

# **Binary Arithmetic**

by

***G. R. Dattatreya***

Copyright G. R. Dattatreya. All rights reserved.

## **I. Introduction**

Human beings generally use the decimal system to deal with numbers. In this system, the number of dots in the following diagram, Fig. 0, is represented as 23. The literal in each position of the representation, such as in 23, is known as a digit. There are ten digits, from 0 through 9. Larger numbers are represented by strings of digits. The value of such a string is given by a weighted sum. For integers, the rightmost position is called the least significant digit and has a weight of one. Successive positions towards the left side of the least significant position have weights increasing by a factor of ten. Arithmetic operations on larger numbers are performed by appropriate sequences of operations on smaller sets of digits. Results of operations on smaller sets of digits are often obtained from memorized tables. For example, we tend to instantaneously answer that three plus four is seven without mentally mimicking the use of an abacus. In a casual discussion, we are more inclined to agree that we memorize multiplication tables than we are, that even addition results are memorized. The decimal number system can also represent fractions through the use of fractional weights.

A dot or point placed in a string of digits specifies the separation between positions of integer and fractional weights. The fractional weights are a tenth, a hundredth, and so forth. Every natural number (positive integer) can be represented by a finite string of digits. However, not every fraction (the ratio of two natural numbers) can be represented by a terminating string of digits. For example,  $\frac{1}{7}=0.142857114285711428571\dots$ . Negative numbers are represented with a – sign as the leftmost symbol.

---



Fig. 0, The number decimal 23

---

The decimal system is not the only one to have the usefulness and advantages of a weighted number system. Indeed, it is possible to use any other natural number in place of ten in a weighted number system. The factor by which the weight increases from a position to its left neighboring position is called the base or the radix of the system. A binary number system uses 2 as the base. 2 is the smallest natural number that can be used as a radix. Each position in a binary natural number (nonnegative integer) has only two possible values, 0 or 1. Each position is called a bit. Furthermore, the sign of a number can also be represented using 0 or 1, since there are only two signs of numbers. The most important advantage of binary representation is that inexpensive physical devices are available to implement any function of such binary (or Boolean) variables. Therefore, computers use the binary number

systems.

The use of a bit to denote the sign of a number offers a few choices of binary number systems. The selection of a complete specification of a binary number system for hardware implementation is guided by the following criteria.

- (1) Simplicity of conversions. At the interface of a hardware system, data (input and output numbers) will have to be converted between the two representations, one inside the system and the other, in the environment.
- (2) Mathematical simplicity of arithmetic operations in the representation.
- (3) Complexity and cost of hardware implementation of arithmetic operations. A representation may be mathematically elegant and simple to understand. But its implementation in hardware may be more expensive than one in another representation. Hardware complexity and cost increases if rules for arithmetic operations have many special cases to deal with.

Within the class of weighted binary number systems, there are two popular choices. The two's complement and the one's complement system. The rest of this section introduces unsigned binary numbers. Section II develops the unified mathematical expression to represent signed integers and easily extends the same to fixed point fractions, in 2's complement notation. Section III presents the main ideas for arithmetic in 2's complement notation by logically deriving the rules for addition and related operations. Development of rules for multiplication and division by 2 are simple exercises carried out in Section IV. Section V deals with techniques to change the size of a number. The 1's complement arithmetic is similarly but concisely dealt with in Section VI. Section VII concludes the chapter.

### A. Unsigned Integers.

The unsigned number  $A$  with the representation  $a_{n-1}a_{n-2} \dots a_0$  has the weighted sum value given by

$$A = \sum_{i=0}^{n-1} a_i 2^i.$$

The contribution of a 1 in position  $i$  is  $2^i$ . Therefore,  $2^i$  is referred to as the weight of the position  $i$ . In a number, each position can take a value 0 or 1. In a variable string such as  $a_{n-1}a_{n-2} \dots a_0$ , each position is also called by different names such as a bit or a Boolean variable.

Example: The value of  $A = 110101$  is given by

$$A = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32 + 16 + 0 \cdot 8 + 4 + 0 \cdot 2 + 1 = 53.$$

It is common to use the same symbol, such as  $A$  in the above example, to denote a number, its binary string pattern, as well as its value. There is usually no cause for confusion.

Conversion from a decimal value to a binary string is not as simple. The procedure starts by dividing the value by 2. If the remainder is 0, the original number is even and therefore the coefficient  $a_0$  of the power  $2^0$  in the above weighted sum is 0. Else,  $a_0$  is 1. In either case, the quotient of the above division by 2 must correspond to a number whose binary representation is  $a_{n-1} \dots a_1$ . Thus the procedure can be repeated. In practice, this is usually done by successive division by 2 and collecting remainders with the remainder obtained first forming the  $a_0$  and the last remainder forming  $a_{n-1}$  for some  $n$  which is also determined during the successive division. The following is an illustration.

Example: Represent 265 as a binary number.

$$\begin{array}{r}
 265 \\
 \text{----} \\
 1 \mid 132 \\
 \text{----} \\
 0 \mid 66 \\
 \text{----} \\
 0 \mid 33 \\
 \text{----} \\
 1 \mid 16 \\
 \text{----} \\
 0 \mid 8 \\
 \text{----} \\
 0 \mid 4 \\
 \text{----} \\
 0 \mid 2 \\
 \text{----} \\
 0 \mid 1 \\
 \text{----} \\
 1 \mid 0
 \end{array}$$

Thus  $(265)_{10}=(100001001)_2$ . The parentheses with the subscript 10 is used to indicate that the number is in decimal notation and on the right hand side (RHS). We similarly have the subscript 2 for the binary number. Such an explicit indication is used only when necessary.

