One Bit At A Time

- We can combine bits, like with base-10 numbers, to represent more data. **8 bits = 1 byte**.
- Computer memory is just a large array of bytes! It is *byte-addressable*; you can't address (store location of) a bit; only a byte.
- Computers still fundamentally operate on bits; we have just gotten more creative about how to represent different data as bits!
  - Images
  - Audio
  - Video
  - Text
  - And more...

Base 2

Most significant bit (MSB)

Least significant bit (LSB)

1 0 1 1

eights fours twos ones

$$= \mathbf{1}*8 + \mathbf{0}*4 + \mathbf{1}*2 + \mathbf{1}*1 = 11_{10}$$

Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store? **minimum = 0** **maximum = 255**

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$
- **Strategy 2:** $2^8 - 1 = 255$

Multiplying by Base

$$1450 \times 10 = 1450\underline{0}$$

$$1100_2 \times 2 = 1100\underline{0}$$

Key Idea: inserting 0 at the end multiplies by the base!

Dividing by Base

$$1450 \;/\; 10 = 145$$

$$1100_2 \;/\; 2 = 110$$

Key Idea: removing 0 at the end divides by the base!

Hexadecimal

- Hexadecimal is *base-16*, so we need digits for 1-15. How do we do this?

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
										10	11	12	13	14	15

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

Hexadecimal

- We distinguish hexadecimal numbers by prefixing them with **`0x`**, and binary numbers with **`0b`**.
- E.g. **`0xf5`** is **`0b11110101`**

$$0x\,\underbrace{f}_{1111}\;\underbrace{5}_{0101}$$

Number Representations

C Declaration	Size (Bytes)
`int`	**4**
`double`	**8**
`float`	**4**
`char`	**1**
`char *`	**8**
`short`	**2**
`long`	**8**

Transitioning To Larger Datatypes

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32}-1, equaling **2^{32} bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024 GB** of memory (RAM)!

