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
    - ~01000001₂ → 10111110₂
  - ~0x00 → 0xFF
    - ~00000000₂ → 11111111₂
  - 0x69 & 0x55 → 0x41
    - 01101001₂ & 01010101₂ → 01000001₂
  - 0x69 | 0x55 → 0x7D
    - 01101001₂ | 01010101₂ → 01111111₂
  - 0x69 ^ 0x55 → 0x4C
    - 01101001₂ ^ 01010101₂ → 00111100₂
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

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

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</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 ≥ $2^w$
    - At most once

**True Sum**

```
0 2^w 2^{w+1}
```

Overflow

Unsigned addition

Unsigned addition (4-bit word)
Addition (3)

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

Signed addition

\[
\begin{align*}
+2^w & \\
+2^{w-1} & \\
0 & \\
-2^{w-1} & \\
-2^w & \\
\end{align*}
\]

Two's complement addition (4-bit word)

Positive overflow

True Sum

Negative overflow

Signed addition

Negative overflow

Positive 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

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

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

**True Product: \( 2^w \) bits**

**Discard \( w \) bits: \( w \) bits**
### 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><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., \( u \ll 3 = u \times 8 \)
  - Both signed and unsigned
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

### Diagram:

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

\[
\begin{array}{c}
\text{u} \\
\times 2^k \\
\end{array}
\]

**True Product:** \( w+k \) bits

\[
\begin{array}{c}
\text{u} \times 2^k \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{UMult}_w(u, 2^k) \\
\text{TMult}_w(u, 2^k) \\
\end{array}
\]}
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

\[ u \quad \text{Operand} \quad / \quad 2^k \quad \text{Division} \quad u / 2^k \quad \text{Result} \]

<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

```
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>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:</th>
<th>$x$</th>
<th>$+$</th>
<th>$2^k - 1$</th>
<th>$\left\lfloor \frac{x}{2^k} \right\rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>00</td>
<td>11111111...11</td>
</tr>
<tr>
<td>$x + (1 &lt;&lt; k) - 1$</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>11111111...11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>$/$</th>
<th>$2^k$</th>
<th>$\left\lfloor \frac{x}{2^k} \right\rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>11111111...11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111...11</td>
</tr>
</tbody>
</table>
Case 2: Rounding

- Biasing adds 1 to final result

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

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

- Incremented by 1
- Binary Point

- Incremented by 1

- Incremented by 1
Compiled signed division code

- Uses arithmetic shift for signed
- Arithmetic shift written as $>>$ in Java

### Compiled Arithmetic Operations

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Operand(s)</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>L3:</td>
<td>sarl $3, %eax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4:</td>
<td>addl $7, %eax</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>L3</td>
<td></td>
</tr>
</tbody>
</table>

### C Function

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

### Explanation

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