Representing Signed Numbers

■■■

"This can't be right . . . it goes into the red!"

LITTLE BOY, WHEN ASKED TO SUBTRACT 36 FROM 24 (CAPTION ON A CARTOON BY UNKNOWN ARTIST)

■■■

This chapter deals with the representation of signed fixed-point numbers by providing an attached sign bit, adding a fixed bias to all numbers, complementing negative values, attaching signs to digit positions, or using signed digits. In view of its importance in the design of fast arithmetic algorithms and hardware, representing signed fixed-point numbers by means of signed digits is further explored in Chapter 3. Chapter topics include:

2.1 SIGNED-MAGNITUDE REPRESENTATION

The natural numbers $0, 1, 2, \ldots, max$ can be represented as fixed-point numbers without fractional parts (refer to Section 1.4). In radix r, the number k of digits needed for representing the natural numbers up to max is

$$k = \lfloor \log_r max \rfloor + 1 = \lceil \log_r (max + 1) \rceil$$

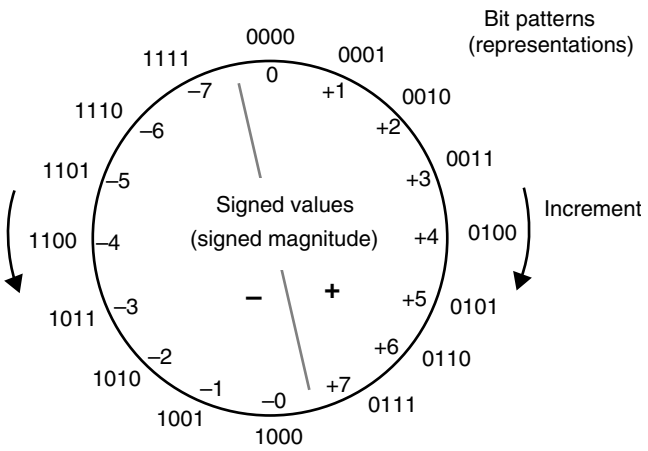

Figure 2.1 A 4-bit signed-magnitude number representation system for integers.

Conversely, with k digits, one can represent the values 0 through $r^k - 1$, inclusive; that is, the interval $[0, r^k - 1] = [0, r^k)$ of natural numbers.

Natural numbers are often referred to as "unsigned integers," which form a special data type in many programming languages and computer instruction sets. The advantage of using this data type as opposed to "integers" when the quantities of interest are known to be nonnegative is that a larger representation range can be obtained (e.g., maximum value of 255, rather than 127, with 8 bits).

One way to represent both positive and negative integers is to use "signed magnitudes," or the sign-and-magnitude format, in which 1 bit is devoted to sign. The common convention is to let 1 denote a negative sign and 0 a positive sign. In the case of radix-2 numbers with a total width of k bits, $k-1$ bits will be available to represent the magnitude or absolute value of the number. The range of k-bit signed-magnitude binary numbers is thus $[-(2^{k-1} - 1), 2^{k-1} - 1]$. Figure 2.1 depicts the assignment of values to bit patterns for a 4-bit signed-magnitude format.

Advantages of signed-magnitude representation include its intuitive appeal, conceptual simplicity, symmetric range, and simple negation (sign change) by flipping or inverting the sign bit. The primary disadvantage is that addition of numbers with unlike signs (subtraction) must be handled differently from that of same-sign operands.

The hardware implementation of an adder for signed-magnitude numbers either involves a magnitude comparator and a separate subtractor circuit or else is based on the use of complement representation (see Section 2.3) internally within the arithmetic/logic unit (ALU). In the latter approach, a negative operand is complemented at the ALU's input, the computation is done by means of complement representation, and the result is complemented, if necessary, to produce the signed-magnitude output. Because the pre- and postcomplementation steps add to the computation delay, it is better to use the complement representation throughout. This is exactly what modern computers do.

Besides the aforementioned extra delay in addition and subtraction, signed-magnitude representation allows two representations for 0, leading to the need for special

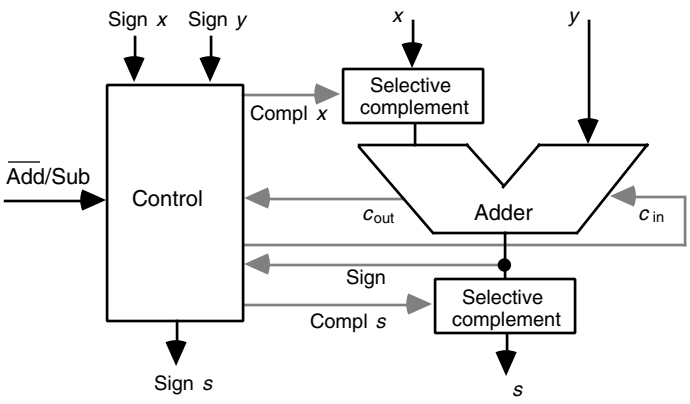

Figure 2.2 Adding signed-magnitude numbers using precomplementation and postcomplementation.

care in number comparisons or added overhead for detecting -0 and changing it to $+0$. This drawback, however, is unavoidable in any radix-2 number representation system with symmetric range.

Figure 2.2 shows the hardware implementation of signed-magnitude addition using selective pre- and postcomplementation. The control circuit receives as inputs the operation to be performed (0 = add, 1 = subtract), the signs of the two operands x and y, the carry-out of the adder, and the sign of the addition result. It produces signals for the adder's carry-in, complementation of x, complementation of the addition result, and the sign of the result. Note that complementation hardware is provided only for the x operand. This is because $x - y$ can be obtained by first computing $y - x$ and then changing the sign of the result. You will understand this design much better after we have covered complement representations of negative numbers in Sections 2.3 and 2.4.

