1. Number Systems

Number Systems and Number Bases

The number system we are most familiar with is called the Arabic number system. It is a positionally based symbol system which makes computation significantly easier than many of the alternatives. Basically it works by placing the least significant digit on the right and progressing to the left toward more significant digits. Each place represents a quantity times a power of the base unit.

Base 10

The most commonly used number base is base 10. In base 10, the right most position is the 1's, the next position to the left is the 10's, then the 100's, and so on. Each position represents a power of 10. The 1's position is 10 to the zeroth power, or 10^0. (Interestingly, *any* number to the zeroth power is equal to 1.) The 10's position is 10^1, the 100's position is 10^2, and so on. Using this method, we can define a numeric value as large as we want. To make a larger value, just create a new position to the left.

Numeric values are defined by using intermediate values in each position, and then summing the quantity. For example, the value represented by 167 can be interpreted as

$$1*10^2 + 6*10^1 + 7*10^0$$

Base 2

Numeric values can be represented in base 2 also. Computers are limited to storing everything in bits, which are usually electrical charges interpreted as on or off, or 1 and 0. Computer scientists and programmers commonly work in base 2 because the 1 and 0 quantities used by computers lend themselves well when attempting to perform most kinds of arithmetic, especially when organized using the Arabic system.

In base 2, the right most position, or bit, is the 1's position. The next position to the left is the 2's position while the next position is the 4's position. Notice that each position is a power of 2, just as each position in base 10 represented a power of 10.

For example, the value represented by the sequence 01001101 can be interpreted as

$$0*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

or

$$0*128 + 1*64 + 0*32 + 0*16 + 1*8 + 1*4 +0*2 + 1*1$$

Can you see a pattern?

To avoid confusion, we often add a subscript to indicate the number base when we are using multiple number bases, or a number base other than base 10.

162_h	h means hexadecimal	162_{16}	16 means base 16
162_d	d means decimal	162_{10}	10 means base 10
162_o	o means octal	162_8	8 means base 8
101_b	b means binary	101_2	2 means base 2

Converting Binary to Decimal

Converting a numeric value represented in binary into decimal can be accomplished with little more than some paper and a calculator. (The calculator makes the arithmetic easier, but isn't essential.)

A method that has worked for me is to write the quantity for the power of 2 over or beneath each bit. For example, the sequence 01001101 can be deciphered as shown below:

128	*64*	*32*	*16*	*8*	*4*	*2*	*1*
0	**1**	**0**	**0**	**1**	**1**	**0**	**1**

Wherever a position is 1, add that positional value to our working sum. Thus, for the example above, the sum would be

$$64 + 8 + 4 + 1$$

or equivalent to 77 in base 10.

Remember that, like base 10, base 2 values can be as large as they need to be to represent a value. In the example above the value was represented by 8 bits.

Converting Decimal to Binary

If converting a value represented in binary into decimal can be performed using addition, then there may be a way to convert a decimal value into binary using subtraction. Just such a method will be used here.

As an example let's convert the decimal vale 167. The first thing to do is to create a list of values representing powers of 2 to the right. (It doesn't have to be to the right, but it needs to go somewhere.

167	128
	64
	32
	16
	8
	4
	2
	1

Going down the list of powers of 2, determine if that value can be subtracted resulting in either a positive or zero remainder. (Technically that can be called a 'non-negative' remainder.) In the case of 128, it can be subtracted. That fact is identified by placing a '1' next to the 128.

167	128	1
-128	64	
39	32	
	16	
	8	
	4	
	2	
	1	

Going down the list, can 64 be subtracted from 32 giving us a non-negative result? No, it can't, so a 0 is placed next to 64.

167	128	1
-128	64	0
39	32	
	16	
	8	
	4	
	2	
	1	

We continue down the list, subtracting where we can.

167	128	1
-128	64	0
39	32	1
-32	16	0
7	8	0
-4	4	1
3	2	1
-2	1	1
1		
-1		
0		

When the result is finally 0, you can stop. Our bit pattern is the right-most column of 1's and 0's. The most significant bit is at the top, and the least significant bit is at the bottom. Thus, in this case, we can write that 10100111 in base 2 is equivalent to 167 in base 10.

Our solution can be verified by using the method we already know to convert the binary number into decimal. Thus, we would write:

$$128 + 32 + 4 + 2 + 1 = 167$$

which gives us the value we started out with in the conversion process. Whenever there is any doubt, you should always be able to verify your solution by reversing the process.

There is another method which requires that you divide by 2. In this method, you need to divide the result from each step using remainders, and the use the result in the next step. For the value 167, the process would look like this:

167 / 2 = 83 r 1 (note that the remainder is to the right of the little 'r')
83 / 2 = 41 r 1
41 / 2 = 20 r 1
20 / 2 = 10 r 0
10 / 2 = 5 r 0
5 / 2 = 2 r 1
2 / 2 = 1 r 0
1 / 2 = 0 r 1

When you get to 0, stop. The bit pattern is the 'remainders' with the least significant bit at the top, and the most significant bit at the bottom. (This is just the opposite of the previous method.) Thus, our bit pattern is 10100111.

