3 BINARY ARITHMETIC OPERATIONS

What to Expect

This chapter develops a number of different binary arithmetic operations. These operations are fundamental in understanding and constructing digital logic circuits using components like adders and logic gates like NOT. Included in this chapter are the following topics.

- Calculating addition/subtraction/multiplication/division problems with binary numbers
- Countering overflow in arithmetic operations
- Representing binary negative numbers
- Contrasting the use of ones-complement and twos-complement numbers
- Developing circuits where bitwise operations like masking are required
- Using and converting codes like ASCII/BCD/Gray

3.1 BINARY ADDITION

Adding binary numbers is a simple task similar to the longhand addition of decimal numbers. As with decimal numbers, the bits are added one column at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of an "Addition Table," as seen in Table 3.1

Inputs			Outputs	
Carry In	**Augend**	**Addend**	**Sum**	**Carry Out**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.1: Addition Table

Just as with decimal addition, two binary integers are added one column at a time, starting from the LSB (the right-most bit in the integer):

```
 1001101
+0010010
 1011111
```

When the sum in one column includes a carry out, it is added to the next column to the left (again, like decimal addition). Consider the following examples:

```
  11  1  <--Carry Bits
 1001001
+0011001
 1100010
```

```
    11   <--Carry Bits
 1000111
+0010110
 1011101
```

The "ripple-carry" process is simple for humans to understand, but it causes a significant problem for designers of digital circuits. Consequently, ways were developed to carry a bit to the left in an electronic adder circuit and that is covered in Section 8.1, page 173.

Binary numbers that include a fractional component are added just like binary integers; however, the radix points must align so the augend and addend may need to be padded with zeroes on either the left or the right. Here is an example:

```
  111 1    <--Carry Bits
 1010.0100
```

```
+0011.1101
 1110.0001
```

3.1.1 *Overflow Error*

One problem circuit designers must consider is a carry out bit in the MSB (left-most bit) in the answer. Consider the following:

```
 11 11    <--Carry Bits
 10101110
+11101101
110011011
```

This example illustrates a significant problem for circuit designers. Suppose the above calculation was done with a circuit that could only accommodate eight data bits. The augend and addend are both eight bits wide, so they are fine; however, the sum is nine bits wide due to the carry out in the MSB. In an eight-bit circuit (that is, a circuit where the devices and data lines can only accommodate eight bits of data), the carry out bit would be dropped since there is not enough room to accommodate it.

The result of a dropped bit cannot be ignored. The example problem above, when calculated in decimal, is $174_{10} + 237_{10} = 411_{10}$. If, though, the MSB carry out is dropped, then the answer becomes 155_{10}, which is, of course, incorrect. This type of error is called an *Overflow Error*, and a circuit designer must find a way to correct overflow. One typical solution is to simply alert the user that there was an overflow error. For example, on a handheld calculator, the display may change to something like *-E-* if there is an error of any sort, including overflow.

3.1.2 *Sample Binary Addition Problems*

The following table lists several binary addition problems that can be used for practice.

Augend	Addend	Sum
10 110.0	11 101.0	110 011.0
111 010.0	110 011.0	1 101 101.0
1011.0	111 000.0	1 000 011.0
1 101 001.0	11 010.0	10 000 011.0
1010.111	1100.001	10 111.000
101.01	1001.001	1110.011

Table 3.2: Binary Addition Problems

3.2 BINARY SUBTRACTION

3.2.1 *Simple Manual Subtraction*

Subtracting binary numbers is similar to subtracting decimal numbers and uses the same process children learn in primary school. The minuend and subtrahend are aligned on the radix point, and then columns are subtracted one at a time, starting with the least significant place and moving to the left. If the subtrahend is larger than the minuend for any one column, an amount is "borrowed" from the column to the immediate left. Binary numbers are subtracted in the same way, but it is important to keep in mind that binary numbers have only two possible values: zero and one. Consider the following problem:

```
 10.1
-01.0
 01.1
```

In this problem, the LSB column is $1 - 0$, and that equals one. The middle column, though, is $0 - 1$, and one cannot be subtracted from zero. Therefore, one is borrowed from the most significant bit, so the problem in middle column becomes $10 - 1$. (Note: do not think of this as "ten minus one" - remember that this is binary so this problem is "one-zero minus one," or two minus one in decimal) The middle column is $10 - 1 = 1$, and the MSB column then becomes $0 - 0 = 0$.

The radix point must be kept in alignment throughout the problem, so if one of the two operands has too few places it is padded on the left or right (or both) to make both operands the same length. As an example, subtract: $101101.01 - 1110.1$:

```
 101101.01
-001110.10
  11110.11
```

There is no difference between decimal and binary as far as the subtraction process is concerned. In each of the problems in this section the minuend is greater than the subtrahend, leading to a positive difference; however, if the minuend is less than the subtrahend, the result is a negative number and negative numbers are developed in the next section of this chapter.

Table 3.3 includes some subtraction problems for practice:

Minuend	Subtrahend	Difference
1 001 011.0	111 010.0	10 001.0
100 010.0	10 010.0	10 000.0
101 110 110.0	11 001 010.0	10 101 100.0
1 110 101.0	111 010.0	111 011.0
11 011 010.1101	101 101.1	10 101 101.0101
10 101 101.1	1 101 101.101	111 111.111

Table 3.3: Binary Subtraction Problems

3.2.2 *Representing Negative Binary Numbers Using Sign-and-Magnitude*

Sign-and-magnitude was used in early computers since it mimics real number arithmetic, but has been replaced by more efficient negative number systems in modern computers.

