CS107 Lecture 3
Byte Ordering & Bitwise Operators

reading:
Bryant & O'Hallaron, Ch. 2.1

Announcements

- Assign 0 due today, late (without penalty) Sunday
- Assign 1 out and due 7/4, late (with cap penalty) 7/5-7/6
- Office Hours are after lecture M/W/F

Casting Between Signed and Unsigned

Converting between two numbers in C can happen explicitly (using a parenthesized cast), or implicitly (without a cast):

explicit

```
1 int tx, ty;
2 unsigned ux, uy;
3 …
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

implicit

```
1 int tx, ty;
2 unsigned ux, uy;
3 …
4 tx = ux; // cast to signed
5 uy = ty; // cast to unsigned
```

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is.

NOTE: Converting a signed number to unsigned preserves the bits not the number!

Casting Between Signed and Unsigned

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is.
You cannot convert a signed number to its unsigned counterpart using a cast!

```
 1 // test_cast.c
 2 #include<stdio.h>
 3 #include<stdlib.h>
 4
 5 int main() {
 6     int v = -12345;
 7     unsigned int uv = (unsigned int) v;
 8
 9     printf("v = %d, uv = %u\n",v,uv);
10
11     return 0;
12 }
```

```
$ ./test_cast
v = -12345, uv = 4294954951
```

Signed -> Unsigned
-12345 goes to 4294954951

Not 12345

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?**

IMPORTANT NOTE

Because Types are just about how we read memory, it is important to note that casting does not impact the values or bits only the meaning that we expect them to have

Casting Between Signed and Unsigned

`printf` has three 32-bit integer representations:

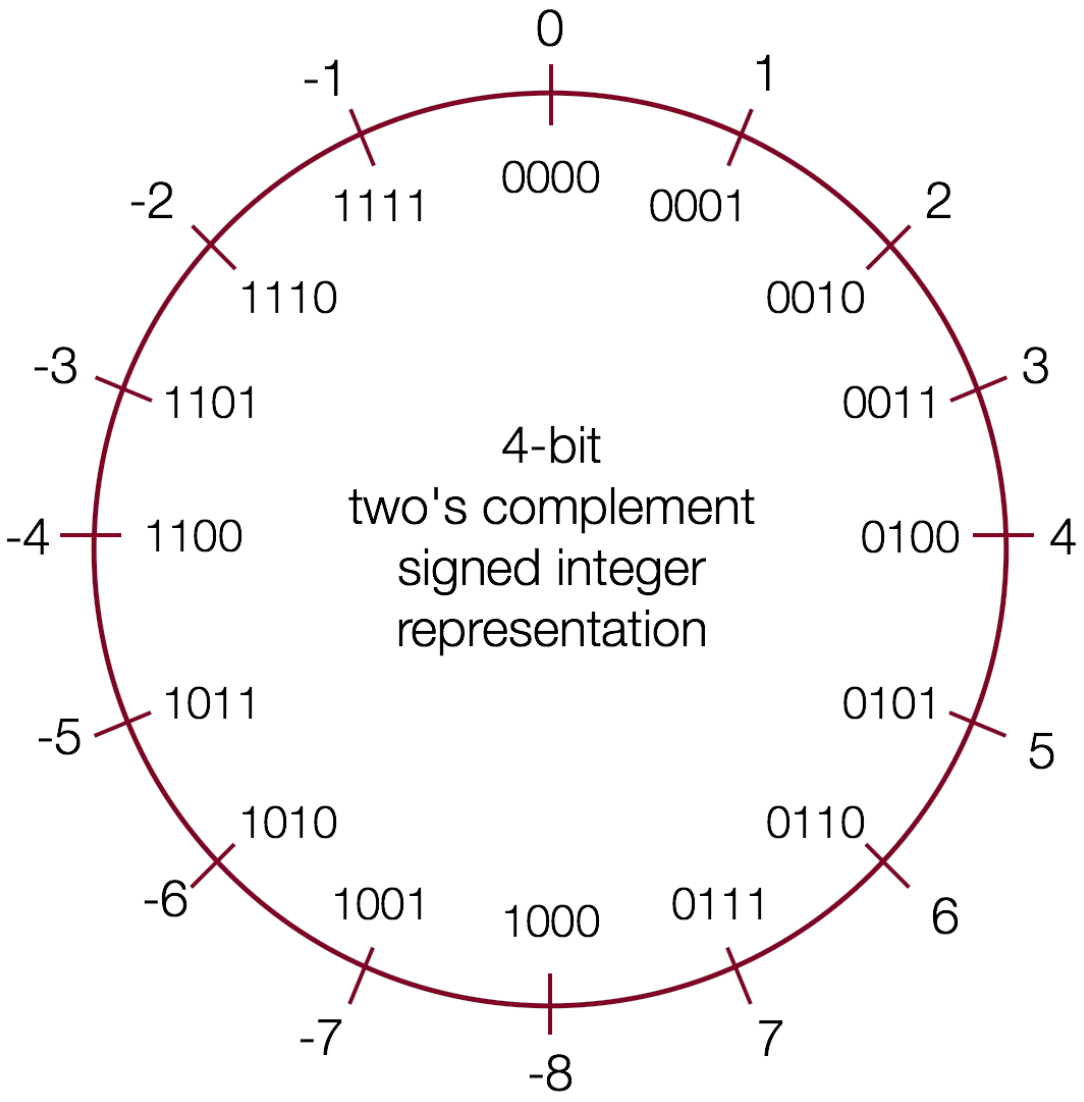
`%d` : signed 32-bit int
`%u` : unsigned 32-bit int
`%x` : hex 32-bit int

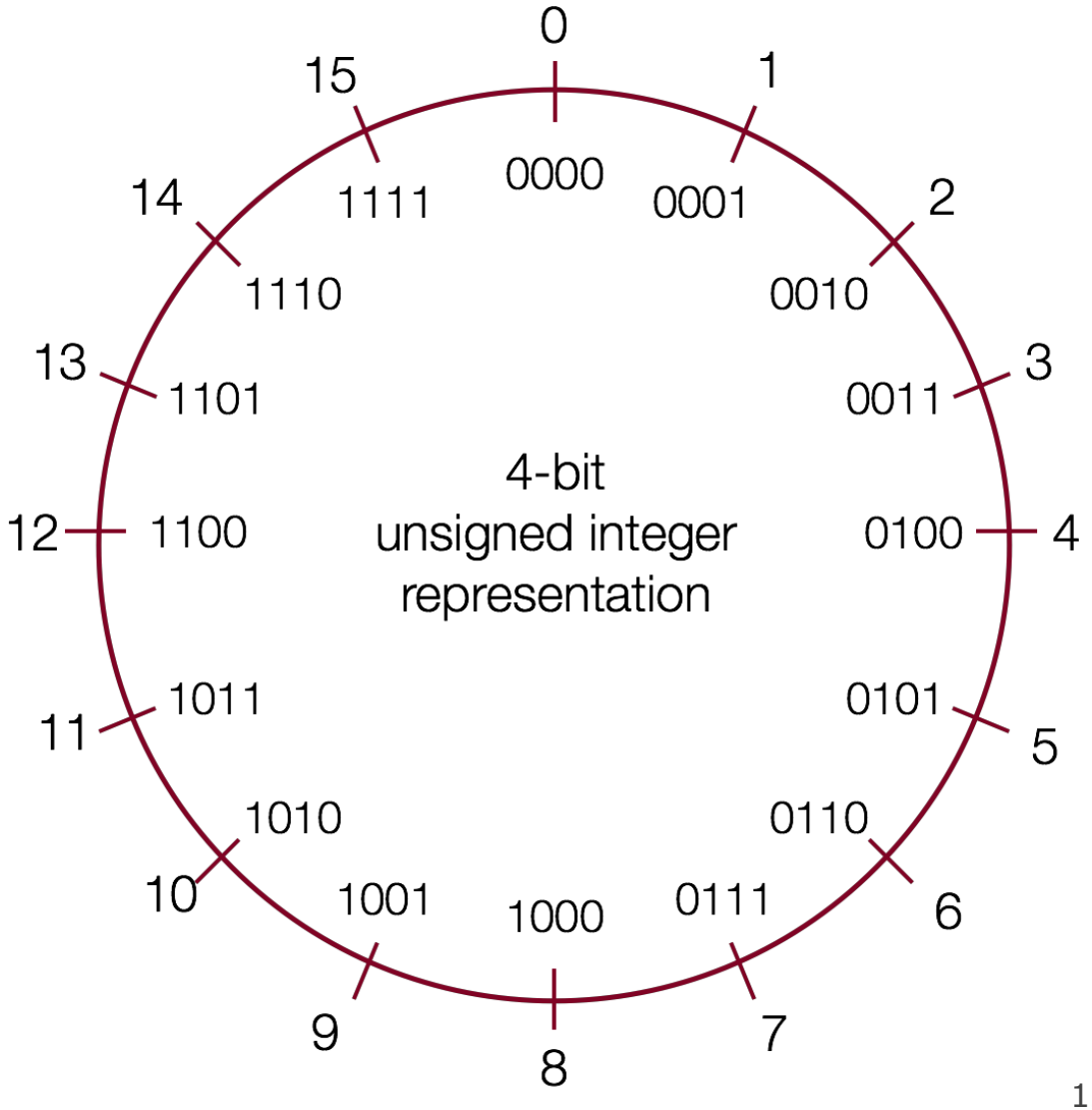
As long as the value is a 32-bit type, printf will treat it according to the formatter it is applying:

```
1    int x = -1;
2    unsigned u = 3000000000; // 3 billion
3
4    printf("x = %u = %d\n", x, x);
5    printf("u = %u = %d\n", u, u);
6
```

```
$ ./test_printf
x = 4294967295 = -1
u = 3000000000 = -1294967296
```

Casting

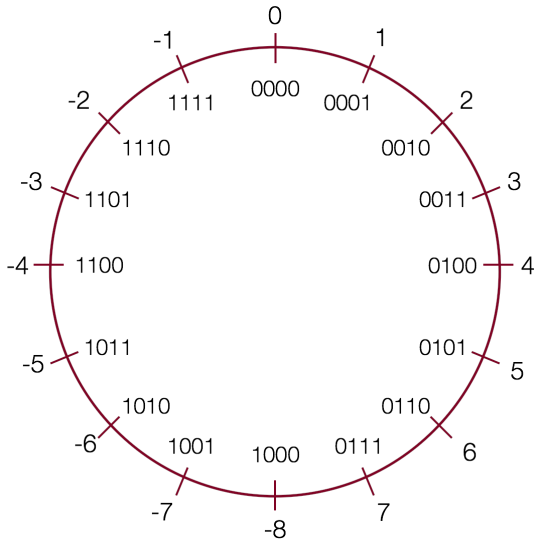

0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
two's complement
signed integer
representation


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

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.

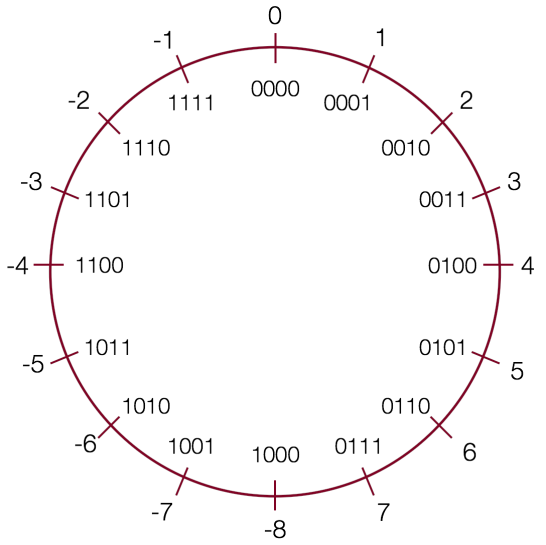
Expression	Type	Evaluation	Correct?
`0 == 0U`			
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

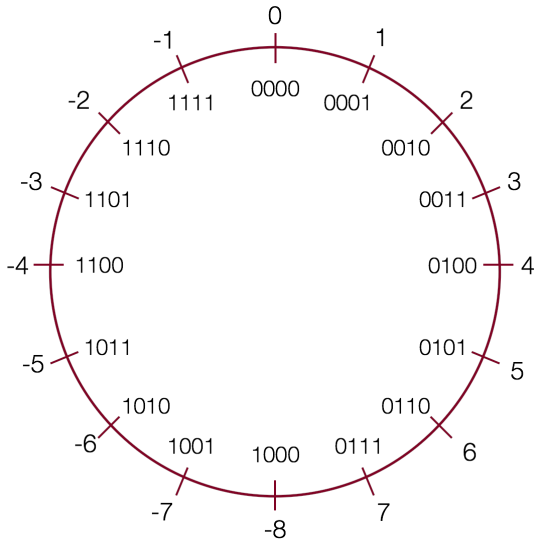
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

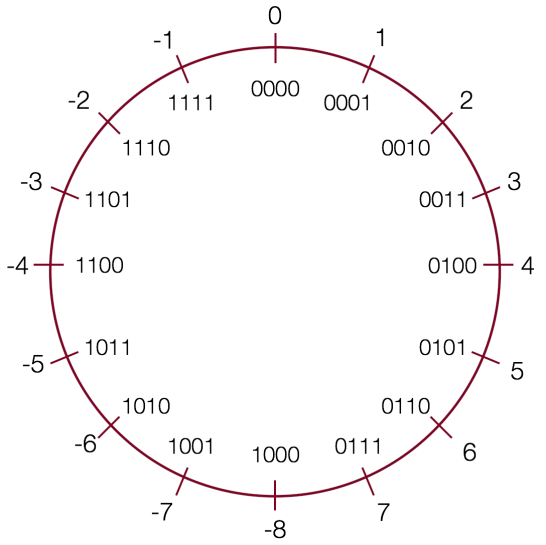
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`			
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