2.2 BIASED REPRESENTATIONS

One way to deal with signed numbers is to devise a representation or coding scheme that converts signed numbers into unsigned numbers. For example, the biased representation is based on adding a positive value *bias* to all numbers, allowing us to represent the integers from *–bias* to *max – bias* using unsigned values from 0 to *max*. Such a representation is sometimes referred to as "excess-*bias*" (e.g., excess-3 or excess-128) coding. We will see in Chapter 17 that biased representation is used to encode the exponent part of a floating-point number.

Figure 2.3 shows how signed integers in the range $[-8, +7]$ can be encoded as unsigned values 0 through 15 by using a bias of 8. With k-bit representations and a bias of 2^{k-1}, the leftmost bit indicates the sign of the value represented (0 = negative, 1 = positive). Note that this is the opposite of the commonly used convention for number signs. With a bias of 2^{k-1} or $2^{k-1} - 1$, the range of represented integers is almost symmetric.

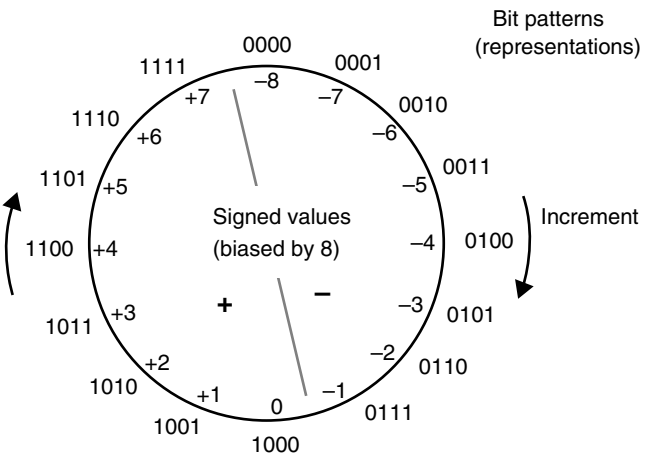

Figure 2.3 A 4-bit biased integer number representation system with a bias of 8.

Biased representation does not lend itself to simple arithmetic algorithms. Addition and subtraction become somewhat more complicated because one must subtract or add the bias from/to the result of a normal add/subtract operation, since

$$x + y + bias = (x + bias) + (y + bias) - bias$$
$$x - y + bias = (x + bias) - (y + bias) + bias$$

With k-bit numbers and a bias of 2^{k-1}, adding or subtracting the bias amounts to complementing the leftmost bit. Thus, the extra complexity in addition or subtraction is negligible.

Multiplication and division become significantly more difficult if these operations are to be performed directly on biased numbers. For this reason, the practical use of biased representation is limited to the exponent parts of floating-point numbers, which are never multiplied or divided.

2.3 COMPLEMENT REPRESENTATIONS

In a complement number representation system, a suitably large complementation constant M is selected and the negative value $-x$ is represented as the unsigned value $M - x$. Figure 2.4 depicts the encodings used for positive and negative values and the arbitrary boundary between the two regions.

To represent integers in the range $[-N, +P]$ unambiguously, the complementation constant M must satisfy $M \geq N + P + 1$. This is justified by noting that to prevent overlap between the representations of positive and negative values in Figure 2.4, we must have $M - N > P$. The choice of $M = N + P + 1$ yields maximum coding efficiency, since no code will go to waste.

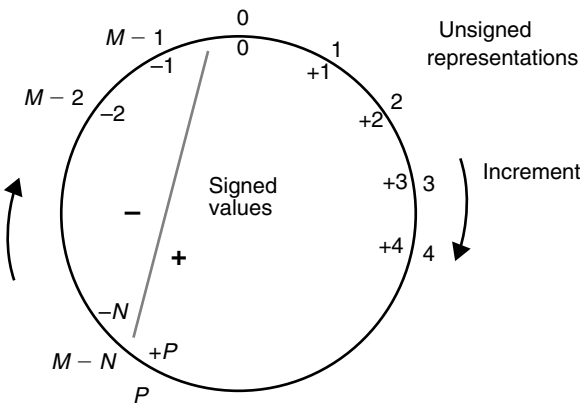

Figure 2.4 Complement representation of signed integers.

Table 2.1 Addition in a complement number system with the complementation constant M and range $[-N, +P]$

Desired operation	Computation to be performed mod M	Correct result with no overflow	Overflow condition
$(+x) + (+y)$	$x + y$	$x + y$	$x + y > P$
$(+x) + (-y)$	$x + (M - y)$	$x - y$ if $y \leq x$ $M - (y - x)$ if $y > x$	N/A
$(-x) + (+y)$	$(M - x) + y$	$y - x$ if $x \leq y$ $M - (x - y)$ if $x > y$	N/A
$(-x) + (-y)$	$(M - x) + (M - y)$	$M - (x + y)$	$x + y > N$

In a complement system with the complementation constant M and the number representation range $[-N, +P]$, addition is done by adding the respective unsigned representations (modulo M). The addition process is thus always the same, independent of the number signs. This is easily understood if we note that in modulo-M arithmetic adding $M - 1$ is the same as subtracting 1. Table 2.1 shows the addition rules for complement representations, along with conditions that lead to overflow.

Subtraction can be performed by complementing the subtrahend and then performing addition. Thus, assuming that a selective complementer is available, addition and subtraction become essentially the same operation, and this is the primary advantage of complement representations.