As noted earlier, the weight of a position  $i$  within the sequence of positions  $n-1, \dots, 0$  is  $2^i$ . The sum of weights of positions  $i-1, \dots, 0$  is  $2^i-1$ . Therefore, the value  $2^i$  that can be represented by a bit 1 in position  $i$ , *cannot* be represented by any other combination of bits in positions  $i-1, \dots, 0$ . The weight of any position  $j > i$  is larger than  $2^i$ . Therefore, the possible contribution of  $2^i$  *cannot* be represented by any other combination of bits. This shows that the representation is unique. That is, each number has a unique bit pattern associated with it. The largest integer that can be represented by  $a_{n-1}\dots a_0$  is  $2^n - 1$ .

*B. Unsigned Fractions.*

Representation of pure unsigned fractions (numbers between 0 and 1) is a simple extension of that for unsigned integers, as follows. The value of

$$B = 0.b_1b_2 \dots b_m \text{ is}$$

$$B = \sum_{j=1}^m b_j 2^{-j}$$

As in the case of unsigned integers, the weights increase from right to left in multiples of 2. Conversion of a binary fraction to its decimal value is realized by evaluating the weighted sum. To represent a decimal fraction as a binary number, we first multiply the given fraction by 2. If the result is 1 or larger, the original value of the fraction is  $2^{-1}$  or larger, giving us  $b_1=1$ . The procedure is repeated on successive fractional remainders.

Example: Represent 0.6875 in binary notation.

$$\begin{array}{r}
 .6875 * 2 \\
 \hline
 1.375 * 2 \quad \text{Remove the leftmost 1 for this multiplication.} \\
 \hline
 0.75 * 2 \\
 \hline
 1.5 * 2 \quad \text{Remove the leftmost 1 for this multiplication.} \\
 \hline
 1.0
 \end{array}$$

The resulting binary number is 0.1011.

A binary fraction may require a recurring bit pattern, just as  $\frac{2}{3}$  is represented by 0.6666... in decimal notation. For example,  $(0.2)_{10} = 0.001100110011\dots$ .

An unsigned binary number with a fractional part and an integer part can also be evaluated by the weighted sum technique, as follows

$$A = a_{n-1}a_{n-2}\cdots a_m.a_{m-1}\cdots a_0$$

$$= [\sum_{i=0}^n a_i 2^i] 2^{-m}$$

That is, the value of the fractional number is the value of the integer obtained by ignoring the “binary point” and multiplying the integer by  $2^{-m}$  where  $m$  is the number of positions on the right side of the binary point. In this representation, the rightmost bit is known as the least significant bit (lsb) and the leftmost bit is known as the most significant bit, due to the corresponding values of the weights they carry. The difference between successive values of representable numbers is the weight of the lsb. It is often referred to as the grain size of the representation. The largest value that can be represented by  $a_{n-1}a_{n-2}\cdots a_m.a_{m-1}\cdots a_0$  is  $(2^n - 1)2^{-m}$ . An unsigned decimal number with an integer part and a fractional part can be converted to an equivalent binary number by converting the two parts separately.

## II. 2’s Complement Notation for Fixed Point Fractions

Consider representing a signed integer  $A$  in an  $n+1$  location (see Fig. 1) binary word  $a_n \cdots a_0$ . Each  $a_i$  is a Boolean variable (also called a bit) and takes a value 0 or 1. The

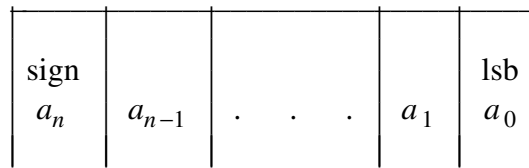


Fig. 1, The  $n+1$  bit binary word

---

leftmost bit,  $a_n$ , is called the sign bit; it is 0 if the number being represented is positive and 1, if negative. Depending on the details of representation,  $A = 0$  can be viewed as a positive or a negative number. It turns out that in the 2's complement notation being developed, the number zero has a unique representation and is identified as positive. In the 1's complement notation discussed later, there are two distinct representations for the zero number, *i. e.*, a *positive zero* and a *negative zero*. The rightmost bit is known as the least significant bit (lsb) for reasons to be evident shortly.

If  $A$  is positive,  $a_n$  will be set to 0 as mentioned earlier. The remaining bits  $a_{n-1}, \dots, a_0$  will be uniquely determined such that

$$A = \sum_{i=0}^{n-1} 2^i a_i. \quad (1)$$

Given  $A$ , the bits  $a_0, \dots, a_n$  are obtained through the usual successive division by 2 and collecting remainders. Obviously, the range of positive integers that may be so represented is from  $A = 0$  resulting in each of the bits  $a_{n-1}, \dots, a_0$  being 0, through  $A = 2^n - 1$  resulting in each bit being 1. Any larger number requires more than  $n$  bits (note that  $a_n$  must be 0 for positive numbers). To represent a negative number,  $A$ , we let  $a_n = 1$  and fill in the bits  $a_{n-1}, \dots, a_0$  by those of the positive number  $(2^n + A)$  as obtained by

$$2^n + A = \sum_{i=0}^{n-1} 2^i a_i \quad \text{or} \quad (2)$$

$$A = -2^n + \sum_{i=0}^{n-1} 2^i a_i. \quad (3)$$

The sign bit  $a_n$  is 1 since the number is negative. Therefore, we can equivalently write the

