15-213
“The Class That Gives CMU Its Zip!”

Bits, Bytes, and Integers
August 27, 2009

Topics

- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C
- Representations of Integers
  - Basic properties and operations
  - Implications for C
Binary Representations

Base 2 Number Representation

- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]…_2$
- Represent $1.5213 \times 10^4$ as $1.11011011011012 \times 2^{13}$

Electronic Implementation

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires
Encoding Byte Values

Byte = 8 bits

- Binary $00000000_2$ to $11111111_2$
- Decimal: $0_{10}$ to $255_{10}$
- Hexadecimal $00_{16}$ to $FF_{16}$
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B$_{16}$ in C as 0xFA1D37B
    » Or 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
Back to bits: 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>Operation</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>( A &amp; B = 1 ) when both ( A=1 ) and ( B=1 )</td>
</tr>
</tbody>
</table>
|           | \[
|          | \begin{array}{c|c|c}
|          | \hline
|          | 0 & 0 & 0 \\
|          | 1 & 1 & 1 \\
|          | \hline
|          | \end{array}
|           | |
| Or        | \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \) |
|           | \[
|          | \begin{array}{c|c|c}
|          | \hline
|          | 0 & 0 & 1 \\
|          | 1 & 1 & 1 \\
|          | \hline
|          | \end{array}
|           | |
| Not       | \( \sim A = 1 \) when \( A=0 \) |
|           | \[
|          | \begin{array}{c|c}
|          | \hline
|          | 0 & 1 \\
|          | 1 & 0 \\
|          | \hline
|          | \end{array}
|           | |
| Exclusive-Or (Xor) | \( A^\wedge B = 1 \) when either \( A=1 \) or \( B=1 \), but not both |
|           | \[
|          | \begin{array}{c|c|c}
|          | \hline
|          | 0 & 1 & 1 \\
|          | 0 & 1 & 0 \\
|          | \hline
|          | \end{array}
|           | |
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \quad | 01010101 & \quad ^ 01010101 & \quad ^ 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

All of the Properties of Boolean Algebra Apply
Bit-Level Operations in C

Operations &, |, ~, ^ Available in C
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)
- ~0x41 --> 0xBE
  - ~01000001_2 --> 10111110_2
- ~0x00 --> 0xFF
  - ~00000000_2 --> 11111111_2
- 0x69 & 0x55 --> 0x41
  - 01101001_2 & 01010101_2 --> 01000001_2
- 0x69 | 0x55 --> 0x7D
  - 01101001_2 | 01010101_2 --> 01111101_2
Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1

Examples (char data type)

- !0x41  -->  0x00
- !0x00  -->  0x01
- !!0x41 -->  0x01
- 0x69 && 0x55  -->  0x01
- 0x69 || 0x55  -->  0x01
- p && *p (avoids null pointer access)

Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming
Shift Operations

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

Right Shift: \( x >> y \)
- 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 right

<table>
<thead>
<tr>
<th>Argument x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( 01100010 )</td>
<td></td>
</tr>
<tr>
<td>( 00010000 )</td>
<td></td>
</tr>
<tr>
<td>( 00011000 )</td>
<td></td>
</tr>
<tr>
<td>( 00011000 )</td>
<td></td>
</tr>
<tr>
<td>( 10100010 )</td>
<td></td>
</tr>
<tr>
<td>( 00010000 )</td>
<td></td>
</tr>
<tr>
<td>( 00101000 )</td>
<td></td>
</tr>
<tr>
<td>( 11101000 )</td>
<td></td>
</tr>
</tbody>
</table>
Memory organization

Programs refer to data by address
- address space viewed as a large array of bytes
- an address is like an index into that array

Any given computer has a “Word Size”
- nominal size of integer-valued data
  - and, usually, of addresses
- 32 bits is still most common
  - though 64 bits is emerging
# Data Representations

## Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned [int]</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>–</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

» Or any other pointer
Encoding Integers

Unsigned

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

Two’s Complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

\begin{align*}
\text{short int } x &= 15213; \\
\text{short int } y &= -15213;
\end{align*}

\begin{table}
\begin{tabular}{|c|c|c|c|}
\hline
Decimal & Hex & Binary \\
\hline
x & 15213 & 00111011 01101101 \\
-15213 & C4 93 & 11000100 10010011 \\
\hline
\end{tabular}
\end{table}

Sign Bit

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Encoding Example (Cont.)

