2

NUMERIC REPRESENTATION

High-level languages shield programmers from the pain of dealing with low-level numeric representation. Writing great code, however, requires that you understand how computers represent numbers, so that is the focus of this chapter. Once you understand internal numeric representation, you'll discover efficient ways to implement many algorithms and avoid the pitfalls associated with common programming practices.

2.1 What Is a Number?

Having taught assembly language programming for many years, I've discovered that most people don't understand the fundamental difference between a number and the representation of that number. Most of the time, this confusion is harmless. However, many algorithms depend on the internal and external representations we use for numbers to operate correctly and efficiently. If you don't understand the difference between the abstract concept of a number and the representation of that number, you'll have trouble understanding, using, or creating such algorithms.

A *number* is an intangible, abstract concept. It is an intellectual device that we use to denote quantity. Let's say I told you that a book has one hundred pages. You could touch the pages—they are tangible. You could even count those pages to verify that there are one hundred of them. However, "one hundred" is simply an abstraction I'm applying to the book as a way of describing its size.

The important thing to realize is that the following is *not* one hundred:

100

This is nothing more than ink on paper forming certain lines and curves (called *glyphs*). You might recognize this sequence of symbols as a representation of one hundred, but this is not the actual value 100. It's just three symbols on this page. It isn't even the only representation for one hundred—consider the following, which are all different representations of the value 100:

100	Decimal representation
C	Roman numeral representation
64_{16}	Base-16 (hexadecimal) representation
1100100_2	Base-2 (binary) representation
144_8	Base-8 (octal) representation
one hundred	English representation

The representation of a number is (generally) some sequence of symbols. For example, the common representation of the value one hundred, "100," is really a sequence of three numeric digits: the digit 1 followed by the digit 0 followed by a second 0 digit. Each of these digits has some specific meaning, but we could have just as easily used the sequence "64" to represent one hundred. Even the individual digits that make up this representation of 100 are not numbers. They are numeric digits, tools we use to represent numbers, but they are not numbers themselves.

Now you might be wondering why you should even care whether a sequence of symbols like "100" is the actual value one hundred or just the representation of it. The reason is that you'll encounter several different sequences of symbols in a computer program that look like numbers (that is, they look like "100"), and you don't want to confuse them with actual numeric values. Conversely, there are many different representations for the value one hundred that a computer could use, and it's important for you to realize that they are equivalent.

2.2 Numbering Systems

A *numbering system* is a mechanism we use to represent numeric values. Today, most people use the *decimal* (or *base-10*) numbering system, and most computer systems use the *binary* (or *base-2*) numbering system. Confusion between the two can lead to poor coding practices.

The Arabs developed the decimal numbering system we commonly use today (this is why the 10 decimal digits are known as *Arabic numerals*). The decimal system uses *positional notation* to represent values with a small group of different symbols. Positional notation gives meaning not only to the symbol itself, but also to the position of the symbol in the sequence of symbols—a scheme that is far superior to other, nonpositional, representations. To appreciate the difference between a positional system and a nonpositional system, consider the *tallyslash* representation of the number 25 in Figure 2-1.

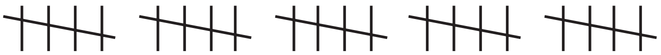
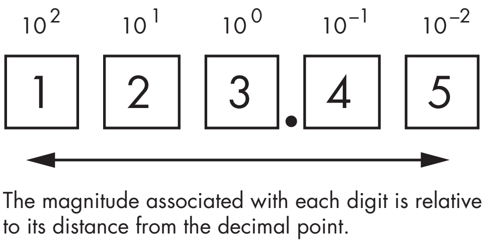


Figure 2-1: Tally-slash representation of 25

The tally-slash representation uses a sequence of *n* marks to represent the value *n*. To make the values easier to read, most people arrange the tally marks in groups of five, as in Figure 2-1. The advantage of the tally-slash numbering system is that it's easy to use for counting objects. However, the notation is bulky, and arithmetic operations are difficult. The biggest problem with the tally-slash representation is the amount of physical space it consumes. To represent the value *n* requires an amount of space proportional to *n*. Therefore, for large values of *n*, this notation becomes unusable.

2.2.1 The Decimal Positional Numbering System

The decimal positional numbering system represents numbers using strings of Arabic numerals, optionally including a decimal point to separate whole and fractional portions of the number representation. The position of a digit in the string affects its meaning: each digit to the left of the decimal point represents a value between 0 and 9, multiplied by an increasing power of 10 (see Figure 2-2). The symbol immediately to the left of the decimal point in the sequence represents a value between 0 and 9. If there are at least two digits, the second symbol to the left of the decimal point represents a value between 0 and 9 times 10, and so forth. To the right of the decimal point, the values decrease.



10^2
10^1
10^0
10^−1
10^−2
1
2
3
.
4
5
The magnitude associated with each digit is relative to its distance from the decimal point.

Figure 2-2: A positional numbering system

The numeric sequence 123.45 represents:

$$(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (4 \times 10^{-1}) + (5 \times 10^{-2})$$

or:

$$100 + 20 + 3 + 0.4 + 0.05$$

To understand the power of the base-10 positional numbering system, consider that, compared to the tally-slash system:

- It can represent the value 10 in one-third the space.
- It can represent the value 100 in about 3 percent of the space.
- It can represent the value 1,000 in about 0.3 percent of the space.

As the numbers grow larger, the disparity becomes even greater. Because of their compact and easy-to-recognize notation, positional numbering systems are quite popular.

2.2.2 Radix (Base) Values

Humans developed the decimal numbering system because it corresponds to the number of fingers ("digits") on their hands. However, decimal isn't the only positional numbering system possible; in fact, for most computer-based applications, it isn't even the best numbering system available. So, let's take a look at how to represent values in other numbering systems.

The decimal positional numbering system uses powers of 10 and 10 unique symbols for each digit position. Because decimal numbers use powers of 10, we call them "base-10" numbers. By substituting a different set of numeric digits and multiplying those digits by powers of some base other than 10, we can devise a different numbering system. The base, or *radix*, is the value that we raise to successive powers for each digit to the left of the *radix point* (note that the term *decimal point* applies only to decimal numbers).

As an example, we can create a base-8 (*octal*) numbering system using eight symbols (0–7) and successive powers of 8. Consider the octal number 123_8 (the subscript denotes the base using standard mathematical notation), which is equivalent to 83_{10}:

$$1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0$$

or:

$$64 + 16 + 3$$

To create a base-n numbering system, you need n unique digits. The smallest possible radix is 2 (for this scheme). For bases 2 through 10, the convention is to use the Arabic digits 0 through $n - 1$ (for a base-n system). For bases greater than 10, the convention is to use the alphabetic digits *a* through *z* or *A* through *Z* (ignoring case) for digits greater than 9. This scheme supports numbering systems through base 36 (10 numeric digits and 26 alphabetic digits). There's no agreed-upon convention for symbols beyond the 10 Arabic numeric digits and the 26 alphabetic digits. Throughout this book, we'll deal with base-2, base-8, and base-16 values because base 2 (binary) is the native representation most computers use, base 8 was popular on older computer systems, and base 16 is more compact than base 2. You'll find that many programs use these three bases, so it's important to be familiar with them.