Complement representation can be used for fixed-point numbers that have a fractional part. The only difference is that consecutive values in the circular representation of Fig. 2.4 will be separated by *ulp* instead of by 1. As a decimal example, given the complementation constant $M = 12.000$ and a fixed-point number range of $[-6.000, +5.999]$, the fixed-point number -3.258 has the complement representation $12.000 - 3.258 = 8.742$.

We note that two auxiliary operations are required for complement representations to be effective: complementation or change of sign (computing $M - x$) and computations of residues mod M. If finding $M - x$ requires subtraction and finding residues mod M

implies division, then complement representation becomes quite inefficient. Thus M must be selected such that these two operations are simplified. Two choices allow just this for fixed-point radix-r arithmetic with k whole digits and l fractional digits:

Radix complement	$M = r^k$
Digit or diminished-radix complement	$M = r^k - ulp$

For radix-complement representations, modulo-M reduction is done by ignoring the carry-out from digit position $k-1$ in a $(k+l)$-digit radix-r addition. For digit-complement representations, computing the complement of x (i.e., $M-x$), is done by simply replacing each nonzero digit x_i by $r - 1 - x_i$. This is particularly easy if r is a power of 2. Complementation with $M = r^k$ and mod-M reduction with $M = r^k - ulp$ are similarly simple. You should be able to supply the details for radix r after reading Section 2.4, which deals with the important special case of $r = 2$.

2.4 2'S- AND 1'S-COMPLEMENT NUMBERS

In the special case of $r = 2$, the radix complement representation that corresponds to $M = 2^k$ is known as *2's complement*. Figure 2.5 shows the 4-bit, 2's-complement integer system ($k = 4, l = 0, M = 2^4 = 16$) and the meanings of the 16 representations allowed with 4 bits. The boundary between positive and negative values is drawn approximately in the middle to make the range roughly symmetric and to allow simple sign detection (the leftmost bit is the sign).

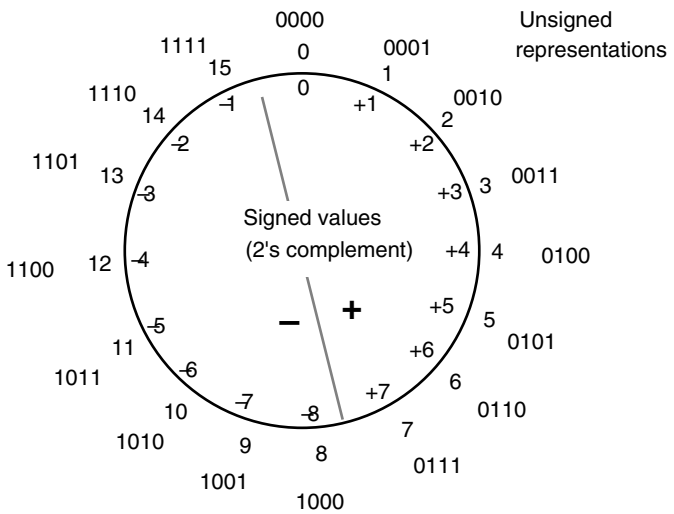

Figure 2.5 A 4-bit, 2's-complement number representation system for integers.

The 2's complement of a number x can be found via bitwise complementation of x and the addition of *ulp*:

$$2^k - x = [(2^k - ulp) - x] + ulp = x^{\text{compl}} + ulp$$

Note that the binary representation of $2^k - ulp$ consists of all 1s, making $(2^k - ulp) - x$ equivalent to the bitwise complement of x, denoted as x^{compl}. Whereas finding the bitwise complement of x is easy, adding *ulp* to the result is a slow process, since in the worst case it involves full carry propagation. We will see later how this addition of *ulp* can usually be avoided.

To add numbers modulo 2^k, we simply drop a carry-out of 1 produced by position $k - 1$. Since this carry is worth 2^k units, dropping it is equivalent to reducing the magnitude of the result by 2^k.

The range of representable numbers in a 2's-complement number system with k whole bits is

$$\text{from} \quad -2^{k-1} \quad \text{to} \quad 2^{k-1} - ulp$$

Because of this slightly asymmetric range, complementation can lead to overflow! Thus, if complementation is done as a separate sign change operation, it must include overflow detection. However, we will see later that complementation needed to convert subtraction into addition requires no special provision.

The name "2's complement" actually comes from the special case of $k = 1$ that leads to the complementation constant $M = 2$. In this case, represented numbers have 1 whole bit, which acts as the sign, and l fractional bits. Thus, fractional values in the range $[-1, 1 - ulp]$ are represented in such a fractional 2's-complement number system. Figure 2.5 can be readily modified to represent this number system by simply inserting a radix point after the leading digit for numbers outside the circle (turning them into 0.000, 0.001, and so on) and replacing each value x inside the circle with $x/8$ (0, 0.125, 0.25, and so on).

The digit or diminished-radix complement representation is known as *1's complement* in the special case of $r = 2$. The complementation constant in this case is $M = 2^k - ulp$. For example, Fig. 2.6 shows the 4-bit, 1's-complement integer system ($k = 4, l = 0, M = 2^4 - 1 = 15$) and the meanings of the 16 representations allowed with 4 bits. The boundary between positive and negative values is again drawn approximately in the middle to make the range symmetric and to allow simple sign detection (the leftmost bit is the sign).

