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.
### 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>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 1

- 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 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 $W = 32$: $T_{MIN} = -2,147,483,648$, $T_{MAX} = 2,147,483,647$

<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>==</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>signed</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 2

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:
    \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \]
### 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>01110101 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

---

Supplied by CMU.
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

\[
\begin{array}{c}
\begin{array}{c}
\hline
\hline
\hline
\text{true} & \text{product} & \text{2 bits} \\
\hline
\hline
\hline
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\hline
\hline
\hline
\text{true} & \text{product} & \text{2 bits} \\
\hline
\hline
\hline
\end{array}
\end{array}
\]

Discard $w$ bits: $w$ bits

\[
\begin{array}{c}
\begin{array}{c}
\hline
\hline
\hline
\text{true} & \text{product} & \text{2 bits} \\
\hline
\hline
\hline
\end{array}
\end{array}
\]

- **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
  
  $u$ $\cdot \cdot \cdot$

  $\times$

  $v$ $\cdot \cdot \cdot$

- **True Product:** $2w$ bits
  
  $u \cdot v$ $\cdot \cdot \cdot$
  $\cdot \cdot \cdot$

- **Discard $w$ bits:** $w$ bits
  
  $\text{TMult}_w(u, v)$ $\cdot \cdot \cdot$

- **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

operands: \( w \) bits

\[
\begin{array}{c}
\text{operands: } w \text{ bits} \\
\hline
u & \cdots & k \\
\hline
* 2^k & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\
\hline
\text{true product: } w+k \text{ bits} & u \times 2^k & \cdots & \cdots & 0 & \cdots & 0 & 0 \\
\hline
\text{discard } k \text{ bits: } w \text{ bits} & \Umult_u(u, 2^k) & \cdots & \cdots & 0 & \cdots & 0 & 0 \\
\hline
\text{UMult}_u(u, 2^k) & \cdots & \cdots & 0 & \cdots & 0 & 0 \\
\hline
\text{TMult}_u(u, 2^k) & \cdots & \cdots & 0 & \cdots & 0 & 0 \\
\end{array}
\]

- 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 \( \lfloor u / 2^k \rfloor \)
  - 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 \gg 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x \gg 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x \gg 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

Supplied by CMU.
**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 \( x < 0 \)

<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</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>E2</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>FC</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59.4257813</td>
<td>FF</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 \( \lceil x / 2^k \rceil \) (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

\[
\begin{array}{c}
\text{dividend:} \\
1 \ldots 0 \ldots 0 \\
\hline
+2^{k-1} \\
0 \ldots 1 \ldots 1 \\
\hline
1 \ldots 1 \ldots 1 \\
\text{divisor:} \\
2^k \\
0 \ldots 0 \ldots 0 \\
\hline
u / 2^k \\
1 \ldots 1 \ldots 1 \\
\end{array}
\]

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

Case 2: rounding

\[
\begin{align*}
\text{dividend:} & \quad x_{1\ldots k} + 2^k - 1 \\
\text{divisor:} & \quad \lfloor x / 2^k \rfloor
\end{align*}
\]

\text{incremented by 1}

Binary point

\text{incremented by 1}

\text{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.)
Byte-Oriented Memory Organization

- Programs refer to data by address
  - conceptually, envision it as a very large array of bytes
    » in reality, it's not, but can think of it that way
  - an address is like an index into that array
    » and, a pointer variable stores an address

- Note: system provides private address spaces to each "process"
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others
Machine Words

• Any given computer has a “word size”
  – nominal size of integer-valued data
    » and of addresses
  – until recently, most machines used 32 bits (4 bytes)
    as word size
    » limits addresses to 4GB ($2^{32}$ bytes)
    » becomes too small for memory-intensive applications
      • leading to emergence of computers with 64-bit word
        size
  – machines still support multiple data formats
    » fractions or multiples of word size
    » always integral number of bytes
### Word-Oriented Memory Organization

- Addresses specify byte locations
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0004</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0012</td>
<td>0003</td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU.
Byte Ordering

- Four-byte integer
  - 0x76543210
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>32</td>
<td>54</td>
<td>76</td>
</tr>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Little-endian

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>54</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Big-endian

Read “Gulliver’s Travels” for an explanation of the egg.
Here we have a four-byte integer one. In the big-endian representation, the address of the integer is the address of the byte containing its most-significant bits (the big end), while in the little-endian representation, the address of the integer is the address of the byte containing its least-significant bits (the little end). Suppose we pass a pointer to this integer to some procedure. However, in a type-mismatch, the procedure assumes that what is passed it is a two-byte integer. On a big-endian system, it would think it was passed a zero, but on a little-endian system, it would think it was passed a one.

This is not an argument in favor of either approach, but simply an observation that behaviors could be different.
int main() {
    long x=1;
    proc((int *)&x);
    return 0;
}

void proc(int *arg) {
    printf("%d\n", *arg);
}

What value is printed on a big-endian 64-bit computer?
   a) 0
   b) 1
   c) $2^{32}$
   d) $2^{32}-1$