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.
Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    » positional notation
    » symbol for 0
  - adopted and modified somewhat later by Arabs
    » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    » (try doing long division with Roman numerals!)
Base 2 is known as “binary” notation.
Base 8 is known as “octal” notation.
Base 10 is known as “decimal” notation.
Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.

Which Base?

• 1999
  – base 10
    » 9·10⁶+9·10¹+9·10²+1·10¹
  – base 2
    » 11111001111
      • 1·2⁸+1·2⁴+1·2²+1·2³+0·2⁴+0+2⁵+1·2⁴+1·2⁸+1·2⁴+1·2¹⁰
  – base 8
    » 3717
      • 7·8⁴+1·8¹+7·8²+3·8³
      » why are we interested?
  – base 16
    » 7CF
      • 15·16⁰+12·16¹+7·16²
      » why are we interested?
Note that a byte consists of two hexadecimal digits, which are sometimes known as “nibbles”. A 32-bit computer word would then have eight nibbles; a 64-bit computer word would have sixteen nibbles.
This routine prints the base base representation of num. (Note that the “…” is not heretofore unexplained C syntax, but is shorthand for “fill this in to the extent needed.”)
"bc" (it stands for basic calculator, or perhaps better calculator) is a standard Unix command that handles arbitrary-precision arithmetic. Among its features is the ability to specify which base to use for input and output of numbers. The default base for both input and output is ten. Setting `obase` to 16 sets the base for output to 16. Similarly, one can change the base for input numbers by setting `ibase`.

```
$ bc
obase=16
1999
7CF
$
```

Or ...

```
$ bc
obase=16
1999
7CF
$
```
Quiz 1

• What’s the decimal (base 10) equivalent of $23_{16}$?
  a) 19
  b) 33
  c) 35
  d) 37
Encoding Byte Values

- **Byte = 8 bits**
  - binary 00000000₂ to 11111111₂
  - decimal: 0₁₀ to 255₁₀
  - hexadecimal 00₁₆ to FF₁₆
    - base 16 number representation
    - use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
Boolean Algebra

- Developed by George Boole in 19th Century
  - algebraic representation of logic
    - encode “true” as 1 and “false” as 0

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU.
General Boolean Algebras

- **Operate on bit vectors**
  - operations applied bitwise

  \[
  \begin{array}{ccc}
  01101001 & 01101001 & 01101001 \\
  \& 01010101 & \| 01010101 & ^ 01010101 & \sim 01010101 \\
  01000001 & 01111101 & 00111100 & 10101010
  \end{array}
  \]

- **All of the properties of boolean algebra apply**

Supplied by CMU.
Example: Representing & Manipulating Sets

- Representation
  - width-w bit vector represents subsets of \{0, ..., w-1\}
  - \(a_j = 1\) iff \(j \in A\)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>{0, 3, 5, 6}</td>
<td>76543210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
<td>76543210</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; intersection</td>
<td>01000001</td>
<td>{0, 6}</td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>01111101</td>
</tr>
<tr>
<td>^ symmetric difference</td>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>~ complement</td>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - apply to any “integral” data type
    » long, int, short, char
  - view arguments as bit vectors
  - arguments applied bit-wise
- Examples (char datatype)
  \[
  \begin{align*}
  \neg 0x41 & \rightarrow 0xBE \\
  \neg 01000001_2 & \rightarrow 10111110_2 \\
  \neg 0x00 & \rightarrow 0xFF \\
  \neg 00000000_2 & \rightarrow 11111111_2 \\
  0x69 \& 0x55 & \rightarrow 0x41 \\
  01101001_2 \& 01010100_2 & \rightarrow 01000001_2 \\
  0x69 \| 0x55 & \rightarrow 0x7D \\
  01101001_2 \| 01010100_2 & \rightarrow 01111110_2
  \end{align*}
  \]

Supplied by CMU.
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    » view 0 as “false”
    » anything nonzero as “true”
    » always return 0 or 1
    » early termination/short-circuited execution

- Examples (char datatype)
  !0x41 → 0x00
  !0x00 → 0x01
  !!0x41 → 0x01

  0x69 && 0x55 → 0x01
  0x69 || 0x55 → 0x01

  p && *p (avoids null pointer access)

Supplied by CMU.
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  - && true && false "false"
- Watch out for && vs. & (and || vs. |)...
- One of the more common oopsies in C programming

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x42 &amp;&amp; 0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>!0x00</td>
<td>0x01</td>
</tr>
<tr>
<td>!0x41</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69 &amp;&amp; 0x55</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69</td>
<td></td>
</tr>
<tr>
<td>p &amp;&amp; *p</td>
<td>(avoids null pointer access)</td>
</tr>
</tbody>
</table>

Supplied by CMU.
The distinction between logical and arithmetic shifts should be clear by the end of this lecture.

