COMPUTER ORGANIZATION REVIEW

BINARY NUMBERS AND CONVERSIONS

Representing numbers

- What is the correct way to represent numbers?
  - *Decimal (base 10)*
  - *Binary (base 2)*
  - *Octal (base 8)*
  - *Hexadecimal (base 16)*
  - *Base 7 or Base 12*
  - *Roman numerals*

Representing numbers

- What is the correct way to represent numbers?
  - *Decimal (base 10)*
  - *Binary (base 2)*
  - *Octal (base 8)*
  - *Hexadecimal (base 16)*
  - *Base 7 or Base 12*
  - *Roman numerals*
- There is no ***correct*** way to represent numbers.
  - A binary number is not just another way to write a decimal number…

Base 10 is not a “universal truth”

- Moses did not descend from Mt. Sinai with a base 10 number system, and base 10 was not given to us from God.
- For our purposes, numbers are adjectives
- Numbers are values, and the number of hash marks on a tape (as in a Turing machine) is just as good a way to represent a number as any numeric grapheme or system of glyphs.
- Fact to learn in this class
  - *No complete system is decidable, and no decidable system is complete.– Godel*

Counting to 15 in decimal, binary, and hexadecimal (hex)

- Note that Binary and Decimal formats will be used for arithmetic in this class, but hexadecimal will only be used as a way to format binary to read it more easily.

Definitions of data sizes

- 8 bits = 1 byte
- 4 bit = ½ byte = 1 nybble (1 hex character)
- 1 word = the size of data transfer in a computer (typically 16, 32, or 64 bits).
- Half word = ½ of a word
- Double word = 2 words
- NOTE: later these will also have meaning concerning alignment

Powers of 2

Power of 2	Base 10 value	Power of 2	Base 10 value
0	1	6	64
1	2	7	128
2	4	8	256
3	8	9	512
4	16	10	1024
5	32	11	2048

Meaning of a base 10 and base 2 numbers

- $4275_{10} = 4*10^3 + 2*10^2 + 7*10^1 + 5*10^0 = 4000 + 200 + 70 + 5$
- $1001\ 0010_2 = 1 * 2^7 + 1 * 2^4 + 1 * 2^1 = 128 + 16 + 2 = 146_{10}$

Base 2 powers of 10

2^{10}	Kilo
2^{20}	Mega
2^{30}	Giga
2^{40}	Terra
2^{50}	Penta
2^{60}	Exa

How do use these values

- $2^{16} = 2^{10} * 2^{6}$ = 64 Kbits
- $2^{32} = 2^{30} * 2^{2}$ = 4 Gbits
- $2^{24} = 2^{20} * 2^{4}$ = 16 Mbits

Converting Decimal to Binary (method 1)

- Easier, since only 0 or 1 of any power can be in the number.
- Check for the largest power of 2 in the number, and work down until there are no powers of 2.

Converting Decimal to Binary (method 2)

- Divide by 2 to find the number of 1’s in the number (remainder is 1 if there is a 1, 0 otherwise). Quotient is the original number with 1’s removed.
- Divide again by 2 to find the number of 2’s in the number (remainder is 1 if there is a 2, 0 otherwise). Quotient is the original number with 1’s and 2’s removed.
- Repeat for until the Quotient is 0.

Adding binary numbers

- When adding two numbers, there is a carry between the digits (other than the first position)
- Adding any three binary digits (X, Y, and a Carray in , or Cin) results in a Sum and a Carry Out (Cout). Specifically

Input			Output	
X	Y	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Addition

```
   11011010
+  00011011
   11110101
```

TWO’S COMPLEMENT

Integer Data Types

- All integer data types (byte, short, int, long, etcetera) are typed as 2’s complement (e.g. the values are stored as 2’s complement, and all operations are for 2’s complement)
- Two issues with 2’s complement
  - *2’s complement operation*
  - *2’s complement type*

2’s complement operation

- A 2’s complement operation converts a number to its negative value (it is a negation operation). Specifically:
  - *It converts positive->negative value*
  - *It converts negative->positive values*
- A 2’s complement operation is performed as follows:
  - *Invert all the bits in the number (0’s->1, 1->0’s)*
  - *Add 1 to the number*
  - *Example 110101100 -> 00100100*

Add 5 + 6 in 4 bit two's complement

  0101
+ 0110
  1011
= -5 (NOT 11)

Add 5 + 6 in 5 bit two’s complement

00101

+ 0<u>0110</u>

01011

= 1011 (answer is correct)

Check for overflow

- Carry-in and Carry-out from the last bit
  - *equal => no overflow*
  - *different => overflow*

