15-213
“The course that gives CMU its Zip!”

Integer Arithmetic Operations
Sept. 7, 2000

Topics

• Basic operations
  – Addition, negation, multiplication

• Programming Implications
  – Consequences of overflow
  – Using shifts to perform power-of-2 multiply/divide
C Puzzles

• Assume machine with 32 bit word size, two’s complement integers
• For each of the following C expressions, either:
  – Argue that is true for all argument values
  – Give example where not true

  • $x < 0 \Rightarrow ((x*2) < 0)$
  • $ux >= 0$
  • $x & 7 == 7 \Rightarrow (x<<30) < 0$
  • $ux > -1$
  • $x > y \Rightarrow -x < -y$
  • $x * x >= 0$
  • $x > 0 && y > 0 \Rightarrow x + y > 0$
  • $x >= 0 \Rightarrow -x <= 0$
  • $x <= 0 \Rightarrow -x >= 0$

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Unsigned Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

\[
\begin{align*}
  s &= UAdd_w(u, v) \\
  &= u + v \mod 2^w \\

  UAdd_w(u,v) &= \begin{cases} \\
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w \\
  \end{cases}
\end{align*}
\]
Visualizing Integer Addition

Integer Addition

- 4-bit integers $u$ and $v$
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with $u$ and $v$
- Forms planar surface
Visualizing Unsigned Addition

Wraps Around
• If true sum $\geq 2^w$
• At most once

True Sum
$2^{w+1}$
$2^w$
0

Overflow

Modular Sum
Mathematical Properties

Modular Addition Forms an Abelian Group

• Closed under addition
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]

• Commutative
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]

• Associative
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]

• 0 is additive identity
  \[ \text{UAdd}_w(u, 0) = u \]

• Every element has additive inverse
  - Let \[ \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Detecting Unsigned Overflow

Task
• Given $s = \text{UAdd}_w(u, v)$
• Determine if $s = u + v$

Application
unsigned s, u, v;
s = u + v;
• Did addition overflow?

Claim
• Overflow iff $s < u$
  $\text{ovf} = (s < u)$
• Or symmetrically iff $s < v$

Proof
• Know that $0 \leq v < 2^w$
• No overflow $\Rightarrow s = u + v \geq u + 0 = u$
• Overflow $\Rightarrow s = u + v - 2^w < u + 0 = u$
Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  
  ```c
  int s, t, u, v;
  s = (int) ((unsigned) u + (unsigned) v);
  t = u + v
  ```

- Will give $s == t$
Characterizing TAdd

Functionality

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

$$TAdd_w(u,v) = \begin{cases} 
  u + v + 2^w & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^w & TMax_w < u + v \quad \text{(PosOver)}
\end{cases}$$
Visualizing 2’s Comp. Addition

Values
- 4-bit two’s comp.
- Range from -8 to +7

Wraps Around
- If sum ≥ $2^{w-1}$
  - Becomes negative
  - At most once
- If sum < $-2^{w-1}$
  - Becomes positive
  - At most once
Detecting 2’s Comp. Overflow

Task
- Given \( s = T\text{Add}_w(u, v) \)
- Determine if \( s = \text{Add}_w(u, v) \)
- Example
  ```
  int s, u, v;
  s = u + v;
  ```

Claim
- Overflow iff either:
  - \( u, v < 0, s \geq 0 \) (NegOver)
  - \( u, v \geq 0, s < 0 \) (PosOver)
- \( \text{ovf} = (u<0 == v<0) \&\& (u<0 != s<0); \)

Proof
- Easy to see that if \( u \geq 0 \) and \( v < 0 \), then \( T\text{Min}_w \leq u + v \leq T\text{Max}_w \)
- Symmetrically if \( u < 0 \) and \( v \geq 0 \)
- Other cases from analysis of \( T\text{Add} \)
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

• \( T\text{Add}_w(u, v) = \text{U2T}(U\text{Add}_w(T2U(u), T2U(v))) \)
  – Since both have identical bit patterns

Two’s Complement Under TAdd Forms a Group

• Closed, Commutative, Associative, 0 is additive identity
• Every element has additive inverse
  Let \( T\text{Comp}_w(u) = \text{U2T}(U\text{Comp}_w(T2U(u))) \)
  \( T\text{Add}_w(u, T\text{Comp}_w(u)) = 0 \)

\[
T\text{Comp}_w(u) = \begin{cases} 
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w 
\end{cases}
\]
Two’s Complement Negation

