Appendices
Number Systems

This appendix introduces background material on various number systems and representations. We start the appendix with a discussion of various number systems, including the binary and hexadecimal systems. When we use multiple number systems, we need to convert numbers from one system to another. We present details on how such number conversions are done. We then give details on integer representations. We cover both unsigned and signed integer representations. We close the appendix with a discussion of the floating-point numbers.

Positional Number Systems

The number systems that we discuss here are based on positional number systems. The decimal number system that we are already familiar with is an example of a positional number system. In contrast, the Roman numeral system is not a positional number system.

Every positional number system has a radix or base, and an alphabet. The base is a positive number. For example, the decimal system is a base-10 system. The number of symbols in the alphabet is equal to the base of the number system. The alphabet of the decimal system is 0 through 9, a total of 10 symbols or digits.

In this appendix, we discuss four number systems that are relevant in the context of computer systems and programming. These are the decimal (base-10), binary (base-2), octal (base-8), and hexadecimal (base-16) number systems. Our intention in including the familiar decimal system is to use it to explain some fundamental concepts of positional number systems.

Computers internally use the binary system. The remaining two number systems—octal and hexadecimal—are used mainly for convenience to write a binary number even though they are number systems on their own. We would have ended up using these number systems if we had 8 or 16 fingers instead of 10.

In a positional number system, a sequence of digits is used to represent a number. Each digit in this sequence should be a symbol in the alphabet. There is a weight associated
with each position. If we count position numbers from right to left starting with zero, the
weight of position \( n \) in a base \( b \) number system is \( b^n \). For example, the number 579 in the
decimal system is actually interpreted as

\[
5 \times (10^2) + 7 \times (10^1) + 9 \times (10^0).
\]

(Of course, \( 10^0 = 1 \).) In other words, 9 is in unit’s place, 7 in 10’s place, and 5 in 100’s
place. More generally, a number in the base \( b \) number system is written as

\[
d_n d_{n-1} \ldots d_1 d_0,
\]

where \( d_0 \) represents the Least Significant Digit (LSD) and \( d_n \) represents the Most Significant Digit (MSD). This sequence represents the value

\[
d_n b^n + d_{n-1} b^{n-1} + \cdots + d_1 b^1 + d_0 b^0.
\]  
(A.1)

Each digit \( d_i \) in the string can be in the range \( 0 \leq d_i \leq (b - 1) \). When we use a number
system with \( b \leq 10 \), we use the first \( b \) decimal digits. For example, the binary system
uses 0 and 1 as its alphabet. For number systems with \( b > 10 \), the initial letters of the
English alphabet are used to represent digits greater than 9. For example, the alphabet
of the hexadecimal system, whose base is 16, is 0 through 9 and A through F, a total of
16 symbols representing the digits of the hexadecimal system. We treat lowercase and
uppercase letters used in a number system such as the hexadecimal system as equivalent.

The number of different values that can be represented using \( n \) digits in a base \( b \n\) system is \( b^n \). Consequently, because we start counting from 0, the largest number that
can be represented using \( n \) digits is \( (b^n - 1) \). This number is written as

\[
\underbrace{(b - 1)(b - 1) \ldots (b - 1)(b - 1)}_{\text{total of } n \text{ digits}}.
\]

The minimum number of digits (i.e., the length of a number) required to represent \( X \n\) different values is given by \( \lceil \log_b X \rceil \), where \( \lceil \rceil \) represents the ceiling function. Note that
\( \lceil m \rceil \) represents the smallest integer that is greater than or equal to \( m \).

**Notation**  The commonality in the alphabet of several number systems gives rise to
confusion. For example, if we write 100 without specifying the number system in which
it is expressed, different interpretations can lead to assigning different values, as shown
below:

<table>
<thead>
<tr>
<th>Number</th>
<th>100</th>
<th>100</th>
<th>100</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>binary</td>
<td>decimal</td>
<td>octal</td>
<td>hexadecimal</td>
</tr>
<tr>
<td>Decimal value</td>
<td>4</td>
<td>100</td>
<td>64</td>
<td>256</td>
</tr>
</tbody>
</table>
Thus, it is important to specify the number system (i.e., specify the base). One common notation is to append a single letter—uppercase or lowercase—to the number to specify the number system. For example, D is used for decimal, B for binary, Q for octal, and H for hexadecimal number systems. Using this notation, 10110111B is a binary number and 2BA9H is a hexadecimal number. Some assemblers use the prefix 0x for hexadecimal and the prefix 0 for octal.

**Decimal Number System**  We use the decimal number system in everyday life. This is a base-10 system presumably because we have 10 fingers and toes to count. The alphabet consists of 10 symbols, digits 0 through 9.

**Binary Number System**  The binary system is a base-2 number system that is used by computers for internal representation. The alphabet consists of two digits, 0 and 1. Each binary digit is called a bit (standing for *binary digit*). Thus, 1021 is not a valid binary number. In the binary system, using $n$ bits, we can represent numbers from 0 through $(2^n - 1)$ for a total of $2^n$ different values.

**Octal Number System**  This is a base-8 number system with the alphabet consisting of digits 0 through 7. Thus, 181 is not a valid octal number. The octal numbers are often used to express binary numbers in a compact way. For example, we need 8 bits to represent 256 different values. The same range of numbers can be represented in the octal system by using only 3 digits.

For example, the number 230Q is written in the binary system as 10011000B, which is difficult to read and error prone. In general, we can reduce the length by a factor of 3. As we show later, it is straightforward to go back to the binary equivalent, which is not the case with the decimal system.

**Hexadecimal Number System**  This is a base-16 number system. The alphabet consists of digits 0 through 9 and letters A through F. In this text, we use capital letters consistently, even though lowercase and uppercase letters can be used interchangeably. For example, FEED is a valid hexadecimal number, whereas GEFF is not.

The main use of this number system is to conveniently represent long binary numbers. The length of a binary number expressed in the hexadecimal system can be reduced by a factor of 4. Consider the previous example again. The binary number 10011000B can be represented as 98H. Debuggers, for example, display information—addresses, data, and so on—in hexadecimal representation.

**Conversion to Decimal**

When we are dealing with several number systems, there is often a need to convert numbers from one system to another. Let us first look at how a number expressed in the base-$b$
system can be converted to the decimal system. To do this conversion, we merely perform the arithmetic calculations of Equation (A.1); that is, multiply each digit by its weight, and add the results. Let’s look at an example next.

**Example A.1 Conversion from binary to decimal.**
Convert the binary number 10100111B into its equivalent in the decimal system.

\[
10100111B = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 167D.
\]

**Conversion from Decimal**

There is a simple method that allows conversions from the decimal to a target number system. The procedure is as follows.

Divide the decimal number by the base of the target number system and keep track of the quotient and remainder. Repeatedly divide the successive quotients while keeping track of the remainders generated until the quotient is zero. The remainders generated during the process, written in the reverse order of generation from left to right, form the equivalent number in the target system.

Let us look at an example now.

**Example A.2 Conversion from decimal to binary.**
Convert the decimal number 167 into its equivalent binary number.

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>167/2 = 83</td>
<td>1</td>
</tr>
<tr>
<td>83/2 = 41</td>
<td>1</td>
</tr>
<tr>
<td>41/2 = 20</td>
<td>1</td>
</tr>
<tr>
<td>20/2 = 10</td>
<td>0</td>
</tr>
<tr>
<td>10/2 = 5</td>
<td>0</td>
</tr>
<tr>
<td>5/2 = 2</td>
<td>1</td>
</tr>
<tr>
<td>2/2 = 1</td>
<td>0</td>
</tr>
<tr>
<td>1/2 = 0</td>
<td>1</td>
</tr>
</tbody>
</table>

The desired binary number can be obtained by writing the remainders generated in the reverse order from left to right. For this example, the binary number is 10100111B. This agrees with the result of Example A.1.
Binary/Octal/Hexadecimal Conversion

Conversion among binary, octal, and hexadecimal number systems is relatively easier and more straightforward. Conversion from binary to octal involves converting three bits at a time, whereas binary to hexadecimal conversion requires converting four bits at a time.

Binary/Octal Conversion  To convert a binary number into its equivalent octal number, form 3-bit groups starting from the right. Add extra 0s at the left-hand side of the binary number if the number of bits is not a multiple of 3. Then replace each group of 3 bits by its equivalent octal digit. Why three bit groups? Simply because $2^3 = 8$. Here is an example.

Example A.3 Conversion from binary to octal.

The following examples illustrate this conversion process.

\[
1000101_B = 001 000 101_B = 105_Q .
\]

\[
10100111_B = 010 100 111_B = 247_Q .
\]

Notice that we have added leftmost 0s (shown in bold) so that the number of bits is 9. Adding 0s on the left-hand side does not change the value of a number. For example, in the decimal system, 35 and 0035 represent the same value.

We can use the reverse process to convert numbers from octal to binary. For each octal digit, write the equivalent 3 bits. You should write exactly 3 bits for each octal digit even if there are leading 0s. For example, for octal digit 0, write the three bits 000.

Example A.4 Conversion from octal to binary.

The following two examples illustrate conversion from octal to binary.

\[
105 = 001 000 101_B ,
\]

\[
247 = 010 100 111_B .
\]

If you want an 8-bit binary number, throw away the leading 0 in the binary number.

Binary/Hexadecimal Conversion  The process for conversion from binary to hexadecimal is similar except that we use 4-bit groups instead of 3-bit groups because
$2^4 = 16$. For each group of 4 bits, replace it by the equivalent hexadecimal digit. If the number of bits is not a multiple of 4, pad 0s at the left. Here is an example.

**Example A.5** *Binary to hexadecimal conversion.*

Convert the binary number 1101011111 into its equivalent hexadecimal number.

\[
1101011111_B = \overbrace{0011}^{3} \overbrace{0101}^{5} \overbrace{1111}^{F} _{B} = \text{35FH}.
\]

As in the octal to binary example, we have added two 0s on the left to make the total number of bits a multiple of 4 (i.e., 12).

The process can be reversed to convert from hexadecimal to binary. Each hex digit should be replaced by exactly four binary bits that represent its value. An example follows.

**Example A.6** *Hex to binary conversion.*

Convert the hexadecimal number B01D into its equivalent binary number.

\[
\begin{align*}
\text{B01DH} & = \overbrace{1011}^{B} \overbrace{0000}^{0} \overbrace{0001}^{1} \overbrace{1101}^{D} _{B} \\
& = 1011\ 0000\ 0001\ 1101\ _{B}.
\end{align*}
\]

**Unsigned Integers**

Now that you are familiar with different number systems, let us turn our attention to how integers (numbers with no fractional part) are represented internally in computers. Of course, we know that the binary number system is used internally. Still, there are a number of other details that need to be sorted out before we have a workable internal number representation scheme.

We begin our discussion by considering how unsigned numbers are represented using a fixed number of bits. We then proceed to discuss the representation for signed numbers in the next section.