expression for the negative number as

$$A = -2^n a_n + \sum_{i=0}^{n-1} 2^i a_i. \quad (4)$$

In the case of positive numbers, since  $a_n = 0$ , using the additional term  $-2^n a_n$  in (4) will not change the representation specified in (1). Thus, (4) is a unique representation for both positive and negative integers. Of course the representation of negative numbers requires that  $(2^n + A)$  be positive and less than  $2^n$  so as to be representable with the  $n$  bits  $a_{n-1}, \dots, a_0$ . This determines the range of negative integers to be from  $-1$  through  $-2^n$ , inclusive of both. Together with the  $2^n$  positive numbers (0 through  $2^n-1$ ), the range of numbers use all the  $2^{n+1}$  combinations available with the  $n+1$  bits. Thus every value for  $A$  has a unique sequence of bits  $a_n, \dots, a_0$  associated with it and vice versa. Table I lists the representations of some interesting choices for  $A$ .

The expression in (4) is a weighted sum representation. The weights of locations increase by a factor of 2 from the weight 1 for  $a_0$  through the weight  $2^{n-1}$  for  $a_{n-1}$ . The weight of the sign bit is  $-2^n$ . The contribution of  $a_0$  to the value of  $A$  is  $a_0$  and has the least absolute weight of all the bits. Hence the name lsb. The above representation is easily extended to fractional numbers by noting that integers merely have zero fractional part. Thus, in Fig. 1, the binary point separating the integer and the fractional part is to the right of the lsb  $a_0$  and is not useful since we did not permit nonzero fractional part. We can exercise a convention that if we move the binary point  $m$  locations to the left, *i. e.*, to be between  $a_m$  and  $a_{m-1}$ , the weight of every location gets multiplied by a factor of  $2^{-m}$ . Applying this procedure to the representation in Fig. 1 and equation (4), we obtain the 2's complement representation of

Table I

Sample Representations

A	sign							lsb
	$a_n$	$a_{n-1}$	$a_{n-2}$	.	.	.	$a_1$	$a_0$
$-2^n$	1	0	0	.	.	.	0	0
$-(2^n - 1)$	1	0	0	.	.	.	0	1
$-2^{n-1}$	1	1	0	.	.	.	0	0
-1	1	1	1	.	.	.	1	1
0	0	0	0	.	.	.	0	0
1	0	0	0	.	.	.	0	1
$2^{n-1}$	0	1	0	.	.	.	0	0
$2^n - 1$	0	1	1	.	.	.	1	1

fixed point fractional numbers as

$$A = (-2^n a_n + \sum_{i=0}^{n-1} 2^i a_i) 2^{-m} \quad (5)$$

depicted in Fig. 2. The binary point is denoted by the symbol • in Fig. 2.

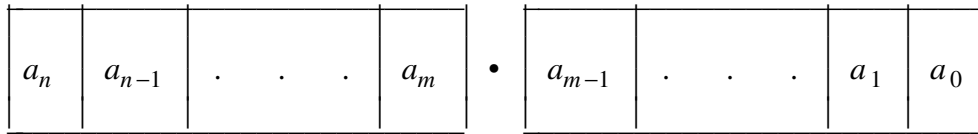


Fig. 2, 2's complement representation of fractional numbers

---

Examples:

$$(1) \quad 110.1011 = 2^0 + 2^1 + 2^3 + 2^5 + (-2^6)]2^{-4} = (1 + 2 + 8 + 32 - 64)0.0625 = -1.3125$$

$$(2) \quad 01101.0101 = 1 + 4 + 8 + 0.25 + 0.0625 = 13.3125$$

In the first example above, the entire bit pattern is first treated as an integer and then multiplied by  $2^{-4}$ . In the second, the correct weights are used in one step.

It is important to note that the bit pattern returns different values based on where the binary point is. There will be no hardware structure inside a digital system to detect the location of the binary point. Within a subsystem, the location of the binary point is predetermined by convention and does not change over a period of time. We need the sign bit to be the left most bit. Hence we can opt to have the binary point anywhere from the position just left of the sign bit ( $m = n + 1$ ) to the position just right of the lsb ( $m = 0$ ). The latter results in pure integer representations and the former represents fractional numbers from  $-2^{-1}$  through  $[2^{-1} - 2^{-(n+1)}]$  in increments of  $2^{-(n+1)}$ . A popular choice is  $m = n$  and results in fractions from  $-1$  through  $(1 - 2^{-n})$  in increments of  $2^{-n}$ .

A brief mention of the procedure to obtain the bit pattern of an unsigned integer is made following (1). Given a positive  $A$ , its fractional part is converted to the 2's complement fractional part through the usual successive multiplication by 2 of remaining fractional parts and collecting the integers resulting from multiplications. If  $A$  is negative, the 2's complement representation of  $|A|$  is first obtained and it is negated as per rules developed in the next section. Given the 2's complement representation of a number, its value in the decimal notation is easily obtained by evaluating the weighted sum (5).

### III. Addition and Related Operations in 2's Complement Notation

This section develops the rules for negation, addition, subtraction, and overflow detection. In the remaining part of the chapter, arithmetic operations are required to be performed on the bit patterns of numbers in the specified notation. The result is also required to be in the same notation (except possibly with different number of bits, as in the case of change of size).

