Define overflow

“As the famous Gandhi in Civ, when things go past where they should, they just loop back to the other end.”

“Overflow is when the number cannot be represented using a fixed bit representation.”

“Overflow is when the "theoretical value" of adding two binary digits is exceeded because its greater than what the binary system for that many digits can hold at max, i.e. too big of a number in too small of a space.”

“When two unsigned number do the operation, the overflow means the bit length exceed the allowed length. When two two's complement do the operation, overflow usually means getting an impossible result, such as getting a negative when adding two positives.”
Define Overflow

“I don't understand 2's complement addition and when it overflows and when it doesn't, and also how some of the bit shifting was "bad" and some was "good". Do you mind explaining it in class?”

“This is all so confusing im going to cry”
OTCR Consulting

Who We Are

20 years
75 students

300+ clients
100% job placement

What We Do

Business Strategy
Market entry, competitor analysis, acquisitions, product analysis and more

Tech Strategy
Analysis of new technologies and potential applications, and more

Tech Implementation
Development of full-stack websites and mobile applications

Who We Work With

Microsoft
KRAFT
Motorola
TEACH FOR AMERICA
STI
Graybar
bp

How To Join

Info Nights
January 17th – 7 pm
Engineering Hall 106B1
January 22nd – 7 pm
Lincoln Hall 1002

Case Training
January 24th – 7 pm
David Kinley Hall 123

Applications Due
Due January 25th
By 11:59 pm

www.otcr.illinois.edu
OTCR Consulting
@otcr_consulting
Number Systems
(in Binary)
Stored state bits can be interpreted with many different encoding schemes

Computer can do 2 things
1) Store state (How do we interpret stored bits?)
2) Manipulate state
233 in one slide!

- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
  1. You will build a simple computer processor. 
     Build and create state machines with **data**, **control**, and **indirection**
  2. You will learn how high-level language code executes on a processor. 
     Time limitations create dependencies in the state of the processor
  3. You will learn why computers perform the way they do. 
     Physical limitations require **locality** and **indirection** in how we access state
  4. You will learn about hardware mechanisms for parallelism. 
     **Locality**, **dependencies**, and **indirection** on performance enhancing drugs

- We will have a SPIMbot contest!
Today’s lecture

- Representing things with bits
  - $N$ bits gets you $2^N$ representations
- Review: Unsigned binary number representation
  - Hexadecimal notation
- Binary Addition & Bitwise Logical Operations
  - Every operation has a width
- Two’s complement signed binary representation
- Bitshift operations
Binary representations review
A code maps each fixed-width string of bits to a meaning

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Marine Mammal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100101</td>
<td>Humpback Whale</td>
</tr>
<tr>
<td>0100110</td>
<td>Leopard Seal</td>
</tr>
<tr>
<td>0100111</td>
<td>Sea Otter</td>
</tr>
<tr>
<td>0101000</td>
<td>West Indian Manatee</td>
</tr>
<tr>
<td>0101001</td>
<td>Bottlenose Dolphin</td>
</tr>
</tbody>
</table>

- This mapping however is rarely stored explicitly
  - Rather it is used when we interpret the bits.
How many bits to encode N possible things?

- 1 bit can encode 2 possibilities (0, 1)

Bits = 1
# encodings = 2
What is the minimum # of bits to encode?

One of the U.S.’s 50 states?

a) 3
b) 4
c) 5
d) 6
e) 7
What is the minimum # of bits to encode?

We are creating a rudimentary hashing function and want to encode the first letter of a person’s first name (a-z) and the number of letters in their name (1-9) as a set of bits.

