CSCI 2330

judge: I sentence you to the maximum punishment, 255 years

Me. May I have one day more?

judge: Ok, you have to serve 0 years in prison

Overview for today

- Recap binary arithmetic
- Logical operators
- Signed numbers

Binary arithmetic

- We will be learning some old tricks and some new tricks here!
- Revisiting grade school arithmetic: +, *
- New stuff: ^, &, |, !, ~

Binary arithmetic

```
  0110
+ 0100
```

Binary arithmetic

```
   1
   0110 =  6₁₀
+  0100 =  4₁₀
   1010 = 10₁₀
```

Binary arithmetic

```
  110
*  11
```

Binary arithmetic

```
   110 =  6₁₀
*   11 =  3₁₀
11
  110
 1100
10010 = 18₁₀
```

Binary and logical operators

`&`	bitwise and
`|`	bitwise or
`^`	bitwise xor
`~`	bitwise negation
`&&`	logical and
`||`	logical or
`!`	logical negation

Truth tables

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~: a unary operator that flips the bit

Binary operations

```
  1100
& 1010
  ????
```

Binary operations

```
  1100
& 1010
  1000
```

Binary operations

```
  1100
| 1010
  ????
```

Binary operations

```
  1100
| 1010
  1110
```

Binary operations

```
  1100
^ 1010
  ????
```

Binary operations

```
  1100
^ 1010
  0110
```

Binary operations

```
~1100 = ????
```

Binary operations

```
~1100 = 0011
```

Logical operations

- &&, ||, !
- These work a little differently than in Java
- In Java: evaluate to a boolean
- In C: evaluate to an int
- In Java: the operands must be booleans
- In C: the operands can be numbers

Logical operations

- In C, operands are reduced:
  - If zero → 0
  - If nonzero → 1
  - Then the normal bit operation is applied

Examples

- 101010110101 && 101010101011 → ?

Examples

- 101010110101 && 101010101011 → ?
- 1 & 1 → ?

Examples

- 101010110101 && 101010101011 → ?
- 1 & 1 → ?
- 1

Examples

- 101010110101010101 && 0 → ?

Examples

- 10101011010101 && 0 → ?
- 1 & 0 → ?

Examples

- 10101011010101 && 0 → ?
- 1 & 0 → ?
- 0 → ?

Examples

- 10101011010101 || 0 → ?

Examples

- 10101011010101 || 0 → ?
- 1 | 0 → ?

Examples

- 10101011010101 || 0 → ?
- 1 | 0 → ?
- 1

Examples

- 0 || 0 → ?

Examples

- 0 || 0 → ?
- 0

Examples

- !101010110101 → ?
- !1 → ?
- 0

Examples

- !0 → ?
- 1

Puzzle

- What expression (in terms of *x*) evaluates to 0 if *x* is zero, 1 otherwise?

Two special operators

- Bit shifts: >>, <<

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101
- 01101 >> 1 → 00110

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101
- 01101 >> 1 → 00110
- 01101 >> 2 → 00011

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101
- 01101 >> 1 → 00110
- 01101 >> 2 → 00011
- 01101 >> 3 → 00001

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101
- 01101 >> 1 → 00110
- 01101 >> 2 → 00011
- 01101 >> 3 → 00001
- 01101 >> 4 → 00000

Two special operators

- Bit shifts: >>, <<
- 01101 >> 0 → 01101
- 01101 >> 1 → 00110
- 01101 >> 2 → 00011
- 01101 >> 3 → 00001
- 01101 >> 4 → 00000
- 01101 >> 5 → 00000

Two special operators

- Bit shifts: >>, <<
- 00011 << 0 → 00011

Two special operators

- Bit shifts: >>, <<
- 00011 << 0 → 00011
- 00011 << 1 → 00110

Two special operators

- Bit shifts: >>, <<
- 00011 << 0 → 00011
- 00011 << 1 → 00110
- 00011 << 2 → 01100

Two special operators

- Bit shifts: >>, <<
- 00011 << 0 → 00011
- 00011 << 1 → 00110
- 00011 << 2 → 01100
- 00011 << 3 → 11000

Two special operators

- Some corner cases to bit shifts that we will discuss later
- For now this is the basic concept

What do shifts do arithmetically?

