Numbers (in C and otherwise)

Q: Why do computer programmers confuse Christmas and Halloween?
A: Because 25 Dec == 31 Oct
The Decimal Number System

Name
• “decem” (Latin) ⇒ ten

Characteristics
• For us, these symbols (Not universal ...)
  • 0 1 2 3 4 5 6 7 8 9
  • Positional
    • $2945 \neq 2495$
    • $2945 = (2 \times 10^3) + (9 \times 10^2) + (4 \times 10^1) + (5 \times 10^0)$

(Most) people use the decimal number system
The Binary Number System

**binary**

*adjective*: being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal.

From Late Latin *bīnārius* ("consisting of two").

**Characteristics**

- Two symbols: 0 1
- Positional: $1010_B \neq 1100_B$

Most (digital) computers use the binary number system

**Terminology**

- **Bit**: a single binary symbol ("binary digit")
- **Byte**: (typically) 8 bits
- **Nibble / Nybble**: 4 bits
# Decimal-Binary Equivalence

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</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>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td>22</td>
<td>10110</td>
</tr>
<tr>
<td>23</td>
<td>10111</td>
</tr>
<tr>
<td>24</td>
<td>11000</td>
</tr>
<tr>
<td>25</td>
<td>11001</td>
</tr>
<tr>
<td>26</td>
<td>11010</td>
</tr>
<tr>
<td>27</td>
<td>11011</td>
</tr>
<tr>
<td>28</td>
<td>11100</td>
</tr>
<tr>
<td>29</td>
<td>11101</td>
</tr>
<tr>
<td>30</td>
<td>11110</td>
</tr>
<tr>
<td>31</td>
<td>11111</td>
</tr>
</tbody>
</table>

... ...
Decimal-Binary Conversion

Binary to decimal: expand using positional notation

\[ 100101_B = (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 32 + 0 + 0 + 4 + 0 + 1 = 37 \]
(Decimal) Integer to binary: do the reverse

- Determine largest power of 2 that’s \( \leq \) number; write template

\[
37 = (\_\_\_\_\_\_2^5) + (\_\_\_\_\_\_2^4) + (\_\_\_\_\_\_2^3) + (\_\_\_\_\_\_2^2) + (\_\_\_\_\_\_2^1) + (\_\_\_\_\_\_2^0)
\]

- Fill in template

\[
37 = (1*2^5) + (0*2^4) + (0*2^3) + (1*2^2) + (0*2^1) + (1*2^0)
\]
Integer-Binary Conversion

Integer to binary division method

- Repeatedly divide by 2, consider remainder

37 / 2 = 18 R 1
18 / 2 =  9 R 0
 9 / 2 =  4 R 1
 4 / 2 =  2 R 0
 2 / 2 =  1 R 0
 1 / 2 =  0 R 1

Read from bottom to top: $100101_B$
The Hexadecimal Number System

Name
• “hexa-” (Ancient Greek ἕξα-) ⇒ six
• “decem” (Latin) ⇒ ten

Characteristics
• Sixteen symbols
  • 0 1 2 3 4 5 6 7 8 9 A B C D E F
• Positional
  • A13DH ≠ 3DA1H

Computer programmers often use hexadecimal or “hex”
• In C: 0x prefix (0xA13D, etc.)
Binary-Hexadecimal Conversion

Observation:
• $16^1 = 2^4$, so every 1 hexit corresponds to 4 bits

Binary to hexadecimal

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010000100111101_B</td>
<td>A 1 3 D_H</td>
</tr>
</tbody>
</table>

Digit count in binary number not a multiple of 4 ⇒ pad with zeros on left

Hexadecimal to binary

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1 3 D_H</td>
<td>1010000100111101_B</td>
</tr>
</tbody>
</table>

Discard leading zeros from binary number if appropriate
Base Conversion Quick Quiz

Convert binary 101010 into decimal and hex

A. 21 decimal, 1A hex
B. 42 decimal, 2A hex
C. 48 decimal, 32 hex
D. 55 decimal, 4G hex

hint: convert to hex first
The Octal Number System

Name
• “octo” (Latin) ⇒ eight

Characteristics
• Eight symbols
  • 0 1 2 3 4 5 6 7
• Positional
  • 17430 ≠ 73140

