COS 217: Introduction to Programming Systems

Crash Course in C (Part 2)

The Design of C Language Features and Data Types and their Operations and Representations
INTEGERS
Integer Data Types

Integer types of various sizes: \{signed, unsigned\} \{char, short, int, long\}

- char is 1 byte
  - Number of bits per byte is unspecified!
    (but in the 21st century, safe to assume it’s 8)
- Sizes of other integer types not fully specified but constrained:
  - int was intended to be “natural word size” of hardware
  - $2 \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$

On ArmLab:

- Natural word size: 8 bytes (“64-bit machine”)
- char: 1 byte
- short: 2 bytes
- int: 4 bytes (compatibility with widespread 32-bit code)
- long: 8 bytes
Integer Literals

- Decimal int: 123
- Octal int: 0173 = 123
- Hexadecimal int: 0x7B = 123
- Use "L" suffix to indicate long literal
- No suffix to indicate char-sized or short integer literals; instead, cast

Examples
- int: 123, 0173, 0x7B
- long: 123L, 0173L, 0x7BL
- short: (short)123, (short)0173, (short)0x7B
Unsigned Integer Data Types

unsigned types: unsigned char, unsigned short, unsigned int, and unsigned long
  • Hold only non-negative integers

Default for short, int, long is signed
  • char is system dependent (on armlab char is unsigned)
  • Use "U" suffix to indicate unsigned literal

Examples
  • unsigned int:
    • 123U, 0173U, 0x7BU
    • Oftentimes the U is omitted for small values: 123, 0173, 0x7B
      • (Technically there is an implicit cast from signed to unsigned, but in these cases it shouldn’t make a difference.)
  • unsigned long:
    • 123UL, 0173UL, 0x7BUL
  • unsigned short:
    • (unsigned short)123, (unsigned short)0173, (unsigned short)0x7B
“Character” Data Type

The C char type

- char is designed to hold an ASCII character
  - Should be used when you’re dealing with characters: character-manipulation functions we’ve seen (such as toupper) take and return char
- char might be signed (-128..127) or unsigned (0..255)
  - But since 0 ≤ ASCII ≤ 127 it doesn’t really matter when used as an actual character
  - If using chars for arbitrary one-byte data, good to specify as unsigned char
Character Literals

Single quote syntax: 'a'

Use backslash (the escape character) to express special characters

- Examples (with numeric equivalents in ASCII):

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>ASCII Value</th>
<th>Hexadecimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>the a character</td>
<td>97, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\141'</td>
<td>the a character, octal form</td>
<td>141, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\x61'</td>
<td>the a character, hexadecimal form</td>
<td>61, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'b'</td>
<td>the b character</td>
<td>98, 01100010&lt;sub&gt;B&lt;/sub&gt;, 62&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'A'</td>
<td>the A character</td>
<td>65, 01000001&lt;sub&gt;B&lt;/sub&gt;, 41&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>the B character</td>
<td>66, 01000010&lt;sub&gt;B&lt;/sub&gt;, 42&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\0'</td>
<td>the null character</td>
<td>0, 00000000&lt;sub&gt;B&lt;/sub&gt;, 0&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'0'</td>
<td>the zero character</td>
<td>48, 00110000&lt;sub&gt;B&lt;/sub&gt;, 30&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'1'</td>
<td>the one character</td>
<td>49, 00110001&lt;sub&gt;B&lt;/sub&gt;, 31&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\n'</td>
<td>the newline character</td>
<td>10, 00001010&lt;sub&gt;B&lt;/sub&gt;, A&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\t'</td>
<td>the horizontal tab character</td>
<td>9, 00001001&lt;sub&gt;B&lt;/sub&gt;, 9&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'\'</td>
<td>the backslash character</td>
<td>92, 01011100&lt;sub&gt;B&lt;/sub&gt;, 5C&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>'''</td>
<td>the single quote character</td>
<td>96, 01100000&lt;sub&gt;B&lt;/sub&gt;, 60&lt;sub&gt;H&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Back in 1970s, English was the only language in the world so we all used this alphabet:

ASCII: American Standard Code for Information Interchange

In the 21st century, it turns out there are other languages!
Modern Unicode

When C was designed, it only considered ASCII, which fits in 7 bits, so C’s chars are 8 bits long.