Supplied by CMU.
Signed Integers

• **Sign-magnitude**

<table>
<thead>
<tr>
<th>$b_{w-1}$</th>
<th>$b_{w-2}$</th>
<th>$b_{w-3}$</th>
<th>...</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

magnitude

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

• two representations of zero!
Signed Integers

• Ones’ complement
  – negate a number by forming its bitwise complement
    » e.g., \((-1)_{11010111} = 10010100\)

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

\[
= \begin{cases} 
\sum_{i=0}^{w-2} b_i \cdot 2^i & \text{if } b_{w-1} = 0 \\
\sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i & \text{if } b_{w-1} = 1 
\end{cases}
\]

Note that the most-significant bit serves as the sign bit.
Note there’s only one zero!

Two’s complement is used on pretty much all of today’s computers to represent signed integers.
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 \(-\text{value}\)?
  \((-\text{value})+1\)
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.

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

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 2

- 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
Numeric Ranges

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

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

- **Other Values**
  - Minus 1

<table>
<thead>
<tr>
<th>Values for $W = 16$</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 1111111</td>
</tr>
<tr>
<td>$T_{Max}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 1111111</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>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 3

- What is $-\text{TMin}$ (assuming two's complement signed integers)?
  a) $\text{TMin}$
  b) $\text{TMax}$
  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 int 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, 
    \textit{signed values implicitly cast to unsigned}
  - including comparison operations $<$, $>$, $==$，$<=$, $>$=
  - examples for $W = 32$: $\text{TMIN} = \text{-2,147,483,648}$, $\text{TMAX} = \text{2,147,483,647}$

<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>$&lt;$</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>$&gt;$</td>
<td>unsigned</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>$&lt;$</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>$&gt;$</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>$&gt;$</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td></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>

Supplied by CMU.
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:
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \)
Sign Extension Example

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 0011011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D 00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 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.
Power-of-2 Multiply with Shift

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

<table>
<thead>
<tr>
<th>Operands: w bits</th>
<th>( u )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \times 2^k )</td>
<td>( k )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True product: w+k bits</th>
<th>( u \times 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{discard } k \text{ bits: w bits} )</td>
<td>( \text{UMult}_u(u, 2^k) )</td>
</tr>
</tbody>
</table>

- 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

Operands:

$$
\begin{array}{c}
\text{u} \\
\downarrow \quad 2^k \\
\text{result:} \\
\end{array} \quad \begin{array}{c}
\lfloor u / 2^k \rfloor \\
\end{array}
$$

Division:

\begin{array}{c}
\text{u} \\
\downarrow \quad 2^k \\
\text{division:} \\
\end{array}

\begin{array}{c}
\lfloor u / 2^k \rfloor \\
\end{array}

\begin{array}{c}
\text{binary point} \\
\end{array}

\begin{array}{c|c|c|c|c}
\text{Division} & \text{Computed} & \text{Hex} & \text{Binary} \\
\hline
x & 15213 & 15213 & 3B 6D & 00111011 01101101 \\
x \gg 1 & 7606.5 & 7606 & 1D B6 & 00011101 10110110 \\
x \gg 4 & 950.8125 & 950 & 03 B6 & 00000011 10110110 \\
x \gg 8 & 59.4257813 & 59 & 00 3B & 00000000 00111011 \\
\end{array}

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>$x$</th>
<th>$\frac{x}{2^k}$</th>
<th>Division:</th>
<th>$x / 2^k$</th>
<th>Result:</th>
<th>RoundDown($x / 2^k$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>111</td>
<td>0000010000</td>
<td>$2^k$</td>
<td>01100000</td>
<td>$x / 2^k$</td>
<td>11100000001</td>
</tr>
</tbody>
</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>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111000 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 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 \( \lfloor x / 2^k \rfloor \) (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>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \ldots 0 \ldots 0</td>
<td>1 \ldots 0 \ldots 1</td>
</tr>
<tr>
<td>( +2^k - 1 )</td>
<td>( 0 \ldots 0 \ldots 1 \ldots 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>divisor: ( / 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \ldots 0 \ldots 0 \ldots 0</td>
</tr>
</tbody>
</table>

**Biasing has no effect**

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

Case 2: rounding

\[
\begin{array}{c}
\text{dividend:} \\
\hline
\text{divisor:}
\end{array}
\]

\[
\begin{array}{c}
x \\
+ 2^k - 1
\end{array}
\]

\[
\begin{array}{c}
[ x / 2^k ] \quad 2^k
\end{array}
\]

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

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

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

\[
\begin{array}{c}
\text{binary point}
\end{array}
\]

\[
\begin{array}{c}
\text{incremented by 1}
\end{array}
\]

\[
\begin{array}{c}
\text{incremented by 1}
\end{array}
\]

**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

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)
    » become 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

Supplied by CMU.
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)
Byte Ordering

- Four-byte integer
  - 0x7654321

- 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>01</td>
<td>23</td>
<td>45</td>
<td>67</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>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

Big-endian
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.
Quiz 4

```c
int main() {
    long x=1;
    proc(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) $2^{32} - 1$