- 1 << 0 → 1
- 1 << 1 → 2
- 1 << 2 → 4
- 1 << 3 → 8
- 1 << 4 → 16

- 3 << 0 → 3
- 3 << 1 → 6
- 3 << 2 → 12
- 3 << 3 → 24
- 3 << 4 → 48

What do shifts do arithmetically?

- 1 << 0 → 1
- 1 << 1 → 2
- 1 << 2 → 4
- 1 << 3 → 8
- 1 << 4 → 16
- $x << n == x * 2^n$

- 3 << 0 → 3
- 3 << 1 → 6
- 3 << 2 → 12
- 3 << 3 → 24
- 3 << 4 → 48

What do shifts do arithmetically?

- $x >> n == x / 2^n$ (floor division)
- $16 >> 3 == 16 / 2^3 == 16 / 8 == 2$
- $17 >> 3 == 17 / 2^3 == 17 / 8 == 2$ (floor division!)

In-class exercise!

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Problem we haven’t directly addressed

- Machine integers have fixed widths

C integer types

- 8 bits → char
- 16 bits → short
- 32 bits → int
- 64 bits → long / long long
- *on the machines we will be working on

Problem we haven’t directly addressed

- Where do bits go when they fall off the edge?

(Assuming 8 bit integers)

- 00001111 >> 1 → 00000111
- 11110000 << 1 → 11100000
- The bit(s) are lost
  - Note: later we will learn how in certain conditions we can recover them!

(Assuming 8 bit integers)

$$
\begin{array}{rll}
 & 10000000 & = 128_{10} \\
+ & \underline{10000000} & = 128_{10} \\
 & ???????? &
\end{array}
$$

(Assuming 8 bit integers)

$$
\begin{array}{rl}
 & 10000000 = 128_{10} \\
+ & \underline{10000000} = 128_{10} \\
 & ?0000000
\end{array}
$$

(Assuming 8 bit integers)

```
  1
    10000000 = 128₁₀
+   10000000 = 128₁₀
    --------
    00000000
```

(Assuming 8 bit integers)

$$
\begin{array}{rll}
 & 10000000 & = 128_{10} \\
+ & \underline{10000000} & = 128_{10} \\
 & 00000000 &
\end{array}
$$

Integer overflow

- For an *n* bit integer, operations are effectively mod 2^n
- chars: mod 256
- shorts: mod 65536
- Et cetera

(Assuming 8 bit integers)

$$
\begin{array}{rl}
 & 10000000 = 128_{10} \\
+ & \underline{10000000} = 128_{10} \\
 & 00000000
\end{array}
$$

$$(128 + 128) \% 256 = ?$$

(Assuming 8 bit integers)

$$
\begin{array}{rl}
 & 10000000 = 128_{10} \\
+ & \underline{10000000} = 128_{10} \\
 & 00000000
\end{array}
$$

$$(128 + 128) \% 256 = 0$$

(Assuming 8 bit integers)

```
        1
  10000001 = 129₁₀
+ 10000001 = 129₁₀
  00000010
```

(129 + 129) % 256 = 2

Also called wraparound

```
x =  11111110 (254)
x++
x == 11111111 (255)
x++
x == 00000000 (0)
```

