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

Integers
Sep 4, 2001

Topics

- **Numeric Encodings**
  - Unsigned & Two’s complement

- **Programming Implications**
  - C promotion rules

- **Basic operations**
  - Addition, negation, multiplication

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

• Taken from Exam #2, CS 347, Spring ‘97
• 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 \implies ((x*2) < 0)
  • ux >= 0
  • x & 7 == 7 \implies (x<<30) < 0
  • ux > -1
  • x > y \implies -x < -y
  • x * x >= 0
  • x > 0 && y > 0 \implies x + y > 0
  • x >= 0 \implies -x <= 0
  • x <= 0 \implies -x >= 0

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Encoding Integers

Unsigned

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

Two’s Complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x}</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>\texttt{y}</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

Sign Bit

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Encoding Example (Cont.)

\[ x = 15213: 00111011 01101101 \]
\[ y = -15213: 11000100 10010011 \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{Sum} \quad 15213 \quad -15213 \]
**Numeric Ranges**

**Unsigned Values**
- $U_{Min} = 0$
  - 000...0
- $U_{Max} = 2^w - 1$
  - 111...1

**Two’s Complement Values**
- $T_{Min} = -2^{w-1}$
  - 100...0
- $T_{Max} = 2^{w-1} - 1$
  - 011...1

**Other Values**
- Minus 1
  - 111...1

**Values for $W = 16$**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{Max}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{Max}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{Min}$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>$-1$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
# Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

### Observations
- $|T_{Min}| = T_{Max} + 1$
  - Asymmetric range
- $U_{Max} = 2 \times T_{Max} + 1$

### C Programming
- `#include <limits.h>`
  - K&R App. B11
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific
Unsigned & Signed Numeric Values

Example Values

- $W = 4$

Equivalence

- Same encodings for nonnegative values

Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

$\Rightarrow$ Can Invert Mappings

- $U2B(x) = B2U^{-1}(x)$
  - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
  - Bit pattern for two’s comp integer

<table>
<thead>
<tr>
<th>$X$</th>
<th>B2U($X$)</th>
<th>B2T($X$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>
Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

<table>
<thead>
<tr>
<th>short int</th>
<th>x =  15213;</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short int</td>
<td>ux = (unsigned short) x;</td>
</tr>
<tr>
<td>short int</td>
<td>y = −15213;</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>uy = (unsigned short) y;</td>
</tr>
</tbody>
</table>

Resulting Value

- No change in bit representation
- Nonnegative values unchanged
  - ux = 15213
- Negative values change into (large) positive values
  - uy = 50323
Relation Between 2’s Comp. & Unsigned

Two’s Complement  \( x \)  \( \xrightarrow{\text{T2B}} \)  \( X \)  \( \xrightarrow{\text{B2U}} \)  \( u_x \)

Maintain Same Bit Pattern

\[
\begin{align*}
    u_x &= \begin{cases} 
    x & x \geq 0 \\
    x + 2^w & x < 0 
\end{cases} \\
    +2^{w-1} - -2^{w-1} &= 2^w 
\end{align*}
\]
Relation Between Signed & Unsigned

\[ u_y = y + 2 \times 32768 = y + 65536 \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>-15213</th>
<th>50323</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>1</td>
<td>16384</td>
</tr>
<tr>
<td>32768</td>
<td>1</td>
<td>-32768</td>
</tr>
<tr>
<td>Sum</td>
<td>-15213</td>
<td>50323</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned in C

Constants
- By default are considered to be signed integers
- Unsigned if have “U” as suffix
  0U, 4294967259U

Casting
- Explicit casting between signed & unsigned same as U2T and T2U
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;
- Implicit casting also occurs via assignments and procedure calls
  tx = ux;
  uy = ty;
Casting Surprises

Expression Evaluation
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for \( W = 32 \)

<table>
<thead>
<tr>
<th>Constant(_1)</th>
<th>Constant(_2)</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>−2147483648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>−2147483648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−1</td>
<td>−2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned) −1</td>
<td>−2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive
Sign Extension

Task:
• Given $w$-bit signed integer $x$
• Convert it to $w+k$-bit integer with same value

Rule:
• Make $k$ copies of sign bit:
• $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$
Sign Extension Example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<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>ix</td>
<td>15213</td>
<td>00 00 C4 92</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Justification For Sign Extension

Prove Correctness by Induction on $k$

- Induction Step: extending by single bit maintains value

• Key observation: $-2^{w-1} = -2^w + 2^{w-1}$

- Look at weight of upper bits:
  $$X \quad -2^{w-1}x_{w-1}$$
  $$X' \quad -2^w x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1}$$
Why Should I Use Unsigned?

