Our goal in this chapter is to show you how simple the computer that you’re using really is. We will describe in detail a simple imaginary machine that has many of the characteristics of real processors at the heart of the computational devices that surround us.

You may be surprised to learn that many, many machines share these same properties, even some of the very first computers that were developed. Accordingly, we are able to tell the story in historical context. Imagine a world without computers, and what sort of device might be received with enthusiasm, and that is not far from what we have! We tell our story from the standpoint of scientific computing—there is an equally fascinating story from the standpoint of commercial computing that we touch on just briefly.

Next, our aim is to convince you that the basic concepts and constructs that we covered in Java programming are not so difficult to implement on a simple machine, using its own machine language. We will consider in detail conditionals, loops, functions, arrays, and linked structures. Since these are the same basic tools that we examined for Java, it follows that several of the computational tasks that we addressed in the first part of this book are not difficult to address at a lower level.

This simple machine is a link on the continuum between your computer and the actual hardware circuits that change state to reflect the action of your programs. As such it is preparation for learning how those circuits work, in the next chapter.

And that still is only part of the story. We end the chapter with a profound idea: we can use one machine to simulate the operation of another one. Thus, we can easily study imaginary machines, develop new machines to be built in future, and work with machines that may never be built.
6.1 Representing Information

The first step in understanding how a computer works is to understand how information is represented within the computer. As we know from programming in Java, everything suited for processing with digital computers is represented as a sequence of 0s and 1s, whether it be numeric data, text, executable files, images, audio, or video. For each type of data, standard methods of encoding have come into widespread use: The ASCII standard associates a seven bit binary number with each of 128 distinct characters; the MP3 file format rigidly specifies how to encode each raw audio file as a sequence of 0s and 1s; the .png image format specifies the pixels in digital images ultimately as a sequence of 0s and 1s, and so forth.

Within a computer, information is most often organized in words, which are nothing more than a sequence of bits of a fixed length (known as the word size). The word size plays a critical role in the architecture of any computer, as you will see. In early computers, 12 or 16 bits were typical; for many years 32-bit words were widely used; and nowadays 64-bit words are the norm.

The information content within every computer is nothing more nor less than a sequence of words, each consisting of a fixed number of bits, each either 0 or 1. Since we can interpret every word as a number represented in binary, all information is numbers, and all numbers are information.

The meaning of a given sequence of bits within a computer depends on the context. This is another of our mantras, which we will repeat throughout this chapter. For example, as you will see, depending on the context, we might interpret the binary string 111101011001110 to mean the positive integer 64,206, the negative integer –1,330, the real number – 55744.0, or the two-character string "eN".

Convenient as it may be for computers, the binary number system is extremely inconvenient for humans. If you are not convinced of this fact, try memorizing the 16-bit binary number 111101011001110 to the point that you can close the book and write it down. To accommodate the computer’s need to communicate in binary while at the same time accommodating our need to use a more compact representation, we introduce in this section the hexadecimal (base 16) number system, which turns out to be a convenient shorthand for binary. Accordingly, we begin by examining hexadecimal in detail.
6.1 Representing Information

**Binary and Hex** For the moment, consider nonnegative integers, or *natural numbers*, the fundamental mathematical abstraction for counting things. Since Babylonian times, people have represented integers using *positional notation* with a fixed *base*. The most familiar to you is certainly *decimal*, where the base is 10 and each positive integer is represented as a string of digits between zero and 9. Specifically, \( d_n d_{n-1} \ldots d_2 d_1 d_0 \) represents the integer

\[
d_n 10^n + d_{n-1} 10^{n-1} + \ldots + d_2 10^2 + d_1 10^1 + d_0 10^0
\]

For example, 10345 represents the integer

\[
10345 = 1 \cdot 10000 + 0 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1.
\]

We can replace the base 10 by any integer greater than 1 to get a different number system where we represent any integer by a string of digits, all between 0 and one less than the base. For our purposes, we are particularly interested in two such systems: *binary* (base 2) and *hexadecimal* (base 16).

**Binary.** When the base is two, we represent an integer as a sequence of 0s and 1s. In this case, we refer to each binary (base 2) digit—either 0 or 1—as a *bit*, the basis for representing information in computers. In this case the bits are coefficients of powers of 2. Specifically, the sequence of bits \( b_n b_{n-1} \ldots b_2 b_1 b_0 \) represents the integer

\[
b_n 2^n + b_{n-1} 2^{n-1} + \ldots b_2 2^2 + b_1 2^1 + b_0 2^0
\]

For example, 1100011 represents the integer

\[
99 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.
\]

Note that the largest integer that we can represent in an \( n \)-bit word in this way is \( 2^n - 1 \), when all \( n \) bits are 1. For example, with 8 bits, 11111111 represents

\[
2^8 - 1 = 255 = 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.
\]

Another way of stating this limitation is that we can represent only \( 2^n \) nonnegative integers (0 through \( 2^n - 1 \)) in an \( n \)-bit word. We often have to be aware of such limitations when processing integers with a computer. Again, a big disadvantage of using binary notation is that the number of bits required to represent a number in binary is much larger than, for example, the number of digits required to represent the same number in decimal. Using binary exclusively to communicate with a computer would be unwieldy and impractical.
**Hexadecimal.** In hexadecimal (or just hex from now on) the sequence of hex digits \(h_n h_{n-1} \ldots h_2 h_1 h_0\) represents the integer

\[
h_n 16^n + h_{n-1} 16^{n-1} + \ldots + h_2 16^2 + h_1 16^1 + h_0 16^0
\]

The first complication we face is that, since the base is 16, we need digits for each of the values 0 through 15. We need to have one character to represent each digit, so we use A for 10, B for 11, C for 12, and so forth, as shown in the table at left. For example, FACE represents the integer

\[
64,206 = 15 \cdot 16^3 + 10 \cdot 16^2 + 12 \cdot 16^1 + 14 \cdot 16^0.
\]

This is the same integer that we represented with 16 bits earlier. As you can see from this example, the number of hex digits needed to represent integers in hexadecimal is only a fraction (about one-fourth) of the number of bits needed to represent the same integer in binary. Also, the variety in the digits makes a number easy to remember. You may have struggled with 111101011001110, but you certainly can remember FACE.

**Conversion between hex and binary.** Given the hex representation of a number, finding the binary representation is easy, and vice-versa, as illustrated in the figure at left. Since the hex base 16 is a power of the binary base 2, we can convert groups of four bits to hex digits and vice versa. To convert from hex to binary, replace each hex digit by the four binary bits corresponding to its value (see the table at right). Conversely, given the binary representation of a number, add leading 0s to make the number of bits a multiple of 4, then group the bits four at a time and convert each group to a single hex digit. You can do the math to prove that these conversions are always correct (see Exercise 5.1.8), but just a few examples should serve to convince you.

\[
\begin{array}{ccc}
\text{decimal} & \text{binary} & \text{hex} \\
0 & 0000 & 0 \\
1 & 0001 & 1 \\
2 & 0010 & 2 \\
3 & 0011 & 3 \\
4 & 0100 & 4 \\
5 & 0101 & 5 \\
6 & 0110 & 6 \\
7 & 0111 & 7 \\
8 & 1000 & 8 \\
9 & 1001 & 9 \\
10 & 1010 & A \\
11 & 1011 & B \\
12 & 1100 & C \\
13 & 1101 & D \\
14 & 1110 & E \\
15 & 1111 & F
\end{array}
\]

Representations of integers from 0 to 15

For example the hex representation of the integer 39 is 27, so the binary representation is 00100111 (and we can drop the leading zeros); the binary representation of 228 is 11100100, so the hex representation is E4. This ability to convert quickly...
from binary to hex and from hex to binary is important as an efficient way to communicate with the computer. You will be surprised at how quickly you will learn this skill, once you internalize the basic knowledge that A is equivalent to 1010, 5 is equivalent to 0101, F is equivalent to 1111, and so forth.

**Conversion from decimal to binary.** We have considered the problem of computing the string of 0s and 1s that represent the binary number corresponding to a given integer as an early programming example. The following recursive program (the solution to Exercise 2.3.15) does the job, and is worthy of careful study:

```java
public static String toBinaryString(int N) {
    if (N == 0) return "";
    if (N % 2 == 0)
        return toBinaryString(N/2) + '0';
    else return toBinaryString(N/2) + '1';
}
```

It is a recursive method based on the idea that the last digit is the character that represents \( N \% 2 \) ('0' if \( N \% 2 \) is 0 and '1' if \( N \% 2 \) is 1) and the rest of the string is the string representation of \( N/2 \). A sample trace of this program is shown at right. This method generalizes to handle hexadecimal (and any other base), and we also are interested in converting string representations to Java datatype values. In the next section, we consider a program that accomplishes such conversions.

When we talk about what is going on within the computer, our language is hex. The contents of an \( n \)-bit computer word can be specified with \( n/4 \) hex digits and immediately translated to binary if desired. You likely have observed such usage already in your daily life. For example, when you register a new device on your network, you need to know its media access control (MAC) address. A MAC address such as `1a:ed:b1:b9:96:5e` is just hex shorthand (using some superfluous colons and lowercase a–f instead of the uppercase A–F that we use) for a 48-bit binary number that identifies your device for the network.
<table>
<thead>
<tr>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>0</td>
<td>32</td>
<td>00100000</td>
<td>20</td>
<td>64</td>
<td>01000000</td>
<td>40</td>
<td>96</td>
<td>01100000</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td>01</td>
<td>33</td>
<td>00100001</td>
<td>21</td>
<td>65</td>
<td>01000001</td>
<td>41</td>
<td>97</td>
<td>01100001</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
<td>02</td>
<td>34</td>
<td>00100010</td>
<td>22</td>
<td>66</td>
<td>01000010</td>
<td>42</td>
<td>98</td>
<td>01100010</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>00000011</td>
<td>03</td>
<td>35</td>
<td>00100011</td>
<td>23</td>
<td>67</td>
<td>01000011</td>
<td>43</td>
<td>99</td>
<td>01100011</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td>04</td>
<td>36</td>
<td>00100100</td>
<td>24</td>
<td>68</td>
<td>01000100</td>
<td>44</td>
<td>100</td>
<td>01100100</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
<td>05</td>
<td>37</td>
<td>00100101</td>
<td>25</td>
<td>69</td>
<td>01000101</td>
<td>45</td>
<td>101</td>
<td>01100101</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>00000110</td>
<td>06</td>
<td>38</td>
<td>00100110</td>
<td>26</td>
<td>70</td>
<td>01000110</td>
<td>46</td>
<td>102</td>
<td>01100110</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
<td>07</td>
<td>39</td>
<td>00100111</td>
<td>27</td>
<td>71</td>
<td>01000111</td>
<td>47</td>
<td>103</td>
<td>01100111</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
<td>08</td>
<td>40</td>
<td>00101000</td>
<td>28</td>
<td>72</td>
<td>01001000</td>
<td>48</td>
<td>104</td>
<td>01101000</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>00001001</td>
<td>09</td>
<td>41</td>
<td>00101001</td>
<td>29</td>
<td>73</td>
<td>01001001</td>
<td>49</td>
<td>105</td>
<td>01101001</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>00001010</td>
<td>0A</td>
<td>42</td>
<td>00101010</td>
<td>2A</td>
<td>74</td>
<td>01001010</td>
<td>4A</td>
<td>106</td>
<td>01101010</td>
<td>6A</td>
</tr>
<tr>
<td>11</td>
<td>00001011</td>
<td>0B</td>
<td>43</td>
<td>00101011</td>
<td>2B</td>
<td>75</td>
<td>01001011</td>
<td>4B</td>
<td>107</td>
<td>01101011</td>
<td>6B</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
<td>0C</td>
<td>44</td>
<td>00101100</td>
<td>2C</td>
<td>76</td>
<td>01001100</td>
<td>4C</td>
<td>108</td>
<td>01101100</td>
<td>6C</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>0D</td>
<td>45</td>
<td>00101101</td>
<td>2D</td>
<td>77</td>
<td>01001101</td>
<td>4D</td>
<td>109</td>
<td>01101101</td>
<td>6D</td>
</tr>
<tr>
<td>14</td>
<td>00001110</td>
<td>0E</td>
<td>46</td>
<td>00101110</td>
<td>2E</td>
<td>78</td>
<td>01001110</td>
<td>4E</td>
<td>110</td>
<td>01101110</td>
<td>6E</td>
</tr>
<tr>
<td>15</td>
<td>00001111</td>
<td>0F</td>
<td>47</td>
<td>00101111</td>
<td>2F</td>
<td>79</td>
<td>01001111</td>
<td>4F</td>
<td>111</td>
<td>01101111</td>
<td>6F</td>
</tr>
<tr>
<td>16</td>
<td>00010000</td>
<td>10</td>
<td>48</td>
<td>00110000</td>
<td>30</td>
<td>80</td>
<td>01010000</td>
<td>50</td>
<td>112</td>
<td>01110000</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>11</td>
<td>49</td>
<td>00110001</td>
<td>31</td>
<td>81</td>
<td>01010001</td>
<td>51</td>
<td>113</td>
<td>01110001</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>00010010</td>
<td>12</td>
<td>50</td>
<td>00110010</td>
<td>32</td>
<td>82</td>
<td>01010010</td>
<td>52</td>
<td>114</td>
<td>01110010</td>
<td>72</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>13</td>
<td>51</td>
<td>00110011</td>
<td>33</td>
<td>83</td>
<td>01010011</td>
<td>53</td>
<td>115</td>
<td>01110011</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
<td>14</td>
<td>52</td>
<td>00110100</td>
<td>34</td>
<td>84</td>
<td>01010100</td>
<td>54</td>
<td>116</td>
<td>01110100</td>
<td>74</td>
</tr>
<tr>
<td>21</td>
<td>00010101</td>
<td>15</td>
<td>53</td>
<td>00110101</td>
<td>35</td>
<td>85</td>
<td>01010101</td>
<td>55</td>
<td>117</td>
<td>01110101</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>00010110</td>
<td>16</td>
<td>54</td>
<td>00110110</td>
<td>36</td>
<td>86</td>
<td>01010110</td>
<td>56</td>
<td>118</td>
<td>01110110</td>
<td>76</td>
</tr>
<tr>
<td>23</td>
<td>00010111</td>
<td>17</td>
<td>55</td>
<td>00110111</td>
<td>37</td>
<td>87</td>
<td>01010111</td>
<td>57</td>
<td>119</td>
<td>01110111</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
<td>18</td>
<td>56</td>
<td>00111000</td>
<td>38</td>
<td>88</td>
<td>01011000</td>
<td>58</td>
<td>120</td>
<td>01111000</td>
<td>78</td>
</tr>
<tr>
<td>25</td>
<td>00011001</td>
<td>19</td>
<td>57</td>
<td>00111001</td>
<td>39</td>
<td>89</td>
<td>01011001</td>
<td>59</td>
<td>121</td>
<td>01111001</td>
<td>79</td>
</tr>
<tr>
<td>26</td>
<td>00011010</td>
<td>1A</td>
<td>58</td>
<td>00111010</td>
<td>3A</td>
<td>90</td>
<td>01011010</td>
<td>5A</td>
<td>122</td>
<td>01111010</td>
<td>7A</td>
</tr>
<tr>
<td>27</td>
<td>00011011</td>
<td>1B</td>
<td>59</td>
<td>00111011</td>
<td>3B</td>
<td>91</td>
<td>01011011</td>
<td>5B</td>
<td>123</td>
<td>01111011</td>
<td>7B</td>
</tr>
<tr>
<td>28</td>
<td>00011100</td>
<td>1C</td>
<td>60</td>
<td>00111100</td>
<td>3C</td>
<td>92</td>
<td>01011100</td>
<td>5C</td>
<td>124</td>
<td>01111100</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>00011101</td>
<td>1D</td>
<td>61</td>
<td>00111101</td>
<td>3D</td>
<td>93</td>
<td>01011101</td>
<td>5D</td>
<td>125</td>
<td>01111101</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>00011110</td>
<td>1E</td>
<td>62</td>
<td>00111110</td>
<td>3E</td>
<td>94</td>
<td>01011110</td>
<td>5E</td>
<td>126</td>
<td>01111110</td>
<td>7E</td>
</tr>
<tr>
<td>31</td>
<td>00011111</td>
<td>1F</td>
<td>63</td>
<td>00111111</td>
<td>3F</td>
<td>95</td>
<td>01011111</td>
<td>5F</td>
<td>127</td>
<td>01111111</td>
<td>7F</td>
</tr>
</tbody>
</table>

Decimal, 8-bit binary, and 2-digit hex representations of integers from 0 to 127
6.1 Representing Information

<table>
<thead>
<tr>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
<th>dec</th>
<th>binary</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>01000000</td>
<td>80</td>
<td>160</td>
<td>10100000</td>
<td>A0</td>
<td>192</td>
<td>11000000</td>
<td>C0</td>
<td>224</td>
<td>11100000</td>
<td>E0</td>
</tr>
<tr>
<td>129</td>
<td>01000001</td>
<td>81</td>
<td>161</td>
<td>10100001</td>
<td>A1</td>
<td>193</td>
<td>11000001</td>
<td>C1</td>
<td>225</td>
<td>11100001</td>
<td>E1</td>
</tr>
<tr>
<td>130</td>
<td>01000010</td>
<td>82</td>
<td>162</td>
<td>10100010</td>
<td>A2</td>
<td>194</td>
<td>11000010</td>
<td>C2</td>
<td>226</td>
<td>11100010</td>
<td>E2</td>
</tr>
<tr>
<td>131</td>
<td>01000011</td>
<td>83</td>
<td>163</td>
<td>10100011</td>
<td>A3</td>
<td>195</td>
<td>11000111</td>
<td>C3</td>
<td>227</td>
<td>11100111</td>
<td>E3</td>
</tr>
<tr>
<td>132</td>
<td>01000100</td>
<td>84</td>
<td>164</td>
<td>10100100</td>
<td>A4</td>
<td>196</td>
<td>11000100</td>
<td>C4</td>
<td>228</td>
<td>11100100</td>
<td>E4</td>
</tr>
<tr>
<td>133</td>
<td>01000101</td>
<td>85</td>
<td>165</td>
<td>10100101</td>
<td>A5</td>
<td>197</td>
<td>11000101</td>
<td>C5</td>
<td>229</td>
<td>11100101</td>
<td>E5</td>
</tr>
<tr>
<td>134</td>
<td>01000110</td>
<td>86</td>
<td>166</td>
<td>10100110</td>
<td>A6</td>
<td>198</td>
<td>11000110</td>
<td>C6</td>
<td>230</td>
<td>11100110</td>
<td>E6</td>
</tr>
<tr>
<td>135</td>
<td>01000111</td>
<td>87</td>
<td>167</td>
<td>10101111</td>
<td>A7</td>
<td>199</td>
<td>11001111</td>
<td>C7</td>
<td>231</td>
<td>11101111</td>
<td>E7</td>
</tr>
<tr>
<td>136</td>
<td>01001000</td>
<td>88</td>
<td>168</td>
<td>10101000</td>
<td>A8</td>
<td>200</td>
<td>11001000</td>
<td>C8</td>
<td>232</td>
<td>11101000</td>
<td>E8</td>
</tr>
<tr>
<td>137</td>
<td>01001001</td>
<td>89</td>
<td>169</td>
<td>10101001</td>
<td>A9</td>
<td>201</td>
<td>11001001</td>
<td>C9</td>
<td>233</td>
<td>11101001</td>
<td>E9</td>
</tr>
<tr>
<td>138</td>
<td>01001010</td>
<td>8A</td>
<td>170</td>
<td>10101010</td>
<td>AA</td>
<td>202</td>
<td>11001010</td>
<td>CA</td>
<td>234</td>
<td>11101010</td>
<td>EA</td>
</tr>
<tr>
<td>139</td>
<td>01001011</td>
<td>8B</td>
<td>171</td>
<td>10101011</td>
<td>AB</td>
<td>203</td>
<td>11001011</td>
<td>CB</td>
<td>235</td>
<td>11101011</td>
<td>EB</td>
</tr>
<tr>
<td>140</td>
<td>01001100</td>
<td>8C</td>
<td>172</td>
<td>10101100</td>
<td>AC</td>
<td>204</td>
<td>11001100</td>
<td>CC</td>
<td>236</td>
<td>11101100</td>
<td>EC</td>
</tr>
<tr>
<td>141</td>
<td>01001101</td>
<td>8D</td>
<td>173</td>
<td>10101101</td>
<td>AD</td>
<td>205</td>
<td>11001101</td>
<td>CD</td>
<td>237</td>
<td>11101101</td>
<td>ED</td>
</tr>
<tr>
<td>142</td>
<td>01001110</td>
<td>8E</td>
<td>174</td>
<td>10101110</td>
<td>AE</td>
<td>206</td>
<td>11001110</td>
<td>CE</td>
<td>238</td>
<td>11101110</td>
<td>EE</td>
</tr>
<tr>
<td>143</td>
<td>01001111</td>
<td>8F</td>
<td>175</td>
<td>10101111</td>
<td>AF</td>
<td>207</td>
<td>11001111</td>
<td>CF</td>
<td>239</td>
<td>11101111</td>
<td>EF</td>
</tr>
<tr>
<td>144</td>
<td>01010000</td>
<td>90</td>
<td>176</td>
<td>10110000</td>
<td>B0</td>
<td>208</td>
<td>11010000</td>
<td>D0</td>
<td>240</td>
<td>11110000</td>
<td>F0</td>
</tr>
<tr>
<td>145</td>
<td>01010001</td>
<td>91</td>
<td>177</td>
<td>10110001</td>
<td>B1</td>
<td>209</td>
<td>11010001</td>
<td>D1</td>
<td>241</td>
<td>11110001</td>
<td>F1</td>
</tr>
<tr>
<td>146</td>
<td>01010010</td>
<td>92</td>
<td>178</td>
<td>10110010</td>
<td>B2</td>
<td>210</td>
<td>11010010</td>
<td>D2</td>
<td>242</td>
<td>11110010</td>
<td>F2</td>
</tr>
<tr>
<td>147</td>
<td>01010011</td>
<td>93</td>
<td>179</td>
<td>10110011</td>
<td>B3</td>
<td>211</td>
<td>11010011</td>
<td>D3</td>
<td>243</td>
<td>11110011</td>
<td>F3</td>
</tr>
<tr>
<td>148</td>
<td>01010100</td>
<td>94</td>
<td>180</td>
<td>10110100</td>
<td>B4</td>
<td>212</td>
<td>11010100</td>
<td>D4</td>
<td>244</td>
<td>11110100</td>
<td>F4</td>
</tr>
<tr>
<td>149</td>
<td>01010101</td>
<td>95</td>
<td>181</td>
<td>10110101</td>
<td>B5</td>
<td>213</td>
<td>11010101</td>
<td>D5</td>
<td>245</td>
<td>11110101</td>
<td>F5</td>
</tr>
<tr>
<td>150</td>
<td>01010110</td>
<td>96</td>
<td>182</td>
<td>10110110</td>
<td>B6</td>
<td>214</td>
<td>11010110</td>
<td>D6</td>
<td>246</td>
<td>11110110</td>
<td>F6</td>
</tr>
<tr>
<td>151</td>
<td>01010111</td>
<td>97</td>
<td>183</td>
<td>10110111</td>
<td>B7</td>
<td>215</td>
<td>11010111</td>
<td>D7</td>
<td>247</td>
<td>11110111</td>
<td>F7</td>
</tr>
<tr>
<td>152</td>
<td>01010000</td>
<td>98</td>
<td>184</td>
<td>10110000</td>
<td>B8</td>
<td>216</td>
<td>11010000</td>
<td>D8</td>
<td>248</td>
<td>11110000</td>
<td>F8</td>
</tr>
<tr>
<td>153</td>
<td>01010001</td>
<td>99</td>
<td>185</td>
<td>10110001</td>
<td>B9</td>
<td>217</td>
<td>11010001</td>
<td>D9</td>
<td>249</td>
<td>11110001</td>
<td>F9</td>
</tr>
<tr>
<td>154</td>
<td>01010010</td>
<td>9A</td>
<td>186</td>
<td>10110010</td>
<td>BA</td>
<td>218</td>
<td>11010010</td>
<td>DA</td>
<td>250</td>
<td>11110010</td>
<td>FA</td>
</tr>
<tr>
<td>155</td>
<td>01010011</td>
<td>9B</td>
<td>187</td>
<td>10110011</td>
<td>BB</td>
<td>219</td>
<td>11010011</td>
<td>DB</td>
<td>251</td>
<td>11110011</td>
<td>FB</td>
</tr>
<tr>
<td>156</td>
<td>01010100</td>
<td>9C</td>
<td>188</td>
<td>10110100</td>
<td>BC</td>
<td>220</td>
<td>11010100</td>
<td>DC</td>
<td>252</td>
<td>11110100</td>
<td>FC</td>
</tr>
<tr>
<td>157</td>
<td>01010101</td>
<td>9D</td>
<td>189</td>
<td>10110101</td>
<td>BD</td>
<td>221</td>
<td>11010101</td>
<td>DD</td>
<td>253</td>
<td>11110101</td>
<td>FD</td>
</tr>
<tr>
<td>158</td>
<td>01011110</td>
<td>9E</td>
<td>190</td>
<td>10111110</td>
<td>BE</td>
<td>222</td>
<td>11011110</td>
<td>DE</td>
<td>254</td>
<td>11111110</td>
<td>FE</td>
</tr>
<tr>
<td>159</td>
<td>01011111</td>
<td>9F</td>
<td>191</td>
<td>10111111</td>
<td>BF</td>
<td>223</td>
<td>11011111</td>
<td>DF</td>
<td>255</td>
<td>11111111</td>
<td>FF</td>
</tr>
</tbody>
</table>

Decimal, 8-bit binary, and 2-digit hex representations of integers from 128 to 255
Later in this chapter, we will be particularly interested in integers less than 256, which can be specified with 8 bits and 2 hex digits. For reference, we have included on the previous two pages a complete table of their representations in decimal, binary and hex. A few minutes studying this table is worth your while, to give you confidence in working with such integers and understanding relationships among these representations. If you believe, after doing so, that the table is a waste of space, then we have achieved our goal!