Also called wraparound

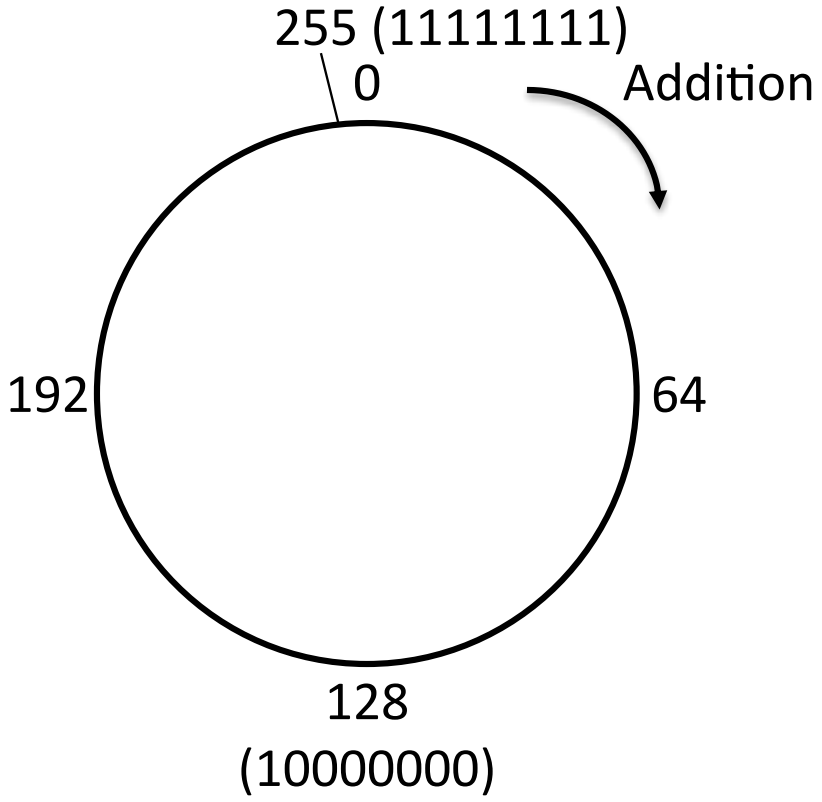
```
x =  11111110 (254)

x++

x == 11111111 (255)

x++

x == 00000000 (0)
```



judge: I sentence you to the maximum punishment, 255 years

Me. May I have one day more?

judge: Ok, you have to serve 0 years in prison


STAY
positive

Signed numbers

- We’ve been talking about *unsigned* numbers
