photo by unsplash user @swimstaralex

COMP201

Computer Systems & Programming

Lecture #03 –Bits and Bitwise Operators, Floating Point

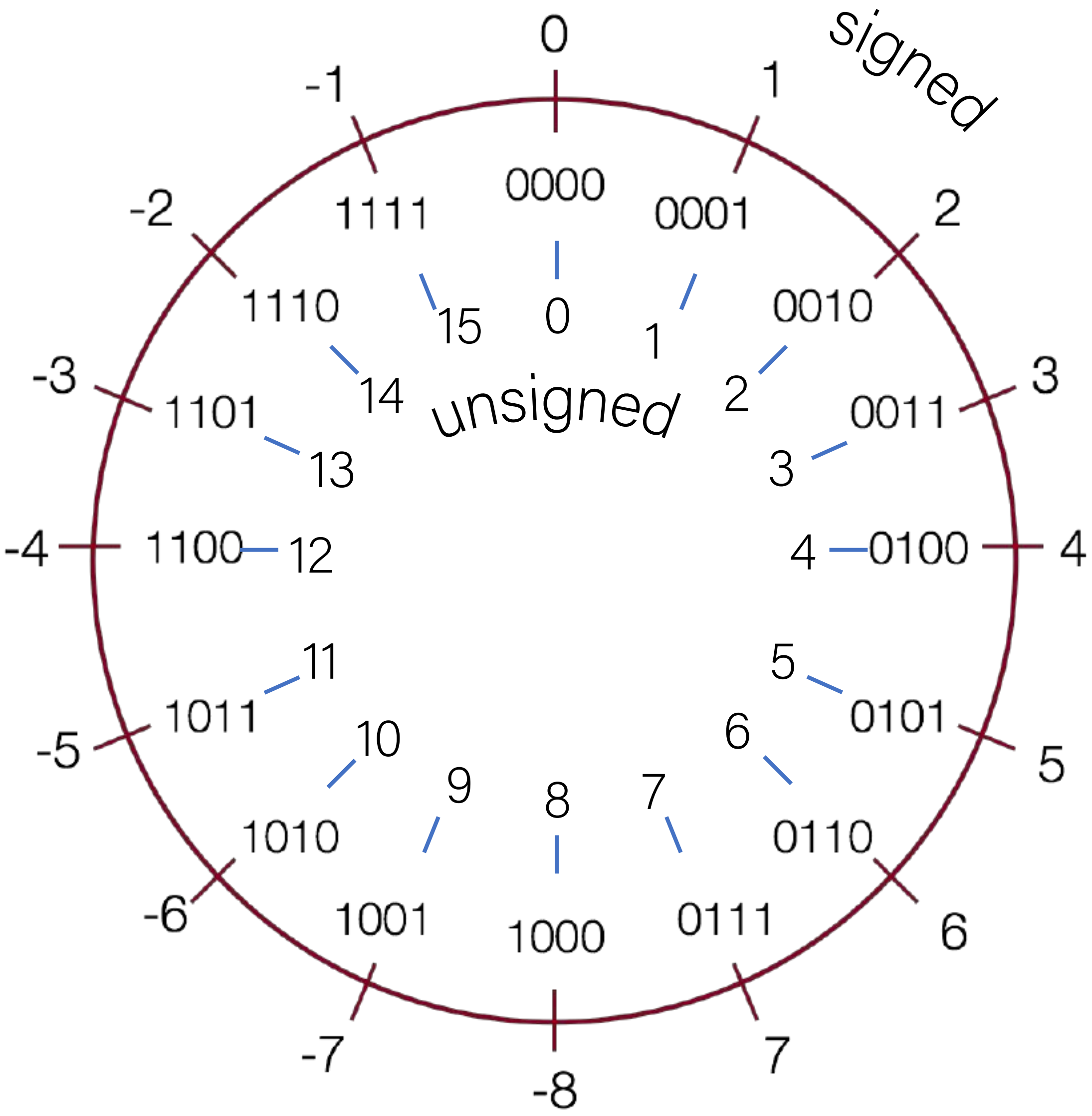