Subtraction

- Add the negative
  - *7 – 5 = 7 + (-5)*
  - *Invert the bits on the second number, add two values, and then add 1*

BOOLEAN AND SHIFT OPERATIONS

Multiplication

- Left shift is multiplication by 2
  - $12\ (1100_2) \ll 1 = 11000_2\ (24)$
  - $12\ (1100_2) \ll 2 = 110000_2\ (48)$
  - $3\ (11_2) \ll 3 = 11000_2\ (24)$
- Can multiply any two numbers by bit shifts and adds
  - $4 * 5 = 4 * (4+1) = 100_2 \ll 2 + 100_2 \ll 0 = 10000_2 + 100_2 = 10100_2\ (20)$

Division

- Division is right shift 1 bit for each power of 2
- Only works for division by powers of 2

Shift Operators

- Left Logical Shift - << n (shift n bits left, padding with 0)
- Right Logical Shift - >>> n (shift n bits rights, padding with 0)
- Arithmetic Right Shirt - >> n (shift n bits rights, padding with the sign bit)
  - *Does not exist in C that I am aware of*
- ~ (tilde) is the complement operator
- &, |, ^ are bitwise AND, OR, and XOR
- Example

```
int a = 1
a = a << 2; // should be  8
```

Example

int x = 1;   // 000… 01

x = x << 31 // 100..00

x = x >> 32 // 111….11, In Java, in C this is implementation dependent

x = x >>> 1 // 000…00, not used in C

x = 0xff

x = x >>> 31 // 000…01, not used in C

ASCII table

- http://www.asciitable.com/

Example

- 0x20 (0010 0000) bit is the difference between upper and lower case in ASCII
- Therefore

```
 char c = 0x20
        ic <- some character
 ic = ic | c ; // Converts to lower case
 ic = ic & ~c // Converts to upper case
```

Example

- Numbers start at 0=0x30 and 9 = 0x39, so “inum = ic – ‘0’ “ will convert input character to number
- Convert a string to a number

```
number = 0;
for (int j = 0; j < s.length; i++)
{
    ic = s.charAt(j);
    ic = ic – ‘0’
    number = number * 10 + ic
}
```

BOOLEAN ALGEBRA

Boolean Algebra

- See notes in separate file

STORAGE BOUNDARIES

Storage Boundaries

- Computers are wired machines!
  - *Not all bits are wired (connected) to all other bits*
  - *Bits are in groups, and wired in groups, and you must align on those groups.*
  - *If you don’t get this, just follow the rules*
- Boundaries
  - *Computers are almost all Byte aligned (I know of none that isn’t)*
  - *MIPS is Byte Addressable (as are most computers)*
  - *Every address is byte addressable (addresses 0, 1, 2, 3, …)*
  - *Every even address is half-word addressable (addresses 0, 2, 4, …)*
  - *Every address divisible by 4 is word addressable (addresses 0, 4, 8, 12, …)*
  - *Every address divisible by 8 is double word addressable (addresses 0, 8, 16, …)*

Advantage of the rules

- The right most bit(s) of a
  - *half word address is 0*
  - *word address is 00*
  - *double word address is 000*
- Instructions always are ***<u>word</u>*** aligned. Thus the two right most bits can be, AND ARE, dropped in instructions. This has implications in branches and jumps.

Problem 1

- At what address would the following be allocated if the last address was 0x1004 020F. Assume each statement will allocate memory at the next address after 0x1004 020F (each allocation is isolated, they are not cumulative)
  - *A: .byte 7 0x10040210*
  - *B: .hw 7 0x10040210*
  - *C: .word 7 0x10040210*
  - *D: .double 7 0x10040210*
  - *E: .asciiz "Test" 0x10040210*
  - *F: .space 15 0x10040210*

Problem 2

- At what address would the following be allocated if the last address was 0x1072 020B. Assume each statement will allocate memory at the next address after 0x1041 020B (each allocation is isolated, they are not cumulative)
  - *A: .byte 7 0x1041 020C*
  - *B: .hw 7 0x1041 020C*
  - *C: word 7 0x1041 020C*
  - *D: .double 7 0x1041 0210*
  - *E: .asciiz “Test” 0x1041 020C*
  - *F: .space 15 0x1041 020C*

Problem 3

- At what address would the following be allocated if the last address was 0x1041 0200. Assume each statement will allocate memory at the next address after 0x1041 0200 (each allocation is isolated, they are not cumulative)
  - *A: .byte 7 0x1041 0201*
  - *B: .hw 7 0x1041 0202*
  - *C: .word 7 0x1041 0204*
  - *D: .double 7 0x1041 0208*
  - *E: .asciiz "Test" 0x1041 0201*
  - *F: .space 15 0x1041 0201*

Problem 4

- At what address would the following be allocated if the last address was 0x1041 020F. Assume that this is code, and you are allocating the variables one after another.