Binary numbers, like decimal numbers, can be both positive and negative. While there are several methods of representing negative binary numbers; one of the most intuitive is using *sign-and-magnitude*, which is essentially the same as placing a "–" in front of a decimal number. With the sign-and-magnitude system, the circuit designer simply designates the MSB as the *sign bit* and all others as the magnitude of the number. When the sign bit is one the number is negative, and when it is zero the number is positive. Thus, -5_{10} would be written as 1101_2.

Unfortunately, despite the simplicity of the sign-and-magnitude approach, it is not very practical for binary arithmetic, especially when done by a computer. For instance, negative five (1101_2) cannot be added to any other binary number using standard addition technique since the sign bit would interfere. As a general rule, errors can easily occur when bits are used for any purpose other than standard place-weighted values; for example, 1101_2 could be misinterpreted as the number 13_{10} when, in fact, it is meant to represent -5. To keep things straight, the circuit designer must first decide how many bits are going to be used to represent the largest numbers in the circuit, add one more bit for the sign, and then be sure to never exceed that bit field length in arithmetic operations. For the above example, three data bits plus a sign bit would limit arithmetic operations to numbers from negative seven (1111_2) to positive seven (0111_2), and no more.

This system also has the quaint property of having two values for zero. If using three magnitude bits, these two numbers are both zero: 0000_2 (positive zero) and 1000_2 (negative zero).

3.2.3 *Representing Negative Binary Numbers Using Signed Complements*

3.2.3.1 *About Complementation*

Before discussing negative binary numbers, it is important to understand the concept of complementation. To start, recall that the *radix* (or base) of any number system is the number of ciphers available for counting; the decimal (or base-ten) number system has ten ciphers $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ while the binary (or base-two) number system has two ciphers $(0, 1)$. By definition, a number plus its complement equals the radix (this is frequently called the *radix complement*). For example, in the decimal system four is the radix complement of six since $4 + 6 = 10$. Another type of complement is the *diminished radix complement*, which is the complement of the radix minus one. For example, in the decimal system six is the diminished radix complement of three since $6 + 3 = 9$ and nine is equal to the radix minus one.

DECIMAL. In the decimal system the radix complement is usually called the *tens* complement since the radix of the decimal system is ten. Thus, the tens complement of eight is two since $8 + 2 = 10$. The diminished radix complement is called the *nines* complement in the decimal system. As an example, the nines complement of decimal eight is one since $8 + 1 = 9$ and nine is the diminished radix of the decimal system.

To find the nines complement for a number larger than one place, the nines complement must be found for each place in the number. For example, to find the nines complement for 538_{10}, find the nines complement for each of those three digits, or 461. The easiest way to find the tens complement for a large decimal number is to first find the nines complement and then add one. For example, the tens complement of 283 is 717, which is calculated by finding the nines complement, 716, and then adding one.

BINARY. Since the radix for a binary number is 10_2, (be careful! this is not ten, it is one-zero in binary) the diminished radix is 1_2. The diminished radix complement is normally called the *ones complement* and is obtained by reversing (or "flipping") each bit in a binary number; so the ones complement of 100101_2 is 011010_2.

The radix complement (or *twos complement*) of a binary number is found by first calculating the ones complement and then adding one to that number. The ones complement of 101101_2 is 010010_2, so the twos complement is $010010_2 + 1_2 = 010011_2$.

3.2.3.2 *Signed Complements*

In circuits that use binary arithmetic, a circuit designer can opt to use ones complement for negative numbers and designate the most significant bit as the sign bit; and, if so, the other bits are the magnitude of the number. This is similar to the *sign-and-magnitude* system discussed on page 39. By definition, when using ones complement negative numbers, if the most significant bit is zero, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is one, then the number is negative and the magnitude of the number is determined by calculating the ones complement of the number. Thus: $0111_2 = +7_{10}$, and $1000_2 = -7_{10}$ (the ones complement for 1000_2 is 0111_2). Table 3.4 may help to clarify this concept:

Decimal	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

Table 3.4: Ones Complement

In a four-bit binary number, any decimal number from -7 to $+7$ can be represented; but, notice that, like the sign-and-magnitude system, there are two values for zero, one positive and one negative. This requires extra circuitry to test for both values of zero after subtraction operations.

To simplify circuit design, a designer can opt to use twos complement negative numbers and designate the most significant bit as the sign bit so the other bits are the number's magnitude. To use twos complement numbers, if the most significant bit is zero, then the number is positive and the magnitude of the number is determined by the remaining bits; but if the most significant bit is one, then the number is negative and the magnitude of the number is determined by taking the twos complement of the number (that is, the ones complement plus one). Thus: $0111 = 7$, and $1001 = -7$ (the ones complement of 1001 is 0110, and $0110 + 1 = 0111$). Table 3.5 may help to clarify this concept:

Decimal	Positive	Negative
0	0000	10000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	N/A	1000

Table 3.5: Twos Complement

The twos complement removes that quirk of having two values for zero. Table 3.5 shows that zero is either 0000 or 10000; but since this is a four-bit number the initial one is discarded, leaving 0000 for zero whether the number is positive or negative. Also, 0000 is considered a positive number since the sign bit is zero. Finally, notice that 1000 is -8 (ones complement of 1000 is 0111, and $0111 + 1 = 1000$). This means that binary number systems that use a twos complement method of designating negative numbers will be asymmetrical; running, for example, from -8 to $+7$. A twos complement system still has the same number of positive and negative numbers, but zero is considered positive, not neutral.

Programmers reading this book may have wondered why the maximum/minimum values for various types of variables is asymmetrical.

All modern computer systems use radix (or twos) complements to represent negative numbers.