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum: 15213, -15213
numeric ranges

unsigned values

- $U_{\text{Min}} = 0$
  000...0
- $U_{\text{Max}} = 2^w - 1$
  111...1

two's complement values

- $T_{\text{Min}} = -2^{w-1}$
  100...0
- $T_{\text{Max}} = 2^{w-1} - 1$
  011...1

other values

- Minus 1
  111...1

values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</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>
Signed vs. unsigned ints in C

Constants
- By default, considered to be signed integers
- Unsigned if have “U” as suffix
  
  0U, 4294967259U

Casting
- Can explicitly cast between signed & unsigned

  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;

- Implicit casting also occurs via assignments (and function calls)
  
  tx = ux;
  uy = ty;
Casting Surprises

Expression Evaluation

- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

<table>
<thead>
<tr>
<th>Constant$_1$</th>
<th>Constant$_2$</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Visual of casting

2’s Comp. $\rightarrow$ Unsigned

- Ordering Inversion
- Negative $\rightarrow$ Big Positive

$T_{Max}$ $\rightarrow$ $UMax$ $UMax - 1$

$T_{Max} + 1$ $T_{Max}$

$T_{Min}$ $0$

2’s Comp. Range $\rightarrow$ Unsigned Range
Sign Extension

Task:
- Given \( w \)-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer with same value

Rule:
- Make \( k \) copies of sign bit:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\[ k \text{ copies of MSB} \]
Sign Extension Example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Unsigned Addition

Operands: $w$ bits

| u | · | · | · | · | · | · |
|+| v | · | · | · | · | · |

True Sum: $w+1$ bits

| $u+v$ | · | · | · | · | · | · |

Discard Carry: $w$ bits

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

| · | · | · | · | · |

Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

$s = \text{UAdd}_w(u, v) = u + v \mod 2^w$

$\text{UAdd}_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}$
Visualizing what should happen

Integer Addition

- As in, actual math, not computer math
- 4-bit integers $u, v$
- Compute true sum $Add_4(u, v)$
- Values increase linearly with $u$ and $v$
- Forms planar surface

$Add_4(u, v)$
Visualizing actually happens

Wraps Around

- If true sum $\geq 2^w$
- At most once

True Sum

\[ 2^{w+1} \]
\[ 2^w \]
\[ 0 \]

Modular Sum

Overflow

$UAdd_4(u, v)$
Two’s Complement Addition

Operands: $w$ bits

\[
\begin{array}{c}
\text{u} \quad \text{u} \\
\text{+} \quad \text{v} \\
\hline
\text{u+v} \quad \text{v}
\end{array}
\]

True Sum: $w+1$ bits

\[
\text{u+v} \quad \text{v}
\]

Discard Carry: $w$ bits

\[
\text{TAdd}_w(u, v) \quad \text{v}
\]