**Parsing and string representations** Converting among different representations of integers is an interesting computational task, which we first considered in Program 1.3.7 and then revisited in Exercise 2.3.15. We have also been making use of Java’s methods for such tasks throughout. Next, to cement ideas about positional number representations with various bases, we will consider a program for converting any number from one base to another.

**Parsing.** Converting a string of characters to an internal representation is called *parsing*. Since Section 1.1, we have been using Java methods like `Integer.parseInt()` and our own methods like `StdIn.readInt()` to convert numbers from the strings that we type to values of Java’s data types. We have been using decimal numbers (represented as strings of the characters between 0 and 9), now we look at a method to parse numbers written in any base. For simplicity, we limit ourselves to bases no more than 36 and extend our convention for hex to use the letters A through Z to represent digits from 10 to 35. *Note:* Java’s `Integer` class has a two-argument `parseInt()` method that has similar functionality, except that it also handles negative integers.

One of the the hallmark features of modern data types is that the *internal representation is hidden from the user*, so we can only use defined operations on data type values to accomplish the task. Specifically, it is best to limit direct reference to the bits that represent a data type value, but to use data-type operations instead.

The first primitive operation that we need to parse a number is a method that converts a character into an integer. Exercise 6.1.12 gives a method `toInt()` that takes a character in the range 0–9 or A–Z as argument and returns an int value between 0 and 35 (0–9 for digits and 10–35 for letters). With this primitive, the rather simple method `parseInt()` in Program 6.1.1 parses the string representation of
6.1 Representing Information

Program 6.1.1 Converting a natural number from one base to another

public class Convert
{
    public static int toInt(char c)
    {
        // See Exercise 5.1.12
    }
    public static char toChar(int i)
    {
        // See Exercise 5.1.13
    }
    public static int parseInt(String s, int d)
    {
        int N = 0;
        for (int i = 0; i < s.length(); i++)
        {
            N = d*N + toInt(s.charAt(i));
        }
        return N;
    }
    public static String toString(int N, int d)
    {
        if (N == 0) return "";
        return toString(N/d, d) + toChar(N % d);
    }
    public static String toString(int N, int d, int w)
    {
        // See Exercise 5.1.15
    }
    public static void main(String[] args)
    {
        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            int baseFrom = StdIn.readInt();
            int baseTo = StdIn.readInt();
            int N = parseInt(s, baseFrom);
            StdOut.println(toString(N, baseTo));
        }
    }
}

This general-purpose conversion program reads strings and pairs of bases from standard input and uses parseInt() and toString() to convert the string from a representation of an integer in the first base to a representation of the same integer in the second base.
an integer in any base \( b \) from 2 to 36 and returns the Java \( \text{int} \) value for that integer. As usual, we can convince ourselves that it does so by reasoning about the effect of the code in the loop: each time through the loop the \( \text{int} \) value \( N \) is the integer corresponding to all the digits seen so far: to maintain this invariant, all we need to do is multiply by the base and add the value of the next digit. The trace shown here illustrates the process: each value of \( N \) is the base times the previous value of \( N \) plus the next digit (in blue). To parse 1101101, we compute 0 \cdot 2 + 1 = 1, 1 \cdot 2 + 1 = 3, 3 \cdot 2 + 0 = 6, 6 \cdot 2 + 1 = 13, 13 \cdot 2 + 1 = 27, 27 \cdot 2 + 0 = 54, \) and 54 \cdot 2 + 1 = 109. To parse FACE as a hex number, we compute 0 \cdot 16 + 15 = 15, 15 \cdot 16 + 10 = 250, 250 \cdot 16 + 12 = 4012, and 4012 \cdot 16 + 14 = 64206. This is a special case of polynomial evaluation via Horner’s method (see Exercise 2.1.32).

For simplicity, we have not included error checks in this code. For example, \( \text{parseInt()} \) should raise an exception if the value returned by \( \text{toInt()} \) is not less than the base. Also, it should throw an exception on overflow, as the input could be a string that represents a number larger than can be represented as a Java \( \text{int} \).

**String representation.** Using a \( \text{toString()} \) method to compute a string representation of a data-type value is also something that we have been doing since the beginning of this book. We use a recursive method that generalizes the decimal-to-binary method (the solution to Exercise 2.3.15) that we considered earlier in this section. Again, it is instructive to look at a method to compute the string representation of an integer in any given base, even though Java’s \( \text{Integer} \) class has a two-argument \( \text{toString()} \) method that has similar functionality.

Again, the first primitive operation that we need is a method that converts an integer into a character (digit). Exercise 6.1.13 gives a method \( \text{toChar()} \) that takes an \( \text{int} \) value between 0 and 35 and returns a character in the range 0–9 (for values less than 10) or A–Z (for values from 10 to 35). With this primitive, the \( \text{toString()} \) method in Program 6.1.1 is even simpler than \( \text{parseInt()} \). It is a recursive method based on the idea that the last digit is the character representation of \( N \mod d \) and the rest of the string is the string representation of \( N \div d \). The computation is essen-
6.1 Representing Information

tially the inverse of the computation for parsing, as you can see from the call trace shown at the bottom of the previous page.

When discussing the contents of computer words, we need to include leading zeros, so that we are specifying all the bits. For this reason, we include a three-argument version of `toString()` in Convert, where the third argument is the desired number of digits in the returned string. For example, the call `toString(15, 16, 4)` returns `000F` and the call `toString(14, 2, 16)` returns `0000000000001110`. Implementation of this version is left for an exercise (see Exercise 6.1.15).

Putting these ideas all together, Program 6.1.1 is a general-purpose tool for computing numbers from one base to another. While the standard input stream is not empty, the main loop in the test client reads a string from standard input, followed by two integers (the base in which the string is expressed and the base in which the result is to be expressed) and performs the specified conversion and prints out the result. To accomplish this task, it uses `parseInt()` to convert the input string to a Java `int`, then it uses `toString()` to convert that Java `int` to a string representation of the number in the specified base. You are encouraged to download and make use of this tool to familiarize yourself with number conversion and representation.
**Integer arithmetic** The first operations that we consider on integers are basic arithmetic operations like addition and multiplication. Indeed, the primary purpose of early computing devices was to perform such operations repeatedly. In the next chapter, we will be studying the idea of building computational devices that can do so, since every computer has built-in hardware for performing such operations. For the moment, we illustrate that the basic methods that you learned in grade school for decimal work perfectly well in binary and hex.

**Addition.** In grade school you learned how to add two decimal integers: add the two least significant digits (rightmost digits); if the sum is more than 10, then carry a 1 and write down the sum modulo 10. Repeat with the next digit, but this time include the carry bit in the addition. The same procedure generalizes to any base. For example, if you are working in hex and the two summand digits are 7 and E, then you should write down a 5 and carry a 1 because $7 + E$ is 15 in hex. If you are working in binary and the two summand bits are 1 and the carry is 1 then you should write down a 1 and carry the 1 because $1 + 1 + 1 = 11$ in binary. The examples at left illustrate how to compute the sum $4567_{10} + 366_{10} = 4933_{10}$ in decimal, hex, and binary. As in grade school, we suppress leading zeros.

**Unsigned integers.** If we want to represent integers within a computer word, we are immediately accepting a limitation on the number and size of integers that we can represent. As already noted, we can represent only $2^n$ integers in an $n$-bit word. If we want just non-negative (or unsigned) integers the natural choice is to use binary for the integers 0 through $2^n - 1$, with leading 0s so that every word corresponds to an integer and every integer within the defined range to a word. The table at right shows the 16 unsigned integers we can represent in a 4-bit word, and the table at left shows the range of representable integers for the 16-bit, 32-bit, and 64-bit word sizes that are used in typical computers.
6.1 Representing Information

Overflow. As you have already seen with Java programming in Section 1.2, we need to pay attention that the value of the result of an arithmetic operation does not exceed the maximum possible value. This condition is called overflow. For addition in unsigned integers, overflow is easy to detect: if the last (leftmost) addition causes a carry, then the result is too large to represent. Testing the value of one bit is easy, even in computer hardware (as you will see), so computers and programming languages typically include low-level instructions to test for this possibility. Remarkably, Java does not do so (see the Q&A in Section 1.2).

Multiplication. Similarly, as illustrated in the diagram at right, the grade-school algorithm for multiplication works perfectly well with any base. (The binary example is difficult to follow because of cascading carries: if you try to check it, add the numbers two at a time.) Actually, computer scientists have discovered multiplication algorithms that are much more suited to implementation in a computer and much more efficient than this method. In early computers, programmers had to do multiplication in software (we will illustrate such an implementation much later, in Exercise 5.3.38). Note that overflow is much more of a concern in developing a multiplication algorithm than for addition, as the number of bits in the result can be twice the number of bits in the operands. That is, when you multiply two \( n \)-bit numbers, you need to be prepared for a \( 2n \)-bit result.

In this book, we certainly cannot describe in depth all of the techniques that have been developed to perform arithmetic operations with computer hardware. Of course, you want your computer to perform division, exponentiation, and other operations efficiently. Our plan is to cover addition/subtraction in full detail and just some brief indication about other operations.

You also want to be able to compute with negative numbers and real numbers. Next, we briefly describe standard representations that allow for that.
Negative numbers  Computer designers discovered early on that it is not difficult to modify the integer data type to include negative numbers, using a representation known as two’s complement.

The first thing that you might think of would be to use a sign-and-magnitude representation, where the first bit is the sign and the rest of bits the magnitude of the number. For example, with 4 bits in this representation 0101 would represent +5 and 1101 would represent −5. By contrast, in \( n \)-bit two’s complement, we represent positive numbers as before, but each negative number \(-x\) with the (positive, unsigned) binary number \(2^n - x\). For example, the table at left shows the 16 two’s complement integers we can represent in a 4-bit word. You can see that 0101 still represents +5 but 1011 represents −5 because \(2^4 - 5 = 11_{10} = 1011_2\).

With one bit reserved for the sign, the largest two’s complement number that we can represent is about half the largest unsigned integer that we could represent with the same number of bits. As you can see from the 4-bit example, there is a slight asymmetry in two’s complement: We represent the positive numbers 1 through 7 and the negative numbers −8 through −1 and we have a single representation of 0. In general, in \( n \)-bits two’s complement, the smallest possible negative number is \(-2^{n-1}\) and the largest possible positive number is \(2^{n-1} - 1\). The table at left shows the smallest and largest (in absolute value) 16-bit two’s complement integers.

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>2</td>
<td>0000000000000010</td>
</tr>
<tr>
<td>3</td>
<td>0000000000000011</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32765</td>
<td>0111111111111101</td>
</tr>
<tr>
<td>32766</td>
<td>0111111111111110</td>
</tr>
<tr>
<td>32767</td>
<td>0111111111111111</td>
</tr>
<tr>
<td>-32768</td>
<td>1000000000000000</td>
</tr>
<tr>
<td>-32767</td>
<td>1000000000000001</td>
</tr>
<tr>
<td>-32766</td>
<td>1000000000000010</td>
</tr>
<tr>
<td>-32765</td>
<td>1000000000000011</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-3</td>
<td>1000000000001101</td>
</tr>
<tr>
<td>-2</td>
<td>1111111111111110</td>
</tr>
<tr>
<td>-1</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

16-bit integers (two’s complement)

There are two primary reasons that two’s complement evolved as the standard over sign-and-magnitude. First, because there is only one representation of 0 (the binary string that is all 0s), testing whether a value is 0 is as easy as possible. Second, arithmetic operations are easy to implement—we discuss this for addition below. Moreover, as with sign-and-magnitude, the leading bit indicates the sign, so testing whether a value is negative is as easy as possible. Building computer hardware is sufficiently difficult that achieving these simplifications just by adopting a convention on how we represent numbers is compelling.
6.1 Representing Information

Addition. Also, adding two $n$-bit two’s complement integers is easy: add them as if they were unsigned integers. For example, $2 + (-7) = 0010 + 1001 = 1011 = -5$. Proving that this is the case when result is within range (between $-2^{n-1}$ and $2^{n-1} - 1$) is not difficult:

- If both integers are nonnegative then standard binary addition as we have described it applies, as long as the result is less than $2^n - 1$.
- If both are negative, then the sum is $(2^n - x) + (2^n - y) = 2^n + 2^n - (x + y)$
- If $x$ is negative, $y$ is positive, and the result is negative, then we have $(2^n - x) + y = 2^n - (x - y)$
- If $x$ is negative, $y$ is positive, and the result is positive, then we have $(2^n - x) + y = 2^n + (y - x)$

In the second and fourth cases, the extra $2^n$ term does not contribute to the $n$-bit result (it is the carry out), so a standard binary addition (ignoring the carry out) gives the result. Detecting overflow is a bit more complicated than for unsigned integers—we leave that for the Q&A.

Subtraction. To compute $x - y$ we compute $x + (-y)$. That is we can still use standard binary addition, if we know how to compute $-y$. It turns out that negating a number is very easy in two’s complement: flip the bits and then add 1. Three examples of this process are shown at left—we leave the proof that it works for an exercise.

Knowing two’s complement is relevant for Java programmers because short, int, and long values are 16-, 32-, and 64-bit two’s complement integers, respectively. This explains the bounds on values of these types that Java programmers have to be aware of (shown in the table at the top of the next page).
Moreover, Java’s two’s complement representation explains the behavior on overflow in Java that we first observed in Section 1.2 (see the Q&A in that section, and Exercise 1.2.10). For example, we saw that, in any of Java integer types, the result of adding 1 to the largest positive integer, the result is the largest negative integer. In 4-bit two’s complement, incrementing 0111 gives 1000; in 16-bit two’s complement, incrementing 0111111111111111 gives 1000000000000000. (Note that this is the only case where incrementing a two’s complement integer does not produce the expected result.) The behavior of the other cases in Exercise 1.2.10 are also as easily explained. For decades, such behavior has bewildered programmers who do not take the time to learn about two’s complement. Here’s one convincing example: in Java, the call Math.abs(-2147483648) returns -2147483648, a negative integer!

Real numbers  How do we represent real numbers? This task is a bit more challenging than integers, as there are many choices to be made. Early computer designers tried many, many options and numerous competing formats evolved during the first few decades of digital computation. Arithmetic on real numbers was actually implemented in software for quite a while, and was quite slow by comparison with integer arithmetic.

By the mid 1980s, the need for a standard was obvious (different computers might get slightly different results for the same computation), so the Institute for Electrical and Electronic Engineers (IEEE) developed a standard known as the IEEE 754 standard that is under development to this day. The standard is extensive—you may not want to know the full details—but we can describe the basic ideas briefly here. We illustrate with a 16-bit version known as the IEEE 754 half-precision binary floating-point format or binary16 for short. The same essential ideas apply to the 32-bit and 64-bit versions used in Java.

**Floating point.**  The real-number representation that is commonly used in computer systems is known as floating-point. It is just like scientific notation, except that everything is representing in binary. In scientific notation, you are used to working with numbers like

<table>
<thead>
<tr>
<th></th>
<th>smallest</th>
<th>largest</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>32-bit</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
</tbody>
</table>

Representable two’s complement integers

---

Anatomy of a floating point number

$$2^{exponent} \times 1.1001000100$$

**sign**

**fraction**

**always 1**

**exponent**

[insert diagram of anatomy of a floating point number]
Typically the number is expressed such that the coefficient is one (non-zero) digit. This is known as a normalization condition. In floating point, we have the same three elements.

**Sign.** The first bit of a floating point number is its sign. Nothing special is involved: the sign bit is 0 if the number is positive (or zero) and 1 if it is negative. Again, checking whether a number is positive or negative is easy.

**Exponent.** The next \( t \) bits of a floating point number are devoted to its exponent. The number of bits used for `binary16`, `binary32`, and `binary64` are 5, 8, and 11, respectively. The exponent of a floating point number is not expressed in two’s complement, but rather offset-binary, where we take \( R = 2^{t-1} - 1 \) (15, 127, and 1023 for \( t = 5, 8, \) and 11, respectively) and represent any decimal number \( x \) between \(-R\) and \( R+1\) with the binary representation of \( x+R \). The table at right shows the 5-bit offset binary representations of the numbers between \(-15\) and +16. In the standard, 0000 and 1111 are used for other purposes.

**Fraction.** The rest of the bits in a floating point number are devoted to the coefficient: 10, 23, and 53 bits for `binary16`, `binary32`, and `binary64`, respectively. The normalization condition implies that the digit before the decimal place in the coefficient is always 1, so we need not include that digit in the representation!

Given these rules, the process of decoding a number encoded in IEEE 754 format is straightforward, as illustrated in the top example in the figure at the top of the next page. According to the standard, the first bit in the given 16-bit quantity is the sign, the next five bits are the offset binary encoding of the exponent \((-6_{10})\), and the next 10 bits are the fraction, which defines the coefficient 1.1012. The process of encoding a number, illustrated in the bottom example, is more complicated, due to the need to normalize and to extend binary conversion to include fractions. Again, the first bit is the sign bit, the next five bits are the exponent, and the next 10 bits are the fraction. These tasks make for
an challenging programming exercise even in a high-level language like Java (see Exercise 6.1.25, but first read about manipulating bits in the next subsection), so you can imagine why floating point numbers were not supported in early computer hardware and why it took so long for a standard to evolve.

The Java Float and Double data types include a floatToIntBits() method that you can use to check floating-point encoding. For example, the call

\[
\text{Convert.toString(Float.floatToIntBits(100.25), 2, 16)}
\]

prints the result 0101011001000100 as expected from the bottom example above.

**Arithmetic.** Performing arithmetic on floating point numbers also makes for an interesting programming exercise. For example, the following steps are required to multiply two floating point numbers:

- Exclusive or the signs.
- Add the exponents.
- Multiply the fractions.
- Normalize the result.

If you are interested, you can explore the details of this process and the corresponding process for addition and for multiplication by working Exercise 5.1.25. Addition is actually a bit more complicated than multiplication, because it is necessary to “unnormalize” to make the exponents match as the first step.

**Computing with floating point numbers is often challenging** because they are most often approximations to the real numbers of interest, and errors in the approximation can accumulate during a long series of calculations. Since the 64-bit format (used in Java’s double data type) has more than twice as many bits in the fraction as the 32-bit format (used in Java’s float data type), most programmers choose to use double to lessen the effects of approximations errors, as we do in this book.
**Java code for manipulating bits**  As you can see from floating-point encoding of real numbers, encoding information in binary can get complicated. Next, we consider the tools available within Java that make it possible to write programs to encode and decode information. These tools are made possible because Java defines integer values to be two’s complement integers, and makes explicit that the values of the `short`, `int`, and `long` data types are 16-, 32-, and 64-bit binary two’s complement, respectively. Not all languages do so, leaving such code to a lower-level language, defining an explicit data type for bit sequences, and/or perhaps requiring difficult or expensive conversion. We focus on 32-bit `int` values, but the operations also work for `short` and `long` values.

**Binary and hex literals.** In Java, it is possible to specify integer literal values in binary (by prepending `0b`) and in hex (by prepending `0x`). This ability substantially clarifies code that is working with binary values. You can use literals like this anywhere that you can use a normal literal; it is just another way of specifying an integer value. If you assign a hex literal to an `int` variable and specify fewer than 8 digits, Java will fill in leading zeros. A few examples are shown in the table at right.

