Discussion 1 Richard Guo
Number Representation 01/21/09

1. **Administrative Stuff**

Email: rfguo@berkeley.edu
Lab Times: WF 11-2, 2-5 PM in 271 Soda

- Unix computer account forms are given out in lab
- Submit HW by placing the files in a directory corresponding to the assignment name and typing "submit <assignment_name>" inside the directory

2. **Keys to Success**

- Use/abuse your TAs, OHs, labs, newsgroup
- Keep up with the material
- Practice coding and on old exams
- Overview – the major topics:
  - Number Representation
    - Interconverting binary, base10, hex
    - Signed vs unsigned
    - Floating point representation
  - The C Language
    - Syntax
    - Pointers
    - Memory management
    - Compiler, assembler, linker, debugger
  - MIPS and Fun with Assembly
    - Types and effects of assembly instructions
    - Tracing through and creating assembly code
  - Digital Logic
    - Truth tables, state elements, logic elements
  - CPU Design
    - Overview and control
    - Pipelining
    - Design and build your own!
  - Cache, VM, I/O

3. **Resources**

- Course webpage: http://inst.eecs.berkeley.edu/~cs61cl/
- Last sem's webpage: http://inst.eecs.berkeley.edu/~cs61c/sp08/
- Newsgroup: ucb.class.cs61c (use http://inst.eecs.berkeley.edu/~webnews)
- HKN site for old exams: http://hkn.eecs.berkeley.edu/cgi-bin/exam-query.cgi?cs61C
- My webpage: http://inst.eecs.berkeley.edu/~cs61cl-tb

4. **Solved Problems**

- **How do we represent numbers on a computer?**

Information in computers are stored in binary format. Binary is a base system (Base 2), as opposed to our Base 10 system. There is also a Base 16 system called Hexadecimal that concatenates binary

numbers to be easier to read by grouping 4 “bits” into a “nibble”. Hex uses A, B, C, D, E, F to represent 10, 11, 12, 13, 14, 15, respectively. Some terminology:

Bit = unit of storage, can be 0 or 1
Nibble = 4 bits, equivalent to 1 hexadecimal digit
Byte = 8 bits (common unit of measurement)
Word = 32 bits (common storage format for integers and such)

Conversion from a number, say 1254 from Base 10 to Binary:
Largest exponential of 2 that is smaller than 1256 is 2^10 = 1024. Difference is 232.
Next largest exponential of 2 is 2^7 = 128. Difference is 104... repeat until nothing's left
Write out total = 0b10011100110

Convert this number to Hex by grouping 4 bits at a time starting from the right hand side.
We get: 0100 1110 0110 = 0x4E6

Test to see if this is equivalent to the original: 4*16^2 + 15*16^1 + 6*16^0 = 1024+224+6 = 1254

Note: In general, it takes log2(M) to represent M things.

- **What is kibi/mebi and how is it different from kilo/mega?**

Kilo = 10^3
Kibi = closest in binary = 1024

In addition to the table on the right (by David Jacobs and Matt Johnson),

IEC Prefixes

Name	Abbr	Factor
Kibi	Ki	2^{10} = 1,024
mebi	Mi	2^{20} = 1,048,576
gibi	Gi	2^{30} = 1,073,741,824
tebi	Ti	2^{40} = 1,099,511,627,776
pebi	Pi	2^{50} = 1,125,899,906,842,624
exbi	Ei	2^{60} = 1,152,921,504,606,846,976
Zebi	Zi	2^{70} = 1,180,591,620,717,411,303,424
yobi	Yi	2^{80} = 1,208,925,819,614,629,174,706,176

- **That's all fine and dandy, but how do we store negative numbers?**

First suggestion: *Sign and magnitude*

Unsigned integers (4 byte, 32 bit, 1 word) in C have a maximum capacity of 2^32 - 1, or 4294967295. If we sacrifice the leading bit and use it to represent the sign (0 for positive, 1 for negative), we have 31 bits for data, leading to a max of 2147483647 and a minimum of -2147483647. This form is easy to read. Just calculate the magnitude from the lower 31 bits and use the upper one to determine sign. However, it has weaknesses in that there are two zeroes (+0 and -0), and adding 1 to 2147483647 results in 0, a weird form of *overflow*. (Prove this to yourself by writing out the bits)

First improvement: *One's complement*

Positive integers stay as they are in sign and magnitude. Negatives are generated by taking the positive number and flipping all the bits.

Example: 01011011 = 91 => 10100100 = -91

This form becomes harder to read, but we get rid of the annoying overflow. Now incrementing from 2147483647 gives -2147483647. This numbering system *wraps around*. Nice! Incidentally, we also preserve the property that the leading bit determines whether the number is positive or negative.

However, we still have the annoying case of two representations of 0.

Final compromise: *Two's complement*

Keep everything the same as one's complement, but subtract 1 from all the negative numbers. This moves the range of numbers represented in a word to -2147483648 to 2147483647. We have eliminated the double zero problem.

Take an example: 0111011110 = 478 => 1000100010 = -478

Converting backwards, we do the same. Invert the bits of -478 and add 1 to the result. Check for yourself.

5. **More Problems (solutions will be on *next* discussion's handout)**

Create a table of numbers from 0 to 15 and their representation in binary, decimal, and hex.

N bits allows you to store a maximum unsigned integer of M. How many more bits are needed to store a max of 15M?

Recently, x86 CPU architectures moved to 64 bits. What is the range of integers that a 64 bit signed integer using 2's complement is able to store?

Write 12345 (base 10) in octal representation (3 bits per digit or base 8).

Why do we prefer to use hex rather than octal?

6. **Intro to C (if we have time...)**

C is different from Scheme in that it is a compiled rather than interpreted language. It is processed into machine code by a compiler. The machine code is specific to the CPU architecture you compile it for. Code compiled for one machine can “generally” only run on machines of the same CPU architecture (e..g x86, SPARC, POWER, ARM). Compiled code is faster to execute and is easier to obfuscate. Sample C program (more than just hello world):

```
#include <stdio.h>

int main (int argc, char* argv[])
{
      int n=0;
      int *n_pt = &n;
```

```
	char* message = "Hello World";
	short short_number = 12345;
	char character = 'a';
	float floating = 12345;
	double more_floating = 0.123454356;

	printf ("%s\n", message);
	printf ("%d\n", n);
	n = 50;
	printf ("%d\n", *n_pt);
	printf ("Integer: %d\n", sizeof(n));
	printf ("Pointer: %d\n", sizeof(n_pt));
	printf ("String: %d\n", sizeof(message));
	printf ("Short: %d\n", sizeof(short_number));
	printf ("Character: %d\n", sizeof(character));
	printf ("Float: %d\n", sizeof(floating));
	printf ("Double: %d\n", sizeof(more_floating));

	return 0;
}

Output:
Hello World
0
50
Integer: 4
Pointer: 8
String: 8
Short: 2
Character: 1
Float: 4
Double: 8
```

Printf output parameters:

specifier	Output	Example
c	Character	a
d or i	Signed decimal integer	392
e	Scientific notation (mantise/exponent) using e character	3.9265e+2
E	Scientific notation (mantise/exponent) using E character	3.9265E+2
f	Decimal floating point	392.65
g	Use the shorter of %e or %f	392.65
G	Use the shorter of %E or %f	392.65
o	Signed octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
p	Pointer address	B800:0000
n	Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored.	