2.2.3 The Binary Numbering System

Since you're reading this book, chances are pretty good that you're already familiar with the base-2, or binary, numbering system; nevertheless, a quick review is in order. The binary numbering system works just like the decimal numbering system, except binary uses only the digits 0 and 1 (rather than 0–9) and uses powers of 2 (rather than powers of 10).

Why even worry about binary? After all, almost every computer language available allows programmers to use decimal notation (automatically converting decimal representation to the internal binary representation). Despite this capability, most modern computer systems talk to I/O devices using binary, and their arithmetic circuitry operates on binary data. Many algorithms depend upon binary representation for correct operation. In order to write great code, then, you'll need a complete understanding of binary representation.

2.2.3.1 Converting Between Decimal and Binary Representation

To appreciate what the computer does for you, it's useful to learn how to convert between decimal and binary representations manually.

To convert a binary value to decimal, add 2^i for each 1 in the binary string, where i is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

or:

$$128 + 64 + 8 + 2$$

or:

$$202_{10}$$

Converting decimal to binary is almost as easy. Here's an algorithm that converts decimal representation to the corresponding binary representation:

1. If the number is even, emit a 0. If the number is odd, emit a 1.
2. Divide the number by 2 and discard any fractional component or remainder.
3. If the quotient is 0, the algorithm is complete.
4. If the quotient is not 0 and the number is odd, insert a 1 before the current string. If the quotient is not 0 and the number is even, prefix your binary string with 0.
5. Go back to step 2 and repeat.

This example converts 202 to binary:

1. 202 is even, so emit a 0 and divide by 2 (101): 0
2. 101 is odd, so emit a 1 and divide by 2 (50): 10
3. 50 is even, so emit a 0 and divide by 2 (25): 010

4. 25 is odd, so emit a 1 and divide by 2 (12): 1010
5. 12 is even, so emit a 0 and divide by 2 (6): 01010
6. 6 is even, so emit a 0 and divide by 2 (3): 001010
7. 3 is odd, so emit a 1 and divide by 2 (1): 1001010
8. 1 is odd, so emit a 2 and divide by 2 (0): 11001010
9. The result is 0, so the algorithm is complete, producing 11001010.

2.2.3.2 Making Binary Numbers Easier to Read

As you can tell by the equivalent representations 202_{10} and 11001010_2, binary representation is not as compact as decimal representation. We need some way to make the digits, or *bits*, in binary numbers less bulky and easier to read.

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. This book will adopt a similar convention for binary numbers; each group of 4 binary bits will be separated with an underscore. For example, the binary value 1010111110110010_2 will be written as $1010\_1111\_1011\_0010_2$.

2.2.3.3 Representing Binary Values in Programming Languages

Thus far, this chapter has used the subscript notation embraced by mathematicians to denote binary values (the lack of a subscript indicates the decimal base). Subscripts are not generally recognized by program text editors or programming language compilers, however, so we need some other way to represent various bases within a standard ASCII text file.

Generally, only assembly language compilers ("assemblers") allow the use of literal binary constants in a program.[1] Because assemblers vary widely, there are many different ways to represent binary literal constants in an assembly language program. This book presents examples using MASM and HLA, so it makes sense to adopt their conventions.

MASM represents binary values as a sequence of binary digits (`0` and `1`) ending with a `b` or `B`. The binary representation for 9 would be `1001b` in a MASM source file.

HLA prefixes binary values with the percent symbol (`%`). To make binary numbers more readable, HLA also allows you to insert underscores within binary strings like so:

```
%11_1011_0010_1101
```

1. Swift also allows you to specify binary numbers, using a `0b` prefix.

2.2.4 The Hexadecimal Numbering System

As noted earlier, binary number representation is verbose. Hexadecimal representation offers two great features: it's very compact, and it's easy to convert between binary and hexadecimal. Therefore, software engineers generally use hexadecimal representation rather than binary to make their programs more readable.

Because hexadecimal representation is base 16, each digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

$$1 \times 16^3 + 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0$$

or:

$$4096 + 512 + 48 + 4$$

or:

$$4660_{10}$$

Hexadecimal representation uses the letters *A* through *F* for the additional six digits it requires (above the 10 standard decimal digits, 0–9). The following are all examples of valid hexadecimal numbers:

$$234_{16} \quad \text{DEAD}_{16} \quad \text{BEEF}_{16} \quad \text{0AFB}_{16} \quad \text{FEED}_{16} \quad \text{DEAF}_{16}$$

2.2.4.1 Representing Hexadecimal Values in Programming Languages

One problem with hexadecimal representation is that it's difficult to differentiate hexadecimal values like "DEAD" from standard program identifiers. Therefore, most programming languages use a special prefix or suffix character to denote hexadecimal values. Here's how you specify literal hexadecimal constants in several popular languages:

- The C, C++, C#, Java, Swift, and other C-derivative programming languages use the prefix `0x`. You'd use the character sequence `0xdead` for the hexadecimal value DEAD_{16}.
- The MASM assembler uses an `h` or `H` suffix. Because this doesn't completely resolve the ambiguity between certain identifiers and literal hexadecimal constants (for example, "deadh" still looks like an identifier to MASM), it also requires that a hexadecimal value begin with a numeric digit. So, you would add `0` to the beginning of the value (because a prefix of 0 does not alter the value of a numeric representation) to get `0deadh`, which unambiguously represents DEAD_{16}.
- Visual Basic uses the `&H` or `&h` prefix. Continuing with the current example, you'd use `&Hdead` to represent DEAD_{16} in Visual Basic.
- Pascal (Delphi) uses the prefix `$`. So, you'd use `$dead` to represent the current example in Delphi/Free Pascal.
- HLA also uses the prefix `$`. As with binary numbers, it also allows you to insert underscores into a hexadecimal number to make it easier to read (for example, `$FDEC_A012`).

In general, this book will use the HLA/Delphi/Free Pascal format except in examples specific to other programming language. Because there are several C/C++ examples in this book, you'll frequently see the C/C++ notation as well.

2.2.4.2 Converting Between Hexadecimal and Binary Representations

Another reason hexadecimal notation is popular is because it's easy to convert between the binary and hexadecimal representations. By memorizing the few simple rules shown in Table 2-1, you can mentally perform this conversion.

