COMP201

Computer Systems & Programming

Lecture #03 –Bits and Bitwise Operators

Aykut Erdem // Koç University // Fall 2025

KOÇ UNIVERSITY

Good news, everyone!

- Labs will start this Friday!

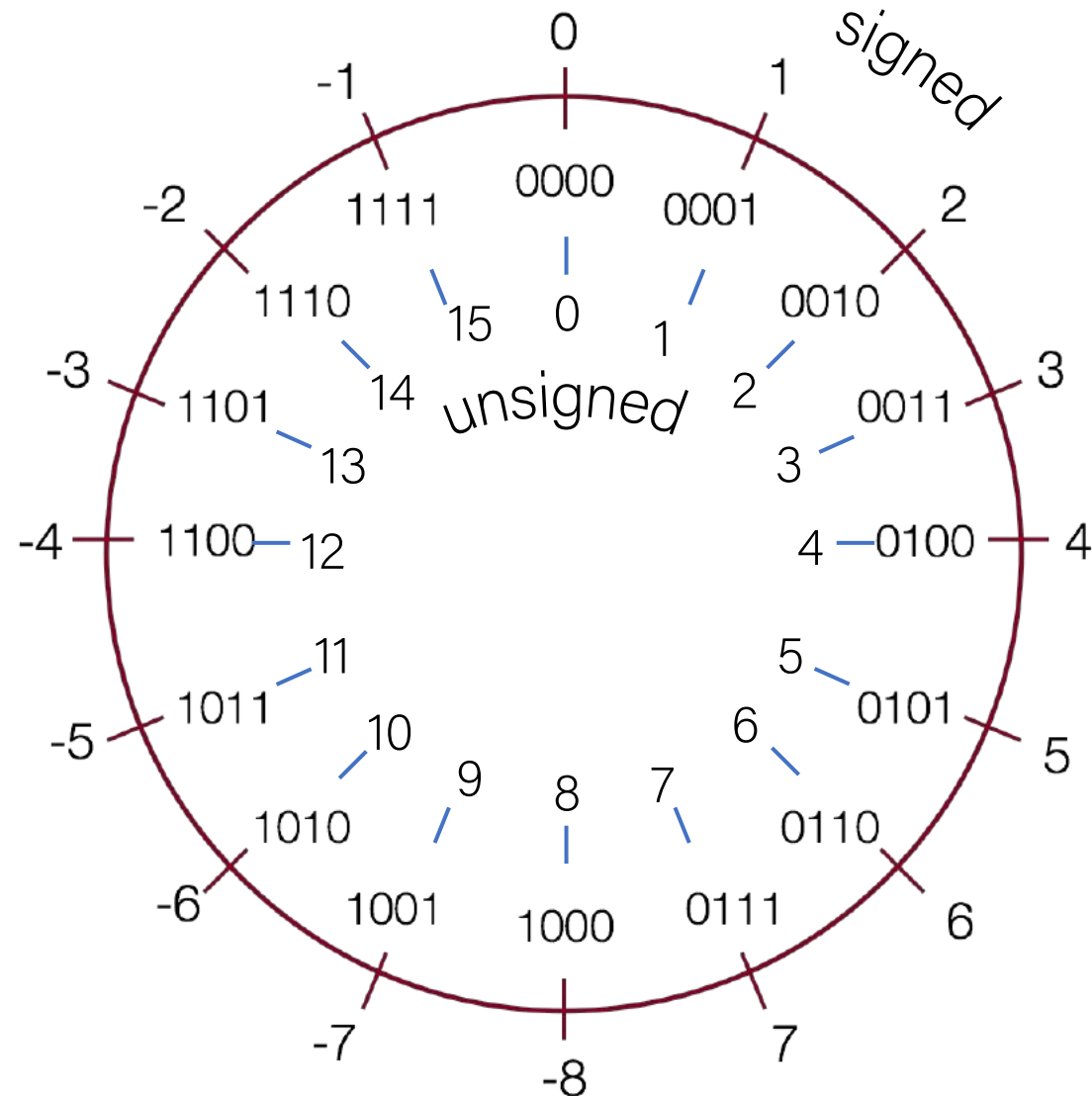- Assg0 will be out this week

Recap

- Bits and Bytes

- Hexadecimal

- Integer Representations

- Unsigned Integers

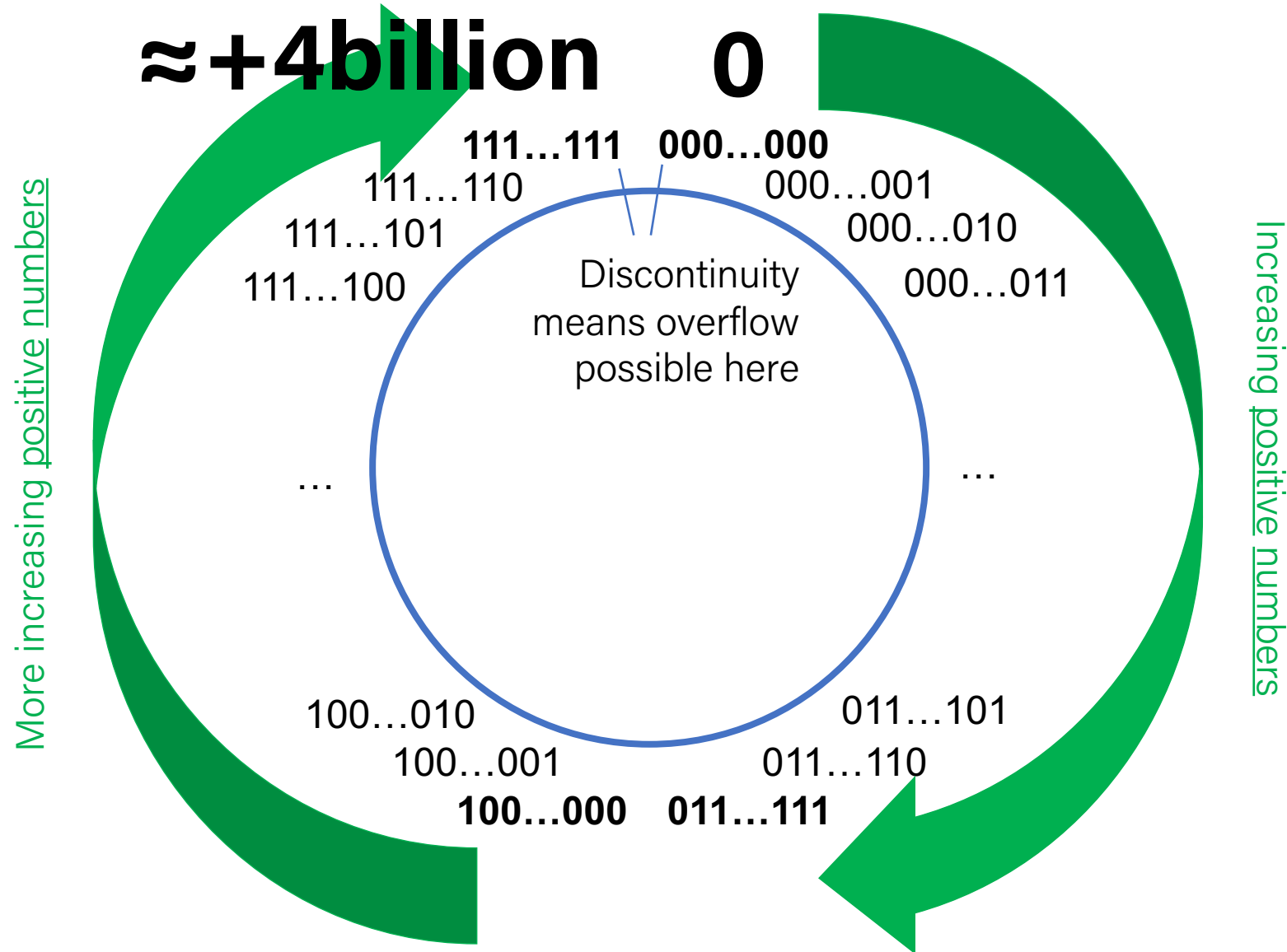- Signed Integers

- Overflow

- Casting and Combining Types

Recap: Min and Max Integer Values

Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

Recap: Unsigned and Signed Integers

Recap: Overflow in Unsigned Integers

≈+4billion **0**

111...111 000...000
111...110 000...001
111...101 000...010
111...100 000...011

More increasing positive numbers

Increasing positive numbers

Discontinuity means overflow possible here

... ...

100...010 011...101
100...001 011...110
100...000 **011...111**

Recap: Overflow in Signed Numbers

-1 **0**

+1

111…111 **000…000**

111…110 000…001

111…101 000…010

111…100 000…011

Negative numbers becoming less negative (i.e. increasing)

Increasing positive numbers

… …

Discontinuity means overflow possible here

100. 011…101

011…110

00 **011…111**

≈+2billion

≈-2billion

Recap: Casting

4-bit two's complement signed integer representation

4-bit unsigned integer representation

Recap: Casting

- What happens at the byte level when we cast between variable types? <mark>The bytes remain the same!</mark> <mark>This means they may be interpreted differently depending on the type.</mark>

