ROM table. The entry

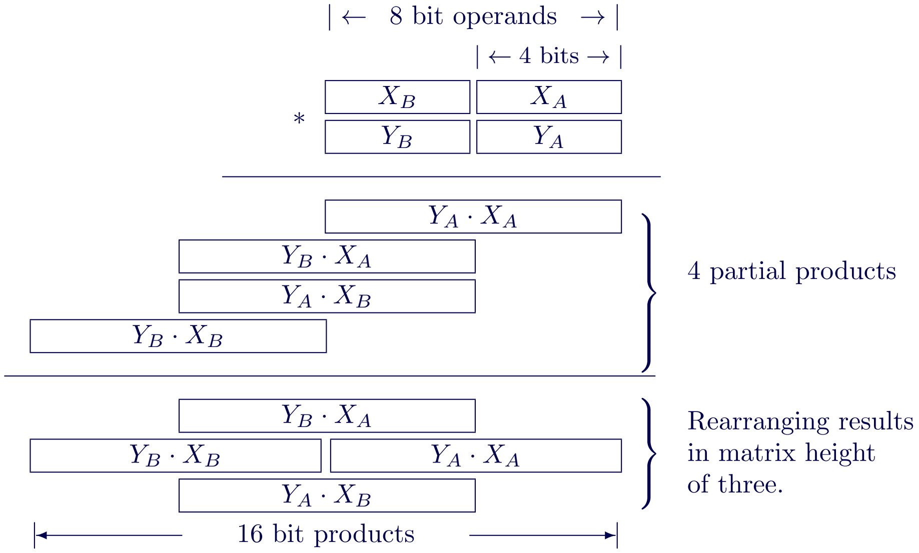


Figure 4.5: Implementation of 8 × 8 multiplication using four 256 × 8 ROMs, where each ROM performs 4 × 4 multiplication.

Table 4.3: Summary of maximum height of the partial products matrix for the various partial generation schemes where n is the multiple size.

		MAX Height of the Matrix							
	General	Number of Bits							
Scheme	Formula	8	16	24	32	40	48	56	64
1 × 1 multiplier (*AND* gate)	n	8	16	24	32	40	48	56	64
4 × 4 multiplier (ROM)	$(n/2) - 1$	3	7	11	15	19	23	27	31
8 × 8 multiplier (ROM)	$(n/4) - 1$	1	3	4	7	9	11	13	15
Modified Booth's algorithm	$(n/2)$	4	8	12	16	20	24	28	32

contains the corresponding 8-bit product. Thus, four tables are required to simultaneously form the products: $Y_{\text{A}} \cdot X_{\text{A}}$, $Y_{\text{B}} \cdot X_{\text{A}}$, $Y_{\text{A}} \cdot X_{\text{B}}$, and $Y_{\text{B}} \cdot X_{\text{B}}$. Note that the $Y_{\text{A}} \cdot X_{\text{A}}$ and the $Y_{\text{B}} \cdot X_{\text{B}}$ terms have disjoint significance; thus, only three terms must be added to form the product. The number of rearranged partial products that must be summed is referred to as the matrix height—the number of initial inputs to the CSA tree.

A generalization of the ROM scheme is shown in Figure 4.6 Texas Instruments, Inc. [1976] for various multiplier arrays of up to 64 × 64. In the latter case, 256 partial products are generated. But upon rearranging, the maximum column height of the matrix is 31. Table 4.3 summarizes the partial products matrix.

These partial products can be viewed as adjacent columns of height h. Now we are ready to discuss the implementations of column reductions.

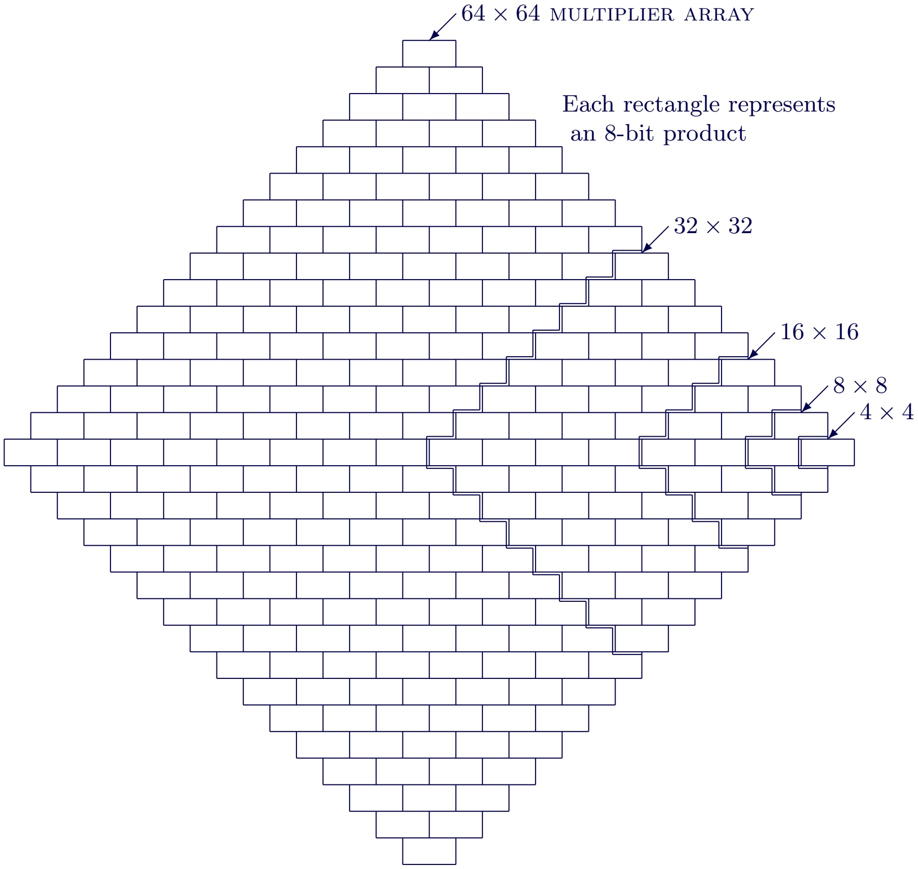


Figure 4.6: Using ROMs for various multiplier arrays for up to 64 × 64 multiplication. Each ROM is a 4×4 multiplier with 8-bit product. Each rectangle represents the 8-bit partial product ($h = 31$).

4.2.3 Partial Products Reduction

As mentioned in the last chapter, the common approach in all of the summand reduction techniques is to reduce the n partial products to two partial products. A carry look ahead is then used to add these two products. One of the first reduction implementations was the Wallace tree Wallace [1964], where carry save adders are used to reduce 3 bits of column height to 2 bits (Figure 4.7). In general, the number of the required carry save adder levels (L) in a Wallace tree to reduce height h to 2 is:

$$L \doteq \left\lceil \log_{3/2}\left(\frac{h}{2}\right)\right\rceil = \left\lceil \log_{1.5}\left(\frac{h}{2}\right)\right\rceil,$$

where h is the number of operands (actions) to be summed and L is the number of CSA stages of delay required to produce the pair of column operands. For 8×8 multiplication using 1×1 multiplier generation, $h = 8$ and four levels of carry-save-adders are required, as illustrated in Figure 4.8. Following we show the number of levels versus various column heights:

Column Height (h)	Number of Levels (L)
3	1
4	2
$4 < n \leq 6$	3
$6 < n \leq 9$	4
$9 < n \leq 13$	5
$13 < n \leq 19$	6
$19 < n \leq 28$	7
$28 < n \leq 42$	8
$42 < n \leq 63$	9

Dadda Dadda [1965] coined the term "parallel (n, m) counter." This counter is a combinatorial network with m outputs and $n(\leq 2^m - 1)$ inputs. The m outputs represent a binary number encoding the number of ones present at the inputs. The carry save adder in the preceding Wallace tree is a (3, 2) counter.

This class of counters has been extended in an excellent article Stenzel *et al.* [1977] that shows the Wallace tree and the Dadda scheme to be special cases. The generalized counters take several successively weighted input columns and produce their weighted sum. Counters of this type are denoted as:

$$(C_{ßR-1},\ C_{ßR-2},\ \cdots,\ C_0,\ d)$$

counters, where R is the number of input columns, C_i is the number of input bits in the column of weight 2^i, and d is the number of bits in the output word. The suggested implementation for such counters is a ROM. For example, a (5, 5, 4) counter can be programmed in $1K \times 4$ ROM, where the ten address lines are treated as two adjacent columns of 5 bits each. Note that the maximum sum of the two columns is 15, which requires exactly 4 bits for its binary representation. Figures 4.9 and 4.10 illustrate the ROM implementation of the (5, 5, 4) counter and show several generalized counters. The use of the (5, 5, 4) counter to reduce the partial products in a 12×12 multiplication is shown in Figure 4.11, where the partial products are generated by 4×4 multipliers.

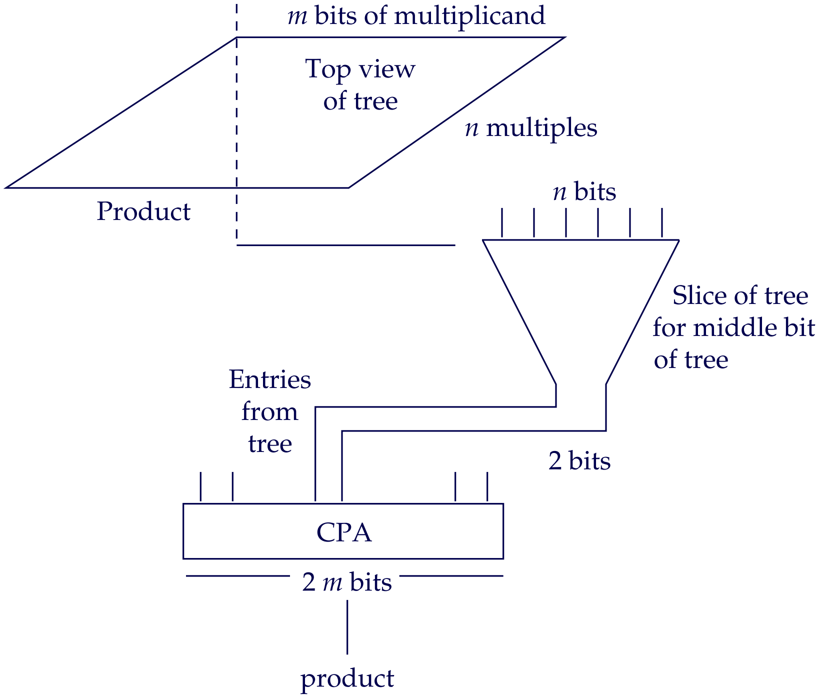


Figure 4.7: Wallace tree.

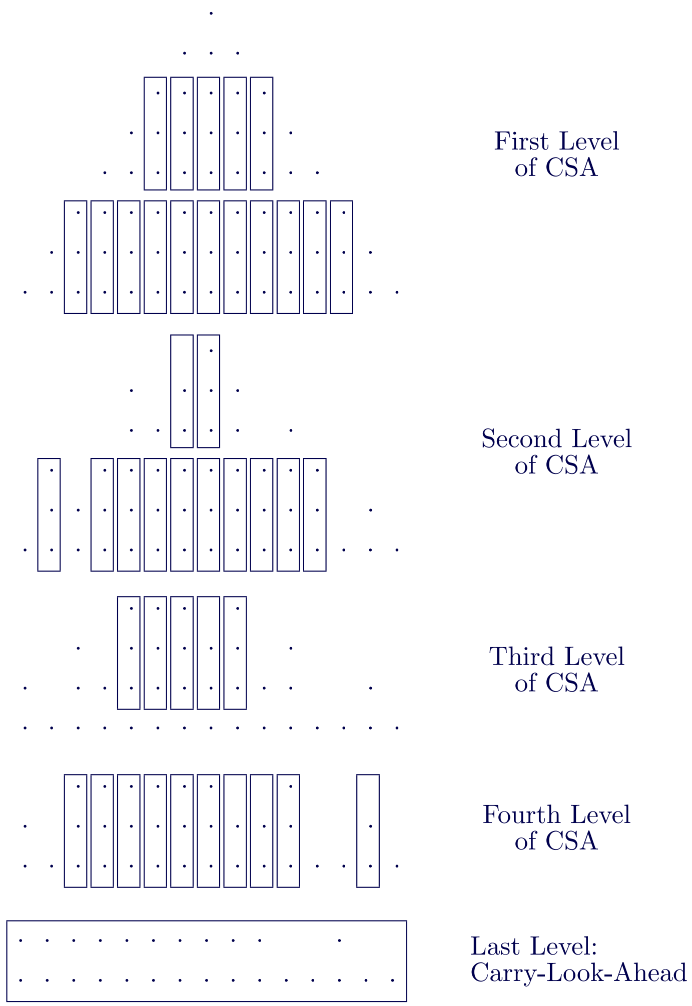
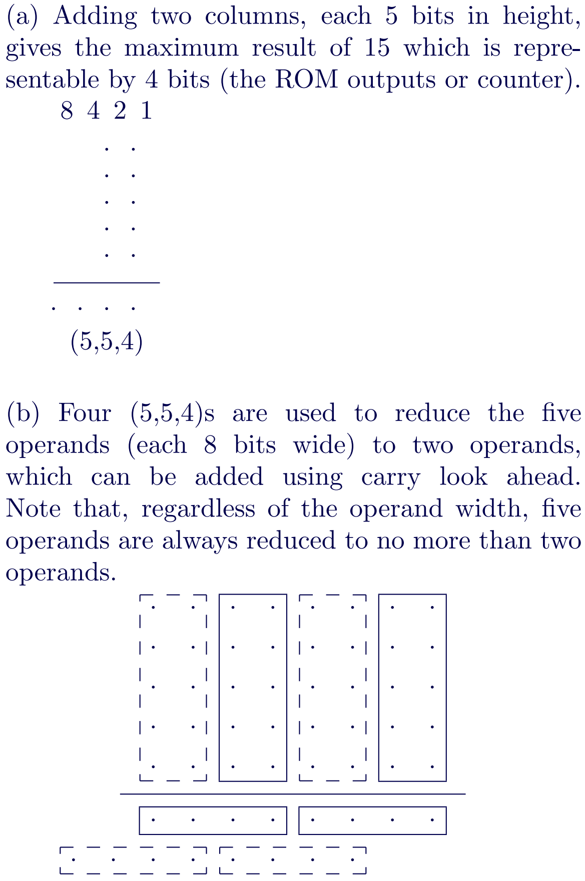


Figure 4.8: Wallace tree reduction of 8 × 8 multiplication, using carry save adders (CSA).



(a) Adding two columns, each 5 bits in height, gives the maximum result of 15 which is representable by 4 bits (the ROM outputs or counter).

(b) Four (5,5,4)s are used to reduce the five operands (each 8 bits wide) to two operands, which can be added using carry look ahead. Note that, regardless of the operand width, five operands are always reduced to no more than two operands.

Figure 4.9: The $(5, 5, 4)$ reduces the five input operands to one operand.

Parallel compressors, which are a subclass of parallel counters, have been introduced by Gajski Gajski [1980]. These compressors are distinctively characterized by the set of inputs and outputs that serves as an interconnection between different packages in a one-dimensional array of compressors. These compressors are said to be more efficient than parallel counters. For more details on this interesting approach, the reader is referred to Gajski's article Gajski [1980].

4.3 Iteration and Partial Products Reduction

4.3.1 A Tale of Three Trees

The Wallace tree can be coupled with iterative techniques to provide cost effective implementations. A basic requirement for such implementation is a good latch to store intermediate results. In this case "good" means that it does not add additional delay to the computation. As we shall see in more detail later, the Earle latch (Figure 4.12), accomplishes this by simply providing a

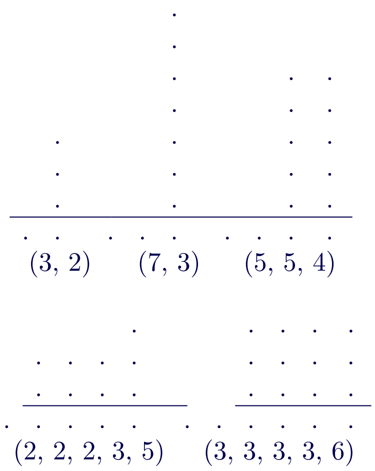


Figure 4.10: Some generalized counters from Stenzel Stenzel *et al.* [1977].

feed–back path from the output of an existing canonic circuit pair to an additional input.

Thus, an existing path delay is not increased (but fan–in requirements are increased by one). In a given logic path, an existing "and–or" pair is replaced by the latch which also performs the "and–or" function.

Use of this latch can significantly reduce implementation costs if appropriate care is taken in design (see Chapter 6).

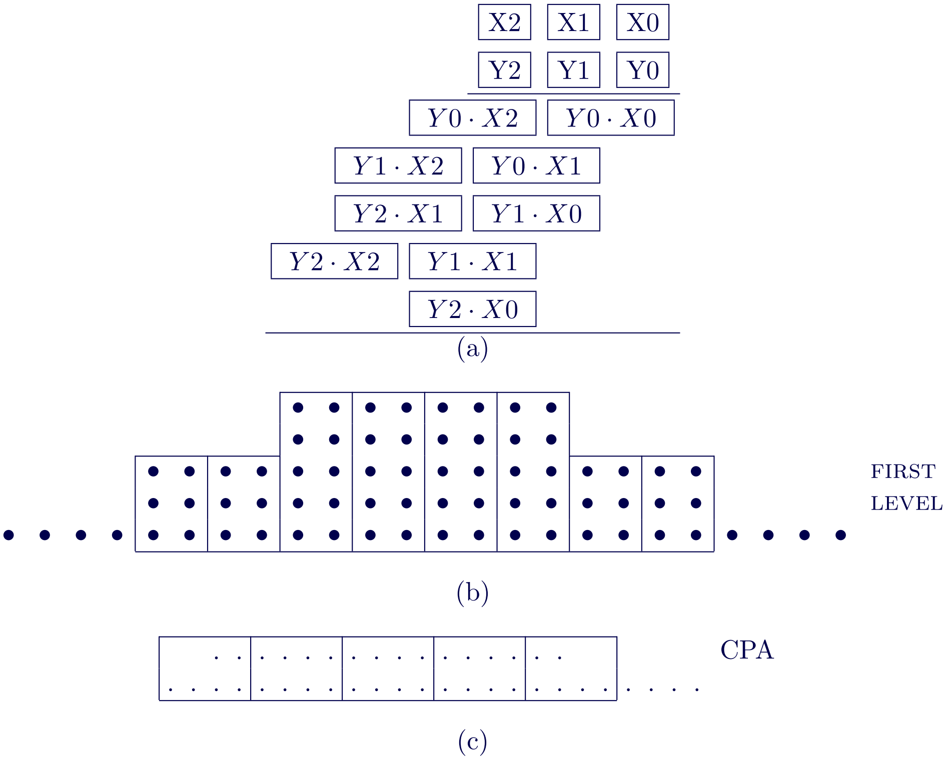
Consider the operation of an n–bit multiplier on an m–bit multiplicand; that is, we reduce n partial products, each of m bits, to two partial products, then propagate the carries to form a single (product) result. Trees of size L actually have some bits of their partial product tree for which L CSA stages are required. Note that for both low–order and high–order product bits the tree size is less.

Now in the case of the simple Wallace tree the time required for multiplication is:

$$\tau \leq L \cdot 2 + \text{CPA}(m + n \text{ bits}),$$

where τ is in unit gate delays. Each CSA stage has 2 serial gates (an *AND–OR* in both sum and carry). The CPA $(m + n)$ term represents the number of unit gate delays for a carry look ahead structure with operand size $m + n$ bits. Actually since the tree height at the less significant bits is smaller than the middle bits, these positions arrive early to the CPA. Thus, the CPA term is somewhat conservative.

The full Wallace tree is "expensive" and, perhaps more important, is topologically difficult to implement. That is, large trees are difficult to map onto planes (printed wire boards) since each CSA communicates with its own slice, transmits carries to the higher order slice, and receives carries from a lower order. This "solid" topology creates both I/0 pin difficulty (if the implementation "spills" over a single board) and wire length (routing) problems.



(a)

(b)

(c)

Figure 4.11: 12×12 bit partial reduction using $(5, 5, 4)$ counters. The $X0$, $Y0$, $X1$, etc., terms each represent four bits of the argument. Thus, the product $X0Y0$ is an 8-bit summand.
(a) Partial products are generated by 4×4 multipliers Stenzel *et al.* [1977].
(b) Eight $(5, 5, 4)$s are used to compress the column height from five to two.
(c) A CPA adder is used to add the last two operands. The output of each counter in (b) produces a 4-bit result: 2 bits of the same significance, and 2 of higher. These are combined in (c).

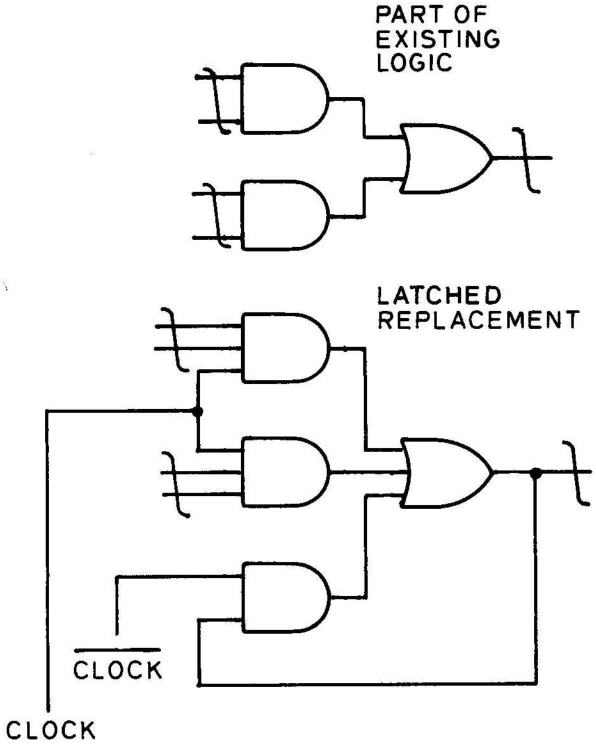


Figure 4.12: Earle latch.

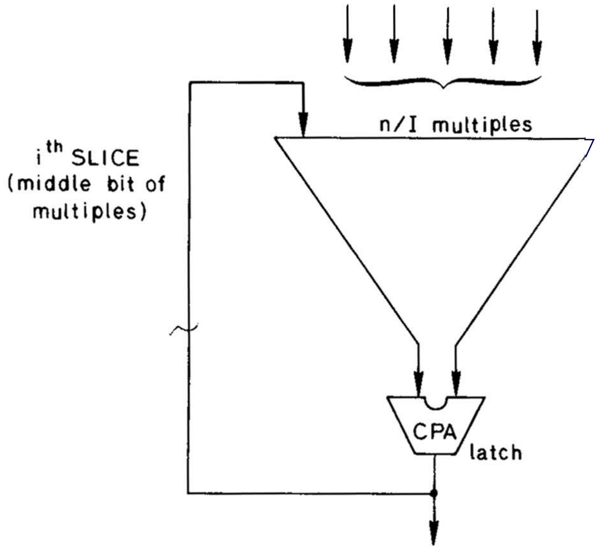


Figure 4.13: Slice of a simple iteration tree showing one product bit.

Iterating on smaller trees has been proposed [Anderson et al., 1967] as a solution. Instead of entering n multiples of the multiplicand we use an n/I tree and perform I iterations on this smaller tree.

Consider three types of tree iterations:

1. Simple iteration: in this case, the n/I multiples of the m–bit multiplicand are reduced and added to form a single partial product. This is latched and fed back into the top of the tree for assimilation on the next iteration.

   The multiplication time is now the number of iterations, I, times the sum of the delay in the CSA tree height (two delays per CSA) and the CPA delay. The CSA tree is approximately $\log_{3/2}$ of the number of inputs $\lceil \frac{n}{\mathrm{I}} + 1 \rceil$ divided by 2, since the tree is reduced to *two* outputs, not *one*. The maximum size of the operands entering the CPA on any iteration is $m + \lceil \frac{n}{\mathrm{I}} + 1 \rceil$;

   $$\tau \approx \mathrm{I}\left(2\left\lceil \log_{3/2}\left\lceil \frac{n}{\mathrm{I}} + 1\right\rceil /2\right\rceil + \mathrm{CPA}\left(m + \left\lceil \frac{n}{\mathrm{I}} + 1\right\rceil\right)\right)$$

   unit gate delays.

2. Iterate on tree only: the above scheme can be improved by avoiding the use of the CPA until the partial products are reduced to two. This requires that the (shifted) two partial results be fed back into the tree *before* entering the CPA, (Figure 4.14). The "shifting" is required since the new $\frac{n}{\mathrm{I}}$ input multiples are at least (could be more, depending on multiplier encoding) $\frac{n}{\mathrm{I}}$ bits more significant than the previous iteration. Thus, each pair of reduced results is returned to the top of the tree and shifted to the correct significance. Therefore, we require only one CPA and I iterations on the CSA tree.

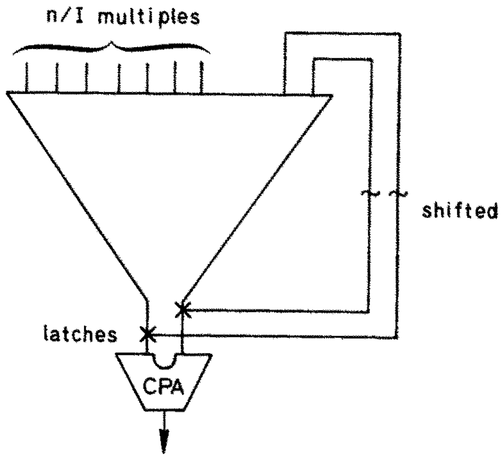


Figure 4.14: Slice of tree iteration showing one product bit.

The time for multiplication is now:

$$\tau \approx I\left(2\left\lceil \log_{3/2}\left\lceil \frac{n}{\mathrm{I}}+2\right\rceil /2\right\rceil\right)+\mathrm{CPA}\,(m+n)$$

3. Iterate on lowest level of tree: In this case the partial products are assimilated by returning them to the lowest level of the tree. When they are assimilated to two partial products again a single CPA is used, (Figure 4.15). Thus:

   As the Earle latch requires (approximately) a minimum of four gate delays for a pipeline stage, returning the CSA output one level back into the tree provides an optimum implementation (Chapter 6 provides more detailed discussion). The tree height is increased; but only the initial set of multiples sees the delay of the total tree. Subsequent sets are introduced at intervals of four gate delays, (Figure 4.15). Thus, the time for multiplication is now:

$$\tau \approx 2\left(\left\lceil \log_{\frac{3}{2}}\left\lceil \frac{n}{\mathrm{I}}\right\rceil /2\right\rceil\right)+\mathrm{I}\cdot 4+\mathrm{CPA}(2m)$$

Note that while the cost of the tree has been significantly reduced, only the $\mathrm{I}\cdot 4$ term differs

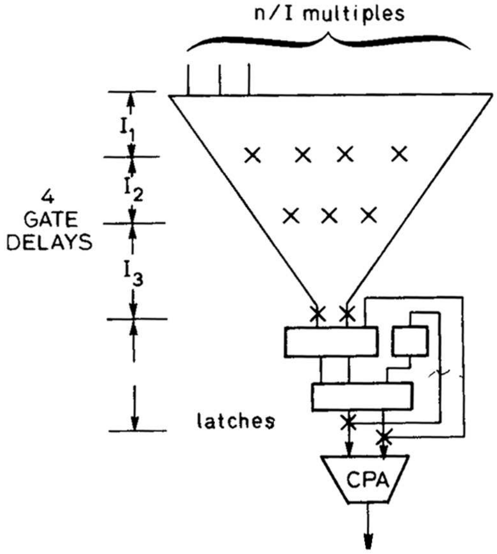


Figure 4.15: Slice of low level tree iteration.

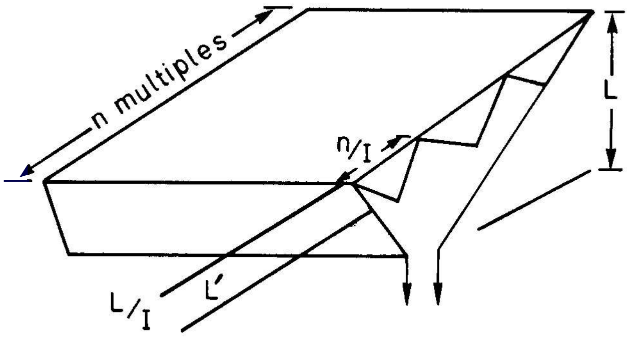


Figure 4.16: Iteration.

from the full Wallace tree time. So long as I is chosen so that this term does not dominate the total time, attractive implementations are possible using a tree of size L′ instead of L levels.

In all of these approaches, we are reducing the height of the tree by inputing significantly fewer terms, about $\frac{n}{\mathrm{I}}$ instead of n. These fewer input terms mean a much smaller tree—fewer components (cost) and quicker generation of a partial result, but now I iterations are needed for a complete result instead of a single pass.

4.4 Iterative Array of Cells

The matrix generation/reduction scheme, (Wallace tree), is the fastest way to perform parallel multiplication; however, it also requires the most hardware. The iterative array of cells requires less hardware but it is slower. The hardware required in the iterative approach can be calculated from the following formula:

$$\text{number of building blocks} = \left\lceil \frac{\mathrm{N} \times \mathrm{M}}{n \times m} \right\rceil$$

where N, M are the number of bits in the final multiplication, and n, m are the number of bits in each building block. For example, nine 4 × 4 multipliers are required to perform 12 × 12 multiplication in the iterative approach (since (12 × 12)/(4 × 4) = 9). By contrast, using the matrix generation technique to do 12 × 12 multiplication requires 13 adders in addition to the nine 4 × 4 multipliers (see Figure 4.11). In general, the iterative array of cells is more attractive for shorter operand lengths since their delay increases linearly with operand length, whereas the delay of the matrix–generation approach increases with the log of the operand length.

The simplest way to construct an iterative array of cells is to use 1–bit cells, which are simply full adders. Figure 4.17 depicts the construction of a 5 × 5 unsigned multiplication from such cells.

In the above equation, we define the arithmetic value (weight) of a logical zero state as μ and a logical one state as ν; then $P(\mu, \nu)$ is a variable with states μ and ν. Thus, a conventional carry-save adder with unsigned inputs performs $X_0+Y_0+C_0 = C_1+S_0$, or $(0,1)+(0,1)+(0,1) = (0,2)+(0,1)$.

In two's complement, the most significant bit of the arguments is the sign bit, thus for 5×5 bit two's complement multiplication of $Y_4Y_3Y_2Y_1Y_0$ by $X_4X_3X_2X_1X_0$, Y_4 and X_4 indicate the sign. Thus, the terms

$$X_0Y_4, X_1Y_4, \ldots, X_3Y_4$$

and

$$X_4Y_3, X_44Y_2, X_4Y_1, X_4Y_0$$

have range $(0,-1)$.

$$X_0Y_4 + 0 + X_1Y_3 \text{ is } (0,-1) + 0 + (0,1) =$$

This can be implemented by type II$'$: $(0,1) + (0,-1) + (0,-1) = (0,-2) + (0,1)$

The negative carry out and the negative X_iY_4 requires type II$'$ circuits along the right diagonal, until negative input X_4Y_3 combines with X_3Y_4 defining a type I$'$ situation wherein all inputs are zero or negative.

Now the type II cell has equations:

$$C_1 = A_0B_0C_0 + \bar{A}_0B_0C_0 + \bar{A}\bar{B}_0C_0 + \bar{A}B_0\bar{C}_0$$

$$S_0 = A_0B_0C_0 + \bar{A}_0\bar{B}_0C_0 + \bar{A}_0B_0\bar{C}_0 + A_0\bar{B}_0\bar{C}_0$$

(recall that $S_0 = (0,-1)$

Figure 4.17 can be generalized by creating a 2 bit adder cell (Figure 4.20) akin to the 1-bit cell of Figure 4.18. For unsigned numbers, an array of circuits of the type called L101:

$$A_1 + B_1 + A_0 + B_0 + C_0 = C_2 + S_1 + S_0$$

$$(0,2) + (0,2) + (0,1) + (0,1) + (0,1) = (0,4) + (0,2) + (0,1)$$

is required.

In (Figure 4.17), if we let

$$\begin{aligned} A_1 &= X_0Y_2 \\ B_1 &= X_1Y_1 \\ A_0 &= X_0Y_1 \\ B_0 &= X_1Y_0 \\ C_0 &= 0 \end{aligned}$$

we capture the upper rightmost 2-bit positions with one 2-bit cell (Figure 4.20). Continuing for the 5×5 unsigned multiplication we need one half the number of cells (plus 1) as shown in Figure 4.17. We can again extend this to two's complement arithmetic.

To perform signed (two's complement) multiplication the scheme is slightly more complex [Pezaris, 1971]. Pezaris illustrates his scheme by building 5×5 multipliers from two types of circuits, using the following 1–bit adder cell (Figure 4.18.):

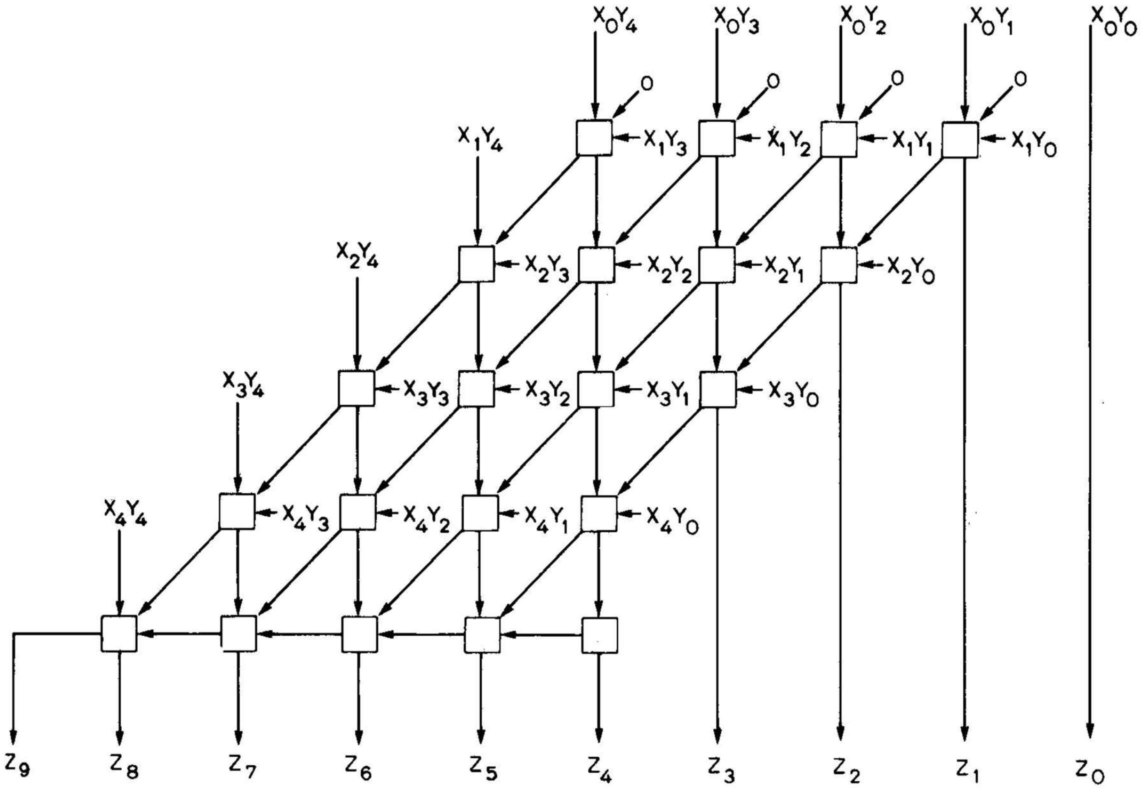


Figure 4.17: 5 × 5 unsigned multiplication.