#### A. Negation.

The problem of negation is to determine the bit pattern of the number  $-A$  starting from the bit pattern of  $A$ . We will assume that  $m = 0$ , *i. e.*, that the numbers are integers. Extension to fractional numbers is mentioned at the end of Section III. Fractional numbers do not require different rules. Let  $B = -A$  with the required bit pattern  $b_n \cdots b_0$ . Then, since  $A + B = 0$ , we have

$$-2^n(a_n + b_n) + \sum_{i=0}^{n-1} (a_i + b_i)2^i = 0. \quad (6)$$

If  $A = 0$ , there is no way the (nonzero) positive and negative contributions from  $b_{n-1}, \dots, b_0$  and  $b_n$ , respectively, cancel each other since the bit  $b_n$  contributes 0 or  $-2^n$  (depending on whether it is 0 or 1) and the remaining bits can contribute 0 through at most  $2^n - 1$ . The only possibility is that all the bits of  $B$  are 0. Another way to draw the same conclusion is to note the earlier statement that every number has a unique representation and hence the bit pattern of  $B = -A$  should be the same as the bit pattern of  $A$ , if  $A = 0$ .

If  $A \neq 0$ , either  $a_n$  or  $b_n$  but not both must be 1 and (6) reduces to

$$\sum_{i=0}^{n-1} (a_i + b_i)2^i = 2^n. \quad (7)$$

Thus the bits  $b_{n-1}, \dots, b_0$  can be obtained by subtracting the smaller unsigned (positive) binary number formed by  $a_{n-1} \dots a_0$  from the bit pattern of the larger unsigned  $2^n$ . Instead of dealing with the sign bit  $b_n$  separately, we can treat  $a_n \dots a_0$  as an unsigned (positive) bit pattern and subtract it from the larger  $2^{n+1}$ . The borrows required to subtract the bit pattern (of the nonzero  $A$ ) ensure that  $b_n$  obtained by such a subtraction is the required opposite of  $a_n$  (called the Boolean complement of  $a_n$  and denoted by  $a_n^-$ ). Finally, note that this procedure is effective even if  $A = 0$ , provided we discard the 1 resulting at a position left of  $b_n$  during the subtraction of zero from  $2^{n+1}$ . Thus, the first of the three procedures discussed in this chapter for negation in 2's complement notation is stated as follows. "Treat the bit pattern  $a_n \dots a_0$  as an unsigned (positive) number and subtract it from the unsigned bit pattern for  $2^{n+1}$ . The  $(n + 1)$  lower significant bits resulting form  $b_n \dots b_0$ , the bit pattern for

$$B = -A.$$

Example: The negative of  $1011101 = -35$  is obtained as

$$\begin{array}{r} 10000000 \\ -1011101 \\ \hline 0100011 \end{array}$$

which evaluates to the required 35.

The same result can be obtained by subtracting from  $(2^{n+1} - 1)$  and then adding 1 to the result of subtraction. Subtraction of the unsigned number  $a_n \cdots a_0$  from  $(2^{n+1} - 1)$  is equivalent to obtaining the bit by bit Boolean complement of  $a_n \cdots a_0$ . Thus, the second procedure for negation follows. “Boolean complement each bit in  $a_n \cdots a_0$ . Add 1 to the resulting bit pattern, viewed as an unsigned number.” The number  $A = -2^n$  poses problems for negation;  $-A = 2^n$  is not within the range of numbers representable in the notation. Such a condition, the result of an operation being outside the representable range, is known as overflow. Any operation with a potential for overflow must be accompanied by a test for overflow. In the case of negation, addition of 1 to the unsigned bit pattern  $a_n^- \cdots a_0^-$  results in a carry  $C_n$  (carry into the sign location during addition) being 1, only if  $a_{n-1}, \dots, a_0$  are all 0s (read as zeros; similarly we use 1s for several 1 bits). Within this case,  $a_n = 0$  produces  $C_{n+1} = 1$  (carry out of sign location) and  $a_n = 1$  produces  $C_{n+1} = 0$ . The latter is the overflow situation which can be detected by checking if  $C_n$  and  $C_{n+1}$  are the same. If they are not, overflow is confirmed. If they are equal, there is no overflow. Note that in every other case (except  $A = 0$ ),  $C_n = C_{n+1} = 0$ . If  $A = 0$ ,  $C_{n+1} = C_n = 1$  occurs. We will find this test of comparing  $C_{n+1}$  and  $C_n$  for overflow to be useful for addition and subtraction as well, in the

next subsection.

Examples:

- (1) The negative of  $0010111 = 23$  is obtained as

$$\begin{array}{r} 1101000 \\ +1 \\ \hline 1101001 \end{array}$$

which evaluates to  $41 - 64 = -23$ .

- (2) Using this approach, the negative of  $1000 = -8$  would be

$$\begin{array}{r} 0111 \\ +1 \\ \hline 1000 \end{array}$$

which is clearly erroneous. However, in the above addition, the Carrys into and out of the sign position are different signifying overflow.

A third procedure for negation results from the observation that during the addition of 1 (after bit by bit complementation), any uninterrupted sequence of 1s from the right side end, followed by the first 0 get complemented back to the sequence of 0s followed by a 1, as they are in the bit pattern  $a_n \cdots a_0$ . The concise statement of the third procedure: “Examine the bits of  $a_n \cdots a_0$  from the lsb  $a_0$ . Leave the first uninterrupted sequence of 0s (from and including the lsb), followed by the first 1, unchanged. Boolean complement the remaining bits. If the first 1 detected happens to be the leftmost (sign) bit, declare overflow.” Among the three negation procedures, the second procedure is most commonly used in digital systems. The third procedure is very suitable for hand calculations.