 Mostly like Integer Negation
  • $T\text{Comp}(u) = -u$

$T\text{Min}$ is Special Case
  • $T\text{Comp}(T\text{Min}) = T\text{Min}$

Negation in C is Actually $T\text{Comp}$
  • $m \times = \neg x$
  • $m \times = T\text{Comp}(x)$
  • Computes additive inverse for $T\text{Add}$
    $x + \neg x = 0$

\[ \begin{array}{cccccccc}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\end{array} \]
Negating with Complement & Increment

In C
\[ \sim x + 1 = -x \]

Complement
- Observation: \[ \sim x + x = 1111...11_2 = -1 \]

\[
\begin{array}{c}
x \ 10011101 \\
+ \sim x \ 01100010 \\
\hline
-1 \ 11111111
\end{array}
\]

Increment
- \[ \sim x + x + (\sim x + 1) = -1 + (\sim x + 1) \]
- \[ \sim x + 1 = -x \]

Warning: Be cautious treating int’s as integers
- OK here: We are using group properties of TAdd and TComp
Comp. & Incr. Examples

\( x = 15213 \)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x  )</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( \sim x )</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>( \sim x + 1 )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

\( T\text{Min} \)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T\text{Min} )</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>( \sim T\text{Min} )</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>( \sim T\text{Min} + 1 )</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
</tbody>
</table>

\( 0 \)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>( \sim 0 )</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>( \sim 0 + 1 )</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Comparing Two’s Complement Numbers

Task

- **Given** signed numbers $u$, $v$
- Determine whether or not $u > v$
  - Return 1 for numbers in shaded region below

![Diagram showing the comparison of two's complement numbers]

Bad Approach

- Test $(u - v) > 0$
  - $TSub(u, v) = TAdd(u, TComp(v))$
- Problem: Thrown off by either Negative or Positive Overflow
Comparing with TSub

Will Get Wrong Results

- **NegOver**: \( u < 0, v > 0 \)
  - but \( u-v > 0 \)
- **PosOver**: \( u > 0, v < 0 \)
  - but \( u-v < 0 \)
Working Around Overflow Problems

Partition into Three Regions

- $u < 0$, $v \geq 0 \implies u < v$
- $u \geq 0$, $v < 0 \implies u > v$
- $u$, $v$ same sign $\implies u-v$ does not overflow
  - Can safely use test $(u-v) > 0$
Multiplication

Computing Exact Product of $w$-bit numbers $x, y$
  - Either signed or unsigned

Ranges
  - **Unsigned**: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2^w$ bits
  - **Two’s complement min**: $x \times y \geq (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - **Two’s complement max**: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $TMin_w^2$

Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
  - Also implemented in Lisp, ML, and other “advanced” languages
Unsigned Multiplication in C

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits

\[
\text{Standard Multiplication Function}
\]
- Ignores high order \( w \) bits

\[
\text{Implements Modular Arithmetic}
\]
\[
\text{UMult}_w(u, v) = u \cdot v \mod 2^w
\]
Unsigned vs. Signed Multiplication

Unsigned Multiplication

unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
• Truncates product to \( w \)-bit number \( up = \text{UMult}_w(ux, uy) \)
• Simply modular arithmetic
  \[ up = ux \cdot uy \mod 2^w \]

Two’s Complement Multiplication

int x, y;
int p = x * y;
• Compute exact product of two \( w \)-bit numbers \( x, y \)
• Truncate result to \( w \)-bit number \( p = \text{TMult}_w(x, y) \)

Relation
• Signed multiplication gives same bit-level result as unsigned
  \( up == (\text{unsigned}) p \)
Multiplication Examples

```c
short int x = 15213;
int txx = ((int) x) * x;
int xx = (int) (x * x);
int xx2 = (int) (2 * x * x);
```

| x      | 15213: | 00111011 01101101 |
| txx    | 231435369: | 00001101 11001011 01101100 01101001 |
| xx     | 27753: | 00000000 00000000 01101100 01101001 |
| xx2    | -10030: | 11111111 11111111 11011000 11010010 |

Observations

- **Casting order important**
  - If either operand `int`, will perform `int` multiplication
  - If both operands `short int`, will perform `short int` multiplication

- **Really is modular arithmetic**
  - Computes for `xx`: $15213^2 \mod 65536 = 27753$
  - Computes for `xx2`: `(int) 55506U = -10030`

- **Note that `xx2 == (xx << 1)`**
Power-of-2 Multiply with Shift

Operation

• $u << k$ gives $u \times 2^k$

• Both signed and unsigned

Examples

• $u << 3 = u \times 8$

• $u << 5 - u << 3 = u \times 24$

• Most machines shift and add much faster than multiply
  
  – Compiler will generate this code automatically
**Unsigned Power-of-2 Divide with Shift**