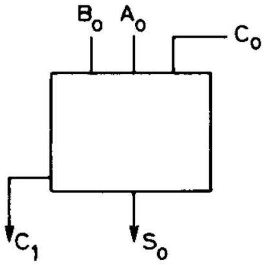


Figure 4.18: 1–bit adder cell.

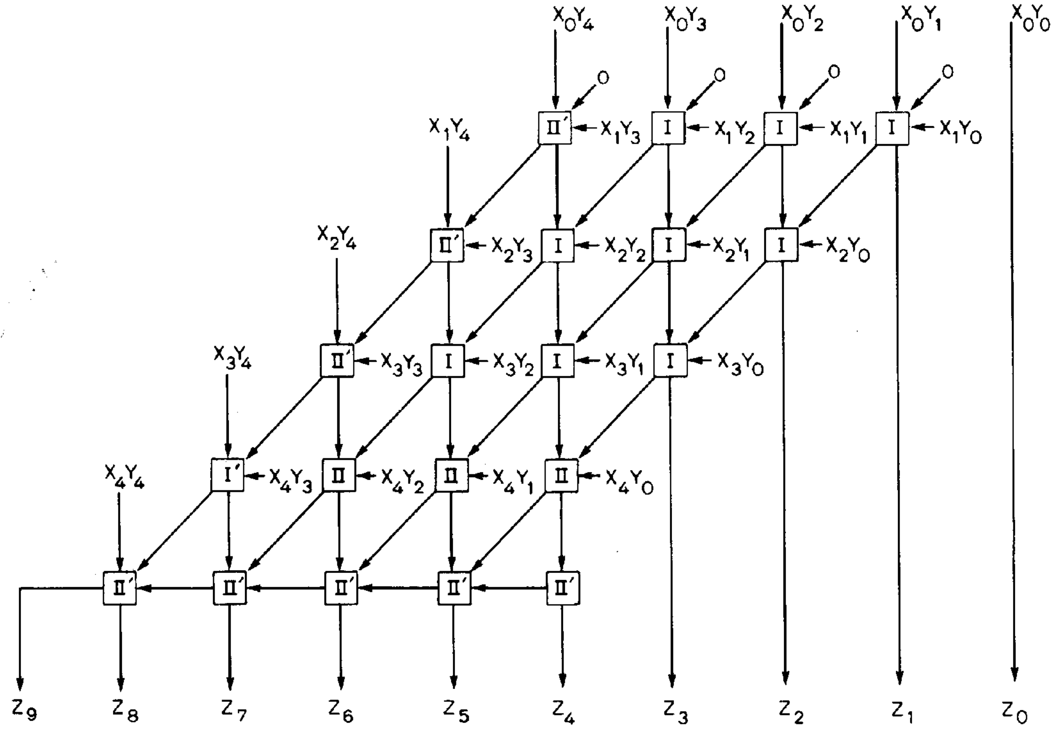


Figure 4.19: 5 × 5 two's complement multiplication [PEZ 70].

	A_0		B_0		C_0		C_1		S_0
TYPE I :	$(0,1)$	$+$	$(0,1)$	$+$	$(0,1)$	$=$	$(0,2)$	$+$	$(0,1)$
TYPE II :	$(0,-1)$	$+$	$(0,1)$	$+$	$(0,1)$	$=$	$(0,2)$	$+$	$(0,-1)$
TYPE II′ :	$(0,1)$	$+$	$(0,-1)$	$+$	$(0,-1)$	$=$	$(0,-2)$	$+$	$(0,1)$
TYPE I′ :	$(0,-1)$	$+$	$(0,-1)$	$+$	$(0,-1)$	$=$	$(0,-2)$	$+$	$(0,-1)$

Type I is the conventional carry save adder, and it is the only type used in Figure 4.17 for the unsigned multiplication. Types I and I′ correspond to identical truth tables (because if $x+y+z=u+v$, then $-x-y-z=-u-v$) and, therefore, to identical circuits. Similarly, types II and II′ correspond to identical circuits. Figure 4.19 shows the entire 5 × 5 multiplication.

Pezaris extends the 1–bit adder cell to a 2–bit adder cell, as shown below:

Implementation of this method with 2–bit adders requires *three* types of circuits, (the L101,

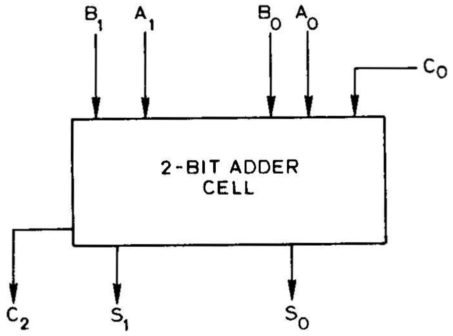


Figure 4.20: 2–bit adder cell.

L102, L103). The arithmetic operations performed by these three types are given below:

$$\begin{array}{lccccccccccccccc}
 & \mathrm{A}_1 & & \mathrm{B}_1 & & \mathrm{A}_0 & & \mathrm{B}_0 & & \mathrm{C}_0 & & \mathrm{C}_2 & & \mathrm{S}_1 & & \mathrm{S}_0 \\
\mathrm{L101}: & (0,2) & + & (0,2) & + & (0,1) & + & (0,1) & + & (0,1) & = & (0,4) & + & (0,2) & + & (0,1) \\
\mathrm{L102}: & (0,2) & + & (0,-2) & + & (0,1) & + & (0,1) & + & (0,1) & = & (0,4) & + & (0,-2) & + & (0,1) \\
\mathrm{L103}: & (0,2) & + & (0,-2) & + & (0,1) & + & (0,-1) & + & (0,1) & = & (0,4) & + & (0,-2) & + & (0,1)
\end{array}$$

Iterative multipliers perform the operation

$$\mathrm{S} = \mathrm{X} \cdot \mathrm{Y} + \mathrm{K},$$

where K is a constant to be added to the product (whereas the matrix generation schemes perform only $\mathrm{S} = \mathrm{X} \cdot \mathrm{Y}$). The device uses the (3–bit) modified Booth's algorithm to halve the number of partial products generated. Figure 4.21 shows the block diagram of the iterative cell. The X_{-1} input is needed in expanding horizontally since the Booth encoder may call for 2X, which is implemented by a left shift. The Y_{-1} is used as the overlap bit during multiplier encoding. Note that outputs S_4 and S_5 are needed only on the most significant portion of each partial product (these 2 bits are used for sign correction). Figure 4.22 shows the implementation of a 12×12 two's complement multiplier. This scheme can be extended to larger cells. For example, in Figure 4.22, the dotted line encloses an 8×8 iterative cell.

4.5 Detailed Design of Large Multipliers

4.5.1 Design Details of a 64×64 Multiplier

In this section, we describe the design of a 64×64 multiplier using the technique of simultaneous generation of partial products. The design uses standard design macros. Four types of macros are needed to implement the three steps of a parallel multiplier.

1. Simultaneous generation of partial products—using an 8×8 multiplier macro.

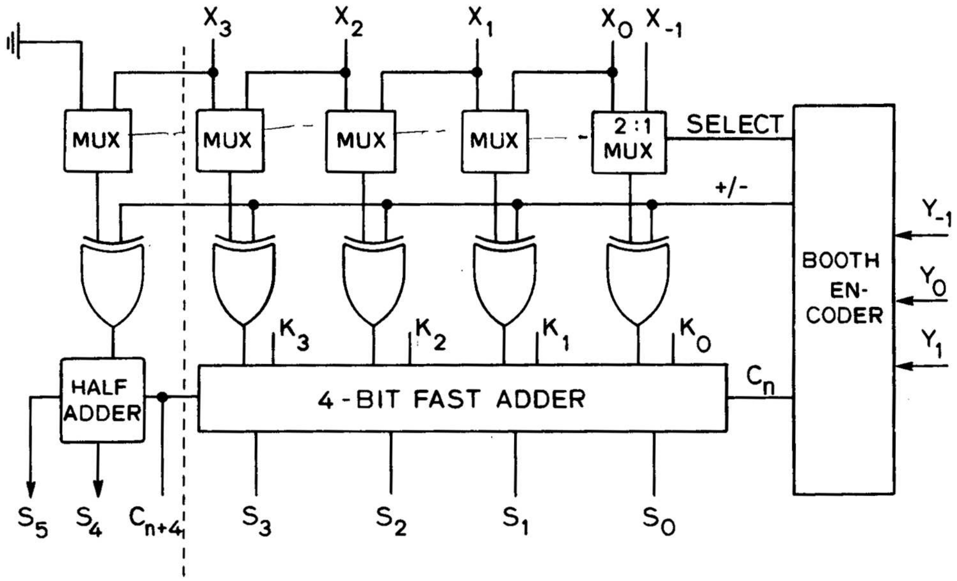


Figure 4.21: Block diagram of 2 × 4 iterative multiplier.

2. Reduction of the partial products to two operands—(5,5,4) counter.
3. Adding the two operands—adders with carry look ahead.

Figure 4.23 depicts the generation of the partial products in a 64 × 64 multiplication, using 8 × 8 multipliers. Each of the two 64-bit operands is made of 8 bytes (byte = 8 bits), which are numbered 0, 1, 2, . . . , 7 from the least to the most significant byte. Thus, 64-bit multiplication involves multiplying each byte of the multiplicand (X) by all 8 bytes of the multiplier (Y). For example, in Figure 4.24, the eight rectangles marked with a dot are those generated from multiplying byte 0 of Y by each of the 8 bytes of X. Note that the product “01” is shifted 8 bits with respect to product “00,” as is “02” with respect to “01,” and so on. These 8-bit shifts are due to the shifted position of each byte within the 64-bit operand. Also, note that for each $X_i \cdot Y_j (i \neq j)$ byte multiplication, there is a corresponding product $X_j Y_i$ with the same weight. Thus, product “01” corresponds to product “10” and product “12” corresponds to product “21.” As before, for $N \times M$ multiplication, the number of $n \times m$ multipliers required is:

$$\frac{N \times M}{n \times m,}$$

and in our specific case:

$$\text{Number of multipliers } = \frac{64 \times 64}{8 \times 8} = 64 \text{ multipliers.}$$

The next step is to reduce the partial products to two operands. As shown in Figure 4.23, a $(5, 5, 4)$ can reduce two columns, each 5 bits high, to two operands. The matrix of the partial

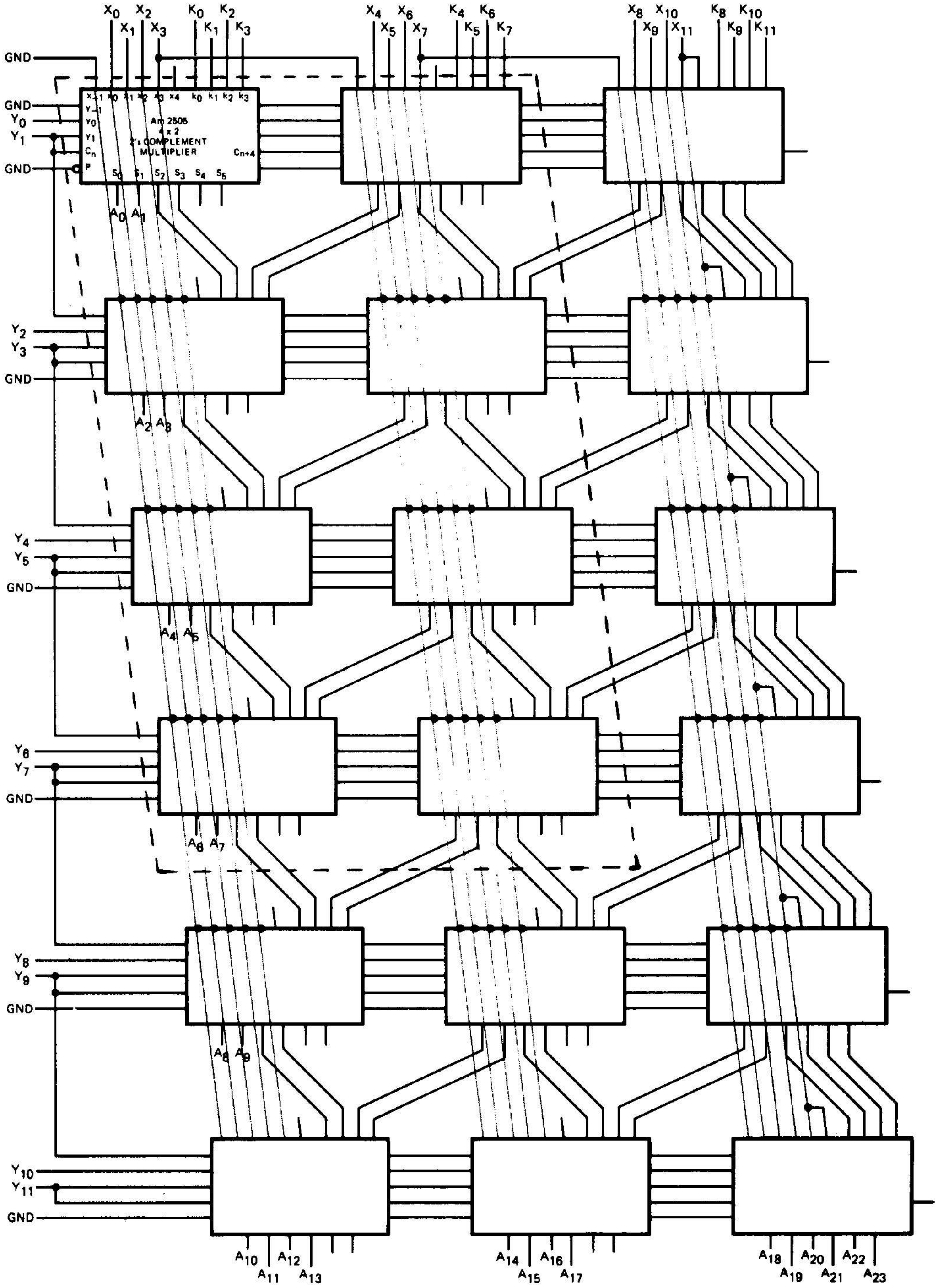


Figure 4.22: 12 × 12 two's complement multiplication A = X · Y + K. Adapted from [Ghest, 1971].

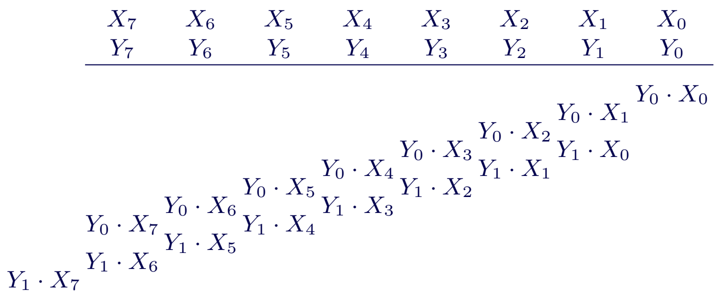


Figure 4.23: A 64×64 multiplier using 8×8 multipliers. Only 16 of the 64 partial products are shown. Each 8×8 multiplier produces a 16–bit result.

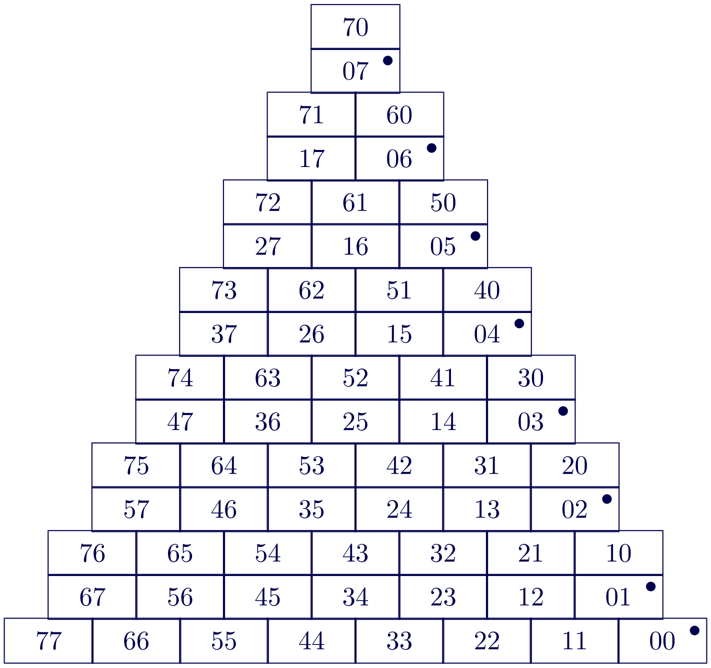


Figure 4.24: Partial products generation of 64×64 multiplication using 8×8 multipliers. Each rectangle represents the 16-bit product of each 8×8 multiplier. These partial products are later reduced to two operands, which are then added by a CPA adder. Each box entry above corresponds to a partial product index pair; e.g., the 70 corresponds to the term $Y7 \cdot X0$.

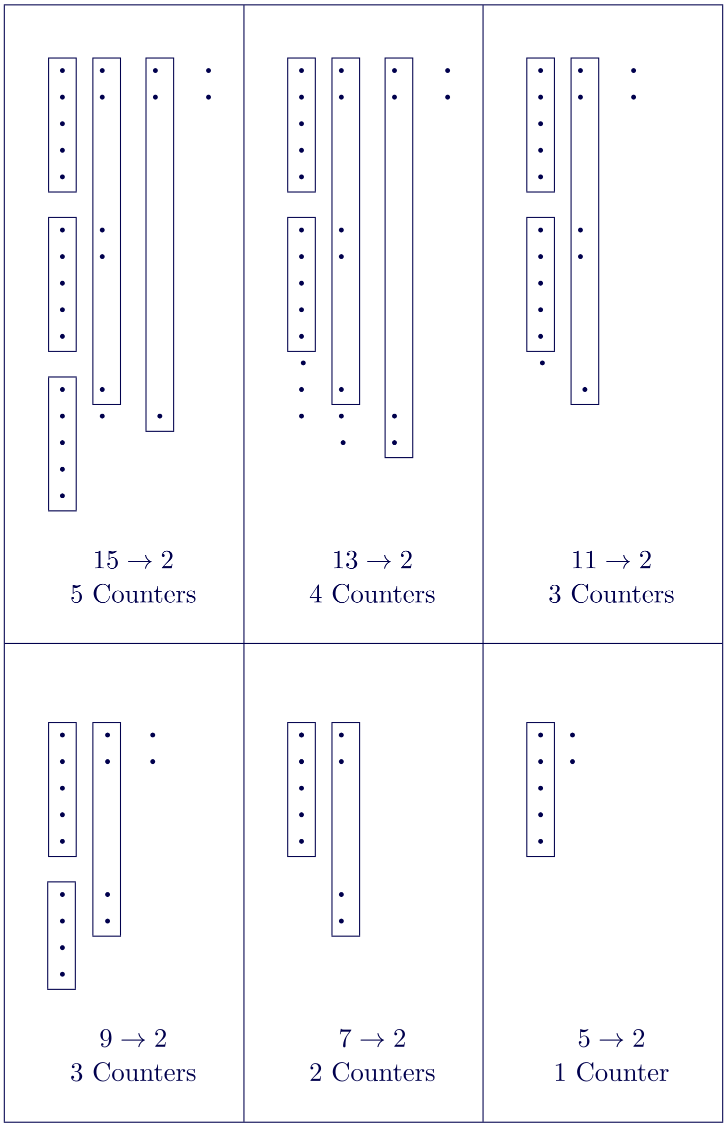


Figure 4.25: Using (5,5,4)s to reduce various column heights to 2 bits high. The 15 → 2 shows the summation as required in Figure 4.23 (height 15). Each other height required in Figure 4.24 is shown in 5,5,4 implementation.

products in Figure 4.24 can be viewed as eight groups of 16 columns each. The height of the eight groups is equal to 1, 3, 5, 7, 9, 11, 13 and 15. Figure 4.25 illustrates with a dot representation the use of (5,5,4)s to reduce the various column heights to 2 bits high. We now can compute the total number of counters required to reduce the partial product matrix of a 64×64 multiplication to two operands:

Number of Columns	Height of Columns	Number of Counters per Two Columns	Number of Counters for all Columns of Same Height
16	15	5	$8 \times 5 = 40$
16	13	4	$8 \times 4 = 32$
16	11	3	$8 \times 3 = 24$
16	9	3	$8 \times 3 = 24$
16	7	2	$8 \times 2 = 16$
16	5	1	$8 \times 1 = 8$
16	3	1	$8 \times 1 = 8$
16	1	—	—
Total number of counters			152

The last step in the 64×64 multiplication is to add the two operands using 4-bit adders and carry look ahead units, as described earlier. Since the double length product is 128 bits long, a 128-bit carry look ahead (CLA) adder is used.

4.5.2 Design Details of a 56×56 Single Length Multiplier

The last section described a 64×64 multiplier with double length precision. In this section, we illustrate the implementation of single length multiplication and the hardware reduction associated with it, as contrasted with double length multiplication. We select 56×56 multiplication because 56 bits is a commonly used mantissa length in long floating point formats.

A single length multiplier is used in fractional multiplication, where the precision of the product is equal to that of each of the operands. For example, in implementing the mantissa multiplier in a hardware floating point processor, input operands and the output product have the range and the precision of:

$$0.5 \leq \text{mantissa range} \leq 1 - 2^{-n},$$

$$\text{mantissa precision} = 2^{-n},$$

where n is the number of bits in each operand.

Figure 4.27 shows the partial products generated by using 8×8 multipliers. The double length product is made of 49 multipliers, since

$$\text{Number of multipliers } = \frac{56 \times 56}{8 \times 8} = 49.$$

However, for single precision, a substantial saving in the number of multipliers can be realized by "chopping" away all the multipliers to the right of the 64 *MSB*s (shaded area); but we need

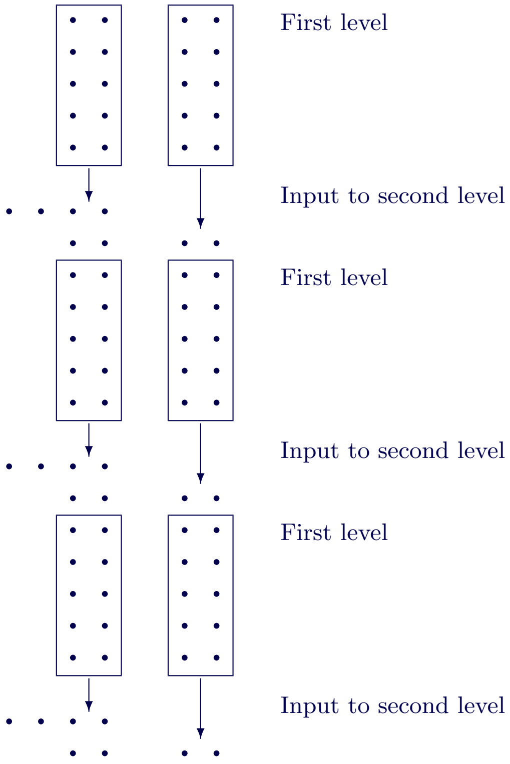


Figure 4.26: Using (5,5,4) counters to implement the reduction of the partial products of height 15, showing only the first level and its output.

This maximum height occurs when partial products 70 ($Y7 \cdot X0$) through 44 (or 33) are to be summed (Figure 4.24). For example, the top 5,5,4 counter above would have inputs from 70, 07, 71, 17, 61, the middle counter inputs from 16, 62, 26, 52, 25, and the lowest from 53, 35, 43, 34, and 44. The three counters above provide three 4-bit outputs (2 bits of the same significance, 2 of higher). Thus, six outputs must be summed in the second level: three from the three shown counters, and three from the three lower order counters.

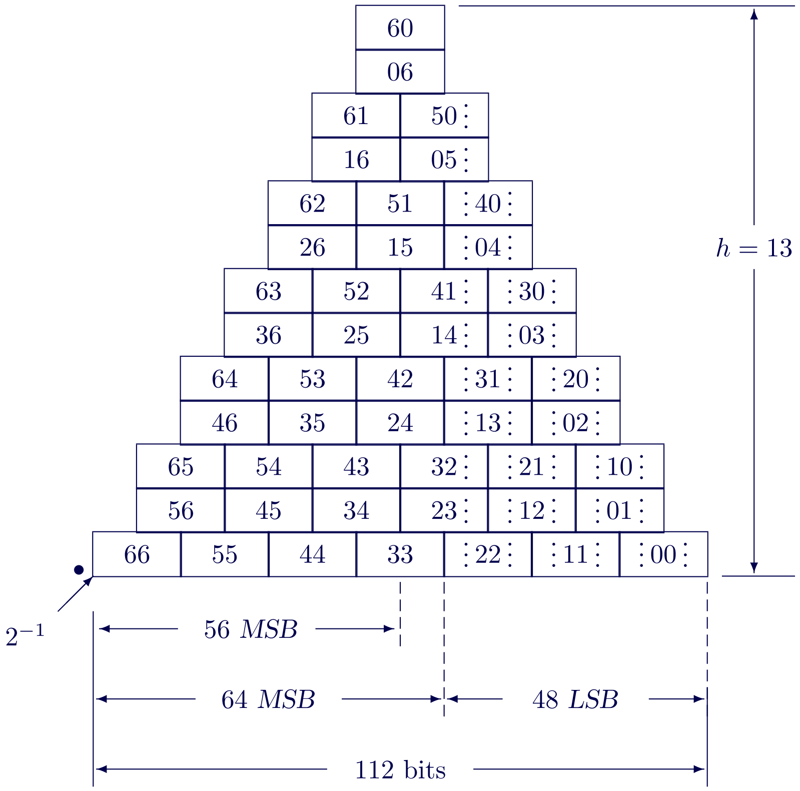


Figure 4.27: Partial products generation in a 56×56 multiplication, using 8×8 multipliers. The shaded multipliers can be removed when a single length product (56 *MSB*s) is required.

to make sure that this chopping does not affect the precision of the 56 *MSB*s. Assume a worst case contribution from the "chopped" area; that is, all ones. The *MSB* of the discarded part has the weight of 2^{-65}, and the column height at this bit position is 11. Thus, the first few terms of the "chopped" area are:

$$\begin{array}{rcl} \text{Max error} & = & 11 * (2^{-65} + 2^{-66} + 2^{-67} + \ldots) \\ \text{But:} & & 11 < 2^4 \\ \text{Therefore:} & & \text{Max error} < (2^{-61} + 2^{-62} + 2^{-63} + \ldots) \end{array}$$

From the last equation, it is obvious that "chopping" right of the 64 *MSB*s will give us 60 correct *MSB*s, which is more than enough to handle the required 56 bits plus the 2 guard bits.

Now we can compute the hardware savings of single length multiplication. From Figure 4.27, we count 15 fully shaded multipliers. We cannot remove the half shaded multipliers, since their most significant half is needed for the 58-bit precision. Thus, a total of 34 instead of 49 multipliers is used for the partial product generation.

Using the technique outlined previously, the number of counters needed for column reduction is easily computed for double and single length.

Number of Columns		Height of Columns	Number of Counters per Two Columns	Number of Counters for All Columns of Same Height	
Double Length	Single Length			Double Length	Single Length
16	16	13	4	32	32
16	8	11	3	24	12
16	8	9	3	24	12
16	8	7	2	16	8
16	8	5	1	8	4
16	8	3	1	8	4
16	8	1	—	—	—
Total Number of Counters				112	72

Finally, adding the resulting two operands in the double length case (112 bits) requires a carry look ahead (CLA) adder. For the single length case, the addition of the two 64 bit operands is accomplished by a 64-bit CLA adder. There is a 34% hardware savings using the single length multiplication. However, there is no speed improvement to a first approximation.

4.6 Problems

Problem 4.1 Design a modified Booth's encoder for a 4-bit multiplier.

Problem 4.2 Find the delay of the encoder in problem 4.1.

Problem 4.3 Construct an action table for modified Booth's algorithm (2-bit multiplier encoded) for sign and magnitude numbers (be careful about the sign bit).

Problem 4.4 Design a modified Booth's encoder for sign and magnitude numbers (4-bit multiplier encoded).

Problem 4.5 Construct the middle bit section of the CSA tree for 48 × 48 multiplication for:

1. Simple iteration.
2. Iterate on tree.
3. Iterate on lowest level of tree.

Problem 4.6 Compute the number of CSA's required for:

1. 1-bit section.
2. Total tree.

Problem 4.7 Derive an improved delay formula (taking into account the skewed significance of the partial products) for:

1. Full Wallace tree.
2. Simple iteration.
3. Iterate on tree.
4. Iterate on lowest level of tree.

Problem 4.8 Suppose 256 × 8 ROMs are used to implement 12 bit × 12 bit multiplication. Find the partial products, the rearranged product matrix, and the delay required to form a 24 bit product.

Problem 4.9 If (5, 5, 4) counters are used for partial product reduction, compute the number of counter levels required for column heights of 8, 16, 32, and 64 bits. How many such counters are required for each height?

Problem 4.10 Suppose (5, 5, 4) counters are being used to implement an iterative multiplication with a column height of 32 bits. Iteration on the tree only is to be used. Compare the tree delay and number of counters required for one, two, and four iterations.

Problem 4.11 Refer to Figure 4.4. Develop the logic on the A_9, B_9, C_9, and D_9 terms to eliminate the required summing of these terms to form the required product. Assume X and Y are 8-bit unsigned integers.

Problem 4.12 Refer to Table 4.1. Show that this table is valid for the two's complement form of the multiplier.

Problem 4.13 Implement a $(5, 5, 4)$ counter using CSAs only. The first priority is to minimize the number of levels used. The second priority is to minimize the number of CSAs.

In your optimum design, show in dot representation all the CSAs in each level. What is the time delay of your implementation? (Assume that the CSA is equivalent to two gate delays.)

Problem 4.14 We wish to build a 16×16 bit multiplier. Simple AND gates are used to generate the partial products ($t_{\text{max}} = 20$ ps, $t_{\text{min}} = 10$ ps).

1. Determine the height of the partial product tree.
2. How many AND gates are required?
3. How many levels of (5,5,4) counters are required? Estimate the total number of counters.
4. If the counters have an access time of $\text{P}_{\text{max}} = 30$ ps, $\text{P}_{\text{min}} = 22$ ps, determine the latency of the multiplier (before CPA).
5. An alternative design for the multiplier would use iteration of the partial product tree. If only one level of (5,5,4) counters is used, how many iterations are required?
6. How many gates are needed for PP generation and how many ROM's for PP reduction?

Problem 4.15 It has been suggested that $1^K \times 8^b$ ROMs could be used as counters to realize a 32-bit CPA ($C_{\text{in}} = 0$). Design such a unit. Show each counter and its DOT configuration or $(C_R, \ldots, d)$ designation. Clearly show how its input and output relate to other counters.

Minimize (with priority):

1. The number of ROM delays.
2. The total number of ROMs.
3. The number of ROM types.

What are the number of ROM delays, the total number of ROMs, and the number of ROM types? Show a complete design.

Problem 4.16 Implement a $(7, 3)$ counter using only CSAs. Use a minimum number of levels and then a minimum number of CSAs.

1. Show in dot representation each CSA in each level.
2. Repeat for a (5,5,4) counter.

Problem 4.17 We want to perform 64×64 bit multiplicand using 8×8 PP generators. What is the PP tree height?

Suppose the tree reduction is to be done by (7,3) counters. Assume a three input CPA is available. Show the counter reduction levels for the max column height (h). How many (7,3) counter levels are required?

Suppose we now wish to iterate on the tree using one level of (7,3) counters.

Suppose PP generation takes $P_{\max} = 400\,\text{ps}$, (7,3) counter takes $P_{\max} = 200\,\text{ps}$, and three input CPA takes $P_{\max} = 300\,\text{ns}$.

How would you arrange to assimilate (reduce) the PP's and find the product? Show the number of PP reduced for each iteration.

The total time for multiplication is how many nsec?

Problem 4.18 Design a circuit to compute the sum of the number of ones in a 128^{b} word, using only $64^{\text{K}} \times 8^{\text{b}}$ ROM's (no CPA's). You should minimize the number of levels, then the number of parts, then the number of types of ROM configurations (contents). Use dot notation.

Problem 4.19 In signal processing and graphics applications it is often required to produce the sum of squares of two numbers: $x^2 + y^2$. You want to design an efficient sum of squares circuit for operands that are 4 bits long.

1. Symbolically, write the full partial products of x^2 and y^2 then minimize them as one partial product array, i.e. do not get each term alone but rather the complete $x^2 + y^2$ together in one PPA.

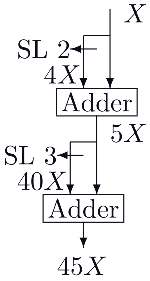
2. Use a dot diagram to indicate how you can connect carry save adders to reduce the PPA.

3. How many CSAs do you need for the whole operation? How many levels of CSAs exist? Estimate the total delay of the operation including the final CPA.

Problem 4.20 A method to perform a DRC multiply (1's complement) has been proposed. It uses an unsigned multiplier which takes two n-bit operands and produces a $2n$-bit result, and a CLA adder which adds the high n-bits to the low n-bits of the multiplier output. Don't worry about checking for overflow. Assume the DRC result will fit in n bits.

Either prove that this will always produce the correct DRC result, or give an example where it will fail.

Problem 4.21 In digital signal processing, there is often a requirement to multiply by specific constants. One simple scheme is to use sub-multiples of the form 2^k or $2^k \pm 1$ to achieve the required operation. As an example, $45X$ is calculated as $5X \times 9 = (2^2+1)X \times (2^3+1)$. First, X is shifted by two bit locations (a simple wiring no gates needed) and added to itself. Then the result is shifted by three bit locations (again no gates) and added to itself. The total hardware needed is two adders.



Multiplication by 45. Shifting is connecting the wires to the required locations.

Assume that in a specific design you need to have four products: $P_1 = 9X$, $P_2 = 13X$, $P_3 = 18X$, $P_4 = 21X$. Show how do you divide the different products. How many adders do you need if each product is done separately? A friend shared the common subexpressions between the required products and claims to complete the four products using only five adders. As a challenge, you try to share more hardware between the units in an attempt to use only four adders, can you?

