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

Bits, Bytes, and Integers
August 20, 2008

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 $15_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]…_2$
- Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

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

Byte = 8 bits

- Binary: 00000000₂ to 11111111₂
- Decimal: 0₁₀ to 255₁₀
  - First digit must not be 0 in C
- Hexadecimal: 00₁₆ to FF₁₆
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B₁₆ 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>
Byte-Oriented Memory Organization

Programs Refer to Virtual Addresses
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

Compiler + Run-Time System Control Allocation
- Where different program objects should be stored
- All allocation within single virtual address space
Machine Words

Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses

- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
    » Users can access 3GB
  - Becoming too small for memory-intensive applications

- High-end systems use 64 bits (8 bytes) words
  - Potential address space $\approx 1.8 \times 10^{19}$ bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes

- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
## 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</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*
Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Big Endian: Sun, PPC Mac
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address
Byte Ordering Example

Big Endian
- Least significant byte has highest address

Little Endian
- Least significant byte has lowest address

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

```
Big Endian
0x100 0x101 0x102 0x103
   01  23  45  67

Little Endian
0x100 0x101 0x102 0x103
  67  45  23  01
```
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
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;
0x11fffffcb8 0x6d
0x11fffffcb9 0x3b
0x11fffffcba 0x00
0x11fffffcbbb 0x00
```
Representing Integers

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

IA32, x86-64 A  Sun A

IA32, x86-64 B  Sun B

IA32 C  x86-64 C  Sun C

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B6D

Two’s complement representation (Covered later)
Representing Pointers

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

Different compilers & machines assign different locations to objects.

<table>
<thead>
<tr>
<th>Sun P</th>
<th>IA32 P</th>
<th>x86-64 P</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td>FD</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>7F</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
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

Compatibility

- Byte ordering not an issue

```
char S[6] = "15213";
```
Boolean Algebra

Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

- A&B = 1 when both A=1 and B=1

<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>

Or

- A|B = 1 when either A=1 or B=1

<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>

Not

- ~A = 1 when A=0

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

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

<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>0</td>
</tr>
</tbody>
</table>
Application of Boolean Algebra

Applied to Digital Systems by Claude Shannon

- 1937 MIT Master’s Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

Connection when

\[ A \& \sim B \quad \sim A \& B \]

\[ = A^{\sim} B \]
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

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

All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

Representation

- **Width** $w$ bit vector represents subsets of $\{0, \ldots, w{-}1\}$
- $a_j = 1$ if $j \in A$

<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>{0, 3, 5, 6}</td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

Operations

- **&** Intersection  
<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000001</td>
<td>{0, 6}</td>
</tr>
</tbody>
</table>

- **|** Union  
<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111101</td>
<td>{0, 2, 3, 4, 5, 6}</td>
</tr>
</tbody>
</table>

- **^** Symmetric difference  
<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
</tbody>
</table>

- **~** Complement  
<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
</tbody>
</table>
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
  - ~01000001₂ --> 10111110₂
- ~0x00 --> 0xFF
  - ~00000000₂ --> 11111111₂
- 0x69 & 0x55 --> 0x41
  - 01101001₂ & 01010101₂ --> 01000001₂
- 0x69 | 0x55 --> 0x7D
  - 01101001₂ | 01010101₂ --> 01111101₂
Contrast: Logic Operations in C

Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Examples (char data type)

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

Strange Behavior
- Shift amount > word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>
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

1. \( x < 0 \) \( \Rightarrow \) \((x \times 2) < 0\)
2. \( ux >= 0 \)
3. \( x \& 7 == 7 \) \( \Rightarrow \) \((x<<30) < 0\)
4. \( ux > -1 \)
5. \( x > y \) \( \Rightarrow \) \(-x < -y\)
6. \( x \times x >= 0 \)
7. \( x > 0 \) \&\& \( y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
8. \( x >= 0 \) \( \Rightarrow \) \(-x <= 0\)
9. \( x <= 0 \) \( \Rightarrow \) \(-x >= 0\)
10. \((x|-x)>>31 == -1\)
11. \( ux >> 3 == ux/8 \)
12. \( x >> 3 == x/8 \)
13. \( x & (x-1) != 0 \)

Initialization

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

Unsigned

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

Two’s Complement

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

- \text{short int } x = 15213;
- \text{short int } y = -15213;

\[ \text{C short 2 bytes long} \]

<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>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

Sign Bit

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

\[ x = \quad 15213: \ 00111011 \ 01101101 \]
\[ y = \quad -15213: \ 11000100 \ 10010011 \]

<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>

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Numeric Ranges

Unsigned Values
- $U_{\text{Min}} = 0$
- $U_{\text{Max}} = 2^w - 1$

Two's Complement Values
- $T_{\text{Min}} = -2^{w-1}$
- $T_{\text{Max}} = 2^{w-1} - 1$

Other Values
- Minus 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></td>
<td>-1 FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
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
- $U_{Max} = 2 \times Tmax + 1$

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

<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

\[ x \rightarrow \text{T2B} \rightarrow \text{B2U} \rightarrow u_x \]

Maintain Same Bit Pattern

\[ \begin{align*}
  u_x & = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0 
  \end{cases} 
\end{align*} \]

Large negative weight → Large positive weight

\[ \begin{align*}
  w-1 & \quad 0 \\
  u_x & = + + + + + + + + + + + \\
  x & = + + + + + + + + + + + 
\end{align*} \]
Signed vs. Unsigned in C

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

Casting
- Explicit casting between signed & unsigned same as U2T and T2U

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

- Implicit casting also occurs via assignments and procedure 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>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive

2’s Comp. Range

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:
  \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \]

\( k \) copies of MSB

\( X \)

\( X' \)
Sign Extension Example

```
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>00110111 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 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
Why Should I UseUnsigned?