Note that compared with the 2's-complement representation of Fig. 2.5, the representation for -8 has been eliminated and instead an alternate code has been assigned to 0 (technically, -0). This may somewhat complicate 0 detection in that both the all-0s and the all-1s patterns represent 0. The arithmetic circuits can be designed such that the all-1s pattern is detected and automatically converted to the all-0s pattern. Keeping -0 intact does not cause problems in computations, however, since all computations are modulo 15. For example, adding $+1$ (0001) to -0 (1111) will yield the correct result of $+1$ (0001) when the addition is done modulo 15.

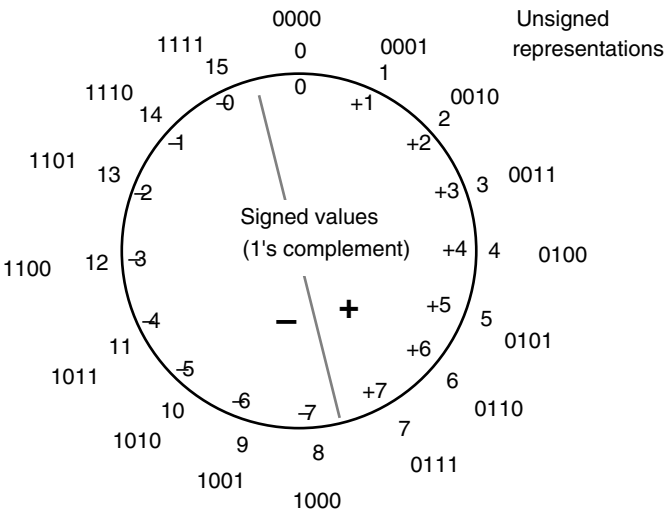

Figure 2.6 A 4-bit, 1's-complement number representation system for integers.

The 1's complement of a number x can be found by bitwise complementation:

$$(2^k - ulp) - x = x^{\text{compl}}$$

To add numbers modulo $2^k - ulp$, we simply drop a carry-out of 1 produced by position $k-1$ and simultaneously insert a carry-in of 1 into position $-l$. Since the dropped carry is worth 2^k units and the inserted carry is worth ulp, the combined effect is to reduce the magnitude of the result by $2^k - ulp$. In terms of hardware, the carry-out of our $(k+l)$-bit adder should be directly connected to its carry-in; this is known as *end-around carry*.

The foregoing scheme properly handles any sum that equals or exceeds 2^k. When the sum is $2^k - ulp$, however, the carry-out will be zero and modular reduction is not accomplished. As suggested earlier, such an all-1s result can be interpreted as an alternate representation of 0 that is either kept intact (making 0 detection more difficult) or is automatically converted by hardware to +0.

The range of representable numbers in a 1's-complement number system with k whole bits is

$$\text{from} \quad -(2^{k-1} - ulp) \quad \text{to} \quad 2^{k-1} - ulp$$

This symmetric range is one of the advantages of 1's-complement number representation.

Table 2.2 presents a brief comparison of radix- and digit-complement number representation systems for radix r. We might conclude from Table 2.2 that each of the two complement representation schemes has some advantages and disadvantages with respect to the other, making them equally desirable. However, since complementation is often performed for converting subtraction to addition, the addition of ulp required in the case of 2's-complement numbers can be accomplished by providing a carry-in of 1 into the least significant, or $(-l)$th, position of the adder. Figure 2.7 shows the required elements for a 2's-complement adder/subtractor. With the complementation disadvantage

Table 2.2 Comparing radix- and digit-complement number representation systems

Feature/Property	Radix complement	Digit complement
Symmetry ($P = N$?)	Possible for odd r (radices of practical interest are even)	Possible for even r
Unique zero?	Yes	No
Complementation	Complement all digits and add *ulp*	Complement all digits
Mod-M addition	Drop the carry-out	End-around carry

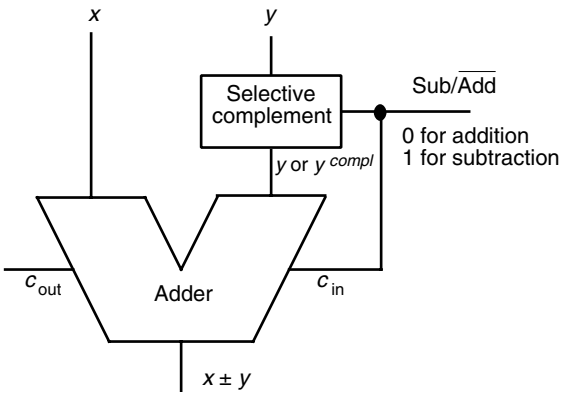

Figure 2.7 Adder/subtractor architecture for 2's-complement numbers.

mitigated in this way, 2's-complement representation has become the favored choice in virtually all modern digital systems.

Interestingly, the arrangement shown in Fig. 2.7 also removes the disadvantage of asymmetric range. If the operand y is -2^{k-1}, represented in 2's complement as 1 followed by all 0s, its complementation does not lead to overflow. This is because the 2's complement of y is essentially represented in two parts: y^{compl}, which represents $2^{k-1} - 1$, and c_{in} which represents 1.

Occasionally we need to extend the number of digits in an operand to make it of the same length as another operand. For example, if a 16-bit number is to be added to a 32-bit number, the former is first converted to 32-bit format, with the two 32-bit numbers then added using a 32-bit adder. Unsigned- or signed-magnitude fixed-point binary numbers can be extended from the left (whole part) or the right (fractional part) by simply padding them with 0s. This type of range or precision extension is only slightly more difficult for 2's- and 1's-complement numbers.

