Numbers, numbers everywhere!

Computer Science 250
Lecture 2

Everyone should have access

TA Office hours posted

- Parker is Head TA
- Grade issues?
  - Talk to TA who graded it first
  - Head TA (Parker) second
  - Me third
  - Skip up the “Chain” in unusual circumstances

Homework 1 posted soon

- Due Sept 14

Last Time in CS250….

Who can remind us what we talked about last time?

- Course policies
- Why to take this course
- Everything is a number ➔ it is all 1s and 0s
- Number representations
  - Decimal
  - Binary
  - Hexadecimal (“hex”)

Converting Numbers

Hex to Binary:
- Expand each hex digit to 4 bits:
  \[ \text{0xFF1} = 11111100001 \]

Binary to Hex:
- Group up 4 bits into a digit:
  \[ 11010110011 = 0x6B3 \]

Binary to Decimal:
- Add up values of bits with 1:
  \[ \text{1001101} = 64 + 8 + 4 + 2 + 1 = 77 \]

Last Time in CS250….

Who can remind us what we talked about last time?

Suppose I want to convert 4872 to binary

Recall from last time and try it out...
Converting Numbers

Converting Decimal to Binary
Suppose I want to convert 4872 to binary
Recall from last time and try it out...

<table>
<thead>
<tr>
<th>4872</th>
<th>4096</th>
<th>2048</th>
<th>1024</th>
<th>512</th>
<th>256</th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 4096</td>
<td>2048</td>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>776</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

What about negative numbers?

This works great for N >= 0…
But what about negative numbers?
• Sometimes we don’t need them: unsigned number
• Sometimes kind of important: signed number

Many options:
• Sign magnitude
• 1’s complement
• 2’s complement
• Biased
Sign Magnitude

Use leftmost bit for + (0) or – (1):
6-bit example (1 sign bit + 5 magnitude bits):
+17 = 010001
-17 = 110001

Pros:
• Conceptually simple
• Easy to convert

Cons:
• Harder to compute (add, subtract, etc) with
• Positive and negative 0: 000000 and 100000

1’s Complement Representation for Integers

Use largest positive binary numbers to represent negative numbers
To negate a number, invert (“not”) each bit:
0->1
1->0

Cons:
• Still two 0s
• Still hard to compute with

2’s Complement Representation for Integers

To negate, flip bits, add 1:

Pros:
• Easy to compute with
• One representation of 0

Cons:
• More complex negation
• Extra negative number (-8)

Ubiquitous choice

Binary Math : Addition

Suppose we want to add two numbers:

00011101
+ 00101011

How do we do this?

• Let’s revisit decimal addition
• Think about the process as we do it

First add one’s digit 5+2 = 7
Binary Math: Addition

Suppose we want to add two numbers:

\[ \begin{array}{c}
00011101 \\
+ 00101011 \\
\hline
27
\end{array} \]

First add one's digit 5+2 = 7
Next add ten's digit 9+3 = 12 (2 carry a 1)

Suppose we want to add two numbers:

\[ \begin{array}{c}
00111011 \\
+ 00101011 \\
\hline
232
\end{array} \]

First add one's digit 5+2 = 7
Next add ten's digit 9+3 = 12 (2 carry a 1)
Last add hundred's digit 1+6+2 = 9

Back to the binary:
First add 1's digit 1+1 = 2 (0 carry a 1)

Suppose we want to add two numbers:

\[ \begin{array}{c}
1111111 \\
00011101 \\
+ 00101011 \\
\hline
01001000
\end{array} \]

Can check our work in decimal

Suppose we want to add two numbers:

\[ \begin{array}{c}
1111111 \\
00011101 \\
+ 00101011 \\
\hline
01001000
\end{array} \]

You all finish it out...
Binary Math: Addition

What about this one:

\[
\begin{align*}
01011101 \\
+ 01101011
\end{align*}
\]

But... that can't be right?
- What do you expect for the answer?
- What is it in 8-bit signed 2's complement?

Integer Overflow

Answer should be 200
- Not representable in 8-bit signed representation
- No right answer

Called Integer Overflow
- Signed addition: CI != CO of last bit
- Unsigned addition: CO != 0 of last bit

Real problem in programs
- SNES SimCity Cheat
- Can easily imagine more serious cases
- Awareness can make you better programmer

Subtraction

2's complement makes subtraction easy:
- Remember: \( A - B = A + (-B) \)
- And: \( -B = \overline{B} + 1 \)
  - that means flip bits ("not")
- So we just flip the bits and start with CI = 1
- Later: No new circuits to subtract (re-use add)

\[
\begin{align*}
0110101 & \rightarrow 0110101 & 1 \\
- 1010010 & + 0101101
\end{align*}
\]

What About Non-integer Numbers?

There are infinitely many real numbers between two integers
Many important numbers are real
- \( \pi = 3.145... \)
- \( \frac{1}{2} = 0.5 \)

How could we represent these sorts of numbers?
- Remember, we only have integers...
- Everyone think for a few minutes
- Feel free to brainstorm with your neighbors...

