Number Representation and Computer Arithmetic

Article to appear in *Encyclopedia of Information Systems*, Academic Press, 2001.

*********** The following outline is not part of the article *********** # pages

****************** end of outline ******************** Total pages 47

Arithmetic is a branch of mathematics that deals with numbers and numerical computation. Arithmetic operations on pairs of numbers x and y include addition, producing the sum $s = x + y$, subtraction, yielding the difference $d = x - y$, multiplication, resulting in the product $p = x \times y$, and division, generating the quotient $q = x / y$ (and, in the case of integer division, the remainder $z = x \bmod y$). Subtraction and division can be viewed as operations that undo the effects of addition and multiplication, respectively. Computer arithmetic is a branch of computer engineering that deals with methods of representing integers and real values (e.g., fixed- and floating-point numbers) in digital systems and efficient algorithms for manipulating such numbers by means of hardware circuits or software routines. On the hardware side, various types of adders, subtractors, multipliers, dividers, square-rooters, and circuit techniques for function evaluation are considered. Both abstract structures and technology-specific designs are dealt with. Software aspects of computer arithmetic include complexity, error characteristics, stability, and certifiability of computational algorithms.

I Natural Numbers

When we think of numbers, it is usually the *natural numbers* that first come to our mind; the type of numbers that sequence book or calendar pages, mark clock dials, flash on stadium scoreboards, and guide deliveries to our houses. The set {0, 1, 2, 3, . . .} of natural numbers, also known as *whole numbers* or *unsigned integers*, forms the basis of arithmetic. We use natural numbers for counting and for answering questions that ask "how many?".

Four-thousand years ago, Babylonians knew about natural numbers and were proficient in arithmetic. Since then, representations of natural numbers have advanced in parallel with the evolution of language. Ancient civilizations used sticks and pebbles to record inventories or accounts. When the need for larger numbers arose, the idea of grouping sticks or pebbles simplified counting and comparisons; for example, 27 was represented by five groups of five sticks, plus two sticks. Eventually, objects of different shapes or colors were used to denote such groups, leading to more compact representations.

Numbers must be differentiated from their representations, sometimes called *numerals*. For example, the number "twenty-seven" can be represented in different ways using various numerals or *numeration systems*; these include:

||||| ||||| ||||| ||||| ||||| ||	sticks or *unary* code
27	radix-10 or *decimal* code
11011	radix-2 or *binary* code
XXVII	Roman numerals

However, we don't always make the distinction between numbers and numerals and may use "decimal numbers" in lieu of "decimal numerals."

Roman numerals were used in Europe during the Middle Ages. Even when Europeans learned that the Arabs had a better way of representing numbers, it took them centuries to adopt the *Arabic system* based on numerals, or *digits*, 0-9 and a radix of 10. In these *decimal numbers*, the worth of each position is 10 times that of the adjacent position to its right, so that the string of digits "5327" represents five thousands, plus three hundreds, plus two tens, plus seven ones.

Other radices have also appeared over the ages (see A. Glaser's *History of Binary and Other Nondecimal Numeration*, Tomash Publishers, 1981). Babylonians used radix-60 numbers, which make dealing with time easy. The radices 12 (*duodecimal*) and 5 (*quinary*) have also been used. The use of radix-2 (*binary*) numbers became popular with the onset of electronic computers, because their use of binary digits, or *bits*, having only two possible values 0 and 1, is compatible with electronic signals. Radix-8 (*octal*) and radix-16 (*hexadecimal*) numbers have been used as shorthand notation for binary numbers. For example, a 24-bit binary number can be represented as an 8-digit octal or a 6-digit hexadecimal number by taking the bits in groups of threes and fours, respectively.

In a general radix-*r positional number system*, with a fixed word width of *k*, a number *x* is represented by a string of *k* digits x_i, with $0 \le x_i \le r - 1$:

$$x = \sum_{i=0 \text{ to } k-1} x_i r^i = (x_{k-1}x_{k-2} \ldots x_1x_0)_r \qquad (1)$$

For example:

$$27 = (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (11011)_{\text{two}}$$

In a k-digit radix-r number system, natural numbers from 0 to $r^k - 1$ can be represented. Conversely, given a desired representation range $[0, M - 1]$, the required number k of digits in radix r is obtained from the following equation:

$$k = \lceil \log_r M \rceil = \lfloor \log_r (M - 1) \rfloor + 1 \tag{2}$$

For example, representing the decimal number 3125 requires 12 bits in radix 2, five digits in radix 5, and four digits in radix 8.

Given a number x represented in radix r, one can obtain its radix-R representation in two ways. If we wish to perform arithmetic in the new radix R, we simply evaluate a polynomial in r whose coefficients are the digits x_i. This corresponds to Equation (1) and can be performed more efficiently by using Horner's rule which involves repeated steps of multiplying by r followed by addition:

$$(x_{k-1}x_{k-2} \, . \, . \, . \, x_1x_0)_r = ((\ldots(x_{k-1}r + x_{k-2})r + \, . \, . \, . \, x_1)r + x_0) \tag{3}$$

This method is particularly suitable for manual conversion from an arbitrary radix r to radix 10, given the relative ease with which we can perform radix-10 arithmetic.

To perform the radix conversion using arithmetic in the old radix r, we repeatedly divide the number x by the new radix R, keeping track of the remainder in each step. These remainders correspond to the radix-R digits X_i, beginning from X_0. For example, we convert the decimal number 23 to radix 2 as follows:

23 divided by 2 yields 11 with remainder 1
11 divided by 2 yields 5 with remainder 1
5 divided by 2 yields 2 with remainder 1
2 divided by 2 yields 1 with remainder 0
1 divided by 2 yields 0 with remainder 1

Reading the computed remainders from bottom to top, we find 23 = $(10111)_{\text{two}}$. Using the same process, we can convert 23 to radix 5 to get 23 = $(43)_{\text{five}}$.

II Digit Sets and Encodings

The standard digit set used for radix-r numbers is {0, 1, . . . , r – 1}. This digit set leads to a unique representation for every natural number. The binary or rdix-2 digit set is {0, 1} which is conveniently representable by electronic signals. Typically, low voltage is used to represent 0 and high voltage denotes 1, but the reverse polarity can also be used. Conceptually, the decimal digit values 0 through 9 could be represented by 10 different voltage levels. However, encoding everything with 0s and 1s makes it easier for electronic circuits to interpret and process the information speedily and reliably.

One way to encode decimal digits using binary signals is to encode each of the digits 0-9 by means of its 4-bit binary representation. The resulting *binary-coded decimal* (BCD) representation is shown below:

Digit	BCD representation
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

The use of digit values 0 through $r - 1$ in radix r is just a convention. We could use more than r digit values (for example, digit values -2 to 2 in radix 4) or use r digit values that do not start with 0 (for example, digit set $\{-1, 0, 1\}$ in radix 3). In the first instance, the resulting number system possesses redundancy in that some numbers will have multiple representations. More on this in Section XVI.

Of course, any finite set of symbols, not just digits, can be encoded using binary signals. The *American Standard Code for Information Interchange* (ASCII) is one such convention that represents upper- and lower-case letters, numerals, punctuation marks, and other symbols in an 8-bit *byte*. For example, the 8-bit ASCII codes for the ten decimal digits are of the form 0011xxxx, where the “xxxx” part is identical to the BCD code discussed earlier. ASCII digits take twice as much space as BCD digits and thus are not used in arithmetic units. Even less compact than ASCII is the 16-bit *Unicode* which can accommodate symbols from many different languages.

III Integers

The set { . . . , –3, –2, –1, 0, 1, 2, 3, . . . } of integers is also referred to as *signed or directed (whole) numbers*. The most straightforward representation of integers consists of attaching a sign bit to any desired representation of natural numbers, leading to *signed-magnitude* representation. The standard convention is to use 0 for positive and 1 for negative and attach the sign bit to the left end of the magnitude. Here are two examples:

+27 in 8-bit signed-magnitude binary code	00011011
–27 in 2-digit decimal code with BCD digits	1 0010 0111

Another option for encoding signed integers in the range $[-N, P]$ is the *biased representation*. If we add the positive value N (the bias) to all numbers in the desired range, unsigned integers in the range $[0, P + N]$ result. Any method for representing natural numbers in $[0, P + N]$ can then be used for representing the original signed integers in $[-N, P]$. This type of biased representation has only limited application in encoding of the exponents in floating-point numbers (see Section XI).

By far the most common machine encoding of signed integers is the *2's-complement representation*. In the k-bit 2's-complement format, a negative value $-x$, with $x > 0$, is encoded as the unsigned number $2^k - x$. Figure 1 shows encodings of positive and negative integers in the 4-bit 2's-complement format. Note that the positive integers 0 through 7 (or $2^{k-1} - 1$, in general) have the standard binary encoding, whereas negative values –1 through –8 (or -2^{k-1}, in general) have been transformed to unsigned values by adding 16 (or 2^k, in general) to them.

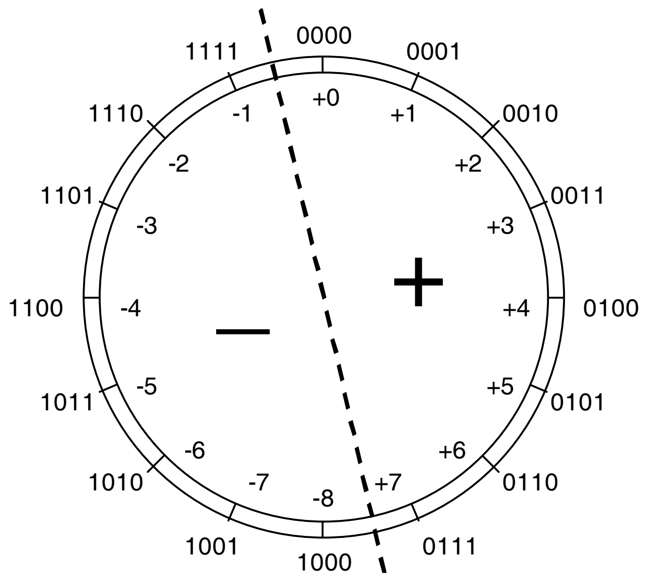

Fig. 1. Schematic representation of 4-bit 2’s-complement code for integers in [–8, +7].