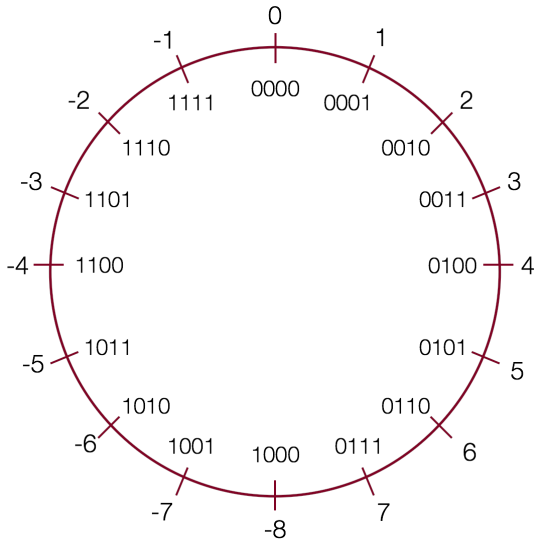
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`			
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

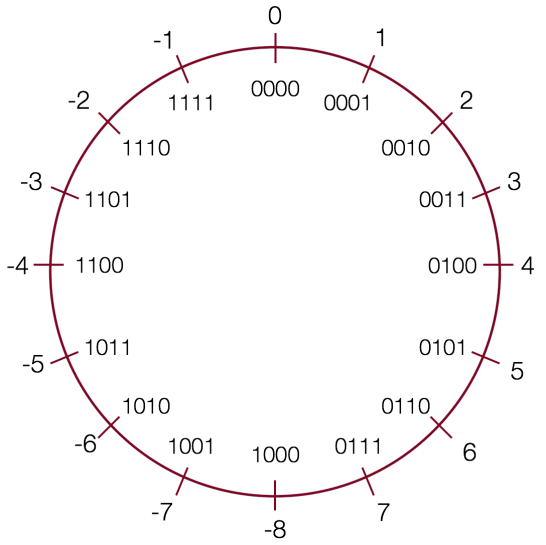
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`			
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

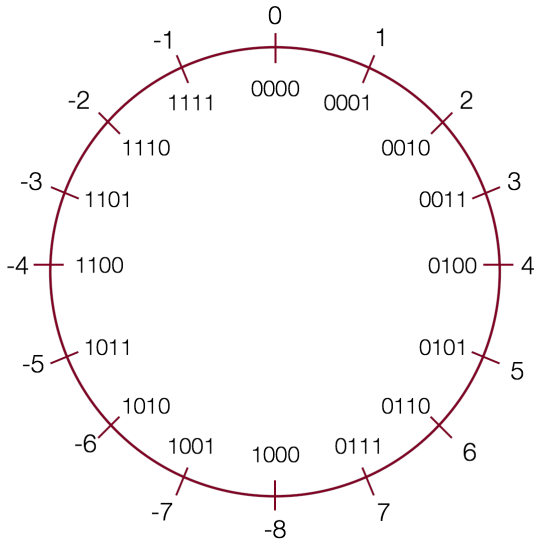
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

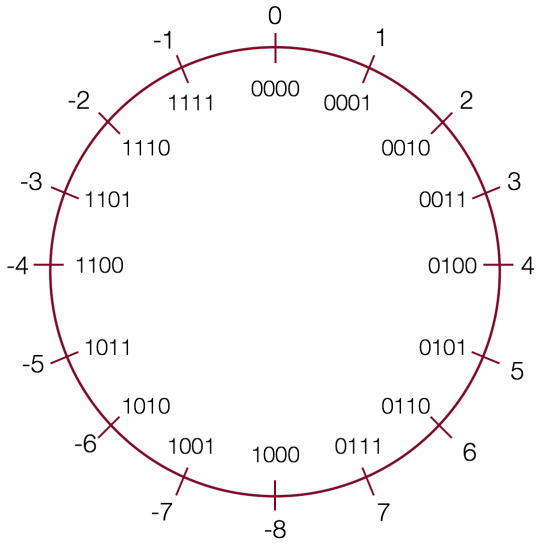
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	No!
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

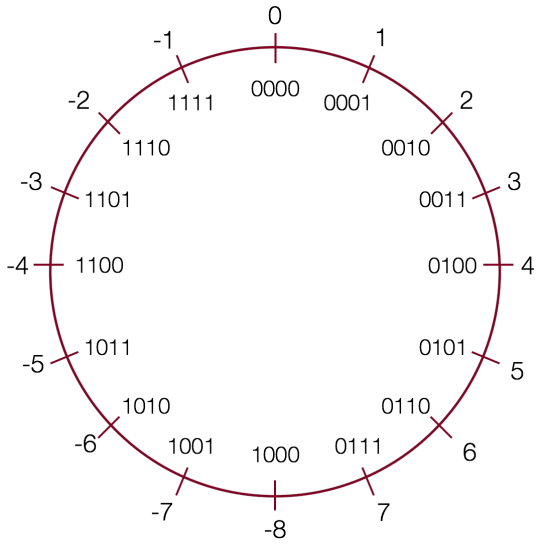
Expression	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	No!
`-1 > -2`	Signed	1	yes
`(unsigned)-1 > -2`			



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



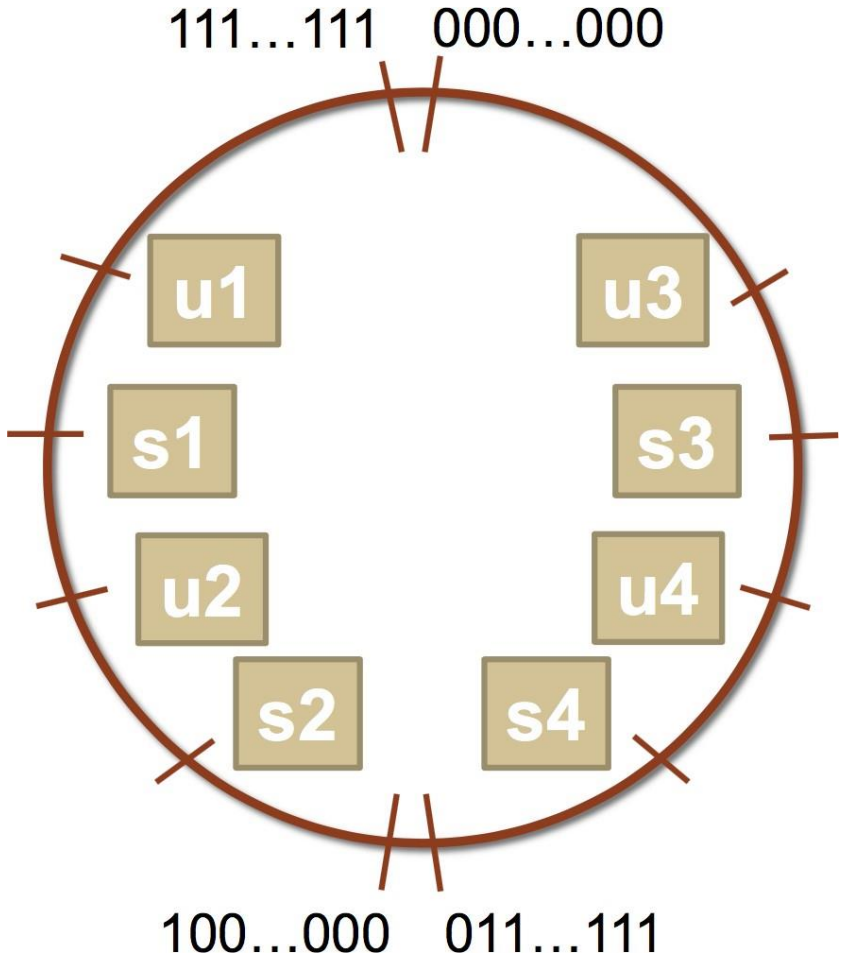
Comparison between signed and unsigned integers

Let's try some more…a bit more abstractly.