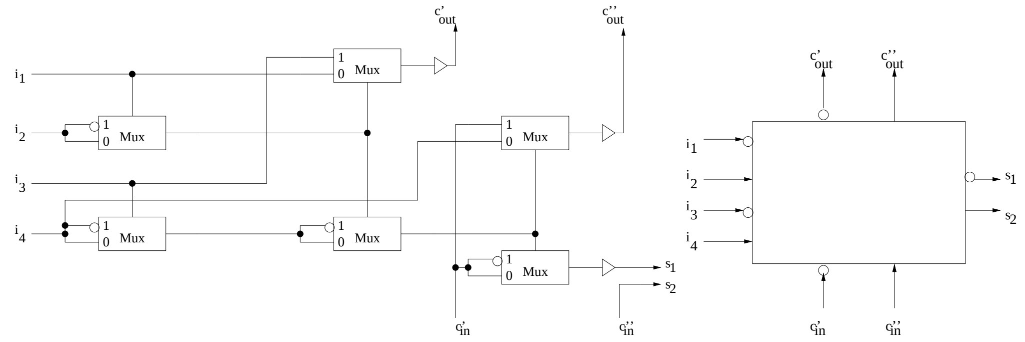
Problem 4.22 The boxes in Fig. 4.28(a) are multiplexers implemented using CMOS pass gates, the triangles are buffers to keep the signal strength and the the circles at the inputs of some multiplexers are inverters. The inputs are $i_1, i_2, i_3, i_4, c'_{in}$ and c''_{in} while the outputs are c'_{out}, c''_{out}, s_1, and s_2.

1. Please indicate the mathematical relation between the inputs and the outputs as implemented by this circuit. Hint: the relation is an equation on one line that has the inputs on the right hand side and the outputs on the left hand side. Only arithmetic (not logical) operators and possibly a multiplication by a constant exist in the relation.
2. If the box of Fig. 4.28(b) is implemented by the circuit of Fig. 4.28(a), now indicate the new mathematical relation.
3. What do you get if you connect an array of the circuit of Fig. 4.28(b) using vertical connections between c_{in} and c_{out}? What do you get if you connect a tree of those arrays using horizontal connections between i and s?
4. Do you still need the inversions anywhere in the whole tree or can you use the circuit of Fig. 4.28(a) everywhere?
5. In your opinion, is the multiplication of signed-digit operands more complicated, less complicated or about the same when compared to regular operands? Why?

Problem 4.23 In an advanced design for a multiplicative division, there is a need to multiply one unsigned number X by another unsigned number Y but in a special manner. The bits of Y are separated to two parts Y_h the most significant part and Y_l the least significant part.

$$Y = \overbrace{y_n \dots y_{i+2} y_{i+1}}^{Y_h} \underbrace{y_i \dots y_1 y_0}_{Y_l}$$

It is required to multiply $X(Y_h - Y_l)$ using a modified Booth-2 algorithm on the original bits of Y without performing a subtraction.



(a) Multiplexer implementation

(b) Using inverters at some inputs and outputs

Figure 4.28: Building blocks for problem 4.22

1. Given that X and Y are unsigned, how are you going to pad the MSB and the LSB?
2. Write the truth table for the Booth recoder within the Y_h part.
3. Write the truth table for the Booth recoder within the Y_l part.
4. Taking into consideration that i may be odd or even, write the truth table for the boundry region between Y_h and Y_l.

Problem 4.24 You are given the following arrangement of partial product bits where $\bar{x}_i = 1 - x_i$ and $\bar{a}_i = 1 - a_i$.

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					$a_4\bar{x}_0$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				$a_4\bar{x}_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			$a_4\bar{x}_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		$a_4\bar{x}_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	$\bar{a}_3x_4$	$\bar{a}_2x_4$	$\bar{a}_1x_4$	$\bar{a}_0x_4$				
	$\bar{a}_4$	0	0	0	a_4				
1	$\bar{x}_4$	0	0	0	x_4				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

1. Prove that this arrangement produces the correct product of two five bits two's complement operands.
2. If the operands were n bits long each, how many rows and columns does such a multiplier have?
3. What are the formulas for obtaining the partial product bit at row i and column j?
4. Briefly compare this multiplier from the area and speed points of view to the parallel multipliers that you studied.

Problem 4.25 You are given the following arrangement of partial product bits where $\overline{a_j x_i} = 1 - (a_j x_i)$.

$$\begin{array}{cccccccccc}
 & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
 & & & & & \overline{a_4x_0} & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & \overline{a_4x_1} & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & \overline{a_4x_2} & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & \overline{a_4x_3} & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \overline{a_3x_4} & \overline{a_2x_4} & \overline{a_1x_4} & \overline{a_0x_4} & & & & \\
1 & & & & 1 & & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

1. Prove that this arrangement produces the correct product of two five-bit two's complement operands.
2. In its current arrangement, the multiplier seems to have six rows. Re-arrange it to have five rows only.
3. Without your re-arrangement, if the operands were each n bits in length, how many rows and columns will the multiplier have?
4. Without your re-arrangement, what are the formulas for obtaining the partial product bit in row i and column j?

Problem 4.26 A hardware accelerator for the following sum of squares loop

for(i=0;i<n;i++)

sum = x[i]*x[i] + y[i]*y[i] +sum

implements a hardware unit to calculate $s = x^2 + y^2 + z$. Both x and y are integers in two's complement format which have n bits while z is an unsigned integer with $2n$ bits.

1. Prove that s is represented correctly using $2n + 1$ bits.
2. Assuming $n = 4$ and labeling the bits of x as x_3, x_2, x_1, and x_0 (from the most to the least significant), use the multiplier arrangement of problem 4.25 to show the partial products array of the x^2 part and reduce it to the minimum number of rows by grouping related terms and moving them to other columns when necessary.
3. Using your result for x^2, show the partial product matrix of the whole computation of s (i.e. without using any carry propagate adders for the different terms) and reduce the constant values to get the minimum number of rows.
4. Use the dot notation to find the minimum number of levels of (3,2) counters needed to reduce that matrix of s to two rows only.

Problem 4.27 In a certain design we need to multiply an unsigned integer binary number by powers of ten. Since $10 = 8 + 2$, we can shift the number to the left by three bits and add another version of the number that is only shifted by one bit to get the result. In this case, we used two bit vectors (the shifted by 3 and the shifted by 1).

1. Given that you can either add or subtract, find a similar way to multiply by $10^2 = 100$ that minimizes the number of bit vectors you use. (Multiply directly, do not repeat the multiplication by ten twice.)

2. Repeat for $10^3 = 1000$ using at most the same number of bit vectors.

3. Design a 'power of ten' multiplier. When given an unsigned binary number and the power of ten, the circuit produces the result of multiplying that binary number by ten raised to the given power. Assume that the binary number has 'n' bits and the power of ten is represented by two bits p_1p_0 that take the values 00, 01, 10, and 11 (binary for 0, 1, 2, and 3).

Chapter 5

Floating over the vast seas

5.1 Motivation and Terminology; or the *why?* and *what?* of floating point.

So far, we have discussed fixed point numbers where the number is written as an integer string of digits and the radix point is a function of the interpretation. In this chapter, we deal with a very important representation of numbers on digital computers, namely the floating point representation. It is important because nearly all the calculations involving fractions use it. Floating point representations have also been the subject of several proprietary as well as international standards. We will attempt to contrast those standards and expose the advantages and disadvantages of each. However, before that, we need to know why did humans use a floating point representation from the first place and what is it exactly that is floating?

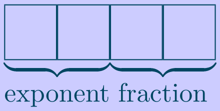
The *dynamic range*, which is one of the properties considered in the choice of a number system, is the ratio of the largest magnitude to the smallest non-zero magnitude representable in the system. The problem with fixed point arithmetic is the lack of dynamic range as illustrated by the following example in the decimal number system.

Example 5.1 In a system with four decimal digits the numbers range from 9 999 to 0 with a dynamic range of $9\,999 \approx 10\,000$. This range is independent of the decimal point position, that is, the dynamic range of 0.9999 to 0.0000 is also $\approx 10\,000$. Since this is a 4-digit number, we may want to represent during the same operation both 9 999 and 0.0001. This is, however, impossible to do in fixed point arithmetic without some form of scaling.

The motivation for a floating point representation is thus to have a larger dynamic range. The floating point representation is similar to scientific notation; that is: fraction $\times$ $(\text{radix})^{\text{exponent}}$.

For example, the number 99 is expressed as 0.99×10^2. In a computer with floating point instructions, the radix is implicit, so only the fraction and the exponent need to be represented explicitly.

Example 5.2 A possible floating point format for a system with four decimal digits is:



What is the dynamic range in this representation assuming positive numbers and positive exponents only?
Solution: Note that the dynamic range is the ratio of the largest number representation to the smallest (nonzero) number representation. Hence,

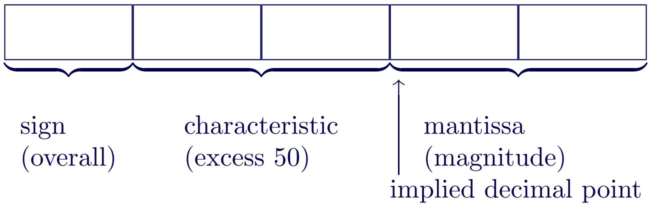
1. Smallest (nonzero) number is $0.01 \times 10^0 = 0.01$.
2. Largest number is 0.99×10^{99}, approximately 10^{99}.
3. Therefore, the dynamic range is approximately $10^{99}/0.01 = 10^{101}$.

Thus, in a floating point representation, the dynamic range is several orders of magnitude larger than that of a fixed point representation.

In practice, floating point numbers may have a negative fraction and negative exponent. There are many formats to represent such numbers, but most of them have the following properties in common:

1. The fraction is an unsigned number called the *mantissa*.
2. The sign of the entire number is represented by the most significant bit of the number.
3. The exponent is represented by a *characteristic*, a number equal to the exponent plus some positive bias.

The following format is an extension of the previous example:



The excess code is a method of representing both negative and positive exponents by adding a bias to the exponent. In the case of binary or binary-based radix ($\beta = 2^k$) where n_{exp} is the number of exponent bits, designers usually choose the bias value close to

$$\frac{1}{2}2^{n_{exp}}$$

For a non-binary based radix with n_{exp} the number of exponent digits the bias is usually taken as $\frac{1}{2}\beta^{n_{exp}}$.

Example 5.3 For the above format where two decimal digits are used for the exponent,

$$\text{bias } = \frac{1}{2}(10)^2 = 50.$$

The biased exponent is called the characteristic and is defined as:

$$\text{Characteristic} = \text{exponent} + \text{bias}.$$

Hence, an exponent of 2 results in

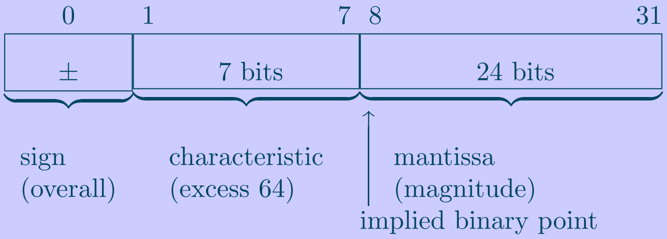
$$\text{Characteristic} = 2 + 50 = 52.$$

Since, by definition in most systems, the number zero is represented by an all zeros bit string, the advantage of excess representation for the exponent is therefore that smaller numbers (i.e., with a negative exponent) uniformly approach zero. Such a scheme simplifies the comparison logic.

The mantissa is the magnitude of the fraction, and its sign is the *MSD* of the format. Usually, a 0 in the sign digit signifies a positive number while a 1 signifies a negative number.

The same approach is used in binary floating point numbers.

Example 5.4 Consider a 32-bit word, where 24 bits are the unsigned mantissa, 7 bits are the characteristic, and the *MSB* is the sign of the number, as follows:



What is the range of the representable numbers as determined by the exponent?
Solution: The largest exponent is $127 - 64 = 63$ and $2^{+63} \simeq 10^{+19}$.
The smallest exponent is $0 - 64 = -64$ and $2^{-64} \simeq 0.5 \times 10^{-20}$.

5.2 Properties of Floating Point Representation

5.2.1 Lack of Unique Representation

So far, we did not clearly indicate why is such formats called *floating point*.

In fact, the word *floating* is used here with the meaning "continually drifting or changing position" and the *point* to which we refer is the fractional point delimiting the integer part of a number from its fractional part. The reason this point is floating is that a normalized scientific notation is often used.

Generally, the magnitude of a floating point number is evaluated by $M \times \beta^{exp}$, where

$$\begin{aligned} M &= \text{mantissa,} \\ \beta &= \text{radix, and} \\ exp &= \text{exponent.} \end{aligned}$$

A floating point system must represent the number zero. According to the definition $M \times \beta^{exp}$, zero is represented by a zero in the fraction and any exponent. Thus, it does not have a unique representation. However, in many systems a "canonic" zero is defined as an all zeros digit string so that both the fraction and the exponent have zero values.

In fact, any number may have multiple representations according to our definition so far. So 0.9 may be represented as 0.9×10^0 or 0.09×10^1. Most floating point systems define a normalized number representation. In such a represention, a number is represented by one non-zero digit preceding the fractional point and the subsequent digits following the point multiplied by the radix of the system raised to some exponent ($d_0.d_{-1} \cdots d_{-n} \times \beta^{exp}$, with $d_0 \neq 0$.) An alternative definition is to say that the number is represented by the digit zero preceding the fractional point and then a fractional part starting with a non-zero digit and all of that multiplied by the radix raised to some exponent ($0.d_{-1}d_{-2} \cdots d_{-n} \times \beta^{exp}$, with $d_{-1} \neq 0$.). For example, in decimal, the number three hundred and forty two can be represented as 3.42×10^2 according to the first definition and as 0.342×10^3 according to the second definition. Using the first definition, if we multiply 3.42×10^2 by 10 the result of 34.2×10^2 must be normalized to 3.42×10^3. The fractional point changed its position after this multiplication, hence the name floating point.

In both definitions of normalization mentioned above, the number zero cannot be correctly represented as a *normalized* number since it does not have any non-zero digits and needs a special treatment. In many systems, the canonical zero is, by definition, the canonic zero defined earlier as an all zeros string. This definition also simplifies the zero detection circuitry. It is interesting to note that a canonical zero in floating point representation is designed to be identical to the fixed point representation of zero.

The two definitions of normalization have been historically used in different implementations. Since the location of the point is implied, it is important to know which definition of normalization is used on a specific computer in order to interpret the bit string representing a number correctly.

Example 5.5 Here are some representations and their values according to a simple format with one digit for the sign, two decimal digits for the exponent (excess 50), and two decimal digits for the mantissa. In this example, the implied point is to the left of the *MSD* of the mantissa as in the second definition of normalization mentioned above.

0 51 78→$+0.78\times10^1$	$= +7.8$	normalized
0 52 07→$+0.07\times10^2$	$= +7.0$	unnormalized
0 47 12→$+0.12\times10^{-3}$	$= +0.00012$	negative exponent
1 51 78→-0.78×10^1	$= -7.8$	negative number
0 52 00→$+0.00\times10^2$	$= +0.0$	non-canonical zero
0 00 00→$+0.00\times10^0$	$= +0.0$	canonical zero

Only mantissas of the form $0.xxx\cdots$ are fractions in reality. When discussing both fraction and other mantissa forms (as in $1.xxx$), people tend to use the more general term *significand*.

5.2.2 Range and Precision

The range is a pair of numbers (smallest, largest) which bounds all representable numbers in a given system. Precision, on the other hand, indicates the smallest difference between the mantissas of any two such representable numbers.

The largest number representable in any normalized floating point system is approximately equal to the radix raised to the power of the most positive exponent, and the absolute value of the smallest nonzero number is approximately equal to the radix raised to the power of the most negative exponent.

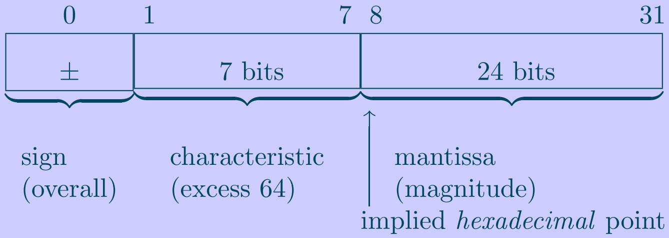
Assuming $M_{\mathbf{max}}$ and $exp_{\mathbf{max}}$ to be the largest mantissa and exponent respectively, we write the largest representable number as:

$$\mathbf{max} = M_{\mathbf{max}} \times \beta^{exp_{\mathbf{max}}}$$

Simililarly, we get the minimum representable number **min** from the minimum *normalized* mantissa $M_{\mathbf{min}}$ and the minimum exponent $exp_{\mathbf{min}}$:

$$\mathbf{min} = M_{\mathbf{min}} \times \beta^{exp_{\mathbf{min}}}$$

Example 5.6 The following IBM System 370 (short) format is similar to the binary format of example 5.4, except that the IBM radix is 16. What is **max** and **min**?



(In the IBM formats, a normalized number has a zero integer part and a fractional part starting with a non-zero digit.)
Solution: Since $\beta = 16$, the 24 bits of the mantissa are viewed as 6 digits each with 4 bits. The maximum mantissa $M_{\mathbf{max}}$ is thus $0.FFFFFF)_{hex} = 1 - 16^{-6}$. The characteristic, however, is still read a regular binary number. Hence, $exp_{\mathbf{max}} = 63$. Therefore, the largest representable number is

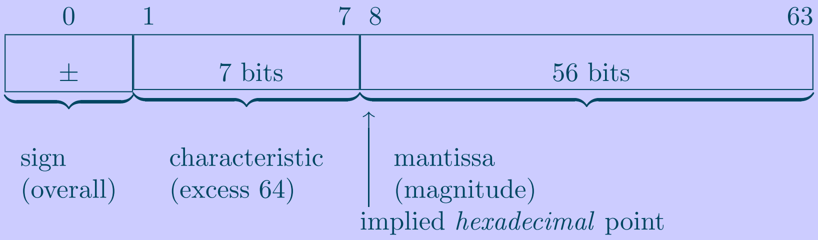
$$\mathbf{max} = 16^{63} \times (1 - 16^{-6}) \simeq 7.23 \times 10^{75}$$

Similarly, the smallest positive normalized number is

$$\mathbf{min} = 16^{-64} \times (16^{-1}) \simeq 5.4 \times 10^{-79},$$

For a given radix, the range is mainly a function of the exponent. By contrast, the precision is a function of the mantissa. *Precision* is the resolution of the system, and it indicates the minimum difference between two mantissa representations, which is equal to the value of the least significant bit of the mantissa. Precision is defined independently of the exponent; it depends only on the mantissa and is equal to the maximum number of significant digits representable in a specific format. In the IBM short format, there are 24 bits in the mantissa. Therefore, the precision is six hexadecimal digits because $16^{-6} = 2^{-24}$. If we convert this to human understandable numbers $2^{-24} \simeq 0.6 \times 10^{-7}$, or approximately seven significant decimal digits. In the litterature, some prefer to express the precision as the difference between two consecutive mantissas so that in the previous example, it would be 16^{-6} and not six.

Example 5.7 More precision is obtained by extending the number of bits in the mantissa; for example, in the IBM System 370, one more word is added to the mantissa in its long format, that is, 32 more bits. The mantissa is 56 bits long and the unit in the last place is 16^{-14} or $2^{-56} \simeq 10^{-17}$. The format with an extended mantissa is commonly called double precision, but in reality the precision is more than doubled. In the IBM case, this is 17 versus 7 decimal digits.



$\overset{?}{\Longrightarrow}$ **Exercise 5.1 What is max and min for the ibm S/370 double format described above?**

5.2.3 Mapping Errors: Overflows, Underflows, and Gap

As discussed in section 1.1, the finitude of the machine number system poses a few challenges. In practice, the problem of overflow in a floating point system is much less severe than in a fixed point system, and most business-type applications are never aware of it. For example, the budget of the U.S. Government, while in the trillions of dollars, requires only thirteen decimal digits to represent—well within the capability of the IBM S/370 floating point format. By contrast, in many scientific applications [Sterbenz, 1974], the computation results in overflows; for example, $e^{200} > 10^{76}$, therefore, e^{200} cannot be represented in the IBM floating point system.

Similarly, $(0.1)^{200}$ cannot be represented either, since $(0.1)^{200} = 10^{-200}$, and the smallest representable number is approximately 10^{-76}. The latter situation is called *underflow*. Thus, mapping from the human infinite number system to a floating point system with finite range may result in an unrepresentable exponent (exponent spill). The exponent spill is called overflow if the absolute value of the result is larger than **max**, and it is called underflow if the absolute value of the result is smaller than **min**.

In order to allow the computation to proceed in a reasonable manner after an exponent spill, a designer might replace an underflow by a canonical zero and an overflow by the largest signed representable number. However, one might also produce a specific bit pattern representing $\pm\infty$, and from that point on this overflow is treated as a genuine $\pm\infty$, for example, $X \div \infty = 0$.

These approximations should not be confused with the computations through overflows in fixed point representation. The latter always produce a correct result, whereas the floating point approximation of overflow and underflow always produces an incorrect result; but this incorrect result may have an acceptable error associated with it.

For example, in computing a function using polynomial approximation, some of the last terms may underflow, but by setting them to zero, no significance is lost. The case in point [Sterbenz, 1974] is $\sin X \simeq X$, which for $|X| < 0.25 \times 16^{-32}$ is good to over 65 hexadecimal digits.

So far, we have discussed the consequences of mapping out of range numbers, and have shown that the resulting overflow or underflow is a function of the range; that is, the exponent portion of the floating point number. Now consider the consequences of the fact that the floating point number system can represent only a finite subset of the set of real numbers. For simplicity, assume that all the real numbers are within the range of floating point numbers; thus, the error in mapping is a function of the mantissa resolution.

For a base β floating point number system with a t-digit mantissa, the value of the gap between two successive normalized floating point numbers is $\beta^{-t}\beta^{exp}$, where exp is the value of the exponent [Garner, 1976]. The gap is thus related to the precision which we defined earlier as β^{-t}. However, while the precision is a function of the mantissa alone, the gap is also a function of the value of the exponent. It is important to note that with an increase in the exponent value by one, the gap between two successive numbers becomes β times larger. The precision is a constant defined by the format and the gap is a variable depending on the specific value of the exponent.

When we represent a real number that falls in the gap between two floating point numbers, we must map it to one of those two numbers. Hence, the magnitude of the mapping error is some fraction of the gap value.

Example 5.8 In the range of 0.5 to 0.999..., the IBM short format has a maximum mapping error (gap) of $2^{-24} \times 16^{0} = 2^{-24} \simeq 10^{-7}$, while the long IBM format reduces the mapping error to $2^{-56} \simeq 10^{-17}$.

5.3 Problems in Floating Point Computations

Our attempt here is to introduce the reader to some general problems that exist in all the floating point systems. These problems are exhibited to different extents in the various standards that we will present shortly. Once the problems are understood, a comparative analysis of the different systems becomes much easier to conduct. We tackle the problems in their logical order: first when we are representing a number in a floating point format, second when we are performing some calculations, and third when we are getting the result.

5.3.1 Representational error analysis and radix tradeoffs

As discussed earlier, it is impossible for a computer using a finite width representation of a weighted positional floating number system to accurately represent irrational numbers such as e and π. Even rational numbers that require a number of digits larger than what the system provides are not representable. For these cases, the closest available machine number is usually substituted. This is one form of representational error and it results from the lack of enough digits in the mantissa for the required significance. The other form of representational error stems from the finite range for the exponents. A number may lie beyond the range of the format which leads to an overflow or underflow error.

Note that a number that is representable in a finite number of digits in one radix may require an infinite number of digits in another radix. For example, if $\beta = 3$ then $(0.1)_{\beta=3} = \left(\frac{1}{3}\right)_{\beta=10} = (0.3333\cdots)_{\beta=10}$, i.e. the number of digits is finite for $\beta = 3$ and infinite for $\beta = 10$. Similarly, $(0.1)_{\beta=10} = (0.0001100110011001100\ldots)_{\beta=2}$. This means that the use of a finite number of digits to represent a number in a certain radix may lead to errors that will not occur if another radix is used. We will see the effect of these errors when we describe decimal floating point arithmetic.

So far, the range of the exponents, significand width, and the choice of the radix in the system were discussed independently. However, for a given number of bits in a format there is a tradeoff between them. Recall the previously mentioned 32 bit format with 24 bits of unsigned mantissa, 7 bits of exponent, and one sign bit. The tradeoffs between the different factors are illustrated by comparing a system with a radix of 16 (hexadecimal) against a system with a radix of 2 (binary system).

	Largest Number	Smallest Number	Precision	Accuracy
Hexadecimal system	7.2×10^{75}	5.4×10^{-79}	16^{-6}	2^{-21}
Binary system	9.2×10^{18}	2.7×10^{-20}	2^{-24}	2^{-24}

While the hexadecimal system has the same resolution as binary, hex-normalization may result in three leading zeros, whereas nonzero binary normalization never has leading zeros. Accuracy is the guaranteed or minimum number of significant mantissa bits excluding any leading zeros. This table indicates that for a given word length, there is a tradeoff between range and accuracy; more accuracy (base 2) is associated with less range, and vice versa (base 16). There is quite a bit of sacrifice in range for a little accuracy.

In base 2 systems there is also a property that can be used to increase the precision, a normalized number must start by 1. In such a case, there is no need to store this 1 with the rest of the bits. Rather, the number is stored starting from the following bit location and that 1 is assumed to be there by the hardware or software manipulating the numbers. This 1 is what we earlier termed the hidden one.

The literature about the precision of various floating-point number systems [Brent, 1973, Cody, 1973, Garner, 1976, Marasa and Matula, 1973, McKeeman, 1967] and the size of the significand part defines two kinds of representational errors: the maximum relative representation error (MRRE) and the average relative representation error (ARRE). The terminology and notation for those errors in the different papers of the literature are not consistent. Hence, we use a simple notation where t is the significand bit width in a system with exponent base β and derive the equations giving those two quantities for any real number x. Then, we proceed to use them in our analysis of the binary and hex-based systems. We assume here that all the t bits in the significand encode a single binary number. Formats that divide the significand into sub-blocks (such as one decimal floating point format described later) need a slightly different treatment.

Let $x = f_x \times \beta^{exp}$ be an exact representation of x assuming that f_x has as many digits as needed (even an infinite number of digits if needed) but that f_x is normalized according to the definition $1/\beta \leq f_x < 1$ (the reader is urged to try the other definition of $1 \leq f_x < \beta$ to check that we get similar results). Let the computer representation of x be $f_R \times \beta^{exp}$. Then the error of the representation is $error(x) = |f_x\beta^{exp} - f_R\beta^{exp}|$. The MRRE is defined as the maximum error relative to x, i.e.

$$MRRE = max(\frac{|f_x\beta^{exp} - f_R\beta^{exp}|}{f_x\beta^{exp}})$$

If the exact number is rounded to the nearest representation then the maximum $error(x)$ is equal to half the unit in the last position (**ulp**) or $max(error(x)) = 1/2 \times 2^{-t} \times \beta^{exp}$. Thus,

$$MRRE = max(\frac{1/2 \times 2^{-t}}{f_x})$$

The denominator should be set to its least possible value which occurs at $f_x = 1/\beta$. Hence the MRRE is given by

$$MRRE = \frac{1/2 \times 2^{-t}}{1/\beta} = 2^{-t-1}\beta$$

In the derivation of the formula for ARRE, we use half of the maximum error since it is assumed that the error is uniform in the range $[-\frac{1}{2}2^{-t}\beta^{exp}, \frac{1}{2}2^{-t}\beta^{exp})$ for any specific $f_x\beta^{exp}$. As for the distribution probability of the numbers $f_x\beta^{exp}$ in the system, it is assumed to be logarithmic and given by $p(f_x) = \frac{1}{f_x \ln \beta}$. This assumption is based on the work of McKeeman [McKeeman, 1967] who suggested that "during the floating point calculations, the distribution of values tends to cluster towards the lower end of the normalization range where the relative representation error tends to be the largest." To get the ARRE we perform the integration

$$\begin{aligned}
ARRE &= \int_{\frac{1}{\beta}}^{1} \frac{1/2 \times (1/2 \times 2^{-t})}{f_x} \frac{1}{f_x \ln \beta}\, df_x \\
&= \frac{2^{-t}}{4\ln\beta} \int_{\frac{1}{\beta}}^{1} \frac{df_x}{f_x^2} \\
&= \frac{2^{-t}}{4\ln\beta} \left[\frac{-1}{f_x}\right]_{\frac{1}{\beta}}^{1} \\
&= \frac{\beta - 1}{4\ln\beta} 2^{-t}
\end{aligned}$$

An analysis of both the MRRE and ARRE of the binary ($\beta = 2$) and hex-based ($\beta = 16$) systems reveals that more bits are needed in the case of hexadecimal digits in order to have the same or better relative errors. If $\beta = 2^k$ and the width is t_k the formulas for MRRE and ARRE become:

$$\begin{aligned}
MRRE(t_k, 2^k) &= 2^{-t_k-1}2^k \\
ARRE(t_k, 2^k) &= \frac{2^k - 1}{4k\ln 2} 2^{-t_k}
\end{aligned}$$

To have the same or better error for a base $\beta = 2^k$ in comparison to the binary-base (2^1), the gaps between two successive floating-point numbers in the larger base must be less than or equal to the gaps in the binary-base. So, if the exponent in base $\beta = 2^k$ is e_k then for base 2^1, $gap_1 = (2^1)^{e_1} 2^{-t_1}$. For base 2^k, $gap_k = (2^k)^{e_k} 2^{-t_k}$. It should be noted that $e_1 = e_k \times k + q$ where $|q| < k$. In fact with this definition, q is always a negative integer as illustrated by the following simplified example

		exp.	*mantissa*
$\beta = 16$		101	$0010xxxxxxx$
$\beta = 2$	*before normalization*	10100	$0010xxxxxxx$
$\beta = 2$	*after normalization*	10010	$10xxxxxxx$

So, The potential left shift for normalization of up to $k-1$ positions makes q negative and it falls in the range $-(k-1) \leq q \leq 0$. Specifically, in the case of $k = 4$, $q \in \{-3, -2, -1, 0\}$. With that in mind, to have the same or better representation for the case of $\beta = 2^k$ the following must hold:

$$\begin{aligned} gap_k &\leq gap_1 \\ (2^k)^{e_k} 2^{-t_k} &\leq (2)^{e_1} 2^{-t_1} \\ ke_k - t_k &\leq e_1 - t_1 \\ ke_k - (ke_k + q) &\leq t_k - t_1 \\ -q &\leq t_k - t_1 \end{aligned}$$

In order to have the last inequality true for all the values of q then

$$t_k - t_1 \geq k - 1$$

If t_k is chosen to be $t_1 + k - 1$ and then substituted in the equation for MRRE, the maximum relative representation error becomes

$$\begin{aligned} MRRE(t_k, 2^k) &= 2^{-t_k-1} 2^k \\ &= 2^{-(t_k-(k-1))} \\ &= 2^{-t_1} \end{aligned}$$

which is intuitive. Equal gaps in both systems means that the same set of numbers out of the real numbers range is being represented in both systems and hence the maximum representation error must be equal.

The average relative representation errors on the other hand are not equal because of the different distribution probability of the numbers. The ratio of $ARRE(t_k, 2^k)$ to $ARRE(t_1, 2^1)$ is

$$\begin{aligned} \frac{ARRE(t_k, 2^k)}{ARRE(t_1, 2^1)} &= \frac{2^k - 1}{k 2^{k-1}} \\ &= \frac{2}{k}(1 - \frac{1}{2^k}) \end{aligned}$$

So, for all $k > 1$, $ARRE(t_k, 2^k) < ARRE(t_1, 2^1)$.

Cody [Cody, 1973] tabulates (Table 5.1) the error as a function of the radix for three 32-bit floating point formats having essentially identical range. (The base 2 entries in the table are without use of the hidden one.)

Table 5.1: Tarde-off between radix and representational errors

				Maximum	Average
	Exponent		Mantissa	Relative	Relative
Base	Bits	Range	Bits	Error	Error
2	9	$2^{255} \simeq 6 \times 10^{76}$	22	0.5×2^{-21}	0.18×2^{-21}
4	8	$4^{127} \simeq 3 \times 10^{76}$	23	0.5×2^{-21}	0.14×2^{-21}
16	7	$16^{63} \simeq 0.7 \times 10^{76}$	24	2^{-21}	0.17×2^{-21}

According to Table 5.1, the binary system seems better in range and accuracy than the hexadecimal system. So why use hexadecimal radix at all? The answer is in the higher computational speed associated with larger base value, as illustrated by the following example.

Example 5.9 Assume a 24-bit mantissa with all bits zero except the least significant bit. Now, compare the maximum number of shifts required for each case of postnormalization.

Binary system: Radix = 2 and 23 shifts are required.

Hexadecimal system: Radix = 16 and we shift four bits at a time (since each hexadecimal digit is made of 4 bits) therefore, the maximum number of shifts is five.

A higher base provides better results for speed. A hexadecimal-base is better than the binary-base for the shifting needed in the alignment of the operands and in the normalization in case of addition as discussed in section 5.5. If the exponent base is binary a shifter capable of shifting to any bit position is needed. On the other hand, if the exponent base is hexadecimal, only shifts to digit boundaries (4 bits boundaries) are needed.

Obviously, if we want to represent the same range in a hexadecimal-based and binary-based system then the width of the exponent field in the hex format is two bits less than its counterpart in the binary format. Those two bits are then used to increase the precision in the hex format as indicated in Table 5.1.

$\overset{?}{\Longrightarrow}$ **Exercise 5.2 It is clear that two additional mantissa bits are not enough to compensate for the loss in MRRE. Use our earlier analysis to indicate how many bits are needed.**

Garner [Garner, 1976] summarizes the tradeoffs:

> "...the designer of a floating number system must make decisions affecting both computational speed and accuracy. Better accuracy is obtained with small base values and sophisticated round-off algorithms, while computational speed is associated with larger base values and crude round-off procedures such as truncation."

We will come to the issue of rounding the result once we explore the errors inherent in the calculations first.

5.3.2 Loss of Significance

The following example illustrates a loss of significance inherent in floating point numbers.

Example 5.10 We assume here a system using the S/370 short format and performing an addition.

$$\begin{array}{lcll}
A & = & 0.100000 & \times 16^{1} \\
B & = & 0.100000 & \times 16^{-10}
\end{array}\Bigg\}\ \text{Original Operands}$$

$$\begin{array}{lcll}
A & = & 0.100000 & \times 16^{1} \\
B & = & 0.000000000001 & \times 16^{1}
\end{array}\Bigg\}\ \text{Alignment}$$

$$\begin{array}{lcll}
A+B & = & 0.100000000001 & \times 16^{1} \quad \text{Addition} \\
A+B & = & 0.100000 & \times 16^{1} \quad \text{Postnormalization}
\end{array}$$

Thus, $A + B = A$, while $B \neq 0$.

