CS107, Lecture 3

Bits and Bytes; Bitwise Operators

reading:

Bryant & O’Hallaron, Ch. 2.1

Based on slides created by Marty Stepp, Cynthia Lee, Chris Gregg, and others.

Plan For Today

- **Recap**: Integer Representations
- Truncating and Expanding
- Bitwise Operators and Masks
- **Demo 1:** Courses
- **Break:** Announcements
- **Demo 2:** Powers of 2
- Bit Shift Operators

Plan For Today

- **Recap: Integer Representations**
- Truncating and Expanding
- Bitwise Operators and Masks
- **Demo 1:** Courses
- **Break:** Announcements
- **Demo 2:** Powers of 2
- Bit Shift Operators

Base 2

1 0 1 1

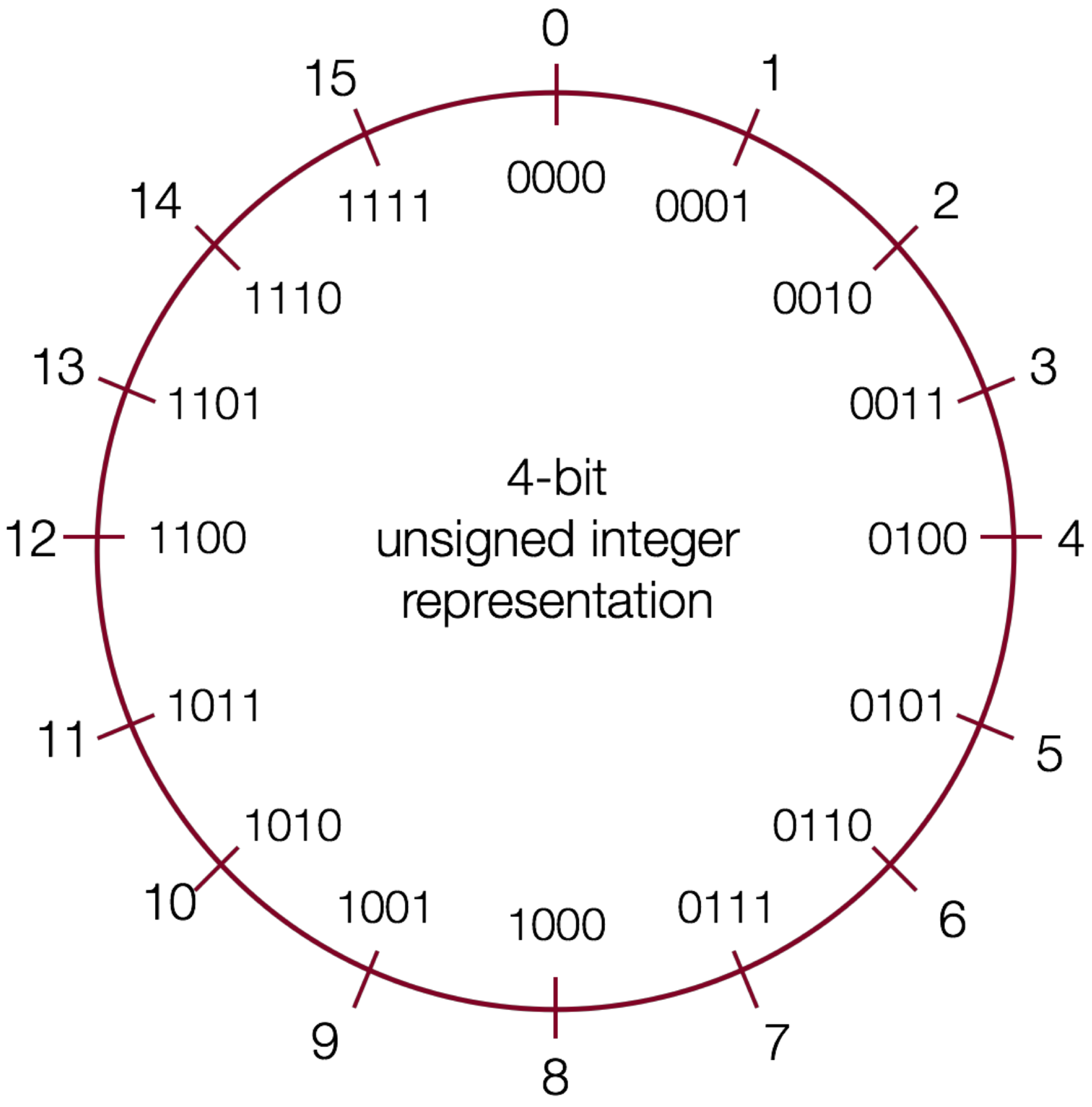
2^3 2^2 2^1 2^0

Hexadecimal

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

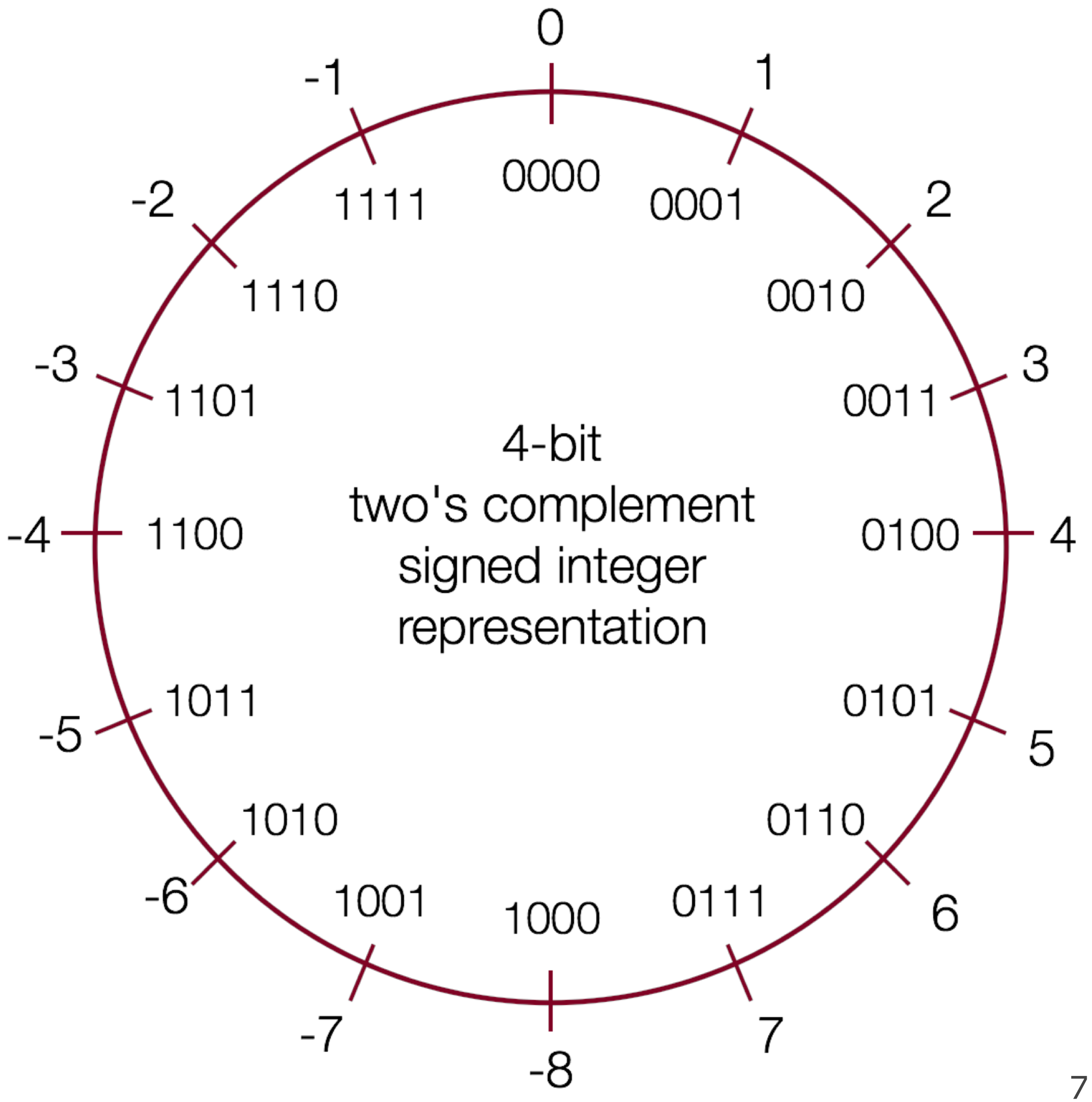
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Unsigned Integers