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
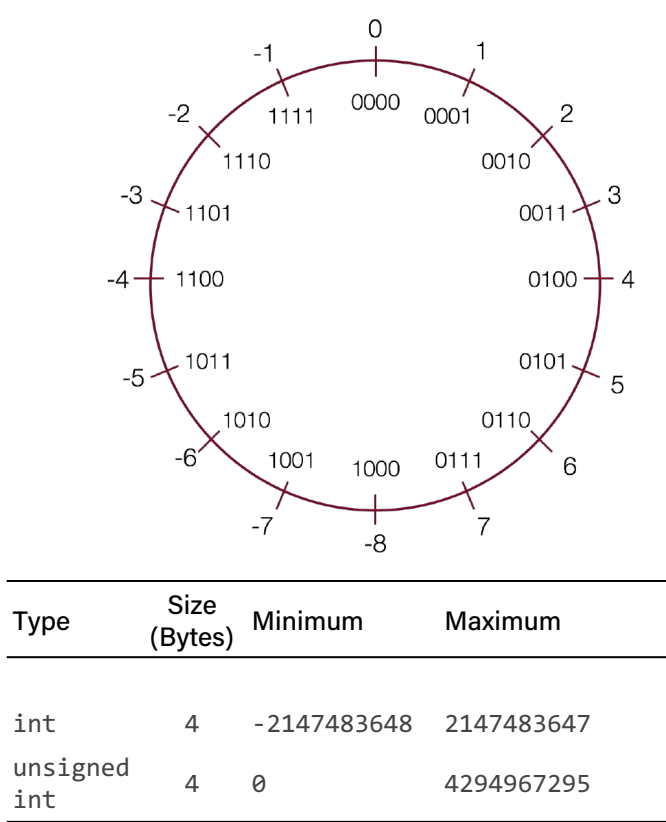```

This prints out: `"v = -12345, uv = 4294954951"`.

- The bit representation for -12345 is
  `0b11111111111111111100111111000111.`

  If we treat this binary representation as a positive number, it's *huge*!

# Recap: 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.

| Expression | Type | Evaluation | Correct? |
|---|---|---|---|
| `0 == 0U` | Unsigned | 1 | yes |
| `-1 < 0` | Signed | 1 | yes |
| `-1 < 0U` | Unsigned | 0 | No! |
| `2147483647 > -2147483647 - 1` | Signed | 1 | yes |
| `2147483647U > -2147483647 - 1` | Unsigned | 0 | No! |
| `2147483647 > (int)2147483648U` | Signed | 1 | No! |
| `-1 > -2` | Signed | 1 | yes |
| `(unsigned)-1 > -2` | Unsigned | 1 | yes |

| Type | Size (Bytes) | Minimum | Maximum |
|---|---|---|---|
| int | 4 | -2147483648 | 2147483647 |
| unsigned int | 4 | 0 | 4294967295 |

# 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 <u>how to manipulate the underlying binary representation</u>!

- This is useful for: more efficient arithmetic, more efficient storing of data, etc.

Plan For Today

- Casting and Combining Types (cont'd.)

