CS107 Lecture 4
Bits and Bytes; Bitwise Operators

reading:

Bryant & O'Hallaron, Ch. 2.1

CS107 Topic 1

How can a computer represent integer numbers?

Why is answering this question important?

- Helps us understand the limitations of computer arithmetic (last time)
- Shows us how to more efficiently perform arithmetic (today)
- Shows us how we can encode data more compactly and efficiently (today)

assign1: implement 3 programs that manipulate binary representations to (1) work around the limitations of arithmetic with addition, (2) simulate an evolving colony of cells, and (3) print Unicode text to the terminal.

Today

Today, we'll learn about a new set of operators to manipulate bits. For example:

```
int x = 2;

// NEW: shift all bits X places to the left or right
x = x << 1; // now x is 4!

// NEW: check if the least significant bit is a 0
if (x & 1 == 0) {...
```

This is useful because we can perform some arithmetic more efficiently, and also store data more compactly in individual bits.

Learning Goals

- Learn about the bitwise C operators and how to use them to manipulate bits
- Understand when to use one bitwise operator vs. another in your program
- Get practice with writing programs that manipulate binary representations

Lecture Plan

- **Recap and continuing**: Integer Representations
- Bitwise Operators
- Bitmasks

Lecture Plan

- **Recap and continuing: Integer Representations**
- Bitwise Operators
- Bitmasks

Bits and Bytes So Far

All data, including integer variables, are ultimately stored in memory in binary:

```
int x = 5;    // really 0b0…0101 in memory!
```

- Unsigned numbers store the direct binary representation of its value
- Signed numbers use **two's complement** to store its positive/negative/0 value
- Overflow occurs when we exceed the the minimum or maximum value of the bit representation – it can cause some funky bugs!

Base 10 vs. Binary vs. Hex

- Let's take a byte (8 bits):

`165`	Base-10: Human-readable, but cannot easily interpret on/off bits
`0b10100101`	Base-2: Yes, computers use this, but not human-readable
`0xa5`	Base-16: Easy to convert to Base-2, More "portable" as a human-readable format (fun fact: a half-byte is called a nibble or nybble)