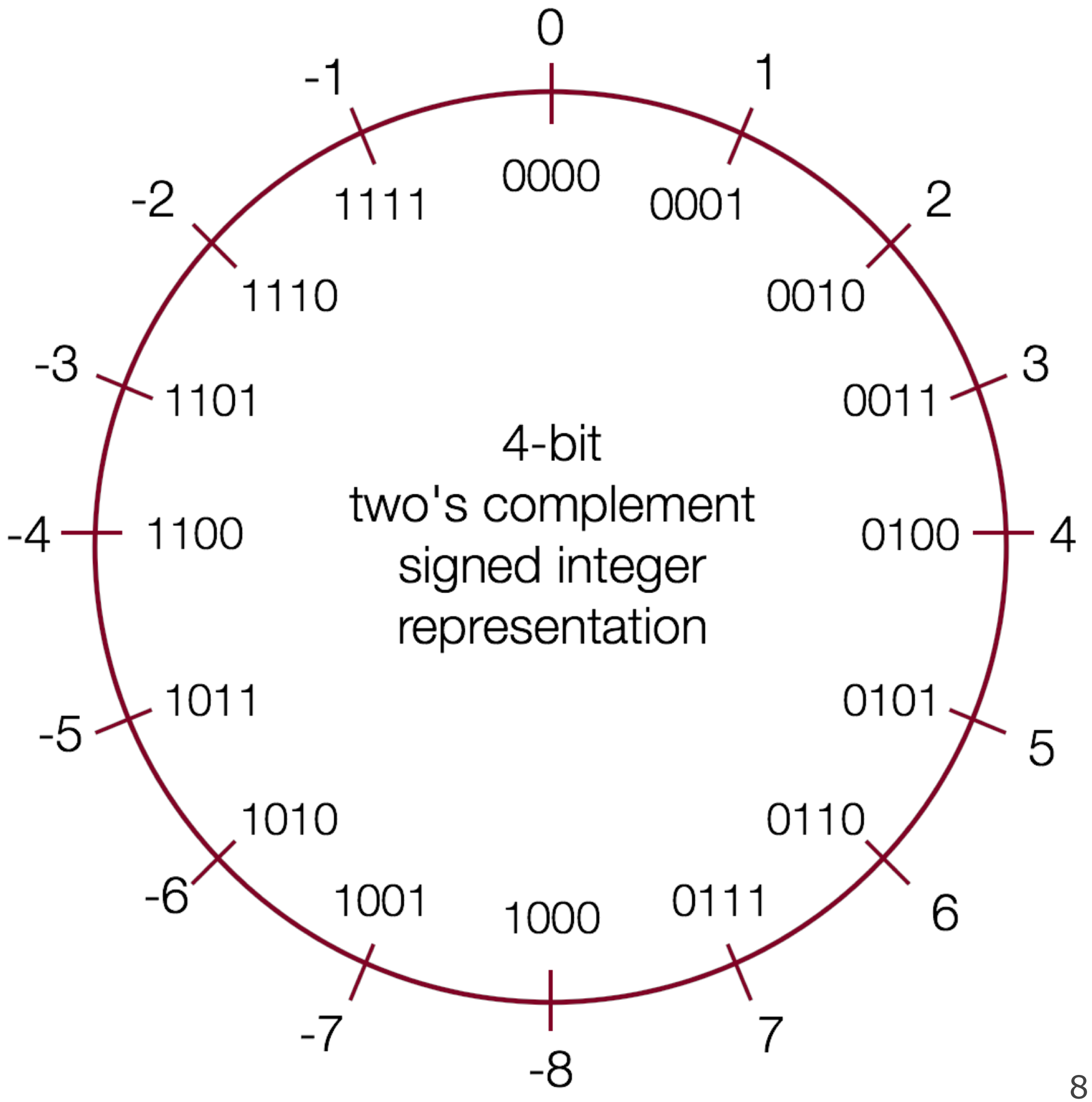
Signed Integers: Two’s Complement

- In **two’s complement**, we represent a positive number as **itself**, and its negative equivalent as the **two’s complement of itself.**
- The **two’s complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!



Signed Integers: Two’s Complement

- **Con:** more difficult to represent, and difficult to convert to/from decimal and between positive and negative.
- **Pro:** only 1 representation for 0!
- **Pro:** all bits are used to represent as many numbers as possible
- **Pro:** it turns out that the most significant bit *still indicates the sign* of a number.
- **Pro:** arithmetic is easy: we just add!



Overflow and Underflow

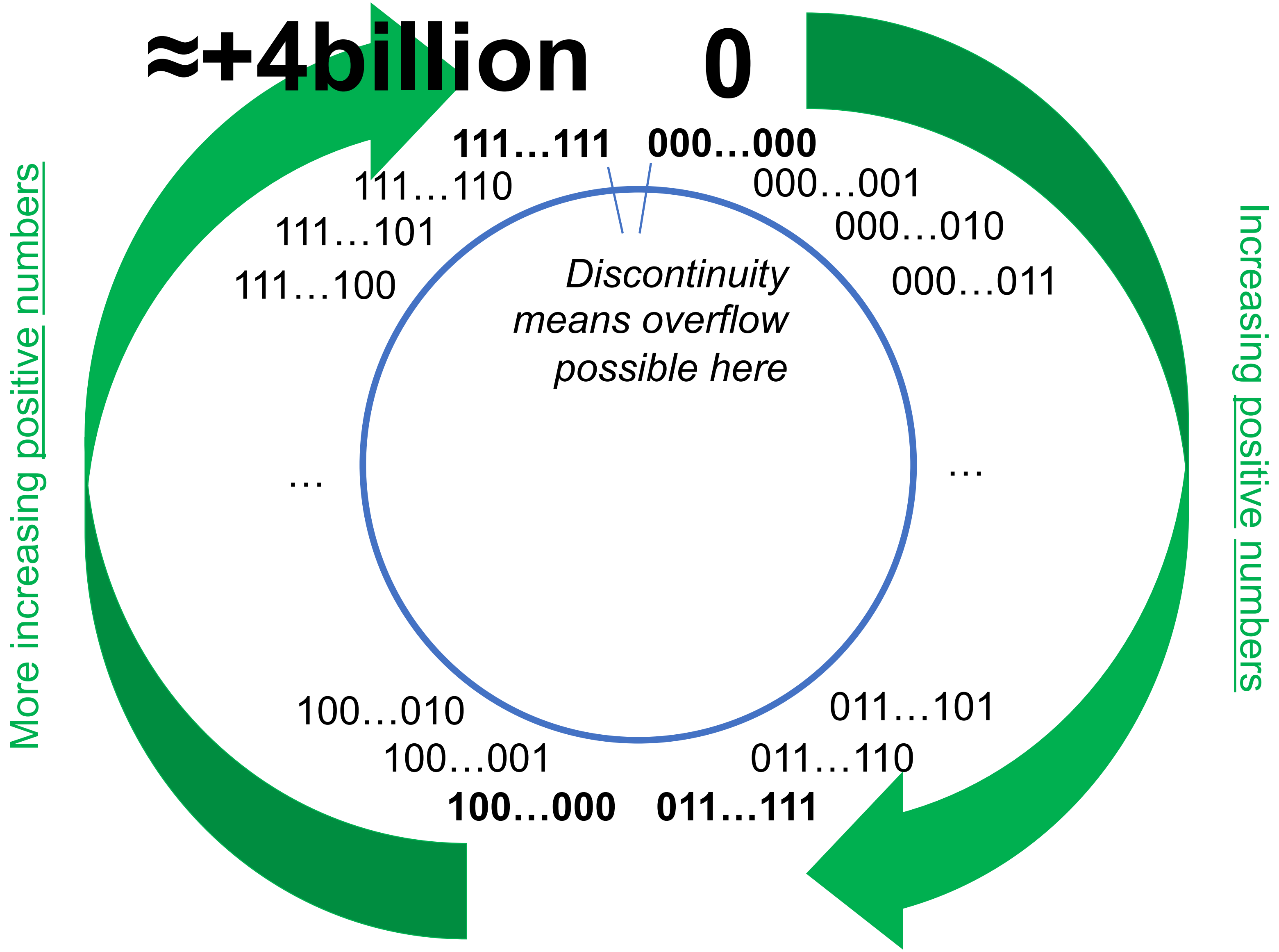
- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

```
0b1111 + 0b1 = 0b0000
```

