Abstraction

Correctness

Performance

Security
Correctness

• Example 1: Is \( x^2 \geq 0 \) ?
  
  • Floats: Yes!
  • Ints: ???
  • DEMO
Correctness

• Example 1: Is “$x^2 \geq 0$”?  
  - Floats: Yes!  
  - Ints: ???  
    - $40000 \times 40000 \rightarrow 16000000000$  
    - $50000 \times 50000 \rightarrow ??$  

• Example 2: Is “$(x + y) + z = x + (y + z)$”?  
  - Ints: Yes!  
  - Floats: ???  
  - DEMO

Source: xkcd.com/571
Correctness

• **Example 1: Is \(x^2 \geq 0\)?**
  - Floats: Yes!
  - Ints: ???
    - \(40000 \times 40000 \rightarrow 1600000000\)
    - \(50000 \times 50000 \rightarrow ??\)
  
  ![Comic Strip](xkcd.com/571)

• **Example 2: Is \((x + y) + z = x + (y + z)\)?**
  - Ints: Yes!
  - Floats:
    - \((2^{30} + -2^{30}) + 3.14 \rightarrow 3.14\)
    - \(2^{30} + (-2^{30} + 3.14) \rightarrow ??\)
Performance

How do these function compare asymptotically?

void copyij(int src[2048][2048],
            int dst[2048][2048]){
    int i,j;
    for (i = 0; i < 2048; i++){
        for (j = 0; j < 2048; j++){
            dst[i][j] = src[i][j];
        }
    }
}

void copyji(int src[2048][2048],
            int dst[2048][2048]){
    int i,j;
    for (j = 0; j < 2048; j++){
        for (i = 0; i < 2048; i++){
            dst[i][j] = src[i][j];
        }
    }
}

4.3ms 81.8ms

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array
void admin_stuff(int authenticated){
    if(authenticated){
        // do admin stuff
    }
}

int dontTryThisAtHome(char * user_input, int size) {
    char data[size];
    int ret = memcpy(*user_input, data);
    return ret;
}
Let’s start at the beginning… Bits
Bits

- a bit is a binary digit that can have two possible values
- can be physically represented with a two-state device
Storing bits

• Static random-access memory (SRAM): stores each bit of data in a flip-flop, a circuit with two stable states
• Dynamic Memory (DRAM): stores each bit of data in a capacitor, which stores energy in an electric field (or not)
• Magnetic Disk: regions of the platter are magnetized with either N-S polarity or S-N polarity
• Optical Disk: stores bits as tiny indentations (pits) or not (lands) that reflect light differently
• Flash Disk: electrons are stored in one of two gates separated by oxide layers
Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic—encode “True” as 1 and “False” as 0

<table>
<thead>
<tr>
<th>And</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Or</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exclusive-Or (Xor)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exercise 1: Boolean Operations

- Evaluate each of the following expressions
  1. $1 \mid (\neg 1)$
  2. $\neg (1 \mid 1)$
  3. $(\neg 1) \& 1$
  4. $\neg (1 \wedge 1)$
Exercise 1: Boolean Operations

• Evaluate each of the following expressions

1. \(1 \mid (~1)\) = \(1 \mid 0 = 1\)
2. \(\neg (1 \mid 1)\) = \(\neg 1 = 0\)
3. \((~1) \& 1\) = \(0 \& 1 = 0\)
4. \(\neg (1 \uparrow 1)\) = \(\neg 0 = 1\)
Bytes and Memory

- **Memory** is an array of bits
Bytes and Memory

- **Memory** is an array of bits

- A **byte** is a unit of eight bits

- An index into the array is an **address**, **location**, or **pointer**
  - Often expressed in hexadecimal

- We speak of the **value** in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address
General Boolean algebras

• Bitwise operations on bytes

\[
\begin{array}{c}
01101001 \\
\& \ 01010101 \\
01000001
\end{array} \quad
\begin{array}{c}
01101001 \\
| \ 01010101 \\
01111101
\end{array} \quad
\begin{array}{c}
01101001 \\
^ \ 01010101 \\
00111100
\end{array} \quad
\begin{array}{c}
\sim \ 01010101 \\
10101010
\end{array}
\]

• How does this map to set operations?
Exercise 2 : Bitwise Operations

• Assume:
  
  \[ a = 01101100 \]
  
  \[ b = 10101010 \]

• What are the results of evaluating the following Boolean operations?