*B. Addition.*

The problem of addition is to obtain the bit pattern  $s_n \cdots s_0$  of the number  $S$  corresponding to the sum of two numbers  $A$  and  $B$ , starting from  $a_n \cdots a_0$  and  $b_n \cdots b_0$ . Also, we need to check for overflow. As in the case of negation, we assume the numbers to be integers, for simplicity of arguments. The value of the sum is

$$S = -2^n (a_n + b_n) + \sum_{i=0}^{n-1} (a_i + b_i) 2^i. \quad (8)$$

Starting from the lsb location, and upto the location  $n-1$ , addition is carried out as in the case of unsigned numbers, because the weights of the carry from a position is the weight of the next position in the left direction (higher significant position).  $C_{i+1}$  is the carry resulting from the addition of  $a_i$ ,  $b_i$ , and  $C_i$ ;  $i = 1, \dots, n$ .  $C_0$  is a possible carry input to the addition at the lsb location.  $C_n$  is the carry into the sign location, and  $C_{n+1}$ , carry out of the sign location (as in the problem of negation). The weight of  $C_n$  is  $2^n$ . But the weight of the next position in the sum,  $s_n$ , is  $-2^n$ . Therefore, whereas  $s_0$  through  $s_{n-1}$  are obtained by

$$s_i = a_i + b_i + C_i, \quad (9)$$

the contribution of  $s_n$  to the number  $S$  is  $-2^n a_n - 2^n b_n + 2^n C_n$  and

$$s_n = a_n + b_n - C_n. \quad (10)$$

Condition for overflow is easily determined to be the possibilities for  $a_n$ ,  $b_n$ , and  $C_n$  that result in  $s_n$  not being 0 or 1. These slightly different procedures for addition of the two parts of the numbers (sign and the remaining part) are inconvenient. Fortunately, it turns out that  $a_n + b_n + C_n$  results in two bits  $C_{n+1}$  (carry out of sign position) and  $r_n$  such that  $r_n = s_n$

for all cases of no overflow. Further,  $C_{n+1} = C_n$  for all cases of no overflow and  $C_{n+1} \neq C_n$  for all cases of overflow. Table II lists the consequences of the two candidate procedures for addition at position  $n$  for all possible values of  $a_n$ ,  $b_n$ , and  $C_n$ . Thus, the effective procedure for addition in 2's complement notation is as follows. "Add the corresponding pairs of bits and carries from lower significant positions as in the case of unsigned numbers. If  $C_{n+1} = C_n$ , the result is the correct sum. If not, declare overflow."

Table II

Comparison of the two procedures for addition at position  $n$

$a_n$	$b_n$	$C_n$	required $s_n = a_n + b_n - C_n$	overflow?	alternative procedure		comparison	
					$C_{n+1}$	$r_n$	$r_n = s_n?$	$C_{n+1} \neq C_n?$
0	0	0	0	no	0	0	yes	no
0	0	1	-1	yes	0	1	no	yes
0	1	0	1	no	0	1	yes	no
0	1	1	0	no	1	0	yes	no
1	0	0	1	no	0	1	yes	no
1	0	1	0	no	1	0	yes	no
1	1	0	2	yes	1	0	no	yes
1	1	1	1	no	1	1	yes	no

Examples:

(1) Add:

$$01010 \equiv 10$$

$$\begin{array}{r}
 +11111 \quad \equiv \quad -1 \\
 \hline
 01001 \quad \equiv \quad 9
 \end{array}$$

$C_4 = C_5 = 1$ ; therefore, there is no overflow.

(2) Add:

$$\begin{array}{r}
 101.110 \quad \equiv \quad -2.25 \\
 +110.101 \quad \equiv \quad -1.375 \\
 \hline
 100.011 \quad \equiv \quad -3.625
 \end{array}$$

$C_5 = C_6 = 1$ ; no overflow.

(3) Add:

$$\begin{array}{r}
 01001.1 \quad \equiv \quad 9.5 \\
 +00011.0 \quad \equiv \quad 2.0 \\
 \hline
 01100.1 \quad \equiv \quad 12.5
 \end{array}$$

$C_5 = C_6 = 0$ ; no overflow.

(4) Add:

$$\begin{array}{r}
 01101.1 \quad \equiv \quad 13.5 \\
 +00111.0 \quad \equiv \quad 7.0 \\
 \hline
 10100.1 \quad \equiv \quad -11.5
 \end{array}$$

$C_5 = 1$  and  $C_6 = 0$  signifying overflow. Therefore, the result 10101.1 is erroneous and should be discarded.

(5) Add:

$$\begin{array}{r}
 1001.1 \quad \equiv \quad -6.5 \\
 +1011.1 \quad \equiv \quad -4.5
 \end{array}$$

$$\begin{array}{r} \text{-----} \\ 0101.0 \end{array} \quad \equiv \quad 5.0$$

$C_4 = 1$  and  $C_5 = 0$  signifying overflow.

### C. Subtraction.

The result of  $A - B$ , starting from  $A$  and  $B$ , is easily obtained by first negating  $B$  and then adding the result of the negation to  $A$ . The second procedure for negation (discussed in Section II A) is ideally suited for the purpose; the required addition of 1 to the bit by bit complements of the bits of  $B$  can be combined with the addition of  $A$  by feeding a 1 at  $C_0$ . Overflow detection procedure is the same as for addition, except possibly for the case of  $B = -2^n$  ( $A$  and  $B$  are integers, *i. e.*,  $m = 0$ ). Fortunately, even in this case ( $B = -2^n$ ), if  $A$  is positive (or zero), the subtraction procedure produces  $C_n = 1$  (since the complemented bits, *i. e.*,  $b_{n-1}^-, \dots, b_0^-$  are all 1s and  $C_0 = 1$ ) and  $C_{n+1} = 0$  (since  $a_n = 0$  due to  $A \geq 0$ ,  $b_n^- = 0$ , and  $C_n = 1$ ) signifying the required overflow condition. If  $B = -2^n$  and  $A < 0$ ,  $A - B$  is representable. In this case,  $C_n = 1$  as earlier and  $C_{n+1} = 1$  (since  $a_n = 1$  due to  $A < 0$ ,  $b_n^- = 0$ , and  $C_n = 1$ ) signifying no overflow, as we would like. Thus the subtraction procedure is effective for all cases.

