This chapter is different from the rest of the chapters in the textbook. It can be skipped (if your instructor does not wish to cover this material) and will not affect the understanding of future chapters. It does not include any computer programs or any further information on how to write programs. No new pseudocode is presented. However, all programs depend on data. It is data, in the form of numbers and text, that is input, processed, and the results created and output. Thus, it is imperative to understand how a computer stores and manipulates data and that is what this chapter is about. Here, you will learn how various types of data are represented in a computer.

After reading this chapter, you will be able to do the following:

- Convert a decimal integer in base 10 to a binary integer in base 2 and do the reverse (convert binary numbers to decimal)
- Convert decimal integers and binary integers to hexadecimal (base 16) integers and do the reverse
- Understand the difference between signed and unsigned integer representation
- Convert decimal integers to sign-and-magnitude binary integers and do the reverse
- Convert decimal integers to binary using one's complement and two's complement formats and do the reverse
- Convert decimal floating point numbers to binary and do the reverse
- Use scientific and exponential notation
- Create normalized floating point binary numbers using the Excess_127 system
- Create single- and double-precision binary floating point numbers
- Represent floating point normalized binary numbers in the hexadecimal system
It Isn’t Magic—It’s Just Computer Code

This chapter is about how data is represented inside a computer’s memory. While different types of data (whole numbers, fractions, text, and more) are represented differently, as you will see, each piece of data is stored as a long (or very long or exceedingly long) series of 0s and 1s. It’s hard to imagine any real-life situation where you might encounter a series of thousands of 0s and 1s in long lines. But have you ever wondered what the self-proclaimed geeks, nerds, and IT people of the twenty-first century movies and television cop shows are doing when they scroll through screens full of 0s and 1s? After working through the material in this chapter, you will still not be able to read a screenful of machine language code, but you will know that it is just data and you will know that, given the right software to translate all those binary digits to some semblance of English, you too could decipher the material on those screens. It isn’t magic; it’s just computer code, written by ordinary people who learned to be programmers, as you are doing right now.

2.1 Decimal and Binary Representation

The number system we all use in everyday life is the decimal number system. It most likely developed from the fact that we have ten fingers so it is natural for us to count by tens. As you will soon see, 10 is a key number in the decimal system, and we refer to this system as a base 10 system. The number 23, for example, is 2 tens and 3 ones (2\times10 + 3\times1). The number 4,657 is 4 thousands, 6 hundreds, 5 tens, and 7 ones (4\times1000 + 6\times100 + 5\times10 + 7\times1).

Bases and Exponents

The decimal system uses a base of 10. This is the system we normally use for all our mathematical operations but it is just one system out of innumerable possibilities. Before we go further into this definition, we need to understand bases and exponents.

The base is the number we are acting on, and the exponent (the number written, usually, as a superscript) tells the reader what to do with the base. In the expression 3^3, we call 3 the base and 5 the exponent. Example 2.1 demonstrates bases and exponents.

Example 2.1 Bases and Exponents

- Any number squared means that number times itself. In the following example, 10 is the base and 2 is the exponent:
  \[10^2 = 10 \times 10 = 100\]

- A number that is cubed means that the number is multiplied by itself 3 times. Here, 4 is the base and 3 is the exponent:
  \[4^3 = 4 \times 4 \times 4 = 64\]
2.1 Decimal and Binary Representation

- When a number is raised to a positive integer power, it is multiplied by itself that number of times. In the following, the base is 5 and the exponent is 6:
  \[5^6 = 5 \times 5 \times 5 \times 5 \times 5 \times 5 = 15,625\]
- In the following, the base is 8 and the exponent is 1:
  \[8^1 = 8\]

Note: When a nonzero number is raised to the power of 0, the result is always 1. This is true no matter what the number is. As you can see from the following examples, bases change but the exponents are all zero.

- \(5,345^0 = 1\)
- \(4^0 = 1\)
- \((-31)^0 = 1\)
- \(0^0\), however, is undefined

Note: Any number raised to a power can be written as \(X^a\), where \(X\) is the base and \(a\) is the exponent. However, in computer programming, a base with an exponent is represented as \(X^a\) where \(X\) is the base, the carat (^) indicates raising a number to an exponent, and \(a\) is the exponent.

All of this relates directly to our decimal number system. Later, we will see how the same concepts apply to the binary and hexadecimal systems. As we said earlier, in a number like 4,657, we say there are 4 thousands, 6 hundreds, 5 tens, and 7 ones. In other words, we can express the number in terms of the 1's column, the 10's column, the 100's column, and the 1,000's column. There is a reason for these specific columns. Each column represents the base of the system (10) raised to a power. The one's column represents 10 raised to the 0th power, the 10's column represents 10 raised to the 1st power. The 100's column represents 10 raised to the 2nd power, the 1,000's column represents 10 raised to the 3rd power, and so forth. Table 2.1 shows the first eight columns of the decimal system.

<table>
<thead>
<tr>
<th>Table 2.1 The decimal system</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first eight columns of the decimal system</td>
</tr>
<tr>
<td>10^7</td>
</tr>
<tr>
<td>10,000,000</td>
</tr>
<tr>
<td>ten millions</td>
</tr>
<tr>
<td>thousands thousands</td>
</tr>
</tbody>
</table>

Expanded Notation

The ten digits that are used in the decimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. This is because the system uses the base of 10. Any number in the decimal system can be written as a sum of each digit multiplied by the value of its column. This is called expanded notation. Example 2.2 shows how expanded notation works.
Example 2.2 Using Expanded Notation

The number 23 in the decimal system actually means the following:

\[3 \times 10^1 = 3 \times 1 = 3\]
\[+ 2 \times 10^1 = 2 \times 10 = 20\]
\[\Rightarrow 23\]

Therefore, 23 can be expressed as \(2 \times 10^1 + 3 \times 10^0\).

The number 6,825 in the decimal system actually means the following:

\[5 \times 10^3 = 5 \times 1 = 5\]
\[+ 2 \times 10^2 = 2 \times 10 = 20\]
\[+ 8 \times 10^0 = 8 \times 100 = 800\]
\[+ 6 \times 10^3 = 6 \times 1,000 = 6,000\]
\[\Rightarrow 6,825\]

Therefore, 6,825 can be expressed as \(6 \times 10^3 + 8 \times 10^2 + 2 \times 10^1 + 5 \times 10^0\).