Option 1: Fixed point

Use normal integers, but \((X \times 2^k)\) instead of \(X\)
- Example: 32 bit int, but use \(X \times 65536\)
- \(3.1415926 \times 65536 = 205887\)
- \(0.5 \times 65536 = 32768\), etc..

Pros:
- Addition/subtraction just like integers ("free")

Cons:
- Mul/div require renormalizing (divide by 64K)
- Range limited (no good rep for large + small)

Can be good in specific situations
Option 2: Rational Numbers

Represent each number as fraction: X / Y

- X and Y are integers (say 16 bits each)
- 0.5 = (1,2)
- 3.1415926 = (31415, 1000)

Pros:
- Represent a lot of numbers
- Precision in some cases (e.g., 1/3)

Cons:
- Hard to work with
- Multiple representations for many numbers

Can we do better?

Think about scientific notation for a second:

- For example: $6.02 \times 10^{23}$

Real number, but comprised of ints:

- 6 generally only 1 digit here
- 2 any number here
- 10 always 10 (base we work in)
- 23 can be positive or negative

Can we do something like this in binary?

Option 3: Floating Point

How about:

+/- X.YYYYYY * 2^+-N

Big numbers: large positive N
Small numbers (<1): negative N
Numbers near 0: small N

This is “floating point” : most common way

IEEE single precision floating point

Specific format called IEEE single precision:

+/- 1.YYYYY * 2^{(N-127)}

“float” in Java, C, C++,...

Assume X is always 1 (save a bit)
1 sign bit (+ = 0, 1 = -)
8 bit biased exponent (do N-127)
Implicit 1 before binary point
23-bit mantissa (YYYYY)

Binary fractions

1.YYYY has a binary point

- Like a decimal point but in binary
- After a decimal point, you have
  - tenths
  - hundredths
  - Thousandths
- ... 

So after a binary point you have...

Binary fractions

1.YYYY has a binary point

- Like a decimal point but in binary
- After a decimal point, you have
  - Tenths
  - Hundredths
  - Thousandths
- ... 

So after a binary point you have...

- Halves
- Quarters
- Eights
- ...
**Floating point example**

Binary fraction example:

\[101.101 = 4 + 1 + \frac{1}{2} + \frac{1}{8} = 5.625\]

For floating point, needs normalization:

\[1.01101 \times 2^2\]

Sign is +, which = 0

Exponent = 127 + 2 = 129 = 1000 0001

Mantissa = 1.011 0100 0000 0000 0000 0000

---

**Floating Point Representation**

Example:

What floating-point number is:

\[0xC1580000?\]

---

**Answer**

What floating-point number is

\[0xC1580000?\]

\[1100 0001 0101 1000 0000 0000 0000 0000\]

\[X = 1100\ 0001\ 0101\ 1000\ 0000\ 0000\ 0000\ 0000\]

\[s = 1\]

\[E = 0101\ 1000\ 0000\ 0000\ 0000\ 0000\]

\[F\]

Sign = 1 which is negative

Exponent = (128+2)-127 = 3

Mantissa = 1.1011

\[-1.1011 \times 2^3 = -1101.1 = -13.5\]

---

**Trick question**

How do you represent 0.0?

• Why is this a trick question?

• 0.0 = 00000000

• But need 1.XXXXX representation?

---

**Trick question**

How do you represent 0.0?

• Why is this a trick question?

• 0.0 = 00000000

• But need 1.XXXXX representation?

Exponent of 0 is denormalized

• Implicit 0. instead of 1. in mantissa

• Allows 0000….0000 to be 0

• Helps with very small numbers near 0

Results in +/- 0 in FP (but they are “equal”)
Other weird FP numbers

Exponent = \text{1111 1111} also not standard
- All 0 mantissa: \pm \infty
- \pm 1/0 = \pm \infty

Non zero mantissa: Not a Number (NaN)
\text{sqrt(-42)} = \text{NaN}

Floating Point Representation

Double Precision Floating point:
- 64-bit representation:
  - 1-bit sign
  - 11-bit (biased) exponent
  - 52-bit fraction (with implicit 1).

"double" in Java, C, C++, ...

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11-bit</td>
<td>52-bit</td>
</tr>
</tbody>
</table>

Ints and Floats

```java
int x = 1930754307;
float f = x;
System.out.println(x);
x = f;
System.out.println(x);
```

Is this Java Legal?
- Java worries you might chop off a fraction
- Requires a cast
- Does not worry about \( f = x \) why not?

Program output:
1930754307
1930754304
Why?

```
int x = 1930754307;
float f = x;
System.out.println(x);
x = (int) f;
System.out.println(x);
```

Program output:
1930754307
1930754304
Hint
0x7314f903
0x7314f900
Why?
Danger: floats cannot hold all ints!

Many programmers think:
- Floats can represent all ints
- NOT true

First summer internship I had:
- Need some floats and some ints: just use floats!
- Bug in their code!
- Other developers shocked as I demonstrated problem...

Doubles can represent all 32-bit ints
(but not all 64-bit ints)

Next time

Memory:
- Where we store all these numbers

Pointers:
- Variables who value is an address

C:
- Low-level programming language
  (let's us shoot ourselves in the foot in new ways)