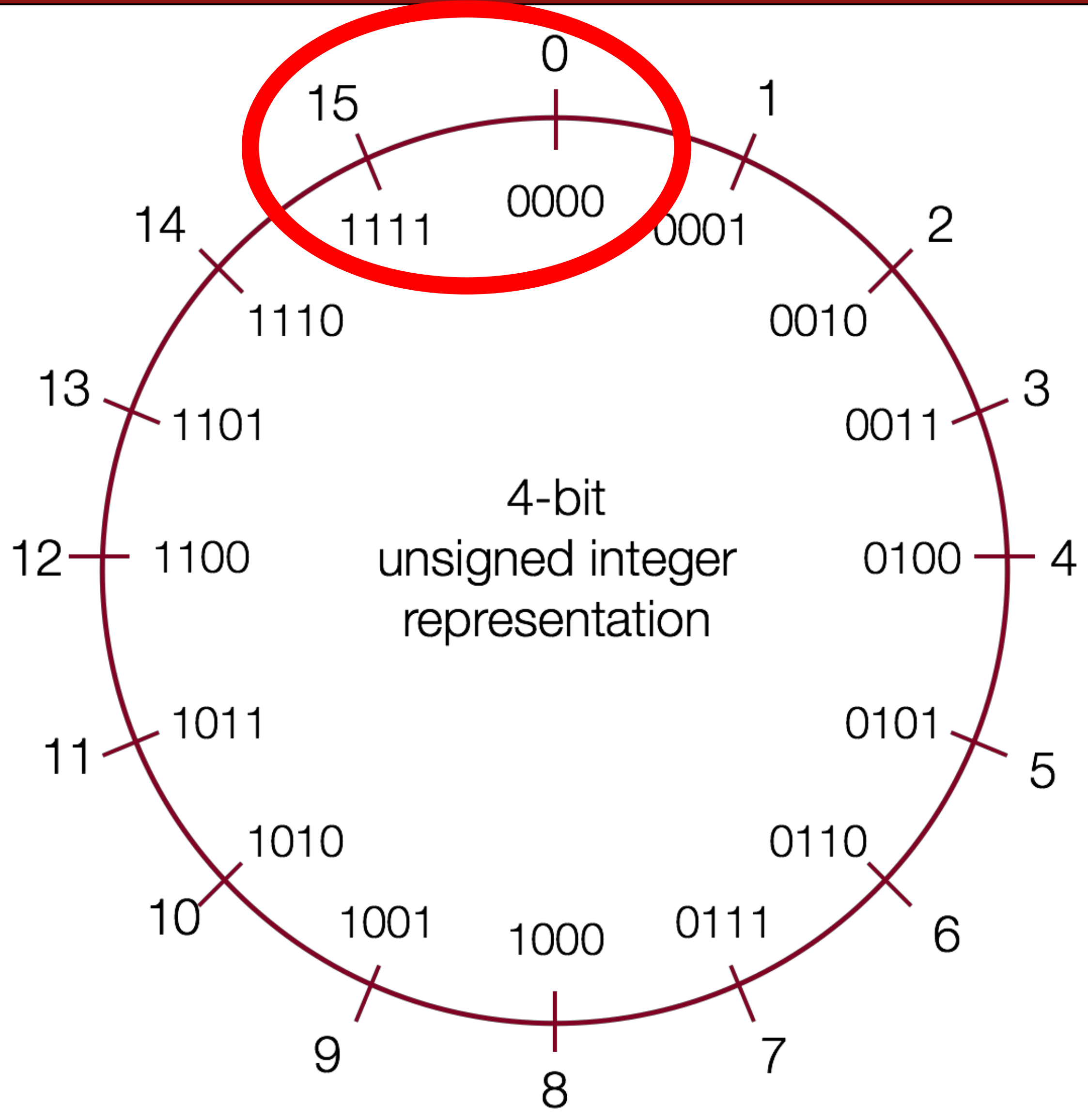
Overflow

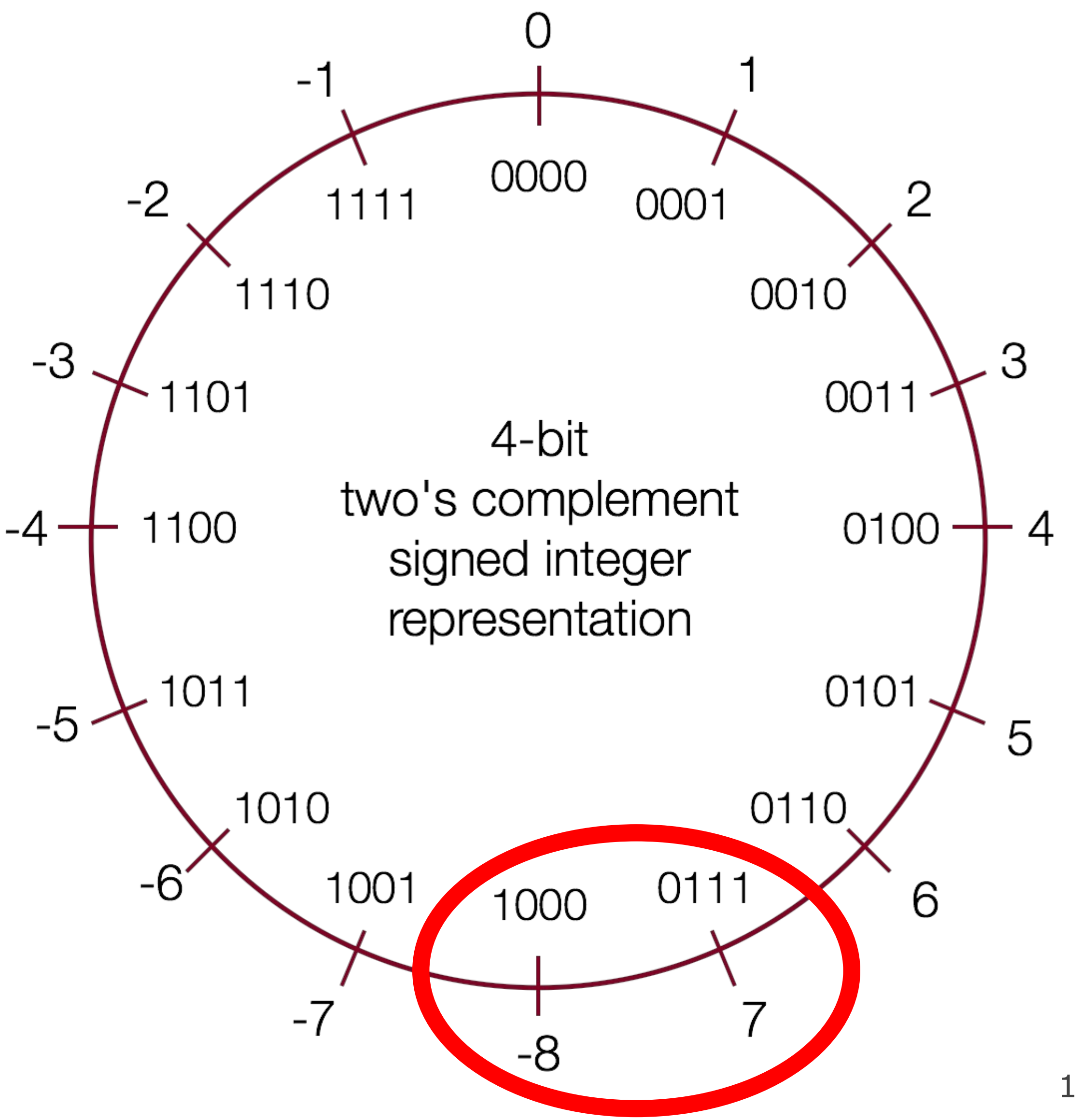
Overflow occurs because we don't have enough bits to store a value.

E.g. if we have **unsigned short x = 65535** and add 2, we get **1**!

```
   1111 1111 1111 1111
   0000 0000 0000 0010
+ ---------------------
  10000 0000 0000 0001
```

Overflow


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
unsigned integer
representation


0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
two's complement
signed integer
representation

Min and Max Integer Values

In C, there are various constants that represent these minimum and maximum values: **`INT_MIN`**, **`INT_MAX`**, **`UINT_MAX`**, **`LONG_MIN`**, **`LONG_MAX`**, **`ULONG_MAX`**, …

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; // -12345!
```

```
 x = 0000 0000 0000 0000 1100 1111 1100 0111
sx =                     1100 1111 1100 0111
```

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; // still -3
```

```
 x = 1111 1111 1111 1111 1111 1111 1111 1101
sx =                     1111 1111 1111 1101
```

Expanding Bit Representations

Sometimes, we want to carry over a value to a larger variable (e.g. make an **int** and set it equal to a **`short`**`).`

- For **unsigned** values, C adds *leading zeros* to the representation ("zero extension")
- For **signed** values, C *repeats the sign of the value* for new digits ("sign extension"

Expanding Bit Representation

If we want to **expand** the bit size of an **unsigned** number, C *adds leading zeros.*

```
unsigned short s = 4;
unsigned int i = s; // still 4
```

```
s =                     0000 0000 0000 0100
i = 0000 0000 0000 0000 0000 0000 0000 0100
```

Expanding Bit Representation

If we want to **expand** the bit size of an **signed** number, C *adds repeats the sign.*

```
short s = -4;
int i = s; // still -4
```

```
s =                     1111 1111 1111 1100
i = 1111 1111 1111 1111 1111 1111 1111 1100
```

Expanding Bit Representation

If we want to **expand** the bit size of an **signed** number, C *adds repeats the sign.*

```
short s = 4;
int i = s; // still 4
```

```
s =                     0000 0000 0000 0100
i = 0000 0000 0000 0000 0000 0000 0000 0100
```