In the decimal system, to express a number in expanded notation, you multiply each digit by the power of 10 in its place value. The decimal number 78,902 in expanded notation is written as follows:

\[(7 \times 10^4) + (8 \times 10^3) + (9 \times 10^2) + (0 \times 10^1) + (2 \times 10^0)\]

The Binary System

The binary system follows the same rules as the decimal system. The difference is that while the decimal system uses a base of 10 and has ten digits (0 through 9), the binary system uses a base of 2 and has two digits (0 and 1). The rightmost column of the binary system is the one's column (2^0). It can contain a 0 or a 1. The next number after one is two, but in binary, a two is represented by a 1 in the two's column (2^1), just as 10 in decimal is represented by a 1 in the 10's column (10^1). One hundred in decimal is represented by a 1 in the 100's (10^2) column and in binary, a 1 in the 2^3 column represents the number 4.

Through examples, we will show how to convert numbers from decimal to binary and back again. Before we do, we need a convention to tell us if the number in question is a decimal number or a binary number. For example, the number 10 is ten in decimal but two in binary. We will use subscripts to indicate whether a number is a decimal or a binary number. Thus, 10_{10} indicates the number ten, base 10 but 10_{2} indicates the number two, base 2.

The easiest way to convert a decimal number to binary is to create a little chart with the values of the columns as shown in Table 2.2 and, as you work through a problem, you can fill in the bottom row.

Look at the decimal number you are given. Then find the largest decimal number in the chart that is less than or equal to the given number. Next, in the row for the binary number you are creating, put a 1 in the column corresponding to the number you found in the chart. For example, suppose the given number is 11. The largest number in the chart that is less than or equal to 11_{10} is 8_{10} which is 2^3. So, put a 1 in the 2^3 column. Now, subtract the decimal number you've found in the chart from the given number. In this example, \(11 - 8 = 3\). Then repeat the process, finding the
largest number in the chart that is less than or equal to 3. This is 2, so we put a 1 in the \(2^1\) column of the binary number we are creating. However, we must also put a 0 in the \(2^0\) column. Now, subtract 2 from 3, getting 1. Put a 1 in the \(2^0\) column. The result, in binary, is 1011₂.

**Table 2.2** Chart to use to convert a decimal number to binary

<table>
<thead>
<tr>
<th>Power of 2</th>
<th>2^7</th>
<th>2^6</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

| Binary representation |

**Example 2.3** Convert the Decimal Number 7₁₀ to Binary

- 7 is less than 8 but greater than 4 so put a 1 in the four’s \(2^2\) column.
- \(7 - 4 = 3\)
- 3 is less than 4 but greater than 2 so put a 1 in the two’s \(2^1\) column.
- \(3 - 2 = 1\)
- You have 1 left so put a 1 in the one’s \(2^0\) column.

<table>
<thead>
<tr>
<th>Power of 2</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

| Binary representation | 1  | 1  | 1  | 1  | 1  | 1   |

- Therefore, \(7_{10} = 111₂\).

**Example 2.4** Convert the Decimal Number 29₁₀ to Binary

- 29 is less than 32 but greater than 16 so put a 1 in the 16’s \(2^4\) column
- \(29 - 16 = 13\)
- 13 is less than 16 but greater than 8 so put a 1 in the eight’s \(2^3\) column
- \(13 - 8 = 5\)
- 5 is less than 8 but greater than 4 so put a 1 in the four’s \(2^2\) column
- \(5 - 4 = 1\)
- 1 is less than 2 so there is nothing in the two’s \(2^1\) column
- Put a 0 in the two’s column
- You have 1 left so put a 1 in the one’s \(2^0\) column

<table>
<thead>
<tr>
<th>Power of 2</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

| Binary representation | 0  | 1  | 1  | 1  | 0  | 1   |

- Therefore, \(29_{10} = 11101₂\).
Example 2.5 Convert the Decimal Number $172_{10}$ to Binary

- There is one 128 in 172 so put a 1 in the 128’s ($2^7$) column.
- $172 - 128 = 44$
- 44 is less than 64 so put a 0 in the 64’s ($2^6$) column.
- 44 is less than 64 but greater than 32 so put a 1 in the 32’s ($2^5$) column.
- $44 - 32 = 12$
- 12 is less than 16 but greater than 8 so put a 0 in the 16’s ($2^4$) column and a 1 in the eight’s ($2^3$) column.
- $12 - 8 = 4$
- Put a 1 in the four’s ($2^2$) column.
- $4 - 4 = 0$
- Put 0s in the last two columns.

<table>
<thead>
<tr>
<th>Power of 2</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Binary representation</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Therefore, $172_{10} = 10101100_2$.

Note: To convert numbers larger than 255, we need to add additional columns ($2^8 = 256, 2^9 = 512$, and so on) to the chart.

Converting Binary to Decimal

To convert a binary number back to a decimal number, just add the value of each column in which a 1 is displayed. The binary number $101_{2}$ has nothing in the one’s column and a 1 in the two’s column. The value of the number is $0 + 2 = 2_{10}$.

Examples 2.6 and 2.7 show how it’s done.

Example 2.6 Convert the Binary Number $1011_2$ to Decimal

- There is a 1 in the one’s column.
- There is a 1 in the two’s column so the value of that column is 2.
- There is a 0 in the four’s column so the value of that is 0.
- There is a 1 in the eight’s column so the value of that column is 8.
- $1 + 2 + 0 + 8 = 11$
- Therefore, $1011_2 = 11_{10}$.

Example 2.7 Convert the Binary Number $10101010_2$ to Decimal

- There is a 0 in the one’s column.
- There is a 1 in the two’s column so the value of that column is 2.
- There is a 0 in the four’s column so the value of that is 0.
- There is a 1 in the eight’s column so the value of that column is 8.
- There is a 0 in the 16’s column.
There is a 1 in the 32's column so the value of that column is 32.
There is a 0 in the 64's column.
There is a 1 in the 128's column so the value of that column is 128.
0 + 2 + 0 + 8 + 0 + 32 + 0 + 128 = 170
Therefore, 10101010₂ = 170₁₀.

Self Check for Section 2.1

2.1 Given the decimal number 5⁷, identify the base and the exponent.
2.2 What is the value of 48₂⁰?
2.3 Convert the following decimal numbers to binary:
   a. 6₁₀
   b. 38₁₀
   c. 189₁₀
2.4 Convert the following binary numbers to decimal:
   a. 001₀₂
   b. 10101₀₂
   c. 111111₁₂

2.2 The Hexadecimal System

Before a computer can execute any of the instructions it receives, it must translate those instructions into a language it understands. Computers only understand the binary system. But it would be almost impossible for a human being to read or write in binary code. It’s hard enough to keep track of a series of eight 0s and 1s. Imagine trying to write hundreds of lines of code if it was all 0s and 1s. One way to make reading and writing code easier is to use a shorthand for binary code. This is why we use the hexadecimal system.