- If you go below the **minimum** value of your bit representation, you *wrap around* or *underflow* back to the **largest** bit representation.

```
0b0000 - 0b1 = 0b1111
```

Unsigned Integers



≈+4billion **0**

111…111 **000…000**

111…110 000…001

111…101 000…010

111…100 000…011

Discontinuity means overflow possible here

… …

100…010 011…101

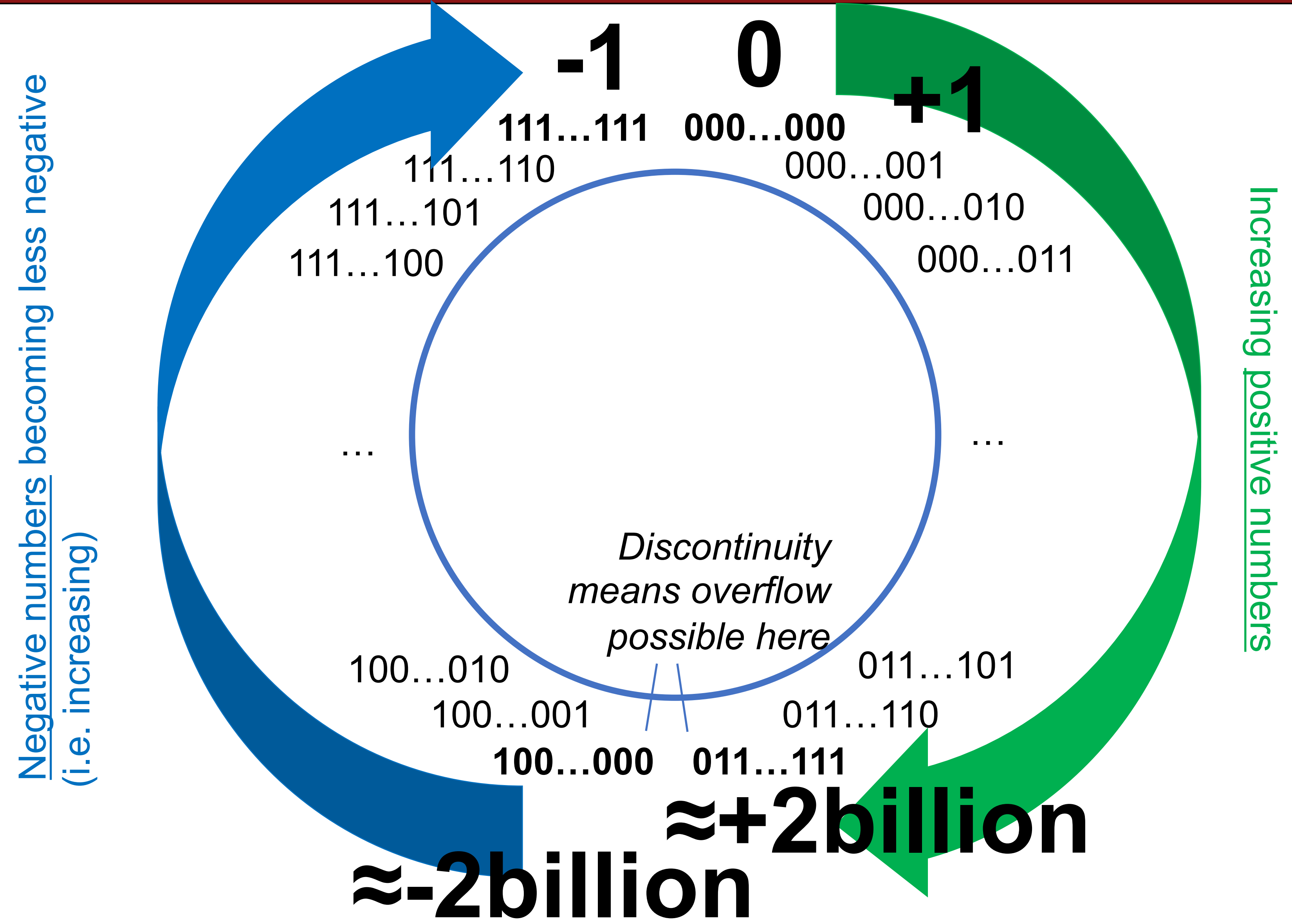
100…001 011…110

100…000 **011…111**

More increasing positive numbers

Increasing positive numbers

Signed Numbers



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

-1 **0** **+1**

111…111 **000…000**

111…110
111…101
111…100

000…001
000…010
000…011

…

…

Discontinuity means overflow possible here

100…010
100…001
100…000

011…101
011…110
011…111

≈+2billion

≈-2billion

Increasing positive numbers

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

printf and Integers

- There are 3 placeholders for 32-bit integers that we can use:
  - %d: signed 32-bit int
  - %u: unsigned 32-bit int
  - %x: hex 32-bit int
- As long as the value is a 32-bit type, `printf` will **treat it according to the placeholder!**

