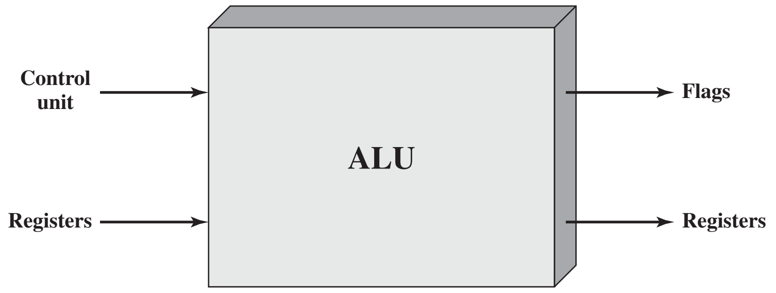


Figure 9.1 ALU Inputs and Outputs

within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

9.2 INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or **radix point**.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

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

[1]See Chapter 19 for a basic refresher on number systems (decimal, binary, hexadecimal).

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

+ 18	= 00010010	
− 18	= 10010010	(sign magnitude)

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \tag{9.1}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 9.3. Another drawback is that there are two representations of 0:

$+0_{10}$	= 00000000	
-0_{10}	= 10000000	(sign magnitude)

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 9.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Most treatments of twos complement representation focus on the rules for producing negative numbers, with no formal proof that the scheme "works." Instead,

[2]In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Table 9.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

our presentation of twos complement integers in this section and in Section 9.3 is based on [DATT93], which suggests that twos complement representation is best understood by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number A ($A < 0$), the sign bit, a_{n-1}, is one. The remaining $n - 1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}. We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, as we will see in Section 9.3, if the weight of the most significant bit is -2^{n-1}. This is the convention used in twos complement representation, yielding the following expression for negative numbers:

$$\textbf{Twos Complement} \qquad A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textbf{(9.2)}$$

Equation (9.2) defines the twos complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines a

Table 9.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

nonnegative integer. When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, yielding a negative integer.

Table 9.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is 1 (2^0) and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 9.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 9.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

Converting between Different Bit Lengths

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Figure 9.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 9.2. The last line can be verified using Equation (9.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

To see why this rule works, let us again consider an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ interpreted as a twos complement integer A, so that its value is

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

If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with $m > n$. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

The two values must be equal:

$$\begin{aligned}
-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i &= -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i \\
-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i &= -2^{n-1} \\
2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i &= 2^{m-1} \\
1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i &= 1 + \sum_{i=0}^{m-2} 2^i \\
\sum_{i=n-1}^{m-2} 2^i a_i &= \sum_{i=n-1}^{m-2} 2^i \\
\Rightarrow \quad a_{m-2} = \cdots = a_{n-2} = a_{n-1} &= 1
\end{aligned}$$

In going from the first to the second equation, we require that the least significant $n - 1$ bits do not change between the two representations. Then we get to the next to last equation, which is only true if all of the bits in positions $n - 1$ through $m - 2$ are 1. Therefore, the sign-extension rule works. The reader may find the rule easier to grasp after studying the discussion on twos-complement negation at the beginning of Section 9.3.

Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

9.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

```
               +18    =    00010010  (twos complement)
bitwise complement    =    11101101
                         +        1
                          ---------
                           11101110 = −18
```

As expected, the negative of the negative of that number is itself:

```
               −18    =    11101110  (twos complement)
bitwise complement    =    00010001
                         +        1
                          ---------
                           00010010 = +18
```

We can demonstrate the validity of the operation just described using the definition of the twos complement representation in Equation (9.2). Again, interpret an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ as a twos complement integer A, so that its value is

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

Now form the bitwise complement, $\overline{a_{n-1}}\,\overline{a_{n-2}}\ldots\overline{a_0}$, and, treating this is an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer B, so that its value is