Table 2-1: Binary/Hexadecimal Conversion Chart

Binary	Hexadecimal
%0000	$0
%0001	$1
%0010	$2
%0011	$3
%0100	$4
%0101	$5
%0110	$6
%0111	$7
%1000	$8
%1001	$9
%1010	$A
%1011	$B
%1100	$C
%1101	$D
%1110	$E
%1111	$F

To convert the hexadecimal representation of a number into binary, substitute the corresponding 4 bits for each hexadecimal digit. For example, to convert `$ABCD` into the binary form `%1010_1011_1100_1101`, convert each hexadecimal digit according to the values in Table 2-1:

A	B	C	D	Hexadecimal
1010	1011	1100	1101	Binary

Converting the binary representation of a number into hexadecimal is almost as easy. First, pad the binary number with 0s to make sure it is a multiple of 4 bits long. For example, given the binary number 1011001010, add two 0 bits to the left of the number to make it 12 bits without changing its value: 001011001010. Next, separate the binary value into groups of 4 bits: 0010_1100_1010. Finally, look up these binary values in Table 2-1 and substitute the appropriate hexadecimal digits: `$2CA`. As you can see, this is much simpler than converting between decimal and binary or between decimal and hexadecimal.

2.2.5 The Octal Numbering System

Octal (base-8) representation was common in early computer systems, so you might still see it in use now and then. Octal is great for 12-bit and 36-bit computer systems (or any other size that is a multiple of 3), but not particularly for computer systems whose bit size is a power of 2 (8-, 16-, 32-, and 64-bit computer systems). Nevertheless, some programming languages allow you to specify numeric values in octal notation, and you can still find some older Unix applications that use it.

2.2.5.1 Representing Octal Values in Programming Languages

The C programming language (and derivatives like C++ and Java), Visual Basic, and MASM support octal representation. You should be aware of the notation they use for octal numbers in case you come across it in programs written in these languages.

- In C, you specify the octal base by prefixing a numeric string with a `0`. For example, `0123` is equivalent to the decimal value 83_{10} and definitely *not* equivalent to the decimal value 123_{10}.
- MASM uses a `Q` or `q` suffix. (Microsoft/Intel probably chose *Q* because it looks like the letter *O* but isn't likely to be confused with a zero.)
- Swift uses a `0o` prefix. For example, `0o14` represents the decimal value 12_{10}.
- Visual Basic uses the prefix `&O` (that's the letter *O*, not a zero). For example, you'd use `&O123` to represent the decimal value 83_{10}.

2.2.5.2 Converting Between Octal and Binary Representation

Converting between binary and octal is similar to converting between binary and hexadecimal, except that you work in groups of 3 bits rather than 4. See Table 2-2 for the list of binary and octal equivalent representations.

Table 2-2: Binary/Octal Conversion Chart

Binary	Octal
%000	0
%001	1
%010	2
%011	3
%100	4
%101	5
%110	6
%111	7

To convert octal into binary, replace each octal digit in the number with the corresponding 3 bits from Table 2-2. For example, when you convert `123q` into a binary value, the final result is `%0_0101_0011`:

1	2	3
001	010	011

To convert a binary number into octal, you break up the binary string into groups of 3 bits (padding with 0s, as necessary) and then replace each triad with the corresponding octal digit from Table 2-2.

To convert an octal value to hexadecimal notation, convert the octal number to binary and then convert the binary value to hexadecimal.

2.3 Numeric/String Conversions

In this section, we'll explore conversions from string to numeric form and vice versa. Because most programming languages (or their libraries) perform these conversions automatically, beginning programmers are often unaware that they're even taking place. For example, consider how easy it is to convert a string to numeric form in various languages:

```
cin >> i;                      // C++
readln( i );                   // Pascal
let j = Int(readLine() ?? "")! // Swift
input i                        // BASIC
stdin.get(i);                  // HLA
```

In each of these statements, the variable `i` can hold some integer number. The input from the user's console, however, is a string of characters. The programming language's runtime library is responsible for converting that string of characters to the internal binary form the CPU requires. Note that Swift only allows you to read a string from the standard input; you must explicitly convert that string to an integer using the `Int()` constructor/type conversion function.

Unfortunately, if you have no idea of the cost of these statements, you won't realize how they can impact your program when performance is critical. It's important to understand the underlying work involved in the conversion algorithms so you won't frivolously use statements like these.

NOTE *For simplicity's sake, we'll discuss unsigned integer values and ignore the possibility of illegal characters and numeric overflow. Therefore, the following algorithms slightly understate the actual work involved.*

Use this algorithm to convert a string of decimal digits to an integer value:

1. Initialize a variable with `0`; this will hold the final value.
2. If there are no more digits in the string, then the algorithm is complete, and the variable holds the numeric value.

3. Fetch the next digit (moving from left to right) from the string and convert it from ASCII to an integer.
4. Multiply the variable by 10, and then add in the digit fetched in step 3.
5. Return to step 2 and repeat.

Converting an integer value to a string of characters takes even more effort:

1. Initialize a string to the empty string.
2. If the integer value is 0, output a `0`, and the algorithm is complete.
3. Divide the current integer value by 10, computing the remainder and quotient.
4. Convert the remainder (always in the range 0..9[2]) to a character, and insert the character at the beginning of the string.
5. If the quotient is not 0, make it the new value and repeat steps 3–5.
6. Output the characters in the string.

The particulars of these algorithms are not important. What *is* important is that these steps execute once for each output character and division is very slow. So, a simple statement like one of the following can hide a fair amount of work from the programmer:

```
printf( "%d", i );    // C
cout << i;            // C++
print i               // BASIC
write( i );           // Pascal
print( i)             // Swift
stdout.put( i );      // HLA
```

To write great code, you don't need to avoid using numeric/string conversions altogether; however, a great programmer will take care to use them only as necessary.

Remember that these algorithms are valid only for unsigned integers. Signed integers require a little more effort to process (though the extra work is almost negligible). Floating-point values, however, are far more difficult to convert between string and numeric form, so keep that in mind when writing code that uses floating-point arithmetic.

2.4 Internal Numeric Representation

Most modern computer systems use an internal binary format to represent values and other objects. However, most systems can only efficiently represent binary values of a given size. In order to write great code, you need to make sure that your programs use data objects that the machine can

2. The ".." notation, taken from Pascal and other programming languages, denotes a range of values. Thus, "0..9" denotes all integer values between 0 and 9.

represent efficiently. This section will describe how computers physically represent values so you can design your programs accordingly.

