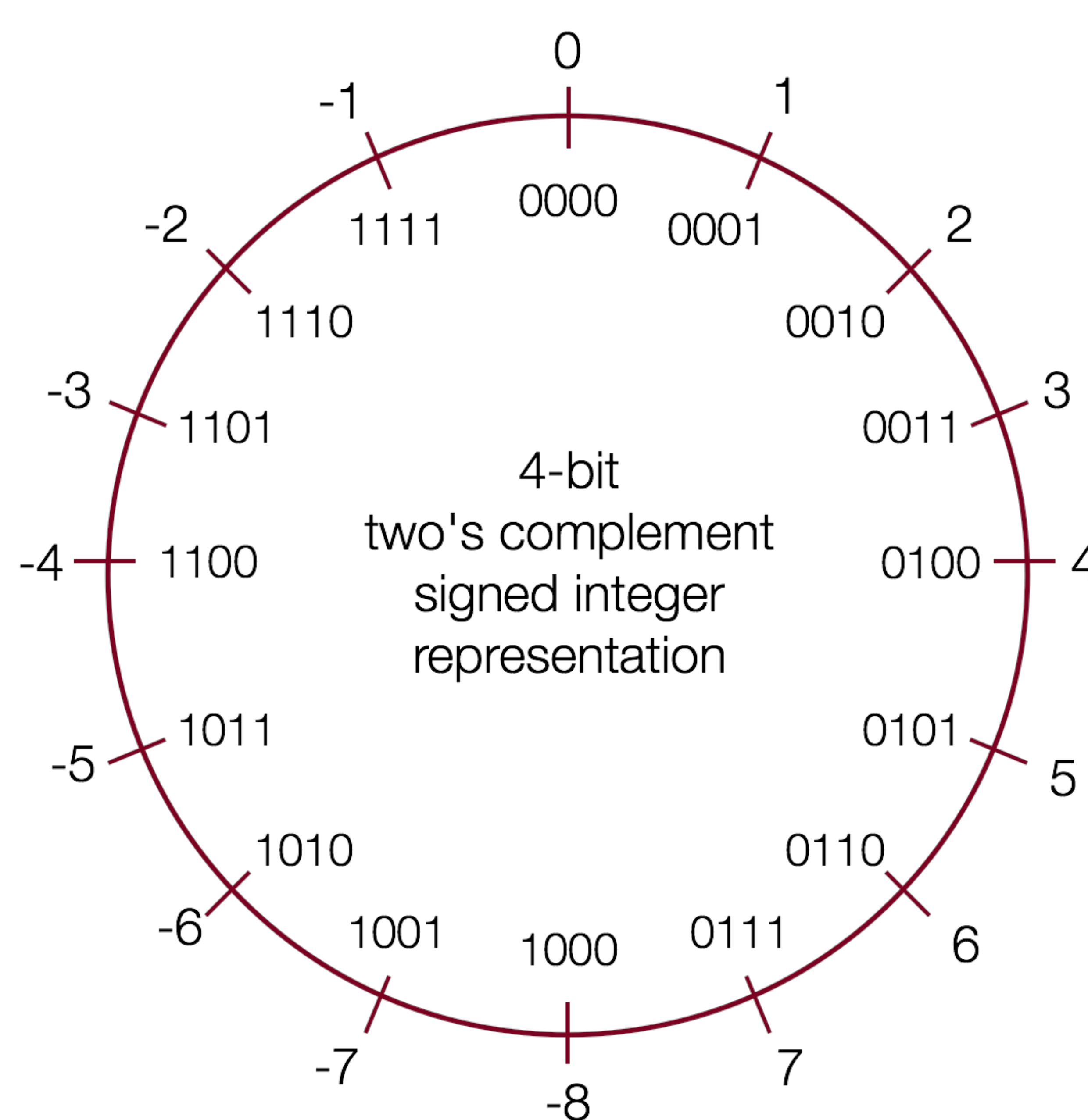
CS 107
Lecture 2: Integer Representations and Bits / Bytes

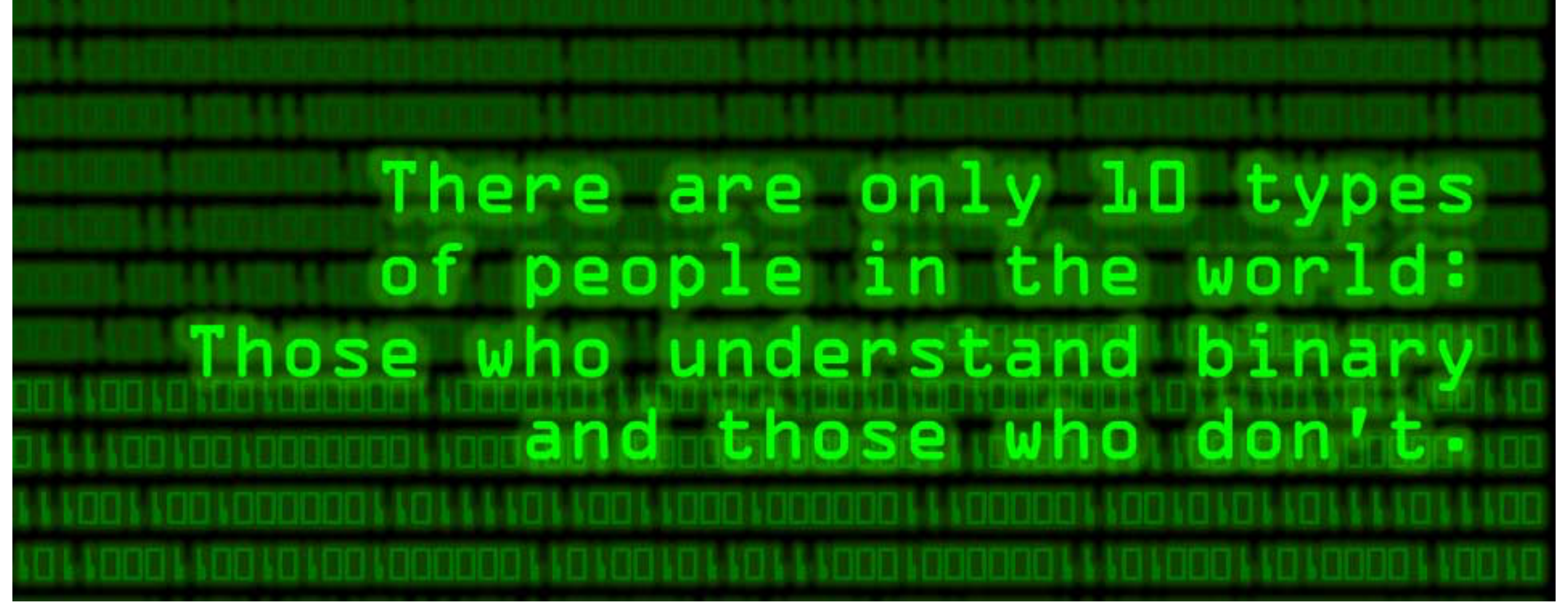
Friday, January 13, 2023



Computer Systems
Winter 2023
Stanford University
Computer Science Department

Reading: Reader: Bits and Bytes, Textbook: Chapter 2.2

Lecturer: Chris Gregg



Today's Topics

- Logistics
  - Assign0 — Due Monday
  - Labs start Wednesday
  - Office hours in full coverage
- Reading: Reader: Bits and Bytes, Textbook: Chapter 2.2 (very mathy…)
- Integer Representations
  - Unsigned numbers
  - Signed numbers
    - two's complement
  - Signed vs Unsigned numbers
  - Casting in C
  - Signed and unsigned comparisons
  - The `sizeof` operator
  - Min and Max integer values
  - Truncating integers
  - two's complement overflow

Today's Topics

- More on extending the bit representation of numbers
- Truncating numbers
- Data Sizes
- Addressing and Byte Ordering
- Boolean Algebra

Integer Representations

Integer Representations

The C language has two different ways to represent numbers, unsigned and signed:

unsigned: can only represent non-negative numbers

signed: can represent negative, zero, and positive numbers

We are going to talk about these representations, and also about what happens when we expand or shrink an encoded integer to fit into a different type (e.g., `int` to `long`)

Unsigned Integers

For positive (unsigned) integers, there is a 1-to-1 relationship between the decimal representation of a number and its binary representation. If you have a 4-bit number, there are 16 possible combinations, and the unsigned numbers go from 0 to 15:

```
0b0000 = 0      0b0001 = 1      0b0010 = 2      0b0011 = 3
0b0100 = 4      0b0101 = 5      0b0110 = 6      0b0111 = 7
0b1000 = 8      0b1001 = 9      0b1010 = 10     0b1011 = 11
0b1100 = 12     0b1101 = 13     0b1110 = 14     0b1111 = 15
```

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits in our integer. For example, a 32-bit `int` can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.

Signed Integers: How do we represent them?

What if we want to represent negative numbers? We have choices!

One way we could encode a negative number is simply to designate some bit as a "sign" bit, and then interpret the rest of the number as a regular binary number and then apply the sign. For instance, for a four-bit number:

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

This might be okay...but we've only represented 14 of our 16 available numbers...

Signed Integers: How do we represent them?

0 001 = 1
0 010 = 2
0 011 = 3
0 100 = 4
0 101 = 5
0 110 = 6
0 111 = 7

1 001 = -1
1 010 = -2
1 011 = -3
1 100 = -4
1 101 = -5
1 110 = -6
1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Signed Integers: How do we represent them?

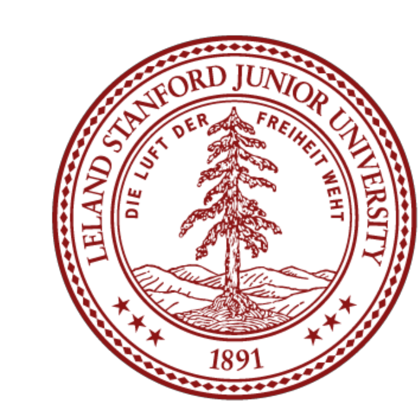
0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.



Signed Integers: How do we represent them?

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

What about 0 000 and 1 000? What should they represent?

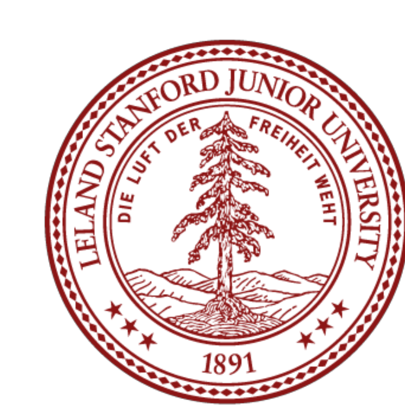
Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.

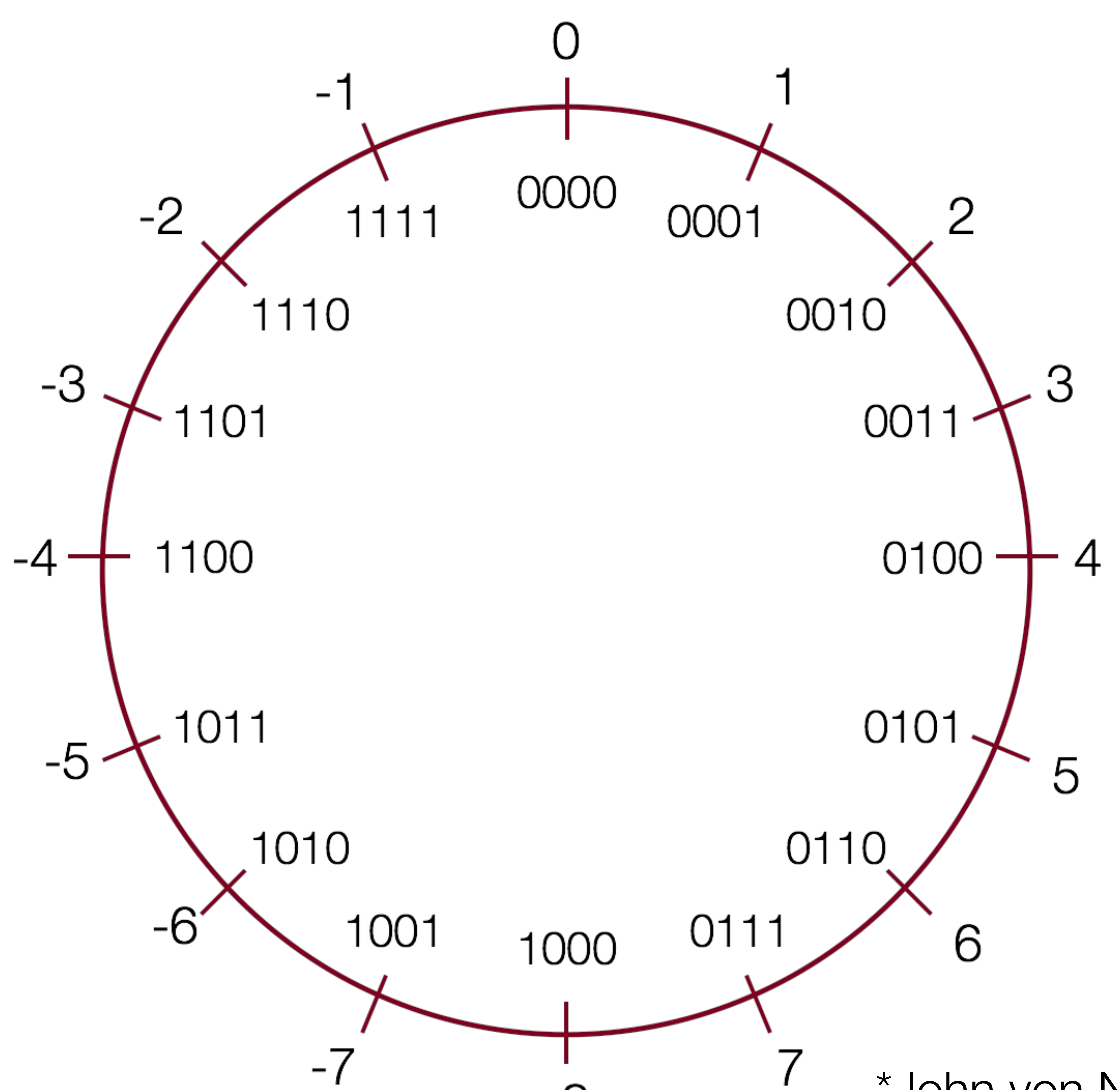
Fine. Let's just make 0 000 to be equal to decimal 0. How does arithmetic work? Well…to add two numbers, you need to know the sign, then you might have to subtract (borrow and carry, etc.), and the sign might change…this is going to get ugly!

Signed Integers: How do we represent them?



Signed Integers: How do we represent them?