This violation of a basic law of algebra is characteristic of the approximations used in the floating point system.

These approximations also lead to a violation of the property of associativity in addition.

Example 5.11 Assuming a decimal system with five digits after the point, check the associativity with $1.12345 \times 10^1 + 1.00000 \times 10^4 - 1.00000 \times 10^4$.
Solution: Given only five decimal digits, the result of

$$\begin{aligned}
(1.12345 \times 10^1 + 1.00000 \times 10^4) - 1.00000 \times 10^4 &= 1.00112 \times 10^4 - 1.00000 \times 10^4 \\
&= 1.12000 \times 10^1.
\end{aligned}$$

On the other hand, the result of

$$\begin{aligned}
1.12345 \times 10^1 + (1.00000 \times 10^4 - 1.00000 \times 10^4) &= 1.12345 \times 10^1 + 0 \\
&= 1.12345 \times 10^1.
\end{aligned}$$

Associativity fails and the first answer lost three digits of significance.

$\overset{?}{\Longrightarrow}$ **Exercise 5.3 Provide an example where the associativity of addition fails in a floating point system and the answer loses *all* the digits of significance.**

The following example [Hasitume, 1977] illustrates another loss of significance problem.

Example 5.12 Assume that two numbers are different by less than 2^{-24}. (The representation is the IBM System 370 short format.)

$$\begin{array}{rcllllllll} A & = & 0.1 & 0 & 0 & 0 & 0 & 0 & \times & 16^1 \\ B & = & 0.F & F & F & F & F & F & \times & 16^0. \end{array}$$

When one is subtracted from the other, the smaller must be shifted right to align the radix points. (Note that the least significant digit of B is now lost.)

$$\begin{array}{rclllllll} A & = & 0.1 & 0 & 0 & 0 & 0 & 0\ \times 16^1 & \\ B & = & 0.0 & F & F & F & F & F\times 16^1 & \\ \cline{3-8} A-B & = & 0.0 & 0 & 0 & 0 & 0 & 1\ \times 16^1 & = 0.1 \times 16^{-4}. \end{array}$$

Now let us calculate the error generated due to loss of digits in the smaller number. The result is (assuming infinite precision):

$$\begin{array}{rcllllllll} A & = & 0.1 & 0 & 0 & 0 & 0 & 0 & 0\ \times 16^1 & \\ B & = & 0.0 & F & F & F & F & F & F\times 16^1 & \\ \cline{3-9} A-B & = & 0.0 & 0 & 0 & 0 & 0 & 0 & 1\ \times 16^1 & = 0.1 \times 16^{-5}. \end{array}$$

Thus, the loss of significance (error) is $0.1\times 16^{-4} - 0.1\times 16^{-5} = 0.F\times 16^{-5} = 93.75\%$ of the correct result. Quite a large relative error!

An obvious solution to this problem is a guard digit, that is, additional bits are used to the right of the mantissa to hold intermediate results. In the IBM format, an additional 4 bits (one hexadecimal digit) are appended to the 24 bits of the mantissa. Thus, with a guard digit the above example will produce no error.

On first thought, one might think that in order to obtain maximum accuracy it is necessary to equate the number of guard bits to the number of bits in the mantissa. All these guard bits are shifted in when a massive cancellation of the most significant bits occur. Let us check this further.

$\overset{?}{\Longrightarrow}$ **Exercise 5.4 A cancellation occurs only in the case of a subtraction. In a non-redundant representation, what is the necessary condition on the exponent difference of the two operands so that there is a possibility of canceling more than one digit at the most significant side? Based on your answer, prove that only one guard digit is needed for the case of massive cancellation?**

The addition, multiplication, and division operations never exhibit this massive cancellation and hence this sort of loss of significance does not affect them. In fact, we analyze the possible ranges for the results of the various operations in section 5.5 for the case of normalized numbers. None of those ranges for any operation (with the exception of subtraction which we have just analyzed now) involves a massive cancellation that leads to a significance loss. Thus, regardless of the operation, no more than one guard digit will enter the final significant result.

To correctly round the result, we must keep some information about the remaining digits beyond the guard digit.

5.3.3 Rounding: Mapping the Reals into the Floating Point Numbers

Rounding in floating point arithmetic [Yohe, 1973] and the associated error analysis [Marasa and Matula, 1973] are probably among the most discussed subjects in floating point literature. Garner [Garner, 1976] provides an extensive list of early papers on the subject.

The following formal definition of rounding is taken from Yohe [Yohe, 1973]:

> *Let $\Re$ be the real number system and let M be the set of machine representable numbers. A mapping* **Round** $: \Re \to M$ *is said to be* **rounding** *if,* $\forall a, b \in \Re$ *where* $\mathbf{Round}(a) \in M$*, and* $\mathbf{Round}(b) \in M$ *we have:*
>
> $$\mathbf{Round}(a) \le \mathbf{Round}(b) \text{ whenever } a \le b.$$
>
> *Further: A rounding is called optimal if* $\forall a \in M$, $\mathbf{Round}(a) = a$.

"Optimal" implies that if $a \in \Re$ and m_1, m_2 are consecutive members of M with $m_1 < a < m_2$, then $\mathbf{Round}(a) = m_1$ or $\mathbf{Round}(a) = m_2$. Rounding is symmetric if $\mathbf{Round}(a) = -\mathbf{Round}(-a)$. For example, $\mathbf{Round}(39.2) = -\mathbf{Round}(-39.2) = 39$.

Several optimal rounding methods may be defined for all $a \in \Re$.

1. Downward directed rounding: $\triangledown a \le a$. This mode is also called round towards minus infinity (RM).
2. Upward directed rounding: $\triangle a \ge a$. This mode is also called round towards plus infinity (RP).
3. Truncation (T), which truncates the digits beyond the rounding point.
4. Rounding toward zero (RZ). In the case of traditional signed magnitute notation such as the one humans use when writing or the one that the IEEE standard (discussed in section 5.4) defines, rounding toward zero is equivalent to truncation.
5. Augmentation or rounding away from zero (RA).
6. Rounding to nearest up (RNU), which selects the closest machine number, and in the case of a tie selects the number whose magnitude is larger. The word "up" in the name is only really accurate if the number is positive. Hence, we will refer to this as round to nearest away (RNA) since it results in the number further away from zero.

The last rounding (RNA) is the most frequently used in human calculations since it produces maximum accuracy. However, since this round to nearest away (RNA) produces a consistent bias in the result, the IEEE standard uses a variation where the case of a tie is rounded to the even number: the number with a *LSB* equal to 0. This latter rounding is called round to nearest even (RNE). The accuracy of both RNA and RNE on a single computation is the same. It is only on a series of computations that the bias of RNA creeps in. RNE is an example of an unbiased rounding. Another example of unbiased rounding is the Round to Nearest Odd (RNO) which is similar but rounds to the number with *LSB* equal to one in case of ties. On the other hand, another example of biased rounding is the Round to Nearest with ties resolved towards zero (RNZ). This RNZ gives the nearest representable number as the result and in the case of a tie the result is the representable number closer to zero.

The two directed roundings $\bigtriangledown$ and $\bigtriangleup$, while not widely available outside of the IEEE standard, are very important in interval arithmetic which is a procedure for computing an upper and lower bound on the true value of a computation. These bounds define an interval which contains the true result.

Example 5.13 Some of the preceding rounding methods are illustrated here using a simple decimal format.

Number	$\bigtriangledown$	$\bigtriangleup$	RZ	RA	RNA	RNE
+38.7	+38	+39	+38	+39	+39	+39
+38.5	+38	+39	+38	+39	+39	+38
+38.2	+38	+39	+38	+39	+38	+38
+38.0	+38	+38	+38	+38	+38	+38
−38.0	−38	−38	−38	−38	−38	−38
−38.2	−39	−38	−38	−39	−38	−38
−38.5	−39	−38	−38	−39	−39	−38
−38.7	−39	−38	−38	−39	−39	−39

This example clarifies a few points:

- Obviously, the $\bigtriangledown$ and $\bigtriangleup$ methods are not symmetric roundings and they depend on the sign of the number. The other methods only depend on the magnitude.
- RNA and RNE only differ in the case of a tie when the larger number is odd. Hence, 39.5 rounds to 40 for both RNA and RNE.
- In the cases presented, the RZ method is the easiest from a computation point of view since it is actually a simple truncation.

Fig. 5.1 presents the rounding methods in a graphical manner. The process of rounding maps a range of real numbers to a specific floating point number. Depending on the format used, that floating point number may be normalized or unnormalized, it may use sign and magnitude notation for the significand or it may use other encodings.

$\overset{?}{\Longrightarrow}$ **Exercise 5.5 Based on Fig. 5.1, give $\bigtriangledown(a)$ and $\bigtriangleup(a)$ in terms of $RZ(a)$ and $RA(a)$ when $a > 0$. Do the same when $a < 0$.**

$\overset{?}{\Longrightarrow}$ **Exercise 5.6 If your hardware only has $\bigtriangledown(a)$, how can you get $\bigtriangleup(a)$ for any a?**

$\overset{?}{\Longrightarrow}$ **Exercise 5.7 For a specific number a, what is the relation between $\bigtriangledown(a)$ and $\bigtriangleup(a)$?**

$\overset{?}{\Longrightarrow}$ **Exercise 5.8 Similarly, for a specific number a, what is the relation between $RA(a)$ and $RZ(a)$?**

Fig. 5.1 clarifies the definitions of the various rounding methods regardless of the specific details of how a floating point number is represented. The specific bit manipulation leading to a certain rounding method in a given floating point format depends on the encoding used in that format.

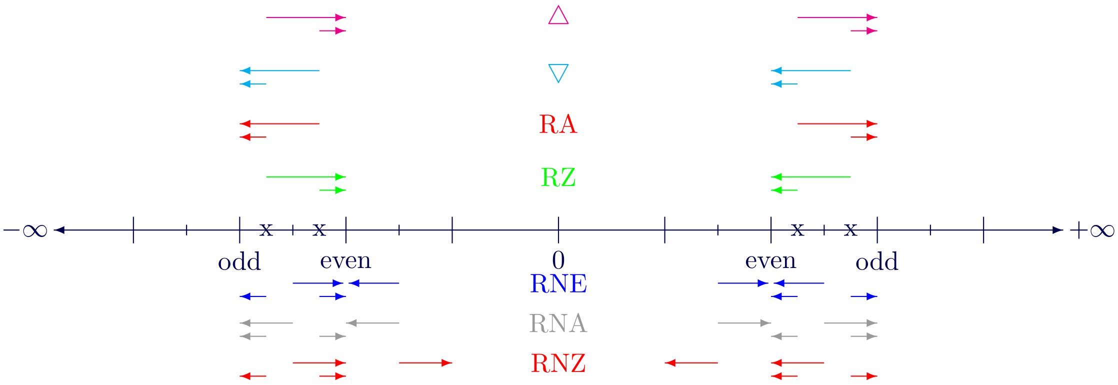


Figure 5.1: Rounding methods on the real number axis. Large tick marks indicate representable numbers, short tick marks indicate the real number exactly in the middle between two representable numbers, and the x marks represent other possible locations of real numbers. Note the difference between RNE, RNA, and RNZ in tie cases.

Example 5.14 To illustrate the issue of representation, indicate what is the effect of truncating the *LSB* in the binary number 1.101 to become 1.10 assuming it is represented as sign-magnitude (bit to the left of the point is sign), two's complement, ones complement, or excess-code (assume in this case the excess is 0.11 in binary).
Solution: The encodings lead to different values of the original and truncated bit patterns.

Code	Original value	Truncated value	Truncated $\leq$ Original
S-M	-0.625	-0.5	No
2's	-0.375	-0.5	Yes
1s	-0.25	-0.25	Yes
excess	$+0.825$	$+0.75$	Yes

Our understanding of Fig. 5.1 leads us to correctly indicate the rounding method corresponding to each of the encodings in the example. In the case of two's complement, any bit except the *MSB* has a positive weight and it increases the value of the number if it is one. Hence, the truncation of such a bit, which is equivalent to making it zero, either retains the current value of the number or reduces it. This is the definition of RM. Similarly, for ones complement and excess-code, the truncation is equivalent to RM. On the other hand, in sign-magnitude notation, the truncation merely decreases the magnitude or the absolute value of the number. Hence, if the number is negative its truncated value is actually larger as shown in the example. Truncation is equal to RZ in a sign-magnitude notation.

5.4 History of floating point standards

It must be clear to the reader by now that if we want to use a floating point system, we need to define several aspects. Those include the radix of the number system, the location of the fractional point and whether the numbers are normalized or not. Because of this need, several formats arose in the history of computer arithmetic. Some of those formats were *de facto* standards used by large companies such as the IBM formats and some were developed by standardization bodies such as the Institute of Electrical and Electronics Engineers (IEEE). The IEEE standard was developed in order to support portability between computers from different manufacturers as well as different programming languages. It emphasizes issues such as rounding the numbers correctly to get reliable answers. At the time of this writing, the IEEE standard is the most widely used in general purpose computation. Hence it will be explained in more detail and contrasted to other formats.

The IEEE produced a standard (IEEE 754) for binary floating point arithmetic in 1985 IEEE Std 754-1985 and a second complementary standard (IEEE 854) for radix independent floating point arithmetic in 1987 IEEE Std 854-1987. Both standards propose two formats for the numbers: a single and a double. The single format of the binary IEEE 754 standard has 32 bits while the double has 64 bits. A revision of the IEEE 754 standard started in 2001 to correct, expand, and update the old standard. Mainly, the revision tried to incorporate aspects of the IEEE 854, add some of the technical advances that occured since the publication of the original standard, and correct some of the past mistakes. The revised standard (IEEE 754–2008) appeared in 2008 IEEE Std 754-2008. During the revision process, it became clear that a simple merge between IEEE 754 and IEEE 854 is not satisfactory. The committee decided to use IEEE 754 as the base document and to amend it. Probably the largest addition in IEEE 754–2008 is the inclusion of decimal floating point, not just binary, and a large number of associated operations. The change of the binary formats names is another noticeable difference to the users of the 1985 and 2008 standards. To have a consistent naming convention, the committee decided to clearly indicate the implied radix of the format (either binary or decimal) and the number of bits in the format (32, 64, or 128). What used to be called the single and double format in IEEE 754–1985 is now called binary32 and binary64 in IEEE 754–2008. Though noticeable, this change of names bears no technical importance. In fact, the IEEE 754–2008 standard clearly states "The names used for formats in this standard are not necessarily those used in programming environments." There are several other changes that will be explained in the coming few sections. For a complete evaluation of the changes, the reader should consult the two documents. In the following discussion, unless explicitly stated otherwise, "IEEE standard" refers to the IEEE 754–2008.

The IEEE standard refers to multiple levels of specifications as a systematic way to approximate real numbers. The first level is the one corresponding to mathematics, i.e. the real numbers as well as the positive and negative infinities. The rounding operation that fits the infinite set of mathematical numbers into a finite set defines the second level named floating point data. The second level includes a special symbol to represent any entity that is not a number which might arise from invalid operations for example as we will see later. A single finite number from level two such as -12.56 may have multiple representations as in -1.256×10^1, -125.6×10^{-1}, or -12560000×10^{-6}. Those various equivalent representations define level three. Finally, the fourth level specifies the exact bit pattern encoding of the representation given in level three. We will now explore the specific bit representations defined.

5.4.1 IEEE binary formats

Fig. 5.2 presents the single and double binary formats of IEEE 754–1985 which were retained as is in IEEE 754–2008. The binary128 is a new format in IEEE 754–2008. The most significant

Sign	Biased exponent	Significand = 1.f (the '1' is hidden)
±	e + bias	f

32 bits:	8 bits, bias = 127	23 + 1 bits, single-precision or short format
64 bits:	11 bits, bias = 1023	52 + 1 bits, double-precision or long format
128 bits:	15 bits, bias = 16383	112 + 1 bits, quad-precision

Figure 5.2: IEEE single (binary32), double (binary64), and quad (binary128) floating point number formats.

Table 5.2: Maximum and minimum exponents in the binary IEEE formats.

Parameter	binary32	binary64	binary128
Exponent width in bits	8	11	15
Exponent bias	+127	+1023	16383
$exp_{\mathbf{max}}$	+127	+1023	16383
$exp_{\mathbf{min}}$	−126	−1022	−16382

bit is the sign bit (*sign*) which indicates a negative number if it is set to 1. The following field denotes the exponent (*exp*) with a constant bias added to it. The remaining part of the number is the significand normalized to have one non-zero bit to the left of the floating point. Since this is a non-redundant binary system, any bit is either 0 or 1. Hence, the normalized numbers must have a bit of value 1 to the left of the floating point. The value of the bit is always known and thus there is no need to store it and it is implied. This implicit bit is called the 'hidden 1.' Only the fractional part (f) of the significand is then stored in the standard format. The standard calls the significand without its *MSD* the trailing significand. To sum up, the normalized number given by the binary standard format has $(-1)^{sign} \times 2^{exp} \times 1.f$ as a value.

The biased exponent has two values reserved for special uses: the all ones and the all zeros. For the binary32 format those values are 255 and 0 giving a maximum allowed real exponent ($exp_{\mathbf{max}}$) of $254 - 127 = 127$ and a minimum exponent ($exp_{\mathbf{min}}$) of -126. Table 5.2 summarizes the maximum and minimum exponents for the binary formats. Note that the standard defines the bias and $exp_{\mathbf{max}}$ to be equal while $exp_{\mathbf{min}} = 1 - exp_{\mathbf{max}}$.

$\overset{?}{\Longrightarrow}$ **Exercise 5.9 If the exponent field has w bits, use Table 5.2 to deduce the relation giving the bias and the maximum and minimum biased exponents in terms of w.**

As for the special values, their interpretation is as shown in Table 5.3. If the exponent field is all ones and the trailing significand field is not zero then this represents what is called 'Not a Number' or NaN in the standard. This is a symbolic entity that might arise from invalid operations such as $+\infty - \infty$.

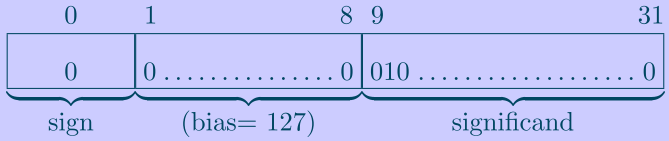
If the exponent field is zero and the trailing significand field is not zero then it represents a subnormal number (was called denormalized number in IEEE 754–1985) which is defined in the standard as: "In a particular format, a non-zero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format." Then the standard specifies the

Table 5.3: Encodings of the special values and their meanings.

Exponent bits	Fraction bits	Meaning
All ones	all zeros	$\pm\infty$ (depending on the sign bit)
All ones	non zero	NaN (Not a Number)
All zeros	all zeros	± 0 (depending on the sign bit)
All zeros	non zero	subnormal numbers

value of a binary subnormal number as $(-1)^{sign} \times 2^{exp_{\mathbf{min}}}(0.f)$.

Example 5.15 According to this definition the following bit string



is equal to $2^{-126} \times 0.01 = 2^{-128}$.

$\overset{?}{\Longrightarrow}$ **Exercise 5.10 It is important to note that the biased exponent for the subnormal numbers is zero, yet we use 2^{-126} and not 2^{-127} as the scaling factor (2^{-1022} not 2^{-1023} for the binary64 format and 2^{-16382} not 2^{-16383} for the binary128 format). Can you figure out why it is defined this way in the standard?**

The subnormal numbers provide a property called gradual underflow which we will detail in section 5.6.

In addition to the three basic binary formats (binary32, binary64, and binary128), the standard also recommends a way to extend those formats in a systematic manner. For these extended cases, if the total number of bits is k, the number of bits in the exponent field is w, and the trailing significand's number of bits is t then the following relations should hold true (where round( ) rounds to the nearest integer).

$$\begin{aligned}
w &= \text{round}\left(4 \times \log_2(k)\right) - 13 \\
t &= k - w - 1 \\
\text{bias} &= 2^{w-1} - 1 \\
exp_{\mathbf{max}} &= 2^{w-1} - 1 \\
exp_{\mathbf{min}} &= 1 - exp_{\mathbf{max}} = 2 - 2^{w-1}
\end{aligned}$$

These relations are defined for the cases where k is both ≥ 128 and a multiple of 32. They extrapolate the values defined in the standard for $k = 64$ and $k = 128$.

5.4.2 Prior formats

Table 5.4 shows the details of three formats that were at some point in time *de facto* competing standards. From the table we see that there is hardly any similarity between the various formats.

Table 5.4: Comparison of floating point specification for three popular computers.

	IBM S/370	DEC PDP-11	CDC Cyber 70

	S = Short
	L = Long
		S = Short
		L =Long	

Word length	S: 32 bits L: 64 bits	S: 32 bits L: 64 bits	60 bits
Exponent	7 bits	8 bits	11 bits
Significand	S: 6 digits L: 14 digits	S: (1)+23 bits L: (1)+55 bits	48 bits
Bias of exponent	64	128	1024
Radix	16	2	2
Hidden '1'	No	Yes	No
Radix point	Left of Fraction	Left of hidden '1'	Right of MSB of Fraction
Range of Fraction (F)	$(1/16) \leq F < 1$	$0.5 \leq F < 1$	$1 \leq F < 2$
F representation	Signed magnitude	Signed magnitude	One's complement
Approximate max. positive number*	$16^{63} \simeq 10^{76}$	$2^{126} \simeq 10^{38}$	$2^{1023} \simeq 10^{307}$
Precision	S: $16^{-6} \simeq 10^{-7}$ L: $16^{-14} \simeq 10^{-17}$	S: $2^{-24} \simeq 10^{-7}$ L: $2^{-56} \simeq 10^{-17}$	$2^{-48} \simeq 10^{-14}$

Approximate maximum positive number for the DEC PDP-11 is 2^{126}, as 127 is a reserved exponent.

This situation, which prohibits data portability produced by numerical software, was the main motivation in 1978 for setting up an IEEE (Computer Society) committee to standardize floating point arithmetic. The main goal of the standardization efforts was to establish a standard which will allow communication between systems at the data level without the need for conversion.

In addition to the respectable goal of "the same format for all computers," the committee wanted to ensure that it would be the best possible standard for a given number of bits. Specifically, the concern was to ensure correct results, that is, the same as those given by the corresponding infinite precision with a maximum error of 1/2 of the *LSB*. Furthermore, to ensure portability of all numerical data, the committee specified exceptional conditions and what to do in each case (overflow, underflow, etc.). Finally, it was desirable to make possible future extensions of the standard such as interval arithmetic which lead to more accurate and reliable computations.

The IEEE Computer Society received several proposals for standard representations for consideration; however, the most complete was the one prepared by Kahan, Coonen, Stone, and Palmer [Coonen, 1979]. This proposal became the basis for the IEEE standard (IEEE 754–1985) floating point representation.

By a simple comparison, it should be clear that the IEEE formats (at least the single format) is very similar to that of the PDP-11 and the VAX machines from the (former) Digital Equipment Corporation (DEC), but it is not identical. For example, the IEEE true significand is in the range $[1, 2($, whereas the DEC significand $\in [0.5, 1($.

For the IEEE, the biased exponent is an 8-bit number between 0 and 255 with the end values of 0 and 255 used for reserved operands. The IEEE bias is 127 so the true exponent is such that $-126 < exp_{\text{IEEE}} < 127$. The DEC format also reserves the end values of the exponent range but uses a bias of 128. Hence, the true exponent range is $-127 < exp_{\text{DEC}} < 126$. The DEC decoding of the reserved operands is also different. Table 5.5 illustrates this difference.

Table 5.5: IEEE and DEC decoding of the reserved operands (illustrated with the SINGLE format).

S	Biased Exp	Significand	Interpretation	
0	0	0	+Zero	IEEE
1	0	0	−Zero	
0/1	0	Not 0	±Denormalized Numbers	
0	255	0	+Infinity	
1	255	0	−Infinity	
X	255	Not 0	NaN (Not a Number)	
S	Biased Exp	Significand	Interpretation	DEC
0	0	Don't care	Unsigned zero	
1	0	Don't care	General purpose reserved operands	

$\overset{?}{\Longrightarrow}$ **Exercise 5.11 Find the value of max and min (largest and smallest representable numbers) for single and double precision in the following systems:**

1. **IEEE standard,**
2. **System/370, and**
3. **PDP-11.**

5.4.3 Comparing the different systems

In light of the previous discussion, let us analyze the good and bad points of each of the above three popular formats from Table 5.4 as well as the IEEE standard.

Representation Error: According to Ginsberg [Ginsberg, 1977], the combination of base 16, short mantissa size, and truncated arithmetic should definitely be avoided. This criticism is of the IBM short format where, due to the hexadecimal base, the MRRE is 2^{-21}. By contrast, the 23 bits combined with the hidden '1' (as on PDP-11) seems to be a more sensible tradeoff between range and precision in a 32 bit word, with MRRE of 2^{-24}.

Range: While the PDP-11 scores well on its precision on the single format, it loses on its double format. In a 64 bit word, the tradeoff between range and precision is unbalanced and more exponent range should be given at the expense of precision. The exponent range of the CYBER 70 seems to be more appropriate for the majority of scientific applications.

Rounding: None of the three formats uses an *unbiased* rounding to the nearest machine number in case of ties [Ginsberg, 1977].

Implementation: The advantage of a radix 16 format (as in IBM S/370) over a radix 2 format is the relative ease of implementation of addition and subtraction. Radix 16 simplifies the shifter hardware, since shifts can only be made on 4 bit boundaries, while radix 2 formats must accommodate shifts to any bit position.

It is clear from this simple comparison that the IEEE 754–1985 attempted to benefit from the earlier experience and pick the best points in each of the earlier formats as much as possible.

However, it is not completely fault free. Several critical points were voiced over the years. As time passed, that standard proved right on some issues and deficient on others.

5.4.4 Who needs decimal and why?

The radix independent IEEE 854 standard of 1987 did not receive a wide adoption in the industry because the hardware necessarily implements a specific radix. On the other hand, a binary radix is easily adapted to digital electronic devices hence the IEEE 754-1985 quickly became a succesful standard. The most important radix beside binary is decimal. Decimal is the natural system for humans hence it is the de facto standard for input to and output from computer programs. But, beside data inspection decimal is important for other reasons that we will explore here.

We have already seen that the fraction $1/10$ is easily described in decimal as $(0.1)_{\beta=10}$ while in binary it leads to a representation with an infinite number of bits as $(0.0001100110011001100\ldots)_{\beta=2}$ which the computer rounds into a finite representation.

$\overset{?}{\Longrightarrow}$ **Exercise 5.12 Write the bits that represent the fraction $1/10$ in the binary32 and binary64 formats of IEEE 754 assuming that round to nearest even is used when converting from decimal to binary.**

The representational error due to conversion may lead to incorrect computational results even if the subsequent arithmetic is correct. A human using a spreadsheet program would expect that if x is set to 0.10 and y to 0.30 then $3x - y = 0.0$ or that three dimes equal thirty cents. A computer using binary arithmetic has a different opinion: $3x - y = 5.6 \times 10^{-17}$. Furthermore, $2x - y + x = 2.8 \times 10^{-17}$. Leading to the wonderful surprise that

$$\left.\frac{3x-y}{2x-y+x}\right|_{(x=0.1,y=0.3)} = 2\ !$$

$\overset{?}{\Longrightarrow}$ **Exercise 5.13 Given that $0.3 = (0.0100110011001100\ldots)_{\beta=2}$, explain why a computer using the double precision (binary64) and round to nearest even leads to that “surprise”.**

The attentive reader will note from the solution of the previous exercise that the sequence of steps to get $3x - y$ when x is set to 0.1 and y to 0.3 have a rounding error in the calculation of $3x$. However, the calculation of $2x - y + x$ is exact without any rounding errors during the calculation. The strange result of $2x - y + x \neq 0$ is due to the rounding errors during conversion. With a decimal radix, such surprises that arise from the errors in conversion do not exist. A decimal radix will also give a correct result for $3x - y$.

A decimal radix is not a solution to everything in life though. A human calculates $(4/3-1)\times 3-1$ as equal to zero. A computer with a finite representation whether using binary or decimal radix yields an incorrect result. Again, the reader is invited to check that simple calculation using a short computer program or a spreadsheet. With binary64 the result is $\approx -2.220446 \times 10^{-16}$.

Ten is the natural number base or radix for humans resulting in a decimal number system while a binary system is natural to computers. In the early days of digital computers, to suite the data provided by the human users many machines included circuits to perform operations on decimal numbers Richards [1955]. Decimal numbers were used even for the memory addressing and partitioning. In his seminal paper in 1959, Buchholz Buchholz [1959] presented many persuasive arguments for using binary representations instead of decimal for such machine related issues as memory addressing.

Buchholz states that "a computer which is to find application in the processing of large files of information and in extensive man-machine communication, must be adept at handling data in human-readable form. This includes decimal numbers, alphabetic descriptions, and punctuation marks. Since the volume of data may be great, it is important that binary-decimal and other conversions not become a burden which greatly reduces the effective speed of the computer."

Buchholz concludes that "a combination of binary and decimal arithmetic in a single computer provides a high-performance tool for many diverse applications. It may be noted that the conclusion might not be the same for computers with a restricted range of functions or with performance goals limited in the interest of economy; the difference between binary and decimal operation might well be considered too small to justify incorporating both. The conclusion does appear valid for high-performance computers regardless of whether they are aimed primarily at scientific computing, business data processing, or real-time control."

Due to the limited capacities of the first integrated circuits in the 1960s and later years, most machines adopted the use of dedicated circuits for binary numbers and dropped decimal numbers. With the much higher capabilities of current processors and the large increase in financial and human oriented applications over the Internet, decimal is regaining its due place. The largest change in the 2008 revision of the IEEE standard for floating point arithmetic IEEE Std 754-2008 is the introduction of the decimal floating point formats and the associated operations. Whether in software or hardware, a standard to represent the decimal data and determine the manner of handling exceptional cases in operations is important.

We have just seen that simple decimal fractions such as 1/10 which might represent a tax amount or a sales discount yield an infinitely recurring number if converted to a binary representation. Hence, a binary number system with a finite number of bits cannot accurately represent such fractions. When an approximated representation is used in a series of computations, the final result may deviate from the correct result expected by a human and required by the law Cowlishaw [2003], European Commission [1997]. One study Cowlishaw [2002b] shows that in a large billing application such an error may be up to $5 million per year.

Banking, billing, and other financial applications use decimal extensively. Such applications may rely on a low-level decimal software library or use dedicated hardware circuits to perform the basic decimal arithmetic operations. Two software libraries were proposed to implement the decimal formats of the IEEE standard 754-2008: one using the densely packed decimal encoding Cowlishaw [2007] and the other using the binary encoded decimal format Cornea et al. [2007] which is widely known as the Binary Integer Decimal (BID) encoding. Those two encodings are defined in the standard and will be explained shortly. Hardware designs were also proposed for addition Wang et al. [2009], multiplication Erle et al. [2007], Raafat et al. [2008], division Nikmehr et al. [2006], Wang and Schulte [2007], square root Wang and Schulte [2005], as well as complete processors Schwarz et al. [2009].

A benchmarking study Wang et al. [2007] estimates that many financial applications spend over 75% of their execution time in Decimal Floating Point (DFP) functions. For this class of applications, the speedup resulting from the use of a fast hardware implementation versus a pure software implementation ranges from a factor of 5.3 to a factor of 31.2 depending on the specific application running. This speedup is for the complete application including the non-decimal parts of the code. Energy savings for the whole application due to the use of dedicated hardware instead of a software layer are of the same order of magnitude as the time savings.

5.4.5 IEEE decimal formats

In the IEEE standard, a finite representable number in base β (β is 2 or 10) has a sign s (0 for positive and 1 for negative), an exponent e, and a significand m to represent its value as $(-1)^s \times \beta^e \times m$. The significand m contains p digits and has an implicit fractional point between its most significant digit and the next lower significant digit. Another permissible view in the standard is to consider the significand as an integer with the implicit fraction point to the right of all the digits and to change the exponent accordingly. The significand considered as an integer c would thus have a corresponding exponent $q = e - (p - 1)$ and the value of the number is $(-1)^s \times \beta^q \times c$.

Decimal formats differ from binary ones in the fact that they are not necessarily normalized with one non-zero digit to the left of the fractional point. Several different representations of the same number are allowed including ones that have leading zeros. For example, $0.000000123456789 \times 10^0$, $0.000012345678900 \times 10^{-2}$, and $1.234567890000000 \times 10^{-7}$ are all representable. They are members of the same 'cohort': the set of all floating-point representations that represent a given floating-point number in a given floating-point format.

This choice for decimal formats follows the human practice. In physical measurements, we distinguish between the case when the mass of a body is reported as $0.050\ kg$ versus $0.05\ kg$ and say that the first measurement is accurate to the nearest gram while the second is only accurate to the nearest ten grams. If both measurements are stored in a normalized form within the computer as 5×10^{-2} they become undistiguishable. Furthermore, storing them in a format with 16 digits as normalized numbers ($5.000000000000000 \times 10^{-2}$) may give the incorrect impression that both measurements were done to a much higher accuracy ($0.050000000000000\ kg$) than what really occured. To maintain the distinction, we should store the first measurement as $0.000000000000050 \times 10^{12}$ and the second measurement as $0.000000000000005 \times 10^{13}$ with all those leading zeros. Both are members of the same cohort. A higher software application layer might make the distinction to a user.

The IEEE standard supports these distinctions by allowing for unnormalized representations. Furthermore, the view of the significand as an integer c with the corresponding exponent q is the 'natural' view used for decimal parts within the standard.

$\overset{?}{\Longrightarrow}$ **Exercise 5.14 In a decimal format with p digits (decimal64 has 16 digits for example as will be defined shortly), if a finite non-zero number has n decimal digits from its most significant non-zero digit to its least significant non-zero digit with $n < p$, how many representations are in the cohort of this number?**

Fig. 5.3 presents the decimal64 and decimal128 formats of the IEEE standard. For decimal, the field encoding the exponent information is explicitly named the combination field since it encodes the exponent, some special cases, and the most significant four bits of the significand. When the most significant five bits of the combination field are 11111 the format encodes Not-a-Number (NaN) which is generated when an invalid operation occurs for example. On the other hand, if those bits are 11110 the encoded value is $\pm\infty$ depending on the sign bit. If neither

	Sign	Combination	Trailing Significand
	±	exponent and MSD	$t = 10J$ bits
64 bits:	1 bit	13 bits, bias = 398	50 bits, $15 + 1$ digits
128 bits:	1 bit	17 bits, bias = 6176	110 bits, $33 + 1$ digits

Figure 5.3: IEEE decimal64 and decimal128 floating point formats.

of those special cases exists, the combination field encodes the exponent q and four significand bits. The possible negative and positive values of the exponent are excess-coded by adding a bias to the exponent value and representing $E = q + \text{bias}$.