One other quirk about the twos complement system is that the decimal value of the binary number can be quickly calculated by assuming the sign bit has a negative place value and all other places are added to it. For example, in the negative number 1010_2, if the sign bit is assumed to be worth -8 and the other places are added to that, the result is $-8 + 2$, or -6; and -6 is the value of 1010_2 in a twos complement system.

3.2.3.3 *About Calculating the Twos Complement*

In the above section, the twos (or radix) complement is calculated by finding the ones complement of a number and then adding one. For machines, this is the most efficient method of calculating the twos complement; but there is a method that is much easier for humans to use to find the twos complement of a number. Start with the LSB (the right-most bit) and then read the number from right to left. Look for the first one and then invert every bit to the left of that one. As an example, the twos complement for $1010\underline{10}_2$ is formed by starting with the least significant bit (the zero on the right), and working to the left, looking for the first one, which is in the second place from the right.

Then, every bit to the left of that one is inverted, ending with: $0101\underline{10}_2$ (the two LSBs are underlined to show that they are the same in both the original and twos complement number).

Table 3.6 displays a few examples:

Number	Twos Complement
0110100	1001100
11010	00110
001010	110110
1001011	0110101
111010111	000101001

Table 3.6: Example Twos Complement

3.2.4 *Subtracting Using the Diminished Radix Complement*

When thinking about subtraction, it is helpful to remember that $A - B$ is the same as $A + (-B)$. Computers can find the complement of a particular number and add it to another number much faster and easier than attempting to create separate subtraction circuits. Therefore, subtraction is normally carried out by adding the complement of the subtrahend to the minuend.

3.2.4.1 *Decimal*

It is possible to subtract two decimal numbers by adding the nines complement, as in the following example:

This method is commonly used by stage performers who can subtract large numbers in their heads. While it seems somewhat convoluted, it is fairly easy to master.

$$\begin{array}{r} 735 \\ -\underline{142} \end{array}$$

Calculate the nines complement of the subtrahend: 857 (that is $9-1$, $9-4$, and $9-2$). Then, add that nines complement to the minuend:

$$\begin{array}{r} 735 \\ +\underline{857} \\ 1592 \end{array}$$

The initial one in the sum (the thousands place) is dropped so the number of places in the answer is the same as for the two addends, leaving 592. Because the diminished radix used to create the subtrahend is one less than the radix, one must be added to the answer; giving 593, which is the correct answer for $735 - 142$.

3.2.4.2 *Binary*

The diminished radix complement (or ones complement) of a binary number is found by simply "flipping" each bit. Thus, the ones complement of 11010 is 00101. Just as in decimal, a binary number can be subtracted from another by adding the diminished radix complement of the subtrahend to the minuend, and then adding one to the sum. Here is an example:

```
 101001
-011011
```

Add the ones complement of the subtrahend:

```
 101001
+100100
1001101
```

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001101_2 and adding one to that number (because the diminished radix is one less than the radix) leaves 1110_2. In decimal, the problem is $41 - 27 = 14$.

Often, diminished radix subtraction circuits are created such that they use *end around* carry bits. In this case, the most significant bit is carried around and added to the final sum. If that bit is one, then that increases the final answer by one, and the answer is a positive number. If, though, the most significant bit is zero, then there is no end around carry so the answer is negative and must be complemented to find the true value. Either way, the correct answer is found.

Here is an example:

```
 0110  (6)
-0010  (2)
```

Solution:

```
 0110  (6)
+1101  (-2 in ones complement)
10011
    1  (End-around carry the MSB)
=0100  (4)
```

Answer: 4 (since there was an end-around carry the solution is a positive number). Here is a second example:

```
 0010  (2)
-0110  (6)
```

Solution:

```
 0010  (2)
+1001  (-6 in ones complement)
 1011  (No end-around carry, so ones complement)
=0100  (-4: no end-around carry so negative answer)
```

Because the diminished radix (or ones) complement of a binary number includes that awkward problem of having two representations for zero, this form of subtraction is not used in digital circuits; instead, the radix (or twos) complement is used (this process is discussed next). It is worth noting that subtracting by adding the diminished radix of the subtrahend and then adding one is awkward for humans, but complementing and adding is a snap for digital circuits. In fact, many early mechanical calculators used a system of adding complements rather than having to turn gears backwards for subtraction.

3.2.5 *Subtracting Using the Radix Complement*

3.2.5.1 *Decimal*

The radix (or tens) complement of a decimal number is the nines complement plus one. Thus, the tens complement of 7 is 3; or $((9-7)+1)$ and the tens complement of 248 is 752 (find the nines complement of each place and then add one to the complete number: $751+1$). It is possible to subtract two decimal numbers using the tens complement, as in the following example:

```
 735
-142
```

Calculate the tens complement of the subtrahend, 142, by finding the nines complement for each digit and then adding one to the complete number: 858 (that is $9-1$, $9-4$, and $9-2+1$). Then, add that tens complement number to the original minuend:

```
 735
+858
1593
```

The initial one in the answer (the thousands place) is dropped so the answer has the same number of decimal places as the addends, leaving 593, which is the correct answer for $735-142$.

3.2.5.2 *Binary*

To find the radix (or twos) complement of a binary number, each bit in the number is "flipped" (making the ones complement) and then one is added to the result. Thus, the twos complement of 11010_2 is 00110_2 (or $(00101_2)+1_2$). Just as in decimal, a binary number can be subtracted from another by adding the radix complement of the subtrahend to the minuend. Here's an example:

```
 101001
-011011
```

Add the twos complement of the subtrahend:

```
 101001
+011011
1001110
```

The most significant bit is discarded so the solution has the same number of bits as for the two addends. This leaves 001110_2 (or 14_{10}). Converting all of this to decimal, the original problem is $41 - 27 = 14$.

