Representing and Manipulating Integers (3/3)

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
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

$2^w$

Unsinged addition
Addition (3)

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

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

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>
- **Power-of-2 multiply with shift**
  - $u << k$ gives $u \times 2^k$
    - e.g., $u << 3 = u \times 8$
  - 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>
<th>True Product: $w+k$ bits</th>
<th>$u \times 2^k$</th>
<th>Discard $k$ bits: $w$ bits</th>
<th>$U\text{Mult}_w(u, 2^k)$</th>
<th>$T\text{Mult}_w(u, 2^k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\bullet \bullet \bullet \bullet$</td>
<td>$0 \bullet \bullet \bullet 0 1 0$</td>
<td>$\bullet \bullet \bullet 0 \bullet \bullet \bullet 0 0$</td>
<td>$\bullet \bullet \bullet 0 \bullet \bullet \bullet 0 0$</td>
<td>$\bullet \bullet \bullet 0 \bullet \bullet \bullet 0 0$</td>
<td>$\bullet \bullet \bullet 0 \bullet \bullet \bullet 0 0$</td>
<td>$\bullet \bullet \bullet 0 \bullet \bullet \bullet 0 0$</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

```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

<table>
<thead>
<tr>
<th>Operands:</th>
<th>$u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division:</th>
<th>$u / 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result:</th>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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

```
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:**

\[
\begin{array}{c|c|c}
  & x & \text{Binary Point} \\
\hline
  & \begin{array}{c}
    \cdots \\
    \cdots \\
  \end{array} & \begin{array}{c}
    \cdots \\
    \cdots \\
  \end{array} \\
\hline
  / \begin{array}{c}
    0 \quad \cdots \\
    0 \quad 1 \quad 0 \\
  \end{array} & 2^k & \begin{array}{c}
    \cdots \\
    \cdots \\
    \cdots \\
  \end{array} \\
\hline
\end{array}
\]

**Division:**

\[
\begin{array}{c|c|c}
  & x / 2^k & \text{Result} \\
\hline
  & \begin{array}{c}
    \cdots \\
    \cdots \\
    \cdots \\
  \end{array} & \begin{array}{c}
    \cdots \\
    \cdots \\
  \end{array} \\
\hline
\end{array}
\]

**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

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>$x$</th>
<th>+2$^k$–1</th>
<th>$x / 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...0</td>
<td>0</td>
<td>0...1</td>
<td>1...1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>$/ 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...0</td>
<td>010...0</td>
</tr>
</tbody>
</table>

Binary Point

\[ \lfloor x / 2^k \rfloor \]
## Case 2: Rounding

- Biasing adds 1 to final result

### Dividend:

- $x$
- $+2^k - 1$

### Divisor:

- $\div 2^k$

- $\left\lceil x / 2^k \right\rceil$

- Incremented by 1

- Binary Point
Division (6)

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

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

Compiled Arithmetic Operations

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

Explanation

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