When the total number of bits is k (a multiple of 32), the number of bits in the combination field is $w + 5$, the trailing significand's number of bits is t, and the number of representable significand digits is p, the standard defines:

$$\begin{aligned}
w &= \tfrac{k}{16} + 4 \\
t &= k - w - 6 && = \tfrac{15k}{16} - 10 \\
p &= \tfrac{3t}{10} + 1 && = \tfrac{9k}{32} - 2 \\
exp_{\mathbf{max}} &= 3 \times 2^{w-1} \\
exp_{\mathbf{min}} &= 1 - exp_{\mathbf{max}} \\
\text{bias} &= exp_{\mathbf{max}} + p - 2 \\
exp_{\mathbf{min}} &\leq q + (p - 1) && \leq exp_{\mathbf{max}}
\end{aligned}$$

The standard specifies two basic decimal formats with $k = 64$ and $k = 128$ as shown in Fig. 5.3. The previous relations provide also for shorter (such as decimal32) and longer values as possible interchange formats to support the exchange of data between implementations.

Those relations also ensure that t the number of bits in the trailing significand is always an exact multiple of 10. This is essential since there are two different encodings defined for this field. The *binary* encoding concatenates the four most significant bits of the significand generated from the combination field to the trailing signifcand field and considers the whole as an integer in unsigned binary notation. The *decimal* encoding considers each 10 bits of the trailing significand to be a 'declet' encoding three decimal digits in the densely packed decimal format Cowlishaw [2002a]. The decoding of the three digits or their encoding back into a declet requires only a few simple boolean logic gates.

The binary encoding of the significand is potentially faster in the case of a software implementation of the standard since the regular integer instructions may be used to manipulate the significand field. The main advantage for the decimal encoding in hardware implementations is that it keeps the decimal digit boundaries accessible. Such accessibility simplifies the shifting of operands by a specific number of digits as well as the rounding of the result at the exact decimal digit boundary required by the standard. The decimal encoding also makes the conversion from or to character strings trivial.

5.5 Floating Point Operations

The standard specifies "Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format". The basic mandated arithmetic operations are: addition, subtraction, multiplication, division, square root, and fused multiply add (multiply then add both implemented as if with unbounded range and precision with only a single rounding after the addition). The standard also defines many other operations for the conversions between formats and for the comparisons of floating point data.

Since the standard allows multiple representations for the same value in decimal formats, it defines a 'preferred exponent' for the result of the operations to select the appropriate member of the result's cohort. If the result of an operation is inexact the cohort member of least possible exponent (i.e. the one where the *MSD* of the n digits of exercise 5.14 coincides with the *MSD* of the p digits) is used to get the maximum number of significant digits. If the result is exact, the cohort member is selected based on the preferred exponent for the operation.

In this section, we introduce some simple ideas to compute the basic arithmetic operations in just enough detail to analyze the resulting consequences.

To simplify this first exposition, we assume that the inputs follow the IEEE definition of significands where $\beta^{-(p-1)} \leq m < \beta$ and that normalized operands have the definition $1 \leq m < \beta$.

5.5.1 Addition and Subtraction

Addition and subtraction require that the exponents of the two operands be equal. This alignment is accomplished by shifting the mantissa of the smaller operand to the right, while proportionally increasing its exponent until it is equal to the exponent of the larger number.

Example 5.16 Assuming a system with four significand digits, calculate $1.324 \times 10^5 + 1.516 \times 10^3$.
Solution: As mentioned above, the first step is to align the operands. Then, we do the calculation.

$$\begin{array}{rll} & 1.324 & \times 10^5 \\ + & 1.516 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 1.324 & \times 10^5 \\ + & 0.01516 & \times 10^5 \\ \hline & 1.33916 & \times 10^5 \\ \approx & 1.339 & \times 10^5 \end{array} \right.$$

$\overset{?}{\Longrightarrow}$ **Exercise 5.15 In general scientific notation, the alignment could be accomplished by the converse proposition, that is, shift the mantissa of the larger number left, while decreasing its exponent. In the case of normalized inputs, why does the floating point hardware shift the smaller number to the right and not the larger to the left?**

If the inputs are not normalized and the larger number has leading zeros then the alignment process becomes more difficult.

Example 5.17 Assuming a system with eight significand digits, calculate $0.0001324 \times 10^5 + 0.0001516 \times 10^3$ as well as $0.1324567 \times 10^5 + 0.1516123 \times 10^3$.
Solution: In the first case, if we shift the smaller number to the right we will drop some digits and the result is inexact. However, it is possible to decrease the exponent of the larger number by shifting it to the *left* as long as it does not overflow.

$$\begin{array}{rll} & 0.0001324 & \times 10^5 \\ + & 0.0001516 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 0.0132400 & \times 10^3 \\ + & 0.0001516 & \times 10^3 \\ \hline & 0.0133916 & \times 10^3 \end{array} \right.$$

This yields an exact result within the number of digits available.
In the second case, however, it is not possible to shift the larger number to the left by the amount of the exponent difference since it will overflow. To minimize the loss due to rounding, we can shift the larger number as much as possible to the left then shift the smaller number to the right by the remaining amount to achieve the required alignment.

$$\begin{array}{rll} & 0.1324567 & \times 10^5 \\ + & 0.1516123 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 1.3245670 & \times 10^4 \\ + & 0.01516123 & \times 10^4 \\ \hline & 1.33972823 & \times 10^4 \\ \approx & 1.3397282 & \times 10^4 \end{array} \right.$$

After the alignment, the two mantissas are added (or subtracted), and the resultant number, with the common exponent, is normalized if needed. The latter operation is called postnormalization or sometimes just normalization. Such a normalization is necessary if the implemented system requires normalized results (as in the IEEE binary formats) and the result has leading zeros.

Example 5.18 Calculate $1.324 \times 10^3 - 1.321 \times 10^3$.
Solution: For a subtraction we use the radix complement.

$$\begin{array}{rll} & 1.324 & \times 10^3 \\ - & 1.321 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 1.324 & \times 10^3 \\ + & 8.679 & \times 10^3 \\ \hline & 0.003 & \times 10^3 \\ = & 3.000 & \times 10^0 \end{array} \right.$$

As seen from the example, in subtraction, the maximum shift (for a nonzero result) required on postnormalization is equal to the number of mantissa digits minus one. The hardware must thus detect the position of the leading non-zero digit to shift the result (to the *left* this time) and adjust the exponent accordingly. The subtraction may produce the special case of a zero result, whereby, instead of shifting, we should generate a 'canonical zero': the bit pattern corresponding to absolute zero as specified in the standard we are implementing.

$\overset{?}{\Longrightarrow}$ **Exercise 5.16 In the addition operation within a normalized system, the postnormalization is a maximum of one *right*-shifted digit. Why? Is it still true if the system does not require normalized numbers as is the case of IEEE decimal formats?**

5.5.2 Multiplication

Multiplication in floating point is conceptually easier than addition. No alignment is necessary. We multiply the significands m_1 and m_2 as if they were fixed point integers and simply add the exponents. Since floating point formats usually use a biased exponent, we must decrement the sum of the two biased exponents by the value of the bias in order to get a correct representation for the exponent of the result.

Example 5.19 If two operands in the IEEE binary32 format have the biased exponents 128 and 130, what is the exponent of the result?
Solution: The IEEE binary32 format has an exponent bias of 127. So, those biased exponents represent true exponents of 1 and 3 respectively. Obviously, the exponent of the result should be 4.
If we add the two biased exponents we get $(1+127)+(3+127) = (4+2\times127)$. We must decrement this sum by the bias value to get a correct characteristic representation of $(4 + 127) = 131$.

$\overset{?}{\Longrightarrow}$ **Exercise 5.17 In the addition and subtraction operations, we did not need to add or decrement the bias. Why?**

$\overset{?}{\Longrightarrow}$ **Exercise 5.18 Given normalized inputs only, can the exponent of the result in a multiplication overflow or underflow?**

The sign bit of the result is equal to the *XOR* of the two operand signs while the resultant significand depends on the two operands. For non-zero numbers, $\beta^{-(p-1)} \leq m_1 < \beta$ and $\beta^{-(p-1)} \leq m_2 < \beta$ so the initial resultant significand is in one of the following ranges:

$\beta^{-(2p-2)} \leq m_1 \times m_2 < \beta^{-(p-1)}$: We should shift the resultant significand to the *left* in order to make it equal to or larger than $\beta^{-(p-1)}$ and decrease the resultant exponent accordingly.

$\beta^{-(p-1)} \leq m_1 \times m_2 < 1$: If the digits in positions below $\beta^{-(p-1)}$ are non-zero and will be rounded then the result is inexact. To improve the accuracy, we may shift the significand to the *left* and decrease the resultant exponent accordingly.

$\overset{?}{\Longrightarrow}$ **Exercise 5.19 How many digits should be shifted and why?**

$1 \leq m_1 \times m_2 < \beta$: No normalization shift is required.

$\beta \leq m_1 \times m_2 < \beta^2$: We must shift the result by one position to the *right* and increase the resultant exponent by one.

Overflow: If any of the above cases (after incrementing the exponent if any) generates an exponent spill, then the postnormalization generates either **max** or a representation of ∞ depending on the rounding (explained below).

Underflow: If any of the above cases (after decrementing the exponent if any) generates an underflow, then the postnormalization generates either **min** or zero depending on the rounding (explained below).

Either operand is zero: The operation should produce a canonical zero.

5.5.3 Division

To perform floating point division, the significands are divided (m_1/m_2) and the exponent of the divisor is subtracted from the exponent of the dividend. For non-zero numbers, $\beta^{-(p-1)} \leq m_1 < \beta$ and $\beta^{-(p-1)} \leq m_2 < \beta$ according to our assumptions, so the initial result is contained by $\beta^{-p} < \frac{m_1}{m_2} < \beta^p$ when $m_2 \neq 0$.

$\overset{?}{\Longrightarrow}$ **Exercise 5.20 Given normalized inputs only, what is the range of the resultant exponent in the case of division? Can it overflow or underflow?**

The sign bit of the result is equal to the *XOR* of the two operand signs while the resultant significand belongs to one of the following cases:

$m_1 = 0, m_2 \neq 0$: The postnormalization produces a canonical zero.

$m_1 \neq 0, m_2 = 0$: The result overflows and the postnormalization produces either **max** or ∞ depending on the format and standard used.

$m_1 = m_2 = 0$: The result is mathematically undefined but usually the implemented standard specifies what to produce. In the case of IEEE, this produces a NaN (Not a Number).

$\beta^{-p} < m_1/m_2 < \beta^{-(p-1)}$: We should shift the resultant significand to the *left* in order to make it equal to or larger than $\beta^{-(p-1)}$ and decrease the resultant exponent accordingly.

$\beta^{-(p-1)} \leq m_1/m_2 < 1$: If the digits in positions below $\beta^{-(p-1)}$ are non-zero and will be rounded then the result is inexact. To improve the accuracy, we may shift the significand to the *left* and decrease the resultant exponent accordingly.

$1 \leq m_1/m_2 < \beta$: No postnormalization is required.

$\beta \leq m_1/m_2 < \beta^p$: We must shift the significand to the *right* and increase the exponent accordingly.

Overflow: If the exponent (after incrementing if any) indicates an overflow, we produce **max**, ∞, or deal with the situation according to the specification of the implemented standard.

Underflow: If the exponent (after decrementing if any) indicates an underflow, we produce **min**, 0, or deal with the situation according to the specification of the implemented standard.

$\overset{?}{\Longrightarrow}$ **Exercise 5.21 When subtracting the two exponents, is there any adjustment needed for the bias?**

5.5.4 Fused Multiply Add

The fused multiply add is an operation that takes three operands. Its most generic form produces the result $\pm A \times B \pm C$ for the operands, A, B, and C with a single rounding operation after the addition. Hence, it gives a more accurate result than what we get from a multiplication then rounding followed by addition then another rounding.

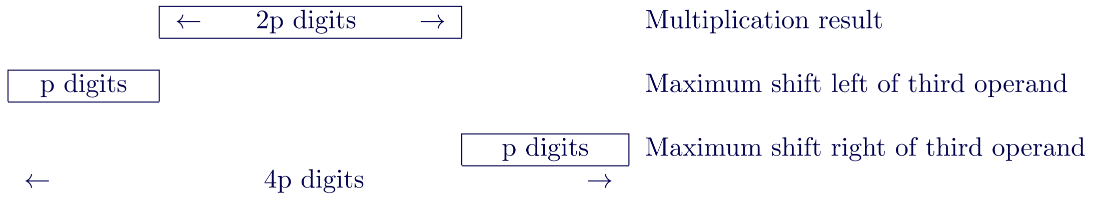

← 2p digits →
Multiplication result
p digits
Maximum shift left of third operand
p digits
Maximum shift right of third operand
← 4p digits →

Figure 5.4: Alignment shift of the third operand with respect to the multiplication result within the FMA.

The FMA was first introduced in the early 1990s after the publication of the IEEE standard 754-1985. Its use was attractive since it produces a more accurate result and it was also faster than two successive operations (the reasons why it is faster will become clear when its detailed implementation is explained). It is the basic operation performed in many program loops to get the sum of a bill of purchases for example or to get the result of a filtering operation in digital signal processing. Furthermore, in the calculation of scalar products, matrix multiplications, or polynomial evaluation we often iterate on an instruction such as $(sum = sum + a_i b_i)$. We can also get the "lower" part of a multiplication using the FMA: $H = ab + 0.0$ gives a rounded result that contains the most significant part of the product. If it is important to know the rounded part (to estimate the error in a calculation for example), we get it easily by $L = ab - H$.

Making this instruction a single operation that is both *faster* and more *accurate* is beneficial. However, since the FMA was not standard compliant, a slower and more inferior result was sometimes the "correct result" to produce. During the revisions of the IEEE standard, it became clear that such a useful operation should be part of the basic operations and it was included in IEEE std754-2008.

Since it involves a multiplication and an addition, it combines the steps of both operations. The absolute values of the first two operands $m_1 \times \beta^{e_1}$ and $m_2 \times \beta^{e_2}$ are multiplied. Their resulting exponent $(e_1 + e_2)$ is compared to the exponent of the third operand $(m_3 \times \beta^{e_3})$ to determine the amount of shifting needed to align the addition operation. Since each operand has p digits, the multiplication results in $2p$ digits as a maximum. A wide datapath may exist in the hardware to allow the alignment of the third operand with the $2p$ digit result of the multiplication by shifting that operand either left or right. The maximum shift occurs when none of the third operand's digits overlap with the multiplication result digits. Fig. 5.4 shows such a scheme.

The result of FMA may lead to an underflow or an overflow. So, both conditions must be checked. The simple scheme of a $4p$ digits wide datapath is not exactly how all real FMAs are implemented. We will see more about those in later chapters.

5.6 Reading the fine print in the standard

In this section, we try to present the fine details related to rounding and exceptional cases handling in the IEEE standard for floating point numbers. We follow that by a short analysis of the standard's strong and weak points.

5.6.1 Rounding

The standard defines five rounding directions:

1. RNE = Unbiased rounding to nearest (in case of a tie round to even).
2. RZ = Round toward zero (sometimes called chop or truncate due to the way it is implemented in sign and magnitude representation).
3. RM = Round toward minus infinity.
4. RP = Round toward plus infinity.
5. RNA = Biased rounding to nearest (in case of a tie round away from zero).

Any compliant implementation must provide RNE, RZ, RP, and RM. An implementation that supports the decimal formats must also provide RNA. The default rounding for the binary formats is RNE. For decimal formats, the IEEE standard leaves the definition of the default rounding to the programing language standards but recommends RNE.

The conventional (to humans at least!) round to nearest away from zero is easy to implement in sign-magnitude encodings by adding 1/2 of the digit to be discarded and then truncating to the desired precision.

Example 5.20 For round to integer, we have:

$$\begin{array}{c} 39.2 \\ 0.5 \\ \hline 39.7 \rightarrow 39 \end{array} \qquad\qquad \begin{array}{c} 39.7 \\ 0.5 \\ \hline 40.2 \rightarrow 40 \end{array}$$

But suppose the number to be rounded is exactly halfway between two numbers: which one is the nearest? To answer the question, let us add the same 0.5 to the two following numbers:

$$\begin{array}{c} 38.5 \\ 0.5 \\ \hline 39.0 \rightarrow 39 \end{array} \qquad\qquad \begin{array}{c} 39.5 \\ 0.5 \\ \hline 40.0 \rightarrow 40 \end{array}$$

Notice that we rounded up in both cases, even though each number was exactly halfway between the smaller and larger numbers.

Therefore, by simply adding 0.5 and truncating, the biased RNA rounding is generated. In order to have an unbiased rounding, we round to even whenever there is a tie between two numbers. Now, using the previous numbers we get:

$$\begin{array}{ccc} 38.5 & \rightarrow & 38 \\ 39.5 & \rightarrow & 40 \end{array}$$

In the first case the number is rounded down, and in the second case the number is rounded up. Therefore, we have statistically unbiased rounding. Of course, it is possible to obtain another unbiased rounding by rounding to odd (instead of even) in the tie case. For such a case, the rounding is:

$$\begin{aligned} 38.5 &\rightarrow 39 \\ 39.5 &\rightarrow 39 \end{aligned}$$

However, rounding to even is preferred because it may result in “nice” integer numbers, as in the following example.

Example 5.21 Round 1.95 and 2.05 to the first fractional position using both the round to nearest even and a round to nearest odd modes.
Solution: In the case of round to nearest even we get “nice” numbers:

$$\begin{aligned} 1.95 &\rightarrow 2.0 \\ 2.05 &\rightarrow 2.0 \end{aligned}$$

whereas rounding to odd results in the more frequent occurrence of noninteger numbers:

$$\begin{aligned} 1.95 &\rightarrow 1.9 \\ 2.05 &\rightarrow 2.1 \end{aligned}$$

To implement the RNE we must determine if the discarded part is exactly a tie or not. If it is not a tie case we must determine to which number it is closer, i.e., if the discarded part is larger than half the *LSB* of the remaining part or not. But, how does the hardware know if it is a tie case from the first place? Let us once more analyze what we as humans do.

The conventional system for rounding adds 1/2 of the *LSD* position of the desired precision to the *MSD* of the portion to be discarded. For RNE, this scheme has a problem: (the $XXXX$ are any additional digits)

$$\begin{array}{ll}
38.5\,X\ X\ X\ X & \leftarrow \text{Number to be rounded} \\
\underline{\ \ 0.5}\ 0\ \ 0\ \ 0\ \ 0 & \leftarrow \text{Add 0.5} \\
39.0\,X\ X\ X\ X & \leftarrow \text{Result} \\
39 & \leftarrow \text{Truncate}
\end{array}$$

Two cases have to be distinguished:

Case 1: $XXXX \neq 0$ (at least one $X = 1$). The rounding is correct since 39 is nearest to $38.5 + \delta$, where $0 < \delta < 0.5$.

Case 2: $XXXX = 0$ (all X bits are 0). Now the rounding is incorrect. It is a tie case that requires the result to be rounded to even (38).

It is obvious that, regardless of the number of X bits, all possible permutations are mapped into one of the two preceding cases. Therefore, one bit is enough to distinguish between the two possibilities. If any of the X bits is one this distinguishing bit becomes one. Otherwise it is zero. This bit is often called the “sticky bit” since any one in the X part “sticks” into this bit. The logic implementation of the sticky bit is simply the *OR* function of all the bits we want to check.

The *MSB* of the discarded part is called the "round bit" since we use it to determine the rounding. If the round bit is zero we are sure that the discarded part is less than half the *LSB* of the remaining part. If the round bit is one, we must check the sticky bit as well. If in this latter case the sticky bit is zero it is a tie otherwise the discarded part is more than half the *LSB* of the remaining part.

It is important now to remember our earlier discussion regarding the normalization shift in addition and subtraction. We discovered then that a right shift by one digit is possible in addition while a left shift by one digit is possible in subtraction. The left shift is of concern since it means that we must keep a "guard digit". In the case of binary this is just a guard bit. In fact, the format is:

1.		L	G	R	S
←	desired precision	→			

where

$L =$ *LSB* at the desired precision,

$G =$ guard bit,

$R =$ round bit, and

$S =$ sticky bit.

In the case of a left shift (normalization after subtraction), S does not participate in the left shift, but instead zeros are shifted into R. In the case of a right shift due to a significand overflow (during magnitude addition or no shift), the S and R guard bits are ORed into S (i.e., $L \rightarrow G$ and $G + R + S \rightarrow S$).

To summarize, while performing the operation, we keep two guard bits (G and R) and group any other bits shifted out into the sticky bit S. After the normalization but just before the rounding, the result has only one guard bit and the sticky bit. At this stage, if we want to use a RNA we just add half '1' to G and truncate as we did earlier. For a RNE, a more elaborate action is required:

1.		L	G	S
←	desired precision	→	a	

where

$L =$ *LSB* at the desired precision,

$G =$ guard bit,

$S =$ sticky bit, and

$a =$ bit to be added to G for proper rounding.

The proper action to obtain unbiased rounding-to-even (RNE) is determined from the following table:

L	G	S	Action	a
X	0	0	Exact result. No rounding is necessary.	0
X	0	1	Inexact result, but significand is rounded properly.	0
0	1	0	The tie case with even significand. No rounding needed.	0
1	1	0	The tie case with odd significand. Round to nearest even.	1
X	1	1	Round to nearest by adding 1.	1

Example 5.22 Here are some numbers with 4-bit significands rounded to nearest even.

		G	S	→ *Action*
a)	1.000X	0	0	→ machine number
b)	1.000X	0	1	→ closer to .000X
c)	1.0000	1	0	→ tie with *LSB* even
d)	1.0001	1	0	→ tie with *LSB* odd; becomes 1.0010
e)	1.000X	1	1	→ round up

$\overset{?}{\Longrightarrow}$ **Exercise 5.22 In some implementations, the designers choose to add the action bit to L instead of G. Check if it makes a difference for the case of RNE discussed so far.**

So far, we have addressed only the unbiased rounding; but there are three more optional modes. The round to zero, RZ, is simply a truncation in the conventional binary system that is used in certain integer related operations Actually, most computers provide truncation as it does not cost them much. The remaining two rounding modes are rounding toward $+\infty$ and rounding toward $-\infty$. These two directed roundings are used in interval arithmetic where one computes the upper and lower bounds of an interval by executing the same sequence of instructions twice, once to find the maximum value of the result and the second to find its minimum value.

$\overset{?}{\Longrightarrow}$ **Exercise 5.23 Let us consider the computation $s = (a \times b) - (c \times d)$ where a, b, c, s are floating point numbers that must be rounded. Find the guaranteed significance interval $[s_{min}, s_{max}]$ in terms of a, b, c, and d, and the rounding operations $\bigtriangledown$, $\bigtriangleup$, RZ, RA, RNA, and RNE.**

The sticky bit, introduced previously, is also essential for the correct directed rounding.

Example 5.23 Let us see the importance of the sticky bit to Directed Upward Rounding when we round to the integer in the following two cases.

Case 1: No sticky bit is used;

38.00001 → 38
38.00000 → 38

Case 2: Sticky bit is used:

38.00001 → 39 (sticky bit = 1)
38.00000 → 38 (sticky bit = 0, exact number).

When the sticky bit is one and we neglect using it, the result is incorrect.

$\overset{?}{\Longrightarrow}$ **Exercise 5.24 A fractional value f_i at bit location i of a signed digit binary number $\cdots x_{i+1}x_ix_{i-1}\cdots x_0$ where each $x_i \in \{-1, 0, 1\}$ can be defined as $f_i = (\Sigma_{j=0}^{j=i-1} 2^j \times x_j)/2^i$.**
The decision of the digit added for rounding is then determined by the fractional value at the rounding position. However, the value to add in order to achieve the correct rounding does not depend only on the fractional range but also on the IEEE rounding mode. In RP and RM modes, the sign of the floating point number affects the decision as well.
Assume that L is the bit at the rounding location, i.e. it is the least significant bit of the number and it is the location where the rounding digit will be added. The fractional value f is calculated at L. Please fill in the following table with the value of the digit to add to achieve correct rounding.

range of f	RNE	RZ	RP		RM	
			$+ve$	$-ve$	$+ve$	$-ve$
$-1 < f < -0.5$						
-0.5						
$-0.5 < f < 0$						
0						
$0 < f < 0.5$						
0.5						
$0.5 < f < 1$						

5.6.2 Exceptions and What to Do in Each Case

The IEEE standard specifies five exceptional conditions that may arise during an arithmetic operation:

1. invalid operation,
2. division by zero,
3. overflow,
4. underflow, and
5. inexact result.

The only exceptions that possibly coincide are inexact with overflow or inexact with underflow. When any of the exceptions occurs, the default behavior is to raise a status flag that remains raised as long as the user did not explicitly lower it. Complying systems have the option to also provide a trapping feature. Hence, exceptions are handled in one of two ways:

1. *TRAP* and possibly supply the necessary information to correct the fault. For example:

   What instruction caused the TRAP?
   What were the values of the input operands?
   Etc.

The standard also specifies the result that must be delivered to the trap handler in the case of overflow, underflow, and inexact exceptions.

2. *DISABLED TRAP* and deliver a specified result. For example on divide by zero: “Set the result to a correctly signed ∞”.

Invalid operations and NaNs

The invalid operation exception occurs during a variety of arithmetic operations that do not produce valid numerical results. However before exaplaining what are the invalid operations, it is important to clarify that the standard has two types of NaNs:

Signaling NaNs in some implementations represent values for uninitialized variables or missing data samples. They are a way to force a trap when needed since any operation on them signals the invalid exception (hence their name of signaling).

Quiet NaNs are supposed to provide retrospective information on the reason of the invalid operation which generated them. One way of achieving this goal is to use the significand part as a pointer into an array where the original operands and the instruction are saved. Another way is to make the significand equal to the address of the offending line in the program code. Most implementations complying to the standard did not do much if anything at all with this feature.

For binary formats, it is up to the implementation to decide on how to distinguish between the two types.

Example 5.24 Most implementations chose to make the distinction based on the *MSB* of the significand field. The Alpha AXP, SPARC, PowerPC, Intel i860, and MC68881 architectures chose to define a quiet NaN by an initial significand field bit of 1 and a signaling NaN by an initial significand field bit of 0.
The HP PA-RISC and the MIPS RISC architectures chose the opposite definition. They have 1 for signaling and 0 for quiet NaNs.

$\overset{?}{\Longrightarrow}$ **Exercise 5.25 In a certain implementation, the system boots with the memory set to all ones. Which of the two previous definitions for signaling and quiet NaNs is more approriate?**

The lesson learned from the differing implementations of sNaN and qNaN for binary formats lead the committee revising the IEEE standard to decide exactly how NaNs are encoded for decimal formats. If the five most significant bits of the combination field are ones then the value of the decimal format is a NaN. If the sixth most significant bit is also 1 then it is sNaN, otherwise it is qNaN.

The invalid operations that lead to an exception are:

1. any operation on a signaling NaN,

2. an effective subtraction of two infinities,

3. a multiplication of a zero by an infinity,

4. a division of a zero by a zero or an infinity by an infinity,

5. a remainder operation (x REM y) when y is zero or x is infinite,

6. a square root if the operand is less than zero, (Note that $\sqrt{-0}$ produces its operand as a result and does *not* cause an exception. It is the only case where the result of a square root is negative.)

7. a conversion of binary floating point number to an integer or decimal format when an overflow, infinity, or NaN is not faithfully represented in the destination format and there is no other way to signal this event, and

8. a comparison involving the less than or greater than operators on *unordered* operands. It should be noted here that a NaN is not considered in order with any number. The standard defines a special operator denoted by a question mark '?' to facilitate the comparisons with NaNs.

Example 5.25 Here are some examples on the above invalid operations.

$$\begin{gathered}(-\infty)+(+\infty)\\(+0)\times(-\infty)\\\frac{(-0)}{(+0)}\\\frac{+\infty}{+\infty}\\+\infty\mathbf{mod_4}\\\sqrt{-5}\end{gathered}$$

It is interesting to note that in the single precision format, there are $2^{23} \simeq 8$ million possible representations in the NaN class. Many more are available in the double precision format. However, such a huge number of representations is not put to much use in most implementations.

The operations in the standard never generate a signaling NaN. An invalid operation generates a quiet NaN if the result of the operation is a floating point format.

Since the quiet NaNs are valid inputs to most operations, it is important to specify exactly what to do in such a case. The standard specifies that when one or more NaNs (none of them signaling) are the inputs to an operation and the result of such an operation is a floating point representation then a quiet NaN is generated. It is recommended that the result be one of the input NaNs.

The issue of NaNs is where a lot of implementations really diverged causing problems for the portability of the results across platforms. When two quiet NaNs appear as inputs, different options were implemented by different designers:

1. Compare the significands or the "payloads" of the NaNs and deliver a result according to some precedence mechanism.

2. Always take as a result the first operand of the NaNs.

3. Always produce a “canonical” NaN regardless of the inputs, i.e. neglect the recommendation of the standard.

The first option is complicated and slow. Hence, it is usually rejected by hardware designers optimizing for speed. It also assumes that the significands bear some meaning.

The second option is fast and easy to implement in the hardware datapath. However, it is not commutative,

$$\begin{aligned} NaN_1 + NaN_2 &= NaN_1 \\ NaN_2 + NaN_1 &= NaN_2 \end{aligned}$$

i.e. a change in the operand order in a “commutative” operation such as addition generates a different result!

The third option is equally fast and keeps commutativity. However, the “canonic” quiet NaN on one implementation is non-canonic on another implementation which causes portability problems.

Division by zero

In a division, if the divisor is zero and the dividend is a finite non-zero number a divide by zero exception occurs. If the trap is disabled, the delivered result is a correctly signed infinity.

Overflow and infinities

The overflow flag is raised whenever the magnitude of what would be the result exceeds **max** in the destination format. When traps are disabled the rounding mode and the sign of the intermediate result determine the final result as follows:

	RNE	RZ	RP	RM
+ve	$+\infty$	**+max**	$+\infty$	**+max**
-ve	$-\infty$	**-max**	**-max**	$-\infty$

The infinities are valid operands in many situations.

Example 5.26 Here are a few valid operations involving infinities.

$$\begin{aligned} +\infty + \text{ finite number} &= +\infty. \\ -\infty + \text{ finite number} &= -\infty. \\ \sqrt{+\infty} &= +\infty. \\ \frac{\text{positive finite number}}{-\infty} &= -0. \end{aligned}$$

The standard thus states that unless there is an invalid exception due to some invalid operation where an infinity is an input operand, the arithmetic on infinities is always exact and signals no exceptions.

On the other hand, an operation that generates an infinity as a result will only produce an exception if it is a division by zero exception or the infinity is produced from finite results in the case of an overflow.

If the overflow trap is enabled when an overflow occurs, a value is delivered to the trap handler that allows the handler to determine the correct result. This value is identical to the normal floating point representation of the result, except that the biased exponent is adjusted by subtracting 192 for single precision and 1536 for double precision. This bias adjust has the effect of wrapping the exponent back into the middle of the allowable range. The intent is to enable the use of these adjusted results, if desired, in subsequent scaled operations within the handler with a smaller risk of causing further exceptions.

Example 5.27 Suppose we multiply two large numbers to produce a single precision result:

$$2^{127} \times 2^{127} = 2^{254} \leftarrow \text{ overflow.}$$

The value delivered to the trap handler would have a biased exponent:

$$254 + 127 - 192 = 189.$$

Underflow and subnormal numbers

Similar to the case of overflow, if the underflow trap is enabled the system wraps the exponent around into the desired range with a bias adjust identical to the overflow case, except that the bias adjust is added instead of subtracted from the bias exponent.

If the underflow trap is disabled the number is denormalized by right shifting the significand and correspondingly incrementing the exponent until it reaches the minimum allowed exponent ($exp = -126$). At this point, the hidden '1' is made explicit and the biased exponent is zero. The following example [Coonen, 1979] illustrates the denormalizing process.

Example 5.28 Assume, for simplicity, that we have a single precision exponent and a significand of 6 bits.

Actual result	$=2^{-130} \times 1.01101\cdot$	$\mid\cdots$	$-130 < -126 \Rightarrow$ denormalize
represented as	$=2^{-126} \times 0.000101$	$\mid 101\cdots$	we round (to nearest)
and rounded	$=2^{-126} \times 0.000110$	$\mid$	= the result to be delivered.

The denormalization as a result of underflow is called *gradual underflow* or *graceful underflow*. Of course, this approach merely postpones the fatal underflow which occurs when all the nonzero bits have been right shifted out of the significand. Note that since denormalized numbers and ± zero have the same biased exponent of zero, such a fatal underflow would automatically produce the properly signed zero. The use of a signed zero indicator is an interesting example of taking a potential disadvantage—two representations for the same value—and turning it (carefully!) into an advantage.

When a denormalized number is an input operand, it is treated the same as a normalized number if the operation is an addition or subtraction. If it is possible to express the result as a normalized number, then the loss of significance in the denormalized operand did not affect the precision of the operation and computation proceeds normally. Otherwise, the result is also denormalized.

If an operation uses a denormalized input operand and produces a normalized result, usually a *loss of accuracy* occurs. As an example, suppose we multiply $0.0010\ldots * 2^{-126}$ by $1.000\ldots * 2^{9}$. The result, $1.000\ldots * 2^{-120}$, is a normalized number, but it has three fewer bits of precision than implied.

Example 5.29 Operations on denormalized operands may produce normalized results with or without exceptions noted to the programmer. Some examples are:

$$\begin{array}{rr@{\;\times\;}ll} & 2^{-126} & 0.1000000 & \text{denormalized number} \\ + & 2^{-126} & 0.1000000 & \text{denormalized number} \\ \hline & 2^{-126} & 1.0000000 & \text{normalized number, no exception} \end{array}$$

$$\begin{array}{rr@{\;\times\;}ll} & 2^{-126} & 0.1110000 & \text{denormalized number} \\ \times & 2^{1} & 1.1110000 & \text{normalized number} \\ \hline & 2^{-125} & 1.1010010 & \text{normalized number, no exception} \end{array}$$

Two events contribute to underflow:

1. the creation of a tiny non-zero number less than **min**

2. and an extraordinary loss of accuracy during the approximation of such a tiny nymber by a subnormal number.

When the underflow trap is disabled and both of these conditions occur the underflow exception is signaled. The original standard of 1985 gives the implementers the option to detect tininess in two ways and the loss of accuracy in two ways as well. This obviously lead to different implementations and problems for portability.

Inexact result

Exact result is obtained whenever both the guard bit and the sticky bit are each equal to zero. Any other combinations of the guard and sticky bit implies that a round off error has taken place, in which case the inexact result flag is raised. Said differently, if the rounded result of an operation is not exact or if it overflows (with the overflow trap disabled) this exception is signaled.

One use of this flag is to allow integer calculations with a fast floating point execution unit. The multiplication or addition of integers is performed with the most significant bits of the floating point result *assumed* to be an integer. In such an implementation, the inexact result flag causes an interrupt whenever the actual result extends outside the allocated floating point precision.

Now, let us see if you were able to follow all of this discussion regarding the exceptions of the IEEE 754 standard.

