Welcome to COMP 273. Let's get started! You all know that computers represent numbers using 0's and 1's. But how exactly this works is probably a mystery for most of you. We start out with a basic and fundamental example, namely how to represent integers. We'll start with positive integers, then negative integers, and then we'll turn to non-integers i.e. real numbers.

Decimal vs. binary representation of positive integers

Up to now in your life, you've represented numbers using a decimal representation (the ten digits from 0,1, ... 9). The reason 10 is special is that we have ten fingers. There is no other reason for using decimal. There is is nothing special otherwise about the number ten.

Computers don't represent numbers using decimal. Instead, they represent numbers using binary. All the algorithms you learned in grade school for addition, subtraction, multiplication and division work for binary as well. Before we review these algorithms, lets make sure we understand what binary representations of numbers are. We'll start with positive integers.

In decimal, we write numbers using *digits* $\{0, 1, \ldots, 9\}$, in particular, as sums of powers of ten. For example,

$$238_{ten} = 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In binary, we represent numbers using *bits* $\{0, 1\}$, in particular, as a sum of powers of two:

$$11010_{two} = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

I have put little subscripts (*ten* and *two*) to indicate that we are using a particular representation (decimal or binary). We don't need to always put this subscript in, but sometimes it helps to remind us of what base we are using. For example, 11010_{two} is not the same thing as 11010_{ten}.

It is trivial to write a decimal number as a sum of powers of ten and it is also trivial to write a binary number as a sum of powers of two, namely, just as I did above. To *convert* from a binary number to a decimal number is also trivial. You need to write the powers of 2 as decimal numbers and then add up these decimal numbers.

$$11010_{two} = 16 + 8 + 2$$

To do so, you need to memorize the powers of 2

$$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32,\ 2^6 = 64,\ 2^7 = 128,\ 2^8 = 256,\ 2^9 = 512,\ 2^{10} = 1024,\ \ldots$$

How do you convert from a decimal number to a binary number? Your first idea would be to find the largest power of 2 that is less than the number, subtract that power of 2, and then repeat to find the smaller powers of 2 in the remainder. This works, but you need to know the powers for 2 to use it.

On the next page, I show another algorithm for converting from decimal to binary, and an example. The algorithm repeatedly divides the decimal number by 2, and concatenates the remainders. Why does this work? For any positive number m, let

$$m = \lfloor m/2 \rfloor * 2 + (m \bmod 2)$$

where $\lfloor m/2 \rfloor$ is the *quotient* and $\lfloor\ \ \rfloor$ rounds down (floor). Let $(m \bmod 2)$ be the *remainder*, i.e. division mod 2. In the lecture slides, this was written with slightly different notation for mod – you should be familiar with both.

When m is already represented as binary number (i.e. "base 2"), the quotient and remainder are trivial to obtain. The remainder is the rightmost bit – called the *least significant bit* (LSB). The quotient is the number with the LSB chopped off. To convince yourself of this, note that writing a positive integer as an n bit binary number means that you write it as a sum of powers of 2,

$$(b_{n-1}\ b_{n-2}\ \ldots\ b_2\ b_1\ b_0)_{two} \equiv \sum_{i=0}^{n-1} b_i\ 2^i = \sum_{i=1}^{n-1} b_i\ 2^i + b_0 = 2 \times \sum_{i=0}^{n-2} b_{i+1}\ 2^i\ + b_0$$

Here is the convertion algorithm:

Algorithm 1 Convert decimal to binary

INPUT: a number m expressed in base 10 (decimal)

OUTPUT: the number m expressed in base 2 using a bit array $b[\,]$

```
$i \leftarrow 0$
while $m > 0$ do
    $b_i \leftarrow m\%2$
    $m \leftarrow m/2$
    $i \leftarrow i + 1$
end while
```

For example,

quotient	remainder	interpretation (not part of algorithm)
$\boxed{241}$		
120	1	241 = 120*2 + 1
60	0	120 = 60*2 + 0
30	0	60 = 30*2 + 0
15	0	30 = 15*2 + 0
7	1	15 = 7*2 + 1
3	1	7 = 3*2 + 1
1	1	3 = 1*2 + 1
0	1	1 = 0*2 + 1
0	0	0 = 0*2 + 0
0	0	
:	:	

Thus,

$$241_{ten} = 11110001_{two}.$$

You need to be able to do this for yourself. So practice it!

If you are not fully clear why the algorithm is correct, consider what happens when you run the algorithm *where you already have the number in binary representation.* (The quotient and remainder of a division by 2 does not depend on how you have represented the number i.e. whether it is represented in decimal or binary.)

(quotient)	(remainder)
$\boxed{11110001}$	
1111000	1
111100	0
11110	0
1111	0
111	1
11	1
1	1
0	1
0	0
0	0
:	:

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 can be truncated. I mention this because in the heat of an exam you might get confused about which way to order the 0's and 1's. Remembering that you have infinitely many 0's when you continue to apply the trick. This tells you that the remainder bits go from right to left.

Performing addition with positive binary numbers

Doing addition with positive integers is as easy with the binary representation as it is with the decimal representation. Let's assume an eight bit representation and compute 26 + 27.

$$\begin{array}{rll} & 00110100 & \leftarrow \text{carry bits (ignore the 0's for now)} \\ & 00011010 & \leftarrow 26 \\ + & \underline{00011011} & \leftarrow 27 \\ & 00110101 & \leftarrow 53 \end{array}$$

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:

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

We would like to apply the grade school algorithm to do subtraction, multiplication and long division in binary. However, there are some subtleties in doing so. For example, when you subtract a big positive number from a small positive number, you end up with a negative number, and I have not yet told you how to represent negative numbers. That's what I'll start with next lecture.