NUMBER REPRESENTATIONS IN THE COMPUTER for COSC 120

First, a reminder of how we represent base ten numbers.
Base ten uses ten (decimal) digits: 0, 1, 2,3, 4, 5, 6, 7, 8, 9.

In base ten, 10 means ten.

Numbers are represented positionally in a string of decimal digits plus a single, optional decimal point and a single, optional sign. The string of digits can be as long as necessary in order to represent the number to some arbitrary, but fixed precision.

The meaning of the string of digits depends on the position of the decimal point. For example, in base 10, the number 923 has the meaning:
$9*10^2 + 2*10^1 + 3*10^0$.

The number 5005.1307 has the meaning:
$5*10^3 + 0*10^2 + 0*10^1 + 5*10^0 + 1*10^{-1} + 3*10^{-2} + 0*10^{-3} + 7*10^{-4}$

Other bases work the same in analogy to base ten. In computer science, the two most common bases are base two and base sixteen (base eight is also used, but less commonly).

Base two uses two (binary) digits: 0, 1.

In base two, 10 means two.

Again, numbers are represented positionally with respect to a binary point. In base two, the number 1101. has the meaning:
$1*10^{11} + 1*10^{10} + 0*10^1 + 1*10^0$
(the 10 in the base is, of course, two)

That expression is a little difficult to read, so let us mix notation, using base ten notation for the base and exponent.
$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

Do you see the similarity to 923_{10} above?

In base two, the number 1011.00101 has the meaning
$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 0*2^{-1} + 0*2^{-2} + 1*2^{-3} + 0*2^{-4} + 1*2^{-5}$

It is simple to convert from base two to base ten, and almost as simple to convert from base ten representation to base two. I'll show you the base two to base ten conversion.

Consider the base 2 number 1011.00101 given in positional notation from above:
$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 0*2^{-1} + 0*2^{-2} + 1*2^{-3} + 0*2^{-4} + 1*2^{-5}$
First calculate the powers of two then give in base 10:
2^3 -- > 8
2^2 -- > 4
2^1 -- > 2
2^0 -- > 1
2^{-1} -- > 1 / 2^1 = 1 / 2 = 0.5
2^{-2} -- > 1 / 2^2 = 1 / 4 = 0.25
2^{-3} -- > 1 / 2^3 = 1 / 8 = 0.125
2^{-4} -- > 1 / 2^4 = 1 / 16 = 0.0625
2^{-5} -- > 1 / 2^5 = 1 / 32 = 0.03125

Now multiply out and add up using base ten.

$1*8 + 0*4 + 1*2 + 1*1 + 0*2^{0.5} + 0*0.25 + 1*0.125 + 0*0.0625 + 1*0.03125 = 11.15625$

Base sixteen ("hex" or "hexadecimal") works the same way. Base sixteen has sixteen digits. Here is a handy table of conversions between hex digits, the equivalent decimal value, and the equivalent **4-bit** binary value.

hex	dec	bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

When converting between hex and binary always use 4 bits for the binary. Pad with zeros on the left as needed to make 4 bits.

Conversion from hex -- > binary and from binary -- > hex is easy.

For hex -- > binary, each hex digit is converted to the equivalent 4-bit binary number (starting on either end is ok, but it's easier if you start at the right).

Example: F A 0 9 -- > 1111 1010 0000 1001
Example: 0 1 2 E -- > 0000 0001 0020 1111

For binary -- > hex, group 4-bit binary numbers (starting from the RIGHT, aka, the binary point)

Example: 0 1 1 0 1 -- > 0 1101 -- > 0 D in hex
Example: 1 0 1 1 1 1 1 0 0 -- > 1 0111 1100 -- >
0001 0111 1100 -- > 1 7 C in hex

A commonly understand notation is to follow the number by a single character to indicate the base:
0D h is the hexadecimal 0D (== > 13 d, thirteen)
01101 b is the binary 01101 (== > 13 d, thirteen)
101 h is the hexadecimal 101 (==> $16^2 + 16^0$ ==> 257 d, two hundred fifty seven)

Operations on binary numbers:

Operations: Arithmetic

addition

+	0	1
0	0	1
1	1	10

multiplication

•	0	1
0	0	0
1	0	1

In mathematics, addition and multiplication commute and associate.

Operations: Logical

and (frequently implied, e.g., a • b is written a b)

•	0	1
0	0	0
1	0	1

or

+	0	1
0	0	1
1	1	1

xor

⊕	0	1
0	0	1
1	1	0

⊕ is useful for cryptography and for computing check sums and parity. We won't be using xor much.

complement, negate, not, !, ~

~	
0	1
1	0

In mathematics, the logical operations commute and associate, as constrained by their priorities.

The priorities from high to low are ~, •, +.

The + symbol is used for both addition and or. The • symbol is used for both multiplication and and. Context should tell you which operation is meant.

Example: what is the value of `(1 • 1) + (0 + 1)`?

```
( 1 • 1 ) -- > 1
( 0 + 1 ) --> 1
1 + 1 -- > 1
```

Example: for a = 1, b = 0, what is the value of `~ a + ~(a • b)`?

```
~ 1 + ~(1 • 0)-- > 0 + ~ 0 -- > 0 + 1 -- > 1
```

When doing logical operation on bit-strings, the operations are applied bit-by-bit.
Example: `0 1 0 1 • 0 1 1 0 == > 0 0 0 0`
Example: `1 1 0 0 + 1 0 1 0 == > 1 1 1 0`
Example: `(0 0 1 0 • 1 1 1 0) + 0 0 0 1     ==> 0010 + 0001`
```
                                ==> 0011
```

When doing logical operations, 1 can mean TRUE and 0 can mean FALSE.