Casting

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

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

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

Casting

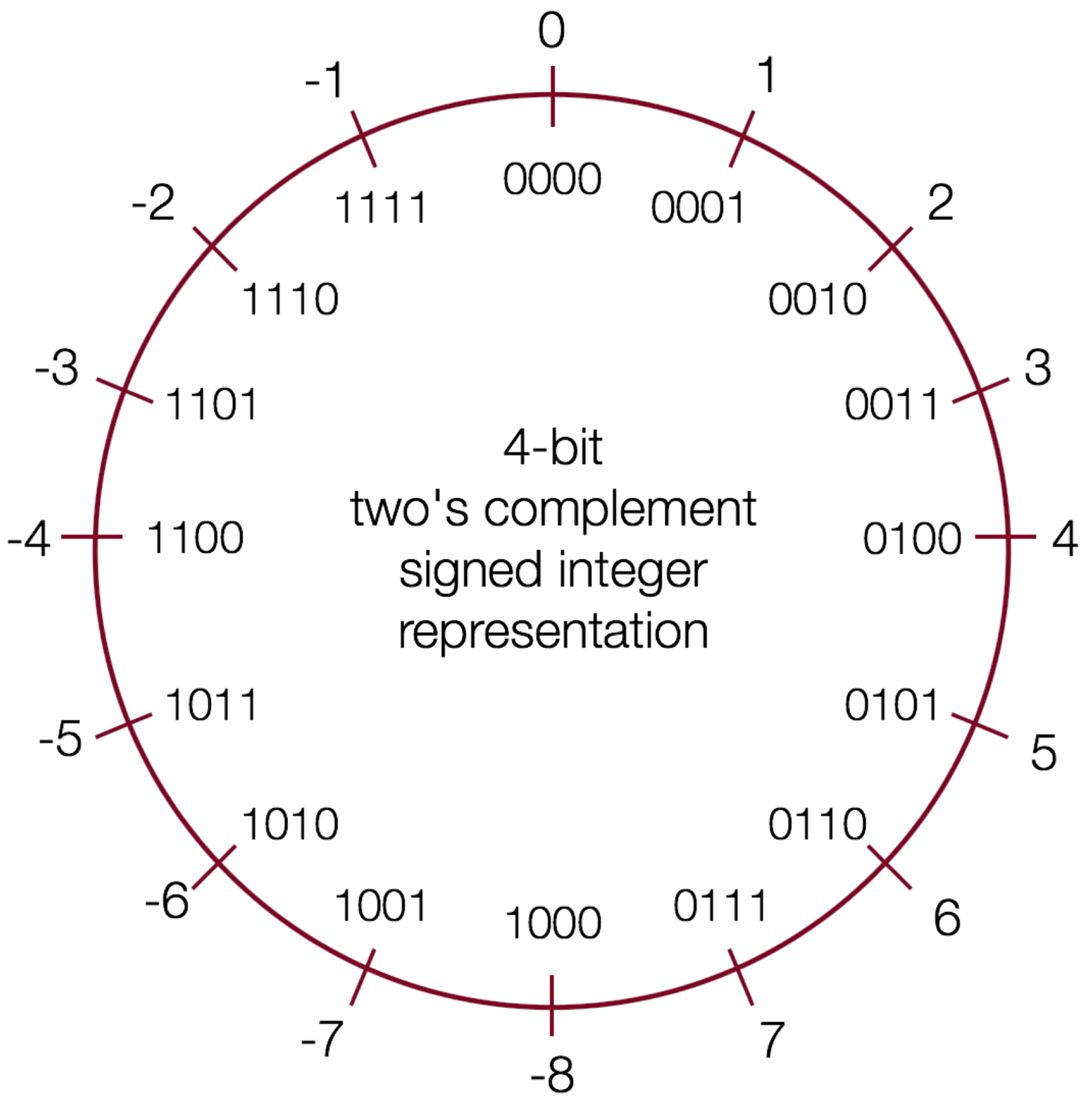
- What happens at the byte level when we cast between variable types? **The bytes remain the same!** **This means they may be interpreted differently depending on the type.**

