CHAPTER TWO

Numbering Systems

Chapter one discussed how computers remember numbers using transistors, tiny devices that act like switches with only two positions, on or off. A single transistor, therefore, can only remember one of two possible numbers, a one or a zero. This isn't useful for anything more complex than controlling a light bulb, so for larger values, transistors are grouped together so that their combination of ones and zeros can be used to represent larger numbers.

This chapter discusses some of the methods that are used to represent numbers with groups of transistors or ***bits***. The reader will also be given methods for calculating the minimum and maximum values of each representation based on the number of bits in the group.

2.1 Unsigned Binary Counting

The simplest form of numeric representation with bits is ***unsigned binary***. When we count upward through the positive integers using decimal, we start with a 0 in the one's place and increment that value until we reach the upper limit of a single digit, i.e., 9. At that point, we've run out of the "symbols" we use to count, and we need to increment the next digit, the ten's place. We then reset the one's place to zero, and start the cycle again.

Ten's place	One's place
	0
	1
	2
	3
	:
	8
	9
1	0

Figure 2-1 Counting in Decimal

Since computers do not have an infinite number of transistors, the number of digits that can be used to represent a number is limited. This

would be like saying we could only use the hundreds, tens, and ones place when counting in decimal.

This has two results. First, it limits the number of values we can represent. For our example where we are only allowed to count up to the hundreds place in decimal, we would be limited to the range of values from 0 to 999.

Second, we need a way to show others that we are limiting the number of digits. This is usually done by adding leading zeros to the number to fill up any unused places. For example, a decimal 18 would be written 018 if we were limited to three decimal digits.

Counting with bits, hereafter referred to as counting in binary, is subject to these same issues. The only difference is that decimal uses ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) while binary only uses two symbols (0 and 1).

To begin with, Figure 2-2 shows that when counting in binary, we run out of symbols quickly requiring the addition of another "place" after only the second increment.

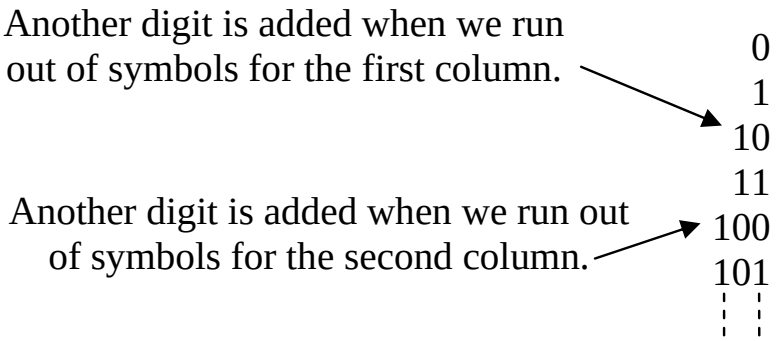


Figure 2-2 Counting in Binary

If we were counting using four bits, then the sequence would look like: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111. Notice that when restricted to four bits, we reach our limit at 1111, which happens to be the fifteenth value. It should also be noted that we ended up with 2 x 2 x 2 x 2 = 16 different values. With two symbols for each bit, we have 2^n possible combinations of symbols where n represents the number of bits.

In decimal, we know what each digit represents: ones, tens, hundreds, thousands, etc. How do we figure out what the different digits in binary represent? If we go back to decimal, we see that each place can contain one of ten digits. After the ones digit counts from 0 to

9, we need to increment the tens place. Subsequently, the third place is incremented after 9 tens and 9 ones, i.e., 99 increments, have been counted. This makes it the hundreds place.

In binary, the rightmost place is considered the ones place just like decimal. The next place is incremented after the ones place reaches 1. This means that the second place in binary represents the value after 1, i.e., a decimal 2. The third place is incremented after a 1 is in both the ones place and the twos place, i.e., we've counted to a decimal 3. Therefore, the third place represents a decimal 4. Continuing this process shows us that each place in binary represents a successive power of two.

Figure 2-3 uses 5 bits to count up to a decimal 17. Examine each row where a single one is present in the binary number. This reveals what that position represents. For example, a binary 01000 is shown to be equivalent to a decimal 8. Therefore, the fourth bit position from the right is the 8's position.

Decimal value	Binary value
0	00000
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000

Decimal value	Binary value
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001

Figure 2-3 Binary-Decimal Equivalents from 0 to 17

This information will help us develop a method for converting unsigned binary numbers to decimal and back to unsigned binary.

Some of you may recognize this as "base-2" math. This gives us a method for indicating which representation is being used when writing a number down on paper. For example, does the number 100 represent a decimal value or a binary value? Since binary is base-2 and decimal is base-10, a subscript "2" is placed at the end of all binary numbers in

this book and a subscript "10" is placed at the end of all decimal numbers. This means a binary 100 should be written as 100_2 and a decimal 100 should be written as 100_{10}.

2.2 Binary Terminology

When writing values in decimal, it is common to separate the places or positions of large numbers in groups of three digits separated by commas. For example, 345323745_{10} is typically written $345,323,745_{10}$ showing that there are 345 millions, 323 thousands, and 745 ones. This practice makes it easier to read and comprehend the magnitude of the numbers. Binary numbers are also divided into components depending on their application. Each binary grouping has been given a name.

To begin with, a single place or position in a binary number is called a ***bit***, short for binary digit. For example, the binary number 0110_2 is made up of four bits. The rightmost bit, the one that represents the ones place, is called the ***Least Significant Bit or LSB***. The leftmost bit, the one that represents the highest power of two for that number, is called the ***Most Significant Bit or MSB***. Note that the MSB represents a bit position. It doesn't mean that a '1' must exist in that position.

The next four terms describe how bits might be grouped together.

- ***Nibble*** – A four bit binary number
- ***Byte*** – A unit of storage for a single character, typically an eight bit (2 nibble) binary number (short for binary term)
- ***Word*** – Typically a sixteen bit (2 byte) binary number
- ***Double Word*** – A thirty-two bit (2 word) binary number

The following are some examples of each type of binary number.

Bit	1_2
Nibble	1010_2
Byte	10100101_2
Word	1010010111110000_2
Double Word	$10100101111100001100111011101101_2$

2.3 Unsigned Binary to Decimal Conversion

As shown in section 2.1, each place or position in a binary number corresponds to a specific power of 2 starting with the rightmost bit

which represents 2^0=1. It is through this organization of the bits that we will convert binary numbers to their decimal equivalent. Figure 2-4 shows the bit positions and the corresponding powers of two for each bit in positions 0 through 7.

Numbered bit position	7	6	5	4	3	2	1	0
Corresponding power of 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal equivalent of power of 2	128	64	32	16	8	4	2	1

Figure 2-4 Values Represented By Each of the First 8 Bit Positions

To begin converting an unsigned binary number to decimal, identify each bit position that contains a 1. It is important to note that we number the bit positions starting with 0 identifying the rightmost bit.

