Data Representation: Bits, Bytes, and Integers
CSci 2021 - Machine Architecture and Organization
Professor Pen-Chung Yew

With slides from Randy Bryant and Dave O'Hallaron

Bits, Bytes, and Integers
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
  - Representations in memory, pointers, strings

Encoding Byte Values
- Byte = 8 bits
  - Binary: 00000000 to 11111111
  - Decimal: 0 to 255
  - Hexadecimal: 0x0 to FF
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37B in C as
      - 0xFA1D37B
      - 0xFa1D37B
      - 0xFa1D37B

Binary Representations
- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions) and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements, e.g. switches, transistors, ...
  - Reliably transmitted on noisy and inaccurate wires

Base-Z number representation
- Decimal to Binary
  - Represent 152\textsubscript{10} as 10011100\textsubscript{2}
  - Represent 0.2\textsubscript{10} as 0.0011\textsubscript{2}
  - Represent 1.5\textsubscript{10} X 2\textsuperscript{2} as 1.0010001\textsubscript{2} X 2\textsuperscript{7}
- Binary to Decimal
  - Represent 10110\textsubscript{2} as 22\textsubscript{10}
  - Represent 101.101\textsubscript{2} as 5.625\textsubscript{10}
  - Represent 1.1011\textsubscript{2} X 2\textsuperscript{3} as 1.35\textsubscript{10}

Example Data Representations
- C Data Type | Typical 32-bit | Typical 64-bit | x86-64
- char | 1 | 1 | 1
- short | 2 | 2 | 2
- int | 4 | 4 | 4
- long | 4 | 8 | 8
- float | 4 | 4 | 4
- double | 8 | 8 | 8
- long double | - | - | 10/16
- pointer | 4 | 8 | 8
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Booleans

- Developed by George Boole in 19th Century
- Algebraic representation of logic
  - Encode "True" as 1 and "False" as 0
- Logical operators
  - And
  - Or
  - Not
- Exclusive-Or (Xor)

Boolean Algebra

- Operate on Bit Vectors
  - Operations applied bitwise

Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \(a_j = 1 \text{ if } j \in A\)
- Operations
  - & Intersection
  - | Union
  - ^ Symmetric difference
  - ~ Complement

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

Bit-Level Operations in C

- Operations &, |, ^ Available in C
  - Apply to any "integral" data type
  - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bitwise
- Examples (char data type – 1 byte)

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
- Examples (char data type)
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - & & || | are logical operators in C
- Example
  - 10 x 10
  - 0x69 & 0x55 = 0x01
  - 0x69 || 0x55 = 0x01

Watch out for & & vs. & (and || vs. |)... one of the more common oopsies in C programming

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
  - Representations in memory, pointers, strings
- Summary

Encoding Example (Binary to Unsigned –B2U )

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum: 15213 -15213

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: x >> y
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
    - Logical shift
      - Fill with 0's on left
      - Arithmetic shift
        - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or 2 word size

Encoding Unsigned and Signed Integers

<table>
<thead>
<tr>
<th>Argument</th>
<th>01000101</th>
<th>00111100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00011000</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>00011000</td>
<td>00011000</td>
</tr>
</tbody>
</table>

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

In C, data type short is 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = -15213</td>
<td>00111101 11101101</td>
<td>(1's)</td>
</tr>
<tr>
<td>y = 15213</td>
<td>00111011 11101011</td>
<td>(1's)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Weight Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^2 = 4</td>
</tr>
<tr>
<td>2^3 = 8</td>
</tr>
<tr>
<td>2^4 = 16</td>
</tr>
<tr>
<td>2^5 = 32</td>
</tr>
<tr>
<td>2^6 = 64</td>
</tr>
<tr>
<td>2^7 = 128</td>
</tr>
<tr>
<td>2^8 = 256</td>
</tr>
<tr>
<td>2^9 = 512</td>
</tr>
<tr>
<td>2^10 = 1024</td>
</tr>
</tbody>
</table>

| 2^11 = 2147483648   |
| 2^12 = 2147483648   |
| 2^13 = 2147483648   |
| 2^14 = 2147483648   |
| 2^15 = 2147483648   |
| 2^16 = 2147483648   |
| 2^17 = 2147483648   |
| 2^18 = 2147483648   |
| 2^19 = 2147483648   |

For 2's complement, most significant bit indicates sign
- 0 for nonnegative
- 1 for negative

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

- In C, data type short is 2 bytes long
Converting Unsigned to Binary (U2B)
Dividing the number repeatedly by 2 until the number becomes 0

49?

