Representations

Prof. David August

COS 217
Goals of Today’s Lecture

• Representations
  ○ Why binary?
  ○ Converting base 10 to base 2
  ○ Octal and hexadecimal

• Integers
  ○ Unsigned integers
  ○ Integer addition, subtraction
  ○ Signed integers

• C bit operators
  ○ And, or, not, and xor
  ○ Shift-left and shift-right
  ○ Function for counting the number of 1 bits
  ○ Function for XOR encryption of a message
Radiohead - OK Computer CD

3 Miles of Music
Pits and Lands

Transition represents a bit state (1/on/red/female/heads)
No change represents other state (0/off/white/male/tails)
Interpretation

As Music:

\[01110101_2 = \text{117/256 position of speaker}\]

As Number:

\[01110101_2 = 1 + 4 + 16 + 32 + 64 = 117_{10} = 75_{16}\]

(Get comfortable with base 2, 8, 10, and 16.)

As Text:

\[01110101_2 = \text{117th character in the ASCII codes = “u”}\]
# Interpretation – ASCII

<table>
<thead>
<tr>
<th>ASCII value</th>
<th>Character</th>
<th>Control character</th>
<th>ASCII value</th>
<th>Character</th>
<th>ASCII value</th>
<th>Character</th>
<th>ASCII value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>(null)</td>
<td>NUL</td>
<td>032</td>
<td>(space)</td>
<td>064</td>
<td>@</td>
<td>096</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>☻</td>
<td>SOH</td>
<td>033</td>
<td>!</td>
<td>065</td>
<td>A</td>
<td>097</td>
<td>a</td>
</tr>
<tr>
<td>002</td>
<td>☠</td>
<td>STX</td>
<td>034</td>
<td>’</td>
<td>066</td>
<td>B</td>
<td>098</td>
<td>b</td>
</tr>
<tr>
<td>003</td>
<td>♦</td>
<td>ETX</td>
<td>035</td>
<td>#</td>
<td>067</td>
<td>C</td>
<td>099</td>
<td>c</td>
</tr>
<tr>
<td>004</td>
<td>♦</td>
<td>EOT</td>
<td>036</td>
<td>$</td>
<td>068</td>
<td>D</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>005</td>
<td>♦</td>
<td>ENQ</td>
<td>037</td>
<td>%</td>
<td>069</td>
<td>E</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>006</td>
<td>♦</td>
<td>ACK</td>
<td>038</td>
<td>&amp;</td>
<td>070</td>
<td>F</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>007</td>
<td>(beep)</td>
<td>BEL</td>
<td>039</td>
<td>'</td>
<td>071</td>
<td>G</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>008</td>
<td>☤</td>
<td>BS</td>
<td>040</td>
<td>(</td>
<td>072</td>
<td>H</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>009</td>
<td>(tab)</td>
<td>HT</td>
<td>041</td>
<td>)</td>
<td>073</td>
<td>I</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>010</td>
<td>(line feed)</td>
<td>LF</td>
<td>042</td>
<td>+</td>
<td>074</td>
<td>J</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>011</td>
<td>(home)</td>
<td>VT</td>
<td>043</td>
<td>‘</td>
<td>075</td>
<td>K</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>012</td>
<td>(form feed)</td>
<td>FF</td>
<td>044</td>
<td>.</td>
<td>076</td>
<td>L</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>013</td>
<td>(carriage return)</td>
<td>CR</td>
<td>045</td>
<td>,</td>
<td>077</td>
<td>M</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>014</td>
<td></td>
<td>SO</td>
<td>046</td>
<td>;</td>
<td>078</td>
<td>N</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>015</td>
<td></td>
<td>SI</td>
<td>047</td>
<td>/</td>
<td>079</td>
<td>O</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>016</td>
<td></td>
<td>DLE</td>
<td>048</td>
<td>0</td>
<td>080</td>
<td>P</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>017</td>
<td>←</td>
<td>DC1</td>
<td>049</td>
<td>1</td>
<td>081</td>
<td>Q</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>018</td>
<td>↑</td>
<td>DC2</td>
<td>050</td>
<td>2</td>
<td>082</td>
<td>R</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>019</td>
<td>↓</td>
<td>DC3</td>
<td>051</td>
<td>3</td>
<td>083</td>
<td>S</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>020</td>
<td>=</td>
<td>DC4</td>
<td>052</td>
<td>4</td>
<td>084</td>
<td>T</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>021</td>
<td>§</td>
<td>NAK</td>
<td>053</td>
<td>5</td>
<td>085</td>
<td>U</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>022</td>
<td></td>
<td>SYN</td>
<td>054</td>
<td>6</td>
<td>086</td>
<td>V</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>023</td>
<td></td>
<td>ETB</td>
<td>055</td>
<td>7</td>
<td>087</td>
<td>W</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>024</td>
<td></td>
<td>CAN</td>
<td>056</td>
<td>8</td>
<td>088</td>
<td>X</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>025</td>
<td></td>
<td>EM</td>
<td>057</td>
<td>9</td>
<td>089</td>
<td>Y</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>026</td>
<td>←</td>
<td>SUB</td>
<td>058</td>
<td>;</td>
<td>090</td>
<td>Z</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>027</td>
<td>←</td>
<td>ESC</td>
<td>059</td>
<td>:</td>
<td>091</td>
<td>[</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>028</td>
<td>(cursor right)</td>
<td>FS</td>
<td>060</td>
<td>&lt;</td>
<td>092</td>
<td>\</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>029</td>
<td>(cursor left)</td>
<td>GS</td>
<td>061</td>
<td>=</td>
<td>093</td>
<td>]</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>(cursor up)</td>
<td>RS</td>
<td>062</td>
<td>&gt;</td>
<td>094</td>
<td>^</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>031</td>
<td>(cursor down)</td>
<td>US</td>
<td>063</td>
<td>?</td>
<td>095</td>
<td>–</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>
Computer Science Building West Wall
Interpretation:
Code and Data (Hello World!)