- 6
- 7
- 8
- 9
- 10

Do we always want to use minimal encodings?
Use Hexadecimal (base-16) as a shorthand for binary numbers

- The hexadecimal system uses 16 digits:
  0 1 2 3 4 5 6 7 8 9 A B C D E F

- We can write our 32-bit number:
  1001 1010 1110 0110 1011 0001 1111 1101
  as
  0x9AE6B1FD (C/Java style)
  32’h9AE6B1FD (Verilog style)

Fun fact: Hex is frequently used to specify things like 32-bit IP addresses and 24-bit colors.
Hexadecimal to Binary

• What is $\text{B4}_{16}$ in binary?
  • A: 10110100
  • B: 1010100
  • C: 1011100
  • D: 11000100
2’s complement representation and binary arithmetic
Unsigned binary representation represents numbers as a weighted positional notation.

\[
a_{N-1} \cdot \cdots \cdot a_2 a_1 a_0
\]

\[
a_{N-1} \cdot \cdots \cdot a_2 a_1 a_0 \cdot\cdot\cdot
\]

\[
\begin{align*}
a_{N-1} \cdot 2^{N-1} + \cdots + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0
\end{align*}
\]
The number wheel (4-bit unsigned #'s)
How do we represent negative numbers?

- What would be ideal is:
  - If we could use the **same algorithm to add signed numbers as we use for unsigned numbers**
  - Then our computers wouldn’t need 2 kinds of adders, just 1.

- This is achieved using the **2’s complement representation**.
2’s complement negates the weight of the most significant bit

\[ a_{N-1} \quad \cdots \quad a_2 \quad a_1 \quad a_0 \]
\[ -\left(2^{N-1}\right) \quad \cdots \quad \frac{a_2}{2^2} \quad \frac{a_1}{2^1} \quad \frac{a_0}{2^0} \]

\[ a_{N-1} \quad \cdots \quad a_2 \quad a_1 \quad a_0 \]
\[ -\left(2^{N-1}\right) \quad \cdots \quad 2^2 \quad \frac{2^1}{2} \quad \frac{2^0}{2} \]
The number wheel (4-bit 2’s complement)
Negating Numbers in 2’s Complement

- To negate a number:
  - Complement each bit and then add 1.

- Example:

  \[
  \begin{array}{c}
  0100 = +4_{10} \quad \text{(a positive number in 4-bit two’s complement)} \\
  1011 \quad \text{(invert all the bits)} \\
  0100 = -4_{10} \quad \text{(and add one)} \\
  0011 \quad \text{(invert all the bits)} \\
  0100 = +4_{10} \quad \text{(and add one)} \\
  \end{array}
  \]

Sometimes, people talk about “taking the two’s complement” of a number. This is a confusing phrase, but it usually means to negate some number that’s already in two’s complement format.
Converting 2’s Complement to Decimal

- **Algorithm 1:**
  - if negative, negate; then do unsigned binary to decimal

- **Algorithm 2:**
  - Same as with n-bit unsigned binary
  - Except, the MSB is worth \(-2^{n-1}\)

\[
-b_{n-1}2^{n-1} + \sum_{k=0}^{n-2} b_k 2^k
\]

- **Example:**

  \[1100_{\text{2’s complement}} = -4_{10}\] (a negative number in 4-bit two’s complement)
2’s Complement Negation

If 01011 is the 5-bit 2’s complement representation for 11, what is the 5-bit 2’s complement representation for -11?

- A: 11011
- B: 10011
- C: 10101
- D: 01011
- E: 10100
2’s Complement Representation

If 01001 is the 5-bit unsigned binary representation for 9, what is the 5-bit 2’s complement representation for 9?

- A: 10110
- B: 10111
- C: 10101
- D: 01001
- E: 01010
You have always implicitly used zero extension

- To subtract 231 and 3, for instance, you can imagine:

  231
  - 003
  - 003

- This works for positive 2’s complement numbers, but not negative ones.
To align negative 2’s complement numbers, you need to extend the sign bit

- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5).
  - But 1100 (-4) should become 1111 1100 (-4).
Add binary numbers just like how you do with decimal numbers

- But remember that it’s binary! For example, $1 + 1 = 10$ and you have to carry!