The hexadecimal system uses a base of 16. This means there is a one’s column (16⁰), a 16s column (16¹), a 256s column (16²), a 4,096s column (16³), a 65,536s column (16⁴), and so forth. We rarely need to deal with anything larger than the 16⁴ column. Table 2.3 shows the first few columns of the hexadecimal system.

Table 2.3 The first five columns of the hexadecimal system

<table>
<thead>
<tr>
<th>16⁴</th>
<th>16³</th>
<th>16²</th>
<th>16¹</th>
<th>16⁰</th>
</tr>
</thead>
<tbody>
<tr>
<td>16×16×16×16</td>
<td>16×16×16</td>
<td>16×16</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>65,536</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Hexadecimal Digits

The decimal system uses 10 digits (0 through 9) in each column because the base is 10. The binary system uses two digits (0 and 1) in each column because the base is 2. The hexadecimal system uses 16 digits in each column because the base is 16. However, it
would make no sense to write 10 in the one’s column to represent the number ten because there would be no way to distinguish “ten” (written as 10) from “sixteen” (also written as 10 → a one in the 16’s column and a zero in the one’s column). So we need another way to write the number 10 in the hexadecimal system. We use uppercase letters to represent the digits 10 through 15. Therefore, the hexadecimal digits are 0 through 9 and A through F.

In hexadecimal:
- 10_{10} is represented as A_{16}  
- 11_{10} is represented as B_{16}  
- 12_{10} is represented as C_{16}  
- 13_{10} is represented as D_{16}  
- 14_{10} is represented as E_{16}  
- 15_{10} is represented as F_{16}  

Note: To avoid confusion, numbers in systems other than decimal are read digit by digit. For example, the number 283 in decimal is read “two hundred eighty-three.” But the binary number 1011 is read “one-zero-one-one” and the hexadecimal number 28A is read “two-eight-A.”

Converting Decimal to Hexadecimal

Decimal numbers can be converted to hexadecimal in the same manner as we converted decimal to binary in the previous section. You can use Table 2.4 to see the hexadecimal values of some binary and decimal numbers. Example 2.8 provides examples.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>10</td>
<td>32</td>
<td>20</td>
<td>...</td>
<td></td>
<td>160</td>
<td>A0</td>
<td>256</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>17</td>
<td>11</td>
<td>33</td>
<td>21</td>
<td>...</td>
<td></td>
<td>161</td>
<td>A1</td>
<td>257</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>18</td>
<td>12</td>
<td>34</td>
<td>22</td>
<td>...</td>
<td></td>
<td>162</td>
<td>A2</td>
<td>258</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>19</td>
<td>13</td>
<td>35</td>
<td>23</td>
<td>...</td>
<td></td>
<td>163</td>
<td>A3</td>
<td>259</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>20</td>
<td>14</td>
<td>36</td>
<td>24</td>
<td>...</td>
<td></td>
<td>164</td>
<td>A4</td>
<td>260</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>21</td>
<td>15</td>
<td>37</td>
<td>25</td>
<td>...</td>
<td></td>
<td>165</td>
<td>A5</td>
<td>261</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>22</td>
<td>16</td>
<td>38</td>
<td>26</td>
<td>...</td>
<td></td>
<td>166</td>
<td>A6</td>
<td>262</td>
<td>106</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>23</td>
<td>17</td>
<td>39</td>
<td>27</td>
<td>...</td>
<td></td>
<td>167</td>
<td>A7</td>
<td>263</td>
<td>107</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>24</td>
<td>18</td>
<td>40</td>
<td>28</td>
<td>...</td>
<td></td>
<td>168</td>
<td>A8</td>
<td>264</td>
<td>108</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>25</td>
<td>19</td>
<td>41</td>
<td>29</td>
<td>...</td>
<td></td>
<td>169</td>
<td>A9</td>
<td>265</td>
<td>109</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>26</td>
<td>1A</td>
<td>42</td>
<td>2A</td>
<td>...</td>
<td></td>
<td>170</td>
<td>AA</td>
<td>266</td>
<td>10A</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>27</td>
<td>1B</td>
<td>43</td>
<td>2B</td>
<td>...</td>
<td></td>
<td>171</td>
<td>AB</td>
<td>267</td>
<td>10B</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>28</td>
<td>1C</td>
<td>44</td>
<td>2C</td>
<td>...</td>
<td></td>
<td>172</td>
<td>AC</td>
<td>268</td>
<td>10C</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>29</td>
<td>1D</td>
<td>45</td>
<td>2D</td>
<td>...</td>
<td></td>
<td>173</td>
<td>AD</td>
<td>269</td>
<td>10D</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>30</td>
<td>1E</td>
<td>46</td>
<td>2E</td>
<td>...</td>
<td></td>
<td>174</td>
<td>AE</td>
<td>270</td>
<td>10E</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>31</td>
<td>1F</td>
<td>47</td>
<td>2F</td>
<td>...</td>
<td></td>
<td>175</td>
<td>AF</td>
<td>271</td>
<td>10F</td>
</tr>
</tbody>
</table>
Example 2.8(a) Convert a Decimal Number to Hexadecimal

Convert $9_{10}$ to hexadecimal.
- $9_{10} = 9_{16}$ because the one's column in hexadecimal takes all digits up to 15

Example 2.8(b) Convert Another Decimal Number to Hexadecimal

Convert $23_{10}$ to hexadecimal.
- Refer to Table 2.3 to see the columns in the hexadecimal system
- There is one 16 in $23_{10}$ so put a 1 in the 16's column
- $23 - 16 = 7$ so put a 7 in the one's column
- Therefore, $23_{10} = 17_{16}$

Before we convert a larger decimal number to hexadecimal notation, we include a brief math refresher.

A Little Math Lesson

We have seen how to convert large decimal numbers to binary and this was relatively easy because for each column in the binary system, the decimal number contains either one or none of the value of that column. But it is a little harder to convert a large decimal number to hexadecimal. The following example demonstrates how to do it.

Convert $89,468_{10}$ to hexadecimal.