When Java was designed, Unicode fit into 16 bits, so Java’s chars are 16 bits long.

Then this happened:

1988:
MY "UNICODE" STANDARD SHOULD HELP REDUCE PROBLEMS CAUSED BY INCOMPATIBLE BINARY TEXT ENCODINGS.

2018:
SENATOR ANGUS KING
@SenAngusKing
GREAT NEWS FOR MAINE—WE'RE GETTING A LOBSTER EMOJI!!! THANKS TO @UNICODE FOR RECOGNIZING THE IMPACT OF THIS CRITICAL CRUSTACEAN IN MAINE AND ACROSS THE COUNTRY.
YOURS TRULY
SENATOR 🦀
2/7/18 3:12PM

WHAT...WHAT HAPPENED IN THOSE THIRTY YEARS?
THINGS GOT A LITTLE WEIRD, OKAY?

https://xkcd.com/1953/
## Integer Types in Java vs. C

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>// 16 bits</td>
<td>unsigned char /* Note 2 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned short</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned (int)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td>Signed types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>// 8 bits</td>
<td>signed char /* Note 2 */</td>
</tr>
<tr>
<td>short</td>
<td>// 16 bits</td>
<td>(signed) short</td>
</tr>
<tr>
<td>int</td>
<td>// 32 bits</td>
<td>(signed) int</td>
</tr>
<tr>
<td>long</td>
<td>// 64 bits</td>
<td>(signed) long</td>
</tr>
</tbody>
</table>

1. Not guaranteed by C, but on armlab, char = 8 bits, short = 16 bits, int = 32 bits, long = 64 bits
2. Not guaranteed by C, but on armlab, char is unsigned

To understand C, must consider the representation of these types!
Representing Unsigned Integers

Mathematics
- Non-negative integers’ range is 0 to $\infty$

Computer programming
- Range limited by computer’s word size
- Word size is $n$ bits $\Rightarrow$ range is 0 to $2^n - 1$
- Exceed range $\Rightarrow$ overflow

Typical computers today
- $n = 32$ or 64, so range is 0 to $2^{32} - 1$ (~4B) or $2^{64} - 1$ (huge ... ~1.8e19)

Pretend computer
- $n = 4$, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4
- All points generalize to word size = $n$ (armlab: 64)
Representing Unsigned Integers

On 4-bit pretend computer

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

Addition

Start at right column
Proceed leftward
Carry 1 when necessary

Beware of overflow

How would you detect overflow programmatically?

Results are mod $2^4$
Subtracting Unsigned Integers

Subtraction

\[
\begin{array}{c}
\text{111} \\
10 - 7 \quad 1010_B \\
\hline
3 - 0111_B \\
\hline
3 - 1010_B \\
\hline
9 - 1001_B \\
\end{array}
\]

Start at right column
Proceed leftward
Borrow when necessary

Beware of overflow

Results are mod \(2^4\)

How would you detect overflow programmatically?
Shifting Unsigned Integers

Bitwise right shift (>> in C): fill on left with zeros

\[ 10 \gg 1 \Rightarrow 5 \]
\[ 1010_B \quad 0101_B \]

\[ 10 \gg 2 \Rightarrow 2 \]
\[ 1010_B \quad 0010_B \]

Bitwise left shift (<< in C): fill on right with zeros

\[ 5 \ll 1 \Rightarrow 10 \]
\[ 0101_B \quad 1010_B \]

\[ 3 \ll 2 \Rightarrow 12 \]
\[ 0011_B \quad 1100_B \]

\[ 3 \ll 3 \Rightarrow 8 \]
\[ 0011_B \quad 1000_B \]

