Converting from decimal to binary (continued)

Recall the algorithm we saw last class for converting from decimal to binary. This algorithm was based on the "mod" operator `(%)` and the integer division operator `(/)`. The former computes the "remainder" of an integer division, and the latter ignores the remainder. e.g. `60 % 7 = 4`, and `60 / 7 = 8`, since `60 = 8 x 7 + 4`. Note that `m % 2` is either 0 or 1, depending on whether `m` is even or odd, respectively. Here is the algorithm again:

```
//      convert a positive integer m to binary
i = 0;
while (m > 0){
   b[i] = m % 2;      //   result is 1 if m is odd and 0 if m is even
   m    = m / 2;      //   truncated i.e.  "floor"
}
```

Why does this algorithm work ? The first intuition of why this might work comes from the observation that for any positive integer m, we have

$$m = 2(m/2) + (m\%2)$$

where the / and % operators were as defined above. Let's see why this observation makes the algorithm work.

Representing a positive integer m in binary means that we write it as a sum of powers of 2 (just as decimal reprsents a number as a sum of powers of 10). Let's suppose we can write the number as:

$$m = \sum_{i=0}^{n-1} b_i \, 2^i$$

where b_i is a bit, $i.e$ either 0 or 1 and so we can write m in binary as $(b_{n-1}\ b_{n-2}\ \ldots\ b_2\ b_1\ b_0)_{\text{two}}$.

For example, if we were looking at $n = 3$ bit numbers, we would write them as 000, 001, 010, 011, 100, 101, 110, 111 which in decimal are the numbers 0 through 7. If we were looking at $n = 8$ bit numbers (called *bytes*), we would write them as $00000000, 00000001, \ldots, 11111111$ and these would correspond to the numbers from 0 through 255.

Getting back to our problem, notice that our n-bit number can be rewritten:

$$m = \sum_{i=0}^{n-1} b_i \, 2^i \ = \ \sum_{i=1}^{n-1} b_i \, 2^i + b_0 \ = \ (2 \sum_{i=0}^{n-2} b_{i+1} \, 2^i) \ + b_0.$$

It follows that

$$m \ \% \ 2 = b_0$$

and

$$m/2 \ = (b_{n-1} \ldots b_2 b_1)_{\text{two}}.$$

Thus, repeatedly dividing by 2 and using the "remainder" bits gives us our b_i values.

If you are still not convinced, let's run another example where we "know" the answer from the start and we'll see that the algorithm does the correct thing. Suppose our number is $m = 241$, which is 1110001 in binary.

i	m	b[i]
0	$\boxed{11110001}$	
1	1111000	1
2	111100	0
3	11110	0
4	1111	0
5	111	1
6	11	1
7	1	1
8	0	1
9	0	0
10	0	0
11	:	:

The remainders are simply the bits used in the binary representation of the number!

Final note: The representation has an infinite number of 0's on the left which is often truncated. That is, often we only use as many bits as necessary to specify the binary number.

Algorithm for addition (in binary)

The algorithm for addition of two positive integers is as easy with the binary representation as it is with the decimal representation which we saw last class. Let's assume an eight bit representation and compute the sum 26 + 27 = 53.

$$\begin{array}{ll} & 00110100 \leftarrow 26 \\ + & \underline{00011011} \leftarrow 27 \end{array}$$

Its the same algorithm that you use with decimal, except that you are only allowed 0's and 1's. Whenever the sum in a column is 2 or more, you carry to the next column:

$$2^i \ + \ 2^i = 2^{i+1}$$

So,

$$\begin{array}{lll} & 00110100 & \leftarrow \text{carry bits} \\ & 00011010 & \leftarrow 26 \\ + & \underline{00011011} & \leftarrow 27 \\ & 00110101 & \leftarrow 53 \end{array}$$

This is much closer to how computers do arithmetic than the base 10 method that I presented last class. In particular, notice that the basic operations that need to be done here involve "summing" three bits, namely the i^{th} bit of each number and the corresponding carry bit.

Binary fractions

Up to now we have only talked about integers. We next talk about binary representations of fractional numbers, that is, numbers that lie between the integers. Take a decimal number such as 22.63. We write this as:

$$(26.375)_{\text{ten}} \;=\; 2*10^1+6*10^0+3*10^{-1}*7*10^{-2}+5*10^{-2}.$$

The "." is called the *decimal point*.

One uses an analogous representation using binary numbers, *e.g.*

$$(11010.011)_{\text{two}} \;=\; 1*2^4+1*2^3+0*2^2+1*2^1+0*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}$$

where "." is called the *binary point*. Check for yourself that this is the same number as above, namely

$$16+8+2+0.25+0.125 \;=\; 26.375.$$

"Scientific Notation" in binary

One typically represents very large decimal numbers or very small decimal numbers by writing, for example:

$$300000000 = 3 \times 10^8$$

$$.00000456 = 4.56 \times 10^{-6}$$

For the expressions on the right, there is exactly one digit to the left of the decimal point and this digit is from 1 to 9 (not 0). (Note that the number 0 cannot be written in this way since you need at least one non-zero digit.)

One uses a similar representation for binary numbers, for example,

$$(1000.01)_{two} = 1.00001 \times 2^3$$

$$(0.111)_{two} = 1.11 \times 2^{-1}$$

The numbers on the right side have the form:

$$1.\underline{\hspace{3cm}} \times 2^E$$

where the $\underline{\hspace{3cm}}$ is called the *significand* (or *mantissa*) and E is the *exponent*. Such numbers are also called "floating point". Again, it is impossible to encode the number 0 in this form.

As you know from COMP 202, computers represent non-integer numbers either as `float` (32 bits) or `double` (64 bits). Such numbers are represented in a standard way, namely a certain number of bits are used for the exponent and mantissa in each case, and one bit is used for the sign (positive or negative). The same representation is used in nearly all computers in the world. You will learn the details (called the IEEE 754 standard) in future courses.

This completes our first step into some of the fundamentals of computer science, in particular, representing numbers in binary. Next lecture, we will begin discussing a very different topic, which will be much more what you were expecting, namely continuing from COMP 202. (Later we will return to discussing algorithms and binary numbers again.)