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  ```
  int s, t, u, v;
  s = (int) ((unsigned) u + (unsigned) v);
  t = u + v
  ```
- Will give $s == t$
Characterizing TAdd

Functionality

- True sum requires \( w+1 \) bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

\[
TAdd(u, v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < Tmin_w \quad \text{(NegOverflow)} \\
  u + v & Tmin_w \leq u + v \leq Tmax_w \\
  u + v - 2^{w-1} & Tmax_w < u + v \quad \text{(PosOverflow)} 
\end{cases}
\]
Visualizing what actually happens

Values
- 4-bit two’s comp.
- Range from -8 to +7

Wraps Around
- If $\text{sum} \geq 2^{w-1}$
  - Becomes negative
  - At most once
- If $\text{sum} < -2^{w-1}$
  - Becomes positive
  - At most once
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

Standard Multiplication Function

- Ignores high order $w$ bits

Implements Modular Arithmetic

$$UMult_w(u, v) = u \cdot v \mod 2^w$$
Signed Multiplication in C

Operands: $w$ bits

$u \cdot v$

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

Standard Multiplication Function

- Ignores high order $w$ bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same
Power-of-2 Multiply with Shift

Operation

- \( u \ll k \) gives \( u \times 2^k \)
- Both signed and unsigned

\[
\begin{array}{c}
\text{Operands: } w \text{ bits} \\
\hline
\text{True Product: } w+k \text{ bits} \\
\hline
\text{Discard } k \text{ bits: } w \text{ bits}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{UMult}_w(u, 2^k) \\
\text{TMult}_w(u, 2^k)
\end{array}
\end{array}
\]

Examples

- \( u \ll 3 \) \( \Rightarrow \) \( u \times 8 \)
- \( u \ll 5 - u \ll 3 \) \( \Rightarrow \) \( u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- \( u \gg k \) gives \( \left\lfloor \frac{u}{2^k} \right\rfloor \)
- Uses logical shift

Operands:

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

Division: \( u \gg k \) gives \( \left\lfloor \frac{u}{2^k} \right\rfloor \)

Result: \( \left\lfloor \frac{u}{2^k} \right\rfloor \)

Division Computed Hex Binary
Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- \( x \gg k \) gives \( \left\lfloor \frac{x}{2^k} \right\rfloor \)
- Uses arithmetic shift
- Rounds wrong direction when \( u < 0 \)

\[
\begin{array}{cccccccccc}
\text{Operands:} & \multicolumn{5}{c}{x} & \text{Binary Point} \\
/ & 2^k & \multicolumn{5}{c}{0 \cdots 010 \cdots 00} \\
\text{Division:} & x / 2^k & \multicolumn{5}{c}{\cdots \cdots \cdots \cdots} \\
\text{Result:} & \text{RoundDown}(x / 2^k) & \multicolumn{5}{c}{\cdots \cdots} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0)
- Compute as \( \left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor \)
  - In C: \( (x + (1<<k)-1) >> k \)
  - Biases dividend toward 0

Case 1: No rounding

\[
\begin{array}{c}
\text{Dividend:} \\
\begin{array}{c}
u \\+2^k-1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
k \\
\end{array}
\]

\[
\begin{array}{c}
\text{Divisor:} \\
\begin{array}{c}
\begin{array}{c}
\frac{u}{2^k}
\end{array}
\end{array}
\end{array}
\]

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend:

\[ x \]
\[ +2^k - 1 \]

Divisor:

\[ \frac{x}{2^k} \]

\[ \left\lfloor \frac{x}{2^k} \right\rfloor \]

**Biasing adds 1 to final result**
Representing Strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code \(0x30\)
    - Digit \(i\) has code \(0x30 + i\)
- String should be null-terminated
  - Final character = 0
What’s next

Recitation on Monday

- autolab, fish machines, datalab
- get started on lab #1 before then…

Floating point (Tues): representations and arithmetic
Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by word size
  - e.g., 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
Byte ordering in multi-byte “words”

Big Endian (e.g., SPARC, Power PC)
- Least significant byte has highest address

Little Endian (e.g., x86)
- Least significant byte has lowest address

Example
- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01 23 45 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>
Examining Data Representations

Code to Print Byte Representation of Data

Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:

%p: Print pointer
%x: Print Hexadecimal
**show_bytes Execution Example**

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

**Result (Linux):**

```c
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Representing Integers

```c
int A = 15213;
int B = -15213;
long int C = 15213;
```

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

Two’s complement representation (Covered later)
Reading Byte-Reversed Listings

Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers

- Value: 0x12ab
- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00
Representing Pointers

```c
int B = -15213;
int *P = &B;
```

Different compilers & machines assign different locations to objects
Connection when $A \& \neg B \mid \neg A \& B$

$= A \hat{\land} B$
Integer C Puzzles

- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

  - $x < 0 \Rightarrow ((x*2) < 0)$
  - $ux >= 0$
  - $x & 7 == 7 \Rightarrow (x<<30) < 0$
  - $ux > -1$
  - $x > y \Rightarrow -x < -y$
  - $x * x >= 0$
  - $x > 0 && y > 0 \Rightarrow x + y > 0$
  - $x >= 0 \Rightarrow -x <= 0$
  - $x <= 0 \Rightarrow -x >= 0$
  - $(x|-x)>>31 == -1$
  - $ux >> 3 == ux/8$
  - $x >> 3 == x/8$
  - $x & (x-1) != 0$

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

**Observations**
- \(|TMin| = Tmax + 1\)
  - Asymmetric range
- \(UMax = 2 \times Tmax + 1\)

**C Programming**
- `#include <limits.h>`
  - K&R App. B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific
## Unsigned & Signed Numeric Values

### Equivalence
- **Same encodings for nonnegative values**

### Uniqueness
- **Every bit pattern represents unique integer value**
- **Each representable integer has unique bit encoding**

⇒ **Can Invert Mappings**
- **U2B(x) = B2U⁻¹(x)**
  - Bit pattern for unsigned integer
- **T2B(x) = B2T⁻¹(x)**
  - Bit pattern for two’s comp integer

<table>
<thead>
<tr>
<th>x</th>
<th>B2U(x)</th>
<th>B2T(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>–8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>–7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>–6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>–5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>–4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>–3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>–2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>–1</td>
</tr>
</tbody>
</table>
Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

Unsigned

\( x \rightarrow T2B \rightarrow B2U \rightarrow u_x \)

\( w-1 \)

\( u_x \)

\( x \)

\( x \geq 0 \)

\( x + 2^w \quad x < 0 \)

Large negative weight

→

Large positive weight

\( u_x = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0
\end{cases} \)
When should I use unsigned?