Computer programmers sometimes use octal (so does Mickey!)
• In C: 0 prefix (01743, etc.)
INTEGERS
Representing Unsigned (Non-Negative) Integers

Mathematics
- Non-negative integers’ range is 0 to ∞

Computers
- Range limited by computer’s word size
- Word size is n bits ⇒ range is 0 to $2^n - 1$
- Exceed range ⇒ overflow

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

<table>
<thead>
<tr>
<th>3</th>
<th>0011\text{B}</th>
</tr>
</thead>
<tbody>
<tr>
<td>+10</td>
<td>1010\text{B}</td>
</tr>
<tr>
<td>--</td>
<td>----</td>
</tr>
<tr>
<td>13</td>
<td>1101\text{B}</td>
</tr>
</tbody>
</table>

Start at right column
Proceed leftward
Carry 1 when necessary

<table>
<thead>
<tr>
<th>7</th>
<th>0111\text{B}</th>
</tr>
</thead>
<tbody>
<tr>
<td>+10</td>
<td>1010\text{B}</td>
</tr>
<tr>
<td>--</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td>0001\text{B}</td>
</tr>
</tbody>
</table>

Beware of overflow

Results are mod $2^4$

How would you detect overflow programmatically?
Subtracting Unsigned Integers

Subtraction

\[
\begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
10 & 1010_B \\
-7 & 0111_B \\
--- & ---- \\
3 & 0011_B \\
\end{array}
\]

Start at right column
Proceed leftward
Borrow when necessary

\[
\begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
3 & 0011_B \\
-10 & 1010_B \\
--- & ---- \\
9 & 1001_B \\
\end{array}
\]

Beware of overflow

Results are mod \(2^4\)

How would you detect overflow programmatically?
Reminder: negative numbers exist
Obsolete Attempt #1: 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>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 indicates sign

- 0 ⇒ positive
- 1 ⇒ negative

Remaining bits indicate magnitude

- \(0101_B = 101_B = 5\)
- \(1101_B = -101_B = -5\)

**Pros and cons**

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

Not used for integers today
Obsolete Attempt #2: 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 \(-(2^{b-1}-1)\)

\[
\begin{align*}
1010_B &= (1\times-7) + (0\times4) + (1\times2) + (0\times1) \\
      &= -5 \\
0010_B &= (0\times-7) + (0\times4) + (1\times2) + (0\times1) \\
      &= 2
\end{align*}
\]

Computing negative = flipping all bits

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Detailed Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pos + pos</strong></td>
<td>6 (0110_B)</td>
<td>11 (0011_B) + 3 (0011_B) -- 6 (0110_B)</td>
</tr>
<tr>
<td><strong>pos + pos (overflow)</strong></td>
<td>-8 (1000_B)</td>
<td>111 (0111_B) + 1 (0001_B) -- -8 (1000_B)</td>
</tr>
<tr>
<td><strong>pos + neg</strong></td>
<td>2 (0010_B)</td>
<td>1111 (1111_B) + -1 (1111_B) -- 2 (0010_B)</td>
</tr>
<tr>
<td><strong>neg + neg</strong></td>
<td>-5 (1011_B)</td>
<td>-3 (1101_B) + -2 (1110_B) -- -5 (1011_B)</td>
</tr>
<tr>
<td><strong>neg + neg (overflow)</strong></td>
<td>5 (0101_B)</td>
<td>-6 (1010_B) + -5 (1011_B) -- 5 (0101_B)</td>
</tr>
</tbody>
</table>

How would you detect overflow programmatically?
How would you compute $3 - 4$?

\[
\begin{array}{c c c c c}
3 & 0011_B \\
- 4 & -0100_B \\
-- & ---- \\
? & ????_B \\
\end{array}
\]
Subtracting Signed Integers

Perform subtraction with borrows

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

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

Compute two’s complement and add

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

\[
\begin{array}{c|c}
5 & 1011_B \\
+2 & 0010_B \\
-- & ---- \\
-3 & 1101_B \\
\end{array}
\]

Perform subtraction with borrows or Compute two’s complement and add
Question: Why does two’s comp arithmetic work?