Next, add the powers of 2 for each position containing a 1. This sum is the decimal equivalent of the binary value. An example of this process is shown in Figure 2-5 where the binary number 10110100_2 is converted to its decimal equivalent.

Bit Position	7	6	5	4	3	2	1	0
Binary Value	1	0	1	1	0	1	0	0

$$10110100_2 = 2^7 + 2^5 + 2^4 + 2^2$$
$$= 128_{10} + 32_{10} + 16_{10} + 4_{10}$$
$$= 180_{10}$$

Figure 2-5 Sample Conversion of 10110100_2 to Decimal

This brings up an important issue when representing numbers with a computer. Note that when a computer stores a number, it uses a limited number of transistors. If, for example, we are limited to eight transistors, each transistor storing a single bit, then we have an upper limit to the size of the decimal value we can store.

The largest unsigned eight bit number we can store has a 1 in all eight positions, i.e., 11111111_2. This number cannot be incremented without forcing an overflow to the next highest bit. Therefore, the largest decimal value that 8 bits can represent in unsigned binary is the sum of all powers of two from 0 to 7.

$$\begin{aligned} 11111111_2 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255_{10} \end{aligned}$$

If you add one to this value, the result is 256 which is 2^8, the power of two for the next bit position. This makes sense because if you add 1 to 11111111_2, then beginning with the first column, 1 is added to 1 giving us a result of 0 with a 1 carry to the next column. This propagates to the MSB where a final carry is passed to the ninth bit. The final value is then $100000000_2 = 256_{10}$.

$$11111111_2 + 1 = 100000000_2 = 256_{10} = 2^8$$

Therefore, the maximum value that can be represented with 8 bits in unsigned binary is $2^8 - 1 = 255$.

It turns out that the same result is found for any number of bits. The maximum value that can be represented with n bits in unsigned binary is $2^n - 1$.

$$\text{Max unsigned binary value represented with n bits} = 2^n - 1 \qquad (2.1)$$

We can look at this another way. Each digit of a binary number can take on 2 possible values, 0 and 1. Since there are two possible values for the first digit, two possible values for the second digit, two for the third, and so on until you reach the n-th bit, then we can find the total number of possible combinations of 1's and 0's for n-bits by multiplying 2 n-times, i.e., 2^n.

How does this fit with our upper limit of 2^n-1? Where does the "-1" come from? Remember that counting using unsigned binary integers begins at 0, not 1. Giving 0 one of the bit patterns takes one away from the maximum value.

2.4 Decimal to Unsigned Binary Conversion

Converting from decimal to unsigned binary is a little more complicated, but it still isn't too difficult. Once again, there is a well-defined process.

To begin with, it is helpful to remember the powers of 2 that correspond to each bit position in the binary numbering system. These were presented in Figure 2-4 for the powers of 2^0 up to 2^7.

What we need to do is separate the decimal value into its power of 2 components. The easiest way to begin is to find the largest power of 2 that is less than or equal to our decimal value. For example if we were converting 75_{10} to binary, the largest power of 2 less than or equal to 75_{10} is $2^6 = 64$.

The next step is to place a 1 in the location corresponding to that power of 2 to indicate that this power of 2 is a component of our original decimal value.

Next, subtract this first power of 2 from the original decimal value. In our example, that would give us $75_{10} – 64_{10} = 11_{10}$. If the result is not equal to zero, go back to the first step where we found the largest power of 2 less than or equal to the new decimal value. In the case of our example, we would be looking for the largest power of 2 less than or equal to 11_{10} which would be $2^3 = 8$.

When the result of the subtraction reaches zero, and it eventually will, then the conversion is complete. Simply place 0's in the bit positions that do not contain 1's. Figure 2-6 illustrates this process using a flowchart.

If you get all of the way to bit position zero and still have a non-zero result, then one of two things has happened. Either there was an error in one of your subtractions or you did not start off with a large enough number of bits. Remember that a fixed number of bits, n, can only represent an integer value up to $2^n – 1$. For example, if you are trying to convert 312_{10} to unsigned binary, eight bits will not be enough because the highest value eight bits can represent is $2^8 – 1 = 255_{10}$. Nine bits, however, will work because its maximum unsigned value is $2^9 – 1 = 511_{10}$.

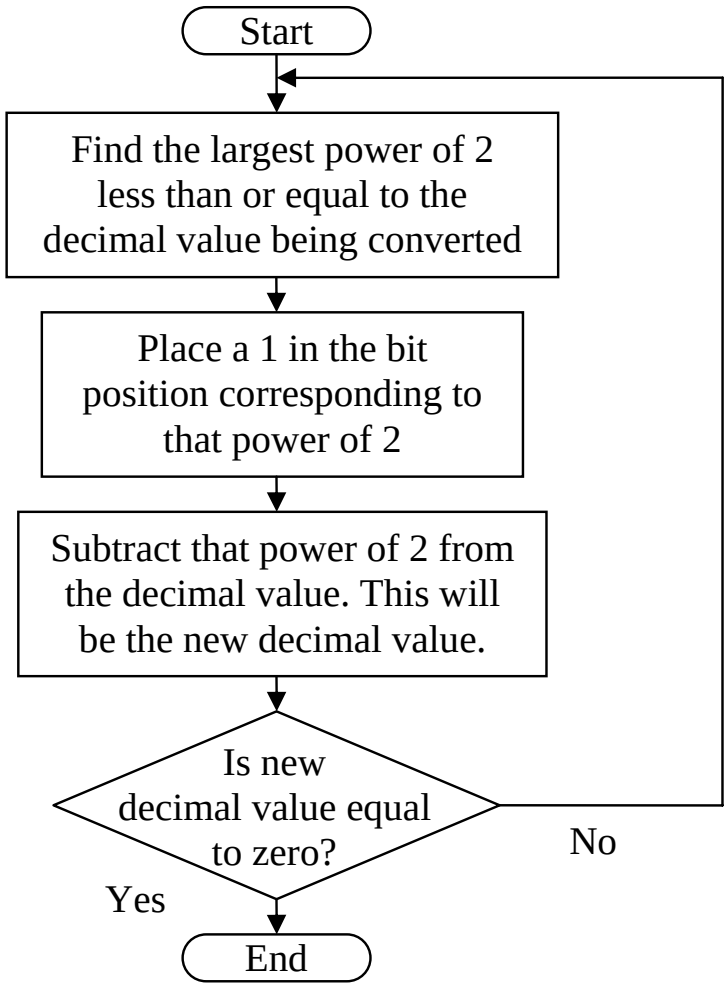


Figure 2-6 Decimal to Unsigned Binary Conversion Flow Chart

Example

Convert the decimal value 133_{10} to an 8 bit unsigned binary number.

Solution

Since 133_{10} is less than $2^8 - 1 = 255$, 8 bits will be sufficient for this conversion. Using Figure 2-4, we see that the largest power of 2 less than or equal to 133_{10} is $2^7 = 128$. Therefore, we place a 1 in bit position 7 and subtract 128 from 133.

