Representing and Manipulating Integers
Part I

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Introduction

- The advent of the digital age
  - Analog vs. digital?

- Compact Disc (CD)
  - 44.1 KHz, 16-bit, 2-channel

- MP3
  - A digital audio encoding with lossy data compression
Representing Information

**Information = Bits + Context**

- Computers manipulate representations of things.
- Things are represented as binary digits.
- What can you represent with N bits?
  - $2^N$ things
  - Numbers, characters, pixels, positions, source code, executable files, machine instructions, ...
  - Depends on what operations you do on them.

<table>
<thead>
<tr>
<th>(char)</th>
<th>'s'</th>
<th>'k'</th>
<th>'k'</th>
<th>'u'</th>
<th>'s'</th>
<th>'e'</th>
<th>'m'</th>
<th>'i'</th>
</tr>
</thead>
<tbody>
<tr>
<td>(int)</td>
<td>01110011 01101011 01101011 01110101 01110011 01100101 01101101 01101001</td>
<td>1969974131</td>
<td>1768777075</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(double)</td>
<td>7.03168990329170818178... x 10^{199}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Binary Representations

- Why not base 10 representation?
  - Easy to store with bistable elements
  - Straightforward implementation of arithmetic functions
  - Reliably transmitted on noisy and inaccurate wires

- Electronic implementation
Encoding Byte Values

- **Byte = 8 bits**
  - **Binary:** $00000000_2$ to $11111111_2$
  - **Octal:** $000_8$ to $377_8$
    - An integer constant that begins with $0$ is an octal number in C
  - **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>
Boolean Algebra (1)

- Developed by George Boole in 1849
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

- And
  - $A \& B = 1$ when both $A=1$ and $B=1$
    
    | & | 0 | 1 |
    |---|---|--|
    | 0 | 0 | 0 |
    | 1 | 0 | 1 |

- Not
  - $\sim A = 1$ when $A=0$
    
    | ~ | 0 | 1 |
    |---|---|--|
    | 0 | 1 |
    | 1 | 0 |

- Or
  - $A | B = 1$ when either $A=1$ or $B=1$
    
    | | 0 | 1 |
    |---|---|--|
    | 0 | 0 | 1 |
    | 1 | 1 | 1 |

- Exclusive-Or (Xor)
  - $A ^ B = 1$ when either $A=1$ or $B=1$, but not both
    
    | ^ | 0 | 1 |
    |---|---|--|
    | 0 | 0 | 1 |
    | 1 | 1 | 0 |
Boolean Algebra (2)

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

X
Y

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

Constant 0

X & Y ; AND

~ (X → Y)

X

~ (Y → X)

Y

X ^ Y ; XOR

X | Y ; OR

~ (X | Y) ; NOR

~ (X ^ Y) ; X-NOR

~ Y

Y → X

~ X

X → Y

~ (X & Y) ; NAND

Constant 1

Basic operations: AND(&), OR(|), NOT(~)

X ^ Y = (X & ~Y) | (~X & Y)

X→Y = ~X | Y

A complete set: NAND = ~ (X & Y)
Unsigned Integers

- Encoding unsigned integers

\[ B = [b_{w-1}, b_{w-2}, \ldots, b_0] \]

\[ D(B) = \sum_{i=0}^{w-1} b_i \cdot 2^i \]

\[ x = 0000 \ 0111 \ 1101 \ 0011_2 \]

\[ D(x) = 2^{10} + 2^9 + 2^8 + 2^7 \]
\[ + 2^6 + 2^4 + 2^1 + 2^0 \]
\[ = 1024 + 512 + 256 + 128 \]
\[ + 64 + 16 + 2 + 1 \]
\[ = 2003 \]

- What is the range for unsigned values with \( w \) bits?
Signed Integers (1)

- **Encoding positive numbers**
  - Same as unsigned numbers

- **Encoding negative numbers**
  - Sign-magnitude representation
  - Ones’ complement representation
  - Two’s complement representation
Signed Integers (2)

- Sign-magnitude representation

\[ S(B) = (-1)^{b_{w-1}} \cdot \left( \sum_{i=0}^{w-2} b_i \cdot 2^i \right) \]

- Two zeros
  - [000...00], [100...00]
- Used for floating-point numbers
Signed Integers (3)

- Ones’ complement representation

\[ O(B) = -b_{w-1}(2^{w-1} - 1) + \left( \sum_{i=0}^{w-2} b_i \cdot 2^i \right) \]

- Easy to find \(-n\)
- Two zeros
  - \([000...00]\), \([111...11]\)
- No longer used
Signed Integers (4)