The most natural way to represent unsigned (i.e., nonnegative) numbers is to use the equivalent binary representation. As discussed before, a binary number with $n$ bits can represent $2^n$ different values, and the range of the numbers is from 0 to $(2^n - 1)$. Padding of 0s on the left can be used to make the binary conversion of a decimal number equal exactly $N$ bits. For example, we can represent 16D as 10000B using 5 bits. However, this can be extended to a byte (i.e., $N = 8$) as 00010000B or to 16 bits as 00000000000010000B. This process is called *zero extension* and is suitable for unsigned numbers.
A problem arises if the number of bits required to represent an integer in binary is more than the $N$ bits we have. Clearly, such numbers are outside the range of numbers that can be represented using $N$ bits. Recall that using $N$ bits, we can represent any integer $X$ such that $0 \leq X \leq 2^N - 1$.

**Signed Integers**

There are several ways in which signed numbers can be represented. These include

- Signed magnitude,
- Excess-M,
- 1’s complement, and
- 2’s complement.

**Signed Magnitude Representation**

In signed magnitude representation, one bit is reserved to represent the sign of a number. The most significant bit is used as the sign bit. Conventionally, a sign bit value of 0 is used to represent a positive number and 1 for a negative number. Thus, if we have $N$ bits to represent a number, $(N - 1)$ bits are available to represent the magnitude of the number. For example, when $N$ is 4, Table A.1 shows the range of numbers that can be represented. For comparison, the unsigned representation is also included in this table.

The range of $n$-bit signed magnitude representation is $-2^{n-1} + 1$ to $+2^{n-1} - 1$. Note that in this method, 0 has two representations: $+0$ and $-0$.

**Excess-M Representation**

In this method, a number is mapped to a nonnegative integer so that its binary representation can be used. This transformation is done by adding a value called *bias* to the number to be represented. For an $n$-bit representation, the bias should be such that the mapped number is less than $2^n$.

To find out the binary representation of a number in this method, simply add the bias $M$ to the number and find the corresponding binary representation. That is, the representation for number $X$ is the binary representation for the number $X + M$, where $M$ is the bias. For example, in the excess-7 system, $-3D$ is represented as

$$-3 + 7 = +4 = 0100B.$$  

Numbers represented in excess-M are called *biased integers* for obvious reasons. Table A.1 gives examples of biased integers using 4-bit binary numbers. This representation, for example, is used to store the exponent values in the floating-point representation (discussed in the next section).
### Table A.1 Number representation using 4-bit binary (all numbers except Binary column in decimal)

<table>
<thead>
<tr>
<th>Unsigned representation</th>
<th>Binary pattern</th>
<th>Signed magnitude</th>
<th>Excess-7</th>
<th>1’s Complement</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>-7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>-6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>-5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>-4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>-3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>-2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>-1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>-0</td>
<td>1</td>
<td>-7</td>
<td>-8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>-1</td>
<td>2</td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>-2</td>
<td>3</td>
<td>-5</td>
<td>-6</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>-3</td>
<td>4</td>
<td>-4</td>
<td>-5</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>-4</td>
<td>5</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>-5</td>
<td>6</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>-6</td>
<td>7</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>-7</td>
<td>8</td>
<td>-0</td>
<td>-1</td>
</tr>
</tbody>
</table>

### 1’s Complement Representation

As in the excess-M representation, negative values are biased in 1’s complement and 2’s complement representations. For positive numbers, the standard binary representation is used. As in the signed magnitude representation, the most significant bit indicates the sign (0 = positive and 1 = negative). In 1’s complement representation, negative values are biased by \( b^n - 1 \), where \( b \) is the base or radix of the number system. For the binary case that we are interested in here, the bias is \( 2^n - 1 \). For the negative value \(-X\), the representation used is the binary representation for \((2^n - 1) - X\). For example, if \( n \) is 4, we can represent \(-5\) as follows.

\[
2^4 - 1 = \overline{1111B} \\
-5 = \overline{-0101B} = \overline{1010B}
\]

As you can see from this example, the 1’s complement of a number can be obtained by simply complementing individual bits (converting 0s to 1s and vice versa) of the number. Table A.1 shows 1’s complement representation using 4 bits. In this method also, 0 has two representations. The most significant bit is used to indicate the sign. To find the
magnitude of a negative number in this representation, apply the process used to obtain the 1's complement (i.e., complement individual bits) again.

Representation of signed numbers in 1's complement representation allows the use of simpler circuits for performing addition and subtraction than the other two representations we have seen so far (signed magnitude and excess-M). Some older computer systems used this representation for integers. An irritant with this representation is that 0 has two representations. Furthermore, the carry bit generated out of the sign bit will have to be added to the result. The 2's complement representation avoids these pitfalls. As a result, 2's complement representation is the choice of current computer systems.

2's Complement Representation

In 2's complement representation, positive numbers are represented the same way as in the signed magnitude and 1's complement representations. The negative numbers are biased by $2^n$, where $n$ is the number of bits used for number representation. Thus, the negative value $-A$ is represented by $(2^n - A)$ using $n$ bits. Because the bias value is one more than that in the 1's complement representation, we have to add 1 after complementing to obtain the 2's complement representation of a negative number. We can, however, discard any carry generated out of the sign bit. For example, $-5$ can be represented as follows.

$$5D = 0101B \rightarrow \text{complement} \rightarrow 1010B$$

$$\text{add 1} \quad 1B$$

$$\quad 1011B$$

Therefore, $1011B$ represents $-5D$ in 2's complement representation. Table A.1 shows the 2's complement representation of numbers using 4 bits. Notice that there is only one representation for 0. The range of an $n$-bit 2's complement integer is $-2^{n-1}$ to $+2^{n-1} - 1$. For example, using 8 bits, the range is $-128$ to $+127$.

To find the magnitude of a negative number in the 2’s complement representation, as in the 1’s complement representation, simply reverse the sign of the number. That is, use the same conversion process: complement and add 1 and discard any carry generated out of the leftmost bit.

Sign Extension

How do we extend a signed number? For example, we have shown that $-5$ can be represented in the 2’s complement representation as $1011B$. Suppose we want to save this as a byte. How do extend these four bits into eight bits? We have seen in Example A.5 that, for unsigned integers, we add zeros on the left to extend the number. However, we cannot use this technique for signed numbers because the most significant bit represents the sign. To extend a signed number, we have to copy the sign bit. In our example, $-5$ is represented using eight bits as

$$-5D = \overline{1111} 1011.$$
We have copied the sign bit to extend the four-bit value to eight bits. Similarly, we can express $-5$ using 16 bits by extending it as follows.

$-5_{10} = \underbrace{1111111111111111}_{\text{sign bit}} 1011.$

This process is referred to as *sign extension*.

## Floating-Point Representation

Using the decimal system for a moment, we can write very small and very large numbers in scientific notation as follows.

\[
1.2345 \times 10^{45},
\]
\[
9.876543 \times 10^{-37}.
\]

Expressing such numbers using the positional number notation is difficult to write and understand, error prone, and requires more space. In a similar fashion, binary numbers can be written in scientific notation. For example,

\[
+1101.101 \times 2^{+11001} = 13.625 \times 2^{25} = 4.57179 \times 10^{8}.
\]

As indicated, numbers expressed in this notation have two parts: a *mantissa* (or *significand*) and an *exponent*. There can be a sign (+ or −) associated with each part.

Numbers expressed in this notation can be written in several equivalent ways, as shown below:

\[
1.2345 \times 10^{45},
\]
\[
123.45 \times 10^{13},
\]
\[
0.00012345 \times 10^{49}.
\]

This causes implementation problems in performing arithmetic operations, comparisons, and the like. This problem can be avoided by introducing a standard form called the *normal form*. Reverting to the binary case, a normalized binary form has the format

\[
\pm 1.X_1X_2 \cdots X_{M-1}X_M \times 2^{\pm Y_N-1Y_{N-2} \cdots Y_1Y_0},
\]

where $X_i$ and $Y_j$ represent a bit, $1 \leq i \leq M$, and $0 \leq j < N$. The normalized form of

\[
+1101.101 \times 2^{+11010}
\]

is

\[
+1.101101 \times 2^{+11101}.
\]
We normally write such numbers as

\[ +1.101101E11101. \]

To represent such normalized numbers, we might use the format shown below:

\[
\begin{array}{c|c|c}
\text{S}_m & \text{exponent} & \text{mantissa} \\
\hline
1 & \text{N bits} & 1 \\
\hline
0 & \text{M bits} & 0 \\
\end{array}
\]

where \( S_m \) and \( S_e \) represent the sign of the mantissa and the exponent, respectively.

Implementation of floating-point numbers varies from this generic format, usually for efficiency reasons or to conform to a standard. From here on, we discuss the format of the IEEE 754 floating-point standard. Such standards are useful, for example, to exchange data among several different computer systems and to write efficient numerical software libraries.

The single-precision and double-precision floating-point formats are shown in Figure A.1. Certain points are worth noting about these formats:

1. The mantissa stores only the fractional part of a normalized number. The 1 to the left of the binary point is not explicitly stored but implied to save a bit. This bit is always 1, therefore there is really no need to store it. However, representing 0.0 requires special attention, as we show later.

2. There is no sign bit associated with the exponent. Instead, the exponent is converted to an excess-M form and stored. For the single-precision numbers, the bias used is 127D (\( = 7FH \)), and for the double-precision numbers, 1023 (\( = 3FFH \)).
Table A.2 Representation of special values in the floating-point format

<table>
<thead>
<tr>
<th>Special number</th>
<th>Sign</th>
<th>Exponent (biased)</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>−0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+∞</td>
<td>0</td>
<td>FFH</td>
<td>0</td>
</tr>
<tr>
<td>−∞</td>
<td>1</td>
<td>FFH</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>0/1</td>
<td>FFH</td>
<td>≠0</td>
</tr>
<tr>
<td>Denormals</td>
<td>0/1</td>
<td>0</td>
<td>≠0</td>
</tr>
</tbody>
</table>

**Special Values**  The representations of 0 and infinity (∞) require special attention. Table A.2 shows the values of the three components used to represent these values. Zero is represented by a zero exponent and fraction. We can have a −0 or +0 depending on the sign bit. An exponent of all ones indicates a special floating-point value. An exponent of all ones with a zero mantissa indicates infinity. Again, the sign bit indicates the sign of the infinity. An exponent of all ones with a nonzero mantissa represents a not-a-number (NaN). The NaN values are used to represent operations such as 0/0 and √−1.

The last entry in Table A.2 shows how denormalized values are represented. The denormals are used to represent values smaller than the smallest value that can be represented with normalized floating-point numbers. For denormals, the implicit 1 to the left of the binary point becomes a 0. The smallest normalized number has a 1 for the exponent (note zero is not allowed) and 0 for the fraction. Thus, the smallest number is $1 \times 2^{-126}$. The largest denormalized number has a zero exponent and all 1s for the fraction. This represents approximately $0.9999999 \times 2^{-127}$. The smallest denormalized number would have zero as the exponent and a 1 in the last bit position (i.e., position 23). Thus, it represents $2^{-23} \times 2^{-127}$, which is approximately $10^{-45}$. A thorough discussion of floating-point numbers can be found in [8].