**Shifting and bitwise operations in Java code.** To allow clients to manipulate the bits in an `int` value, Java supports the operations shown in the table at the bottom of this page. We can complement the bits, do bitwise logical operations, and shift left or right a given number of bit positions. For shift right, there are two options: *logical* shift, where 0s are filled in at the left, and *arithmetic* shift, where vacated

<table>
<thead>
<tr>
<th>values</th>
<th>32-bit integers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typical literals</code></td>
<td><code>0b00000000000000000000000000000000</code> <code>0b1111 0xF</code> <code>0x1234</code></td>
</tr>
<tr>
<td><code>operations</code></td>
<td></td>
</tr>
<tr>
<td><code>bitwise complement</code></td>
<td><code>~</code></td>
</tr>
<tr>
<td><code>bitwise and</code></td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td><code>bitwise or</code></td>
<td>`</td>
</tr>
<tr>
<td><code>bitwise xor</code></td>
<td><code>^</code></td>
</tr>
<tr>
<td><code>shift left</code></td>
<td><code>&lt;&lt;</code></td>
</tr>
<tr>
<td><code>shift right</code></td>
<td><code>&gt;&gt;&gt;</code></td>
</tr>
<tr>
<td><code>shift right arithmetic</code></td>
<td><code>&gt;&gt;</code></td>
</tr>
</tbody>
</table>

*Bit manipulation operators for Java's built-in `int` data type*
positions are filled with the sign bit (see the Q&A at the end of this section). Examples of several of these operations are shown at right.

**Shifting and masking.** One of the primary uses of such operations is masking, where we isolate a bit or a group of bits from the others in the same word. Usually we prefer to specify masks as hex constants. For example, the mask 0x8000000 can be used to isolate the leftmost bit in a 32-bit word, the mask 0x000000FF can be used to isolate the rightmost 8 bits, and the mask 0x007FFFF can be used to isolate the rightmost 23 bits.

Going a bit further, we often do shifting and masking to extract the integer value that a contiguous group of bits represent, as follows:

- Use a shift right instruction to put the bits in the rightmost position.
- If we want $k$ bits, create a literal mask whose bits are all 0 except its $k$ rightmost bits, which are 1.
- Use a bitwise and to isolate the bits. The 0s in the mask lead to zeros in the result; the 1s in the mask give the bits of interest in the result.

This sequence of operations puts us in a position to use the result as we would any other int value, which is often what is desired. Later in this chapter we will be interested in shifting and masking to isolate hex digits, as shown in the example at left.

**Shifting and masking example**

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0008A2B &gt;&gt; 8</td>
<td>0x0000008A</td>
</tr>
<tr>
<td>(0x0008A2B &gt;&gt; 8) &amp; 0xF</td>
<td>0x0000000A</td>
</tr>
</tbody>
</table>

As an example of a practical application of bitwise operations, Program 6.1.2 illustrates the use of shifting and masking to extract the sign, exponent and fraction from a floating point number. Most computer users are able to work comfortably without dealing with data representations at this level (indeed, we have hardly needed it so far in this book), but, as you will see, bit manipulation plays an important role in all sorts of applications.
6.1 Representing Information

Program 6.1.2  Extracting the components of a floating point number

```java
public class ExtractFloat
{
    public static void main(String[] args)
    {
        while (!StdIn.isEmpty())
        {
            float x = StdIn.readFloat();
            int t = Float.floatToIntBits(x);
            if ((t & 0x7FFFFFFF) == 1)
                StdOut.println("     Sign: -");
            else StdOut.println("     Sign: +");
            int exp = ((t >> 23) & 0xFF) - 127;
            StdOut.println("Exponent: "+ exp);
            double frac = 1.0 * (t & 0x007FFFFF) / (1 << 23);
            StdOut.println("Fraction: "+ frac);
            StdOut.println((float) (Math.pow(2, exp) * (1 + frac)));
        }
    }
}
```

This program illustrates the use of Java bit manipulation operations by extracting the sign, exponent and fraction fields from float values entered on standard input, then using the exponent and fraction to recompute the absolute value of the number.

% java ExtractFloat
-100.25
    Sign: -
    Exponent: 6
    Fraction: 0.56640625
100.25
    Sign: +
    Exponent: 1
    Fraction: 0.5707963705062866
3.141592653589793
    Sign: +
    Exponent: 1
    Fraction: 0.5707963705062866
3.1415927410125732

829
Characters

In order to process text, we need a binary encoding for characters. The basic method is quite simple: a table defines the correspondence between characters and $n$-bit unsigned binary integers. With six bits, we can encode 64 different characters; with seven bits, 128 different characters, with eight bits, 256 different characters and so forth. As with floating point, many different schemes evolved as computers came into use, and people still use different encodings in different situations.

ASCII. the American Standard Code for Information Interchange (ASCII) code was developed as a standard in the 1960s, and has been in widespread use ever since. It is a 7-bit code, though in modern computing it most often is used in 8-bit bytes with the leading bit ignored.

One of the primary reasons for the development of ASCII was for communication via teletypewriters that could send and receive text. Accordingly, many of the encoded characters are control characters for such machines. Some of the control characters were for communications protocols (for example, ACK means “acknowledge”); others controlled the printing aspect of the machine (for example, BS means “backspace” and CR means “carriage return”).

The table at left is a definition of ASCII that provides the correspondence that you need to convert from 8-bit binary (equivalently, 2-digit hex) to a character and back. Use the first hex digit as a row index and the second hex digit as a column index to find the character that it encodes. For example, 31 encodes the digit 1, 4A encodes the letter J, and so forth. This table is for 7-bit ASCII, so the first hex digit must be 7 or less. Hex numbers starting with 0 and 1 (and the numbers 20 and 7F) correspond to non-printing control characters such as CR, which now means “new line” (most of the others are rarely used in modern computing).

Unicode. In the connected world of the 21st century, it is necessary to work with many more than the 100 or so ASCII characters from the 20th century, so a new standard known as Unicode is emerging. By using 16 bits for most characters (and
up to 24 or 32 for some characters), Unicode can support tens of thousands of characters and a broad spectrum of the world’s languages. The UTF-8 encoding (from sequences of characters to sequences of 8-bit bytes and vice-versa, most characters mapping to two bytes) is rapidly emerging as a standard. The rules are complicated, but comprehensive, and implemented in most modern systems (such as Java) so programmers generally need not worry much about the details. ASCII survives within Unicode: all ASCII characters map to 1 byte, so ASCII files are special cases of UTF-8 files (and backwards compatible).

\[
\begin{align*}
\text{ASCII (two chars) to binary} \quad & \quad \text{PC} \\
& \quad 0101000001000011 \\
\text{binary to ASCII (two chars)} \quad & \quad 0010101001110110 \\
\text{ASCII-binary conversion examples} \quad & \quad \text{V}
\end{align*}
\]

We generally pack as much information as possible in a computer word, so it is possible to encode two ASCII characters in 16 bits (as shown in the example at right), four characters in 32 bits, eight characters in 64 bits, and so forth. In high-level languages such as Java, such details and UTF-8 encoding and decoding are implemented in the `String` data type, which we have been using throughout the book. Still, it is often important for Java programmers to understand some basic facts about the underlying representation, as it can certainly affect the resource requirements of programs. For example, many programmers discovered that the memory usage of their programs suddenly doubled when Java switched from ASCII to Unicode in the 2000s, and began using a 16-bit `char` to encode each ASCII character. Experienced programmers know how to pack two ASCII characters per `char` to save memory when critical.
Summary  Generally, it is wise to write programs that function properly independent of the data representation. Many programming languages fully support this point of view. But it can stand in direct opposition to the idea of taking full advantage of the capability of a computer, by using its hardware the way it was designed to be used. Java’s primitive types are intended to support this point of view. For example, if the computer has hardware to add or multiply 64-bit integers, then, if we have a huge number of such operations to perform, we would like each add or multiply to reduce to a single instruction so that our program can run as fast as possible. For this reason, it is wise for the programmer to try to match data types having performance-critical operations with the primitive types that are implemented in the computer hardware. Achieving the actual match might involve deeper understanding of your system and its software, but striving for optimal performance is a worthwhile endeavor.

You have been writing programs that compute with various types of data. Our message in this section is that since every sequence of bits can be interpreted in many different ways, the meaning of any given sequence of bits within a computer depends on the context. You can write programs to interpret bits any way that you want. You cannot tell from the bits alone what type of data they represent, or even whether they represent data at all, as you will see.

To further emphasize this point, the table below gives several different 16-bit strings along with their values if interpreted as hex integers, unsigned integers, two’s complement integers, binary16 floating point numbers, and pairs of characters. This are but a few early examples of the myriad available ways of representing information within a computer.

<table>
<thead>
<tr>
<th>binary</th>
<th>hex</th>
<th>unsigned</th>
<th>2’s comp</th>
<th>floating point</th>
<th>ASCII chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100100110100</td>
<td>1234</td>
<td>4,660</td>
<td>4,660</td>
<td>0.0007572174072265625</td>
<td>DC2 4</td>
</tr>
<tr>
<td>111111111111111</td>
<td>FFFF</td>
<td>65,535</td>
<td>−1</td>
<td>−131008.0</td>
<td>DEL DEL</td>
</tr>
<tr>
<td>1111101011001110</td>
<td>FACE</td>
<td>64,206</td>
<td>−1,330</td>
<td>−55744.0</td>
<td>e N</td>
</tr>
<tr>
<td>0101011001000100</td>
<td>5644</td>
<td>22,052</td>
<td>22,052</td>
<td>100.25</td>
<td>V D</td>
</tr>
<tr>
<td>10000000000000001</td>
<td>8001</td>
<td>32,769</td>
<td>−32,767</td>
<td>−0.0000305473804473876953125</td>
<td>NUL SOH</td>
</tr>
<tr>
<td>0101000001000011</td>
<td>5043</td>
<td>20,547</td>
<td>20,547</td>
<td>34.09375</td>
<td>P C</td>
</tr>
<tr>
<td>000111001010111</td>
<td>1CAB</td>
<td>7,339</td>
<td>7,339</td>
<td>0.004558563232421875</td>
<td>FS +</td>
</tr>
</tbody>
</table>

Five ways to interpret various 16-bit values
Q. How do I find out the word size of my computer?

A. You need to find out the name of its processor, then look for the specifications of that processor. Most likely, you have a 64-bit processor. If not, it may be time to get a new computer.

Q. Why does Java use 32 bits for `int` values when most computers have 64-bit words?

A. That was a design decision made a long time ago. Java is unusual in that it completely specifies the representation of an `int`. The advantage of doing so is that old Java programs are more likely to work on new computers than in languages where machines might use different representations. The disadvantage is that 32 bits is often not enough. For example, in 2014 Google had to change from a 32-bit representation for view count after it became clear that the video *Gangham Style* would be watched more than 2,147,483,647 times. In Java, you can switch to `long`.

Q. This seems like something that could be taken care of by the system, right?

A. Some languages, for example Python, place no limit on the size of integers, leaving it to the system to use multiple words for integer values when necessary. In Java, you can use the `BigInteger` class.

Q. What’s the `BigInteger` class?

A. It allows you to compute with integers without worrying about overflow. For example, if you import `java.math.BigInteger`, then the code

```java
BigInteger x = new BigInteger("2");
StdOut.println(x.pow(100));
```

prints 1267650600228229401496703205376, the value of $2^{100}$. You can think of a `BigInteger` as a string (the internal representation is more efficient than that), and the class provides methods for standard arithmetic operations and many other operation. For example, this method is useful in cryptography, where arithmetic operations on numbers with hundreds of digits play a critical role in some systems. The implementation works with many digits as necessary, so overflow is not a concern. Of course, operations are much more expensive than built-in `long` or `int`
operations, so Java programmers do not use `BigInteger` for integers that fit in the range supported by `long` or `int`.

**Q.** Why hexadecimal? Aren’t there other bases that would do the job?

**A.** Base 8, or octal, was widely used for early computer systems with 12-bit, 24-bit, or 36-bit words, because the contents of a word could be expressed with 4, 8, or 12 octal digits, respectively. An advantage over hex in such systems was that only the familiar decimal digits 0-7 were needed, so that primitive I/O devices like numeric keypads could be used both for decimal numbers and octal numbers. But octal is not convenient for 32-bit and 64-bit word sizes, because those word sizes are not divisible by 3. (They are not divisible by 5 or 6 either, so no switch to a larger base is likely.)

**Q.** How can I guard against overflow?

**A.** It is not so easy, as a different check is needed for each operation. For example, if you know that `x` and `y` are both positive and you want to compute `x + y`, you could check that `x < Integer.MAX_VALUE - y`.

**A.** Another approach is to “upcast” to a type with a bigger range. For example, if you are calculating with `int` values, you could convert them to `long` values, then convert the result back to `int` (if it is not too big).

**A.** In Java 8, you can use `Math.AddExact()`, which throws an exception on overflow.

**Q.** How might hardware detect overflow for two’s complement addition?

**A.** The rule is simple, though it is a bit tricky to prove: check the values of the carry in to the leftmost bit position and the carry out of the leading bit position. Overflow is indicated if they are different (see the examples at right).

**Q.** What is the purpose of the arithmetic shift?
6.1 Representing Information

A. For two’s complement integers, arithmetic shifting right by 1 is the same as integer division by 2. For example, the value of \((-16) \gg 1\) is \(-2\), as illustrated at right. To test your understanding of this operator, figure out the values of \((-3) \gg 1\) and \((-1) \gg 1\). This convention is called “sign extension,” as opposed to “zero extension” for logical shifts.

Q. What is \(x \gg y\) if \(y\) is negative or greater than 31?

A. Java only uses the five low-order bits of the second operand. This behavior coincides with the physical hardware on typical computers.

Q. I never really understood the examples in the Q&A in Section 1.2 that claim that \((0.1 + 0.1 == 0.2)\) is true but \((0.1 + 0.1 + 0.1 == 0.3)\) is false. Can you elaborate, now?

A. A literal like \(.1\) or \(.3\) in Java source code is converted to the nearest 64-bit IEEE-754 number (the one whose least significant bit is 0 in case of a tie), a Java double value. Here are the values for the literals \(.1\), \(.2\), and \(.3\):

<table>
<thead>
<tr>
<th>literal</th>
<th>nearest 64-bit IEEE 754 number</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td>0.1000000000000000000000000055511151231257827021181583404541015625</td>
</tr>
<tr>
<td>.2</td>
<td>0.2000000000000000000000000111022302462515654042363166809082031250</td>
</tr>
<tr>
<td>.3</td>
<td>0.29999999999999999999999988977697537484345957636833190917968750</td>
</tr>
</tbody>
</table>

As you can see, \(.1 + .1\) is equal to \(.2\), but \(.1 + .1 + .1\) is greater than \(.3\). The situation is not so different from noticing that, for integers, \(2/5 + 2/5\) is equal to \(4/5\) (they are both 0), but \(2/5 + 2/5 + 2/5\) is not equal to \(6/5\).

Q. System.out.println(.1) prints .1, not the value in the above table. Why?

A. Few programmers need that much precision, so println() truncates for readability. There must be at least one digit to represent the fractional part, and beyond that as many, but only as many, more digits as are needed to uniquely distinguish the argument value from adjacent values of type double. You can use printf() for more precise control over the format, and BigDecimal for extended precision.
6.1.1 Convert the decimal number 92 to binary.
Answer: 1011100.

6.1.2 Convert the octal number 31415 to binary.
Answer: 01100110001101.

6.1.3 Convert the octal number 314159 to decimal.
Answer: That is not an octal number! You can do the computation, even with Convert, to get the result 104561, but 9 is just not a legal octal digit. The version of Convert on the booksite includes such legality checks (see also Exercise 5.1.12). It is not unusual for a teacher to try this trick on a test, so beware!

6.1.4 Convert the hexadecimal number BB23A to octal.
Answer: First convert to binary 1011 1011 0010 0011 1010, then consider the bits three at a time 10 111 011 001 000 111 010, and convert to octal 2731072.

6.1.5 Add the two hexadecimal numbers 23AC and 4B80 and give the result in hexadecimal. Hint: add directly in hex instead of converting to decimal, adding, and converting back.

6.1.6 Assume that \( m \) and \( n \) are positive integers. How many 1 bits are there in the binary representation of \( 2^{m+n} \)?

6.1.7 What is the only decimal integer whose hexadecimal representation has its digits reversed?
Answer: 53 is 35 in hex.

6.1.8 Prove that converting a hexadecimal number one digit at a time to binary and vice versa always gives the correct result.

6.1.9 IPv4 is the protocol developed in the 1970s that dictates how computers on the Internet communicate. Each computer on the Internet needs its own Internet address. IPv4 uses 32 bit addresses. How many computers can the Internet handle? Is this enough for every mobile phone and every toaster to have their own?
6.1.10 IPv6 is an Internet protocol in which each computer has a 128 bit address. How many computers would the Internet be able to handle if this standard is adopted? Is this enough?

*Answer: $2^{128}$. That at least enough for the short term—5000 addresses per square micrometer of the Earth's surface!*

6.1.11 Fill in the values of the expressions in this table:

<table>
<thead>
<tr>
<th>expression</th>
<th>~0xFF</th>
<th>0x3 &amp; 0x5</th>
<th>0x3</th>
<th>0x5</th>
<th>0x3 &amp; 0x5</th>
<th>0x3</th>
<th>0x5</th>
<th>0x1234 &lt;&lt; 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.1.12 Develop an implementation of the `toInt()` method specified in the text for converting a character in the range 0–9 or A–Z into an int value between 0 and 35.

*Answer:*

```java
public static int toInt(char c) {
    if ((c >= '0') && (c <= '9')) return c - '0';
    return c - 'A' + 10;
}
```

6.1.13 Develop an implementation of the `toChar()` method specified in the text for converting an int value between 0 and 35 into a character in the range 0–9 or A–Z.

*Answer:*

```java
public static char toChar(int i) {
    if (i < 10) return (char) ('0' + i);
    return (char) ('A' + i - 10);
}
```
6.1.14 Modify Convert (and the answers to the previous two exercises) to use long, test for overflow, and check that the digits in the input string are within the range specified by the base.

Answer: See Convert.java on the booksite.

6.1.15 Add to Convert a version of the toString() method that takes a third argument, which specifies the length of the string to be produced. If the specified length is less than needed, return only the rightmost digits; if it is greater, fill in with leading 0 characters. For example, toString(64206, 16, 3) should return "ACE" and toString(15, 16, 4) should return "000F". Hint: First call the two-argument version.

6.1.16 Compose a Java program TwosComplement that takes an int value i and a word size w from the command line and prints the w-bit two's complement representation of i and the hex representation of that number. Assume that w is a multiple of 4. For example, your program should behave as follows:

```
% java TwosComplement -1 16
1111111111111111 FFFF
% java TwosComplement 45 8
00101101 2D
% java TwosComplement -1024 32
11111111111111111111110000000000 FFFFC00
```

6.1.17 Modify ExtractFloat to develop a program ExtractDouble that accomplishes the same task for double values.

6.1.18 Write a Java program EncodeDouble that takes a double value from the command line and encodes it as a floating-point number according to the IEEE 754 binary32 standard.
### 6.1.19 Fill in the blanks in this table.

<table>
<thead>
<tr>
<th>binary</th>
<th>floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010001000110100</td>
<td></td>
</tr>
<tr>
<td>1000000000000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.09375</td>
</tr>
<tr>
<td></td>
<td>1024</td>
</tr>
</tbody>
</table>

### 6.1.20 Fill in the blanks in this table.

<table>
<thead>
<tr>
<th>binary</th>
<th>hex</th>
<th>unsigned</th>
<th>2's comp</th>
<th>ASCII chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001000110100111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–131</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
6.1.21 IP addresses and IP numbers An IP address (IPV4) is comprised of integers \(w, x, y,\) and \(z\) and is typically written as the string \(w.x.y.z\). The corresponding IP number is given by \(16777216w + 65536x + 256y + z\). Given an IP number \(N\), the corresponding IP address is derived from \(w = (N / 16777216) \mod 256, x = (N / 65536) \mod 256, y = (N / 256) \mod 256, z = N \mod 256\). Write a function that takes an IP number and returns a String representation of the IP address, and another function takes an IP address and returns an int corresponding to the IP number. For example, given \(3401190660\) the first function should return \(202.186.13.4\).

6.1.22 IP address. Write a program that takes a 32 bit string as a command line argument, and prints out the corresponding IP address in dotted decimal form. That is, take the bits 8 at a time, convert each group to decimal, and separate each group with a dot. For example, the binary IP address \(01010000000100000000000000000001\) should be converted to \(80.16.0.1\).

6.1.23 MAC address. Write functions to convert back-and-forth between MAC addresses and 48-bit long values.

6.1.24 Base64 encoding. Base64 encoding is a popular method for sending binary data over the Internet. It converts arbitrary data to ASCII text, which can be emailed back between systems without problems. Write a program to read in an arbitrary binary file and encode it using Base64.

6.1.25 Floating point software. Write a class \ Código de clase del punto flotante \(\text{FloatingPoint}\) that has three instance variables \(\text{sign}, \text{exponent},\) and \(\text{fraction}\). Implement addition and multiplication. Include \(\text{toString()}\) and \(\text{parseFloat()}\). Support 16-, 32-, and 64-bit formats.
6.1.26 DNA encoding. Develop a class DNA that supports an efficient representation of strings that are comprised exclusively of a, c, t, or g characters. Include a constructor that converts a string to the internal representation, a toString() method to convert the internal representation to a string, a charAt() method that returns the character at the specified index, and an indexOf() method that takes a String p as argument and returns the first occurrence of p in the represented string. For the internal representation, use an array of int values, packing 16 characters in each int (two bits per character).
6.2 TOY

To help you better understand the nature of computation on your computer, we introduce in this section TOY, an imaginary machine designed for this book that is very similar to the computers that first came into widespread use in the 1970s. We study it today because it also shares the essential characteristics of the modern day microprocessors found in your mobile device and your computer and everywhere else, not to mention countless other computing devices developed in the intervening years. The figure at the bottom of this page depicts a PDP-8, a real computer from the 1970s, and TOY, our imaginary computer.

TOY demonstrates that simple computational models can perform useful and nontrivial calculations, and also serves a reference point that can help you understand the basic characteristics of your own computer. One of the remarkable facts of the evolution of computation over the past several decades is that all computers share the same basic architecture, an approach that was widely adopted almost immediately after it was first articulated by John von Neumann in 1945.

We begin by introducing the basic constituent parts of the TOY machine. There are only a few, and the purpose of each is easy to understand. All computers are made up of similar components.

Then we describe how to use and program the TOY machine. Starting with the basic ways of representing information that we covered in the previous section, we look at operations on such information. In other words, we are working...
with the data types that the TOY machine hardware implements: sets of values and operations on those values. Working at this level is known as machine language programming. Studying programming at the machine language level will help you better understand the relationship between your Java programs and your computer, and also to better understand the nature of computation itself. Machine language programming is actually still used today in performance-critical applications such as video processing, audio processing, and scientific computing. As you will see, it is not difficult to learn how to program in machine language.

In Chapter 7, we describe how to build such a machine in hardware. This will be the final step in demystifying your computer, helping you to better understand the connection between your Java programs and the physical world.

This essential layer of abstraction is found in all computers. A complete description of precisely what a processor can do provides a target language for translating a program written in a high-level language like Java while at the same time providing a blueprint for building a circuit that can implement the machine.

**Brief historical note** It is perhaps a bit difficult for you to imagine a modern world without computers. To pick a point in time, consider the 1950s, when the world was becoming industrialized after World War II.

At that time, there were cars, airplanes, satellites, and all sorts of other technological developments that are recognizable today, but there were no computers available to the average person or even the average scientist or engineer.

