CSCI-UA.0201

Computer Systems Organization

Data Representation – Integers and Floating points

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
What happens if you change the type of a variable (aka type casting)?
Signed vs. Unsigned in C

- **Constants**
  - By default, signed integers
  - Unsigned with “U” as suffix
    - \(0U\), \(4294967259U\)

- **Casting**
  - Explicit casting between signed & unsigned
    ```
    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;
    ```
General Rule for Casting:
signed <-> unsigned

Follow these two steps:
1. Keep the bit presentation
2. Re-interpret

Effect:
• Numerical value may change.
• Bit pattern stays the same.
### Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</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>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>
Casting Surprises

• Expression Evaluation
  – If there is a mix of unsigned and signed in single expression,
    *signed values implicitly cast to unsigned*
  – Including comparison operations <, >, ==, <=, >=

If there is an expression that has many types, the compiler follows these rules.
Example

```
#include <stdio.h>

int main() {
    int i = -7;
    unsigned j = 5;

    if(i > j) {
        printf("Surprise!\n");
        return 0;
    }
}
```

Condition is TRUE!
Expanding & Truncating a variable
Expanding

• Convert \( w \)-bit signed integer to \( w+k \)-bit with same value
• Convert unsigned: pad \( k \) 0 bits in front
• Convert signed: make \( k \) copies of sign bit
Sign Extension Example

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

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

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

- Example: from int to short (i.e. from 32-bit to 16-bit)
- High-order bits are truncated
- Value is altered → must reinterpret
- Can lead to buggy code! → So don't do it!
Addition, negation, multiplication, and shifting
Negation: Complement & Increment

- The two's complement of x satisfies
  \[ \text{TC}(x) + x = 0 \]
  where \( \text{TC}(x) = \sim x + 1 \)

- Proof sketch
  - Observation: \( \sim x + x = 1111\ldots111 = -1 \)
    \[ \rightarrow \sim x + x + 1 = 0 \]
    \[ \rightarrow (\sim x + 1) + x = 0 \]
    \[ \rightarrow \text{TC}(x) + x = 0 \]

\[
\begin{array}{c}
\times & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & \sim x & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
-1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

Hardware Rules for addition/subtraction

- The hardware must work with two operands of the same length.
- The hardware produces a result of the same length as the operands.
- The hardware does not differentiate between signed and unsigned.
Two's Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- If sum $\geq 2^{w-1}$, becomes negative (positive overflow)
- If sum $<-2^{w-1}$, becomes positive (negative overflow)
Signed Overflow in C

- **CAUTION:** signed overflow has undefined behavior in C!
- The compiler may assume that signed overflow never happens and exploit this in optimizations.
- Example:
  ```c
  int x = INT_MAX;
  if (x + 1 < x) printf("Overflow!");
  ```
  GCC assumes this is always FALSE!
Multiplication

• Exact Product of $w$-bit numbers $x$, $y$
  – Either signed or unsigned

• Ranges
  – Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  – Two’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  – Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
Power-of-2 Multiply with Shift

• Operation
  – $u << k$ gives $u \times 2^k$
  – Both signed and unsigned

• Examples
  – $u << 3 = u \times 8$
  – $(u << 5) - (u << 3) = u \times 24$
  – Most machines shift and add faster than multiply
    • Compiler generates this code automatically
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
  \(-u \gg k\) gives \(\lfloor u / 2^k \rfloor\)

Examples:

<table>
<thead>
<tr>
<th></th>
<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</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>(x \gg 1)</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>(x \gg 4)</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>(x \gg 8)</td>
<td>59.4257813</td>
<td>59</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
  - $x \gg k$ gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift

Examples

<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>$y \gg 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>111111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>111111111 11000100</td>
</tr>
</tbody>
</table>
Floating Points

Some slides and information about FP are adopted from Prof. Michael Overton book:
Numerical Computing with IEEE Floating Point Arithmetic
Turing Award 1989 to William Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)
Background: Fractional binary numbers

• What is $1011.101_2$?
Background: Fractional Binary Numbers

- Value: \[ \sum_{k=-j}^{i} b_k \times 2^k \]
Fractional Binary Numbers: Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.111₂</td>
</tr>
</tbody>
</table>
Why not fractional binary numbers?

• Not efficient

  – $3 \times 2^{100} \rightarrow 10100000000 \ldots 0$

  100 zeros

  – Given a finite length (e.g. 32-bits), cannot represent very large numbers nor numbers very close to 0