Integer Representation

Announcements

assign0 due tonight

assign1 out tomorrow

Labs start this week

SCPD

- Note on office hours on Piazza
- Will get an email tonight about labs

Goals for Today

Introduction to computer arithmetic

- Binary addition and subtraction
- Overflow

Represent negative numbers

Compare/contrast unsigned and signed integers

Add two 1-bit numbers

a	b	a + b
0	0	00
0	1	01
1	0	01
1	1	10

Add two 1-bit numbers

a	b	a + b
0	0	00
0	1	01
1	0	01
1	1	10

LSB looks like XOR, next bit looks like AND

```
a + b = ((a & b) << 1) | (a ^ b)
```

(Only when a and b are 1-bit)

Add two 1-bit numbers

```
a   b   a + b
0   0     00
0   1     01
1   0     01
1   1     10
```

LSB looks like XOR, next bit looks like AND

`a + b = ((a & b) << 1) | (a ^ b)`

(Only when a and b are 1-bit)

This is called a half adder

"Full" addition requires us to handle carries

Addition in Binary

```
   1
  107
+  58
-----
  165
```

Addition in Binary

```
   1
  107        0110 1011
+  58      + 0011 1010
-----      -----------
  165
```

Addition in Binary

```
   1
  107        0110 1011
+  58      + 0011 1010
-----      -----------
  165               1
```

Addition in Binary

```
  1                1
  107          0110 1011
+  58      + 0011 1010
-----      -----------
  165              01
```

Addition in Binary

```
  1              1  1
 107          0110 1011
+ 58        + 0011 1010
-----       ----------
 165            0101
```

Addition in Binary

```
  1          1111  1
 107          0110 1011
+ 58        + 0011 1010
-----       ----------
 165          1010 0101
```

Same as decimal addition, just a lot more carrying

Overflow

```
  255        1111 1111
+   1    + 0000 0001
-----    -----------
```

What happens when we run out of bits?

Overflow

```
                    1
   255        1111 1111
 +   1    + 0000 0001
------    -----------
                    0
```

What happens when we run out of bits?

Overflow

```
   255          1 1111 111
 +   1        +  1111 1111
-------      + 0000 0001
             ------------
             1 0000 0000
```

What happens when we run out of bits?

Overflow

```
   255          1111 111
 +   1        + 0000 0001
-----        -----------
     0          0000 0000
```

What happens when we run out of bits?

- Overflow: extra bit is dropped
- No warning, no indication

Overflow

```
   255          1111 111
 +   1        + 0000 0001
-------      -----------
     0          0000 0000
```