- Programs consist of Code and Data
- Code and Data are Encoded in Bits

```
IA-64 Binary (objdump)

00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000 .ELF..............
...
00000260: 5002 0000 0000 0000 006c 6962 632e 736f P.......libc.so
00000270: 2e36 2e31 0070 7269 6e74 6600 5f5f 6c69 .6.1.printf._li
00000280: 6263 5f73 7461 7274 5f6d 6169 6e00 474c bc_start_main.GL
00000290: 4942 435f 322e 3200 0000 0020 0020 0000 IBC_2.2........
...
00000860: 4865 6c6c 6f62 576f 7261 6d65 0d00 0000 Hello world!....
...
40000000000000: 00 10 15 08 80 05 [MII] alloc r34=ar.pfs,5,4,0
40000000000005: 30 02 30 00 42 20 mov r35=r12
40000000000006: 04 00 c4 00 mov r33=0
40000000000007: 0a 20 81 03 00 24 [MMI] add1 r36=96,r1;;
40000000000008: 40 02 90 30 20 00 ld8 r36=[r36]
40000000000009: 04 08 00 84 mov r32=r1
4000000000000a: 1d 00 00 00 01 00 [MFB] nop.m 0x0
4000000000000b: 00 00 00 02 00 00 nop.f 0x0
4000000000000c: b8 fd ff 58 [MII] br.call.sptk_many b0=400000000000000460;;
4000000000000d: 00 08 00 40 00 21 mov r1=r32
4000000000000e: 80 00 00 00 42 00 mov r8=r0
4000000000000f: 20 02 aa 00 mov.i ar.pfs=r34
40000000000010: 00 00 00 00 01 00 [MII] nop.m 0x0
40000000000011: 00 08 05 80 03 80 mov b0=r33
40000000000012: 01 18 01 84 mov r12=r35
40000000000013: 1d 00 00 00 01 00 [MFB] nop.m 0x0
40000000000014: 00 00 00 02 00 80 nop.f 0x0
40000000000015: 08 00 84 00 br.ret.sptk_many b0;;
```
Interpretation:
Numbers