Bit position	7	6	5	4	3	2	1	0
	1							

$$133 - 128 = 5$$

Our new decimal value is 5. Since this is a non-zero value, our next step is to find the largest power of 2 less than or equal to 5. That would be $2^2 = 4$. So we place a 1 in the bit position 2 and subtract 4 from 5.

Bit position	7	6	5	4	3	2	1	0
	1					1		

$$5 - 4 = 1$$

Our new decimal value is 1, so find the largest power of 2 less than or equal to 1. That would be $2^0 = 1$. So we place a 1 in the bit position 0 and subtract 1 from 1.

Bit position	7	6	5	4	3	2	1	0
	1					1		1

$$1 - 1 = 0$$

Since the result of our last subtraction is 0, the conversion is complete. Place zeros in the empty bit positions.

Bit position	7	6	5	4	3	2	1	0
	1	0	0	0	0	1	0	1

And the result is:

$$133_{10} = 10000101_2$$

2.5 Binary Representation of Analog Values

Converting unsigned (positive) integers to binary is only one of the many ways that computers represent values using binary bits. This chapter still has two more to cover, and Chapter 3 will cover even more.

This section focuses on the problems and solutions of trying to map real world values such as temperature or weight from a specified range to a binary integer. For example, a computer that uses 8 bits to represent an integer is capable of representing 256 individual values from 0 to 255. Temperature, however, is a floating-point value with

CHAPTER THREE

Binary Math and Signed Representations

Representing numbers with bits is one thing. Doing something with them is an entirely different matter. This chapter discusses some of the basic mathematical operations that computers perform on binary numbers along with the binary representations that support those operations. These concepts will help programmers better understand the limitations of doing math with a processor, and thereby allow them to better handle problems such as the upper and lower limits of variable types, mathematical overflow, and type casting.

3.1 Binary Addition

Regardless of the numbering system, the addition of two numbers with multiple digits is performed by adding the corresponding digits of a single column together to produce a single digit result. For example, 3 added to 5 using the decimal numbering system equals 8. The 8 is placed in the same column of the result where the 3 and 5 came from. All of these digits, 3, 5, and 8, exist in the decimal numbering system, and therefore can remain in a single column.

In some cases, the result of the addition in a single column might be more than 9 making it necessary to place a '1' overflow or carry to the column immediately to the left. If we add 6 to 5 for example, we get 11 which is too large to fit in a single decimal digit. Therefore, 10 is subtracted from the result leaving 1 as the new result for that column. The subtraction of 10 is compensated for by placing a carry in the next highest column, the ten's place. Another way of saying this is that 6 added to 5 equals 1 with a carry of 1. It is important to note that the addition of two digits in decimal can never result in a value greater than 18. Therefore, the carry to the next highest position will never be larger than 1.

Binary addition works the same way except that we're limited to two digits. Three of the addition operations, 0+0, 0+1, and 1+0, result in 0 or 1, digits that already exist in the binary numbering system. This means no carry will be needed.

Adding 1 to 1, however, results in a decimal 2, a digit which does not exist in binary. In this case, we need to create a carry or overflow that will go to the next column.

The next highest bit position represents $2^1 = 2$. Just as we did with decimal, we subtract one instance of the next highest bit position from our result. In the case of 1+1=2, we subtract 2 from 2 and get 0. Therefore, 0 is the result that is placed in the current column, and the subtraction of 2 becomes a carry to the next column. Therefore, 1+1 in binary equals 0 with a carry of 1. Each of the possible binary additions of two variables is shown in Figure 3-1.

```
    0        0        1        1
  + 0      + 1      + 0      + 1
  ---      ---      ---      ---
    0        1        1       10
```

Figure 3-1 Four Possible Results of Adding Two Bits

The last addition $1_2 + 1_2 = 10_2$ is equivalent to the decimal addition $1_{10} + 1_{10} = 2_{10}$. Converting 2_{10} to binary results in 10_2, the result shown in the last operation of Figure 3-1, which confirms our work.

Now we need to figure out how to handle a carry from a previous column. In decimal, a carry from a previous column is simply added to the next column. This is the same as saying that we are adding three digits where one of the digits, the carry, is always a one.

In binary, accounting for a carry adds four new scenarios to the original four shown in Figure 3-1. Just like decimal, it is much like adding three values together: 1+0+0, 1+0+1, 1+1+0, or 1+1+1. The four additional cases where a carry is added from the previous column are shown in Figure 3-2.

```
Previous          1        1        1
 Carry →    1        1        1        1
            0        0        1        1
          + 0      + 1      + 0      + 1
          ---      ---      ---      ---
            1       10       10       11
```

Figure 3-2 Four Possible Results of Adding Two Bits with Carry

The second and third cases are similar to the last case presented in Figure 3-1 where two 1's are added together to get a result of 0 with a carry. The last case in Figure 3-2, however, has three 1's added together which equals 3_{10}. Subtracting 2 from this result places a new result of 1 in the current column and sends a carry to the next column. And just as in decimal addition, the carry in binary is never greater than 1.

Now let's try to add binary numbers with multiple digits. The example shown below presents the addition of 10010110_2 and 00101011_2. The highlighted values are the carries from the previous column's addition, and just as in decimal addition, they are added to the next most significant digit/bit.

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

3.2 Binary Subtraction

Just as with addition, we're going to use the decimal numbering system to illustrate the process used in the binary numbering system for subtraction.

There are four possible cases of single-bit binary subtraction: 0 – 0, 0 – 1, 1 – 0, and 1 – 1. As long as the value being subtracted from (the minuend) is greater than or equal to the value subtracted from it (the subtrahend), the process is contained in a single column.

```
Minuend →      0        1        1
Subtrahend →   - 0      - 0      - 1
               ---      ---      ---
               0        1        0
```

But what happens in the one case when the minuend is less than the subtrahend? As in decimal, a borrow must be taken from the next most significant digit. The same is true for binary.

A "borrow" is made from the next highest bit position

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

Pulling 1 from the next highest column in binary allows us to add 10_2 or a decimal 2 to the current column. For the previous example, 10_2 added to 0 gives us 10_2 or a decimal 2. When we subtract 1 from 2, the result is 1.

Now let's see how this works with a multi-bit example.

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

Starting at the rightmost bit, 1 is subtracted from 1 giving us zero. In the next column, 0 is subtracted from 1 resulting in 1. We're okay so far with no borrows required. In the next column, however, 1 is subtracted from 0. Here we need to borrow from the next highest digit.

The next highest digit is a 1, so we subtract 1 from it and add 10 to the digit in the 2^2 column. (This appears as a small "1" placed before the 0 in the minuend's 2^2 position.) This makes our subtraction 10 - 1 which equals 1. Now we go to the 2^3 column. After the borrow, we have 0 – 0 which equals 0.