2.4.1 Bits

The smallest unit of data on a binary computer is a single bit. Because a bit can represent only two different values (typically `0` or `1`), you might assume that you can't use it for much. But in fact, there's an infinite number of two-item combinations you can represent with a single bit. Here are some examples (with arbitrary binary encodings I've created):

- Zero (`0`) or one (`1`)
- False (`0`) or true (`1`)
- Off (`0`) or on (`1`)
- Male (`0`) or female (`1`)
- Wrong (`0`) or right (`1`)

You're not limited to representing binary data types, either (that is, those objects that have only two distinct values). You could also use a single bit to represent any two distinct items:

- The numbers 723 (`0`) and 1,245 (`1`)
- The colors red (`0`) and blue (`1`)

You could even represent two unrelated objects with a single bit. For example, you could use the bit value `0` to represent the color red and the bit value `1` to represent the number 3,256. You can represent *any* two different values with a single bit—but *only* two different values. Therefore, individual bits aren't sufficient for most computational needs. To overcome the limitations of a single bit, we create *bit strings* from a sequence of multiple bits.

2.4.2 Bit Strings

By combining bits into a sequence, we can form binary representations that are equivalent to other representations of numbers, like hexadecimal and octal. Most computer systems don't let you combine an arbitrary number of bits, so you have to work with bit strings of certain fixed lengths.

A *nibble* is a collection of 4 bits. Most computer systems don't provide efficient access to nibbles in memory. Notably, it takes exactly 1 nibble to represent a single hexadecimal digit.

A *byte* is 8 bits and is the smallest addressable data item on many CPUs; that is, the CPU can efficiently retrieve data in groups of 8 bits from memory. For this reason, the smallest data type that many languages support consumes 1 byte of memory (regardless of the actual number of bits the data type requires).

Because the byte is the smallest unit of storage on most machines, and many languages use bytes to represent objects that require fewer than 8 bits, we need some way of denoting individual bits within a byte. To describe the bits within a byte, we'll use *bit numbers*. As Figure 2-3 shows, bit 0 is the *low-order (LO)*, or *least significant*, bit, and bit 7 is the *high-order (HO)*, or *most significant*, bit of the byte. We'll refer to all other bits by their number.

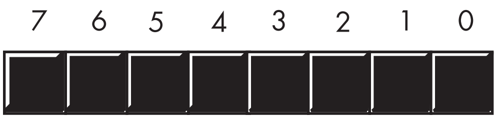


Figure 2-3: Bit numbering in a byte

A *word* is defined differently depending on the CPU: it may be a 16-bit, 32-bit, or 64-bit object. This book adopts the 80x86 terminology and defines a word as a collection of 16 bits. As with bytes, we'll use bit numbers for a word, starting with bit number 0 for the LO bit and working our way up to bit 15, the HO bit (see Figure 2-4).

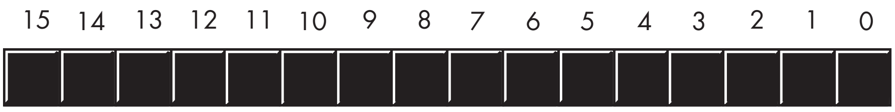


Figure 2-4: Bit numbers in a word

Notice that a word contains exactly 2 bytes. Bits 0 through 7 form the LO byte, and bits 8 through 15 form the HO byte (see Figure 2-5).

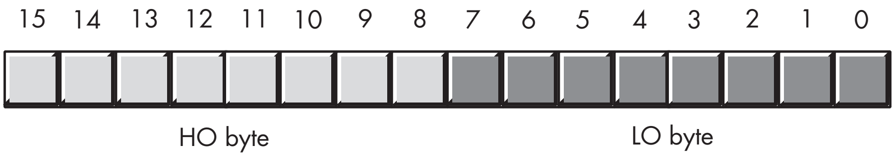


Figure 2-5: The 2 bytes in a word

A *double word* (or *dword*) is exactly what its name implies—a pair of words. Therefore, a double-word quantity is 32 bits long, as shown in Figure 2-6.

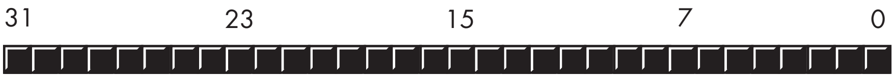


Figure 2-6: Bit layout in a double word

Figure 2-7 shows that a double word comprises 2 words or 4 bytes.

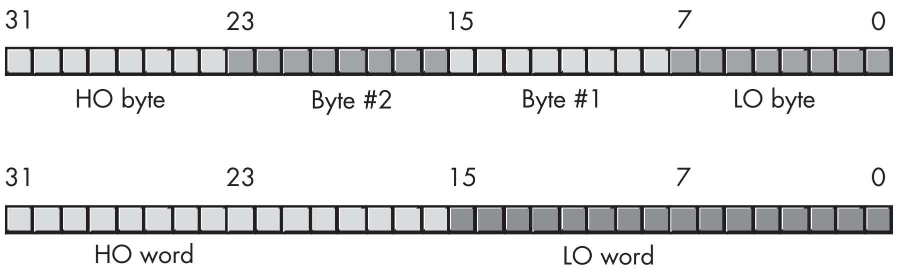


Figure 2-7: Bytes and words in a double word

As noted, most CPUs efficiently handle objects up to a certain size (typically 32 or 64 bits on contemporary systems). That doesn't mean you can't work with larger objects, simply that it's less efficient to do so. You typically won't see programs handling numeric objects much larger than about 128 or 256 bits. Some programming languages make 64-bit integers available, and most languages support 64-bit floating-point values, so for these data types we'll use the term *quad word*. Finally, we'll use *long word* to describe 128-bit values; although few languages today support them,[3] this gives us some room to grow.

We can break down quad words into 2 double words, 4 words, 8 bytes, or 16 nibbles. Likewise, we can break down long words into 2 quad words, 4 double words, 8 words, or 16 bytes.

Intel 80x86 platforms also support an 80-bit type that Intel calls a *tbyte* (short for "ten byte") object. The 80x86 CPU family uses tbyte variables to hold extended precision floating-point values and certain binary-coded decimal (BCD) values.

In general, with an *n*-bit string you can represent up to 2^n different values. Table 2-3 shows the number of possible objects you can represent with nibbles, bytes, words, double words, quad words, and long words.

Table 2-3: Number of Values Representable with Bit Strings

Size of bit string (in bits)	Number of possible combinations (2n)
4	16
8	256
16	65,536
32	4,294,967,296
64	18,446,744,073,709,551,616
128	340,282,366,920,938,463,463,374,607,431,768,211,456

3. HLA supports 128-bit values.

2.5 Signed and Unsigned Numbers

The binary number 0...00000[4] represents 0; 0...00001 represents 1; 0...00010 represents 2; and so on toward infinity. But what about negative numbers? To represent signed values, most computer systems use the *two's complement* numbering system. The representation of signed numbers places some fundamental restrictions on them, so it's important that you understand how signed and unsigned numbers are represented differently in a computer system in order to use them efficiently.