The necessary columns in the hexadecimal system for this number are as follows:

<table>
<thead>
<tr>
<th>$16^4$</th>
<th>$16^3$</th>
<th>$16^2$</th>
<th>$16^1$</th>
<th>$16^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>65,536</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

- Look at the number: $89,468_{10}$.
- There is one 65,536 in 89468 so we put a 1 in the $16^4$ column. (Use the table below, which is already filled in but you will fill in the bottom row yourself as you work through subsequent problems.
- Subtract: $89468 - 65536 = 23932$.
- Now, we must find out how many 4096s are there in 23932.
- Divide: $23932 \div 4096 = 5.8427\ldots$ This means there are five 4096s in 23932 with some left over. Put a 5 in the $16^3$ column.
- What is left over? In other words, what does the fractional part of $5.8427\ldots$ represent? To do this:
  - Multiply 4096 by 5: $4096 \times 5 = 20480$. This is what was used by putting 5 in the $16^3$ column.
  - Subtract 20480 from 23932: $23932 - 20480 = 3452$. So 3452 is the amount left (the remainder after using five 4096s).
• Now, go through a similar process to fill in the rest of the columns:
  • The $16^2$ column: $3452 \div 256 = 13.48437 \ldots$ There are thirteen 256s so put a 3 in the $16^2$ column.
  • $13 \times 256 = 3328$
  • $3452 - 3328 = 124$ (This is the remainder.)
  • To fill in the $16^1$ column, find how many 16s are there in 124:
    $124 \div 16 = 7.75$ so put a 7 in the $16^1$ column.
  • What is left over goes in the $16^0$ column:
    • $7 \times 16 = 112$
    • $124 - 112 = 12$ (the remainder)
  • There are twelve 1s so put a C in the $16^0$ column to represent 12.

<table>
<thead>
<tr>
<th></th>
<th>16⁴</th>
<th>16³</th>
<th>16²</th>
<th>16¹</th>
<th>16⁰</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,048,576</td>
<td>65,536</td>
<td>4,096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>D</td>
<td>7</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

Now we know that $89,468_{10} = 15D7C_{16}$.

### Example 2.8(c) One More Conversion

Convert $875_{10}$ to hexadecimal.

• Refer to Table 2.3 to see the columns in the hexadecimal system.
• $875$ is less than $4,096$ but greater than $256$ so there is nothing in the $4,096$'s ($16^3$) column.
• Divide $875$ by $256$ to see how many $256$s are there.
• $875 \div 256 = 3$ with a remainder of $107$. (Refer to the previous Making It Work section to see how this remainder is obtained.)
• Put a 3 in the $256$'s column.
• $107 \div 16 = 6$ with a remainder of $11$.
• Put a 6 in the 16's column.
• 11 in decimal notation = B in hexadecimal notation.
• Put a B in the one's column.
• Therefore, $875_{10} = 36B_{16}$.

You can use expanded notation to check your work!

Check your answer by writing $36B_{16}$ in expanded notation:

$$(3 \times 256) + (6 \times 16) + (11 \times 1) = 768 + 96 + 11 = 875$$
Converting Hexadecimal to Decimal
To convert a hexadecimal number to decimal, find the decimal value of each digit in the hexadecimal number and add the values.

Example 2.9(a) Convert a Hexadecimal Number to Decimal

Convert A2₁₆ to decimal.
- In expanded notation, this hexadecimal number is A×₁₆ + 2×₁.
- The digit A in hexadecimal means 10 in decimal. The A in this number is in the 16's column.
- A×₁₆ = 10×₁₆ = 160.
- 2×₁ = 2.
- Add these decimal values: 160 + 2 = 162.
- Therefore, A2₁₆ = 16₂ₐ₁₀.

Example 2.9(b) Convert Another Hexadecimal Number to Decimal

Convert 123D₁₆ to decimal.
- In expanded notation, this hexadecimal number is: (1×₄₀₉₆) + (2×₂₅₆) + (3×₁₆) + (D×₁)
- D in hexadecimal is 13 in decimal, so 4,096 + 512 + 48 + 13 = 4,669
- Therefore, 123D₁₆ = 4,669₁₀.

Using Hexadecimal Notation
A computer has no problem dealing with binary notation. It handles enormous strings of data, all represented as 0s and 1s (binary notation) with ease. But, unless you are a truly unusual human being, it would be difficult for you to do the following computation:

01011100111101101 + 111111101011001 + 1000000101011001 + 00011111010101010

Imagine working with hundreds of such long strings of binary numbers! But computers store data in groups of binary numbers. Each binary digit is called a bit. Four bits can represent the decimal numbers 0 through 15. This makes hexadecimal notation a natural choice for a shorthand representation of long binary numbers. Each hexadecimal digit can represent the numbers 0 through 15 in binary. For example, 3₁₀ = 3₁₆ = 0011₂ and 1₄₁₀ = E₁₆ = 1110₂. Table 2.5 shows the conversions for numbers 0 through 15 in decimal, hexadecimal, and binary.

Frequently, binary data is written in hexadecimal. For example, when creating a graphic for a website, colors are represented by six hexadecimal digits. Each hexadecimal digit represents an amount of a color. So white is represented by FFFFFF and black is 000000. Red is FF0000, blue is 0000FF, and green is 00FF00 in the red–green–blue (RGB) system.
Table 2.5  Decimal, hexadecimal, and binary equivalents

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

Converting Binary to Hexadecimal

It is common to write a long binary number in hexadecimal notation. Example 2.10 shows how this is done.

Example 2.10(a)  Convert a Binary Number to Hexadecimal

Convert the following binary number to hexadecimal notation: $10010011_2$.
- First, separate the binary number into sets of 4 digits: $1001$ $0011$.
- If necessary, refer to Table 2.5 to make the conversions:
  $1001_2 = 9_{16}$ and $0011_2 = 3_{16}$
- Therefore, $10010011_2$ is $93_{16}$.

Example 2.10(b)  Convert Another Binary Number to Hexadecimal

Convert the following binary number to hexadecimal notation: $1000111101111_2$.
- First, separate the binary number into sets of 4 digits:
  $1000$ $1111$ $0111$
- Refer to Table 2.5, if necessary, to make the conversions:
  $1000_2 = 8_{16}$ and $1111_2 = F_{16}$ and $0111_2 = 7_{16}$
- Therefore, $1000111101111_2$ is $8F7_{16}$.