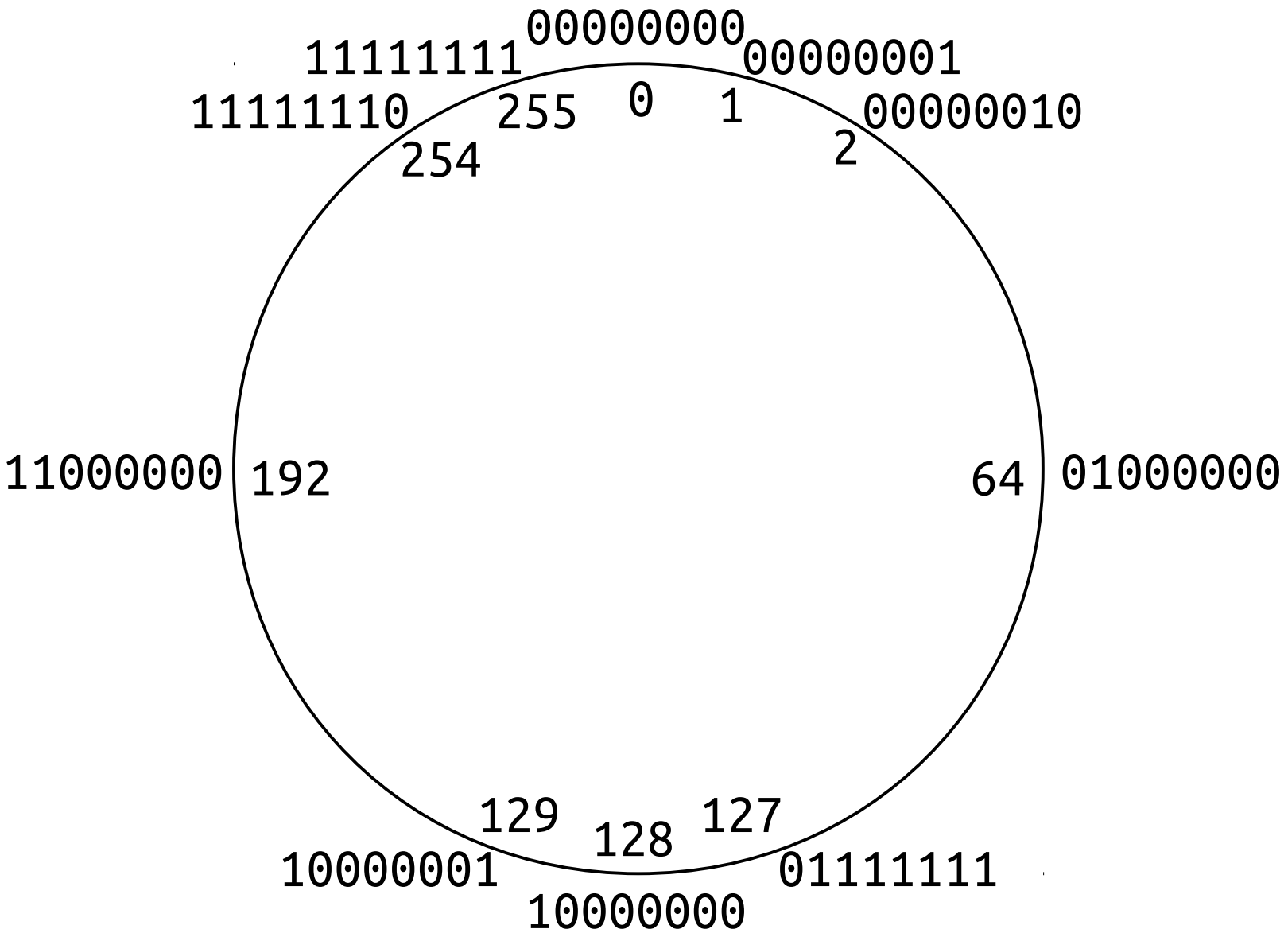
Modular arithmetic

Overflow leads to wrapping around, mod by 256

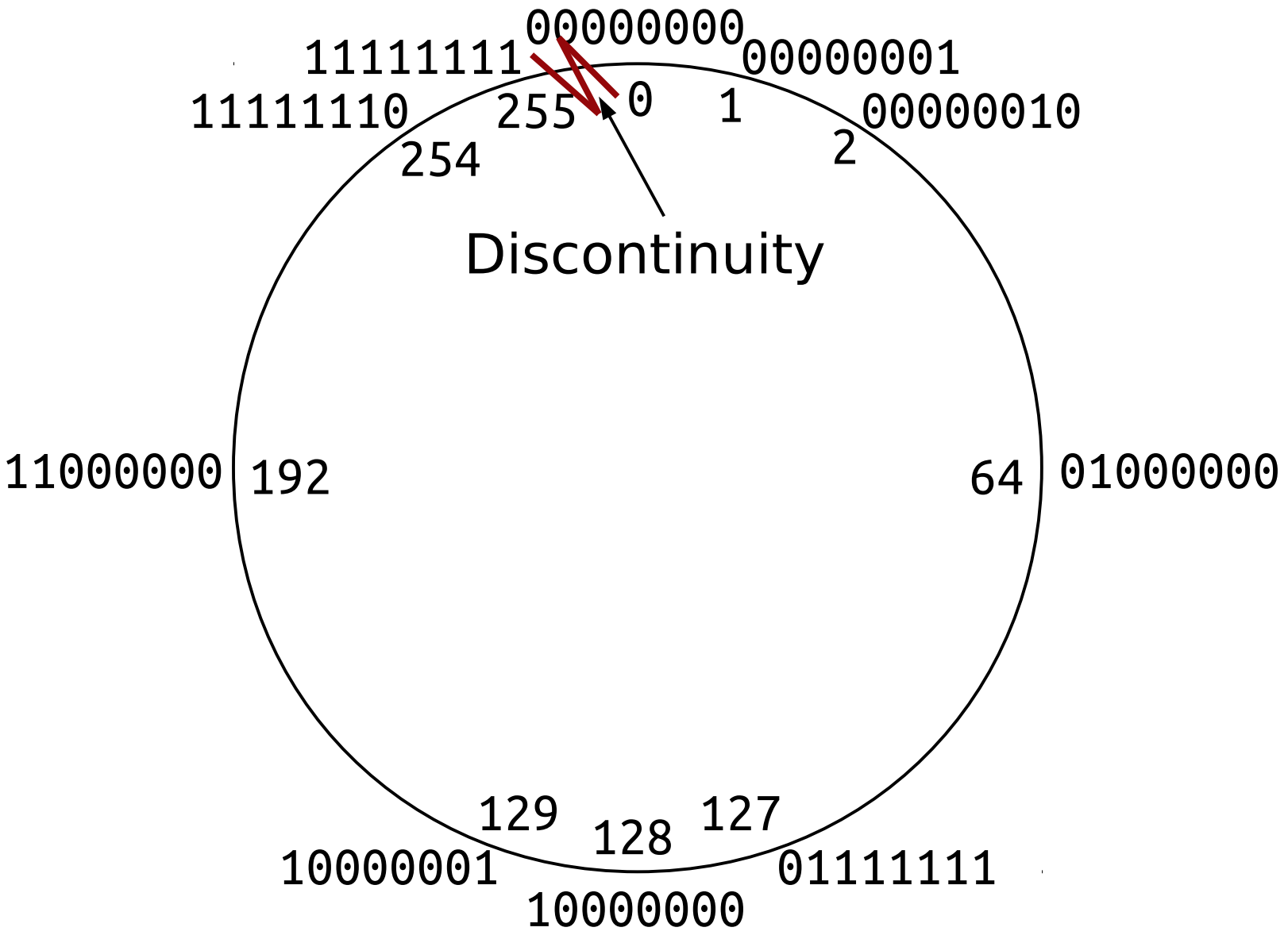
`0x100 % 256 = (1 0000 0000) % 256 = 0`

$256 = 2^8$

`unsigned char` Number Circle



`unsigned char` Number Circle



What about subtraction?

