Manipulating Integers

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)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~0x41 → 0xBE</td>
<td>~01000001₂ → 10111110₂</td>
</tr>
<tr>
<td>~0x00 → 0xFF</td>
<td>~00000000₂ → 11111111₂</td>
</tr>
<tr>
<td>0x69 &amp; 0x55 → 0x41</td>
<td>01101001₂ &amp; 01010101₂ → 01000001₂</td>
</tr>
<tr>
<td>0x69</td>
<td>0x55 → 0x7D</td>
</tr>
<tr>
<td>0x69 ^ 0x55 → 0x3C</td>
<td>01101001₂ &amp; 01010101₂ → 00111100₂</td>
</tr>
</tbody>
</table>
Logic Operations in C

- &&, ||, !
  - View 0 as “False”, anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

- Examples (char data type)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>!0x41</td>
<td>0x00</td>
</tr>
<tr>
<td>!0x00</td>
<td>0x01</td>
</tr>
<tr>
<td>!!0x41</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69 &amp;&amp; 0x55</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69</td>
<td></td>
</tr>
</tbody>
</table>

if (p && *p) ... // avoids null pointer access
Shift Operations

- **Left shift:** \( x << y \)
  - Shift bit-vector \( x \) left \( y \) positions
    - Throw away extra bits on left
    - Fill with 0’s on right

- **Right shift:** \( x >> y \)
  - 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 (I)

- 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

\[
\begin{align*}
\text{Overflow} & \quad 2^{w+1} \\
\text{Unsigned addition} & \quad 0 \\
\text{True Sum} & \quad 2^w
\end{align*}
\]
Addition (3)

- Signed addition
  - Drop off MSB
  - Treat remaining bits as two’s complement integers

```
Positive overflow

Signed addition

True Sum

+2^w

+2^{w-1}

0

-2^{w-1}

-2^w

Negative overflow

Positive overflow

Negative overflow
```

Two's complement addition (4-bit word)
Addition (4)

- Signed addition in C
  - Ignores carry output
  - The low order $w$ bits are identical to unsigned addition

### Examples for $w = 3$

<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><strong>Unsigned</strong></td>
<td>4 [100]</td>
<td>3 [011]</td>
<td>7 [0111]</td>
<td>7 [111]</td>
</tr>
<tr>
<td><strong>Unsigned</strong></td>
<td>4 [100]</td>
<td>7 [111]</td>
<td>11 [1011]</td>
<td>3 [011]</td>
</tr>
<tr>
<td><strong>Unsigned</strong></td>
<td>3 [011]</td>
<td>3 [011]</td>
<td>6 [0110]</td>
<td>6 [110]</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>
Multiplication (1)

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

\[
UMult_w(u, v) = u \cdot v \mod 2^w
\]

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits
Multiplication (3)

- Signed multiplication in C
  - Ignores high order $w$ bits
  - The low-order $w$ bits are identical to unsigned multiplication

| Mode          | $x$   | $y$   | $x \cdot y$     | Truncated $x \cdot y$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>

Examples for $w = 3$
Multiplication (4)

- Power-of-2 multiply with shift
  - $u \ll k$ gives $u \times 2^k$
    - e.g. $u \ll 3 = u \times 8$, $(u \ll 5) - (u \ll 3) = u \times 24$
  - Both signed and unsigned
  - Most machines shift and add faster than multiply

Operands: $w$ bits

True Product: $w+k$ bits

Discard $k$ bits: $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th>$\times 2^k$</th>
<th>$u \cdot 2^k$</th>
<th>$\text{UMult}_w(u, 2^k)$</th>
<th>$\text{TMult}_w(u, 2^k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$2^k$</td>
<td>$u \cdot 2^k$</td>
<td>$0 \ldots 010 \ldots 00$</td>
<td>$0 \ldots 00$</td>
</tr>
</tbody>
</table>

$k$
Multiplication (5)

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

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

Compiled Arithmetic Operations

- `leal (%eax, %eax, 2), %eax` ; t ← x + x * 2
- `sall $2, %eax` ; return t << 2
Division (1)

- Unsigned power-of-2 divide with shift
  - $u >> k$ 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 10101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 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

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

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

<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 &gt;&gt; 1 )</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y &gt;&gt; 4 )</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y &gt;&gt; 8 )</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
<td>1111111 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:</th>
<th>( x )</th>
<th>( +2^k - 1 )</th>
<th>( \frac{x}{2^k} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 ( \cdots ) 0 ( \cdots ) 0 0</td>
<td>0 ( \cdots ) 0 0 1 ( \cdots ) 1 1</td>
<td>0 ( \cdots ) 0 1 0 ( \cdots ) 0 0</td>
</tr>
</tbody>
</table>

Binary Point

Divisor: \( / 2^k \)
Division (5)

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

\[
x + 2^k - 1
\]

\[
\left\lfloor \frac{x}{2^k} \right\rfloor
\]

\[
\frac{x}{2^k}
\]

\[
k
\]

**Dividend:**

\[
\begin{array}{c}
x  \\
+2^k - 1
\end{array}
\]

**Divisor:**

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

**Binary Point**

Incremented by 1
Divison (6)

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

### Compiled Arithmetic Operations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>testl</td>
<td>%eax, %eax</td>
</tr>
<tr>
<td>js</td>
<td>L4</td>
</tr>
<tr>
<td>L3:</td>
<td></td>
</tr>
<tr>
<td>sarl</td>
<td>$3, %eax</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>L4:</td>
<td></td>
</tr>
<tr>
<td>addl</td>
<td>$7, %eax</td>
</tr>
<tr>
<td>jmp</td>
<td>L3</td>
</tr>
</tbody>
</table>

### C Function

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

### Explanation

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