• Base 10
  ◦ Each digit represents a power of 10
  ◦ \(4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0\)

• Base 2
  ◦ Each bit represents a power of 2
  ◦ \(10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22\)

  Divide repeatedly by 2 and keep remainders

\[\begin{align*}
12/2 &= 6 \quad \text{R} = 0 \\
6/2 &= 3 \quad \text{R} = 0 \\
3/2 &= 1 \quad \text{R} = 1 \\
1/2 &= 0 \quad \text{R} = 1 \\
\end{align*}\]

Result = 1100
Writing Bits is Tedious for People

- **Octal (base 8)**
  - Digits 0, 1, ..., 7
  - In C: 00, 01, ..., 07

- **Hexadecimal (base 16)**
  - Digits 0, 1, ..., 9, A, B, C, D, E, F
  - In C: 0x0, 0x1, ..., 0xf

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

Thus the 16-bit binary number 1011 0010 1010 1001 converted to hex is B2A9.
Interpretation: Colors

• Three primary colors
  ◦ Red
  ◦ Green
  ◦ Blue

• Strength
  ◦ 8-bit number for each color (e.g., two hex digits)
  ◦ So, 24 bits to specify a color

• In HTML, on the course Web page
  ◦ Red: <font color="#FF0000"><i>Symbol Table Assignment Due</i></font>
  ◦ Blue: <font color="#0000FF"><i>Fall Recess</i></font>

• Same thing in digital cameras
  ◦ Each pixel is a mixture of red, green, and blue
Binary Representation of Integers

• Fixed number of bits in memory
  - char: 8 bits
  - short: usually 16 bits
  - int: 16 or 32 bits
  - long: 32 bits
  - long long: 64 bits

• Unsigned integers
  - Always positive or 0
  - All arithmetic is modulo 2^n
  - unsigned char
  - unsigned short
  - unsigned int
  - unsigned long
  - unsigned long long

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1{n}</td>
<td>2^{n-1}</td>
</tr>
</tbody>
</table>
Size and Overflow in Unsigned Integers

<table>
<thead>
<tr>
<th>Bits</th>
<th>Integer Range</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 - 255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0 - 65,535</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>0 - 4,294,967,295</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>0 - 18,446,744,073,709,551,615</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Number of bits determines unsigned integer range

Overflow:
• 8-bit integer $\rightarrow 11111111_2$ \( (255_{10}) \)
• Add 1
• What happens?
Adding Two Integers: Base 10

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column

```
+  1  9  8
  2  6  4
  ---
 Sum 4  6  2
 Carry 0  1  1
```

```
+  0  1  1
  0  0  1
  ---
 Sum 1  0  0
 Carry 0  1  1
```
### Binary Sums and Carries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum</th>
<th></th>
<th></th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XOR**

\[
\begin{array}{llll}
0100 & 0101 & \Rightarrow & 69 \\
+    & 0110 & 0111 & \Rightarrow 103 \\
\hline
1010 & 1100 & \Rightarrow 172
\end{array}
\]
Overflow in Unsigned Addition

Operands: $w$ bits

| $u$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
+----+---------+---------+---------+---------|
| $v$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

True Sum: $w + 1$ bits

| $u + v$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

Discard Carry: $w$ bits

$\text{UAdd}_w(u, v)$

$U\text{Add}_w(u,v) = \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}$

Modulo Arithmetic: $\text{UAdd}_w(u, v) = u + v \mod 2^w$
Detecting Unsigned Overflow

- **Task:**
  - Given \( s = \text{UAdd}_w(u, v) \)
  - Determine if \( s = u + v \)

- **Claim:**
  - Overflow iff \( s < u \)
  - \( \text{ovf} = (s < u) \)
  - By symmetry iff \( s < v \)