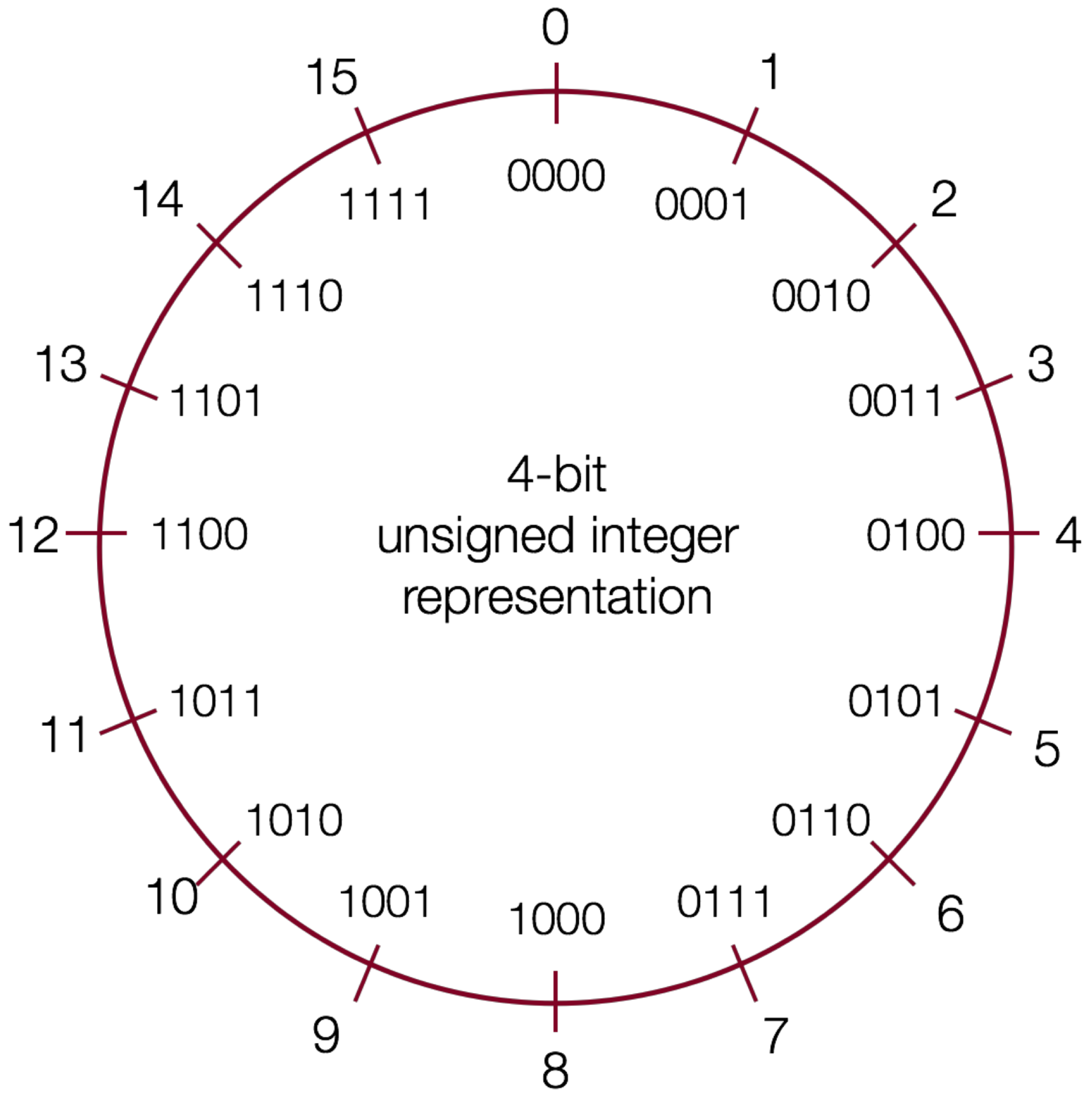
```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

-12345 in binary is **`1111 1111 1111 1111 1100 1111 1100 0111`**.

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

Casting

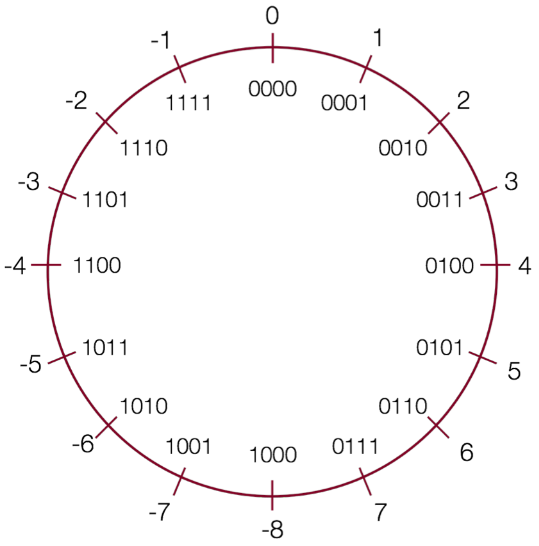

0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-8
1000
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111
4-bit
two's complement
signed integer
representation


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
4-bit
unsigned integer
representation

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.

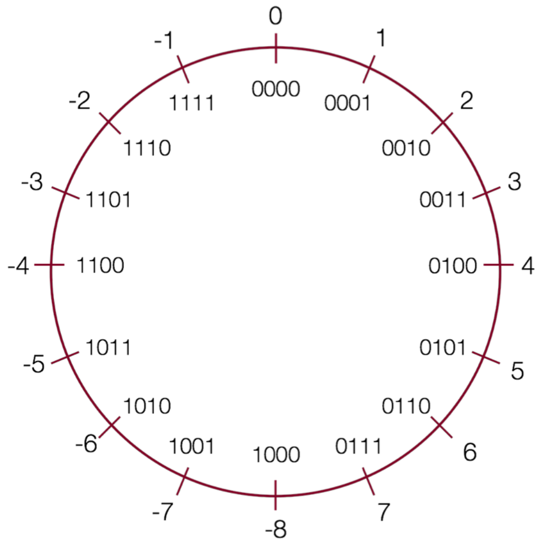
`Expression`	Type	Evaluation	Correct?
`0 == 0U`			
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

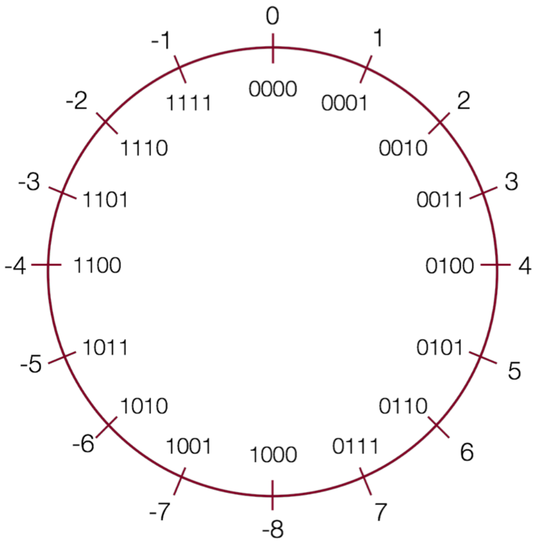
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

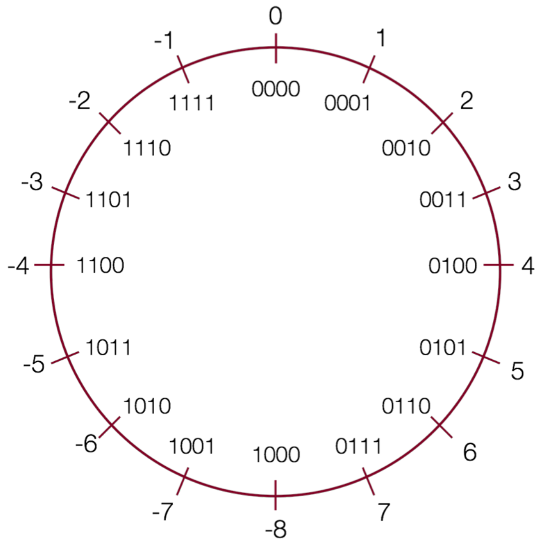
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

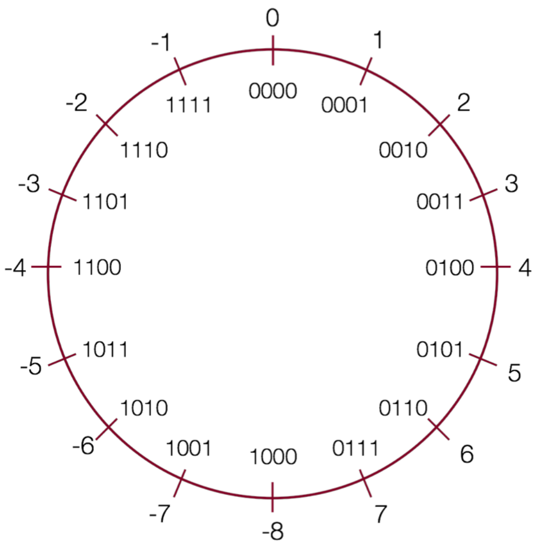
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

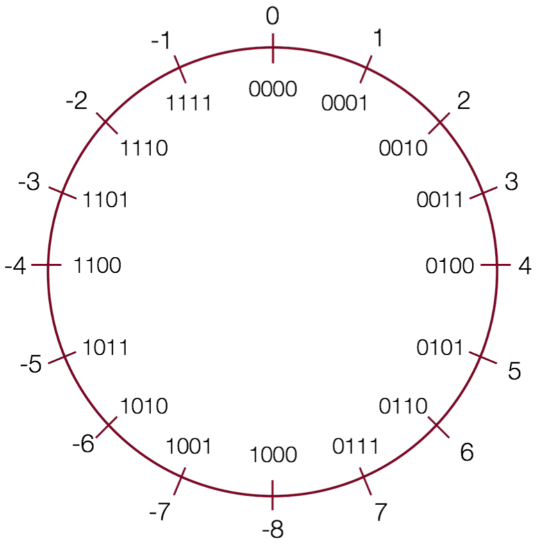
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`	Signed	true	yes
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

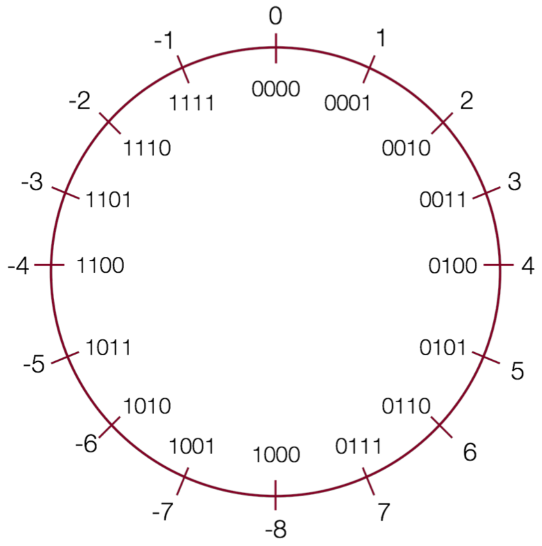
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`	Signed	true	yes