Could use same approach as addition

- Start with 1-bit numbers, then deal with borrowing

What about subtraction?

Could use same approach as addition

- Start with 1-bit numbers, then deal with borrowing
- But we don't like inventing new hardware

Idea: Subtraction is just adding the negative

- `a - b` becomes `a + (-b)`
- New problem: How do we represent negatives?

Deriving -1

$$\begin{array}{rr} & 1 \\ + & -1 \\ \hline & 0 \end{array}$$

Deriving -1

```
     1          0000 0001
+   -1      + ???? ????
-------     -----------
     0          0000 0000
```

Let's backsolve for -1

We know that `1 + (-1) = 0`

Deriving -1

```
      1          0000 0001
+    -1     +    ???? ???1
-------    ---------------
      0          0000 0000
```

Deriving -1

```
                        11
       1        0000 0001
 +    -1     +  ???? ??11
 ---------  --------------
       0        0000 0000
```

Deriving -1

```
                1 1111 111
       1           0000 0001
 +    -1         + 1111 1111
 ---------       ------------
       0           0000 0000
```

Deriving -1

```
      1          1 1111 111
  +  -1        + 1111 1111
-------      ---------------
      0          1 0000 0000
```

Did it...work?

What about that extra 1?

Deriving -1

```
      1          1111 111
+    -1        + 1111 1111
-------        ----------
      0          0000 0000
```

Did it...work?

What about that extra 1?

Arithmetic is % 256, so it's dropped

What have we learned?

For 1-byte numbers

-1 is `0xff = 1111 1111 (bin)`

Note: same bit pattern as 255 in unsigned

`0 = 0x100 % 256 = 1 0000 0000 (bin) % 256`

But how do we generalize for other negatives?

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
```

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
```

Reminder: `0x100 = 256 = 1 0000 0000 (bin)`

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
   = (0xff + 1) - n
```

Reminder: `0xff = 255 = 1111 1111 (bin)`

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
   = (0xff + 1) - n
   = (0xff - n) + 1
```

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
   = (0xff + 1) - n
   = (0xff - n) + 1
```

Wouldn't it be awesome if there were a simple way to get `0xff - n`?

0xff - n

```
   0xff       1111 1111
 - 0x35     - 0011 0101
```

0xff - n

```
   0xff       1111 1111
 - 0x35     - 0011 0101
----------  -----------
                     10
```

0xff - n

```
   0xff       1111 1111
 - 0x35     - 0011 0101
-------    -----------
   0xca       1100 1010
```

There's always a 1 to take, never need to borrow

0xff - n

```
   0xff       1111 1111
 - 0x35     - 0011 0101
-------    ------------
   0xca       1100 1010
```

```
0xff - n = ~n
```

For 1-byte n

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
   = (0xff + 1) - n
   = (0xff - n) + 1
```

Wouldn't it be awesome if there were a simple way to get `0xff - n`?

Arithmetic Negation

Let's try something different...

Let n be a 1-byte number

```
-n = 0 - n
   = 0x100 - n
   = (0xff + 1) - n
   = (0xff - n) + 1
```