Behold: the "two's complement" circle:



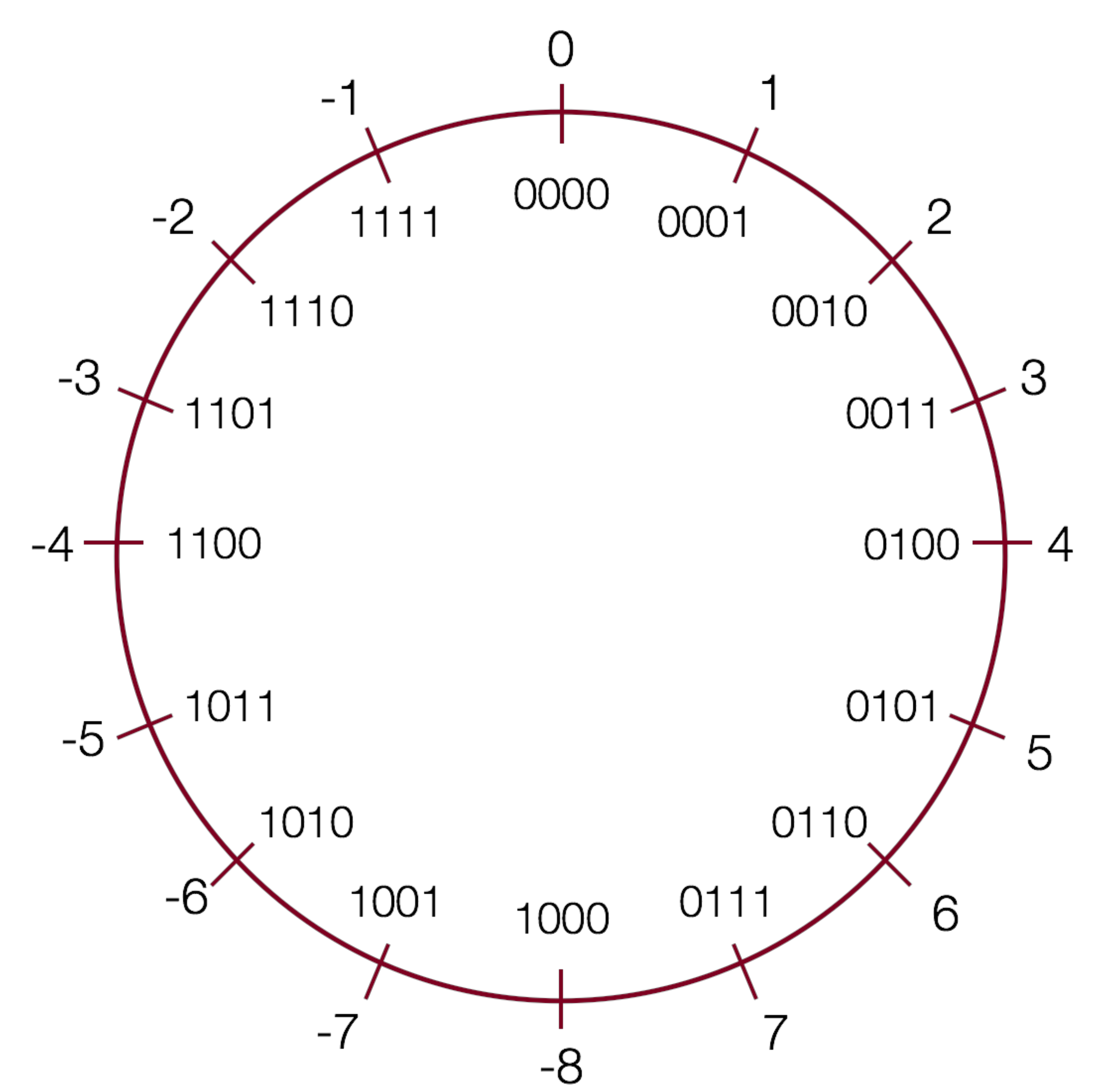
In the early days of computing*, two's complement was determined to be an excellent way to store binary numbers.

In two's complement notation, positive numbers are represented as themselves (phew), and negative numbers are represented as the two's complement of themselves (definition to follow).

This leads to some amazing arithmetic properties!

*John von Neumann suggested it in 1945, for the EDVAC computer.

Two's Complement



A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

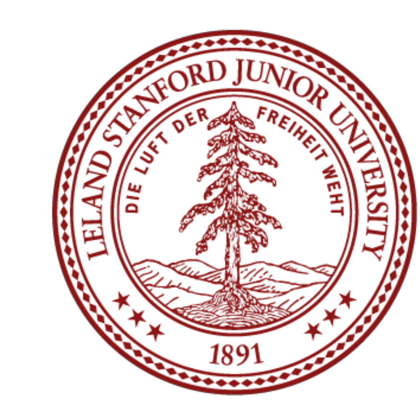
Definition: For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of an w-bit integer $x_{w-1}x_{w-2}\ldots x_0$ is given by the following formula:

$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$

$B2T_w$ means "Binary to Two's complement function"

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1.

*Inverting all the bits of a number is its "one's complement"



Two's Complement

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

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1, or: `x = ~x + 1`

Example: The number 2 is represented as normal in binary: 0010

-2 is represented by inverting the bits, and adding 1:

0010 ☞ 1101

```
  1101
+    1
  1110
```

*Inverting all the bits of a number is its "one's complement"

Two's Complement

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

Trick: to convert a positive number to its negative in two's complement, start from the right of the number, and write down all the digits until you get to a 1. Then invert the rest of the digits:

Example: The number 2 is represented as normal in binary: 0010

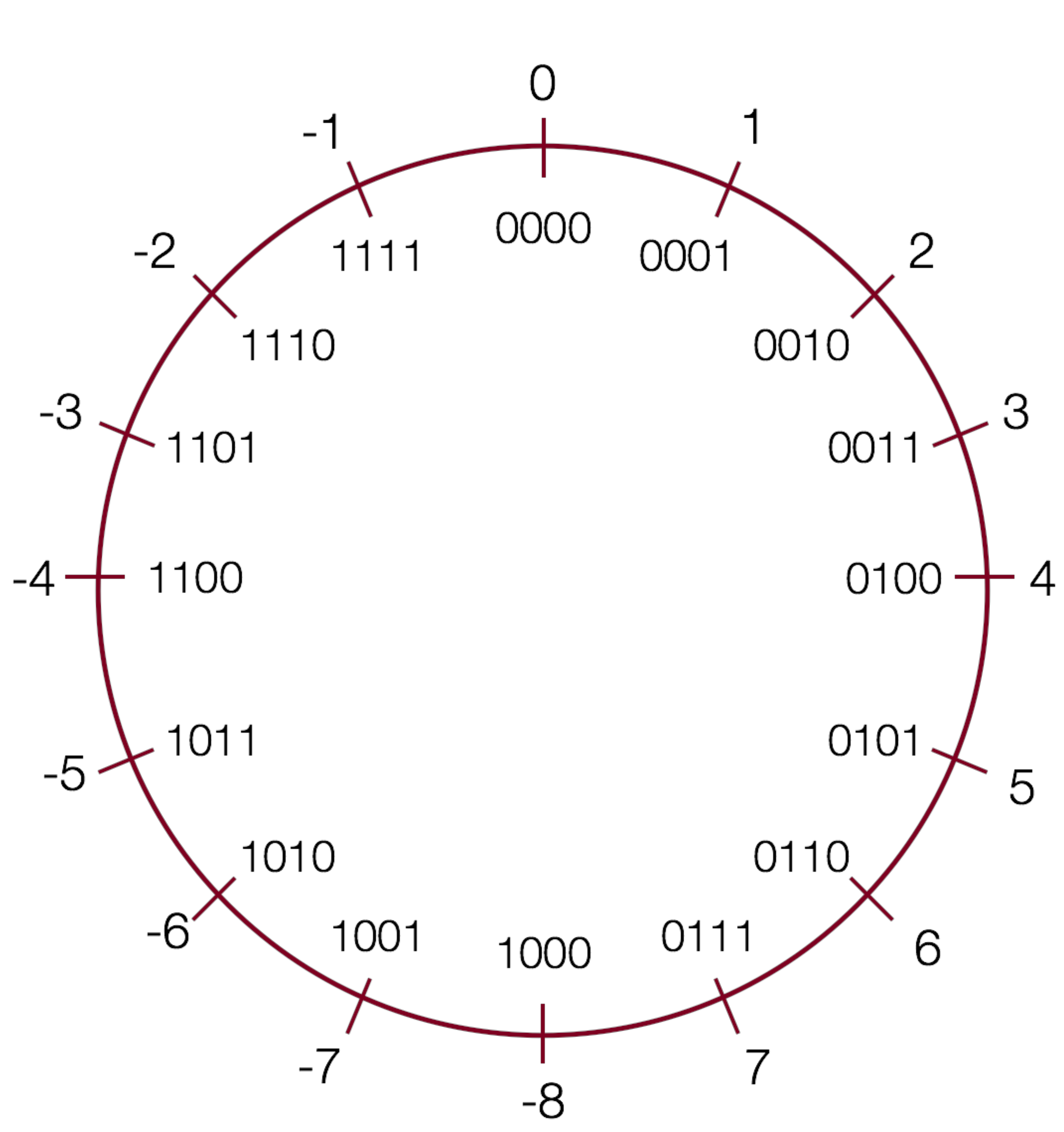
Going from the right, write down numbers until you get to a 1:

10

Then invert the rest of the digits:
1110

*Inverting all the bits of a number is its "one's complement"

Two's Complement



To convert a negative number to a positive number, perform the same steps!

Example: The number -5 is represented in two's complements as: 1011

5 is represented by inverting the bits, and adding 1:

1011 ☞ 0100

```
 0100
+   1
 0101
```

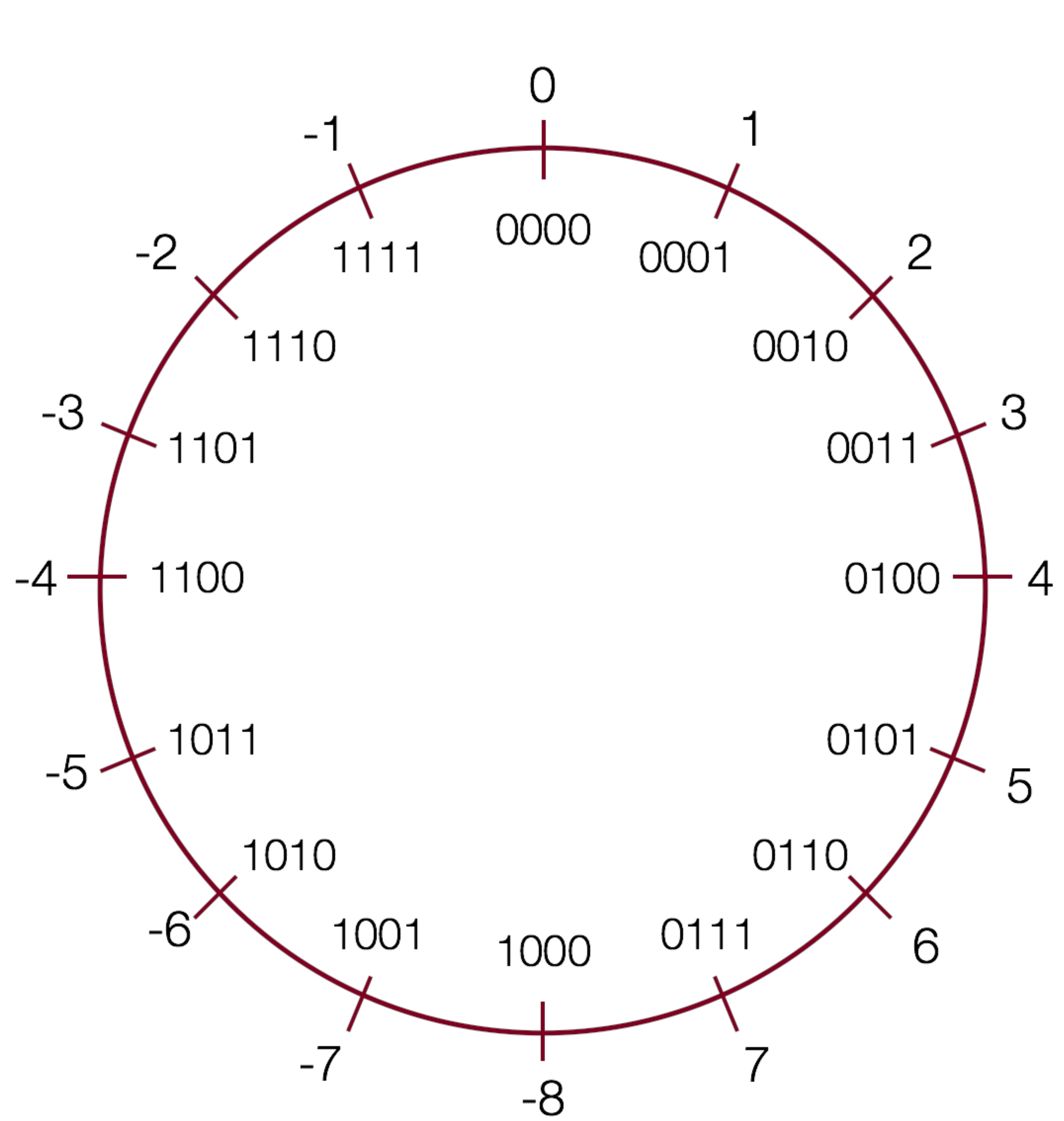
Shortcut: start from the right, and write down numbers until you get to a 1:

   1

Now invert all the rest of the digits:

0101

Two's Complement: Neat Properties

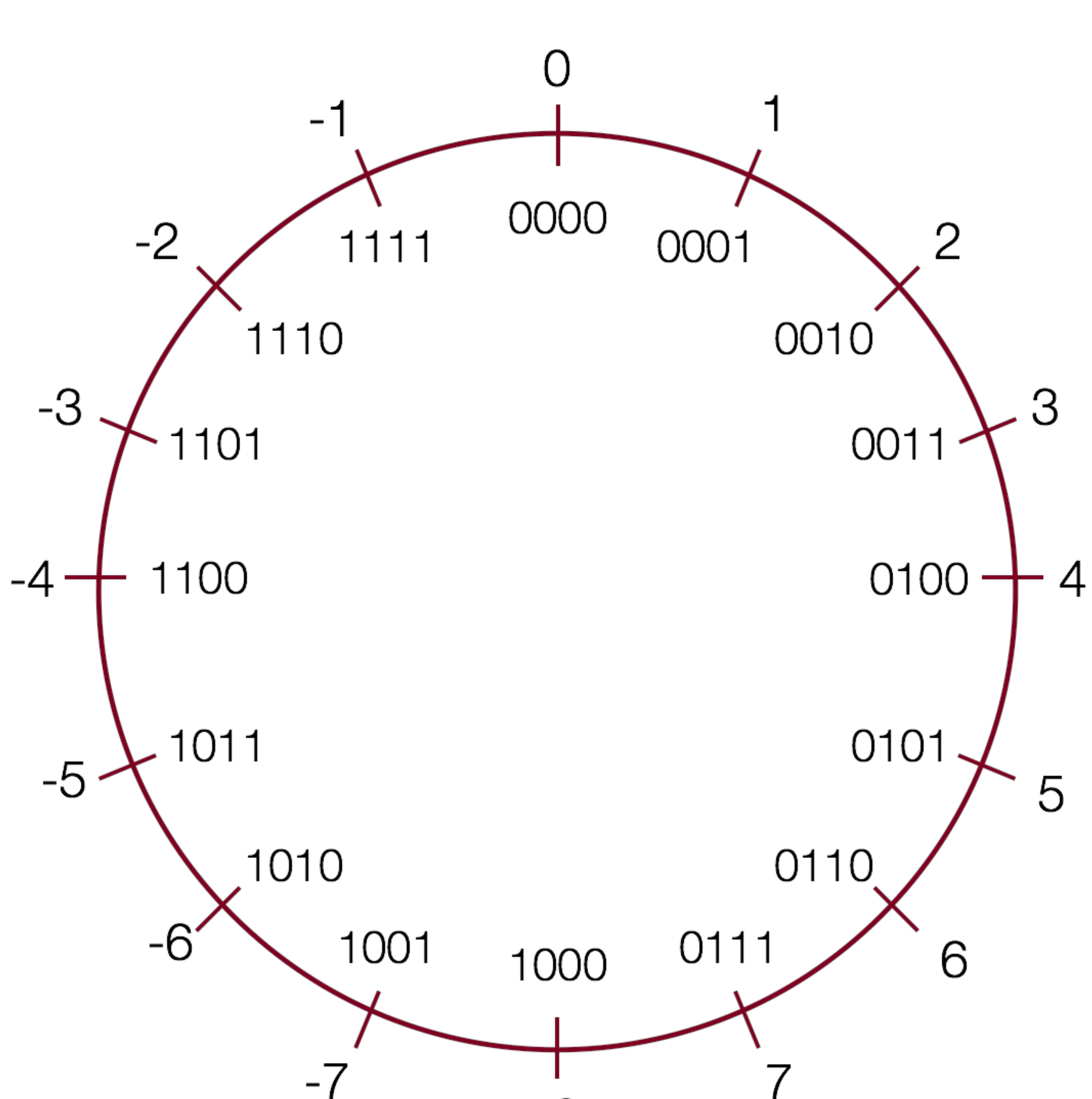


There are a number of useful properties associated with two's complement numbers:

1. There is only one zero (yay!)
2. The highest order bit (left-most) is 1 for negative, 0 for positive (so it is easy to tell if a number is negative)
3. Adding two numbers is just…adding! Example:
   2 + -5 = -3

 0010 ☞ 2
+1011 ☞ -5
 1101 ☞ -3 decimal (wow!)