Here are two worked out examples:

Calculate $0110_2 - 0010_2$ (or $6_{10} - 2_{10}$):

```
 0110  (6)
-0010  (2)
```

Solution:

```
 0110  (6)
+1110  (-2 in twos complement)
10100  (Discard the MSB, the answer is 4)
```

Calculate $0010_2 - 0110_2$ (or $2_{10} - 6_{10}$)

```
 0010  (2)
-0110  (6)
```

Solution:

```
 0010  (2)
+1010  (-6 in twos complement)
 1100
 0100  (Twos complement of the sum, -4)
```

3.2.6 *Overflow*

One caveat with signed binary numbers is that of overflow, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the allotted number of bits. Remember that the sign bit is defined as the most significant bit in the number. For example, with a six-bit number, five bits are used for magnitude, so there is a range from 00000_2 to 11111_2, or 0_{10} to 31_{10}. If a sign bit is included, and using twos complement, numbers as high as 011111_2 ($+31_{10}$) or as low as 100000_2 (-32_{10}) are possible. However, an addition problem with two signed six-bit numbers that results in a sum greater than $+31_{10}$ or less than -32_{10} will yield an incorrect answer. As an example, add 17_{10} and 19_{10} with signed six-bit numbers:

```
 010001  (17)
+010011  (19)
 100100
```

The answer (100100_2), interpreted with the most significant bit as a sign, is equal to -28_{10}, not $+36_{10}$ as expected. Obviously, this is not correct. The problem lies in the restrictions of a six-bit number field. Since the true sum (36) exceeds the allowable limit for our designated bit field (five magnitude bits, or $+31$), it produces what is called an overflow error. Simply put, six places is not large enough to represent the correct sum if the MSB is being used as a sign bit, so whatever sum is obtained will be incorrect. A similar error will occur if two negative numbers are added together to produce a sum that is too small for a six-bit binary field. As an example, add -17_{10} and -19_{10}:

```
 -17 = 101111
 -19 = 101101

 101111  (-17)
+101101  (-19)
1011100
```

The solution as shown: $011100_2 = +28_{10}$. (Remember that the most significant bit is dropped in order for the answer to have the same number of places as the two addends.) The calculated (incorrect) answer for this addition problem is 28 because true sum of $-17 + -19$ was too small to be properly represented with a five bit magnitude field.

Here is the same overflow problem again, but expanding the bit field to six magnitude bits plus a seventh sign bit. In the following example, both $17 + 19$ and $(-17) + (-19)$ are calculated to show that both can be solved using a seven-bit field rather than six-bits.

Add 17 + 19:

```
 0010001  (17)
+0010011  (19)
 0100100  (36)
```

Add (-17) + (-19):

```
 -17 = 1101111
 -19 = 1101101

 1101111  (-17)
+1101101  (-19)
11011100  (-36)
```

The correct answer is only found by using bit fields sufficiently large to handle the magnitude and sign bits in the sum.

3.2.6.1 *Error Detection*

Overflow errors in the above problems were detected by checking the problem in decimal form and then comparing the results with the binary answers calculated. For example, when adding +17 and +19, the answer was supposed to be +36, so when the binary sum was −28, something had to be wrong. Although this is a valid way of detecting overflow errors, it is not very efficient, especially for computers. After all, the whole idea is to reliably add binary numbers together and not have to double-check the result by adding the same numbers together in decimal form. This is especially true when building logic circuits to add binary quantities: the circuit must detect an overflow error without the supervision of a human who already knows the correct answer.

The simplest way to detect overflow errors is to check the sign of the sum and compare it to the signs of the addends. Obviously, two positive numbers added together will give a positive sum and two negative numbers added together will give a negative sum. With an overflow error, however, the sign of the sum is always opposite that of the two addends: $(+17) + (+19) = -28$ and $(-17) + (-19) = +28$. By checking the sign bits an overflow error can be detected.

It is not possible to generate an overflow error when the two addends have opposite signs. The reason for this is apparent when the nature of overflow is considered. Overflow occurs when the magnitude of a number exceeds the range allowed by the size of the bit field. If a positive number is added to a negative number then the sum will always be closer to zero than either of the two added numbers; its magnitude must be less than the magnitude of either original number, so overflow is impossible.

3.3 BINARY MULTIPLICATION

3.3.1 *Multiplying Unsigned Numbers*

Multiplying binary numbers is very similar to multiplying decimal numbers. There are only four entries in the Binary Multiplication Table:

$0X0 = 0$
$0X1 = 0$
$1X0 = 0$
$1X1 = 1$

Table 3.7: Binary Multiplication Table

To multiply two binary numbers, work through the multiplier one number at a time (right-to-left) and if that number is one, then shift left and copy the multiplicand as a partial product; if that number is zero, then shift left but do not copy the multiplicand (zeros can be used as placeholders if desired). When the multiplying is completed add all partial products. This sounds much more complicated than it actually is in practice and is the same process that is used to multiply two decimal numbers. Here is an example problem.

```
    1011   (11)
  X 1101   (13)
    ----------
    1011
   0000
  1011
 1011
 -------------
10001111   (143)
```

3.3.2 *Multiplying Signed Numbers*

The simplest method used to multiply two numbers where one or both are negative is to use the same technique that is used for decimal numbers: multiply the two numbers and then determine the sign from the signs of the original numbers: if those signs are the same then the result is positive, if they are different then the result is negative. Multiplication by zero is a special case where the result is always zero.

