Manipulating Integers

Jinkyu Jeong (jinkyu@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>Binary Value</th>
<th>Result</th>
<th>Binary Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~0x41 → 0xBE</td>
<td>~01000001₂ → 10111110₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~0x00 → 0xFF</td>
<td>~00000000₂ → 11111111₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x69 &amp; 0x55 → 0x41</td>
<td>01101001₂ &amp; 01010101₂ → 01000001₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x69</td>
<td>0x55 → 0x7D</td>
<td>01101001₂ &amp; 01010101₂ → 01111101₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x69 ^ 0x55 → 0x3C</td>
<td>01101001₂ &amp; 01010101₂ → 00111100₂</td>
<td></td>
<td></td>
<td></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 \( y \geq \text{word size} \)
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 two’s complement integers
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>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>
Multiplication (I)

- **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

\[
UMult_w(u, v) = u \cdot v \mod 2^w
\]
Multiplication (3)

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

<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>Unsigned</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>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 \( u \cdot 2^k \)
    - e.g. \( u \ll 3 = u \cdot 8 \), \( (u \ll 5) - (u \ll 3) = u \cdot 24 \)
  - Both signed and unsigned
  - Most machines shift and add faster than multiply

- True Product: \( w+k \) bits
- Discard \( k \) bits: \( w \) bits

Operands: \( w \) bits

<table>
<thead>
<tr>
<th>True Product: ( w+k ) bits</th>
<th>Discard ( k ) bits: ( w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \cdot 2^k )</td>
<td>UMult(_w(u, 2^k))</td>
</tr>
<tr>
<td>Discard ( k ) bits: ( w ) bits</td>
<td>TMult(_w(u, 2^k))</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

```assembly
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

Operations:

<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>$x &gt;&gt; 1$</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

```assembly
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 \( x < 0 \))

**Operands:**

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

**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 &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>11111111 11000100</td>
</tr>
</tbody>
</table>
Division (4)

• Correct power-of-2 divide
  – Want \( \lfloor x / 2^k \rfloor \) (Round Toward 0) when \( x < 0 \)
  – Compute as \( \lfloor (x + 2^k - 1) / 2^k \rfloor \)
    • In C: \( (x + (1 << k) - 1) >> k \)
    • Biases dividend toward 0

• Case 1: No rounding
  – Biasing has no effect

\[
\begin{array}{c}
\text{Dividend:} \\
\hline
x \quad & 1 \, \cdots \, 0 \, \cdots \, 0 \, 0 \\
+2^k - 1 \quad & 0 \, \cdots \, 0 \, 0 \, 1 \, \cdots \, 1 \, 1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{Divisor:} \\
\hline
/ \, 2^k \\
\hline
\lfloor x / 2^k \rfloor \quad & 1 \, \cdots \, 1 \, \cdots \, 1 \, 1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{Binary Point} \\
\hline
1 \, \cdots \, 1 \, 1 \\
\hline
\end{array}
\]
Division (5)

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

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

\[
\begin{array}{c}
\text{Divisor:} \\
2^k \\
\hline
\left\lceil \frac{x}{2^k} \right\rceil \\
\end{array}
\]

- Increased by 1
- Biasing adds 1 to final result
Division (6)

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

### Compiled Arithmetic Operations

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

### C Function

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

### Explanation

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