Representing and Manipulating Integers

Part II

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
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
    ~0x01000001 --> 10111110
  - ~0x00 --> 0xFF
    ~0x00000000 --> 11111111
  - 0x69 & 0x55 --> 0x41
    01101001 & 01010101 --> 01000001
  - 0x69 | 0x55 --> 0x7D
    01101001 | 01010101 --> 01111101
  - 0x69 ^ 0x55 --> 0x4C
    01101001 ^ 01010101 --> 00111100
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`
  - `if (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 MSB on right
    - Useful with two’s complement integer representation

- **Undefined if** \( y < 0 \) **or** \( y \geq \) word size
Addition (1)

- **Integer addition example**
  - 4-bit integers $u$, $v$
  - Compute true sum
  - True sum requires one more bit ("carry")
  - Values increase linearly with $u$ and $v$
  - Forms planar surface
Addition (2)

- **Unsigned addition**
  - Ignores carry output
  - Wraps around
    - If true sum $\geq 2^w$
    - At most once

---

**True Sum**

$2^{w+1}$

Overflow

$2^w$

$0$

**Unsigned addition**
Addition (3)

- **Signed addition**
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer

\[ +2^w \]

- **True Sum**

- Positive overflow

- Negative overflow

- Signed addition

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

\[ 0 \]

\[ -2^{w-1} \]

\[ -2^w \]

Two’s complement addition (4-bit word)

Positive overflow

Negative overflow
Addition (4)

- **Signed addition in C**
  - Ignores carry output
  - The low-order \( w \) bits are identical to unsigned addition

<table>
<thead>
<tr>
<th>Mode</th>
<th>x</th>
<th>y</th>
<th>x + y</th>
<th>Truncated x + y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned</td>
<td>4 [100]</td>
<td>3 [011]</td>
<td>7 [0111]</td>
<td>7 [111]</td>
</tr>
<tr>
<td>Two’s comp.</td>
<td>3 [011]</td>
<td>3 [011]</td>
<td>6 [0110]</td>
<td>-2 [110]</td>
</tr>
</tbody>
</table>

Examples for \( w = 3 \)
Multiplication (1)

- **Ranges of \((x \times y)\)**
  - Unsigned: up to \(2^w\) bits
    \[
    0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1
    \]
  - Two’s complement min: up to \(2^{w-1}\) bits
    \[
    x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}
    \]
  - Two’s complement max: up to \(2^w\) bits (only for TMin\(^2\))
    \[
    x \times y \leq (-2^{w-1})^2 = 2^{2w-2}
    \]

- **Maintaining exact results**
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Multiplication (2)

- **Unsigned multiplication in C**
  - Ignores high order \( w \) bits
  - Implements modular arithmetic

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

### Diagram

**Operands:** \( w \) bits

**True Product:** \( 2^w \) bits \( u \cdot v \)

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

\[ \text{UMult}_w(u, v) \]
### Signed multiplication in C

- Ignores high order \( w \) bits
- The low-order \( w \) bits are identical to unsigned multiplication

#### Examples for \( w = 3 \)

<table>
<thead>
<tr>
<th>Mode</th>
<th>( x )</th>
<th>( y )</th>
<th>( x \cdot y )</th>
<th>Truncated ( x \cdot y )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned</strong></td>
<td>5 [101]</td>
<td>3 [011]</td>
<td>15 [001111]</td>
<td>7 [111]</td>
</tr>
<tr>
<td><strong>Unsigned</strong></td>
<td>4 [100]</td>
<td>7 [111]</td>
<td>28 [011100]</td>
<td>4 [100]</td>
</tr>
<tr>
<td>Two’s comp.</td>
<td>-4 [100]</td>
<td>-1 [111]</td>
<td>4 [000100]</td>
<td>-4 [100]</td>
</tr>
<tr>
<td><strong>Unsigned</strong></td>
<td>3 [011]</td>
<td>3 [011]</td>
<td>9 [001001]</td>
<td>1 [001]</td>
</tr>
<tr>
<td>Two’s comp.</td>
<td>3 [011]</td>
<td>3 [011]</td>
<td>9 [001001]</td>
<td>1 [001]</td>
</tr>
</tbody>
</table>
Multiplication (4)

- **Power-of-2 multiply with shift**
  - $u << k$ gives $u \cdot 2^k$
    - e.g., $u << 3 == u \cdot 8$
  - Both signed and unsigned
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdot 2^k$</td>
<td>$0 \cdots 010 \cdots 00$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product: $w+k$ bits</th>
<th>$u \cdot 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UMult_w(u, 2^k)$</td>
<td>$\cdots 0 \cdots 00$</td>
</tr>
<tr>
<td>$TMult_w(u, 2^k)$</td>
<td>$\cdots 0 \cdots 00$</td>
</tr>
</tbody>
</table>
Compiled multiplication code

- C compiler automatically generates shift/add code when multiplying by constant

### C Function

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

### Compiled Arithmetic Operations

- `leal (%eax, %eax, 2), %eax` ; \( t \leftarrow x + x \times 2 \)
- `sall $2, %eax` ; return \( t \ll 2 \)
Divison (1)

- **Unsigned power-of-2 divide with shift**
  - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift

<table>
<thead>
<tr>
<th>Expression</th>
<th>Division</th>
<th>Result</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>
Division (2)

- Compiled unsigned division code
  - Uses logical shift for unsigned
  - Logical shift written as >>> in Java

C Function

```c
unsigned udiv8 (unsigned x) {
    return x / 8;
}
```

Compiled Arithmetic Operations

```asm
shrl $3, %eax ; return t >> 3
```
Division (3)

- Signed power-of-2 divide with shift
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - Uses arithmetic shift (rounds wrong direction if \( x < 0 \))

Operands:

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

Division:

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

Result:

\[
\text{RoundDown}(x / 2^k)
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Division</th>
<th>Result</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y \gg 1 )</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Division (4)

- **Correct power-of-2 divide**
  - Want $\left\lfloor \frac{x}{2^k} \right\rfloor$ (Round Toward 0) when $x < 0$
  - 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**
  - Biasing has no effect

<table>
<thead>
<tr>
<th>Dividend: $x$</th>
<th>$+2^k-1$</th>
<th>$2^k$</th>
<th>$\left\lfloor \frac{x}{2^k} \right\rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 ... 0 ... 0 0$</td>
<td>$0 ... 0 0 1 ... 1 1$</td>
<td>$0 ... 0 1 0 ... 0 0$</td>
<td>$1 ... 1 1 ... 1 1$</td>
</tr>
</tbody>
</table>

Binary Point
**Division (5)**

- **Case 2: Rounding**
  - Biasing adds 1 to final result

![Division Diagram](image)

- **Dividend:**
  - $x + 2^k - 1$
  - Incremented by 1

- **Divisor:**
  - $2^k$
  - Incremented by 1

- **Binary Point**
Division (6)

- Compiled signed division code
  - Uses arithmetic shift for signed
  - Arithmetic shift written as >> in Java

```
C Function

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

Explanation

if (x < 0)
    x += 7;
return x >> 3;
```