With n bits, we can represent only 2^n different objects. Because negative values are objects in their own right, we'll have to divide these 2^n combinations between negative and non-negative values. So, for example, a byte can represent the negative values -128 through -1 and the non-negative values 0 to 127. With a 16-bit word, we can represent signed values in the range -32,768 to +32,767. With a 32-bit double word, we can represent values in the range -2,147,483,648 to +2,147,483,647. In general, with n bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1} - 1$.

The two's complement system uses the HO bit as a *sign bit*. If the HO bit is `0`, the number is non-negative and has the usual binary encoding; if the HO bit is `1`, the number is negative and uses the two's complement encoding. Here are some examples using 16-bit numbers:

- `$8000` (`%1000_0000_0000_0000`) is negative because the HO bit is `1`.
- `$100` (`%0000_0001_0000_0000`) is non-negative because the HO bit is `0`.
- `$7FFF` (`%0111_1111_1111_1111`) is non-negative.
- `$FFFF` (`%1111_1111_1111_1111`) is negative.
- `$FFF` (`%0000_1111_1111_1111`) is non-negative.

To negate a number, you can use the two's complement operation as follows:

1. Invert all the bits in the number; that is, change all the `0`s to `1`s and vice versa.
2. Add `1` to the inverted result (ignoring any overflow).

If the result is negative (has its HO bit set), then this is the two's complement form of the non-negative value.

For example, these are the steps to compute the 8-bit equivalent of the decimal value –5:

1. `%0000_0101` 5 (in binary).
2. `%1111_1010` Invert all the bits.
3. `%1111_1011` Add 1 to obtain –5 (in two's complement form).

4. The ellipses (. . .) have the standard mathematical meaning: repeat a string of zeros an indefinite number of times.

If we take −5 and negate it, the result is 5 (`%0000_0101`), just as we expect:

1. `%1111_1011` Two's complement for −5.
2. `%0000_0100` Invert all the bits.
3. `%0000_0101` Add 1 to obtain 5 (in binary).

Let's look at some 16-bit examples and their negations.
First, negate 32,767 (`$7FFF`):

1. `%0111_1111_1111_1111` +32,767, the largest 16-bit positive number.
2. `%1000_0000_0000_0000` Invert all the bits (`8000h`).
3. `%1000_0000_0000_0001` Add 1 (`8001h`, or -32,767).

Now negate 16,384 (`$4000`):

1. `%0100_0000_0000_0000` 16,384.
2. `%1011_1111_1111_1111` Invert all the bits (`$BFFF`).
3. `%1100_0000_0000_0000` Add 1 (`$C000` or -16,384).

And now negate -32,768 (`$8000`):

1. `%1000_0000_0000_0000` -32,768, the smallest 16-bit negative number.
2. `%0111_1111_1111_1111` Invert all the bits (`$7FFF`).
3. `%1000_0000_0000_0000` Add 1 (`$8000` or -32768).

`$8000` inverted becomes `$7FFF`, and after adding 1 we obtain `$8000`! Wait, what's going on here: −(-32,768) is -32,768? Of course not. However, the 16bit two's complement numbering system cannot represent the value +32,768. In general, you cannot negate the smallest negative value in the two's complement numbering system.

2.6 Useful Properties of Binary Numbers

Here are some properties of binary values that you might find useful in your programs:

- If bit position 0 of a binary (integer) value contains `1`, the number is an odd number; if this bit contains `0`, then the number is even.
- If the LO n bits of a binary number all contain `0`, then the number is evenly divisible by 2^n.
- If a binary value contains a `1` in bit position n, and `0`s everywhere else, then that number is equal to 2^n.
- If a binary value contains all `1`s from bit position 0 up to (but not including) bit position n, and all other bits are `0`, then that value is equal to $2^n - 1$.

- Shifting all the bits in a number to the left by one position multiplies the binary value by 2.
- Shifting all the bits of an unsigned binary number to the right by one position effectively divides that number by 2 (this does not apply to signed integer values). Odd numbers are rounded down.
- Multiplying two *n*-bit binary values together may require as many as 2 × *n* bits to hold the result.
- Adding or subtracting two *n*-bit binary values never requires more than *n* + 1 bits to hold the result.
- Inverting all the bits in a binary number (that is, changing all the `0`s to `1`s and vice versa) is the same thing as negating (changing the sign) of the value and then subtracting 1 from the result.
- *Incrementing* (adding 1 to) the largest unsigned binary value for a given number of bits always produces a value of `0`.
- *Decrementing* (subtracting 1 from) 0 always produces the largest unsigned binary value for a given number of bits.
- An *n*-bit value provides 2^n unique combinations of those bits.
- The value 2^n-1 contains *n* bits, each containing the value `1`.

It's a good idea to memorize all the powers of 2 from 2^0 through 2^{16} (see Table 2-4), as these values come up in programs all the time.

Table 2-4: Powers of 2

n	**2^n**
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536

2.7 Sign Extension, Zero Extension, and Contraction

With the two's complement system, a single negative value is represented differently depending on the size of the representation. An 8-bit signed value must be converted for use in an expression involving a 16bit number. This conversion and its converse—converting a 16-bit value to 8 bits—are the *sign extension* and *contraction* operations, respectively.

Consider the value −64. The 8-bit two's complement value for this number is `$C0`. The 16-bit equivalent is `$FFC0`. Clearly, these are not the same bit pattern. Now consider the value +64. The 8- and 16-bit versions of this value are `$40` and `$0040`, respectively. We extend the size of negative values differently than we extend the size of non-negative values.

To *sign-extend* a value, copy the sign bit into the additional HO bits in the new format. For example, to sign-extend an 8-bit number to a 16-bit number, copy bit 7 of the 8-bit number into bits 8 through 15 of the 16-bit number. To sign-extend a 16-bit number to a double word, copy bit 15 into bits 16 through 31 of the double word.

When adding a byte quantity to a word quantity, you need to sign extend the byte to 16 bits before adding the two numbers. Other operations may require a sign extension to 32 bits.

Table 2-5 provides several examples of sign extension.