`2147483647U > -2147483647 - 1`	Unsigned	false	no!
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

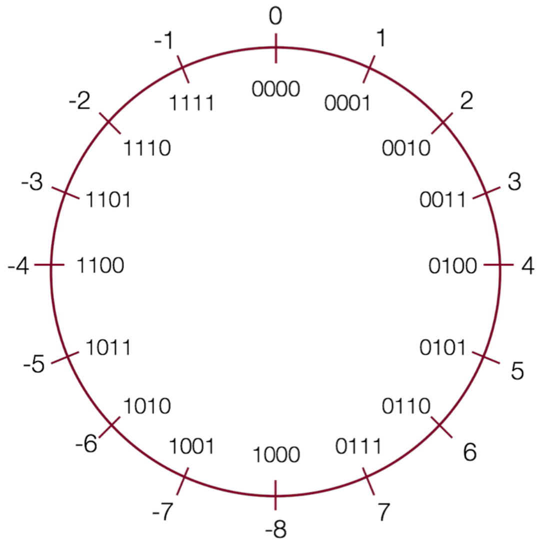
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`	Signed	true	yes
`2147483647U > -2147483647 - 1`	Unsigned	false	no!
`2147483647 > (int)2147483648U`	Signed	true	no!
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

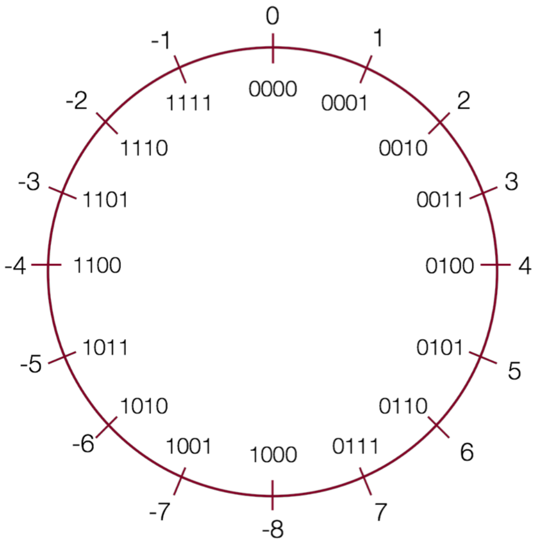
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`	Signed	true	yes
`2147483647U > -2147483647 - 1`	Unsigned	false	no!
`2147483647 > (int)2147483648U`	Signed	true	no!
`-1 > -2`	Signed	true	yes
`(unsigned)-1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



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.

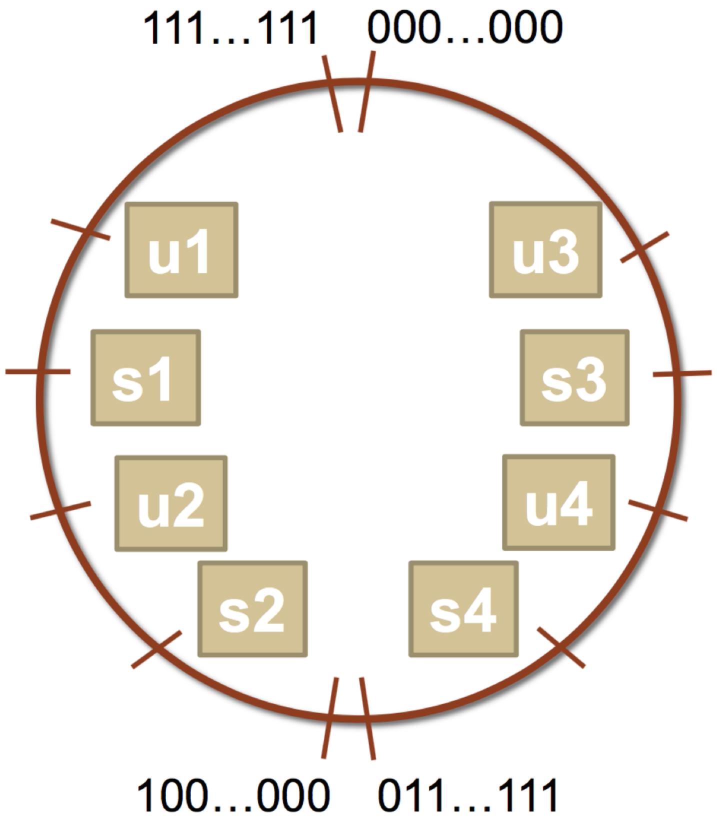
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	no!
`2147483647 > -2147483647 - 1`	Signed	true	yes
`2147483647U > -2147483647 - 1`	Unsigned	false	no!
`2147483647 > (int)2147483648U`	Signed	true	no!
`-1 > -2`	Signed	true	yes
`(unsigned)-1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



Comparisons Between Different Types

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

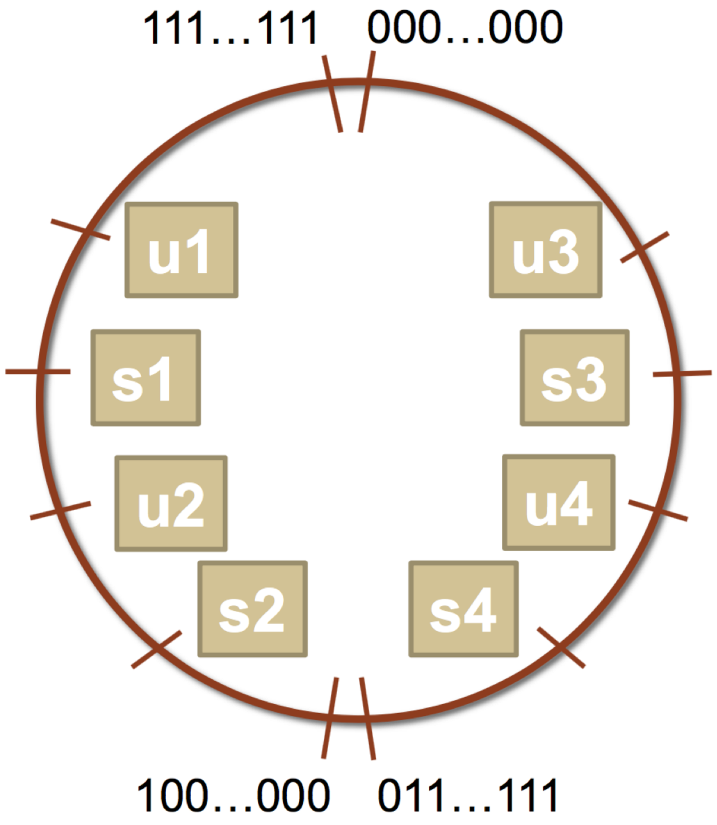
1. `s3 > u3`
2. `u2 > u4`
3. `s2 > s4`
4. `s1 > s2`