- Byte Ordering

- Bitwise Operators

- Bitmasks

- Bit Shift Operators

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Lecture Plan

- Casting and Combining Types (cont'd.)
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*
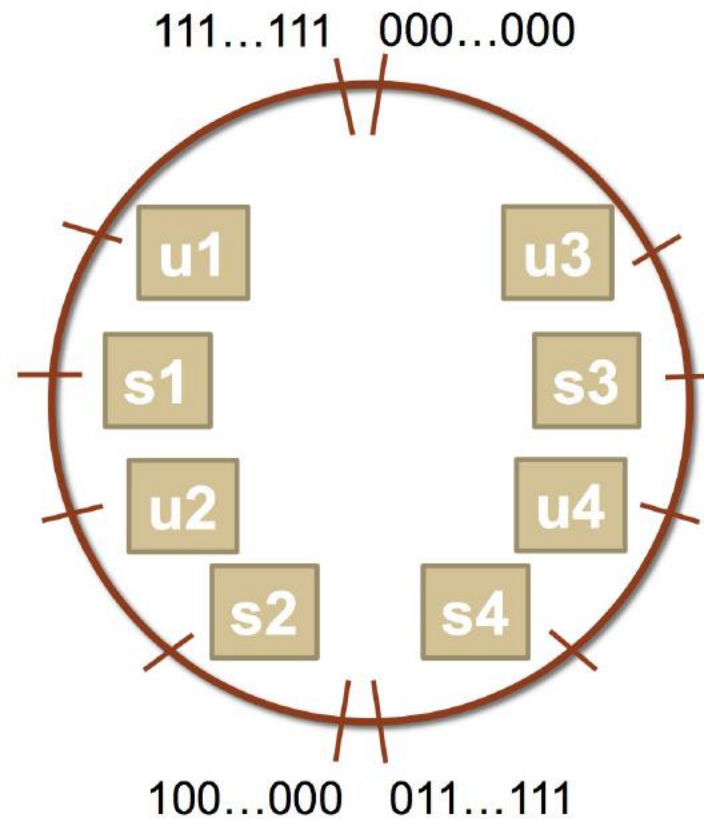
s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*
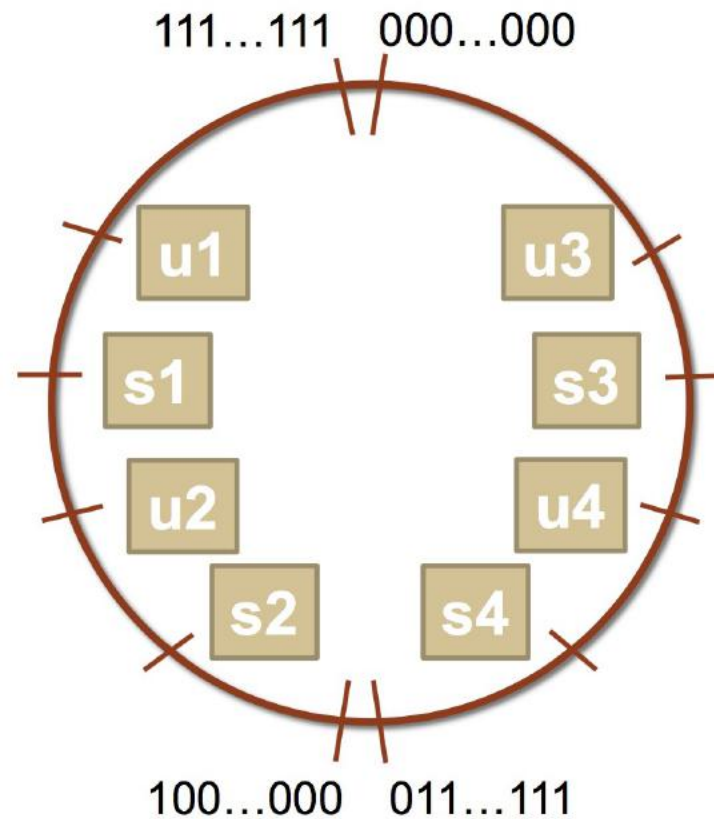
s3 > u3 - true
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

s3 > u3 - true
u2 > u4 - true
s2 > s4
s1 > s2
u1 > u2
s1 > u3

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*
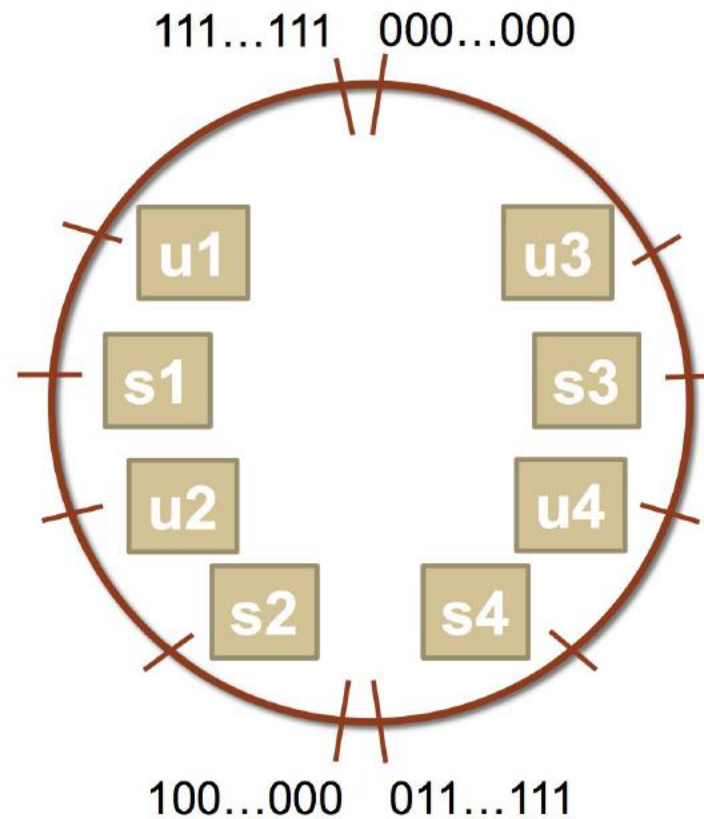
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2
u1 > u2
s1 > u3