Table 2-5: Sign Extension Examples

8 bits	16 bits	32 bits	Binary (two's complement)
$80	$FF80	$FFFF_FF80	%1111_1111_1111_1111_1111_1111_1000_0000
$28	$0028	$0000_0028	%0000_0000_0000_0000_0000_0000_0010_1000
$9A	$FF9A	$FFFF_FF9A	%1111_1111_1111_1111_1111_1111_1001_1010
$7F	$007F	$0000_007F	%0000_0000_0000_0000_0000_0000_0111_1111
n/a	$1020	$0000_1020	%0000_0000_0000_0000_0001_0000_0010_0000
n/a	$8086	$FFFF_8086	%1111_1111_1111_1111_1000_0000_1000_0110

Zero extension converts small unsigned values to larger unsigned values. Zero extension is very easy—just store `0`s in the HO byte(s) of the larger operand. For example, to zero-extend the 8-bit value `$82` to 16 bits, you insert a `0` for the HO byte, yielding `$0082`.

Further examples are listed in Table 2-6.

Table 2-6: Zero Extension Examples

8 bits	16 bits	32 bits	Binary
$80	$0080	$0000_0080	%0000_0000_0000_0000_0000_0000_1000_0000
$28	$0028	$0000_0028	%0000_0000_0000_0000_0000_0000_0010_1000
$9A	$009A	$0000_009A	%0000_0000_0000_0000_0000_0000_1001_1010
$7F	$007F	$0000_007F	%0000_0000_0000_0000_0000_0000_0111_1111
n/a	$1020	$0000_1020	%0000_0000_0000_0000_0001_0000_0010_0000
n/a	$8086	$0000_8086	%0000_0000_0000_0000_1000_0000_1000_0110

Many high-level language compilers automatically handle sign and zero extension. The following examples in C demonstrate how this works:

```
signed char sbyte;   // Chars in C are byte values.
short int sword;     // Short integers in C are *usually* 16-bit values.
long int sdword;     // Long integers in C are *usually* 32-bit values.
 . . .
sword = sbyte;       // Automatically sign-extends the 8-bit value to 16 bits.
sdword = sbyte;      // Automatically sign-extends the 8-bit value to 32 bits.
sdword = sword;      // Automatically sign-extends the 16-bit value to 32
bits.
```

Some languages (such as Ada or Swift) require an explicit cast from a smaller size to a larger size. Check the reference manual for your particular language to see if this is necessary. The advantage of a language that requires you to provide an explicit conversion is that the compiler never does anything behind your back. If you fail to do the conversion yourself, the compiler emits a diagnostic message.

The important thing to realize about sign and zero extension is that they aren't always free. Assigning a smaller integer to a larger integer may require more machine instructions (taking longer to execute) than moving data between two like-sized integer variables. Therefore, you should be careful about mixing variables of different sizes within the same arithmetic expression or assignment statement.

Sign contraction—converting a value with some number of bits to the same value with a fewer number of bits—is a little more troublesome. For example, consider the value −448. As a 16-bit hexadecimal number, its representation is `$FE40`. The magnitude of this number is too large to fit into 8 bits, so you can't sign-contract it to 8 bits.

To properly sign-contract one value to another, you must look at the HO byte(s) that you want to discard. First, the HO bytes must all contain either `0` or `$FF`. Second, the HO bit of your resulting value must match *every* bit you've removed from the number. Here are some examples of converting 16-bit values to 8-bit values (including a couple of failures):

- `$FF80` (`%1111_1111_1000_0000`) can be sign-contracted to `$80` (`%1000_0000`).
- `$0040` (`%0000_0000_0100_0000`) can be sign-contracted to `$40` (`%0100_0000`).
- `$FE40` (`%1111_1110_0100_0000`) cannot be sign-contracted to 8 bits.
- `$0100` (`%0000_0001_0000_0000`) cannot be sign-contracted to 8 bits.

Some high-level languages, like C, will simply store the LO portion of the expression into a smaller variable and discard the HO component—at best, the C compiler may give you a warning about the loss of precision that may occur. You can often quiet the compiler, but it still doesn't check for invalid values. Typically, you'd use code like the following to sign-contract a value in C:

```
signed char sbyte;    // Chars in C are byte values.
short int sword;      // Short integers in C are *usually* 16-bit values.
```

```
long int sdword;      // Long integers in C are *usually* 32-bit values.
 . . .
sbyte = (signed char) sword;
sbyte = (signed char) sdword;
sword = (short int) sdword;
```

The only safe solution in C is to compare the result of the expression to an upper- and lower-bound value before attempting to store the value into a smaller variable. Here's the preceding code with these checks in place:

```
if( sword >= -128 && sword <= 127 )
{
    sbyte = (signed char) sword;
}
else
{
    // Report appropriate error.
}

// Another way, using assertions:

assert( sword >= -128 && sword <= 127 )
sbyte = (signed char) sword;

assert( sdword >= -32768 && sdword <= 32767 )
sword = (short int) sdword;
```

This code gets pretty ugly. In C/C++, you'd probably want to turn this into a macro (`#define`) or a function so your code would be a bit more readable.

Some high-level languages (such as Free Pascal and Delphi) automatically sign-contract values and then check the value to ensure it fits in the destination operand.[5] Such languages raise some sort of exception (or stop the program) if a range violation occurs. To take corrective action, you'll either need to write some exception handling code or use an `if` statement sequence similar to the one in the C example just given.

2.8 Saturation

You can also reduce the size of an integer value through *saturation*, which is useful when you're willing to live with a possible loss of precision. To convert a value via saturation, you copy the LO bits of the larger object into the smaller object. If the larger value is outside the smaller object's range, then you *clip* the larger value by setting the smaller object to the largest (or smallest) value within the smaller value's range.

For example, when converting a 16-bit signed integer to an 8-bit signed integer, if the 16-bit value is in the range -128 through +127, you simply copy

5. Borland's compilers require the use of a special compiler directive to activate this check. By default, the compiler does not do the bounds check.

the LO byte into the 8bit object. If the 16-bit signed value is greater than +127, then you clip the value to +127 and store +127 into the 8-bit object. Likewise, if the value is less than -128, you clip the final 8-bit object to -128. Saturation works the same way when you clip 32-bit values to smaller values.

If the larger value is outside the range of the smaller value, there will be a loss of precision during the conversion. While clipping the value is never desirable, sometimes it's better than raising an exception or otherwise rejecting the calculation. For many applications, such as audio or video, the clipped result is still recognizable to the end user, so this is a reasonable conversion scheme.

Many CPUs support saturation arithmetic in their special "multimedia extension" instruction sets—for example, the MMX/SSE/AVX instruction extensions on the Intel 80x86 processor family. Most CPUs' standard instruction sets, as well as most high-level languages, do not provide direct support for saturation, but the technique is not difficult. Consider the following Free Pascal/Delphi code, which uses saturation to convert a 32-bit integer to a 16-bit integer:

```
var
    li  :longint;
    si  :smallint;
        . . .
    if( li > 32767 ) then

        si := 32767;

    else if( li < -32768 ) then

        si := -32768;

    else
        si := li;
```

2.9 Binary-Coded Decimal Representation

The *binary-coded decimal (BCD)* format, as its name suggests, encodes decimal values using a binary representation. Common general-purpose high-level languages (like C/C++, Pascal, and Java) rarely support decimal values. However, business-oriented programming languages (like COBOL and many database languages) do. So, if you're writing code that interfaces with a database or some language that supports decimal arithmetic, you may need to deal with BCD representation.