Casting

You can cast something to another type (treat as other type temporarily) by putting that type in parentheses in front of the value:

```
short s = -12345;
...(unsigned short)s...
```

Casting between variable types can cause tricky issues; **the bits remain the same but are interpreted differently.**

Here, **s** is -12345, but casted it is 53191! (**1100 1111 1100 0111** in binary)

Casting

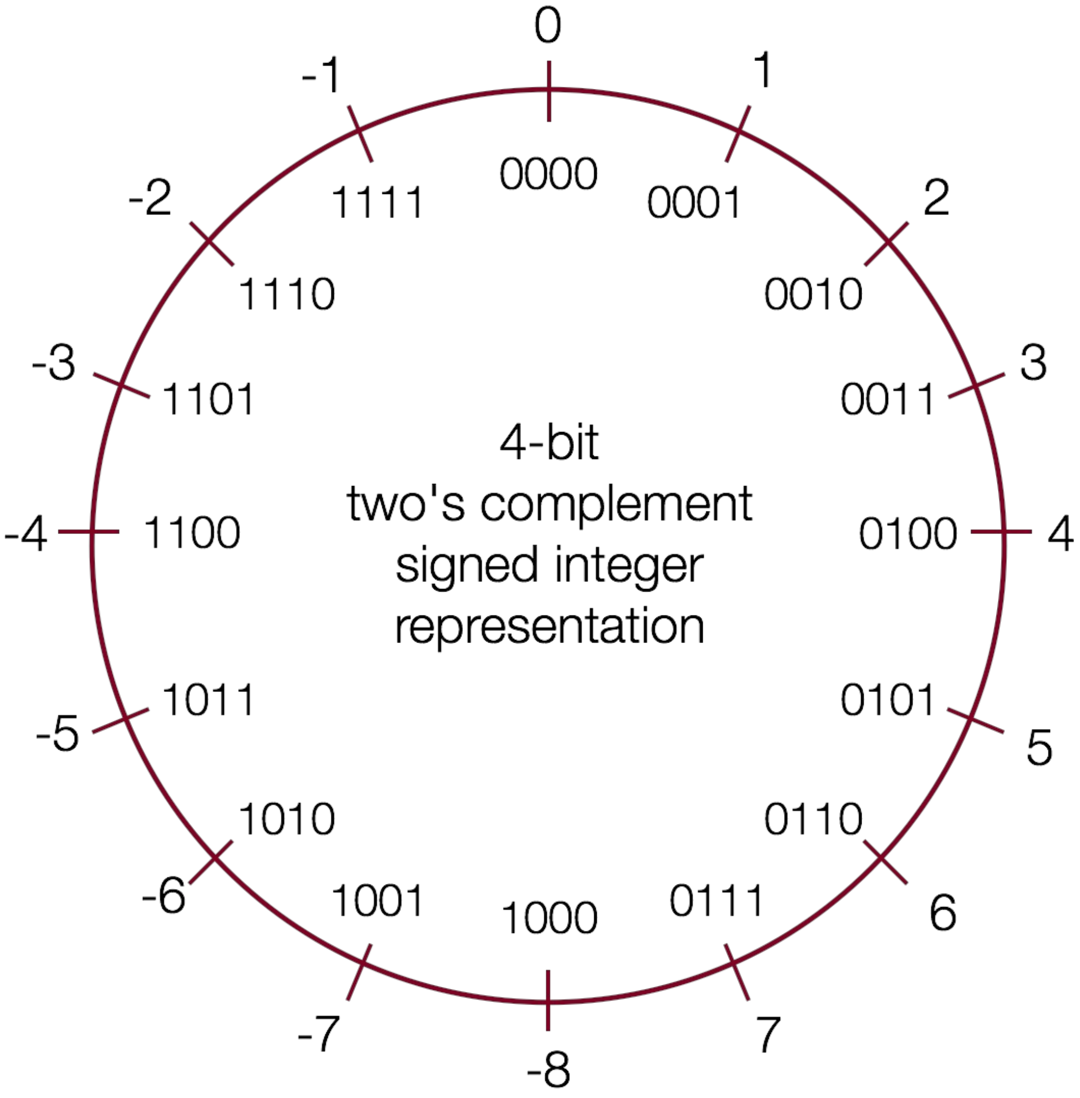
You can store the result as well:

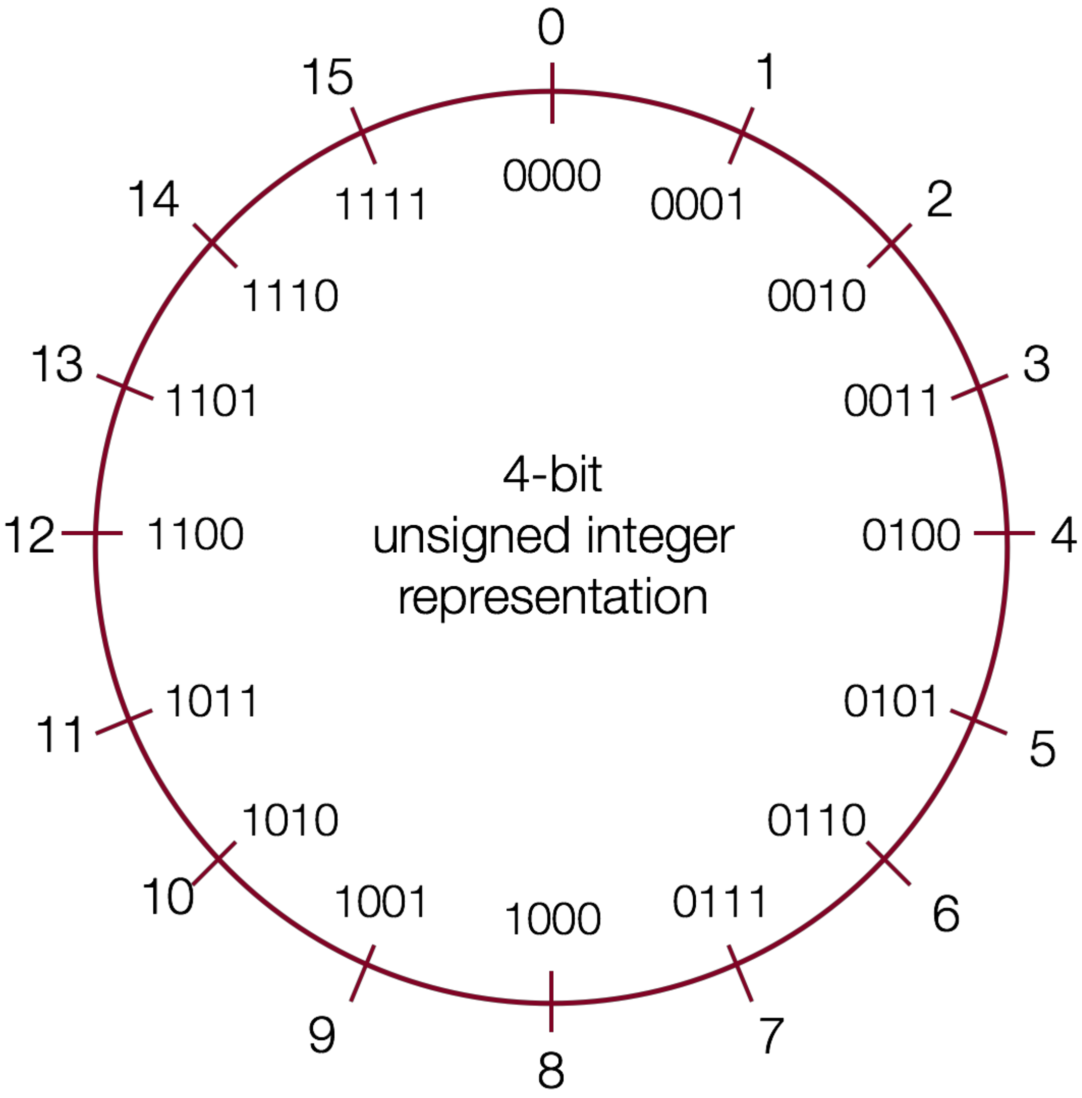
```
short s = -12345;
unsigned short us = (unsigned short)s;  // 53191!
```