<table>
<thead>
<tr>
<th>Divide by</th>
<th>Number</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1 × 2^5 + 1 × 2^4 + 0 × 2^3 + 0 × 2^2 + 0 × 2^1 + 1 × 2^0 = 49

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>1,073,741,823</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>TMax</td>
<td>127</td>
<td>32,767</td>
<td>5,368,709,127</td>
<td>8,589,934,468,055,881,337</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-5,368,709,127</td>
<td>-8,589,934,468,055,881,337</td>
</tr>
</tbody>
</table>

Observations
- |TMin| = |TMax| + 1
- Asymmetric range
- UMax = 2^W - 1

C Programming
- #include <limits.h>
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN

Values platform specific
- Need to know to avoid overflow

Today: Bits, Bytes, and Integers
- Representing information as bits
- Bit-level manipulations

Integers
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Numeric Ranges

<table>
<thead>
<tr>
<th>Unsigned Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMin = 0</td>
</tr>
<tr>
<td>UMax = 2^W - 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two’s Complement Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tmin = -2^W - 1</td>
</tr>
<tr>
<td>Tmax = 2^W - 1</td>
</tr>
<tr>
<td>Other Values</td>
</tr>
<tr>
<td>Minus 1 = -2^W - 1</td>
</tr>
</tbody>
</table>

Values for W = 16

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>11111111110</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>0000000000 0000000000 0000000000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 00</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

Unsigned & Signed Integer Values

<table>
<thead>
<tr>
<th>x</th>
<th>B2U(x)</th>
<th>B2T(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

Unsigned
- Equivalence
  - Same encodings for nonnegative integer values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- Can invert mappings
  - U2B(x) = B2U(-x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T(-x)
    - Bit pattern for two’s comp integer

Mapping Between Signed & Unsigned

Two's Complement

<table>
<thead>
<tr>
<th>Maintain Same Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

Unsigned
<table>
<thead>
<tr>
<th>Maintain Same Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
</tbody>
</table>
Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Relation between Signed & Unsigned

Two's Complement

\[ x \rightarrow \text{Signed} \rightarrow \text{Two's Complement} \rightarrow \text{Unsigned} \]

Maintain Same Bit Pattern

\[ w = \begin{cases} x & x > 0 \\ x + 2^w & x \leq 0 \end{cases} \]

Large negative weight becomes Large positive weight

Conversion Visualized

<table>
<thead>
<tr>
<th>2's Comp. → Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering Inversion</td>
</tr>
<tr>
<td>Negative → Big Positive</td>
</tr>
<tr>
<td>Warning: Can cause a lot of confusion and bugs in C!!</td>
</tr>
</tbody>
</table>

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
  - 0U, 4294967259U

- **Casting (i.e. conversion)**
  - **Explicit casting** between signed & unsigned same as U2T and T2U
    
    ```c
    int tx, ty;
    unsigned uw, uy;
    tx = (int) uw;
    uy = (unsigned) ty;
    ```
  - **Implicit casting** also occurs via assignments and procedure calls
    
    ```c
    tx = uw;
    uy = ty;
    ```

Summary

**Casting Signed ↔ Unsigned: Basic Rules**

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^n
- Expression containing signed and unsigned int
  - _int is cast to unsigned!!_
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