- **Proof:**
  - \( 0 \leq v < 2^w \)
  - No overflow \( \Rightarrow s = u + v \geq u + 0 = u \)
  - Overflow \( \Rightarrow s = u + v - 2^w < u + 0 = u \)
Modulo Arithmetic

- Consider only numbers in a range
  - E.g., five-digit car odometer: 0, 1, …, 99999
  - E.g., eight-bit numbers 0, 1, …, 255

- Roll-over when you run out of space
  - E.g., car odometer goes from 99999 to 0, 1, …
  - E.g., eight-bit number goes from 255 to 0, 1, …

- Adding $2^n$ doesn’t change the answer
  - For eight-bit number, $n=8$ and $2^n=256$
  - E.g., $(37 + 256) \mod 256$ is simply 37

- This can help us do subtraction…
  - Suppose you want to compute $a – b$
  - Note that this equals $a + (256 -1 - b) + 1$
Modulo Arithmetic

Modulo Addition Forms an Abelian Group

- **Closed under addition**
  - $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$

- **Commutative**
  - $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$

- **Associative**
  - $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$

- **0 is additive identity**
  - $\text{UAdd}_w(u, 0) = u$

- **Every element has additive inverse**
  - Let $\text{UComp}_w(u) = 2^w - u$
  - $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$
What about Negative Numbers?

- We have been looking at unsigned numbers.
- What about negative or signed numbers?

- Need new interpretation of bits.
- Some patterns interpreted as negative numbers.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Patterns</th>
<th>Binary</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>256</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>4,294,967,296</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>18,446,744,073,709,551,616</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1{n}</td>
<td>2^n</td>
</tr>
</tbody>
</table>
### Key Standard Pattern Assignments

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Sign Magnitude</th>
<th>One’s Complement</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>+0</td>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>-0</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-3</td>
<td>-0</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Which one is best?**
  - Balance
  - Zeros
  - Ease of operations
**Most Common: Two’s Complement**

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+2</td>
</tr>
<tr>
<td>011</td>
<td>+3</td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
</tbody>
</table>

- “Invert and Add 1” to negate
- Sign Bit
- Zeros, Range
- What about arithmetic?
Unsigned and Two’s Complement

- **Unsigned Values**
  \[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]
  - \( U\text{Min} = 0 \)
  - \( U\text{Max} = 2^w - 1 \)

- **Two’s Complement**
  \[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]
  - \( T\text{Min} = -2^{w-1} \)
  - \( T\text{Max} = 2^{w-1} - 1 \)

**Values for \( W = 16 \)**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Two’s Compliment Range

$T_{Max}$

$T_{Min}$

$U_{Max}$

$U_{Max} - 1$

$T_{Max} + 1$

$T_{Max}$

Unsigned Range

Representation Relationship
### Sizes and C Data Types

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>MIPS, x86</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>long int</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

**char, short, int, long int**

- Refer to number of bits of integer
- Most machines: signed two’s complement

**unsigned <type>**

- Same number of bits as signed counterparts
- Unsigned integer
Sign Extension

cchar minusFour = -4;
short moreBits;
moreBits = (short) minusFour;

Given $w$ bit signed integer, return equivalent $w+k$ bit signed integer

Sign Extend:
Sign Extension
Proof of Correctness Outline

• Prove Correctness by Induction on k
• Induction Step: extending by single bit maintains value
Two’s Complement Addition

• TAdd and UAdd have identical Bit-Level Behavior!