Given a 2's-complement number $x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l}$, extension can be achieved from the left by replicating the sign bit (*sign extension*) and from the right by padding it with 0s.

$$\cdots x_{k-1}x_{k-1}x_{k-1}x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l}000\cdots$$

To justify the foregoing rule, note that when the number of whole (fractional) digits is increased from k (l) to k' (l'), the complementation constant increases from $M = 2^k$ to

$M' = 2^{k'}$. Hence, the difference of the two complementation constants

$$M' - M = 2^{k'} - 2^k = 2^k(2^{k'-k} - 1)$$

must be added to the representation of any negative number. This difference is a binary integer consisting of $k' - k$ 1s followed by k 0s; hence the need for sign extension.

A 1's-complement number must be sign-extended from both ends:

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

Justifying the rule above for 1's-complement numbers is left as an exercise.

An unsigned binary number can be multiplied or divided by 2^h via an h-bit left or right shift, essentially changing the location of the radix point within the original digit-vector. To perform similar operations on 2's- and 1's-complement numbers, the operand must be first extended, so that the vacated positions on the right or left side of the fixed-width number after shifting receive the correct digit values. Put another way, in performing an h-bit right shift for dividing a number by 2^h, copies of the sign bit must be shifted in from the left. In the case of an h-bit left shift to multiply an operand by 2^h, we need to shift in the sign bit for 1's complement and 0s for 2's complement.

2.5 DIRECT AND INDIRECT SIGNED ARITHMETIC

In the preceding pages, we dealt with the addition and subtraction of signed numbers for a variety of number representation schemes (signed-magnitude, biased, complement). In all these cases, signed numbers were handled directly by the addition/subtraction hardware (*direct signed arithmetic*), consistent with our desire to avoid using separate addition and subtraction units.

For some arithmetic operations, it may be desirable to restrict the hardware to unsigned operands, thus necessitating *indirect signed arithmetic*. Basically, the operands are converted to unsigned values, a tentative result is obtained based on these unsigned values, and finally the necessary adjustments are made to find the result corresponding to the original signed operands. Figure 2.8 depicts the direct and indirect approaches to signed arithmetic.

Indirect signed arithmetic can be performed, for example, for multiplication or division of signed numbers, although we will see in Parts III and IV that direct algorithms are also available for this purpose. The process is trivial for signed-magnitude numbers. If x and y are biased numbers, then both the sign removal and adjustment steps involve addition/subtraction. If x and y are complement numbers, these steps involve selective complementation.

This type of preprocessing for operands, and postprocessing for computation results, is useful not only for dealing with signed values but also in the case of unacceptable or inconvenient operand values. For example, in computing $\sin x$, the operand can be brought to within $[0, \pi/2]$ by taking advantage of identities such as $\sin(-x) = -\sin x$ and $\sin(2\pi + x) = \sin(\pi - x) = \sin x$. Chapter 22 contains examples of such transformations.

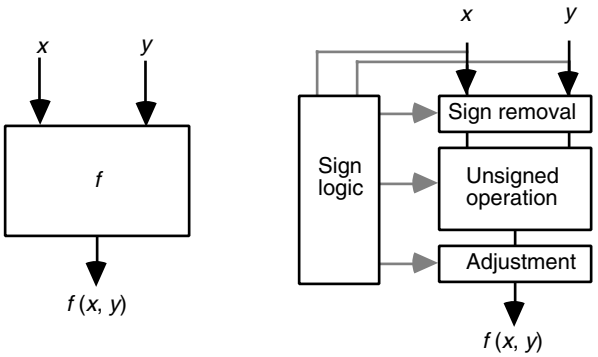

Figure 2.8 Direct versus indirect operation on signed numbers.

As a second example, some division algorithms become more efficient when the divisor is in a certain range (e.g., close to 1). In this case, the dividend and divisor can be scaled by the same factor in a preprocessing step to bring the divisor within the desired range (see Section 15.1).

2.6 USING SIGNED POSITIONS OR SIGNED DIGITS

The value of a 2's-complement number can be found by using the standard binary-to-decimal conversion process, except that the weight of the most significant bit (sign position) is taken to be negative. Figure 2.9 shows an example 8-bit, 2's-complement number converted to decimal by considering its sign bit to have the negative weight -2^7.

This very important property of 2's-complement systems is used to advantage in many algorithms that deal directly with signed numbers. The property is formally expressed as follows:

$$\begin{aligned} x &= (x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{2'\text{s}-\text{compl}} \\ &= -x_{k-1}2^{k-1} + \sum_{i=-l}^{k-2} x_i 2^i \end{aligned}$$

The proof is quite simple if we consider the two cases of $x_{k-1} = 0$ and $x_{k-1} = 1$ separately. For $x_{k-1} = 0$, we have

$$\begin{aligned} x &= (0x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{2'\text{s}-\text{compl}} \\ &= (0x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{\text{two}} \\ &= \sum_{i=-l}^{k-2} x_i\, 2^i \end{aligned}$$

$$
\begin{array}{llllllllll}
x = (& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0)_{\text{2's-compl}} \\
 & -2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\
 & -128 & + & 32 & & & + 4 & + 2 & = & -90
\end{array}
$$

Check:

$$
\begin{array}{llllllllll}
x = (& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0)_{\text{2's-compl}} \\
-x = (& 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0)_{\text{two}} \\
 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\
 & & 64 & + & 16 + 8 & & + & 2 & = & 90
\end{array}
$$

Figure 2.9 Interpreting a 2's-complement number as having a negatively weighted most significant digit.

For $x_{k-1} = 1$, we have

$$
\begin{aligned}
x &= (1x_{k-2} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-l})_{2'\text{s}-\text{compl}} \\
&= -[2^k - (1x_{k-2} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-l})_{\text{two}}] \\
&= -2^{k-1} + \sum_{i=-l}^{k-2} x_i \, 2^i
\end{aligned}
$$

Developing the corresponding interpretation for 1's-complement numbers is left as an exercise.