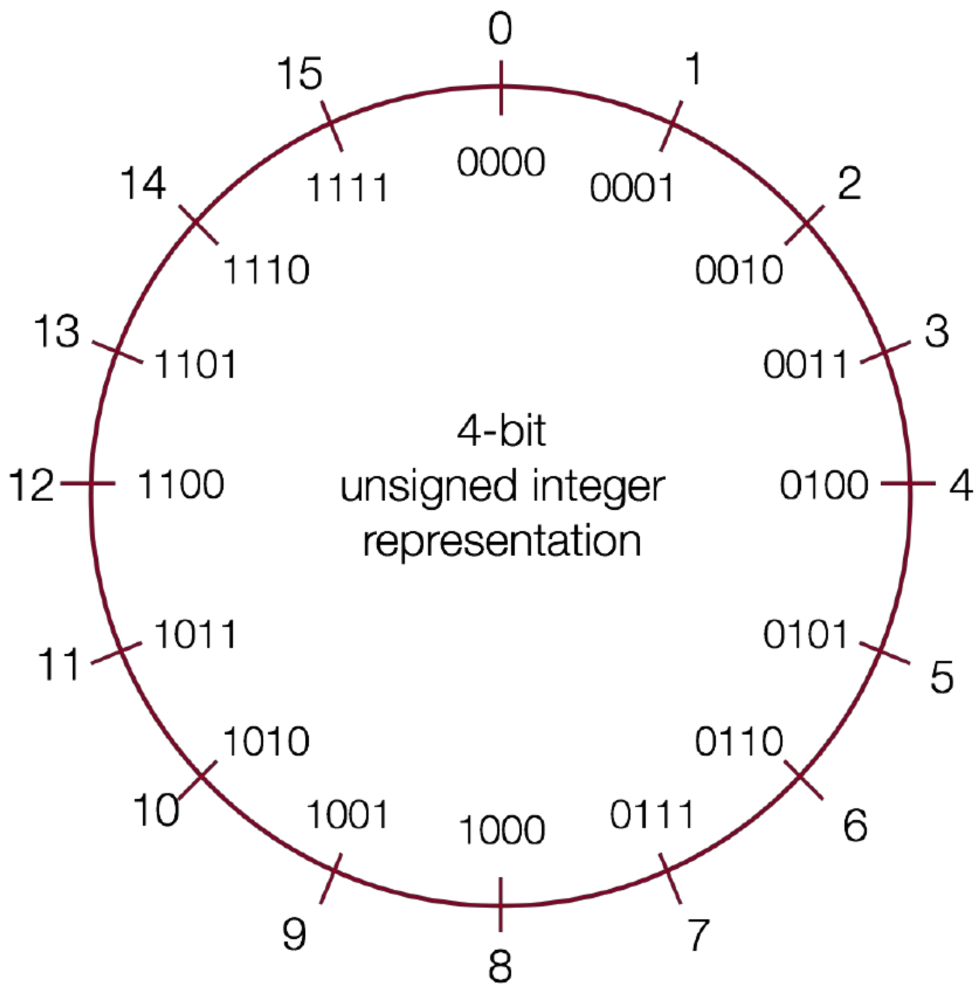
Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- We have already discussed converting between decimal and binary, which is a nice 1:1 relationship. Examples:

```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

- The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits. E.g. a 32-bit integer can represent 0 to $2^{32} - 1$ (4,294,967,295).

Unsigned Integers



Sign Magnitude Representation

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

- We've only represented 15 of our 16 available numbers!

Sign Magnitude Representation

- **Pro:** easy to represent, and easy to convert to/from decimal.