```
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

What is the value of this expression?

```
u1 > s3
```

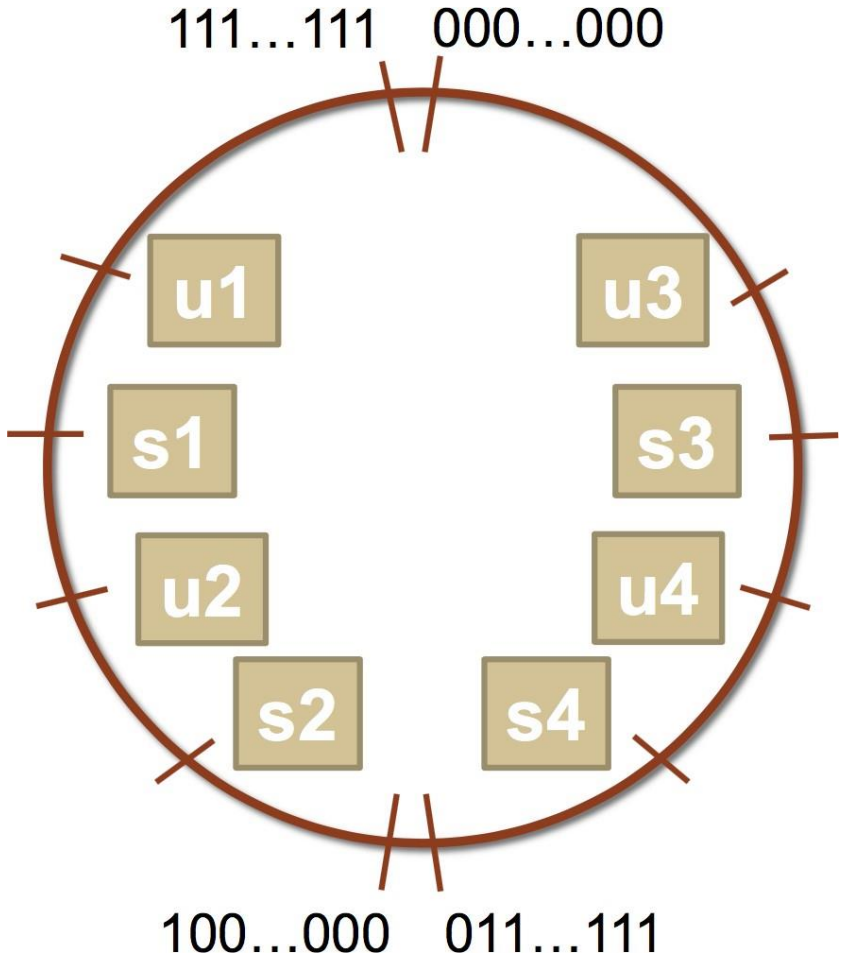


Comparison between signed

Let's try some more…a bit more abstractly.

```
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

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

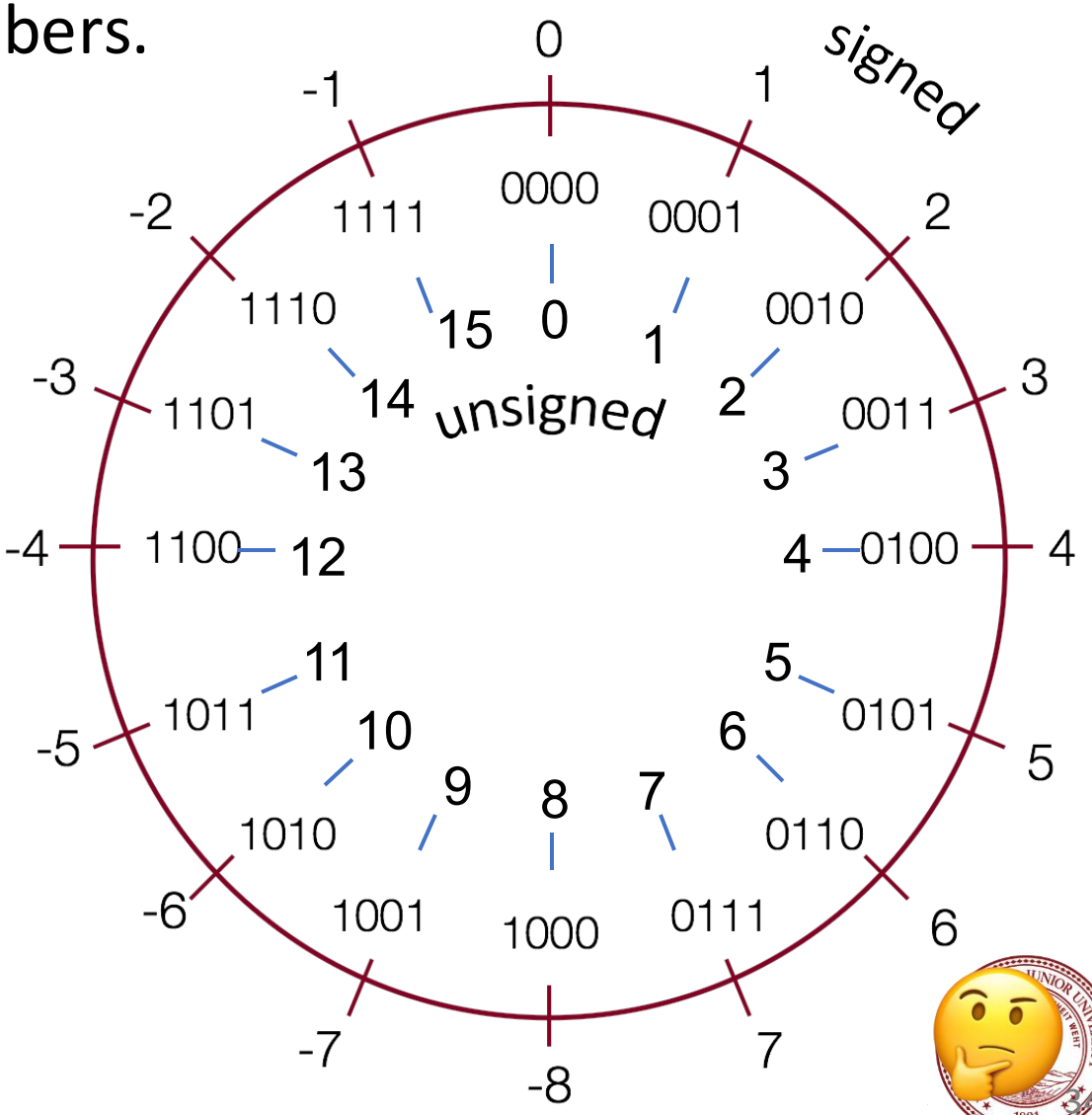