- Two’s complement representation

\[ O(B) = -b_{w-1} \cdot 2^{w-1} + \left( \sum_{i=0}^{w-2} b_i \cdot 2^i \right) \]

- Unique zero
- Easy for hardware
  - leading 0 ≥ 0
  - leading 1 < 0
- Used by almost all modern machines
Signed Integers (5)

- Two’s complement representation (cont’d)
  - Following holds for two’s complement
    \[ \sim x + 1 = -x \]

  - Complement
    - Observation: \[ \sim x + x = 1111...12 = -1 \]

  - Increment
    \[ \sim x + x = -1 \]
    \[ \sim x + x + (-x + 1) = -1 + (-x + 1) \]
    \[ \sim x + 1 = -x \]
### Numeric Ranges (1)

#### Unsigned values
- **UMin** = 0 \([000...00]\)
- **UMax** = \(2^w - 1\) \([111...11]\)

#### Two’s complement values
- **TMin** = \(-2^{w-1}\) \([100...00]\)
- **TMax** = \(2^{w-1} - 1\) \([011...11]\)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>TMin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Numeric Ranges (2)

- Values for different word sizes

<table>
<thead>
<tr>
<th></th>
<th>w = 8</th>
<th>w = 16</th>
<th>w = 32</th>
<th>w = 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
  - $|\text{TMin}| = \text{TMax} + 1$ (Asymmetric range)
  - $\text{UMax} = 2 \times \text{TMax} + 1$

- In C programming
  - `#include <limits.h>`
  - `INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, UINT_MAX, ...`
  - Values platform-specific
Type Conversion (1)

- **Unsigned:** \( w \) bits \( \rightarrow \) \( w+k \) bits
  - Zero extension: just fill \( k \) bits with 0’s

```c
unsigned short x = 2003;
unsigned ix = (unsigned) x;
```
Type Conversion (2)

- **Signed:** $w$ bits $\rightarrow w+k$ bits
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value