  A: .space 14 0x1041 0210

  B: .hw 7 0x1041 021D

  C: .word 7   0x1041 0220

  D: .double 7 0x1041 0228

  E: .asciiz "Test"   0x1041 0230

  F: .word 7   0x1041 0238

  G: .space 15 0x1041 023C

FLOATING POINT NUMBERS

Scientific Notation

- $n = f \times 10^e$
- f is the fraction, or mantissa
- e is the exponent
- Avogadro's Number
  - 6.0221415×10^{23}
- Number of atoms in the observable
  - between 4×10^{79} and 1×10^{81}
- Weight of an electron
  - 9×10^{-28}

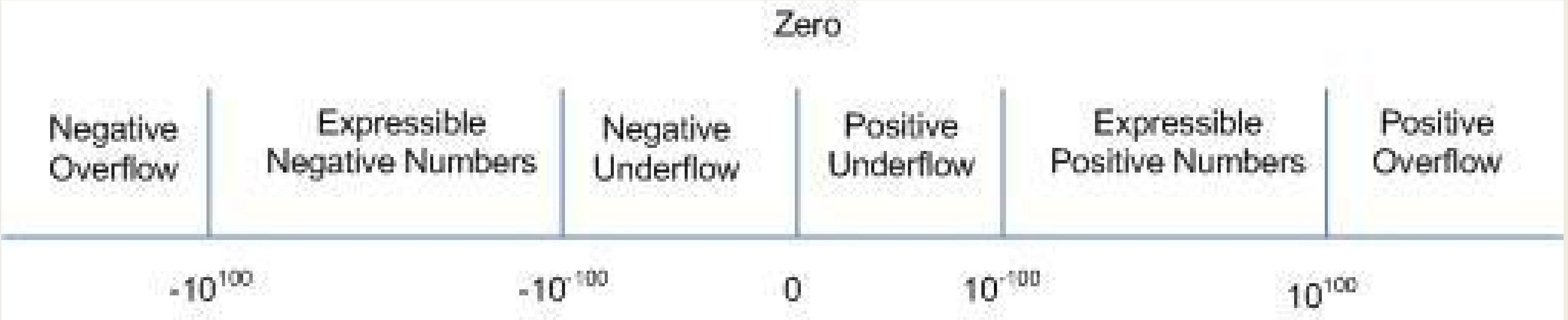
Some interesting notes about Floating Point Number

- Floating point numbers are how we express scientific notation in binary.
- There is always an error in floating point numbers
  - Remember 0.1 CANNOT be represented exactly in binary!
  - Floating point number are **not continuous**!
  - ~ 7 decimal digits precision for single
  - ~ 15 decimal digits precision for double
- The infinity of floating point number is infinitely larger than the infinity of fractions!

Facts about floating point numbers

- 7 regions
- Consider a representation which has a signed, 3 digit fraction, and a signed 2 digit exponent.

Floating Point Ranges



IEEE 754 format

- Web site to define IEEE 754 format
  - http://en.wikipedia.org/wiki/IEEE_754
  - http://en.wikipedia.org/wiki/IEEE_754-1985
- Here is a web site to calculate numbers
  - http://babbage.cs.qc.edu/IEEE-754/
- Note that you need to know HOW this site gets the answers. I do not really grade homework, so just copying answers doesn't help. I do grade exams, and you will not have this site when you sit for the exam!

IEEE 754 Format
Single Precision

Sign bit	Exponent 8 bits Excess 127	Fraction Normalized, Leading 1 dropped 23 bits

IEEE 754 Format
Double Precision

Sign bit	Exponent 11 bits Excess 127	Fraction Normalized, Leading 1 dropped 52 bits

Some Examples

- 7 (in binary and hex)
  - 0 10000001 11000000000000000000000
    - NOTE: Leading 1 is dropped!
  - 40E00000
- 0.125
  - 0 01111100 00000000000000000000000
    - Note: Fraction is 0! (leading 1 is dropped!)
  - 3E000000

Some IEEE 754 Constants

Type	**Sign and Exponent (total 9 bits)**	**Fraction**
Zero	[01] 0x00	0
Denormalized numbers	[01] 0x00	non zero
Normalized numbers	1 to 2^e -2	any
Infinities	0 0xff (positive infinity) 1 0xff (negative infinity)	0
NaNs	[01] 0xff	non zero