#### CS107 Lecture 3 Byte Ordering & Bitwise Operators

reading: Bryant & O'Hallaron, Ch. 2.1

This document is copyright (C) Stanford Computer Science, Lisa Yan, and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved. Based on slides created by Marty Stepp, Cynthia Lee, Chris Gregg, Jerry Cain and others.

### Announcements

- Assign 0 due late today
- Lecture attendance 6/26 posted, please confirm
- Assign 1 out and due 7/3
- Assignment 1 IntelliCopilot Assistant Posted
- Office Hours calendar up
- Lab enrollment due today, labs start next week

## **Practice: Two's Complement**

Fill in the below table:

It's easier to compute base-10 for positive numbers, so use two's complement first if negative.

decimal binary decimal binary	
1. 0b1111 1100	
2. 0b0001 1000	
3. 0b0010 0100	
4. 0b1101 1111	



### **Expanding Bit Representations**

- Sometimes, we need to convert between two integers of different sizes (e.g. short to int, or int to long).
- We might not be able to convert from a bigger data type to a smaller data type and retain all information, but we should always be able to convert from a **smaller** data type to a **larger** data type.
- For **unsigned** values, we can prepend *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension")
- Note: when doing <, >, <=, >= comparison between different size types, it
  will promote the smaller type to the larger one.



#### **Expanding Bit Representation**



### **Expanding Bit Representation**

```
short s = 4;
// short is a 16-bit format, so
                            s = 0000 0000 0000 0100b
int i = s;
— or —
short s = -4;
// short is a 16-bit format, so
                           s = 1111 1111 1111 1100b
int i = s;
```



### **Truncating Bit Representation**

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit int), 53191:

#### 0000 0000 0000 0000 1100 1111 1100 0111

When we cast x to a short, it only has 16-bits, and C *truncates* the number: **1100 1111 1100 0111** 

This is -12345! And when we cast sx back an int, we sign-extend the number. **1111 1111 1111 1111 1100 1111 1100 0111** // still -12345



### **Truncating Bit Representation**

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

int x = -3;short sx = x;int y = sx;

What happens here? Let's look at the bits in x (a 32-bit int), -3:

#### 1111 1111 1111 1111 1111 1111 1111 1101

When we cast x to a short, it only has 16-bits, and C truncates the number:

1111 1111 1111 1101



### **Truncating Bit Representation**

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit unsigned int), 128000: **0000 0000 0001 1111 0100 0000 0000** When we cast x to a short, it only has 16-bits, and C *truncates* the number: **1111 0100 0000 0000** 



# Now that we understand values are really stored in binary, 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 (&)

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

#### output = a & b;

а	b	output
0	0	0
0	1	0
1	0	0
1	1	1

& with 1 to let a bit through, & with 0 to zero out a bit

# **Or (|)**

OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1.

outp	ut = a	b;
а	b	output
0	0	0
0	1	1
1	0	1
1	1	1

with 1 to turn on a bit, | with 0 to let a bit go through

## Not (~)

NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise.

output	= ~a;
а	output
0	1
1	0

## Exclusive Or (^)

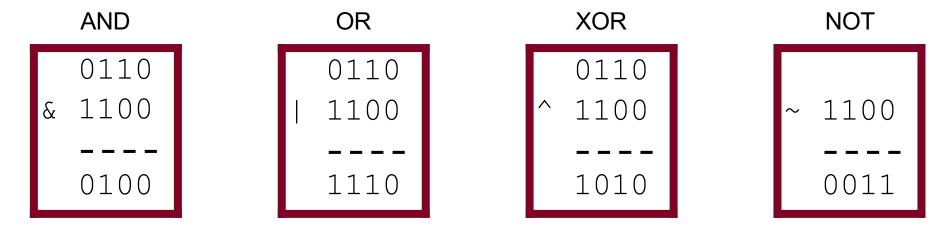
Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if *exactly* one of the bits is 1, or 0 otherwise.

oucp		U y
а	b	output
0	0	0
0	1	1
1	0	1
1	1	0

output = a ^ b;