$\overset{?}{\Longrightarrow}$ **Exercise 5.26 According to the definitions of the different exceptions, what is the result of $(+\infty)/(-0)$ and of $\sqrt{-0}$ when traps are disabled and what are the exceptions raised?**

$\overset{?}{\Longrightarrow}$ **Exercise 5.27 Indicate which of the following expressions is valid (i.e. left hand side equals right hand side) under all conditions within the IEEE standard for floating point arithmetic. You must justify your answer whether you think the expression is valid or invalid.**

$$\begin{align}
x/2 &= x * 0.5 \tag{5.1}\\
1 * x &= x/1 = x \tag{5.2}\\
x/x &= 1.0 \tag{5.3}\\
x - y &= x + (-y) = (-y) + x \tag{5.4}\\
x - y &= -(y - x) \tag{5.5}\\
x - x &= 0 \tag{5.6}\\
0 * x &= 0 \tag{5.7}\\
x + 0 &= x \tag{5.8}\\
x - 0 &= x \tag{5.9}\\
-x &= 0 - x \tag{5.10}
\end{align}$$

$\overset{?}{\Longrightarrow}$ **Exercise 5.28 During your task to build the control system for the new rocket launchers, you wrote the code fragment:**

```
if a < b then r = sqrt(b^ 2 - a^ 2)
else r = sqrt(a^ 2 - b^ 2)
```

to run on a processor supporting the IEEE 754-2008 standard. Your manager objects and changes the code to

```
if a < b then r = sqrt(b^ 2 - a^ 2)
else if a >= b then r = sqrt(a^ 2 - b^ 2)
else <do something NOW before the rocket fails>
```

Explain why the manager did so. After careful examination of the new code you discover that it is still unsafe and you propose further improvements. Why is it still unsafe and how can you improve it?

$\overset{?}{\Longrightarrow}$ **Exercise 5.29 For a system that follows the IEEE 754-2008 standard and uses the decimal128 format ($e_{max} = 6144$, $p = 34$), what are the results and flags raised (inexact, overflow, underflow, invalid, and divide by zero) corresponding to the following operations?**

1. $(-1 \times 10^{0}) + (-48881942410202340748464871347 \times 10^{-35})$, **rounded to zero.**
2. $(+1969500000000000000000000000000000 \times 10^{-2626}) + (-8 \times 10^{-2591})$, **rounded to zero.**
3. $fma(-10 \times 10^{3898}, +2 \times 10^{-3898}, -37 \times 10^{0})$, **rounded to plus infinity.**
4. $fma(-16850102 \times 10^{-6176}, +2 \times 10^{-6176}, -0 \times 10^{-2858})$, **rounded to nearest even.**
5. $(-33000000000000000000000000000 \times 10^{-2826})/(+2000000000 \times 10^{-2260})$, **rounded to plus infinity.**
6. $(-1410659489160695397301633101904734 \times 10^{1578})/(-12090 \times 10^{-4546})$, **rounded to minus infinity.**
7. $(-101 \times 10^{5080}) \times (+125958990999010496132607 \times 10^{1032})$, **rounded to minus infinity.**
8. $(+76 \times 10^{-6176}) \times (+1 \times 10^{-6176})$, **rounded to plus infinity.**
9. $sqrt(+9 \times 10^{-6140})$, **rounded to minus infinity.**
10. $sqrt(+276676 \times 10^{-6176})$, **rounded to zero.**

5.6.3 Analysis of the IEEE 754 standard

There seems to be general agreement that the following features of the standard are best for the given number of bits.

- The format of:

S	E	F

- The two levels of precision (SINGLE and DOUBLE).
- The various rounding modes.
- The specification of arithmetic operations.
- The identification of conditions causing exceptions.

However, on a more detailed level, there seem to be many controversial issues, which we outline next.

Gradual underflow

This is an area where a large controversy exists. The obvious advantage of the gradual underflow is the extension of the range for small numbers, and similarly, the compression of the gap between the smallest representable number and zero. For example, in SINGLE precision the gap is $2^{-126} \simeq 1.2\times10^{-38}$ for normalized numbers, whereas the use of denormalized numbers narrows the gap to $2^{-149} \simeq 1.4 \times 10^{-45}$. However, the argument is that gradual underflow is needed not so much to extend the exponent range as to allow further computation with some sacrifice of precision in order to defer as long as possible the need to decide whether the underflow will have significant consequences.

In the early literature regarding the issue, several objections to gradual underflow exist:

1. Payne [Payne and Strecker, 1978] maintains that the range is extended only from 10^{-38} to 10^{-45} (coupled with complete loss of precision at 10^{-45}) and it makes sense only if single precision frequently generates intermediate results in the range 10^{-38} to 10^{-45}. However, for such cases, she believes that the use of single precision (for intermediate results) is generally inappropriate. In fact, since the publication of the standard most implementations on general purpose processors used the double precision or even a wider extended precision for intermediate results.

2. Fraley [Fraley, 1978] objects to the use of gradual underflow for three reasons:

   (a) There are nonuniformities in the treatment of gradual underflow;

   (b) There is no sufficient documented need for it;

   (c) There is no mechanism for the confinement of these values.

3. Another objection to the gradual underflow is the increased implementation cost in floating point hardware. It is much more economical and faster to simply generate a zero output on underflow, and not have to recognize a denormalized number as a valid input.

An alternative approach to denormalized numbers is the use of a pointer to a heap on occurrence of underflow [Payne and Strecker, 1978]. In this scheme, a temporary extension of range can be implemented on occurrence of either underflow or overflow without sacrifice of precision. Furthermore, multiplication (and division) work as well as addition and subtraction. While this scheme seems adequate, or even better than gradual underflow, it also has the same cost disadvantage outlined in number (3) above.

On the other hand, the presence of the subnormal numbers and the gradual underflow preserve an important mathematical property: if M is the set of representable numbers according to the standard then

$$\forall x, y \in M, \qquad x - y = 0 \Longleftrightarrow x = y.$$

In a system that flushes any underflow to zero and does not use denormalized representations, if the difference between two numbers is lower than **min** the returned result is zero.

Example 5.30 Assume that a system uses the single precision format of IEEE but without denormalized numbers. In such a system, what is the result of $1.0 \times 2^{-120} - 1.1111 \cdots 1 \times 2^{-121}$?
Solution: The exact result is obviously

$$\begin{array}{rll} & 1.000 \cdots 0 & \times 2^{-120} \\ - & 0.111 \cdots 1|1 & \times 2^{-120} \\ \hline & 0.000 \cdots 0|1 & \times 2^{-120} = 2^{-144} \end{array}$$

which is not representable in this system. Hence the returned result is zero although the two numbers are not equal.

The systems that flush to zero potentialy have multiple additive inverses to any number.

The use of number representations with less than the normal accuracy in the denormalized range prevents such an effect. It allows all sufficiently small add and subtract operations to be performed exactly.

$\overset{?}{\Longrightarrow}$ **Exercise 5.30 We saw that the subtraction of normalized numbers may produce denormalized numbers. The same effect does not exist within the subnormal range. That is to say, the difference of two denormalized numbers is always a valid denormalized number (or zero if the two numbers are equal). Explain why.**

Significand range and exponent bias.

For binary formats, the standard has a significand in the range $[1, 2($, and the exponent is biased by 127 (in the single precision). These yield a number system with a magnitude between 2^{-126} and $\approx 2^{128}$, thus, the system is asymmetric in such a way that overflow is presumably less likely to happen than underflow. However, if gradual underflow is not used, then the above rationale disappears and one can go back to a PDP-11 format with significand in $[0.5, 1($ and an exponent biased by 128. The PDP-11 single precision numbers have a magnitude between 2^{-128} and $\approx 2^{128}$, such that overflows and underflows are symmetric.

Zeros and infinities.

The IEEE standard has two zero values ($+0$ and -0) and two infinities ($+\infty$ and $-\infty$), and has been called the *two zero system*. An alternate approach, the *three zero system*, is suggested by Fraley [Fraley, 1978]. His system has values $+0$, -0, and 0, $+\infty$, $-\infty$, and ∞.

The basic properties of the two systems are shown below:

2-Zero	3-Zero	Difference
$+0 = -0$	$-0 < 0 < +0$	3 zero system introduces an unsigned zero
$-\infty < +\infty$ or $-\infty = +\infty$	$-\infty < +\infty$ or ∞ not comparable	
$x - x = +0$	$x - x = 0$	
$1/+0 = +\infty$	$1/+0 = +\infty$	
$1/-0 = -\infty$	$1/-0 = -\infty$	
	$1/0 = \infty$	

The main advantage of the three zeros system is the availability of a true zero and a true infinity in the algebraic sense. This is illustrated by the following points.

1. Suppose $f(x) = e^{1/x}$. In the two zeros system we have:

   $$\begin{aligned} f(-0) &= +0, \\ f(+0) &= +\infty; \end{aligned}$$

   thus, $f(-0) \neq f(+0)$, even though $-0 = +0$ as defined by the standard.

   This, of course, is a contradiction of the basic theorem:

   $$\text{if } x = y \text{ then } f(x) = f(y).$$

   By contrast, in the three zeros system, this theorem holds since $-0 \neq +0$.

2. If gradual underflow is not implemented then a two zeros system fails to distinguish zeros that result from underflow from those which are mathematically zero. The result of $x - x$ is $+0$ in the two zeros system. In the three zeros system, $x - x = 0$, whereas $+0$ is the result of an underflow of a positive number; that is,

   $$0 < +0 < \text{smallest representable number.}$$

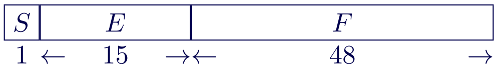
3. In the IEEE 754 standard, if the sum of two operands with different signs or the difference of two operands with the same sign is exactly zero the delivered result is -0 in the case of rounding toward $-\infty$. In all the other rounding modes, the result is $+0$. This choice stands against the definition of $+0 = -0$. A three zeros system delivers the true 0 in such a case for all the rounding modes.

5.7 Cray Floating Point

The IEEE standard is an attempt to provide functionality and information to the floating point user. All floating point designs are necessarily compromises between user functionality and engineering requirements; between function and performance. A clear illustration of this is the Cray Research Corporation floating point design (as used in the CRAY-1 and CRAY-XMP). The Cray format is primarily organized about high speed considerations, providing an interesting contrast to the IEEE standard.

5.7.1 Data Format

As before, the format ($\beta = 2$) consists of sign bit, biased exponent and fraction (mantissa):



where

$$\begin{aligned} S &= \text{sign bit of fraction} \\ E &= \text{biased exponent} \\ F &= \text{fraction} \end{aligned}$$

then

$$\begin{aligned} e &= \text{true exponent} = E\text{-bias} \\ f &= \text{true mantissa} = 0.\text{F} \end{aligned}$$

A normalized nonzero number X would be represented as

$$X = (-1)^S \times 2^{E-\text{bias}} \times (0.\text{F})$$

with a bias $= 2^{14} = 16384$.

5.7.2 Machine Maximum

$$\textbf{max} = 2^{2^{13}-1}(1 - 2^{-48}) = 2^{8191}(1 - 2^{-48}).$$

Note that overflow is strictly defined by the exponent value. Any result with an exponent containing two leading ones is said to have *overflowed*.

5.7.3 Machine Minimum

$$\textbf{min} = 2^{-(2^{13})} \cdot 2^{-1} = 2^{-8193}.$$

Any result with an exponent containing two leading zeros is said to have *underflowed*. There are no underflow interrupt flags on the Cray machines; underflowed results are to be set to zero. Notice the relatively large range of representations that are designated "underflowed" or "overflowed."

To further simplify (and speed up) implementations, the range tests (tests for nonzero numbers which exceed **max** or are under **min**) are largely performed *before postnormalization*! (There is an exception.) To expedite matters still further, range testing is *not* done on input operands (except zero testing)!

This gives rise to a number of curious results:

1. A number below **min**, call it s, can participate in computations. Thus,

   - $(\mathbf{min}+s)-\mathbf{min}=s$, where s is 2^{-2} to 2^{-48} times **min**, since $\mathbf{min}+s>\mathbf{min}$ *before postnormalization.*

     The machine normalizes such results producing a number up to 2^{-48} smaller than **min**. This number is not set to zero.
   - $\mathbf{min}+s$ is produced as the sum of **min** and s.
   - $s+0=0$, since now the invalid exponent of s is detected in the floating point adder result.
   - $s\times 1.0=0$ if s is less than $2^{-1}\times\mathbf{min}$, since the sum of exponents is less than **min** (recall 1.0 is represented by exponent = 1, fraction = 1/2).
   - $s\times 1.0=s$ if $\mathbf{min}>s>2^{-1}\times\mathbf{min}$, since the sum of the exponents *before* postnormalization is equal to **min**.
   - $s\times Y=0$ if the exponent of Y is not positive enough to bring $\exp(s)+\exp(Y)$ into range.
   - $s\times Y=s\times Y$ if $\exp(s)+\exp(Y)\geq\exp(\mathbf{min})$.

2. On overflow, the machine may be interrupted (maskable). An uninterrupted overflow is represented by $\exp(\mathbf{max})+1$ or 11000..0 (bias $+2^{13}$) in the exponent field (actually, 11xx...x indicates an overflow condition). The fraction may be anything.

3. Overflow checking is performed on multiply. If the upper bits of the exponent are “1”, the result is set to “overflow” (exponent = 1100...0) unless the other argument is zero (exponent = 000..0, fraction = xx...x), in which case the result is zero (all zeros exponent and fraction).

4. Still, it is possible to have the following:

   $$\mathbf{max}\times 1.0=\mathbf{max}\ \textit{with overflow flag set}.$$

   This is because 1.0 has exp = 1, which causes the result exponent to overflow *before* postnormalization.

5. The input multiplier operands are not checked for underflow, as just illustrated.

5.7.4 Treatment of Zero

Cray represents zero as all 0's (i.e., positive sign) and sets a detected underflowed number to zero. The machine checks input operands for zero by exponent inspection only. Further, the Cray machine uses the floating point multiplier to do integer operations. Integers are detected by having all zeros in the “exponent” field for *both* operands. If only one operand of the multiplier has a zero exponent, that operand is interpreted as floating point zero and the result of multiplication is zero regardless of the value of the other operand. Thus,

$$(zero)\times(+overflow)=zero,$$

since zero takes precedence. Zero is (almost) always designated as +00...0. Thus, even in this case:

$$(+zero)\times(-overflow)=+zero.$$

However, in the case of

$$(+zero) \times (-zero) = -zero,$$

since both exponents are zero, the operands are interpreted as valid *integer* operands and the sign is computed as such. However,

$$(-zero) \times (Y) = (+zero)$$

for any nonzero value of Y, since +zero is "always" the result of multiplication with a zero exponent operand.

5.7.5 Operations

The Cray systems have three floating point functional units:

- Floating Point Add/Subtract.
- Floating Point Multiplication.
- Floating Point Reciprocal.

On floating point add/subtract, the fraction result is checked for all zeros. In this case, the sign is set and the exponent is set to all zeros. No such checking is performed in multiplication.

5.7.6 Overflow

As mentioned earlier, overflow is detected on the results of add and multiply, and on the input operands of multiply. In overflow detection, the result exponent is set to exp(**max**)+1—two leading exponent "1"s followed by "0"s. The fraction for all operations is unaffected by results in an overflow condition.

The exceptions to the test for over/underflow on result (only) before postnormalization are two:

- The input argument to multiply are tested for overflow.
- The result of addition is tested for overflow (also) after postnormalization. This is (in part) a natural consequence of the operation

  $$\mathbf{max} + \mathbf{max} = overflow$$

  and the overflow flag is set. Also,

  $$(-\mathbf{max}) + (-\mathbf{max}) = -overflow.$$

  The sign of the overflow designation is correctly set.

Table 5.6: Underflow/overflow designations in Cray machines.

		test on input	test on output before post-normalization	test on output after post-normalization
underflow	+/−	No	Yes	No
	×	No	Yes	No
overflow	+/−	No	Yes	Yes
	×	Yes	Yes	No

Thus, the "overflow" designation is somewhat "firmer" than "underflow." Table 5.6 illustrates the difference.

Since fractions are not checked on multiply, some anomalies may result, such as:

$$\text{overflow} \times 0.0 \times 2^1 = \text{overflow with } 0.0 \text{ fraction.}$$

This quest for speed at the cost of correct functionality sometimes is justified in some specific applications. When it comes to three dimensional graphics animation, an error in a few pixels in a frame that flashes on the screen and is followed by so many other frames within a second is definitely tolerable. In general signal processing whether the signal is audio, video, or something else is a domain that tolerates a number of errors and the designer should not restrict the design with the requirements of a standard such as the IEEE 754.

The danger comes, however, when such a design philosophy is applied beyond the original application of the design. If fast and inaccurate results are delivered in scientific or financial computations catastrophes might occur. Due diligence is required to handle the numbers correctly and to report any exceptions that occur. The software getting such exceptions must also deal with them wisely. Otherwise, an accident similar to the blast of the Ariane V space shuttle[1] in 1996 might repeat.

5.8 Additional Readings

Sterbenz [Sterbenz, 1974] is an excellent introduction to the problem of floating point computation. It is a comprehensive treatment of the earlier approaches to floating point representation and their difficulties.

The January 1980 and March 1981 issues of IEEE *Computer* have several valuable articles on the proposed standard; Stevenson [Stevenson, 1981] provides a precise description of what was proposed in 1981 for the IEEE 754 standard with good introductory remarks.

Cody [Cody, 1981] provides a detailed analysis of the three major proposals in 1981 and shows the similarity between all of them.

[1] The cause in that accident was an overflow of a conversion from a floating point to integer operation. The ADA language used in that system had a policy of aborting on any "arithmetic error". In this case the overflow was not a serious error but the system aborted and equipements worth millions of dollars were blown in the air!

Coonen [Coonen, 1981] gives an excellent tutorial on underflows and denormalized numbers. He attempts to clear the misconceptions about gradual underflows and shows how it fits naturally into the proposed standard.

Hough [Hough, 1981] describes applications of the standard for computing elementary functions such as trigonometric and exponential functions. This interesting article also explains the need for some of the unique features of the standard: extended formats, unbiased rounding, and infinite operands.

Coonen [Coonen, 1980] also published a guide for the implementation of the standard. His guide provides practical algorithms for floating point arithmetic operations and suggests the hardware/software mix for handling exceptions. His guide also includes a narrative description of the standard, including the QUAD format.

Kahan provides [Kahan, 1996] more details on the status of the standard, features, and examples. A recent interview with him [Severance, 1998]) describes the history of the standard.

5.9 Summary

Pairs of *signed* integers can be used to represent approximations to real numbers called floating point numbers. Floating point representations broadly involve tradeoffs between precision, range, and implementation problems. With the relatively decreasing importance of implementation costs, the possibility of defining more suitable floating point representations has led to efforts toward a standard floating point representation.

We discussed the details of the IEEE 754 standard and contrasted it to other prior *de facto* standards. If in a specific design, the features of the standard are deemed too cumbersome a designer can use the tools we presented in this chapter to evaluate the points of strengths and weaknesses in any proposed alternatives. However, the designer should wisely define the operations on the chosen format and clearly define the outcomes in the case of exceptions. Such a clear definition enables future designers to decide whether such choices are suitable to their systems or not.

5.10 Problems

Problem 5.1 For a variety of reasons, a special purpose machine is built that uses 32-bit representation for floating point numbers. A minimum of 24 bits of precision is required.

Compare a IBM S/370-like (radix=16 and with truncation only) system to the IEEE binary32 system with respect to

1. the range,
2. the precision, and
3. the associative, commutative, and distributive properties of basic arithmetic operations. (In which cases do the properties fail?)

Problem 5.2 Indicate which of the following expressions is valid (i.e. left hand side equals right hand side) under all conditions within the IEEE standard for floating point arithmetic. You must justify your answer whether you think the expression is valid or invalid.

$$x/2 = x * 0.5 \tag{5.11}$$
$$1 * x = x/1 = x \tag{5.12}$$
$$x/x = 1.0 \tag{5.13}$$
$$x - y = x + (-y) = (-y) + x \tag{5.14}$$
$$x - y = -(y - x) \tag{5.15}$$
$$x - x = 0 \tag{5.16}$$
$$0 * x = 0 \tag{5.17}$$
$$x + 0 = x \tag{5.18}$$
$$x - 0 = x \tag{5.19}$$
$$-x = 0 - x \tag{5.20}$$

Problem 5.3 For IEEE single precision (binary32), if $A = (1).0100\ldots\times 2^{-126}$, $B = (1).000\ldots\times 2^{-3}$, and $C = (1).000\ldots\times 2^{5}$ (A, B, and C are positive):

1. What is the result of $A * B * C$, RM round, if performed $(A * B) * C$?
2. Repeat, if performed $A * (B * C)$.
3. Find $A + B + C$, RP round.
4. If $D = (1).01000\ldots\times 2^{122}$, find $C * D$, RP round.
5. Find $(2 * C) * D$, RZ round.

Problem 5.4 All of the floating point representations studied use sign and magnitude to represent the mantissa, and excess code for the exponent. Instead, consider a floating point representation system that uses radix 2 complement coding for both the mantissa and the exponent for a binary based system.

1. If the magnitude of a normalized mantissa is in the range $1/2 < m < 1$, where is the implied binary point?
2. Can this representation make use of a technique similar to the hidden one technique studied in class? If so, which bit is hidden and what is its value? If not, why not?

Problem 5.5 In each of the following, you are given the ALU output of floating point operations *before* post-normalization and rounding. An IEEE-type format is assumed, but (for problem simplicity) only four bits of fraction are used (i.e., a hidden "1".xxx, plus three bits)—and three fraction bits are stored.

> M is the most significant bit, immediately to the left of the radix point.
> X are intermediate bits.
> L is the least significant bit.
> S is the sign (1 = neg., 0 = pos.)

(1) Show results after post-normalization and rounding—exactly the way the fraction will be stored. (2) Note the effects (the change) in exponent value.

1. Result after subtraction, round RZ

S	$M.X\,X\,L\,G\,R\,S$
1	0 0 0 0 1 0 0

Result after post-normalization and round:

S significand change to exponent

_ __________ __________________

2. Result after multiplication, round RNE

S	$M.X\,X\,L\,G\,R\,S$
0	1 0 1 0 1 0 1 0

Result after post-normalization and round:

S significand change to exponent

_ __________ __________________

3. Result after multiplication, round RNE

S	$M.X\,X\,L\,G\,R\,S$
0	1 1 1 1 1 1 0 0

Result after post-normalization and round:

S significand change to exponent

_ __________ __________________

4. Result after addition, RM

S	$M.X\,X\,L\,G\,R\,S$
1	1 0 1 1 0 0 0 1

Result after post-normalization and round:

S	significand	change to exponent
_	__________	________________

Problem 5.6 On page 168, there is an action table for RNE. Create a similar table for RP. State all input bits used and all actions on the final round bit, A.

Problem 5.7 For a system that follows the IEEE 754-2008 standard and uses the decimal64 format ($e_{max} = 384$, $p = 16$), what are the results and flags raised (inexact, overflow, underflow, invalid, and divide by zero) corresponding to the following operations?

1. $(+0 \times 10^{-216}) + (-0 \times 10^{-306})$, rounded away from zero.
2. $(-20000000000000 \times 10^{-14}) \times (-5 \times 10^{-398})$, rounded towards zero.
3. $(-80 \times 10^{362}) \times (-125 \times 10^{19})$, rounded to nearest ties to even.
4. $(+20 \times 10^{-30}) \times (+5657950712797142 \times 10^{-368})$, rounded to nearest ties away from zero.
5. $(-8672670147962662 \times 10^{159})/$ sNaN, rounded to nearest ties to even.
6. $(-8628127745585310 \times 10^{-214})/(+4403614193461964 \times 10^{207})$, rounded to minus infinity.
7. $(+1712988626697436 \times 10^{-375})/(-2308774070921686 \times 10^{96})$, rounded to plus infinity.
8. $(+9999999999969645 \times 10^{369}) - (-303540000023000 \times 10^{359})$, rounded to nearest ties to zero.

Problem 5.8 Assume that we have a 'single precision decimal system' with two digits. If we add 0.54×10^2 and 0.42×10^4 the exact result is 0.4254×10^4 but suppose that the hardware uses an internal three digits notation and rounds the result to 0.425×10^4. When this internal result is saved, the round to nearest mode yields 0.42×10^4. However, if the round to nearest mode were applied to the original exact result we get 0.43×10^4. This is an example of what is known as "double rounding" errors.

1. In the regular binary floating point representation, does double rounding lead to a problem in the round to zero mode? What about the round to nearest up (the 'normal' rounding for humans)?
2. If we round the sum $x + y$ of two floating point numbers x and y each having t-bits in its significand to a precision t' such that $t' \geq 2t + 2$ prove that a second rounding to t bits yields the same result as a direct rounding to t bits of the exact result regardless of the rounding mode. (That is to say double rounding does not cause a problem if the first rounding is to a wide enough precision.)

3. Show that the statement of question 2 holds also for multiplication. (Note: it is also true for division and square root but you do not need to prove it for those two now!)
4. A designer claims that an IEEE double precision floating point unit for addition, multiplication, division, and square root can always produce correct results for the single precision calculations as well. Discuss the correctness of this claim based on the results of this problem.

Problem 5.9 In a system with an odd radix β, a number whose value is $\mathcal{D} = \sum_i d_i \beta^i$ is represented by $d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots$ where $\forall_i \frac{-\beta+1}{2} \leq d_i \leq \frac{\beta-1}{2}$.

1. Is this a redundant system?
2. Prove that for any $j \leq n$, $\left|\sum_{i=-\infty}^{j-1} d_i \beta^i\right| \leq \frac{1}{2}\beta^j$.
3. Is the round to zero equal to a truncation in such a system?
4. If the representation is finite (i.e. the minimum i is not $-\infty$), prove that the round to nearest is equal to a truncation.

Problem 5.10 In a system with an *even* positive radix β, a number whose value is $\mathcal{D} = \sum_i d_i \beta^i$ is represented by $d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots$ where $-\frac{\beta}{2} \leq d_i \leq \frac{\beta}{2}$ for all values of i and if $|d_i| = \frac{\beta}{2}$ then the first non-zero digit that follows on the right has the opposite sign, that is, the largest $j < i$ such that $d_j \neq 0$ satisfies $d_i \times d_j < 0$.

1. For a finite representation (i.e. the minimum i is not $-\infty$ but a certain finite value ℓ), do all numbers have unique representations in this system? Clearly give your reasons.
2. Prove that for any $j \leq n$, $\left|\sum_{i=\ell}^{j-1} d_i \beta^i\right| \leq \frac{1}{2}\beta^j$ and that the only way of representing $\frac{1}{2}\beta^j$ in this system starting from position $j-1$ and going down to position ℓ is $(\frac{\beta}{2}000 \cdots 000)$.
3. Prove that the truncation of a number at a certain position is equivalent to a type of round to nearest. Please indicate clearly what happens in the tie cases (when the number is exactly at the mid point between the two nearest numbers).
4. Does this number system suffer from the double rounding problem in the type of round to nearest mentioned above? What about round to zero, round to plus infinity, and round to minus infinity?

Problem 5.11 Indicate whether each of the following statements is true or false. If it is true prove it, otherwise give a counterexample.

1. According to the IEEE 754 floating point standard, the results of $\sqrt{1/b}$ and $1/\sqrt{b}$ are the same for all the possible inputs and will give the same exceptions.
2. For a normalized binary floating point number b with a significand represented in p bits, if the result of $\sqrt{b}$ is exact it can never be represented in exactly $p+1$ bits. Hence, in the round to nearest, we never get the tie case.

Problem 5.12 Let $\underset{\nabla}{\times}$ denote the result of multiplication rounded down to minus infinity and $\overset{\Delta}{\times}$ denote the result of multiplication rounded up to plus infinity. Given an interval representation of two numbers $A = [a_1, a_2]$ and $B = [b_1, b_2]$ where a_1, a_2, b_1, and b_2 are floating point numbers that may take positive or negative values, i.e. the interval may even contain zero as one of its elements. Derive the rules to calculate the result of the interval representing $A \times B$.

Problem 5.13 In interval arithmetic, the intervals may be represented as $[\ell, u]$ where ℓ is the lower bound and u is the upper bound of the interval (called "inf-sup" notation) or as (m, r) where m is the middle point and r the radius of the interval (called "mid-rad" notation). For example a resistor with a nominal resistance value of 3000Ω which has a 10% tolerance may be represented as $[2700, 3300]$ or as $(3000, 300)$.

1. Your colleague suggests that the conversion from the first to the second notation may be done as $m = 0.5(\ell + u)$ and $r = 0.5(u - \ell)$. You strongly object and suggest that $m = \ell + 0.5(u - \ell)$ and $r = 0.5(u - \ell)$ is safer in finite range and precision arithmetic and gives the correct result more often than your colleague's form. Please explain why it is safer.

2. Another colleague suggests yet a different form: $m = 0.5\ell + 0.5u$ and $r = 0.5u - 0.5\ell$. Which is better, your form or his? Why?

3. Try to minimize the number of addition and multiplication operations needed in each of the three forms. Do not assume that division by two is a simple shift since the numbers may be decimal not binary. You should rather assume that the multiplication by 0.5 is an operation.

4. To check if the inf-sup and mid-rad notations are equivalent you are asked: "are there intervals represented in one notation that cannot be represented in the other?" If yes, please give examples. If no, prove it.

Problem 5.14 Your friend claims that for finite floating point numbers (binary or decimal formats), the two successive program instructions $c = a + b$ and $d = c - b$ lead to $d = a$ always and uses this idea in a program. You should either prove this identity as a general statement for all cases of the IEEE standard or disprove it by an example if it fails under some conditions.

Problem 5.15 You are adding support for the division operation to a system that uses the binary and decimal formats of the IEEE 754–2008 standard.

1. Prove: "For binary floating point numbers a and b with significands represented in p bits each, if the result of $\frac{a}{b}$ is exact it can never be represented in exactly $p + 1$ bits. Hence, in the round to nearest, we never get the tie case." (Remember to check for the normal and subnormal numbers.)

2. Assume that you want to implement the following rounding directions for binary FP division: Round To Zero (RTZ), Round Away from Zero (RAZ), Round to Plus Infinity (RPI), Round to Minus Infinity (RMI), as well as three round to nearest with the tie cases resolved as: to even (RNE), to zero (RNZ), and away from zero (RNA). Given that the statement of part 1 is true, indicate which of those rounding directions are exactly equivalent and explain why.

3. Prove: "For decimal floating point numbers a and b with significands represented in p digits each, if the result of $\frac{a}{b}$ is exact it is possible in some cases to be represented in exactly $p+1$ digits. Hence, in the round to nearest, we may get the tie case."

Problem 5.16 You are adding support for the square root operation to a system that uses the IEEE decimal64 format.

1. Consider the following statement: "The square root operation never raises the divide by zero, overflow, or underflow flags; the only flags that may be raised due to this operation are the invalid and inexact flags." Indicate your reasons to say whether the statement is true or false.
2. Prove: "For a decimal floating point number d with a significand represented in p digits, if the result of $\sqrt{d}$ is exact it can never be represented in exactly $p+1$ digits. Hence, in the round to nearest, we never get the tie case."
3. Assume that you want to implement the following rounding directions: Round To Zero (RTZ), Round Away from Zero (RAZ), Round to Plus Infinity (RPI), Round to Minus Infinity (RMI), as well as three round to nearest with the tie cases resolved as: to even (RNE), to zero (RNZ), and away from zero (RNA). Given that the statement of part 2 is true, indicate which of those rounding directions are exactly equivalent and explain why. Hint: remember that the result of the square root is always positive (except for -0 but you may assume that it is handled separately).

Problem 5.17 A designer of a fast fourier transform uses two's complement representation for the numbers.

1. What are the necessary bits to check in order to correctly round the results according to each of the following directions:
   (a) Round to zero,
   (b) Round to positive infinity,
   (c) Round to negative infinity,
   (d) Round to nearest where tie cases should round to the odd number, and
   (e) Round to nearest where tie cases should round up?
2. Write the truth table for the bit to add in order to achieve the correct rounding according to each of the directions mentioned above.

Problem 5.18 The normal technique in the Round to Nearest Even (RNE) mode of the IEEE standard is to add the rounding action to the least significant bit (L) of the result. Instead of that, a friend suggests to add a new rounding action to the guard bit (G). He calls this new rounding action the rounding carry (C_r) since it is added to G and propagates a carry to L only in the needed cases. After he adds C_r and any possible propagation occurs, he truncates the positions of the old G and S where S is the sticky bit.

1. Write the truth table giving the value of C_r for all the combinations of L, G, and S. Clearly indicate all the “don’t care” cases.

2. Now, your friend wants to integrate his new idea for the rounding with the adder in order to decide the rounding on the pre-normalized bits N', L', G', and $(R' + S')$. (N' is the bit next to L' and R' is the rounding bit which is ORed with S' to form the new sticky $R' + S'$). You should analyze the case where the result *may need* a single bit right shift. Write a truth table (such as the following one) for C_r(pre) based on the pre-normalized bits and for the final C_r(post) indicating the don’t care cases for both.

Without right shift					After 1-bit right shift			
N'	L'	G'	$(R' + S')$	C_r(pre)	L	G	S	C_r(post)
0	0	0	0		0	0	0	
⋮					⋮			
0	1	0	0		0	1	0	
⋮					⋮			
1	0	1	1		1	0	1	
⋮					⋮			
1	1	1	1		1	1	1	

3. Now choose the values of the don’t care cases in order to minimize the difference between the two C_r columns and give the logical equation giving C_r(pre) based on N', L', G', and $(R' + S')$.

4. Repeat the above for the case when the result may need a single bit left shift.

5. Finally, combine all the ideas taking into consideration that a right shift may occur only in the case of an effective addition and a left shift may occur only in the case of an effective subtraction.

Problem 5.19 In the IEEE 754-2008 standard, the function $hypot(a, b) = \sqrt{a^2 + b^2}$ is defined among many other elementary functions in clause 9.

1. In each of the binary64 and decimal64 formats, what is the result of $c = hypot(a, b)$ if $a = 3 \times 10^{160}$ and $b = 4 \times 10^{160}$ and the function is implemented directly as $sqrt(a^2 + b^2)$? Why?

2. To implement a better version, a colleague suggests:

```
x = maxabs(a,b);
n = minabs(a,b);
c = (x == 0.0) ?  0.0 :  x*sqrt(1 + (n/x)^ 2);
```

   where `maxabs` is the maximum absolute value and `minabs` is the minimum absolute value. What is the result for the values given above for a and b in each of the two formats? Why?

3. The standard further specifies the special cases $hypot(\pm 0, \pm 0) = +0$, $hypot(\pm\infty, qNaN) = +\infty$, and $hypot(qNaN, \pm\infty) = +\infty$. Suggest an implementation that provides these results correctly as well as avoiding the problems you noted in the previous two points.

4. What should the result be if both operands are $qNaN$? What if one or both operands are $sNaN$?

Chapter 6

Division (Incomplete chapter)

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.

6.1 Subtractive Algorithms: General Discussion

6.1.1 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.