KOÇ UNIVERSITY

Aykut Erdem // Koç University // Spring 2021

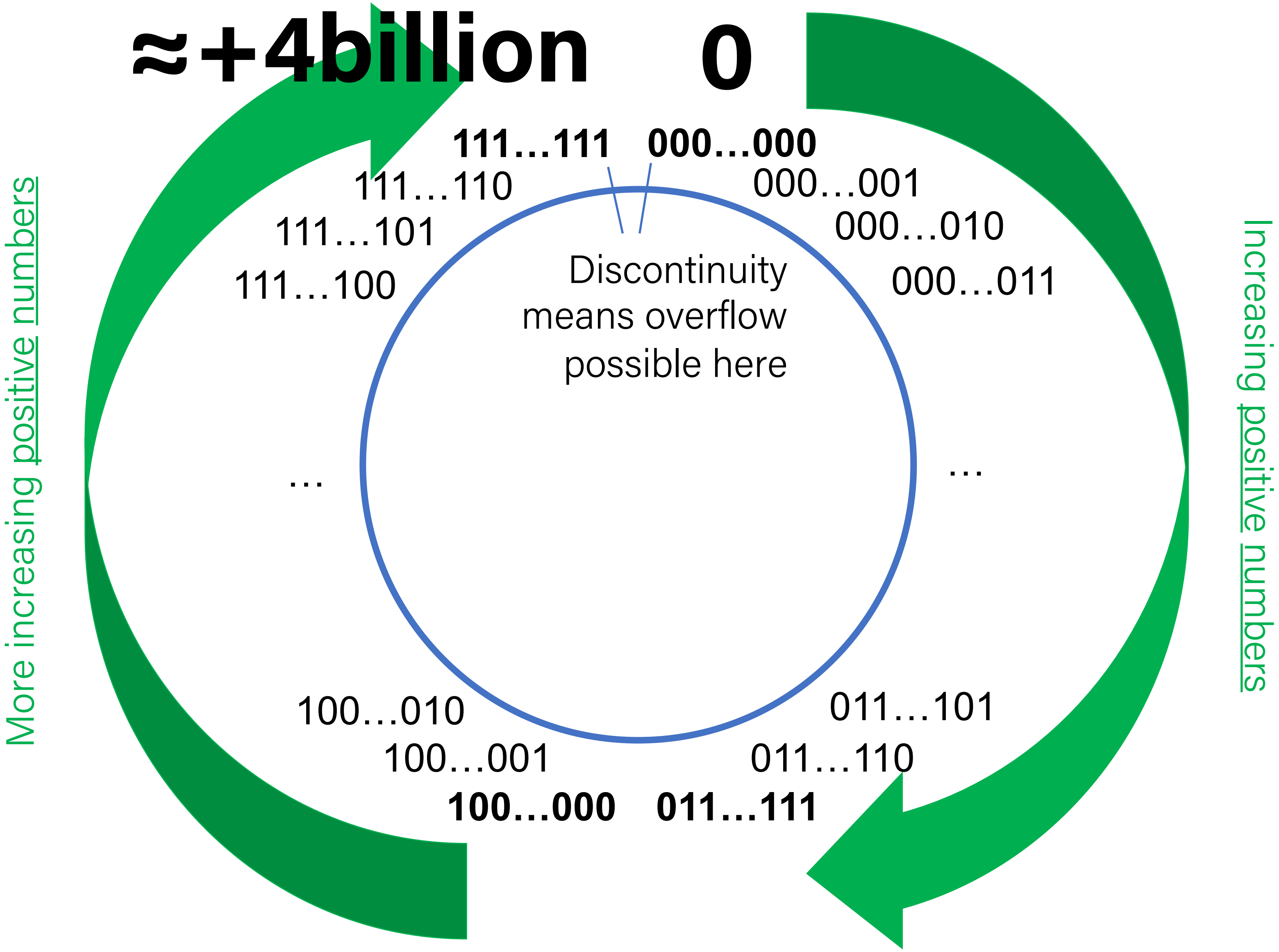
Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