We need to make a borrow again in the third column from the left, the 2^6 position, but the 2^7 position of the minuend is zero and does not have anything to borrow. Therefore, the next highest digit of the minuend, the 2^8 position, is borrowed from. The borrow is then cascaded down until it reaches the 2^6 position so that the subtraction may be performed.

3.3 Binary Complements

In decimal arithmetic, every number has an additive complement, i.e., a value that when added to the original number results in a zero. For example, 5 and -5 are additive complements because 5 + (-5) = 0. This section describes the two primary methods used to calculate the complements of a binary value.

3.3.1 One's Complement

When asked to come up with a pattern of ones and zeros that when added to a binary value would result in zero, most people respond with, "just flip each bit in the original value." This "inverting" of each bit, substituting 1's for all of the 0's and 0's for all of the 1's, results in the ***1's complement*** of the original value. An example is shown below.

Previous value	1	0	0	1	0	1	1	1
1's complement	0	1	1	0	1	0	0	0

The 1's complement of a value is useful for some types of digital functions, but it doesn't provide much of a benefit if you are looking for the additive complement. See what happens when we add a value to its 1's complement.

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

If the two values were additive complements, the result should be zero, right? Well, that takes us to the 2's complement.

3.3.2 Two's Complement

The result of adding an n-bit number to its one's complement is always an n-bit number with ones in every position. If we add 1 to that result, our new value is an n-bit number with zeros in every position and an overflow or carry to the next highest position, the $(n+1)^{th}$ column which corresponding to 2^n. For our 8-bit example above, the result of adding 10010110_2 to 01101001_2 is 11111111_2. Adding 1 to this number gives us 00000000_2 with an overflow carry of 1 to the ninth or 2^8 column. If we restrict ourselves to 8 bits, this overflow carry can be ignored.

This gives us a method for coming up with the additive complement called the ***2's complement*** representation. The 2's complement of a value is found by first taking the 1's complement, then incrementing that result by 1. For example, in the previous section, we determined that the 1's complement of 10010111_2 is 01101000_2. If we add 1 to this value, we get:

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

Therefore, the 2's complement of 10010111_2 is 01101001_2. Let's see what happens when we try to add the value to its 2's complement.

```
1 1 1 1 1 1 1 1
  1 0 0 1 0 1 1 1
+ 0 1 1 0 1 0 0 1
  0 0 0 0 0 0 0 0
```

The result is zero! Okay, so most of you caught the fact that I didn't drop down the last carry which would've made the result 100000000_2. This is not a problem, because in the case of signed arithmetic, the carry has a purpose other than that of adding an additional digit representing the next power of two. As long as we make sure that the two numbers being added have the same number of bits, and that we keep the result to that same number of bits too, then any carry that goes beyond that should be discarded.

Actually, discarded is not quite the right term. In some cases we will use the carry as an indication of a possible mathematical error. It should not, however, be included in the result of the addition. This is simply the first of many "anomalies" that must be watched when working with a limited number of bits.

Two more examples of 2's complements are shown below.

Original value (10_{10})	0	0	0	0	1	0	1	0
1's complement	1	1	1	1	0	1	0	1
2's complement (-10_{10})	1	1	1	1	0	1	1	0

Original value (88_{10})	0	1	0	1	1	0	0	0
1's complement	1	0	1	0	0	1	1	1
2's complement (-88_{10})	1	0	1	0	1	0	0	0

Now let's see if the 2's complement representation stands up in the face of addition. If $88_{10} = 01011000_2$ and $-10_{10} = 11110110_2$, then the addition of these two numbers should equal $78_{10} = 01001110_2$.

```
1 1 1 1
  0 1 0 1 1 0 0 0
+ 1 1 1 1 0 1 1 0
  0 1 0 0 1 1 1 0
```

There is also a "short-cut" to calculating the 2's complement of a binary number. This trick can be used if you find the previous way too cumbersome or if you'd like a second method in order to verify the result you got from using the first.

The trick works by copying the zero bit values starting with the least significant bit until you reach your first binary 1. Copy that 1 too. If the least significant bit is a one, then only copy that bit. Next, invert all of the remaining bits. Figure 3-3 presents an example of the short-cut.

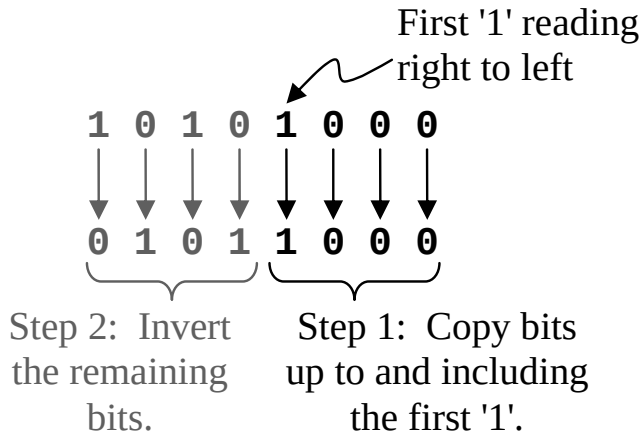


Figure 3-3 Two's Complement Short-Cut

This result matches the result for the previous example.

In decimal, the negative of 5 is -5. If we take the negative a second time, we return to the original value, e.g., the negative of -5 is 5. Is the same true for taking the 2's complement of a 2's complement of a binary number? Well, let's see.

The binary value for 45_{10} is 00101101_2. Watch what happens when we take the 2's complement twice.

Original value = 45	0	0	1	0	1	1	0	1
1's complement of 45	1	1	0	1	0	0	1	0
2's complement of 45 = -45	1	1	0	1	0	0	1	1
1's complement of -45	0	0	1	0	1	1	0	0
2's complement of -45 = 45	0	0	1	0	1	1	0	1

It worked! The second time the 2's complement was taken, the pattern of ones and zeros returned to their original values. It turns out that this is true for any binary number of a fixed number of bits.

3.3.3 Most Significant Bit as a Sign Indicator

As was stated earlier, 2's complement is used to allow the computer to represent the additive complement of a binary number, i.e., negative numbers. But there is a problem. As we showed earlier in this section, taking the 2's complement of $45_{10} = 00101101_2$ gives us $-45_{10} = 11010011_2$. But in Chapter 2, the eight bit value 11010011_2 was shown to be equal to $2^7 + 2^6 + 2^4 + 2^1 + 2^0 = 128 + 64 + 16 + 2 + 1 = 211_{10}$. So did we just prove that -45_{10} is equal to 211_{10}? Or maybe 00101101_2 is actually -211_{10}.

It turns out that when using 2's complement binary representation, half of the binary bit patterns must lose their positive association in order to represent negative numbers. So is 11010011_2 -45_{10} or 211_{10}? It turns out that 11010011_2 is one of the bit patterns meant to represent a negative number, so in 2's complement notation, $11010011_2 = -45_{10}$. But how can we tell whether a binary bit pattern represents a positive or a negative number?