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

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

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

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

Integer types of various sizes: \{signed, unsigned\} \{char, short, int, long\}
- Shortcuts: “signed” assumed for short/int/long; “unsigned” means unsigned int
- 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

What decisions did the designers of Java make?
### Integer Types in Java vs. C

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned types</strong></td>
<td>char // 16 bits</td>
<td>unsigned char /* Note 1 */</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><strong>Signed types</strong></td>
<td>byte // 8 bits</td>
<td>signed char /* Note 2 */</td>
</tr>
<tr>
<td></td>
<td>short // 16 bits</td>
<td>(signed) short</td>
</tr>
<tr>
<td></td>
<td>int // 32 bits</td>
<td>(signed) int</td>
</tr>
<tr>
<td></td>
<td>long // 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
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
Integer Literals in C

- 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
- Use "U" suffix to indicate unsigned literal

Examples

- int: 123, 0173, 0x7B
- long: 123L, 0173L, 0x7BL
- short: (short)123, (short)0173, (short)0x7B
- unsigned: 123U, 0173U, 0x7BU
- unsigned long: 123UL, 0173UL, 0x7BUL
- unsigned short: (unsigned short)123, (unsigned short)0173, (unsigned short)0x7B
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
OPERATIONS ON NUMBERS
Reading / Writing Numbers

Motivation

- Must convert between external form (sequence of character codes) and internal form
- Could provide `getchar()`, `putshort()`, `getint()`, `putfloat()`, etc.
- Alternative implemented in C: parameterized functions

`scanf()` and `printf()`

- Can read/write any primitive type of data
- First parameter is a format string containing conversion specs: size, base, field width
- Can read/write multiple variables with one call

See King book for details
Operators in C

• Typical arithmetic operators: + − * / %
• Typical relational operators: == !!= < <= > >=
  • Each evaluates to FALSE ⇒ 0, TRUE ⇒ 1
• Typical logical operators: ! && ||
  • Each interprets 0 ⇒ FALSE, non-0 ⇒ TRUE
  • Each evaluates to FALSE ⇒ 0, TRUE ⇒ 1
• Cast operator: (type)
• Bitwise operators: ~ & | ^ >> <<
Shifting Unsigned Integers

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

\[
\begin{align*}
10 & >> 1 \Rightarrow 5 \\
1010_2 & \quad 0101_2
\end{align*}
\]

\[
\begin{align*}
10 & >> 2 \Rightarrow 2 \\
1010_2 & \quad 0010_2
\end{align*}
\]

What is the effect arithmetically?

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

\[
\begin{align*}
5 & << 1 \Rightarrow 10 \\
0101_2 & \quad 1010_2
\end{align*}
\]

\[
\begin{align*}
3 & << 2 \Rightarrow 12 \\
0011_2 & \quad 1100_2
\end{align*}
\]

\[
\begin{align*}
3 & << 3 \Rightarrow 8 \\
0011_2 & \quad 1000_2
\end{align*}
\]

What is the effect arithmetically?

Results are mod 2^4
Other Bitwise Operations on Unsigned Integers

Bitwise NOT (~ in C)
• Flip each bit

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

Useful for “masking” bits to 0

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

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

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

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
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 0000010</td>
</tr>
<tr>
<td>&amp;&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</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 0000010</td>
</tr>
<tr>
<td>&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</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
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. \(u | = \sim(1 << k);\)
D. \(u \&= \sim(1 << k);\)
E. \(u = \sim u ^ (1 << k);\)
Aside: Using Bitwise Ops for Arithmetic

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

\[ x \times 2^y = x << y \]
- \[ 3 \times 4 = 3 \times 2^2 = 3 \times 2 \Rightarrow 12 \]

\[ x \div 2^y = x >> y \]
- \[ 13 \div 4 = 13 \div 2^2 = 13 \div 2 \Rightarrow 3 \]

\[ x \% 2^y = x \& (2^y - 1) \]
- \[ 13 \% 4 = 13 \% 2^2 = 13 \& (2^2 - 1) = 13 \& 3 \Rightarrow 1 \]

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!
Shifting Signed Integers

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

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

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

\[
\begin{align*}
6 & \gg 1 \Rightarrow 3 \\
0110_2 & \quad 0011_2 \\
-6 & \gg 1 \Rightarrow -3 \\
1010_2 & \quad 1101_2
\end{align*}
\]

What is the effect arithmetically?