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Now, we want $A = -B$, which means $A + B = 0$. This is easily shown to be true:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i})\right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

$$\begin{array}{rcr} 0 & = & 00000000 \text{ (twos complement)} \\ \text{bitwise complement} & = & 11111111 \\ & & \underline{+ \qquad\qquad 1} \\ & & 100000000 = 0 \end{array}$$

There is *carry* out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

$$\begin{array}{rcr} -128 & = & 10000000 \text{ (twos complement)} \\ \text{bitwise complement} & = & 01111111 \\ & & \underline{+ \qquad\qquad 1} \\ & & 10000000 = -128 \end{array}$$

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is 2^n, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

Addition and Subtraction

Addition in twos complement is illustrated in Figure 9.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

```
   1001  = −7                 1100  = −4
  +0101  =  5                +0100  =  4
   1110  = −2                10000  =  0

(a) (−7) + (+5)             (b) (−4) + (+4)

   0011  = 3                  1100  = −4
  +0100  = 4                 +1111  = −1
   0111  = 7                 11011  = −5

(c) (+3) + (+4)             (d) (−4) + (−1)

   0101  = 5                  1001  = −7
  +0100  = 4                 +1010  = −6
   1001  = Overflow          10011  = Overflow

(e) (+5) + (+4)             (f) (−7) + (−6)
```

Figure 9.3 Addition of Numbers in Twos Complement Representation

Figures 9.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 9.4. The last two examples demonstrate that the overflow rule still applies.

```
         0010 =  2                    0101 =  5
        +1001 = −7                   +1110 = −2
         1011 = −5                   10011 =  3

(a) M = 2 = 0010              (b) M = 5 = 0101
    S = 7 = 0111                  S = 2 = 0010
   −S =     1001                 −S =     1110

         1011 = −5                    0101 = 5
        +1110 = −2                   +0010 = 2
        11001 = −7                    0111 = 7

(c) M = −5 = 1011             (d) M =  5 = 0101
    S =  2 = 0010                 S = −2 = 1110
   −S =      1110                −S =      0010

         0111 = 7                     1010 = −6
        +0111 = 7                    +1100 = −4
         1110 = Overflow             10110 = Overflow

(e) M =  7 = 0111             (f) M = −6 = 1010
    S = −7 = 1001                 S =  4 = 0100
   −S =      0111                −S =      1100
```

Figure 9.4 Subtraction of Numbers in Twos Complement Representation (M − S)