The multiplication method discussed above works fine for paper-and-pencil; but is not appropriate for designing binary circuits. Unfortunately, the mathematics for binary multiplication using an algorithm that can become an electronic circuit is beyond the scope of this book. Fortunately, though, ICs already exist that carry out multiplication of both signed and floating-point numbers, so a circuit designer can use a pre-designed circuit and not worry about the complexity of the multiplication process.

3.4 binary division

Binary division is accomplished by repeated subtraction and a right shift function; the reverse of multiplication. The actual process is rather convoluted and complex and is not covered in this book. Fortunately, though, ICs already exist that carry out division of both signed and floating-point numbers, so a circuit designer can use a pre-designed circuit and not worry about the complexity of the division process.

3.5 BITWISE OPERATIONS

It is sometimes desirable to find the value of a given bit in a byte. For example, if the LSB is zero then the number is even, but if it is one then the number is odd. To determine the "evenness" of a number, a bitwise mask is multiplied with the original number. As an example, imagine that it is desired to know if 1001010_2 is even, then:

```
          1001010   <- Original Number
     BitX 0000001   <- "Evenness" Mask
          0000000
```

The bits are multiplied one position at a time, from left-to-right. Any time a zero appears in the mask that bit position in the product will be zero since any number multiplied by zero yields zero. When a one appears in the mask, then the bit in the original number will be copied to the solution. In the given example, the zero in the least significant bit of the top number is multiplied with one and the result is zero. If that LSB in the top number had been one then the LSB in the result would have also been one. Therefore, an "even" original number would yield a result of all zeros while an odd number would yield a one.

3.6 CODES

3.6.1 *Introduction*

Codes are nothing more than using one system of symbols to represent another system of symbols or information. Humans have used codes to encrypt secret information from ancient times. However, digital logic codes have nothing to do with secrets; rather, they are only concerned with the efficient storage, retrieval, and use of information.

3.6.1.1 *Morse Code*

As an example of a familiar code, Morse code changes letters to electric pulses that can be easily transmitted over a radio or telegraph wire. Samuel Morse's code uses a series of dots and dashes to represent letters so an operator at one end of the wire can use electromagnetic pulses to send a message to some receiver at a distant end. Most people are familiar with at least one phrase in Morse code: *SOS*. Here is a short sentence in Morse:

```
-.-. --- -.. . ...   .- .-. .   ..-. ..- -.
 c    o   d  e  s    a   r  e    f    u  n
```

3.6.1.2 *Braille Alphabet*

As one other example of a commonly-used code, in 1834 Louis Braille, at the age of 15, created a code of raised dots that enable blind people to read books. For those interested in this code, the Braille alphabet can be found at http://braillebug.afb.org/braille_print.asp.

3.6.2 *Computer Codes*

The fact is, computers can only work with binary numbers; that is how information is stored in memory, how it is processed by the CPU, how it is transmitted over a network, and how it is manipulated in any of a hundred different ways. It all boils down to binary numbers. However, humans generally want a computer to work with words (such as email or a word processor), ciphers (such as a spreadsheet), or graphics (such as photos). All of that information must be encoded into binary numbers for the computer and then decoded back into understandable information for humans. Thus, binary numbers stored in a computer are often codes used to represent letters, programming steps, or other non-numeric information.

3.6.2.1 *ASCII*

Computers must be able to store and process letters, like those on this page. At first, it would seem easiest to create a code by simply making A=1, B=2, and so forth. While this simple code does not work for a number of reasons, the idea is on the right track and the code that is actually used for letters is similar to this simple example.

In the early 1960s, computer scientists came up with a code they named American Standard Code for Information Interchange (ASCII) and this is still among the most common ways to encode letters and other symbols for a computer. If the computer program knows that a particular spot in memory contains binary numbers that are actually ASCII-coded letters, it is a fairly easy job to convert those codes to letters for a screen display. For simplicity, ASCII is usually represented by hexadecimal numbers rather than binary. For example, the word *Hello* in ASCII is: 048 065 06C 06C 06F.

ASCII code also has a predictable relationship between letters. For example, capital letters are exactly 20_{16} higher in ASCII than their lower-case version. Thus, to change a letter from lower-case to upper-case, a programmer can add 20_{16} to the ASCII code for the lower-case letter. This can be done in a single processing step by using what is known as a *bit-wise AND* on the bit representing 20_{16} in the ASCII code's binary number.

An ASCII chart using hexadecimal values is presented in Table 3.8. The most significant digit is read across the top row and the least significant digit is read down the left column. For example, the letter *A* is 41_{16} and the number *6* is 36_{16}.

	0	1	2	3	4	5	6	7
0	NUL	DLE		0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	∧	n	~
F	SI	US	/	?	O	_	o	DEL

Table 3.8: ASCII Table

Teletype operators from decades past tell stories of sending 25 or more 07_{16} codes (ring the bell) to a receiving terminal just to irritate another operator in the middle of the night.

ASCII 20_{16} is a space character used to separate words in a message and the first two columns of ASCII codes (where the high-order nibble are zero and one) were codes essential for teletype machines, which were common from the 1920s until the 1970s. The meanings of a few of those special codes are given in Table 3.9.

NUL	All Zeros (a "null" byte)
SOH	Start of Header
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquire (is the remote station on?)
ACK	Acknowledge (the station is on)
BEL	Ring the terminal bell (get the operator's attention)

Table 3.9: ASCII Symbols

Table 3.10 contains a few phrases in both plain text and ASCII for practice.