Bitwise logical right shift: fill on left with zeros

\[
\begin{align*}
6 & \gg 1 \Rightarrow 3 \\
0110_2 & \quad 0011_2 \\
-6 & \gg 1 \Rightarrow 5 \\
1010_2 & \quad 0101_2
\end{align*}
\]

What is the effect arithmetically?

In C, right shift (\texttt{>>}) 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
Assignment Operator

Many high-level languages provide an assignment statement

C provides an assignment operator
- Performs assignment, and then evaluates to the assigned value
- Allows assignment to appear within larger expressions
Assignment Operator Examples

Examples

```c
i = 0;
    /* Side effect: assign 0 to i.
       Evaluate to 0.
    
    j = i = 0; /* Assignment op has R to L associativity */
    /* Side effect: assign 0 to i.
       Evaluate to 0.
       Side effect: assign 0 to j.
       Evaluate to 0. */

while (((i = getchar()) != EOF) ...)
    /* Read a character.
       Side effect: assign that character to i.
       Evaluate to that character.
       Compare that character to EOF.
       Evaluate to 0 (FALSE) or 1 (TRUE). */
```
Special-Purpose Assignment in C

Motivation

- The construct `a = b + c` is flexible
- The construct `i = i + c` is somewhat common
- The construct `i = i + 1` is very common

Assignment in C

- Introduce `+=` operator to do things like `i += c`
- Extend to `-=`  `*=`  `/=`  `~=`  `&=`  `^=`  `<==`  `>=`
- All evaluate to whatever was assigned
- Pre-increment and pre-decrement: `++i`  `--i`
- Post-increment and post-decrement (evaluate to old value): `i++`  `i--`
Q: What are i and j set to in the following code?

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

A. 5, 7  
B. 7, 5  
C. 7, 11  
D. 7, 12  
E. 7, 13
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  
B. 2  
C. 3  
D. 22 
E. 33
APPENDIX: 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
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.bbbbbbbbbb b b b b b b b b b b b b b b b b

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.bbbbbbbbbbbbb b b b b b b b b b b b b b b b b b b b b b b b b
When was floating-point invented?

mantissa (noun): decimal part of a logarithm, 1865, \( \Rightarrow \) Answer: long before computers!

from Latin mantissa “a worthless addition, makeweight”
Floating Point Example

Sign (1 bit):
• 1 ⇒ negative

Exponent (8 bits):
• $10000011_2 = 131$
• $131 - 127 = 4$

Mantissa (23 bits):
• $1.10110110000000000000000_2$
• $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5}) +
  (1*2^{-6})+(1*2^{-7}) +(0*2^{-\cdots}) = 1.7109375$

Number:
• $-1.7109375 * 2^4 = -27.375$
Floating Point Consequences

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

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

For double: $\epsilon \approx 2 \times 10^{-16}$
- 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

<table>
<thead>
<tr>
<th>Decimal Approx</th>
<th>Rational Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3</td>
<td>3/10</td>
</tr>
<tr>
<td>.33</td>
<td>33/100</td>
</tr>
<tr>
<td>.333</td>
<td>333/1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Approx</th>
<th>Rational Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0/2</td>
</tr>
<tr>
<td>0.01</td>
<td>1/4</td>
</tr>
<tr>
<td>0.010</td>
<td>2/8</td>
</tr>
<tr>
<td>0.0011</td>
<td>3/16</td>
</tr>
<tr>
<td>0.00110</td>
<td>6/32</td>
</tr>
<tr>
<td>0.001101</td>
<td>13/64</td>
</tr>
<tr>
<td>0.0011010</td>
<td>26/128</td>
</tr>
<tr>
<td>0.00110011</td>
<td>51/256</td>
</tr>
</tbody>
</table>
What does the following code print?

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

```c
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");
```

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.