**Summary**

We discussed how numbers are represented using the positional number system. Positional number systems are characterized by a base and an alphabet. The familiar decimal system is a base-10 system with the alphabet 0 through 9. Computer systems use the binary system for internal storage. This is a base-2 number system with 0 and 1 as the alphabet. The remaining two number systems—octal (base-8) and hexadecimal (base-16)—are mainly used for convenience in writing a binary number. For example, debuggers use the hexadecimal numbers to display address and data information.

When we use several number systems, there is often a need to convert numbers from one system to another. Conversion among binary, octal, and hexadecimal systems is sim-
ple and straightforward. We also discussed how numbers are converted from decimal to binary and vice versa.

The remainder of the chapter was devoted to internal representation of numbers. Representation of unsigned integers is straightforward and uses binary representation. There are, however, several ways of representing signed integers. We discussed four methods to represent signed integers. Of these four methods, current computer systems use the 2’s complement representation.

Floating-point representation on most computers follows the IEEE 754 standard. There are three components of a floating-point number: mantissa, exponent, and the sign of the mantissa. There is no sign associated with the exponent. Instead, the exponent is stored as a biased number.
Character Representation

This appendix discusses character representation. We identify some desirable properties that a character-encoding scheme should satisfy in order to facilitate efficient character processing. Our focus is on the ASCII encoding; we don’t discuss other character sets such as UCS and Unicode. The ASCII encoding, which is used by most computers, satisfies the requirements of an efficient character code.

Character Representation

As computers have the capability to store and understand the alphabet 0 and 1, characters should be assigned a sequence over this alphabet; that is, characters should be encoded using this alphabet. For efficient processing of characters, several guidelines have been developed. Some of these are mentioned here.

1. Assigning a contiguous sequence of numbers (if treated as unsigned binary numbers) to letters in alphabetical order is desired. Upper- and lowercase letters (A through Z and a through z) can be treated separately, but a contiguous sequence should be assigned to each case. This facilitates efficient character processing such as case conversion, identifying lowercase letters, and so on.

2. In a similar fashion, digits should be assigned a contiguous sequence in numerical order. This would be useful in numeric-to-character and character-to-numeric conversions.

3. A space character should precede all letters and digits.

These guidelines allow for efficient character processing including sorting by names or character strings. For example, to test if a given character code corresponds to a lowercase letter, all we have to do is to see if the code of the character is between that of a and
z. These guidelines also aid in applications requiring sorting, for instance, sorting a class list by last name.

Because computers are rarely used in isolation, exchange of information is an important concern. This leads to the necessity of having some standard way of representing characters. Most computers use the American Standard Code for Information Interchange (ASCII) for character representation. The standard ASCII uses 7 bits to encode a character. Thus, $2^7 = 128$ different characters can be represented. This number is sufficiently large to represent uppercase and lowercase characters, digits, special characters such as !, ^, and control characters such as CR (carriage return), LF (linefeed), and the like.

We store the bits in units of a power of 2, therefore we end up storing 8 bits for each character, even though ASCII requires only 7 bits. The eighth bit is put to use for two purposes.

1. **To parity encode for error detection:** The eighth bit can be used to represent the parity bit. This bit is made 0 or 1 such that the total number of 1s in a byte is even (for even parity) or odd (for odd parity). This can be used to detect simple errors in data transmission.

2. **To represent an additional 128 characters:** By using all eight bits we can represent a total of $2^8 = 256$ different characters. This is referred to as the extended ASCII. These additional codes are used for special graphics symbols, Greek letters, and so on.

The standard ASCII character code is presented in the following two tables. You will notice from these tables that ASCII encoding satisfies the three guidelines mentioned earlier. For instance, successive bit patterns are assigned to uppercase letters, lowercase letters, and digits. This assignment leads to some good properties. For example, the difference between the uppercase and lowercase characters is constant. That is, the difference between the character codes of a and A is the same as that between n and N, which is 32. This characteristic can be exploited for efficient case conversion.

Another interesting feature of ASCII is that the character codes are assigned to the 10 digits such that the lower-order four bits represent the binary equivalent of the corresponding digit. For example, digit 5 is encoded as 0110101. If you take the rightmost four bits (0101), they represent 5 in binary. This feature, again, helps in writing an efficient code for character-to-numeric conversion. Such a conversion, for example, is required when you type a number as a sequence of digit characters.

### ASCII Character Set

The following tables give the standard ASCII character set. We divide the character set into control and printable characters. The control character codes are given next and the printable ASCII characters follow.
### Control Codes

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>NUL</td>
<td>NULL</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>SOH</td>
<td>Start of heading</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>STX</td>
<td>Start of text</td>
</tr>
<tr>
<td>03</td>
<td>3</td>
<td>ETX</td>
<td>End of text</td>
</tr>
<tr>
<td>04</td>
<td>4</td>
<td>EOT</td>
<td>End of transmission</td>
</tr>
<tr>
<td>05</td>
<td>5</td>
<td>ENQ</td>
<td>Enquiry</td>
</tr>
<tr>
<td>06</td>
<td>6</td>
<td>ACK</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>07</td>
<td>7</td>
<td>BEL</td>
<td>Bell</td>
</tr>
<tr>
<td>08</td>
<td>8</td>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>09</td>
<td>9</td>
<td>HT</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>0A</td>
<td>10</td>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>0B</td>
<td>11</td>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>0C</td>
<td>12</td>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>0D</td>
<td>13</td>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>0E</td>
<td>14</td>
<td>SO</td>
<td>Shift out</td>
</tr>
<tr>
<td>0F</td>
<td>15</td>
<td>SI</td>
<td>Shift in</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>DLE</td>
<td>Data link escape</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>DC1</td>
<td>Device control 1</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
<td>DC2</td>
<td>Device control 2</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
<td>DC3</td>
<td>Device control 3</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>15</td>
<td>21</td>
<td>NAK</td>
<td>Negative acknowledgment</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>17</td>
<td>23</td>
<td>ETB</td>
<td>End of transmission block</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>19</td>
<td>25</td>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>1A</td>
<td>26</td>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>1B</td>
<td>27</td>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>1D</td>
<td>29</td>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>1E</td>
<td>30</td>
<td>RS</td>
<td>Record separator</td>
</tr>
<tr>
<td>1F</td>
<td>31</td>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>7F</td>
<td>127</td>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>
## Printable Character Codes

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Character</th>
<th>Hex</th>
<th>Decimal</th>
<th>Character</th>
<th>Hex</th>
<th>Decimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>32</td>
<td>Space</td>
<td>40</td>
<td>64</td>
<td>@</td>
<td>60</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>33</td>
<td>!</td>
<td>41</td>
<td>65</td>
<td>A</td>
<td>61</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>22</td>
<td>34</td>
<td>&quot;</td>
<td>42</td>
<td>66</td>
<td>B</td>
<td>62</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>23</td>
<td>35</td>
<td>#</td>
<td>43</td>
<td>67</td>
<td>C</td>
<td>63</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
<td>$</td>
<td>44</td>
<td>68</td>
<td>D</td>
<td>64</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>%</td>
<td>45</td>
<td>69</td>
<td>E</td>
<td>65</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>26</td>
<td>38</td>
<td>&amp;</td>
<td>46</td>
<td>70</td>
<td>F</td>
<td>66</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>27</td>
<td>39</td>
<td>'</td>
<td>47</td>
<td>71</td>
<td>G</td>
<td>67</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>28</td>
<td>40</td>
<td>(</td>
<td>48</td>
<td>72</td>
<td>H</td>
<td>68</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>29</td>
<td>41</td>
<td>)</td>
<td>49</td>
<td>73</td>
<td>I</td>
<td>69</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>2A</td>
<td>42</td>
<td>*</td>
<td>4A</td>
<td>74</td>
<td>J</td>
<td>6A</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>2B</td>
<td>43</td>
<td>+</td>
<td>4B</td>
<td>75</td>
<td>K</td>
<td>6B</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>2C</td>
<td>44</td>
<td>,</td>
<td>4C</td>
<td>76</td>
<td>L</td>
<td>6C</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>2D</td>
<td>45</td>
<td>-</td>
<td>4D</td>
<td>77</td>
<td>M</td>
<td>6D</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>2E</td>
<td>46</td>
<td>.</td>
<td>4E</td>
<td>78</td>
<td>N</td>
<td>6E</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>2F</td>
<td>47</td>
<td>/</td>
<td>4F</td>
<td>79</td>
<td>O</td>
<td>6F</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>30</td>
<td>48</td>
<td>0</td>
<td>50</td>
<td>80</td>
<td>P</td>
<td>70</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>31</td>
<td>49</td>
<td>1</td>
<td>51</td>
<td>81</td>
<td>Q</td>
<td>71</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>32</td>
<td>50</td>
<td>2</td>
<td>52</td>
<td>82</td>
<td>R</td>
<td>72</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>33</td>
<td>51</td>
<td>3</td>
<td>53</td>
<td>83</td>
<td>S</td>
<td>73</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>34</td>
<td>52</td>
<td>4</td>
<td>54</td>
<td>84</td>
<td>T</td>
<td>74</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>35</td>
<td>53</td>
<td>5</td>
<td>55</td>
<td>85</td>
<td>U</td>
<td>75</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>36</td>
<td>54</td>
<td>6</td>
<td>56</td>
<td>86</td>
<td>V</td>
<td>76</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>37</td>
<td>55</td>
<td>7</td>
<td>56</td>
<td>87</td>
<td>W</td>
<td>77</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>38</td>
<td>56</td>
<td>8</td>
<td>58</td>
<td>88</td>
<td>X</td>
<td>78</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>39</td>
<td>57</td>
<td>9</td>
<td>59</td>
<td>89</td>
<td>Y</td>
<td>79</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>3A</td>
<td>58</td>
<td>;</td>
<td>5A</td>
<td>90</td>
<td>Z</td>
<td>7A</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>3B</td>
<td>59</td>
<td>:</td>
<td>5B</td>
<td>91</td>
<td>[</td>
<td>7B</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>3C</td>
<td>60</td>
<td>&lt;</td>
<td>5C</td>
<td>92</td>
<td>\</td>
<td>7C</td>
<td>124</td>
<td>]</td>
</tr>
<tr>
<td>3D</td>
<td>61</td>
<td>=</td>
<td>5D</td>
<td>93</td>
<td>]</td>
<td>7D</td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>3E</td>
<td>62</td>
<td>&gt;</td>
<td>5E</td>
<td>94</td>
<td>^</td>
<td>7E</td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>3F</td>
<td>63</td>
<td>?</td>
<td>5F</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that 7FH (127 in decimal) is a control character listed in the Control Codes table.
MIPS Instruction Set Summary

This appendix lists the MIPS instructions implemented by the SPIM simulator. These instructions can be divided into two groups: instructions and pseudoinstructions. The first group consists of the instructions supported by the processor. The pseudoinstructions are supported by the assembler; these are not the processor instructions. These pseudoinstructions are translated into one or more processor instructions. For example, abs is a pseudoinstruction, which is translated into the following two-instruction sequence.

\[ \text{abs} \quad \text{Rdest,Rsrc} \]

\[ \text{bgez} \quad \text{Rsrc,8} \]
\[ \text{sub} \quad \text{Rdest,$0,Rsrc} \]

In this appendix, as in the main text, the pseudoinstructions are indicated by a †. In the following, instructions are presented in alphabetical order.