Two important properties of 2’s-complement numbers are worth noting. First, the leftmost bit of the representation acts a the sign bit (0 for positive values, 1 for negative ones). Second, the value represented by a particular bit pattern can be derived without a need to follow different procedures for negative and positive values. We simply use Equation (1), as we did for unsigned integers, except that the sign bit is considered to have a negative weight. Here are two examples:

$$(01011)_{2\text{'s-compl}} = (-0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = +11$$

$$(11011)_{2\text{'s-compl}} = (-1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = -5$$

The reason for the popularity of 2’s-complement representation will become clear when we discuss addition and subtraction in Section VI.

Other complement representation systems can also be devised, but none is in widespread use. Choosing any *complementation constant* M, that is at least as large as $N + P + 1$, allows us to represent signed integers in the range $[–N, P]$, with the positive numbers in $[0, +P]$ corresponding to the unsigned values in $[0, P]$ and negative numbers in $[–N, –1]$ represented as unsigned values in $[M – N, M – 1]$. Sometimes, M itself is used as an alternate code for 0 (actually, –0). For example, the k-bit *1's-complement* system is based on $M = 2^k – 1$ and includes numbers in the range $[–(2^{k/2} – 1), 2^{k/2} – 1]$, with 0 having two representations: the all-0s string and the all-1s string.

IV Counting

The natural numbers are ordered. Each natural number x has a successor *succ*(x), which is the next number in this order and, except when $x = 0$, it has a predecessor *pred*(x). *Counting* is the act of going through the successor or predecessor sequence, beginning with some initial value. So, (3, 4, 5, 6, . . .) is an up-counting sequence and (15, 14, 13, 12, . . .) is a down-counting sequence (for a detailed treatment of various types of counters, see R.M.M. Oberman's *Counting and Counters*, Macmillan, 1981). Hardware circuits that mimic this process, advancing to the successor number or retrogressing to the predecessor each time a count control signal is asserted, are known as *up-counters* and *down-counters*, respectively. A k-bit modulo-2^k down-counter would go into negative 2's-complement values if it is decremented past zero. An *up/down-counter* can go in either direction, depending on the value of a direction control signal.

In what follows, we focus on up-counters that hold unsigned values. *Asynchronous counters* can be built of edge-triggered storage elements (flip-flops) and nothing else. The more commonly used *synchronous counters* are built of a register, a hardware incrementer, and some logic that allows the incremented count or an initial value to be loaded into the register when appropriate control signals are asserted. Figure 2 shows the block diagram for a synchronous counter.

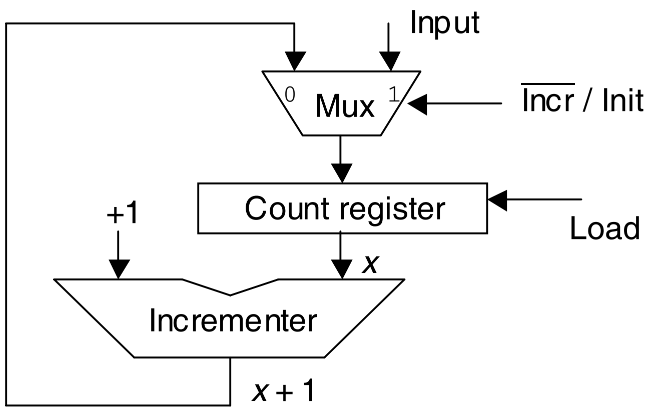

Fig. 2. Synchronous binary counter with initialization capability.

The counter design shown in Fig. 2 is adequate for most applications. It can be made faster by using a fast incrementer with carry-lookahead feature similar to that used in fast adders, to be discussed in Section VII. If still higher speed is required, the counter can be divided into blocks. A short initial block (say, 3 bits wide) can easily keep up with the fast incoming signals. Increasingly wider blocks to the left of the initial block need not be as fast because they are adjusted less and less frequently.

V Fixed-Point Numbers

A fixed-point number consists of a *whole or integral* part and a *fractional* part, with the two parts separated by a *radix point* (*decimal point* in radix 10, *binary point* in radix 2, and so on). The position of the radix point is almost always implied and thus the point is not explicitly shown. If a fixed-point number has k whole digits and l fractional digits, its value is obtained from the formula:

$$x = \sum_{i=-l \text{ to } k-1} x_i r^i = (x_{k-1}x_{k-2} \, . \, . \, . \, x_1x_0 \bullet x_{-1}x_{-2} \, . \, . \, . \, x_{-l})_r \tag{4}$$

In other words, the digits to the right of the radix point are given negative indices and their weights are negative powers of the radix. For example:

$$2.375 = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = (10\bullet011)_{\text{two}}$$

In a $(k + l)$-digit radix-r fixed-point number system with k whole digits, numbers from 0 to $r^k - r^{-l}$, in increments of r^{-l}, can be represented. The step size or resolution r^{-l} is often referred to as *ulp*, or *unit in least position*. For example, in a (2 + 3)-bit binary fixed-point number system, we have $ulp = 2^{-3}$ and the values $0 = (00\bullet000)_{\text{two}}$ through $2^2 - 2^{-3} = 3.875 = (11\bullet111)_{\text{two}}$ are representable. For the same total number $k + l$ of digits in a fixed-point number system, increasing k will lead to enlarged *range* of numbers, whereas increasing l leads to greater *precision*. Therefore, there is a tradeoff between range and precision.

Signed fixed-point numbers can be represented by the same methods discussed for signed integers: signed-magnitude, biased format, and complement method. In particular, for 2's-complement format, a negative value $-x$ is represented as the unsigned value $2^k - x$. Figure 3 shows encodings of positive and negative integers in the (1 + 3)-bit fixed-point 2's-complement format. Note that the positive values 0 to 7/8 (or $2^{k-1} - 2^{-l}$, in general) have the standard binary encoding, whereas negative values –1/8 to –1 (or -2^{-l} to -2^{k-1}, in general) are transformed to unsigned values by adding 2 (or 2^k, in general) to them.

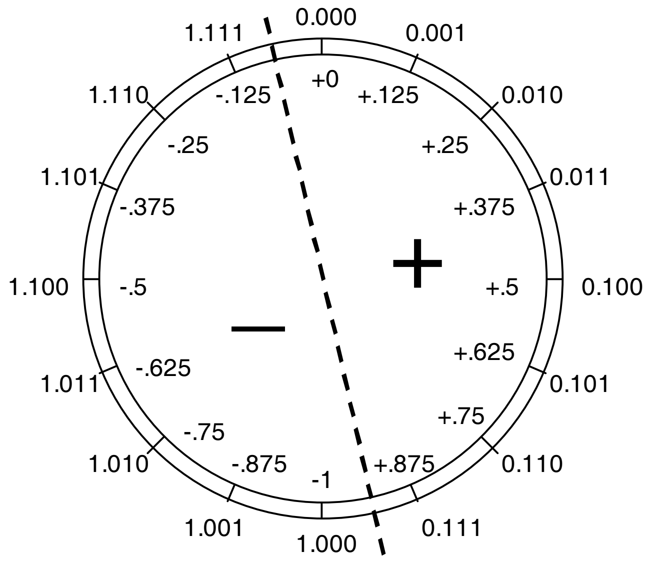

Fig. 3. Schematic representation of 4-bit 2's-complement encoding for (1 + 3)-bit fixed-point numbers in the range [–1, +7/8].

The two important properties of 2's-complement numbers, previously mentioned in connection with integers, are valid here as well; namely, the leftmost bit of the number acts a the sign bit, and the value represented by a particular bit pattern can be derived by considering the sign bit as having a negative weight. Here are two examples:

$$(01.011)_{\text{2's-compl}} = (-0 \times 2^{1}) + (1 \times 2^{0}) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = +1.375$$

$$(11.011)_{\text{2's-compl}} = (-1 \times 2^{1}) + (1 \times 2^{0}) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = -0.625$$

Conversion of fixed-point numbers from radix r to another radix R is done separately for the whole and fractional parts. Converting the whole part was discussed in Section I. To convert the fractional part, we can again use arithmetic in the new radix R or in the old radix r, whichever is more convenient. With radix-R arithmetic, we simply evaluate a polynomial in r^{-1} whose coefficients are the digits x_i. The simplest way to do this is to view the fractional part as an l-digit integer, convert this integer to radix R, and divide the result by r^{l}.

To perform radix conversion using arithmetic in the old radix r, we repeatedly multiply the fraction y by the new radix R, noting and removing the integer part in each step. These integer parts correspond to the radix-R digits X_{-i}, beginning from X_{-1}. For example, we convert .175 to radix 2 as follows:

.175 multiplied by 2 yields .350 with integer part 0
.350 multiplied by 2 yields .700 with integer part 0
.700 multiplied by 2 yields .400 with integer part 1
.400 multiplied by 2 yields .800 with integer part 0
.800 multiplied by 2 yields .600 with integer part 1
.600 multiplied by 2 yields .200 with integer part 1
.200 multiplied by 2 yields .400 with integer part 0
.400 multiplied by 2 yields .800 with integer part 0

Reading the recorded integer parts from top to bottom, we find .175 = $(.00101100)_{\text{two}}$. This equality is approximate because the result did not converge to 0. In general a fraction in one radix may not have an exact representation in another radix. In any case, we simply carry out the process above until the required number of digits in the new radix have been obtained.