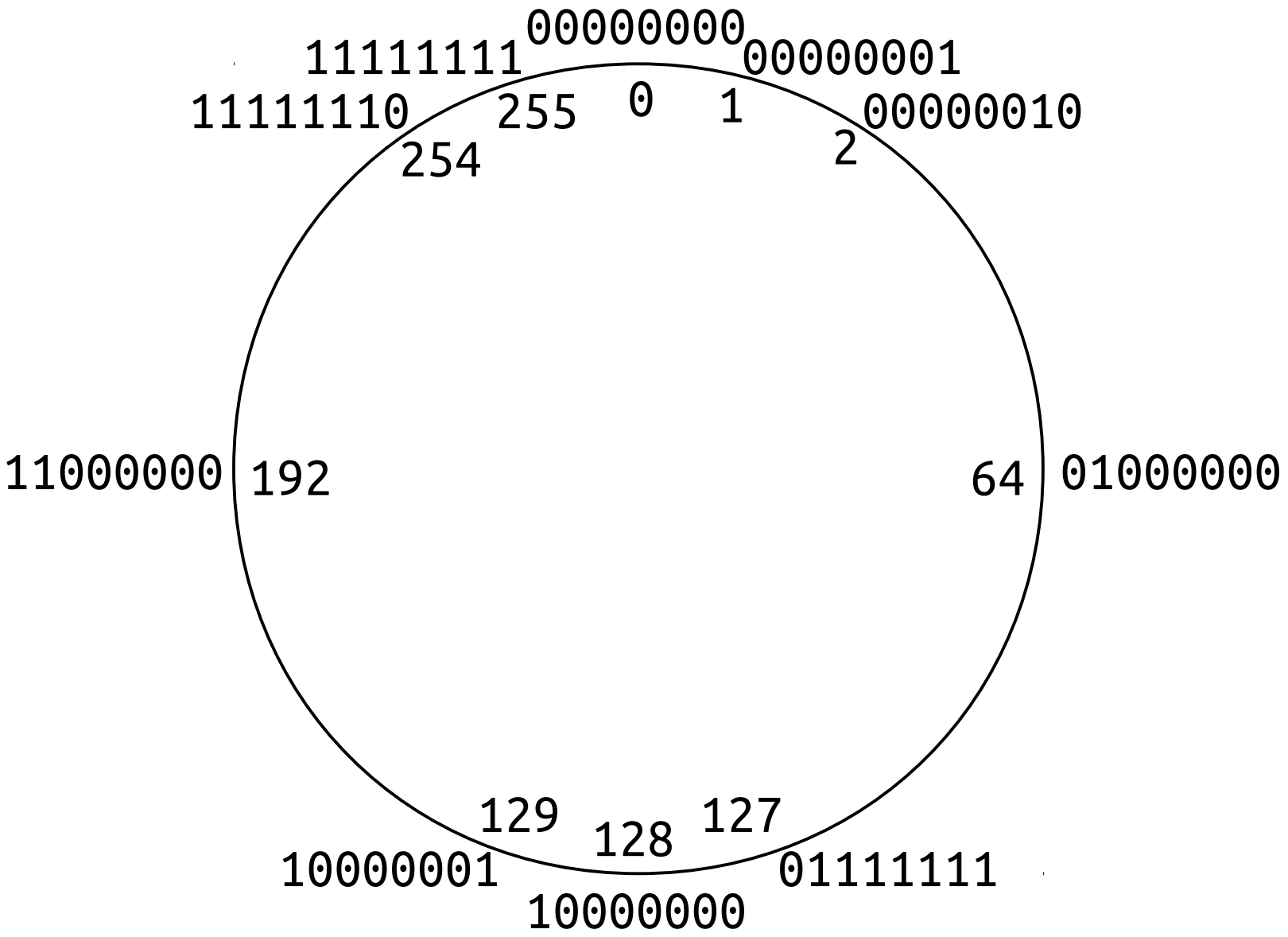
`-n = ~n + 1`

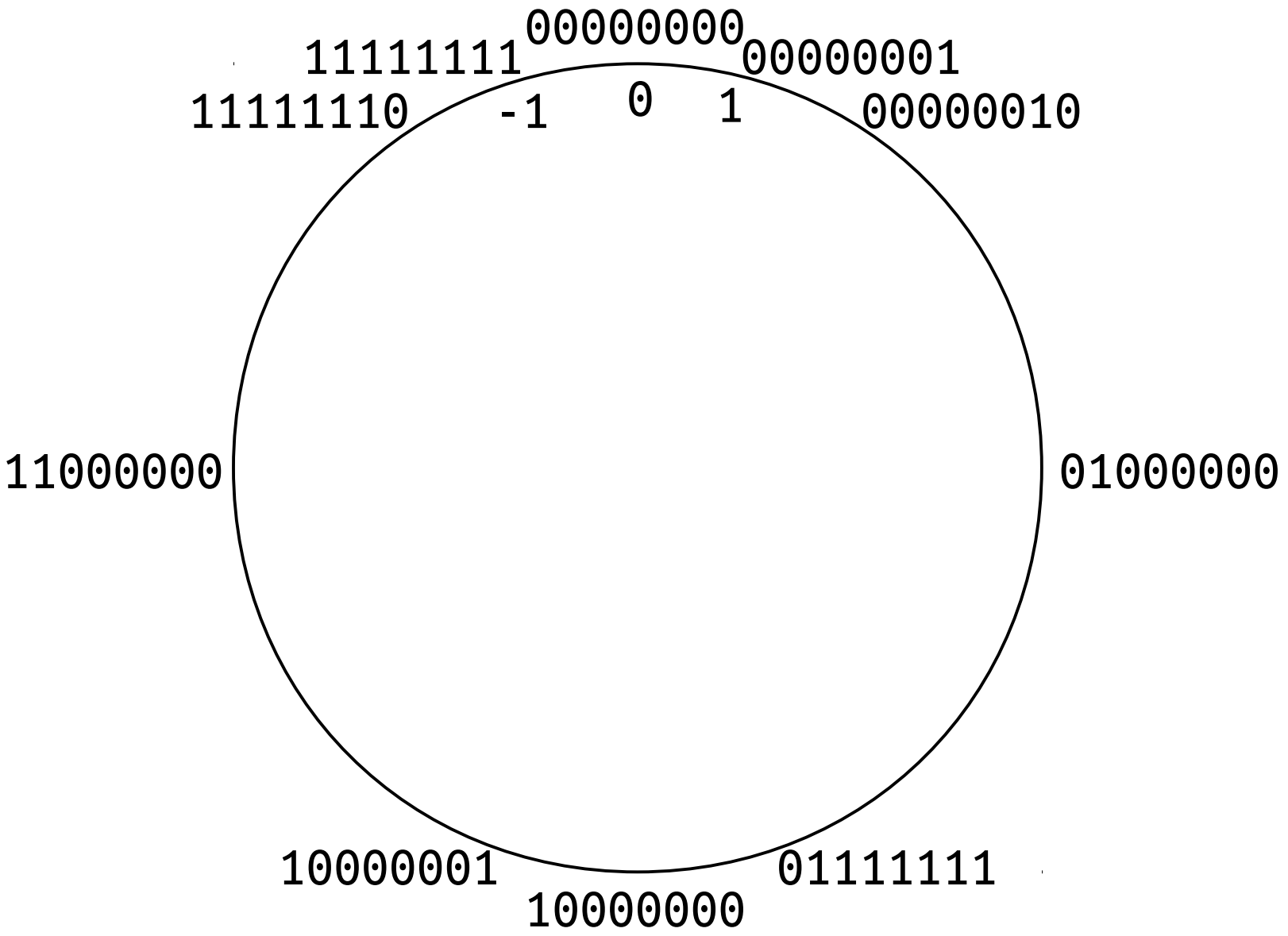
!!