111…111   000…000

u1   u3

s1   s3
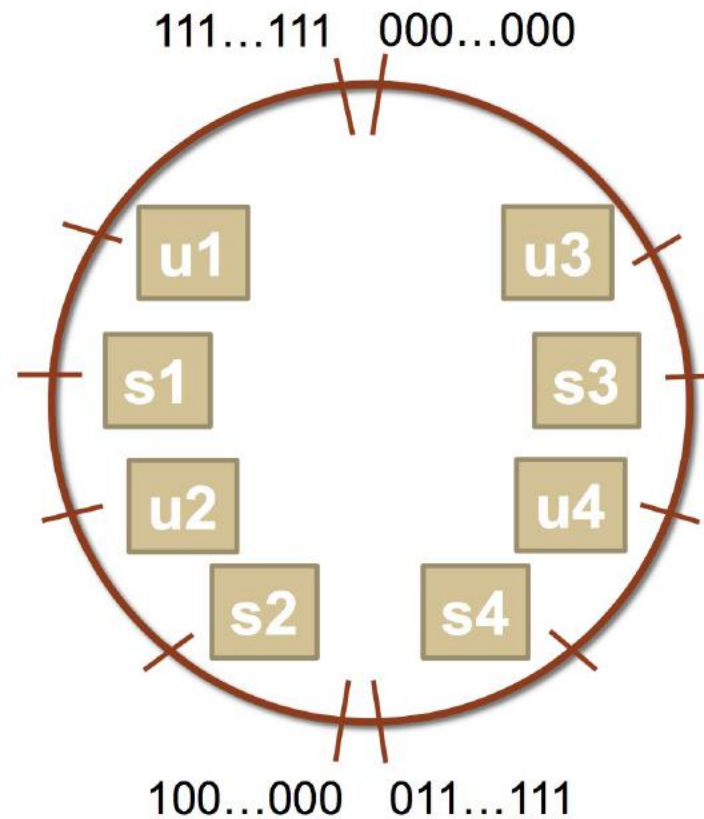
u2   u4

s2   s4

100…000   011…111

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2
s1 > u3

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*
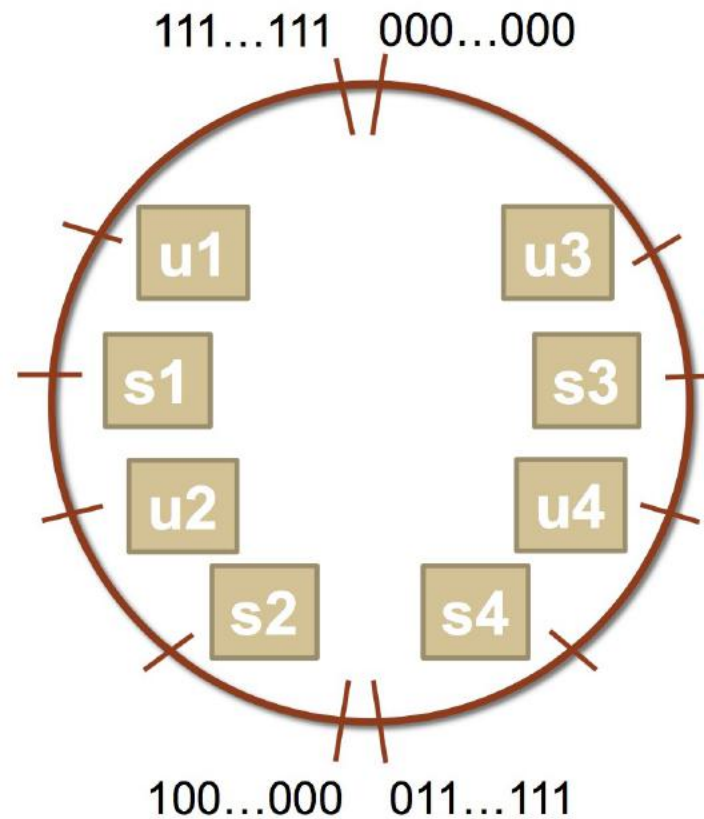
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2 - true
s1 > u3