Also note that, in all the instructions, \textit{Src2} can be either a register or a 16-bit integer. The assembler translates the general form of an instruction to its immediate form if \textit{Src2} is a constant. For reference, we also include the immediate form instructions. In these instructions, \textit{Imm} represents a 16-bit integer.

<table>
<thead>
<tr>
<th>abs†</th>
<th>— Absolute value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format:</strong></td>
<td>abs Rdest,Rsrc</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Places the absolute value of Rsrc in Rdest.</td>
</tr>
</tbody>
</table>
### add — Add with overflow

**Format:**  
```
add Rdest, Rsrl, Src2
```

**Description:** Rdest receives the sum of Rsrl and Src2. The numbers are treated as signed integers. In the case of an overflow, an overflow exception is generated.

### addi — Add immediate with overflow

**Format:**  
```
addi Rdest, Rsrl, Imm
```

**Description:** Rdest receives the sum of Rsrl and Imm. The numbers are treated as signed integers. In the case of an overflow, an overflow exception is generated.

### addiu — Add immediate with no overflow

**Format:**  
```
addiu Rdest, Rsrl, Imm
```

**Description:** Rdest receives the sum of Rsrl and Src2. The numbers are treated as signed integers. No overflow exception is generated.

### addu — Add with no overflow

**Format:**  
```
addu Rdest, Rsrl, Src2
```

**Description:** Rdest receives the sum of Rsrl and Src2. The numbers are treated as signed integers. No overflow exception is generated.

### and — Logical AND

**Format:**  
```
and Rdest, Rsrl, Src2
```

**Description:** Bitwise AND of Rsrl and Src2 is stored in Rdest.
**MIPS Instruction Set**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>andi</strong></td>
<td>Logical AND immediate</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td>andi Rdest,Rsrc1,Imm</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Bitwise AND of Rsrc1 and Imm is stored in Rdest.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>b</strong></td>
<td>Branch</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td>b label</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Unconditionally transfer control to the instruction at label. Branch instruction uses a signed 16-bit offset. This allows jumps to $2^{15} - 1$ instructions (not bytes) forward, or $2^{15}$ instructions backward.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bczf</strong></td>
<td>Branch if coprocessor Z is false</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td>bczf label</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Conditionally transfer control to the instruction at label if coprocessor’s Z flag is false.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bczt</strong></td>
<td>Branch if coprocessor Z is true</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td>bczt label</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Conditionally transfer control to the instruction at label if coprocessor’s Z flag is true.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>beq</strong></td>
<td>Branch if equal</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td>beq Rsrc1,Src2,label</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Conditionally transfer control to the instruction at label if Rsrc1 = Src2.</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>beqz†</strong> — Branch if equal to zero</td>
<td>Conditionally transfer control to the instruction at label if Rscc = 0.</td>
</tr>
</tbody>
</table>

**Format:** 
beqz Rscc, label

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bge†</strong> — Branch if greater or equal (signed)</td>
<td>Conditionally transfer control to the instruction at label if Rscc1 ≥ Src2. The contents are treated as signed numbers.</td>
</tr>
</tbody>
</table>

**Format:** 
bge Rscc1, Src2, label

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bgeu†</strong> — Branch if greater or equal (unsigned)</td>
<td>Conditionally transfer control to the instruction at label if Rscc1 ≥ Src2. The contents are treated as unsigned numbers.</td>
</tr>
</tbody>
</table>

**Format:** 
bgeu Rscc1, Src2, label

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bgez</strong> — Branch if greater than or equal to zero</td>
<td>Conditionally transfer control to the instruction at label if Rscc ≥ 0.</td>
</tr>
</tbody>
</table>

**Format:** 
bgez Rscc, label

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bgezal</strong> — Branch if greater than or equal to zero and link</td>
<td>Conditionally transfer control to the instruction at label if Rscc ≥ 0. Save the next instruction address in register 31.</td>
</tr>
</tbody>
</table>

**Format:** 
bgezal Rscc, label
**bgt† — Branch if greater (signed)**

**Format:** \( \text{bgt} \quad \text{Rsrc1}, \text{Src2}, \text{label} \)

**Description:** Conditionally transfer control to the instruction at label if \( \text{Rsrc1} > \text{Src2} \). The contents are treated as signed numbers.

---

**bgtu† — Branch if greater (unsigned)**

**Format:** \( \text{bgtu} \quad \text{Rsrc1}, \text{Src2}, \text{label} \)

**Description:** Conditionally transfer control to the instruction at label if \( \text{Rsrc1} > \text{Src2} \). The contents are treated as unsigned numbers.

---

**bgtz — Branch if greater than zero (signed)**

**Format:** \( \text{bgtz} \quad \text{Rsrc}, \text{label} \)

**Description:** Conditionally transfer control to the instruction at label if \( \text{Rsrc} > 0 \). The contents are treated as signed numbers.

---

**ble† — Branch if less than or equal (signed)**

**Format:** \( \text{ble} \quad \text{Rsrc1}, \text{Src2}, \text{label} \)

**Description:** Conditionally transfer control to the instruction at label if \( \text{Rsrc1} \leq \text{Src2} \). The contents are treated as signed numbers.

---

**bleu† — Branch if less than or equal (unsigned)**

**Format:** \( \text{bleu} \quad \text{Rsrc1}, \text{Src2}, \text{label} \)

**Description:** Conditionally transfer control to the instruction at label if \( \text{Rsrc1} \leq \text{Src2} \). The contents are treated as unsigned numbers.
**blez** — Branch if less than or equal to zero (signed)

**Format:** `bltz Rsrc,label`

**Description:** Conditionally transfer control to the instruction at `label` if `Rsrrc ≤ 0`. The contents are treated as signed numbers.

---

**blt**† — Branch if less than (signed)

**Format:** `blt Rsrrc1,Srcc2,label`

**Description:** Conditionally transfer control to the instruction at `label` if `Rsrrc1 < Srcc2`. The contents are treated as signed numbers.

---

**bltu**† — Branch if less than (unsigned)

**Format:** `bltu Rsrrc1,Srcc2,label`

**Description:** Conditionally transfer control to the instruction at `label` if `Rsrrc1 < Srcc2`. The contents are treated as unsigned numbers.

---

**bltz** — Branch if less than zero (signed)

**Format:** `bltz Rsrrc,label`

**Description:** Conditionally transfer control to the instruction at `label` if `Rsrrc < 0`. The contents are treated as signed numbers.

---

**bltzal** — Branch if less than zero and link

**Format:** `bltzal Rsrrc,label`

**Description:** Conditionally transfer control to the instruction at `label` if `Rsrrc < 0`. Save the next instruction address in register 31.
### MIPS Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bne</strong> — Branch if not equal</td>
<td>Conditionally transfer control to the instruction at label if Rsrc1 ≠ Src2.</td>
<td><code>bne Rsrc1, Src2, label</code></td>
</tr>
<tr>
<td><strong>bnez</strong>† — Branch if not equal to zero</td>
<td>Conditionally transfer control to the instruction at label if Rsrc ≠ 0.</td>
<td><code>bnez Rsrc, label</code></td>
</tr>
<tr>
<td><strong>break</strong> — Exception</td>
<td>Causes exception n. Exception 1 is reserved for the debugger.</td>
<td><code>break n</code></td>
</tr>
<tr>
<td><strong>div</strong> — Divide (signed)</td>
<td>Performs division of two signed numbers in Rsrc1 and Rsrc2 (i.e., Rsrc1/Rsrc2). The quotient is placed in register lo and the remainder in register hi.</td>
<td><code>div Rsrc1, Rsrc2</code></td>
</tr>
<tr>
<td><strong>divu</strong> — Divide (unsigned)</td>
<td>Same as <code>div</code> above except that the numbers in Rsrc1 and Rsrc2 are treated as unsigned.</td>
<td><code>div Rsrc1, Rsrc2</code></td>
</tr>
</tbody>
</table>
**div† — Divide (signed)**

**Format:** `div Rdest,Rsrc1,Src2`

**Description:** Performs division of two signed numbers in `Rsrc1` and `Src2` (i.e., `Rsrc1/Src2`). The quotient is placed in register `Rdest`. `Src2` can be a register or a 16-bit immediate value.

**divu† — Divide (signed)**

**Format:** `divu Rdest,Rsrc1,Src2`

**Description:** Same as the last `div` pseudoinstruction except that the numbers in `Rsrc1` and `Src2` are treated as unsigned.

**j — Jump**

**Format:** `j label`

**Description:** Unconditionally transfer control to the instruction at `label`. Jump instruction uses a signed 26-bit offset. This allows jumps to $2^{25} - 1$ instructions forward, or $2^{25}$ instructions backward.

**jal — Jump and link**

**Format:** `jal label`

**Description:** Unconditionally transfer control to the instruction at `label`. Save the next instruction address in register 31.

**jalr — Jump and link register**

**Format:** `jalr Rsrc`

**Description:** Unconditionally transfer control to the instruction whose address is in `Rsrc`. Save the next instruction address in register 31.
**jr** — Jump register

**Format:** jr Rsrc

**Description:** Unconditionally transfer control to the instruction whose address is in Rsrc. This instruction is used to return from procedures.

**la†** — Load address

**Format:** la Rdest,address

**Description:** Load address into register Rdest.

**lb** — Load byte (signed)

**Format:** lb Rdest,address

**Description:** Load the byte at address into register Rdest. The byte is sign-extended.

**lbu** — Load byte (unsigned)

**Format:** lbu Rdest,address

**Description:** Load the byte at address into register Rdest. The byte is zero-extended.

**ld†** — Load doubleword

**Format:** ld Rdest,address

**Description:** Load the doubleword (64 bits) at address into registers Rdest and Rdest+1.

**lh** — Load halfword (signed)

**Format:** lh Rdest,address

**Description:** Load the halfword (16 bits) at address into register Rdest. The halfword is sign-extended.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lhu</code> — Load halfword (unsigned)</td>
<td>Load the halfword (16 bits) at address into register <code>Rdest</code>. The halfword is zero-extended.</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>lhu</code> <code>Rdest,address</code></td>
<td></td>
</tr>
<tr>
<td><code>li</code> — Load immediate</td>
<td>Load the immediate value <code>Imm</code> into register <code>Rdest</code>.</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>li</code> <code>Rdest,Imm</code></td>
<td></td>
</tr>
<tr>
<td><code>lui</code> — Load upper immediate</td>
<td>Load the 16-bit immediate <code>Imm</code> into the upper halfword of register <code>Rdest</code>. The lower halfword of <code>Rdest</code> is set to 0.</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>lui</code> <code>Rdest,Imm</code></td>
<td></td>
</tr>
<tr>
<td><code>lw</code> — Load word</td>
<td>Load the word (32 bits) at address into register <code>Rdest</code>.</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>lw</code> <code>Rdest,address</code></td>
<td></td>
</tr>
<tr>
<td><code>lwc\_z</code> — Load word from coprocessor <code>z</code></td>
<td>Load the word (32 bits) at address into register <code>Rdest</code> of coprocessor <code>z</code>.</td>
</tr>
<tr>
<td><strong>Format:</strong> <code>lwc\_z</code> <code>Rdest,address</code></td>
<td></td>
</tr>
</tbody>
</table>
lwl — Load word left

**Format:** lwl Rdest, address

**Description:** Load the left bytes from the word at address into register Rdest. This instruction can be used along with lwr to load an unaligned word from memory. The lwl instruction starts loading bytes from (possibly unaligned) address until the lower-order byte of the word. These bytes are stored in Rdest from the left. The number of bytes stored depends on the address. For example, if the address is 1, it stores the three bytes at addresses 1, 2, and 3. As another example, if the address is 2, it stores the two bytes at addresses 2 and 3. See lwr for more details.

lwr — Load word right

**Format:** lwr Rdest, address

**Description:** Load the right bytes from the word at address into register Rdest. This instruction can be used along with lwl to load an unaligned word from memory. The lwr instruction starts loading bytes from (possibly unaligned) address until the higher-order byte of the word. These bytes are stored in Rdest from the right. As in the lwl instruction, the number of bytes stored depends on the address. However, the direction is opposite to that used in the lwl instruction. For example, if the address is 4, it stores just one byte at address 4. As another example, if the address is 6, it stores the three bytes at addresses 6, 5, and 4. In contrast, the lwl instruction with address 6 would load the two bytes at addresses 6 and 7.