From the earlier description of the 2's complement short-cut, you can see that except for two cases, the MSB of the 2's complement is always the inverse of the original value. The two cases where this isn't true are when all bits of the number except the most significant bit equal 0 and the most significant bit is a 0 or a 1. In both of these cases, the 2's complement equals the original value.

In all other cases, when we apply the shortcut we will always encounter a 1 before we get to the MSB when reading right to left. Since every bit after this one will be inverted, then the most significant bit must be inverted toggling it from its original value. If the original value has a zero in the MSB, then its 2's complement must have a one and vice versa. Because of this characteristic, the MSB of a value can be used to indicate whether a number is positive or negative and is called a ***sign bit***.

A binary value with a 0 in the MSB position is considered positive and a binary value with a 1 in the MSB position is considered negative. This makes it vital to declare the number of bits that a signed binary number uses. If this information is not given, then the computer or the user looking at a binary number will not know which bit is the MSB.

Since the MSB is being used to indicate the sign of a signed binary number, it cannot be used to represent a power of 2, i.e., if a number is said to represent a 2's complement value, only n-1 of its n bits can be

used to determine the magnitude since the MSB is used for the sign. This cuts in half the number of positive integers n bits can represent.

And the special cases? Well, a binary number with all zeros is equal to a decimal 0. Taking the negative of zero still gives us zero. The other case is a bit trickier. In the section on minimums and maximums, we will see that an n-bit value with an MSB equal to one and all other bits equal to zero is a negative number, specifically, $-2^{(n-1)}$. The largest positive number represented in 2's complement has an MSB of 0 with all the remaining bits set to one. This value equals $2^{(n-1)} - 1$. Therefore, since $2^{(n-1)} > 2^{(n-1)} - 1$, we can see that there is no positive equivalent to the binary number $100\ldots00_2$.

3.3.4 Signed Magnitude

A second, less useful way to represent positive and negative binary numbers is to take the MSB and use it as a sign bit, much like a plus or minus sign, and leave the remaining bits to represent the magnitude. The representation is called ***signed magnitude*** representation. For example, –45 and +45 would be identical in binary except for the MSB which would be set to a 1 for –45 and a 0 for +45. This is shown below for an 8-bit representation.

$+45_{10}$ in binary	0	0	1	0	1	1	0	1
-45_{10} using signed magnitude	1	0	1	0	1	1	0	1

3.3.5 MSB and Number of Bits

Since the MSB is necessary to indicate the sign of a binary value, it is vital that we know how many bits a particular number is being represented with so we know exactly where the MSB is. In other words, the leading zeros of a binary value may have been removed making it look like the binary value is negative since it starts with a one.

For example, if the binary value 10010100_2 is assumed to be an 8-bit signed number using 2's complement representation, then converting it to decimal would give us -108_{10}. (We will discuss converting signed values to decimal later in this chapter.) If, however, it was a 10-bit number, then the MSB would be 0 and it would convert to the positive value 148_{10}.

3.3.6 Issues Surrounding the Conversion of Binary Numbers

Since computers don't use an infinite number of bits to represent values, the software must know two things before it can interpret a binary value: the number of bits and the type of binary representation being used. This usually is confusing for the novice.

Identifying 10100110_2 as an 8-bit number isn't enough. Note that the MSB is equal to 1. Therefore, this value represents one number in unsigned binary, another number in 2's complement, and yet a third in signed magnitude.

First, let's do the conversion of 10100110_2 assuming it is an 8-bit, unsigned binary like those described in Chapter 2.

$$10100110_2 = 2^7 + 2^5 + 2^2 + 2^1 = 128 + 32 + 4 + 2 = 166_{10}$$

Now let's do the conversion in 2's complement. Before we do, however, let's examine the process. First, if the MSB is equal to 0, then the value is a positive number. In 2's complement notation, positive numbers look just like unsigned binary and should be treated exactly the same when performing a conversion to decimal.

If, however, the MSB is equal to 1, then the value represented by this pattern of ones and zeros is negative. To turn it into a negative number, someone had to apply the process of taking the 2’s complement to the original positive value. Therefore, we must remove the negative sign before we do the conversion.

It was shown earlier how a second application of the 2's complement conversion process returns the number back to its original positive value. If taking the 2's complement of a negative number returns it to its positive value, then the positive value can be converted to decimal using the same process used for an unsigned binary value. Adding a negative sign to the decimal result completes the conversion. Figure 3-4 presents a flow chart showing this process graphically.

A second method of converting an n-bit 2's complement value to decimal is to perform the conversion as you would an unsigned binary value except that the MSB digit is treated as -2^{n-1} instead of 2^{n-1}. For example, the MSB of an 8-bit 2's complement value would represent $-2^{-7} = -128$.

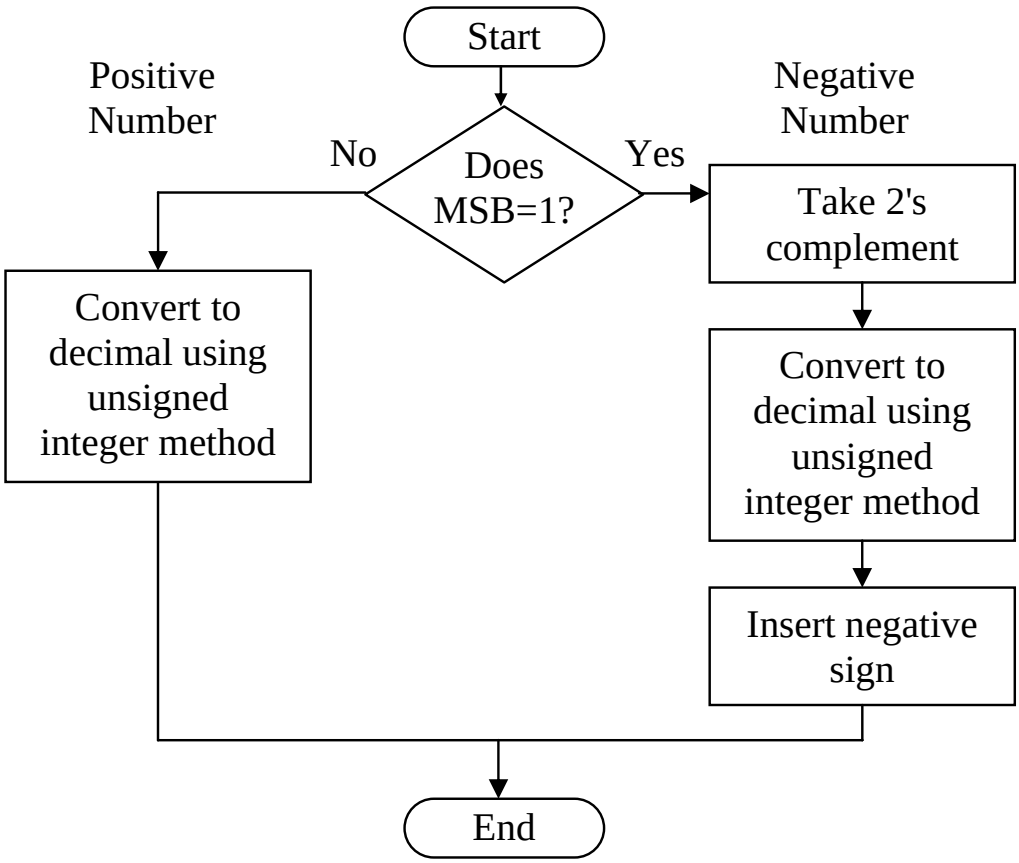


Figure 3-4 Converting a Two's Complement Number to a Decimal

In the case of 10100110_2, the MSB is a 1. Therefore, it is a negative number. By following the right branch of the flowchart in Figure 3-4, we see that we must take the two's complement to find the positive counterpart for our negative number.