Plain Text	ASCII
codes are fun	63 6f 64 65 73 20 61 72 65 20 66 75 6e
This is ASCII	54 68 69 73 20 69 73 20 41 53 43 49 49
365.25 days	33 36 35 2e 32 35 20 64 61 79 73
It's a gr8 day!	49 74 27 73 20 61 20 67 72 38 20 64 61 79 21

Table 3.10: ASCII Practice

While the ASCII code is the most commonly used text representation, it is certainly not the only way to encode words. Another popular code is Extended Binary Coded Decimal Interchange Code (EBCDIC) (pronounced like "Eb See Deck"), which was invented by IBM in 1963 and has been used in most of their computers ever since.

Since the early 2000's, computer programs have begun to use Unicode character sets, which are similar to ASCII but multiple bytes are combined to expand the number of characters available for non-English languages like Cyrillic.

3.6.2.2 *Binary Coded Decimal (BCD)*

It is often desirable to have numbers coded in such a way that they can be easily translated back and forth between decimal (which is easy for humans to manipulate) and binary (which is easy for computers to manipulate). BCD is the code used to represent decimal numbers in binary systems. BCD is useful when working with decimal input (keypads or transducers) and output (displays) devices.

There are, in general, two types of BCD systems: non-weighted and weighted. Non-weighted codes are special codes devised for a single purpose where there is no implied relationship between one value and the next. As an example, 1001 could mean one and 1100 could mean two in some device. The circuit designer would create whatever code meaning is desired for the application.

Weighted BCD is a more generalized system where each bit position is assigned a "weight," or value. These types of BCD systems are far more common than non-weighted and are found in all sorts of applications. Weighted BCD codes can be converted to decimal by adding the place value for each position in exactly the same way that Expanded Positional Notation is used for to covert between decimal and binary numbers. As an example, the weights for the Natural BCD system are $8-4-2-1$. (These are the same weights used for binary numbers; thus the name "natural" for this system.) The code 1001_{BCD} is converted to decimal like this:

$$\begin{aligned} 1001_{BCD} &= (1X8) + (0X4) + (0X2) + (1X1) \\ &= (8) + (0) + (0) + (1) \\ &= 9_{10} \end{aligned} \tag{3.1}$$

Because there are ten decimal ciphers $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$, it requires four bits to represent all decimal digits; so most BCD code systems are four bits wide. In practice, only a few different weighted BCD code systems are commonly used and the most common are shown in Table 3.11.

Decimal	8421 (Natural)	2421	Ex3	5421
0	0000	0000	0011	0000
1	0001	0001	0100	0001
2	0010	0010	0101	0010
3	0011	0011	0110	0011
4	0100	0100	0111	0100
5	0101	1011	1000	1000
6	0110	1100	1001	1001
7	0111	1101	1010	1010
8	1000	1110	1011	1011
9	1001	1111	1100	1100

Table 3.11: BCD Systems

Remember that BCD is a code system, not a number system; so the meaning of each combination of four-bit codes is up to the designer and will not necessarily follow any sort of binary numbering sequence.

The name of each type of BCD code indicates the various place values. Thus, the 2421 BCD system gives the most significant bit of the number a value of two, not eight as in the natural code. The *Ex3* code (for "Excess 3") is the same as the natural code, but each value is increased by three (that is, three is added to the natural code).

In each of the BCD code systems in Table 3.11 there are six unused four-bit combinations; for example, in the *Natural* system the unused codes are: $1010, 1011, 1100, 1101, 1110$, and 1111. Thus, any circuit designed to use BCD must include some sort of check to ensure that if unused binary values are accidentally input into a circuit it does not create an undefined outcome.

Normally, two BCD codes, each of which are four bits wide, are packed into an eight-bit byte in order to reduce wasted computer memory. Thus, the packed BCD 01110010 contains two BCD numbers: 72. In fact, a single 32-bit word, which is common in many computers, can contain 8 BCD codes. It is a trivial matter for software to either pack or unpack BCD codes from a longer word.

It is natural to wonder why there are so many different ways to code decimal numbers. Each of the BCD systems shown in Table 3.11

has certain strengths and weaknesses and a circuit designer would choose a specific system based upon those characteristics.

converting between bcd and other systems. One thing that makes BCD so useful is the ease of converting from BCD to decimal. Each decimal digit is converted into a four-bit BCD code, one at a time. Here is 37_{10} in Natural BCD:

```
0011 0111
  3    7
```

It is, generally, very easy to convert Natural BCD to decimal since the BCD codes are the same as binary numbers. Other BCD systems use different place values, and those require more thought to convert (though the process is the same). The place values for BCD systems other than Natural are indicated in the name of the system; so, for example, the 5421 system would interpret the number $1001_{BCD5421}$ as:

$$\begin{aligned} 1001_{BCD5421} &= (1X5) + (0X4) + (0X2) + (1X1) \\ &= (5) + (0) + (0) + (1) \\ &= 6_{10} \end{aligned} \tag{3.2}$$

Converting from decimal to BCD is also a rather simple process. Each decimal digit is converted to a four-bit BCD equivalent. In the case of Natural BCD the four-bit code is the binary equivalent to the decimal number, other weighted BCD codes would be converted with a similar process.

```
  2    4    5
0010 0100 0101
```

To convert binary to BCD is no trivial exercise and is best done with an automated process. The normal method used is called the *Shift Left and Add Three* algorithm (or, frequently, *Double-Dabble*). The process involves a number of steps where the binary number is shifted left and occasionally three is added to the resulting shift. Wikipedia (https://en.wikipedia.org/wiki/Double_dabble) has a good explanation of this process, along with some examples.