VI Addition and Subtraction

We will cover only integer addition and subtraction. Fixed-point numbers that are both in the same format can be added like integers by simply ignoring the implied radix point. When two bits are added, the sum is a value in the range [0, 2] which can be represented by a *sum bit* and a *carry bit*. The circuit that can compute the sum and carry bits is known as a *half-adder* (HA) with its truth table and symbolic representation shown in Fig. 4. The carry output is the logical AND of the two inputs, while the sum output is the exclusive OR (XOR) of the inputs. By adding a carry input to a half-adder, we get a binary *full-adder* (FA) whose truth table and schematic diagram are depicted in Fig. 5.

Inputs		Outputs	
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

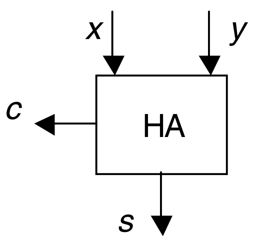

Fig. 4. Truth table and schematic diagram for a binary half-adder.

Inputs			Outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

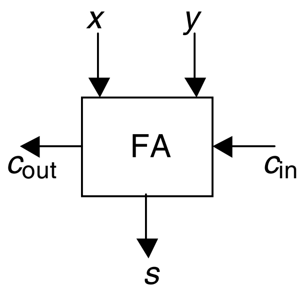

Fig. 5. Truth table and schematic diagram for a binary full-adder.

A full-adder, connected to a flip-flop for holding the carry bit from one cycle to the next, functions as a *bit-serial adder*. The inputs of a bit-serial adder are supplied in synchrony with a clock signal, one bit from each operand per clock cycle, beginning from the least-significant bits. One bit of the output is produced per clock cycle and the carry from one cycle is held and used as input in the next cycle. A *ripple-carry adder*, on the other hand, unfolds this sequential behavior into space, using a cascade of *k* full-adders to add two *k*-bit numbers (Fig. 6).

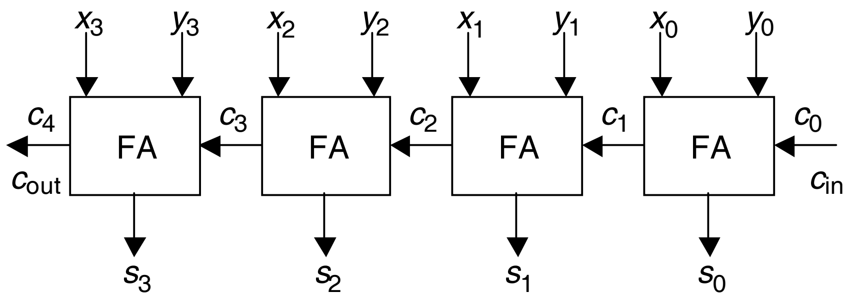

Fig. 6. Four-bit ripple-carry binary adder.

The ripple-carry design of Fig. 6 becomes a radix-r adder if each binary full-adder is replaced by a radix-r full-adder that accepts two radix-r digits (each encoded in binary) and a carry-in signal, producing a radix-r sum digit and a carry-out signal. Because the carry signals are always binary and their propagation is independent of the radix r, in the rest of this section and in Section VII, we do not deal with radices other than 2.

An adder/subtractor for 2's-complement numbers can be built by adding a controlled complementation circuit (a two-way multiplexer with y and its bitwise complement as the two inputs) to an adder of any type. The resulting circuit is shown in Fig. 7. To justify this design, note that $x - y$ can be computed by forming the 2's-complement of y and adding it to x. However, the 2's-complement of y, which is $2^k - y$, can be computed by adding 1 to $(2^k - 1) - y$, the bitwise complement of y. The addition of 1 is accomplished by inserting a carry-in of 1 into the adder when it is used to perform subtraction.

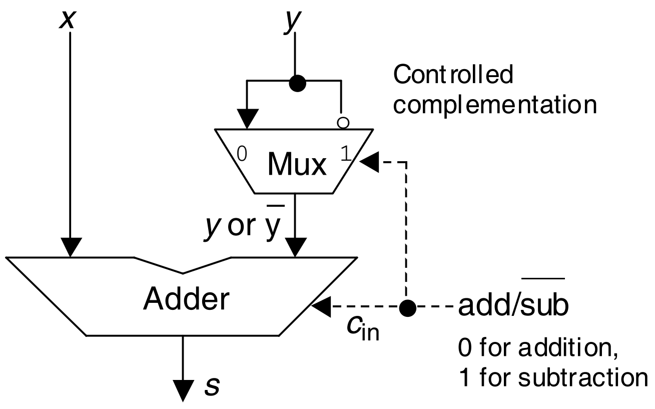

Fig. 7. Two's-complement adder/subtractor.

VII Fast Adders

Ripple-carry adders are quite simple and easily expandable to any desired width. However, they are rather slow, because carries may propagate across the full width of the adder. This happens, for example, when the two 8-bit numbers 10101011 and 01010101 are added. Because each full-adder requires some time to generate its carry output, cascading k such units together implies k times as much signal delay in the worst case. This linear amount of time becomes unacceptable for wide words (say, 32 or 64 bits) or in high-performance computers, though it might be acceptable in an embedded system that is dedicated to a single task and is not expected to be fast.

A variety of fast adders can be designed that require logarithmic, rather than linear, time. In other words, the delay of such fast adders grows as the logarithm of k. The best-known and most widely used such adders are *carry-lookahead adders*. The basic idea in carry-lookahead addition is to form the required intermediate carries directly from the inputs rather than from the previous carries, as was done in Fig. 6. For example, the carry c_3 in Fig. 6, which is normally expressed in terms of c_2 using the carry recurrence

$$c_3 = x_2y_2 + (x_2 \oplus y_2)c_2$$

can be directly derived from the inputs based on the logical expression:

$$c_3 = x_2y_2 + (x_2 \oplus y_2)x_1y_1 + (x_2 \oplus y_2)(x_1 \oplus y_1)x_0y_0 + (x_2 \oplus y_2)(x_1 \oplus y_1)(x_0 \oplus y_0)c_0$$

To simplify the rest of our discussion of fast adders, we define the *carry generate* and *carry propagate* signals and use them in writing a carry recurrence that relates c_{i+1} to c_i:

$$g_i = x_i\, y_i$$

$$p_i = x_i \oplus y_i$$

$$c_{i+1} = g_i + p_i\, c_i$$

The expanded form of c_3, derived earlier, now becomes:

$$c_3 = g_2 + p_2\, g_1 + p_2\, p_1\, g_0 + p_2\, p_1\, p_0\, c_0$$

It is easy to see that the expanded expression above would grow quite large for a wider adder that requires c_{31} or c_{52}, say, to be derived. A variety of *lookahead carry networks* exist that systematize the preceding derivation for all the intermediate carries in parallel and make the computation efficient by sharing parts of the required circuits whenever possible. Various designs offer tradeoffs in speed, cost, VLSI chip area, and power consumption. Information on the design of fast carry networks and other types of fast adders can be found in books on computer arithmetic.

Here, we present just one example of a fast carry network. The building blocks of this network are the *carry operator* which combines the generate and propagate signals for two adjacent blocks $[i + 1, j]$ and $[h, i]$ of digit positions into the respective signals for the wider combined block $[h, j]$. In other words

$$[i + 1, j] ¢ [h, i] = [h, j]$$

where $[a, b]$ stands for $(g_{[a,b]}, p_{[a,b]})$ representing the pair of generate and propagate signals for the block extending from digit position a to digit position b. Because the problem of determining all the carries c_i is the same as computing the cumulative generate signals $g_{[0,i]}$, a network built of ¢ operator blocks, such as the one depicted in Fig. 9, can be used to derive all the carries in parallel. If a c_{in} signal is required for the adder, it can be accommodated as the generate signal g_{-1} of an extra position on the right.

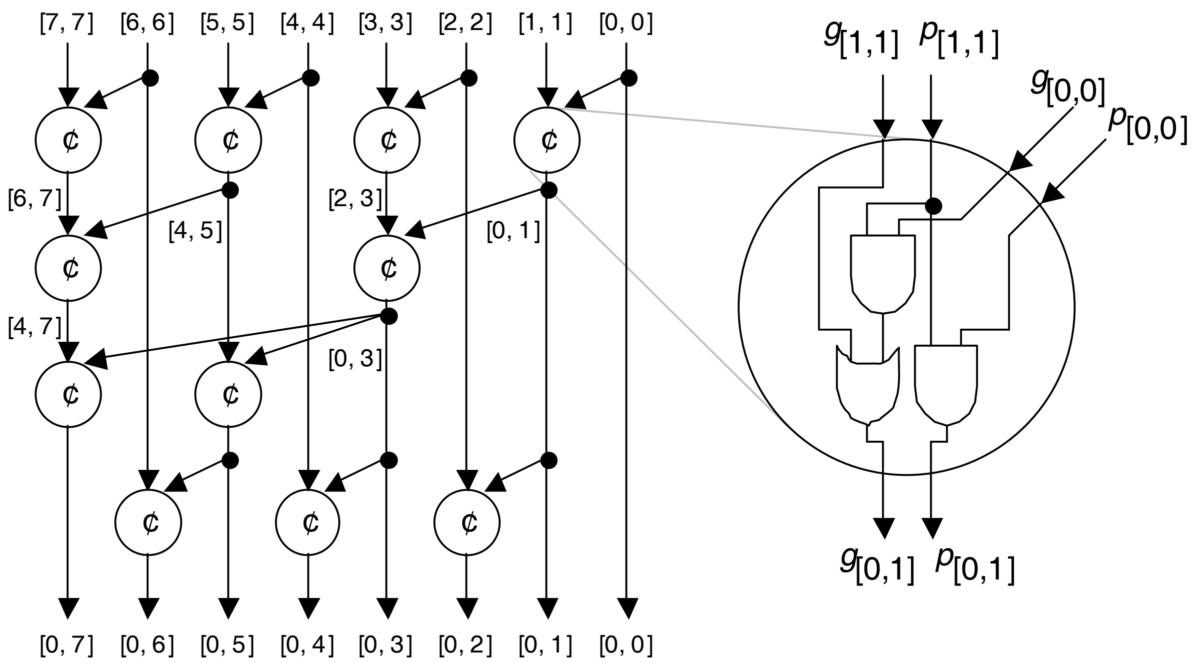

Fig. 9. Brent-Kung lookahead carry network for an 8-digit adder.

