CS107 Winter 2020, Lecture 3
Bits and Bytes; Bitwise Operators

reading:
Bryant & O’Hallaron, Ch. 2.1
Announcements

• Assignment 0 deadline tonight at 11:59PM PST
• Assignment 1 (Bit operations!) goes (or rather, went) out today
  • Saturated arithmetic
  • Cell Automata
  • Unicode and UTF-8
• Labs start this week!
Plan For Today

• Bitwise Operators and Masks
• Demo 1: Fun With Bits
• Demo 2: Powers of 2
• Bit Shift Operators
• Demo 3: Color Wheel
Plan For Today

- Bitwise Operators and Masks
- Demo 1: Fun With Bits
- Demo 2: Powers of 2
- Bit Shift Operators
- Demo 3: Color Wheel
Now that we understand binary representations, how can we manipulate them at the bit level?
Bitwise Operators

• You’re already familiar with many operators in C:
  • Arithmetic operators: +, -, *, /, %
  • Comparison operators: ==, !=, <, >, <=, >=
  • Logical Operators: &&, ||, !
• Today, we’re introducing a new category of operators: **bitwise operators:**
  • &, |, ~, ^, <<, >>
And (&)

& is a binary operator. The & of 2 bits is 1 if both bits are 1, and 0 otherwise.

\[
\text{output} = a \& b;
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
| is a binary operator. The | of 2 bits is 1 if either (or both) bits is 1.

\[
\text{output} = a \mid b;
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Not (~)

~ is a unary operator. The ~ of a bit is 1 if the bit is 0, or 1 otherwise.

output = ~a;

<table>
<thead>
<tr>
<th>a</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exclusive Or (^)

^ is a binary operator. The ^ of 2 bits is 1 iff *exactly* one of the bits is 1.

\[
\text{output} = a \ ^ \ b;
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Operators on Multiple Bits

When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

- **AND**
  - 0110 & 1100 = 0100

- **OR**
  - 0110 | 1100 = 1110

- **XOR**
  - 0110 ^ 1100 = 1010

- **NOT**
  - ~ 1100 = 0011

Note: these are different from the logical operators AND (&&), OR (||) and NOT (!).
Operators on Multiple Bits

• When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110 &amp; 1100 ---- 0100</td>
<td>0110</td>
<td>0110 ^ 1100 ---- 1010</td>
<td>~ 1100 ---- 0011</td>
</tr>
</tbody>
</table>

This is different from logical AND (&&). The logical AND returns true if both are nonzero, or false otherwise.
Operators on Multiple Bits

- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

\[
\begin{align*}
\text{AND} & : 0110 \& 1100 = 0100 \\
\text{OR} & : 0110 \mid 1100 = 1110 \\
\text{XOR} & : 0110 \^ 1100 = 1010 \\
\text{NOT} & : \sim 1100 = 0011
\end{align*}
\]

This is different from logical NOT (!). The logical NOT returns true if this is zero, and false otherwise.
Bit Vectors and Sets

- We can use bit vectors (ordered collections of bits) to represent finite sets, and perform functions such as union, intersection, and complement.

- **Example:** we can represent current courses taken using a `char`.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS161</td>
<td>CS109</td>
<td>CS103</td>
<td>CS110</td>
<td>CS107</td>
<td>CS106X</td>
<td>CS106B</td>
<td>CS106A</td>
</tr>
</tbody>
</table>
• How do we find the union of two sets of courses taken? Use |:

```
00100011
| 01100001
```

```
01100011
```
Bit Vectors and Sets

How do we find the intersection of two sets of courses taken? Use &:

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

CS161  |  CS109  |  CS103  |  CS110  |  CS107  |  CS106X |  CS106B |  CS106A

- 00100011
- 01100001
- 00100001
Bit Masking

• We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A **bitmask** is a constructed bit pattern that we can use, along with bit operators, to do this.

• **Example:** how do we update our bit vector to indicate we’ve taken CS107?

```
0 0 1 0 0 0 0 1 1
```

00100011

```
| 00001000
-------
```

00101011
Bit Masking

```c
#define CS106A 0x1 /* 0000 0001 */
#define CS106B 0x2 /* 0000 0010 */
#define CS106X 0x4 /* 0000 0100 */
#define CS107 0x8 /* 0000 1000 */
#define CS110 0x10 /* 0000 1000 */
#define CS103 0x20 /* 0001 0000 */
#define CS109 0x40 /* 0010 0000 */
#define CS161 0x80 /* 0100 0000 */

char myClasses = ...;
myClasses = myClasses | CS107; // Add CS107
```
#define CS106A 0x1 /* 0000 0001 */
#define CS106B 0x2 /* 0000 0010 */
#define CS106X 0x4 /* 0000 0100 */
#define CS107 0x8 /* 0000 1000 */
#define CS110 0x10 /* 0000 1000 */
#define CS103 0x20 /* 0001 0000 */
#define CS109 0x40 /* 0010 0000 */
#define CS161 0x80 /* 0100 0000 */

char myClasses = ...;
myClasses |= CS107;  // Add CS107
Bit Masking

**Example:** how do we update our bit vector to indicate we’ve *not* taken CS103?

```plaintext
00100011 & 11011111 ---- ---- 00000011
```

```
char myClasses = ...;
myClasses = myClasses & ~CS103;  // Remove CS103
```
**Bit Masking**

- **Example:** how do we update our bit vector to indicate we’ve *not* taken CS103?