Comparisons Between Different Types

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*
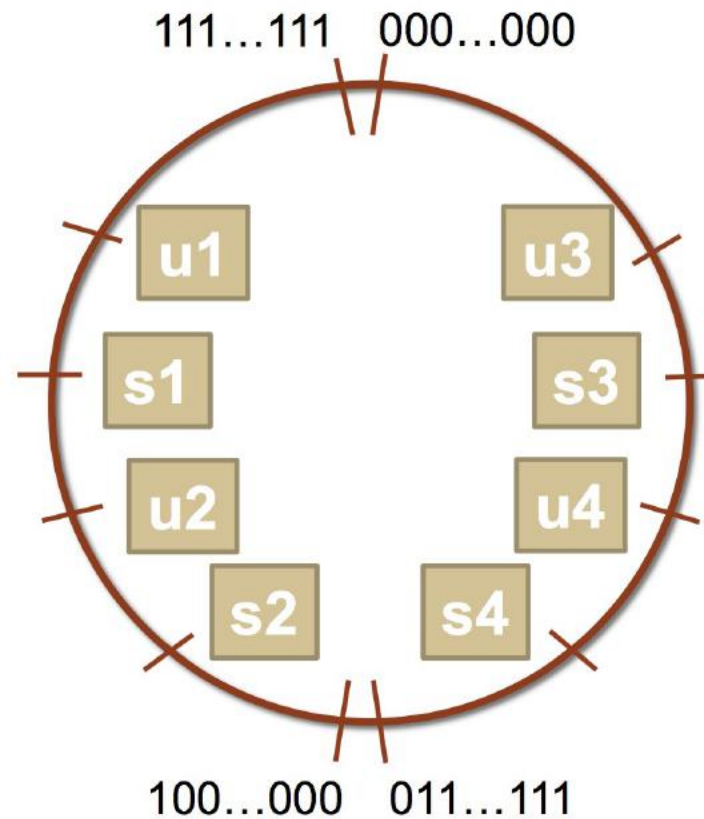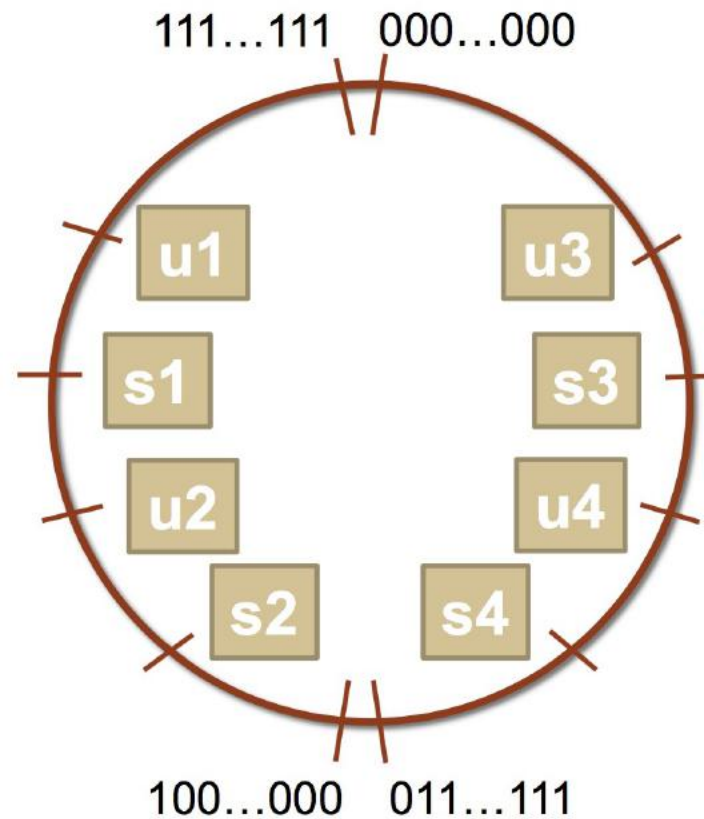
```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2 - true
s1 > u3 - true
```

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*.

Expanding Bit Representation

```
unsigned short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b


unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

Expanding Bit Representation

```
short s = 4;
// short is a 16-bit format, so                          s = 0000 0000 0000 0100b


int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b


— or —


short s = -4;
// short is a 16-bit format, so                          s = 1111 1111 1111 1100b


int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Truncating Bit Representation

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

```
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 <u>*more significant bits*</u>.

```
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

This is -3! **If the number does fit, it will convert fine.** `y` looks like this:

1111 1111 1111 1111 1111 1111 1111 1101 // still -3

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 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

This is 62464! **Unsigned numbers can lose info, too**. Here is what `y` looks like:

0000 0000 0000 0000 1111 0100 0000 0000 // still 62464

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 as a parameter and returns the size of that type, in bytes.

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 130; // 0b1000 0010
char cx = x;
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x;
```

```
short x = 25; // 0b1 1001
char cx = x;
```

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 130; // 0b0000 0000 1000 0010
char cx = x;
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x;
```

```
short x = 25; // 0b0000 0000 0001 1001
char cx = x;
```

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 130; // 0b0000 0000 1000 0010
char cx = x; // -126
```

```
short x = -132 // 0b1111 1111 0111 1100
char cx = x; // 124
```

```
short x = 25; // 0b0000 0000 0001 1001
char cx = x; // 25
```

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 390; // 0b1 1000 0110
char cx = x;
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x;
```

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 390; // 0b0000 0001 1000 0110
char cx = x;
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x;
```

Practice: Truncation

What are the values of **cx** for the passages of code below?