6.1.2 Pencil and Paper Division

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

$$\begin{array}{rllll}
 & 1 & 5 & 1 & 2 \\ \cline{2-5}
3) & 4 & 5 & 3 & 7 \\
 & 3 & & & \\ \cline{2-3}
 & 1 & 5 & & \\
 & 1 & 5 & & \\ \cline{2-3}
 & & & 3 & \\
 & & & 3 & \\ \cline{4-5}
 & & & & 7 \\
 & & & & 6 \\ \cline{5-5}
 & & & & 1
\end{array}$$

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:

$$\begin{array}{rl}
 & 1512 \leftarrow \text{Quotient} \\ \cline{2-2}
3) & 4537 \leftarrow \text{Dividend} \\
 & 3000 \leftarrow \text{Divisor}*q(MSD)*10^3 \\ \cline{2-2}
 & 1537 \leftarrow \text{Partial remainder} \\
 & 1500 \\ \cline{2-2}
 & 0037 \\
 & 0030 \\ \cline{2-2}
 & 0007 \\
 & 0006 \\ \cline{2-2}
 & 0001 \leftarrow \text{Remainder}
\end{array}$$

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:

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

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:

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

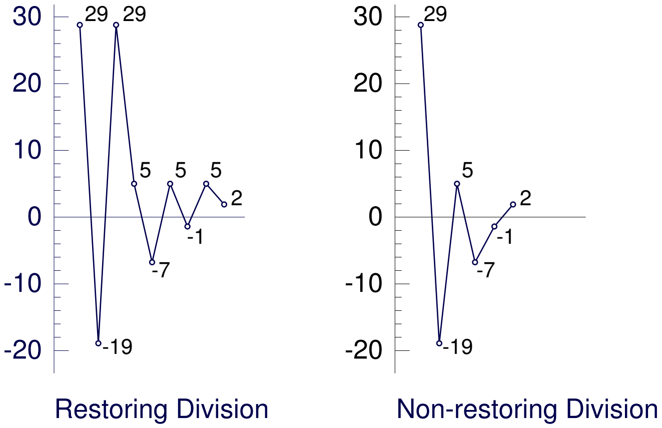


Figure 6.1: Graphical illustration of partial remainder computations in restoring and nonrestoring division.

The left side of Figure 6.1 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.*

6.2 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 Flynn [1970] 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.

6.2.1 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 $1+x$.

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

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

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

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 Anderson et al. [1967], 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 $\left|X^{32}\right|\leq 2^{-32}$; 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 $b_0\cdots b_i xxx\cdots$ have same leading digits, they will map into the same table entry *regardless* of the value of $0.00\cdots0xxxx\cdots$. How, then, does the algorithm use the different trailing digits to form the proper quotient?

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

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 $.b_0 \cdots b_8 xxxx \cdots$). Then, since $b = 1+X$:

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

$$\underbrace{\underbrace{\underbrace{(1+X)(1-X)}_{(1-X^2)}(1+X^2)}_{(1-X^4)}(1+X^4)}_{(1-X^8)}$$

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, ϵ_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 $1.xx \cdots x$, 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 ϵ_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 ϵ_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.

6.2.2 The Newton–Raphson Division

The Newton–Raphson iteration is based on the following procedure to solve the equation $f(X) = 0$ Thomas [1962]:

- 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.

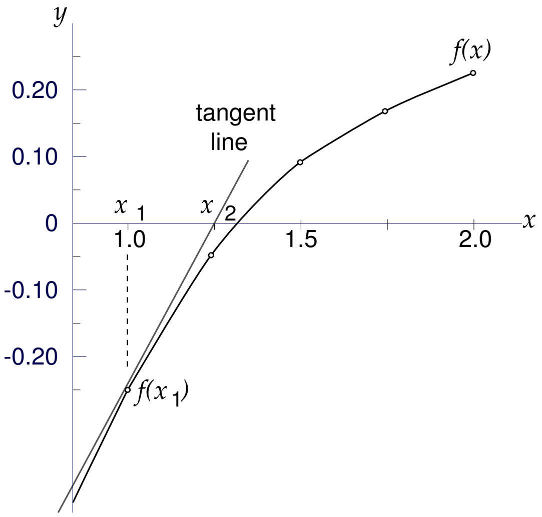


Figure 6.2: Plot of the curve $f(X) = 0.75 - \frac{1}{\mathrm{X}}$ and its tangent at $f(X_1)$, where $X_1 = 1$ (first guess). $f'(x_1) = \frac{\delta y}{\delta x}$.

- From Figure 6.2, 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 $f(X) = \frac{1}{\mathrm{X}} - b = 0$ 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 6.1 Find $\frac{1}{b}$ to at least three decimal digits where $b = 0.75$. Include a calculation of the error $= \epsilon$.
Solution: We start by $X_0 = 1$ and iterate.

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

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

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

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 $b < 1$).

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 1024×8 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

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

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

Thus, the Newton–Raphson iteration on $f(X) = \frac{1}{\mathrm{X}} - b$ and the binomial expansion of $\frac{1}{b} = \frac{1}{1+\mathrm{X}}$ are different ways of viewing the same algorithm Flynn [1970].

6.3 Additional Readings

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

Undheim [Undheim, 1980] 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 [Birkner, 1980] 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 [Cheng and K., 1980] and McMinn [McMinn, 1980] 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 [Robertson, 1956] was a leader in the development of such algorithms. Both Hwang [Hwang, 1978] and Spaniol [Spaniol, 1981] contain reviews of this literature.

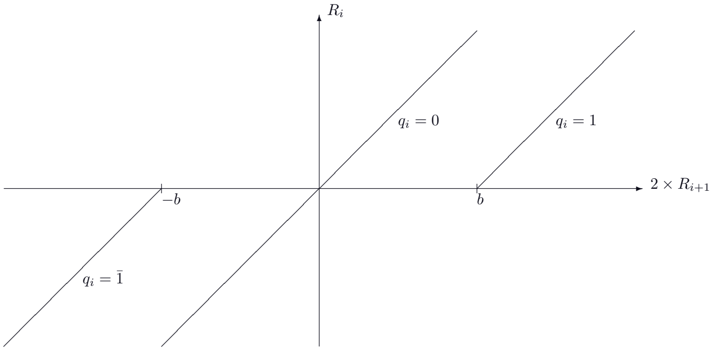
Flynn [Flynn, 1970] provides a review of multiplicative division algorithms.

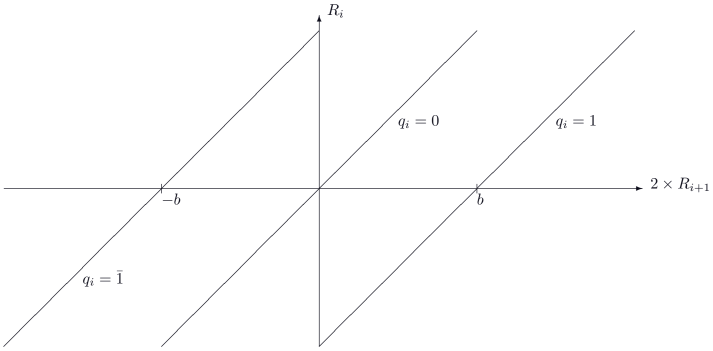
6.4 Problems

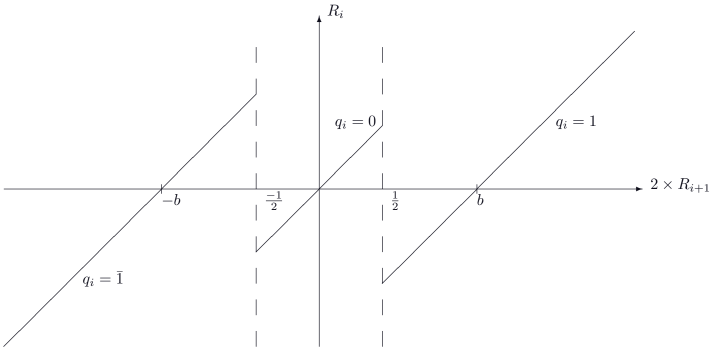
Problem 6.1 Using restoring two's complement division, perform $\frac{a}{b}$ where $a = 0.110011001100$ and $b = 0.100111$. Show each iteration.

Repeat the above using nonrestoring two's complement division.

Problem 6.2 Use each of the following diagrams to perform a/b where $a = 0.110011001100$ and $b = 0.100111$. Show each iteration.







Problem 6.3 Using the Newton-Raphson iteration $x_{i+1} = x_i(2 - bx_i)$, compute $1/b$ where

$b = 0.9$; $b = 0.6$; $b = 0.52$.

Problem 6.4 Another suggested approach uses:

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

(This recursion is unstable and converges very slowly.) Repeat problem 6.3 for this function.

Problem 6.5 Construct a look-up table for two decimal digits (20 entries only; i.e., divisors from 0.60 to 0.79). Use this table to find $b = 0.666$.

Problem 6.6 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.

1. Find the iteration.
2. Compute the error term.
3. Use this to find $1/b$ when $b = 0.9$ and $b = 0.6$.
4. Comment on this algorithm as compared to that described in the text.

Problem 6.7 A hardware cube-root function $a^{1/3}$ is desired based on the Newton-Raphson iteration technique. Using the function

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

1. Find the iteration ($x_{i+1} = \text{to}1cm$).
2. Find the first two approximations to $.58^{1/3}$ using the iteration found in (a).
3. 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!*

Problem 6.8 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.

Problem 6.9 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:

1. Will the iteration converge to $1/b$?

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

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

2. Find the iteration.

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

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

3. Is this a practical scheme—will it work in a processor?

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

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

4. Is it better than the scheme outlined in the chapter?

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

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

Problem 6.10 Use the regular Newton-Raphson technique and the function $f(x) = \frac{1}{x^2} - b$ to find out the iteration equation for the inverse square root of b.

What are the first three iterations for $b = 1.5$ and the initial approximation $x_0 = 1$?

If we define $d_0 = 1 - x_0^2 b$, then $\frac{1}{\sqrt{b}} = x_0(1 - d_0)^{-0.5}$ and we get the iterations

$$x_{i+1} = x_i(1 + \frac{1}{2}d_i + \frac{3}{8}d_i^2 + \cdots).$$

What iteration do you get if only the first two terms up to $\frac{d_i}{2}$ are used?

Prove that, if three terms are used (i.e. up to $\frac{3}{8}d_i^2$), the error ϵ_{i+1} after the iteration $(i+1)$ is dominated by a term with the factor ϵ_i^3. (The iteration produces roughly 3 times the number of correct bits in the previous approximation.)

Problem 6.11 Assume that we want to use the Newton-Raphson method to build the hardware within a floating point unit implementing the IEEE standard to find $\beta = \frac{1}{b^{\frac{1}{p}}}$ for integer values of p. A friend suggested using the function

$$\phi(x) = 1 + \frac{1}{b(x-1)^p}$$

1. The iterations on $\phi(x)$ will not converge directly to β, but to a related quantity α. Find an expression for α.

2. What is the value of p for the reciprocal?

3. For the case of the reciprocal,

   (a) find the iteration,

   (b) find the error term, and

   (c) comparing it to the standard iteration using $f(x) = \frac{1}{x} - b$, which algorithm is better and why?

4. Another friend claims that it is not possible to find a value for p to use in $\phi(x)$ in order to get the reciprocal square root. Prove or disprove this statement by defining the set of possible values of p.

5. For the *positive* values within the set of possible values of p obtained in the previous part, show that the relation giving the error term $\epsilon_{i+1} = x_{i+1} - \alpha$ is a function of ϵ_i^2 and higher order terms only. Your relation should have p as a parameter in it. You might need the binomial expansion: $(u+v)^n = \sum_{m=0}^{m=n} \frac{n!}{m!(n-m)!} u^{n-m} v^m$.

Solutions

We show here the solutions to the exercises present in the book. It is really quite important that you try solving each exercise before looking at the solution in order to benefit from it. Even if you think that you managed to solve the exercise, it is still recommended to read the solution since we sometimes discuss other related topics and point to new things.

Solutions to Exercises

Exercise 1.1. We simply use the definition of the modulo operation and the properties of arithmetic. $N' = N\mathbf{mod}_\mu$ and $M' = M\mathbf{mod}_\mu$ mean that $N = N' + k\mu$ and $M = M' + \ell\mu$ where k and ℓ are integeres. Hence,

$$\begin{aligned}(N[+,-,\times]M)\mathbf{mod}_\mu &= ((N'+k\mu)[+,-,\times](M'+\ell\mu))\mathbf{mod}_\mu \\ &= ((N'[+,-,\times]M') + (k[+,-,\times]\ell)\mu))\mathbf{mod}_\mu \\ &= (N'[+,-,\times]M')\mathbf{mod}_\mu\end{aligned}$$

Exercise 1.1

Exercise 1.2(a) In signed division, the magnitude of the quotient is independent of the signs of the divisor and dividend.

Numerator	Denominator	Quotient	Remainder
11	5	2	1
11	-5	-2	1
-11	5	-2	-1
-11	-5	2	-1

↩

Exercise 1.2(b) For the modulus division, the remainder is the least positive residue.

Numerator	Denominator	Quotient	Remainder
11	5	2	1
11	-5	-2	1
-11	5	-3	4
-11	-5	3	4

↩

Exercise 1.3. For a number N and divisor D, $\frac{N}{D} = q + \frac{r}{D}$ as long as $D \neq 0$.

Case 1: $(N \geq 0) \Rightarrow q_s = q_m$ and $r_s = r_m$

Case 2a: $(N < 0,\ D > 0$ and $r_m = 0) \Rightarrow q_s = q_m$ and $r_s = r_m$

Case 2b: $(N < 0,\ D > 0$ and $r_m > 0) \Rightarrow q_s = q_m + 1$ and $r_s = r_m - D$

Case 3a: $(N < 0,\ D < 0$ and $r_m = 0) \Rightarrow q_s = q_m$ and $r_s = r_m$

Case 3b: $(N < 0,\ D < 0$ and $r_m > 0) \Rightarrow q_s = q_m - 1$ and $r_s = r_m + D$

Moral of the story: Always look for the limiting cases.

Exercise 1.3

Exercise 1.4. Here too, we assume $D \neq 0$ and $\frac{N}{D} = q + \frac{r}{D}$.

Case 1: $(D > 0) \Rightarrow q_f = q_m$ and $r_f = r_m$

Case 2a: $(D < 0, \text{ and } r_m = 0) \Rightarrow q_f = q_m \text{ and } r_f = r_m$

Case 2b: $(D < 0 \text{ and } r_m \neq 0) \Rightarrow q_f = q_m - 1 \text{ and } r_f = r_m + D$

Once more, always look for the limiting cases.

Exercise 1.4

Exercise 1.5. The maximum value is when all the digit values (d_i) are equal to $\beta - 1$. Hence, the maximum value of

$$\begin{aligned} N &= \sum_{i=0}^{i=m} (\beta - 1)\beta^i \\ &= \sum_{i=0}^{i=m} \beta^{i+1} - \sum_{i=0}^{i=m} \beta^i \\ &= (\beta^{m+1} + \beta^m + \beta^{m-1} + \cdots + \beta) - (\beta^m + \beta^{m-1} + \cdots + \beta + 1) \\ &= \beta^{m+1} - 1 \end{aligned}$$

Thus, $\beta^{m+1} > N \geq 0$.

Exercise 1.5

Exercise 1.6. According to our definitions, N is represented as $d_m \cdots d_0$ with $m = n - 1$. Hence, N is less than β^n and the operation $\beta^n - N$ is actually:

$$\overbrace{\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ - & d_m & d_{m-1} & d_{m-2} & d_{m-3} & \cdots & d_0 \\ \hline \end{array}}^{n \text{ digits}}, \text{ recall } m = n - 1.$$

For all lower order digits which satisfy $d_0 = d_1 = \cdots = d_i = 0$, the operation is

$$\begin{array}{cccc} & 0 & \cdots & 0 \\ - & 0 & \cdots & 0 \\ \hline & 0 & \cdots & 0. \end{array}$$

and the $\mathbf{RC}(d)_i$ is

$$\mathbf{RC}(d)_0 = \mathbf{RC}(d)_1 = \cdots = \mathbf{RC}(d)_i = 0.$$

For position $i + 1$ where $d_{i+1} \neq 0$, the first (lower order) nonzero element in N, we *borrow* from the more significant digits and the sequence of numbers at the top row becomes $0\ (\beta - 1)\ (\beta - 1)\ \cdots (\beta - 1)\ (beta)$. The resulting digits are

$$\mathbf{RC}(d)_{i+1} = \beta - d_{i+1},$$

and for all elements d_j thereafter, $m \geq j \geq i + 2$,

$$\mathbf{RC}(d)_j = \beta - 1 - d_j.$$

Exercise 1.6

Exercise 1.7. With N represented by $d_m d_{m-1} \cdots d_1 d_0$ we have two possibilities.

If N is positive then $d_m = 0$ and the statement is true by the definition of binary number representation.

If N is negative then $d_m = 1$ and the absolute value of N is equal to $(2^n - N)$ where $n = m+1$.

$$\begin{aligned}
N &= -(2^{m+1} - \sum_{i=0}^{m} d_i 2^i) \\
&= \sum_{i=0}^{m-1} d_i 2^i + d_m 2^m - 2^{m+1} \\
&= \sum_{i=0}^{m-1} d_i 2^i + (d_m - 2)2^m. \text{ However, } d_m = 1 \text{ in this case. Thus,} \\
N &= \sum_{i=0}^{m-1} d_i 2^i + (-1)d_m 2^m.
\end{aligned}$$

Hence, the statement is true for N either positive or negative and this concludes the proof.

Exercise 1.7

Exercise 1.8. No, it is not since the representation $99 \cdots 99$ is equivalent to 0.

Exercise 1.8

Exercise 1.9. First, we calculate the nines' complement then add the 'end-around' carry.

$$\begin{array}{rcr}
250 & \Rightarrow & 250 \\
-245 & & +754 \\
\hline
 & & 1004 \\
 & & \Downarrow \\
 & & 004 \\
 & & +1 \\
\hline
 & & 5
\end{array}$$

Exercise 1.9

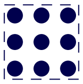
Exercise 1.10. The number of bits needed for the representation of the product is derived from analyzing the multiplication of the two largest representable unsigned operands:

$$P = (2^n - 1) \times (2^n - 1) = 2^{2n} - 2^{n+1} + 1 = 2^{2n-1} + \underbrace{2^{2n-1} - 2^{n+1}}_{\text{Positive number}} + 1.$$

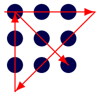
Thus, the largest product P_{max} is bounded such that $2^{2n} > P > 2^{2n-1}$. Hence, $2n$ bits are necessary and sufficient to represent it.

Exercise 1.10

Exercise 1.11. Most people when confronted with this exercise for the first time will over-restrict themselves by thinking within the "virtual box" drawn by the dots:



However, the requirements never mentioned anything related to the box. The whole idea is to actually go far and beyond the box as in:



In table 1.1, we saw several binary coding schemes giving completely different values to the same bit pattern. It was quite obvious that the way we 'look' at a pattern determines our interpretation of its value. The moral of the story in this exercise is to check your assumptions. How are you 'looking' and 'interpreting' the problem. Are you over-restricting yourself by implicitly forcing something that is not required in the system that you are studying? Keep this idea with you while reading the rest of the book! Exercise 1.11

Exercise 1.12(a) Since this is a weighted positional number system and the radix is $\beta = -1+j$ then the weight of position k is $\beta^k = (-1+j)^k = (\sqrt{2})^k(\frac{-1+j}{\sqrt{2}})^k$ which yields:

$$\cdots \quad -16+16j \quad 16 \quad -8-8j \quad 8j \quad 4-4j \quad -4 \quad 2+2j \quad -2j \quad -1+j \quad 1.$$

Based on these weights $0\,1011\,0011 \Rightarrow -8-11j$ and $1\,1101\,0001 \Rightarrow 5$.

↩

Exercise 1.12(b) From the weights we see that each four digits form a group where the imaginary part of the two most significant digits might cancel each other. For example, the least significant group is the positions with the weights: $\{2+2j, -2j, -1+j, 1\}$. To have a real number represented by this group the most significant two bits must be either both ones or both zeros and the second digit from the right must always be zero. This leads to the simple test:

> If $d_{4l+3} = d_{4l+2}$ and $d_{4l+1} = 0$ for $l = 0, 1, 2, \cdots$ then the number represented is a real number.

↩

Exercise 1.12(c) If we look at a group of four position as one digit D_l, we see that such a digit takes the following values to give a real number:

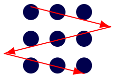
$$\begin{array}{ccc} 0000 & \Rightarrow & 0 \\ 0001 & \Rightarrow & 1 \\ 1100 & \Rightarrow & 2 \\ 1101 & \Rightarrow & 3 \end{array}$$

with each D_l having a weight equal to -4 times that of D_{l-1}. In fact, if we define $\alpha = -4$ and n as the number of 4 bits groups then the system at hand represents real numbers as $X = \sum_{l=0}^{l=n-1} D_l \alpha^l$. Since the values of D_l go from 0 to $|\alpha| - 1$ then this system is similar to the negabinary system presented in Table 1.2 and is capable of representing all integers.

For $n = 1$, i.e. a single group of four bits, the numbers represented are the values of $D_0 \in \{0, 1, 2, 3\}$. For $n = 2$, $(D_1 \times \alpha^1) \in \{0, -4, -8, -12\}$ and each of those values can be combined with any value for D_0 which results in $X \in \{-12, -11, -10, \cdots, 0, 1, 2, 3\}$. For $n = 3$, $(D_2 \times \alpha^2) \in \{0, 16, 32, 48\}$ and $X \in \{-12, -11, \cdots, 48\}$. This process can be continued and the system is able to represent any positive or negative number given enough bits.

↩

Exercise 1.13. Once more, the basic idea is to re-check your implicit assumptions. Did the problem state that these dots are infinitly small? Most people attempt to force the line to pass through the center of the dots assuming them to be single points. However, once we get rid of this false assumption we can easily solve it as:



In this section of 'going far and beyond', we are explicitly re-checking our assumptions about number representations and trying to see if there are better ways to deal with the various arithmetic operations. Exercise 1.13

Exercise 1.14. If a redundant system with base β has both positive and negative digits with $d_i = \beta$ or $d_i = -\beta$ as possible values then it has multiple representations of zero. For example, in a system with $-1 \leq d_i \leq \beta$, we represent zero as 00 or $\bar{1}\beta$.

If the system has digits at one side only of zero but still including zero, i.e. $\alpha \leq d_i \leq 0$ or $0 \leq d_i \leq \gamma$, then a unique representation of zero exist. An example of these systems is the carry save representation $(\beta = 2, d_i \in \{0, 1, 2\})$used in parallel multipliers. Exercise 1.14

Exercise 1.15. Once more, a person should check the implicit assumptions. The problem did not state the kind of pen you use to draw the line. People tend to think of a line in the mathematical sense where it has zero width. Real lines on paper are physical entities that exhibit some non-zero width. Times and again while dealing with arithmetic on computers we are reminded that we are not really working with a pure mathematical system but rather with a physical computer. With that said, just use a pen with a wide enough tip and you are able to cover all the dots in one line! In fact, a simple extension to this idea is to just place your thumb on the nine dots and realize: "I covered them all with only one 'dot', no lines at all!"

In section 1.5, we intentionally went 'far and beyond' and attempted to think out of the box in many dimensions to facilitate our job later when designing circuits. This final exercise is here to teach you that, in general, our instinct will lead us to the correct solution. However, we might be sometimes blinded by our own previous expositions to some subject to the point of not seeing a solution. We should then relax, go back to the basics, and if we cannot solve it, recheck our assumptions (our 'box'). Otherwise, ask a four year old child for help! Exercise 1.15

Exercise 2.1. Recalling that $-x_i$ and $m_i - x_i$ are congruent $\mathbf{mod}_{\mathbf{m_i}}$ we get the residue of $X^c = M - X$ at position i as

$$\begin{aligned}
(M - X)\mathbf{mod}_{\mathbf{m_i}} &= M\mathbf{mod}_{\mathbf{m_i}} + (-X)\mathbf{mod}_{\mathbf{m_i}} \\
&= 0 + (-x_i)\mathbf{mod}_{\mathbf{m_i}} \\
&= (m_i - x_i)\mathbf{mod}_{\mathbf{m_i}} \\
&= x_i^c.
\end{aligned}$$

Hence, $X^c = [x_i^c]$. Exercise 2.1

Exercise 2.2. The total range for the moduli 32, 31, 15 is

$$32 \times 31 \times 15 = 14\,880,$$

which means that the range of signed integers is

$$-7440 \leq x \leq 7439.$$

To perform the operation $(123 - 283 = -160)$, we must first convert each of the operands to this residue system.

$$\begin{aligned} 123 &= [27, 30, 3] \\ 283 &= [27, 4, 13]. \end{aligned}$$

Hence,

$$-283 = [5, 27, 2].$$

Then, we add the residues:

$$\begin{aligned} [27, 30, 3] + [5, 27, 2] &= [32, 57, 5] \\ &= [0, 26, 5]. \end{aligned}$$

As a check:

$$\begin{aligned} 160 &= [0, 5, 10] \\ -160 &= [0, 26, 5]. \end{aligned}$$

The results match, and the arithmetic is performed correctly. Exercise 2.2

Exercise 2.3. Without paying attention, one might think that the range of representable numbers in this system is equal to $4 \times 3 \times 2 = 24$. However, since 4 and 2 share a common factor, the range is only up to the least common multiple of the bases which is 12. Beyond that, the same representations repeat.

It is important to note that some combinations of residues never appear. It is impossible to have 1 or 3 as a residue for the modulus 4 (which means the number is odd) while having 0 as the residue for 2 (which means the number is even). Similarly, it is impossible to get 0 or 2 as a residue for 4 if we have 1 as the residue for 2. These ‘four’ impossible combinations when multiplied by the three possibilities for the residue of the base 3 yield the 12 representations that never occur.

The following table summaries the results.

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

Only the representations for the numbers from zero to eleven are unique. Exercise 2.3

Exercise 2.4. The equations derived so far still apply. However, there are two simpler cases of special interest:

$\beta = km_j$ results in $A\mathbf{mod_{m_j}} = a_0\mathbf{mod_{m_j}}$.

$m_j = \beta^k$ results in $A\mathbf{mod_{m_j}}$ being equal to the least significant k digits of A.

Exercise 2.4

Exercise 2.5. Since $X = \sum x_i\beta^i$ then

$$\begin{aligned} X\mathbf{mod}_{\beta-\mathbf{1}} &= \left(\sum x_i\beta^i\right)\mathbf{mod}_{\beta-\mathbf{1}} \\ &= \left(\sum (x_i\beta^i)\mathbf{mod}_{\beta-\mathbf{1}}\right)\mathbf{mod}_{\beta-\mathbf{1}} \\ &= \left(\sum (x_i\mathbf{mod}_{\beta-\mathbf{1}}\beta^i\mathbf{mod}_{\beta-\mathbf{1}})\mathbf{mod}_{\beta-\mathbf{1}}\right)\mathbf{mod}_{\beta-\mathbf{1}}. \end{aligned}$$

However, $\beta^i\mathbf{mod}_{\beta-\mathbf{1}} = (\beta\mathbf{mod}_{\beta-\mathbf{1}})^i = (1)^i = 1$. Hence,

$$X\mathbf{mod}_{\beta-\mathbf{1}} = \left(\sum (x_i\mathbf{mod}_{\beta-\mathbf{1}})\right)\mathbf{mod}_{\beta-\mathbf{1}}.$$

Exercise 2.5

Exercise 2.6. This algorithm is equivalent to a modulo 9 operation. Assume we are adding two decimal numbers A and B with digits a_i and b_i to form the sum S. First we show that this algorithm for adding up the digits of the numbers gives the same result as taking the $\mathbf{mod_9}$ of the numbers.

$$\begin{aligned} A\mathbf{mod_9} &= (\sum a_i10^i)\mathbf{mod_9} = [\sum (a_i10^i\mathbf{mod_9})]\mathbf{mod_9} \\ A\mathbf{mod_9} &= [\sum ((a_i\mathbf{mod_9})(10^i\mathbf{mod_9}))]\mathbf{mod_9} \\ A\mathbf{mod_9} &= [\sum ((a_i)(10\mathbf{mod_9})^i)]\mathbf{mod_9} = [\sum a_i]\mathbf{mod_9} \end{aligned}$$

Therefore, this algorithm gives the $\mathbf{mod_9}$ of each number and since:

$$(A\mathbf{mod}_\mu + B\mathbf{mod}_\mu)\mathbf{mod}_\mu \equiv S\mathbf{mod}_\mu,$$

this checksum algorithm will always work.

Historically, this algorithm has been also known as casting out 9's because whenever you get a 9 you can replace it by zero. It is possible to teach this checksum to young children learning to add multi-digit numbers to check their operations. Exercise 2.6

Exercise 2.7. Two functions are different if they have different values for at least one combination of the inputs. Since this is a d-valued system, each combination leads to d different possible functions. Hence the number of logic functions is $d^{\text{number of combinations}}$.

For r inputs we have d^r different possible combinations which yields d^{d^r} logic functions for the (r, d) system. Exercise 2.7

Exercise 4.1. Yes this algorithm will work. The variable `count` is a dummy variable and we can substitute it by `count=Y-Z` so that

1. initially `count =0` means `Z=Y`,
2. the condition `count<Y` becomes `Z>0`, and
3. the incrementing step `count=count+1` reduces to `Z=Z-1`.

This substitution transforms this new algorithm to algorithm 4.1 which proves that it is equivalent and will work correctly.

Despite their mathematical equivalence, the hardware implementations of both algorithms are different. In this proposed method the comparison is with Y which is an unknown variable at the design time. Hence, the comparison requires a bit by bit matching of Y and the current value of `count`. This matching is done by a row of *XOR* gates. Each gate has a zero output if the two corresponding bits match correctly. The outputs of that row of *XOR* gates are then fed to a tree to form their *NOR* function as in algorithm 4.1.

The comparison with a data-dependent quantity should, in general, be avoided. We will see this same recommendation when we discuss the division algorithms. Exercise 4.1

Exercise 4.2. The generation of a partial product in binary is quite simple. Let us assume that X and Y are represented by the bits strings $x_{n-1}\cdots x_1 x_0$ and $y_{n-1}\cdots y_1 y_0$ respectively.

The i^{th} partial product PP_i (corresponding to y_i) is given by $PP_i = (X)(y_i)$ and its j^{th} bit is equal to $(x_j)(y_i)$. The result of multiplying those two bits is

x_i	y_i	$(x_i)(y_i)$
0	0	0
0	1	0
1	0	0
1	1	1

which is equivalent to the operation of an *AND* gate. Hence, to generate the required PP, we use a row of two inputs *AND* gates where one of the inputs in all the gates is driven by y_i while the other input in each gate receives the corresponding bit of X. Exercise 4.2

Exercise 4.3. Yes. Algorithm 4.2 is still a loop and the loop condition must be checked. In this case, we initialize a register with the value of n, decrement it every cycle, and compare with zero to check the end of the loop. Fig. 4.1 is not a complete implementation. A designer must remember any hidden costs not explicitly presented in some diagrams when evaluating a new design.

Another "hidden" cost is the use of a shift register instead of a regular register with only a parallel loading capability. Such a shift register is slightly more complicated than the registers needed in algorithm 4.1.

Exercise 4.3

Exercise 4.4. In the two implementations, each cycle both the adder and the multilpexer are used and consume practically the same power. This power is wasted. A probably better way to reduce the power in the case of Fig. 4.2 is to really skip over the zeros by turning off the power supply to the adder. In such a case, extra power is consumed only when there is a 1 in the *LSB* of Y.

Exercise 4.4

Exercise 4.5. The regular cycles have a delay time equal to that of the sequence of the adder and the multiplexer while the faster cycles have the delay of the multiplexer only. Those fast cycles occur when there is a bit with a zero value in Y. The question is then how many bits, on average, are of zero value? This depends on the kind of data input to the multiplier. A designer should analyze the anticipated data and find the probability p_0 of a bit being zero. Then, if the time taken by the multiplexer is $\log_4 n$ and that by the adder is $\log_r n$, the total time delay is

$$t = \mathcal{O}\left(n \times (p_0 \log_4 n + (1 - p_0)(\log_r(n) + \log_4 n))\right).$$

A study of the anticipated workload of a computing system is often useful for a designer to tune the design for the best performance.

Exercise 4.5

Exercise 4.6. sol

Exercise 4.6

Exercise 5.1. In the double format the characteristic is still 7 bits as in the short format, hence $exp_{\mathbf{max}} = 63$ and $exp_{\mathbf{min}} = -64$. For the mantissa, $M_{\mathbf{max}} = 1 - 16^{-14}$ and $M_{\mathbf{min}} = 16^{-1}$. Thus, the largest representable number is

$$\mathbf{max} = 16^{63} \times (1 - 16^{-14}).$$

and the smallest positive normalized number is, as in the short format,

$$\mathbf{min} = 16^{-64}(16^{-1}).$$

Exercise 5.1

Exercise 5.2. To have the same or better accuracy in a larger base, the analysis of MRRE leads to $t_k - t_1 \geq k - 1$. For the case of $k = 4$ or $\beta = 16$ this relation becomes $t_{\beta=16} \geq t_{\beta=2} + 3$. Hence, three additional mantissa bits must exist in the hexadecimal format to achieve the same accuracy as the binary format.

Another way of looking at this result is to think that those additional 3 bits compensate for the leading zeros that might exist in the *MSD*. Obviously, if a hexadecimal base is used, we would use a full additional digit of 4 bits which leads to a higher accuracy in the hexadecimal format at the expense of a larger storage space. Exercise 5.2

Exercise 5.3. Taking the decimal system with five digits after the point, we can provide the following as an example of a total loss of significance: $(1.12345 \times 10^1 + 1.00000 \times 10^{10}) - 1.00000 \times 10^{10}$ which yields $(1.00000 \times 10^{10} - 1.00000 \times 10^{10}) = 0$ while the mathematicaly correct answer is 1.12345×10^1 as given by associating the second and third terms together first.

The heart rate of some readers might now be running a bit faster than usual since they thought of the question: *does this mean that the computers handling my bank account can lose all the digits representing my money?*

Well, fortunately enough there is another branch of science called numerical analysis. People there study the errors that might creep into calculations and try to give some assurance of correctness to the results. Floating point hardware can lend them some help as we will see when we discuss the directed rounding and interval arithmetic. Exercise 5.3

Exercise 5.4. Obviously, if the exponent difference is equal to zero a massive cancellation might occur as in $1.11111 \times 2^{20} - 1.11110 \times 2^{20} = 0.00001 \times 2^{20}$. The most significant digits may also cancel each other if the exponent difference is equal to one as in the case

$$\begin{aligned} 1.00000 \times 2^{20} - 1.11110 \times 2^{19} &= (1.00000 - 0.11111) \times 2^{20} \\ &= 0.00001 \times 2^{20} \\ &= 1.00000 \times 2^{15} \end{aligned}$$

where a binary system is used as an example. We prove here that for a non-redundant system with base β, if the exponent difference is two or larger, a massive cancellation is impossible. The minimum value of the significand of the larger number is $1.0000 \cdots 0$ while the maximum value of the significand of the smaller number (after alignment) is $0.0(\beta - 1)(\beta - 1) \cdots (\beta - 1)$. The difference is thus

$$\begin{array}{cccccccc} & 1. & 0 & 0 & 0 & \cdots & 0 \\ - & 0. & 0 & \beta - 1 & \beta - 1 & \cdots & \beta - 1 \\ \hline & 0. & \beta - 1 & 0 & 0 & \cdots & 1 \end{array}$$

which requires only a one digit left shift for normalization. Hence the necessary condition for massive cancellation is that the exponent difference is equal to zero or one.

