Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective.” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
There’s only one zero!

Two’s complement is used on pretty much all of today’s computers to represent signed integers.

Note that the high-order (most-significant) bit represents $-2^{w-1}$. All the other bits represent positive numbers.
To negate a two’s-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.
Signed Integers

- Negating two’s complement (continued)

\[
\text{value} + (\sim\text{value} + 1) \\
= (\text{value} + \sim\text{value}) + 1 \\
= (2^{w-1}) + 1 \\
= 2^w \\
= \underbrace{1\ 0\ 0\ 0\ \ldots\ 0\ 0\ 0}_W
\]

If we add to the two’s complement representation of a w-bit number the result of adding one to its bitwise complement, we get a \(w+1\)-bit number whose low-order \(w\) bits are zeroes and whose high-order bit is one. However, since we’re constrained to only \(w\) bits, the result is a \(w\)-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.
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

Why the signed integer types use the arithmetic right shift will be clear by the end of this lecture.
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>UMax</td>
<td>65535</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

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

- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform-specific

Supplied by CMU.
Quiz 2

- What is \(-\text{TMin}\) (assuming two’s complement signed integers)?
  a) TMin
  b) Tmax
  c) 0
  d) 1
Unsigned computer arithmetic is performed modulo 2 to the power of the computer’s word size. The outer ring of the figure demonstrates arithmetic modulo $2^4$. To see the result, for example, of adding 3 to 2, start at 2 and go around the ring three units in the clockwise direction. If we add 5 to 14, we start at 14 and move 5 units clockwise, to 3. Similarly, to subtract 3 from 1, we start at one and move three units counterclockwise to 14.

What about two's-complement computer arithmetic? We know that the values encoded in a 4-bit computer word range from -8 to 7. How do we arrange them in the ring? As shown in the second ring, it makes sense for the non-negative numbers to be in the same positions as the corresponding unsigned values. It clearly makes sense for the integer coming just before 0 to be -1, the integer just before -1 to be -2, etc. Thus, since we have a ring, the integer following 7 is -8. Now we can see how arithmetic works for two’s-complement numbers. Adding 3 to 2 works just as it does for unsigned numbers. Subtracting 3 from 1 results in -2. But adding 3 to 6 results in -7; and adding 5 to -2 results in 3.

The innermost ring shows the bit encodings for the unsigned and two’s-complement values. The point of all this is that, with only one implementation of arithmetic, we can handle both unsigned and two’s-complement values. Thus adding unsigned 5 and 9 is equivalent to adding two’s-complement 5 and -7. The result will 1110, which, if interpreted as an unsigned value is 14, but if interpreted as a two’s-complement value is -2.
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
    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
    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 $W = 32$: $TMIN = -2,147,483,648$, $TMAX = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant₁</th>
<th>Constant₂</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int)2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Quiz 3

What is the value of

\[(\text{long})\, \text{ULONG_MAX} - (\text{unsigned long})\, -1\]

???

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:
  - $X' = x_{w-1},\ldots,x_{w-1},x_{w-1},x_{w-2},\ldots,x_0$

Supplied by CMU.
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 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
Sign extension clearly works for positive and zero values (where the sign bit is zero). But does it work for negative values? The first line of the slide shows the computation of the value of a $w$-bit item with a sign bit of one (i.e., it’s negative). The next two lines show what happens if we extend this to a $w+1$-bit item, extending the sign bit. What had been the sign bit becomes one of the value bits, and its contribution to the value is now positive rather than negative. But this is compensated by the new sign bit, whose contribution is a negative value, twice as large as the original sign bit. Thus the net effect is for there to be no change in the value.

We do this again, extending to a $w+2$-bit item, and again, the resulting value is the same as what we started with.
**Unsigned Multiplication**

Operands: $w$ bits

```
  u
* v
```

True Product: $2^w$ bits

```
  u * v
```

Discard $w$ bits: $w$ bits

**UMult_w(u, v)**

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

---

Supplied by CMU.
Signed Multiplication

Operands: \( w \) bits

\[
\begin{array}{c}
\text{u} \\
\times \\
\text{v}
\end{array}
\]

True Product: \( 2^w \) bits

\[
\text{u} \times \text{v} \\
\text{TMult}_w(u, v)
\]

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

Supplied by CMU.
Power-of-2 Multiply with Shift

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

  \[
  \begin{array}{c}
  \text{operands: } w \text{ bits} \\
  \hline
  u & \cdots & \cdots \\
  \hline
  \text{true product: } w+k \text{ bits} \\
  \hline
  u \times 2^k & \cdots & \cdots & 0 & 0 & \cdots & 0 & 0 \\
  \hline
  \text{discard } k \text{ bits: } w \text{ bits} \\
  \hline
  \text{UMult}(u, 2^k) & \cdots & 0 & 0 & \cdots & 0 & 0 \\
  \hline
  \text{TMult}(u, 2^k) & \cdots & 0 & 0 & \cdots & 0 & 0 \\
  \end{array}
  \]

- Examples
  - \( u \ll 3 = u \times 8 \)
  - \( u \ll 5 - u \ll 3 = u \times 24 \)

  - most machines shift and add faster than multiply

 /compiler generates this code automatically

Supplied by CMU.
Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned by power of 2
  - \( u \gg k \) gives \( \left\lfloor \frac{u}{2^k} \right\rfloor \)
  - 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</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03</td>
<td>00000001 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

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

\[
x
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - want \( \lceil x / 2^k \rceil \) (round toward 0)
  - compute as \( \lceil (x+2^k-1) / 2^k \rceil \)
    » in C: \((x + (1<<k)-1) >> k\)
    » biases dividend toward 0

Case 1: no rounding

\[
\begin{array}{c}
\text{dividend:} \\
\hline
u & 11 \cdots 0 \cdots 0 \\
+2^k-1 & 01 \cdots 011 \cdots 11 \\
\end{array}
\]

\[
\begin{array}{c}
\text{divisor:} \\
\hline
\div & 11 \cdots 1 \cdots 1 \\
/ 2^k & 01 \cdots 010 \cdots 00 \\
\lceil u / 2^k \rceil & 11 \cdots 110 \cdots 11 \\
\end{array}
\]

**Biasing has no effect**

Supplied by CMU.
Correct Power-of-2 Divide (Cont.)

Case 2: rounding

\[
\begin{array}{c}
\text{dividend:} \\
\quad x \\
\quad + 2^k - 1 \\
\hline
\text{divisor:} \\
\quad / 2^k \\
\quad \lfloor x / 2^k \rfloor
\end{array}
\]

- \quad \begin{array}{c}
\quad 1 \quad \ldots \quad \ldots \\
+ \quad 0 \quad \ldots \quad 001 \quad \ldots \quad 11
\end{array}

\quad \begin{array}{c}
\quad 1 \quad \ldots \quad \ldots \\
\quad \ldots
\end{array}

\text{binary point}

\quad \text{incremented by 1}

\quad \text{incremented by 1}

\textbf{Biasing adds 1 to final result}

Supplied by CMU.
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

Supplied by CMU.

Note that “sizeof” returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)