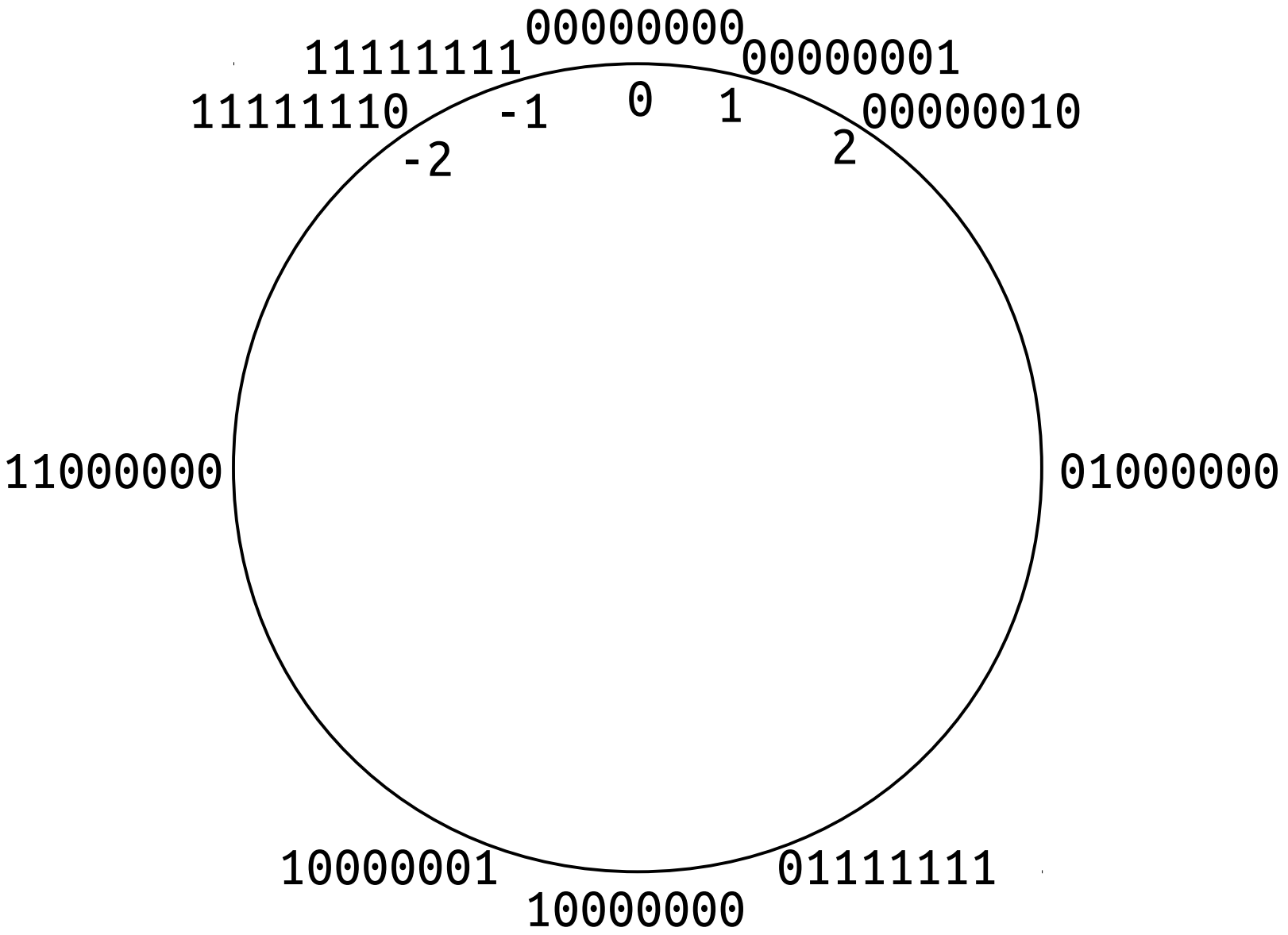
This is called 2's complement

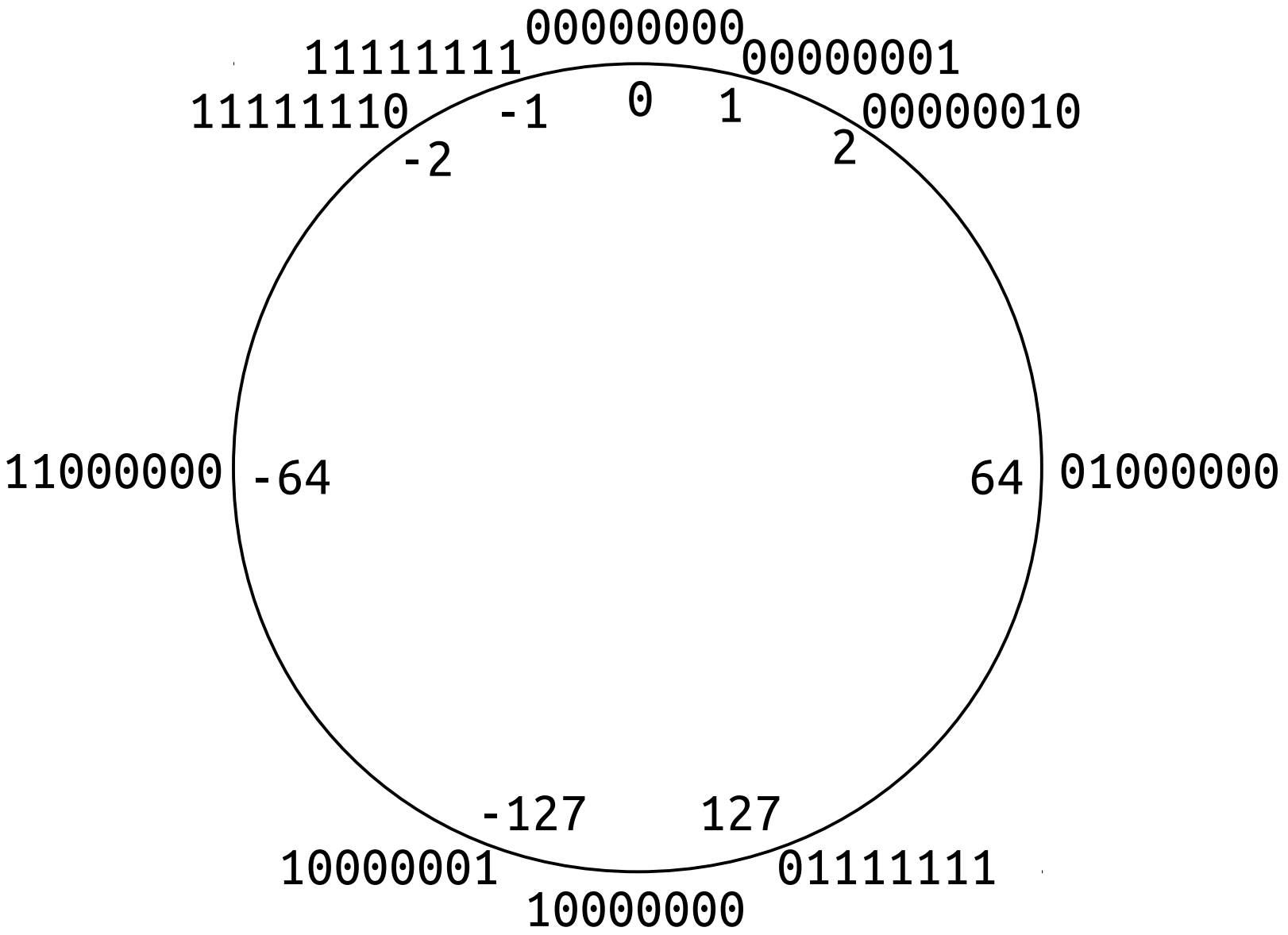
Works for all integer types, not just 1-byte