```
u1 > s3 :  true
```

Overflow

- What is happening here? Assume 4-bit numbers.

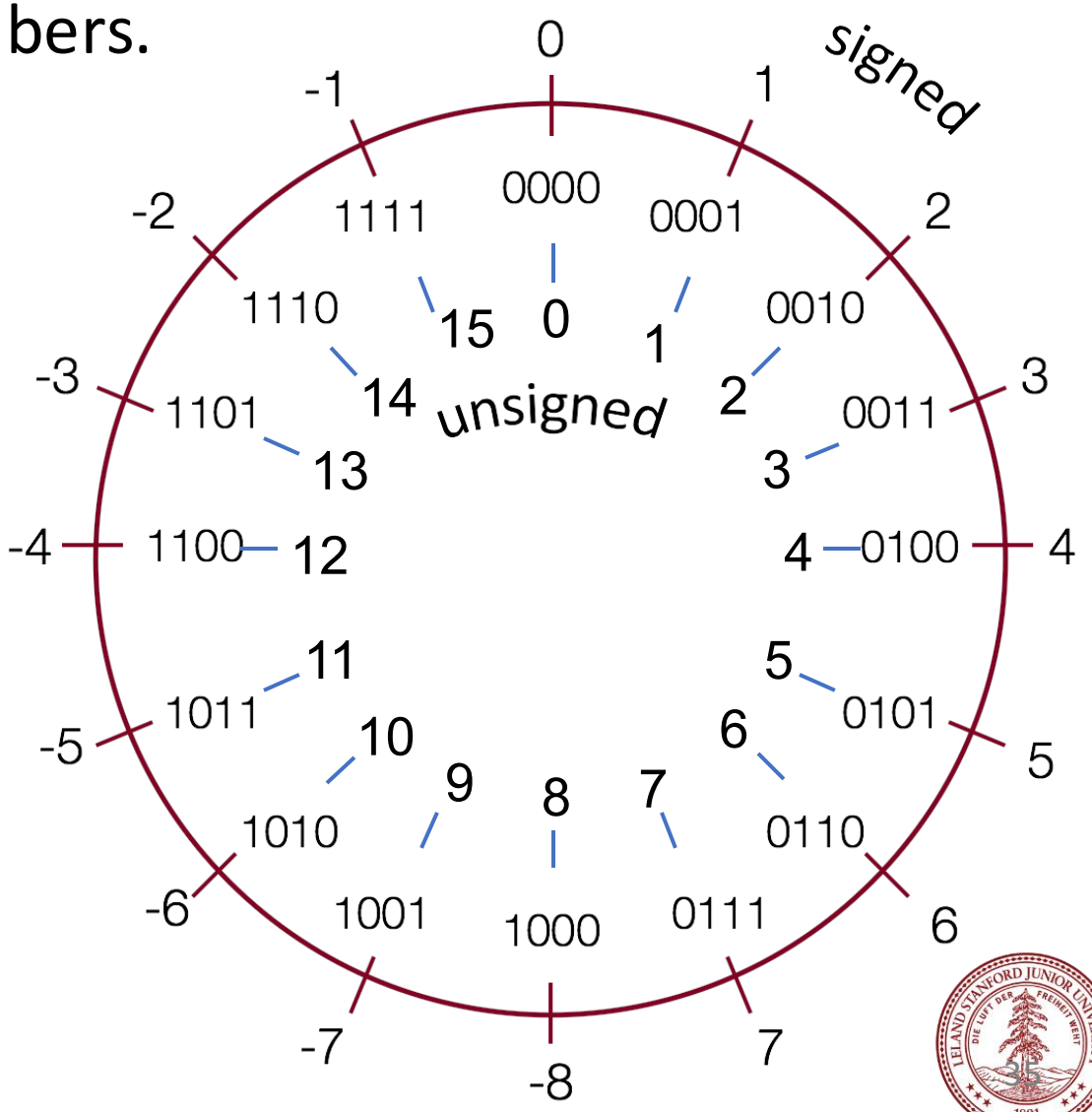
```
  0b1101
+ 0b0100
---------
```



Overflow

- What is happening here? Assume 4-bit numbers.

```
  0b1101
+ 0b0100
--------
```



Signed	Unsigned
-3 + 4 = 1	13 + 4 = 1
No overflow	Overflow

Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`			
`int`			

2. Will the following `char` comparisons evaluate to `true` or `false`?

i. `-7 < 4`

ii. `-7 < 4U`

iii. `(char) 130 > 4`

iv. `(char) -132 > 2`

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.

The `sizeof` Operator

As we have seen, integer types are limited by the number of bits they hold. On the 64-bit myth machines, we can use the `sizeof` operator to find how many bytes each type uses:

```
int main() {
    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    return 0;
}
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

Type	Width in bytes	Width in bits
char	1	8
short	2	16
int	4	32
long	8	64
void *	8	64

MIN and MAX values for integers

Because we now know how bit patterns for integers works, we can figure out the maximum and minimum values, designated by `INT_MAX`, `UINT_MAX`, `INT_MIN`, (etc.), which are defined in `limits.h`

Type	Width (bytes)	Width (bits)	Min in hex (name)	Max in hex (name)
char	1	8	80 (CHAR_MIN)	7F (CHAR_MAX)
unsigned char	1	8	0	FF (UCHAR_MAX)
short	2	16	8000 (SHRT_MIN)	7FFF (SHRT_MAX)
unsigned short	2	16	0	FFFF (USHRT_MAX)
int	4	32	80000000 (INT_MIN)	7FFFFFFF (INT_MAX)
unsigned int	4	32	0	FFFFFFFF (UINT_MAX)
long	8	64	8000000000000000 (LONG_MIN)	7FFFFFFFFFFFFFFF (LONG_MAX)
unsigned long	8	64	0	FFFFFFFFFFFFFFFF (ULONG_MAX)

- You can also find constants in the standard library that define the max and min for each type on that machine(architecture)

- Visit <limits.h> or <cstdint.h> and look for variables like:

```
INT_MIN
INT_MAX
UINT_MAX
LONG_MIN
LONG_MAX
ULONG_MAX
...
```

Expanding Bit Representations

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

Expanding Bit Representation

```
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 *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

Now that we understand values are really stored in binary, how can we manipulate them at the bit level?

Bitwise Operators

- You're already familiar with many operators in C:
  - **Arithmetic operators:** +, -, *, /, %
  - **Comparison operators:** ==, !=, <, >, <=, >=
  - **Logical Operators**: &&, ||, !
- Today, we're introducing a new category of operators: **bitwise operators:**
  - &, |, ~, ^, <<, >>

And (&)

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

```
output = a & b;
```

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
```