Two's Complement: Neat Properties



More useful properties:

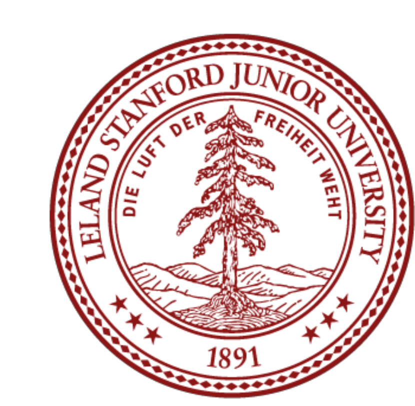
4. Subtracting two numbers is simply performing the two's complement on one of them and then adding. Example:
   4 - 5 = -1

0100 ☞ 4, 0101 ☞ 5

Find the two's complement of 5: 1011
add:

  0100 ☞ 4
+1011 ☞ -5

  1111 ☞ -1 decimal



Two's Complement: Neat Properties

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

More useful properties:

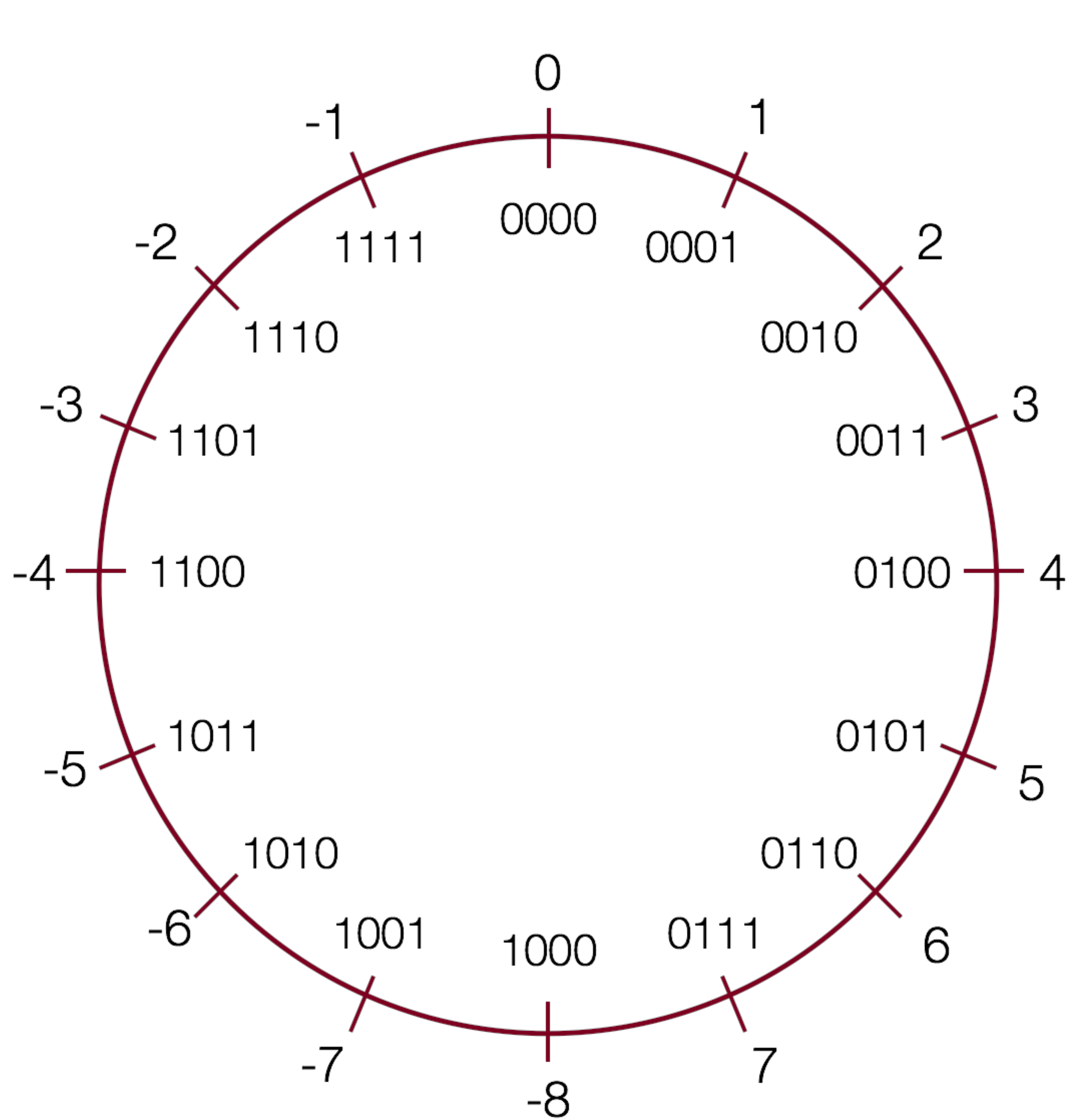
5. Multiplication of two's complement works just by multiplying (throw away overflow digits).

Example: -2 * -3 = 6

```
    1110 ☞  -2
   x1101 ☞  -3
    1110
   0000
  1110
+1110
10110110 ☞  6
```

(In the result, the leading "1011" overflow digits are struck through; **0110** is shown in bold.)

Two's Complement: Powers of two remain!



For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of an w-bit integer $x_{w-1}x_{w-2}\ldots x_0$ is given by the following formula:

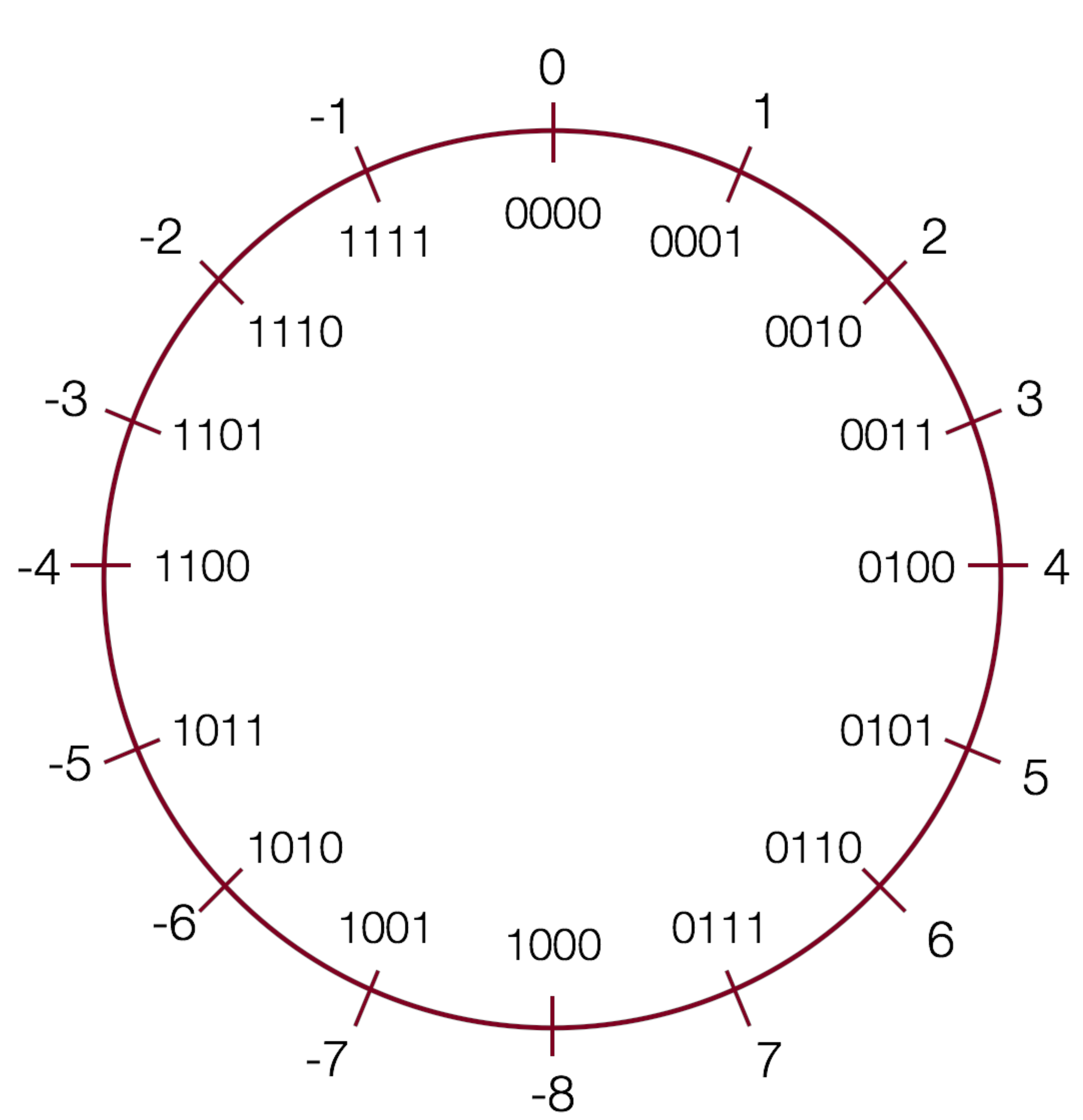
$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$

From the definition of a two's complement number, we can see that we are still dealing with bits being equal to their powers-of-two place: there isn't anything magical about the placement of the bits:

-5 = 1 0 1 1

(1 * -2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0)