5. `u1 > u2`
6. `s1 > u3`



Comparisons Between Different Types

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

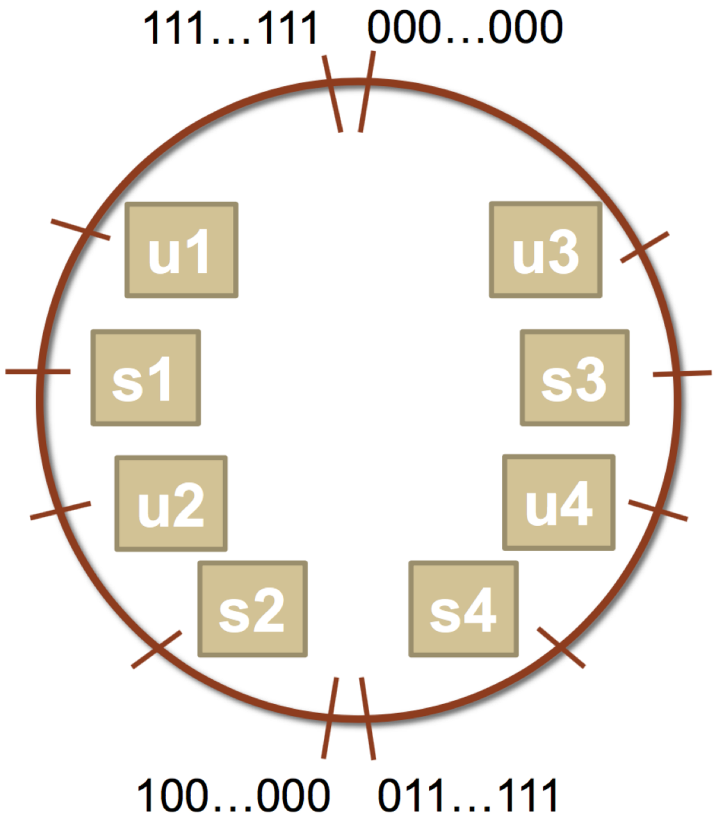
```
1. s3 > u3 - true
2. u2 > u4
3. s2 > s4
4. s1 > s2
5. u1 > u2
6. s1 > u3
```



Comparisons Between Different Types

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

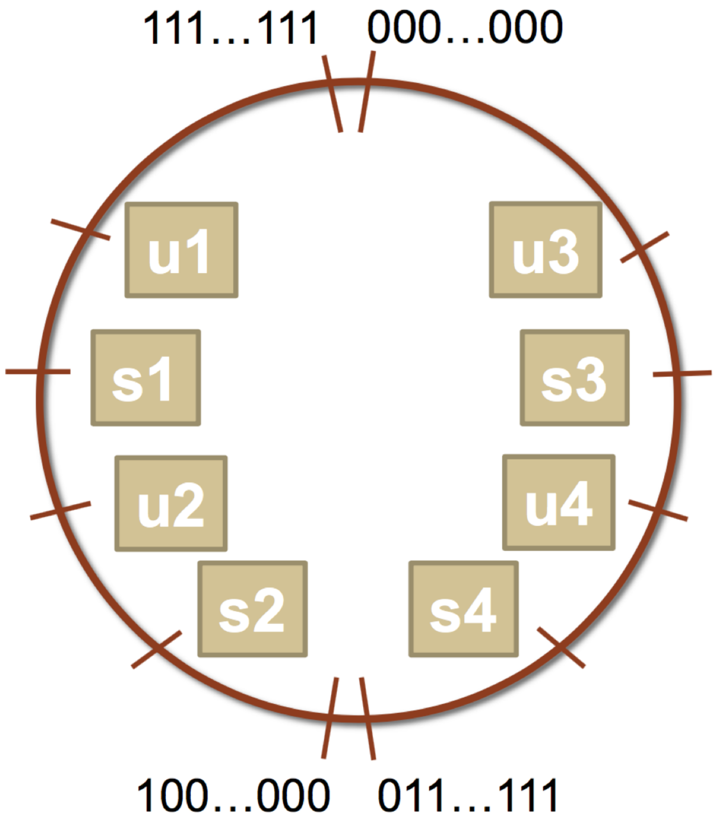
```
1. s3 > u3 - true
2. u2 > u4 - true
3. s2 > s4
4. s1 > s2
5. u1 > u2
6. s1 > u3
```



Comparisons Between Different Types

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

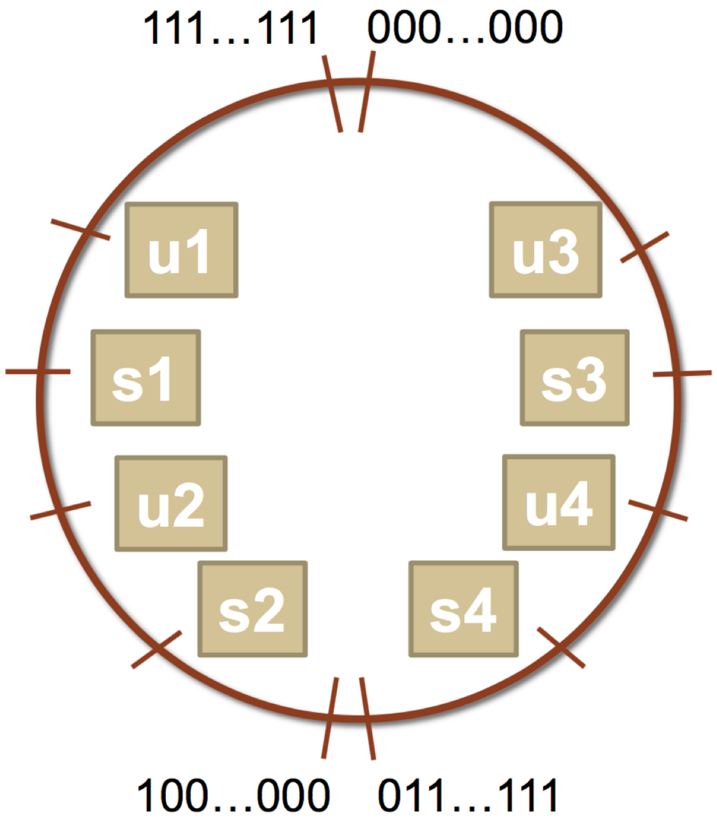
```
1. s3 > u3 - true
2. u2 > u4 - true
3. s2 > s4 - false
4. s1 > s2
5. u1 > u2
6. s1 > u3
```



Comparisons Between Different Types

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

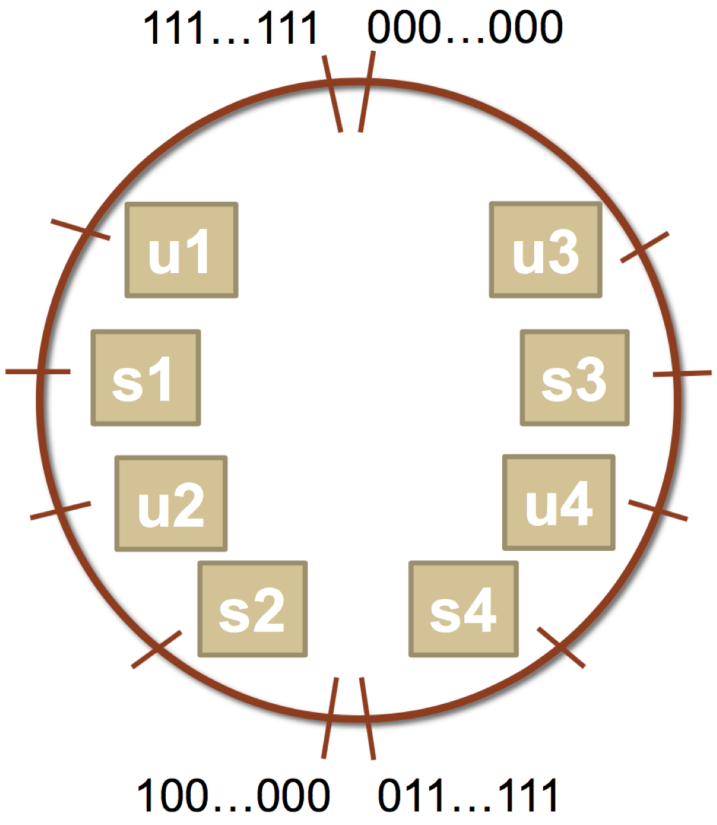
```
1. s3 > u3 - true
2. u2 > u4 - true
3. s2 > s4 - false
4. s1 > s2 - true
5. u1 > u2
6. s1 > u3
```



Comparisons Between Different Types

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

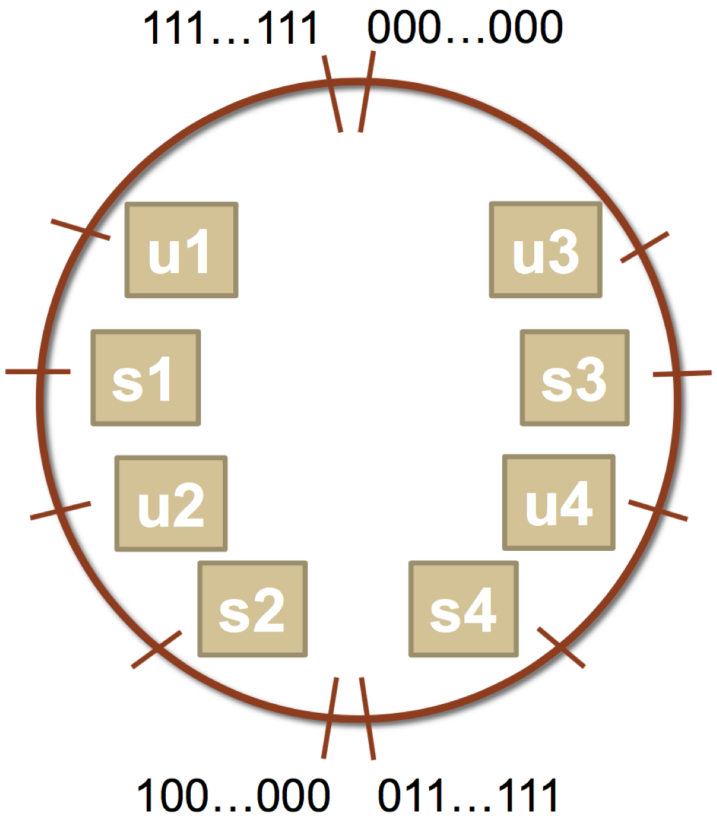
```
1. s3 > u3 - true
2. u2 > u4 - true
3. s2 > s4 - false
4. s1 > s2 - true
5. u1 > u2 - true
6. s1 > u3
```



Comparisons Between Different Types

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