- How do we encode that a number is negative?
- Remember: all memory is just 0s and 1s

Many different ways!

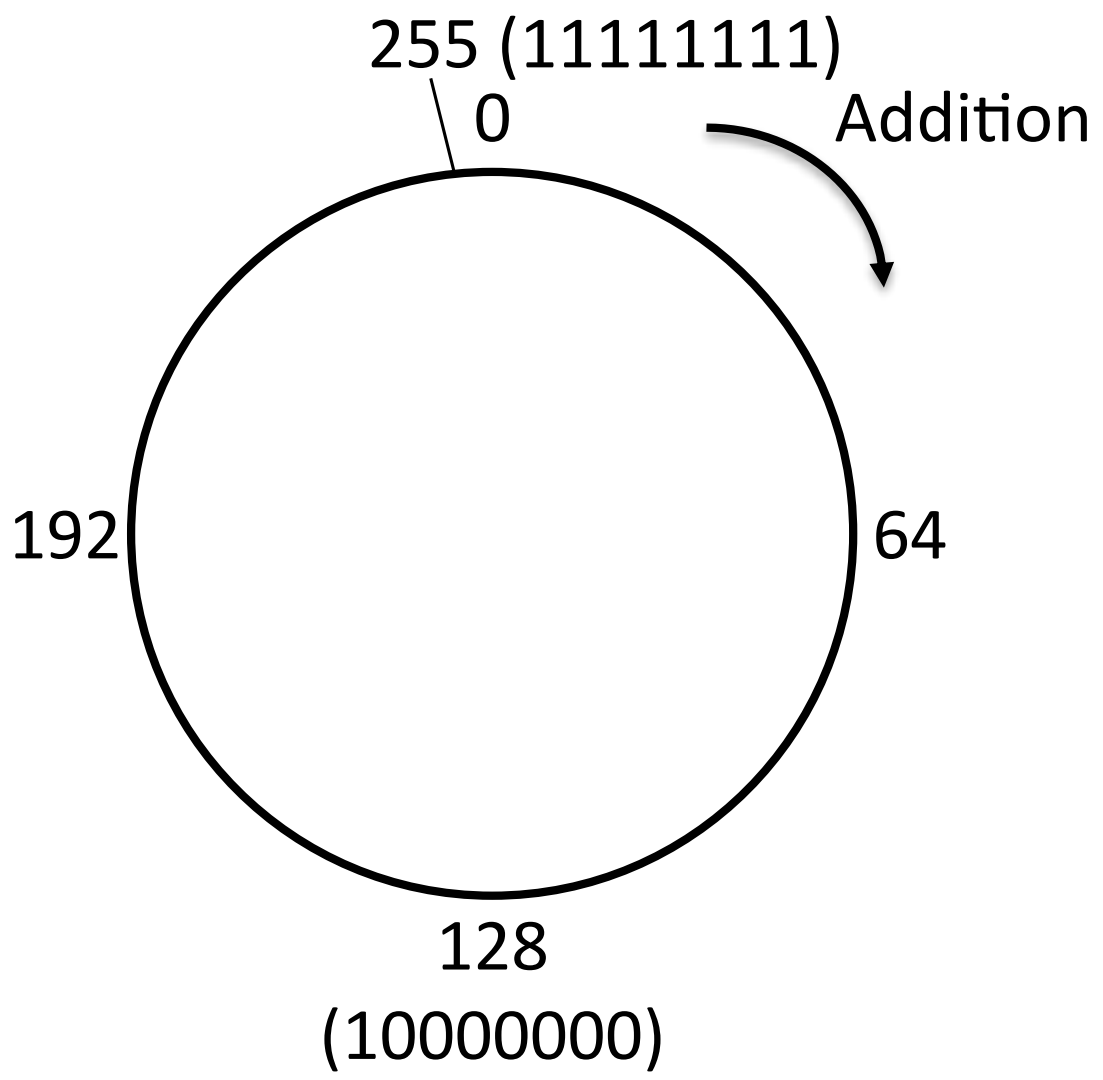
- Simplest way: sign bit
- Say you have 8-bit number
- Use the highest order bit to represent sign
- Traditionally, 0 positive, 1 negative
- 00000101 → 5
- 10000101 → -5
- “Signed magnitude” approach