- **Con:** +-0 is not intuitive
- **Con:** we lose a bit that could be used to store more numbers
- **Con:** arithmetic is tricky: we need to find the sign, then maybe subtract (borrow and carry, etc.), then maybe change the sign. This complicates the hardware support for something as fundamental as addition.

Can we do better?

A Better Idea

Decimal	Positive	Negative
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

Decimal	Positive	Negative
8	1000	1000
9	1001 (same as -7!)	NA
10	1010 (same as -6!)	NA
11	1011 (same as -5!)	NA
12	1100 (same as -4!)	NA
13	1101 (same as -3!)	NA
14	1110 (same as -2!)	NA
15	1111 (same as -1!)	NA

There Seems Like a Pattern Here…

0101	0011	0000
+ 1011	+ 1101	+ 0000
0000	0000	0000

- The negative number is the positive number **inverted, plus one!**

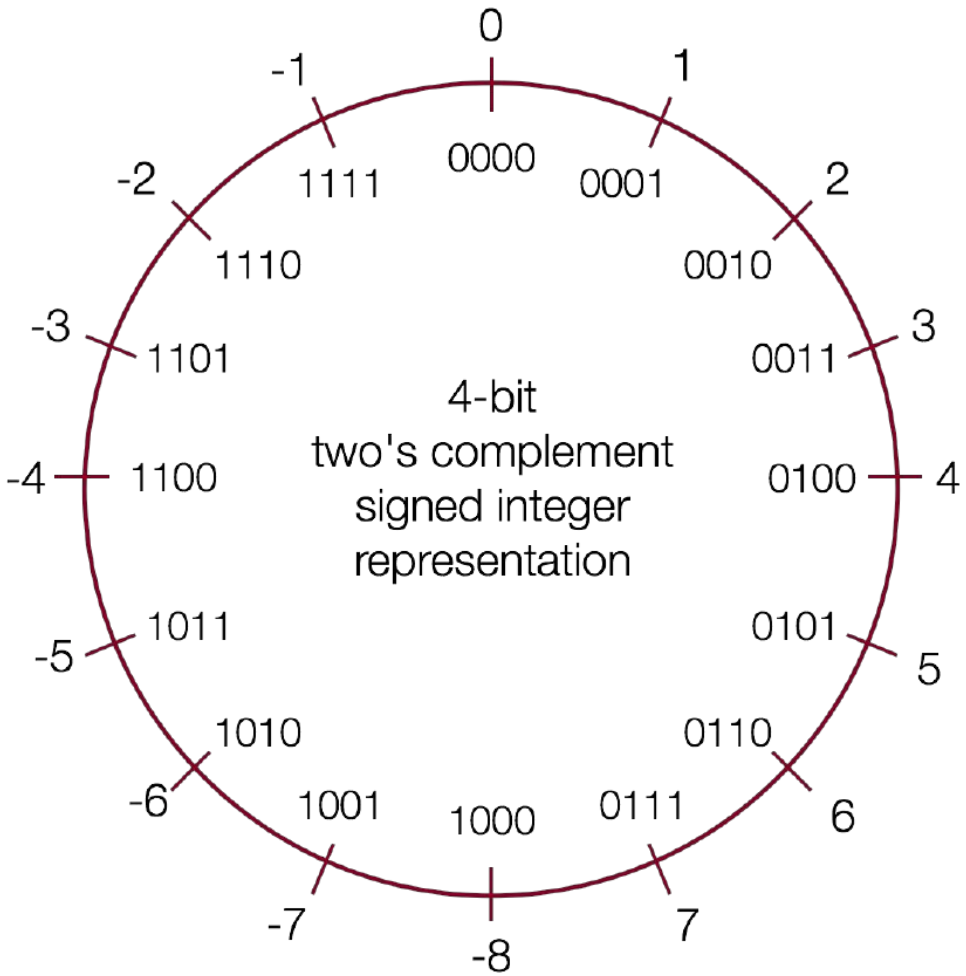
Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\ {\color{red}011100} \\
\hline
000000
\end{array}
$$

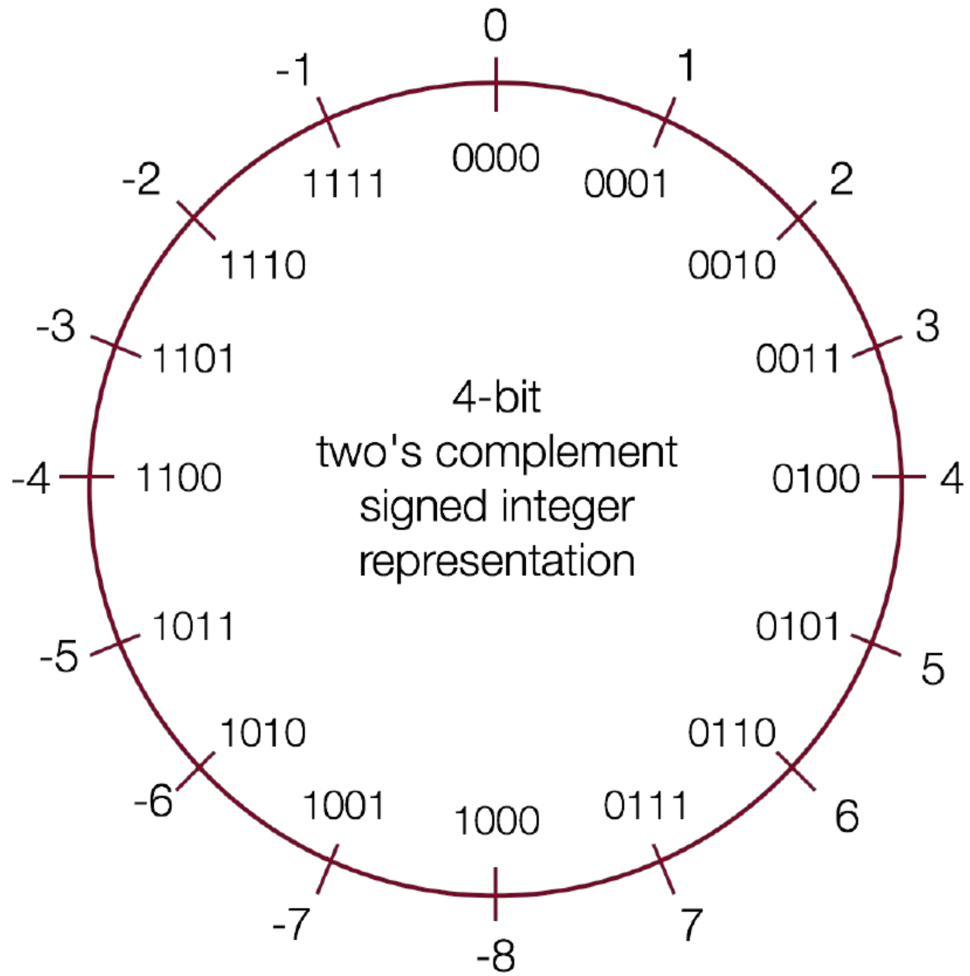
Two’s Complement

- 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!



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:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!



Overflow

- 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 *overflow* back to the **largest** bit representation.