What is the effect arithmetically?

\[ 1010 \quad 0101 \]
\[ 1010 \quad 0010 \]

\[ 0101 \quad 1010 \]
\[ 1100 \]

\[ 0011 \quad 1000 \]

\[ \rightarrow \text{Results are mod } 2^4 \]
Other Bitwise Operations on Unsigned Integers

Bitwise NOT (~ in C)
- Flip each bit

\[
\begin{align*}
\sim 10 & \Rightarrow 5 \\
1010_B & \quad 0101_B \\
\sim 5 & \Rightarrow 10 \\
0101_B & \quad 1010_B
\end{align*}
\]

Bitwise AND (& in C)
- AND (1=True, 0=False) corresponding bits

\[
\begin{align*}
10 & \quad 1010_B \\
& \quad & \quad 0111_B \\
\& 7 & \quad \& \quad ---- \\
-- & \quad \& \quad ---- \\
2 & \quad 0010_B \\
\end{align*}
\]

\[
\begin{align*}
10 & \quad 1010_B \\
& \quad \& \quad 0010_B \\
\& 2 & \quad \& \quad ---- \\
-- & \quad \& \quad ---- \\
2 & \quad 0010_B \\
\end{align*}
\]

Useful for “masking” bits to 0
Other Bitwise Operations on Unsigned Ints

Bitwise OR: (| in C)
- Logical OR corresponding bits

```
  10  1010_B  
|  1 |  0001_B  
--  ----  
 11  1011_B  
```

Useful for “masking” bits to 1

Bitwise exclusive OR (^ in C)
- Logical exclusive OR corresponding bits

```
  10  1010_B  
^ 10  ^ 1010_B  
--  ----  
  0   0000_B  
```

x ^ x sets all bits to 0
A Bit Complicated

How do you set bit k (where the least significant bit is bit 0) of unsigned variable u to zero (leaving everything else in u unchanged)?

A. \( u &= (0 << k); \)
B. \( u |= (1 << k); \)
C. \( u |= ~(1 << k); \)
D. \( u &= ~(1 << k); \)
E. \( u = ~u ^ (1 << k); \)

D:

\[ 1 << k \Rightarrow 0{\{n-1-k\}10{\{k\}}} \]

\[ ~(1 << n) \Rightarrow 1{\{n-1-k\}01{\{k\}}} \]

\[ u &= ~(1 << k); \Rightarrow u_i{\{n-1-k\}0u_i{\{k\}}} \]
Aside: Using Bitwise Ops for Arith

Can use <<, >>, and & to do some arithmetic efficiently

\[ x \times 2^y = x \ll y \]
\[ \text{• } 3 \times 4 = 3 \times 2^2 = 3 \ll 2 = 12 \]

\[ x \div 2^y = x \gg y \]
\[ \text{• } 13 \div 4 = 13 \div 2^2 = 13 \gg 2 = 3 \]

\[ x \mod 2^y = x \& (2^y - 1) \]
\[ \text{• } 13 \mod 4 = 13 \mod 2^2 = 13 \& (2^2 - 1) \]
\[ = 13 \& 3 = 1 \]

Fast way to **multiply**
by a power of 2

Fast way to **divide**
**unsigned** by power of 2

Fast way to **mod**
by a power of 2

Many compilers will
do these transformations
automatically!
Unfortunate reminder: negative numbers exist
## Sign-Magnitude

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

**Definition**

High-order bit indicates sign
- 0 ⇒ **positive**
- 1 ⇒ **negative**

Remaining bits indicate magnitude
- $0101_B = 101_B = 5$
- $1101_B = -101_B = -5$
### Sign-Magnitude (cont.)