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

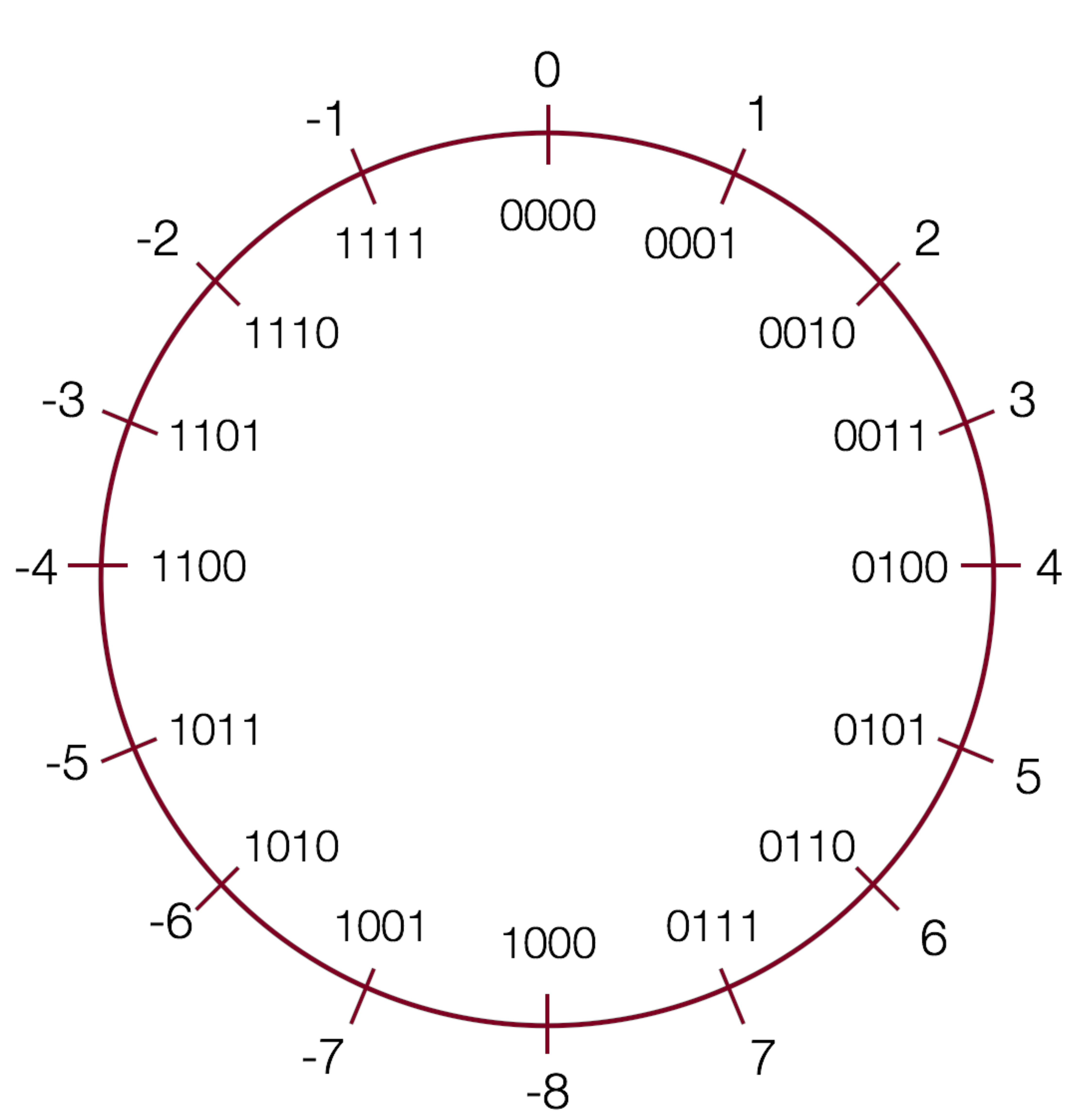
a. -4 (1100) ☞

b. 7 (0111) ☞

c. 3 (0011) ☞

d. -8 (1000) ☞

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

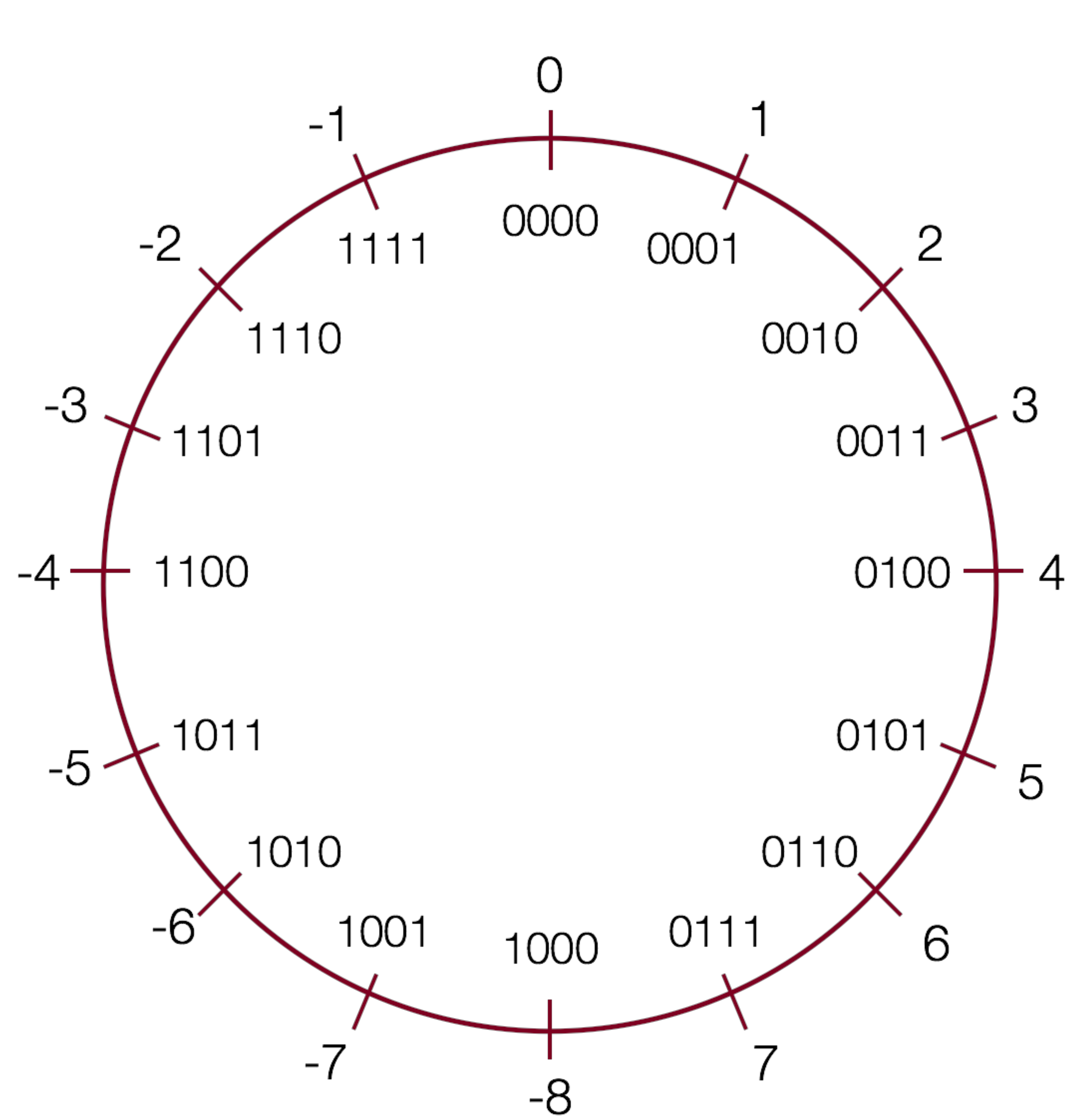
a. -4 (1100) ☞ 0100

b. 7 (0111) ☞ 1001

c. 3 (0011) ☞ 1101

d. -8 (1000) ☞ 1000 (?! If you look at the chart, +8 cannot be represented in two's complement with 4 bits!)

Practice



Convert the following 8-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. -4 (11111100) ☞ 00000100

b. 27 (00011011) ☞ 11100101

c. -127 (10000001) ☞ 01111111

d. 1 (00000001) ☞ 11111111

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. You cannot convert a signed number to its unsigned counterpart using a cast!

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
```

Casting Between Signed and Unsigned

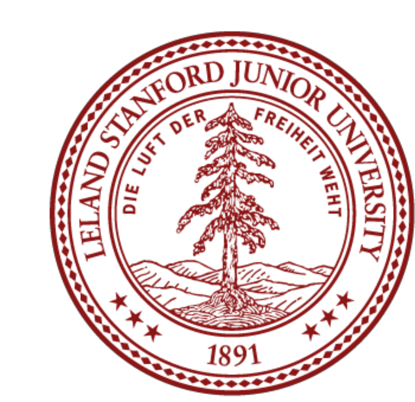
`printf` has three 32-bit integer representations:

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

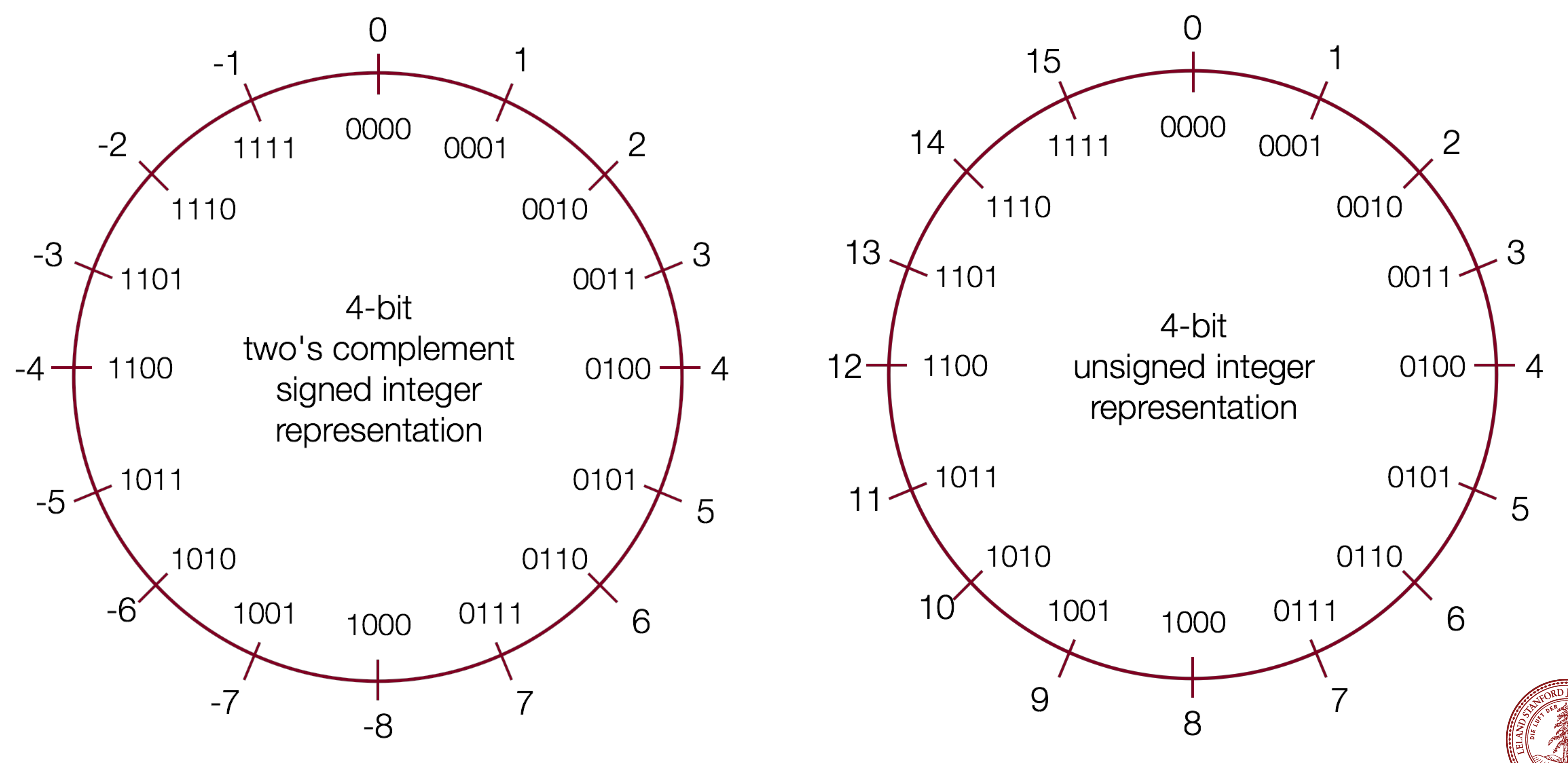
As long as the value is a 32-bit type, printf will treat it according to the formatter it is applying:

```
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);
```

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



Signed vs Unsigned Number Wheels

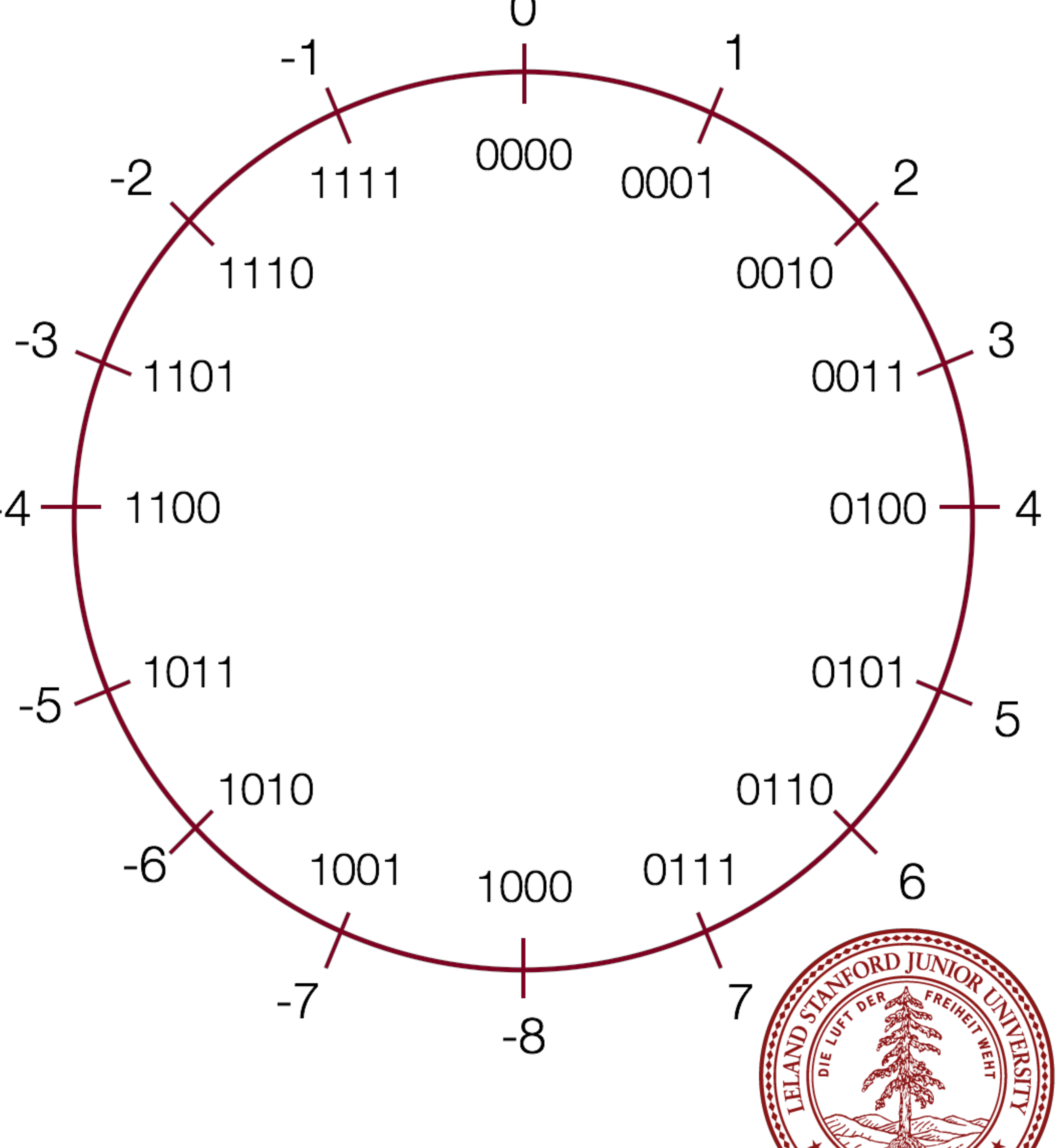


Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

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

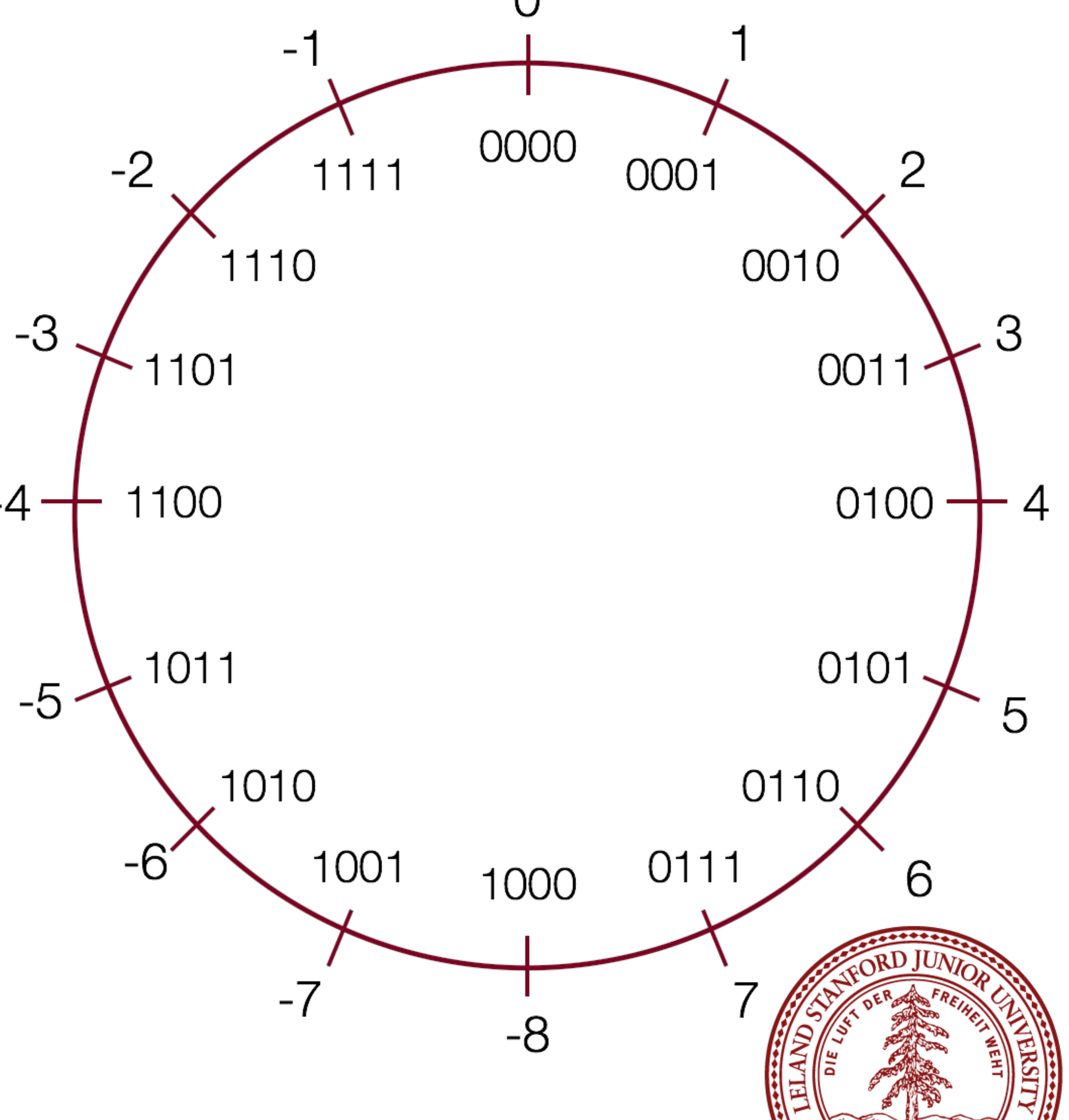


Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

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



Note: In C, 0 is false and everything else is true. When C produces a boolean value, it allways chooses 1 to represent true.

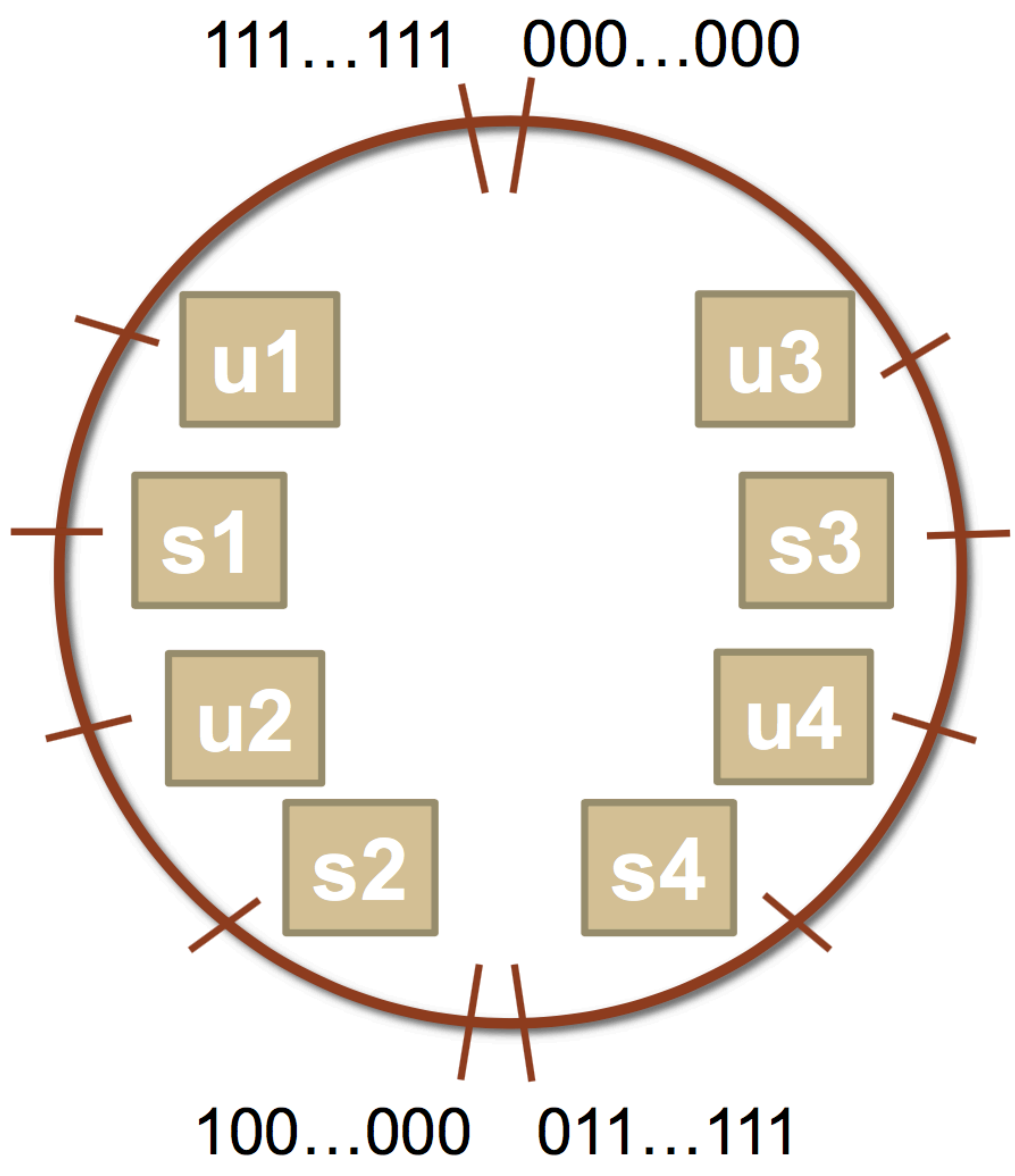
Comparison between signed and unsigned integers

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

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

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
```



