Representing Integers

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
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 (I)

• 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

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

- Two zeros
  - \([000\ldots00]\), \([100\ldots00]\)
- 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: \[ \sim x + x = 1111...11_2 = -1 \]

– 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
  – |TMin| = Tmax + 1
    (Asymmetric range)
  – UMax = 2 * 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

```
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
Type Conversion (3)

• Sign extension example
  – Converting from smaller to larger integer type
  – C automatically performs sign extension

```
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 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 \(x\ mod\ 2^w\)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>3405691582</td>
<td>CA FE BA BE</td>
<td>11001010 11111110 10111010 10111110</td>
</tr>
<tr>
<td>(ix)</td>
<td>47806</td>
<td>BA BE</td>
<td>10111010 10111110</td>
</tr>
<tr>
<td>(y)</td>
<td>537116399</td>
<td>20 03 BE EF</td>
<td>00100000 00000011 10111110 11101111</td>
</tr>
<tr>
<td>(iy)</td>
<td>-16657</td>
<td>BE EF</td>
<td>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></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>07 D3</td>
<td>00000111 11010011</td>
</tr>
<tr>
<td>y</td>
<td>47806</td>
<td>BA BE</td>
<td>10111010 1011110</td>
</tr>
<tr>
<td>iy</td>
<td>-17730</td>
<td>BA BE</td>
<td>10111010 10111110</td>
</tr>
</tbody>
</table>
Type Conversion (6)

- Signed $\rightarrow$ Unsigned
  - Ordering inversion
  - Negative $\rightarrow$ Big positive

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

<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>y</td>
<td>-2003</td>
<td>F8 2D</td>
</tr>
</tbody>
</table>

0 2^{w-1} 2^w 0 \quad \text{Two's complement}

0 \quad \text{Signed}

0 2^{w-1} \quad \text{Unsigned}

short $x = 2003$;
unsigned short $\text{ix} = (\text{unsigned short}) x$;
short $y = -2003$;
unsigned short $\text{iy} = (\text{unsigned short}) y$;
Type Casting in C

• Constants
  – By default, considered to be signed integers
  – Unsigned if they have “U” or “u” as suffix
    • e.g. 0U, 12345U, 0x1A2Bu

• Type casting
  – Explicit casting

  – Implicit casting via
    • Assignments
    • Procedure calls

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

int f(unsigned);
    tx = ux;
f(ty);
```
Expression Evaluation in C

- 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><code>unsigned</code></td>
<td><code>True</code></td>
</tr>
<tr>
<td><code>-1 &lt; 0</code></td>
<td><code>signed</code></td>
<td><code>True</code></td>
</tr>
<tr>
<td><code>-1 &lt; 0U</code></td>
<td><code>unsigned</code></td>
<td><code>?</code></td>
</tr>
<tr>
<td><code>-1 &gt; -2</code></td>
<td><code>signed</code></td>
<td><code>?</code></td>
</tr>
<tr>
<td><code>(unsigned) -1 &gt; -2</code></td>
<td><code>unsigned</code></td>
<td><code>?</code></td>
</tr>
<tr>
<td><code>2147483647 &gt; -2147483647-1</code></td>
<td><code>signed</code></td>
<td><code>?</code></td>
</tr>
<tr>
<td><code>2147483647U &gt; -2147483647-1</code></td>
<td><code>unsigned</code></td>
<td><code>?</code></td>
</tr>
<tr>
<td><code>2147483647 &gt; (int) 2147483648U</code></td>
<td><code>signed</code></td>
<td><code>?</code></td>
</tr>
</tbody>
</table>
Example 1

```c
#include <stdio.h>

int main ()
{
    unsigned i;
    for (i = 10; i >= 0; i--)
        printf ("%u\n", i);
}
```
Example 2

```c
#include <stdio.h>

#define DELTA sizeof(int)

int main ()
{
    int i;
    for (i = 10; i - DELTA >= 0; i -= DELTA)
        printf ("%d\n", i);
}
```
Example 3

```c
#include <string.h>

int strlonger (char *s, char *t)
{
    return (strlen(s) - strlen(t)) > 0;
}
```
Example 4

```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 5

```c
#include <stdio.h>

int main ()
{
    unsigned char  c;

    while ((c = getchar()) != EOF)
        putchar (c);
}
```