#### Computing negative

\[ \text{neg}(x) = \text{flip high order bit of } x \]

- \( \text{neg}(0101_B) = 1101_B \)
- \( \text{neg}(1101_B) = 0101_B \)

#### Pros and cons

- easy to understand, easy to negate
- symmetric
- two representations of zero
- need different algorithms to add signed and unsigned numbers

<table>
<thead>
<tr>
<th>Integer</th>
<th>Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>1111</td>
</tr>
<tr>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>-5</td>
<td><strong>1101</strong></td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>-0</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td><strong>0101</strong></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>
Ones’ Complement

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

**Definition**

High-order bit has weight \(-(2^{b-1}-1)\)

\[
\begin{align*}
1010_\text{B} &= (1*7) + (0*4) + (1*2) + (0*1) \\
&= -5
\end{align*}
\]

\[
\begin{align*}
0010_\text{B} &= (0*7) + (0*4) + (1*2) + (0*1) \\
&= 2
\end{align*}
\]

Similar pros and cons to sign-magnitude
Two’s Complement

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

**Definition**

High-order bit has weight \(-(2^{b-1})\)

\[
1010_B = (1 \times -8) + (0 \times 4) + (1 \times 2) + (0 \times 1) = -6
\]

\[
0010_B = (0 \times -8) + (0 \times 4) + (1 \times 2) + (0 \times 1) = 2
\]
Two’s Complement (cont.)

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

Computing negative

\[ \text{neg}(x) = \sim x + 1 \]

\[ \text{neg}(x) = \text{onescomp}(x) + 1 \]

\[ \text{neg}(0101_B) = 1010_B + 1 = 1011_B \]

\[ \text{neg}(1011_B) = 0100_B + 1 = 0101_B \]

Pros and cons

- not symmetric
  - (“extra” negative number)
- one representation of zero
- same algorithm adds signed and unsigned integers
Adding Signed Integers

- **pos + pos**
  - 3 $+$ 3 $\rightarrow$ 6 $0110_2$
  - 11 $+$ 0011$_2$ $\rightarrow$ 1111$_2$
  - 2 $0010_2$

- **pos + pos (overflow)**
  - 7 $+$ 1 $\rightarrow$ -8 $1000_2$
  - 111 $+$ 001$\rightarrow$ 1111$_2$
  - -8 $1000_2$

- **pos + neg**
  - 3 $+$ -1 $\rightarrow$ 2 $0010_2$
  - 1111$+$ 1111$_2$ $\rightarrow$ 1111$_2$
  - 2 $0010_2$

- **neg + neg**
  - -3 $+$ -2 $\rightarrow$ -5 $1011_2$
  - 11 $+$ 1110$_2$ $\rightarrow$ 1111$_2$
  - -5 $1011_2$

- **neg + neg (overflow)**
  - -6 $+$ -5 $\rightarrow$ 5 $0101_2$
  - 11 $+$ 1011$_2$ $\rightarrow$ 1111$_2$
  - 5 $0101_2$

How would you detect overflow programmatically?
Subtracting Signed Integers

Perform subtraction with borrows

\[
\begin{array}{c|c}
3 & 0011_B \\
4 & 0100_B \\
\hline
-1 & 1111_B \\
\end{array}
\]

or

Compute two’s comp and add

\[
\begin{array}{c|c}
3 & 0011_B \\
4 & 1100_B \\
\hline
-1 & 1111_B \\
\end{array}
\]

\[
\begin{array}{c|c}
5 & 1011_B \\
2 & 1110_B \\
\hline
-3 & 1101_B \\
\end{array}
\]

\[
\begin{array}{c|c}
5 & 1011_B \\
2 & 0010_B \\
\hline
-3 & 1101_B \\
\end{array}
\]
Negating Signed Ints: Math

Question: Why does two’s comp arithmetic work?

