Lecture 5:
Data Representation

Announcements

- Project 1 due in one week
  - July 13th , 11:55pm
- Make sure to ask questions on Sakai
- Additional TA
  - Info will be posted within the next dat

Decimal Numbers

- Base 10
- 10 different digits:
  - {0 , 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Numbers written as $d_n \dots d_2 d_1 d_0$
- The decimal value is $\sum_{i=0}^{n} d_i \times 10^i$
- Example:

$$54372$$

$$= 5(10^4) + 4(10^3) + 3(10^2) + 7(10^1) + 2(10^0)$$

$$= 5(10000) + 4(1000) + 3(100) + 7(10) + 2(1)$$

Binary Numbers

- Base 2
- 2 different digits: {0,1}
- Numbers written as $d_n \ldots d_2 d_1 d_0$
- The decimal value is $\sum_{i=0}^{n} d_i \times 2^i$
- Example:

$$10110101$$

$$= 1\,(2^7) + 0(2^6) + 1(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$$

$$= 1\,(128) + 0(64) + 1(32) + 1(16) + 0(8) + 1(4) + 0(2) + 1(1)$$

$$= 181$$

- Binary Representation is used in computers
  - Easy to represent using switches (on/off)
  - Manipulation by digital logic in hardware
  - Each digit is known as a bit and 1 byte consists of 8 bits

Hexadecimal Numbers

- Base 16
- 16 different digits:
  - {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}
  - First 10 digits are the same as in decimal
  - Last 6 (values 10-15) are the first 6 letters in alphabet
- Numbers written as $d_n \ldots d_2 d_1 d_0$
- The decimal value is $\sum_{i=0}^{n} d_i \times 16^i$
- Example:

$$\begin{aligned} &FFA9 \\ &= 15(16^3) + 15(16^2) + 10(16^1) + 9(16^0) \\ &= 15(4096) + 15(256) + 10(16) + 9(1) \\ &= 65440 \end{aligned}$$

- Used as binary representation can be too verbose
- Hexadecimals typically prefixed with "0x" (ex. 0xFFF)
- Each hex digit is 4 bits long

Numbering Systems

System	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

- What is the range of value that can be represented with 1 byte (8 bits)
- Binary: 00000000_2 - 11111111_2
- Decimal: 0_{10} - 255_{10}
- Hexadecimal: 00_{16} – FF_{16}

Bit Patterns from N Bits

Number of Bits	Number of Patterns	Number of Patterns as Power of Two
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4

- Number of possible patterns with N bits $= 2^N$
- How many patterns can be formed with
  - 10 bits? $= 2^{10} = 1024$
  - 20 bits? $= 2^{20} = 2^{10} * 2^{10} = 1048576$
  - 30 bits? $= 2^{30} = 2^{10} * 2^{20} = 1073741824$
  - 40 bits? $= 2^{40} = 2^{10} * 2^{30} = 1.0995116e+12$
  - 50 bits? $= 2^{50} = 2^{10} * 2^{40} = 1.1258999e+15$
  - 60 bits? $= 2^{60} = 2^{10} * 2^{50} = 1.1529215e+18$

Unsigned Integers

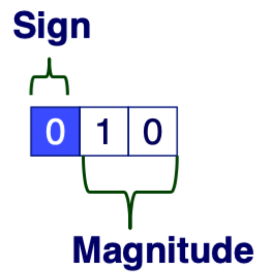
- Given a 4 bytes, what is the range of unsigned integers can we represent?
- Recall 1 byte = 8 Bits
- 8 bits = 2^8 different values
- 4 bytes = $2^{(8 \times 4)} = 2^{32}$ = 4294967296 different values
- We can represent 0 to 2^{32} -1
- Range: 0 to 4294967295

Negative Integers

- How can we represent negative integers using binary representation?
  - 9_{10} negated = -9_{10}
  - 1001_2 negated = -1001_2?
- Unfortunately a computer can only holds 0's and 1's
- Suppose you want to represent an equal number of positive and negative integers with 3 bits
  - 2^3=8 possible patterns
  - 4 positive and 4 negative numbers
- Solution: Use 1 bit to represent the sign

Sign Magnitude

- Use the leftmost bit to indicate the sign
- The remaining bits indicate the magnitude



- 3-bit register
  - Represent numbers in the range $[-(2^{n-1}-1), 2^{n-1}-1]$

000	001	010	011	100	101	110	111
0	1	2	3	-0	-1	-2	-3

Sign Magnitude (Issues)

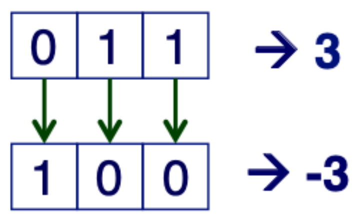
- There are some problems with using simple sign magnitude for signed integers
- Have two zeros
  - 100 (-0) and 000 (+0)
- Inconvenient for arithmetic computations

```
      010          2
  +   101      +  -1
  ---------   -------
      111         -3 (wrong!)
```

- Binary addition does not work if numbers have unlike signs
  - Would need a special subtraction algorithm
- Another solution: One's complement

One's Complement

- Represent negative numbers by complementing its positive counterpart
  - $\overline{N} = (2^n - 1) - N$
    - n is the number of bits
    - N is a positive integer
    - $\overline{N}$ is $-N$ in 1's complement
- Similar to inverting each bit of the n-bit binary number



- 3-bits register
  - Represent numbers in range $[-(2^{n-1}-1), 2^{n-1}-1]$

000	001	010	011	100	101	110	111
0	1	2	3	-3	-2	-1	-0

One’s Compliment (Benefits and Issues)