Examples:

(1)  $0110.1 - 1100.0$ :

0110.1

$$\begin{array}{r}
 +0011.1 \\
 +.1 \\
 \hline
 1010.1
 \end{array}$$

$C_4 = 1$  and  $C_5 = 0$  signifying overflow.

(2)  $1101.1 - 1110.1$ :

$$\begin{array}{r}
 1101.1 \\
 +0001.0 \\
 +.1 \\
 \hline
 1110.1
 \end{array}$$

$C_4 = C_5 = 0$ ; no overflow.

As a final remark for this section, if  $A$  and  $B$  are fractional numbers with the same  $m$ , the arguments are identical, except that all the weights are multiplied by  $2^{-m}$ . The values of numbers and portions of numbers also change accordingly. The final rules for negation, addition, and subtraction are unaffected. The examples above illustrate the same.

#### IV. Multiplication and Division by 2 in 2's Complement Notation

##### A. Multiplication.

Let  $*$  denote multiplication. The value of

$$B = 2*A = A + A \tag{11}$$

Adding  $a_0$  with itself results in 0 for the  $b_0$ , the lsb of  $2*A$ . The carry from the lsb,  $C_1$  is 0 if  $a_0$  is 0 and 1 if  $a_0$  is 1. That is,

$$C_0 = a_0 \quad (12)$$

Similarly, at position  $i$ ,

$$a_i + a_i + C_i = 2*a_i + C_i$$

which produces a two bit output with as

$$b_i = C_i \text{ and} \quad (13)$$

$$C_{i+1} = a_i. \quad (14)$$

This in turn reduces to

$$b_i = C_i = a_{i-1}. \quad (15)$$

Continuing further,  $C_n = a_{n-1}$  and  $C_{n+1} = a_n$  for use in overflow detection. The consolidated rule for multiplication by 2 follows. ‘‘If the two leftmost bits are identical, shift the bits once left. Reset the lsb to 0. If not, declare overflow.’’

Example:  $A = 1101.10 = -2.5$ ;  $2*A = 1011.00 = -5.0$ .

### B. Division.

Starting from (5),

$$\frac{A}{2} = (-2^{n-1}a_n + \sum_{i=1}^{n-1} 2^{i-1}a_i)2^{-m} + 2^{-m-1}a_0. \quad (16)$$

First of all, we need to discard the contribution of  $a_0$ , since  $2^{-m-1}$  falls between two successive numbers representable in the notation. For the other part,  $-2^{n-1}a_n = -2^n a_n + 2^{n-1}a_n$  and

$$\frac{A}{2} = (-2^n a_n + 2^{n-1}a_n + \sum_{i=1}^{n-1} 2^{i-1}a_i)2^{-m} \quad (17)$$

$$= (-2^n a_n + \sum_{i=0}^{n-1} 2^i a_{i+1})2^{-m}. \quad (18)$$

The rule for division follows. “Shift the bits right once. Retain the sign bit at its location (in addition to duplicating it at its next right location). Discard the lsb that has no position to occupy when right shifted.” An important conclusion often overlooked by students is that any fraction discarded is always positive (or zero). So, if a negative number is divided by 2, the result may be more negative than it should be, as opposed to being closer to zero than it should be. Sometimes, dividing  $-2^{-m}$  by 2 resulting in zero value is considered to be a special condition termed underflow.

Example:  $A = 1011.11 = -4.25$ .  $\frac{A}{2} = 1101.11 = -2.25$  which is really half of  $-4.5$ . The discrepancy is due to the effect of the value of the original lsb, 0.25, being lost.

## V. Change of Size

It is interesting to study ways of changing the number of bits in a 2's complement representation, without changing its value. We can append positions to the right of the lsb and fill them with 0s. If the original representation is for pure integers, an explicit binary point must be used at a place just right of the original lsb. The new representation is

$$A = (-2^n a_n + \sum_{i=k}^{n-1} 2^i a_i)2^{-m}. \quad (19)$$

$k$  can be any negative integer and  $a_k$  must be set to 0 for all  $k < 0$ . This results in an increase in the number of bits. On the other hand, if  $a_i = 0$ ,  $i = 0, \dots, (k-1)$  and  $(1 \leq k \leq m)$ , the positions  $a_0, \dots, a_{k-1}$  can be eliminated to reduce the size of the number.

At the left side, if the sign bit replicates itself in an uninterrupted sequence (all on the left side of the binary point), we can eliminate all except one of them. Indeed,

$$A = (-2^n a_n + 2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i) 2^{-m} \quad (20)$$

$$= (-2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i) 2^{-m} \quad (21)$$

if  $a_n = a_{n-1}$ . By the same argument, we can create extra locations to the left of the sign bit and fill them all with bits identical to the sign bit. Of course, after any of these changes, the leftmost bit assumes the role of the sign bit. Such size changes are useful mainly for hand calculations, as in aligning the binary points, the lsbs, and the sign bits, respectively, of the two numbers to be added.

Examples:

(1)  $00110.11 = 0110.1100 = 6.75$ .