A simple generalization of the notion above immediately suggests itself [Kore81]. Let us assign negative weights to an arbitrary subset of the $k + l$ positions in a radix-r number and positive weights to the rest of the positions. A vector

$$\lambda = (\lambda_{k-1} \lambda_{k-2} \cdots \lambda_1 \lambda_0 . \lambda_{-1} \lambda_{-2} \cdots \lambda_{-l})$$

with elements λ_i in $\{-1, 1\}$, can be used to specify the signs associated with the various positions. With these conventions, the value represented by the digit vector x of length $k + l$ is

$$(x_{k-1} x_{k-2} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-l})_{r,\lambda} = \sum_{i=-l}^{k-1} \lambda_i x_i r^i$$

Note that the scheme above covers unsigned radix-r, 2's-complement, and negative-radix number systems as special cases:

$$
\begin{array}{llllllllll}
\lambda = & 1 & 1 & 1 & \cdots & 1 & 1 & 1 & 1 & \text{Positive radix} \\
\lambda = & -1 & 1 & 1 & \cdots & 1 & 1 & 1 & 1 & 2'\text{s complement} \\
\lambda = & & & & \cdots & -1 & 1 & -1 & 1 & \text{Negative radix}
\end{array}
$$

We can take one more step in the direction of generality and postulate that instead of a single sign vector λ being associated with the digit positions in the number system (i.e.,

with all numbers represented), a separate sign vector is defined for each number. Thus, the digits are viewed as having signed values:

$$x_i = \lambda_i |x_i|, \quad \text{with } \lambda_i \in \{-1, 1\}$$

Here, λ_i is the sign and $|x_i|$ is the magnitude of the ith digit. In fact once we begin to view the digits as signed values, there is no reason to limit ourselves to signed-magnitude representation of the digit values. Any type of coding, including biased or complement representation, can be used for the digits. Furthermore, the range of digit values need not be symmetric. We have already covered some examples of such signed-digit number systems in Section 1.4 (see Examples 1.1, 1.3, and 1.4).

Basically, any set $[-\alpha, \beta]$ of r or more consecutive integers that includes 0 can be used as the digit set for radix r. If exactly r digit values are used, then the number system is irredundant and offers a unique representation for each value within its range. On the other hand, if more than r digit values are used, $\rho = \alpha + \beta + 1 - r$ represents the *redundancy index* of the number system and some values will have multiple representations. In Chapter 3, we will see that such redundant representations can eliminate the propagation of carries in addition and thus allow us to implement truly parallel fast adders.

As an example of nonredundant signed-digit representations, consider a radix-4 number system with the digit set $[-1, 2]$. A k-digit number of this type can represent any integer from $-(4^k - 1)/3$ to $2(4^k - 1)/3$. Given a standard radix-4 integer using the digit set [0, 3], it can be converted to the preceding representation by simply rewriting each digit of 3 as $-1 + 4$, where the second term becomes a carry of 1 that propagates leftward. Figure 2.10 shows a numerical example. Note that the result may require $k + 1$ digits.

The conversion process of Fig. 2.10 stops when there remains no digit with value 3 that needs to be rewritten. The reverse conversion is similarly done by rewriting any digit of -1 as 3 with a borrow of 1 (carry of -1).

More generally, to convert between digit sets, each old digit value is rewritten as a valid new digit value and an appropriate transfer (carry or borrow) into the next

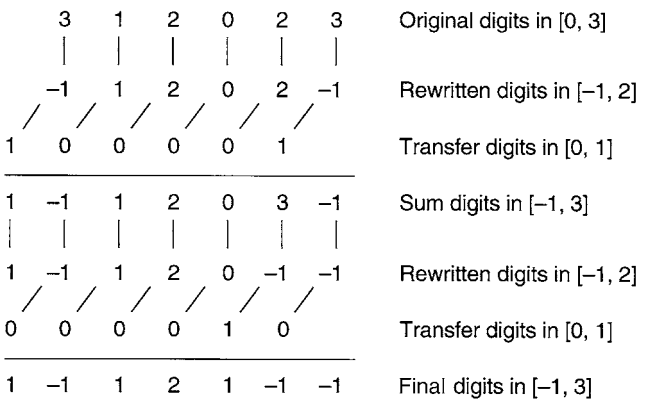

Figure 2.10 Converting a standard radix-4 integer to a radix-4 integer with the nonstandard digit set $[-1, 2]$.

higher digit position. Because these transfers can propagate, the conversion process is essentially a digit-serial one, beginning with the least-significant digit.

As an example of redundant signed-digit representations, consider a radix-4 number system with the digit set $[-2, 2]$. A k-digit number of this type can represent any integer from $-2(4^k - 1)/3$ to $2(4^k - 1)/3$. Given a standard radix-4 number using the digit set [0, 3], it can be converted to the preceding representation by simply rewriting each digit of 3 as $-1 + 4$ and each digit of 2 as $-2 + 4$, where the second term in each case becomes a carry of 1 that propagates leftward. Figure 2.11 shows a numerical example.

In this case, the transfers do not propagate, since each transfer of 1 can be absorbed by the next higher position that has a digit value in $[-2, 1]$, forming a final result digit in $[-2, 2]$. The conversion process from conventional radix-4 to the preceding redundant representation is thus carry-free. The reverse process, however, remains digit-serial.