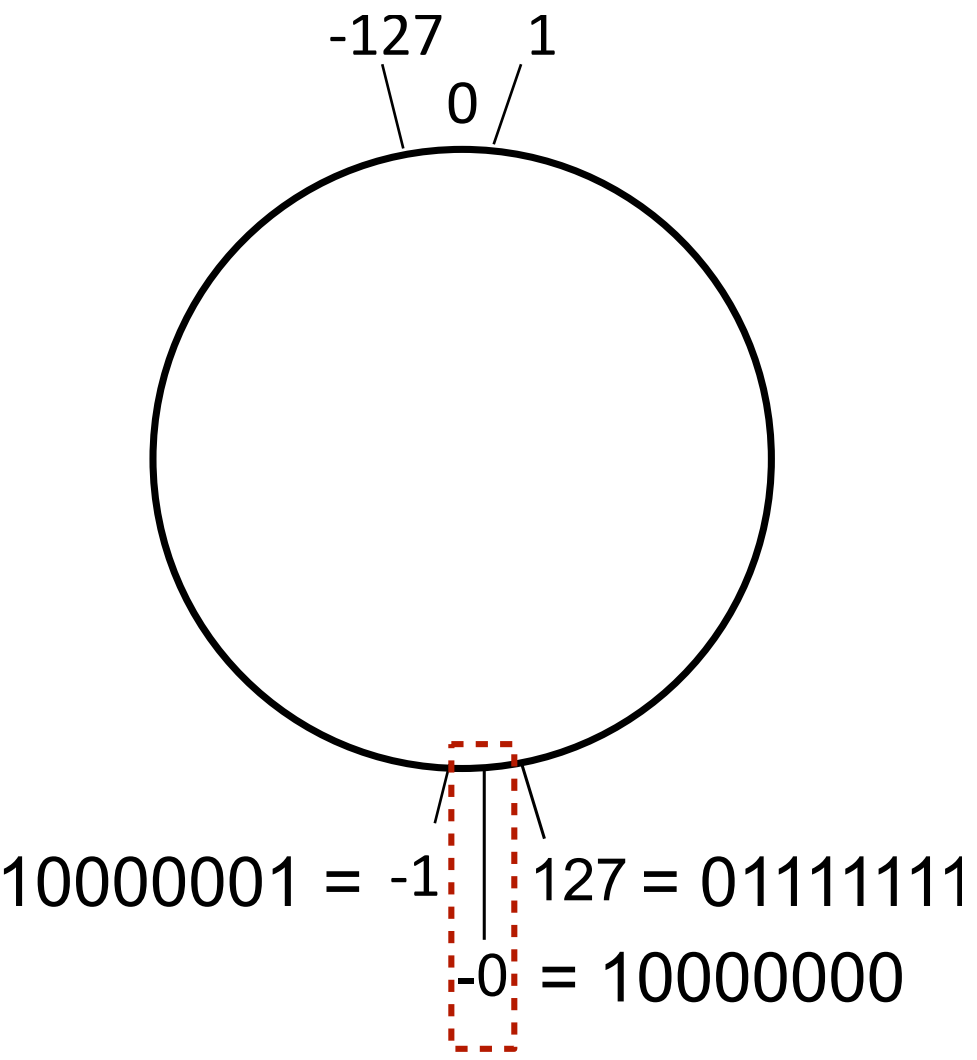
Signed magnitude approach

- With 8 bits:
  - What is the largest number we can represent?
  - What is the smallest number?

Signed magnitude approach

- With 8 bits:
  - What is the largest number we can represent?
    - 01111111 == 127
  - What is the smallest number?
    - 11111111 == -127


255 (11111111)
0
Addition
192
64
128
(10000000)


-127
1
0
10000001 = -1
127 = 01111111
-0 = 10000000

Pros and cons

- Pros
  - Simple to understand
  - Simple to flip sign
- Cons
  - Positive and negative zero!?
  - Grade school arithmetic is broken

(Assuming 4 bit integers)

$$
\begin{array}{rll}
 & 0101 & = \ 5_{10} \\
+ & \underline{1010} & = -2_{10} \\
 & ???? &
\end{array}
$$

(Assuming 4 bit integers)

$$
\begin{array}{rll}
 & 0101 = & 5_{10} \\
+ & \underline{1010} = & -2_{10} \\
 & 1111 = & ?
\end{array}
$$

(Assuming 4 bit integers)

$$
\begin{array}{rll}
 & 0101 & = \ \ 5_{10} \\
+ & \underline{1010} & = -2_{10} \\
 & 1111 & = -7_{10}
\end{array}
$$

(Assuming 4 bit integers)

$$
\begin{array}{rll}
 & 0010 = & 2_{10} \\
+ & \underline{1010} = & -2_{10} \\
 & ???? &
\end{array}
$$

(Assuming 4 bit integers)

```
  1
 0010 =  2₁₀
+1010 = -2₁₀
 1100 =  ?
```

(Assuming 4 bit integers)

```
  1
 0010 =  2₁₀
+1010 = -2₁₀
 1100 = -4₁₀
```

Let’s try another approach

- Modern computers do not use signed magnitude
- Let’s think about another way…
- We want the desirable property that the negation of the negation is the original number
  - e.g., -(-2) == 2
- Signed magnitude satisfies this criterion by flipping one bit
- Why not negate by flipping all the bits?

Let’s try…

- +2 == 0010
- -2 == 1101

$$
\begin{array}{rl}
 & 0101 = 5_{10} \\
+ & \underline{1101} = -2_{10} \\
 & ????
\end{array}
$$

Off by one!

- +2 == 0010
- -2 == 1101

```
 11 1
  0101 =  5₁₀
+ 1101 = -2₁₀
  0010 =  2₁₀
```

Two’s complement

- This implementation is close but off by one
- Let’s start again from first principles
- For an 8-bit int *x*, does adding $2^8 = 256$ change its value? $x + 256 = ?$

Two’s complement

- This implementation is close but off by one
- Let’s start again from first principles
- For an 8-bit int *x*, does adding $2^8 = 256$ change its value? $x + 256 = x$

$$\begin{aligned} & \texttt{xxxxxxxx} = x_{10} \\ + \; & \texttt{1}\underline{\texttt{00000000}} = 256_{10} \\ & \texttt{1xxxxxxxx} = x_{10} \end{aligned}$$