The original motivation for building computers was to be able to perform calculations for applications of all sorts in science, engineering, and commerce. The war itself proved the point, from the ballistics tables computed by John von Neumann to the code-breaking machine developed by Alan Turing, not to mention the calculations done at Los Alamos that enabled the development of the atomic bomb. And imagine running a bank or building a car without a computer.

The most important tool used for calculations by a typical science or engineering student before the 1970s was the slide rule, a decidedly non-electronic and non-digital device that nonetheless was very useful, particularly for computing logarithms and doing multiplications. Another tool in common use was a book of tables of functions: for example, to compute \( \sin(x) \) you would look it up in the book!
Once computers did start to become available, they were generally shared by a group of people and they were cumbersome to use, as you will see. Still, it was immediately apparent that computing would be a vast improvement over slide rules and function tables. Within a very short amount of time, people were sharing large computers in ways that made slide rules and tables of functions obsolete. For many years people used calculators, based on the same technology as computers but specialized for calculations, and handheld. For simple calculations, calculators persist. For complex calculations, scientists, engineers, and applications developers wrote computer programs, then as now. It is quite remarkable that the basic design of the first devices that were created for such purposes has persisted to the present day and still supports the ocean of applications that have transformed the world.

**TOY components** We begin with an overview of the basic design components that have been found in virtually all computers for over half a century, but are reduced to their essentials in our TOY machine.

**Memory.** The memory is a critical component of any computer. It not only holds data to be processed and the results of the computation, but also any program that the machine is to run. TOY’s memory consists of 256 words, each 16 bits. That cer-

```
TOY memory dump (256 16-bit words)

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | A  | B  | C  | D  | E  | F  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0A | 8A | 15  | 8A | 2B | 7101| 7101| 7800| 8AFF| 7101| 7101| BBOE| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 1  | 7BEF| 8B16| 8B2C| 75FF| 7A00| 8CF8| 8BFF| A90A| A90A| 1EE1| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 2  | 9AFF| 1CAB| 2CAB| 7901| 7B01| CC55| 7101| 140A| 180A| 900E| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 3  | 9BFF| 9C17| CC29| 2C59| 894C| 188C| 7900| 7800| C98F| 1EE1| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 4  | 7101| 0000| DC27| CC3B| C94B| 188C| 7900| 7800| C98F| 1EE1| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 5  | 7900| 0000| 2BBA| 1991| 9AFF| 98FF| C26B| 2991| 2CBC| 1EE1| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 6  | 22B9| 0005| C022| 1A09| 1CAB| 0000| 1CA9| 2441| CC96| EF00| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 7  | C200| 0000| EF00| 8B3D| 1AB0| 0000| 8DFF| AC04| 1991| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 8  | 1CA9| 0000| 0000| FF22| 1BC0| 0000| BD0C| 2EBC| 1809| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| 9  | AD0C| 0000| 00C3| 2AA1| 2991| 0000| 1991| DE7B| A909| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| A  | 9DFF| 0000| 0111| CA3C| 0044| 0000| C064| 1B0C| DCBE| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| B  | 1441| 0000| 0000| 9B3E| 0000| 0000| 1AA9| C074| 1981| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| C  | C006| 0000| 0000| 0000| 0000| 000C| FF60| BBOA| EF00| 1809| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| D  | 0000| 0000| 0000| 005B| 0000| FF70| EF00| 0000| A909| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| E  | 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
| F  | 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000| 0000|
```
tainly is not much by today’s standards, but you will be surprised at the range of calculations it can support. And here is a sobering thought: Those $256 \times 16 = 4096$ bits have $2^{4096}$ different possible values, so the vast majority of things that TOY can do will never be observed in this universe.

With hex notation, we can specify the contents of a memory word with four hex digits. Furthermore, we consider the words to be numbered from 0 to 255, so that we can refer to each word with 2-digit hex number known as its address. For example, we might say that “the value of the word at address 1E is 0FA2.” or just “memory location 1E is 0FA2.” For economy, we often just use array notation and say that “M[1E] is 0FA2.” We can specify the contents of the TOY memory using a 16-row table like the one at the bottom of the previous page. The first column gives values for locations 00 to 0F; the second column locations 10 to 1F, and so forth. Such a table is known as a memory dump. In this case, the memory contains all the programs that we consider in this chapter (!)

*Instructions.* A critical characteristic of the TOY machine (and virtually all other computers) is that the contents of a memory word might be interpreted either as data or as an instruction, depending on the context. For example, you know from the previous section that the value 1234 might be interpreted as representing the integer $4660_{10}$ or the real number $0.00302886962890625$; in this section you will learn that it might also represent a machine instruction that adds two numbers. It is up to the programmer to ensure that data is treated as data, and that instructions are treated as instructions. We will examine how all TOY instructions are encoded shortly, so that you will know how to decode any 16-bit value as an instruction (and how to encode any instruction as a 16-bit value).

*Registers.* A register is a machine component that holds a sequence of bits, much like a word in main memory. Registers serve the function of holding intermediate results during computation. You can think of them as playing the role of variables in TOY programming. TOY has 16 registers, numbered from 0 through F. As with memory, we use array notation and refer to the registers with the designations R[0] through R[F]. Since they are 16 bits, the same as memory words, we represent the contents of each register with a 4-digit hex value, and the contents of all the registers with a table of 16 4-digit hex numbers. The table at right shows the contents of the registers during a typical computation, which we examine later. By convention, R[0] is always 0000.
**Arithmetic and logic unit.** The arithmetic and logic unit (ALU) is TOY’s computational engine—the workhorse of the machine that performs all its calculations. Typically, a TOY instruction directs the ALU to compute some function that takes two registers as arguments and put the result in a third register. For example, the TOY instruction 1234 says to direct the contents of R[2] and R[3] to the ALU, add them, and then direct the result to R[4]. Later in this section, we will describe how to compose TOY programs based upon such instructions.

**Program counter and instruction register.** The 8-bit program counter (PC) is an internal register that holds the address of the next instruction to be executed. The 16-bit instruction register (IR) is an internal register that contains the current instruction being executed. These registers are central to the operation of the machine. The IR is not directly accessed by programs, but programmers are always aware of its contents.

The diagram at the bottom of this page is a schematic that includes these basic components. In Chapter 7, we will consider how to create a circuit that implements all of them. For the rest of this chapter, we will consider how they operate and how a programmer can control them to get them to perform a desired computation.

**Fetch-increment-Execute Cycle** The TOY machine executes instructions by taking a specific sequence of actions, repeatedly. First it checks the value of the PC and fetches (copies) the contents of this memory location into the IR. Next, it increments the program counter by 1. (For example, if the program counter is 10, it gets

```
Components of the TOY machine
```
incremented to 11.) Finally, it interprets the 16-bit value in the IR as an instruction and executes it according to the rules that characterize the TOY machine, which we will describe shortly. Each instruction can modify the contents of various registers, main memory, or even the program counter itself. After executing the instruction, the machine repeats the whole fetch-increment-execute cycle, using the new value of the program counter to find the next instruction. This process continues forever, or until the machine executes a halt instruction. As with Java, it is possible to write programs that go into infinite loops. To stop the TOY machine in an infinite loop, a programmer would have to turn it off, or perhaps even to unplug it.

**Instructions** Any 16-bit value (the contents of any memory word) can be interpreted as a TOY instruction. The purpose of each instruction is to modify the state of the machine (the value of a memory word, a register, or the PC) in some way. To describe the operation of the instructions, we use pseudo-code, which is much like Java code except that it refers to memory words, registers, and the PC directly.

**Anatomy of an instruction.** We use hex to encode instructions: each 16-bit instruction is four hex digits. The first digit of an instruction is its opcode, which specifies the operation it performs. There are 16 different instructions, so one hex digit suffices to specify one of them. The second digit of an instruction specifies a register—each instruction uses or changes the value of some register. Since there are 16 registers, one hex digit suffices to specify one of them.

Most of the instructions are coded in one of two instruction formats, which tell us how to interpret the third and fourth hex digits. In RR-format instructions each of the two remaining hex digits refer to a register. In A-format instructions, the third and fourth hex digits (together) specify a memory address.

**Instruction set.** On the next page is a table that describes all of TOY’s instructions. This table is a complete reference guide to programming in TOY, to be consulted when you compose TOY programs. Next, we describe the instructions in detail.
<table>
<thead>
<tr>
<th>opcode</th>
<th>description</th>
<th>format</th>
<th>pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>RR</td>
<td>R[d] = R[s] + R[t]</td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>RR</td>
<td>R[d] = R[s] - R[t]</td>
</tr>
<tr>
<td>3</td>
<td>bitwise and</td>
<td>RR</td>
<td>R[d] = R[s] &amp; R[t]</td>
</tr>
<tr>
<td>4</td>
<td>bitwise xor</td>
<td>RR</td>
<td>R[d] = R[s] ^ R[t]</td>
</tr>
<tr>
<td>5</td>
<td>left shift</td>
<td>RR</td>
<td>R[d] = R[s] &lt;&lt; R[t]</td>
</tr>
<tr>
<td>6</td>
<td>right shift</td>
<td>RR</td>
<td>R[d] = R[s] &gt;&gt; R[t]</td>
</tr>
<tr>
<td>7</td>
<td>load address</td>
<td>A</td>
<td>R[d] = addr</td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>A</td>
<td>R[d] = M[addr]</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>A</td>
<td>M[addr] = R[d]</td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>RR</td>
<td>R[d] = M[R[t]]</td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>RR</td>
<td>M[R[t]] = R[d]</td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>A</td>
<td>if (R[d] == 0) PC = addr</td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>A</td>
<td>if (R[d] &gt; 0) PC = addr</td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>–</td>
<td>PC = R[d]</td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>A</td>
<td>R[d] = PC + 1; PC = addr</td>
</tr>
</tbody>
</table>

**Halt.** Opcode 0, the most basic instruction, simply directs the machine to halt—stop the fetch-increment-execute cycle. At this point, the programmer can examine the contents of memory to see the results of the computation. TOY ignores the other three hex digits of a halt, so 0000, 0123, and 0FFF are all halt instructions.

**Arithmetic instructions.** Opcodes 1 and 2 are arithmetic instructions, which invoke the ALU to perform an arithmetic operation on two registers (R[s] and R[t]), putting the result in a third (R[d]). For example, the instruction 1234 means “add R[3] to R[4] and put the result in R[2]” and 2AAC means “subtract R[C] from R[A] and put the result in R[A].”

**Memory address instructions.** Opcodes 7, A, and B are memory address instructions, which we use to manipulate addresses in TOY. For example, the instruction 7423 means “set R[4] to the value 0023” or just R[4] = 0023 (note the leading 0s). Then opcodes A and B can be used to reference memory indirectly via the address in a register. Understanding these instructions is a helpful way to better understand references in Java. We will examine their use in more detail later when we look at implementing arrays and linked structures.

Another important use of opcode 7 is as an integer-data-type instruction: once the bits are loaded into the register, we can use them in an arithmetic instruction (interpret them as representing an integer). For example, we use the instruction 7C01 to set R[C] to 0001 (R[C] = 0001).

**Logical instructions.** Opcodes 3 through 6 are logical instructions, which invoke the ALU to perform operations on the bits in the registers, like the operations that we considered for Java in Section 5.1. Opcode 3 is “bitwise and” where each bit in R[d] is set to 1 if the corresponding bits in R[s] and R[t] are both 1; otherwise it
6.2 TOY

is set to 0. Similarly, opcode 4 is “bitwise xor” where each bit in $R[d]$ is set to 1 if the corresponding bits in $R[s]$ and $R[t]$ are the same; otherwise it is set to 0. Opcode 5 leaves in $R[d]$ the result of shifting the bits in $R[s]$ left the number of bit positions given in $R[t]$, discarding the bits shifted out and shifting in 0 bits as needed. Opcode 6 is similar, but bits are shifted right and the bits shifted in match the sign bit (see the Q&A in Section 6.1). The logical instructions complete the implementation of TOY’s int data type in a manner that corresponds to Java. In TOY, as in Java, we sometimes bend data abstraction rules by treating the values just as sequences of 16 bits, not integers. Shift and bitwise logical instructions are useful in implementing and decoding all types of data, just as in Java code.

**Memory instructions.** Opcodes 8 and 9 are memory instructions, which transfer data between memory and the registers. For example, the instruction 8234 means “load into $R[2]$ the value of memory word $M[34]$,” or $R[2] = M[34]$ for short; and the instruction 9234 means “store into memory word $M[34]$ the value of $R[2]$,” or $M[34] = R[2]$.

**Flow of control instructions.** Opcodes C through F are flow of control instructions, which modify the PC and are essential for implementing the flow-of-control constructs like conditionals, loops, and functions that are fundamental in programming. For example, the instruction C212 means “set the PC to 12 if $R[2]$ is 0” and D212 means “set the PC to 12 if $R[2]$ is positive.” Note in particular that C0xx means “set the PC to xx”, since $R[0]$ is always zero. This is known as an unconditional branch. Changing the value of the PC has the effect of changing the flow of control because the next instruction is always taken from the memory address given by the PC. We will examine opcodes E and F in more detail later when we look at implementing functions.

Your first reaction to this set of instructions might be that it is minimal at best. That is certainly true, but one of the goals of this chapter is to convince you that a small set of instructions like this suffices to write programs equivalent to the Java programs that you learned in the first half of this book, or any program. For the moment, the most important thing to remember is that you now have the information you need to decode any 16-bit value as a TOY instruction.
Your first TOY program  "Hello, World" for TOY is a program that adds two integers, shown in Program 6.2.1 on the opposite page. As with HelloWorld.java, in Section 1.1, we start with such a simple program to allow us to focus on the the details of running the program. The code in Program 6.2.1, known as machine code, illustrates the various conventions that we use for TOY programs:

- All data and code relevant to a given program is included.
- Each line gives a 2-digit (hex) memory address and the 4-digit (hex) value of the word at that address.
- The starting value for the PC is always the address of the first instruction, highlighted in blue.
- The third column gives pseudo code for each instruction.

The program itself is just the five 4-digit hex numbers stored in memory locations 10-14. The pseudo-code makes this program very easy to understand—it reads much like a Java program.

To trace a TOY program, we simply write down the PC and IR values for each instruction executed and the value of any affected register or memory word after execution. The table below the code gives such a trace for Program 6.2.1. To specify the result of the computation, we list the contents of memory when the halt instruction is reached, with the halt instruction itself and any memory locations whose values have changed highlighted in blue. In this case, only one memory value changes: location 17 gets the computed result 000D.

This process seems exceedingly simple, but we still have not described the process of actually getting the program to run. For Java, we were able to describe to you how to use an editor to create a file containing the program, the using a compiler and Java runtime to execute it and view the results in the terminal window. For TOY, you have to consider that there is no operating system, no applications, certainly no editor, terminal emulator, compiler, or runtime—not even a keyboard or a display.

Indeed, the picture at the bottom of the facing page shows the “result” of running Program 6.2.1: the lights at the bottom of the front panel of the computer display the computed result 000D, in binary: 000000000001101. Next, we consider, step by step, the process of getting this program to run on the TOY machine and to achieve this result.
Program 6.2.1 Your first TOY program

10 8A15 R[A] = M[15]  \hspace{1cm} \text{load first summand into a}
11 8B16 R[B] = M[16]  \hspace{1cm} \text{load second summand into b}
12 1CAB R[C] = R[A] + R[B]  \hspace{1cm} c = a + b
13 9C17 M[17] = R[C]  \hspace{1cm} \text{store result}
14 0000 halt
15 0008 \hspace{0.5cm} \text{integer value 8}
16 0005 \hspace{0.5cm} \text{integer value 5}
17 0000 \hspace{0.5cm} \text{result}

Started with the PC at 10, this program adds the two numbers at memory locations 15 and 16 and puts the result 000D in memory location 17.

memory dump

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8A15</td>
<td>0008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8B16</td>
<td>0008</td>
<td>0005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1CAB</td>
<td>0008</td>
<td>0005</td>
<td>000D</td>
<td>000D</td>
</tr>
<tr>
<td>13</td>
<td>9C17</td>
<td>0008</td>
<td>0005</td>
<td>000D</td>
<td>000D</td>
</tr>
<tr>
<td>14</td>
<td>0000</td>
<td>0008</td>
<td>0005</td>
<td>000D</td>
<td>000D</td>
</tr>
</tbody>
</table>

\text{instruction execution trace}
Operating the machine  You communicate with your computer through I/O devices like the keyboard, display, and trackpad. TOY has I/O devices, too—in a sense. Next, we describe how programmers would communicate with machines like TOY, to actually run programs. At left is a depiction of our the front panel of our imaginary TOY machine. It has just three simple devices for control, input, and output: pushbuttons, switches, and lights. All of them are simple on/off mechanisms, but there are very few of them: just 24 switches and lights and just 4 pushbuttons. Nothing else. No keyboard, printer, or display. No internet connection, wireless card, speakers, or touchpad. Just buttons for control, switches for input, and lights for output. Still, this is enough to perform worthwhile computations with a general-purpose computer with the same basic characteristics as your own, as we now describe.

Pushbuttons.  Perhaps the most basic control for a computer is to turn it on or off. The PDP–8 had a key for this purpose; TOY has a pushbutton. The first thing a programmer would do is to turn the machine on (and the last thing would be to turn it off). Additionally, three other basic functions are controlled by pushbuttons:

- Load a word into the computer’s memory.
- Look at the value of a word in the computer’s memory.
- Run a program.

We will discuss each of these functions in context shortly.

Switches.  Programmers using such machines specify binary values with on-off switches. A switch in the up position denotes 1; a switch in the down position denotes 0. The TOY machine has two banks of switches: an 8-switch bank for specifying a location in the memory and a 16-switch bank for specifying a memory-word value. These switches are TOY’s input devices.

Lights.  TOY’s output devices are the banks of lights under the switches. Again, an 8-light bank specifies a location in the memory and a 16-light bank specifies a memory-word value.

Running a program.  To run a program on a machine such as TOY, a programmer would typically reserve time with the machine and show up at the appointed
time with the program written down on a piece of paper. To fix ideas, we consider in full detail the steps needed to run our first program. For brevity, we “think in hex” by specifying hex values for switches and lights, when in reality each switch or light corresponds to a bit. For example, when we say “set the DATA switches to 1CAB” we mean “turn on the DATA switches corresponding to 1s in the bitstring 0001110010101011.” That said, here is how a programmer would implement and run Program 6.2.1:

- Turn on the machine (press the ON/OFF button).
- Set the ADDR switches to 10 and the DATA switches to 8A15, then press LOAD.
- Set the ADDR switches to 11 and the DATA switches to 8B16, then press LOAD.
- Set the ADDR switches to 12 and the DATA switches to 1CAB, then press LOAD.
- Set the ADDR switches to 13 and the DATA switches to 9C01, then press LOAD.
- Set the ADDR switches to 14 and the DATA switches to 0000, then press LOAD.
- Set the ADDR switches to 15 and the DATA switches to 0008, then press LOAD.
- Set the ADDR switches to 16 and the DATA switches to 0005, then press LOAD.
- Set the ADDR switches to 10, then press the RUN button.
- Set the ADDR switches to 17, then press the LOOK button.
- Write down the computed answer, shown in the DATA lights (000D).
- (Typically) repeat the previous five steps with other data values.
- Turn off the machine (press the ON/OFF button).

In summary, when we turn the machine on, we cannot assume anything about the values in the memory, registers, and PC (except that R[0] is 0000), so we need to load the program and the data (the seven steps after turning the machine on above) before we can run the program. After running the program, we need to examine the memory location containing the result to learn the answer.

It is worthwhile to reflect on the truly fundamental nature of this interface. Essentially, programmers would communicate with the machine one bit at a time. Crude as this process was, people put up with it because developing programs was far superior to doing calculations with pencil and paper, slide rules, or mechanical calculators, the only other widely available alternatives at the time. We will soon see many examples where short programs yield computed values that would otherwise be very difficult to learn.

Better input/output devices such as keyboards, printers, paper tapes, and magnetic tapes soon followed, of course, but this method of programming was certainly a starting point for many scientists, engineers, and students (yes, this is how students at universities would learn to program in the early 1970s).
**Conditionals and loops** To consider increasingly interesting TOY programs, we are following a similar approach to the one we used when you first learned programming, in Chapter 1. We have considered data types and basic operations in our description of machine instructions; now we move to control constructs.

The first control flow constructs that you learned in Chapter 1 were conditionals and loops. Accordingly, we next consider implementations of these constructs with TOY's branch statements.

As an example, we consider *Euclid's algorithm* for computing the greatest common divisor (GCD) of two positive integers \(a\) and \(b\). We studied a version of this algorithm in Section 2.3 that is based on integer division (with remainder). Since TOY has no division instruction, we will start with the following version, implemented in Java.

```java
public static int gcd(int a, int b) {
    while (a != b) {
        if (b > a) b = b - a;
        else a = a - b;
    }
    return a;
}
```

This code is based on a simple idea: if \(b\) is greater than \(a\), any number that divides both \(a\) and \(b\) also divides both \(a\) and \(b - a\) (in particular the GCD) and if \(a\) is greater than \(b\), any number that divides both \(a\) and \(b\) also divides both \(b\) and \(a - b\). When \(a\) and \(b\) are positive, each iteration of the loop decreases the larger of the two (but it stays positive), so the process must eventually end with \(a\) and \(b\) equal—to the GCD of all the numbers encountered, including the original \(a\) and \(b\). A trace of the values of \(a\) and \(b\) for sample inputs is shown at right.

Euclid’s algorithm was first articulated over two thousand years ago, and many versions of it have been studied since. This particular version can sometimes be slow: for example, you may have noticed in the trace that 1092 is subtracted six times before \(a\) becomes less than \(b\). (Taking the remainder accomplishes the same task with one division.) If one of the numbers is very small, then the algorithm can take time proportional to the magnitude of the other. For example the algorithm takes 7214 iterations to find that the greatest common divisor of 7215 and 7214 is 1. Rest assured that versions of Euclid’s algorithm that are much more efficient...
than this have been studied in quite some detail, even for machines like TOY (see Exercise 6.2.20).

For the moment, we will concentrate on implementing the code in the body of this function, assuming that the programmer will enter the input data in specified memory locations, as in Program 6.2.1. In the next section, we will discuss how to package the code as a function.

The key to the implementation is effective use of TOY’s branch statements to implement loops and conditionals, as illustrated by the diagrams on this page.

To implement a while loop (diagram at right), we put the code that computes the value of the expression starting at some memory location yy and implement the loop with the instruction C0yy, an unconditional branch to yy. Within the loop, we put the code for evaluating the expression, arranging that it leave zero in some register (say R[1]) if and only if the value is false. Then we use the conditional branch C0xx to transfer control to the instruction after the loop (at address xx) if the register is zero. The implementation of an if statement, illustrated at left, is similar.