The initial carry in is implicitly 0

```
  0 1 1 1 0  Carry in
     0 1 0 1 1  Augend
   + 0 1 1 1 0  Addend
  -------
    1 1 0 0 1  Sum
```

most significant bit, or MSb

least significant bit, or LSb
Binary addition with 2’s Complement

- You can add two’s complement numbers just as if they are unsigned numbers.
  - Recall, this was the whole reason for this representation

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 0 & + & (-4) \\
\hline
0 & 0 & 1 & 1 & 1 & 7 \\
\end{array}
\]
We can implement subtraction by negating the 2nd input and then adding

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 1 & 13 \\
- & 0 & 1 & 0 & 1 & 0 & -10
\end{array}
\rightarrow
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 1 & 13 \\
+ & 1 & 0 & 1 & 1 & 0 & +(-10)
\end{array}
\Rightarrow
\begin{array}{cccccc}
0 & 0 & 0 & 1 & 1 & 3
\end{array}
\]
Why does this work?

- For n-bit numbers, the negation of B in two’s complement is $2^n - B$ (this is alternative way of negating a 2’s-complement number).
  
  $$A - B = A + (-B) = A + (2^n - B) = (A - B) + 2^n$$

- If $A \geq B$, then $(A - B)$ is a positive number, and $2^n$ represents a carry out of 1. Discarding this carry out is equivalent to subtracting $2^n$, which leaves us with the desired result $(A - B)$.

- If $A < B$, then $(A - B)$ is a negative number $-(B - A)$ and we have $2^n - (B - A)$. This corresponds to the desired result, $(A - B)$, in two’s complement form.
2’s Complement Subtraction

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
- & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 \\
\end{array}
\]

- A: 0111
- B: 0011
- C: 1000
- D: 0101
- E: 1001
2’s Complement Subtraction

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
- & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 1 & 1 \\
\end{array}
\]

- A: 1011
- B: 1010
- C: 0001
- D: 0101
- E: 1111
“Carry-out” is a procedure, “Overflow” is an interpretation

<table>
<thead>
<tr>
<th>Carry-out</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Occurs at every bit-position</td>
<td>▪ Can only be seen after completing an entire mathematical operation</td>
</tr>
<tr>
<td>▪ The process of moving larger numbers to higher bit positions</td>
<td>▪ When the interpretation of a set of bits does not match the expected value after a mathematical operation</td>
</tr>
<tr>
<td>▪ Focuses on bit-wise operations</td>
<td>▪ Focuses on representational range (i.e., 4 bits represent 0-15)</td>
</tr>
</tbody>
</table>
The number wheel (4-bit unsigned #'s)
The number wheel (4-bit 2’s complement)
Overflow clicker

4-bit unsigned integers

\[
\begin{array}{c}
1110 \\
+ 0011 \\
\hline
0001
\end{array}
\]

14 + 3 = 17

4-bit 2’s comp integers

\[
\begin{array}{c}
1110 \\
- 2 \\
+ 0011 \\
\hline
-1
\end{array}
\]

1110 + 3 = 0001

a) Neither overflows
b) Only unsigned addition overflows
c) Only 2’s comp addition overflows
d) Both overflow
Overflow clicker

4-bit unsigned integers

\[
\begin{array}{c}
\begin{array}{c}
0 0 1 0 \\
+ 0 1 1 0 \\
\hline
1 0 0 0
\end{array}
\end{array}
\]

4-bit 2’s comp integers

\[
\begin{array}{c}
\begin{array}{c}
0 0 1 0 \\
+ 0 1 1 0 \\
\hline
1 0 0 0
\end{array}
\end{array}
\]

a) Neither overflows
b) Only unsigned addition overflows
c) Only 2’s comp addition overflows
d) Both overflow
Overflow clicker

4-bit unsigned integers 4-bit 2’s comp integers

a) Neither overflows
b) Only unsigned addition overflows
c) Only 2’s comp addition overflows
d) Both overflow
Overflow

In which circumstance can overflow not occur?

