Bits, Bytes, and Integers
August 26, 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 $15_{10}$ 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

![Diagram showing voltage levels and binary representation]
Encoding Byte Values

Byte = 8 bits

- **Binary**: \(00000000_2\) to \(11111111_2\)
- **Decimal**: \(0_{10}\) to \(255_{10}\)
  - First digit must not be 0 in C
- **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>
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 ≈ 1.8 X 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)
## 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

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
</tr>
</tbody>
</table>
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):

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

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

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
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
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>( A )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Or

- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)

<table>
<thead>
<tr>
<th>( A )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not

- \( \sim A = 1 \) when \( A=0 \)

<table>
<thead>
<tr>
<th>( A )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sim )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

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

<table>
<thead>
<tr>
<th>( A )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</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

\[ A \& \neg B \lor \neg A \& B \]

= \( A^{\land} B \)
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & ~ 01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

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>Binary</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110101</td>
<td>{0, 3, 5, 6}</td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

Operations

- & Intersection
  - Binary: 01000001
  - Set: \{0, 6\}
- | Union
  - Binary: 01111101
  - Set: \{0, 2, 3, 4, 5, 6\}
- ^ Symmetric difference
  - Binary: 00111100
  - Set: \{2, 3, 4, 5\}
- ~ Complement
  - Binary: 10101010
  - Set: \{1, 3, 5, 7\}
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)

- \( \sim 0x41 \rightarrow 0xBE \)
  - \( \sim 01000001_2 \rightarrow 10111110_2 \)
- \( \sim 0x00 \rightarrow 0xFF \)
  - \( \sim 00000000_2 \rightarrow 11111111_2 \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 01101001_2 \& 01010101_2 \rightarrow 01000001_2 \)
- \( 0x69 \mid 0x55 \rightarrow 0x7D \)
  - \( 01101001_2 \mid 01010101_2 \rightarrow 01111101_2 \)
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: $x \ll y$
- 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 right

Strange Behavior
- Shift amount > word size

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. $\gg 2$</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. $\gg 2$</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. $\gg 2$</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. $\gg 2$</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

\[
\begin{align*}
\text{• } x < 0 & \implies ((x \times 2) < 0) \\
\text{• } u_x \geq 0 & \\
\text{• } x \& 7 == 7 & \implies (x<<30) < 0 \\
\text{• } u_x > -1 & \\
\text{• } x > y & \implies -x < -y \\
\text{• } x \times x \geq 0 & \\
\text{• } x > 0 \&\& y > 0 & \implies x + y > 0 \\
\text{• } x >= 0 & \implies -x <= 0 \\
\text{• } x <= 0 & \implies -x >= 0 \\
\text{• } (x|\neg x) >> 31 == -1 & \\
\text{• } u_x >> 3 == u_x/8 & \\
\text{• } x >> 3 == x/8 & \\
\text{• } x \& (x-1) != 0 &
\end{align*}
\]

Initialization

```c
int x = foo();
int y = bar();
unsigned u_x = x;
unsigned u_y = y;
```
Encoding Integers

Unsigned

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

Two’s Complement

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

C short 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 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.)

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

<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>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>0</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
  - 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>
### 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
- $|T\text{Min}| = T\text{Max} + 1$
- Asymmetric range
- $U\text{Max} = 2 \times T\text{Max} + 1$