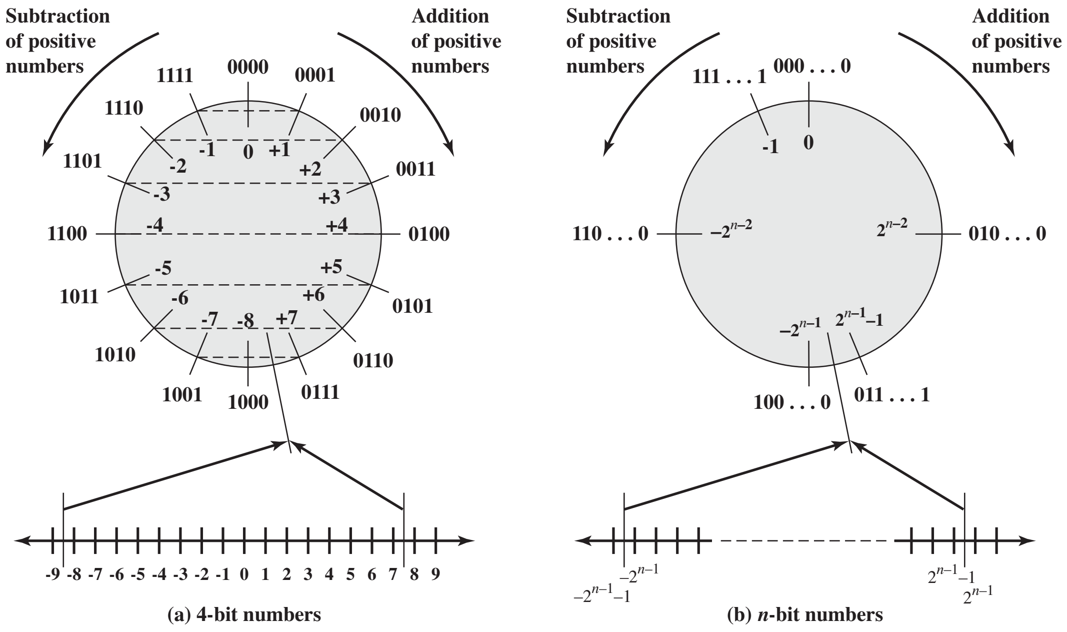
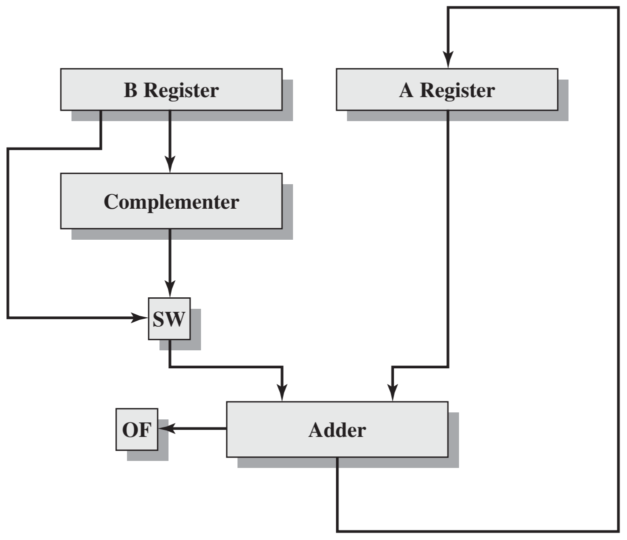


Figure 9.5 Geometric Depiction of Twos Complement Integers

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 9.5. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive k (or subtract negative k) to that number by moving k positions clockwise, and we can subtract positive k (or add negative k) from that number by moving k positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

All of the examples of Figures 9.3 and 9.4 are easily traced in the circle of Figure 9.5.

Figure 9.6 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Chapter 20.) For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag ($0 =$ no overflow; $1 =$ overflow). For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure 9.6 only shows the data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.



OF = Overflow bit
SW = Switch (select addition or subtraction)

Figure 9.6 Block Diagram of Hardware for Addition and Subtraction

Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

UNSIGNED INTEGERS Figure 9.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.

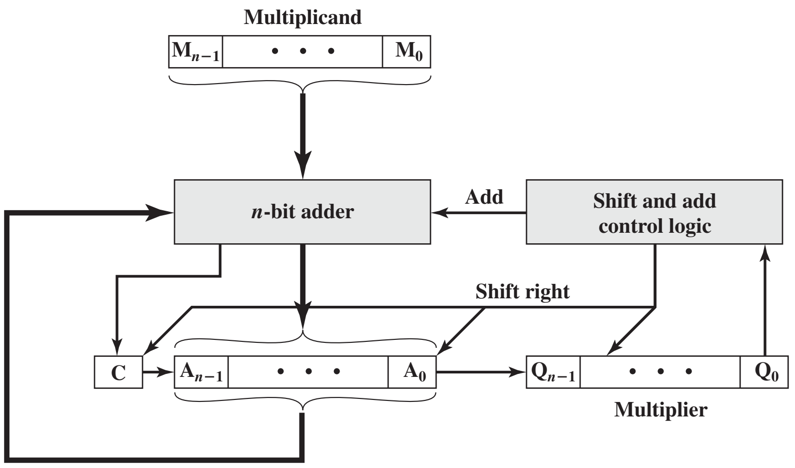
```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  }
   0000   }
  1011    }  Partial products
 1011     }
10001111     Product (143)
```

Figure 9.7 Multiplication of Unsigned Binary Integers

3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 9.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third register,



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 9.8 Hardware Implementation of Unsigned Binary Multiplication