Two’s complement

- Anything + 256 = That same thing
- $-x + 256 = -x$
  $\Rightarrow -x = 256 - x$
- For an n-bit number x:
- $-x = 2^n - x$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2 (-x = 2^n - x)
  == ?

$$
\begin{array}{rl}
 & 0101 = 5_{10} \\
+ & \underline{1110} = -2_{10} \\
 & ????
\end{array}
$$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == ???? (binary)

0101 = 5_{10}

\+ 1110 = -2_{10}

????

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

0101 = 5_{10}

\+ 1110 = -2_{10}

????

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

$$
\begin{array}{rl}
 & 0101 = 5_{10} \\
+ & \underline{1110} = -2_{10} \\
 & \quad 1
\end{array}
$$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

$$
\begin{array}{rl}
 & 0101 = 5_{10} \\
+ & \underline{1110} = -2_{10} \\
 & \quad 11
\end{array}
$$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
  1
  0101 =  5₁₀
+ 1110 = -2₁₀
   011
```

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
 11
  0101 =  5₁₀
+ 1110 = -2₁₀
  0011 =  ?
```

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
 11
  0101 =  5₁₀
+ 1110 = -2₁₀
  0011 =  3₁₀
```

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

$$
\begin{array}{rl}
 & 0010 = \ \ 2_{10} \\
+ & \underline{1110} = -2_{10} \\
 & ????
\end{array}
$$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

$$
\begin{array}{rl}
 & 0010 = \ \ 2_{10} \\
+ & \underline{1110} = -2_{10} \\
 & \ \ \ 0
\end{array}
$$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
  1
 0010 =  2₁₀
+1110 = -2₁₀
   00
```

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
  11
  0010 =  2₁₀
+ 1110 = -2₁₀
   000
```

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

```
 111
  0010 =  2₁₀
+ 1110 = -2₁₀
  0000 =  0₁₀
```

Trick for negating in binary

- $-x = 256 - x$ (for an 8-bit number)
- $-x = 2^n - x$ (for an n-bit number)
- What is $x + \sim x$?

```
  1100111101011101
+ 0011000010100010
  ????????????????
```

Trick for negating in binary

- $-x = 256 - x$ (for an 8-bit number)
- $-x = 2^n - x$ (for an n-bit number)
- What is $x + \sim x$?

```
  1100111101011101
+ 0011000010100010
  1111111111111111
```

Trick for negating in binary

- $-x = 256 - x$ (for an 8-bit number)
- $-x = 2^n - x$ (for an n-bit number)
- What is $x + \sim x$?