Negative value	1	0	1	0	0	1	1	0
1's comp. of negative value	0	1	0	1	1	0	0	1
2's comp. of negative value	0	1	0	1	1	0	1	0

Now that we have the positive counterpart for the 2's complement value of the negative number 10100110_2, we convert it to decimal just as we did with the unsigned binary value.

$$01011010_2 = 2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 = 90_{10}$$

Since the original 2's complement value was negative to begin with, the value 10100110_2 in 8-bit, 2's complement form is –90.

We can duplicate this result using the second method of conversion, i.e., converting 10100110_2 using the unsigned binary method while treating the MSB as -2^{-7}. In this case, there is a 1 in the -2^{-7}, 2^5, 2^2, and 2^1 positions.

$$10100110_2 = 2^{-7} + 2^5 + 2^2 + 2^1 = -128 + 32 + 4 + 2 = -90_{10}$$

Next, let's do the conversion assuming 10100110_2 is in 8-bit signed magnitude where the MSB represents the sign bit. As with the 2's complement form, an MSB of 1 means that the number is negative.

The conversion of a signed magnitude binary number to decimal is different than 2's complement. In the case of signed magnitude, remove the MSB and convert the remaining bits using the same methods used to convert unsigned binary to decimal. When done, place a negative sign in front of the decimal result only if the MSB equaled 1.

Meaning of bit position	Sign	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary value	1	0	1	0	0	1	1	0

To convert this value to a positive number, remove the sign bit. Next, calculate the magnitude just as we would for the unsigned case.

$$0100110_2 = 2^5 + 2^2 + 2^1 = 32 + 4 + 2 = 38_{10}$$

Since the MSB of the original value equaled 1, the signed magnitude value was a negative number to begin with, and we need to add a negative sign. Therefore, 10100110_2 in 8-bit, signed magnitude representation equals -38_{10}.

But what if this binary number was actually a 10-bit number and not an 8 bit number? Well, if it's a 10 bit number (0010100110_2), the MSB is 0 and therefore it is a positive number. This makes our conversion much easier. *The method for converting a positive binary value to a decimal value is the same for all three representations.* The conversion goes something like this:

Bit position	MSB	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary value	0	0	1	0	1	0	0	1	1	0

$$0010100110_2 = 2^7 + 2^5 + 2^2 + 2^1 = 128 + 32 + 4 + 2 = 166_{10}$$

This discussion shows that it is possible for a binary pattern of ones and zeros to have three interpretations. It all depends on how the computer has been told to interpret the value.

In a programming language such as C, the way in which a computer treats a variable depends on how it is declared. Variables declared as *unsigned int* are stored in unsigned binary notation. Variables declared as *int* are treated as either 2's complement or signed magnitude depending on the processor and/or compiler.

3.3.7 Minimums and Maximums

When using a finite number of bit positions to store information, it is vital to be able to determine the minimum and maximum values that each binary representation can handle. Failure to do this might result in bugs in the software you create. This section calculates the minimum and maximum values for each of the three representations discussed in this and the previous chapter using a fixed number of bits, n.

Let's begin with the most basic representation, unsigned binary. The smallest value that can be represented with unsigned binary representation occurs when all the bits equal zero. Conversion from binary to decimal results in 0 + 0 + ... + 0 = 0. Therefore, for an n bit number:

$$\text{Minimum n-bit unsigned binary number} = 0 \qquad (3.1)$$

The largest value that can be represented with unsigned binary representation is reached when all n bits equal one. When we convert this value from binary to decimal, we get $2^{n-1} + 2^{n-2} + ... + 2^0$. As was shown in Chapter 2, adding one to this expression results in 2^n. Therefore, for an n-bit unsigned binary number, the maximum is:

$$\text{Maximum n-bit unsigned binary number} = 2^n - 1 \qquad (3.2)$$

Next, let's examine the minimum and maximum values for an n-bit 2's complement representation. Unlike the unsigned case, the lowest decimal value that can be represented with n-bits in 2's complement representation is not obvious. Remember, 2's complement uses the MSB as a sign bit. Since the lowest value will be negative, the MSB should be set to 1 (a negative value). But what is to be done with all of the remaining bits? A natural inclination is to set all the bits after the

MSB to one. This should be a really big negative number, right? Well, converting it to decimal results in something like the 8 bit example below:

2's comp. value	1	1	1	1	1	1	1	1
Intermediate 1's complement	0	0	0	0	0	0	0	0
Positive value of 2's comp.	0	0	0	0	0	0	0	1

This isn't quite what we expected. Using the 2's complement method to convert 11111111_2 to a decimal number results in -1_{10}. This couldn't possibly be the lowest value that can be represented with 2's complement.

It turns out that the lowest possible 2's complement value is an MSB of 1 followed by all zeros as shown in the 8 bit example below. For the conversion to work, you must strictly follow the sequence presented in Figure 3-4 to convert a negative 2's complement value to decimal.

2's comp. value	1	0	0	0	0	0	0	0
Intermediate 1's complement	0	1	1	1	1	1	1	1
Positive value of 2's comp.	1	0	0	0	0	0	0	0

Converting the positive value to decimal using the unsigned method shows that $10000000_2 = -2^7 = -128$. Translating this to n-bits gives us:

$$\text{Minimum n-bit 2's complement number} = -2^{(n-1)} \qquad (3.3)$$

The maximum value is a little easier to find. It is a positive number, i.e., an MSB of 0. The remaining n-1 bits are then treated as unsigned magnitude representation. Therefore, for n bits:

$$\text{Maximum n-bit 2's complement number} = 2^{(n-1)} - 1 \qquad (3.4)$$

Last of all, we have the signed magnitude representation. To determine the magnitude of a signed magnitude value, ignore the MSB and use the remaining n–1 bits to convert to decimal as if they were in unsigned representation. This means that the largest and smallest values represented with an n-bit signed magnitude number equals the positive and negative values of an (n–1)-bit unsigned binary number.

$$\text{Minimum n-bit signed magnitude number} = -(2^{(n-1)} - 1) \quad (3.5)$$

$$\text{Maximum n-bit signed magnitude number} = (2^{(n-1)} - 1) \quad (3.6)$$

As an example, Table 3-1 compares the minimum and maximum values of an 8-bit number for each of the binary representations. The last column shows the number of distinct integer values possible with each representation. For example, there are 256 integer values between 0 and 255 meaning the 8-bit unsigned binary representation has 256 possible combinations of 1's and 0's, each of which represents a different integer in the range.