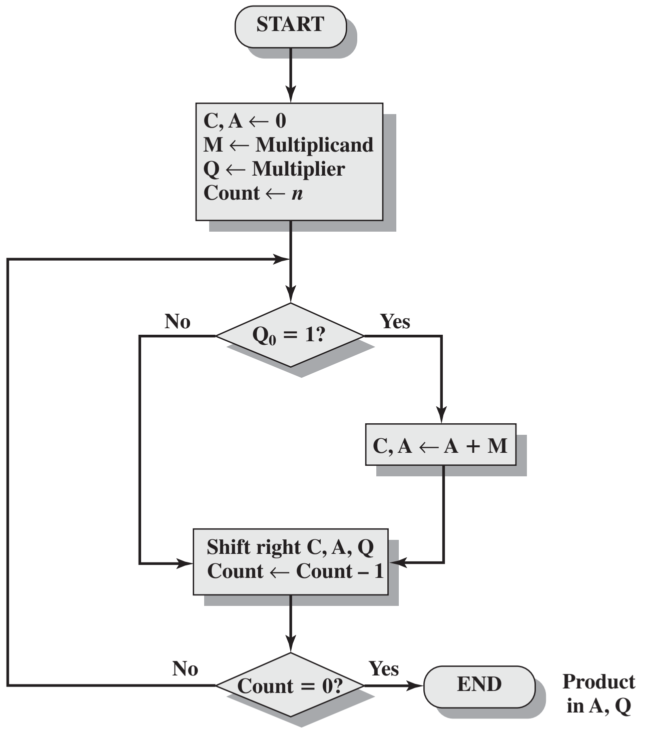


Figure 9.9 Flowchart for Unsigned Binary Multiplication

the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers. A flowchart of the operation is shown in Figure 9.9, and an example is given in Figure 9.8b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

TWOS COMPLEMENT MULTIPLICATION We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\underline{0011} \\ 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7 (1001) to 3 (0011) to get -4 (1100).

```
    1011
  × 1101
--------
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
--------
10001111
```

Figure 9.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 9.7. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have -5 (1011) times -3 (1101) equals -113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 9.7 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

$$\begin{aligned} 1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 2^3 + 2^2 + 2^0 \end{aligned}$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 9.10 recasts Figure 9.7 to make the generation of partial products by multiplication explicit. The only difference in Figure 9.10 is that it recognizes that the partial products should be viewed as $2n$-bit numbers generated from the n-bit multiplicand.

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This is demonstrated in Figure 9.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted as

```
    1001  (9)                      1001  (−7)
  × 0011  (3)                    × 0011  (3)
--------                       --------
00001001  1001 × 2^0           11111001  (−7) × 2^0 = (−7)
00010010  1001 × 2^1           11110010  (−7) × 2^1 = (−14)
--------                       --------
00011011  (27)                 11101011  (−21)
```

(a) Unsigned integers (b) Twos complement integers

Figure 9.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

the twos complement value -7, then each partial product must be a negative twos complement number of $2n$ (8) bits, as shown in Figure 9.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 9.12 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers,

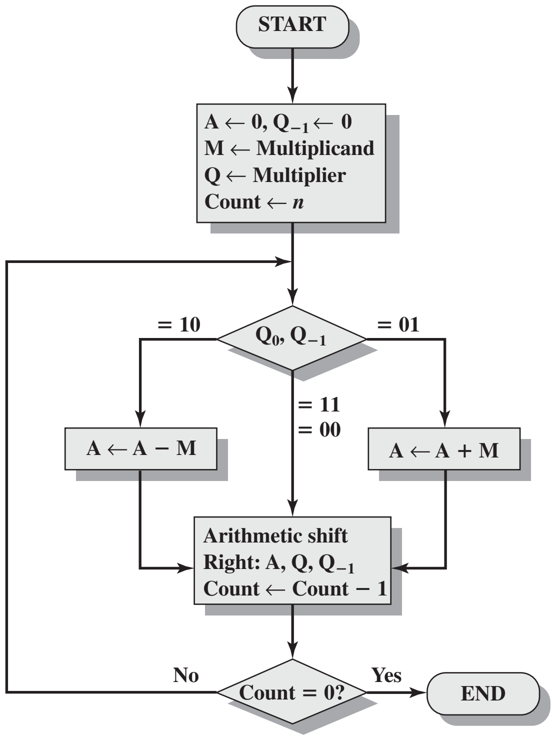


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 9.13 Example of Booth's Algorithm (7 × 3)

respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit.