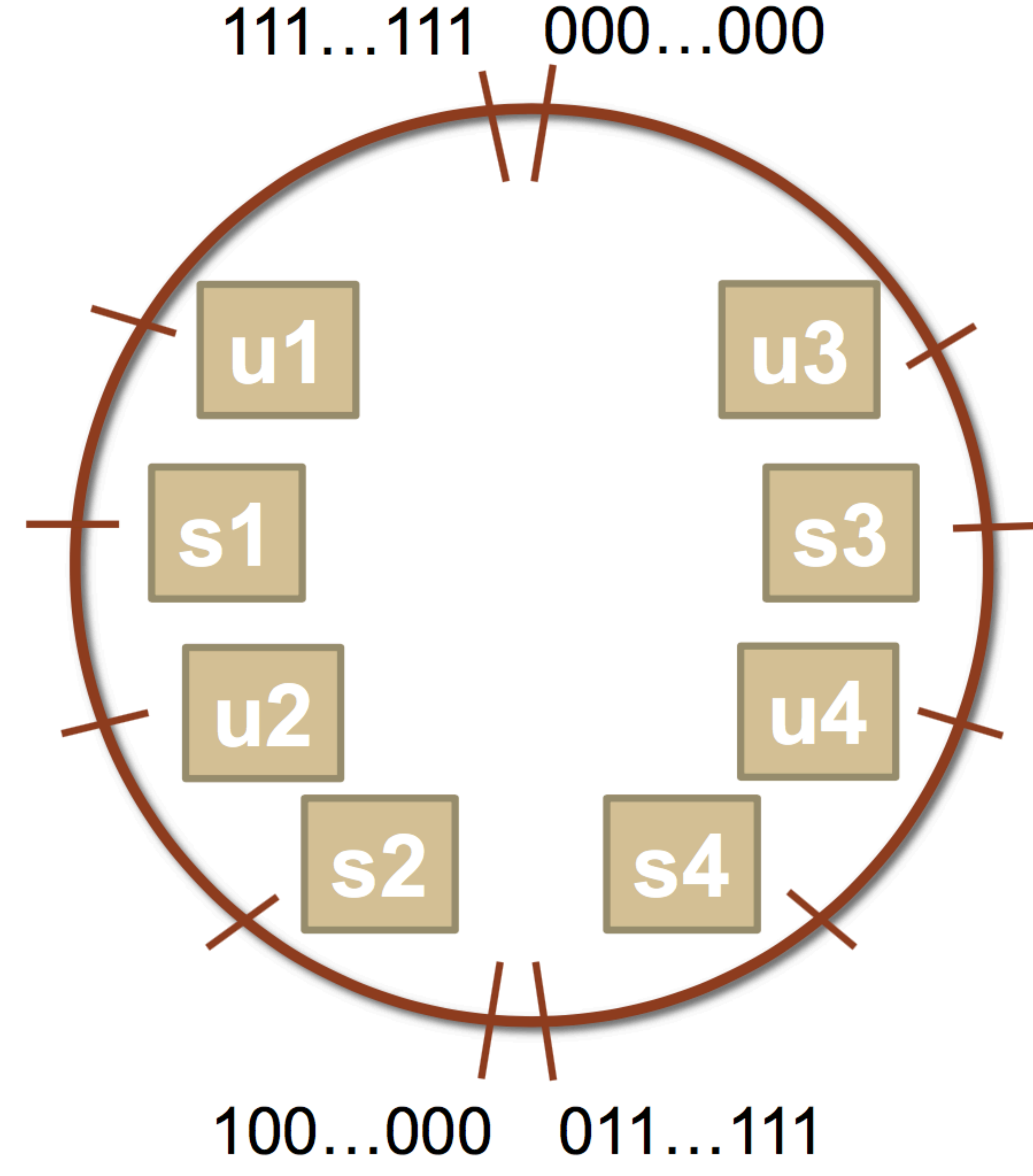
Comparison between signed and unsigned integers

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

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

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
```



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)

LELAND STANFORD JUNIOR UNIVERSITY · DIE LUFT DER FREIHEIT WEHT · 1891

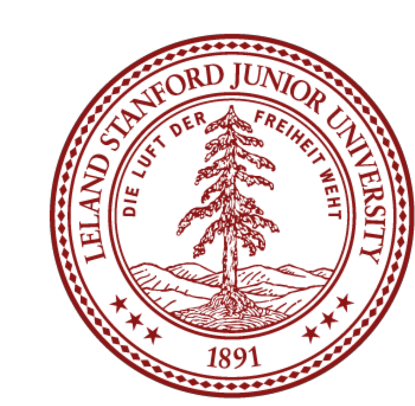
Expanding the bit representation of a number

Sometimes we want to convert between two integers having different sizes. E.g., a **`short`** to an **`int`**, or an **`int`** to a **`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.

This is easy for unsigned values: simply add leading zeros to the representation (called "zero extension").

```
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 the bit representation of a number

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a "sign extension" by repeating the sign of the value for the new digits. E.g.,

```
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 —

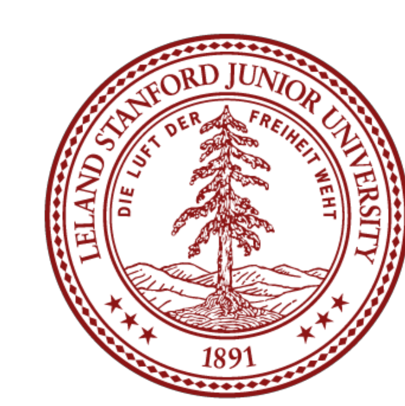
```
short s = -4;
// short is a 16-bit format, so                      s = 1111 1111 1111 1100b

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

Sign-extension Example

```
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    short sx = -12345;          // -12345
    unsigned short usx = sx; // 53191
    int x = sx;                 // -12345
    unsigned ux = usx;          // 53191

    printf("sx = %d:\t", sx);
    show_bytes((byte_pointer) &sx, sizeof(short));
    printf("usx = %u:\t", usx);
    show_bytes((byte_pointer) &usx, sizeof(unsigned short));
    printf("x  = %d:\t", x);
    show_bytes((byte_pointer) &x, sizeof(int));
    printf("ux = %u:\t", ux);
    show_bytes((byte_pointer) &ux, sizeof(unsigned));

    return 0;
}
```

```
$ ./sign_extension
sx = -12345:   c7 cf
usx = 53191:   c7 cf
x  = -12345:   c7 cf ff ff
ux = 53191:    c7 cf 00 00
```

(careful: this was printed on the little-endian myth machines!)



Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```
int x = 53191;
short sx = (short) 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`

What is this number in decimal? Well, it must be negative (b/c of the initial 1), and it is `-12345`.

Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```
int x = 53191;          // 53191
short sx = (short) x; // -12345
int y = sx;
```

This is a form of *overflow*! We have altered the value of the number. Be careful!

We don't have enough bits to store the int in the short for the value we have in the `int`, so the strange values occur.

What is y above? We are converting a short to an int, so we sign-extend, and we get -12345!

`1100 1111 1100 0111` becomes

`1111 1111 1111 1111 1100 1111 1100 0111`

Play around here: http://www.convertforfree.com/twos-complement-calculator/

Truncating Numbers: Signed

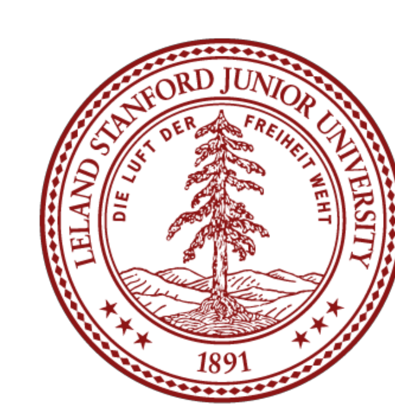
If the number does fit into the smaller representation in the current form, it will convert just fine.