- **Sign extension**
  - Make $k$ copies of sign bit
### Sign extension example

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

```c
short int x = 2003;
int ix = (int) x;
short int y = -2003;
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>2003</td>
<td>07 D3</td>
<td>00000111 11010011</td>
</tr>
<tr>
<td>ix</td>
<td>2003</td>
<td>00 00 07 D3</td>
<td>00000000 00000000 00000111 11010011</td>
</tr>
<tr>
<td>y</td>
<td>-2003</td>
<td>F8 2D</td>
<td>11111000 00101101</td>
</tr>
<tr>
<td>iy</td>
<td>-2003</td>
<td>FF FF F8 2D</td>
<td>11111111 11111111 11111000 00101101</td>
</tr>
</tbody>
</table>
Type Conversion (4)

- Unsigned & Signed: \( w+k \) bits \( \rightarrow \) \( w \) bits
  - Just truncate it to lower \( w \) bits
  - Equivalent to computing \( x \mod 2^w \)

| unsigned int | \( x = 0xcafebabe; \) |
| unsigned short | \( ix = (\text{unsigned short}) x; \) |
| int | \( y = 0x2003beef; \) |
| short | \( iy = (\text{short}) y; \) |

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>3405691582</td>
<td>CA FE BA BE 11001010 11111110 10111010 10111110</td>
</tr>
<tr>
<td>( ix )</td>
<td>47806</td>
<td>BA BE 10111010 10111110</td>
</tr>
<tr>
<td>( y )</td>
<td>537116399</td>
<td>20 03 BE EF 00100000 00000011 10111110 11101111</td>
</tr>
<tr>
<td>( iy )</td>
<td>-16657</td>
<td>BE EF 10111110 11101111</td>
</tr>
</tbody>
</table>
### Type Conversion (5)

#### Unsigned → Signed

- The same bit pattern is interpreted as a signed number

\[
U2T_w(x) = \begin{cases} 
  x, & x < 2^{w-1} \\
  x - 2^w, & x \geq 2^{w-1}
\end{cases}
\]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>203</td>
<td>07 D3 00000111 11010011</td>
</tr>
<tr>
<td>ix</td>
<td>203</td>
<td>07 D3 00000111 11010011</td>
</tr>
<tr>
<td>y</td>
<td>47806</td>
<td>BA BE 10111010 10111110</td>
</tr>
<tr>
<td>iy</td>
<td>-17730</td>
<td>BA BE 10111010 10111110</td>
</tr>
</tbody>
</table>

unsigned short x = 2003;
short ix = (short) x;
unsigned short y = 0xbabe;
short iy = (short) y;
### Type Conversion (6)

- **Signed → Unsigned**
  - Ordering inversion
  - Negative → Big positive

\[
T2U_w(x) = \begin{cases} 
  x + 2^w, & x < 0 \\
  x, & x \geq 0 
\end{cases}
\]

- **Example**
  - `short`  
    - \(x = 2003\);
    - `unsigned short` \(ix = (\text{unsigned short}) x;\)
  - `short`  
    - \(y = -2003\);
    - `unsigned short` \(iy = (\text{unsigned short}) y;\)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2003</td>
<td>07 D3</td>
</tr>
<tr>
<td>iy</td>
<td>63533</td>
<td>F8 2D</td>
</tr>
<tr>
<td>iix</td>
<td>2003</td>
<td>07 D3</td>
</tr>
<tr>
<td>iy</td>
<td>-2003</td>
<td>F8 2D</td>
</tr>
<tr>
<td>iy</td>
<td>63533</td>
<td>F8 2D</td>
</tr>
</tbody>
</table>

- **Two’s complement**
  - \(+2^{w-1}\) to \(+2^w - 1\)
  - \(-2^{w-1}\) to \(-2^w + 1\)

- **Unsigned Two’s complement**
  - \(x = 2003\);
  - `unsigned short` \(ix = (\text{unsigned short}) x;\)
  - `short`  
    - \(y = -2003\);
    - `unsigned short` \(iy = (\text{unsigned short}) y;\)
Type Casting in C (1)

- **Constants**
  - By default, considered to be signed integers
  - Unsigned if have “U” or “u” as suffix
    - 0U, 12345U, 0x1A2Bu

- **Type casting**
  - Explicit casting
    - ```
        int tx, ty;
        unsigned ux, uy;
        tx = (int) ux;
        uy = (unsigned) ty;
      ```
  - Implicit casting via
    - Assignments
    - Procedure calls
    - ```
        int f(unsigned);
        tx = ux;
        f(ty);
      ```
## Type Casting in C (2)

### Expression evaluation

- If mix unsigned and signed in single expression, signed values implicitly cast to **unsigned**.
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0 == 0U</code></td>
<td>unsigned</td>
<td>True</td>
</tr>
<tr>
<td><code>-1 &lt; 0</code></td>
<td>signed</td>
<td>True</td>
</tr>
<tr>
<td><code>-1 &lt; 0U</code></td>
<td>unsigned</td>
<td>?</td>
</tr>
<tr>
<td><code>-1 &gt; -2</code></td>
<td>signed</td>
<td>?</td>
</tr>
<tr>
<td><code>(unsigned) -1 &gt; -2</code></td>
<td>unsigned</td>
<td>?</td>
</tr>
<tr>
<td><code>2147483647 &gt; -2147483647-1</code></td>
<td>signed</td>
<td>?</td>
</tr>
<tr>
<td><code>2147483647U &gt; -2147483647-1</code></td>
<td>unsigned</td>
<td>?</td>
</tr>
<tr>
<td><code>2147483647 &gt; (int) 2147483648U</code></td>
<td>signed</td>
<td>?</td>
</tr>
</tbody>
</table>
Type Casting in C (3)

- Example 1-1

```c
int main ()
{
    unsigned i;
    for (i = 10; i > 0; i--)
        printf (“%u\n”, i);
}
```

- Example 1-2

```c
int main ()
{
    unsigned i;
    for (i = 10; i >= 0; i--)
        printf (“%u\n”, i);
}
```
Type Casting in C (4)

- Example 2

```c
int sum_array (int a[], unsigned len)
{
    int i;
    int result = 0;

    for (i = 0; i <= len - 1; i++)
        result += a[i];

    return result;
}
```
Example 3-1

```c
void copy_mem1 (char *src, char *dest, unsigned len)
{
    unsigned i;
    for (i = 0; i < len; i++)
        *dest++ = *src++;
}
```

Example 3-2

```c
void copy_mem2 (char *src, char *dest, unsigned len)
{
    int i;
    for (i = 0; i < len; i++)
        *dest++ = *src++;
}
```
Example 3-3

```c
void copy_mem3 (char *src, char *dest, unsigned len)
{
    for (; len > 0; len--)
        *dest++ = *src++;
}
```

Example 3-4

```c
void copy_mem4 (char *src, char *dest, unsigned len)
{
    for (; (int) len > 0; len--)
        *dest++ = *src++;
}
```
Type Casting in C (7)

- Example 4

```c
#include <stdio.h>

int main ()
{
    unsigned char c;

    while ((c = getchar()) != EOF)
        putchar (c);
}
```
Type Casting in C (8)

- Lessons
  - There are many tricky situations when you use unsigned integers – hard to debug
  - Do not use just because numbers are nonnegative
  - Use only when you need collections of bits with no numeric interpretation ("flags")
  - Few languages other than C support unsigned integers