From these constructions, it is clear that that any loop or conditional in a Java program can be directly implemented in a TOY program. Each line of Java code corresponds to just a few TOY branches. As further evidence, several of the exercises at the end of this section address implementations of Java programs that you learned early in this book. As with Java, conditionals and loops take us from a world of
programming where we execute only a few instructions to one where we can execute thousands or millions of instructions, or more.

With TOY programs, we are not even limited to these ways of implementing conditionals and loops. For example, the implementation in Program 6.2.2 actually happens to evaluate just one conditional expression for both the while loop and the if statement. Indeed, programmers can use branch instructions to set up flow-of-control structures in TOY that cannot be naturally expressed with conditionals and loops. It took many years for people to become comfortable with the idea that it is best to use just the few building blocks (conditionals, loops, and nesting) that we use in modern programming. For clarity, we take a middle-of-the-road approach in this book, where we are guided by the constructs just considered (and we use Java-like code as documentation), but we take shortcuts that simplify the code when appropriate.

Thus, a programmer could use the switches to enter Program 6.2.2 and its data in memory locations 20 through 2D, then set the address switches to 20, press RUN, and observe the result in the lights by examining location 2D. The program computes the GCD of 195 and 273, which is 39. The programmer could run the program as often as desired, entering new pairs of numbers in 2B and 2C, resetting the address switches to 20, pressing RUN, and observing the result in 2D.

Following the trace at right is a worthwhile way to develop more comfort with TOY programming: First, the program loads 195 (00C3) into R[A] and 273 (0111) into R[B]. Then it subtracts them, putting the result −78 (FFC3) into R[C]. Since this value is not zero, it enters the loop, and then tests whether the difference is negative or positive. Since it is negative, it subtracts R[A] from R[B] leaving 78 (004E) in R[B] and then goes back for another iteration of the loop. In the next iteration of the loop, it subtracts R[B] from R[A] leaving 117 (0075) in R[A]. Then it subtracts R[B] from R[A] again, leaving 39 (0027) in R[A]. The last iteration of the loop subtracts R[A] from R[B] leaving the result 39 (0027) in both registers.

This is a classic computation, and we can imagine the excitement experienced by early programmers realizing that such computations could so easily be relegated to the computer. Would anyone rather perform such computations by hand? The ability to quickly implement algorithms like this appealed to the mathematicians and scientists who became early programmers, stimulated the search for faster algorithms, and unleashed research on developing efficient algorithms that continues to this day.
Program 6.2.2  Conditionals and loops: Euclid’s algorithm

22  2CAB  R[C] = R[A] - R[B]  while (a != b)
23  CC29  {
24  DC27  if (b > a)
26  C022  else
28  C022  }
29  9A2D  return a (= b = GCD)  return a

2A  0000  halt
2B  00C3  p: integer value 195
2C  0111  q: integer value 273
2D  0000  result

Started with the PC at 20, this program finds the greatest common divisor of the two numbers at memory locations 2B and 2C and puts the result in memory location 2D.
**Stored-program computing**  One of the essential characteristics of the TOY machine is that it stores computer programs as numbers, and both data and programs are stored in the same main memory. This is a profound idea with a fascinating history that is crucial to understanding the basic nature of computation.

*Instructions as data and data as instructions.* As an illustration of the fundamental idea, consider Program 6.2.3, which adds a sequence of numbers. The sequence could be of any length, terminated by 0000. A trace of the operation of this program is shown at right, and is worthy of careful study (R[1] and M[1B] are omitted from the trace because each of their values change only once). The computation starts out in a manner similar to Program 6.2.1, but then does something quite different. After adding the first two numbers, and leaving the result in R[A], the program loads the instruction 8B1D at location 12 into R[D], adds 1 to it, then stores the result 8B1E back into location 12. Then, it branches back to location 12, where that instruction loads into R[B] the next number to be added to R[A], continuing in a loop until encountering 0000 in the data.

Such code is known as self-modifying code. We have included this program because it succinctly illustrates the fundamental concept of stored-program computing. Is the contents of memory location 12 an instruction or data? It is both! When we add 1 to it, it is data, but when the PC refers to it and it is loaded into the IR, it is an instruction. Since the program and data share the same memory, the machine can modify its data or the program itself while it is executing. That is, code and data are the same, or at least they can be. The memory is interpreted as an instruction when the program counter references it, and as data when an instruction references it.

Self-modifying code liked this is rarely used in modern computing because it is difficult to understand, debug, and maintain. We will consider an alternative method of adding a sequence of numbers in the next section. But the ability to process instructions as data is fundamental in computing, as discussed next and throughout the rest of this chapter.

*Some implications.* On reflection, you can see that ability to treat the program as data is crucial and essential in our modern computational infrastructure:

- Any *application* is treated as data while you are downloading or installing it, but as a program once you launch it.
- *Compilers* are programs that read in other programs as input data and produce machine-language programs as output data, so all *programming*
**Program 6.2.3  Self-modifying code: Compute a sum**

10  7101  R[1] = 1  
11  8A1C  R[A] = M[1C]  load first number into a  
12  8B1D  R[B] = M[1D*]  while (b != 0)  
13  CB19  } 
14  1AAB  a = a + b  
16  1D1D  R[D] = R[D] + 1  to load next number into b 
17  9D12  M[12] = R[D]  on next iteration  
18  C012  }  
19  9A1B  store result  
1A  0000  halt  
1B  0000  result  
1C  0001  data  
1D  0008  
1E  0018  
1F  0040  
20  0000

Started with the PC at 10, this program adds the sequence of numbers at memory locations 1C through 1F (terminated by 0000) and stores the result in memory location 1B.
languages depend on this capability.

- Modern cloud computing is based on the concept of a virtual machine, where one computer runs a program written for another computer. Indeed, TOY itself is a virtual machine, as you will see in Section 6.4.

These are but a few examples, and we will be revisiting this concept throughout this chapter.

Treating programs as data is not without its perils. For example, computer viruses are (malicious) programs that propagate by writing new programs or modifying existing ones. And, as we saw in Chapter 5, it is a consequence of Turing’s theory that there is no effective way in general to tell the difference between a malicious virus, a useful application, or data. This practical downside is an inescapable consequence of the stored-program model. We will examine a specific example in the context of our TOY machine in the next section.

Von Neumann machines As already mentioned, by the 1940s and 1950s scientists and engineers were performing extensive calculations, not just for wartime applications such as ballistics, atomic weapons and cryptography, but also for peacetime applications like space flight and meteorology. The idea that electronic components could be much faster than mechanical ones was a powerful one.

But many early computers were emulating mechanical calculators. Operators had to “program” the computer by plugging cables and setting banks of switches, which was tedious, time-consuming, and error-prone. Any memory was devoted to data. One such machine was the ENIAC, being developed in the mid-1940s at the University of Pennsylvania by Eckert and Mauchly.

At the same time (actually a bit earlier, in the 1930s) there was great excitement in the field of mathematics because Alan Turing had developed the ingenious theoretical constructs that we just considered in Chapter 5. Turing’s work provided deep insights into the true nature of computation.

Princeton scholar John von Neumann worked as a consultant with Eckert and Mauchly on the ENIAC project and its planned successor, the EDVAC. He was interested both in the ballistics calculations that were the primary purpose of the machine and in the extensive calculations that were needed for the development of the atom bomb.

In 1945, while on a train from Princeton to Los Alamos, von Neumann wrote up his report on planned improvements to ENIAC. This memo, First Draft of a Report on the EDVAC, is a complete description of the stored-program model of
computing. Since von Neumann was a professor at Princeton while Turing was a graduate student there, he certainly was influenced by Turing’s ideas, and he was in a unique position to bridge the gap between Turing’s theories and the practical challenges faced by Eckert and Mauchly. Soon after von Neumann’s arrival at Los Alamos, a young lieutenant named Herman Goldstine recognized that there would be intense interest in the idea, and he circulated the memo widely. Scientists around the world immediately saw the value of the stored-program model, and computers based on the model (virtually all computers) have been called *von Neumann machines* ever since. Many historians believe that Eckert and Mauchly deserve credit for the idea (as does Turing!) but von Neumann’s memo was certainly the spark that made it happen around the world.

The stored-program model enables computers to perform any type of computation, without requiring the user to physically alter or reconfigure the hardware. This simple but fundamental model has been used in virtually every computer since von Neumann first articulated it.

In hindsight, the von Neumann architecture may seem obvious. However, plenty of research groups were working in a different direction at the time, and the question of whether a computer built around a stored program model can be as powerful as a computer that can be re-wired and reconfigured was debated. In fact, Turing’s theory shows that the ability to physically reconfigure a computer does not enable it to solve more problems, so long as basic instruction set is rich enough (as is the case even with our TOY machine). Other than Turing himself, von Neumann was one of the few people in the world in a position to appreciate this. His ability to fully articulate the idea in a practical context on a long train ride was a serendipitous development that profoundly changed the world.
Q. Did programmers really flip switches to enter programs?

A. Yes. Many, many people learned this skill. Even when better input/output devices became available, it was necessary to enter a program through switches that could drive one of the devices.

Q. What's the difference between a register and a memory word?

A. Both store 16-bit integers, but they play different roles within the computer. The purpose of the memory is to hold programs and data—we want the memory to be as large as possible. The purpose of the registers is to provide an intermediate staging ground to get data to and from the ALU—only a limited number of registers are really needed. Typically, computers use more expensive technology for registers because they are involved in virtually every instruction. The number of registers in a computer is a design decision.

Q. Are special purpose computers or microprocessors still fabricated today?

A. Yes, because it is possible to do simple things faster in hardware than software.

Q. It's hard to imagine programming without thinking in terms of loops and conditionals. Did people really work that way?

A. Most certainly. Programmers designed their logic with flowcharts, and early high-level languages had a “goto” statement that translated to a machine-language branch. The idea of “structured programming” using just loops, conditionals, and functions emerged in academia but was not taken seriously by many programmers until the 1970s. One famous turning point was a letter to the editor of the Communications of the ACM by E. W. Dijkstra in 1968, entitled “Goto considered harmful.” In this letter he argued for the “goto” statement to be abolished in all higher-level programming languages because programs that use it are too difficult to understand, debug and maintain. This point of view was universally embraced within a decade, and structured programming has been taken for granted since.
6.2.1 How many bits of memory does TOY have? Include all registers (including the PC) and main memory.

6.2.2 TOY uses 8-bit memory addresses which means that it is possible to access 256 words of memory. How many words of memory can we address with 32-bit addresses? 64-bit addresses?

6.2.3 Suppose that we want to use the same instruction format as TOY, but with 32-bit addresses. What word size would we need? Describe a problem with this design.

6.2.4 Give a single instruction that changes the program counter to memory address 15 regardless of the contents of any registers or memory cells.

Answer: C015 or F015. Both instructions rely on the fact that R[0] is always 0000.

6.2.5 List seven instructions (all having different opcodes) that put 0000 into register A.

Answer: 1A00, 2Axx, 3A0x, 4Axx, 5A0x, 6A0x, 7A00, where x is any hex digit.

6.2.6 List three ways (different opcodes) to set the program counter to 00 without changing the contents of any register or memory cell.

Answer: C000, E0xy, F000.

6.2.7 List five instructions (all having different opcodes) that are no-ops. Exclude cases where the second digit is 0.

Answer: 1xx0, 1x0x, 2xx0, 3xxx, 5xx0, 6xx0, or D0xx, where x is any hex digit other than 0.

6.2.8 List six ways to assign the contents of R[B] into R[A].

Answer: 1AB0, 1A0B, 2AB0, 3ABB, 4A0B, 4AB0, 5AB0, and 6AB0.

6.2.9 There is no branch if nonnegative operator in TOY. Explain how to jump to memory address 15 if R[A] is greater than or equal to 0.

Answer: Use branch if positive and branch if zero, one after the other: CA15 DA15.
6.2.10 Fill in the blanks in this table:

<table>
<thead>
<tr>
<th>binary</th>
<th>hex</th>
<th>TOY instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111111111</td>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td>1111101011001110</td>
<td>FACE</td>
<td></td>
</tr>
<tr>
<td>0101011001000100</td>
<td>5644</td>
<td></td>
</tr>
<tr>
<td>1000000000000001</td>
<td>8001</td>
<td></td>
</tr>
<tr>
<td>0101000001000011</td>
<td>5043</td>
<td></td>
</tr>
<tr>
<td>0001111001010111</td>
<td>1CAB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R[F] = R[F] &amp; R[F]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R[8] = M[88]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (R[C] == 0) PC = CC</td>
</tr>
</tbody>
</table>

6.2.11 There is no absolute value function in TOY. Give a sequence of TOY instructions that sets R[d] to the absolute value of R[s].

6.2.12 There is no bitwise NOR operator in TOY. Give a sequence of three TOY instructions that sets each bit of Rd to 1 if and only if either or both of the corresponding bits in R[s] and R[t] are 0.

6.2.13 There is no bitwise OR operator in TOY. Give a sequence of three TOY instructions that sets each bit of R[d] to 0 if and only if either or both of the corresponding bits in R[s] and R[t] are 1.

Answer: 3DAB 4EAB 1CDE.

6.2.14 There is no bitwise NAND operator in TOY. Give a sequence of three TOY instructions that sets each bit of R[d] to 0 if and only if both of the corresponding bits in R[s] and R[t] are 1.

6.2.15 There is no bitwise NOT operator in TOY. Give a sequence of three TOY instructions that sets each bit of R[d] to the opposite of the corresponding bit value in R[s].

Answer: 7101 2B01 4BAB or 7101 2B01 2BBA.
6.2.16 Show that the subtract operator is redundant. That is, explain how to compute \( Rd = Rs - Rt \) by using a sequence of TOY instructions that do not involve opcode 2.

6.2.17 Which of the 16 TOY instructions do not use all 16 bits?

Answer: halt (only uses first 4 bits), load indirect (does not use third hex digit), store indirect (does not use third hex digit), jump register (does not use two hex digits).

6.2.18 We interpret some TOY integers to be negative according to two's complement notation. What are the only instructions for which this matters?

Answer: The branch positive instruction treats integers between 0001 and 7FFF as positive. The right shift instruction is an arithmetic shift so that if the leftmost bit is 1, then vacated positions are filled with 1s. All other instructions (even subtract!) do not depend on whether TOY has negative integers or not.

6.2.19 Improve the TOY function for computing the GCD given in the text by including a variable c initialized at 0 and modifying the while loop as follows:

- If \( a \) and \( b \) are even, \( \gcd(a, b) = 2 \gcd(a/2, b/2) \) so divide both \( a \) and \( b \) by 2 and increment \( c \) by 1.
- If \( a \) is even and \( b \) is odd, \( \gcd(a, b) = \gcd(a/2, b) \) so divide \( a \) by 2.
- If \( a \) is odd and \( b \) is even, divide \( b \) by 2 (same reasoning).
- Otherwise, both \( a \) and \( b \) are odd, so proceed as before (replacing the larger by their difference, which, by the way, is even).

At the end, shift the result left by \( c \) positions to account for the factors of 2 cast out. Include a client that reads in two integers from standard input and punches their greatest common divisor to standard output. (Analysis beyond the scope of this book shows that the worst-case running time of this algorithm is quadratic in the number of bits in the numbers, much faster than the version given in the text.)
6.3 Machine Language Programming

In this section we continue the process of describing how the Java-language mechanisms that we considered earlier in the book can be implemented in TOY, in the context of TOY programs that accomplish interesting tasks. Our primary goals are to convince you that TOY programming is every bit as interesting and satisfying as Java programming, and that TOY is a much more powerful machine than you might think.

Specifically, we consider TOY programs that implement functions, arrays, standard input and output, and linked structures, the basic building blocks of programming. In principle, these constructs indicate that it is possible to develop machine-language programs corresponding to any Java program that we write. Indeed, we know that to be the case, since the Java compiler does just that.

Of course, there may be resource constraints. Can we really do useful computing with 4096 bits of memory? One of the goals of this section is to convince you that we certainly can. Still, the advance of technology has minimized such constraints. In Section 6.4, we discuss extensions to TOY that make it look a bit more like a modern computer in that respect, without changing the programming model. On such a machine, you certainly could write a TOY program that could do any computation that any Java program can do on your computer.

On a practical level, you will see that it is quite feasible to develop machine-language implementations that accomplish all sorts of tasks. Indeed, many of the first applications programs were implemented in this way because, for many years, the performance penalty of using a high-level language was too much to pay. Actually, most such code was written in assembly language, which is like machine language except that it allowed symbolic names for opcodes, registers, and memory locations (see Exercise 6.4.12). In the 1970s, it was not unusual to see assembly-language programs that stretched to tens of thousands of lines of code. So we are covering a bit of history as well. But even now, people write code of this sort for performance-critical applications.

Most important, our goal is to give you some insight into what your computer is actually doing when running your program.
Functions  After conditionals and loops, the next flow-of-control construct that you learned, in Chapter 2, is the function. The TOY jump instructions are designed for this purpose. Since our primary goal is to demonstrate the concept, we choose one of the simplest of the many possible ways to proceed. To implement a function, we have to:

- Divert flow of control to the function.
- Pass arguments from the client to the function.
- Return a value from the function to the client.
- Return control to the client.

Our choice is to use registers to help accomplish these tasks. For Euclid’s algorithm, we proceed as follows:

- Use TOY’s jump and link instruction to give control to the function. This instruction also saves the return address (the memory address of the next instruction in the client) in a register (we use R[F]).
- Use R[A] and R[B] for arguments and return value.
- Use TOY’s jump register instruction to return control to the client. Specifically, EF00 sets the PC to the saved return address in R[F].

Typical control flow for a function call is shown at right.

For example, with these choices, it is easy to convert the code in Program 6.2.2 to implement a function that computes the GCD of R[A] and R[B], leaving the result in both R[A] and R[B]: change M[29] to the jump register instruction EF00 so that a client can use the jump and link instruction FF22 to call the function. This implementation is shown at the bottom of this page.

```
23  CC2C
24  DC2A
26  C022  {  if (b > a)
28  C022    else
29  EF00          a = a - b
                     }  return [RA = RB = GCD]
```

Implementing a function call

**Typical control flow for a function call**

- **yy**: FFxx — call (R[F] = yy+1, branch to xx)
- **yy+1**: <more client code>
- **yy**: return (branch to address in R[F])
- **xx**: <function code>
- **zz**: <client code>

---

*a function to compute the GCD*
Program 6.3.1 is a client that uses this GCD function to test whether an integer is prime. (Again, this task would be easier if we were to have a division instruction!) The method is simple: a number is prime if and only if the GCD of it and each smaller positive number is 1 (if the number has a factor greater than 1, the GCD of the number and the factor is the factor). As with Program 2.3.1, we can stop the computation after reaching an upper bound on the smallest factor: since we do not have a square root function, we just use 255, since \(255^2\) is larger than any positive 16-bit two’s complement number. (Alternatively, we might easily compute some other upper bound on the square root of the given number.)

The trace below the program shows the values of \(R[9]\), \(R[A]\) and \(R[B]\) before and after the function call and when the halt instruction is reached.

Note carefully that the function uses \(R[C]\), so that the calling program cannot expect \(R[C]\) to have the same value after the function call as before it. And, of course, the function cannot use \(R[9]\) because the calling program is keeping an index there or \(R[F]\) because the return address is kept there. With resources so scarce, “contracts” of this sort among programs play an important role in programming at this level. Such contracts are commonplace.

As with conditionals and loops, we can implement function-call mechanisms in TOY that are not conveniently implemented in Java. For example, we might use several registers as return values. At one extreme, some programmers would save all the registers before calling a function; at another extreme, some programmers would require that each function be responsible for saving the values of all registers before proceeding and then restoring them to their original values upon returning from the function. Modern function-call mechanisms tend towards the latter.

The function-call mechanism used in Program 6.3.1 does not work for recursive functions, but a more general mechanism is easy to develop using a stack (see Exercise 6.3.27). As usual, our intent is not to cover all the details, but just to convince you that the basic constructs that we have in Java are not so difficult to implement on a machine like TOY.

This example illustrates that the benefits of modular programming that we have been discussing for Java also apply to TOY programs. Once our code for computing the GCD or for testing primality has been debugged, we can make use of it in other programs, accessible with a single TOY instruction. This ability made possible the development of layers of abstraction that quickly raised machine-language programming to a workable level and led to the development of many aspects of the software infrastructure that we still use today.
Program 6.3.1  Calling a function: Testing primality

30  7101  R[1] = 1
31  75FF  R[5] = 255
32  7901  R[9] = 1  i = 1
33  2C59  while (i < 255)
34  CC3B  {
35  1991  i = i + 1
36  1A09  R[A] = i  a = i
37  8B3D  R[B] = M[3D]  b = q
38  FF22  a = b = gcd(a, b)
39  2AA1  if (gcd(a, b) != 1) break
3A  CA36  }
3B  9B3E  M[3E] = 1 iff p is prime
3C  0000
3D  005B  q: integer value 9110
3E  0000  result

Started with the PC at 30, this program tests whether the number in 3D is prime. The result is 1 if so and the smallest factor greater than 1 otherwise. It uses the \texttt{gcd()} function derived from Program 6.3.2 that is shown on the previous page, calling it with with the instruction \texttt{FF22}.
Standard output  Of course, one of the very first developments after the computer itself was a better means of communicating with the computer than switches and lights. A variety of different types of devices were widely used. We choose for TOY a means of communication that is stripped to the bare essentials: punched paper tape. As with TOY itself, this method may seem extremely crude to you, but it was widely used for at least a decade.

Punched paper tape was a simple medium that encoded binary numbers in a very visible way. For TOY, we use an encoding that is very similar to the one used on the old PDP-8 computers that we have already discussed, as illustrated at the bottom of the previous page. Each 16-bit binary number is encoded on two successive rows on the tape, where each row can encode eight bits, with a hole punched in positions corresponding to a 1 (and no hole in positions corresponding to a 0). Along the center of the tape is a sequence of regularly spaced small holes that were used to pull the tape through the tape punch via a toothed gear. The tape punch would take a 16-bit binary word from the computer, punches the holes corresponding to that word (in two rows), and advance the tape to get ready to punch the next word. Rather than using lights and switches, a programmer could write a program to punch information on a tape, then look at the tape (or, as we will see, feed it back into the machine later on).
6.3 Machine Language Programming

Program 6.3.2 Standard output: Fibonacci numbers

40  7101  R[1] = 1
41  7A00  R[A] = 0           a = 0
42  7B01  R[B] = 1           b = 1
44  C94B  if (R[9] == 0) PC = 4B  while (i > 0) {
45  9AFF  R[A] to stdout        print(a)
46  1CAB  R[C] = R[A] + R[B]   c = a + b
47  1AB0  R[A] = R[B]          a = b
48  1BC0  R[B] = R[C]          b = c
49  2991  R[9] = R[9] - 1     i = i - 1
4A  C044  PC = 44             }
4B  0000
4C  000C

N: integer value 12_{10}

Started with the PC at 40, this program writes to standard output the first \( N \) nonzero Fibonacci numbers, where \( N \) is the integer value at memory location 4C.
How do we direct our TOY computer to output a word on the tape? The answer to this question is simple: we reserve memory location FF for this purpose, and connect our hardware such that every time a program stores a word in that location, the paper tape punch is activated to punch the contents of that word on the tape.