For a k-digit adder, the number of operator blocks on the critical path of the carry network exemplified by Fig. 9 is $2(\lceil \log_2 k \rceil - 1)$. Many other carry networks can be designed that offer speed-cost tradeoffs.

An important method for fast adder design, that often complements the carry-lookahead scheme, is *carry-select*. In the simplest application of the carry-select method, a k-bit adder is built of a $(k/2)$-bit adder in the lower half, two $(k/2)$-bit adders in the upper half (forming two versions of the $k/2$ upper sum bits with $c_{k/2} = 0$ and $c_{k/2} = 1$), and a multiplexer for choosing the correct set of values once $c_{k/2}$ becomes known. A hybrid design, in which some of the carries (say, c_8, c_{16}, and c_{24} in a 32-bit adder) are derived via carry-lookahead and are then used to select one of two versions of the sum bits that are produced for 8-bit blocks concurrently with the operation of the carry network, is quite popular in modern arithmetic units.

VIII Multiplication

The simplest machine multipliers are designed to follow a variant of the pencil-and-paper multiplication algorithm depicted in Fig. 9, where each row of dots in the *partial products bit-matrix* is either all 0s (if the corresponding $y_i = 0$) or the same as x (if $y_i = 1$). When we perform a $k \times k$ multiplication manually, we form all of the k partial products and add the resulting k numbers to obtain the product p.

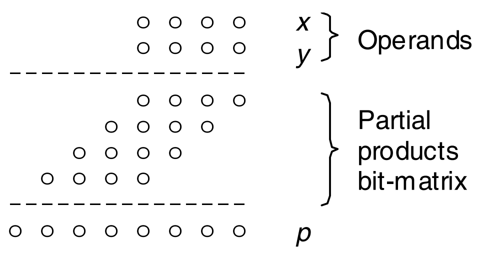

Fig. 9. Multiplication of 4-bit numbers in dot notation.

For machine execution, it is easier if the *cumulative partial product* is initialized to 0, each row of the bit-matrix added to it as the corresponding term is generated, and the result of addition shifted to the right by one bit to achieve proper alignment with the next term, as depicted in Fig. 9. In fact, this is exactly how *programmed multiplication* is performed on a machine that does not have a hardware multiply unit. The recurrence equation describing the process above is:

$$p^{(j+1)} = \underbrace{(\underbrace{p^{(j)} + y_j\, x\, 2^k}_{\text{add}})\, 2^{-1}}_{\text{shift right}} \quad \text{with } p^{(0)} = 0 \text{ and } p^{(k)} = p \tag{5}$$

Because by the time we are done, the right shifts will have caused the first partial product to be multiplied by 2^{-k}, we pre-multiply x by 2^k to offset the effect of these right shifts. This is not an actual multiplication but is done by aligning x with the upper half of the $2k$-bit cumulative partial product in the addition steps. Figure 10 depicts a possible hardware realization of the foregoing *shift-add multiplication* algorithm. The shifting of the partial product need not be done in a separate step but can be incorporated in the connecting wires that go from the adder output to the doublewidth register.

After k iterations, recurrence (5) leads to:

$$p^{(k)} = xy + p^{(0)}2^{-k}$$

Thus if $p^{(0)}$ is initialized to $2^k z$ (z padded with k zeros) instead of 0, the expression $xy + z$ will be evaluated. This *multiply-add operation* is quite useful for many applications and is performed at essentially no extra cost compared to plain shift-add multiplication.

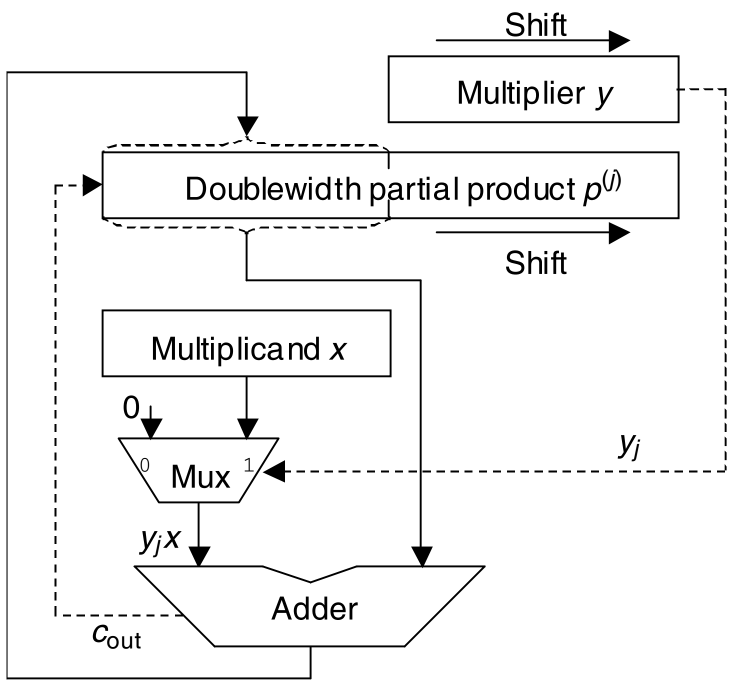

Fig. 10. Hardware multiplier based on the shift-add algorithm.

The preceding bit-at-a-time multiplication scheme can be easily extended to a digit-at-a-time algorithm in a higher radix such as 4, 8, or 16. In this case, the multiplexer in Fig. 10 (which is really a bit-by-number multiplier) must be replaced with a digit-by-number multiplier circuit (perhaps implemented as a multioperand adder), the single-bit shifts replaced by h-bit shifts for radix-2^h algorithm, and the number of iterations reduced to from k to k/h. These faster *high-radix multipliers* may still be too slow for some applications. In such cases, fully combinational *tree multipliers* are used in which the addition of the partial products bit-matrix is done by means of a tree-structured combinational circuit.

IX Fast Multipliers

Instead of developing the partial products one at a time in radix 2 or in radix 2^h, we can form all of them simultaneously, thus reducing the multiplication problem to n-operand addition, where $n = k$ in radix 2, $n = k/2$ in radix 4, and so on. For example, a 16×16 multiplication becomes a 16-operand addition problem in radix 2 or an 8-operand addition problem in radix 4.

In *tree multipliers*, the n operands thus formed are added in two stages. In stage 1, a tree built of *carry-save adders* or similar *compression circuits* is used to reduce the n operands to two operands that have the same sum as the original n numbers. A carry-save adder (see Section XVI) reduces three values to two values, for a reduction factor of 1.5, thus leading to a $\lceil \log_{1.5}(n/2) \rceil$-level circuit for reducing n numbers to two. The two numbers thus derived are then added by a fast logarithmic-time adder, leading to an overall logarithmic latency for the multiplier (Fig. 11).

Such a *full-tree multiplier* is rather complex and its speed may not be needed for all applications. In such cases, more economical *partial-tree multipliers* might be implemented. For example, if about half of the partial products are accommodated by the tree part, then two passes through the tree can be used to form the two numbers representing the desired product, with the results of the first pass fed back to the inputs and combined with the second set of partial products (Fig. 11). A partial-tree multiplier can be viewed as a (very-) high-radix multiplier. For example, if 12 partial products are combined in each pass, then a radix-2^{12} multiplication is effectively performed.

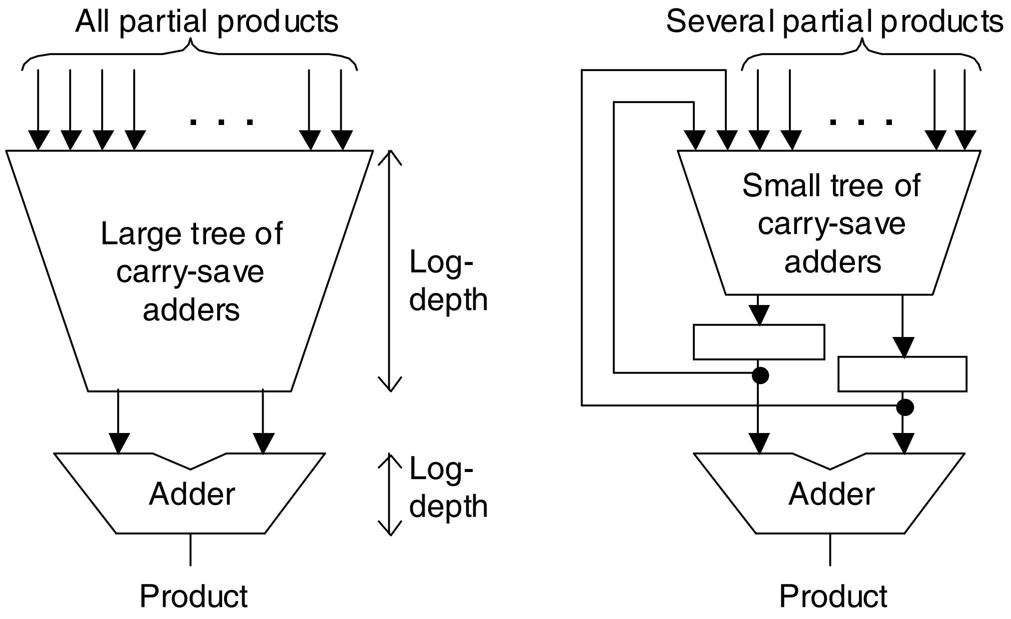

Fig. 11. Schematic diagrams for full-tree and partial-tree multipliers.