```
short x = 390; // 0b0000 0001 1000 0110
char cx = x; // -122
```

```
short x = -15; // 0b1111 1111 1111 0001
char cx = x; // -15
```

In Sum: Basic Rules of Expanding, Truncating

- Expanding (e.g., `short int` to `int`)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., `unsigned` to `unsigned short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Lecture Plan

- Casting and Combining Types (cont'd.)
- **Byte Ordering**
- Bitwise Operators
- Bitmasks
- Bit Shift Operators

Number Representations

C Declaration	Intel IA32	X86-64
`int`	4	4
`double`	8	8
`float`	4	4
`char`	1	1
`char *`	4	8
`short`	2	2
`long`	4	8

Byte Ordering

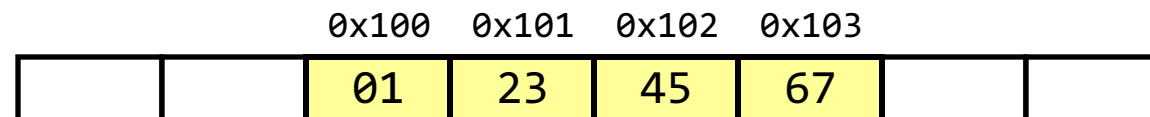- So, how are the bytes within a multi-byte word ordered in memory?

Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - **Big Endian**: Sun (Oracle SPARC), PPC Mac, Internet
    - Least significant byte has highest address
  - **Little Endian**: x86, ARM processors running Android, iOS, and Linux
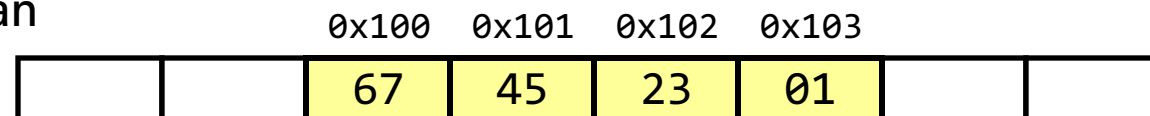   - Least significant byte has lowest address

Byte Ordering Example

- **Big Endian:** Sun (Oracle SPARC), PPC Mac, Internet
  - Least significant byte has highest address

- **Little Endian:** x86, ARM processors running Android, iOS, and Linux
  - Least significant byte has lowest address

- **Example:**
  - Variable `x` has 4-byte value of `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

On Holy Wars and a Plea for Peace

Danny Cohen
Information Sciences Institute

This article was written in an attempt to stop a war. I hope it is not too late for peace to prevail again. Many believe that the central question of this war is, What is the proper byte order in messages? More specifically, the question is, Which bit should travel first—the bit from the little end of the word or the bit from the big end of the word?

Followers of the former approach are called Little Endians, or Lilliputians; followers of the latter are called Big Endians, or Blefuscuians. I employ these Swiftian terms because this modern conflict is so reminiscent of the holy war described in *Gulliver's Travels*.[1]

Approaches to serialization

The above question arises as a result of the serialization process performed on messages to allow them to be sent through communication media. If the unit of communication is a message, this question has no meaning. If the units are computer words, one must determine their size and the order in which they are sent.

Since they are sent virtually at once, there is no need to determine the order of the elements of these words.

If the unit of transmission is an eight-bit byte, questions about bytes are meaningful but questions about the order of the elementary particles that constitute these bytes are not.
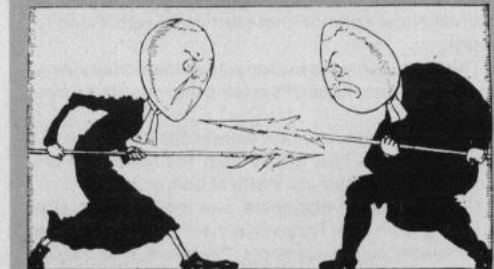
If the units of communication are bits, the atoms (quarks?) of computation, the only meaningful question concerns the order in which the bits are sent. Most modern communication is based on a single stream of information, the bit-stream. Hence, bits, rather than bytes or words, are the units of information that are actually transmitted over channels such as wires and satellites.

Notes on Swift's *Gulliver's Travels*

Swift's hero, Gulliver, is shipwrecked and washed ashore on Lilliput, whose six-inch inhabitants are required by law to break their eggs only at the little ends. Of course, all those citizens who habitually break their eggs at the big ends are angered by the proclamation. Civil war breaks out between the Little Endians and the Big Endians, resulting in the Big Endians taking refuge on a nearby island, the kingdom of Blefuscu. The controversy is ethically and politically important for the Lilliputians. In fact, Swift has 11,000 Lilliputian rebels die over the egg question. The issue might seem silly, but Swift is satirizing the actual causes of religious or holy wars.

Swift's point is that the difference between breaking an egg at the little end and breaking it at the big end is trivial. He suggests that everyone do it in his preferred way.

Of course, we are making the opposite point. We agree that the difference between sending information with the little or the big end first is trivial, but insist that everyone must do it in the same way to avoid anarchy.

Reproduced from *The Annotated Gulliver's Travels* by Isaac Asimov. Published by Crown Publishers, Inc.

Facing page: "Audience with the Emperor of Lilliput," Gavarni, 1850's. Courtesy Library of Congress.

The origin of "endian"

"Gulliver finds out that there is a law, proclaimed by the grandfather of the present ruler, requiring all citizens of Lilliput to break their eggs only at the little ends. Of course, all those citizens who broke their eggs at the big ends were angered by the proclamation. Civil war broke out between the Little-Endians and the Big-Endians, resulting in the Big-Endians taking refuge on a nearby island, the kingdom of Blefuscu."

– Danny Cohen, On Holy Wars and A Plea For Peace (1980)
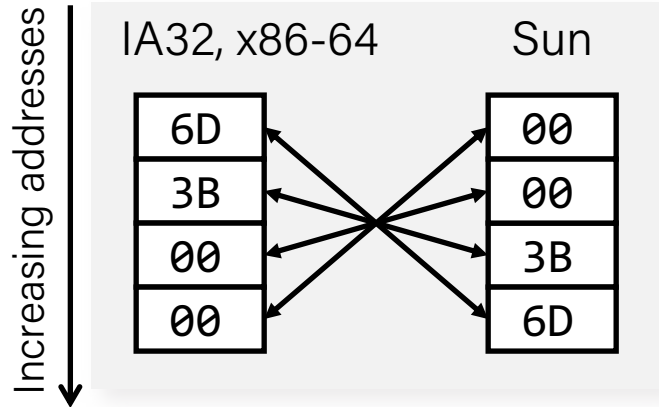

Illustration by Chris Beatrice

Representing Integers

int A = 15213;

int B = -15213;

long int C = 15213;

Two's complement representation

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

- Casting and Combining Types (cont'd.)
- Byte Ordering
- **Bitwise Operators**
- Bitmasks
- Bit Shift Operators

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	OR	XOR	NOT