Program 6.3.2 is an example that computes Fibonacci numbers and punches them out on a paper tape. The computation is straightforward: we maintain the previous two Fibonacci numbers in R[A] and R[B], with R[A] initialized at 0 and R[B] initialized at 1. Then we enter into a loop where we compute the next Fibonacci number by adding R[A] and R[B] with the result in R[C] and then update R[A] and R[B] by copying R[B] to R[A] and R[C] to R[B]. Each time through the loop, we execute the instruction 9AFF, which punches the contents on R[A] on the tape. The resulting output tape is shown below the program. If you compare the tape to the contents of the trace to its right, you can read the Fibonacci numbers in binary on the tape, just as people who programmed machines like TOY once did.

Note that the TOY program makes no explicit reference to paper tape: it just executes 9AFF instructions. This simple abstraction makes it possible to eventually replace the paper tape punch with a different output device, perhaps a teleprinter or a magnetic tape device, without changing the TOY program at all! Such arrangements are the forerunner of the standard output abstraction that we still use today.

In particular, paper tape implements one of the most important characteristics of standard output: there is no intrinsic limit on the length of the tape. This additional capability all of a sudden makes it possible to write programs that can produce an unlimited amount of output. Not only does this ability have useful practical implications (even though TOY is a tiny machine, it can do a lot of computing and produce a lot of output) but also reflects profound implications in the theory of computing, as we have seen in Chapter 5.

Standard input Of course, devices to take input from paper tape came along at the same time as paper tape punches. With a light on one side and sixteen sensors on the other side, two rows of the tape could be quickly read as a binary word with 1 bits corresponding to the punched holes and 0 bits corresponding to positions with no holes. Again, a toothed gear matched with the small holes in the center of the tape would pull the tape into position ready to read the next word.

To read a word from the input tape, you may have already guessed that we again use memory location FF for this purpose, connecting our hardware such
Program 6.3.3  Standard input: Compute the sum

```
50 7800  R[8] = 0                int sum = 0;
51 8CFF  R[C] = stdin          while ((c = read()) != 0)
52 CC55  if (R[C] == 0) PC = 55   {
54  C051  PC = 51                }
55  98FF  stdout = R[8]       print(sum)
56  0000
```

Started with the PC at 50, this program reads a sequence of numbers from standard input, computes their sum, and writes the result on standard output. The program adheres to the convention that a 0000 on the tape marks the end of the sequence.
that every time a program loads a word from that location, the paper tape punch is activated to read 16 bits from the tape and load them into the specified register.

With standard input, data processing with TOY is quite simple, as illustrated by Program 6.3.3. With just seven TOY instructions, we can compute the sum of the numbers on the input tape. The example shown confirms that

\[
1 + 8 + 27 + 64 + 125 + 216 + 343 + 512 = 1296.
\]

This program clearly generalizes to handle all sorts of calculations on the input data, and such computers were heavily used in this way. Indeed, it was not unusual for a machine to be loaded with a small program and then be devoted to an operator simply setting up tapes, running the program, and collecting the output on an output tape all day long. There is no question of the value of being able to efficiently process data. And again, the amount of input data is unlimited, a concept that reflects profound concepts from the theory of computing.

**Arrays** The ability to read a substantial amount of data immediately leads to the need to save the data in memory, to process it. Of course, this leads us to our first data structure in TOY, the array. We use a natural array representation much like the one we first described for Java, storing the array entries in a contiguous sequence of memory words, but we store the length at the end of the array, instead of at the beginning, as shown at right.

We refer to the array by the address of that word. This convention works as well as the convention of putting the length at the beginning: it maintains the essential characteristic that we can compute the address of \( a[i] \) by adding \( i \) to the address of \( a[0] \), and it easy to compute the address of \( a[0] \) from the array address (subtract the length).

One of the primary purposes of the TOY load indirect and store indirect instructions is to support array processing. We maintain the address of \( a[i] \) in a register. Then, to load/store a value in the \( i \)th array word, we use load/store indirect, specifying that register.

Program 6.3.4 illustrates the process of loading the contents of a paper tape into an array. It is packaged as a function that reads the number of words in the array and the address where it is to be stored from the paper tape, then enters a
Program 6.3.4 Array processing: Read an array

```
60  8AFF  R[A] = stdin         a = address of a[0]
61  8BFF  R[B] = stdin         N = read()
62  7101  R[1] = 1
63  7900  R[9] = 0
65  C26B  if (R2 == 0) PC = 6B  {
67  8DFF  R[D] = stdin        i = i + 1
68  BD0C  M[R[C]] = R[D]      }                    a.length = N
6A  C064  PC = 64               address of a[] (TOY standard)
6C  BB0A  M[R[A]] = R[B]      return (array addr in R[A])
6D  EF00
```

Started with the PC at 60, this program reads an array from punched paper tape and stores it in TOY-standard format. Specifically, it reads an address a and an integer N from standard input and reads N integers from standard input, storing them at memory locations M[a], M[a+1], . . . , M[a+N-1]. Then it stores N in M[a+N] (and returns a+N in R[a]).
simple loop that reads each word, maintains an index $i$ to keep track of the next array entry, and stores the result in $a[i]$. After everything has been read and stored, the value of $i$ is the length of the array—the instruction before the halt stores the length at the end of the array. The return value in $R[A]$ is the address of that value, at the end of the array (TOY standard format).

Once data has been loaded into a TOY array, array processing code can refer to the $i$th item in the array precisely as in instructions 66 and 68 in Program 6.3.4: add the index $i$ to the address of $a[0]$ and then use indirection to load or store the item. Another alternative is to maintain a pointer to $a[i]$ and just increment that index to move to the next element, again using indirection to access array elements.

The table on the facing page gives two examples of code that illustrate the utility of passing arrays as arguments to functions. The first example in the table is a typical array-processing program that finds the maximum value in the array whose address is in $R[A]$. It iterates $R[A]$ through the array indices, using load indirect to load each entry and compare it against the largest seen so far, updating that value if necessary. The second is a client that reads an array from punched paper tape, finds the maximum element in the array, and punches the maximum value. Building a set of functions that support array processing of all sorts is not a difficult endeavor.

Actually, one of the motivations for developing media like paper tape in the first place was to carry data produced by devices making experimental measurements, in all sorts of applications. The idea of data processing—storing data on physical media like punched cards or punched paper tape and then processing it with some sort of machine—preceded the development of computers by several decades. Businesses used punched cards to store customer data as early as the 1900s, and a machine was developed to sort punched cards (with some manual intervention) in 1901! Indeed, one of today’s most successful computer companies successfully developed such precursors to computers for half a century before introducing its first computers in the 1950s—the International Business Machines company, which you know as IBM.

It does not take much imagination to realize that the ability to do such calculations, even on computing devices not much more complicated than TOY, would have tremendous practical impact in a world where people were using slide rules and calculators.
6.3 Machine Language Programming

function to find the maximum

\[ R[A] = \text{array address} \quad \text{(TOY standard)} \]

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>7101 R[1] = 1</td>
<td>int i = a.length</td>
</tr>
<tr>
<td>71</td>
<td>A90A R[9] = M[R[A]]</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>140A R[4] = R[A]</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>7800 R[B] = 0</td>
<td>int max = 0</td>
</tr>
<tr>
<td>74</td>
<td>C97C if (R[9] == 0) PC = 7C</td>
<td>while (i &gt; 0)</td>
</tr>
<tr>
<td>75</td>
<td>2991 R[9] = R[9] - 1</td>
<td>{ i = i - 1</td>
</tr>
<tr>
<td>77</td>
<td>AC04 R[C] = M[R4]</td>
<td>d = a[i]</td>
</tr>
<tr>
<td>78</td>
<td>2EBC R[E] = R[B] - R[C]</td>
<td>if (d &gt; max)</td>
</tr>
<tr>
<td>79</td>
<td>DE7B if (R[9] == 0) PC = 7B</td>
<td>max = d</td>
</tr>
<tr>
<td>80</td>
<td>1B0C R[B] = R[C]</td>
<td>}</td>
</tr>
<tr>
<td>81</td>
<td>C074 PC = 74</td>
<td>return (max in R[B])</td>
</tr>
<tr>
<td>82</td>
<td>EF00</td>
<td></td>
</tr>
</tbody>
</table>

read an array and output the maximum (using Program 5.3.5 and the function above).

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5C</td>
<td>FF60 R[F] = 5D; PC = 60</td>
<td>read a[] from stdin</td>
</tr>
<tr>
<td>5D</td>
<td>FF70 R[F] = 5E; PC = 70</td>
<td>R[B] = max(a[])</td>
</tr>
<tr>
<td>5E</td>
<td>9BBF stdout = R[B]</td>
<td>write result</td>
</tr>
<tr>
<td>5F</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

Typical array-processing code
Linked structures Other data structures that we have considered are not difficult to implement in TOY. As an example, we consider binary search trees (BSTs—see Program 4.4.3). For simplicity we consider BSTs built from integer keys, such as the one illustrated at right.

As a sample client, suppose that we need a program to dedup a punched paper tape taken as input: produce a tape with all duplicate values removed. To perform this task, we read a new value, search in the BST to see if it is already there (and we can therefore ignore it) and insert it in the BST and write it on standard output if it has not been seen before. Program 6.3.5 gives an implementation of a TOY function that we can use for this purpose. We will examine the dedup client after we have looked more closely at this function.

To represent a BST node, we use three TOY words: one for the key, one for the left link, and one for the right link, with 0000 representing null links. Typically, the nodes are kept contiguously in memory, though more general arrangements are certainly possible.

To represent a tree, we use a sequence of nodes contiguous in memory, as depicted at left. Each time we need a new node, we add it to the end of the sequence. Accordingly, note that all the links point downwards in this diagram.

The BST search and insert function in Program 6.3.5 takes as parameters the address of the root of the BST in R[A], the search key in R[B], and the address of the next place to put a new node in R[E]. If the tree is empty, it just creates a new node, as described in the next paragraph. If the tree is nonempty, it compares the key to the key at the current node (using a load indirect instruction to load that key) and returns with a nonzero value in R[9] if it is equal (successful search) If not, follows the left link or right link as appropriate (again using a load indirect instruction to load the link) and loops until either finding the key or reaching a 0000 link.
Program 6.3.5  Linked structures: search/insert in a BST

```assembly
7101 R[1] = 1
A90A R[9] = root
180A R[8] = R[A]
C98F if (R[9] == 0) PC = 8F
AC09 R[C] = M[R[9]]
2CBC R[C] = R[B] - R[C]
CC96 if (R[C] == 0) PC = 96
DC8E if (R[C] > 0) PC = 8E
C083 PC = 83
BE08 M[R[8]] = R[E]
BB0E M[R[E]] = key
1EE1 R[E] += 1
900E M[R[E]] = 0
1EE1 R[E] += 1
900E M[RE] = 0
1EE1 R[E] += 1
EF00 PC = RF
```

Called with FF80, this program searches the BST rooted at R[A] for the key given in R[B], allocating memory for new nodes starting at the address in R[E]. The return value in R[9] is nonzero if the search is successful and zero if the search is unsuccessful (and a node was added).
The BST *insert* function is performed once a search has determined that the key is not in the tree so that a new node has to be linked in to the tree at the point where the search terminates. The trick to accomplishing this is to save the address of the last link (in R8 at instructions 82, 88 and 8C) followed. When that link is 0000, we can use the instruction BE08 to replace the null link where the search terminated with a link to the new node. To create a new node we store the new key and two 0000 links and update RE as necessary (this code is in instructions 90–95) and then return with R[9] = 0.

Program 6.3.5 is essentially a full symbol-table implementation, which is likely to be fast and efficient in all sorts of practical situations. In many ways, this implementation is simpler and easier to understand than the Java version. At the very least, it is a very helpful way to understand the essential nature of linked structures.

The function in Program 6.3.5 enables implementation of our punched paper tape dedup client in less than ten TOY instructions, as illustrated below. When you first read about TOY at the beginning of this chapter, you may not have imagined that it might have the power to perform useful calculations of this sort. And, of course, this is/was only the beginning!

```
A0 7AB0 R[A] = B0  
A1 7EB1 R[E] = B1  
A2 8BFF RB = stdin 
A3 CBA8 if (RB == 0) PC = A8  
A4 FF80           \  \  x = searchinsert(b);  
A5 C9A2           \  |  if (x == 0) continue;  
A6 9BFF stdout = RB |  |  print(b);  
A7 C0A2 PC = A2  
A8 0000
```

*dedup a punched paper tape*

*Typical symbol-table client*
Why learn machine language programming? Revisiting this question is worthwhile now that you have a better idea of what is involved, and before you take on the challenge of actually composing some machine-language programs yourself. There are three primary reasons to do so:

- Machine-language programs are still often preferred in applications.
- Viewing a computation at this lowest level can expose its essence.
- You can better appreciate programs such as compilers that create machine-language programs for your computer.

We will briefly expand upon each of these reasons here, summarizing the discussion sprinkled throughout this chapter.

First, machine-language programs are still often preferred in performance-critical situations. Scientists, engineers, and applications programmers continue to push the boundaries of what is possible, and it is very often the case that a low-level implementation of a critical part of a computation will run one or two orders of magnitude faster than one written in a high-level language. You should at least be aware of this option.

Second, machine language programming often does seem to capture the essential aspects of a computation, as it can strip away system and machine dependencies to expose what is really going on. Data structures, in particular, are often much more transparent in machine-language programs than in higher-level languages. The BST example that we have just considered is a fine example of this phenomenon, and we explore others in the exercises.

Third, once you know a machine language, you can think about the idea of writing programs in high-level languages like Java that can produce programs in that language. You depend on such programs whenever you use your computer, as everything that runs on your computer ultimately has been reduced to machine language. We explore this aspect of programming in more detail in the next section. Writing programs that produce programs is a satisfying and worthwhile experience that everyone should consider doing.

In the context of this book, our purpose is to demystify what goes on inside your computer by describing the nature of the interface between its hardware and its software. Our hope is that you will view learning to program in TOY as an opportunity to be in a position to appreciate these important ideas without having to cope with more complex real-world computers. It is a placeholder for the concept of programming in machine language that we can use to expose a number of funda-
mental ideas. Alternatively, you might view learning TOY as preparation for coping with real-world computers.

Either way, the significance of machine language programming in this book, and, indeed, in the development of our computational infrastructure over the past several decades, is that it provides an intermediate level of abstraction between your Java programs and your computer. First, you are now in a position to better understand the relationship between the Java programs that you write and the machine-language programs that your computer actually executes. The next section is devoted to exploring this relationship in more detail. Second, you are now in a position to better understand the possibility of creating a physical artifact that can execute machine language programs. In Chapter 7, we complete our demystification effort by examining what is involved to design and build a circuit that can execute machine-language programs. Again, familiarity with a low-level machine like TOY plays a critical role in your ability to understand how such a circuit operates.
Q. Do I really need to practice writing TOY programs?

A. Well, you probably do not need to refine your skills to the extent you have done in Java. But you might think of TOY as another programming language. Certainly, knowing Java makes it easier for you to learn TOY, and you can see that each new language that you learn makes it easier to learn the next one.

Q. Where can I find more information on TOY?

A. There is a substantial amount of information on the booksite, developed by students, teaching assistants and faculty over the two decades we have been developing this book. In particular, you can find an interactive TOY simulator and practice entering programs with switches and lights yourself, if you are interested. (We use that for teaching demos.)

Q. Any other sources of information about TOY?

A. No, not at all. It is an imaginary machine.

Q. So maybe I should learn some other machine language?

A. Sure, you can learn about programming on a real machine. As mentioned in the text, knowing TOY should be good preparation for that. You might learn about the widely-used IA-32 architecture, but be warned that the complete software developer’s manual is thousands of pages long!

A. Or, you can learn MIX, another imaginary machine language developed by D. E. Knuth for his classic series of books The Art of Computer Programming. You can read his reasons for choosing a machine language in the preface to those books; one of them is the following: “A person who is more than casually interested in computers should be well schooled in machine language, since it is a fundamental part of a computer.” And if you learn MIX, you can read about numerous algorithms in Knuth’s books that are expressed only in MIX.
Important note. These exercises are much, much easier to complete if you have access to a TOY machine to run and debug programs, so you are advised to read Section 6.4 before attempting them.

6.3.1 Write a program sort3.toy that reads in three integers from standard input and punches them out to standard output in ascending order.

6.3.2 Write a program powers2.toy that punches to standard output all of the positive powers of 2 that can be represented in a two’s complement TOY word.

6.3.3 Write a program sum_1-n.toy that reads in an integer \(N\) from standard input and punches out the sum \(1 + 2 + 3 + \ldots + N\).

6.3.4 Given an integer \(x\), the next integer in its Collatz sequence is defined by replacing it with \(x/2\) if \(x\) is even, and \(3x + 1\) if \(x\) is odd, and repeating until \(x\) is 1 (see Exercise 2.3.29). Write a program collatz.toy that reads an integer from standard input and prints its Collatz sequence to standard output. \textit{Hint}: use right shift to perform integer division by two.

6.3.5 Draw a diagram like the ones in the text for if and while that shows how to implement a for loop in TOY.

6.3.6 Write a program chop.toy that reads in an integer \(N\) from standard input and punches out powers of 2 that sum to \(N\). For example if \(N\) is 012A, then the program should punch

\[
\begin{align*}
0002 \\
0008 \\
0020 \\
0100
\end{align*}
\]

on the output tape since \(012A = 0002 + 0008 + 0020 + 0100\).

6.3.7 Write a program that reads in an integer from standard input, cubes it, and punches the result. For multiplication, use FF90 to call the multiplication function given in Exercise 6.3.38.

6.3.8 Write a TOY code fragment that swaps the contents of R[A] and R[B], without writing to main memory or any other registers. \textit{Hint}: use the XOR instruction.
6.3.9 This question tests the difference between *load address*, *load*, and *load indirect*. For each of the following TOY programs, give the contents of R[1], R[2], and R[3] upon termination.

(a)  
10: 7211  
11: 7110  
12: 2321  
13: 0000  

(b)  
10: 8211  
11: 8110  
12: 2312  
13: 0000  

(c)  
10: 7211  
11: A102  
12: 2312  
13: 0000  

6.3.10 Consider the following TOY program. What is the value of R[3] when it halts?

10: 7101  
11: 7207  
12: 7301  
13: 1333  
14: 2221  
15: D213  
16: 0000  

6.3.11 For each of the following boolean expressions, give a TOY code fragment that reads in an integer *a* from standard input and writes 0001 to standard output if the condition is true and 0000 if it is false.

```java
a = 3
a > 3
a < 3
a != 3
a >= 3
a <= 3
```

6.3.12 Suppose that you load the following into locations 10–17 of TOY, set the PC to 10, and press RUN.
What, if anything, is printed to standard output if standard input is 1112 1112?

**Hint:** The first value is stored into M[15] where it is eventually executed as code.

**6.3.13** Repeat the previous question, but now with the following data on standard input: C011 C011 1112 1112.

**6.3.14** Write a TOY function that takes $a$, $b$, and $c$ as arguments in R[A], R[B], and R[C] and computes the discriminant $d = b^2 - 4ac$, returning the result in R[D]. Put your code in locations 10–... and use FF90 to call the multiplication function given in Exercise 5.3.38.

**6.3.15** List all input values between 0123 and 3210 for which the following program writes 0000 to standard output before halting.

```assembly
10: 8AFF   R[A] = stdin
11: 7101   R[1] = 1
12: 2BA1   R[B] = R[A] - 1
14: 9CFF   stdout = R[C]
15: 0000   halt
```

**Answer:** 0200 0400 0800 1000 2000. It returns 1 for all inputs that have at most one 1 in their binary representation, i.e, the hexadecimal integers 0000, 0001, 0002, 0004, 0008, 0010, ..., 8000.

**6.3.16** Suppose that you load the following program into locations 10-20 of TOY,
10: 7101
11: 7A30
12: 7B08
13: 130B
14: C320
15: 1400
16: 2543
17: C51E
18: 16A4
19: A706
1A: 1717
1B: B706
1C: 1414
1D: C016
1E: 6331
1F: C014
20: 0000

Now suppose that you load 0001 0002 0003 0004 0004 0003 0002 0001 into memory locations 30 through 37, set the PC to 10, and press RUN. What will be the contents of memory locations 30 through 37 when the program halts?

6.3.17 Translate the TOY program in the previous exercise into Java code by filling in the ????

for (int i = N; i > ???; i = i ???)
    for (int j = 0; j < ???; j = j ???)
        a[ ??? ] = ??? ;

6.3.18 Compose a program that computes the dot product of two vectors stored in TOY arrays. Package it as as a function that takes the array addresses in RA and RB and returns the dot product in RE. Use FF90 to call the multiplication function given in Exercise 6.3.37. **Note:** you need a more general function-call convention than we have considered so far: save and restore the return address R[F] so that the same register can be used for the return address when calling the multiply function. See Exercise 6.3.27.
6.3.19 Suppose that you load the following into locations 10–1B of TOY and that standard input has the values 1CAB EF00 0000 4321 1234. When you set the PC to 10 and press RUN, what value is punched on standard output?

10: 7101 R[1] = 1
12: 8AFF R[A] = stdin
13: CA17 if (R[A] == 0) PC = 17
14: BA02 M[R[2]] = R[A]
15: 1221 R[2]++
16: C012 PC = 12
17: 8AFF R[A] = stdin
18: 8BFF R[B] = stdin
19: FF30 see previous exercise
1A: 9CFF stdout = R[C]
1B: 0000 halt

6.3.20 Answer the previous exercise for the case when standard input has the values 2CAB EF00 0000 4321 1234.

6.3.21 Consider the following code for traversing a linked list.

10: 7101
11: 72D0
12: 1421
13: A302
14: 93FF
15: A204
16: D212
17: 0000

Each node is two consecutive words in memory, a value followed by a link (address of the next node), with the link value 0000 marking the end of this list. Suppose that memory locations D0–DB contain the values

0001 00D6 0000 0000 0004 0000 0002 00DA 0000 0000 0003 00D4

Give the values punched if by this program (set the PC to 10 and press RUN).

Answer: 1 2 3 4 5 6 7.
6.3.22 Specify how to change one word of memory in the previous exercise so that it prints out 1 2 6 7 instead of 1 2 3 4 5 6 7 (linked list deletion).

6.3.23 Specify how to change three words of memory (overwriting one, and using two more) so that it prints out 1 2 3 4 8 5 6 7 (linked list insertion).