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

Sign Extension Example

```
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>$0011101101101101$</td>
</tr>
<tr>
<td>$ix$</td>
<td>$00000000000000000000000000000000$</td>
<td>$01110101$</td>
</tr>
<tr>
<td>$y$</td>
<td>$-15213$</td>
<td>$1100010010010011$</td>
</tr>
<tr>
<td>$iy$</td>
<td>$FF$</td>
<td>$11111111111111111111111111111111$</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Summary:

Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
- For small numbers yields expected behaviour

Negation: Complement & Increment

- Claim: Following Holds for 2’s Complement
  - $x + 1 = \bar{x}$
- Two’s Complement
  - Observation: $\bar{\bar{x}} + x = 1111\ldots11 \rightarrow -1$
  - $x$
  - $\bar{x}$
  - $\bar{x} + x$
  - $-1$
Complement & Increment Examples

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 15213</td>
<td></td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x = 15214</td>
<td></td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>x = 15213 + 1</td>
<td></td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

x = 0

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>~0 + 1</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

What is \(-TMIN\)? (pp.37)

Unsigned Addition

- Standard Addition Function
- Ignores carry output
- Implements Modular Arithmetic

\[ s = UAdd(u, v) = (u + v) \mod 2^w \]

\[ UAdd(u, v) = \begin{cases} 
 u + v & \text{if true sum } \leq 2^w \\
 u + v - 2^w & \text{otherwise} 
\end{cases} \]

Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers \( u, v \)
  - Compute true sum \( Add(u, v) \)
  - Values increase linearly with \( u \) and \( v \)
  - Forms planar surface

Visualizing Unsigned Addition

- Wraps Around
  - If true sum \( \geq 2^w \)
  - At most once

True Sum

\[ 2^w \]

Modular Sum

\[ UAdd(u, v) \]

Two’s Complement Addition

- Operands: \( w \) bits
- True Sum: \( w + 1 \) bits
- Discard Carry: \( w \) bits

- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C
  - `t = (int)((unsigned) u + (unsigned) v);`
  - Will give \( t = t \)

TAdd Overflow

- Functionality
  - True sum requires \( w + 1 \) bits
  - Drop off Most Significant Bit (MSB)
  - Treat remaining bits as 2’s comp. integer

True Sum

\[ 2^{w+1} - 1 \]

TAdd Result

\[ 011_1 \]

Overflow

\[ 100_0 \]
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum ≈ 2^w-1
    - Becomes negative
    - At most once
  - If sum ≈ 2^0
    - Becomes positive
    - At most once

Charaterizing TAdd

- **Functionality**
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer

Multiplication

- **Computing Exact Product of w-bit numbers** x, y
  - Either signed or unsigned
- **Ranges**
  - **Unsigned**: Up to 2w bits
    - Result range: 0 ≤ x * y ≤ (2w-1)^2 = 2^2w - 2^w + 1
  - Two’s complement min (negative): Up to 2w-1 bits
    - Result range: x * y ≥ 2^w*(2^w-1) = 2^2w - 2^w
  - Two’s complement max (positive): Up to 2w bits, but only for \( TMin_y \)
    - Result range: x * y ≤ (2^w-1)^2 = 2^(2w-2)
- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages

Unsigned Multiplication in C

- **Operands**: w bits
  - \( u \) \( v \)
- **True Product**: 2w bits
  - \( u * v \)
- **Discard w bits**: w bits
  - \( TMult(u, v) \)

Signed Multiplication in C

- **Operands**: w bits
  - \( u \) \( v \)
- **True Product**: 2w bits
  - \( u \cdot v \)
- **Discard w bits**: w bits
  - \( TMult(u, v) \)

Power-of-2 Multiply with Shift

- **Operation**
  - \( u \ll k \) gives \( u \cdot 2^k \)
  - Both signed and unsigned
- **Operands**: w bits
  - \( u \ll v \)
- **True Product**: \( w \ll k \) bits
  - \( UMult(u, v) \)
- **Discard k bits**: w bits
  - \( TMult(u, v) \)

- **Examples**
  - \( u \ll 3 \)
    - \( u \cdot 8 \)
  - \( u \ll 5 - u \ll 3 \)
    - \( (u \cdot 32) - (u \cdot 8) \)
    - \( u \cdot 24 \)
  - 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 \( u / 2^k \)
  - Uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16211</td>
<td>3B</td>
<td>00111011 00110101</td>
</tr>
<tr>
<td>2</td>
<td>7606.5</td>
<td>1B</td>
<td>00111101 10100100</td>
</tr>
<tr>
<td>4</td>
<td>390.8125</td>
<td>35</td>
<td>00000100 00110100</td>
</tr>
<tr>
<td>8</td>
<td>59.4257813</td>
<td>39</td>
<td>00</td>
</tr>
</tbody>
</table>

**Case 1:**

**Dividend:**
\[
\frac{u}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 -00000000
\]
**Divisor:**
\[
\frac{1}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 -00000000
\]

**Result:**
\[
\frac{u}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 01111111
\]

**Correct Power-of-2 Divide**

- Quotient of Negative Number by Power of 2
  - Want \( [x / 2^k] \) (Round Toward 0)
  - Compute as \( \left[ (x 2^k + 1) / 2^k \right] \)
  - In C: \( x + (1<<k) \gg k \)
  - Biases dividend toward 0

**Case 1:** No rounding

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16211</td>
<td>3B</td>
<td>00111011 00110101</td>
</tr>
<tr>
<td>2</td>
<td>7606.5</td>
<td>1B</td>
<td>00111101 10100100</td>
</tr>
<tr>
<td>4</td>
<td>390.8125</td>
<td>35</td>
<td>00000100 00110100</td>
</tr>
<tr>
<td>8</td>
<td>59.4257813</td>
<td>39</td>
<td>00</td>
</tr>
</tbody>
</table>

**Case 2:**

**Dividend:**
\[