**Quotient of Unsigned by Power of 2**

- \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</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>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 4 )</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 8 )</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
2’s Comp Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- $u > k$ gives $\lfloor u / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$

\[ u > > k \]

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

\[ u / 2^k \]

\[ \text{RoundDown}(u / 2^k) \]

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</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>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

• Want \( \left\lfloor \frac{u}{2^k} \right\rfloor \) (Round Toward 0)
• Compute as \( \left\lfloor \frac{u + 2^k - 1}{2^k} \right\rfloor \)
  – In C: \((u + (1<<k)-1) >> k\)
  – Biases dividend toward 0

Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>1</th>
<th>⋯</th>
<th>⋯</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2^k + 1</td>
<td>0</td>
<td>⋯</td>
<td>0</td>
<td>1</td>
<td>⋯</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend:

\[ u \]

\[ +2^k + 1 \]

Divisor:

\[ \div 2^k \]

\[ \left\lfloor u / 2^k \right\rfloor \]

Incremented by 1

Incremented by 1

Biasing adds 1 to final result
## Correct Power-of-2 Divide Examples

<table>
<thead>
<tr>
<th>$y/2^k$</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y+1$</td>
<td>-15212</td>
<td>C4 94</td>
<td>11000100 10010100</td>
</tr>
<tr>
<td>$(y+1) &gt;&gt; 1$</td>
<td>-7606.5</td>
<td>E2 4A</td>
<td>11100010 01001010</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y+15$</td>
<td>-15197</td>
<td>C4 A2</td>
<td>11000100 10100010</td>
</tr>
<tr>
<td>$(y+15) &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>FC 4A</td>
<td>11111100 01001010</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y+255$</td>
<td>-14958</td>
<td>C5 92</td>
<td>11000101 10010010</td>
</tr>
<tr>
<td>$(y+255) &gt;&gt; 8$</td>
<td>-59.4257813</td>
<td>FF C5</td>
<td>11111111 11000101</td>
</tr>
</tbody>
</table>
Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms
Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \[
  0 \leq \text{UMult}_w(u, v) \leq 2^w - 1
  \]
- Multiplication Commutative
  \[
  \text{UMult}_w(u, v) = \text{UMult}_w(v, u)
  \]
- Multiplication is Associative
  \[
  \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)
  \]
- 1 is multiplicative identity
  \[
  \text{UMult}_w(u, 1) = u
  \]
- Multiplication distributes over addition
  \[
  \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))
  \]
Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

• Unsigned multiplication and addition
  – Truncating to \( w \) bits
• Two’s complement multiplication and addition
  – Truncating to \( w \) bits

Both Form Rings

• Isomorphic to ring of integers mod \( 2^w \)

Comparison to Integer Arithmetic

• Both are rings
• Integers obey ordering properties, e.g.,
  \[ u > 0 \implies u + v > v \]
  \[ u > 0, v > 0 \implies u \cdot v > 0 \]
• These properties are not obeyed by two’s complement arithmetic
  \[ \text{TMax} + 1 = \text{TMin} \]
  \[ 15213 \times 30426 = -10030 \] (16-bit words)
C Puzzle Answers

• Assume machine with 32 bit word size, two’s complement integers
• \( T_{Min} \) makes a good counterexample in many cases

- \( x < 0 \) \( \Rightarrow \) \( ((x*2) < 0) \) \( \text{False:} \ T_{Min} \)
- \( ux >= 0 \) \( \text{True:} \ 0 = U_{Min} \)
- \( x & 7 == 7 \) \( \Rightarrow \) \( (x<<30) < 0 \) \( \text{True:} \ x_1 = 1 \)
- \( ux > -1 \) \( \text{False:} \ 0 \)
- \( x > y \) \( \Rightarrow \) \( -x < -y \) \( \text{False:} \ -1, \ T_{Min} \)
- \( x * x >= 0 \) \( \text{False:} \ 65535 \)
  \( (x*x = -131071) \)
- \( x > 0 && y > 0 \) \( \Rightarrow \) \( x + y > 0 \) \( \text{False:} \ T_{Max}, \ T_{Max} \)
- \( x >= 0 \) \( \Rightarrow \) \( -x <= 0 \) \( \text{True:} \ -T_{Max} < 0 \)
- \( x <= 0 \) \( \Rightarrow \) \( -x >= 0 \) \( \text{False:} \ T_{Min} \)