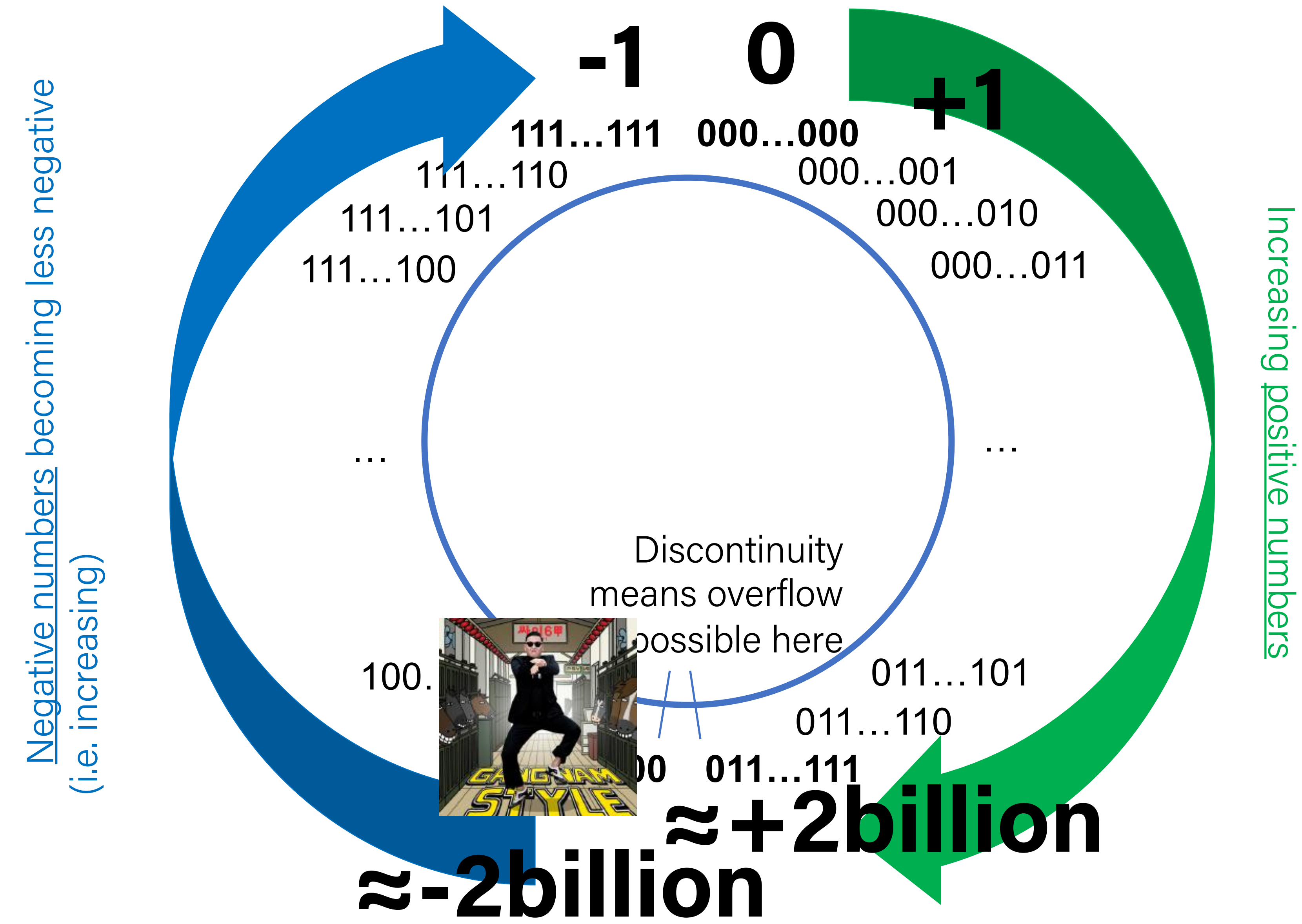
Recap: Unsigned and Signed Integers



Recap: Overflow in Unsigned Integers



Recap: Overflow in Signed Numbers



Recap: Expanding Bit Representations

- Sometimes, we want 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, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *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 to the larger type*.