(2)  $11101.1000 = 101.1 = -2.5$ .

## VI. 1's Complement Notation and Arithmetic

The detailed development of 2's complement arithmetic in the previous sections enables a quick treatment of the related 1's complement arithmetic. With  $n+1$  locations  $a_n, \dots, a_0$ , a binary point located between  $a_m$  and  $a_{m-1}$ , the number  $A$  has the representation

$$A = [-(2^n - 1)a_n + \sum_{i=0}^{n-1} 2^i a_i]2^{-m}, \quad (22)$$

in 1's complement notation.

*A. Straightforward Conclusions.*

- (a) The range of numbers representable is from  $-(2^n - 1)2^{-m}$  through  $(2^n - 1)2^{-m}$ , inclusive of both, in increments of  $2^{-m}$ . The bit  $a_n$  serves as the sign bit; the number is positive (or zero) if  $a_n = 0$  and negative (or zero) if  $a_n = 1$ .
- (b) To obtain the representation of a negative number, the bits of the corresponding positive number are all simply Boolean complemented.
- (c) The 1's complement representation of a positive fractional number is the same as its 2's complement representation, and is obtained by the procedure outlined at the end of Section II. The decimal value of any 1's complement representation (of either sign) is obtained by evaluating the weighted sum (22).
- (d) Negation of any number (of any sign) is effected by bit by bit complementation. Therefore the number zero has two distinct representations; all 0 bits and all 1 bits. Negation never results in overflow. Rules for many operations are easily inferred by first obtaining them for positive numbers and then translating them to the case of negative numbers.
- (e) Multiplication by 2 is realized by left circular shifting, if the two left most bits are equal. If not, multiplication results in overflow.
- (f) Division by 2 is realized by right shifting the bits, discarding the lsb, and retaining the sign bit (in addition to duplicating it at a position next to the sign location). If the

discarded bit is the same as the sign bit, division is exact. If not, the absolute value of the result is less than the desired absolute value by an amount equal to half the contribution of the lsb, *i. e.*, by  $2^{-m-1}$ .

- (g) The size of the representation of a number can be changed (without changing the value) as follows. On both left and right side, any number of additional locations can be created and filled with bits identical to the sign bit (if the original representation is for pure integers, a binary point must be explicitly used at a place just right of the original lsb). On the right side of the binary point, an uninterrupted sequence of identical bits starting from and to the left of the lsb, and equal to the sign bit, can be deleted. On the left side of the binary point and starting from the sign bit, all but the right most of an uninterrupted sequence of identical bits, each equal to the sign bit, can be deleted.

### B. Addition.

Consider adding two integers  $A$  and  $B$  represented with  $n+1$  bits. Fractional numbers do not require different rules and are commented upon later. The sum

$$S = A + B = -(2^n - 1)(a_n + b_n) + \sum_{i=0}^{n-1} 2^i (a_i + b_i) \quad (23)$$

is required to be represented as an integer in 1's complement notation with  $n+1$  bits  $s_n \cdots s_0$ . The bits of  $A$  and  $B$  from location 0 to location  $n-1$  are added as in the case of unsigned numbers, with carries from previous positions. The required contribution of the sign bits  $a_n, b_n$ , and the carry into the sign position  $C_n$  is

$$-(2^n - 1)(a_n + b_n) + 2^n C_n = -(2^n - 1)(a_n + b_n - C_n) + C_n. \quad (24)$$

As in the case of 2's complement addition, we seek simpler alternatives. Again, it turns out that we can modify the procedure and add  $a_n$ ,  $b_n$ , and  $C_n$  to obtain two bits; the sum bit  $r_n$  and the carry out of sign position  $C_{n+1}$ . Using Table I developed for addition in 2's complement notation, we find that  $C_{n+1} = C_n$  for six of the eight cases. In each of these six cases, the contribution due to the last term  $C_n$  on the RHS of (24) can be equivalently accounted for by adding  $C_{n+1}$  at the lsb position which is easily implemented by feeding  $C_{n+1}$  at  $C_0$ . If  $C_{n+1}$  is 0, its addition does not change any result bit. If it is 1, it will change some bits of the result  $s_{n-1}, \dots, s_0$  but will not change the  $C_n$  which is 1 even before adding the  $C_{n+1}$  bit. As in the case of 2's complement addition, the value of  $r_n$  is the same as the required  $s_n$  for these six cases of no overflow, and  $C_{n+1} = C_n$ .

Of the remaining two cases, consider  $a_n + b_n - C_n = -1$ . That is,  $a_n = b_n = 0$  and  $C_n = 1$ , which also implies  $C_{n+1} = 0$ . The required contribution of these bits is  $2^n$ . The contribution of  $s_n$  to  $S$  is negative or zero and the maximum the other bits can contribute to  $S$  is  $2^n - 1$ . Thus,  $a_n + b_n - C_n = -1$  is a case of overflow. This is also accompanied by  $C_{n+1} \neq C_n$ . The last of the eight possibilities is  $a_n + b_n - C_n = 2$ . That is,  $a_n = b_n = 1$  and  $C_n = 0$ , which implies  $C_{n+1} = 1$ . The required contribution of  $a_n$ ,  $b_n$ , and  $C_n$  together is  $-2(2^n - 1)$ . If the result of addition upto position  $s_{n-1}$  turns out to be all 1s, half of  $-2(2^n - 1)$  can be added to the contribution  $(2^n - 1)$  due to  $s_{n-1}, \dots, s_0$  and the other half can be represented by  $s_n = 1$ . This can equivalently be achieved by adding the generated  $C_{n+1} = 1$  at  $C_0$  which changes the original  $s_{n-1}, \dots, s_0 = 1, \dots, 1$  to all 0 bits and changes the original  $C_n = 0$  to 1, which in turn changes the original  $a_n + b_n + C_n$  from 10 to 11

ensuring that  $r_n$  finally settles down to the required  $s_n = 1$ . Note also that the final values of  $C_{n+1}$  and  $C_n$  are both equal to 1.