Example 2.10(c)  One More Conversion

Convert the following binary number to hexadecimal notation: $11101010000011111_2$.
- First, separate the binary number into sets of 4 digits:
  $1110$ $1010$ $0000$ $1111$
- Refer to Table 2.5, if necessary, to make the conversions:
  $1110_2 = E_{16}$ and $1010_2 = A_{16}$ and $0000_2 = 0_{16}$ and $1111_2 = F_{16}$
- Therefore, $11101010000011111_2$ is $EA0F_{16}$.
Any Base Will Do, If You’re a Mathematician Looking to Have Fun

We suggested that our “normal” number system uses 10 as the base because human beings have ten fingers and ten toes so it might have seemed natural to count to ten and then start over. This may or may not be true but it does seem logical. We do know, however, that the computer uses the binary system because of the structure of a computer. All processes in a computer are a product of combining switches that are either on or off or, as we have seen in Chapter 1, represented as either true or false or 0 or 1. This makes the binary system a necessity for computer programming. The hexadecimal system was chosen for use with many programming languages because, as we have just seen, a single hexadecimal digit represents four bits.

But a number system can be constructed with any base. If an alien species arrives on planet Earth tomorrow with five legs and five arms, they might use a base 5 number system. It is easy, now that you know how to convert decimal to base 2 and base 16, to convert our number system to a system with any base. Let’s take base 5 as an example.

In base 5, the only available digits would be 0, 1, 2, 3, and 4. The first five columns in base 5 would look as follows:

<table>
<thead>
<tr>
<th>The first five columns of a base 5 system</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5^4$</td>
</tr>
<tr>
<td>$5^45^35^25^15^0$</td>
</tr>
<tr>
<td>625</td>
</tr>
</tbody>
</table>

Some conversions from decimal to base 5 would look like this:

$\begin{align*}
3_{10} &= 3_5 \\
5_{10} &= 10_5 \\
27_{10} &= 102_5 \\
2131_{10} &= 32011_5
\end{align*}$

Can you see why?

- There are three 1s in $3_{10}$ so $3_{10} = 3_5$
- There is one 5 and no 1s in $5_{10}$ so $5_{10} = 10_5$
- There is one 25, no 5s, and two 1s in $27_{10}$ so $27_{10} = 102_5$
- There are three 625s, two 125s, no 25s, one 5 and one 1 in $2131_{10}$ so $2131_{10} = 32011_5$

Self Check for Section 2.2

2.5 Explain why a system with base 16 needs to use letters to represent some digits.

2.6 What are the hexadecimal values of the following numbers:

a. $10_2$

b. $10_{10}$
2.7 Convert the following decimal numbers to hexadecimal notation:
   a. $64_{10}$
   b. $159_{10}$
   c. $76,458_{10}$

2.8 Convert the following binary numbers to hexadecimal notation:
   a. $1110_2$
   b. $111111110100010110_2$
   c. $0100010100001101000_2$

2.9 Explain why hexadecimal notation is often used to represent binary numbers.

2.3 Integer Representation

The manner in which computers process numbers depends on each number’s type. Integers are stored and processed in quite a different manner from floating point numbers. But even within the broad categories of integers and floating point numbers, there are more distinctions. Integers can be stored as unsigned numbers (all nonnegative) or as signed numbers (positive, negative, and zero). Floating point numbers also have several variations. This section will give a general overview of the various ways integers can be stored and processed by a computer. Floating point numbers are discussed later in this chapter.

Unsigned Integer Format

In the previous section, we saw how to convert a decimal number to binary form. The number $2_{10}$ in decimal is $10_2$ in binary and the number $101101_2$ is $45_{10}$. Notice that $10_2$ uses two binary digits and $101101_2$ uses six binary digits. However, a computer stores information in memory locations that are normally sixteen to sixty-four bits in length. To store the numbers $10_2$ and $101101_2$ in a computer, both of these numbers must have the same length as a storage location. We do this by adding 0s to the left of the number to fill up as many places as needed for a memory location. This is called the unsigned form of an integer. When a decimal number is converted to an unsigned binary format, the integer value of the number in binary must be calculated and it must also match the number of bits it takes up in the computer’s memory. Therefore, after you change the number to binary, if the result has fewer bits than the number allocated by that computer for integer representation, you need to add 0s to the left of the number. Following are a few examples:

- Store the decimal integer $6_{10}$ in a 4-bit memory location:
  - Convert $6_{10}$ to binary: $110_2$.
  - Add a 0 to the left to make 4 bits: $0110_2$.
- Store the decimal integer $5_{10}$ in an 8-bit memory location:
  - Convert $5_{10}$ to binary: $101_2$.
  - Add five 0s to the left to make 8 bits: $00000101_2$.
- Store the decimal integer $928_{10}$ in a 16-bit memory location:
  - Convert $928_{10}$ to binary: $1101000000_2$.
  - Add six 0s to the left to make 16 bits: $000001101000000_2$. 
Overflow

If you try to store an unsigned integer that is bigger than the maximum unsigned value that can be handled by that computer, you get a condition called overflow. This kind of error occurs in programming frequently and you need to be aware of it. If overflow occurs in your programming, you will have to find another way to write that part of the code. Following are a few examples of overflow:

- Store the decimal integer $23_{10}$ in a 4-bit memory location:
  - The range of integers available in a 4-bit location is $0_{10}$ through $15_{10}$. Therefore, attempting to store $23_{10}$ in a 4-bit location will give you an overflow.

- Store the decimal integer $65,537_{10}$ in a 16-bit memory location:
  - The range of integers available in a 16-bit location is $0_{10}$ through $65,535_{10}$. Therefore, attempting to store this number in a 16-bit location will give you an overflow.

Unsigned integers are the easiest to convert from decimal and also take up the least amount of room in the computer's memory. However, they do not allow for much flexibility because they are limited in how many numbers can be represented. The range of a given method, with a specific number of bits allowed, is the span of numbers—from the smallest to the largest—that can be represented. Table 2.6 provides sample ranges of unsigned integers.

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$0 \ldots 255$</td>
</tr>
<tr>
<td>16</td>
<td>$0 \ldots 65,535$</td>
</tr>
<tr>
<td>32</td>
<td>$0 \ldots 4,294,967,295$</td>
</tr>
<tr>
<td>64</td>
<td>$0 \ldots 18,446,740,000,000,000,000$ approximately</td>
</tr>
</tbody>
</table>

Sign-and-Magnitude Format

The simple unsigned integer method of converting a decimal integer to binary works well to represent positive integers and zero. However, we need a way to represent negative integers. The sign-and-magnitude format provides one way. In sign-and-magnitude format, the leftmost bit is reserved to represent the sign. The other bits represent the magnitude (or the absolute value) of the integer. Table 2.7 shows sample ranges of sign-and-magnitude integers.