Converting BCD to any other system (like hexadecimal) is most easily done by first converting to binary and then to whatever base is desired. Unfortunately, converting BCD to binary is not as simple as concatenating two BCD numbers; for example, 01000001 is 41 in BCD, but those two BCD numbers concatenated, 01000001, is 65 in binary. One way to approach this type of problem is to use the reverse of the *Double-Dabble* process: *Shift Right and Subtract Three*. As in converting binary to BCD, this is most easily handled by an automated process.

SELF-COMPLEMENTING. The Excess-3 code (called *Ex3* in the table) is self-complementing; that is, the nines complement of any decimal number is found by complementing each bit in the Ex3 code. As an example, find the nines complement for 127_{10}:

1	127_{10}	Original Number
2	$0100\ 0101\ 1010_{Ex3}$	Convert 127 to Excess 3
3	$1011\ 1010\ 0101_{Ex3}$	Ones Complement
4	872_{10}	Convert to Decimal

Table 3.12: Nines Complement for 127

Thus, 872_{10}, is the nines complement of 127_{10}. It is a powerful feature to be able to find the nines complement of a decimal number by simply complementing each bit of its Ex3 BCD representation.

REFLEXIVE. Some BCD codes exhibit a reflexive property where each of the upper five codes are complementary reflections of the lower five codes. For example, 0111_{Ex3} (4) and 1000_{Ex3} (5) are complements, 0110_{Ex3} (3) and 1001_{Ex3} (6) are complements, and so forth. The reflexive property for the 5421 code is different. Notice that the codes for zero through four are the same as those for five through nine, except for the MSB (zero for the lower codes, one for the upper codes). Thus, 0000_{5421} (zero) is the same as 1000_{5421} (five) except for the first bit, 0001_{5421} (one) is the same as 1001_{5421} (six) except for the first bit, and so forth. Studying Table 3.11 should reveal the various reflexive patterns found in these codes.

PRACTICE. Table 3.13 shows several decimal numbers in various BCD systems which can be used for practice in converting between these number systems.

Dec	8421	2421	Ex3	5421
57	0101 1110	10111 1101	1000 1010	1000 1010
79	0111 1001	1101 1111	1010 1100	1010 1100
28	0010 1000	0010 1110	0101 1011	0010 1011
421	0100 0010 0001	0100 0010 0001	0111 0101 0100	0100 0010 0001
903	1001 0000 0011	1111 0000 0011	1100 0011 0110	1100 0000 0011

Table 3.13: BCD Practice

ADDING BCD NUMBERS. BCD numbers can be added in either of two ways. Probably the simplest is to convert the BCD numbers to binary, add them as binary numbers, and then convert the sum back to BCD. However, it is possible to add two BCD numbers without converting. When two BCD numbers are added such that the result is less than ten, then the addition is the same as for binary numbers:

```
  0101   (5)
 +0010   (2)
  0111   (7)
```

However, four-bit binary numbers greater than 1001_2 (that is: 9_{10}) are invalid BCD codes, so adding two BCD numbers where the result is greater than nine requires a bit more effort:

```
  0111   (7)
 +0101   (5)
  1100   (12 -- not valid in BCD)
```

When the sum is greater than nine, then six must be added to that result since there are six invalid binary codes in BCD.

```
  1100   (12 -- from previous addition)
 +0110   (6)
1 0010   (12 in BCD)
```

When adding two-digit BCD numbers, start with the Least Significant Nibble (LSN), the right-most nibble, then add the nibbles with carry bits from the right. Here are some examples to help clarify this concept:

```
  0101 0010   (52)
 +0011 0110   (36)
  1000 1000   (88 in BCD)

  0101 0010   (52)
 +0101 0110   (56)
  1010 1000   (1010, MSN, invalid BCD code)
 +0110 0000   (Add 6 to invalid code)
1 0000 1000   (108 in BCD)

  0101 0101   (55)
 +0101 0110   (56)
  1010 1011   (both nibbles invalid BCD code)
 +0000 0110   (Add 6 to LSN)
  1011 0001   (1 carried over to MSN)
 +0110 0000   (Add 6 to MSN)
1 0001 0001   (111 in BSD)
```

NEGATIVE NUMBERS. BCD codes do not have any way to store negative numbers, so a sign nibble must be used. One approach to this problem is to use a sign-and-magnitude value where a sign nibble is prefixed onto the BCD value. By convention, a sign nibble of 0000 makes the BCD number positive while 1001 makes it negative. Thus, the BCD number 000000100111 is 27, but 100100100111 is −27.

A more mathematically rigorous, and useful, method of indicating negative BCD numbers is use the tens complement of the BCD number since BCD is a code for decimal numbers, exactly like the twos complement is used for binary numbers. It may be useful to review Section 3.2.3.1 on page 39 for information about the tens complement. In the *Natural BCD* system the tens complement is found by adding one to the nines complement, which is found by subtracting each digit of the original BCD number from nine. Here are some examples to clarify this concept:

```
0111 (7 in BCD)
0010 (2, the 9's complement of 7 since 9-7=2)
0011 (3, the 10's complement of 7, or 2+1)

0010 0100 (24 in BCD)
0111 0101 (75, the 9's complement of 24)
0111 0110 (76, the 10's complement of 24)
```

Also, some BCD code systems are designed to easily create the tens complement of a number. For example, in the 2421 BCD system the tens complement is found by nothing more than inverting the MSB. Thus, three is the tens complement of seven and in the 2421 BCD system $0011_{BCD2421}$ is the tens complement of $1101_{BCD2421}$, a difference of only the MSB. Therefore, designers creating circuits that must work with negative BCD numbers may opt to use the 2421 BCD system.