An *array multiplier* uses the same two-stage computation scheme of a tree multiplier, with the difference that the tree of carry-save adders is one-sided (has the maximum possible depth) and the final adder is of ripple-carry type (quite slow). An example 4 x 4 array multiplier is depicted in Fig. 12. HA and FA cells are half- and full-adders defined in Figs. 4 and 5, respectively, and MA cells are modified full-adders, one of whose inputs is internally formed as the logical AND of x_i and y_j.

The reader may well ask why such a slow tree-type multiplier is of any interest at all. The answer is that array multipliers are quite suitable for VLSI realization, given their highly regular design and efficient wiring pattern. They can also be readily pipelined by inserting latches between some of the rows of cells, thus allowing several multiplications to be performed on the same hardware structure.

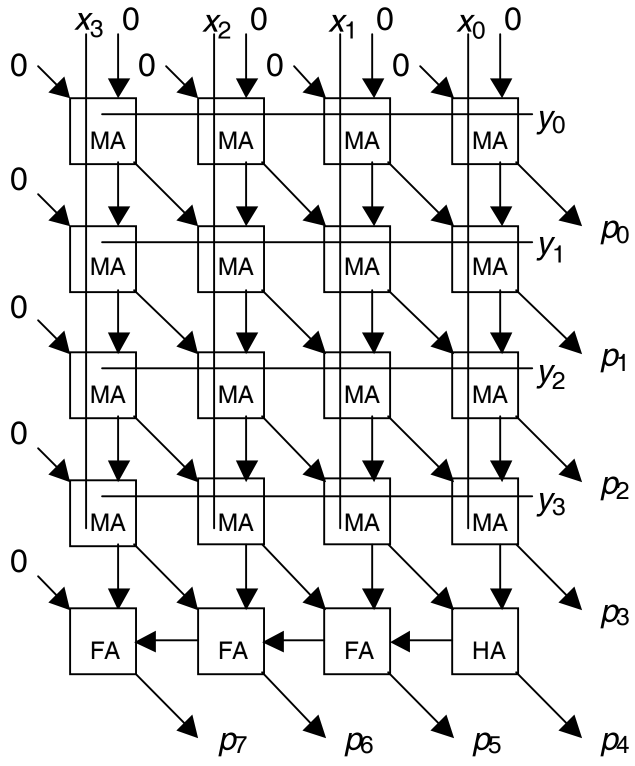

Fig. 12. Array multiplier for 4-bit unsigned operands.

X Division

Like multipliers, the simplest machine dividers are designed to follow a variant of the pencil-and-paper division algorithm depicted in Fig. 13, where each row of dots in the subtracted bit-matrix is either all 0s (if the corresponding $q_i = 0$) or the same as y (if $q_i = 1$). When we perform a $2k / k$ division manually, we form the subtracted terms one at a time by "guessing" the value of the next *quotient digit*, subtract the appropriate term (0 or a suitably shifted version of y) from the *partial remainder* (initialized to the value of the dividend x), and proceed until all k bits of q have been determined. At this time, the partial remainder becomes the final remainder z.

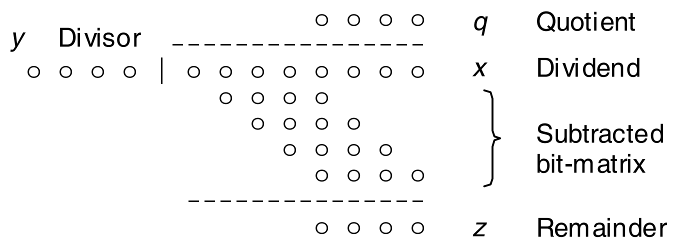

Fig. 13. Division of an 8-bit number by a 4-bit number in dot notation.

For hardware or software implementation, a recurrence equation describing the process above is used:

$$z^{(j+1)} = \underbrace{\underbrace{2\,z^{(j)}}_{\text{shift}} - q_{k-j}\, y\, 2^k}_{\text{subtract}} \quad \text{with} \quad z^{(0)} = x \quad \text{and} \quad z^{(k)} = 2^k z \tag{6}$$

Because by the time we are done, the left shifts will have caused the partial remainder to be multiplied by 2^k, the true remainder is obtained by multiplying the final partial remainder by 2^{-k} (shifting it to the right by k bits). Figure 14 depicts a possible hardware realization of the foregoing *shift-subtract division* algorithm. As in multiplication, the shifting of the partial remainder need not be done in a separate step but can be incorporated in the wires connecting the adder output to the partial remainder register.

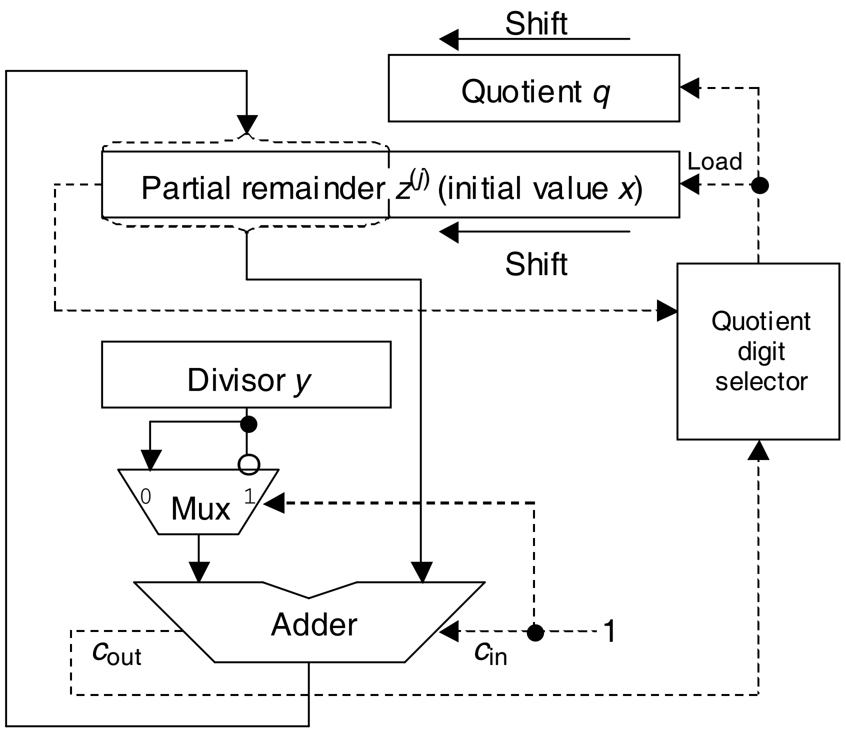

Fig. 14. Hardware divider based on the shift-subtract algorithm.

A comparison of Figs. 10 and 14 reveals that multipliers and dividers are quite similar and can be implemented with shared hardware within an *arithmetic/logic unit* (ALU) that performs different operations based on an externally supplied *function code*.

As in the case of multipliers, *high-radix dividers* speed up the division process by producing several bits of the quotient in each cycle. Whereas there is no counterpart to fast tree multipliers for performing division, *array dividers* do exist and are structurally quite similar to the array multiplier of Fig. 12. It is also possible to perform division by using a sequence of multiplications instead of additions. Even though multiplications are slower and costlier to implement than additions, advantage over additive schemes may be gained because far fewer multiplications are needed to perform division. Details can be found in any book on computer arithmetic.

XI Real Numbers

Integers in a prescribed range can be represented exactly for automatic processing, but most real numbers must be approximated within the machine's finite word width. Some real numbers can be represented as, or approximated by, $(k + l)$-bit fixed-point numbers (see Section V). A problem with fixed-point representations is that they are not very good for dealing with very large and extremely small numbers at the same time. Consider the two (8 + 8)-bit fixed-point numbers shown below:

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

The relative representation error due to truncation or rounding of digits beyond the –8th position is quite significant for x, but it is much less severe for y. On the other hand, neither y^2 nor y / x is representable in this number format.

The most common representation of numbers for computations dealing with values in a wide range is the *floating-point format*. Old computers used many different floating-point number formats, and some specialized machines still do, but for the most part, the IEEE floating-point standard format (ANSI/IEEE Standard 754-1985; available from IEEE Press) has been adopted by the computer industry. We thus formulate our discussion of floating-point numbers and arithmetic exclusively in terms of this standard format. Other formats will differ in their parameters and representation details, but the basic tradeoffs and algorithms remain the same.

A floating-point number has three components: sign ±, exponent e, and significand s, together representing the value $\pm 2^e s$. The *exponent* is a signed integer represented in biased format (a fixed bias is added to it to make it into an unsigned number). The *significand* is a fixed-point number in the range [1, 2). Because the binary representation of the significand always starts with “1.”, this fixed 1 is hidden and only the fractional part of the significand is explicitly represented.