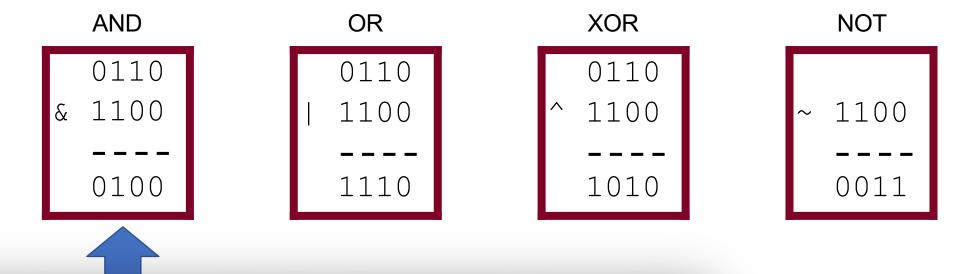
^ with 1 to flip a bit, ^ with 0 to let a bit go through

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



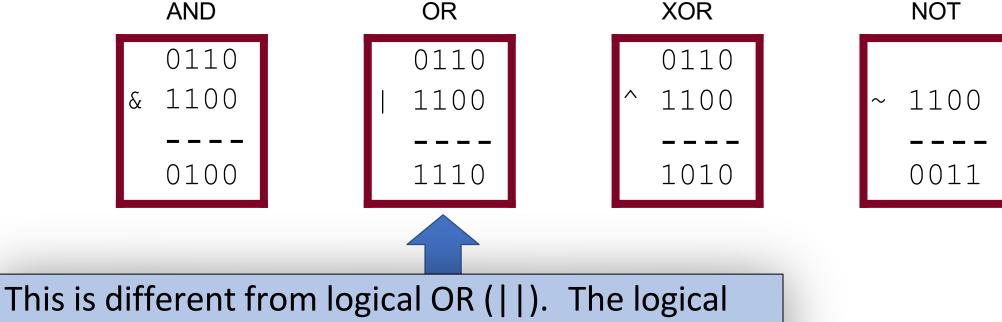
**Note:** these are different from the logical operators AND (&&), OR (||) and NOT (!).

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



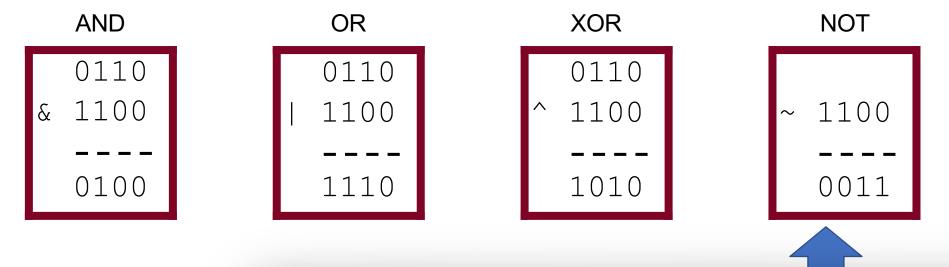
This is different from logical AND (&&). The logical AND returns true if both are nonzero, or false otherwise. With &&, this would be 6 && 12, which would evaluate to **true** (1).

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



OR returns true if either are nonzero, or false otherwise. With ||, this would be 6 || 12, which would evaluate to **true** (1).

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



This is different from logical NOT (!). The logical NOT returns true if this is zero, and false otherwise. With !, this would be !12, which would evaluate to **false** (0).

## **Demo: Bits Playground**



## Bitmasks

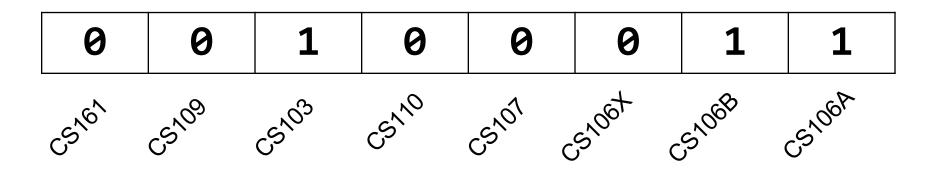
We will frequently want to manipulate or otherwise isolate specific bits in a larger collection of them. A **bitmask** is a constructed bit pattern that we can use, along with standard bit operators like &, |,  $^, ~, ~, <<$ , and >>, to do this.