We end this chapter by extending the dot notation of Section 1.6 to include negatively weighted bits, or negabits, which are represented as small hollow circles. Using this extended dot notation, positive-radix, 2's-complement, and negative-radix numbers, compared earlier in this section, can be represented graphically as in Fig. 2.12. Also, arithmetic algorithms on such numbers can be visualized for better understanding. For example, Fig. 2.13 depicts the operands, intermediate values, and final results when

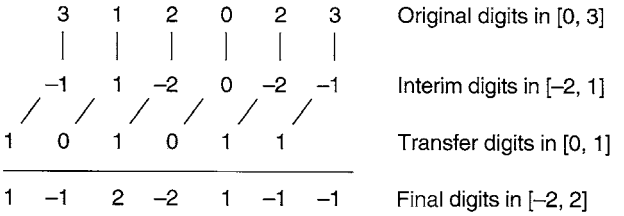

Figure 2.11 Converting a standard radix-4 integer to a radix-4 integer with the nonstandard digit set $[-2, 2]$.

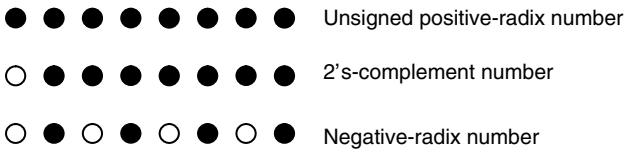

Figure 2.12 Extended dot notation depicting various number representation formats.

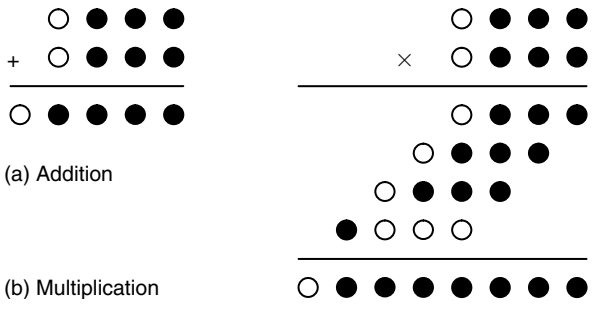

Figure 2.13 Example arithmetic algorithms represented in extended dot notation.

adding or multiplying 2's-complement numbers. As a case in point, Fig. 2.13b helps us understand that to multiply 2's-complement numbers, we need a process that allows us to add partial results containing a mix of posibits and negabits, in a way that yields a final result that includes only 1 negabit.

PROBLEMS

2.1 Signed-magnitude adder/subtractor

Design the control circuit of Fig. 2.2 so that signed-magnitude inputs are added correctly regardless of their signs. Include in your design a provision for overflow detection in the form of a fifth control circuit output.

2.2 Arithmetic on biased numbers

Multiplication of biased numbers can be done in a direct or an indirect way.

a. Develop a direct multiplication algorithm for biased numbers. *Hint:* Use the identity $xy + bias = (x + bias)(y + bias) - bias[(x + bias) + (y + bias) - bias] + bias$.
b. Present an indirect multiplication algorithm for biased numbers.
c. Compare the algorithms of parts a and b with respect to delay and hardware implementation cost.
d. Repeat the comparison for part c in the special case of squaring a biased number.

2.3 Representation formats and conversions

Consider the following five ways for representing integers in the range $[-127, 127]$ within an 8-bit format: (a) signed-magnitude, (b) 2's complement, (c) 1's complement, (d) excess-127 code (where an integer x is encoded using the binary representation of $x + 127$), (e) excess-128 code. Pick one of three more conventional and one of the two "excess" representations and describe conversion of numbers between the two formats in both directions.

2.4 Representation formats and conversions

a. Show conversion procedures from k-bit 2's-complement format to k-bit biased representation, with $bias = 2^{k-1}$, and vice versa. Pay attention to possible exceptions.
b. Repeat part a for $bias = 2^{k-1} - 1$.
c. Repeat part a for 1's-complement format.
d. Repeat part b for 1's-complement format.

2.5 Complement representation of negative numbers

Consider a k-bit integer radix-2 complement number representation system with the complementation constant $M = 2^k$. The range of integers represented is taken to be from $-N$ to $+P$, with $N + P + 1 = M$. Determine all possible pairs of values for N and P (in terms of M) if the sign of the number is to be determined by:

a. Looking at the most significant bit only.
b. Inspecting the three most significant bits.

c. A single 4-input OR or AND gate.
d. A single 4-input NOR or NAND gate.

2.6 Complement representation of negative numbers

Diminished radix complement was defined as being based on the complementation constant $r^k - ulp$. Study the implications of using an "augmented radix complement" system based on the complementation constant $r^k + ulp$.

2.7 1's- and 2's-complement number systems

We discussed the procedures for extending the number of whole or fractional digits in a 1's- or 2's-complement number in Section 2.4. Discuss procedures for the reverse process of shrinking the number of digits (e.g., converting 32-bit numbers to 16 bits).

2.8 Interpreting 1's-complement numbers

Prove that the value of the number $(x_{k-1}x_{k-2}\cdots x_1x_0.x_{-1}x_{-2}\cdots x_{-l})_{1's-compl}$ can be calculated from the formula $-x_{k-1}(2^{k-1} - ulp) + \sum_{i=-l}^{k-2} x_i 2^i$.

2.9 1's- and 2's-complement number systems

a. Prove that $x - y = (x^c + y)^c$, where the superscript "c" denotes any complementation scheme.
b. Find the difference between the two binary numbers 0010 and 0101 in two ways: First by adding the 2's complement of 0101 to 0010, and then by using the equality of part a, where "c" denotes bitwise complementation. Compare the two methods with regard to their possible advantages and drawbacks.