Table 3-1 Representation Comparison for 8-bit Binary Numbers

Representation	Minimum	Maximum	Number of integers represented
Unsigned	0	255	256
2's Complement	-128	127	256
Signed Magnitude	-127	127	255

So why can 8-bit signed magnitude only represent 255 possible values instead of 256? It is because in signed magnitude 00000000_2 and 10000000_2 both represent the same number, a decimal 0.

3.4 Floating Point Binary

Binary numbers can also have decimal points, and to show you how, we will once again begin with decimal numbers. For decimal numbers with decimal points, the standard way to represent the digits to the right of the decimal point is to continue the powers of ten in descending order starting with -1 where 10^{-1}=1/10th = 0.1. That means that the number 6.5342 has 5 increments of 10^{-1} (tenths), 3 increments of 10^{-2} (hundredths), 4 increments of 10^{-3} (thousandths), and 2 increments of 10^{-4} (ten-thousandths). The table below shows this graphically.

Exponent	3	2	1	0	-1	-2	-3	-4
Position value	1000	100	10	1	0.1	0.01	0.001	0.0001
Sample values	0	0	0	6	5	3	4	2

Therefore, our example has the decimal value 6*1 + 5*0.1 + 3*0.01 + 4*0.001 + 2*0.0001 = 6.5342.

Binary representation of real numbers works the same way except that each position represents a power of two, not a power of ten. To convert 10.01101 to decimal for example, use descending negative powers of two to the right of the decimal point.

Exponent	2	1	0	-1	-2	-3	-4	-5
Position value	4	2	1	0.5	0.25	0.125	0.0625	0.03125
Sample values	0	1	0	0	1	1	0	1

Therefore, our example has the decimal value 0*4 + 1*2 + 0*1 +0*0.5 + 1*0.25 + 1*0.125 + 0*0.0625 + 1*0.03125 = 2.40625. This means that the method of conversion is the same for real numbers as it is for integer values; we've simply added positions representing negative powers of two.

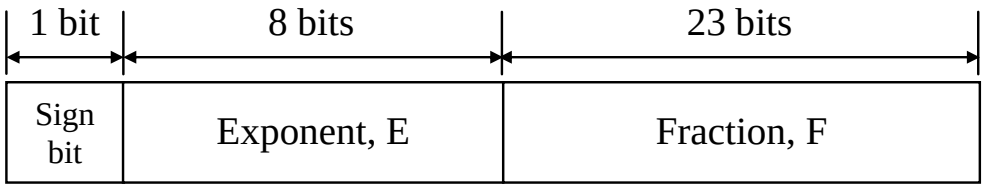
Computers, however, use a form of binary more like scientific notation to represent floating-point or real numbers. For example, with scientific notation we can represent the large value 342,370,000 as 3.4237×10^8. This representation consists of a decimal component or mantissa of 3.4237 with an exponent of 8. Both the mantissa and the exponent are signed values allowing for negative numbers and for negative exponents respectively.

Binary works the same way using 1's and 0's for the digits of the mantissa and exponent and using 2 as the multiplier that moves the decimal point left or right. For example, the binary number 100101101.010110 would be represented as:

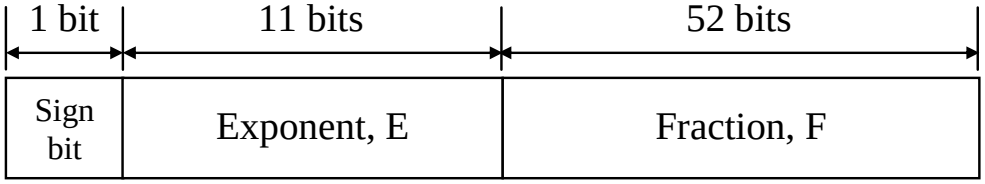
$$1.00101101010110 * 2^8$$

The decimal point is moved left for negative exponents of two and right for positive exponents of two.

The IEEE Standard 754 is used to represent real numbers on the majority of contemporary computer systems. It utilizes a 32-bit pattern to represent single-precision numbers and a 64-bit pattern to represent double-precision numbers. Each of these bit patterns is divided into three parts, each part representing a different component of the real number being stored. Figure 3-5 shows this partitioning for both single- and double-precision numbers.



a) Single-Precision



b) Double-Precision

Figure 3-5 IEEE Standard 754 Floating-Point Formats

Both formats work the same differing only by the number of bits used to represent each component of the real number. In general, the components of the single-precision format are substituted into Equation 3.7 where the sign of the value is determined by the sign bit (0 – positive value, 1 – negative value). Note that E is in unsigned binary representation.

$$(\pm)1.F \times 2^{(E-127)} \tag{3.7}$$

Equation 3.8 is used for the double-precision values.

$$(\pm)1.F \times 2^{(E-1023)} \tag{3.8}$$

In both cases, F is preceded with an implied '1' and a binary point. There are, however, some special cases. These are as follows:

- Positive, E=255, F=0: represents positive infinite;
- Negative, E=255, F=0: represents negative infinite; and
- Positive or negative, E=0, F=0: represents zero.

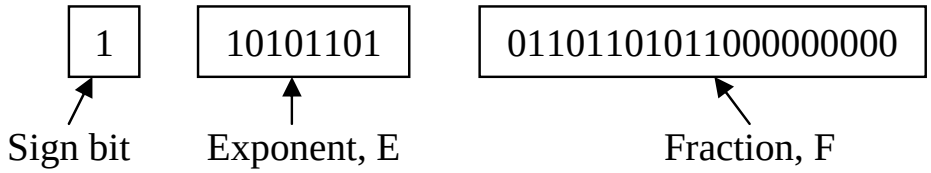
Example

Convert the 32-bit single-precision IEEE Standard 754 number shown below into its binary equivalent.

11010110101101101011000000000000

Solution

First, break the 32-bit number into its components.



A sign bit of 1 means that this will be a negative number.