#### Motivating Example: Bit vectors

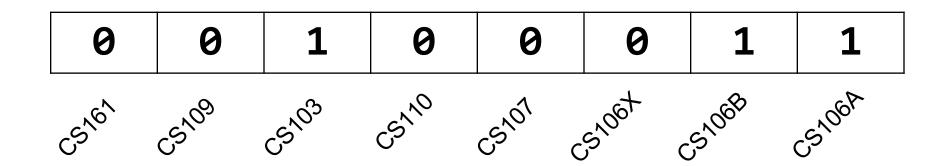
**Aside:** C++ relies on bit vectors to efficiently implement **vector<bool>**.

## **Bit Vectors and Sets**

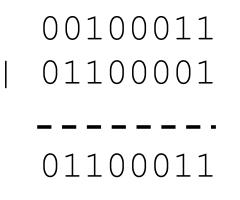
- We can use bit vectors (ordered collections of bits) to represent finite sets, and perform functions such as union, intersection, and complement.
- Example: we can represent current courses taken using a char.



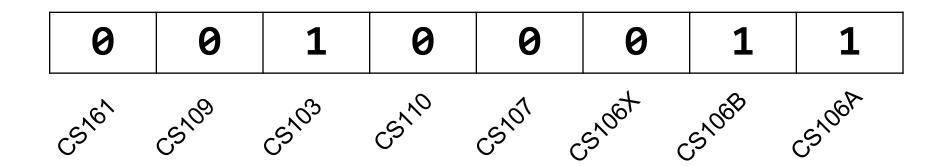
### **Bit Vectors and Sets**



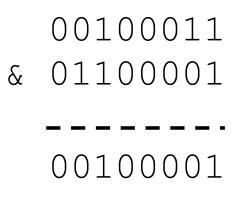
• How do we find the union of two sets of courses taken? Use OR:



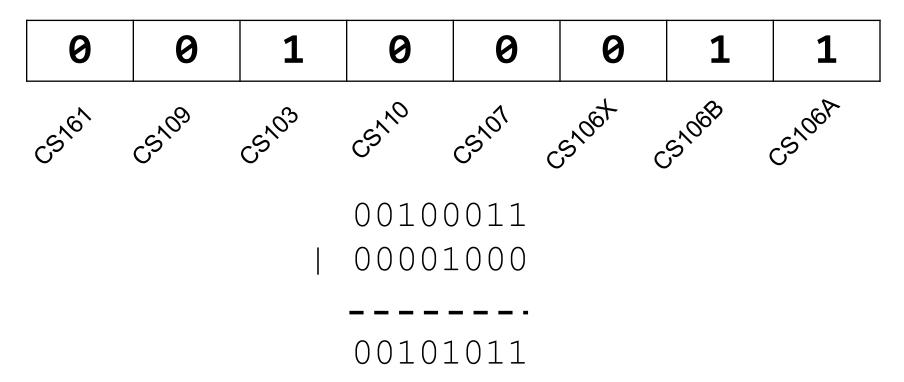
### **Bit Vectors and Sets**



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



- 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?



#define CS106A 0x1 / #define CS106B 0x2 / #define CS106X 0x4 / #define CS107 0x8 / #define CS110 0x10 / #define CS103 0x20 / #define CS109 0x40 / #define CS161 0x80 /

/*	0000	0001	*/
/*	0000	0010	*/
/*	0000	0100	*/
/*	0000	1000	*/
/*	0001	0000	*/
/*	0010	0000	*/
/*	0100	0000	*/
/*	1000	0000	*/