Addition can be performed using similar rules as used for base ten: use a "carry" digit when the sum of two operands requires 2 bits.

Example: 1 0 1 1 + 0 1

```
         1 1      (< -- the carry bits)
      1 0 1 1
   +  0 0 0 1
--------------
      1 1 0 0
```

Example: 0 1 1 1 + 0 0 1 1

```
      1 1 1
      0 1 1 1
    + 0 0 1 1
--------------
      1 0 1 0
```

Cover up the right hand side and test your understanding

Problem	Solution
arithmetic operations	
`11 + 1001`	`1100`
`(11 + 1001) * 100`	`11 0000`
`1 + 10 + 11 + 100`	`1010`
`1 * 10 * 11 * 100`	`1 1000`
logical operations	
`(1 + 0) • (0 + 1)`	`0`
`~(1 + 0) • ~(0 + 1)`	`0`
`~1101 + 01`	`0011`
`(101 • 011) + ~ 100`	`011`

Conversions bin -> hex

0 1 1 0	6 h
1 0 1 1 0	16 h
0 0 0 0 1 0 0 0 0 0 0 0	0 1 0 h
1 0 1 1 1 1 1	5 F h

Conversions hex -> bin

0 1 2	0000 0001 0010
A B	1010 1011
7 7 7 F	0111 0111 0111 1111
1 9	0001 1001
F 7 E 6	1111 0111 1110

How numbers are stored in the computer

Storage in the computer is accomplished by storing "words" in addressed locations. A value is retrieved by specifying its address. A word is a grouping of bits based on the architecture of the computer. A particular architecture has one word size.

In early computers, the most common and then most influential word size was 8 bits. 8 bits is a byte. In more modern computers, the word size is larger, 32 bits for example.

In the most primitive approach, the most primitive "type" of a number will fit into a single word. For an 8-bit word, the type of this kind of number is called `byte`. To store a positive integer into 8 bits, only the bit strings 0000 0000 through 1111 1111 (00h - FFh). Thus, in 8 bits, the largest possible integer is 255. If a computer has a larger word size, the can store larger numbers. But, in any fixed word size, there will be a largest possible number that can fit. In scientific and engineering computing, that number will never be large enough.

There is a second primitive type of a number commonly called `float`. Without going into details, a part of a word is assigned to be an exponent, and the remaining bits are used for the significand. For example, suppose we use 4 places and base ten numbers.

The largest integer that can be stored in 4 places is 9999. If two of those places are used for an exponent, then the largest number that can be stored is $99 * 10^{99} >> 9999$. But notice that even if using something like scientific notation, there are still only 10,000 unique values that can be stored in 4 places (with decimal digits). That means that the *precision* of floats is sacrificed for the *range* of floats. I.e., the range is greater, but not every (or even most) of the numbers within the range can be represented exactly.

You can look at http://blogs.mathworks.com/seth/2008/12/14/representing-numbers-integers-and-fixed-point/ for another exposition on this topic (*fixed-point* is much like *float*).

MatLab, like all software environments, builds on these primitive types. This means that there is a limit on how large or how small a number can be; also numbers often not precise. Be aware of this when you are solving scientific and engineering problems: you are likely to need extensions or modules that will improve range and precision.

Practice: cover up the right side to test your understanding

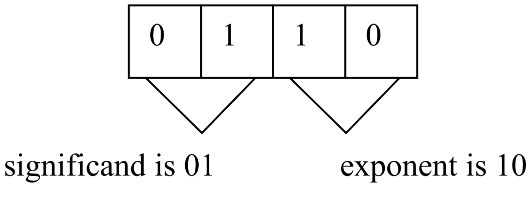
1. Suppose you are given a 4-bit word in a
computer for storing integer-type positive numbers.

a. How many possible numbers can be stored? There are 16 unique patterns

b. What is the largest possible number? 1111 b == > 15 d

c. What is the smallest possible number? 0 0 0 0 b == > 0 d

2. Suppose you are given a 4-bit word in a computer for storing float type positive numbers. These are base two digits. The format for specifying a float number is fixed as follows: the first two bits (the leftmost bits) are used for the significand, the last two bits (the rightmost bits) are used for the exponent, the binary point is understood to be to the right of the significand numbers. The base for the exponent is two (not ten as in scientific notation). Here is a picture and an example number:



The value of the stored number is $01. * 2^{10}$
I.e., $1 * 2^2$ == > 4

a. How many possible numbers are there? Again, there are 16 unique patterns,
(but notice the problem in part c!)

b. What is the largest possible value? $11 * 2^{11}$ ==> 24 d

c. This particular format for float is extremely simple, but it is not a very good one. One reason it is not very good is that there are four different ways of storing the number zero. What are they?

0000 == > $0 * 2^{0}$
0001 == > $0 * 2^{1}$
0010 == > $0 * 2^{2}$
0011 == > $0 * 2^{3}$

Why do <u>you</u> need to know this binary, hexadecimal, and number type stuff?

(1) to understand why there are built-in errors in numerical calculation on computing devices,
(2) to be able to better communicate with any computer scientists/engineers on your future teams,
(3) to better understand computational principles. The more you understand the principles, the easier it is to leverage your knowledge to something new.