```
1. s3 > u3 - true
2. u2 > u4 - true
3. s2 > s4 - false
4. s1 > s2 - true
5. u1 > u2 - true
6. s1 > u3 - true
```



Plan For Today

- **Recap**: Integer Representations
- **Truncating and Expanding**
- Bitwise Operators and Masks
- **Demo 1:** Courses
- **Break:** Announcements
- **Demo 2:** Powers of 2
- Bit Shift Operators

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 = 32772;
// short is a 16-bit format, so                    s = 1000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 1000 0000 0000 0100b
```

Expanding Bit Representation

```
unsigned short s = 32772;
// short is a 16-bit format, so                      s = 1000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 1000 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 *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`

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

sizeof

sizeof takes a variable type as a parameter and returns its size in bytes.

`sizeof(`*`type`*`)`

For example:

```
sizeof(char) => 1
sizeof(short) => 2
sizeof(int) => 4
sizeof(unsigned int) => 4
sizeof(long) => 8
sizeof(char *) => 8
```

Now that we understand binary representations, how can we manipulate them at the bit level?

Plan For Today

- **Recap**: Integer Representations
- Truncating and Expanding
- **Bitwise Operators and Masks**
- **Demo 1:** Courses
- **Break:** Announcements
- **Demo 2:** Powers of 2
- Bit Shift Operators

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

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

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 (^)

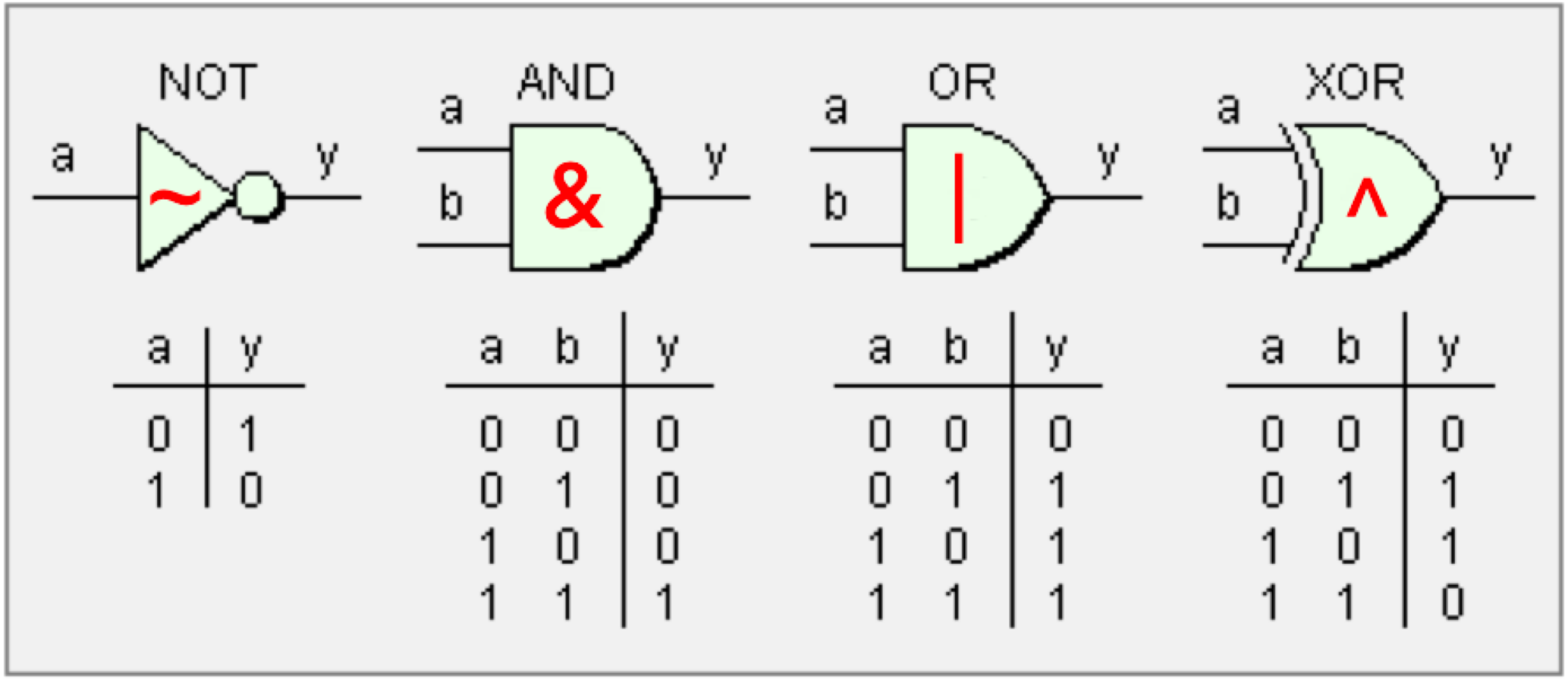
Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if *exactly* one of the bits is 1, or 0 otherwise.

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

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

An Aside: Boolean Algebra

- These operators are not unique to computers; they are part of a general area called **Boolean Algebra**. These are applicable in math, hardware, computers, and more!



NOT

a	y
0	1
1	0

AND

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

a	b	y
0	0	0
0	1	1
1	0	1
1	1	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

```
  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.

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.

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
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

Bit Vectors and Sets

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

- 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
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

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

```
  00100011
& 0110001
  --------
  0010001
```

Bit Masking

- We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A **bitmask** is a constructed bit pattern that we can use, along with bit operators, to do this.
- **Example:** how do we update our bit vector to indicate we’ve taken CS107?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

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

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS106X 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS110  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 CS106X 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS110  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?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  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	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

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

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  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	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

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

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

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

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

Demo: Bitmasks and GDB

Bit Masking

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

- **Example:** If I have a 32-bit integer **`j`**, what operation should I perform if I want to 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.

- **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.

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
```

Plan For Today

- **Recap**: Integer Representations
- Truncating and Expanding
- Bitwise Boolean Operators and Masks
- **Demo 1:** Courses
- **Break: Announcements**
- **Demo 2:** Powers of 2
- Bit Shift Operators

Announcements

- Please send us any OAE letters or athletics conflicts as soon as possible.
- Assignment 0 deadline tonight at 11:59PM PST
- Assignment 1 (Bit operations!) goes out tonight at Assignment 0 deadline
  - Saturated arithmetic
  - Cell Automata
  - Unicode and UTF-8
- Lab 1 this week!

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

Plan For Today

- **Recap**: Integer Representations
- Truncating and Expanding
- Bitwise Boolean Operators and Masks
- **Demo 1:** Courses
- **Break**: Announcements
- **Demo 2:** Powers of 2
- **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 of the end are lost.

```
    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 of the end are lost.

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

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

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

```
short x = 2;  // 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 of the end are lost.

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

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

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

```
short x = -2; // 1111 1111 1111 1110
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 of the end are lost.

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

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

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

Solution: let's fill with the sign bit!

Right Shift (>>)

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

```
x >> 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 of the end are lost.

```
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;
```

Recap

- **Recap**: Integer Representations
- Truncating and Expanding
- Bitwise Boolean Operators and Masks
- **Demo 1:** Courses
- **Break:** Announcements
- **Demo 2:** Powers of 2
- Bit Shift Operators

Next time: *How can a computer represent and manipulate more complex data like text?*

Extra Slides

Demo: Absolute Value