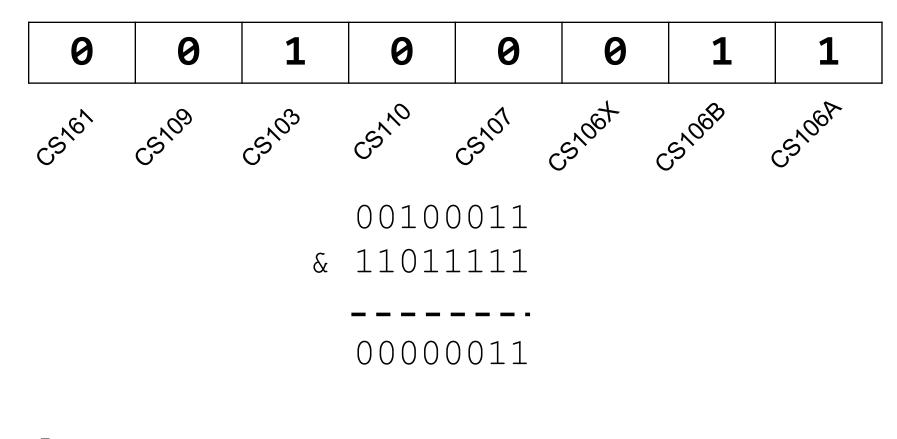
char myClasses = ...;
myClasses = myClasses | CS107; // Add CS107

#define CS106A 0x1
#define CS106B 0x2
#define CS106X 0x4
#define CS107 0x8
#define CS100 0x10
#define CS103 0x20
#define CS109 0x40
#define CS161 0x80

/*	0000	0001	*/
/*	0000	0010	*/
/*	0000	0100	*/
/*	0000	1000	*/
/*	0001	0000	*/
/*	0010	0000	*/
/*	0100	0000	*/
/*	1000	0000	*/

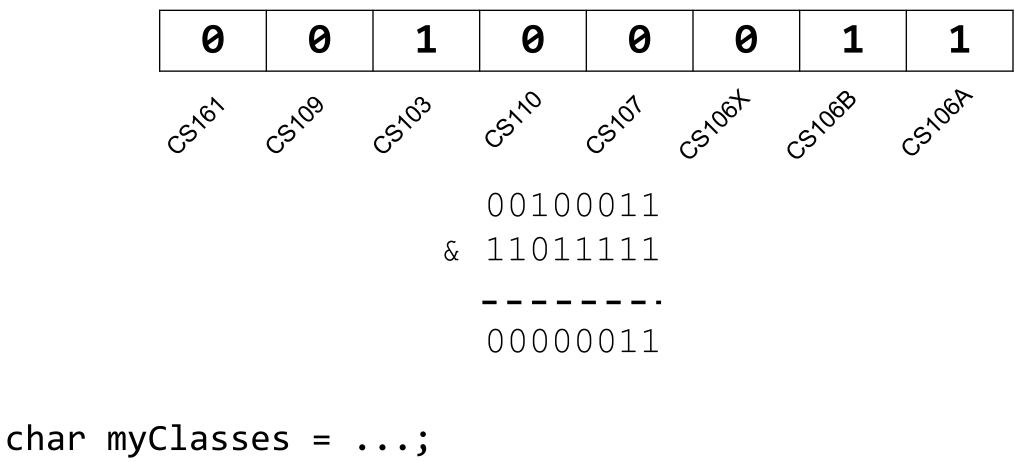
```
char myClasses = ...;
myClasses |= CS107; // Add CS107
```