Figure 15 shows the details of short (32-bit) and long (64-bit) floating-point formats. The short format has adequate range and precision for most common applications (magnitudes ranging from 1.2×10^{-38} to 3.4×10^{38}). The long format is used for highly precise computations or those involving extreme variations in magnitude (from about 2.2×10^{-308} to 1.8×10^{308}). Of course in both of these formats, as explained thus far, zero has no proper representation. To remedy this problem, and to be able to represent certain other special values, the smallest and largest exponent codes (all 0s and all 1s in the biased exponent field) are not used for ordinary numbers. An all-0s word (0s in sign, exponent, and significand fields) represents +0; similarly, –0 and ±∞ have special representations, as does any nonsensical or indeterminate value, known as “not a number” (NaN). Certain other details of this standard are beyond the scope of this article.

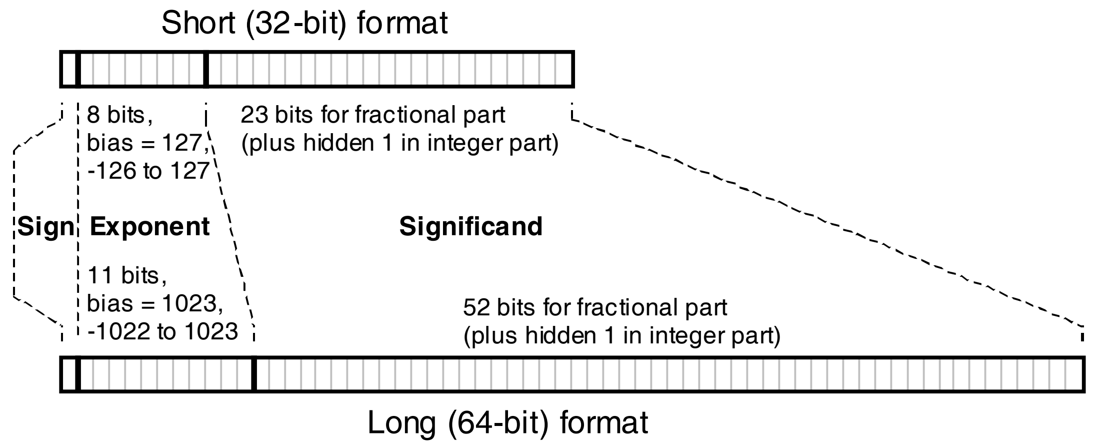
Short (32-bit) format
8 bits, bias = 127, -126 to 127
23 bits for fractional part (plus hidden 1 in integer part)
Sign
Exponent
Significand
11 bits, bias = 1023, -1022 to 1023
52 bits for fractional part (plus hidden 1 in integer part)
Long (64-bit) format

Fig. 15. The ANSI/IEEE standard floating-point formats.

When an arithmetic operation produces a result that is not exactly representable in the format being used, the result must be rounded to some representable value. The ANSI/IEEE standard prescribes four rounding options. The default rounding mode is “*round to nearest even*”: choose the closest representable value and, in case of a tie, choose the value with its least-significant bit 0. There are also three *directed rounding* modes: “*round toward* +∞” (choose the next higher value), “*round toward* –∞” (choose the next lower value), and “*round toward* 0” (choose the closest value that is less than the value at hand in magnitude). With the round-to-nearest option, the maximum rounding error is 0.5 *ulp*, while with directed rounding schemes, the error can be up to 1 *ulp*.

XII Floating-Point Arithmetic

The floating-point operations of multiplication and division are not much different from their fixed-point counterparts. For multiplication, exponents of the two operands are added and their significands are multiplied:

$$(\pm 2^{e1} s1) \times (\pm 2^{e2} s2) = \pm 2^{e1+e2}(s1 \times s2)$$

Thus, a hardware floating-point multiplier consists of a significand multiplier and an exponent adder that together compute $2^e s$, with $e = e1 + e2$ and $s = s1 \times s2$. The result's sign is easily obtained from the signs of the operands. This is not all, however. With $s1$ and $s2$ in [1, 2), their product will lie in [1, 4) and may thus be outside the permitted range. If the product of the two significands is in [2, 4), dividing it by 2 via a 1-bit right shift will put it back into the desired range. When this *normalization* is needed, the exponent e must be incremented by 1 to compensate for the division of s by 2.

Floating-point division is similar and is governed by the equation:

$$(\pm 2^{e1} s1) / (\pm 2^{e2} s2) = \pm 2^{e1-e2}(s1 / s2)$$

Again, given that the ratio $s1 / s2$ of the two significands lies in (0.5, 2), normalization may be required for results that are less than 1. This normalization consists of multiplying the significand by 2 via a 1-bit left shift and decrementing the resulting exponent by 1 to compensate for the doubling of the significand. Of course, throughout the operation and ensuing adjustments, for both multiplication and division, the hardware must check for exceptions such as *overflow* (exponent too large) and *underflow* (exponent too small).

We next discuss floating-point addition. Subtraction can be converted to addition by changing the sign of the subtrahend. To perform addition, the exponents of the two operands must be equalized, if needed. Consider the addition of $\pm 2^{e1}s1$ and $\pm 2^{e2}s2$, with $e1 > e2$. By making both exponents equal to $e1$, the addition becomes:

$$(\pm 2^{e1}s1) + (\pm 2^{e1}(s2/2^{\,e1-e2})) = \pm 2^{e1}(s1 \pm s2/2^{\,e1-e2})$$

We thus see that $s2$ must be right-shifted by $e1 - e2$ bits before being added to $s1$. This *alignment shift* is also called *preshift* (so named to distinguish it from the *postshift* needed for normalizing a floating-point result). Figure 16 shows a complete example of floating-point addition, including preshift, addition of aligned significands, and final rounding. In this example, no postshift is needed because the result is already normalized. In general, though, the result may need a 1-bit right shift, when it is in [2, 4), or a multibit left shift when the addition of operands with different signs leads to *cancellation* or *loss of significance* and one or more leading 0s appear in the result.

Numbers to be added:

$$x = 2^5 \times 1.00101101$$
$$y = 2^1 \times 1.11101101 \longleftarrow$$

Operand with smaller exponent to be preshifted

Operands after alignment shift:

$$x = 2^5 \times 1.00101101$$
$$y = 2^5 \times 0.000111101101$$

Result of addition:

Extra bits to be rounded off

$$s = 2^5 \times 1.01001011\overbrace{1101}$$
$$s = 2^5 \times 1.01001100 \longleftarrow$$

Rounded sum

Fig. 16. Alignment shift and rounding in floating-point addition.

A simplified block diagram for a hardware floating-point adder is shown in Fig. 17. It is seen that once the operands are unpacked, their exponents and significands are processed in two separate tracks whose functions are coordinated by the block labeled "Control & sign logic." For example, based on the difference of the two exponents, this unit decides which operand must be preshifted. To economize on hardware, usually only one preshifter is provided, say for the left operand of the adder. If the other operand needs to be preshifted, the operands are physically swapped. Also, in adding operands with unlike signs, the operand that is not preshifted is complemented. This time-saving strategy may lead to the computation of $y - x$ when in fact $x - y$ is the desired result. The control unit corrects this problem by forcing the complementation of the result if needed. Finally, normalization and rounding are preformed and the exponent is adjusted accordingly.

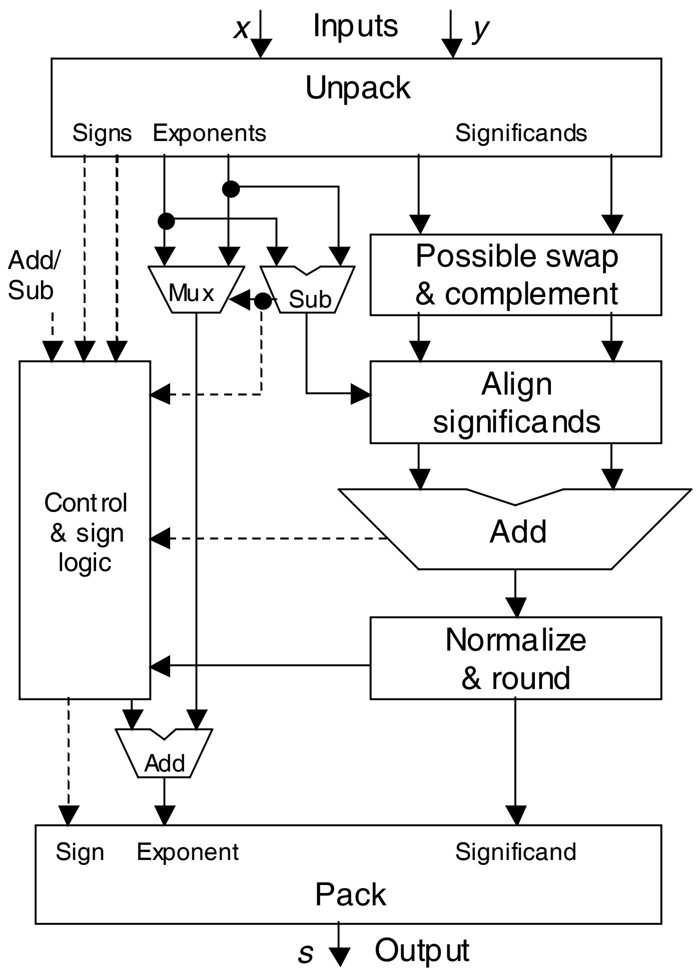

Fig. 17. Simplified schematic of a floating-point adder.