6.3.24 Suppose that the TOY memory contains the following values and that you set the program counter to 30 and hit RUN. What, if anything, is printed to standard output? List the contents of R[2] and R[3] when the machine halts.

```
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>7101</td>
<td>7101</td>
<td>0003</td>
<td>0002</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>7200</td>
<td>7200</td>
<td>0000</td>
<td>0050</td>
</tr>
<tr>
<td>2</td>
<td>0000</td>
<td>8329</td>
<td>8329</td>
<td>0005</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>0000</td>
<td>1221</td>
<td>A403</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>0000</td>
<td>1331</td>
<td>1224</td>
<td>0004</td>
<td>0000</td>
</tr>
<tr>
<td>5</td>
<td>0000</td>
<td>A303</td>
<td>1331</td>
<td>0052</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>0000</td>
<td>D333</td>
<td>A303</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>7</td>
<td>0000</td>
<td>92FF</td>
<td>D343</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>8</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td>9</td>
<td>005A</td>
<td>0000</td>
<td>0000</td>
<td>0060</td>
<td>0000</td>
</tr>
<tr>
<td>A</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>B</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0058</td>
<td>0000</td>
</tr>
<tr>
<td>C</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>D</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>E</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>F</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
```

6.3.25 Develop a pair of TOY functions that implement a pushdown stack, passing the values to push and pop in R[B] and the address of the stack in R[A].

6.3.26 Write a TOY function that traverses a BST and punches out the keys in sorted order. Hint: Use a pushdown stack.
6.3.27 Use your solution from Exercise 6.3.25 to develop two TOY functions: one that saves \( R[A] \) through \( R[F] \) on a stack and another that restores them from the stack. Use the functions to develop a recursive program that punches the ruler function on the output tape.

6.3.28 Implement a version of our BST function (Program 6.3.5) that saves space by packing the two links for each node into a single word.
6.3.29 32-bit integers. Write a TOY function that, considering R[A] and R[B] to be a 32-bit two’s complement integer and R[C] and R[D] to be a second 32-bit two’s complement integer, adds the two and leaves the result in R[A] and R[B].

6.3.30 Gray codes. Write a TOY program graycode.toy that reads in an integer \( n \) (between 1 and 15) from standard input and then prints out \( (i \gg 1) \land i \) to standard output for \( i \) decreasing from \( 2^n - 1 \) through 0. The resulting sequence is called a Gray code of order \( n \). See the discussion associated with Exercise 2.3.3.

6.3.31 Dot product. Compute the dot product of two arrays, which start at locations RA and RB and have length RC.

6.3.32 Axpy. Write a TOY function that takes a scalar \( a \) in R[A], a vector \( b \) stored in a TOY array whose address is in R[B], and another vector \( c \) stored in a TOY array whose address is in R[C], computes the vector \( ab + c \), leaving the result in a TOY array and the address of that array in R[D].

6.3.33 One-time pad. Implement a one-time pad in TOY to encrypt and decrypt 256 bit messages. Assume that the key is stored in memory location 30–3F and that the input consists of sixteen 16-bit integers.

6.3.34 Find the singleton number. Suppose that a sequence of \( 2N + 1 \) 16-bit integers appears on standard input such that \( N \) integers appear exactly twice and one integer appears only once. Write a TOY program to punch out the singelton integer. Hint: XOR all of the integers together.

6.3.35 Efficient multiplication. Implement the algorithm that you learned in grade school to multiply two integers. Specifically, let \( b_i \) denote the \( i \)th bit of \( b \), so that

\[
b = (b_{15} \times 2^{15}) + (b_{14} \times 2^{14}) + \ldots + (b_1 \times 2^1) + (b_0 \times 2^0)
\]

Now, to compute \( a \times b \), use the distributive law:

\[
a \times b = (a \times b_{15} \times 2^{15}) + (a \times b_{14} \times 2^{14}) + \ldots + (a \times b_1 \times 2^1) + (a \times b_0 \times 2^0)
\]

Naively, this appears to reduce the problem of performing one multiplication to 32 multiplication, two for each of the 16 terms. Fortunately, each of these 32 multipli-
cations are of a very special type, because \(a \times 2^i\) is the same as left shifting \(a\) by \(i\) bits. Since \(b_i\) is either 0 or 1 the \(i\)th term is either \(a \ll i\) or 0.

**Answer:**

```
90: 7101   R[1] = 1
91: 7C00   R[C] = 0       c = 0;
92: 7216   R[2] = 16      for (i = 16; i > 0; i--)
97: C41B   if (R[4] == 0) PC = 99   if b[i] == 1
99: D293   if (R[2] == 0) PC = 93   } return
```

6.3.36 *Power function.* Using the multiplication function of the previous exercise, implement a TOY function that computes \(a^b\), taking the values of \(a\) and \(b\) as arguments in \(R[A]\) and \(R[B]\).

6.3.37 *Polynomial evaluation.* Using the power and multiplication functions of the previous two exercises, implement a TOY function that takes the address of a TOY array containing coefficients \(a_0, a_1, a_2, \ldots, a_n\) and an integer \(x\) as arguments in \(R[A]\) and \(R[B]\) and evaluates the polynomial

\[ p(x) = a_n x^n + \ldots + a_2 x^2 + a_1 x^1 + a_0 x^0 \]

returning the value in \(R[C]\). Polynomial evaluation was one *raison d’etre* for early computers (for preparing ballistics tables).

6.3.38 *Horner’s method* is a clever alternative to directly evaluating a polynomial that is more efficient and easier to code. The basic idea is to judiciously sequence the way in which terms are multiplied, as in the following example:

\[ p_4 x^4 + p_3 x^3 + p_2 x^2 + p_1 x^1 + p_0 x^0 = (((((p_4)x + p_3)x + p_2)x + p_1)x + p_0) \]

Implement a TOY function based on this idea that uses only \(n\) multiplications to evaluate polynomial of degree \(n\). This method was published in 19th century by
British mathematician W. G. Horner, but the algorithm was used by Isaac Newton over a century earlier (see Exercise 2.1.32).

**6.3.39 Number conversion.** Implement a TOY program that uses Horner’s method (see Program 6.1.1) to convert a decimal integer to binary representation. Read the decimal integers as hex numbers of the form 000x from standard input and punch the binary result on standard output, one bit at a time.
6.4 TOY Virtual Machine

Given that TOY is an imaginary machine, how are we able to run and debug programs? In this section we provide a complete answer to that question, and then discuss implications. In short, our answer is that we can easily write a Java program known as a virtual machine that can run any TOY program. Indeed, as you will see, the virtual machine is the definition of TOY. It precisely describes the effect of every possible TOY instruction, always maintaining complete information about the state of the TOY machine.

A virtual machine is a definition of a machine that executes programs like a real machine, but need not have direct correspondence to any physical hardware. We use the term broadly to refer both to the definition of the machine and to any software (or hardware) that implements it.

The idea of a virtual machine is a bit discomforting at first. Our purpose in providing full details for a specific case is to make sure that you completely understand the idea because it is of great practical importance and also sits at the heart of the fundamental ideas in the theory of computing that we discussed in Chapter 5. Indeed, the Church-Turing thesis implies that any operating environment can perform the same computation as any other, given enough memory, so every computer can be a virtual machine on every other computer.

The core concept is the idea of programs that process programs. You may recall in our proof of the halting problem that the idea that a program might take another program as input seemed a bit strange at first. Actually, the idea is a fundamental computer science concept, which we explore in detail in this section.

As a warmup, we consider some simple practical applications of the concept for TOY programming, including a significant and unavoidable pitfall. These provide the context that we need to relate these ideas to Java and then to consider the TOY virtual machine, the centerpiece of this section. Then we consider implications for modern and future computing.

This section covers a lot of ground, from TOY to implications of the halting problem to server farms and cloud computing. That we are able to do so in a few dozen pages is testimony to the power, elegance, and lasting importance of the concepts that underly these topics. Our treatment is necessarily brief, but an understanding of the issues that arise is important for anyone who is engaged with
computation. Indeed, one prime purpose of our entire presentation of TOY is to put you in a position to be able to understand these issues.

**Booting and dumping** We begin by examining the essential characteristic of any von Neumann machine like TOY—it can process data of any type, not just numbers. Indeed *one program's instructions can be another program's data*. Specifically, we examine two practical consequences of this property in the context of TOY programming.

**Booting.** Consider the punched paper tape shown at right. Does it look familiar to you? By this time, you might recognize some of these 4-digit hex numbers as TOY instructions. Actually we have seen them before—they are the sequence of instructions for Program 6.3.3 (which computes the sum of the numbers on the input tape), preceded by the address of the first instruction and the count of the number of instructions. But, as we have emphasized since the very beginning of this chapter, *the meaning of a given sequence of bits within a computer depends on the context*. Thus, a program could also interpret these bits as two's complement numbers. Indeed, this tape could also be the input to Program 6.3.3, to compute the sum

\[ 50_{16} + 7_{16} + 7800_{16} + (-7301_{16}) + (-33AB_{16}) + 18CC_{16} + (-3FAF_{16}) + (-6301_{16}). \]

Of much more significance is the idea that this tape could also be the input to our array-reading program Program 6.3.4—we consider the tape to be a program, and we view Program 6.3.4 as a program that loads other programs into memory. Indeed, we could have tapes for each of the TOY programs that we have considered, and load each of them into memory in this way.

Or, we could have a single tape for all of the memory locations 10–FF and fill the whole memory with our code and data. This process is called *booting* the computer, terminology that persists to this day. We use some special process (in TOY’s case, the switches) to load a small program into memory that can load up the rest of the memory from an external device.

At left in the table at the bottom of the next page is code taken from Program 6.3.4, but retooled from being a function to being *boot code*, to be entered via the switches by a programmer after turning on the computer at the beginning of the work day. A typical convention was to reserve the first few words of memory (locations 00–0F, in our case) for the boot program. A computer like TOY would
have a program like this taped on the front panel. Because that would be the only program that needed to be entered via the switches, programmers would pride themselves on the speed with which they could operate the switches to get it loaded, fingers flying like those of a concert pianist (well, not exactly).

Beyond not being packaged as a function, the boot program differs from Program 6.3.4 in two ways. The first difference is that there is no need to store the length (that was a convention for arrays). The second (and much more profound) difference is that the boot program ends with the statement EA00. Since $R[A]$ contains the address of the first word loaded, this instruction passes control to the data just loaded, a quintessential example of data changing roles to an instruction in a von Neumann machine.

Modern computers use the same basic process, so the terminology lingers to this day. When you *reboot* your phone or tablet or your computer, your operating system and many basic applications are loaded into the device from storage external to the processor by a small program that is loaded into the processor memory by some special process. Then a branch instruction gives that program control.

**Dumping.** One easy way to write a TOY program to punch a paper tape for loading by the boot program is to modify the boot program to punch the address and the length (instead of reading them) and to punch each memory word (instead of reading it) within the loop. These changes are highlighted in blue in the dump program shown at right below. The first two instructions specify the address and the length—the programmer can set them (via the switches) to any value whatsoever.

```plaintext
02 8AFF R[A] = stdin
03 8BFF R[B] = stdin
04 7101 R[1] = 1
05 7900 R[9] = 1
07 C20D if R[2]==0 PC = 0D
08 1CA9 R[C] = R[A] + R[9] { address of a[i]
09 8DFF R[D] = stdin
0A BD0C M[R[C]] = R[D]
0C C006 PC = 06
0D EA00 PC = R[A]
```

```
read/write
read/write address of a[i]
b = length of a[i]
```

```plaintext
00 7A10 R[A] = 10
01 7BEF R[B] = 239,10
02 9AFF stdout = R[A]
03 9BFF stdout = R[B]
04 7101 R[1] = 1
05 7900 R[9] = 1
07 C20D if R[2]==0 PC = 0D
08 1CA9 R[C] = R[A] + R[9]
09 AD0C R[D] = M[R[C]]
0A 9DFF stdout = R[D]
0C C006 PC = 06
0D 0000
```

*boot (see Program 5.3.5)*

*dump*

**Booting and dumping**
ever. The values 10 and EF specify a “full dump”: we only dump the words from 10 through FE to reserve 00–0F for the dump/boot programs themselves and to take into account the fact that FF is reserved for standard input and standard output. This process is called dumping the contents of memory.

These two simple processes vastly simplified the programmer’s workflow for computers like TOY. A programming session would start by entering the boot program with the switches, and then running it to load up various programs from paper tape. If the day’s work involved entering new code (via the switches), then a few changes could be made (locations 00–03 and 09–0A, in our case) to convert the boot to a dump, and a paper tape could be punched to save the new code for use on another day.

For example, after entering and debugging our program to compute the Fibonacci numbers (Program 6.3.2), we could save that work by noting that it comprises 0D words to be loaded in locations 40 through 4C, then using the switches to change the instruction at 00 to 7A40 and the instruction at 01 to 7B0D, then setting the address switches to 00 and pressing RUN to run the dump program and produce the tape at right. At any later time, we could boot from that tape to load that program.

You can imagine that programmers would quickly develop a collections of punched paper tapes containing their code. You might think of such a collection as a very early manifestation of external storage and our boot program as a very early manifestation of the installer that you regularly use to put programs on your mobile device.

A note of caution  Next, we consider an example that illustrates that the von Neumann architecture, while marvelous in many ways, can also be dangerous.

A typical workflow on machines like TOY might have been for a scientist to develop a program for processing experimental data, then use that program over a period of weeks or months to actually process the data. Since this activity would just involve loading the program (either by entering the boot program via the switches or by just checking that it has been previously loaded and then using it) and then loading the data on the paper tape reader for the program to process, it was typical to hire a machine operator (who need not have programming or scientific skills)
to do these jobs. The machine operator would load the machine with a scientist’s
program, then run it on different sets of experimental data, perhaps for hours.

Now imagine that someone is regularly using a large program starts by load-
ing an array, using Program 6.3.4 (which resides in memory
locations 60 through 6D), and one day a colleague asks to use
the program and provides the operator with the tape shown at
right. The operator puts the tape on the reader presses RUN, and
(perhaps) goes to lunch. The tape says to load a 21-word array
starting at 50. What happens?

Analyzing the situation is not difficult if we adopt the
point of view of the computer. Whatever the instruction at the
address specified by the PC, we know that the machine fetches
that instruction, increments the PC, executes the instruction,
and continues in that cycle until encountering a halt. In this
case, the boot program loads the 16 words on the tape that fol-
low the address and length as expected, but when R9 is 5F and
then is incremented, we need to take a closer look.

When we increment 5F, we get the result 60, so the next
word of data is stored in 60, then the next one in 61, and so
forth, as illustrated in the table at the top of the next page. This
process leads to results that are perhaps unexpected, because the
array reading program is overwriting itself. Eventually it comes
time to execute an instruction that was once data on the tape.
In this case, the instruction branches back to a previous instruc-
tion (which was also once data on the tape) with the result that
the machine goes into an infinite loop punching 8888 on the
tape, as fast as it can. That operator would return from lunch to
find an unexpected situation, to say the least.

This fictional story is intended to reduce to a simple form
the idea of taking control of a computer in an unexpected way.
That three-line loop to punch 8888 might have been any pro-
gram at all. And it need not have happened just with the boot
program: you can likely arrange to take control in a similar
manner from our array input program or any other program
that reads and stores data.
The situation is similar to the very real viruses that have plagued computer users for many decades. In a great many situations, computer systems have been easily tricked into transferring control to an area of memory that was supposed to be data, with all sorts of dire consequences. Just to pick one example: typical programs written in the C programming language are subject to a buffer overflow attack, where a user provides a string argument that is longer than expected to a function. Since the function code appears just after the buffer that is supposed to hold the string in the memory, a malicious user can encode a program in a string that is longer than expected, just as in our example. The system transfers control to the memory location where the function is supposed to be, but that gives control to the bad actor. In the case of a virus, that code contacts and infects other computers, and the situation quickly escalates. Documented cases of this sort have plagued millions of computer users, and continue to do so.

Couldn’t we write a program to check for this? Doesn’t virus protection software help? Unfortunately, these sorts of program just scan for known viruses and do not try to figure out what a given sequence of instructions might do. Indeed, in general, it is a consequence of the undecidability of the halting problem (see Section 5.4) that it is not possible to write a program that can check whether or not any given program is a virus.
Programs that process programs  On a much more positive note, there are many situations where it is extremely helpful to write programs that take other programs as input (or output). We have informally discussed such programs since talking about the Java compiler and the Java Virtual Machine in Section 1.1, but in the context of TOY we can provide a bit more detail.

Assembler. Programming directly with hex numbers is inconvenient and error-prone, so one of the first developments for many computers was assembly language, which allowed the use of symbolic names for operations and machine addresses. An assembler is a program that takes an assembly language program as input and produces a machine language program as output. Writing an assembler for TOY in Java is not difficult (see Exercise 6.4.12). Doing the job in TOY code is a bit more challenging, but early programmers met such challenges for all sorts of computers. Assembly language programming is widespread to this day.

Interpreter. An interpreter is a program that directly executes instructions written in a programming language. We have already seen a simple example of an interpreter: Program 4.3.5, which evaluates arithmetic expressions. An arithmetic expression specifies a computation in a simple programming language, and Program 4.3.5 performs that computation. This computation is sufficiently simple that you could imagine implementing it in TOY (see Exercise 6.4.14). On a much larger scale, many modern programming languages are intended to be processed with an interpreter. A primary reason to use a system based on interpretation is that it can be interactive—you can type in instructions one at a time. A primary reason not to use such a system is that it can be inefficient, as each source language instruction has to be parsed and processed each time it is encountered.

Compiler. A compiler is a program that transforms source code in one computer language into another computer language (often machine language), typically to create an executable program. To better understand this concept, you are encouraged to work Exercise 6.4.13, where you are asked to convert our arithmetic expression evaluator into a compiler. For example, where the interpreter performs an addition when encountering a + sign, the compiler will output a machine instruction that performs an addition. After the whole source program has been processed, the result is a machine-code.
language program. Most industrial-strength programming systems are based on compilation because modern compilers can produce machine-language programs that are as efficient as hand-coded solutions (or even more so).

**Virtual machine.** A virtual machine is a definition of a machine that executes programs like a real machine, but need not have direct correspondence to any physical hardware. For sure, TOY is a virtual machine! Historically, the term has evolved significantly. So as not to have to rewrite existing software, every new computer design includes software or hardware called an emulator that can run programs written for its predecessor. One of the earliest uses of virtual machines was *timesharing*, software that gives the illusion of multiple copies of a computer, all running on a single computer. Another early use, which persists to this day, was to define an intermediate level of abstraction between high-level languages and machine hardware (this is Java's approach, which we examine next). In modern computing, it makes sense to use the term *virtual machine* to encompass all of these.

**JVM** The Java virtual machine (JVM) is a prototypical example. Rather than having to develop a Java compiler for every kind of processor, the designers of the Java system knew that it would be much better to define a virtual machine with many of the characteristics of real machines (registers, memory, a program counter, instructions that perform arithmetic and logical operations, transfer information between registers and memory and implement branches and jumps), but with an instruction set called a bytecode that is designed for efficient execution by an interpreter. Then they could put all their effort into developing a compiler from Java to the JVM. The process is depicted in the diagram at the bottom of this page. To make Java work on any particular computer, it is enough to write an interpreter for the
JVM, a much easier task than developing a new compiler for Java. Even though Java was developed decades ago, it is successfully used on new machines even today. As further evidence of the utility of the idea, new languages have been developed that compile to the Java virtual machine. These languages then work on any device that can run Java programs.

All of these types of programs are fascinating, and you will likely encounter them in a variety of forms as you become more engaged with computation. The only difference between a program and data is the context. In TOY, if the PC has the address of a memory word, it is an instruction; otherwise it is data. This essential characteristic of von Neumann machines is not just a trick; it is a critical aspect of modern computing. Turing conceived of the idea, von Neumann grasped its importance in practice, and the world has reaped the benefits ever since.

In the present context, our overriding interest is in the relationship between Java and TOY, so we address that next.

**TOY in Java** The Java program on the facing page deserves careful study. In a very real sense, this program is the TOY machine, as we used it to implement and debug all of the TOY programs in this book (and we encourage you to use it yourself to implement and debug some TOY programs). Our intent is for you to be surprised at how easy it is to understand this program. Indeed, one of the primary design considerations for TOY itself was that this program had to fit on a single page. This program is complete except for the code for standard input and standard output, which is omitted for the moment to allow us to focus on the core machine.

**Parsing a TOY instruction.** Suppose that we have a TOY instruction in an int variable IR. It is a 32-bit value, but TOY instructions are just 16 bits, so only the rightmost 16 bits are relevant. With the shifting and masking operations that we considered in Section 6.1, we can isolate the opcode, registers, and address for later use:

```java
int op   = (IR >> 12) & 0xF;
int d    = (IR >>  8) & 0xF;
int s    = (IR >>  4) & 0xF;
int t    = (IR >>  0) & 0xF;
int addr = (IR >>  0) & 0xFF;
```

For any particular instruction, we use s and t or addr, but not both, but it is easiest to just compute all the values for every instruction.
public class TOY {
    private int[] R = new int[16];
    private int[] M = new int[256];
    private int PC;
    public TOY(String filename) // Constructor; see text.
    public void run()
    {
        while (true)
        {
            int IR = M[PC++]; // Fetch and increment.
            int op = (IR >> 12) & 0xF;
            if (op == 0) break;
            int d = (IR >> 8) & 0xF;
            int s = (IR >> 4) & 0xF;
            int t = (IR >> 0) & 0xF;
            int addr = (IR >> 0) & 0xFF;
            switch (op)
            {
                case 1: R[d] = R[s] + R[t]; break;
                case 2: R[d] = R[s] - R[t]; break;
                case 3: R[d] = R[s] & R[t]; break;
                case 4: R[d] = R[s] ^ R[t]; break;
                case 5: R[d] = R[s] << R[t]; break;
                case 6: R[d] = (short) R[s] >>> R[t]; break;
                case 7: R[d] = addr; break;
                case 8: R[d] = M[addr]; break;
                case 9: M[addr] = R[d]; break;
                case 10: R[d] = M[R[t]]; break;
                case 11: M[R[t]] = R[d]; break;
                case 12: if ((short) R[d] == 0) PC = addr; break;
                case 13: if ((short) R[d] > 0) PC = addr; break;
                case 14: PC = R[d] & 0xFF; break;
                case 15: R[d] = PC; PC = addr; break;
            }
            R[d] = R[d] & 0xFFFF;
            R[0] = 0;
        }
    }
    public static void main(String[] args)
    { (new TOY(args[0])).run(); }
}
The state of the machine. We noted at the outset that the behavior of the TOY machine is completely determined by the contents of the registers (in particular, the PC) and the contents of the memory. This fact leads naturally to our choice of instance variables in PROGRAM 6.4.1: we use an array of 16 int values for the registers, an array of 256 int values for the memory, and a single int value for the PC. Again, we actually use only the rightmost 16 bits of those values (see the Q&A at the end of this section).

Importantly, when we take standard input into account, the size of the machine state is unlimited, even though the machine itself is a finite state machine. There are just 4,160 bits in the memory and the registers, but the number of bits on the input tape is unlimited.

Booting the machine. To slightly streamline our workflow when running the TOY simulator, we boot the machine within the constructor, shown in the code at the bottom of this page. The client provides a file name as argument—the file is to contain a starting address, a word count, and a sequence of words. The example at left illustrates fib.toy, the 13-word program that we presented as PROGRAM 6.3.3, which is to be loaded at M[40] and run by setting the PC to 40. That is, we store each TOY program in a file, then boot from that file just by providing the file name in the constructor. The constructor loads the TOY memory by reading the specified number of words, storing them in sequence starting at the given location. Then it sets the PC value to the address of the first instruction, ready for the program to run.