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
\[ \sim x + 1 \equiv -x \]

Complement

- Observation: \[ \sim x + x \equiv 1111...11_2 \equiv -1 \]

\[
\begin{array}{c}
  x \\
  10011101
\end{array}
\]
\[
\begin{array}{c}
  + \sim x \\
  01100010
\end{array}
\]
\[
\begin{array}{c}
  \hline
  \sim x + 1 \\
  11111111
\end{array}
\]

Increment

- \[ \sim x + x + (\sim x + 1) \equiv -1 + (\sim x + 1) \]
- \[ \sim x + 1 \equiv -x \]

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>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

Operands: \( w \) bits

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

=True Sum: \( w+1 \) bits

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

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 Integer Addition

Integer Addition

- 4-bit integers $u$, $v$
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with $u$ and $v$
- Forms planar surface
Visualizing Unsigned Addition

Wraps Around

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

True Sum

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

Modular Sum

Overflow

Overflow

\[ \text{Overflow} \]

\[ U\text{Add}_4(u, v) \]
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
  \[ \text{Let } \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Two’s Complement Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

\[ \begin{array}{c}
  \text{Discard Carry: } w \text{ bits}
\end{array} \]

\[ \begin{array}{c}
  \text{TAdd}_w(u, v)
\end{array} \]

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  ```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

\[
\text{TAdd}(u, v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v \quad \text{(PosOver)}
\end{cases}
\]
Visualizing 2’s Comp. Addition

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

Wraps Around
- If sum $\geq 2^{w-1}$
  - Becomes negative
  - At most once
- If sum $< -2^{w-1}$
  - Becomes positive
  - At most once
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- \( TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \)
  - 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

\[
TComp_w(u) = \begin{cases} 
-u & u \neq Tmin_w \\
Tmin_w & u = Tmin_w 
\end{cases}
\]
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:** $x \times y \geq (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - 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 $\text{TMin}_w^2$

Maintaining Exact Results

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

Operands: \( w \) bits

Standard Multiplication Function

- Ignores high order \( w \) bits

Implements Modular Arithmetic

\[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
Signed Multiplication in C

Operands: $w$ bits

True Product: $2^w$ 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

Operands: $w$ bits

\[
\begin{array}{c}
\text{u} \times 2^k \\
\end{array}
\]

True Product: $w+k$ bits

\[
\begin{array}{c}
\text{u} \times 2^k \\
\end{array}
\]

Discard $k$ bits: $w$ bits

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

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

Examples

- $u \ll 3 == u \times 8$
- $u \ll 5 - u \ll 3 == u \times 24$
- Most machines shift and add faster than multiply

- Compiler generates this code automatically
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

- C compiler automatically generates shift/add code when multiplying by constant
Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- $u >> k$ gives $\left\lfloor \frac{u}{2^k} \right\rfloor$
- Uses logical shift

```
               k
          u       Binary Point
```

Operands:

```
/  2^k  0 1 0 1 1 0 0
```

Division:

```
u / 2^k
```

Result:

```
\left\lfloor \frac{u}{2^k} \right\rfloor
```

<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 &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
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 >>>
**Signed Power-of-2 Divide with Shift**

### Quotient of Signed by Power of 2

- \( x >> k \) gives \( \lfloor x / 2^k \rfloor \)
- Uses arithmetic shift
- Rounds wrong direction when \( u < 0 \)

### Operands:

\[
\begin{array}{c}
x \\
/ 2^k \\
x / 2^k
\end{array}
\]

### Division:

\[
\begin{array}{c}
x \\
/ 2^k \\
x / 2^k
\end{array}
\]

### Result:

RoundDown(\( x / 2^k \))

### Table:

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-15213)</td>
<td>(-15213)</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>(-7606.5)</td>
<td>(-7607)</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>(-950.8125)</td>
<td>(-951)</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>(-59.4257813)</td>
<td>(-60)</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 \[ \lceil \frac{x}{2^k} \rceil \] (Round Toward 0)
- Compute as \[ \lfloor \frac{x+2^k-1}{2^k} \rfloor \]
  - In C: \( x + (1<<k)-1 \gg k \)
  - Biases dividend toward 0

Case 1: No rounding

\[
\begin{array}{c}
\text{Dividend:} \\
\hline
u & 1\ldots\ldots\ldots\ldots\ldots0\ldots\ldots\ldots00 \\
+2^k-1 & 0\ldots\ldots001\ldots\ldots11 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{Divisor:} \\
\hline
\frac{u}{2^k} & 0\ldots\ldots01\ldots\ldots00 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{Quotient:} \\
\hline
\lceil u/2^k \rceil & 1\ldots\ldots111\ldots\ldots11 \\
\hline
\end{array}
\]

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

Case 2: Rounding

Dividend:

\[
\begin{array}{c}
x \\
+2^k - 1
\end{array}
\]

Divisor:

\[
\begin{array}{c}
x \div 2^k \\
\left\lfloor x / 2^k \right\rfloor
\end{array}
\]

Biasing adds 1 to final result

Incremented by 1
Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

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

For Java Users

```java
int idiv8(int x)
{
    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.,
  \[
  u > 0 \implies u + v > v \\
  u > 0, v > 0 \implies u \cdot v > 0
  \]
- These properties are not obeyed by two’s comp. arithmetic
  \[
  T_{\text{Max}} + 1 = T_{\text{Min}}
  \]

\[
15213 \times 30426 = -10030 \quad (16 \text{ bit words})
\]
Integer C Puzzles Revisited

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

Initialization

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