XIII Function Evaluation

In many numeric calculations, there is a need to evaluate functions such as square-root, logarithm, sine, or tangent. One approach to function evaluation is the use of an *approximating function* that is easier to evaluate than the original function. Polynomial approximations, derived from Taylor-series and other expansions, allow function evaluation by means of addition, multiplication, and division. Here are a few examples:

$$\ln x = 2(z + z^3/3 + z^5/5 + z^7/7 + \ldots) \quad \text{where } z = (x - 1)/(x + 1)$$

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + x^4/4! + \ldots$$

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - \ldots$$

$$\tan^{-1} x = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - \ldots$$

A second approach is *convergence computation*: begin with a suitable approximation and proceed to refine the value with iterative evaluation. For example, if $q^{(0)}$ is an approximation to the square root of x, the following recurrence can be used to refine the value, using one addition, one division, and a 1-bit right shift per iteration:

$$q^{(i+1)} = 0.5(q^{(i)} + x/q^{(i)})$$

The initial approximation can be obtained via table lookup based on a few high-order bits of x or simply be taken to be a constant. As an example, suppose that we want to use this method to extract the square root of a floating-point number. To do this, we must halve the exponent and find the square root of the significand. If the exponent is odd, we can subtract 1 from it and double the significand, before applying this method. As a result, the adjusted significand will be in [1, 4). We may thus take 1.5 as our initial approximation. The better the initial approximation, the fewer the number of iterations needed to achieve a certain level of accuracy. A special case of convergence computation is when each iteration leads to the determination of one digit of the result, beginning with the most-significant digit. Such *digit recurrence* schemes are similar in nature to shift-subtract division discussed in Section X.

There are many other methods for function evaluation. As our third and final example, we consider a method based on lookup tables which is becoming increasingly popular, given that tables can be implemented efficiently and compactly in VLSI technology. Within a reasonably narrow interval $[x^{(i)}, x^{(i+1)})$, a function $f(x)$ can be approximated by the linear function $a + b(x - x^{(i)})$. This *interpolation scheme* leads to the hardware implementation depicted in Fig. 18. The range of x values is divided into 2^h intervals based on the h high-order bits of x, which define the value x_H. For each of these intervals $[x_H, x_H + 2^{-h})$, the corresponding a and b values of the approximating linear function $a + b(x - x_H) = a + bx_L$ are stored in two tables. Function evaluation with this method thus involves two table accesses, one multiplication, and one addition.

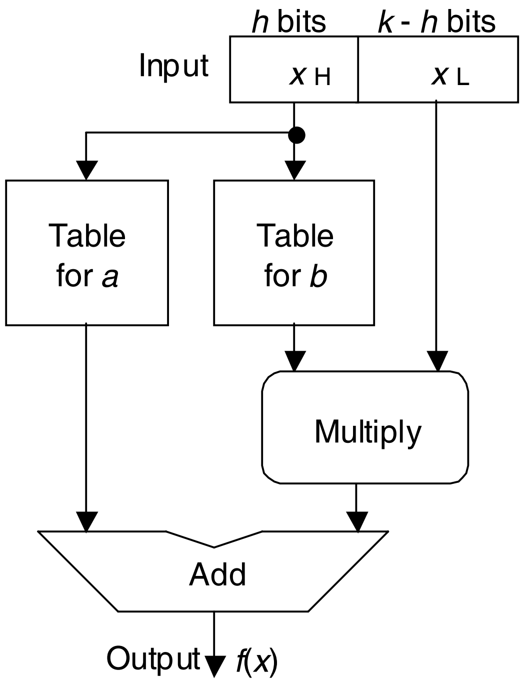

Fig. 18. Function evaluation by table lookup and linear interpolation.

XIV Precision and Errors

Machine arithmetic is inexact for two different reasons. First, many numbers do not have exact binary representations within a finite word format. This is referred to as *representation error*. Second, even for values that are exactly representable, floating-point arithmetic produces inexact results. For example, the exactly computed product of two short floating-point numbers will have a 48-bit significand that must be rounded to fit in 23 bits (plus the hidden 1). This is characterized as *computation error*.

It is important for both the designers of arithmetic circuits and for the users of machine arithmetic to be mindful of these errors and to learn methods for estimating and controlling them. There are documented instances of arithmetic errors leading to disasters in computer-controlled critical systems. Even a small per-operation error of 0.5 *ulp*, when accumulated over many millions, perhaps billions, of operations needed in some applications, can lead to highly inaccurate or totally incorrect results.

We limit our discussion of errors, their sources, and countermeasures to a few examples from floating-point arithmetic (see D. Goldberg's article in "further reading" for more detailed discussion and other examples).

One way to avoid excessive error accumulation is to carry extra precision in the course of a computation. Even inexpensive calculators use extra digits that are invisible to the user but help ensure greater accuracy for the results. Without these *guard digits*, the computation of 1/3 will produce 0.333 333 333 3, assuming a 10-digit calculator.

Multiplying this value by 3 will yield 0.999 999 999 9, instead of the expected 1. In a calculator with two guard digits, the value of 1/3 is evaluated and stored as 0.333 333 333 333, but still displayed as 0.333 333 333 3. If we now multiply the stored value by 3, and use rounding to derive the result to be displayed, the expected value 1 will be obtained.

Use of guard digits improves the accuracy of floating-point arithmetic but does not totally eliminate some incorrect and highly surprising results. For example, floating-point addition is not associative in that the algebraically equivalent computations $(x + y) + z$ and $x + (y + z)$ may yield very different results. Similarly. Many other laws of algebra do not hold for floating-point arithmetic, causing difficulties in result predictability and certification. An optimizing compiler that switches the order of evaluation for the sake of computation speed-up may inadvertently change the result obtained!

One of the sources of difficulties is loss of precision that occurs when subtraction is performed with operands of comparable magnitudes. Such a subtraction produces a result that is close to 0, making the effect of previous roundings performed on the operands quite significant in relative terms. Such an event is referred to as *catastrophic cancellation*. For example, when the algebraically correct equation

$$A = [s(s - a)(s - b)(s - c)]^{1/2}$$

with $s = (a + b + c)/2$, is used to calculate the area of a needlelike triangle (a triangle for which one side a is approximately equal to the sum $b + c$ of the other two sides), a large error can be produced due to the catastrophic cancellation in computing $s - a$. A user or programmer who is aware of this problem can use an alternate formula that is not prone to producing such large errors.

Because of the anomalies and surprises associated with floating-point arithmetic, there is some interest in *certifiable arithmetic*. An example is offered by *interval arithmetic* whereby each number is represented by a pair of values, a lower bound and an upper bound. We represent x by the interval $[x_l, x_u]$ if we are certain that $x_l \le x \le x_u$. Given interval representations of x and y, arithmetic operations can be defined in such a way as to ensure containment of the result in the interval that is produced as output. For example:

$$[x_l, x_u] + [y_l, y_u] = [x_l + y_l, x_u + y_u]$$

In this way, we always have a guaranteed error bound and will know when a result is too imprecise to be trusted.

The ultimate in result certification is *exact arithmetic*, which may be feasible in some applications through the use of *rational numbers* or other forms of exact representation. For example, if each value is represented by a sign and a pair of integers for the numerator and denominator, then numbers such as 1/3 will have exact representations and an expression such as $(1/3) \times 3$ will always produce the exact result. However, besides limited applicability, exact rational arithmetic also implies significant hardware and/or time overhead.

XV Speed and Reliability

We would, of course, like to design arithmetic circuits to be as fast as possible. The adder or multiplier must indeed be very fast if a machine is to execute one billion arithmetic operations per second. We now have this level of performance on some desktop computers, with 10^3 times greater performance in the largest available supercomputers, and 10^6 times more being worked on in research laboratories. Therefore, methods for speeding up arithmetic algorithms and their circuit implementations form an important part of the field of computer arithmetic.

With modern VLSI technologies, design for high speed is a challenging undertaking. The crossing of the gigahertz milestone in microprocessor clock rates signals the fact that many hardware operations are occurring with subnanosecond latencies. Because an electronic signal can travel only a few centimeters in one nanosecond, the roles of interconnects and package boundary crossings are becoming increasingly important. Given that clock distribution accounts for a significant portion of long wires and power dissipation on a modern VLSI chip, there is significant incentive to do away with the clocked or synchronous design style and adopt a fully asynchronous approach.

Speed is but one of several parameters that a designer must consider. In recent years, compactness and power economy have emerged as important attributes of an implementation. Design for compactness requires careful attention to implementation and packaging technologies and their various constraints. One important aspect to consider is the pin limitation at the chip and other levels of the hardware packaging hierarchy. Power economy is somewhat related to compactness, but it also depends on signal transition activity and circuit speed (faster circuit technologies use more power). Signal transition activity can be reduced via judicious choice of encoding schemes and careful algorithm design. Circuit speed can be reduced, while still keeping the same level of performance through various architectural schemes.

A particularly useful method of designing high-throughput arithmetic circuits without a need for ultrahigh-speed circuit elements is *pipelining*. We explain the use of pipelining and the various tradeoffs involved in its implementation through an example. Consider the array multiplier of Fig. 12. The critical (longest) signal path through this circuit goes through four MA, one HA, and three FA blocks. This path begins at the upper left corner of the diagram, proceeds diagonally to the lower right and then horizontally to the lower left corner. Assuming, for simplicity, that all three block types have the same unit-time latency, one multiplication can be performed every 8 time units.

By cutting the design of Fig. 12 in half, right below the last row of MA blocks, and inserting temporary storage platforms (latches) to hold intermediate signal values at that point, we can double the throughput of our multiplier. Once the intermediate signals from one multiplication are stored in the latches, a second multiplication can begin in the upper part of the circuit, while the lower part completes the first multiplication. Of course, if we insert more latches, the throughput will increase even further. The improvement is not indefinite, though, because the insertion of more latches introduces increasing overheads in cost and time.

When results of an arithmetic unit are critical to the safety of humans or success of a mission, some form of checking must be performed. Design methods for *fault-tolerant arithmetic* form active research areas. Here, we just mention one simple method based on *residue checking* (for more information on this and other types of checking, see T.R.N. Rao's *Error Codes for Arithmetic Processors*, Academic Press, 1974).

