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>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>~0x41</td>
<td>0x41</td>
<td>0xBE</td>
</tr>
<tr>
<td>~0x00</td>
<td>0x00</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x69 &amp; 0x55</td>
<td>0x69</td>
<td>0x41</td>
</tr>
<tr>
<td>0x69</td>
<td>0x55</td>
<td>0x7D</td>
</tr>
<tr>
<td>0x69 ^ 0x55</td>
<td>0x69</td>
<td>0x3C</td>
</tr>
</tbody>
</table>

\[ \begin{array}{|c|c|}
\hline
\text{~0x41} & \text{0xBE} \\
\hline
\text{~0x00} & \text{0xFF} \\
\hline
\text{0x69} & \text{0x55} \rightarrow \text{0x41} \\
\hline
\text{0x69} & \text{0x55} \rightarrow \text{0x7D} \\
\hline
\text{0x69} & \text{0x55} \rightarrow \text{0x3C} \\
\hline
\end{array} \]
### 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>

```c
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

<table>
<thead>
<tr>
<th>Argument x</th>
<th>&lt;&lt; 3</th>
<th>Log. &gt;&gt; 2</th>
<th>Arith. &gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100010</td>
<td>00010000</td>
<td>00011000</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>&lt;&lt; 3</th>
<th>Log. &gt;&gt; 2</th>
<th>Arith. &gt;&gt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100010</td>
<td>00010000</td>
<td>00101000</td>
<td>11101000</td>
</tr>
</tbody>
</table>
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
Addition (3)

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

**Positive overflow**

**Negative overflow**

**True Sum**

**Signed addition**

Two's complement addition (4-bit word)

Negative overflow

Positive overflow
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 Tmin²)
    
    \[
    x \times y \leq (-2^{w-1})^2 = 2^{2w-2}
    \]

• 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

\[
\begin{array}{c}
\text{Operands: } \mathbf{w} \text{ bits} \\
\text{True Product: } 2^w \text{ bits} \\
\text{Discard } w \text{ bits: } w \text{ bits}
\end{array}
\]

\[
\begin{array}{c}
\text{UMult}_w(u, v)
\end{array}
\]
## Multiplication (3)

- **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**
  - $u \ll k$ 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

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u \times 2^k$</th>
<th>True Product: $w+k$ bits</th>
<th>Discard $k$ bits: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$\times 2^k$</td>
<td>$u \times 2^k$</td>
<td>UMult$_w(u, 2^k)$</td>
</tr>
<tr>
<td></td>
<td>$0 \cdots 010 \cdots 00$</td>
<td>$\cdots 0 \cdots 00$</td>
<td>TMult$_w(u, 2^k)$</td>
</tr>
</tbody>
</table>

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Multiplication (5)

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

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

C Function

Compiled Arithmetic Operations

```
leal (%eax, %eax, 2), %eax ; t ← x + x * 2
sll $2, %eax ; return t << 2
```
Division (I)

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

**Operands:**

\[
\frac{u}{2^k} = \begin{array}{c}
\begin{array}{cccc}
0 & \cdots & 0 & 1 \cdots 0 \cdots 0
\end{array}
\end{array}
\]

**Division:**

\[
\frac{u}{2^k} = \begin{array}{c}
\begin{array}{cccc}
\cdots & \cdots & \cdots & \cdots
\end{array}
\end{array}
\]

**Result:**

\[
\lfloor \frac{u}{2^k} \rfloor = \begin{array}{c}
\cdots
\end{array}
\]

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

C Function

Compiled Arithmetic Operations

```
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:

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

\[
\begin{array}{c}
0 \quad \cdots \quad 0 \quad 1 \quad 0 \quad \cdots \quad 0 \quad 0
\end{array}
\]

Division:

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

\[
\begin{array}{c}
\cdots \quad \cdots \quad \cdots \quad \cdots
\end{array}
\]

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>111111100 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 $\left\lfloor x / 2^k \right\rfloor$ (Round Toward 0) when $x < 0$
  – Compute as $\left\lfloor (x + 2^k - 1) / 2^k \right\rfloor$
    • 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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>⋯ 0 ⋯ 0 0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>⋯ 0 0 1 ⋯ 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>⋯ 1 ⋯ 1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>$x / 2^k$</td>
<td>/ 2$^k$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>⋯ 11 ⋯ ⋯</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>⋯ 0 10 ⋯ 0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>⋯ 1 11 ⋯ 1</td>
</tr>
<tr>
<td>Binary Point</td>
<td>1</td>
<td>⋯ 11</td>
</tr>
</tbody>
</table>

SSE2030: Introduction to Computer Systems, Spring 2018, Jinkyu Jeong (jinkyu@skku.edu)
Division (5)

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

Dividend:

\[
x + 2^k - 1 = \underbrace{0\cdots001\cdots11}_{+2^k-1}
\]

Divisor:

\[
x / 2^k = \underbrace{0\cdots010\cdots00}_{/2^k}
\]

\[
x / 2^k = \underbrace{1\cdots111\cdots}_{\lfloor x / 2^k \rfloor}
\]

Incremented by 1

Binary Point
Division (6)

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

Compiled Arithmetic Operations

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Register</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>testl</td>
<td>%eax, %eax</td>
<td>L6</td>
<td></td>
</tr>
<tr>
<td>js</td>
<td></td>
<td>L4</td>
<td></td>
</tr>
<tr>
<td>L3:</td>
<td>sarl $3, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4:</td>
<td>addl $7, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td></td>
<td>L3</td>
<td></td>
</tr>
</tbody>
</table>

C Function

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

Explantion

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