Both methods work well, and can be verified.

Hexadecimal Number System [Base-16]

The hexadecimal number system uses *sixteen* values to represent numbers. The values are,

0 1 2 3 4 5 6 7 8 9 A B C D E F

with 0 having the least value and F having the greatest value. Columns are used in the same way as in the decimal system, in that the left most column is used to represent the greatest value.

Hexadecimal is often used to represent values [numbers and memory addresses] in computer systems.

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

Hexadecimal values are often used with a notation so that you, as programmer, know that the value is in hexadecimal. Depending on the system the number may be preceded by #, 0#, 0&, 0c, or some other notation.

Converting Decimal to Hexadecimal

To convert a decimal number to a hexadecimal, use the method used earlier to convert decimal to binary, but divide by 16 instead of by 2.

232 / 16 = 14 with a remainder of 8
14 / 16 = 0 with a remainder of E (14 decimal = E)

= $E8_{16}$

Another method is to convert the decimal value into a binary number, and then convert that result as shown below.

Converting Hexadecimal to Decimal

Let's look at an example converting 176 in hexadecimal to decimal, or base 10.
Each column represents a power of 16,

176_{16} =
$6 * 16^0 = 6$
$7 * 16^1 = 112$
$1 * 16^2 = \underline{256}$
374 in base 10

Another example converting FD:

FD_{16} =
$D * 16^0 = 13$
$F * 16^1 = \underline{240}$
253 in base 10

Converting Binary to Hexadecimal

Let's consider the case of converting the binary number 10110 to hexadecimal. Each hexadecimal digit represents 4 binary bits. Split the binary number into groups of 4 bits, starting from the right. In this case, 10110 is equivalent to 00010110 (we can always add zeros to the left to make the number 'big' enough), so we can split the binary number like this:

0001 0110

The left group, 0001, is equal to 1.
The right group, 0110, is equal to 6.

Because both digits are less than ten, we can simply combine them for the answer.

0001 0110
1 6
= 16 in hexadecimal, or 16_{16}

Adding Binary Numbers

Binary numbers can be added using the same techniques you learned in grade school because they use the Arabic numeral system, which is to say that each position represents a quantity of that power of the number base. (Extra points if you actually understood that last sentence.) In grade school you learned to add decimal numbers by placing one number over another so that the columns line up, and sum each column. In base 10 this would look something like this:

```
 22
+46
 68
```

Where necessary you carry a portion to the next power of 10, as in this example:

```
 1
 27
+46
 73
```

The same techniques can be applied to adding numbers in base 2. We will use the values already shown above, but already converted to base 2.

```
 00010110    (22)
+00101110    (46)
```

We know a couple of values in base 2: 0+0=0, 1+0=1, 0+1=1, 1+1=10 (2 in base 10). Starting from the right (just like you learned in grade school) add the two values in the column, in this case 0+0 (=0).

```
 00010110    (22)
+00101110    (46)
        0
```

Moving to the next column to the left, we add 1+1 (=10). We place a 0 and carry the 1.

```
       1
 00010110    (22)
+00101110    (46)
       00
```

The next column could be difficult, but shouldn't be too bad. Add 1+1+1 (=11 in base 2). Place the right 1 and carry the left 1.

```
     11
 00010110    (22)
+00101110    (46)
      100
```

This can be continued until all columns have been summed.

```
   11111
  00010110    (22)
 +00101110    (46)
  01000100    (64*1 + 4*1)
```

We can verify the solution by converting it to decimal (which is 22 + 46 = 64 + 4 = 68).

Negative Numbers in Binary

Negative values are interesting in computer science because, by definition, they split the range of values in two, half being positive, and the other half being negative. (This isn't quite true, but its close enough for now.) Practical limitations also mean that the programmer must decide in advance how many bits will be used in a binary word. For example, an 8 bit word, which usually ranges from 0 to 255, is now -128 to 127 (Why are the two values different by 1?). This may not seem significant, but can easily become so. Thus, the first thing you as a programmer have to decide is how big your binary word is going to be.

There are at least three ways to represent negative numbers in binary. The simplest is to simply make one of the bits the 'negative' bit. This is called *signed magnitude*.
In this the left–most bit is simply turned on (made high, or 1) and the remaining bits hold the absolute value of the number. For example, 10010100 is negative because the left most bit (**1**0010100)(MSB) is a 1. This has worked but requires that the programmer always identify that the number is negative, and then treat it in a special way. At the very least this is inconvenient. There are severe limitations to this method and it is not used often but has the advantage of making it easy to see that a value is negative.