- A: subtracting a positive number from a negative number
- B: subtracting a negative number from zero
- C: adding two negative numbers
- D: subtracting a negative number from a positive number
- E: subtracting a negative number from a negative number
How can we know if overflow has occurred?

- The easiest way to detect signed overflow is to look at all of the sign bits.

  \[
  \begin{array}{cccc}
  0100 & (+4) & 1100 & (-4) \\
  + & 0101 & (+5) & + & 1011 & (-5) \\
  \hline
  1001 & (-7) & 0111 & (+7) \\
  \end{array}
  \]

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.

- Overflow cannot occur if you add a positive number to a negative number. Do you see why?
Bitshifting
Bit-wise left shift moves bits toward more significant bit positions

unsigned char f = b << 5;

(always left shift in zeros)

(left shift)
Bit-wise right shift considers a variable's signedness.

**unsigned char g = f >> 2;**

*(right shift logical)*

If unsigned, right shift in zeros:

**unsigned char i = h >> 2;**

*(right shift arithmetic)*

If signed, sign extend MSB

Note: `x >> 1` not the same as `x/2` for negative numbers; compare `(-3)>>1` with `(-3)/2`
Bit-shifting has lower precedence than arithmetic but higher than bitwise operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher</td>
<td>* / %</td>
<td>Multiplication, division, and modulus</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise shifting</td>
</tr>
<tr>
<td></td>
<td>&amp; ^</td>
<td></td>
</tr>
<tr>
<td>Lower</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
</tbody>
</table>
Bit-wise Logical & Shifting

- We have 2 unsigned 8-bit words:
  \[ x = a_7a_6a_5a_4a_3a_2a_1a_0 \]
  \[ y = b_7b_6b_5b_4b_3b_2b_1b_0 \]

- And we want the 8-bit word:
  \[ z = b_3b_2b_1b_0a_3a_2a_1a_0 \]

- A: \( z = (x \& 0xf) \mid (y \& 0x0f << 4) \)
- B: \( z = (x \& 0xf) \mid (y \& 0xf0) \)
- C: \( z = (x >> 4) \mid (y << 4) \)
- D: \( z = (x \& 0x0f) \mid (y << 4) \)
- E: \( z = (x \& 0xf) \mid (y >> 4) \)
Bit-shifting is useful for extracting bits

- We have the unsigned 8-bit word: \( b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \)
- And we want the 8-bit word: \( 0 0 0 0 0 b_5 b_4 b_3 \)
  - i.e., we want to extract bits 3-5.

- We can do this with bit-wise logical & shifting operations
  - \( y = (x \gg 3) \& \text{0x7}; \)

\[
\begin{align*}
x & \quad b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\
x \gg 3 \\
(x \gg 3) \& \text{0x7}
\end{align*}
\]
Useful for merging two bit patterns

- We have 2 unsigned 8-bit words: $a_7a_6a_5a_4a_3a_2a_1a_0$
  $b_7b_6b_5b_4b_3b_2b_1b_0$
- And we want the 8-bit word: $a_7b_6a_5b_4a_3b_2a_1b_0$
Bit-wise Logical & Shifting

- We have 2 unsigned 8-bit words:
  \[ x = a_7a_6a_5a_4a_3a_2a_1a_0 \]
  \[ y = b_7b_6b_5b_4b_3b_2b_1b_0 \]

- And we want the 8-bit word:
  \[ z = a_3a_2a_1a_0b_3b_2b_1b_0 \]

- A:
  \[ z = (x >> 4) \lor (y << 4) \]

- B:
  \[ z = (x \& 0x0f << 4) \lor (y \& 0xf) \]

- C:
  \[ z = (x >> 4) \lor (y \& 0xf) \]

- D:
  \[ z = (x \& 0xf0) \lor (y \& 0xf) \]

- E:
  \[ z = (x << 4) \lor (y \& 0x0f) \]