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



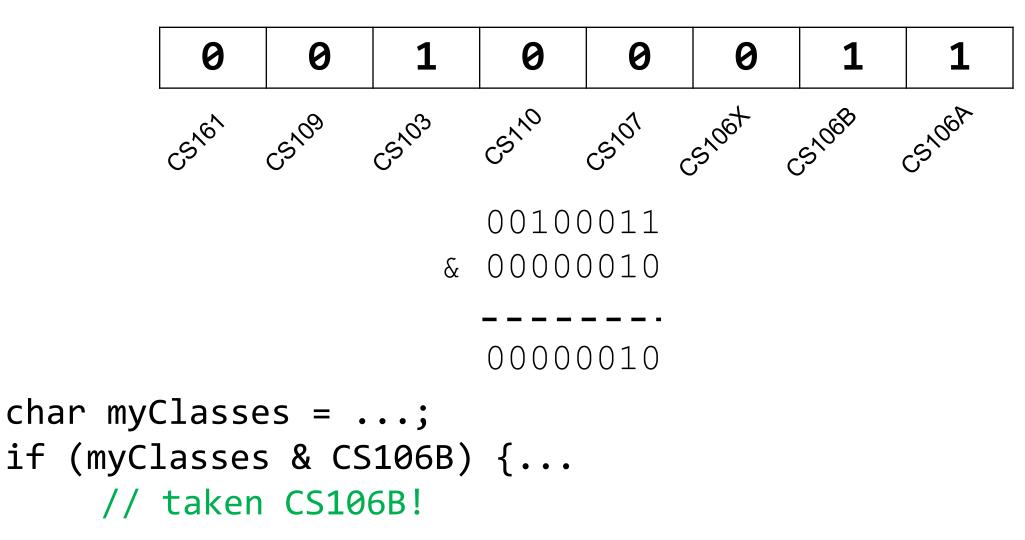
char myClasses = ...;
myClasses = myClasses & ~CS103; // Remove CS103

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

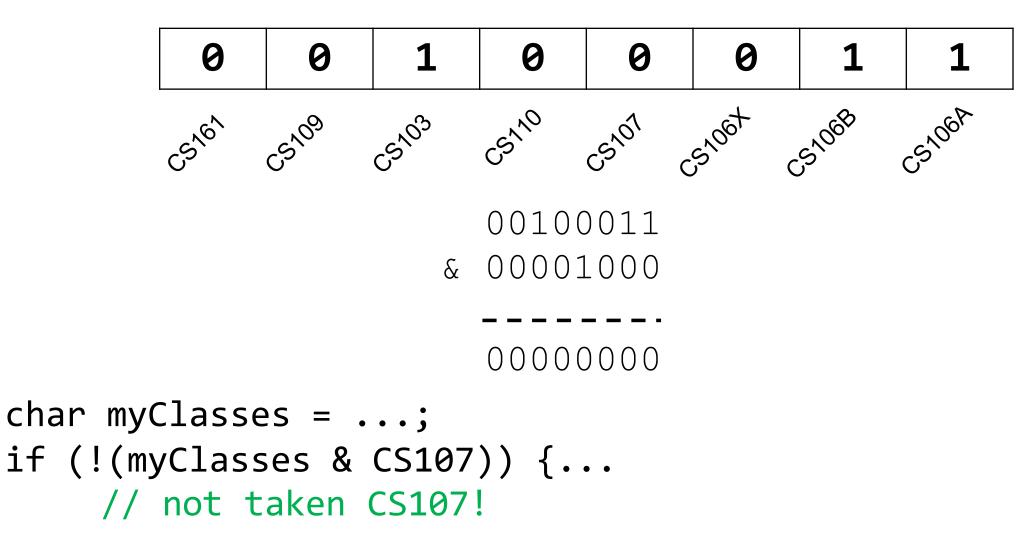


myClasses &= ~CS103; // Remove CS103

• Example: how do we check if we've taken CS106B?



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



## **Bitwise Operator Tricks**

- | with 1 is useful for turning select bits on
- & with 0 is useful for turning select bits off
- | is useful for taking the union of bits
- & is useful for taking the intersection of bits
- ^ is useful for flipping isolated bits
- ~ is useful for flipping all bits

## **Introducing GDB**

# Is there a way to step through the execution of a program and print out values as it's running? e.g., to view binary representations? **Yes!**

## **The GDB Debugger**

- GDB is a **command-line debugger**, a text-based debugger with similar functionality to other debuggers you may have used, such as in Qt Creator
- It lets you put **breakpoints** at specific places in your program to pause there
- It lets you step through execution line by line
- It lets you print out values of variables in various ways (including binary)
- It lets you track down where your program crashed
- And much, much more!

GDB is essential to your success in CS107 this quarter! We'll be building our familiarity with GDB over the course of the quarter.