The exponent, E, will be used to determine the power of two by which our mantissa will be multiplied. To use it, we must first convert it to a decimal integer using the unsigned method.

$$\begin{aligned} \text{Exponent, E} &= 10101101_2 \\ &= 2^7 + 2^5 + 2^3 + 2^2 + 2^0 \\ &= 128 + 32 + 8 + 4 + 1 \\ &= 173_{10} \end{aligned}$$

Substituting these components into Equation 3.7 gives us:

$$\begin{aligned} (\pm)1.F \times 2^{(E-127)} &= -1.01101101011000000000000 \times 2^{(173-127)} \\ &= -1.01101101011 \times 2^{46} \end{aligned}$$

Example

Create the 32-bit single-precision IEEE Standard 754 representation of the binary number 0.000000110110100101

Solution

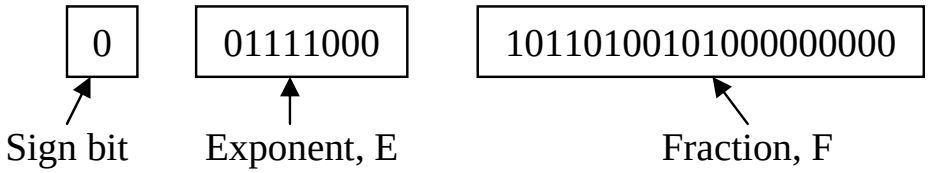
Begin by putting the binary number above into the binary form of scientific notation with a single 1 to the left of the decimal point. Note that this is done by moving the decimal point seven positions to the right giving us an exponent of –7.

$$0.000000110110100101 = 1.10110100101 \times 2^{-7}$$

The number is positive, so the sign bit will be 0. The fraction (value *after* the decimal point and not including the leading 1) is 10110100101 with 12 zeros added to the end to make it 23 bits. Lastly, the exponent must satisfy the equation:

$$E - 127 = -7$$
$$E = -7 + 127 = 120$$

Converting 120_{10} to binary gives us the 8-bit unsigned binary value 01111000_2. Substituting all of these components into the IEEE 754 format gives us:



Therefore, the answer is 00111100010110100101000000000000.

3.5 Hexadecimal Addition

At the beginning of this chapter, it was shown how binary addition (base 2) with its two digits, 1 and 0, is performed the same way decimal addition (base 10) is with its ten digits, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The only difference is the limitation placed on the addition by the number of digits. In binary, the addition of two or three ones results in a carry since the result goes beyond 1, the largest binary digit. Decimal doesn't require a carry until the result goes beyond 9.

Hexadecimal numbers (base 16) can be added using the same method. The difference is that there are more digits in hexadecimal than there are in decimal. For example, in decimal, adding 5 and 7 results in 2 with a carry to the next highest position. In hexadecimal, however, 5 added to 7 does not go beyond the range of a single digit. In this case, $5 + 7 = C_{16}$ with no carry. It isn't until a result greater than F_{16} is reached (a decimal 15_{10}) that a carry is necessary.

In decimal, if the result of an addition is greater than 9, subtract 10_{10} to get the result for the current column and add a carry to the next column. In binary, when a result is greater than 1, subtract 10_2 (i.e., 2_{10}) to get the result for the current column then add a carry to the next

column. In hexadecimal addition, if the result is greater than F_{16} (15_{10}) subtract 10_{16} (16_{10}) to get the result for the current column and add a carry to the next column.

$$D_{16} + 5_{16} = 13_{10} + 5_{10} = 18_{10}$$

By moving a carry to the next highest column, we change the result for the current column by subtracting 16_{10}.

$$\begin{aligned} 18_{10} &= 2_{10} + 16_{10} \\ &= 2_{16} \text{ with a carry to the next column} \end{aligned}$$

Therefore, D_{16} added to 5_{16} equals 2_{16} with a carry to the next column.

Just like decimal and binary, the addition of two hexadecimal digits never generates a carry greater than 1. The following shows how adding the largest hexadecimal digit, F_{16}, to itself along with a carry from the previous column still does not require a carry larger than 1 to the next highest column.

$$\begin{aligned} F_{16} + F_{16} + 1 &= 15_{10} + 15_{10} + 1 = 31_{10} \\ &= 15_{10} + 16_{10} \\ &= F_{16} \text{ with a 1 carry to the next column} \end{aligned}$$

When learning hexadecimal addition, it might help to have a table showing the hexadecimal and decimal equivalents such as that shown in Table 3-2. This way, the addition can be done in decimal, the base with which most people are familiar, and then the result can be converted back to hex.

Table 3-2 Hexadecimal to Decimal Conversion Table

Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
0_{16}	0_{10}	4_{16}	4_{10}	8_{16}	8_{10}	C_{16}	12_{10}
1_{16}	1_{10}	5_{16}	5_{10}	9_{16}	9_{10}	D_{16}	13_{10}
2_{16}	2_{10}	6_{16}	6_{10}	A_{16}	10_{10}	E_{16}	14_{10}
3_{16}	3_{10}	7_{16}	7_{10}	B_{16}	11_{10}	F_{16}	15_{10}

Example

Add $3DA32_{16}$ to $4292F_{16}$.

Solution

Just like in binary and decimal, place one of the numbers to be added on top of the other so that the columns line up.

```
  3 D A 3 2
+ 4 2 9 2 F
-----------
```

Adding 2_{16} to F_{16} goes beyond the limit of digits hexadecimal can represent. It is equivalent to $2_{10} + 15_{10}$ which equals 17_{10}, a value greater than 16_{10}. Therefore, we need to subtract 10_{16} (16_{10}) giving us a result of 1 with a carry into the next position.

```
      1
  3 D A 3 2
+ 4 2 9 2 F
-----------
        1
```

For the next column, the 16^1 position, we have 1 + 3 + 2 which equals 6. This result is less than 16_{10}, so there is no carry to the next column.

```
      1
  3 D A 3 2
+ 4 2 9 2 F
-----------
      6 1
```

The 16^2 position has $A_{16} + 9_{16}$ which in decimal is equivalent to $10_{10} + 9_{10} = 19_{10}$. Since this is greater than 16_{10}, we must subtract 16_{10} to get the result for the 16^2 column and add a carry in the 16^3 column.

```
    1   1
  3 D A 3 2
+ 4 2 9 2 F
-----------
    3 6 1
```

For the 16^3 column, we have $1_{16} + D_{16} + 2_{16}$ which is equivalent to $1_{10} + 13_{10} + 2_{10} = 16_{10}$. This gives us a zero for the result in the 16^3 column with a carry.

```
  1 1   1
  3 D A 3 2
+ 4 2 9 2 F
-----------
    0 3 6 1
```

Last of all, 1 + 3 + 4 = 8 which is the same in both decimal and hexadecimal, so the result is $3DA32_{16} + 4292F_{16} = 80361_{16}$:

```
  1 1   1
  3 D A 3 2
+ 4 2 9 2 F
-----------
  8 0 3 6 1
```

3.6 BCD Addition

When we introduced Binary Coded Decimal numbers, we said that the purpose of these numbers was to provide a quick conversion to binary that would not be used in mathematical functions. It turns out, however, that BCD numbers can be added too, there's just an additional step that occurs when each column of digits is added.

When two BCD numbers are added, the digits 1010, 1011, 1100, 1101, 1110, and 1111 must be avoided. This is done by adding an additional step anytime the binary addition of two nibbles results in one of these illegal values or if a carry is generated. When this happens, the invalid result is corrected by adding 6 to skip over the illegal values. For example:

```
 BCD       Decimal

 0011        3
+1000       +8
 1011     Invalid
+0110       +6
10001       11
```