```
int x = -3;          // -3
short sx = (short) -3; // -3
int y = sx; // -3
```

x: `1111 1111 1111 1111 1111 1111 1111 1101` becomes

sx: `1111 1111 1111 1101`

Play around here: http://www.convertforfree.com/twos-complement-calculator/



Truncating Numbers: Unsigned

We can also lose information with unsigned numbers:

```
unsigned int x = 128000;
unsigned short sx = (short) x;
unsigned int y = sx;
```

Bit representation for x = 128000 (32-bit unsigned int):

`0000 0000 0000 0001 1111 0100 0000 0000`

Truncated unsigned short sx:

`1111 0100 0000 0000`

which equals 62464 decimal.

Converting back to an unsigned int, y = 62464

Overflow in Unsigned Addition

When integer operations overflow in C, the runtime does not produce an error:

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for UINT_MAX

int main() {
    unsigned int a = UINT_MAX;
    unsigned int b = 1;
    unsigned int c = a + b;

    printf("a = %u\n",a);
    printf("b = %u\n",b);
    printf("a + b = %u\n",c);

    return 0;
}
```

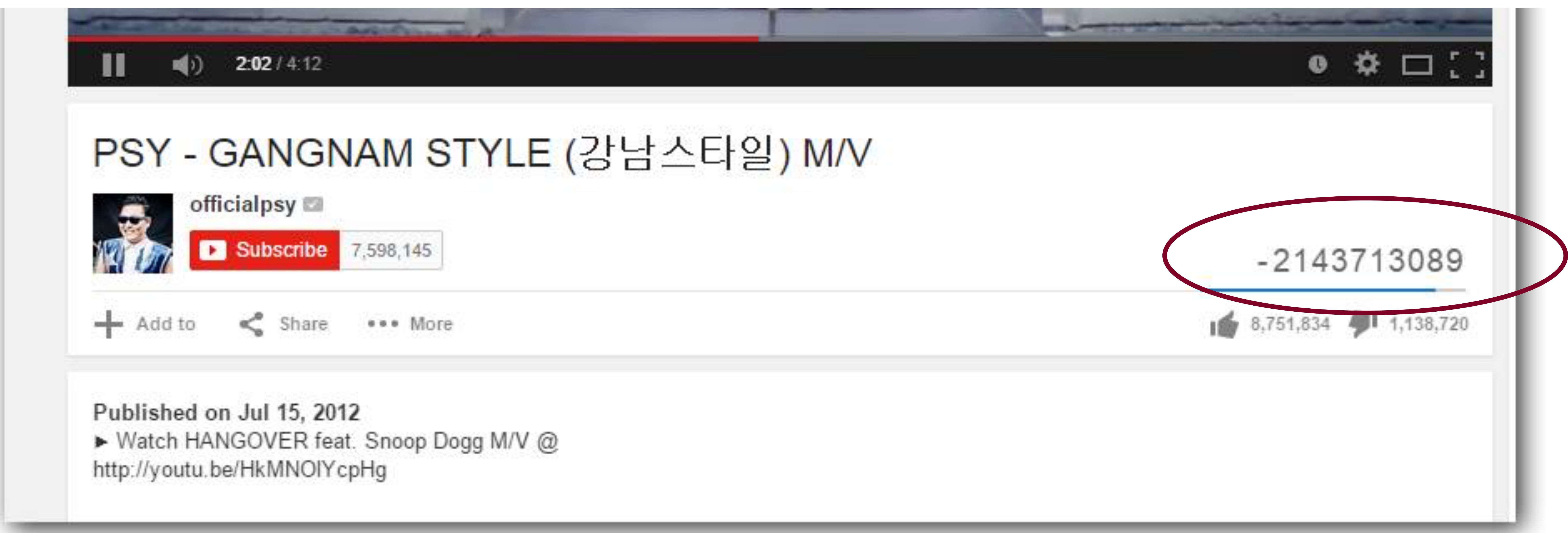
```
$ ./unsigned_overflow
a = 4294967295
b = 1
a + b = 0
```

Technically, unsigned integers in C don't overflow, they just wrap. You need to be aware of the size of your numbers. Here is one way to test if an addition will fail:

```
// for addition
#include <limits.h>
unsigned int a = <something>;
unsigned int x = <something>;
if (a > UINT_MAX - x) /* `a + x` would overflow */;
```

Overflow in Signed Addition

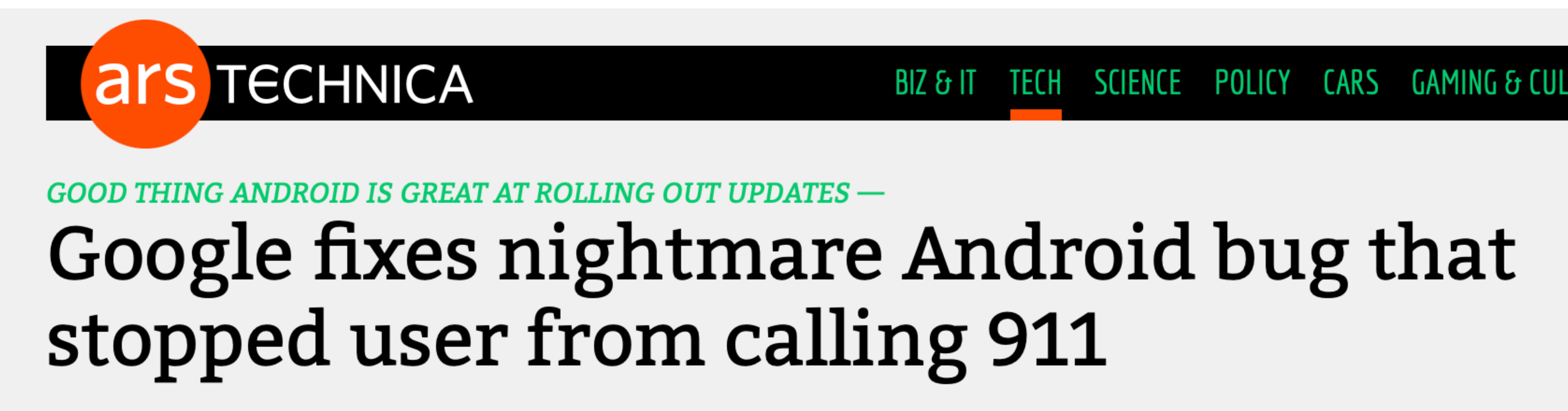
Signed overflow wraps around to the negative numbers:



YouTube fell into this trap — their view counter was a signed, 32-bit int. They fixed it after it was noticed, but for a while, the view count for Gangnam Style (the first video with over `INT_MAX` number of views) was negative.

Overflow in Signed Addition

In the news on January 5, 2022 (!):


ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CUL

GOOD THING ANDROID IS GREAT AT ROLLING OUT UPDATES —

Google fixes nightmare Android bug that stopped user from calling 911

An integer overflow/underflow crash lets misbehaving apps lock users out of 911.

RON AMADEO - 1/5/2022, 3:09 PM

https://arstechnica.com/gadgets/2022/01/google-fixes-nightmare-android-bug-that-stopped-user-from-calling-911/

Overflow in Signed Addition

Signed overflow wraps around to the negative numbers.

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for INT_MAX

int main() {
    int a = INT_MAX;
    int b = 1;
    int c = a + b;

    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("a + b = %d\n",c);

    return 0;
}
```

```
$ ./signed_overflow
a = 2147483647
b = 1
a + b = -2147483648
```

Technically, signed integers in C *produce undefined behavior* when they overflow. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:

```
// for addition
#include <limits.h>
int a = <something>;
int x = <something>;
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;
if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
```

3 Minute Break

WHEN CS107 GOES TOO FAST

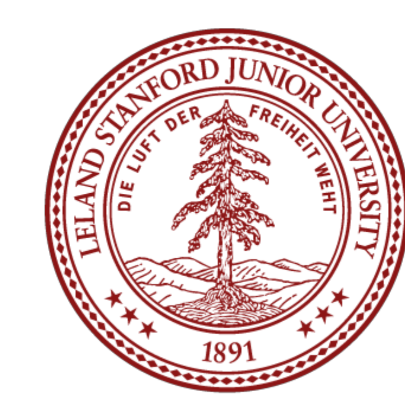
TAKE A BREAK

Data Sizes

Data Sizes

On the myth computers (and most 64-bit computers today), the `int` representation is comprised of 32-bits, or four 8-bit bytes. but the C language does not mandate this. To the right is Figure 2.3 from your textbook:

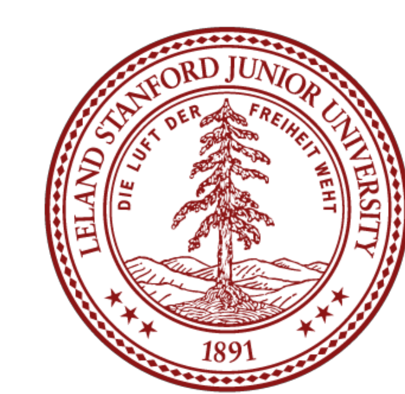
C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8



Data Sizes

There are guarantees on the lower-bounds for type sizes, but you should expect that the myth machines will have the numbers in the 64-bit column.

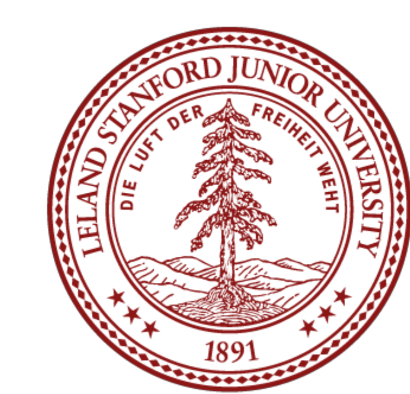
C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8



Data Sizes

You can be guaranteed the sizes for `int32_t` (4 bytes) and `int64_t` (8 bytes)

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8



Data Sizes

C allows a variety of ways to order keywords to define a type. The following all have the same meaning:

```
unsigned long
unsigned long int
long unsigned
long unsigned int
```

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Addressing and Byte Ordering

On the myth machines, pointers are 64-bits long, meaning that a program can "address" up to 2^{64} bytes of memory, because each byte is individually addressable.

This is a lot of memory! It is 16 *exa*bytes, or 1.84×10^{19} bytes. Older, 32-bit machines could only address 2^{32} bytes, or 4 Gigabytes.

64-bit machines can address 4 *billion* times more memory than 32-bit machines...

Machines will not need to address more than 2^{64} bytes of memory for a long, long time.

Addressing and Byte Ordering

We've already talked about the fact that a memory address (pointer) points to a particular byte. But, what if we want to store a data type that has more than one byte?

The **`int`** type on our machines is 4 bytes long. So, how is a byte stored in memory?

We have choices!

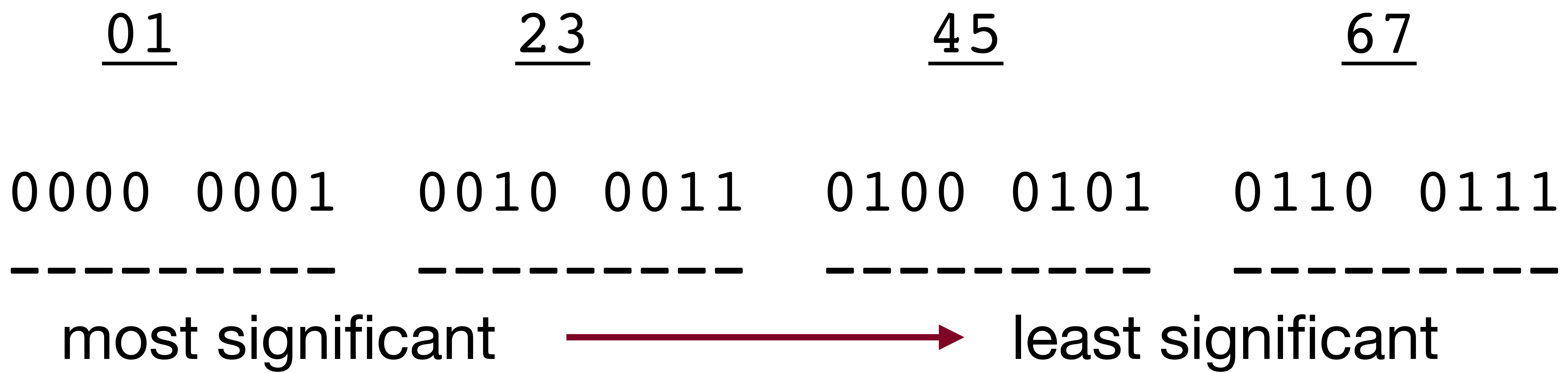
First, let's talk about the ordering of the bytes in a 4-byte hex number. We can represent an **`ints`** as 8-digit hex numbers:

`0x01234567`

We can separate out the bytes:

`0x  01 23 45 67`

Addressing and Byte Ordering



- Some machines choose to store the bytes ordered from least significant byte to most significant byte, called “little endian” (because the “little end” comes first).

- Other machines choose to store the bytes ordered from most significant byte to least significant byte, called “big endian” (because the “big end” comes first).

Addressing and Byte Ordering

- Our `0x01234567` number would look like this in memory for a little endian computer (which, by the way, is the way the myth computers store ints):

byte:	67	45	23	01
address:	0x100	0x101	0x102	0x103

- A big-endian representation would look like this:

byte:	01	23	45	67
address:	0x100	0x101	0x102	0x103

Many times we don't care how our integers are stored, but in cs107 we will! Let's look at a sample program and dig under the hood to see how little-endian works.

Addressing and Byte Ordering

- Our `0x01234567` number would look like this in memory for a little endian computer (which, by the way, is the way the myth computers store ints):

address:	`0x100`	`0x101`	`0x102`	`0x103`
value:	`67`	`45`	`23`	`01`

- A big-endian representation would look like this:

address:	`0x100`	`0x101`	`0x102`	`0x103`
value:	`01`	`23`	`45`	`67`

Many times we don't care how our integers are stored, but in cs107 we will! Let's look at a sample program and dig under the hood to see how little-endian works.

Addressing and Byte Ordering

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3
 4 int main() {
 5     // a variable
 6     int a = 0x01234567;
 7
 8     // print the variable in big endian format
 9     printf("a's value: 0x%.8x\n",a);
10     return 0;
11 }
```

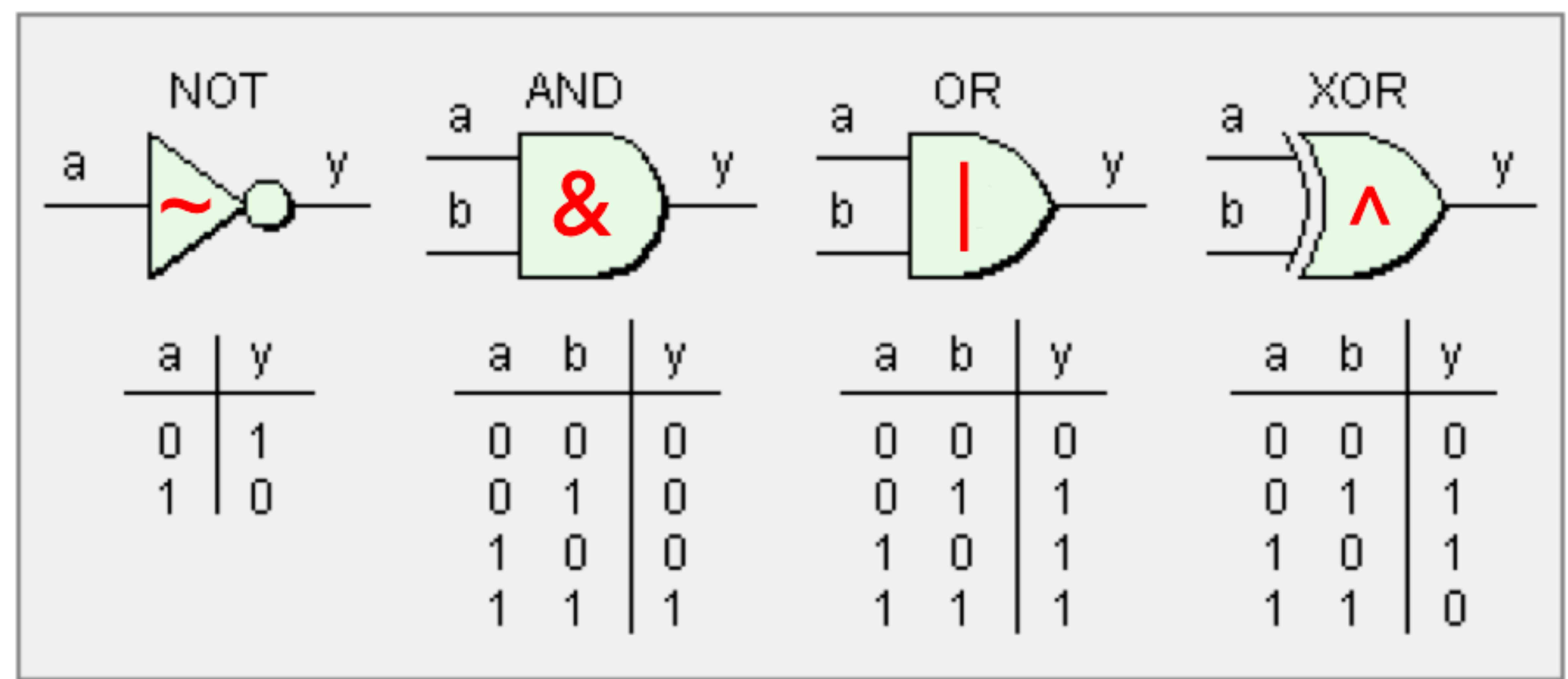
Addressing and Byte Ordering

```
$ gcc -g -O0 -std=gnu99 big_endian.c -o big_endian
$ ./big_endian
a's value: 0x01234567

$ gdb big_endian
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
...
(gdb) break main
Breakpoint 1 at 0x400535: file big_endian.c, line 6.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/107/lectures/lecture2_bits_bytes_continued/big_endian

Breakpoint 1, main () at big_endian.c:6
6        int a = 0x01234567;
(gdb) n
9        printf("a's value: 0x%08x\n",a);
(gdb) p/x a
$1 = 0x1234567
(gdb) p &a
$2 = (int *) 0x7fffffffe98c
(gdb) x/16bx &a
0x7fffffffe98c: 0x67    0x45    0x23    0x01    0x00    0x00    0x00    0x00
0x7fffffffe994: 0x00    0x00    0x00    0x00    0x45    0x2f    0xa3    0xf7
(gdb)
```

Note the ordering: `0x01234567` is stored as Little Endian!