Operands: $w$ bits

True Sum: $w + 1$ bits

Discard Carry: $w$ bits

$$TAdd_w(u, v)$$
Characterizing TAdd

- True sum requires \( w+1 \) bits
- Drop MSB

\[
TAdd_w(u,v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < TMin_w \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v 
\end{cases}
\]
Detecting Two’s Complement Overflow

- **Task:**
  - Given \( s = T\text{Add}_w(u, v) \)
  - Determine if \( s = \text{Add}_w(u, v) \)

- **Claim:**
  - Overflow iff either:
    - \( u, v < 0, s \geq 0 \) (NegOver)
    - \( u, v \geq 0, s < 0 \) (PosOver)
  - \( \text{ovf} = (u<0 == v<0) && (u<0 != s<0); \)

- **Proof:**
  - Obviously, if \( u \geq 0 \) and \( v < 0 \), then \( \text{TMin}_w \leq u + v \leq \text{TMax}_w \)
  - Symmetrically if \( u < 0 \) and \( v \geq 0 \)
  - Other cases from analysis of \( T\text{Add} \)
Negation vs. Inversion

**Inversion:**
- A bit-wise operation
- Flip all 0’s to 1’s and vice versa: 0011 => 1100
- What does this do to the two’s complement value?

**Negation:**
- Two’s complement: invert all bits and add 1
- Example:
  \[ 3_{10} = 0011 \]
  \[ \text{invert}(0011) + 1 \rightarrow 1100 + 1 \rightarrow 1101 \]
  \[ 1101 = -3_{10} \]
Two’s Complement Negation

• Mostly like Integer Negation
  ○ TComp(u) = \(-u\)

• Tmin is Special Case
  ○ TComp(TMin) = TMin
  ○ Note Also: TComp(0) = 0

• Negation in C (\(x = -x;\)) is Actually TComp
Comparing Two’s Complements

- Given signed numbers $u$, $v$
- Determine whether or not $u > v$
- Return true for shaded region:

```
< 0   > 0
< 0   > 0
```

- **Bad Approach:**
  - Test $(u - v) > 0$
  - Problem: Thrown off by Overflow
Representation: A Collection of Bits

- Treat unsigned int as a collection 32 independent bits
- Good for tracking 32 individual binary conditions
  - True/False
  - Yes/No
  - Black/White

- Can also treat unsigned int as:
  - 16 2-bit values
  - 8 4-bit values
  - 4 8-bit values
  - 8 1-bit value, 4 2-bit values, 2 4-bit values, and 1 8-bit value
Bitwise Operators: AND and OR

• Bitwise AND (&)

<table>
<thead>
<tr>
<th>&amp;</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>

- Mod on the cheap!
  - E.g., h = 53 & 15;

53 \[0 0 1 1 0 1 0 1\]
& 15 \[0 0 0 0 1 1 1 1\]

\[\text{\underline{5}}\] \[0 0 0 0 0 1 0 1\]

• Bitwise OR (|)

<table>
<thead>
<tr>
<th></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>
Bitwise Operators: Not and XOR

• One’s complement (\(\sim\))
  ◦ Turns 0 to 1, and 1 to 0
  ◦ E.g., set last three bits to 0
    \(-x = x \& \sim7;\)

• XOR (\(^\wedge\))
  ◦ 0 if both bits are the same
  ◦ 1 if the two bits are different

\[
\begin{array}{c|cc}
^\wedge & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Bitwise Operators: Shift Left/Right

- **Shift left (<<): Multiply by powers of 2**
  - Shift some # of bits to the left, filling the blanks with 0
    - Example:
      - 53 (00110100) << 2
      - Result: 53 << 2 = 11010000

- **Shift right (>>): Divide by powers of 2**
  - Shift some # of bits to the right
    - For unsigned integer, fill in blanks with 0
    - What about signed integers? Varies across machines…
      - Can vary from one machine to another!
    - Example:
      - 53 (00110100) >> 2
      - Result: 53 >> 2 = 00001101
Count Number of 1s in an Integer

- **Function** bitcount(unsigned x)
  - Input: unsigned integer
  - Output: number of bits set to 1 in the binary representation of x

- **Main idea**
  - Isolate the last bit and see if it is equal to 1
  - Shift to the right by one bit, and repeat

```c
int bitcount(unsigned int x) {
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 1)
            b++;
    return b;
}
```
XOR Encryption