As an example, let us look at an unaligned word stored at addresses 1, 2, 3, and 4. We could use the lwl with address 1 to store the bytes at addresses 1, 2, 3; the lwr with address 4 can be used to store the byte at address 4. At the end of this two-instruction sequence, the unaligned word is stored in Rdest.

mfcz — Move from coprocessor z

**Format:** mfcz Rdest, CPsrc

**Description:** Move contents of coprocessor z’s register CPsrc to CPU register Rdest.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mfhi</strong> — Move from hi</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> mfhi Rdest</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Copy contents of hi register to Rdest.</td>
<td></td>
</tr>
<tr>
<td><strong>mflo</strong> — Move from lo</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> mflo Rdest</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Copy contents of lo register to Rdest.</td>
<td></td>
</tr>
<tr>
<td><strong>move†</strong> — Move</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> move Rdest,Rs</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Copy contents of Rs to Rdest.</td>
<td></td>
</tr>
<tr>
<td><strong>mtcz</strong> — Move to coprocessor z</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> mtcz Rs,CPdest</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Move contents of CPU register Rs to coprocessor z’s register CPdest.</td>
<td></td>
</tr>
<tr>
<td><strong>mthi</strong> — Move to hi</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> mfhi Rs</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Copy contents of Rdest to hi register.</td>
<td></td>
</tr>
<tr>
<td><strong>mtlo</strong> — Move to lo</td>
<td></td>
</tr>
<tr>
<td><strong>Format:</strong> mflo Rs</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong> Copy contents of Rdest to lo register.</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>mul†</strong></td>
<td>Signed multiply (no overflow)</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td><code>mul Rdest,Rsrc1,Src2</code></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Perform multiplication of two signed numbers in Rsrc1 and Src2. The result is placed in register Rdest. Src2 can be a register or a 16-bit immediate value. No overflow exception is generated.</td>
</tr>
<tr>
<td><strong>mulo†</strong></td>
<td>Signed multiply (with overflow)</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td><code>mulo Rdest,Rsrc1,Src2</code></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Perform multiplication of two signed numbers in Rsrc1 and Src2. The result is placed in register Rdest. Src2 can be a register or a 16-bit immediate value. If there is an overflow, an overflow exception is generated.</td>
</tr>
<tr>
<td><strong>mulou†</strong></td>
<td>Signed multiply (with overflow)</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td><code>mulou Rdest,Rsrc1,Src2</code></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Perform multiplication of two unsigned numbers in Rsrc1 and Src2. The result is placed in register Rdest. Src2 can be a register or a 16-bit immediate value. If there is an overflow, an overflow exception is generated.</td>
</tr>
<tr>
<td><strong>mult</strong></td>
<td>Multiply (signed)</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td><code>mult Rsrc1,Rsrc2</code></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Perform multiplication of two signed numbers in Rsrc1 and Rsrc2. The lower-order word of result is placed in register lo and the higher-order word in register hi.</td>
</tr>
<tr>
<td><strong>multu</strong></td>
<td>Multiply (unsigned)</td>
</tr>
<tr>
<td><strong>Format:</strong></td>
<td><code>multu Rsrc1,Rsrc2</code></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Same as mult but treat the numbers as unsigned.</td>
</tr>
</tbody>
</table>
neg† — Negation (with overflow)

Format: neg Rdest,Rsrc
Description: Place the negative value of the integer in Rsrc in Rdest. This pseudoinstruction generates overflow exception.

negu† — Negation (no overflow)

Format: neg Rdest,Rsrc
Description: Place the negative value of the integer in Rsrc in Rdest. No overflow exception is generated.

dop — No operation

Format: dop
Description: Do nothing.

nor — Logical NOR

Format: nor Rdest,Rsrc1,Src2
Description: Place the logical NOR of Rsrc1 and Src2 in Rdest.

not† — Logical NOT

Format: not Rdest,Rsrc
Description: Place the logical NOT of Rsrc in Rdest.

or — Logical OR

Format: or Rdest,Rsrc1,Src2
Description: Place the logical OR of Rsrc1 and Src2 in Rdest.
ori — Logical OR immediate

**Format:** ori Rdest,Rsrc1,Imm

**Description:** Place the logical OR of Rsrc1 and Imm in Rdest.

rem† — Remainder (signed)

**Format:** rem Rdest,Rsrc1,Src2

**Description:** Place the remainder from dividing two signed numbers in Rsrc1 and Src2 (Rsrc1/Src2) in register Rdest. Src2 can be a register or a 16-bit immediate value. If an operand is negative, the remainder is unspecified by the MIPS architecture. The corresponding SPIM value depends on the machine it is running.

remu† — Remainder (unsigned)

**Format:** remu Rdest,Rsrc1,Src2

**Description:** Same as rem except that the numbers are treated as unsigned.

rol† — Rotate left

**Format:** rol Rdest,Rsrc1,Src2

**Description:** Rotate the contents of register Rsrc1 left by the number of bit positions indicated by Src2 and place the result in Rdest.

ror† — Rotate left

**Format:** ror Rdest,Rsrc1,Src2

**Description:** Rotate the contents of register Rsrc1 right by the number of bit positions indicated by Src2 and place the result in Rdest.
### sb — Store byte

**Format:** \( \text{sb} \quad \text{Rsrc}, \text{address} \)

**Description:** Store the lowest byte from register \( \text{Rdest} \) at address.

### sd — Store doubleword

**Format:** \( \text{sd} \quad \text{Rsrc}, \text{address} \)

**Description:** Store the doubleword (64 bits) from registers \( \text{Rdest} \) and \( \text{Rdest}+1 \) at address.

### seq† — Set if equal

**Format:** \( \text{seq} \quad \text{Rdest}, \text{Rsrc1}, \text{Src2} \)

**Description:** Set register \( \text{Rdest} \) to 1 if \( \text{Rsrc1} \) is equal to \( \text{Src2} \); otherwise, \( \text{Rdest} \) is 0.

### sge† — Set if greater than or equal (signed)

**Format:** \( \text{sge} \quad \text{Rdest}, \text{Rsrc1}, \text{Src2} \)

**Description:** Set register \( \text{Rdest} \) to 1 if \( \text{Rsrc1} \) is greater than or equal to \( \text{Src2} \); otherwise, \( \text{Rdest} \) is 0. \( \text{Rsrc1} \) and \( \text{Src2} \) are treated as signed numbers.

### sgeu† — Set if greater than or equal (unsigned)

**Format:** \( \text{sgeu} \quad \text{Rdest}, \text{Rsrc1}, \text{Src2} \)

**Description:** Same as \( \text{sge} \) except that \( \text{Rsrc1} \) and \( \text{Src2} \) are treated as unsigned numbers.
### sgt† — Set if greater than (signed)

**Format:**  
\[
\text{sgt} \quad \text{Rdest}, \text{Rsrec1}, \text{Src2}
\]

**Description:**  
Set register Rdest to 1 if Rsrec1 is greater than Src2; otherwise, Rdest is 0. Rsrec1 and Src2 are treated as signed numbers.

### sgtu† — Set if greater than (unsigned)

**Format:**  
\[
\text{sgtu} \quad \text{Rdest}, \text{Rsrec1}, \text{Src2}
\]

**Description:**  
Same as sgt except that Rsrec1 and Src2 are treated as unsigned numbers.

### sh — Store halfword

**Format:**  
\[
\text{sh} \quad Rsrc, \text{address}
\]

**Description:**  
Store the lower halfword (16 bits) from register Rsrc at address.

### sle† — Set if less than or equal (signed)

**Format:**  
\[
\text{sle} \quad \text{Rdest}, \text{Rsrec1}, \text{Src2}
\]

**Description:**  
Set register Rdest to 1 if Rsrec1 is less than or equal to Src2; otherwise, Rdest is 0. Rsrec1 and Src2 are treated as signed numbers.

### sleu† — Set if less than or equal (unsigned)

**Format:**  
\[
\text{sleu} \quad \text{Rdest}, \text{Rsrec1}, \text{Src2}
\]

**Description:**  
Same as sle except that Rsrec1 and Src2 are treated as unsigned numbers.
**sll** — Shift left logical

**Format:** `sll  Rdest,Rsrc1,count`

**Description:** Shift the contents of register `Rsrc1` left by `count` bit positions and places the result in `Rdest`. Shifted-out bits are filled with zeros.

**sllv** — Shift left logical variable

**Format:** `sllv  Rdest,Rsrc1,Rsrc2`

**Description:** Shift the contents of register `Rsrc1` left by the number of bit positions indicated by `Rsrc2` and places the result in `Rdest`. Shifted-out bits are filled with zeros.

**slt** — Set if less than (signed)

**Format:** `slt  Rdest,Rsrc1,Src2`

**Description:** Set register `Rdest` to 1 if `Rsrc1` is less than `Src2`; otherwise, `Rdest` is 0. `Rsrc1` and `Src2` are treated as signed numbers.

**slti** — Set if less than immediate (signed)

**Format:** `slti  Rdest,Rsrc1,Imm`

**Description:** Set register `Rdest` to 1 if `Rsrc1` is less than `Imm`; otherwise, `Rdest` is 0. `Rsrc1` and `Imm` are treated as signed numbers.

**sltiu** — Set if less than immediate (unsigned)

**Format:** `sltiu  Rdest,Rsrc1,Imm`

**Description:** Same as `slti` except that `Rsrc1` and `Imm` are treated as unsigned numbers.
**sltu** — Set if less than (unsigned)

**Format:** `sltu Rdest, Rsrlc, Src2`

**Description:** Same as `slt` except that `Rsrlc` and `Src2` are treated as unsigned numbers.

**sne †** — Set if not equal

**Format:** `sne Rdest, Rsrlc, Src2`

**Description:** Set register `Rdest` to 1 if `Rsrlc` is not equal to `Src2`; otherwise, `Rdest` is 0.

**sra** — Shift right arithmetic

**Format:** `sra Rdest, Rsrlc, count`

**Description:** Shift the contents of register `Rsrlc` right by `count` bit positions and place the result in `Rdest`. Shifted-out bits are filled with the sign bit.