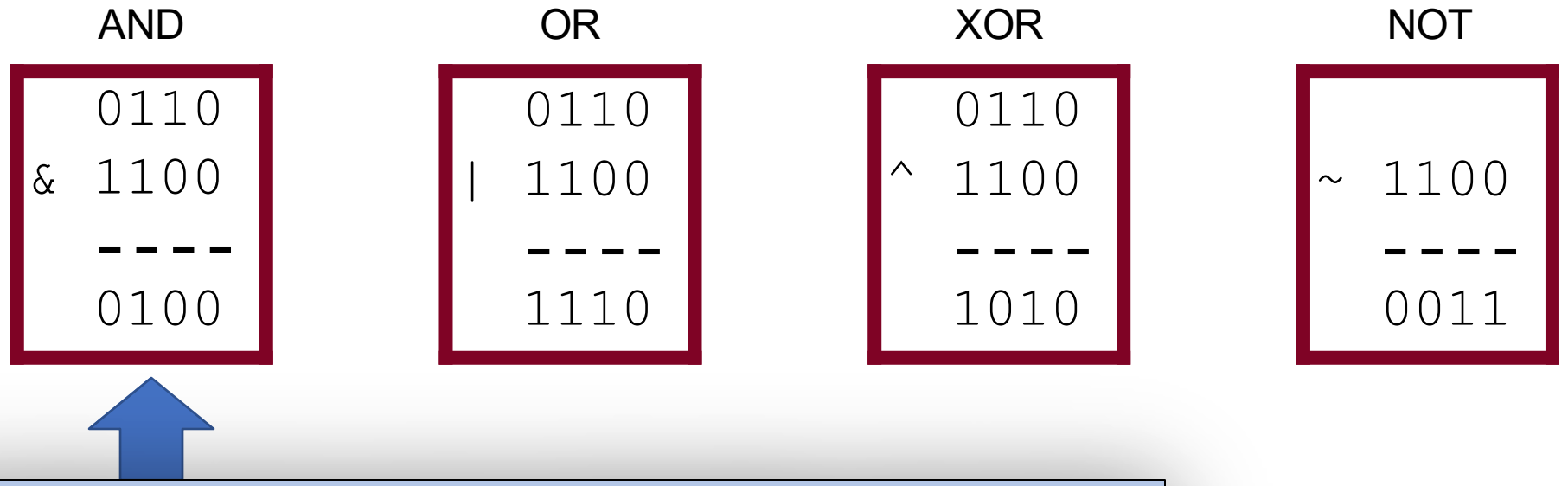
NOT

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

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

Operators on Multiple Bits

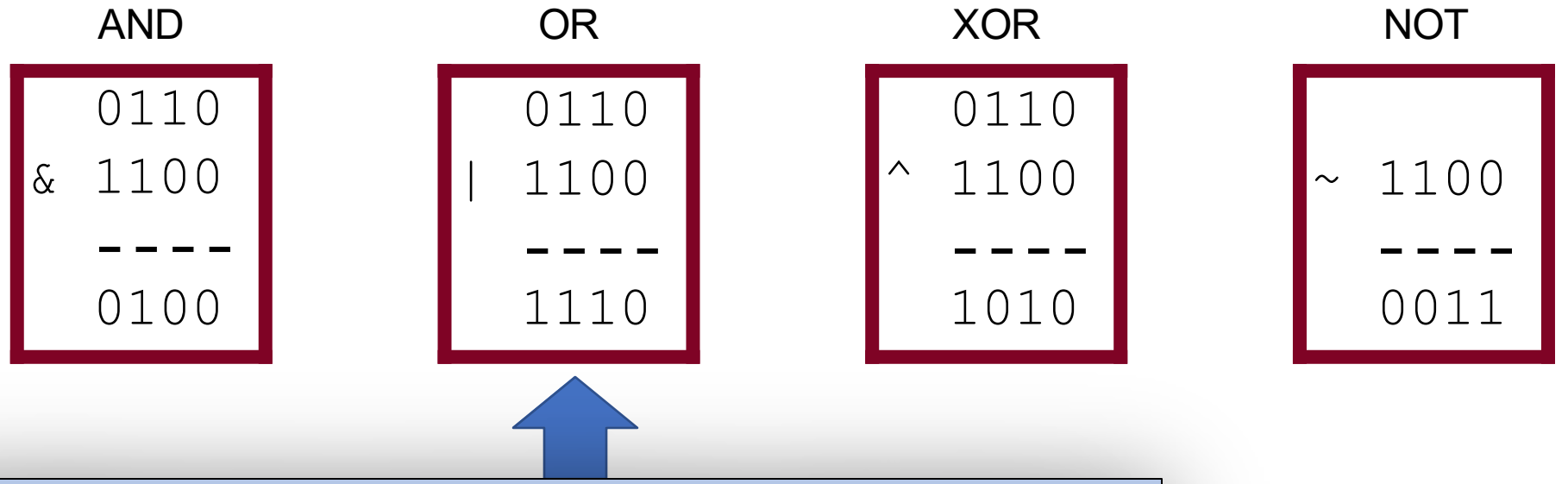
- When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:



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:



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).

Demo: Bits Playground

Bitmasks

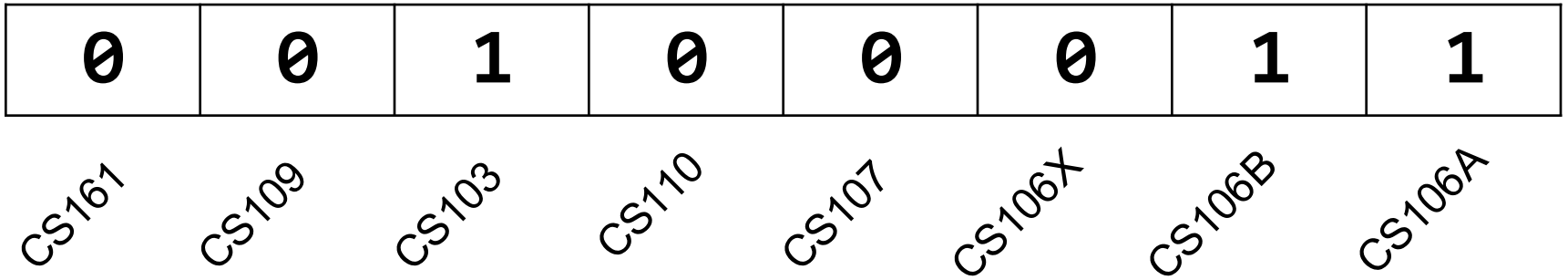
We will frequently want to manipulate or otherwise isolate specific bits in a larger collection of them. A **bitmask** is a constructed bit pattern that we can use, along with standard bit operators like **&**, **|**, **^**, **~**, **<<**, and **>>**, to do this.

Motivating Example: Bit vectors

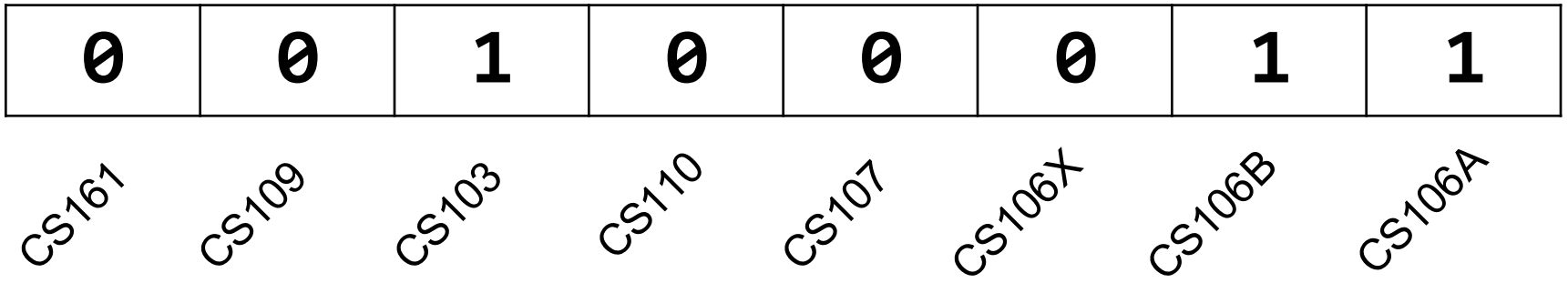
Aside: C++ relies on bit vectors to efficiently implement **vector<bool>**.