- Benefit: Binary addition works with end-around carry
  - End-around carry = add any resulting carry back into the resulting sum

```
   11              ← Carry
    010        2
+   110    +  -1
  ______    ____
    000            ← Not the correct answer
      1       +1   ← Add carry
  ______
    001        1   ← Correct answer
```

- Issue: Still have two zeros
  - 111(-0) and 000 (+0)

Two's Complement

- Represent negative numbers by complementing its positive counterpart with respect to 2^n
  - $\overline{N} = 2^{\mathrm{n}} - N$
    - n is the number of bits
    - N is a positive integer
    - $\overline{N}$ is $-N$ in 1's complement
- Simply one's complement plus one
  1. Invert each bit
  2. Add 1
- 3-bit register
  - Represent numbers in range $[-2^{n-1}, 2^{n-1}-1]$

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

  - Notice one more negative integer can be represented than positive integers

Two's Complement

- Benefits
  - Only one zero (000)
  - One more negative number can be represented
  - Arithmetic still works
- Given a two's complement n-bit value written as $d_{n-1}d_n \ldots d_1 d_0$
  - Decimal value is interpreted as $-d_{n-1}2^{n-1} + \sum_{i=0}^{n-2} d_i 2^i$
  - Works for both positive and negative numbers

Signed Integers

- Two's complement is the standard and most common way to represent signed integers
- Integer size on 32-bit and 64-bit architecture
  - 4 bytes
- What range of values by a 4 byte int?
  - 4 bytes = 32 bits
  - Represent numbers in range $[-2^{n-1}, 2^{n-1}-1]$
  - 4 byte int can represent $[-2^{32-1}, 2^{32-1}-1]$
  - -2147483648 to 2147483647

Floating Point Numbers

- Real numbers (a number that contains a fractional part)
- How do we represent real numbers in an integer computer?
    - Integers written in decimal form
        - E.g. 1, 10, 100, 1000, 10000, 12456897, etc.
    - Can also be written in scientific notation
        - E.g. 1×10^4, 1.2456897×10^7
    - What about binary numbers?
        - Works the same way: $0b100 = 0b1 \times 2^2$
- Scientific notation gives a natural way for thinking about floating point numbers
    - $0.25 = 2.5 \times 10^{-1} = 0b1 \times 2^{-2}$
- How are floating point numbers stored in computers?

IEEE floating point standard

- Most computers follow IEEE 754 standard
- Data is split up into three sections:

s	**exp**	**mantissa**

  - s: sign field determines if the number is negative (s=1)
  - exp: exponent
  - mantissa: fractional number in binary
- Value = $(-1)^S \times 2^E \times F$
  - $E = exp - (2^{(k-1)}-1)$ where k is the number exp bits
  - F = 1.<mantissa> or 0.<mantissa> (we’ll see later on)
  - $(2^{(k-1)}-1)$ is what’s known as the bias

Floating Point in C

- Three precisions
  - Single precision (32-bits) (float)
  - Double precision (64-bits) (double)
  - Extended precision (80-bit) (long double)
- 32-bit single (type float)
  - 1 bit for sign, 8 bits for exponent, 23 bits for fraction
  - Have 2 zero's
  - Range is approximately -10^{38} to 10^{38}
- 64-bit double precision (type double)
  - 1 bit for sign, 11 bits for exponent, 52 bits for fraction
  - Majority of new bits for fraction
    - Allows for higher precision
  - Range is approximately -10^{308} to 10^{308}

Floating Point Numerical Values

- Three different cases
- Normalized values
  - When exp is not all 0s or not all 1s
  - $E = exp - (2^{(k-1)}-1)$
  - F = 1.<mantissa>
- Denormalized Values
  - When exp is 0
  - $E = 1 - (2^{(k-1)}-1)$ -> (e.g for float: 1- 127 = -126)
  - F = 0.<mantissa>
  - Represents 0 and values very close to 0
- Special Values
  - When exp all 1's
  - When mantissa is all 0's
    - Positive or negative Infinity ($\pm\infty$) depending on sign
  - Else when mantissa is not all 0's
    - NaN = Not a number

Converting Floating Point To Decimal

- Recall: Single precision (32-bits) (float)
  - 1 bit for sign, 8 bits for exponent, 23 bits for fraction
- Recall: Value = $(-1)^S \times 2^E \times F$
  - Where $E = exp - (2^{(k-1)}-1)$ and F = 1.<manitissa>
- Example:

1	**10000011**	**01010100000000000000001**

  - S = 1, so negative number
  - $E = 131 - (2^{8-1}-1) = 131 - 127 = 4$
  - F = 1. 01010100000000000000001
  - $(-1)^1 \times 2^4 \times 1.01010100000000000000001$
  - = -10101.01100000000000000001
  - = - 21.3750009536743164062

Converting Decimal to Floating Point

- Convert 5.625 to 32-bit floating point
- Convert decimal to binary
  - Note: $2^{-1} = 0.5$, $2^{-2} = 0.25$, $2^{-3} = 0.125$
  - 5.625 = 101.101
  - $101.101 = 1.01101 \times 2^2$
- Add bias to get exp
  - 2 + 127 = 129
  - 129 in binary = 10000001
- Positive value so signed bit is 0
- We get S=0, E=10000001, M=01101

0	**10000001**	**01101000000000000000000**

Rounding in Floating Point

- Round to the nearest number
- Example:
  - Assume 4 bit mantissa
  - 1.10011001
  - Which one is closer? 1.1001 or 1.1010?
  - Round up to 1.1010 because it's closer
- What happens if tie?
  - Round to even number number
- Example:
  - 1.10011
    - If we round down we get an odd number 1.1001
    - Round up to even number 1.1010
  - 1.10001
    - If we round up we get 1.1001 which is not even
    - Round down to even number 1.1000