**srav** — Shift right arithmetic variable

**Format:** `srav Rdest, Rsrlc, Rsrlc2`

**Description:** Shift the contents of register `Rsrlc` right by the number of bit positions indicated by `Rsrlc2` and place the result in `Rdest`. Shifted-out bits are filled with the sign bit.

**srl** — Shift right arithmetic

**Format:** `srl Rdest, Rsrlc, count`

**Description:** Shift the contents of register `Rsrlc` right by `count` bit positions and place the result in `Rdest`. Shifted-out bits are filled with zeros.
**srlv** — Shift right arithmetic variable

**Format:**  
srlv Rdest,Rsrc1,Rsrc2

**Description:** Shift the contents of register Rsrc1 right by the number of bit positions indicated by Rsrc2 and place the result in Rdest. Shifted-out bits are filled with zeros.

**sub** — Subtract with overflow

**Format:**  
sub Rdest,Rsrc1,Src2

**Description:** Rdest receives the difference of Rsrc1 and Src2 (i.e., Rsrc1 − Src2). The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.

**subu** — Subtract with no overflow

**Format:**  
subu Rdest,Rsrc1,Src2

**Description:** Same as sub but no overflow exception is generated.

**sw** — Store word

**Format:**  
sw Rsrrc,address

**Description:** Store the word from register Rsrrc at address.

**swcz** — Store word coprocessor z

**Format:**  
sw Rsrrc,address

**Description:** Store the word from register Rsrrc of coprocessor z at address.
swl — Store word left

**Format:**  
```
swl  Rsrrc,address
```

**Description:**  
Copy the left bytes from register `Rsrrc` to memory at address. This instruction can be used along with `swr` to store a word in memory at an unaligned address. The `swl` instruction starts storing the bytes from the most-significant byte of `Rsrrc` to memory at address until the lower-order byte of the word in memory is reached. For example, if the address is 1, it stores the three most significant bytes of `Rsrrc` at addresses 1, 2, and 3. As another example, if the address is 2, it stores the two most significant bytes of `Rsrrc` at addresses 2 and 3.

swr — Store word right

**Format:**  
```
swr  Rsrrc,address
```

**Description:**  
Copy the right bytes from register `Rsrrc` to memory at address. This instruction can be used along with `swl` to store a word in memory at an unaligned address. The `swr` instruction starts storing the bytes from the least-significant byte of `Rsrrc` to memory at address until the higher-order byte of the word in memory is reached. For example, if the address is 1, it stores the two least significant bytes of `Rsrrc` at addresses 1 and 0. As another example, if the address is 2, it stores the three least significant bytes of `Rsrrc` at addresses 2, 1, and 0.

ulh† — Unaligned load halfword (signed)

**Format:**  
```
ulh  Rdest,address
```

**Description:**  
Load the halfword (16 bits) from the word at address into register `Rdest`. The address could be unaligned. The halfword is sign-extended.
ulhu† — Unaligned load halfword (unsigned)

**Format:** ulhu  Rdest,address

**Description:** Load the halfword (16 bits) from the word at address into register Rdest. The address could be unaligned. The halfword is zero-extended.

ulw† — Unaligned load word

**Format:** ulw  Rdest,address

**Description:** Load the word (32 bits) at address into register Rdest. The address could be unaligned.

ush† — Unaligned store halfword

**Format:** ush  Rs src,address

**Description:** Store the lower halfword (16 bits) from register Rs src at address. The address could be unaligned.

usw† — Unaligned store word

**Format:** usw  Rs src,address

**Description:** Store the word (32 bits) from register Rs src at address. The address could be unaligned.

xor — Logical XOR

**Format:** xor  Rdest,Rsrc1,Src2

**Description:** Place the logical XOR of Rsrc1 and Src2 in Rdest.

xori — Logical XOR immediate

**Format:** xori  Rdest,Rsrc1,Imm

**Description:** Place the logical XOR of Rsrc1 and Imm in Rdest.
Programming Exercises

This appendix gives several programming exercises. These exercises can be used to practice writing programs in the MIPS assembly language.

1. Modify the addigits.asm program given in Example 10.1 on page 174 such that it accepts a string from the keyboard consisting of digit and nondigit characters. The program should display the sum of the digits present in the input string. All nondigit characters should be ignored. For example, if the input string is

   ABC1?5wy76:~2

   the output of the program should be

   sum of individual digits is: 21

2. Write an assembly language program to encrypt digits as shown below:

   input digit: 0 1 2 3 4 5 6 7 8 9
   encrypted digit: 4 6 9 5 0 3 1 8 7 2

   Your program should accept a string consisting of digit and nondigit characters. The encrypted string should be displayed in which only the digits are affected. Then the user should be queried whether he or she wants to terminate the program. If the response is either “y” or “Y” you should terminate the program; otherwise, you should request another input string from the keyboard.

   The encryption scheme given here has the property that when you encrypt an already encrypted string, you get back the original string. Use this property to verify your program.

3. Write a program to accept a number in the hexadecimal form and display the decimal equivalent of the number. A typical interaction of your program is (user input is shown in bold):
Please input a positive number in hex (4 digits max.): \textbf{A10F}  
The decimal equivalent of A10FH is 41231  
Do you want to terminate the program (Y/N): \textbf{Y}

You can refer to Appendix A for an algorithm to convert from base \(b\) to decimal. You should do the required multiplication by the left-shift instruction. Once you have converted the hex number into the equivalent in binary, you can use the \texttt{print\_int} system call to display the decimal equivalent.

4. Write a program that reads an input number (given in decimal) between 0 and 65,535 and displays the hexadecimal equivalent. You can read the input using the \texttt{read\_int} system call.

5. Modify the above program to display the octal equivalent instead of the hexadecimal equivalent of the input number.

6. Write a procedure \texttt{locate} to locate a character in a given string. The procedure receives a pointer to a NULL-terminated character string and the character to be located. When the first occurrence of the character is located, its position is returned to \texttt{main}. If no match is found, a negative value is returned. The \texttt{main} procedure requests a character string and a character to be located and displays the position of the first occurrence of the character returned by the \texttt{locate} procedure. If there is no match, a message should be displayed to that effect.

7. Write a procedure that receives a string and removes all leading blank characters in the string. For example, if the input string is (\(\square\) indicates a blank character)

\[
\square \square \square \square \text{Read}\square \square \text{my}\square \square \text{lips}.
\]

it will be modified by removing all leading blanks as

\[
\text{Read}\square \square \text{my}\square \square \text{lips}.
\]

Write a main program to test your procedure.

8. Write a procedure that receives a string and removes all leading and duplicate blank characters in the string. For example, if the input string is (\(\square\) indicates a blank character)

\[
\square \square \square \square \text{Read}\square \square \square \text{my} \square \square \square \square \square \text{lips}.
\]

it will be modified by removing all leading and duplicate blanks as

\[
\text{Read}\square \text{my}\square \square \text{lips}.
\]

Write a main program to test your procedure.

9. Write a procedure to read a string, representing a person’s name, in the format

\[
\text{first-name}\square \text{MI} \square \text{last-name}
\]

and display the name in the format

\[
\text{last-name}, \square \text{first-name}\square \text{MI}
\]
where ⊙ indicates a blank character. As indicated, you can assume that the three names—first name, middle initial, and last name—are separated by single spaces. Write a main program to test your procedure.

10. Modify the last exercise to work on an input that can contain multiple spaces between the names. Also, display the name as in the last exercise but with the last name in all capital letters.

11. Write a complete assembly language program to read two matrices \( A \) and \( B \) and display the result matrix \( C \), which is the sum of \( A \) and \( B \). Note that the elements of \( C \) can be obtained as

\[
\]

Your program should consist of a main procedure that calls the \textit{read_matrix} procedure twice to read data for \( A \) and \( B \). It should then call the \textit{matrix_add} procedure, which receives pointers to \( A, B, C \), and the size of the matrices. Note that both \( A \) and \( B \) should have the same size. The \texttt{main} procedure calls another procedure to display \( C \).

12. Write a procedure to perform multiplication of matrices \( A \) and \( B \). The procedure should receive pointers to the two input matrices (\( A \) of size \( l \times m \), \( B \) of size \( m \times n \)), the product matrix \( C \), and values \( l \), \( m \), and \( n \). Also, the data for the two matrices should be obtained from the user. Devise a suitable user interface to read these numbers.

13. Modify the program of the last exercise to work on matrices stored in the column-major order.

14. Write a program to read a matrix (maximum size \( 10 \times 10 \)) from the user and display the transpose of the matrix. To obtain the transpose of matrix \( A \), write rows of \( A \) as columns. Here is an example.

If the input matrix is

\[
\begin{bmatrix}
12 & 34 & 56 & 78 \\
23 & 45 & 67 & 89 \\
34 & 56 & 78 & 90 \\
45 & 67 & 89 & 10
\end{bmatrix}
\]

the transpose of the matrix is

\[
\begin{bmatrix}
12 & 23 & 34 & 45 \\
34 & 45 & 56 & 67 \\
56 & 67 & 78 & 89 \\
78 & 89 & 90 & 10
\end{bmatrix}
\]

15. Write a program to read a matrix (maximum size \( 10 \times 15 \)) from the user and display the subscripts of the maximum element in the matrix. Your program should consist of two procedures: \texttt{main} is responsible for reading the input matrix and for
displaying the position of the maximum element. Another procedure `mat_max` is responsible for finding the position of the maximum element. For example, if the input matrix is

\[
\begin{bmatrix}
12 & 34 & 56 & 78 \\
23 & 45 & 67 & 89 \\
34 & 56 & 78 & 90 \\
45 & 67 & 89 & 10 \\
\end{bmatrix}
\]

the output of the program should be

The maximum element is at (2,3),

which points to the largest value (90 in our example).

16. Write a program to read a matrix of integers, perform cyclic permutation of rows, and display the result matrix. Cyclic permutation of a sequence \(a_0, a_1, a_2, \ldots, a_{n-1}\) is defined as \(a_1, a_2, \ldots, a_{n-1}, a_0\). Apply this process for each row of the matrix. Your program should be able to handle up to \(12 \times 15\) matrices. If the input matrix is

\[
\begin{bmatrix}
12 & 34 & 56 & 78 \\
23 & 45 & 67 & 89 \\
34 & 56 & 78 & 90 \\
45 & 67 & 89 & 10 \\
\end{bmatrix}
\]

the permuted matrix is

\[
\begin{bmatrix}
34 & 56 & 78 & 12 \\
45 & 67 & 89 & 23 \\
56 & 78 & 90 & 34 \\
67 & 89 & 10 & 45 \\
\end{bmatrix}
\]

17. Generalize the last exercise to cyclically permute by a user-specified number of elements.

18. Write a complete assembly language program to do the following.

- Read the names of students in a class into a one-dimensional array.
- Read test scores of each student into a two-dimensional marks array.
- Output a letter grade for each student in the format:
  
  student name letter grade

You can use the following information in writing your program:

- Assume that the maximum class size is 20.
- Assume that the class is given four tests of equal weight (i.e., 25 points each).
- Test marks are rounded to the nearest integer so you can treat them as integers.
- Use the following table to convert percentage marks (i.e, sum of all four tests) to a letter grade.
19. Modify the program for the last exercise to also generate a class summary stating the number of students receiving each letter grade in the following format:

<table>
<thead>
<tr>
<th>Grade</th>
<th>Marks range</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>85–100</td>
</tr>
<tr>
<td>B</td>
<td>70–84</td>
</tr>
<tr>
<td>C</td>
<td>60–69</td>
</tr>
<tr>
<td>D</td>
<td>50–59</td>
</tr>
<tr>
<td>F</td>
<td>0–49</td>
</tr>
</tbody>
</table>

A = number of students receiving A,
B = number of students receiving B,
C = number of students receiving C,
D = number of students receiving D,
F = number of students receiving F.