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$-127 \ldots +127$</td>
</tr>
<tr>
<td>16</td>
<td>$-32,767 \ldots +32,767$</td>
</tr>
<tr>
<td>32</td>
<td>$-2,147,483,647 \ldots +2,147,483,647$</td>
</tr>
</tbody>
</table>
The absolute value of a number is its value, ignoring the sign. The magnitude of a number is its absolute value. For example, the absolute value of \(-3\) is 3, the absolute value of 5 is 5, and the absolute value of 0 is 0.

**Representation of Sign-and-Magnitude Integers**

In this format, if the leftmost bit is 0, the number is positive and if the leftmost bit is 1, the number is negative. The rest of the bits give the magnitude of the number, as in unsigned integer format. So the sign-and-magnitude number 0111₂ represents \(+7_{10}\) and the number 1111₂ represents \(-7_{10}\). To convert a decimal integer to sign-and-magnitude number, you need to know both the sign of the number and the number of bits allocated for storing integers. In an \(N\)-bit allocation, \(N-1\) bits are used to store the magnitude of the number and the \(N\)th bit is used to represent the sign. Examples 2.11 and 2.12 show how this works.

**Example 2.11(a) Convert a Decimal Number to Sign-and-Magnitude Binary Format**

Store the decimal integer \(+23_{10}\) in an 8-bit memory location using sign-and-magnitude format:

- Convert \(23_{10}\) to binary: 10111₂.
- Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
- The number 10111₂ uses 5 bits so add two 0s to the left to make up 7 bits: 0010111₂.
- Finally, look at the sign. This number is positive so add a 0 in the leftmost place to show the positive sign.
- Therefore, \(+23_{10}\) in sign-and-magnitude format in an 8-bit location is 00010111₂.

**Example 2.11(b) Convert Another Decimal Number to Sign-and-Magnitude Binary Format**

Store the decimal integer \(-19_{10}\) in an 8-bit memory location using sign-and-magnitude format:

- Convert \(-19_{10}\) to binary: 10011₂.
- Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
- The number 10011₂ uses 5 bits so add two 0’s to the left to make up 7 bits: 0010011₂.
- Finally, look at the sign. This number is negative so add a 1 in the leftmost place to show the negative sign.
- Therefore, \(-19_{10}\) in sign-and-magnitude format in an 8-bit location is 10010011₂.
Example 2.12(a) Convert a Sign-and-Magnitude Binary Number to Decimal

a. Given that 00110111₂ is an 8-bit binary integer in sign-and-magnitude format, what is its decimal equivalent?
   - First convert the rightmost 7 bits to decimal to get 55₁₀.
   - Look at the leftmost bit; it is a 0. That means this number is positive.
   - Therefore, 00110111₂ represents the decimal integer +55₁₀.

Example 2.12(b) Convert Another Sign-and-Magnitude Binary Number to Decimal

Given that 10001110₂ is an 8-bit binary integer in sign-and-magnitude format, what is its decimal equivalent?
   - First convert the rightmost 7 bits to decimal to get 14₁₀.
   - Look at the leftmost bit; it is a 1. That means this number is negative.
   - Therefore, 10001110₂ represents the decimal integer -14₁₀.

The Zero

One serious problem faced by programmers is how to represent zero in binary. As you will see, by the following examples, the sign-and-magnitude form has two ways to represent zero.

Example 2.13 Two Ways to Represent Zero in Sign-and-Magnitude Format

(a) Store the decimal integer 0₁₀ in an 8-bit memory location using sign-and-magnitude format:
   - Convert 0₁₀ to binary: 0₂.
   - Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
   - The number 0₂ uses 1 bit so add six 0s to the left to make up 7 bits: 0000000₂.
   - Finally, look at the sign. Zero is considered a non-negative number so you should add a 0 in the leftmost place to show that it is not negative.
   - Therefore, 0₁₀ in sign-and-magnitude in an 8-bit location is 0000000₂.

(b) ... but ... given that 1000000₂ is an 8-bit binary integer in sign-and-magnitude form, find its decimal value:
   - First convert the rightmost 7 bits to decimal to get 0₁₀.
   - Look at the leftmost bit; it is a 1. That means this number is negative.
   - Therefore, 1000000₂ represents the decimal integer -0₁₀.

We see, then, that using sign-and-magnitude format allows both 0000000₂ and 1000000₂ to represent the same number. This can wreak havoc in a computer program.
One's Complement Format

The fact that 0 has two possible representations in sign-and-magnitude format is one of the main reasons why computers usually use a different method to represent signed integers. There are two other formats that may be used to store signed integers. The one's complement method is not often used; but it is explained here because it helps to understand the most common format: two's complement.

In sign-and-magnitude format, the number \(+6_{10}\) in a 4-bit allocation, is written as \(0110_2\). The leftmost bit is reserved for the sign of the number. So the 0 on the left here represents a positive sign. The number \(-6_{10}\) in a 4-bit allocation, is written as \(1110_2\), where the leftmost bit, a 1, represents a negative sign. The one's complement method is slightly different.

To complement a binary digit, you simply change a 1 to a 0 or a 0 to a 1. In the one's complement method, positive integers are represented as they would be in sign-and-magnitude format. The leftmost bit is still reserved as the sign bit. So \(+6_{10}\) in a 4-bit allocation, is still \(0110_2\). But in one's complement, \(-6_{10}\) is just the complement of \(+6_{10}\). So \(-6_{10}\) becomes \(1001_2\). This means that the range of one's complement integers is the same as the range of sign-and-magnitude integers. It also means that there are still two ways to represent the zero. Following are a few examples of how decimal numbers are converted to signed binary integers using the one's complement method.

Example 2.14(a) Convert a Decimal Number With One's Complement Format

Store the decimal integer \(+78_{10}\) in an 8-bit memory location using one's complement format:

- Convert \(78_{10}\) to binary: \(1001110_2\).
- Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
- The number \(1001110_2\) uses all 7 bits.
- Finally, look at the sign. This number is positive so add a zero in the leftmost place to show the positive sign.
- Therefore, \(+78_{10}\) in one's complement in an 8-bit location is \(01001110_2\).