```
  1100111101011101
+ 0011000010100010
  1111111111111111 = 2^n-1
```

Trick for negating in binary

- $-x = 256 - x$ (for an 8-bit number)
- $-x = 2^n - x$ (for an n-bit number)
- $x + \sim x = 2^n - 1$
- $\Rightarrow x - 2^n = -\sim x - 1$
- $\Rightarrow -x + 2^n = \sim x + 1$
- $\Rightarrow -x = \sim x + 1$

Let’s try…

- +2 == 0010 (binary)
- -2 == 2^4-2
  == 14
  == 1110 (binary)

~0010 + 1

== 1101 + 1

== 1110

Why flipping all bits didn’t work

- $-x = \sim x + 1$
- This is why simply flipping all the bits was off by one
- We need to also add 1!

Two’s complement

- Only one zero:

  -0000 = ?

Two’s complement

- Only one zero:

  -0000 = ~0000 + 1 = ?

Two’s complement

- Only one zero:

  -0000 = ~0000 + 1 = 1111 + 1 = ?

Two’s complement

- Only one zero:

  -0000 = ~0000 + 1 = 1111 + 1 = 0

Two’s complement

- Only one zero:

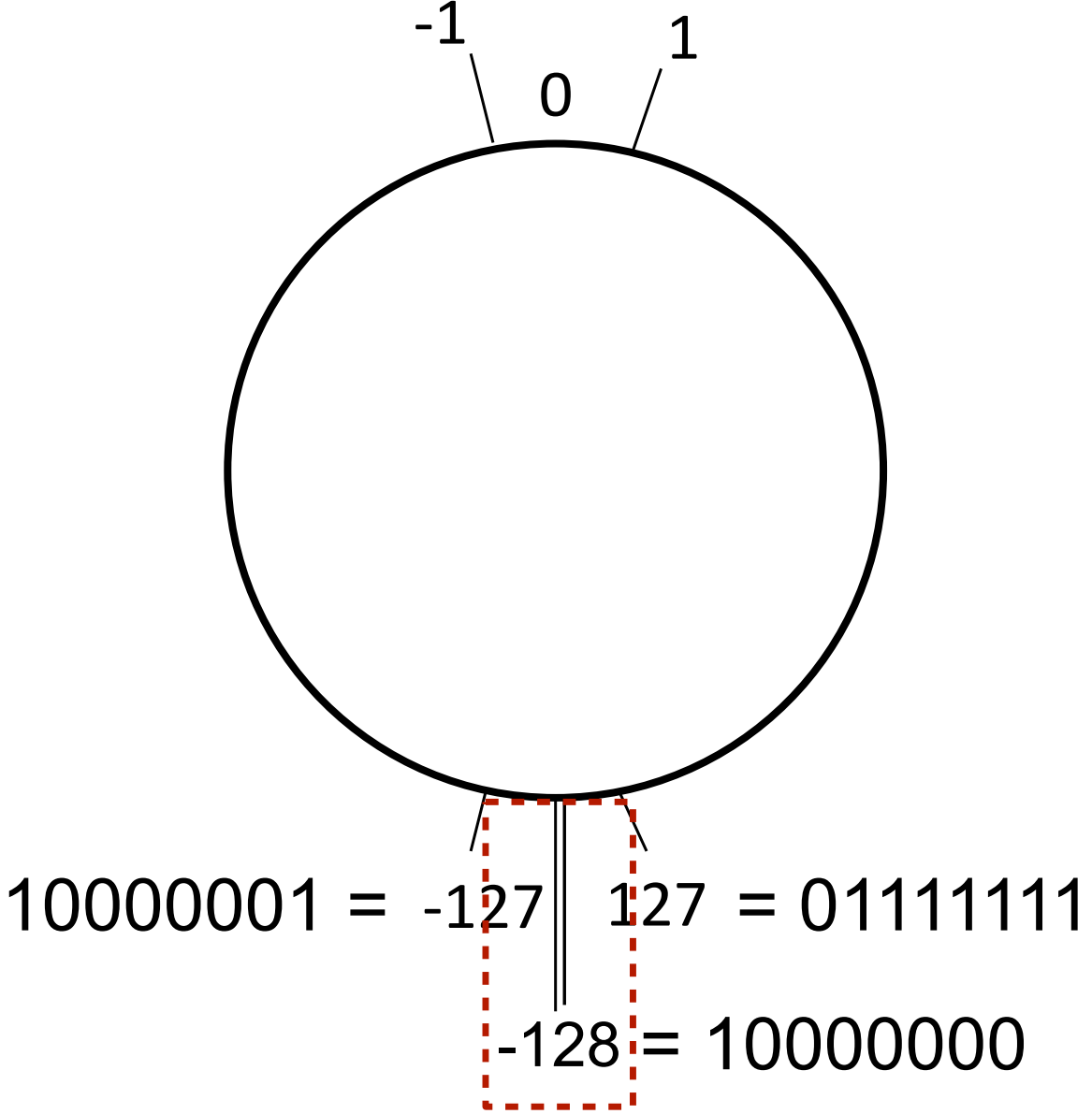
  -0000 = ~0000 + 1 = 1111 + 1 = 0

- Why is it called *two*’s complement?

  $-x = 2^n - x$

- Can quickly identify if a number is negative:

  if and only if the high bit is set


-1
0
1
10000001 = -127
127 = 01111111
-128 = 10000000

TODO

- Assignment 0 is assigned (due September 11)
- Assignment 1 will be assigned end of next week