Recap: 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

Bits and Bytes So Far

- all data is ultimately stored in memory in binary
- When we declare an integer variable, under the hood it is stored in binary

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

- Until now, we only manipulate our integer variables in base 10 (e.g. increment, decrement, set, etc.)
- Today, we will learn about how to manipulate the underlying binary representation!
- This is useful for: more efficient arithmetic, more efficient storing of data, etc.

Plan For Today

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Aside: ASCII

- ASCII is an encoding from common characters (letters, symbols, etc.) to bit representations (chars).
  - E.g. 'A' is 0x41
- Neat property: all uppercase letters, and all lowercase letters, are sequentially represented!
  - E.g. 'B' is 0x42

More on this next week!

Lecture Plan

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

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 (&)

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

```
  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:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

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).

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
```

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

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
```

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`).

Lecture Plan

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

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`**.

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

Bit Vectors and Sets

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

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

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

Bit Vectors and Sets

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

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

```
  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 COMP202?

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

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

Bit Masking

```
#define COMP100 0x1      /* 0000 0001 */
#define COMP106 0x2      /* 0000 0010 */
#define COMP132 0x4      /* 0000 0100 */
#define COMP201  0x8     /* 0000 1000 */
#define COMP202  0x10    /* 0001 0000 */
#define COMP291  0x20    /* 0010 0000 */
#define COMP301  0x40    /* 0100 0000 */
#define COMP302  0x80    /* 1000 0000 */

char myClasses = ...;
myClasses = myClasses | COMP201; // Add COMP201
```

Bit Masking

```
#define COMP100 0x1      /* 0000 0001 */
#define COMP106 0x2      /* 0000 0010 */
#define COMP132 0x4      /* 0000 0100 */
#define COMP201  0x8     /* 0000 1000 */
#define COMP202  0x10    /* 0001 0000 */
#define COMP291  0x20    /* 0010 0000 */
#define COMP301  0x40    /* 0100 0000 */
#define COMP302  0x80    /* 1000 0000 */

char myClasses = ...;
myClasses |= COMP201;   // Add COMP201
```

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken COMP132?

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

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

```
char myClasses = ...;
myClasses = myClasses & ~COMP132;     // Remove COMP132
```

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken COMP132?

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

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

```
char myClasses = ...;
myClasses &= ~COMP132;  // Remove COMP132
```

Bit Masking

- **Example:** how do we check if we've taken COMP301?

COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302
0	0	1	0	0	0	1	1

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

```
char myClasses = ...;
if (myClasses & COMP301) {...
     // taken COMP301!
```

Bit Masking

- **Example:** how do we check if we've *not* taken COMP201?

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

```
  00100011
& 00010000
  --------
  00000000
```

```
char myClasses = ...;
if (!(myClasses & COMP201)) {...
     // not taken COMP201!
```

Bit Masking

- **Example:** how do we check if we've *not* taken COMP201?

0	0	1	0	0	0	1	1
COMP100	COMP106	COMP132	COMP201	COMP202	COMP291	COMP301	COMP302

```
  00100011      00000000
& 00010000    ^ 00010000
  --------      --------
  00000000      00010000
```

```
char myClasses = ...;
if ((myClasses & COMP201) ^ COMP201) {...
    // not taken COMP201!
```

Practice: Bitwise Operations

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

`0b00001101`

1. ...turn **on** a particular set of bits? **OR**
2. ...turn off a particular set of bits? **AND**
3. ...flip a particular set of bits? **XOR**

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

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

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

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

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`**?

```
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 | 0xff
```

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 all other bytes.

```
j ^ ~0xff
```

Powers of 2

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

`is_power_of_2`

Lecture Plan

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

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
```

8-bit examples:

```
00110111 << 2 results in 11011100
01100011 << 4 results in 00110000
10010101 << 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.