Example 2.14(b) Convert Another Decimal Number With One's Complement Format

Store the decimal integer \(-37_{10}\) in an 8-bit memory location using one's complement format:

- Convert \(37_{10}\) to binary: \(100101_2\).
- Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
- The number \(100101_2\) uses 6 bits so add one 0 to the left to make up 7 bits: \(0100101_2\).
- Now, look at the sign. This number is negative.
- Complement all the digits by changing all the 0s to 1s and all the 1s to 0s.
- Add a 1 in the 8th bit location because the number is negative.
- Therefore, \(-37_{10}\) in one's complement in an 8-bit location is \(11011010_2\).
Example 2.14(c) Another Conversion

Store the decimal integer $+139_{10}$ in an 8-bit memory location using one's complement format:

- Convert $139_{10}$ to binary: $10001011_2$.
- Since this is an 8-bit memory location, only 7 bits may be used for storing the magnitude of the number.
- The magnitude of this number requires 8 bits.
- Therefore, $139_{10}$ cannot be represented in an 8-bit memory location in one's complement format.

To convert a one's complement number back to decimal, simply look at the leftmost bit to determine the sign. If the leftmost bit is 0, the number is positive and can be converted back to decimal immediately. If the leftmost bit is a 1, the number is negative. Uncomplement the other bits (change all the 0s to 1s and all the 1s to 0s) and then convert the binary bits back to decimal. Remember to include the negative sign when displaying the result!

The Zero Again

Unfortunately, the one's complement method does not solve the problem of a dual representation for zero. With one's complement, as shown in the following, there are still two representations for zero.

Example 2.15 Representing Zero With One's Complement Format

(a) Store the decimal integer $0_{10}$ in an 8-bit memory location using one's complement format:

- Convert $0_{10}$ to binary: $0_2$.
- Since this is an 8-bit memory location, 7 bits are used for storing the magnitude of the number.
- The number 0, uses 1 bit so add six 0's to the left to make up 7 bits: $0000000_2$.
- Now, look at the sign. Zero is considered a non-negative number so you probably should add a zero in the leftmost place to show that it is not negative.
- Therefore, $0_{10}$ in one's complement in an 8-bit location is $0000000_2$.

(b) But... given that $11111111_2$ is a binary number in one's complement form, find its decimal value:

- First look at the leftmost bit. It is a 1 so you know the number is negative.
- Since the leftmost bit is 1, you also know that all the other bits have been complemented. You need to "un-complement" them to find the magnitude of the number.
- When you un-complement $11111111_2$, you get $00000000_2$.
- Therefore, $11111111_2$ in one's complement represents the decimal integer $-0_{10}$. 
Why So Much Fuss About Nothing?

You may wonder why there is so much fuss about the zero. Why not just define zero in binary as 0000₂ (or 00000000₂, or 0000000000000000₂, depending on how many bits you are using) and be done with it? However, in, for example, a 4-bit allocation, the bit-pattern 1111₂ still exists. When calculations are performed, this number could be the result. Unless the computer knows what to do with it, the program will get an error. It might even not work at all. Here’s one possible scenario: If the result of a calculation using one’s complement was 1111₂, the computer would read this as -0, as we have seen in Example 2.15, part (b). If you then try to add 1 to it, what would the answer be? The number that follows 1111₂ in a 4-bit allocation is 0000₂. (We will learn later how to add binary integers.) So that would mean, using one’s complement, that -0 + 1 = +0. This certainly would not be an irrelevant issue!

To avoid the complications caused by two representations of zero, programmers usually use a slightly more complicated but more accurate form called two’s complement to represent integers.

Two’s Complement Format

In the one’s complement method, positive integers are represented as they would be in sign-and-magnitude format. The leftmost bit is reserved as the sign bit. A negative integer is just the complement of the positive integer, with a 1 in the leftmost spot to represent the negative sign. We have seen that this creates some problems, especially in the case of zero. The two’s complement solves the problem of the zero.

The following is how to find the two’s complement of an $x$-bit number:

1. If the number is positive, just convert the decimal integer to binary and you are finished.
2. If the number is negative, convert the number to binary and find the one’s complement.
3. Add a binary 1 to the one’s complement of the number.
4. If this results in an extra bit (more than $x$ bits), discard the leftmost bit.

Binary Addition

To do the addition (Step 3), we need to learn to add in binary. Binary arithmetic operations are a bit more complicated than what we show here, but this brief explanation is enough for the purpose of finding the two’s complement of a number.

- In binary, adding 1 to 0 results in 1, adding 0 to 1 also results in 1, and adding 0 to 0 results in 0 (as you might expect).
- In the decimal system, if you add 1 to 9, you put a zero in the one’s column and carry a 1 to the 10’s column. In the binary system, you do virtually the same thing. If you add 1 to 1 in binary, you put a zero in that column and carry a 1 to the next column. This is because 1 + 1 in binary = 2 but there is no digit “2” in binary. Instead, you must carry a 2 to the next column.
- To complete the addition you need for two’s complement, you need to remember two rules of binary addition, as shown in Table 2.8. Examples 2.16–2.18 provide examples.
Table 2.8  The two rules of binary addition

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$1 + 0 = 1$</td>
<td>$1 + 1 = 10$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$+ 1$</td>
<td>$+ 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 + 0$</td>
<td>$+ 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 1 1$</td>
<td>$1 0 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$+ 1$</td>
<td>$+ 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 0 0$</td>
<td>$1 0 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1 0 1$</td>
<td>$1 1 0$</td>
<td></td>
</tr>
</tbody>
</table>

Example 2.16(a)  Finding the Two’s Complement of a 4-bit Binary Integer

Find the two’s complement of $+2_{10}$ as a 4-bit binary integer.

- Convert $2_{10}$ to binary: $10_2$.
- Add zeros to the left to complete 4 bits: 0010.
- Since this is already a positive integer, you are finished.

Example 2.16(b)  Finding the Two’s Complement of Another 4-bit Binary Integer

Find the two’s complement of $-2_{10}$ as a 4-bit binary integer:

- Convert $2_{10}$ to binary: $10_2$.
- Add zeros to the left to complete 4 bits: 0010.
- Since the number is negative, do the one’s complement to get: 1101.
- Now add binary 1 to this number:
  
  \[
  \begin{array}{c}
  1101 \\
  + 1 \\
  \hline
  1110 \\
  \end{array}
  \]
- Therefore, $-2_{10}$ in two’s complement in a 4-bit location is 1110.

Example 2.17(a)  Finding the Two’s Complement of an 8-bit Binary Integer

Find the two’s complement of $+3_{10}$ as an 8-bit binary integer:

- Convert $3_{10}$ to binary: $101011_2$.
- Add zeros to the left to complete 7 bits: 0010111.
- Since this is already a positive integer, you are finished.
Example 2.17(b) Finding the Two’s Complement of Another 8-bit Binary Integer

Find the twos complement of -43\textsubscript{10} as an 8-bit binary integer:

- Convert 43\textsubscript{10} to binary: 101011\textsubscript{2}.
- Add zero’s to the left to complete 8 bits: 00101011.
- Since the number is negative, do the one’s complement to get: 11010100.
- Now add binary 1 to this number:
  \[11010100 + 1 = 11010101\]
- Therefore, -43\textsubscript{10} in two’s complement in an 8-bit location is 11010101.