BCD values consist of a sequence of nibbles, with each nibble representing a value in the range 0 to 9. (The BCD format uses only 10 of the possible 16 values represented by a nibble.) With a byte we can represent values containing two decimal digits (0..99), as shown in Figure 2-8. With a word, we can represent four decimal digits (0..9999). A double word can represent up to eight decimal digits.

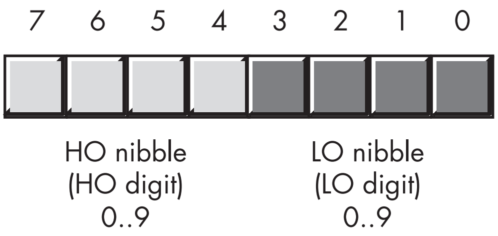


Figure 2-8: BCD data representation in a byte

An 8-bit BCD variable can represent values in the range 0 to 99, while that same 8 bits, holding a binary value, could represent values in the range 0 to 255. Likewise, a 16-bit binary value can represent values in the range 0 to 65535, while a 16-bit BCD value can represent only about a sixth of those values (0..9999). Inefficient storage isn't the only problem with BCD, though—BCD calculations also tend to be slower than binary calculations.

The BCD format does have two saving graces: it's very easy to convert BCD values between the internal numeric representation and their decimal string representations, and it's also very easy to encode multidigit decimal values in hardware when using BCD—for example, when using a set of dials, with each dial representing a single digit. For these reasons, you're likely to see people using BCD in embedded systems (such as toaster ovens and alarm clocks) but rarely in general-purpose computer software.

A few decades ago, people thought that calculations involving BCD (or just decimal) arithmetic were more accurate than binary calculations. Therefore, they would often perform important calculations, like those involving monetary units, using decimal-based arithmetic. Certain calculations can produce more accurate results in BCD, but for most calculations, binary is more accurate. This is why most modern computer programs represent all values (including decimal values) in a binary form. For example, the Intel 80x86 *floating-point unit (FPU)* supports a pair of instructions for loading and storing BCD values. Internally, the FPU converts these BCD values to binary. It only uses BCD as an external (to the FPU) data format. This approach generally produces more accurate results.

2.10 Fixed-Point Representation

There are two ways computer systems commonly represent numbers with fractional components: fixed-point representation and floating-point representation.

Back in the days when CPUs didn't support floating-point arithmetic in hardware, fixed-point arithmetic was very popular with programmers writing high-performance software that dealt with fractional values. There's less software overhead needed to support fractional values in a fixed-point format than in floating-point. However, CPU manufacturers added FPUs to their CPUs to support floating-point in hardware, and today, it's fairly rare to see someone attempt fixed-point arithmetic on a general-purpose CPU. It's usually more cost-effective to use the CPU's native floating-point format.

Although CPU manufacturers have worked hard at optimizing the floating-point arithmetic on their systems, in certain circumstances, carefully written assembly language programs that use fixed-point calculations will run faster than the equivalent floating-point code. Certain 3D gaming applications, for example, may produce faster computations using a 16:16 (16-bit integer, 16-bit fractional) format rather than a 32-bit floatingpoint format. Because there are some very good uses for fixed-point arithmetic, this section discusses fixed-point representation and fractional values using the fixed-point format.

NOTE *Chapter 4 will discuss the floating-point format.*

As you've seen, positional numbering systems represent fractional values (values between 0 and 1) by placing digits to the right of the radix point. In the binary numbering system, each bit to the right of the binary point represents the value 0 or 1 multiplied by some successive negative power of 2. We represent that fractional component of a value using sums of binary fractions. For example, the value 5.25 is represented by the binary value 101.01. The conversion to decimal yields:

$$1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} = 4 + 1 + 0.25 = 5.25$$

When using a fixed-point binary format, you choose a particular bit in the binary representation and implicitly place the binary point before that bit. You choose the position of the binary point based on the number of significant bits you require in the fractional portion of the number. For example, if your values' integer components can range from 0 to 999, you'll need at least 10 bits to the left of the binary point to represent this range of values. If you require signed values, you'll need an extra bit for the sign. In a 32-bit fixed-point format, this leaves either 21 or 22 bits for the fractional part, depending on whether your value is signed.

Fixed-point numbers are a small subset of the real numbers. Because the number of values between any two integer values is infinite, fixedpoint values cannot exactly represent every single one (doing so would require an infinite number of bits). With fixed-point representation, we have to approximate most of the real numbers. Consider the 8bit fixed-point format, which uses 6 bits for the integer portion and 2 bits for the fractional component. The integer component can represent values in the range 0 to 63 (or signed values in the range -32 to +31). The fractional component can represent only four different values: 0.0, 0.25, 0.5, and 0.75. You cannot exactly represent 1.3 with this format; the best you can do is approximate it by choosing the value closest to it (1.25). This introduces error. You can reduce this error by adding further bits to the right of the binary point in your fixed-point format (at the expense of reducing the range of the integer component or adding more bits to your fixed-point format). For example, if you move to a 16-bit fixed-point format using an 8bit integer and an 8-bit fractional component, then you can approximate 1.3 using the binary value 1.01001101. The decimal equivalent is as follows:

$$1 + 0.25 + 0.03125 + 0.15625 + 0.00390625 = 1.30078125$$

Adding more bits to the fractional component of your fixed-point number will give you a more accurate approximation of this value (the error is only 0.00078125 using this format, compared to 0.05 in the previous format).

In a fixed-point binary numbering system, there are certain values you can never accurately represent regardless of how many bits you add to the fractional part of your fixed-point representation (1.3 just happens to be such a value). This is probably the main reason why people (mistakenly) feel that decimal arithmetic is more accurate than binary arithmetic (particularly when working with decimal fractions like 0.1, 0.2, 0.3, and so on).

To contrast the comparative accuracy of the two systems, let's consider a fixed-point decimal system (using BCD representation). If we choose a 16-bit format with 8 bits for the integer portion and 8 bits for the fractional portion, we can represent decimal values in the range 0.0 to 99.99 with two decimal digits of precision to the right of the decimal point. We can exactly represent values like 1.3 in this BCD notation using a hex value like `$0130` (the implicit decimal point appears between the second and third digits in this number). As long as you use only the fractional values 0.00 to 0.99 in your computations, this BCD representation is more accurate than the binary fixed-point representation (using an 8-bit fractional component).

