Discussion #1: Number Representation

Written by Justin Hsia (6/20/2011)

Base Representation:

A number is represented in a particular *base* by a series of *digits*. Each digit is typically an alphanumeric character and represents a number between 0 and $base - 1$. We are taught to think in base 10 (decimal); computers think in base 2 (binary). In decimal, the value of the i^{th} digit d in base b is $d \times b^{i-1}$. The power is $i - 1$ because the first digit ($i = 1$) is for $b^0 = 1$. Alternatively, you can use $d \times b^i$ if you declare the right-most digit to be the 0^{th} digit. The full number is the sum of all of the digits. In the following examples, the subscript after a number is the base it is being represented in (no subscript means base 10):

$$1011_{\text{two}} = 2^3 + 2^1 + 2^0 = 11_{\text{ten}} = 1 \times 3^2 + 2 \times 3^0 = 12_{\text{three}}$$

$$-1234_{\text{five}} = -1 \times 5^3 - 2 \times 5^2 - 3 \times 5^1 - 4 \times 5^0 = -194$$

Though not shown, all numbers can be assumed to have an infinite number of *leading zeros*, which would appear to the left of all the shown digits.

Note: We're still in the realm of pure math here, so the negative sign sits separately from the number and does its own thing.

Computer Bases:

Computers store everything as either high (1) or low (0) electronic signals, so they naturally use base two (*binary* or *bin* for short). Each binary digit is called a *bit*. Collections of bits can be combined into bases that are powers of two. Commonly used collections are three bits (base 2^3 = base 8, which is called *octal* or *oct*) and four bits (base 2^4 = base 16, which is called *hexadecimal* or *hex*). P&H uses the aforementioned subscript notation for bases, in C we use the following prefixes:

bin – '0b', oct – '0', hex – '0x'

Decimal	Binary	Octal	Hexadecimal		Decimal	Binary	Octal	Hexadecimal
0	0b0000	000	0x0		8	0b1000	010	0x8
1	0b0001	001	0x1		9	0b1001	011	0x9
2	0b0010	002	0x2		10	0b1010	012	0xa
3	0b0011	003	0x3		11	0b1011	013	0xb
4	0b0100	004	0x4		12	0b1100	014	0xc
5	0b0101	005	0x5		13	0b1101	015	0xd
6	0b0110	006	0x6		14	0b1110	016	0xe
7	0b0111	007	0x7		15	0b1111	017	0xf

Signed Numbers:

Now we get into the practical implementation of numbers on computers. We need some way of representing both positive and negative numbers with just 0's and 1's. We have the following **number representations**, which define how we interpret collections of bits on a computer into numbers. For the following table, assume we are looking at a collection of i bits. The left-most bit is called the *most significant bit* (MSB) and the right-most bit is called the *least-significant bit* (LSB).

	Unsigned	**Sign & magnitude**	**1's complement**	**2's complement**
Most (+) #	$2^i - 1$	$2^{i-1} - 1$	$2^{i-1} - 1$	$2^{i-1} - 1$
Most (+) # (bin)	`0b111…1`	`0b011…1`	`0b011…1`	`0b011…1`
Increment (+) #	+ 1	+ 1	+ 1	+ 1
Most (−) #	0	$-(2^{i-1} - 1)$	$-(2^{i-1} - 1)$	-2^{i-1}
Most (−) # (bin)	`0b000…0`	`0b111…1`	`0b100…0`	`0b100…0`
Increment (−) #	N/A	- 1	+ 1	+ 1
Zero (bin)	`0b000…0`	`+0 = 0b000…0` `-0 = 0b100…0`	`+0 = 0b000…0` `-0 = 0b111…1`	`0 = 0b000…0`
Negation procedure	Doesn't do negative #s	Flip the sign bit (MSB)	Flip all of the bits	Flip all of the bits and add 1
Sign extension	Add leading zeros	Add leading zeros, move old MSB to new MSB	Copy old MSB into leading bits	Copy old MSB into leading bits

General things we like in our number representation:

- Having about equal positive and negative numbers (unsigned fails this)
- Zero is represented with all zero bits
- Having only one zero (sign & magnitude and 1's complement fail this)
- Incrementing positive and negative numbers the same way (sign & magnitude fails this)

Because of this, 2's complement is now used ubiquitously. Unsigned is useful in certain cases and still allowed if specially declared.

Note: Although not specifically designed to be this way, the MSB in a 1's complement or 2's complement number can still be thought of as a sign bit.

Powers of Two

With the rapid growth of computing, we often need to specify very large powers of 2. There are a very nice set of prefixes to allow us to do this rapidly!

Note: The standard prefixes such as kilo-, mega-, and giga- mean different things in different contexts. In the SI system, they mean powers of $10^3 = 1000$. When talking about computer-related quantities, they often refer to powers of $2^{10} = 1024$. To avoid this confusion, a relatively recent (1999) set of prefixes have been defined to unambiguously refer to powers of 1024.

The following table is taken from http://en.wikipedia.org/wiki/Binary_prefix:

Prefixes for **bit** and **byte** multiples

Decimal (SI)			Binary (IEC)		
Value	**Symbol**	**Full**	**Value**	**Symbol**	**Full**
1000	k	kilo	1024	Ki	kibi
1000^2	M	mega	1024^2	Mi	mebi
1000^3	G	giga	1024^3	Gi	gibi
1000^4	T	tera	1024^4	Ti	tebi
1000^5	P	peta	1024^5	Pi	pebi
1000^6	E	exa	1024^6	Ei	exbi
1000^7	Z	zetta	1024^7	Zi	zebi
1000^8	Y	yotta	1024^8	Yi	yobi

The names come from shortened versions of the original SI prefixes and "bi" is short for "binary," but pronounced "bee."

Because the binary prefixes are powers of 2^{10}, we can convert as follows:

2^{XY} means…

$Y = 0 \Rightarrow 1$
$Y = 1 \Rightarrow 2$
$Y = 2 \Rightarrow 4$
$Y = 3 \Rightarrow 8$
$Y = 4 \Rightarrow 16$
$Y = 5 \Rightarrow 32$
$Y = 6 \Rightarrow 64$
$Y = 7 \Rightarrow 128$
$Y = 8 \Rightarrow 256$
$Y = 9 \Rightarrow 512$

$+$

$X = 0 \Rightarrow 0$
$X = 1 \Rightarrow$ kibi
$X = 2 \Rightarrow$ mebi
$X = 3 \Rightarrow$ gibi
$X = 4 \Rightarrow$ tebi
$X = 5 \Rightarrow$ pebi
$X = 6 \Rightarrow$ exbi
$X = 7 \Rightarrow$ zebi
$X = 8 \Rightarrow$ yobi

$+$ bits/bytes

Examples: 2^{33} bits is 8 gibibits!

To hold 13.2 TiB of memory, you would need a 44-bit address space ($2^{44} = 16$ TiB).

For possible mnemonics to help you remember the order of these prefixes, see:
http://inst.eecs.berkeley.edu/~cs61c/fa06/mnem.html

Or use one of the ones we came up with in discussion!