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

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

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 bit
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!

```
short x = 2;   // 0000 0000 0000 0010
x >>= 1;       // 0000 0000 0000 0001
printf("%d\n", x); // 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;    // evaluates to x shifted to the right by k bit
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!

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

Right Shift (>>)

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

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<<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:

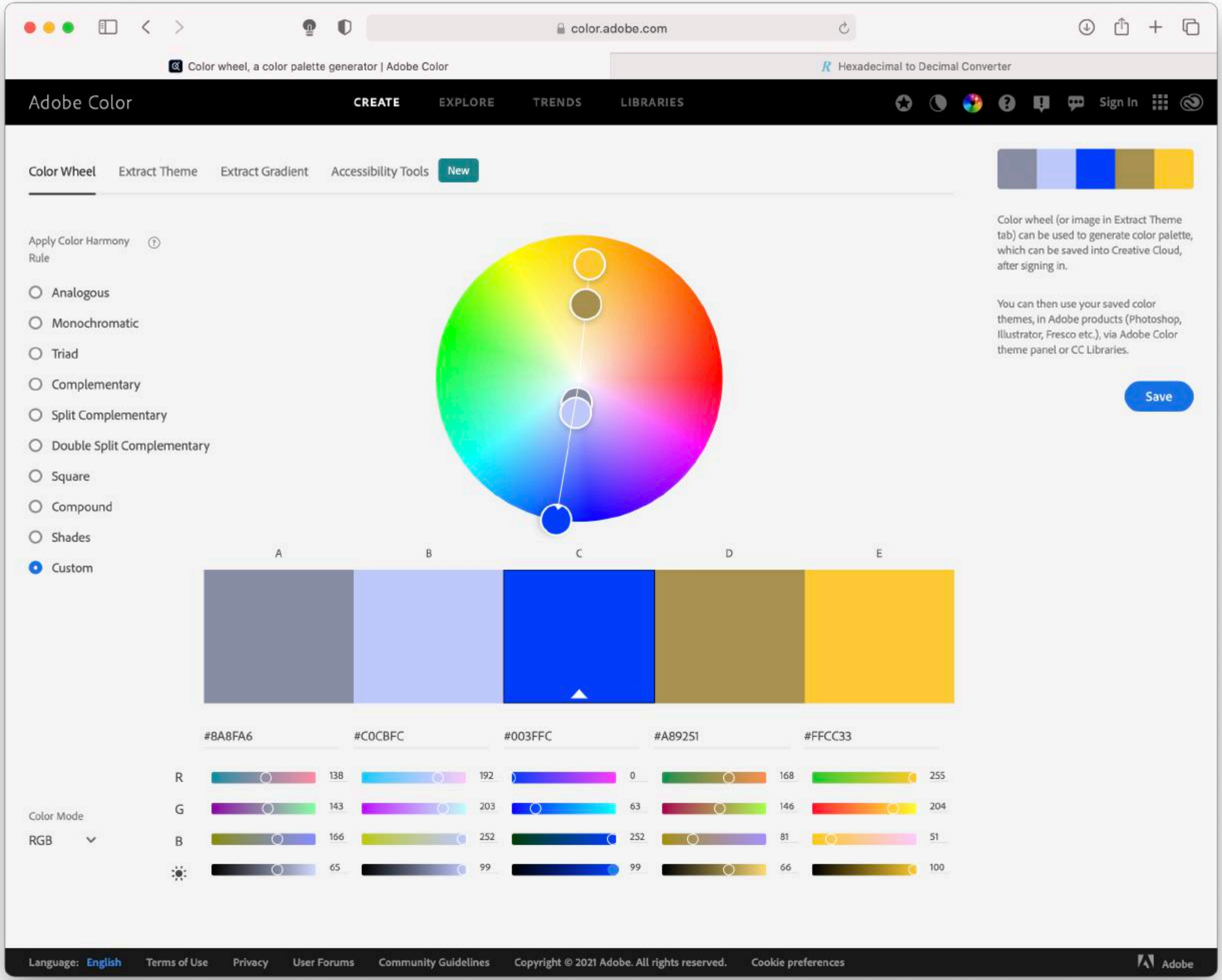
```
long num = 1 << 32;
```

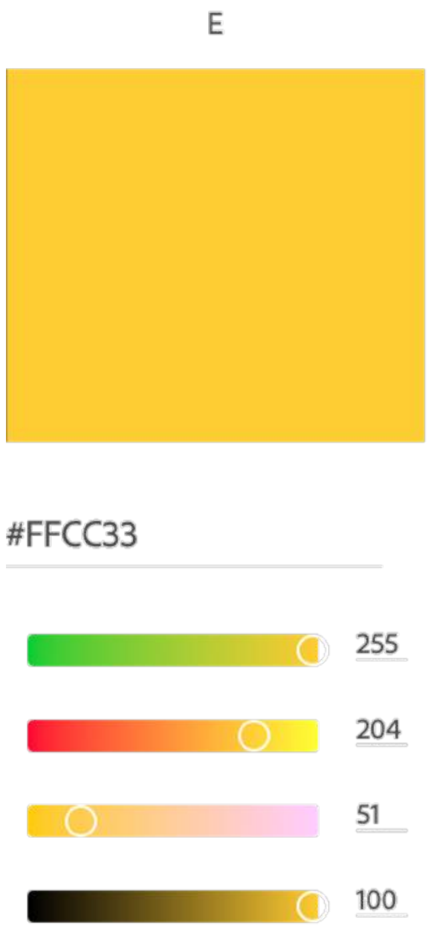
- This doesn't work! 1 is by default an **int**, and you can't shift an int by 32 because it only has 32 bits. You must specify that you want 1 to be a **long**.

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

Demo: Color Wheel

`color_wheel.c`





0xFFCC33 =
4294954035U

<u>COMP201 Topic 2</u>: How can a computer represent real numbers in addition to integer numbers?

Learning Goals

Understand the design and compromises of the floating point representation, including:

- Fixed point vs. floating point
- How a floating point number is represented in binary
- Issues with floating point imprecision
- Other potential pitfalls using floating point numbers in programs

Lecture Plan

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Real Numbers

- We previously discussed representing integer numbers using two’s complement.
- However, this system does not represent real numbers such as 3/5 or 0.25.
- How can we design a representation for real numbers?