Don’t Use Just Because Number Nonzero
- C compilers on some machines generate less efficient code
  ```c
  unsigned i;
  for (i = 1; i < cnt; i++)
    a[i] += a[i-1];
  ```
- Easy to make mistakes
  ```c
  for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
  ```

Do Use When Performing Modular Arithmetic
- Multiprecision arithmetic
- Other esoteric stuff

Do Use When Need Extra Bit’s Worth of Range
- Working right up to limit of word size
Negating with Complement & Increment

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

Complement

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

\[
\begin{array}{c}
\text{x} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & \sim x & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
-1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

Increment

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

Warning: Be cautious treating \texttt{int}'s as integers

- OK here
### 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>(~x)</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>(~x+1)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011\textsuperscript{1}</td>
</tr>
<tr>
<td>(y)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
<tr>
<td>(~0)</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>(~0+1)</td>
<td>0</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

Standard Addition Function
• Ignores carry output

Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \mod 2^w$$

$$\text{UAdd}_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}$$
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
\[ \begin{align*}
2^{w+1} \\
2^w \\
0
\end{align*} \]

Modular Sum

Overflow
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 \[ \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
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

\[
\text{TAdd}(u, v) = \begin{cases} 
  u + v + 2^{w-1} & \text{if } u + v < TMin_w \\
  u + v & \text{if } TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & \text{if } TMax_w < u + v 
\end{cases}
\]
Visualizing 2’s Comp. Addition

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

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

Task

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

Claim

• Overflow iff either:
  
  \[
  u, v < 0, s \geq 0 \quad \text{(NegOver)}
  \]
  
  \[
  u, v \geq 0, s < 0 \quad \text{(PosOver)}
  \]

  \[
  \text{ovf} = (u<0 == v<0) && (u<0 != s<0);
  \]
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- \( TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \)
  - 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 \( TComp_w(u) = U2T(UComp_w(T2U(u))) \)

\[ TAdd_w(u, TComp_w(u)) = 0 \]

\[ TComp_w(u) = \begin{cases} -u & \text{if } u \neq TMin_w \\ u & \text{if } u = TMin_w \end{cases} \]
Negating with Complement & Increment

In C

\[ \sim{x} + 1 == -x \]

Complement

- Observation: \[ \sim{x} + x == 1111\ldots11_2 == -1 \]

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

Increment

- \[ \sim{x} + x + (-x + 1) == -1 + (-x + 1) \]
- \[ \sim{x} + 1 == -x \]

Warning: Be cautious treating int’s as integers

- OK here: We are using group properties of TAdd and TComp
Multiplication

Computing Exact Product of $w$-bit numbers $x, y$

- Either signed or unsigned

Ranges

- **Unsigned**: $0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- **Two’s complement min**: $x \cdot y \geq (-2^{w-1})^2(2^{w-1} - 1) = -2^{2w-2} + 2^w$
  - Up to $2w-1$ bits
- **Two’s complement max**: $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - 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, and other “advanced” languages
### Unsigned Multiplication in C

**Standard Multiplication Function**
- Ignores high order \( w \) bits

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

Unsigned Multiplication

```c
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

```c
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
- \(up == (\text{unsigned}) p\)
Power-of-2 Multiply with Shift

Operation

• $u << k$ gives $u \cdot 2^k$
• Both signed and unsigned

Examples

• $u << 3 = u \cdot 8$
• $u << 5 - u << 3 = u \cdot 24$
• 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>15213</td>
<td>3B 6D 001111011 01101101</td>
</tr>
<tr>
<td>$x \gg 1$</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>$x \gg 8$</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
2’s Comp Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

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

**Operands:**

- $u$
- $2^k$

**Division:**

- $u / 2^k$

**Result:**

- $\text{RoundDown}(u / 2^k)$

---

### Table:

<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>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>$y &gt;&gt; 1$</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
</tr>
<tr>
<td>$y &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
</tr>
<tr>
<td>$y &gt;&gt; 8$</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
</tr>
</tbody>
</table>

---

class03.ppt
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>( u )</th>
<th>( u ) + ( 2^k-1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2^k + 1</td>
<td>1 ( \cdots ) 0 ( \cdots ) 0 0</td>
<td>0 ( \cdots ) 0 0 1 ( \cdots ) 1 1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
\hline
& k \\
\hline
\text{Divisor:} & \frac{u}{2^k} \\
\hline
\end{array}
\]

\[
\frac{u}{2^k} = 1 \cdots 1 1 1 \cdots \cdot 1 | \cdots | 1 1 \cdot \cdots \cdot 0 0
\]

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

Case 2: Rounding

Dividend:

\[ u \]

\[ +2^k + 1 \]

\[ \overline{+} \]

Divisor:

\[ / 2^k \]

\[ \left\lceil u / 2^k \right\rceil \]

Biasing adds 1 to final result
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.,
  \[
  u > 0 \quad \Rightarrow \quad u + v > v \\
  u > 0, \ v > 0 \quad \Rightarrow \quad u \cdot v > 0
  \]
• These properties are not obeyed by two’s complement arithmetic
  \[
  TMax + 1 = TMin \\
  15213 \times 30426 = -10030 \quad \text{(16-bit words)}
  \]
C Puzzle Answers

• Assume machine with 32 bit word size, two’s complement integers
• $TMin$ makes a good counterexample in many cases

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