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.
In the diagram, x is an int occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).
If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where \( w \) is the number of bits in the word.
Signed Integers

- **Sign-magnitude**

  \[
  b_{w-1} \ b_{w-2} \ b_{w-3} \ldots \ b_2 \ b_1 \ b_0
  \]

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

- two representations of zero
  - computer must have two sets of instructions
    - one for signed arithmetic, one for unsigned arithmetic

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the “sign bit”: 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.
In ones' complement, a number is positive if its leftmost bit is zero negative otherwise. We negate a number by complementing all its bits. Thus if the leftmost bit is zero, a one in position \( i \) of the remaining bits contributes a value of \( 2^i \) and a zero contributes nothing. But if the leftmost bit is one, a zero in position \( i \) contributes a value of \(-2^i\) and a one contributes nothing.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.
Signed Integers

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

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

  - one zero!

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.
### Example

- **w = 4**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>7</td>
</tr>
<tr>
<td>1010</td>
<td>6</td>
</tr>
<tr>
<td>1011</td>
<td>5</td>
</tr>
<tr>
<td>1100</td>
<td>4</td>
</tr>
<tr>
<td>1101</td>
<td>3</td>
</tr>
<tr>
<td>1110</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>
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)

\[
\text{value} + (\sim\text{value} + 1) \\
= (\text{value} + \sim\text{value}) + 1 \\
= (2^w - 1) + 1 \\
= 2^w \\
= \begin{array}{cccccc}
1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0
\end{array}
\]

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 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 FF FF</td>
<td>11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 00 00</td>
<td>00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU.
## Values for Different Word Sizes

<table>
<thead>
<tr>
<th></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**
  - \(|T_{\text{Min}}| = T_{\text{Max}} + 1\)
  - Asymmetric range
  - \(U_{\text{Max}} = 2 \times T_{\text{Max}} + 1\)

- **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 \(-T_{\text{Min}}\) (assuming two’s complement signed integers)?
  a) \(T_{\text{Min}}\)
  b) \(T_{\text{Max}}\)
  c) 0
  d) 1
Unsigned computer arithmetic is performed modulo $2^n$ 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;
    ```

Note that the kind of casting done here is what we called "intimidation" in the previous lecture: no actual conversion takes place, but the value is reinterpreted according to the cast.
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 \( W = 32 \):
    \[ \text{TMIN} = -2,147,483,648, \quad \text{TMAX} = 2,147,483,647 \]

<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><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>

Supplied by CMU.
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

```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>00110101 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

\[
\begin{array}{ll}
\hline
u & \cdots \\
\ast & \cdots \\
v & \cdots \\
\hline
\end{array}
\]

True Product: \( 2w \) bits

\[
\begin{array}{ll}
\hline
u \ast v & \cdots \\
UMult_w(u, v) & \cdots \\
\hline
\end{array}
\]

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

- **Standard multiplication function**
  - ignores high order \( w \) bits

- **Implements modular arithmetic**

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

---

Supplied by CMU.
How is it that the "true product" is different from that of unsigned multiplication? Consider what the true product should be is the multiplier is -1 and the multiplicand is 1. Thus the multiplier is a w-bit word of all ones and the multiplicand is a w-bit word of all zeroes except for the least-significant bit, which is 1. The high-order w bits of the true product should be all ones (since it's negative), but with unsigned multiplication they'd be all zeroes. However, since we're ignoring the high-order w bits, this doesn't matter.
Power-of-2 Multiply with Shift

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

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

- 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
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>Operands</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u / 2^k$</td>
<td>$u / 2^k$</td>
<td>$u / 2^k$</td>
<td>$u / 2^k$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result: $\lfloor u / 2^k \rfloor$</th>
<th>$\lfloor u / 2^k \rfloor$</th>
<th>$\lfloor u / 2^k \rfloor$</th>
<th>$\lfloor u / 2^k \rfloor$</th>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
</tr>
<tr>
<td>$2^k$</td>
<td>$2^k$</td>
<td>$2^k$</td>
<td>$2^k$</td>
<td>$2^k$</td>
</tr>
</tbody>
</table>

Supplied by CMU.
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>Operands:</th>
<th>( x \gg k )</th>
<th>( 2^k )</th>
<th>( x / 2^k )</th>
<th>Result: RoundDown( (x / 2^k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</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>11100100 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>

Supplied by CMU.
Correct Power-of-2 Divide

- Quotient of negative number by power of 2
  - want \( \lfloor x / 2^k \rfloor \) (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: (u)</th>
<th>(2^k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 ... 0 ... 0</td>
<td>11 ... 1 ... 1</td>
</tr>
<tr>
<td>+2^k - 1</td>
<td></td>
</tr>
<tr>
<td>01 ... 1 ... 1</td>
<td></td>
</tr>
</tbody>
</table>

**Divisor:**

<table>
<thead>
<tr>
<th>(u/2^k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 ... 1 ... 1</td>
</tr>
</tbody>
</table>

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

Case 2: rounding

\[
\begin{array}{c}
\text{dividend:} \\
\hline
k & \cdot & \cdot & \cdot \\
\hline
+2^k-1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{divisor:} \\
\hline
\hline
2^k & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
x / 2^k & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\end{array}
\]

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.)