Real Numbers

Problem: unlike with the integer number line, where there are a finite number of values between two numbers, there are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1

Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001,…

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers*.

Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: $1/6_{10} = 0.16666666....._{10}$

Base 2: $1/10_{10} = 0.000110011001100110011..._{2}$

Therefore, by representing in base 2, *we will not be able to represent all numbers*, even those we can exactly represent in base 10.

Fixed Point

- **Idea:** Like in base 10, let's add binary decimal places to our existing number representation.

5	9	3	4	.	2	1	6
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1	0	1	1	.	0	1	1
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

Lecture Plan

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Fixed Point

- **Idea:** Like in base 10, let's add binary decimal places to our existing number representation.

1	0	1	1	.	0	1	1
8s	4s	2s	1s		1/2s	1/4s	1/8s

- **Pros:** arithmetic is easy! And we know exactly how much precision we have.

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

.	0	1	1	0	0	1	1	
	1/2s	1/4s	1/8s	...				
1	0	1	1	0	.	1	1	
16s	8s	4s	2s	1s		1/2s	1/4s	

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

Base 10		Base 2
5.07E30	=	$10\underbrace{\ldots}_{\text{100 zeros}}0.1$
9.86E-32	=	$0.0\underbrace{\ldots}_{\text{100 zeros}}01$

To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible "floating" decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior

Recap

- Bitwise Operators
- Bitmasks
- Bit Shift Operators
- Representing real numbers
- Fixed Point

Next time: *More on how can a computer represent floating point numbers?*