Answer: \([-b] \mod 2^4 = \text{twoscomp}(b) \mod 2^4\)

\[
\begin{align*}
[-b] \mod 2^4 &= [2^4 - b] \mod 2^4 \\
&= [2^4 - 1 - b + 1] \mod 2^4 \\
&= [(2^4 - 1 - b) + 1] \mod 2^4 \\
&= [\text{onescomp}(b) + 1] \mod 2^4 \\
&= \text{twoscomp}(b) \mod 2^4
\end{align*}
\]

So: \([a - b] \mod 2^4 = [a + \text{twoscomp}(b)] \mod 2^4\)

\[
\begin{align*}
[a - b] \mod 2^4 &= [a + 2^4 - b] \mod 2^4 \\
&= [a + 2^4 - 1 - b + 1] \mod 2^4 \\
&= [a + (2^4 - 1 - b) + 1] \mod 2^4 \\
&= [a + \text{onescomp}(b) + 1] \mod 2^4 \\
&= [a + \text{twoscomp}(b)] \mod 2^4
\end{align*}
\]
Ring theory.

If $n > 0$, $\mathbb{Z}/(n)$ is a finite commutative ring, with properties:

$$\overline{a_n + b_n} = (\overline{a} + \overline{b})_n; \overline{a_n - b_n} = (\overline{a} - \overline{b})_n; \overline{a_n b_n} = (\overline{ab})_n$$
Shifting Signed Integers

Bitwise left shift (<< in C): fill on right with zeros

\[ \begin{align*}
3 & \ll 1 \Rightarrow 6 \\
0011_B & \quad 0110_B \\
-3 & \ll 1 \Rightarrow -6 \\
1101_B & \quad 1010_B \\
-3 & \ll 2 \Rightarrow 4 \\
1101_B & \quad 0100_B
\end{align*} \]

What is the effect arithmetically?

Results are mod 2^4

Bitwise right shift: fill on left with ???
Shifting Signed Integers (cont.)

Bitwise arithmetic right shift: fill on left with sign bit

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0110(_B)</td>
<td>0011(_B)</td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>1010(_B)</td>
<td>1101(_B)</td>
<td></td>
</tr>
</tbody>
</table>

What is the effect arithmetically?

Bitwise logical right shift: fill on left with zeros

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0110(_B)</td>
<td>0011(_B)</td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1010(_B)</td>
<td>0101(_B)</td>
<td></td>
</tr>
</tbody>
</table>

What is the effect arithmetically???

In C, right shift (\(>>\)) could be logical or arithmetic

- Not specified by standard (happens to be arithmetic on armlab)
- Best to avoid shifting signed integers
Other Operations on Signed Ints

Bitwise NOT (~ in C)
  • Same as with unsigned ints

Bitwise AND (& in C)
  • Same as with unsigned ints

Bitwise OR: (| in C)
  • Same as with unsigned ints

Bitwise exclusive OR (^ in C)
  • Same as with unsigned ints

Best to avoid with signed integers
Issue: Should C provide tailored assignment operators?

Thought process

- The construct `a = b + c` is flexible
- The construct `i = i + c` is somewhat common
- The construct `i = i + 1` is very common
- Special-purpose operators make code more expressive
  - Might reduce some errors
  - May complicate the language and compiler

Decisions

- Introduce `+=` operator to do things like `i += c`
- Extend to `-=` `/=` `~=` `&=` `|=` `^=` `<<=` `>>=`
- Special-case increment and decrement: `i++` `i--`
- Provide both pre- and post-inc/dec: `x = ++i; y = i++;`
Q: What are i and j set to in the following code?

```
    i = 5;
j = i++;  
j += ++i;
```

<table>
<thead>
<tr>
<th>Option</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. 5, 7</td>
<td>5, 5</td>
<td>7</td>
</tr>
<tr>
<td>B. 7, 5</td>
<td>?</td>
<td>5</td>
</tr>
<tr>
<td>C. 7, 11</td>
<td>i=5;</td>
<td>?</td>
</tr>
<tr>
<td>D. 7, 12</td>
<td>j = i++;</td>
<td>5</td>
</tr>
<tr>
<td>E. 7, 13</td>
<td>j += ++i;</td>
<td>12</td>
</tr>
</tbody>
</table>