`unsigned char` Number Circle

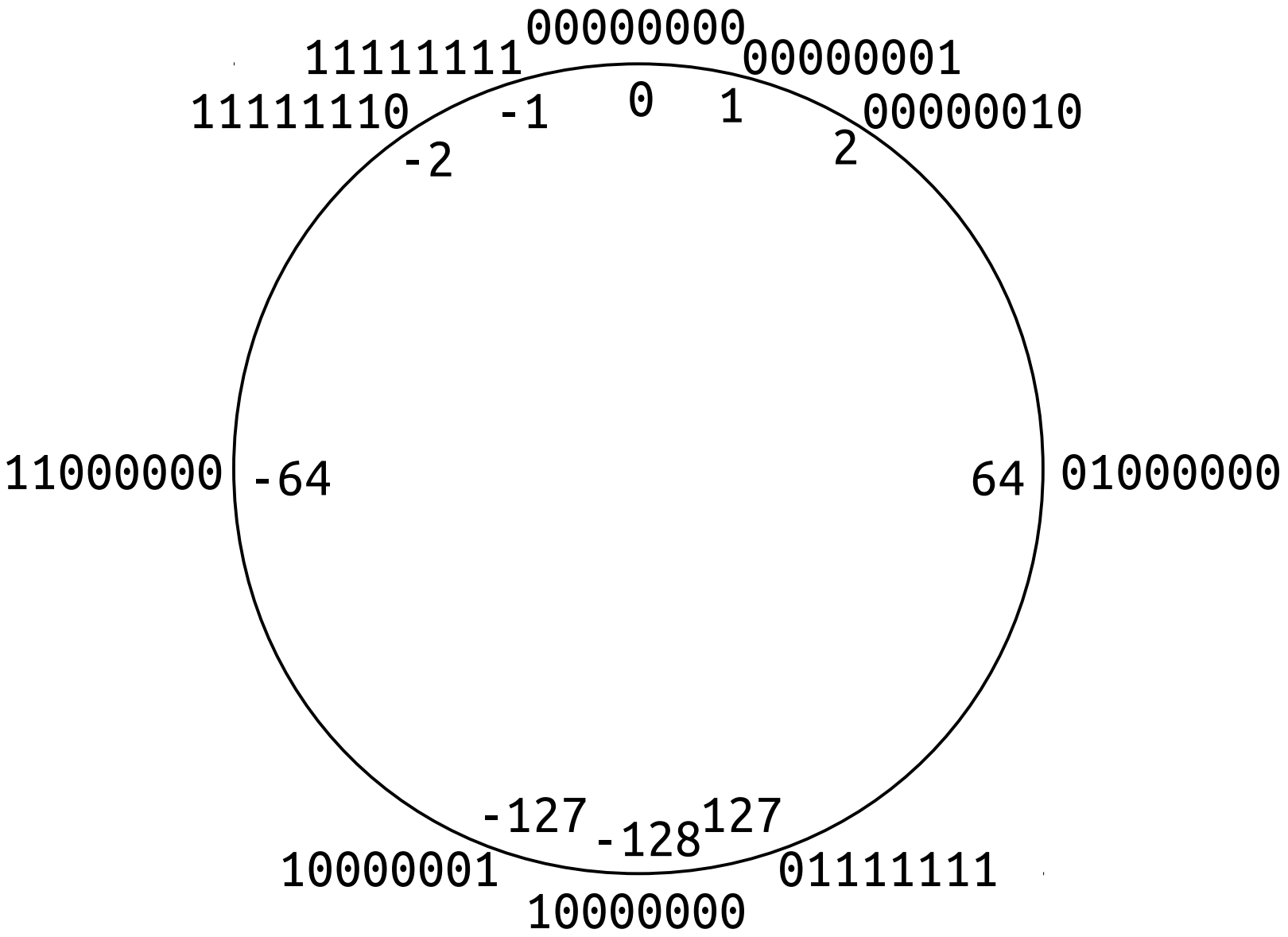



00000000
0
00000001
1
11111111
-1
00000010
11111110
11000000
01000000
10000001
01111111
10000000