Suppose that each value is represented by appending to it the residue modulo A, where A is a suitably chosen check constant. Then addition can be checked in the manner shown in Fig. 19. If the mod-A sum of the two check tags does not match the mod-A value of the computed sum s, then the presence of an error is signaled. Special attention must be paid to the design of the various components in Fig. 19 to ensure that matching of the two residues implies fault-free operation with very high probability.

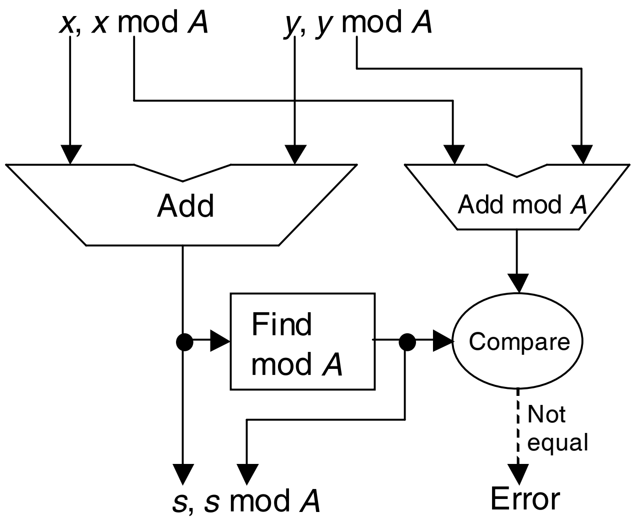

Fig. 19. Arithmetic unit with residue checking.

XVI Unconventional Number Systems

Thus far, our discussion of computer arithmetic was based mostly on standard representations that are widely used in digital systems; namely, 2's-complement binary and ANSI/IEEE floating-point format. Other number representation systems are either invisible to the computer user (used to encode intermediate values for speeding up arithmetic operations) or are applied in the design of application-specific systems. In this section, we briefly review two examples of number systems in each of these categories. These are only meant to give the reader a sense that other options exist (see B. Parhami's textbook in "further reading" for more detailed discussion and other examples).

The *carry-save* (or *stored-carry*) representation for binary numbers is extensively used in the design of fast multipliers and other arithmetic circuits. A carry-save number is essentially a pair of binary numbers whose sum is the value being represented. Given such a carry-save value, it can be added to an ordinary binary number, yielding a carry-save result at very high speed, because the addition does not involve any carry propagation. Figure 20 shows the addition of a binary number x to a carry-save number composed of y and z, yielding the carry-save number composed of s and c. Comparing Fig. 20 to Fig. 6 reveals the origins of the name "carry-save"; note that here, unlike in Fig. 6, carries are not connected to the next FA block but are saved along with the sum bits to form a carry-save number.

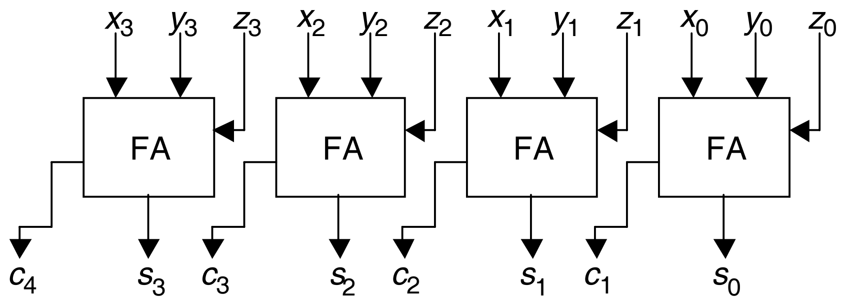
x_3 y_3 z_3 x_2 y_2 z_2 x_1 y_1 z_1 x_0 y_0 z_0
FA
FA
FA
FA
c_4 s_3 c_3 s_2 c_2 s_1 c_1 s_0

Fig. 20. Three binary numbers reduced to two numbers by a carry-save adder.

Carry-save numbers can be viewed as radix-2 numbers that use the digit set {0, 1, 2}. From this observation, it is easy to generalize to other redundant representations such as *signed-digit numbers*. For example, radix-8 numbers with the digit set [–5, 5] can be added without carry propagation chains; the carry only affects the next higher position and nothing else. Details are beyond the scope of this article.

The *logarithmic number system* (LNS) is an example of number representation for application-specific systems. In LNS, a value x is represented by its sign and the logarithm of its absolute value in some base. For example, if we use base-2 logarithms, with 4 whole and 4 fractional bits, the numbers 8 and 11 are represented as 0011**.**0000 and 0011**.**0111, respectively. The key advantage of LNS is that it allows us to multiply or divide numbers through addition or subtraction of their logarithms. For example, the product and ratio of 11 and 8 are found as follows:

$$\log_2 11 + \log_2 8 = (0011.0111)_{\text{two}} + (0011.0000)_{\text{two}} = (0110.0111)_{\text{two}}$$

$$\log_2 11 - \log_2 8 = (0011.0111)_{\text{two}} + (0011.0000)_{\text{two}} = (0000.0111)_{\text{two}}$$

Addition and subtraction, on the other hand, become somewhat more complicated but they are still manageable with help from lookup tables. Details are beyond the scope of this article. Suffice it to say that practical applications of this representation have thus far been limited to short word widths; however, available methods are improving and an LNS with range equivalent to the short ANSI/IEEE floating-point format has recently been implemented as part of a European microprocessor project.

The *residue number system* (RNS) has a long history dating back to the ancient Chinese. Use of this representation method in computer arithmetic was proposed in the late 1950s. Despite this long history, applications have remained limited due to the fact that RNS makes some key arithmetic operations, such as division and magnitude comparison, quite difficult. Its main advantages are simple addition and multiplication, thus making applications in which these two operations are predominant more suitable for RNS implementation. In RNS, each number is represented by a set of residues with respect to a set of pairwise relatively prime *moduli*. For example, given the moduli set $\{3, 5, 7\}$, the numbers 11 and 8 are represented by the set of their residues with respect to the moduli, thus leading to the codes (2, 1, 4) and (2, 3, 1), respectively. The number of distinct natural numbers that are represented is $3 \times 5 \times 7 = 105$. This set of values can be used to represent the natural numbers 0 to 104 or signed values –52 to +52.

In our example RNS with the moduli set $\{3, 5, 7\}$, adding or multiplying the numbers 11 and 8 is done by separately operating on the three residues, with each operation performed modulo the corresponding modulus. We thus get:

$$(2, 1, 4) + (2, 3, 1) = (4 \bmod 3, 4 \bmod 5, 5 \bmod 7) = (1, 4, 5)$$

$$(2, 1, 4) \times (2, 3, 1) = (4 \bmod 3, 3 \bmod 5, 4 \bmod 7) = (1, 3, 4)$$

Despite lack of widespread applications, both LNS and RNS remain important from a theoretical standpoint. Additionally, there are indications that with advances in arithmetic algorithms and VLSI technology, these two methods may find greater use in future (for information on applications of RNS arithmetic in signal processing, see *Residue Number System Arithmetic*, edited by M.A. Soderstrand, W.K. Jenkins, G.A. Jullien, and F.J. Taylor, IEEE Press, 1986).

XVII Research Directions

Computer arithmetic has played a central role in the development of digital computers. A.W. Burkes, H.H. Goldstine, and J. von Neumann, in their now classic 1946 report entitled “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” set the stage for many ingenious schemes to perform fast arithmetic on early digital computers, at a time when even the simplest circuits where bulky and expensive. After more than half a century, research and development is still continuing unabated, for even though many of the theoretical underpinnings of the field are now well understood, new application challenges must be faced and old solution schemes must be adapted to emerging technological constraints and opportunities.

Examples of active research issues in computer arithmetic at the turn of the twenty-first century include reducing power consumption, efficient handling of low-precision multimedia data, subnanosecond operations, area-efficient implementations, configurable processing elements, function evaluation with no error other than that dictated by the mandatory rounding, certifiable arithmetic, compatibility/portability of numerical software, applying novel computational paradigms, and fundamental theoretical limits. New results in the field appear in *Proceedings of the Symposia on Computer Arithmetic*, currently held in odd-numbered years, and in archival technical journals such as *IEEE Transactions on Computers*, which occasionally devotes entire special issues to the topic. Web resources can be accessed via the author's home page at http://www.ece.ucsb.edu.

Further Reading

[1] Flynn, M.J. and S.S. Oberman, *Advanced Computer Arithmetic Design*, Wiley, 2001. [Reference for advanced topics in arithmetic design and implementation.]

[2] Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 5-48, Mar. 1991.

[3] Knuth, D.E., *The Art of Computer Programming – Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 3rd Edition, 1997. [Authoritative reference work.]

[4] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000. [Basic textbook on computer arithmetic.]

[5] Swartzlander, E.E., Jr., *Computer Arithmetic,* Vols. I & II, IEEE Computer Society Press, 1990. [Compendium of key papers of historical or practical significance.]

Number Representation and Computer Arithmetic

Article to appear in *Encyclopedia of Information Systems*, Academic Press, 2001.

Behrooz Parhami
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106-9560

E-mail: parhami@ece.ucsb.edu

Web: http://www.ece.ucsb.edu/Faculty/Parhami/

List of terms specific to the topic

Array multiplier
Carry lookahead
Convergence method
Fast carry network
Fixed-point number
Floating-point number
Full-adder
High-radix arithmetic
Redundant number
Residue arithmetic
Ripple-carry adder
Rounding
Signed magnitude
Table-lookup scheme
Tree multiplier
Two’s complement