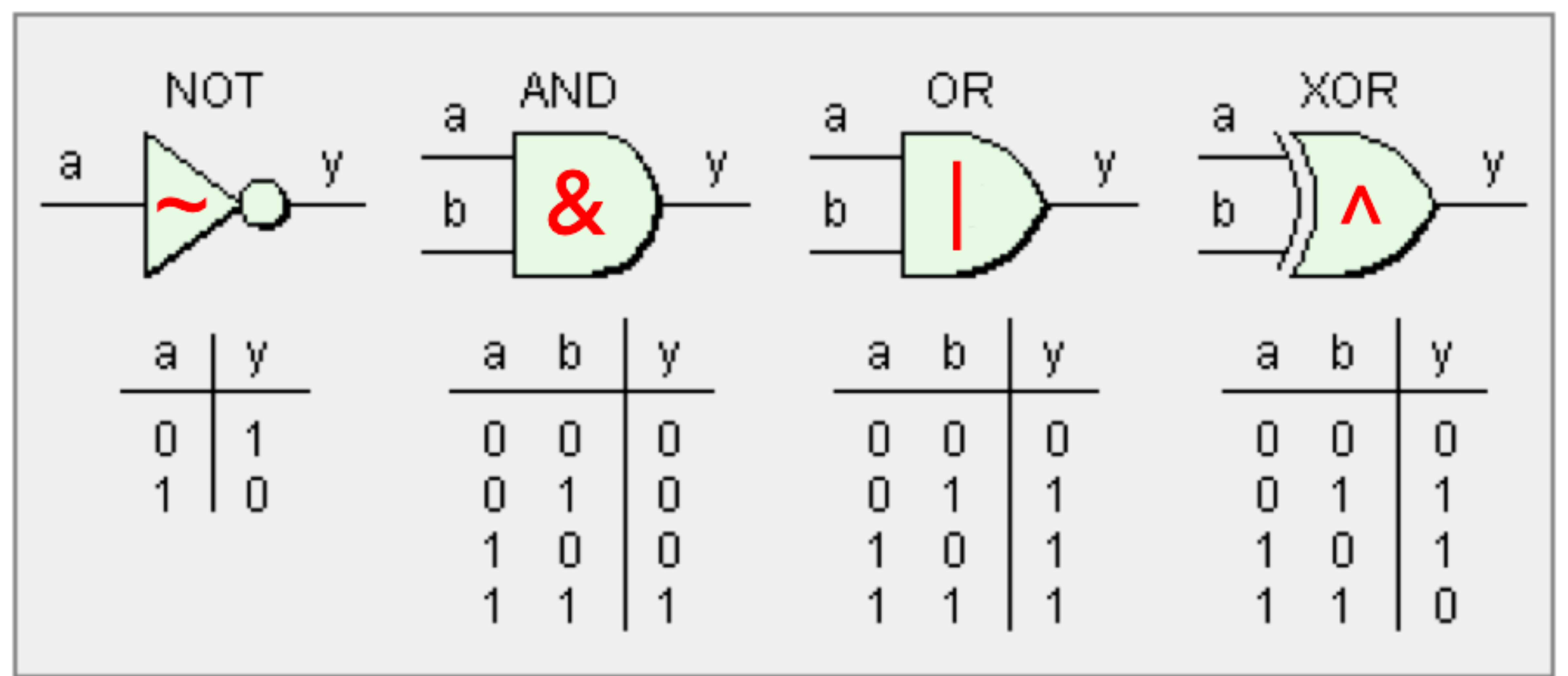
Boolean Algebra

Boolean Algebra



- Because computers store values in binary, we need to learn about boolean algebra. Most of you have already studied this in some form in math classes before, but we are going to quantify it and discuss it in the context of computing and programming.
- We can define Boolean algebra over a 2-element set, 0 and 1, where 0 represents `false` and 1 represents `true`.
- The symbols are: ~ for NOT, `&` for AND, | for OR, and ^ for "exclusive or," which means that if one and only one of the values is true, the expression is true.

Boolean Algebra



a	y
0	1
1	0

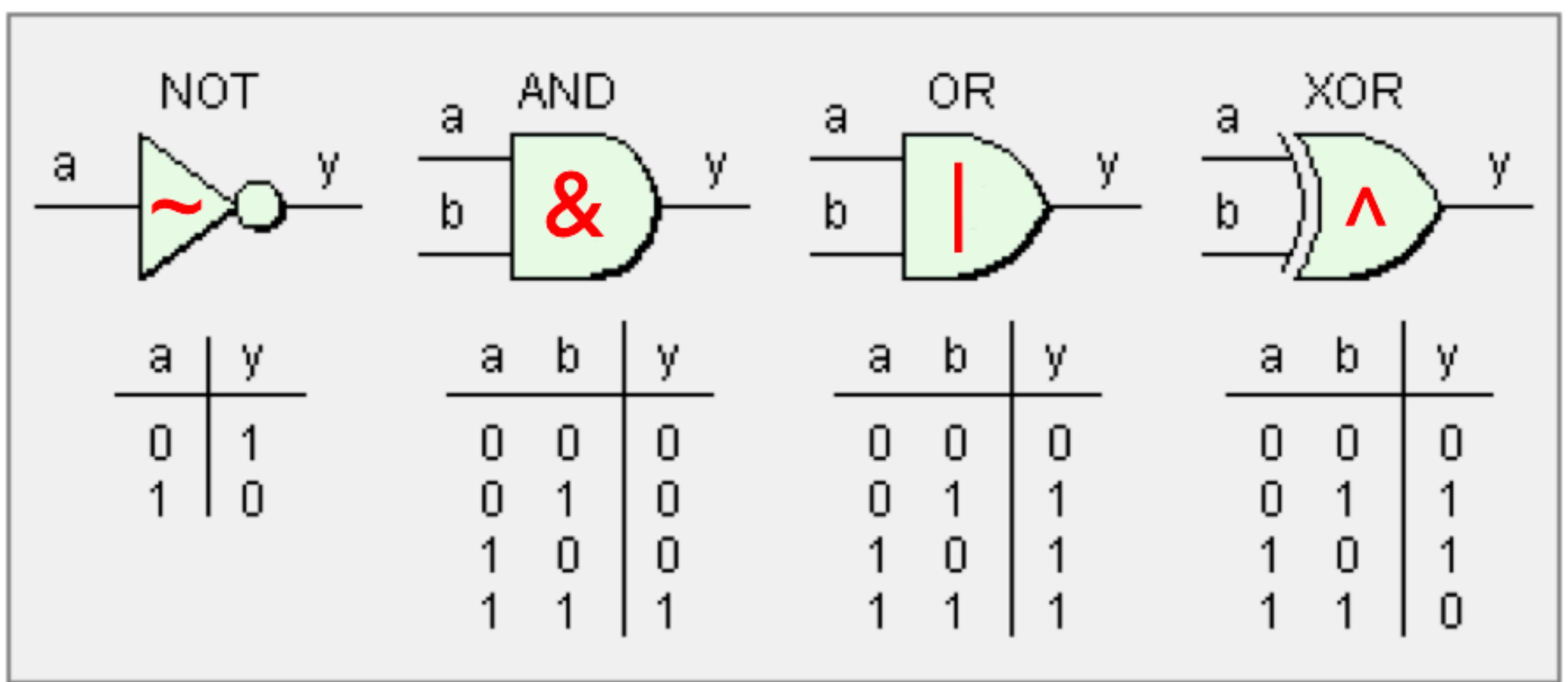
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

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

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

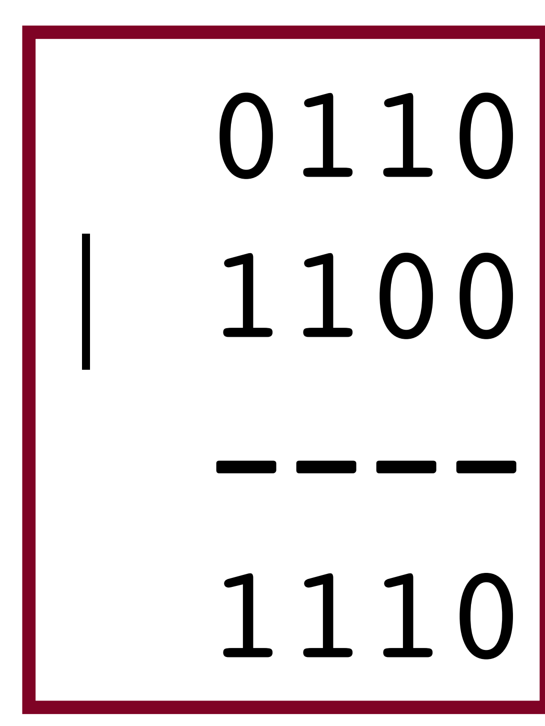
- Be careful! There are *logical* analogs to some of these that you have used in C++ and other programming languages: `!` (logical NOT), `&&` (logical AND), and `||` (logical OR), but we are now talking about *bit* operations that result in 0 or 1 for each bit in a number.
- The bitwise operators use single character representations for AND and OR, not double-characters.

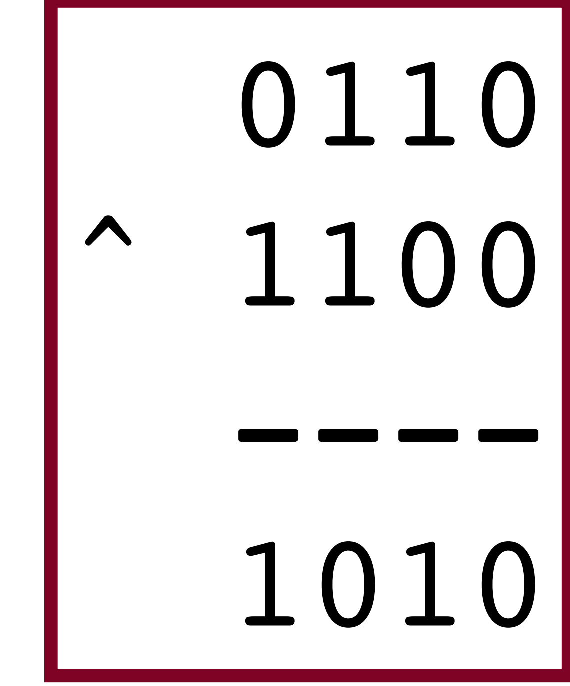
Boolean Algebra

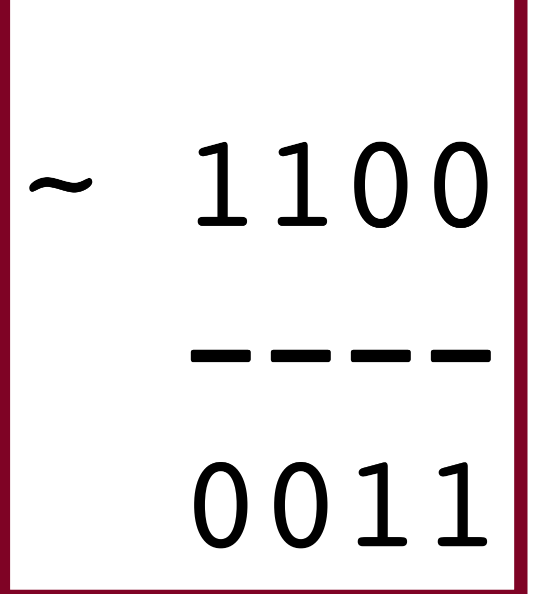


- When a boolean operator is applied to two numbers (or, in the case of ~, a single number), the operator is applied to the corresponding bits in each number. For example:

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







Boolean Algebra: Mystery Function

- Let's look at a mystery function!

```
 1 // mystery1.c
 2 #include<stdlib.h>
 3 #include<stdio.h>
 4 
 5 void mystery(int *x, int *y) {
 6     if (x != y) {
 7         *y = *x ^ *y;
 8         *x = *x ^ *y;
 9         *y = *x ^ *y;
10     }
11 }
12 
13 int main(int argc, char *argv[]) {
14     int x = atoi(argv[1]);
15     int y = atoi(argv[2]);
16 
17     printf("x:%d, y:%d\n",x,y);
18 
19     mystery(&x,&y);
20 
21     printf("x:%d, y:%d\n",x,y);
22     return 0;
23 }
```

```
$ ./mystery 4 5
```

Boolean Algebra: Mystery Function

- Let's look at a mystery function!

```
1  // mystery1.c
2  #include<stdlib.h>
3  #include<stdio.h>
4
5  void mystery(int *x, int *y) {
6      if (x != y) {
7          *y = *x ^ *y;
8          *x = *x ^ *y;
9          *y = *x ^ *y;
10     }
11 }
12
13 int main(int argc, char *argv[]) {
14     int x = atoi(argv[1]);
15     int y = atoi(argv[2]);
16
17     printf("x:%d, y:%d\n",x,y);
18
19     mystery(&x,&y);
20
21     printf("x:%d, y:%d\n",x,y);
22     return 0;
23 }
```

```
$ ./mystery 4 5
x:4, y:5
x:5, y:4
```

https://en.wikipedia.org/wiki/XOR_swap_algorithm

This relies on the fact that `x^x == 0`, and the associativity and commutativity of the exclusive or function.

Incidentally, if you XOR a number with all 1s, you get the complement!