\frac{x}{2^k} = \frac{x + (1<<k)}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 00111111
\]

**Divisor:**
\[
\frac{1}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 -00000000
\]

**Result:**
\[
\frac{x}{2^k} = \frac{1}{2^k} \quad \text{Bin.} -11111111 01111111
\]

**Signed Power-of-2 Divide with Shift**

- Quotient of Signed by Power of 2
  - \( x \gg k \) gives \( x / 2^k \)
  - Uses arithmetic shift
  - Ok when \( x \geq 0 \), but 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>1</td>
<td>-16211</td>
<td>-3B</td>
<td>11000100 11101010</td>
</tr>
<tr>
<td>2</td>
<td>-7606.5</td>
<td>-1B</td>
<td>11001101 11011010</td>
</tr>
<tr>
<td>4</td>
<td>-390.8125</td>
<td>-35</td>
<td>11111111 11110100</td>
</tr>
<tr>
<td>8</td>
<td>-59.4257813</td>
<td>-39</td>
<td>1100010 01001001</td>
</tr>
</tbody>
</table>

**Arithmetic: Basic Rules**

- Addition:
  - **Unsigned/signed:** Normal addition followed by truncate, same operation on bit level
  - **Unsigned:** addition mod 2^n
    - Mathematical addition + possible subtraction of 2^n
  - **Signed:** modified addition mod 2^n (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^n

- Multiplication:
  - **Unsigned/signed:** Normal multiplication followed by truncate, same operation on bit level
  - **Unsigned:** multiplication mod 2^n
  - **Signed:** modified multiplication mod 2^n (result in proper range)
### Arithmetic: Basic Rules

- **Left shift (Multiplication)**
  - Unsigned/signed: multiplication by $2^k$
  - Always logical left shift
- **Right shift (Division)**
  - Unsigned: logical right shift, div (division + round to zero) by $2^k$
  - Signed: arithmetic shift
  - Positive numbers: div (division + round to zero) by $2^k$
  - Negative numbers: div (division + round away from zero) by $2^k$
  - Use biasing to fix

### Why Should I Use Unsigned?

- **Don’t use without understanding implications**
  - Easy to make mistakes
  - for (i = cnt-2; i >= 0; i--)
    - a[i] += a[i+1];
  - Can be very subtle
    - #define DELTA sizeof(int)
    - int i;
    - for (i = CNT; i-Delta >= 0; i+=DELTA)
      - ...

### Counting Down with Unsigned

- **Proper way to use unsigned as loop index**
  - unsigned i;
    - for (i = cnt-2; i < cnt; i--)
      - a[i] += a[i+1];
  - **See Robert Seacord, Secure Coding in C and C++**
    - C Standard guarantees that unsigned addition will behave like modular arithmetic
      - $0 - 1 \rightarrow \text{UMax}$
  - **Even better**
    - size_t i;
      - for (i = cnt-2; i < cnt; i--)
        - a[i] += a[i+1];
    - Data type size_t defined as unsigned value with length = word size
    - Code will work even if cnt = UMax
    - What if cnt is signed and < 0?

### Why Should I Use Unsigned? (cont.)

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

### Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings**

### 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 Machine Has a "Word Size"
  - Nominal size of integer-valued data
    - Including addresses
  - Many current machines use 32 bits (4 bytes) words
    - Limits addresses to $2^{32} = 4$GB
    - Becoming too small for memory-intensive applications
  - Most high-end systems use 64 bits (8 bytes) words
    - Potential address space $= 2^{64}$ bytes (exabytes)
  - Machines 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 a word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
- Need to distinguish the notion of address vs. data stored in that address

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address, i.e. least significant byte first
  - Little Endian: x86, ARM processors running Android, iOS and Windows
    - Least significant byte has lowest address, i.e. least significant byte first

Example

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Reading Byte-Reversed Listings

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:51</td>
<td>pop   %ebx</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:81</td>
<td>c3    12 00 00</td>
<td>add $0x12ab, %ebx</td>
</tr>
<tr>
<td>8048368:83</td>
<td>bb    28 00 00 00</td>
<td>x movl %eax,0x2814%ebx</td>
</tr>
</tbody>
</table>

Deciphering Numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Representing Integers

int A = 15213;
long int C = 15213;
Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;
void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p \t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Print directives:
- %p: Print pointer
- %x: Print Hexadecimal

```
typedef unsigned char *pointer;
void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p \t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

**show_bytes Execution Example**

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):
```
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

Representing Pointers

- Different compilers & machines assign different locations to objects
- Even get different results each time run program

```
int B = -15213;
int *P = &B;
```

Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
  - Digit i has code 0x30+i
  - String should be null-terminated
  - Final character = 0
- **Compatibility**
  - Byte ordering not an issue

```
char S[6] = "18243";
```

```
char S[6] = "18243";
```

```
char S[6] = "18243";
```