20. If we are given a square matrix (i.e., a matrix with the number of rows equal to the number of columns), we can classify it as a diagonal matrix if only its diagonal elements are nonzero; as an upper triangular matrix if all the elements below the diagonal are 0; and as a lower triangular matrix if all elements above the diagonal are 0. Some examples are:

Diagonal matrix:
\[
\begin{bmatrix}
28 & 0 & 0 & 0 \\
0 & 87 & 0 & 0 \\
0 & 0 & 97 & 0 \\
0 & 0 & 0 & 65
\end{bmatrix};
\]

Upper triangular matrix:
\[
\begin{bmatrix}
19 & 26 & 35 & 98 \\
0 & 78 & 43 & 65 \\
0 & 0 & 38 & 29 \\
0 & 0 & 0 & 82
\end{bmatrix};
\]

Lower triangular matrix:
\[
\begin{bmatrix}
76 & 0 & 0 & 0 \\
44 & 38 & 0 & 0 \\
65 & 28 & 89 & 0 \\
87 & 56 & 67 & 54
\end{bmatrix}.
\]

Write an assembly language program to read a matrix and output the type of matrix.
21. In Appendix A, we discussed the format of the single-precision floating-point numbers. Write a program that reads the floating-point internal representation from the user as a string of eight hexadecimal digits and displays the three components—mantissa, exponent, and sign—in binary. For example, if the input to the program is 429DA000, the output should be:

   sign = 0
   mantissa = 1.0011101101
   exponent = 110.

22. Modify the program for the last exercise to work with the double-precision floating-point representation.

23. Ackermann’s function \( A(m, n) \) is defined for \( m \geq 0 \) and \( n \geq 0 \) as

\[
A(0,n) = N + 1 \quad \text{for } n \geq 0 \\
A(m,0) = A(m - 1,1) \quad \text{for } m \geq 1 \\
A(m,n) = A(m - 1,A(m,n - 1)) \quad \text{for } m \geq 1, n \geq 1.
\]

Write a recursive procedure to compute this function. Your main program should handle the user interface to request \( m \) and \( n \) and display the final result.

24. Write a program to solve the Towers of Hanoi puzzle. The puzzle consists of three pegs and \( N \) disks. Disk 1 is smaller than disk 2, which is smaller than disk 3, and so on. Disk \( N \) is the largest. Initially, all \( N \) disks are on peg 1 such that the largest disk is at the bottom and the smallest at the top (i.e., in the order \( N, N - 1, \ldots, 3, 2, 1 \) from bottom to top). The problem is to move these \( N \) disks from peg 1 to peg 2 under two constraints: you can move only one disk at a time and you must not place a larger disk on top of a smaller one. We can express a solution to this problem by using recursion. The function

   \text{move}(N, 1, 2, 3)

moves \( N \) disks from peg 1 to peg 2 using peg 3 as the extra peg. There is a simple solution if you concentrate on moving the bottom disk on peg 1. The task \text{move}(N, 1, 2, 3) \) is equivalent to

   \text{move}(N-1, 1, 3, 2) \\
   \text{move the remaining disk from peg 1 to 2} \\
   \text{move}(N-1, 3, 2, 1)

Even though the task appears to be complex, we write a very elegant and simple solution to solve this puzzle. Here is a version in C.

   \text{void move (int n, int x, int y, int z)} \\
   \{ \\
   \text{if (n == 1)} \\
   \text{printf("Move the top disk from peg %d to %d\n",x,y);}
else
    move(n-1, x, z, y)
    printf("Move the top disk from peg %d to %d\n",x,y);
    move(n-1, z, y, x)
}

int main (void)
{
    int    disks;

    scanf("%d", &disks);
    move(disks, 1, 2, 3);
}

Test your program for a very small number of disks (say, less than 6). Even for 64 disks, it takes hundreds of years on whatever PC you have!

25. Write a procedure str_str that receives two pointers to strings string and substring and searches for substring in string. If a match is found, it returns the starting position of the first match. Matching should be case sensitive. A negative value is returned if no match is found. For example, if

    string = Good things come in small packages.

and

    substring = in

the procedure should return 8 indicating a match of in in things.

26. Write a procedure strncpy to mimic the strncpy function provided by the C library. The function strncpy receives two strings, string1 and string2, and a positive integer num. Of course, the procedure receives only the string pointers but not the actual strings. It should copy at most the first num characters from string2 to string1.

27. A palindrome is a word, verse, sentence, or number that reads the same backward or forward. Blanks, punctuation marks, and capitalization do not count in determining palindromes. Here are some examples:

    1991
    Able was I ere I saw Elba
    Madam! I’m Adam

Write a program to determine if a given string is a palindrome. The procedure returns 1 if the string is a palindrome; otherwise, it returns 0.

28. Write an assembly language program to read a string of characters from the user and print the vowel count. For each vowel, the count includes both uppercase and lowercase letters. For example, the input string
29. Merge sort is a technique to combine two sorted arrays. Merge sort takes two sorted input arrays X and Y—say of size m and n—and produces a sorted array Z of size m + n that contains all elements of the two input arrays. The pseudocode of merge sort is as follows.

```pseudocode
mergesort(X, Y, Z, m, n)
   i := 0 {index variables for arrays X, Y, and Z}
   j := 0
   k := 0
   while ((i < m) AND (j < n))
      if (X[i] <= Y[j]) {find largest of two}
         then
            Z[k] := X[i] {copy and update indices}
            k := k+1
            i := i+1
         else
            Z[k] := Y[j] {copy and update indices}
            k := k+1
            j := j+1
         end if
      end while
   if (i < m) {copy remainder of input array}
      while (i < m)
         Z[k] := X[i]
         k := k+1
         i := i+1
      end while
   else
      while (j < n)
         Z[k] := Y[j]
         k := k+1
         j := j+1
      end while
```
The merge sort algorithm scans the two input arrays while copying the smallest of the two elements from X and Y into Z. It updates indices appropriately. The first while loop terminates when one of the arrays is exhausted. Then the other array is copied into Z.

Write a merge sort procedure and test it with two sorted arrays. Assume that the user enters the two input arrays in sorted (ascending) order.
Bibliography


Index

Symbols
.ALIGN, 164
.FLOAT, 163
.GLOBL, 164
.HALF, 163
.SPACE, 163
1’s complement, 332
1-address machines, 16
2’s complement, 333
2-address machines, 15
3-address machines, 14

A
absolute address, 23, 247
accumulator machines, 16
Ackermann’s function, 370
activation record, 42, 292, 295
address
absolute, 247
PC-relative, 247
addresses
0-address machines, 17
1-address machines, 16
2-address machines, 15
3-address machines, 14
accumulator machines, 16
comparison, 18
number of, 14–22
stack machines, 17
addressing modes, 33, 50, 58, 83, 100, 125, 212
immediate addressing mode, 33, 100
immediate index addressing mode, 83
immediate update index addressing mode, 84
in ARM, 125
in MIPS, 50, 212
in PowerPC, 83
in SPARC, 58
index addressing mode, 84, 100
indirect addressing mode, 100
register addressing mode, 33
advanced load, 107
architecture
CISC, 5
Itanium, 98
load/store, 20
RISC, 5, 43
arithmetic instructions, 35
ARM
addressing modes, 125
general-purpose registers, 123
instruction set, 131
processor modes, 123
program status registers, 123
registers, 123
write-back, 127
ARM instructions
adc, 137
add, 136
and, 139
asr, 132
b, 142
bic, 139

379
bl, 143
clz, 139
cmp, 141
eor, 139
ldm, 135
ldmf, 144
ldr, 134
ldrb, 135
ldrh, 135
ldrsb, 135
ldrsh, 135
lsl, 132
lsr, 132
mla, 138
mov, 131
mul, 138
mvn, 133
orr, 139
ror, 132
rrx, 132
rsb, 138
rsc, 138
sbc, 137
smlal, 138
smull, 138
stm, 136
stmf, 144
str, 136
sub, 137
teq, 140
tst, 140
umlal, 138
umull, 138
arrays
  column-major order, 216
  multidimensional, 215
  one-dimensional, 214
  row-major order, 215
ASCII string, 163
ASCIIIZ string, 163
assembler directives, 160, 162
assembly language, 4, 7–10

advantages, 10
what is it, 7–9

B
binary numbers, 327
conversion, 329
bit manipulation, 277
  clearing bits, 271
  cutting and pasting, 274
  isolating bits, 273
  toggling, 275
branch
  absolute address, 23
  conditional, 24
    set-then-jump, 24
    test-and-jump, 24
  overview, 22
  PC-relative, 23
  unconditional, 23
branch elimination, 112, 113
branch handling, 112–114
branch hints, 112, 119
branch prediction, 28–32, 113
  dynamic strategy, 30
  fixed strategy, 28
  in Itanium, 119
  static strategy, 29
branch speedup, 112

call-by-reference, 184
call-by-value, 184
character representation, 339–340
  extended ASCII, 340
CISC processors
  evolution, 40
  VAX-11/780, 40, 43
column-major order, 216
conditional branch, 24
control speculation, 118
CTR register, 83
current frame marker register, 101

D
data dependency
  ambiguous, 115
  read-after-write, 115
  write-after-read, 115
  write-after-write, 115
data movement instructions, 34
data speculation, 115
delayed procedure call, 27
denormalized values, 336
dynamic branch prediction strategy, 30

E
eンド of procedure, 26
EPIC design, 97
equivalence function, 91
even parity, 272, 274
excess-M number representation, 331
executable instructions, 160
explicit parallel instruction computing, see EPIC design

F
factorial, 291–295
  recursive procedure, 292
Fibonacci function, 295, 303
  MIPS procedure, 295
fixed branch prediction strategy, 28
floating-point, 334–336
  denormals, 336
  IEEE 754, 335
  representation, 334
  special values, 336
  ∞, 336
  NaN, 336
  zero, 336
FPSCR register, 81

G
general-purpose registers, 48, 81
  in ARM, 123
  in Itanium processor, 98
  in MIPS, 48
  in PowerPC, 81
  usage in MIPS, 48