<table>
<thead>
<tr>
<th></th>
<th>CS161</th>
<th>CS109</th>
<th>CS103</th>
<th>CS110</th>
<th>CS107</th>
<th>CS106X</th>
<th>CS106B</th>
<th>CS106A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

00100011

\& 11011111

----------

00000011

```cpp
char myClasses = ...;
myClasses &= ~CS103;  // Remove CS103
```
### Bit Masking

**Example:** how do we check if we’ve taken CS106B?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CS161</td>
<td>CS109</td>
<td>CS103</td>
<td>CS110</td>
<td>CS107</td>
<td>CS106X</td>
<td>CS106B</td>
<td>CS106A</td>
<td></td>
</tr>
</tbody>
</table>

```
00100011
& 00000010
-----
00000010
```

cchar myClasses = ...;
if (myClasses & CS106B) {...
    // taken CS106B!
```
Bit Masking

**Example:** how do we check if we’ve *not* taken CS107?

```
char myClasses = ...;
if (!(myClasses & CS107)) {
    // not taken CS107!
```
Bit Masking

- **Example:** how do we check if we’ve *not* taken CS107?

```
char myClasses = ...;
if (((myClasses & CS107) ^ CS107) {...
  // not taken CS107!
```
Demo: Fun with Bits in gdb
Bit Masking

Bit masking is also useful for integer representations as well. For instance, we might want to check the value of the most-significant bit, or just one of the middle bytes.

• Example: If I have a 32-bit integer $j$, what operation should I perform if I want to extract just the lowest byte in $j$?

```c
int j = ...;
int k = j & 0xff; // mask to get just lowest byte
```
Practice: Bit Masking

- **Practice 1:** write an expression that, given a 32-bit integer $j$, sets its least-significant byte to all 1s, but preserves all other bytes.
  
  $j \mid 0xff$

- **Practice 2:** write an expression that, given a 32-bit integer $j$, flips ("complements") all but the least-significant byte, and preserves all other bytes.
  
  $j \ ^\^ \sim 0xff$
Without using loops, how can we detect if a binary number is a power of 2? What is special about its binary representation and how can we leverage that?
Demo: Powers of 2
Plan For Today

- Bitwise Operators and Masks
- **Demo 1:** Courses
- **Demo 2:** Powers of 2
- **Bit Shift Operators**
The LEFT SHIFT operator shifts a bit pattern a certain number of positions to the left. New lower order bits are filled in with 0s, and bits shifted off the end are lost.

\[ x \ll k; \quad \text{// shifts } x \text{ to the left by } k \text{ bits} \]

8-bit examples:

- \(00110111 \ll 2\) results in \(11011100\)
- \(01100011 \ll 4\) results in \(00110000\)
- \(10010101 \ll 4\) results in \(01010000\)
Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```c
x >> k;  // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Idea:** let’s follow left-shift and fill with 0s.

```c
short x = 2;         // 0000 0000 0000 0010
int y = x >> 1;      // 0000 0000 0000 0001
printf("%d\n", y);   // 1
```
Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;    // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Idea:** let’s follow left-shift and fill with 0s.

```
short x = -2;       // 1111 1111 1111 1110
int y = x >> 1;     // 0111 1111 1111 1111
printf("%d\n", y);  // 32767!
```
Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Problem:** always filling with zeros means we may change the sign bit.

**Solution:** let’s fill with the sign bit!
Right Shift (\texttt{>>})

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

\[
x \gg k; \quad // \text{shifts } x \text{ to the right by } k \text{ bits}
\]

\textbf{Question}: how should we fill in new higher-order bits?

\textbf{Solution}: let’s fill with the sign bit!

\begin{verbatim}
short x = 2; \quad // 0000 0000 0000 0010
int y = x >> 1; \quad // 0000 0000 0000 0001
printf("%d\n", y); \quad // 1
\end{verbatim}
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

```c
x >> k;    // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Solution:** let’s fill with the sign bit!

```c
short x = -2;    // 1111 1111 1111 1110
int y = x >> 1;  // 1111 1111 1111 1111
printf("%d\n", y);    // -1!
```
There are two kinds of right shifts, depending on the value and type you are shifting:

• **Logical Right Shift:** fill new high-order bits with 0s.

• **Arithmetic Right Shift:** fill new high-order bits with the most-significant bit.

*Unsigned numbers* are right-shifted using **Logical Right Shift**.

*Signed numbers* are right-shifted using **Arithmetic Right Shift**.

This way, the sign of the number (if applicable) is preserved!
1. *Technically*, the C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. However, almost all compilers/machines use arithmetic, and you can most likely assume this.

2. Operator precedence can be tricky! For example:

\[ 1 \ll 2 + 3 \ll 4 \text{ means } 1 \ll (2 + 3) \ll 4 \] because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:

\[ (1 \ll 2) + (3 \ll 4) \]
Shift Operator Pitfalls

• The default type of a number literal in your code is an `int`.
• Let’s say you want a long with the index-32 bit as 1:

```java
long num = 1 << 32;
```

• This doesn’t work! 1 is by default an `int`, and you can’t shift an int by 32 because it only has 32 bits. You must specify that you want 1 to be a `long`.

```java
long num = 1L << 32;
```
Demo: Color Wheel
Recap

• Bitwise Operators and Masks
• **Demo 1:** Fun With Bits
• **Demo 2:** Powers of 2
• Bit Shift Operators
• **Demo 3:** Color Wheel

**Next time:** *How can a computer represent and manipulate more complex data like text?*
Bonus Demo: Bitwise $abs$