This special boot process actually mirrors the approach that eventually emerged for real computers, where machines are booted through some special process that is different from the input devices used by programs. The precise mechanism is inconsequential: our primary interest is in what

```
% more fib.toy
40
0D
7101
7A00
7B01
894C
C94B
9AFF
1CAB
1AB0
1BC0
2991
C044
0000
000C
```

```
public TOY(String filename)
{
    In in = new In(filename);
    PC = Integer.parseInt(in.readString(), 16) & 0xFF;
    int len = Integer.parseInt(in.readString(), 16) & 0xFF;
    for (int i = 0; i < len; i++)
        M[PC + i] = Integer.parseInt(in.readString(), 16) & 0xFFFF;
}
```

Constructor for TOY virtual machine
happens when the memory has been loaded and the PC has been initialized with the specified address by the constructor: we call the run() method. This action simulates the action of an operator pressing the RUN button after having entered or booted the program and having set the switches to the start address.

**Run.** The run() method is the heart of the simulator, and its implementation is extremely simple. We fetch the instruction whose address is in the PC into an int variable IR and increment the PC (in one statement). Then we decode all the constituent pieces of the instruction (the opcode, result register, argument registers, and address), as just described. With this information extracted, the changes in state of the machine are all one-line implementations, within the switch statement. That is, we simulate the execution of the instruction. It is easy to see what each instruction does because the Java code for each instruction is identical to the description that we gave when we first introduced it.

What happens next depends completely on the instructions and the changes in the machine’s state that they cause. In the same way that TOY executes instructions as per its PC, the virtual machine executes instructions as per its PC variable, continuing until it encounters a halt instruction (opcode 0). For our example program fib.toy, shown at right, the result is to print the Fibonacci numbers on standard output, as expected.

**Standard input and output.** Our boot process takes the program to be loaded from a file, so we can use StdIn for standard input and StdOut for standard output. Specifically, we need to read a value from stan-

```java
private void stdin(int addr, int op, int t)
{
    if ((addr == 0xFF && op == 8) || (R[t] == 0xFF && op == 10))
        M[0xFF] = Convert.parseInt(StdIn.readString(), 16);
}
private void stdout(int addr, int op, int t)
{
    if ((addr == 0xFF && op == 9) || (R[t] == 0xFF && op == 11))
        StdOut.println(Convert.toString(M[0xFF], 16, 4));
}
```

% java TOY fib.toy
0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
000A
000B
000C
000D
000E
000F
0010
0011
0012
0013
0014
0015
0016
0017
0018
0019
001A
001B
001C
001D
001E
001F
0020
0021
0022
0023
0024
0025
0026
0027
0028
0029
002A
002B
002C
002D
002E
002F
0030
0031
0032
0033
0034
0035
0036
0037
0038
0039
003A
003B
003C
003D
003E
003F
0040
0041
0042
0043
0044
0045
0046
0047
0048
0049
004A
004B
004C
004D
004E
004F
0050
0051
0052
0053
0054
0055
0056
0057
0058
0059

Standard input and standard output for TOY virtual machine
standard input when a load (opcode 8) or load indirect (opcode A) instruction accesses memory location FF and to write a value to standard output when a store (opcode 9) or store indirect (opcode B) instruction accesses memory location FF. This code is encapsulated in the methods stdin() and stdout() given at the bottom of the previous page. To add standard input and standard output to Program 6.4.1, just add the call stdin(addr, op, t) just before the main switch statement and the call stdout(addr, op, t) just after. As an alternative, we might arrange to make these checks just before accessing memory, which is actually closer to what the hardware itself might do, as we shall see in Chapter 6. Again, the details of the implementation are inconsequential—our purpose here is just to accurately simulate the behavior of the machine. Note also that we have resisted the temptation to burden you with binary input and output like our punched paper tape (but see Exercise 6.4.2).

The example at left shows the contents of sum.toy, a file representing the 7-word program that we presented as Program 6.3.3, and sum.data, the sample data for that program, which is to be present on standard input, simulating its presence on a punched paper tape. When invoked, the single-line test client in TOY.java loads this program at M[50], sets the PC to 50, and calls run(). Arranging for simulating standard input is easy—we redirect standard input to come from sum.data, as illustrated at right. The simulator calls stdin() to fill M[FF] from standard input each time the 8CFF instruction is executed, so the program reads all the numbers and adds them all together. Finally, it writes the result to standard output with a 98FF instruction and halts. You can see that this same process will be effective for any TOY program.

Developing TOY programs. If needed, we can easily instrument Program 6.4.1 to provide a trace of the PC, register contents, and affected memory cells while the program is running and to provide a dump of memory whenever appropriate (see Exercise 5.4.2). Programmers spend a great deal of time with memory dumps well into the 1980s, as that was the only way to figure out what had gone on within the machine for many sorts of bugs. Indeed you might think of Program 6.4.1 as an extensible TOY development environment—you can use it to implement and debug a TOY program. And you can instrument it however you want.
to give you all the information that you need in order to understand what your program is doing. All of this is accomplished much more easily than if you had to do so on an actual TOY machine (which, in this case, would be impossible, since no TOY machines exist). You can go ahead and add whatever you think you might need to help develop your programs to your version of Program 6.4.1. Indeed, you can find on the booksite a simulator (written by a student like you) that includes a graphic display showing the switches and lights, supports traces and dumps, stepping through a program one instruction at a time, and many other features.

**Coping with Moore’s law.** For at least the last six or seven decades, the speed and memory of state-of-the-art computers has roughly doubled every eighteen months. This rule of thumb, known as Moore’s law presents us all with a constant challenge: how do we go about building a new computer, without wasting all of the effort that we have put into developing software? Virtual machines play an essential role in this process because an early (if not the first) step in the design of any new computer is to build a virtual machine like Program 6.4.1 on the old one.

- Software can be developed on the new computer even before it exists.
- Once the computer is built, its actual behavior can be checked against the behavior of the virtual machine.
- An early (if not the first) piece of software to be developed for the new computer is a virtual machine for the old computer! That way, any software developed on the old computer can be run on the new one.

For example, could we create a TOY program that implement the TOY virtual machine itself? Of course! We had no difficulty translating many other Java programs that we considered into TOY programs, and Program 6.4.1 is a relatively simple Java program (see Exercise 6.4.9). Once we have a TOY virtual machine, we can modify it to improve it: we could add more instructions, more registers, a different word size, or whatever else we might want to try. The resulting TOY virtual machine is “more powerful” than the original TOY machine. This idea is called *bootstrapping*: once we build one machine, we can use it to create “more powerful” machines. This fundamental idea has played an essential role in the design of computers for decades. This famous quote, while perhaps apocryphal, well conveys the idea: “Seymour Cray, founder of Cray Research and father of several generations of supercomputers, heard that Apple had bought a Cray to simulate computer design. Cray was amused, remarking, Funny, I am using an Apple to simulate the Cray-3.”
Here is a question worth pondering. Does TOY exist? There are no physical TOY machines, but we can implement and debug TOY programs, whether or not there exists a physical TOY machine. Indeed, in this sense, TOY is no different than Java. We implement and debug Java programs, but no physical Java machine exists. In fact, Program 6.4.1 is proof that TOY is every bit as real as Java. There may be billions of real machines that implement the Java virtual machine—even one of them also implements TOY. In other words, to the extent that Java exists, so does TOY. Java runs on billions of devices; so does TOY.

Any of the programs that we considered in the previous section (indeed, any TOY program at all) can easily be stored in a file and any data at all can be presented on standard input. The constructor in Program 6.4.1 will load the program into the TOY virtual machine, simulate its operation on the given input and present its output (if any) on standard output.

More important, anyone (even you) can implement any machine of their own design and write programs for it. This is a very powerful idea that has taken our computational infrastructure to where it is and will carry us into the future, as we discuss next.

The TOY family of imaginary computers Our imaginary machine has just 256 words of memory, each 16 bits. Despite the fact that we have demonstrated that one could in principle develop any program in TOY that we could develop in Java, one’s immediate reaction to that constraint is that TOY certainly does not have enough memory to do any important calculations in real applications.

But that reaction would be completely wrong. Computers like TOY were used for all sorts of important applications in the years after they were first introduced. Just to pick one example, the Apollo Guidance Computer that took men to the moon on six occasions had 1024 16-bit words of memory, the equivalent of just four TOY machines!

While external memory devices were improving, from punched paper tape and punched cards to magnetic tapes and disk storage, programmers were realizing that they could make do with a relatively small amount of (expensive) internal memory by organizing their programs in phases that could fit into memory, do their job, and then read in the code for the next phase from external memory. By the 1970s, this attitude led to the concept of virtual memories, where the operating system maintains the illu-
sion that programs have available memory much larger than the machine’s physical memory. This idea still plays a central role in modern computing.

Still, as technology marches on, we find ourselves with bigger memory and faster machines, on a continual basis. Modern computers have billions of bits of memory. How can they really relate to our tiny TOY machine?

To be sure, the scope of the technical advances on all fronts has been incredible, but the fundamental point remains that the essential nature of machine language programs on modern computers is much less different from TOY programming than you might suspect.

**TOY-64.** To relate TOY to modern computers, we imagine a 64-bit TOY machine, which we refer to as TOY-64. For such a machine, we can describe instructions that are precisely the same as the ones we have been considering, but with many more bits devoted to specifying registers and memory locations. Specifically, we can devote 40 bits to memory addresses and 20 bits to register addresses. This would mean that TOY-64 could accommodate more than 68 billion 64 bit words and use more than 250 thousand registers, certainly more like today’s computers than TOY or the PDP-8.

Programming for this machine would be precisely the same as for TOY, except with a much bigger word size, many more registers, and much more memory. Even without us presenting the details, you can see that everything would be represented with 16 hex digits and that all the representations that we have considered extend in a natural way. For example, the number 40544F592D363421 might represent the number 4,635,417,160,900,293,665, the character string “@TOY-64!” or an instruction calling for the bitwise exclusive or of registers 592D3 and 63421 to be computed and stored in register 0544F. It is easy to see how we could convert a program developed for TOY into a program that would work on TOY-64. Because of this ease of translation, the ability to make use of a growing mountain of old software quickly became a compelling design goal in building a new computer. Indeed, a substantial amount of software that we use today was developed decades ago, and we can use it because new machines retain compatibility with old ones.

Most likely, TOY-64 would not have switches and lights, just a wireless interface and an on/off button. The technical details are unimportant. All that matters is that the machine has access to input/output streams that are unbounded in length from the point of view of the program.
The most significant difference between TOY-64 and modern computers is the instruction set. Typical machines devote more bits to the opcode, allowing for a richer set of instructions to perform tasks in hardware, from floating point operations to memory manipulation to external memory support. However, the pendulum swings in computer hardware design. The relative ease of developing software compared with the extreme difficulty of developing reliable high-performance software led to the development of reduced instruction set computing (RISC), which persists to this day. Typical modern computers might have two to four times more instructions than TOY-64, but not much more than that.

Another significant difference between TOY-64 and modern computers is that not many modern computers have so many registers, or instructions like our RR instructions. We will not dwell on this difference, except to note that all computers have a hierarchy of memories ranging from expensive, fast, and small to cheap, slow, and large. Our registers are a placeholder for the idea that differences in memory technologies have to be accounted for in any computer architecture.

The situation is little different from our client-API-implementation model of modular programming. Where should the boundary lie between software and hardware? Many, many computers have settled on an interface sufficiently similar to TOY that you could begin to write programs on them. The effort required to do so might be similar to that required to learn a new programming language.

**TOY-8.** To relate TOY to circuits that implement computers, we also imagine in chapter 7 an 8-bit TOY machine, with 32 words of memory and one register, which we refer to as TOY-8. Writing programs for such a machine would certainly seem to be a challenge, but note that all of the programs that we have considered in this section have taken less than 32 words of memory. When you take into account the

<table>
<thead>
<tr>
<th></th>
<th>TOY</th>
<th>TOY-64</th>
<th>TOY-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits per word</td>
<td>16</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>number of registers</td>
<td>16</td>
<td>262,144</td>
<td>1</td>
</tr>
<tr>
<td>words in memory</td>
<td>256</td>
<td>68,719,476,736</td>
<td>32</td>
</tr>
<tr>
<td>bits per opcode</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>bits per register address</td>
<td>4</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>bits per memory address</td>
<td>8</td>
<td>40</td>
<td>5</td>
</tr>
</tbody>
</table>

*TOY family parameters*
fact that a program can read more code from paper tape, you can see that it actually is possible to get something done with such a tiny machine.

It is sobering to realize that even in TOY–8 the number of possible states of the memory is $2^{256}$, not even taking the external memory into account, so we can never know what TOY–8 can do (most TOY–8 programs will never exist in this universe).

But the point of TOY–8 is to exhibit a complete machine with nearly all the characteristics of TOY, just with less of everything. Our purpose in defining TOY–8 is to be able to exhibit a complete circuit for a computer that contains all the essential elements of TOY (and many other computers). By knowing the characteristics of TOY programming, you can conceive how programs on a machine like TOY–64 and your own computer operate; by knowing how TOY–8 can be built, you can conceive how TOY, TOY–64 and your own computer can be built.

The parameters defining our family of imaginary computers is summarized in the table at the bottom of the facing page and in the figure at the top of this page. Of course, there is a huge gap between TOY and TOY–64 because we have not accounted for the 32-bit computers that were in widespread use for many decades. We leave that for an exercise.

Virtual memory. It would seem that one of TOY’s biggest restrictions is the limited amount of memory that it has, so you might be wondering how we could simulate a machine having more memory than available on our actual machine. This problem actually was addressed very early on, due to the high cost of memory. With paper tape, it is a bit hard to imagine, but within a short amount of time, magnetic tapes and discs became available that could provide a substantial amount of external memory. With such devices the idea of having a virtual memory soon
emerged: Most programs work with a relatively small area of the memory at a given time, so programs would be given access to a large virtual memory that actually mostly resides on external storage and it is the job of the operating system to make sure that the part of the memory that the program needs to access is available in real memory at the appropriate time. In this situation, when more real memory becomes available, programs just work better, because there is less traffic between the real memory and the external memory.

**Backward compatibility.** After many generations of this process, it is an amazing fact that a substantial amount of the software that we use today was written a long time ago on a much less powerful machine. Indeed, it is typical for no one to know anything about the software other than what it does. When you buy a new computer, only a small fraction of the software that you use was written for that computer. This greatly speeds the pace of progress: if the virtual machine works, then all the software works! But after some time has passed, difficulties may arise. A famous example of such a difficulty is the Y2K problem, where old software systems of all sorts had to be rewritten before the year 2000 because they represented the year with just two digits, resulting in people being assigned negative ages and all sorts of other effects.

**Server farms.** Why stop at one computer? Our TOY implementation in Program 6.4.1 is a data type, so we can easily write a client that can create a thousand or a million TOY computers and run them all. And modern cloud computing makes it possible to run them all simultaneously on server farms consisting of large numbers of real processors. Indeed, it is actually becoming less and less likely that significant computing applications will be addressed anywhere other than on a virtual machine in a server farm. When your mobile device recognizes your voice or enhances a photo, there’s a good chance that a virtual computer in a server farm is helping it do the job.

Our imaginary computing machine is certainly a modest contribution to the pantheon of computing devices that have been devised, but in its simplicity we can better appreciate the deep insights into fundamental questions surrounding the nature of computing that we considered in Chapter 5. Exactly what is a computer, or a computer program? Which aspects of a computer are essential? Do there exist new approaches to computing that can significantly expand our horizons? Is there some underlying explanation why emulation seems to tie such a broad spectrum
of different computing devices into a unified whole? There is a direct path from the rigorous examination of these sorts of questions to the practical application of programs that process programs in the form of installers, interpreters, compilers, and other such programs.

Finally, we need to demystify TOY itself by examining how to design a circuit that implements it. That is the subject of the next chapter.
Q. Why not use `short` for `R[]` and `M[]`? They are 16-bit values, as is `short`.

A. Java converts `short` to `int` values for arithmetic, so it is not worth the trouble.

Q. What happens if the PC is set to FF? What value is in FF?

A. Of course we can arrange for any behavior we want, but to answer these questions you have to study the code of `PROGRAM 6.4.1`, which is the definition of `T0Y`. Apparently, `M[FF]` is initially zero, then it holds last value that was read with `stdin()` or written with `stdout()`. Certainly, some hacker can figure out how to exploit that behavior! Maybe it would be better to set it to zero in `stdin()` and `stdout()` so that the machine would halt in this circumstance.

Q. Is T0Y universal (Turing-equivalent)?

A. Not quite, as we have defined it, though replacing the paper tape punch and reader with a read-write bidirectional magnetic tape unit would do the trick. Indeed, that was one of the first upgrades to many old computers.
6.4.1 How many bytes of memory are there in the TOY-64 server farm depicted at the end of the chapter?

6.4.2 Add a command line argument to TOY.java that takes a memory address and a register number and then add code so that TOY.java prints the contents of that register just before the instruction at that memory address is executed, every time that the PC takes on that value. If the argument is 0, print out a dump of the contents of memory on completion, in the form shown in the text (at the beginning of Section 5.3.1).

6.4.3 Develop versions of stdin() and stdout() for TOY.java that simulate punched paper tape: for each 16-bit value, use two 8-character lines, with a blank corresponding to each 0 bit and a * corresponding to each 1 bit.

6.4.4 Modify TOY.java to replace the subtract instruction with a multiply instruction. Be sure to specify and implement a way to cope with the fact that the result of multiplying two 16-bit integers is a 32-bit integer.

6.4.5 Modify TOY.java to support a $2^{16}$ word TOY memory by changing the meaning of every memory reference instruction to be indirect, through the first 256 words of the memory. For example, the instruction 8A23 should load into R[A] the memory work whose 16-bit address is in M[23]. Implement a version of sum.toy for this machine that can add 10,000 16-bit two’s complement values.
6.4.6  One register machine. Design a 16-bit computer with one register, sixteen instructions, and 4096 words of memory. Every two-operand instruction takes one of the operands from the register and the other from memory and leaves the result in the register. Write a simulator for your machine.

6.4.7  Virtual memory. Suppose that the machine has a new external memory device with $2^{32}$ addressable 32-bit words and that its interface to TOY implements a virtual memory, as follows: Two consecutive writes to FF provide a 32-bit address. Then, if the next two references to FF are store (or store indirect) they constitute an instruction to write a 32-bit value to that address, and if the next two references to two reads to FF are load (or load indirect) they constitute an instruction to read a 32-bit value from address. Write a version of sum.toy that can add up to 1 million 32-bit two’s complement values (see Exercise 5.4.6).

6.4.8  Parallel TOY. Modify TOY.java to take an integer N from the command line and then simulate a machine that maintains N PCs, numbered from 0 to $N - 1$. The machine performs fetch-increment-execute for all the PCs simultaneously on each cycle. If two PCs call for different changes to a register on each cycle, then the one with the lower index prevails.

6.4.9  TOY in TOY. Develop a program TOY.toy that implements the TOY virtual machine. Start by assuming that the machine has 32 words of memory, 8 registers, and no standard input/output. You can use the rest of the memory (and standard input/output) to develop your program. Then add standard input/output, more memory, and more registers, to the point that you can run any of the programs in Section 6.3.

6.4.10 String TOY. Design and build a simulator for an imaginary 16-bit string processing machine, with the same registers and memory as TOY, but with string processing operations. Assume that strings are stored as a sequence of words, two ASCII characters per word, terminated with 00. Include operations for string search, substring extraction, and standard input/output. Write a program to sort an array of (references to) strings, using insertion sort.
6.4.11 **Performance.** Is TOY faster than Java? Run doubling tests to compute the ratio of running times of our TOY program and a Java program that uses BSTs for deduping files of random 16-bit integers.

6.4.12 **Assembler.** Write a Java program that takes as input a TOY program written in a slightly higher level language known as *assembly language* and produces as output a TOY program in the format suitable for input to the boot program given in the text. Assembly language supports the use of *symbolic names* for addresses, opcodes, and registers. For example, the following is an assembly language version of Program 6.3.4 (which punches the Fibonacci numbers to standard output).

```
LA one, 1
LA a, 0
LA b, 1
L i, N
loop BZ i, done
ST a, stdout
A c, a, b
A a, b, 0
A b, c, 0
S i, i, one
BZ 0, loop
done H
N 000C
```

You should maintain a 1-1 correspondence between lines of assembly language code and TOY instructions, but details are otherwise left to you. One of the big advantages of assembly language over machine language is that the program can be loaded anywhere (the assembler computes the addresses), so your program should take a starting address as a command-line argument.

6.4.13 **Expression compiler.** Modify Dijkstra’s algorithm (Program 4.3.5) to print a TOY program that can compute the given expression.

6.4.14 **Expression interpreter.** Develop a TOY implementation of Dijkstra’s algorithm (Program 4.3.5), omitting the square root function. Assume that the input expression is on standard input, using the convention that all operands are non-
negative 15-bit unsigned numbers, and use negative numbers to encode operators
and delimiters: 8001, for +, 8002 for −, 8003 for *, 8004 for /, 8005 for (, and 8006
for ). Use your stack implementation from Exercise 6.3.25 and the multiplication
implementation from Exercise 6.3.35.

6.4.15 32-bit TOY. Design a 32-bit TOY computer. Defend each design choice that
you make. Implement a virtual machine for your TOY-32 design.

6.4.16 Bouncing ball. Develop a TOY program that produces instructions for a
drawing machine. Specifically, consider a plotting machine that takes 16-bit com-
mands, as follows. The first hex digit is an op-
code; the others may contain information. For
the purposes of this exercise we are only inter-
ested in two opcodes: The 0 opcode pushes the
12-bit value in the rest of the word on a stack
and the 1 opcode pops three 12-bit values from
the stack (r, then y, then x) from the stack and
draws a circle of radius r centered at (x, y). For
eexample, the code at right produces a sequence
of instruction on standard output that can be
used to instruct the device to plot a moving ball
(left to right, wrapping back to the left edge when
moving off the right edge). Extend this program
to produce the instructions to plot a bouncing
ball, as in Program 1.5.6.

10: 7AEE  x
11: 7BFF  y
12: 710F  dx
13: 7C32  r
14: 9CFF  push r
15: 9BFF  push y
16: 1AA1  x += dx
18: 3AA3  x &= mask
1A: 9AFF  push x
1B: 871D  R[7] = instruction
1C: 97FF  push instruction
1D: C014  loop
1E: 1010
1F: 0FFF

Creating instructions for a drawing device

6.4.17 Virtual drawing machine. Write a Java
program DrawingTOY.java that uses StdDraw to simulate the plotting device de-
scribed in the previous exercise to produce the specified animated drawing. Extend
your machine to include squares, lines, and polygons, and then write TOY code to
produce an interesting graphic design.
6.4 TOY Virtual Machine