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

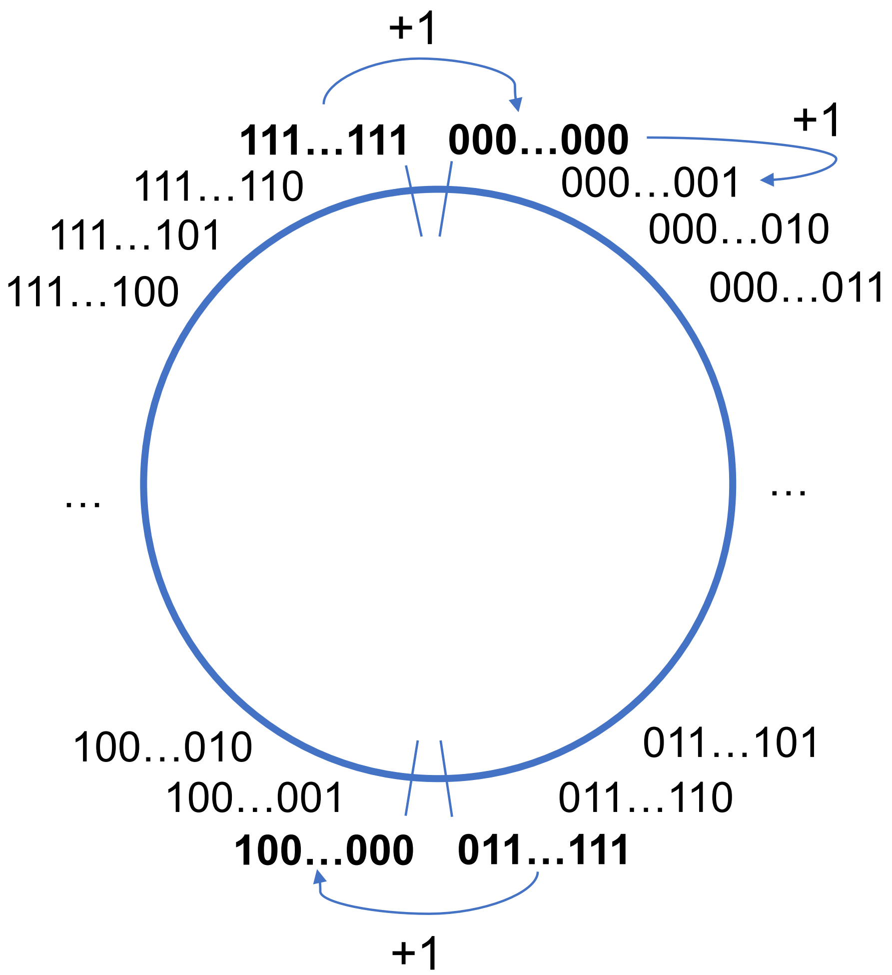
Min and Max Integer Values

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

Min and Max Integer Values

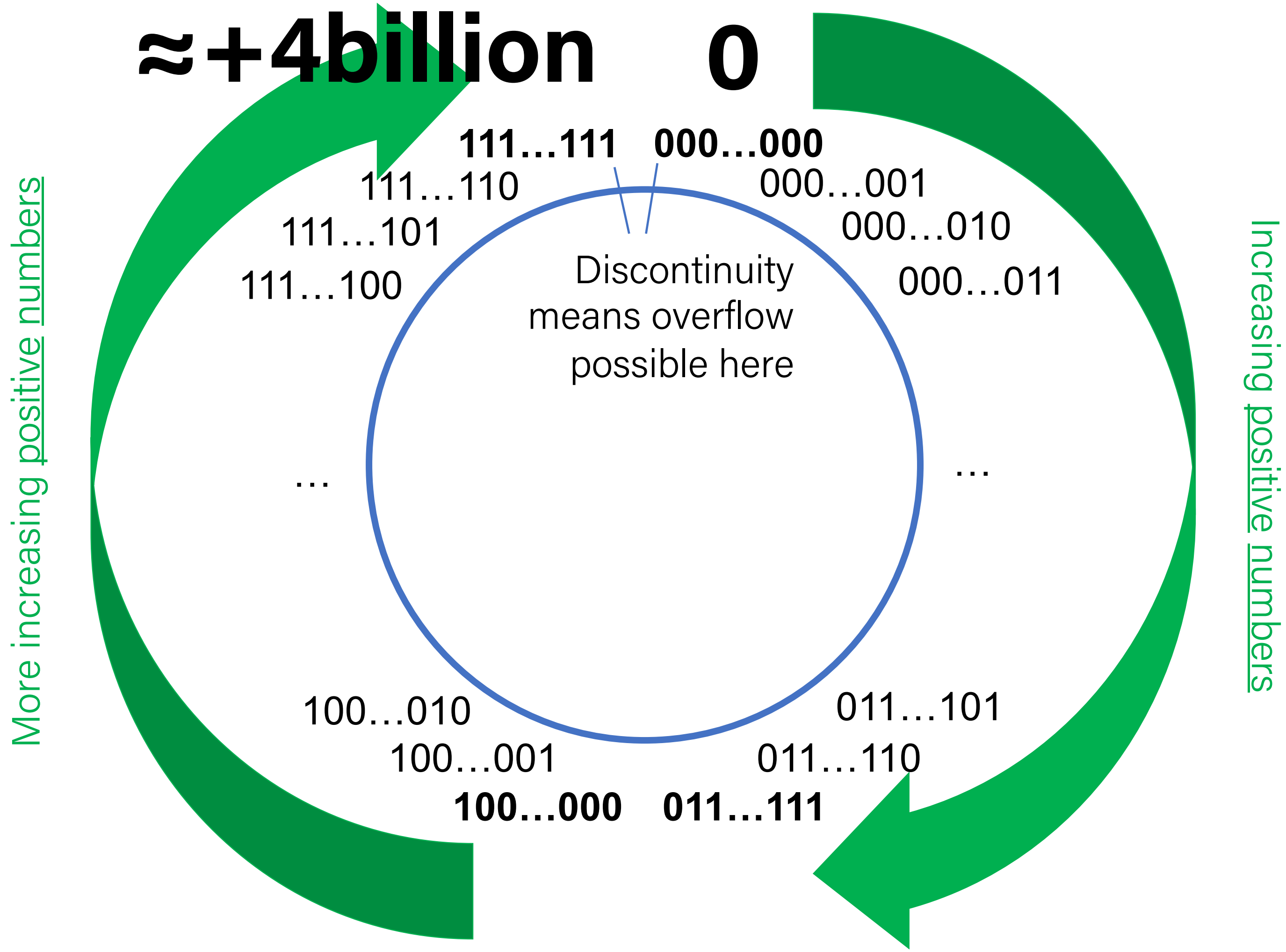
`INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX, …`

Overflow

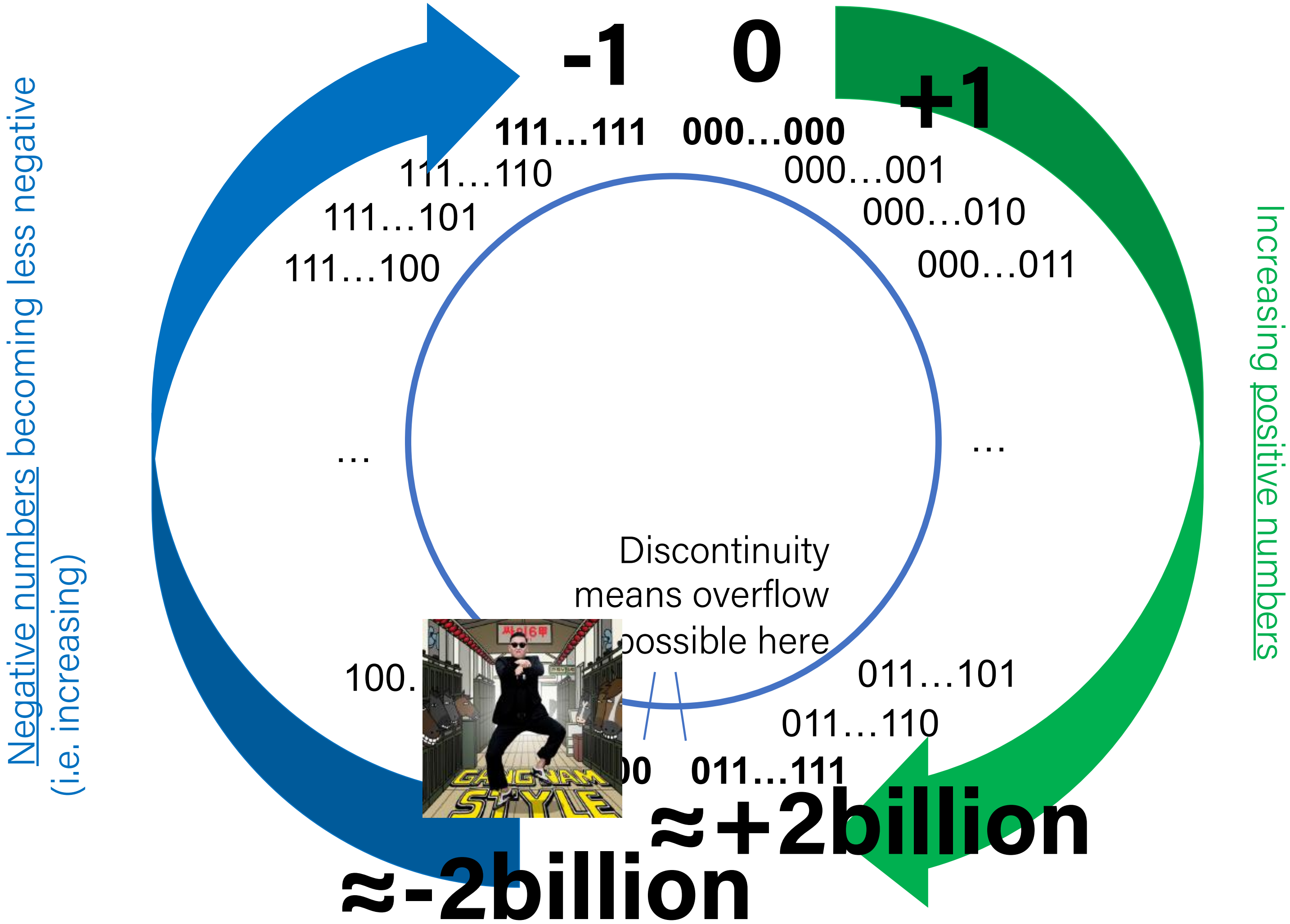


+1
111…111
000…000
+1
111…110
000…001
111…101
000…010
111…100
000…011
…
…
100…010
011…101
100…001
011…110
100…000
011…111
+1

Unsigned Integers



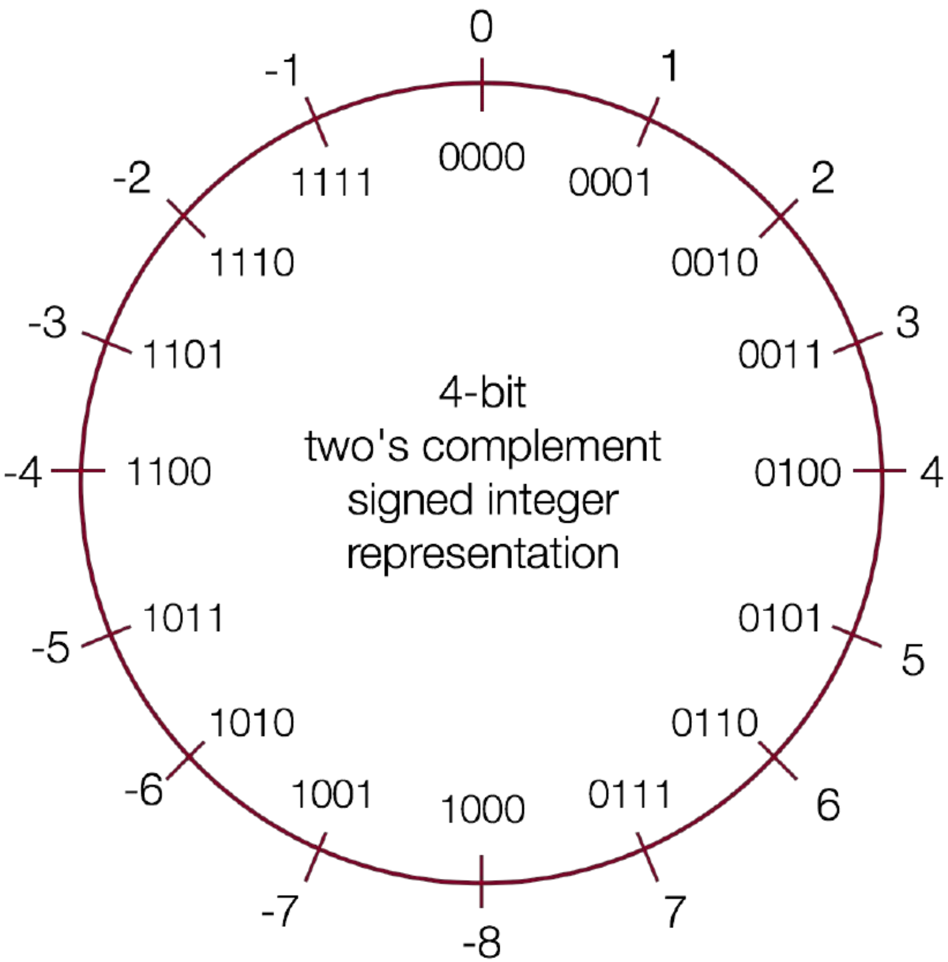
Signed Numbers

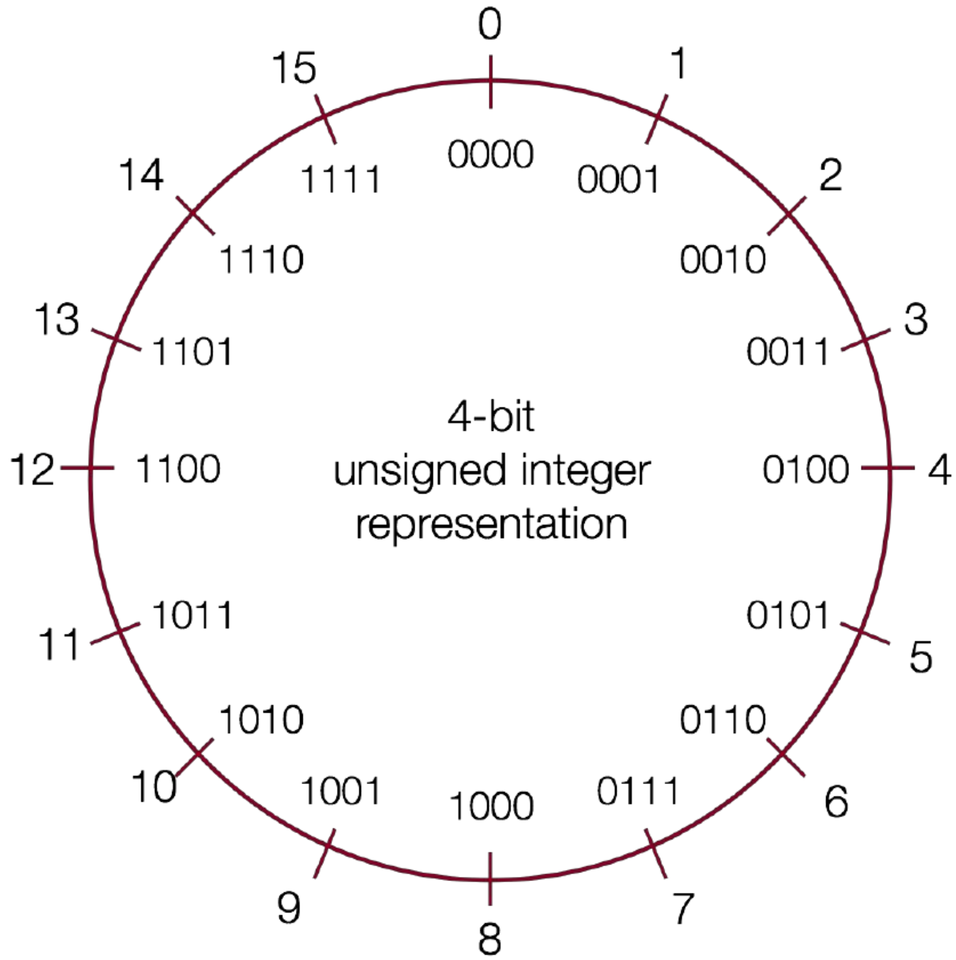


`printf` and Integers