``` 0110```
```& 1100```
```----```
``` 0100```
	``` 0110```
	```| 1100```
	```----```
	``` 1110```
		``` 0110```
		```^ 1100```
		```----```
		``` 1010```
			```~ 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

- Casting and Combining Types (cont'd.)
- Byte Ordering
- Bitwise Operators
- **Bitmasks**
- Bit Shift Operators

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?

0	0	1	0	0	0	1	1

COMP100 COMP106 COMP132 COMP201 COMP202 COMP291 COMP301 COMP302

```
  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        0b00001101        0b00001101
  0b00000010        0b11111011        0b00000110
_____        _____        _____
  0b00001111        0b00001001        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`?
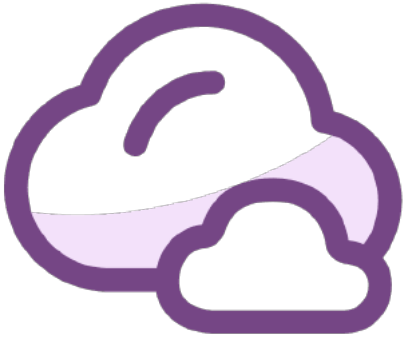
```
int j = ...;
int k = j & 0xff;  // mask to get just lowest byte
```

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.

Write an expression that, given a 32-bit integer j, sets its least-significant byte to all 1s, but preserves all other bytes.
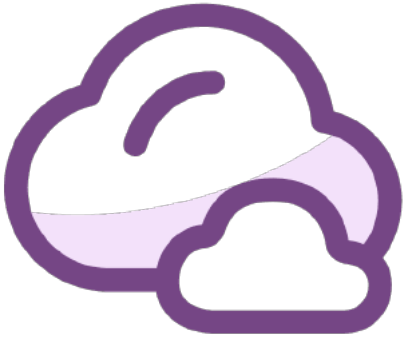
slido

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.

Write an expression that, given a 32-bit integer j, flips ("complements") all but the least-significant byte, and preserves all other bytes.

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

Any power of 2 minus 1 is all ones: $(2^N - 1 = 111....b)$

2 = 2^1   2-1 = 1   (**1b**)
4 = 2^2   4-1 = 3   (**11b**)
8 = 2^3   8-1 = 7   (**111b**)

Example: Take 8   1000 & 0111 = 0000

`is_power_of_2`

Lecture Plan

- Casting and Combining Types (cont'd.)

- Byte Ordering

- Bitwise Operators

- Bitmasks

- **Bit Shift Operators**

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. <mark>Operator precedence can be tricky</mark>! 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;
```

Recap

- Casting and Combining Types (cont'd.)
- Byte Ordering
- Bitwise Operators
- Bitmasks
- Bit Shift Operators

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