Bit Vectors and Sets

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



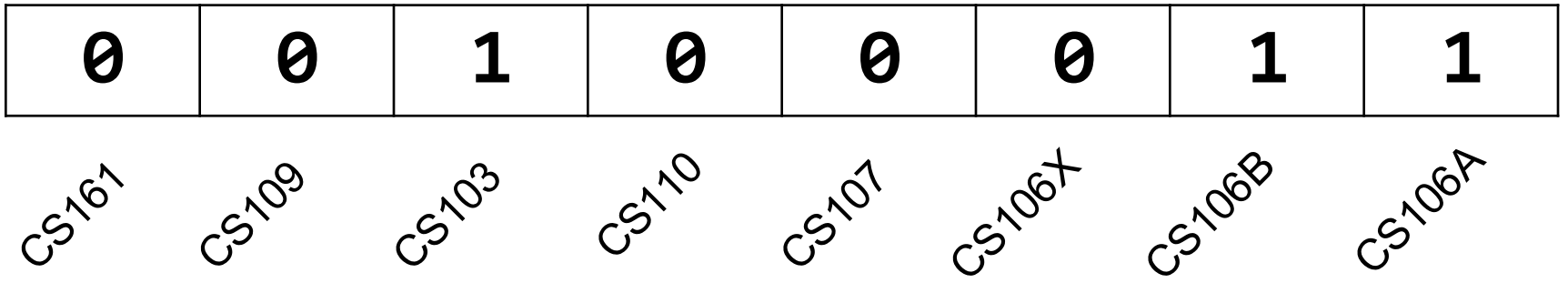
Bit Vectors and Sets



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

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

Bit Vectors and Sets

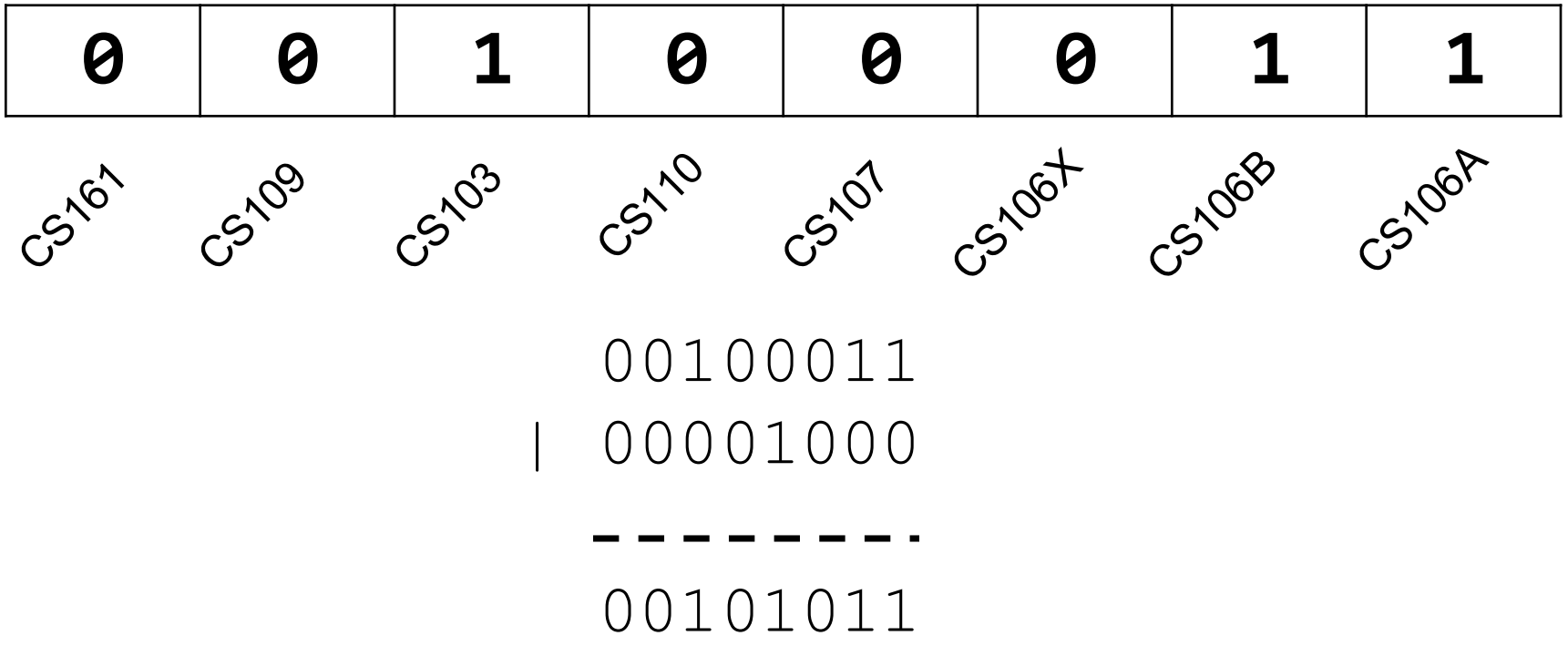


- How do we find the intersection of two sets of courses taken? Use AND:

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

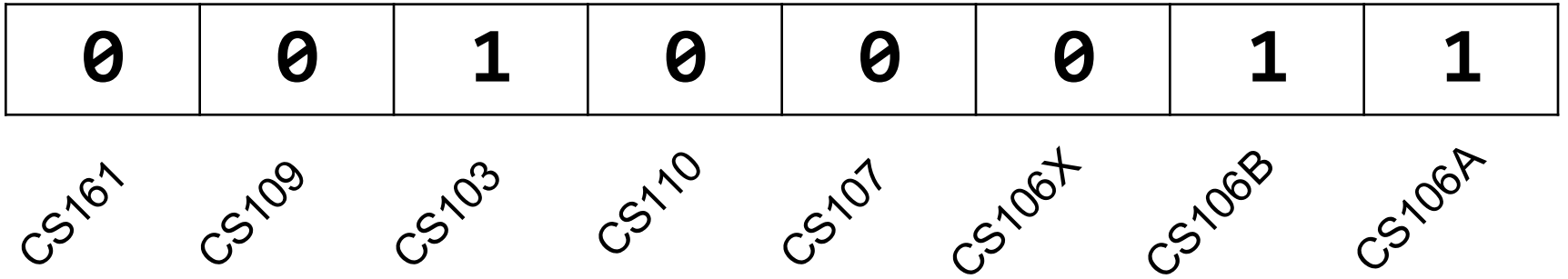
Bit Masking

```
#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?

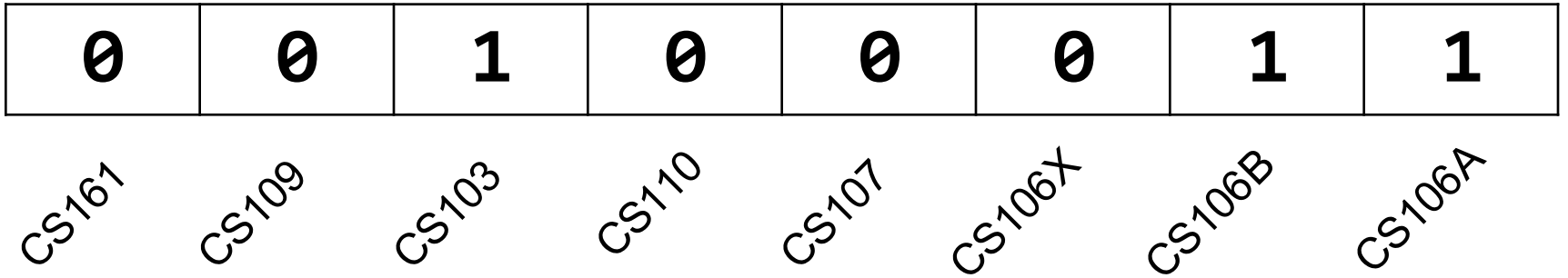