2.10 Shifting of signed numbers

Left/right shifting is used to double/halve the magnitude of unsigned binary integers.

a. How can we use shifting to accomplish the same for 1's- or 2's-complement numbers?
b. What is the procedure for doubling or halving a biased number?

2.11 Arithmetic on 1's-complement numbers

Discuss the effect of the end-around carry needed for 1's-complement addition on the worst-case carry propagation delay and the total addition time.

2.12 Range and precision extension for complement numbers

Prove that increasing the number of integer and fractional digits in 1's-complement representation requires sign extension from both ends (i.e., positive numbers are extended with 0s and negative numbers with 1s at both ends).

2.13 Signed digits or digit positions

a. Present an algorithm for determining the sign of a number represented in a positional system with signed positions.
b. Repeat part a for signed-digit representations.

2.14 Signed digit positions

Consider a positional radix-r integer number system with the associated position sign vector $\lambda = (\lambda_{k-1}\lambda_{k-2}\cdots\lambda_1\lambda_0), \lambda_i \in \{-1, 1\}$. The additive inverse of a number x is the number $-x$.

a. Find the additive inverse of the k-digit integer Q all of whose digits are $r - 1$.
b. Derive a procedure for finding the additive inverse of an arbitrary number x.
c. Specialize the algorithm of part b to the case of 2's-complement numbers.

2.15 Generalizing 2's complement: 2-adic numbers

Around the turn of the twentieth century, K. Hensel defined the class of p-adic numbers for a given prime p. Consider the class of 2-adic numbers with infinitely many digits to the left and a finite number of digits to the right of the binary point. An infinitely repeated pattern of digits is represented by writing down a single pattern (the period) within parentheses. Here are some example 2-adic representations using this notation:

$$\begin{array}{ll} 7 = (0)111. = \cdots 00000000111. & 1/7 = (110)111. = \cdots 110110110111. \\ -7 = (1)001. = \cdots 11111111001. & -1/7 = (001). = \cdots 001001001001. \\ 7/4 = (0)1.11 & 1/10 = (1100)110.1 \end{array}$$

We see that 7 and -7 have their standard 2's-complement forms, with infinitely many digits. The representations of 1/7 and $-1/7$, when multiplied by 7 and -7, respectively, using standard rules for multiplication, yield the representation of 1. Prove the following for 2-adic numbers:

a. Sign change of a 2-adic number is similar to 2's complementation.
b. The representation of a 2-adic number x is ultimately periodic if and only if x is rational.
c. The 2-adic representation of $-1/(2n + 1)$ for $n \geq 0$ is (σ), for some bit string σ, where the standard binary representation of $1/(2n + 1)$ is $(0.\sigma\sigma\sigma\cdots)_{\text{two}}$.

2.16 Biased-number representation

Consider radix-2 fractional numbers in the range $[-1, 1)$, represented with a bias of 1.

a. Develop an addition algorithm for such biased numbers.
b. Show that sign change for such numbers is identical to 2's complementation.
c. Use the results of parts a and b to design an adder/subtractor for such numbers.

2.17 Signed digits or digit positions

a. Present an algorithm for determining whether a number represented in a positional system with signed digit positions is 0. Note that such a number has fixed signs permanently associated with the different digit positions and is thus different from a signed-digit number.

b. Repeat part a for signed-digit representations.

2.18 2's-complement numbers

Consider a 2's-complement number representation system with the range $[-1, 1 - ulp]$ in which complementing the number -1 or performing the multiplication $(-1) \times (-1)$ leads to overflow. Can one change the range to $[-1 + ulp, 1]$, and would this solve the problems? [Swar07]

2.19 10's- and 9's-complement decimal representation

Discuss the 10's- and 9's-complement number systems that are the radix-10 counterparts to 2's- and 1's-complement representations. In particular, describe any changes that might be needed in arithmetic algorithms.

2.20 Extended dot notation

a. Show the subtraction of 2's-complement numbers in extended dot notation (see Fig. 2.13a).

b. Show the division of 2's-complement numbers in extended dot notation (see Fig. 2.13b).

REFERENCES AND FURTHER READINGS

[Aviz61] Avizienis, A., "Signed-Digit Number Representation for Fast Parallel Arithmetic," *IRE Trans. Electronic Computers*, Vol. 10, pp. 389–400, 1961.

[Gosl80] Gosling, J. B., *Design of Arithmetic Units for Digital Computers*, Macmillan, 1980.

[Knut97] Knuth, D. E., *The Art of Computer Programming*, 3rd ed., Vol. 2: *Seminumerical Algorithms*, Addison-Wesley, 1997.

[Kore81] Koren, I., and Y. Maliniak, "On Classes of Positive, Negative, and Imaginary Radix Number Systems," *IEEE Trans. Computers*, Vol. 30, No. 5, pp. 312–317, 1981.

[Korn94] Kornerup, P., "Digit-Set Conversions: Generalizations and Applications," *IEEE Trans. Computers*, Vol. 43, No. 8, pp. 622–629, 1994.

[Parh90] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 89–98, 1990.

[Parh98] Parhami, B., and S. Johansson, "A Number Representation Scheme with Carry-Free Rounding for Floating-Point Signal Processing Applications," *Proc. Int'l. Conf. Signal and Image Processing*, pp. 90–92, 1998.

[Scot85] Scott, N. R., *Computer Number Systems and Arithmetic*, Prentice-Hall, 1985.

[Swar07] Swartzlander, E. E. Jr., "The Negative Two's Complement Number System," *J. VLSI Signal Processing*, Vol. 49, No. 1, pp. 177–183, 2007.