*Don’t Use Just Because Number Nonzero*

- Easy to make mistakes
  
  ```c
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
      a[i] += a[i+1];
  ```

- Can be very subtle
  
  ```c
  #define DELTA sizeof(int)
  int i;
  for (i = CNT; i-DELTA >= 0; i-= DELTA)
      ...
  ```

*Do Use When Performing Modular Arithmetic*

- Multiprecision arithmetic

*Do Use When Need Extra Bit’s Worth of Range*

- Working right up to limit of word size
Negating with Complement & Increment

Claim: Following Holds for 2’s Complement

\[ \neg x + 1 = -x \]

Complement

- Observation: \[ \neg x + x = 1111\ldots 1_{2} = -1 \]

\[
\begin{array}{ccc}
\times & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\neg x & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
-1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

Increment

- \[ \neg x + x + (\neg x + 1) = -1 + (\neg x + 1) \]
- \[ \neg x + 1 = -x \]

Warning: Be cautious treating int’s as integers

OK here
### Comp. & Incr. Examples

**x = 15213**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>~x</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>~x+1</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td></td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td></td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>
Mathematical Properties

Modular 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 \( \text{UComp}_w(u) = 2^w - u \)
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- \( T\text{Add}_w(u, v) = \text{U}2\text{T}(U\text{Add}_w(\text{T}2\text{U}(u), \text{T}2\text{U}(v))) \)
  - Since both have identical bit patterns

Two’s Complement Under TAdd Forms a Group

- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse

\[
T\text{Comp}_w(u) = \begin{cases} 
-u & u \neq \text{TMin}_w \\
\text{TMin}_w & u = \text{TMin}_w 
\end{cases}
\]
Multiplication

Computing Exact Product of $w$-bit numbers $x, y$

- Either signed or unsigned

Ranges

- **Unsigned**: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits

- **Two’s complement min**: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits

- **Two’s complement max**: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2w$ bits, but only for $(TMin_w)^2$

Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages
Compiled Multiplication Code

C Function

```c
int mul12(int x) {
    return x*12;
}
```

Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
Compiled Unsigned Division Code

C Function

```c
unsigned udiv8(unsigned x) {
    return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned

For Java Users

- Logical shift written as >>>>
Compiled Signed Division Code

C Function

```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```asm
testl %eax, %eax
js    L4
L3:
sarl $3, %eax
ret
L4:
addl $7, %eax
jmp   L3
```

Explanation

```asm
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- Arith. shift written as >>

For Java Users

- int idiv8(int x) {
  return x/8;
}
- if x < 0
  x += 7;
  return x >> 3;
Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms
Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
- Multiplication Commutative
  \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
- Multiplication is Associative
  \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
- 1 is multiplicative identity
  \[ \text{UMult}_w(u, 1) = u \]
- Multiplication distributes over addition
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to $w$ bits
- Two’s complement multiplication and addition
  - Truncating to $w$ bits

Both Form Rings

- Isomorphic to ring of integers mod $2^w$

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,
  \[
  \begin{align*}
  &u > 0 \implies u + v > v \\
  &u > 0, v > 0 \implies u \cdot v > 0
  \end{align*}
  \]
- These properties are not obeyed by two’s comp. arithmetic

\[
T_{Max} + 1 = T_{Min}
\]
\[
15213 \times 30426 = -10030 \text{ (16-bit words)}
\]
Integer C Puzzles Revisited

• \( x < 0 \) \( \Rightarrow \) \((x*2) < 0)\)
• \( ux >= 0 \)
• \( x & 7 == 7 \) \( \Rightarrow \) \((x<<30) < 0)\)
• \( ux > -1 \)
• \( x > y \) \( \Rightarrow \) \(-x < -y\)
• \( x * x >= 0 \)
• \( x > 0 && y > 0 \) \( \Rightarrow \) \(x + y > 0\)
• \( x >= 0 \) \( \Rightarrow \) \(-x <= 0)\)
• \( x <= 0 \) \( \Rightarrow \) \(-x >= 0)\)
• \((x|-x)>>31 == -1\)
• \( ux >> 3 == ux/8\)
• \( x >> 3 == x/8\)
• \( x & (x-1) != 0\)

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```