You can also use the **U** suffix after a number literal to treat it as unsigned:

```
-12345U
```

Casting


4-bit
two's complement
signed integer
representation
0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-8
1000
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111


4-bit
unsigned integer
representation
0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
8
1000
9
1001
10
1010
11
1011
12
1100
13
1101
14
1110
15
1111

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

```
int x = -1;    // 1111 1111 1111 1111 1111 1111 1111 1111
unsigned int y = 0;
if (x < y) { ...  // will be false!!
```

Note: when doing <, >, <=, >= comparison between different ***size*** types, it will *promote to the larger type*.

The `sizeof` Operator

```
long sizeof(type);

    // Example
    long int_size_bytes = sizeof(int);     // 4
    long short_size_bytes = sizeof(short); // 2
    long char_size_bytes = sizeof(char);   // 1
```

`sizeof` takes a variable type (or a variable itself) as a parameter and returns the size of that type, in bytes.

Lecture Plan

- **Recap and continuing**: Integer Representations
- **Bitwise Operators**
- Bitmasks

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;
```

a	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.

```
output = a | b;
```

a	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;
```

a	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.

```
output = a ^ b;
```

a	b	output
0	0	0
0	1	1
1	0	1
1	1	0

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

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	OR	XOR	NOT
`  0110` `& 1100` `  ----` `  0100`	`  0110` `| 1100` `  ----` `  1110`	`  0110` `^ 1100` `  ----` `  1010`	`~ 1100` `  ----` `  0011`

Bit Operators

```
int x = 6;  // 0000 ... 0110
int y = 5;  // 0000 ... 0101

// 4
int anded = x & y; // 0000 ... 0100

// 7
int ored  = x | y; // 0000 ... 0111

// -7
int notX  = ~x; // 1111 ... 1111 1111 1001

int xored = x ^ y; // what would this give us?
```

If x = 6 (0110) and y = 5 (0101), what would x ^ y be?

- 0
- 1
- 2
- 3
- 4

If x = 6 (0110) and y = 5 (0101), what would x ^ y be?

0	0%
1	0%
2	0%
3	0%
4	0%

If x = 6 (0110) and y = 5 (0101), what would x ^ y be?

0	0%
1	0%
2	0%
3	0%
4	0%

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	OR	XOR	NOT
`  0110` `& 1100` `  ----` `  0100`	`  0110` `| 1100` `  ----` `  1110`	`  0110` `^ 1100` `  ----` `  1010`	`~ 1100` `  ----` `  0011`

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

Bit Operators

```
int x = 4;  // 0000 ... 0100
int y = 5;  // 0000 ... 0101

// This is checking if x and y are both nonzero
if (x && y) { …

// This is checking if the result of x & y is nonzero
if (x & y) { …
```

Lecture Plan

- **Recap and continuing**: Integer Representations
- Bitwise Operators
- **Bitmasks**
- **Demo:** Bitmasks and GDB
- More practice

Bitmasks

We will frequently want to manipulate or isolate out specific bits in a larger collection of bits.

Motivating Example: Bit vectors

Bit Vectors and Sets

Instead of using arrays of e.g., Booleans in our programs, sometimes it's beneficial to store that information in bits instead – more compact.

- **Example:** we can represent current courses taken using a **`char`** and manipulate its contents using bit operators.