• Program to encrypt text with a key
  ◦ Input: original text in stdin
  ◦ Output: encrypted text in stdout

• Use the same program to decrypt text with a key
  ◦ Input: encrypted text in stdin
  ◦ Output: original text in stdout

• Basic idea
  ◦ Start with a key, some 8-bit number (e.g., 0110 0111)
  ◦ Do an operation that can be inverted
    – E.g., XOR each character with the 8-bit number

  \[
  \begin{align*}
    \text{0100 0101} & \quad \text{0010 0010} \\
    \wedge \text{0110 0111} & \quad \wedge \text{0110 0111} \\
    \hline
    \text{0010 0010} & \quad \text{0100 0101}
  \end{align*}
  \]
XOR Encryption, Continued

• But, we have a problem
  ◦ Some characters are control characters
  ◦ These characters don’t print

• So, let’s play it safe
  ◦ If the encrypted character would be a control character
  ◦ … just print the original, unencrypted character
  ◦ Note: the same thing will happen when decrypting, so we’re okay

• C function iscntrl()
  ◦ Returns true if the character is a control character
XOR Encryption, C Code

```c
#define KEY '\&'

int main(void) {
    int orig_char, new_char;

    while ((orig_char = getchar()) != EOF) {
        new_char = orig_char ^ KEY;
        if (iscntrl(new_char))
            putchar(orig_char);
        else
            putchar(new_char);
    }
    return 0;
}
```
Stupid Programmer Tricks

• Where do I use bitwise & most?
  ◦ Bit vectors

• What’s a bit vector?
  ◦ Lots of booleans packed into an int/long
  ◦ Often used to indicate some condition(s)
  ◦ Less storage space than lots of fields
  ◦ More explicit storage than compiled-defined bit fields

• Your compiler can do this?
  typedef struct Blah {
    int b_onoff:1;
    int b_temperature:7;
    char b_someChar;
  }

Example From Real Code

- #define DONTCACHE_REQNOSTORE 0x000001
- #define DONTCACHE_AUTHORIZED 0x000002
- #define DONTCACHE_MISSINGVARIANTHDR 0x000004
- #define DONTCACHE_USERORPASS 0x000008
- #define DONTCACHE_BYPASSFILTER 0x000010
- #define DONTCACHE_NONCACHEMETHOD 0x000020
- #define DONTCACHE_CTLPRIVATE 0x000040
- #define DONTCACHE_CTLNOSTORE 0x000080
- #define DONTCACHE_ISQUERY 0x000100
- #define DONTCACHE_EARLYEXPIRE 0x000200
- #define DONTCACHE_NOLASTMOD 0x000400
- #define DONTCACHE_NONEGCACHING 0x000800
- #define DONTCACHE_INSTANTEXPIRE 0x001000
- #define DONTCACHE_FILETOOBIG 0x002000
- #define DONTCACHE_FILEGREWTOOBIG 0x004000
- #define DONTCACHE_ICPPROXYONLY 0x008000
- #define DONTCACHE_LARGEFILEBLAST 0x010000
- #define DONTCACHE_PERSISTLOGLOADING 0x020000
- #define DONTCACHE_NEWERCOPYEXISTES 0x040000
- #define DONTCACHE_BADVARYFIELDS 0x080000
- #define DONTCACHE_SETCOOKIE 0x100000
- #define DONTCACHE_HTTPSTATUSCODE 0x200000
- #define DONTCACHE_OBJECTINCOMPLETE 0x400000
Conclusions

- Computer represents everything in binary
  - Integers, floating-point numbers, characters, addresses, …
  - Pixels, sounds, colors, etc.

- Binary arithmetic through logic operations
  - Sum (XOR) and Carry (AND)
  - Two’s complement for subtraction

- Binary operations in C
  - AND, OR, NOT, and XOR
  - Shift left and shift right
  - Useful for efficient and concise code, though sometimes cryptic