Example 2.18 Carrying 1s with Binary Addition

Find the two’s complement of -24\textsubscript{10} as an 8-bit binary integer:

- Convert 24\textsubscript{10} to binary: 11000\textsubscript{2}.
- Add zeros to the left to complete 7 bits: 00011000.
- Since the number is negative, do the one’s complement to get: 11100111.
- Now add binary 1 to this number:
  \[11100111 + 1 = 11101000\]
- Therefore, -24\textsubscript{10} in two’s complement in an 8-bit location is 11101000.

Pay close attention to the math in Example 2.18. Starting at the rightmost column, you add a 1 to the 1 in the one’s column: 1 + 1 = 0 with a carry. When you carry the 1 to the two’s column, you also get 1 + 1 = 0 with another carry. You carry this 1 to the four’s column where you also get 1 + 1 = 0 with another carry. But in the eight’s column, you add 1 + 0 so the result in the eight’s column is a 1 with no carry.

But what happens when the two’s complement cannot be done? This is shown in Example 2.19.

Example 2.19 When the Two’s Complement Cannot Be Done

Find the two’s complement of -159\textsubscript{10} as an 8-bit binary integer:

- Convert 159\textsubscript{10} to binary: 10011111\textsubscript{2}.
- 10011111 already takes up 8 bits so there is nothing left for the sign bit.
- Therefore, -159\textsubscript{10} cannot be represented as a two’s complement binary number in an 8-bit location.

We will leave the process of converting from a two’s complemented number back to binary for a more advanced course.
The Two’s Complement Zero

You will recall that, in both sign-and-magnitude and in one’s complement representation, there are two ways to represent binary 0. This problem is solved by using two’s complement. You will see why in Example 2.20, which shows how to represent +0 and -0 in two’s complement.

**Example 2.20 The Solution to the Problem of Zero!**

(a) Find the two’s complement of \( +0_{10} \) as an 8-bit binary integer:
- Convert \( 0_{10} \) to 8-bit binary: \( 00000000_2 \).
- The number is positive, so nothing more needs to be done.
- Therefore, +0 in two’s complement in an 8-bit location is \( 00000000 \).

(b) Find the two’s complement of \( -0_{10} \) as an 8-bit binary integer:
- Convert \( 0_{10} \) to 7-bit binary: \( 0000000_2 \).
- Since the number is negative, do the one’s complement, including a 1 as the eighth bit, to get: \( 11111111_2 \).
- Now add binary 1 to this number:

\[
\begin{array}{c}
11111111 \\
\hline
+1 \\
\hline
10000000
\end{array}
\]

- Recall that Step 4 in the rules for converting to two’s complement states that, after the addition of 1 to the one’s complement, any digits to the left of the maximum number of bits (here, 8 bits) should be discarded.
- Discard the leftmost 1.
- Therefore, \( -0_{10} \) in two’s complement in an 8-bit location is \( 00000000 \) which is exactly the same as \( +0_{10} \).

The problem of representing zero in two ways is solved by the two’s complement method!

**Why the Two’s Complement Works**

We have seen how complementing all the bits turns a positive integer into a negative one using one’s complement. The two’s complement may seem even more confusing. How in the world does flipping digits and then adding 1 somehow end up with the negative of the number you started with? But it does make mathematical sense, especially if you think like a computer. That means you must think in the binary system.

We will use a 4-bit allocation for our example, since this is easiest to manage. A 4-bit allocation allows for 16 binary numbers ranging from 0000 to 1111 or \( 0_{10} \) to \( 15_{10} \). In this situation, we define the “flip side” of any number between 0 and 16 to be 16 minus that number. On the other hand, if we had 8 bits available, there would be 256 possible numbers (\( 0_{10} \) to \( 255_{10} \)), so the flip side of a number between 0 and 256 would be 256 minus that number. For example, the flip side of 4 is \( 16 - 4 = 12 \). In two’s complement, the negative of a number is represented as the flip side of its positive value. Thus, using two’s complement notation, \( -3_{10} \) is represented as the
flip side of \( +3_{10} \). In a 4-bit location, this would be \( 16 - 3 = 13 \). In an 8-bit location, this would be \( 256 - 3 = 253 \) because \( 2^8 = 256 \).

In mathematical terms, this can be expressed as follows: Assume you are working with an X-bit memory allocation. Then, for a number, \( N \), the two’s complement is \( 2^X - |N| \), where \( |N| \) denotes the absolute value of the given number, \( N \).

The following example illustrates how to use this formula.

**Example 2.21 Using a Formula to Find the Two’s Complement**

Use the formula:

\[
2^X - |N|
\]

to find the two’s complement of \( -37_{10} \) stored in an 8-bit memory location.

By the formula shown above, the answer should be as follows:

- \( -37 = (2^8 - |37|) = 256 - 37 = 219 \) (since, in this example, \( N = -37 \))
- \( 219_{10} \) converted to binary, is \( 11011011_2 \).

Check, using the first method described earlier:

- Convert \( 37_{10} \) to 8-bit binary to get \( 00100101_2 \).
- Since this is a negative number, complement each bit to get \( 11011010 \).
- Add a binary 1 to this number to get \( 11011011 \).
- \( 11011011_2 = 219_{10} \)

The two’s complement is the standard representation for storing positive, zero, and negative integers in computers today. We have already discussed (in depth!) how important this method is when it comes to representing the zero.

### Self Check for Section 2.3

2.10 Convert the following decimal numbers to binary using the sign-and-magnitude format, using 8 bits:

- a. \( 48_{10} \)
- b. \( -39_{10} \)
- c. \( -284_{10} \)
- d. \( 0_{10} \)

2.11 Convert the following decimal numbers to binary using the one’s complement format, using 8 bits:

- a. \( 48_{10} \)
- b. \( -39_{10} \)
- c. \( -284_{10} \)
- d. \( 0_{10} \)
2.12 Convert the following decimal numbers to binary using the two’s complement format, using 8 bits:
   a. $48_{10}$
   b. $-39_{10}$
   c. $-284_{10}$
   d. $0_{10}$

2.13 Briefly explain why two’s complement is the best format for representing signed integers in a computer.