  • \( \neg a \)
  • \( \neg b \)
  • \( a \& b \)
  • \( a \mid b \)
  • \( a \^\ b \)
Exercise 2 : Bitwise Operations

• Assume:
  
  \[
  a = 01101100 \\
  b = 10101010 \\
  \]

• What are the results of evaluating the following Boolean operations?

  \[
  \begin{align*}
    \sim a &= \sim 01101100 = 10010011 \\
    \sim b &= \sim 10101010 = 01010101 \\
    a \& b &= 01101100 \& 10101010 = 00101000 \\
    a \mid b &= 01101100 \mid 10101010 = 11101110 \\
    a \wedge b &= 01101100 \wedge 10101010 = 11000110 \\
  \end{align*}
  \]
Bitwise vs Logical Operations in C

- **Bitwise Operators** &, |, ~, ^
  - View arguments as bit vectors
  - Operations applied bit-wise in parallel

- **Logical Operators** &&, ||, !
  - View 0 as “False”
  - View anything nonzero as “True”
  - Always return 0 or 1
  - With short circuiting
Exercise 3: Bitwise vs Logical Operations

- \sim 01000001
- \sim 00000000
- \sim \sim 01000001

- \! 01000001
- \! 00000000
- \! \! 01000001

- 01101001 \& 01010101
- 01101001 \mid 01010101

- 01101001 \&\& 01010101
- 01101001 \mid\mid 01010101
Exercise 3: Bitwise vs Logical Operations

- \( \sim 01000001 \) \( \rightarrow 10111110 \)
- \( \sim 00000000 \) \( \rightarrow 11111111 \)
- \( \sim \sim 01000001 \) \( \rightarrow 01000001 \)
- \( ! 01000001 \) \( \rightarrow 00000000 \)
- \( ! 00000000 \) \( \rightarrow 00000001 \)
- \( !! 01000001 \) \( \rightarrow 00000001 \)
- \( 01101001 \& 01010101 \) \( \rightarrow 01000001 \)
- \( 01101001 \mid 01010101 \) \( \rightarrow 01111101 \)
- \( 01101001 \& & 01010101 \) \( \rightarrow 00000001 \)
- \( 01101001 \mid \mid 01010101 \) \( \rightarrow 00000001 \)
Bit Shifting

- **Left Shift:** \( x \ll y \)
  - Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

- **Right Shift:** \( x \gg y \)
  - Shift bit-vector \( x \) right \( y \) positions
  - Throw away extra bits on right
  - **Logical shift:** Fill with 0’s on left
  - **Arithmetic shift:** Replicate most significant bit on left

**Undefined Behavior** if you shift amount \(< 0\) or \(\geq\) word size

**Choice between logical and arithmetic depends on the type of data**
Example: Bit Shifting

- $01101001 \ll 4 = 10010000$
- $01101001 \gg 2 = 00011010$
- $01101001 \gg_a 4 = 00000110$
Exercise 4: Bit Shifting

- $10101010 \ll 4$  $\quad 10100000$
- $10101010 \gg_l 4$  $\quad 00001010$
- $10101010 \gg_a 4$  $\quad 11111010$
Bits and Bytes Require Interpretation

00000000 00110101 00110000 00110001
might be interpreted as

- The integer $3,485,745_{10}$
- A floating-point number close to $4.884569 \times 10^{-39}$
- The string “105”
- A portion of an image or video
- An address pointing to another place in memory
- Or… some user-defined type
Information is Bits + Context