Lecture 10: 0100100001101001

This lecture describes methods of using strings of 0's and 1's to represent integers, floating point numbers and characters.

Binary numbers

Just as we use the digits 0 through 9 to represent decimal numbers, or numbers base 10, we can use the digits 0 and 1 to represent binary numbers, or numbers base 2. The binary number 1101 represents the integer 13 as follows:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13.$$

This gives us one algorithm to convert from a binary number to an integer: process the binary digits (bits) right to left, keeping track of the corresponding power of 2, accumulating the power of 2 to a total if the bit is 1 and not if the bit is 0.

What integer is represented when we interpret the 16-bit string of 0's and 1's in the title of this lecture as a binary number?

$$0100100001101001 \Rightarrow 2^{14} + 2^{11} + 2^6 + 2^5 + 2^3 + 2^0 = 18537.$$

How can we do the converse, that is, convert an integer into its binary representation? Two observations give us a nice method. The first that the rightmost bit of a binary number is 0 if the number is even and 1 if the number is odd, for example:

$$\begin{aligned} 1101 &\Rightarrow 13 \\ 1100 &\Rightarrow 12 \end{aligned}$$

The second is that when we multiply a number by 2, we affix a 0 at the right of its binary representation, for example:

$$\begin{aligned} 1101 &\Rightarrow 13 \\ 11010 &\Rightarrow 26 \end{aligned}$$

Conversely, to divide an even number by 2, we can simply erase the rightmost bit 0. Putting these together, we can repeatedly determine the rightmost bit of the number, subtract the bit (to make it even), and divide by 2. As a shortcut, we can use the fact that `(quotient n 2)` has the same effect as subtracting the rightmost bit and dividing by two. For example:

```
(remainder 13 2) => 1
(quotient 13 2) => 6
(remainder 6 2) => 0
(quotient 6 2) => 3
(remainder 3 2) => 1
(quotient 3 2) => 1
(remainder 1 2) => 1
(quotient 1 2) => 0
```

The successive remainders: 1, 0, 1, 1, give the bits of the binary representation of 13 in reverse order.

Another method of converting an integer into its binary representation is to find the highest power of two that is less than or equal to the number, subtract that power of two and repeat. So, for 13, we find the highest power of 2 that does not exceed 13, namely $2^3 = 8$, subtract 8 from 13 to get 5, and repeat the process with 5. Doing this, we find that

$$13 = 2^3 + 2^2 + 2^0,$$

which we convert into a binary representation: 1101.

For a fixed number of bits, for example 16, how many integers can we represent? There are 2^{16} different bit patterns, and each one represents a different integer, from 0 to $2^{16} - 1 = 65535$. In general, n bits can be used to represent integers from 0 to $2^n - 1$.

Floating point numbers

What if we want to represent non-integers, like 3.75? We could choose a small unit, for example, 0.001, and express our numbers as integral multiples of that unit, so that 3.75 would be represented by the integer 3750. This would work, but with 16 bits, the largest number we could represent would now be 65.535, which isn't very large.

In scientific computation we often have a need for very large or very small numbers that have a comparable number of significant digits. This need is answered by scientific notation: express a number as a decimal number times an appropriate power of 10. In Scheme, `32.456e4` denotes the number

$$32.456 \cdot 10^4 = 324560,$$

and `0.172e-2` denotes the number

$$0.172 \cdot 10^{-2} = 0.00172.$$

The idea of floating-point representation is based on the idea of scientific notation, but generally uses base 2 instead of base 10. First we have to be clear about binary fractions. What number should be represented by the binary fraction:

$$0.1101?$$

Analogous to the decimal case, the first position to the right of the decimal point is the $2^{-1} = 1/2$ position, the next is the $2^{-2} = 1/4$ position, and so on. Thus, the binary fraction 0.1101 represents:

$$1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 1/2 + 1/4 + 1/16 = 13/16 = 0.8125.$$

For a floating point number, the information we need to represent is a sign (whether the number is positive or negative), a binary fraction (the mantissa), and an exponent, which is positive, zero, or negative integer.

The text book (ICS) specifies a 16-bit floating point format, which is used in the exercises. This format is not used in practice, but it illustrates the basic ideas of formats that are used in practice. ¿From the leftmost bit to rightmost bit, the format allocates 1 bit for the sign of the number (0 for positive, 1 for negative), 9 bits for the mantissa of the number (with an implied binary point to the left of the first bit), 1 bit for the sign bit of the exponent (0 for positive, 1 for negative), and 5 bits for the exponent (base 2). The layout for the 16 bits can be pictured:

```
---------------------------------
| | | | | | | | | | | | | | | |
---------------------------------
|s|      m          |t|    e    |
---------------------------------
```

In this diagram, s is the sign bit, m is the mantissa, t is the sign of the exponent, and e is the exponent. How would we convert the bit pattern in the title of this lecture into the floating point number it represents in this format? We begin by dividing the bit string into its parts:

```
---------------------------------
|0|1|0|0|1|0|0|0|0|1|1|0|1|0|0|1|
---------------------------------
|s|      m          |t|    e    |
---------------------------------
```

Because $s = 0$, the sign of the number is positive. The mantissa m is the binary fraction 0.100100001, which we convert to

$$289/512 = 0.564453125.$$

The sign of the exponent is 1, meaning that the exponent is negative. Finally, the absolute value of the exponent is the integer represented by the binary number 01001, or 9. Putting all this together, the number represented is:

$$+0.564453125 \cdot 2^{-9} = 0.001102447509765625.$$

Thus, this bit pattern represents a number that is about 1/1000.

Note that we can represent $2^{30} = 1073741824$ as well as $2^{-32} = 2.3283064365386962e-10$, numbers that are nearly 19 decimal orders of magnitude apart. But since we have only 16 bits, we cannot be representing more than $2^{16} = 65536$ different numbers. In fact, because some numbers are represented in more than one way, strictly fewer than 65536 numbers are representable. (How many ways can you think of to represent 0.0 in this format?)

Very big numbers, very small numbers, but fewer than 65536 of them: doesn't something have to give? Yes, very many numbers between the largest and smallest cannot be represented exactly. This contrasts with the integer case, where all the integers between the largest and smallest representable integers are themselves representable. For example, see if you can find an exact representation of the number 1025 in this format. Its neighbor 1024 is exactly representable in this format: 0100000000001011, but the number 1025 is not exactly representable. So what do we do about the numbers that are not representable? We approximate them. A limited number of significant figures is a familiar situation for an experimental scientist – this situation is analogous.

Enough numbers!

We give one more interpretation of the string of 16 0's and 1's in the title of this lecture. In 16 bits, we can store two bytes, that is 8-bit quantities. Thus, the bit string in the title could be thought of as the two bytes: 01001000 and 01101001. Referrring to the table of ASCII codes on page 128 of ICS, we see that each character is represented by a particular pattern of 8 bits. Decoding the two bytes as characters, we get *H* and *i*, so the 16 bits of the title can also be interpreted as representing the two characters: *Hi*.