Boolean Algebra: Operations on bit flags

We can represent finite sets with bit vectors, where we can perform set functions such as union, intersection, and complement. For example:

bit vector a = [01101001] encodes the set A = {0,3,5,6} (reading the 1 positions from *right to left*, with #0 being the right-most, #7 being the left-most)

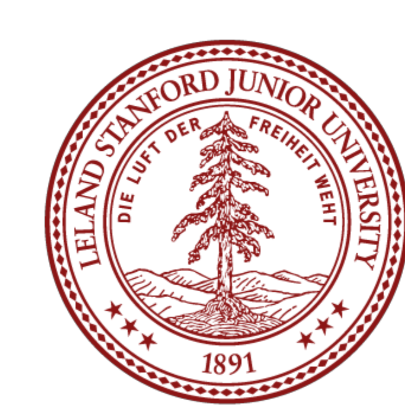
bit vector b = [01010101] encodes the set B = {0,2,4,6}

The `|` operator produces a set union:

`a | b` → [01111101], or $A \cup B = \{0,2,3,4,5,6\}$

The `&` operator produces a set intersection:

`a & b` → [01000001], or $A \cap B = \{0,6\}$

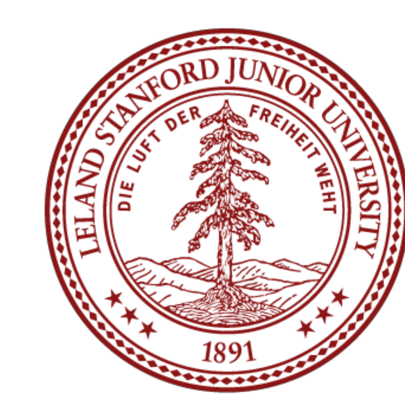


Boolean Algebra: Bit Masking

A common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that will be used to choose a selected set of bits in a word. For example, the mask of **`0xFF`** means the lowest byte in an integer. To get the low-order byte out of an integer, we simply use the bitwise AND operator with the mask:

```
int j = 0x89ABCDEF;
int k = j & 0xFF; // k now holds the value 0xEF,
                  // which is the low-order byte of j
```

A useful expression is `~0`, which makes an integer with all 1s, regardless of the size of the integer.



Boolean Algebra: Bit Masking

Challenge 1: write an expression that sets the least significant byte to all ones, and all other bytes of the number (assume it is the variable `j`) left unchanged E.g.

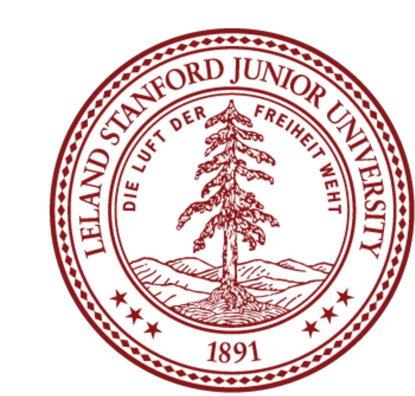
`0x87654321` → `0x876543FF`

Possible answer: `j | 0xFF`

Challenge 2: write an expression that complements all but the least significant byte of `j`, with the least significant byte unchanged. E.g.

`0x87654321` → `0x789ABC21`

Possible answer: `j ^ ~0xFF`



Boolean Algebra: Shift Operations

C provides operations to shift bit patterns to the left and to the right.

The `<<` operator moves the bits to the left, replacing the lower order bits with zeros and dropping any values that would be bigger than the type can hold:

`x << k` will shift **`x`** to the left by **`k`** number of bits.

Examples for an 8-bit binary number:

```
00110111 << 2 returns 11011100
01100011 << 4 returns 00110000
10010101 << 4 returns 01010000
```

Boolean Algebra: Shift Operations

There are actually two flavors of *right* shift, which work differently depending on the value and type of the number you are shifting.

A *logical* right shift moves the values to the right, replacing the upper bits with 0s.

An *arithmetic* right shift moves the values to the right, replacing the upper bits with a copy of the most significant bit. This may seem weird! But, we will see why this is useful soon!

Examples for an 8-bit binary number:
Logical right shift:

00110111 >> 2 returns *00001101*
10110111 >> 2 returns *00101101*
01100011 >> 4 returns *00000110*
10010101 >> 4 returns *00001001*

Examples for an 8-bit binary number:
Artithmetic right shift:

00110111 >> 2 returns *00001101*
10110111 >> 2 returns *11101101*
01100011 >> 4 returns *00000110*
10010101 >> 4 returns *11111001*

Right shift: arithmetic -vs- logical

The right-shift (>>) operator behaves differently for unsigned and signed numbers:

- **Unsigned** numbers are **logically**-right shifted (by shifting in 0s, always)
- **Signed** numbers are **arithmetically**-right shifted (by shifting in the sign bit)

```
$ ./right_shift
a = 1048576:       00 00 10 00
a >> 8 = 4096:     00 10 00 00
b = -1048576:      00 00 f0 ff
b >> 8 = -4096:    00 f0 ff ff
```

(run on a little-endian machine)

```
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    int a = 1048576;
    int a_rs8 = a >> 8;

    int b = -1048576;
    int b_rs8 = b >> 8;

    printf("a = %d:\t", a);
    show_bytes((byte_pointer) &a, sizeof(int));

    printf("a >> 8 = %d:\t", a_rs8);
    show_bytes((byte_pointer) &a_rs8, sizeof(int));

    printf("b = %d:\t", b);
    show_bytes((byte_pointer) &b, sizeof(int));

    printf("b >> 8 = %d:\t", b_rs8);
    show_bytes((byte_pointer) &b_rs8, sizeof(int));
    return 0;
}
```

Shift Operation Pitfalls

There are two important things you need to consider when using the shift operators:

1. The C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. *Almost all* compilers / machines use arithmetic shifts for signed integers, and you can most likely assume this. Don't be surprised if some Internet pedant yells at you about it some day. :) All *unsigned* integers will always use a logical right shift (more on this later!)

2. Operator precedence can be tricky! Example:

   `1<<2 + 3<<4` means this: `1 << (2 + 3) << 4`, because *addition and subtraction have a higher precedence than shifts!*

   Always parenthesize to be sure:
   `(1<<2) + (3<<4)`

Practice!

Let's take a look at lots of examples:

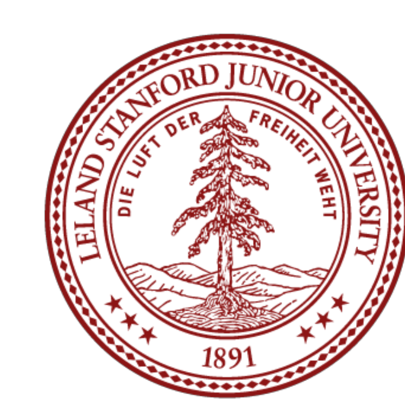
If you want to try the examples out yourself. On myth:

```
$ cd CS107
$ cp -r /afs/ir/class/cs107/lecture-code/lect2 .
cd lect2
make
ls # to see the files
```

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/

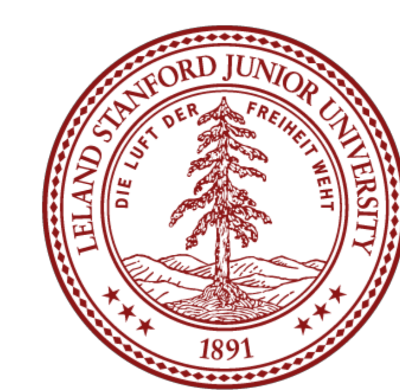
- **Advanced Reading:**
  - Signed overflow: https://stackoverflow.com/questions/16056758/c-c-unsigned-integer-overflow
  - Integer overflow in C: https://www.gnu.org/software/autoconf/manual/autoconf-2.62/html_node/Integer-Overflow.html
  - https://stackoverflow.com/questions/34885966/when-an-int-is-cast-to-a-short-and-truncated-how-is-the-new-value-determined

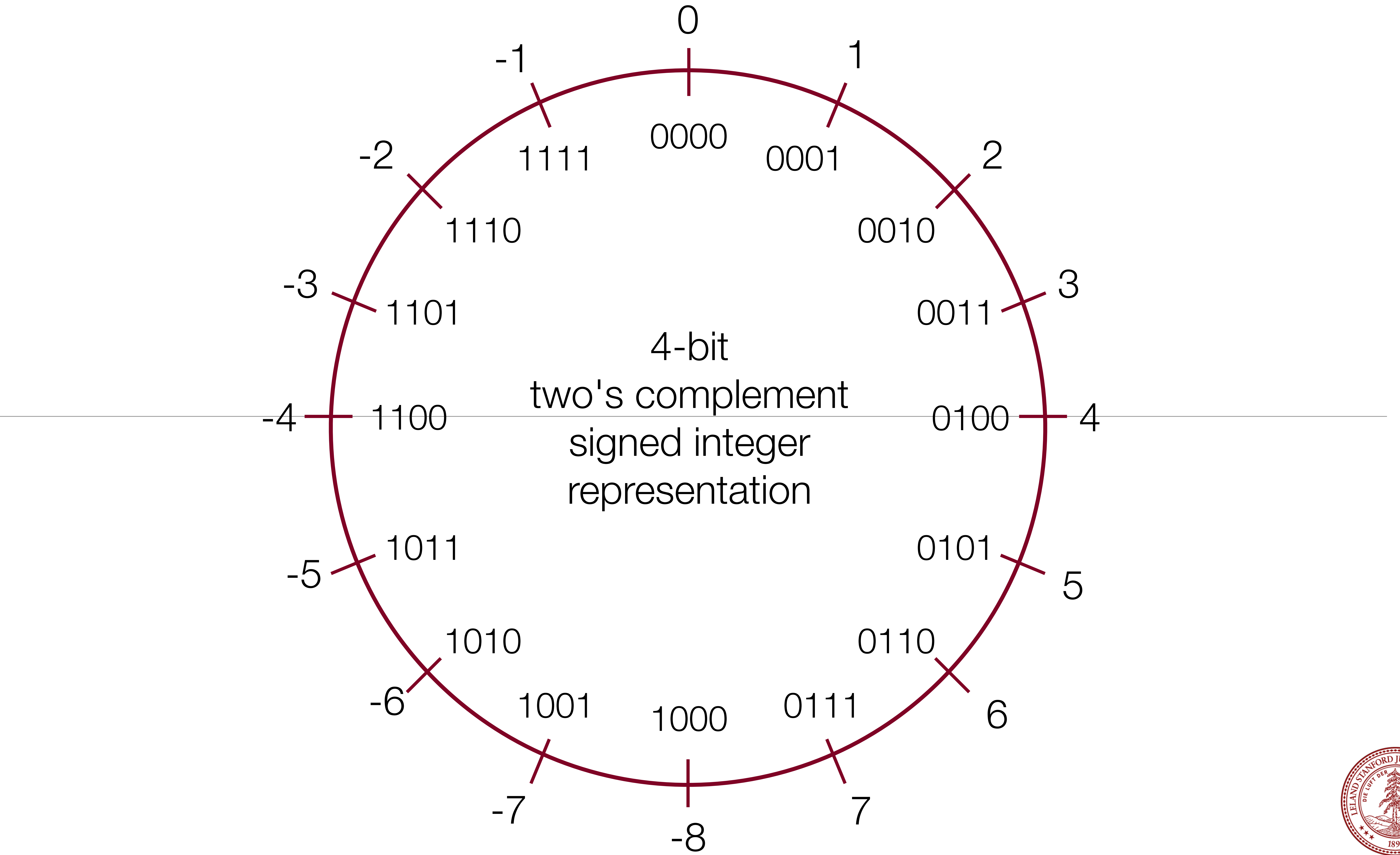


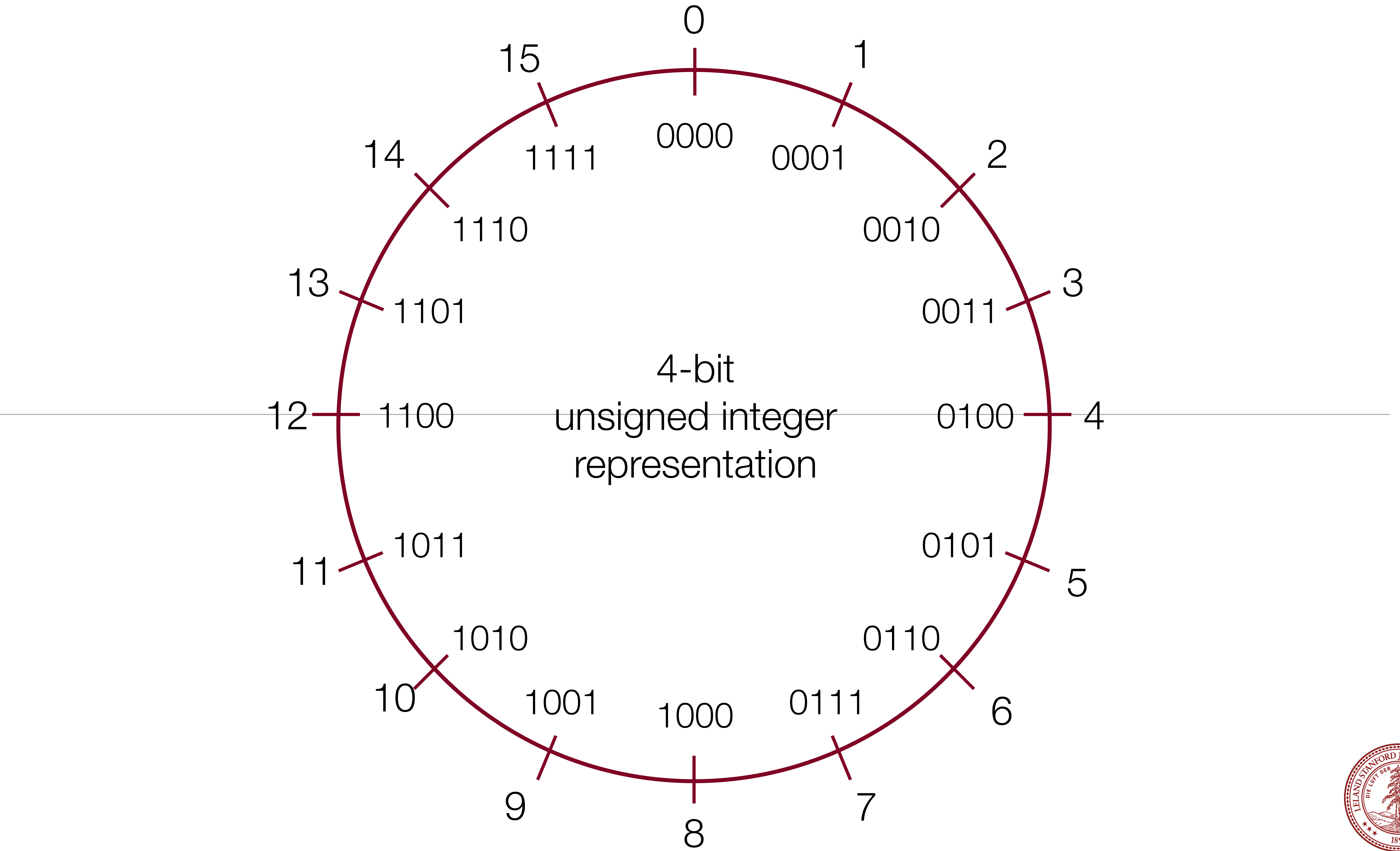
References and Advanced Reading

- **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




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
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