- 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
- The placeholder—not the expression filling in the placeholder—dictates what gets printed!

Casting

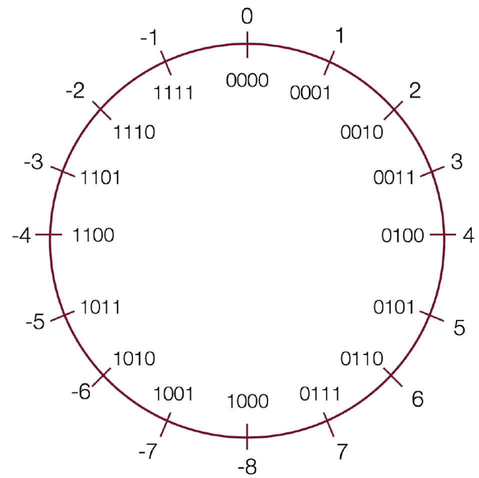

4-bit
two's complement
signed integer
representation
0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-8
1000
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111


4-bit
unsigned 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

Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

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

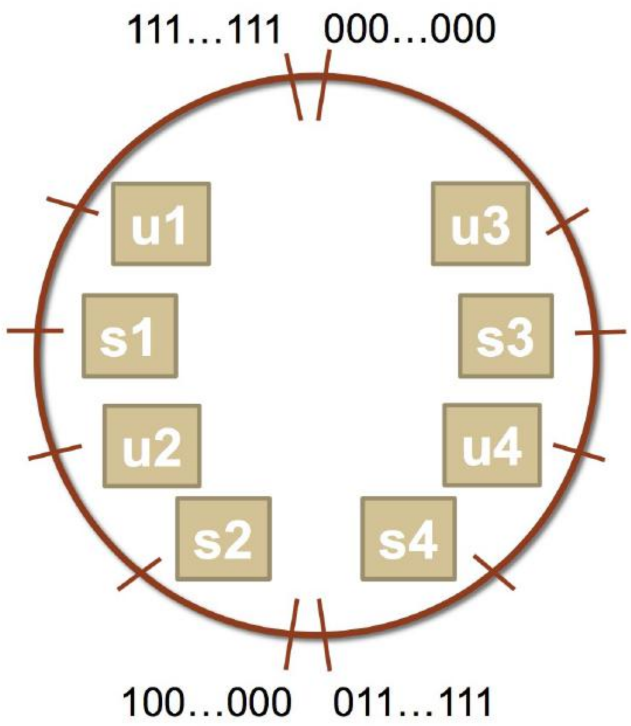


Type	Size (Bytes)	Minimum	Maximum
`int`	4	-2147483648	2147483647
`unsigned int`	4	0	4294967295

Comparisons Between Different Types

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

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



Expanding Bit Representations

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

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

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.

Summary: Basic Rules of Expanding, Truncating

- Expanding (e.g., `short int` to `int`)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., `unsigned` to `unsigned short`)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Slide credit: R.E. Bryant and D.R. O'Hallaron

Byte Ordering

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

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

Byte Ordering Example

- Big Endian: Sun (Oracle SPARC), PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86, ARM processors running Android, iOS, and Linux
  - Least significant byte has lowest address
- Example:
  - Variable `x` has 4-byte value of `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian

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

Little Endian

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

Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

`int A = 15213;`　　`int B = -15213;`　　`long int C = 15213;`