D
Q: What does the following code print?

```c
int i = 1;
switch (i++) {
    case 1: printf("%d", ++i);
    case 2: printf("%d", i++);
}
```

A. 1
i++ increments i to 2, but evaluates to i’s old value: 1

B. 2
So switch on 1, not 2!

C. 3
++i evaluates to new value, so 3 is printed

D. 22
FALL THROUGH GOTCHA!

E. 33
i++ evaluates to old value, so 3 is printed again
Incremental Iteration

Q: What does the following code print?

```c
int i = 1;
switch (i=i++) {
    case 1: printf("%d", ++i);
    case 2: printf("%d", i++);
}
```

A. 1
   i++ increments i to 2, but evaluates to i’s old value: 1

B. 2
   i = 1 overwrites our just-incremented 2 back to 1

C. 3
   ... switch on 1, so now continue into case 1 with i as 1,
   so we end up printing 22.

D. 22

E. 33
sizeof Operator

Issue: How to determine the sizes of data?

Thought process
- The sizes of most primitive types are un- or under-specified
- Provide a way to find size of a given variable programmatically

Decisions
- Provide a sizeof operator
  - Applied at compile-time
  - Operand can be a data type
  - Operand can be an expression, from which the compiler infers a data type

Examples, on armlab using gcc217
- sizeof(int) evaluates to 4
- sizeof(i) – where i is a variable of type int – evaluates to 4
Q: What is the value of the following sizeof expression on the armlab machines?

```c
int i = 1;
sizeof(i + 2L)
```

A. 3  
B. 4  
C. 8  
D. 12  
E. error

Promote i to long, add 1L + 2L. Result, 3L, is a long. Longs are 8 bytes on armlab.
LOGICAL TYPES
Logical Data Types

• No separate logical or Boolean data type

• Represent logical data using type char or int
  • Or any primitive type! :/

• Conventions:
  • Statements (if, while, etc.) use  0 ⇒ FALSE, ≠0 ⇒ TRUE
  • Relational operators (<, >, etc.) and logical operators (!, &&, ||)
    produce the result 0 or 1, specifically
Using integers to represent logical data permits shortcuts

```
...  
int i;  
...  
if (i)  /* same as (i != 0) */  
    statement1;  
else  
    statement2;  
...  
```

It also permits some really bad code...

```
i = (1 != 2) + (3 > 4);  
```
Logical Data Type Dangers