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS107E 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS111  0x10   /* 0001 0000 */
#define CS103  0x20   /* 0010 0000 */
#define CS109  0x40   /* 0100 0000 */
#define CS161  0x80   /* 1000 0000 */

// Bit pattern: 0000 1011
unsigned char myClasses = CS106A | CS106B | CS107;
```

Bit Vectors and Sets

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

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

```
  00100011
| 01100001
  --------
  01100011
```

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS107E 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS111  0x10   /* 0001 0000 */
#define CS103  0x20   /* 0010 0000 */
#define CS109  0x40   /* 0100 0000 */
#define CS161  0x80   /* 1000 0000 */

unsigned char myClasses = CS106A | CS106B | CS107;
unsigned char otherClasses = CS106A | CS106B | CS103;

// 0010 1011
unsigned char either = myClasses | otherClasses;
```

Bit Vectors and Sets

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

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

```
   00100011
&  01100001
   --------
   00100001
```

Bit Masking

Example: how do we update our bit vector to indicate we've taken CS107?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

```
  00100011
| 00001000
  --------
  00101011
```

Bit Masking

Example: how do we update our bit vector to indicate we've taken CS107?

A bitmask is a constructed bit pattern that we can use, along with bit operators, to manipulate a value.

CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A
0	0	1	0	0	0	1	1

```
  00100011
| 00001000
  --------
  00101011
```

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS107E 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS111  0x10   /* 0001 0000 */
#define CS103  0x20   /* 0010 0000 */
#define CS109  0x40   /* 0100 0000 */
#define CS161  0x80   /* 1000 0000 */

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

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS107E 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS111  0x10   /* 0001 0000 */
#define CS103  0x20   /* 0010 0000 */
#define CS109  0x40   /* 0100 0000 */
#define CS161  0x80   /* 1000 0000 */

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

Bit Masking

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

CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A
0	0	1	0	0	0	1	1

```
  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?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

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

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

Bit Masking

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

CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A
0	0	1	0	0	0	1	1

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

```
char myClasses = ...;
if (myClasses & CS106B) {...
    // taken CS106B!
```

Bit Masking

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

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS107E	CS106B	CS106A

```
  00100011
& 00001000
  --------
  00000000
```

```
char myClasses = ...;
if (!(myClasses & CS107)) {...
    // 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 select bits
- ~ is useful for flipping all bits

Introducing GDB

Is there a way to step through the execution of a program and print out its 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.

More next time…

Recap

- **Recap and continuing**: Integer Representations
- Bitwise Operators
- Bitmasks

Lecture 4 takeaways: We can use bit operators like &, |, ~, etc. to manipulate the binary representation of values. A number is a bit pattern that can be manipulated arithmetically or bitwise at your convenience!

Extra Practice

Hexadecimal and Truncation

For each initialization of x, what will be printed?

i. `x = 130;  // 0x82`

ii. `x = -132; // 0xff7c`

iii. `x = 25;   // 0x19`

```
short x = ______;
char cx = x;
printf("%d", cx);
```

Hexadecimal and Truncation

For each initialization of x, what will be printed?

-126 i. `x = 130;  // 0x82`

124 ii. `x = -132; // 0xff7c`

25 iii. `x = 25;   // 0x19`

```
short x = ______;
char cx = x;
printf("%d", cx);
```

Limits and Comparisons

2. Will the following `char` comparisons evaluate to `true` or `false`?
   1. `-7 < 4` **true**
   2. `-7 < 4U` **false**
   3. `(char) 130 > 4` **false**
   4. `(char) -132 > 2` **true**

By default, numeric constants in C are signed ints, unless they are suffixed with u (unsigned) or L (long).

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

0b00001101

1. …turn **on** a particular set of bits?

   0b00001101

   ______

   0b00001111

2. …turn **off** a particular set of bits?

   0b00001101

   ______

   0b00001001

3. …flip a particular set of bits?

   0b00001101

   ______

   0b00001011

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

`0b00001101`

1. …turn **on** a particular set of bits? **OR**

```
0b00001101
0b00000010 |
-----------
0b00001111
```

2. …turn **off** a particular set of bits? **AND**

```
0b00001101
0b11111011 &
-----------
0b00001001
```

3. …**flip** a particular set of bits? **XOR**

```
0b00001101
0b00000110 ^
-----------
0b00001011
```