### 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

<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>

### 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
Relation between Signed & Unsigned

Two’s Complement

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

Maintain Same Bit Pattern

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

Large negative weight \( \rightarrow \) Large positive weight
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
  ```c
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;
  ```
- Implicit casting also occurs via assignments and procedure calls
  ```c
  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>$\text{Constant}_1$</th>
<th>$\text{Constant}_2$</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>$(\text{unsigned})$ $-1$</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>$(\text{int})$ $2147483648U$</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:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \)
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>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
Why 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

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

Complement

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

\[
\begin{array}{c}
\sim x + x \\
\hline
\end{array}
\]

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

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

\[ 0 \]

<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

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\[
\begin{align*}
\text{UAdd}_w(u, v) &= \begin{cases} 
  u + v & \text{if } u + v < 2^w \\
  u + v - 2^w & \text{if } u + v \geq 2^w
\end{cases}
\end{align*}
\]

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

Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic
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

$\text{Add}_4(u, v)$
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

UAdd$_4(u, v)$

Graph showing the relationship between true sum and modular sum with overflow indication.
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 \( \text{UComp}_w(u) = 2^w - u \)
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Two’s Complement Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits  \( \text{TAdd}_w(u, v) \)

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  \[
  \begin{align*}
  \text{int } s, t, u, v; \\
  s &= \text{(int) ((unsigned) } u + \text{ (unsigned) } v); \\
  t &= u + v
  \end{align*}
  \]
- Will give \( s == t \)
Characterizing TAdd

### Functionality
- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

#### True Sum

<table>
<thead>
<tr>
<th>u</th>
<th>v</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>100</td>
<td>011...1</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>000...0</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
<td>100...0</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

#### TAdd Result

- **PosOver**: $u + v + 2^{w-1}$ for $u + v < Tmin_w$
- **NegOver**: $u + v - 2^{w-1}$ for $TMax_w < u + v$

$$TAdd_w(u,v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < Tmin_w \\
  u + v & Tmin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v
\end{cases}$$
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

$TAdd_4(u, v)$

NegOver
PosOver
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 \((TMin_w)^2\)

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

\[
\begin{array}{c}
v \\
\cdot \\
u
\end{array}
\]

True Product: \( 2^w \) bits

\[
\begin{array}{c}
u \cdot v
\end{array}
\]

Discard \( w \) bits: \( w \) bits

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

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

$$u \cdot v \quad TMult_w(u, v)$$

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

Operands: \( w \) bits

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

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

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

Examples

- \( u \ll 3 \) \( \equiv \) \( u \times 8 \)
- \( u \ll 5 - u \ll 3 \) \( \equiv \) \( u \times 24 \)
- 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

```
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
Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses logical shift

<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>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 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 >>>

~52~

15-213: Intro to Computer Systems
Fall 2009 ©
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 \( u < 0 \)

\[
\begin{array}{c}
\text{Operands: } x \\
\hline
/ 2^k \\
\hline
\text{Division: } x / 2^k \\
\hline
\text{Result: RoundDown}(x / 2^k)
\end{array}
\]

<table>
<thead>
<tr>
<th>y</th>
<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</td>
<td>10010011</td>
</tr>
<tr>
<td>(-7606.5)</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010</td>
<td>01001001</td>
</tr>
<tr>
<td>(-950.8125)</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100</td>
<td>01001001</td>
</tr>
<tr>
<td>(-59.4257813)</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111</td>
<td>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

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( u )</th>
<th>( +2^k-1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divisor:</td>
<td>( / 2^k )</td>
<td>( \left\lfloor u / 2^k \right\rfloor )</td>
</tr>
</tbody>
</table>

\( u \) has leading 1:

Binaries:

- Biases dividend toward 0

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

Case 2: Rounding

Dividend:
\[ x \newline +2^k - 1 \]

Divisor:
\[ \div 2^k \newline \left\lfloor \frac{x}{2^k} \right\rfloor \]

\[ \begin{array}{c}
 1 \cdots 0 \cdots 1 \cdots 1 \\
 0 \cdots 0 \cdots 0 \cdots 1 \\
 0 \cdots 0 \cdots 0 \cdots 1 \\
\end{array} \]

\[ \begin{array}{c}
 1 \cdots 1 \cdots 1 \cdots 1 \\
 0 \cdots 0 \cdots 0 \cdots 0 \\
 1 \cdots 1 \cdots 1 \cdots 1 \\
\end{array} \]

Biasing adds 1 to final result

Incremented by 1

Binary Point

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
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.,
  \[
  \begin{align*}
  u > 0 & \quad \Rightarrow \quad u + v > v \\
  u > 0, \ v > 0 & \quad \Rightarrow \quad u \cdot v > 0
  \end{align*}
  \]
- These properties are not obeyed by two’s comp. arithmetic
  \[
  Tmax + 1 =\quad Tmin
  \]
  \[
  15213 \times 30426 =\quad -10030 \quad (16\text{ bit words})
  \]
Integer C Puzzles Revisited

Initialization

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

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