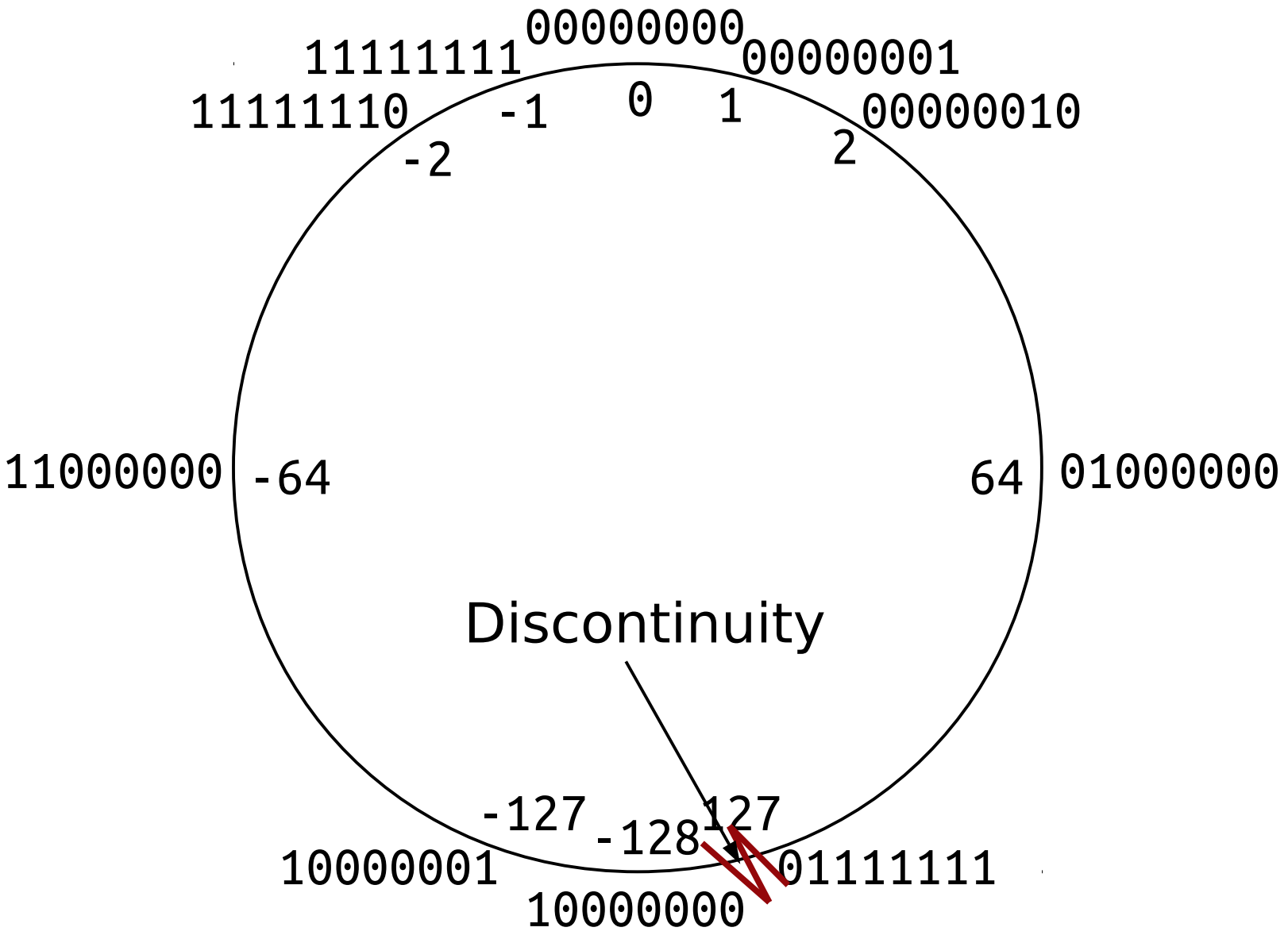

00000000
0
00000001
1
00000010
2
11111111
-1
11111110
-2
01000000
01111111
10000000
10000001
11000000


00000000
0
00000001
1
00000010
2
11111111
-1
11111110
-2
11000000
-64
64
01000000
-127
127
10000001
01111111
10000000

`signed char` Number Circle



`signed char` Number Circle



`signed char`

`-n = ~n + 1`

Same 256 bit patterns

- Half the bit patterns used for negatives
- MSB = 1 means negative (sign bit)

Discontinuity

- Different place than unsigned

Asymmetry

- What is `-(-128)`?

(signed char) -(-128)

```
-(-128) = -0x80
        = ~0x80 + 1
        = 0x7f + 1
        = 0x80 = -128
```

```
-(1000 0000)
~(1000 0000) + 1
  0111 1111  + 1
  1000 0000
```

-128 is its own arithmetic inverse

Larger Integer Types

C integer types

All but `char` default to signed (`char`'s default is not defined)

Data type sizes

`char`: 1 byte, -128 to 127

`short`: 2 bytes, +/- 32k

`int`: 4 bytes, +/- 2 billion

`long`: 8 bytes, +/- [big number]

Code Examples

Signed vs. Unsigned

What's the same?

- Most bitwise ops (`&`, `|`, `^`, `~`, `<<`)
- Most arithmetic ops (`+`, `-`, `*`)

What's different?

- Comparison (`>`, `<`)
- Right shift: arithmetic vs .logical
  - Fill with sign bit vs. zero
- Assignment to larger type (e.g. assign `short` to `int`)
  - Sign extend vs. zero extend
  - Preserves value

Conclusions

In the end, all operations manipulate bits

- Assignment, parameter passing just copy bits around

Signed and unsigned are interpretations of bit patterns

- Bits aren't fundamentally one or the other
- Just like binary, decimal, hex are ways to read/write and think about bits

Summary

Introduction to computer arithmetic

- Binary addition and subtraction
- Overflow

Represent negative numbers

Compare/contrast unsigned and signed integers

This week's lab and assignment

- Practice with bits, bitwise ops, ints

Next time: pointers and memory