## **GDB** as an **Interpreter**

- gdb live\_session
   run gdb on live\_session executable
- p print variable (p varname) or evaluated expression (p 3L << 10)
  - p/t, p/x binary and hex formats.
  - p/d, p/u, p/c
- <enter>

Execute last command again Quit gdb

• q

**Important** When first launching gdb:

- Gdb is not running any program and therefore can't print variables
- It can still process operators on constants

## gdb on a program

- gdb live\_session
   b
   Set breakpoint on a function (e.g., b main) or line (b 42)
  - Run with provided args
- n, s, continue control forward execution (next, step into, continue)
- p print variable (p varname) or evaluated expression (p 3L << 10)
  - p/t, p/x binary and hex formats.
  - p/d, p/u, p/c

•r 82

• info args, locals

**Important**: gdb does not run the current line until you hit "next"

# **Demo: Bitmasks and GDB**



## gdb: highly recommended

At this point, setting breakpoints/stepping in gdb may seem like overkill for what could otherwise be achieved by copious **printf** statements.

However, gdb is incredibly useful for **assign1** (and all assignments):

- A fast "C interpreter": p + <expression>
  - Sandbox/try out ideas around bitshift operators, signed/unsigned types, etc.
  - Can print values out in binary!
  - Once you're happy, then make changes to your C file
- Tip: Open two terminal windows and SSH into myth in both
  - Keep one for emacs, the other for gdb/command-line
  - Easily reference C file line numbers and variables while accessing gdb
- **Tip**: Every time you update your C file, **make** and then rerun gdb.

Gdb takes practice! But the payoff is tremendous! ©

#### gdb step, next, finish

I've seen a few students who have been frustrated with stepping through functions in gdb. Sometimes, they will accidentally step into a function like strlen or printf and get stuck.

There are three important gdb commands about stepping through a program:

**step** (abbreviation: s) : executes the next line and *goes into* function calls.

**next** (abbreviation: n) : executes the next line, and *does not go into function calls.* I.e., if you want to run a line with strlen or printf but don't want to attempt to go into that function, use **next**.

display (abbreviation: disp) : displays a variable (or other item) after each step.

finish (abbreviation: fin) : completes a function and returns to the calling function. This is the command you want if you accidentally go into a function like strlen or printf! This continues the program until the end of the function, putting you back into the calling function.

#### **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 get *just the lowest byte* in **j**?

#### **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.

**Practice 2:** write an expression that, given a 32-bit integer j, flips ("complements") all but the least-significant byte, and preserves the last 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 | 0xff

**Practice 2:** write an expression that, given a 32-bit integer j, flips ("complements") all but the least-significant byte, and preserves the last byte.

#### **Powers of 2**

Without using loops, how can we detect if a number **num** is a power of 2? What's special about its binary representation and how can we take advantage of that?

# Code: Powers of 2

# bool is\_power\_of\_2(unsigned long num){ return (num != 0) && ((num & (num -1)) == 0) }

#### Left Shift (<<)

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 << k; // evaluates to x shifted to the left by k bits
x <<= k; // shifts x to the left by k bits</pre>

8-bit examples:

00110111 << 2 results in 11011100 01100011 << 4 results in 00110000 10010101 << 4 results in 01010000

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; // evaluates to x shifted to the right by k bits
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; // 0000 0000 0000 0010
x >>= 1; // 0000 0000 0000 0001
printf("%d\n", x); // 1

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; // evaluates to x shifted to the right by k
x >>= k; bit
// 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
x >>= 1; // 0111 1111 1111 1111
printf("%d\n", x); // 32767!

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

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!

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; // evaluates to x shifted to the right by k
x >>= k; bit
// 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!

short x = 2; // 0000 0000 0000 0010
x >>= 1; // 0000 0000 0000 0001
printf("%d\n", x); // 1

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; // evaluates to x shifted to the right by k
x >>= k; bit
// 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!

short x = -2; // 1111 1111 1111 1110
x >>= 1; // 1111 1111 1111 1111
printf("%d\n", x); // -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!