SUBTRACTING BCD NUMBERS. A BCD number can be subtracted from another by changing it to a negative number and adding. Just like in decimal, $5 - 2$ is the same as $5 + (-2)$. Either a nines or tens complement can be used to change a BCD number to its negative, but for this book, the tens complement will be used. If there is a carry-out bit then it can be ignored and the result is positive, but if there is no carry-out bit then answer is negative so the magnitude must be found by finding the tens complement of the calculated sum. Compare this process with subtracting regular binary numbers. Here are a few examples:

```
7 - 3 = 4

     0111   (7 in BCD)
    +0111   (add the 10's complement of 3)
     1110   (invalid BCD code)
    +0110   (add 6 to invalid BCD code)
   1 0100   (4 - drop the carry bit)

7 - 9 = -2

     0111   (7 in BCD)
    +0001   (10's complement of 9)
     1000   (valid BCD code)
     0010   (10's complement)

32 - 15 = 17

     0011 0010   (32 in BCD)
    +1000 0101   (10's complement of 15)
     1011 0111   (MSB is invalid BCD code)
    +0110 0000   (add 6 to MSB)
   1 0001 0111   (17, drop the carry bit)

427 - 640 = -213

     0100 0010 0111   (427 in BCD)
    +0011 0110 0000   (10's complement of 640)
     0111 1000 0111   (no invalid BCD code)
     0010 0001 0011   (10's complement)

369 - 532 = -163

     0011 0110 1001   (369 in BCD)
    +0100 0110 1000   (10's complement of 532)
     0111 1100 0001   (two invalid BCD codes)
    +0000 0110 0110   (add 6 to invalid codes)
     1000 0011 0111   (sum)
     0001 0110 0011   (10's complement)
```

Here are some notes on the last example: adding the LSN yields $1001 + 1000 = 10001$. The initial one is ignored, but this is an invalid BCD code so this byte needs to be corrected by adding six to it. Then the result of that addition includes an understood carry into the next nibble after the correction is applied. In the same way, the middle nibble was corrected: $1100 + 0110 = 10011$ but the initial one in this answer is carried to the Most Significant Nibble (MSN) and added there.

3.6.2.3 *Gray Code*

It is often desirable to use a wheel to encode digital input for a circuit. As an example, consider the tuning knob on a radio. The knob is attached to a shaft that has a small, clear disk which is etched with a code, similar to Figure 3.1. As the disk turns, the etched patterns pass or block a laser beam from reaching an optical sensor, and that pass/block pattern is encoded into binary input.

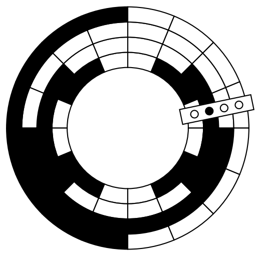
Figure 3.1: Optical Disc

One of the most challenging aspects of using a mechanical device to encode binary is ensuring that the input is stable. As the wheel turns past the light beam, if two of the etched areas change at the same time (thus, changing two bits at once), it is certain that the input will fluctuate between those two values for a tiny, but significant, period of time. For example, imagine that the encoded circuit changes from 1111 to 0000 at one time. Since it is impossible to create a mechanical wheel precise enough to change those bits at exactly the same moment in time (remember that the light sensors will "see" an input several million times a second), as the bits change from 1111 to 0000 they may also change to 1000 or 0100 or any of dozens of other possible combinations for a few microseconds. The entire change may form a pattern like 1111 − 0110 − 0010 − 0000 and that instability would be enough to create havoc in a digital circuit.

The solution to the stability problem is to etch the disk with a code designed in such a way that only one bit changes at a time. The code used for that task is the Gray code. Additionally, a Gray code is cyclic, so when it reaches its maximum value it can cycle back to its minimum value by changing only a single bit. In Figure 3.1, each of the concentric rings encodes one bit in a four-bit number. Imagine that the disk is rotating past the fixed laser beam reader —the black areas ("blocked light beam") change only one bit at a time, which is characteristic of a Gray code pattern.

It is fairly easy to create a Gray code from scratch. Start by writing two bits, a zero and one:

```
0
1
```

Then, reflect those bits by writing them in reverse order underneath the original bits:

```
0
1
```

```
-----
  1
  0
```

Next, prefix the top half of the group with a zero and the bottom half with a one to get a two-bit Gray code.

```
00
01
11
10  (2-bit Gray code)
```

Now, reflect all four values of the two-bit Gray code.

```
  00
  01
  11
  10
------
  10
  11
  01
  00
```

Next, prefix the top half of the group with a zero and the bottom half with a one to get a three-bit Gray code.

```
000
001
011
010
110
111
101
100  (3-bit Gray code)
```

Now, reflect all eight values of a three-bit Gray code.

```
  000
  001
  011
  010
  110
  111
  101
  100
-------
  100
```

```
101
111
110
010
011
001
000
```

Finally, prefix the top half of the group with a zero and the bottom half with a one to get a four-bit Gray code.

```
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000  (four-bit Gray code)
```

The process of reflecting and prefixing can continue indefinitely to create a Gray code of any desired bit length. Of course, Gray code tables are also available in many different bit lengths. Table 3.14 contains a two-bit, three-bit, and four-bit Gray code:

2-Bit Code	3-Bit Code	4-Bit Code
00	000	0000
01	001	0001
11	011	0011
10	010	0010
	110	0110
	111	0111
	101	0101
	100	0100
		1100
		1101
		1111
		1110
		1010
		1011
		1001
		1000

Table 3.14: Gray Codes