Figure 9.13 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3. More compactly, the same operation is depicted in Figure 9.14a. The rest of Figure 9.14 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

```
    0111                    0111
  × 0011    (0)           × 1101    (0)
11111001    1–0         11111001    1–0
0000000     1–1         0000111     0–1
000111      0–1         111001      1–0
00010101    (21)        11101011    (–21)
```

(a) (7) × (3) = (21)   (b) (7) × (−3) = (−21)

```
    1001                    1001
  × 0011    (0)           × 1101    (0)
00000111    1–0         00000111    1–0
0000000     1–1         1111001     0–1
111001      0–1         000111      1–0
11101011    (–21)       00010101    (21)
```

(c) (−7) × (3) = (−21)   (d) (−7) × (−3) = (21)

Figure 9.14 Examples Using Booth's Algorithm

Why does Booth's algorithm work? Consider first the case of a positive multiplier. In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (for example, 00011110). As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} \mathrm{M} \times (00011110) &= \mathrm{M} \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= \mathrm{M} \times (16 + 8 + 4 + 2) \\ &= \mathrm{M} \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \qquad \textbf{(9.3)}$$

$$\begin{aligned} \mathrm{M} \times (00011110) &= \mathrm{M} \times (2^5 - 2^1) \\ &= \mathrm{M} \times (32 - 2) \\ &= \mathrm{M} \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} \mathrm{M} \times (01111010) &= \mathrm{M} \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= \mathrm{M} \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

To show that the same scheme works for a negative multiplier, we need to observe the following. Let X be a negative number in twos complement notation:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3}\ldots x_1x_0\}$$

Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0) \qquad \textbf{(9.4)}$$

The reader can verify this by applying the algorithm to the numbers in Table 9.2.

The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111\ldots10x_{k-1}x_{k-2}\ldots x_1x_0\} \qquad \textbf{(9.5)}$$

Then the value of X is

$$\mathrm{X} = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \qquad \textbf{(9.6)}$$

From Equation (9.3), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Rearranging

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \qquad \textbf{(9.7)}$$

Substituting Equation (9.7) into Equation (9.6), we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \qquad \textbf{(9.8)}$$

At last we can return to Booth's algorithm. Remembering the representation of X [Equation (9.5)], it is clear that all of the bits from x_0 up to the leftmost 0 are handled properly because they produce all of the terms in Equation (9.8) but (-2^{k+1}) and thus are in the proper form. As the algorithm scans over the leftmost 0 and encounters the next 1 (2^{k+1}), a 1–0 transition occurs and a subtraction takes place (-2^{k+1}). This is the remaining term in Equation (9.8).

As an example, consider the multiplication of some multiplicand by (−6). In twos complement representation, using an 8-bit word, (−6) is represented as 11111010. By Equation (9.4), we know that

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

which the reader can easily verify. Thus,

$$\text{M} \times (11111010) = \text{M} \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Using Equation (9.7),

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^1)$$

which the reader can verify is still M × (−6). Finally, following our earlier line of reasoning,

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^2 - 2^1)$$

We can see that Booth's algorithm conforms to this scheme. It performs a subtraction when the first 1 is encountered (1–0), an addition when (01) is encountered, and finally another subtraction when the first 1 of the next block of 1s is encountered. Thus, Booth's algorithm performs fewer additions and subtractions than a more straightforward algorithm.

Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 9.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder*. From this point on,