```
  00100011
& 11011111
----------
  00000011
```

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

Bit Masking

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

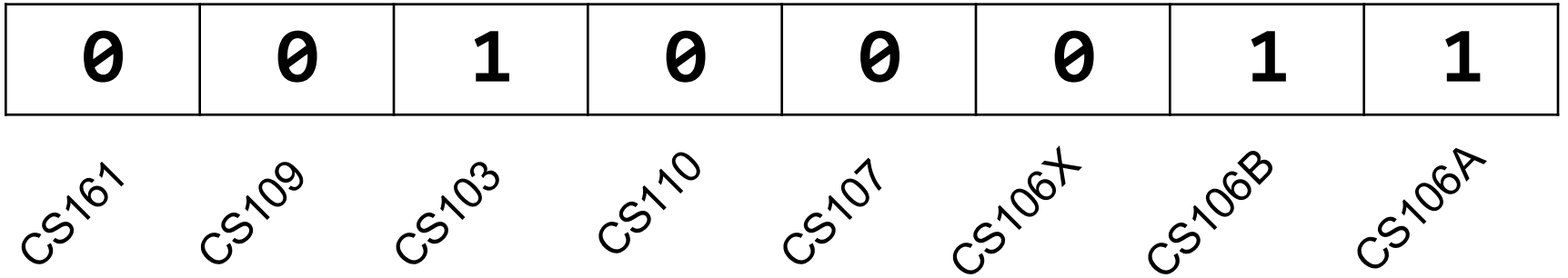



```
  00100011
& 11011111
----------
  00000011
```

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

Bit Masking

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

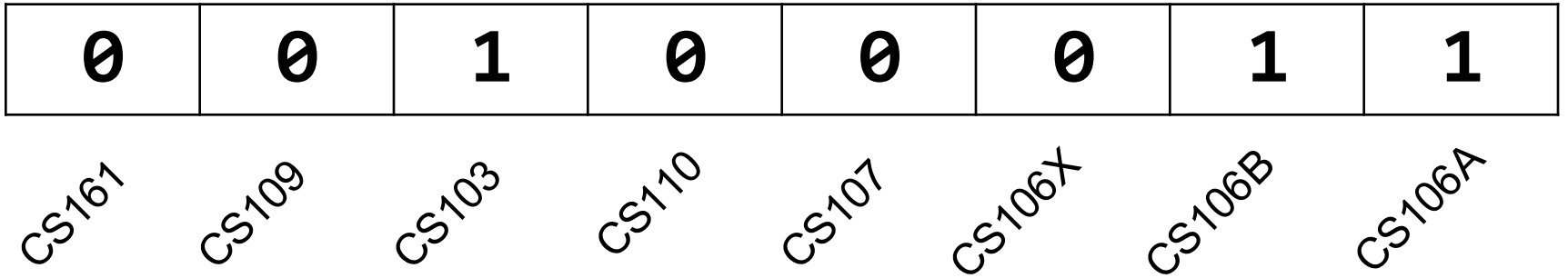



```
  00100011
& 00000010
----------
  00000010
```

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

Bit Masking

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




```
  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 isolated bits
- ~ is useful for flipping all bits

Introducing GDB

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

The GDB Debugger

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

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

GDB as an Interpreter

- `gdb live_session` run gdb on live_session executable
- `p` print variable (`p varname`) or evaluated expression (`p 3L << 10`)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `<enter>` Execute last command again
- `q` Quit gdb

Important When first launching gdb:

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

gdb on a program

- `gdb live_session` run gdb on executable
- `b` Set breakpoint on a function (e.g., `b main`) or line (`b 42`)
- `r 82` Run with provided args
- `n, s, continue` control forward execution (next, step into, continue)
- `p` print variable (`p varname`) or evaluated expression (`p 3L << 10`)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `info` `args, locals`

Important: gdb does not run the current line until you hit “next”

Demo: Bitmasks and GDB

gdb: highly recommended

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

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

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

Gdb takes practice! But the payoff is tremendous! ©

gdb step, next, finish

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

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

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

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

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

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

Bit Masking

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

Example: If I have a 32-bit integer **`j`**, what operation should I perform if I want to get *just the lowest byte* in **`j`**?

```
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 the last byte.

Practice: Bit Masking

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

```
j | 0xff
```

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

```
j ^ ~0xff
```

Powers of 2

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

Code: Powers of 2

```
bool is_power_of_2(unsigned long num){
        return (num != 0) && ((num & (num -1)) == 0)
}
```

Left Shift (<<)

The LEFT SHIFT operator shifts a bit pattern a certain number of positions to the left. New lower order bits are filled in with 0s, and bits shifted off the end are lost.

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

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

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

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

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

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;
```

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
```

More Exercises

Suppose we have a 64-bit number.

```
long x = 0b1010010;
```

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

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

More Exercises

Suppose we have a 64-bit number.

```
long x = 0b1010010;
```

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

`x | (1L << i)`

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

`x & (-1L << i)`

On your own

- Print a variable
- Print (in binary, then in hex) result of left-shifting 14 and 32 by 4 bits.
- Print (in binary, then in hex) result of subtracting 1 from 128

`1 << 32`

- Why is this zero? Compare with `1 << 31`.
- Print in hex to make it easier to count zeros.

References and Advanced Reading

- **References:**
  - Two's complement calculator: http://www.convertforfree.com/twos-complement-calculator/
  - Wikipedia on Two's complement: https://en.wikipedia.org/wiki/Two%27s_complement
  - The `sizeof` operator: http://www.geeksforgeeks.org/sizeof-operator-c/

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

References and Advanced Reading

- **References:**
  - argc and argv: http://crasseux.com/books/ctutorial/argc-and-argv.html
  - The C Language: https://en.wikipedia.org/wiki/C_(programming_language)
  - Kernighan and Ritchie (K&R) C: https://www.youtube.com/watch?v=de2Hsvxaf8M
  - C Standard Library: http://www.cplusplus.com/reference/clibrary/
  - https://en.wikipedia.org/wiki/Bitwise_operations_in_C
  - http://en.cppreference.com/w/c/language/operator_precedence

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