#### **Shift Operation Pitfalls**

- 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<<2 + 3<<4 means 1 << (2+3) << 4 because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:

(1<<2) + (3<<4)

#### **Bit 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:

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**.

long num = 1 << 32;

# **Code: Absolute Value**

```
long abs_val(long num){
    long sign = num >> sizeof(long) * CHARBIT; // gives me 64 sign bits
    return (num ^ sign) - sign;
```

#### **Bitwise Warmup**

How can we use bitmasks + bitwise operators to...

#### 0b00001101

1.	turn <b>on</b> a particular set of bits?	2.	turn <b>off</b> a particular set of bits?	Э	<ol> <li>flip a particular set of bits?</li> </ol>	
	0b00001101		0b00001101		0b00001101	
_	0b000011 <u>1</u> 1	_	0b00001 <u>0</u> 01		0b00001 <u>01</u> 1	

#### **Bitwise Warmup**

How can we use bitmasks + bitwise operators to...

#### 0b00001101

<ol> <li>turn on a particular set of bits? OR</li> </ol>	<ol> <li>turn off a particular set of bits? AND</li> </ol>	<ol> <li>flip a particular set of bits? XOR</li> </ol>
0b00001101 0b00000010	0b00001101 0b11111011 &	0b00001101 0b00000110 ^
0b000011 <u>1</u> 1	0b0001 <u>0</u> 01	0b00001 <u>01</u> 1

#### **More Exercises**

- Suppose we have a 64-bit number. long x = 0b1010010; How can we use bit operators, and the constant 1L or -1L to...
  - ...design a mask that turns on the i-th bit of a number for any i (0, 1, 2, ..., 63)?

 ...design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?



#### **More Exercises**

- Suppose we have a 64-bit number. long x = 0b1010010; How can we use bit operators, and the constant 1L or -1L to...
- ...design a mask that turns on the i-th bit of a number for any i (0, 1, 2, ..., 63)?
- x | (1L << i)
- ...design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?



- Print a variable
- Print (in binary, then in hex) result of left-shifting 14 and 32 by 4 bits.
- Print (in binary, then in hex) result of subtracting 1 from 128
- 1 << 32
- Why is this zero? Compare with 1 << 31.
- Print in hex to make it easier to count zeros.

#### **References and Advanced Reading**

#### •References:

- •Two's complement calculator: <u>http://www.convertforfree.com/twos-complement-</u> calculator/
- •Wikipedia on Two's complement: <u>https://en.wikipedia.org/wiki/</u> <u>Two%27s\_complement</u>
- The sizeof operator: <a href="http://www.geeksforgeeks.org/sizeof-operator-c/">http://www.geeksforgeeks.org/sizeof-operator-c/</a>

#### Advanced Reading:

- Signed overflow: <u>https://stackoverflow.com/questions/16056758/c-c-unsigned-integer-overflow</u>
- Integer overflow in C: <u>https://www.gnu.org/software/autoconf/manual/</u> <u>autoconf-2.62/html\_node/Integer-Overflow.html</u>
- <u>https://stackoverflow.com/questions/34885966/when-an-int-is-cast-to-a-short-and-truncated-how-is-the-new-value-determined</u>



#### **References and Advanced Reading**

#### •References:

- argc and argv: <u>http://crasseux.com/books/ctutorial/argc-and-argv.html</u>
- The C Language: <a href="https://en.wikipedia.org/wiki/C">https://en.wikipedia.org/wiki/C</a> (programming language)
- Kernighan and Ritchie (K&R) C: <u>https://www.youtube.com/watch?v=de2Hsvxaf8M</u>
- C Standard Library: http://www.cplusplus.com/reference/clibrary/
- <u>https://en.wikipedia.org/wiki/Bitwise\_operations\_in\_C</u>
- <u>http://en.cppreference.com/w/c/language/operator\_precedence</u>

#### Advanced Reading:

- After All These Years, the World is Still Powered by C Programming
- Is C Still Relevant in the 21st Century?
- Why Every Programmer Should Learn C