H
hexadecimal numbers, 327

I
IEEE 754 floating-point standard, 335
  immediate addressing mode, 33, 100
  immediate index addressing mode, 83, 84
  index addressing mode, 84, 100
  indirect addressing mode, 100
  indirect jump, 259
  instruction format, 35–36, 51, 59
  instruction set design, 32–36
    issues, 32
    addressing modes, 33
    instruction types, 34
    operand types, 32
  instruction types, 34
    arithmetic, 35
    data movement, 34
    logical, 35
  instruction-level parallelism, 97, 105–106
  Itanium instructions, 102–112
    add, 108
    advanced load, 107
    and, 110
    br, 111
    branch hints, 112
    call, 112
    cmp, 110, 114
    instruction format, 102
    ld8, 107
    ldSZ, 106
loop, 112
mov, 108
movl, 108
return, 112
shl, 110
shladd, 109
shr, 110
shr.u, 110
speculative load, 107
stSZ, 107

Itanium processor, 97–119
addressing modes, 100
architectural features, 98
architecture, 98
arithmetic instructions, 108
branch elimination, 112, 113
branch handling, 112–114
branch hints, 119
branch instructions, 111
branch prediction, 119
branch speedup, 112
comparison instructions, 110
data dependency, 115
data transfer instructions, 106
EPIC design, 97
immediate addressing mode, 100
index addressing mode, 100
indirect addressing mode, 100
instruction bundles, 106
instruction format, 102
instruction-level parallelism, 97, 105–106
logical instructions, 109
Not-a-Thing bit, 98
predication, 113
procedure call, 101
register renaming, 101
registers, 98
shift instructions, 110
speculative execution, 114–119
control speculation, 118
data speculation, 115

stack frame, 101

L
linear search, 256
load instructions, 166
load/store architecture, 20, 100
logical instructions, 35
LR register, 82

M
machine language, 4, 7
masking bit, 271
merge sort, 372
microprogram, 6
MIPS, 47–52
addressing modes, 50
architecture, 47–52
at register use, 229
instruction format, 51
memory layout, 52
recursion, 292–303
MIPS instructions
abs, 227, 343
add, 169, 226, 344
addi, 226, 344
addiu, 344
addu, 169, 226, 344
and, 271, 272, 344
andi, 272, 345
b, 171, 246, 249, 345
bczf, 345
bczt, 345
beq, 171, 248, 249, 345
beqz, 172, 248, 250, 346
bge, 249, 346
bgeu, 249, 346
bgez, 227, 250, 346
bgezl, 346
bgt, 248, 249, 347
bgtu, 248, 249, 347
bgtz, 250, 347
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ble</td>
<td>249</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>bleu</td>
<td>249</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>blez</td>
<td>250</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>blt</td>
<td>249</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>bltu</td>
<td>249</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>bltz</td>
<td>250</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>bltzal</td>
<td></td>
<td></td>
<td>348</td>
</tr>
<tr>
<td>bne</td>
<td>249</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>250</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td></td>
<td></td>
<td>246</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>229</td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>comparison</td>
<td></td>
<td></td>
<td>244</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
<td>165</td>
</tr>
<tr>
<td>transfer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>229</td>
<td>349</td>
<td>350</td>
</tr>
<tr>
<td>divu</td>
<td>349</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>246</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>186</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>jalr</td>
<td>350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td>188</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td></td>
<td></td>
<td>246</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>la</td>
<td>166</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>lb</td>
<td>167</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>lbu</td>
<td>166</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>351</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lh</td>
<td>166</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>lhu</td>
<td>166</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>166</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
<td></td>
<td>166</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lui</td>
<td>352</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>166</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>lwcz</td>
<td>352</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw1</td>
<td>353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lwrc</td>
<td>353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfcz</td>
<td>353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfhi</td>
<td>228</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>mflo</td>
<td>228</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>move</td>
<td>166</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>mtcz</td>
<td>354</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mthi</td>
<td>228</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>mtlo</td>
<td>228</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>228</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>mulo</td>
<td>228</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>mulou</td>
<td>228</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>228</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>multu</td>
<td></td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>neg</td>
<td>227</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>negu</td>
<td>356</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>356</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>272</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>272</td>
<td>276</td>
<td>356</td>
</tr>
<tr>
<td>or</td>
<td>272</td>
<td>273</td>
<td>356</td>
</tr>
<tr>
<td>ori</td>
<td>167</td>
<td>272</td>
<td>357</td>
</tr>
<tr>
<td>rem</td>
<td>230</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>remu</td>
<td>357</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rol</td>
<td>282</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>ror</td>
<td>281</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>rotate</td>
<td></td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>167</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>sd</td>
<td>358</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>245</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>sge</td>
<td>245</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>sgeu</td>
<td>245</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>sgt</td>
<td>245</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>sgtu</td>
<td>245</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>sh</td>
<td>168</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>shift</td>
<td></td>
<td></td>
<td>170</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td>276</td>
</tr>
<tr>
<td>sle</td>
<td>245</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>sleu</td>
<td>245</td>
<td>359</td>
<td></td>
</tr>
<tr>
<td>sl1</td>
<td>170</td>
<td>278</td>
<td>360</td>
</tr>
<tr>
<td>sl1v</td>
<td>276</td>
<td>278</td>
<td>360</td>
</tr>
<tr>
<td>sll</td>
<td>244</td>
<td>245</td>
<td>360</td>
</tr>
<tr>
<td>slt</td>
<td>244</td>
<td>245</td>
<td>360</td>
</tr>
<tr>
<td>slti</td>
<td>245</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td>360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sltu</td>
<td>245</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>sne</td>
<td>245</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>sra</td>
<td>278</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>srav</td>
<td>278</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>srl</td>
<td>278</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>srlv</td>
<td>278</td>
<td>362</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>169</td>
<td>226</td>
<td>362</td>
</tr>
<tr>
<td>subu</td>
<td>169</td>
<td>227</td>
<td>362</td>
</tr>
<tr>
<td>sw</td>
<td>168</td>
<td>362</td>
<td></td>
</tr>
<tr>
<td>swcz</td>
<td>362</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
swl, 363
swr, 363
ulh, 363
ulhu, 364
ulw, 364
ush, 364
usw, 364
xor, 272, 364
xori, 272, 364

MIPS processor
addressing modes, 212
multidimensional arrays, 215

N
number of addresses, 14–22
number representation
floating-point, 334–336
signed integer, 331
1’s complement, 332
2’s complement, 333
excess-M, 331
signed magnitude, 331
unsigned integer, 330
number systems, 325
base, 325
binary, 325, 327
conversion, 327–329
decimal, 325, 327
floating-point, 334–336
hexadecimal, 325, 327
notation, 326
octal, 325, 327
radix, 325

O
octal numbers, 327
one’s complement, 332
one-dimensional arrays, 214
opcode, 35, 36
operand types, 32

P
parameter passing, 27, 71, 184
call-by-reference, 184
call-by-value, 184
variable number of parameters, 207–210
parity conversion, 275
PC-relative, 23
PC-relative address, 247
PowerPC instructions
xor, 275
pipelining
branch prediction, 28–32
PowerPC, 79–95
addressing modes, 83
immediate index addressing mode, 83
immediate update index addressing mode, 84
index addressing mode, 84
instruction set, 86–95
instruction format, 84
registers, 81
condition register, 81
CTR register, 83
FPSCR register, 81
LR register, 82
XER register, 83
PowerPC instructions, 86–95
add, 88
add., 88
adde, 88
adi, 89
addition instructions, 88
addo, 88
addo., 88
and, 90
andc, 90
andi., 90
arithmetic instructions, 88
b, 94
ba, 94
bc, 94
bca, 94
bcctr, 95
bcctrl, 95
bcl, 94
bcla, 94
bclr, 95
bclrl, 95
bl, 94
bla, 94
branch conditions, 94
branch hints, 94, 95
branch instructions, 93
cmp, 93
cmpd, 93
cmpi, 93
comparison instructions, 93
data transfer instructions, 86
divide instructions, 90
divw, 90
instruction format, 84
la, 89
lbz, 86
lbzu, 86
lbzux, 86
lbzx, 86
lha, 87
lhau, 87
lhaux, 87
lhaux, 87
lhax, 87
li, 89
logical instructions, 90
mr, 91
mulhw, 90
mulli, 90
mullw, 89
multiply instructions, 89
nand, 91
nand, 91
neg, 89
nor, 91
nor, 91
ori, 91
rlwnm, 92
rotate instructions, 91
rotlw, 92
shift instructions, 91
slw, 91, 92
sraw, 92
sraw, 92
srawi, 92
srawi, 92
stb, 87
stbu, 87
stbux, 87
stbx, 87
stmu, 88
subf, 89
subi, 89
subtract instructions, 89
xor, 91
xor, 91
predicated execution, 113
procedure call, 26
delayed call, 27
end, 26
in Itanium, 101
overview, 26
parameter passing, 27
return address, 26
processor registers, 22, 48
program status registers, 123

Q
quicksort
algorithm, 299
MIPS procedure, 298

R
recursion, 291–303
activation record, 292
factorial, 291, 292
Fibonacci function, 295
Fibonacci number, 303
in MIPS, 292–303
  factorial procedure, 292
  quicksort procedure, 298
  quicksort algorithm, 298, 299
  versus iteration, 303
register addressing mode, 33
register renaming, 101
registers, 22
return address, 26
RISC processors, 41–44
  characteristics, 41–42
  design principles, 43–44
  PowerPC, 79–95
row-major order, 215

S
  selection sort, 252
  sign bit, 331
  sign extension, 334
  signed integer, 331
    1’s complement, 332
    2’s complement, 333
    excess-M, 331
    signed magnitude representation, 331
  signed magnitude representation, 331
  space-efficiency, 10
  SPARC, 55–76
    addressing modes, 58
    instruction format, 59
    instruction set, 59–76
    parameter passing, 71
    stack implementation, 72
    window management, 72
  SPARC instructions, 59–76
    add instructions, 63
    arithmetic instructions, 62
    branch instructions, 65
    compare instructions, 65
    data transfer instructions, 59
    division instructions, 64
    logical instructions, 64
    multiplication instructions, 63
    procedure calls, 69
    procedure instructions, 70
    shift instructions, 64
    subtract instructions, 63
  speculative execution, 114–119
    control speculation, 118
    data speculation, 115
  speculative load, 107
  SPIM, 149–157
    assembler directives, 162
    data directives, 163
    debugging, 154
    loading, 153
    miscellaneous directives, 163
    running, 153
    segments, 162
    simulator settings, 152
    string directives, 162
    system calls, 161
  stack, 192–193
    MIPS implementation, 193
    SPARC implementation, 72
    stack frame, 101
    top-of-stack, 192, 193
    use, 195
    what is it, 192
    stack depth, 18
    stack frame, 101
    stack machines, 17
      stack depth, 18
    static branch prediction strategy, 29
    string directives, 162
    system calls, 161

T
  time-efficiency, 10
  top-of-stack, 192, 193
  towers of Hanoi, 370
  two’s complement, 333
U
unconditional branch, 23
unsigned integer representation, 330

V
variable number of parameters, 207–210
VAX-11/780, 40, 43

W
window management, 72

X
XER register, 83

Z
zero extension, 330