According to this condition on the exponent, we are sure that the right shift for alignment is, at most, by one digit. Hence, a *single* guard digit is enough for the case of massive cancellation.

(Note that these results hold even for the case of subnormal numbers defined in the IEEE standard.)

Exercise 5.4

Exercise 5.5. When $a > 0$, $\bigtriangledown(a) = RZ(a)$ and $\bigtriangleup(a) = RA(a)$. On the other hand, when $a < 0$, $\bigtriangledown(a) = RA(a)$ and $\bigtriangleup(a) = RZ(a)$.

For systems where a single zero is defined all these rounding methods yield same result for $a = 0$ which is a. In systems where two signed zeros are defined (such as in the IEEE standard) the exact definition to round each of the two representation in the various rounding directions should be followed. Exercise 5.5

Exercise 5.6. We deduce from Fig. 5.1 that $\bigtriangleup(a) = - \bigtriangledown(-a)$. Exercise 5.6

Exercise 5.7. If $\bigtriangleup(a) = a$ (i.e. if it a is representable) then $\bigtriangleup(a) = \bigtriangledown(a)$, otherwise $\bigtriangleup(a) = \bigtriangledown(a) + 1\mathbf{ulp}$. Exercise 5.7

Exercise 5.8. If $RA(a) = a$ (i.e. if it a is representable) then $RA(a) = RZ(a)$, otherwise $|RA(a)| = |RZ(a) + 1\mathbf{ulp}|$. Exercise 5.8

Exercise 5.9. The standard defines the binary bias as $2^{w-1} - 1$. Hence, $exp_{\mathbf{max}} + \text{bias} = 2^w - 2$ which is a string of all ones except for the *LSB* which is 0, i.e. one less than the special value of all ones. Similarly, $exp_{\mathbf{min}} + \text{bias} = 1$ which is one more than the special value of all zeros. Exercise 5.9

Exercise 5.10. In the binary32 format, we have $\mathbf{min} = 2^{-126} \times 1.0$. Using the definition of the standard, the largest subnormal number is

$$\begin{aligned} 2^{-126} \times 0.11\cdots1 &= 2^{-126} \times (1.0 - 0.00\cdots1) \\ &= \mathbf{min} - 2^{-126} \times 0.00\cdots1 \end{aligned}$$

which means that there is a continuation between the normal and subnormal numbers. If we use 2^{-127} as the scaling factor for the subnormal numbers a gap will exist. The case of the other formats is similar. Exercise 5.10

Exercise 5.11. As indicated in the text, both the IEEE standard and the PDP-11 have reserved exponents. In contrast, the IBM system does not reserve exponents for a special use. We only show the case of normalized numbers in the IEEE standard. For the single precision we get:

	IEEE	PDP-11	S/370
Radix	2	2	16
Significand	(1)+23 bits	(1)+23 bits	6 digits
Minimum Significand	1	0.5	1/16
Maximum Significand	≈ 2	≈ 1	≈ 1
Exponent	8 bits	8 bits	7 bits
Bias	127	128	64
Minimum exponent	-126	-127	-64
Maximum exponent	127	126	63
Minimum number	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-128} \approx 3 \times 10^{-39}$	$16^{-65} \approx 5.4 \times 10^{-79}$
Maximum number	$2^{128} \approx 3.4 \times 10^{38}$	$2^{126} \approx 8.5 \times 10^{37}$	$16^{63} \approx 7.2 \times 10^{75}$

For the double precision, the results are:

	IEEE	PDP-11	S/370
Radix	2	2	16
Significand	(1)+52 bits	(1)+55 bits	14 digits
Minimum Significand	1	0.5	1/16
Maximum Significand	≈ 2	≈ 1	≈ 1
Exponent	11 bits	8 bits	7 bits
Bias	1023	128	64
Minimum exponent	-1022	-127	-64
Maximum exponent	1023	126	63
Minimum number	$2^{-1022} \approx 2.2 \times 10^{-308}$	$2^{-128} \approx 3 \times 10^{-39}$	$16^{-65} \approx 5.4 \times 10^{-79}$
Maximum number	$2^{1024} \approx 1.8 \times 10^{308}$	$2^{126} \approx 8.5 \times 10^{37}$	$16^{63} \approx 7.2 \times 10^{75}$

It is clear that the extra bits in the PDP-11 and S/370 serve to improve the precision but not the range of the system.

Exercise 5.11

Exercise 5.12. The exact representation is the infinite binary sequence

$$(0.0001100110011001100\ldots)_{\beta=2}.$$

The binary32 is normalized with a hidden one, hence the sequence becomes

$$(1).1001100110011001100110011001100\ldots \times 2^{-4}$$

when normalized. The significand has 23 bits whatever lies beyond that in the infinite sequence must be rounded. The rounded part is larger than half of the unit in the least significant place of the part remaining so we add one to the unit of the last place to get:

$$\begin{array}{l} (1).10011001100110011001100 \,|\underbrace{1100\ldots}_{\text{rounded}} \times 2^{-4} \\ \quad\Downarrow \\ (1).10011001100110011001101 \ \ \times 2^{-4}. \end{array}$$

The biased exponent is $127 - 4 = 123$ and the representation is thus:

$$0\ 01111011\ 10011001100110011001101$$

which is slightly more than 0.1. (It is about 0.1000000015.)

Similarly, binary64 yields

$$0\ 01111111011\ 1001100110011\ldots00110011010$$

a quantity also larger than 0.1.

Exercise 5.12

Exercise 5.13. In binary64 and round to nearest even, 0.3 is represented as

$$0\ 01111111101\ 0011001100110011\ldots001100110011$$

which is slightly less than 0.3 due to rounding. The multiplication of $x \approx 0.1$ (represented as $0\ 01111111011\ 1001100110011\ldots00110011010 = (1).1001100110011\ldots00110011010 \times 2^{-4}$) by

3 yields $100.11001100110011\ldots0011001110 \times 2^{-4} \Rightarrow (1).00110011\ldots001100110011 \mid 10 \times 2^{-2}$ which is rounded to $(1).00110011\ldots001100110100 \times 2^{-2}$ and represented by

$$0\ 01111111101\ 0011001100110011\ldots001100110100$$

a quantity larger than 0.3. Now $3x - y$ gives $0.000\ldots001 \times 2^{-2}$ which is normalized to $(1).000\ldots00 \times 2^{-54} \approx 5.551 \times 10^{-17}$.

On the other hand, $2x$ yields

$$0\ 01111111100\ 1001100110011\ldots00110011010$$

with just a change in the exponent from the representation of x. Then $2x - y$ gives

$$\begin{array}{rl} & (1).100110011001100\ldots1100110011010\times2^{-3} \\ - & (1).0011001100110011\ldots001100110011\times2^{-2} \end{array}$$

which becomes

$$\begin{array}{rl} & (0).1100110011001100\ldots110011001101 \mid 0\times2^{-2} \\ - & (1).0011001100110011\ldots001100110011 \quad\ \ \times2^{-2} \end{array}$$

after the alignment yielding a result of

$$-\quad (0).01100110011001100\ldots11001100110 \mid 0\times2^{-2}$$

When normalized and rounded $2x - y$ is

$$-\quad (1).100110011001100\ldots1100110011000\times2^{-4}.$$

Hence, $2x - y + x$ gives

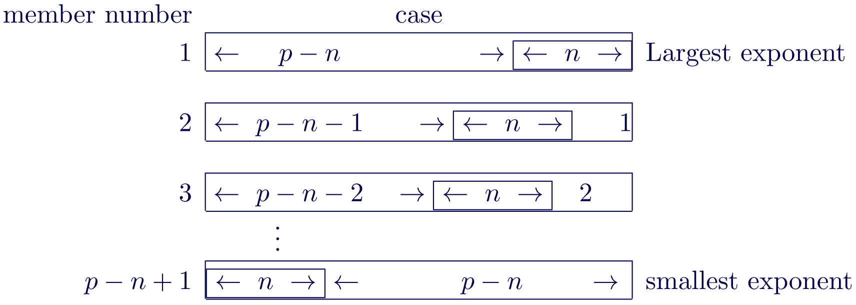
$$\begin{array}{rl} - & (1).100110011001100\ldots1100110011000\times2^{-4} \\ + & (1).100110011001100\ldots1100110011010\times2^{-4} \\ \hline + & (0).000000000000000\ldots0000000000010\times2^{-4} \end{array}$$

which is equal to $2^{-55} \approx 2.776 \times 10^{-17}$. Now,

$$\left.\frac{3x-y}{2x-y+x}\right|_{(x=0.1,y=0.3)} = \frac{2^{-54}}{2^{-55}} = 2\ !$$

Exercise 5.13

Exercise 5.14. The member of the cohort with the largest exponent is the one where the least significant of the n digits coincides with the right most (least significant) of the p digits. A left shift by one position of the n digits leads to an increment of the exponent by one. This can be continued till the *MSD* of the n digits is at the *MSD* of the p digits.



So the total number of members in the cohort is $p - n + 1$. Exercise 5.14

Exercise 5.15. As we have seen in example 5.16, in a normalized floating point hardware system with a fixed width datapath, the digits that are shifted are eventually droped or used for rounding. Such a choice is acceptable since these are the digits of lowest significance in the smaller number. On the other hand, if we shift the larger number to the left and drop the shifted digits, those digits would be the most significant ones which is not acceptable.

In some specific cases, a designer might opt to increase the datapath width to allow for a left shift without loosing the *MSD*s. However, the common practice is to use the right shift for operands alignement. Exercise 5.15

Exercise 5.16. If the significand of the larger operand is m_l while that of the smaller operand is m_s then $1 \leq m_l < \beta$ and $1 \leq m_s < \beta$ because they are both originally normalized according to our assumptions so far.

After alignment we get $0 \leq m_{s_{align}} < \beta$ which yields

$$1 \leq m_l + m_{s_{align}} < 2\beta.$$

If the result is in the range $1 \leq m_l + m_{s_{align}} < \beta$, there is no need to shift. However, if $\beta \leq m_l + m_{s_{align}} < 2\beta$ we must shift the result to the right by one digit to normalize it. Such a shift is equivalent to dividing the result by β to reach the new range $[1, 2($. This range is representable as a normalized number even when $\beta = 2$. In the case of a right shift, the exponent is increased by 1. If this results in exponent spill, i.e. the exponent reaching its maximum value, the postnormalization sets the number to its largest possible value or to ∞ according to the rounding direction currently used. We will explain the exponent spill shortly.

If the system does not require normalized numbers we start with $0 \leq m_l < \beta$ and $0 \leq m_s < \beta$ then follow the same steps to reach the same conclusion. Exercise 5.16

Exercise 5.17. In the addition and subtraction operations, we only care about the difference between the values of the two exponents. That difference is the same whether we use the true exponents or the biased exponents. The result in the case of addition or subtraction then has the exponent of the larger operand. That exponent value might change due to normalization but that change can be done on the biased exponent without causing trouble. Exercise 5.17

Exercise 5.18. Let us denote the two (true) exponents by exp_1 and exp_2 and denote the exponent of the **max** by $exp_{\mathbf{max}}$ and that of **min** by $exp_{\mathbf{min}}$. Since

$$\begin{array}{rcl} exp_{\mathbf{min}} \leq & exp_1 & \leq exp_{\mathbf{max}} \\ exp_{\mathbf{min}} \leq & exp_2 & \leq exp_{\mathbf{max}} \end{array}$$

then the exponent of the result lies in the range

$$2 \times exp_{\mathbf{min}} \leq exp_1 + exp_2 \leq 2 \times exp_{\mathbf{max}}.$$

Since exp_{min} is negative,

$$2 \times exp_{min} < exp_{min}$$

which indicates that an underflow is possible under these conditions. Obviously, an overflow is also possible.

Note that in a format with subnormals such as the IEEE binary32 format, an underflow in mutliplication occurs *always* when both inputs are subnormal numbers. An underflow might happen even if only one operand is subnormal, can you see why? Exercise 5.18

Exercise 5.19. We should shift as many significant digits as possible from those below the $\beta^{-(p-1)}$ position. The words "significant" and "as possible" are the important keywords.

Obviously, there is a non-ending sequence of non-significant zeros to the right of the number. We do not need to shift those. However, if the zero digit is significant (as a result from multiplication of 2 by 5 in base 10 for example) then we should take it into account and shift it in. This explains "significant".

For the "as possible", we should not shift to the point of losing the *MSD* of the result. So the shifting stops when the most significant non-zero digit reaches the β^0 position. Another point on the "as possible" side is the exponent range. Since the exponent is decremented by one for each left shift of one position, we should stop if the exponent reaches the underflow limit. Exercise 5.19

Exercise 5.20. We have

$$\begin{aligned} exp_{\mathbf{min}} \le \quad & exp_1 \quad \le exp_{\mathbf{max}} \\ exp_{\mathbf{min}} \le \quad & exp_2 \quad \le exp_{\mathbf{max}} \end{aligned}$$

then the exponent of the result lies in the range

$$exp_{\mathbf{min}} - exp_{\mathbf{max}} \le exp_1 - exp_2 \le exp_{\mathbf{max}} - exp_{\mathbf{min}}$$

which indicates that an underflow is possible.

Because $exp_{\mathbf{min}} < 0$, the upper bound is larger than $exp_{\mathbf{max}}$ and an overflow may occur. A special case in the division operation is the division by zero which is usually treated as an overflow leading to a result of **max** or ∞.

Note that in a format with subnormals such as the IEEE binary32 format, an underflow in division might occur more frequently. Exercise 5.20

Exercise 5.21. Obviously yes. If we subtract the two biased exponents then the bias of the first cancels that of the second. To get a correct biased exponent for the result we add the bias back. Exercise 5.21

Exercise 5.22. For this simple RNE implementation, whether we add A to G or L we get the same logic value. This is not always the case as we will see later. This exercise is here just to remind you to look "far and beyond" as we learned in section 1.5. Some simple changes can have big effects sometimes! Exercise 5.22

Exercise 5.23. In the following, the numbers (a, b, c, d) themselves are rounded first according to the statement of the problem. Then, the operations are rounded to give us the maximum and minimum value of the result.

Case 1: If c and d are of the same sign their product is subtracted from s. For s_{max} we want to reduce their magnitude, and for s_{min} we want to increase their magnitude. Hence, for s_{max} we use $\bigtriangledown(RZ(c) \times RZ(d))$. This helps us to combine the case for both numbers being positive or both being negative. If we insist on using $\bigtriangleup$ and $\bigtriangledown$ only then we must split this formula into $\bigtriangledown(\bigtriangleup c \times \bigtriangleup d)$ for c and d negative and $\bigtriangledown(\bigtriangledown c \times \bigtriangledown d)$ for c and d positive. Notice that to *decrease* the magnitude of a *negative*number we use $\bigtriangleup$. Similar arguments hold for s_{min} and for the other cases below.

a and b of the same sign: Then

$$\begin{aligned} s_{max} = &\quad \bigtriangleup(\bigtriangleup(RA(a) \times RA(b)) - \bigtriangledown(RZ(c) \times RZ(d))\,) \\ s_{min} = &\quad \bigtriangledown(\bigtriangledown(RZ(a) \times RZ(b)) - \bigtriangleup(RA(c) \times RA(d))\,) \end{aligned}$$

a and b of opposite signs: Then

$$\begin{aligned} s_{max} = &\quad \bigtriangleup(\bigtriangleup(RZ(a) \times RZ(b)) - \bigtriangledown(RZ(c) \times RZ(d))\,) \\ s_{min} = &\quad \bigtriangledown(\bigtriangledown(RA(a) \times RA(b)) - \bigtriangleup(RA(c) \times RA(d))\,) \end{aligned}$$

Case 2: If c and d are of opposite signs their product adds to s. For s_{max} we want to increase their magnitude, and for s_{min} we want to decrease their magnitude.

a and b of the same sign: Then

$$\begin{aligned} s_{max} = &\quad \bigtriangleup(\bigtriangleup(RA(a) \times RA(b)) - \bigtriangledown(RA(c) \times RA(d))\,) \\ s_{min} = &\quad \bigtriangledown(\bigtriangledown(RZ(a) \times RZ(b)) - \bigtriangleup(RZ(c) \times RZ(d))\,) \end{aligned}$$

a and b of opposite signs: Then

$$\begin{aligned} s_{max} = &\quad \bigtriangleup(\bigtriangleup(RZ(a) \times RZ(b)) - \bigtriangledown(RA(c) \times RA(d))\,) \\ s_{min} = &\quad \bigtriangledown(\bigtriangledown(RA(a) \times RA(b)) - \bigtriangleup(RZ(c) \times RZ(d))\,) \end{aligned}$$

Since the IEEE standard does not provide the 'RA' rounding, a compliant implementation without 'RA' will even do more calculations!

It is clear from this exercise how the software dealing with floating point numbers and aiming to provide a good interval analysis must switch the rounding mode quite often and recalculate (in fact, just re-round) some results. Exercise 5.23

Exercise 5.24. Since the fractional value can be either positive or negative, the value added for rounding may be positive, negative or zero. Compared to conventional IEEE rounding logic, more complicated situations arise in some of the rounding cases for this redundant digit design. For example, the rounding to zero (RZ) mode of the IEEE is not just a simple truncation but a -1 is added to the number at the rounding location if the fractional value is negative. The decision is according to the following table where the given value is added to L the bit at the rounding location.

range	*RNE*	*RZ*	*RP*		*RM*	
			$+ve$	$-ve$	$+ve$	$-ve$
$-1 < f < -0.5$	-1	-1	0	-1	-1	0
-0.5	$-\lvert L\rvert$	-1	0	-1	-1	0
$-0.5 < f < 0$	0	-1	0	-1	-1	0
0	0	0	0	0	0	0
$0 < f < 0.5$	0	0	1	0	0	1
0.5	$\lvert L\rvert$	0	1	0	0	1
$0.5 < f < 1$	1	0	1	0	0	1

Two points in this table challenge our previous assumptions about rounding.

1. RZ is not always a truncation.
2. We may sometimes subtract instead of add to get the rounded result.

Exercise 5.24

Exercise 5.25. Since the signaling NaN indicates uninitialized FP number, it is better if the signaling NaN is the representation used in the booting status of the memory (a value of all ones). Otherwise, at the declaration of each variable in a high level language the compiler must insert a few instructions to modify one position in the bit string.

Hence, for such a system, the definition where a 1 in the significand's *MSB* indicates a signaling NaN and a 0 indicates a quiet NaN is preferred. Exercise 5.25

Exercise 5.26. The result for $(+\infty)/(-0)$ is $-\infty$. Note that the division by zero is signaled only when the dividend is a *finite* non-zero number. As mentioned earlier the arithmetic on infinities is considered exact and does not signal the inexact nor the overflow exceptions. Hence, we get the result of $-\infty$ with no exceptions at all!

As for $\sqrt{-0}$, according to the standard and the explanation provided in section 5.6.2 (point 6 of the invalid exception), $\sqrt{-0} = -0$. This is somewhat an arbitrary decision. It could have been defined as equal to $+0$. Again, no exceptions raised at all!

This exercise shows us that it is important to really read the fine details of standards and stick to them. A complying system must correctly implement even these rare occurances.

An alert reader might feel that the lesson here contradicts the spirit of going 'far and beyond' that we got from exercise 1.11 and its follow-ups. This is not true. It is really important to think freely at the early stages of design. However, once the design team agrees on some decisions, everyone should implement them. Otherwise, chaos occurs when the different parts of the design are grouped.

The maxim is *"An early standardization stifles innovation and a late standardization leads to chaos"*.

This does not, however, prevent a designer from innovating within the premises of the standard. We will discuss a number of such innovations in the subsequent chapters. Exercise 5.26

Exercise 5.27. These expressions are given in the C-language standard as guidelines for compiler writers on what can be optimized without problem and what cannot.

1. $x/2 = x * 0.5$

   Although similar transformations involving inexact constants generally do not yield numerically equivalent expressions, if the constants are exact then such transformations can be made on IEEE 754 compliant machines and others that round perfectly.

2. $1 * x = x/1 = x$

   The expressions 1 * x, x / 1, and x are equivalent for the same reasons as above.

3. $x/x \neq 1.0$

   The expressions x/x and 1.0 are not equivalent if x can be zero, infinite, or NaN.

4. $x - y = x + (-y) = (-y) + x$

   The expressions $x - y$, $x + (-y)$, and $(-y) + x$ are equivalent since the subtraction is treated as a negation followed by addition and those operations are still commutative.

   Note that commutativity is valid for two operands only. If three terms should be summed then commutativity is not always valid for example $1.0 \times 10^{20} - 1.0 \times 10^{20} + 1.0 \times 10^{2} \neq 1.0 \times 10^{20} + 1.0 \times 10^{2} - 1.0 \times 10^{20}$.

5. $x - y \neq -(y - x)$

   The expressions $x - y$ and $-(y - x)$ are not equivalent because $1 - 1$ is $+0$ but $-(1 - 1)$ is -0 (in the default rounding direction).

6. $x - x \neq 0$

   The expressions $x - x$ and 0.0 are not equivalent if x is a NaN or infinite.

7. $0 * x \neq 0$

   The expressions $0 * x$ and 0.0 are not equivalent if x is a NaN, infinite, or -0.

8. $x + 0 \neq x$

   The expressions $x + 0$ and x are not equivalent if x is -0, because $(-0) + (+0)$ yields $+0$ (in the default rounding direction), not -0.

9. $x - 0 \neq x$

   $(+0) - (+0)$ yields -0 when rounding is downward (toward $-\infty$), but $+0$ otherwise, and $(-0) - (+0)$ always yields -0. Hence, the expression is not always valid under all rounding directions and the compiler should not make this optimization unless the rounding direction is guaranteed not to be towards minus infinity.

10. $-x \neq 0 - x$

    The expressions $-x$ and $0 - x$ are not equivalent if x is $+0$, because $-(+0)$ yields -0, but $0 - (+0)$ yields $+0$ (unless rounding is towards minus infinity).

Exercise 5.27

Exercise 5.28. It is important to remember that, according to the standard, the result of a comparison involving a NaN is always false (even if it is compared to check if it is equal to itself). Hence, in the original code, if one of the inputs is a NaN, the result will become NaN and will be used for further calculations in the program which might result in the "rocket failing".

The second piece of code uses what is sometimes called "defensive coding" where the programmer checks and isolates the problematic cases. Here, the manager changed the code so that if one of the inputs is a NaN the first two lines will not be executed and the third line will take the appropriate action.

Another problem still remains in this second piece of code: the cases of one of the inputs or both being equal to $\pm\infty$. If only one input is infinite then the conditions in the program will cause the value of r to become $+\infty$. Note that comparisons involving infinities assume that $+\infty$ is larger than any finite number and that $-\infty$ is smaller than any finite number. A result $r = +\infty$ could cause the rocket to fail hence this is problematic. Furthermore, if both inputs are infinite we end up with $r = \text{NaN}$ whether the two infinities are of different signs or the same sign. This NaN result will again cause problems for our rocket. Hence, a better code may be to check for the NaN and infinity before attempting the arithmetic part. The standard provides the predicates `isNaN()` and `isInfinite()` to perform such checks. These predicates return a boolean value equivalent to TRUE if the floating point variable is of the class specified by the predicate. So, the code becomes:

```
if (isNaN(a) or isNaN(b) or isInfinite(a) or isInfinite(b))
     <do something NOW before the rocket fails>
else if a < b then r = sqrt(b^ 2 - a^ 2)
else if a >= b then r = sqrt(a^ 2 - b^ 2)
```

Alas, this third code is still vulnerable to errors. If one of the inputs or both are of finite values but the square is larger than the maximum then an overflow will occur and the value of the square may be recorded as infinity. A value $r = \sqrt{+\infty - \text{finite}} = +\infty$ results if the square of the larger input only becomes infinity. A value $r = \text{NaN}$ results if the squares of both inputs become infinities or the square of the smaller input only becomes infinity. This latter case is possible if for example $a = -\text{max}$ and $b = 1$ which leads to $r = \sqrt{1 - \infty} = \text{NaN}$. An even better code is:

```
if (isNaN(a) or isNaN(b) or isInfinite(a) or isInfinite(b))
     <do something NOW before the rocket fails>
else if a < b then r = abs(b)*sqrt(1 - (a/b)^ 2)
else if a >= b then r = abs(a)*sqrt(1 - (b/a)^ 2)
```

where the `abs()` function gives the absolute value. This fourth piece of code will not cause an overflow if the square of the larger number overflows. However, it does not solve the cases similar to $a = -\text{max}$ and $b = 1$. The following code which compares the absolute values correctly handles these cases:

```
if (isNaN(a) or isNaN(b) or isInfinite(a) or isInfinite(b))
     <do something NOW before the rocket fails>
else if abs(a) < abs(b) then r = abs(b)*sqrt(1 - (a/b)^ 2)
else if abs(a) >= abs(b) and abs(a) not=0 then r = abs(a)*sqrt(1 - (b/a)^ 2)
else r = 0
```

This fifth piece of code also correctly handles the case of zero inputs in order not to divide by zero. A slight optimization gives us the sixth piece of code:

```
if (isNaN(a) or isNaN(b) or isInfinite(a) or isInfinite(b))
     <do something NOW before the rocket fails>
else if abs(a) < abs(b) then r = abs(b)*sqrt(1 - (a/b)^ 2)
else if abs(a) not=0 then r = abs(a)*sqrt(1 - (b/a)^ 2)
else r = 0
```

Can you find further problems in this last piece of code? Hopefully, the enhancement of this code teaches you to program safely while using floating point numbers. Exercise 5.28

Exercise 5.29. The results are

1. -1000000488819424102023407484648713E-33, inexact
2. -7980305000000000000000000000000000E-2624, no flags raised
3. -57E0, no flags raised
4. -0E-6176, inexact and underflow
5. -16500000000000000000E-566, no flags raised
6. +9999999999999999999999999999999999E6111, inexact and overflow
7. -127218580909000601093933070E6111, no flags raised
8. +1E-6176, inexact and underflow
9. +3E-3070, no flags raised
10. +526E-3088, no flags raised

Exercise 5.29

Exercise 5.30. The main reason that caused the appearance of a denormalized number out of normalized ones is the alignment shift to equate the exponents before the subtraction.

For the subnormal range, all the numbers have the same exponent and no alignment shift is required. Hence, the result is always exact which is either another denormalized number or zero. Exercise 5.30

Bibliography

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.

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

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

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

Richard P. Brent. On the precision attainable with various floating-point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.

Wilfried Buchholz. Fingers or fists? (the choice of decimal or binary representation). *Communications of the ACM*, 2(12):3–11, December 1959.

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

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

William J. Cody, JR. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.

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.

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

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

M. Cornea, C. Anderson, J. Harrision, P. Tang, E. Schneider, and C. Tsen. A software implementation of the IEEE 754r decimal floating-point arithmetic using the binary encoding. In *Proceedings of the IEEE International Symposium on Computer Arithmetic, Montpellier, France*, June 2007.

Michael F. Cowlishaw. Densely packed decimal encoding. *IEE Proceedings. Computers and Digital Techniques*, 149(3):102–104, 2002a. ISSN 1350-2387.

Michael F. Cowlishaw. The 'telco' benchmark. World-Wide Web document., 2002b. URL http://speleotrove.com/decimal/telco.html.

Michael F. Cowlishaw. Decimal floating-point: algorism for computers. In Jean Claude Bajard and Michael Schulte, editors, *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: proceedings: Santiago de Compostela, Spain, June 15–18, 2003*, pages 104–111, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2003. IEEE Computer Society Press. ISBN 0-7695-1894-X. URL http://www.acsel-lab.com/arithmetic/arith16/papers/ARITH16_Cowlishaw.pdf;http://www.dec.usc.es/arith16/papers/paper-107.pdf. IEEE Computer Society order number PR01894. Selected papers republished in *IEEE Transactions on Computers*, **54**(3) (2005) Schulte and Bajard [2005].

Mike Cowlishaw. *The decNumber C library*. IBM Corporation, April 2007. URL http://download.icu-project.org/ex/files/decNumber/decNumber-icu-340.zip. Version 3.40.

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

Mark A. Erle, Michael J. Schulte, and Brian J. Hickmann. Decimal floating-point multiplication via carry-save addition. In *Proceedings of the IEEE International Symposium on Computer Arithmetic, Montpellier, France*, pages 46–55, June 2007.

European Commission. *The Introduction of the Euro and the Rounding of Currency Amounts*. European Commission Directorate General II Economic and Financial Affairs, Brussels, Belgium, 1997. URL http://www.ciceuta.es/euro/doc/1/eup22en.pdf.

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

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

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

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

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

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

Harvey L. Garner. A survey of some recent contributions to computer arithmetic. *IEEE Transactions on Computers*, C-25(12):1277–1282, December 1976.

R. C. Ghest. *A Two's Complement Digital Multiplier, the Am25S05*. Advanced Micro Devices, Sunnyvale, CA, 1971.

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

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

B. Hasitume. Floating point arithmetic. *Byte*, November 1977.

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

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

Muhammad ibn Musa Al-Khawarizmi. *The keys of Knowledge*, مَفَاتِيحُ العُلُومِ لِمُحَمَّدٍ بنِ مُوسَى الخَوَارِزمِيّ. circa 830 C.E.

IEEE Std 754-1985. IEEE standard for binary floating-point arithmetic, August 1985. (ANSI/IEEE Std 754-1985).

IEEE Std 754-2008. IEEE standard for floating-point arithmetic, August 2008. (IEEE Std 754-2008).

IEEE Std 854-1987. IEEE standard for radix-independent floating-point arithmetic, October 1987. (ANSI/IEEE Std 854-1987).

William Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic, May 1996. URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`.

Rolf Landauer. Minimal energy requirements in communication. *Science*, 272:1914–1918, June 1996.

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

John D. Marasa and David W. Matula. A simulative study of correlated error propagation in various finite-precision arithmetics. *IEEE Transactions on Computers*, C-22(6):587–597, June 1973.

Grant W. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay.* PhD thesis, Stanford University, June 1997.

W. M. McKeeman. Representation error for real numbers in binary computer arithmetic. *IEEE Transactions on Electronic Computers*, pages 682–683, October 1967.

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

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

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

Hooman Nikmehr, Braden Phillips, and Cheng-Chew Lim. Fast decimal floating-point division. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):951–961, September 2006.

Roelof Maarten Marie Oberman. *Digital circuits for binary arithmetic.* The MacMillan Press LTD, London, 1979. ISBN 0-333-25535-6.

Behrooz Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, January 1990. URL http://www.ece.ucsb.edu/Faculty/Parhami/publications.htm.

Behrooz Parhami. *Computer Arithmetic Algorithms And Hardware Designs.* Oxford University Press, New York, 2000. URL http://www.oup-usa.org.

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

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

S. D. Pezaris. A 40ns 17-bit by 17-bit array multiplier. *IEEE Transactions on Computers*, April 1971.

Ramy Raafat, Amira M. Abdel-Majeed, Rodina Samy, Tarek ElDeeb, Yasmin Farouk, Mostafa Elkhouly, and Hossam A. H. Fahmy. A decimal fully parallel and pipelined floating point multiplier. In *Forty-Second Asilomar Conference on Signals, Systems, and Computers, Asilomar, California, USA*, October 2008.

Thammavarapu R. N. Rao. *Error coding for arithmetic processors.* Academic Press, Inc., 111 Fifth Avenue, New York, New York 10003, USA, 1974. ISBN 0-12-580750-3.

R. K. Richards. *Arithmetic Operations in Digital Computers.* D. Van Nostrand, New York, NY, USA, 1955.

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

Michael J. Schulte and Jean-Claude Bajard. Guest Editors' introduction: Special issue on computer arithmetic. *IEEE Transactions on Computers*, 54(3):241–242, March 2005. ISSN 0018-9340. URL http://csdl.computer.org/comp/trans/tc/2005/03/t0241.pdf;http://csdl.computer.org/dl/trans/tc/2005/03/t0241.htm.

Eric M. Schwarz, J. S. Kapernick, and Michael F. Cowlishaw. Decimal floating-point support on the IBM system z10 processor. *IBM Journal of Research and Development*, 53(1), 2009.

Norman Ross Scott. *Computer number systems and arithmetic.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, USA, 1985. ISBN 0-13-164211-1.

Charles Severance. IEEE 754: An interview with William Kahan. *IEEE Computer magazine*, 31(3):114–115, March 1998.

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

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

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

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

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

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

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

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

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

Ivan E. Sutherland and Robert F. Sproull. Logical effort: Designing for speed on the back of an envelope. In *Proceedings of the 1991 Advanced Research in VLSI, University of California, Santa Cruz*, pages 1–16, 1991.

Ivan E. Sutherland, Robert F. Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, 1999.

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

Texas Instruments, Inc. TTL Data Book, 1976.

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

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

Alvaro Vazquez, Elisardo Antelo, and Paolo Montuschi. A new family of high-performance parallel decimal multipliers. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic, June 25–27, 2007, Montpellier, France*, pages 195–204, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2007. IEEE Computer Society Press. ISBN 0-7695-2854-6. doi: http://dx.doi.org/10.1109/ARITH.2007.6. URL `http://www.lirmm.fr/arith18/papers/vazquez-DecimalMultiplier.pdf`.

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

L.-K. Wang and M. J. Schulte. Decimal floating-point square root using Newton–Raphson iteration. In Stamatis Vassiliadis, Nikitas J. Dimopoulos, and Sanjay Vishnu Rajopadhye, editors, *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors: ASAP 2005: 23–25 July 2005, Samos, Greece*, pages 309–315, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2005. IEEE Computer Society Press. ISBN 0-7695-2407-9. URL `http://mesa.ece.wisc.edu/publications/cp_2005-05.pdf`.

Liang-Kai Wang and Michael J. Schulte. A decimal floating-point divider using Newton–Raphson iteration. *Journal of VLSI Signal Processing*, 49(1):3–18, October 2007. ISSN 0922-5773. doi: http://dx.doi.org/10.1007/s11265-007-0058-5.

Liang-Kai Wang, Charles Tsen, Michael J. Schulte, and Divya Jhalani. Benchmarks and performance analysis of decimal floating-point applications. *IEEE*, pages 164–170, 2007.

Liang-Kai Wang, Michael J. Schulte, John D. Thompson, and Nandini Jairam. Hardware designs for decimal floating-point addition and related operations. *IEEE Transactions on Computers*, 58(3):322–335, March 2009.

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.

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

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

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

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

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

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

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

Index