The lack of a logical data type hampers compiler's ability to detect some errors

```
... int i;
... i = 0;
... if (i = 5)
   statement1;
...```

What happens in Java?

What happens in C?
Logical vs. Bitwise Ops

Logical AND (&&) vs. bitwise AND (&)

- 2 (TRUE) && 1 (TRUE) => 1 (TRUE)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
<tr>
<td>&amp;&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------</td>
</tr>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
</tbody>
</table>

- 2 (TRUE) & 1 (TRUE) => 0 (FALSE)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
<tr>
<td>&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------</td>
</tr>
<tr>
<td>0</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

Implication:

- Use logical AND to control flow of logic
- Use bitwise AND only when doing bit-level manipulation
- Same for OR and NOT
Agenda

Thus far:

- Integer types in C
- Finite representation of unsigned integers
- Finite representation of signed integers
- Logical types (or lack thereof) in C

Up next:

- Finite representation of rational (floating-point) numbers
FLOATING POINT
Rational Numbers

Mathematics
• A rational number is one that can be expressed as the ratio of two integers
• Unbounded range and precision

Computer science
• Finite range and precision
• Approximate using floating point number
Floating Point Numbers

Like scientific notation: e.g., c is $2.99792458 \times 10^8 \text{ m/s}$

This has the form

$\text{(multiplier)} \times \text{(base)}^{\text{(power)}}$

In the computer,

- Multiplier is called mantissa
- Base is almost always 2
- Power is called exponent
Floating-Point Data Types

C specifies:

- Three floating-point data types: float, double, and long double
- Sizes unspecified, but constrained:
  - `sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)`

On ArmLab (and on pretty much any 21st-century computer using the IEEE standard)

- float: 4 bytes
- double: 8 bytes

On ArmLab (but varying across architectures)

- long double: 16 bytes
Floating-Point Literals

How to write a floating-point number?

• Either fixed-point or “scientific” notation
• Any literal that contains decimal point or "E" is floating-point
• The default floating-point type is double
• Append "F" to indicate float
• Append "L" to indicate long double

Examples

• double: 123.456, 1E-2, -1.23456E4
• float: 123.456F, 1E-2F, -1.23456E4F
• long double: 123.456L, 1E-2L, -1.23456E4L
IEEE Floating Point Representation

Common finite representation: IEEE floating point
  • More precisely: ISO/IEEE 754 standard

Using 32 bits (type `float` in C):
  • 1 bit: sign (0⇒positive, 1⇒negative)
  • 8 bits: exponent + 127
  • 23 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbb

Using 64 bits (type `double` in C):
  • 1 bit: sign (0⇒positive, 1⇒negative)
  • 11 bits: exponent + 1023
  • 52 bits: binary fraction of the form
    1.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
When was floating-point invented?

mantissa (noun): decimal part of a logarithm, 1865,  ➡️Answer: long before computers!
from Latin mantissa “a worthless addition, makeweight”

![Common Logarithms Table](image-url)
Floating Point Example

Sign (1 bit):
• $1 \Rightarrow$ negative

Exponent (8 bits):
• $10000011_B = 131$
• $131 - 127 = 4$

Mantissa (23 bits):
• $1.10110110000000000000000_B$
• $1 + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + (0 \times 2^{-5}) + (1 \times 2^{-6}) + (1 \times 2^{-7}) + (0 \times 2^{-\cdots}) = 1.7109375$

Number:
• $-1.7109375 \times 2^4 = -27.375$

32-bit representation:
$10000011101101100000000000000000$
Floating Point Consequences

“Machine epsilon”: smallest positive number you can add to 1.0 and get something other than 1.0

For float: $\varepsilon \approx 10^{-7}$
- No such number as 1.000000001
- Rule of thumb: “almost 7 digits of precision”

For double: $\varepsilon \approx 2 \times 10^{-16}$
- Rule of thumb: “not quite 16 digits of precision”

These are all relative numbers
Floating Point Consequences, cont

Just as decimal number system can represent only some rational numbers with finite digit count...
  • Example: 1/3 cannot be represented

Binary number system can represent only some rational numbers with finite digit count
  • Example: 1/5 cannot be represented

Beware of round-off error
  • Error resulting from inexact representation
  • Can accumulate
  • Be careful when comparing two floating-point numbers for equality
Floating away ...

What does the following code print?

```
double sum = 0.0;
double i;
for (i = 0.0; i != 10.0; i++)
    sum += 0.1;
if (sum == 1.0)
    printf("All good!\n");
else
    printf("Yikes!\n");
```

A. All good!
B. Yikes!
C. (Infinite loop)
D. (Compilation error)

B: Yikes!

... loop terminates, because we can represent 10.0 exactly by adding 1.0 at a time.

... but sum isn’t 1.0 because we can’t represent 1.0 exactly by adding 0.1 at a time.
Summary

<table>
<thead>
<tr>
<th>Integer types in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite representation of unsigned integers</td>
</tr>
<tr>
<td>Finite representation of signed integers</td>
</tr>
<tr>
<td>Logical types in C (or lack thereof)</td>
</tr>
<tr>
<td>Floating point types in C</td>
</tr>
<tr>
<td>Finite representation of rational (floating-point) numbers</td>
</tr>
</tbody>
</table>

Essential for proper understanding of
- C primitive data types
- Assembly language
- Machine language