CS 33

Data Representation, Part 2
Signed Integers

- Two’s complement
  \[ b_{w-1} = 0 \implies \text{non-negative number} \]
  \[
  \text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i
  \]
  \[ b_{w-1} = 1 \implies \text{negative number} \]
  \[
  \text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i
  \]
Signed Integers

• Negating two’s complement

\[
\text{value} = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i
\]

– how to compute \(-\text{value}\)?

\((\sim\text{value})+1\)
Signed Integers

• Negating two’s complement (continued)

\[ value + (\sim value + 1) \]

\[ = (value + \sim value) + 1 \]

\[ = (2^w - 1) + 1 \]

\[ = 2^w \]

\[ = \begin{array}{cccccccccc}
1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\end{array} \]
Quiz 1

• We have a computer with 4-bit words that uses two’s complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 0111
  b) 1001
  c) 1110
  d) 1111
Signed vs. Unsigned in C

- char, short, int, and long
  - signed integer types
  - right shift (>>) is arithmetic
- unsigned char, unsigned short, unsigned int, unsigned long
  - unsigned integer types
  - right shift (>>) is logical
Numeric Ranges

• Unsigned Values
  – $UMin = 0$
    000...0
  – $UMax = 2^w - 1$
    111...1

• Two’s Complement Values
  – $TMin = -2^{w-1}$
    100...0
  – $TMax = 2^{w-1} - 1$
    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>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</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

<table>
<thead>
<tr>
<th>Tmin</th>
<th>=</th>
<th>Tmax + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric range</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• C Programming

<table>
<thead>
<tr>
<th>include &lt;limits.h&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>declares constants, e.g.,</td>
</tr>
<tr>
<td>ULONG_MAX</td>
</tr>
<tr>
<td>LONG_MAX</td>
</tr>
<tr>
<td>LONG_MIN</td>
</tr>
<tr>
<td>values platform-specific</td>
</tr>
</tbody>
</table>
Quiz 2

• What is –TMin (assuming two’s complement signed integers)?
  a) TMin
  b) Tmax
  c) 0
  d) 1
4-Bit Computer Arithmetic
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
    ```c
    int tx, ty;
    unsigned ux, uy; // “unsigned” means “unsigned int”
    tx = (int) ux;
    uy = (unsigned int) ty;
    ```

  – implicit casting also occurs via assignments and procedure calls
    ```c
    tx = ux;
    uy = ty;
    ```
Casting Surprises

- Expression evaluation
  - if there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - including comparison operations `<`, `>`, `==`, `<=`, `>=`
  - examples for $W = 32$: \( T\text{MIN} = -2,147,483,648 \), \( T\text{MAX} = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>(\text{Constant}_1)</th>
<th>(\text{Constant}_2)</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int)2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Quiz 3

What is the value of

\[(\text{long})\text{ULONG}_\text{MAX} - (\text{unsigned long})-1\]

???

a) -1
b) 0
c) 1
d) ULONG_MAX
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

```
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 3B 6D</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
Does it Work?

\[ \text{val}_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ \text{val}_{w+1} = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ \text{val}_{w+2} = -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]

\[ = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]
Unsigned Multiplication

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard multiplication function**
  - ignores high order $w$ bits
- **Implements modular arithmetic**
  \[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
Signed Multiplication

Operands: \( w \) bits

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

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

- Standard multiplication function
  - ignores high order \( w \) bits
  - some of which are different from those of unsigned multiplication
  - lower bits are the same
**Power-of-2 Multiply with Shift**

- **Operation**
  - \( u << k \) gives \( u \times 2^k \)
  - both signed and unsigned

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u &lt;&lt; 3 )</td>
<td>( u \times 8 )</td>
</tr>
<tr>
<td>( u &lt;&lt; 5 - u &lt;&lt; 3 )</td>
<td>( u \times 24 )</td>
</tr>
</tbody>
</table>

- most machines shift and add faster than multiply
  - compiler generates 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

### Division

<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</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>59.42578125</td>
<td>59</td>
<td>00 3B</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of signed by power of 2
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - uses arithmetic shift
  - rounds wrong direction when \( x < 0 \)

<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>111000100 10010011</td>
</tr>
<tr>
<td>( y \gg 1 )</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>111111100 01001001</td>
</tr>
<tr>
<td>( y \gg 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 \( \lceil x / 2^k \rceil \) (round toward 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

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( u )</th>
<th>( +2^k - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1\cdots00\cdots00 )</td>
<td>( 0\cdots001\cdots11 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( 2^k )</th>
<th>( u / 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0\cdots010\cdots00 )</td>
<td>( 1\cdots11\cdots11 )</td>
<td></td>
</tr>
</tbody>
</table>

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

Case 2: rounding

dividend: \[ x + 2^k - 1 \]

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

\[ \begin{array}{c}
1 \cdots \cdots \cdots \hspace{1cm} 1 \hspace{1cm} 1 \hspace{1cm} \cdots \hspace{1cm} \cdots \\
\end{array} \]

divisor: \[ \frac{x}{2^k} \]

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

\[ \begin{array}{c}
1 \cdots \cdots \cdots \hspace{1cm} 1 \hspace{1cm} 1 \hspace{1cm} 1 \hspace{1cm} \cdots \\
\end{array} \]

Biasing adds 1 to final result
Why Should I Use Unsigned?

• *Don’t* use just because number nonnegative
  – easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  – can be very subtle
    ```c
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
        ...
    ```
• *Do* use when performing modular arithmetic
  – multiprecision arithmetic
• *Do* use when using bits to represent sets
  – logical right shift, no sign extension