The next most common is a method called *one's complement*. In this method, all of the bits are simply inverted, i.e. 1's become 0's, and 0's become 1's. Thus, the binary number 01101001 becomes 10010110. It has an advantage similar to signed magnitude in that the left most bit is 1 when the value is negative. But few use one's complement directly because of problems of representing zero (0).

The third method, *two's complement*, also uses the left-most (most significant) bit to easily identify a negative number, and uses one's complement in part, but adds a value. This value is one (1). To convert an integer into it's negative counterpart, invert all of the bits as in one's complement, and then increment, or add the value of one.

For example, the value 27_{10} is 00011011_2 using 8 bits. The two's complement of 27, or -27, is obtained by inverting the bits, to get 11100100, and then incrementing, or adding the value one to get 11100101.

00011011	27_{10}
11100100	invert all of the bits
1	increment, or add 1
11100101	result represents -27_{10}

If the result of the increment results in a carry after the left-most bit, that bit is simply discarded with no impact on the value. That is, if carrying during addition would place a one in a bit too far to the left – more than the defined size of the 'word' - then that bit is ignored.

One of the more interesting details is that values processed by two's complement can be reversed by applying two's complement. (The technical term for that is *reflective*.)

11100101	-27_{10}
00011010	invert all of the bits
1	increment, or add one
00011011	result is back to 27_{10}

This feature is convenient because it follows the mathematical rule that –(-a) = a.

Notice that in all cases a bit has been lost from the possible magnitude of the value in order to accommodate the range of negative values. This means that a 16 bit integer can have as many as 65536 values, and either have a range of 0 to 65535, or -32768 to 32767. (Why are the two values different by 1?)

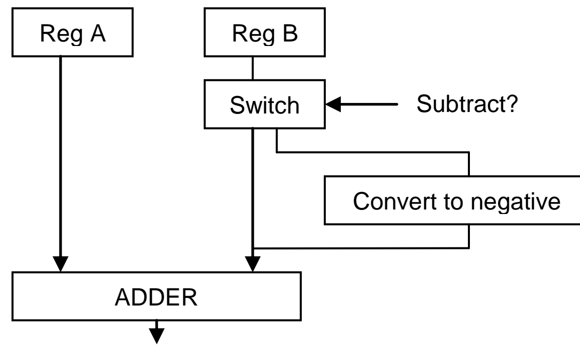
Whenever possible, try to avoid the least negative value in a word. For example, in an eight bit word (256 possible values), avoid -128. Why do you think this might be so? Try it and see for yourself.

Subtraction in Binary

Subtraction can be particularly difficult in binary. To make is easier we alter the rules of the game slightly. When considering subtraction, we commonly think of a pair of number separated by a minus sign, as in 46-21. But if you think back you may recall that another way to examine this same relationship is 46 + (-21). Thus, the easiest way to subtract two numbers is to make the subtrahend negative. We can see this if we consider the following example:

7 – 4 = 7 + (-4) (hint: the 4 is the subtrahend)

This allows us to use the addition method we already know and are comfortable with, and simply convert the second value to a negative. This simplifies many aspects of computer hardware design, though not necessarily software. In the case of hardware, a bypass is provided in the ALU (arithmetic logic unit) and a separate subtraction component in the ALU isn't needed.



Binary values created using two's complement can be used for addition as well as multiplication.

Multiplication in Binary

Binary multiplication is based on the fact that 1 * 1 = 1, and that 1 * 0 = 0. In most cases you will see column multiplication, just like you did in grade school with decimal numbers. The primary difference is that there is even *less* arithmetic when multiplying binary values. This is because of the fact that any number multiplied by 1 is equal to itself, and any value multiplied by 0 (zero) is equal to 0.

In the examples below, you can use exactly the same techniques for large number multiplication that you learned in grade school. As a point of information, 1+1+1=11 in binary.

Extra space has been added to place carry values, which are in *italics*.

Here is an example if 11 x 11:

```
    11
  x 11
   11         carry values
    11
   11_
  1001
```

Another example, 1001 x 10:

```
  1001
 x  10
  0000        no carry values
 1001_
 10010
```

One more example, 1011 x 11:

```
   1011
  x  11
1111          carry values
   1011
  1011_
100001
```

Multiplication and Number Size

A challenge facing most programmers is how 'large' to make integers. While most compilers set integer size to 32 bits (signed values range from 0 to) the programmer often has the option to define larger or smaller word size, usually in powers of two.

For example, most C compilers give the programmer the option of 8, 16, 32, or even 64 bits for the integer. The programmer must select a integer size that is large *enough*. The reason is that multiplication can *double* the word size.

For example,

Multiplying Negative Values

Negative numeric values using two's complement can be multiplied with valid results. There are just two things to keep in mind when performing this kind of operation.

First, you must determine in advance how 'large' your numeric values are expected to be. By this, I mean the number bits. For example, if you