In general, however, the binary format is more accurate. The binary format lets you exactly represent 256 different fractional values, whereas BCD lets you represent only 100. If you pick an arbitrary fractional value, it's likely the binary fixed-point representation provides a better approximation than the decimal format (because there are over two and a half times as many binary versus decimal fractional values). (You can extend this comparison to larger formats: for example, with a 16-bit fractional component, the decimal/BCD fixed-point format gives you exactly four digits of precision; the binary format, on the other hand, offers over six times the resolution—65,536 rather than 10,000 fractional values.) Decimal fixed-point format has the advantage only when you regularly work with the fractional values that it can exactly represent. In the United States, monetary computations commonly produce these fractional values, so programmers figured the decimal format is better for monetary computations. However, given the accuracy most financial computations require (generally four digits to the right of the decimal point is the minimum precision), it's usually better to use a binary format.

If you absolutely, positively need to exactly represent the fractional values between 0.00 and 0.99 with at least two digits of precision, the binary fixed-point format is not a viable solution. Fortunately, you don't have to use a decimal format; as you'll soon see, there are other binary formats that will let you exactly represent these values.

2.11 Scaled Numeric Formats

Fortunately, there's a numeric representation that combines the exact representation of certain decimal fractions with the precision of the binary format. Known as the *scaled numeric* format, this representation is also efficient to use and doesn't require any special hardware.

Another advantage of the scaled numeric format is that you can choose any base, not just decimal, for your format. For example, if you're working with ternary (base-3) fractions, you can multiply your original input value by 3 (or a power of 3) and exactly represent values like $^1/_3$, $^2/_3$, $^4/_9$, $^7/_{27}$, and so on—something you can't do in either the binary or decimal numbering systems.

To represent fractional values, you multiply your original value by some value that converts the fractional component to a whole number. For example, if you want to maintain two decimal digits of precision to the right of the decimal point, multiply your values by 100 upon input. This translates values like 1.3 to 130, which we can exactly represent using an integer value. Assuming you do this calculation with all your fractional values (and they have the same two digits of precision to the right of the decimal point), you can manipulate your values using standard integer arithmetic operations. For example, if you have the values 1.5 and 1.3, their integer conversion produces 150 and 130. If you add these two values, you get 280 (which corresponds to 2.8). When you need to output these values, you divide them by 100 and emit the quotient as the integer portion of the value and the remainder (zero-extended to two digits, if necessary) as the fractional component. Other than needing to write specialized input and output routines that handle the multiplication and division by 100 (as well as dealing with the decimal point), you'll find that this scaled numeric scheme is almost as easy as doing regular integer calculations.

If you scale your values as described here, you've limited the maximum range of the integer portion of your numbers. For example, if you need two decimal digits of precision to the right of your decimal point (meaning you multiply the original value by 100), then you may only represent (unsigned) values in the range 0 to 42,949,672 rather than the normal range of 0 to 4,294,967,296.

When you're doing addition or subtraction with a scaled format, both operands must have the same scaling factor. If you've multiplied the left operand by 100, you must multiply the right operand by 100 as well. For example, if you've scaled the variable `i10` by 10 and you've scaled the variable `j100` by 100, you need to either multiply `i10` by 10 (to scale it by 100) or divide `j100` by 10 (to scale it down to 10) before attempting to add or subtract these two numbers. This ensures that both operands have the radix point in the same position (note that this applies to literal constants as well as to variables).

In multiplication and division operations, the operands do not require the same scaling factor prior to the operation. However, once the operation is complete, you may need to adjust the result. Suppose you have two values you've scaled by 100 to produce two digits of precision after the decimal point, `i = 25` (0.25) and `j = 1` (0.01). If you compute `k = i * j` using standard integer arithmetic, you'll get 25 (25 × 1 = 25), which is interpreted as 0.25, but the result should be 0.0025. The computation is correct; the problem is understanding how the multiplication operator works. We're actually computing:

$$(0.25 \times (100)) \times (0.01 \times (100))$$
$$=$$
$$0.25 \times 0.01 \times (100 \times 100) \text{ (commutative laws allow this)}$$
$$=$$
$$0.0025 \times (10{,}000)$$
$$=$$
$$25$$

The final result actually gets scaled by 10,000 because both `i` and `j` have been multiplied by 100; when you multiply their values, you wind up with a value multiplied by 10,000 (100 × 100) rather than 100. To solve this problem, you should divide the result by the scaling factor once the computation is complete. For example, `k = (i * j)/100`.

The division operation suffers from a similar problem. Suppose we have the values `m = 500` (5.0) and `n = 250` (2.5) and we want to compute `k = m/n`. We would normally expect to get the result 200 (2.0, which is 5.0/2.5). However, here's what we're actually computing:

$$(5 \times 100) \ / \ (2.5 \times 100)$$
$$=$$
$$500/250$$
$$=$$
$$2$$

At first blush this may look correct, but the result is really 0.02 after you factor in the scaling operation. The result we need is 200 (2.0). Division by the scaling factor eliminates the scaling factor in the final result. Therefore, to properly compute the result, we need to compute `k = 100 * m/n`.

Multiplication and division place a limit on the precision you have available. If you have to premultiply the dividend by 100, then the dividend must be at least 100 times smaller than the largest possible integer value, or an overflow will occur (producing an incorrect result). Likewise, when you're multiplying two scaled values, the final result must be 100 times less than the maximum integer value, or an overflow will occur. Because of these issues, you may need to set aside additional bits or work with small numbers when using scaled numeric representation.

2.12 Rational Representation

One big problem with the fractional representations we've seen is that they provide a close approximation, but not an exact representation, for all rational values.[6] For example, in binary or decimal you cannot exactly represent the value $^1/_3$. You could switch to a ternary (base-3) numbering system and exactly represent $^1/_3$, but then you wouldn't be able to exactly represent fractional values like $^1/_2$ or $^1/_{10}$. We need a numbering system that can represent *any* rational fractional value.

Rational representation uses pairs of integers to represent fractional values. One integer represents the numerator (n) of a fraction, and the other represents the denominator (d). The actual value is equal to n/d. As long as n and d are "relatively prime" (that is, not both evenly divisible by the same value), this scheme provides a good representation for fractional values within the bounds of the integer representation you're using for n and d. Arithmetic is quite easy; you use the same algorithms to add, subtract, multiply, and divide fractional values that you learned in grade school when dealing with fractions. However, certain operations may produce really large numerators or denominators (to the point where you get integer overflow in these values). Other than this problem, you can represent a wide range of fractional values using this scheme.

2.13 For More Information

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* 3rd ed. Boston: Addison-Wesley, 1998.

6. It isn't possible to provide an exact computer representation of an irrational number, so we won't even try.