If, on the other hand,  $a_n + b_n - C_n = 2$  and  $s_{n-1}, \dots, s_0$  are not all 1s, the required contribution  $-2(2^n - 1)$  of  $a_n, b_n$ , and  $C_n$  cannot be distributed over  $s_n$  and  $s_{n-1}, \dots, s_0$ . This is another case of overflow. In this case, adding the generated  $C_{n+1} = 1$  to the lsb position through the use of  $C_0 = 1$  will not change the original  $C_n$  from 0 to 1 since we know that  $s_{n-1}, \dots, s_0$  are not all 1s and any new carry generated by adding  $C_{n+1}$  will not propagate to the position  $C_n$ . Thus  $C_n$  continues to be 0 and  $C_{n+1}$ , 1, signifying the required overflow condition. The following summarizes the rule for addition in 1's complement notation. "Add all the  $(n + 1)$  pairs of corresponding bits and carries from previous positions, identical to the procedure for the case of unsigned numbers. Add the resulting carry out of the sign position,  $C_{n+1}$ , to the lsb position, through the use of the carry into the lsb,  $C_0$ . If at this time,  $C_{n+1} = C_n$ , the  $(n + 1)$  resulting sum bits form the correct sum. If not, declare overflow." Subtraction is easily carried out by negating the subtrahend through bit by bit complementation and adding it to the minuend. As in the case of 2's complement notation, the rules for addition are not affected if we have fractional numbers for  $A$  and  $B$ , provided their binary points stand aligned (*i. e.*,  $m$  is the same for both numbers). The arguments are the same except that the weights and values get multiplied by  $2^{-m}$ .

The feed back addition of  $C_{n+1}$  at the  $C_0$  position is known as the *end around carry*. This feed back in an adder implementation with combinational circuits does render the system *asynchronous sequential*. To illustrate this, feed the two numbers  $A = 0 \dots 01$  and

$B = 1 \cdots 11$ . This generates a  $C_{n+1} = 1$  which gets fed back to  $C_0$ . Now, if the lsb of  $A$  is changed to 0 with no other change in the set up, the existing  $C_0 = 1$  reinforces  $C_{n+1} = 1$ , although the new  $A$  and  $B$  require  $C_{n+1}$  to be 0. Fortunately, although the erroneous possibility of  $C_{n+1}$  being 1 when  $A = 0 \cdots 00$  and  $B = 1 \cdots 11$  can lead to the wrong bit pattern of  $0 \cdots 00$  instead of the correct  $1 \cdots 11$  for the result, both these bit patterns evaluate to the same value zero. The generalization of this example corresponds to bit patterns for  $A$  and  $B$  being bit by bit complements of each other so that a spurious  $C_0 = 1$  can reinforce an apparently wrong  $C_{n+1} = 1$ . On the other hand, if  $C_{n+1}$  is required to be 1, it is so only because of the bit patterns of  $A$  and  $B$ . In this case, a spurious  $C_0 = 0$  is quickly corrected by the persistent  $C_{n+1} = 1$ .

If  $C_{i+1} = 1$  is the first carry generated (scanning to the left from  $C_1$ ), the corresponding  $s_i = 0$ . If this carry propagates to generate a  $C_{n+1} = 1$ , this in turn can generate more 1 carries on the lower significant side of the location  $i$ . However, since  $s_i$  was 0, the propagating carry cannot change the original carry  $C_{i+1} = 1$ , or any carry  $C_j, j > i+1$ . Thus, the first 1 carry generated never propagates beyond  $n$  locations (of the  $n+1$  locations) confirming that addition in 1's complement notation is not slower than the same in 2's complement notation.

## VII. Conclusion

This chapter logically, yet concisely, derives rules for arithmetic operations in the two useful binary notations for fixed point fractions. The approach is suitable for adoption in class room teaching of undergraduate students. The establishment of algebraic (weighted sum) expression for a notation creates a setting in which the required properties of a number

at hand, or of an arithmetic operation, can be studied by concentrating on just the few affecting bits. For example, rules for addition depend only on  $a_n$ ,  $b_n$ , and  $C_n$  (even  $C_{n+1}$  is not directly involved, since it is a function of  $a_n$ ,  $b_n$ , and  $C_n$ ). This approach enables us to easily exhaust all possible cases and hence is superior to the conventional approach of description of rules followed by a few illustrative examples. The approach also lays solid foundations for the study of advanced topics such as the Booth's algorithm for multiplication of two numbers.

### Exercises

Only some interesting questions are included here at this time.

- (1) Prove that the maximum value that can be represented by the binary unsigned integer  $a_{n-1}\cdots a_0$  is  $2^n - 1$ .
- (2) Develop the table for overflow signal during addition, as a function of the two input signs and the
  - (a) output signs only.
  - (b) Carry out of sign position only.
- (3) Determine the conditions on the range of values for  $A + B$  that will produce
  - (a)  $C_{n+1} = 0$ ,
  - (b)  $C_{n+1} = 1$ ,
  - (c)  $C_n = 0$ , and
  - (d)  $C_n = 1$ .

- (4) Show that the decimal equivalent of a nonrecurring binary fraction will also be nonrecurring.