CS 31: Intro to Systems
Binary Arithmetic

Vasanta Chaganti & Kevin Webb

Swarthmore College

September 14, 2023

Reading Quiz

So far: Unsigned Integers

- With N bits, we can represent values: 0 to 2^n-1

- We can always add 0's to the front of a number without changing it:

10110 = <u>0</u>10110 = <u>000</u>10110 = <u>00000</u>10110

- `1 byte:  char, unsigned char`
- `2 bytes: short, unsigned short`
- `4 bytes: int, unsigned int, float`
- `8 bytes: long long, unsigned long long, double`
- `4 or 8 bytes: long, unsigned long`

Unsigned Integers

- Suppose we had one byte
  - Can represent 2^8 (256) values
  - If unsigned (strictly non-negative): 0 – 255

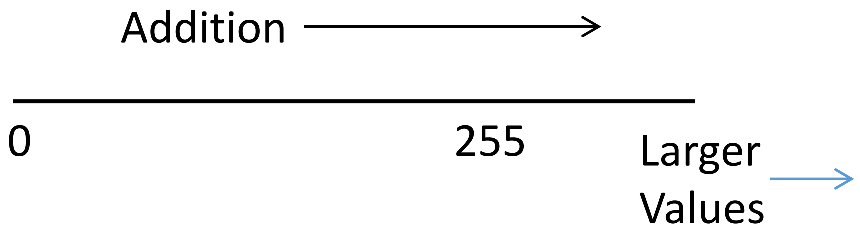
252 = 11111100

253 = 11111101

254 = 11111110

255 = 11111111

Traditional number line:



Unsigned Integers

- Suppose we had one byte
  - Can represent 2^8 (256) values
  - If unsigned (strictly non-negative): 0 – 255

252 = 11111100

253 = 11111101

254 = 11111110

255 = 11111111

What if we add one more?

Car odometer “rolls over”.

Any time we are dealing with a finite storage space we cannot represent an infinite number of values!

Unsigned Integers

- Suppose we had one byte
  - Can represent 2^8 (256) values
  - If unsigned (strictly non-negative): 0 – 255

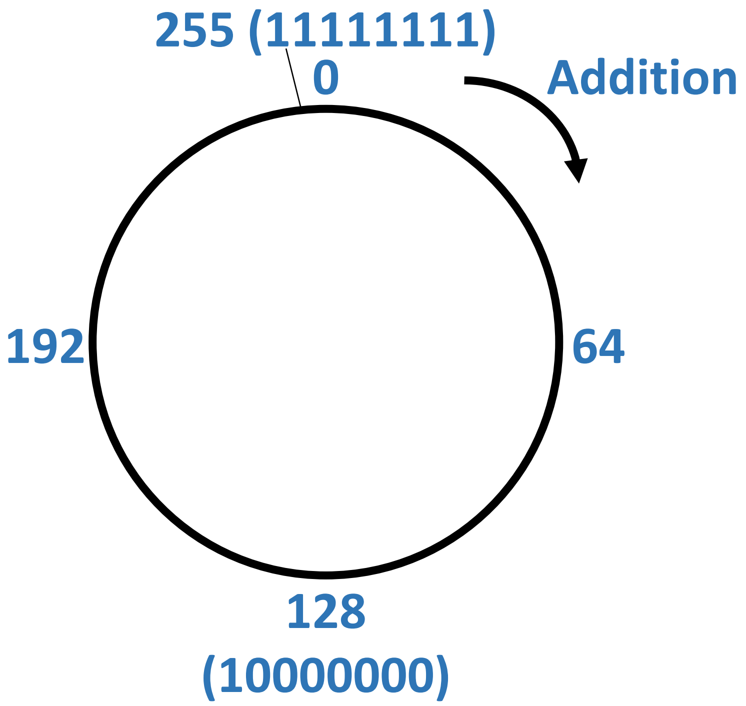
252 = 11111100

253 = 11111101

254 = 11111110

255 = 11111111

What if we add one more?



Modular arithmetic: Here, all values are modulo 256.

Unsigned Addition (4-bit)

- Addition works like grade school addition:

```
    1
    0110      6
+   0100    + 4
--------    ---
    1010     10
```

Four bits give us range: 0 - 15

Unsigned Addition (4-bit)

- Addition works like grade school addition:

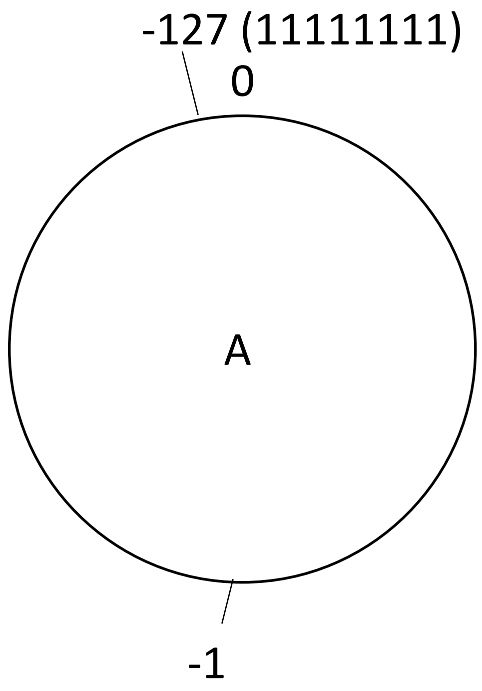
```
   1
   0110      6          1100     12
+  0100   +  4       +  1010    +10
-------   ----       -------    ---
   1010     10       1 0110       6
                     ^carry out
```

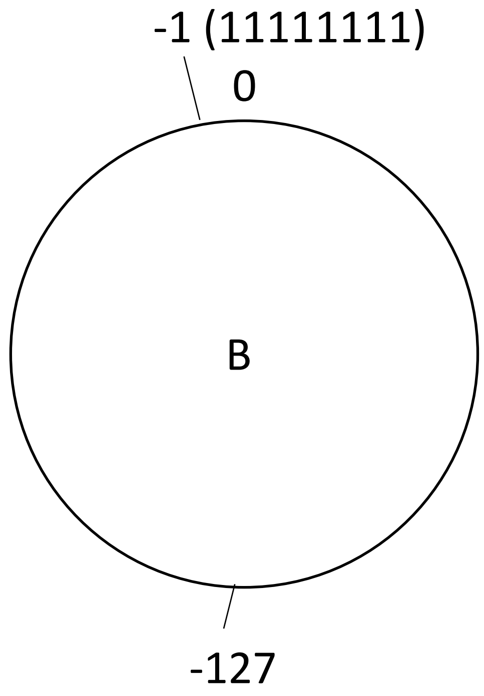
Overflow!

Four bits give us range: 0 - 15

Carry out is indicative of something having gone wrong when adding unsigned values

Suppose we want to support signed values too (positive **and** negative). Where should we put -1 and -127 on the circle? Why?





C: Put them somewhere else.

Not Used: Signed Magnitude

- One bit (usually left-most) signals:
  - 0 for positive
  - 1 for negative

For one byte:

1 = 00000001

-1 = 10000001

Pros: Negation is very simple!

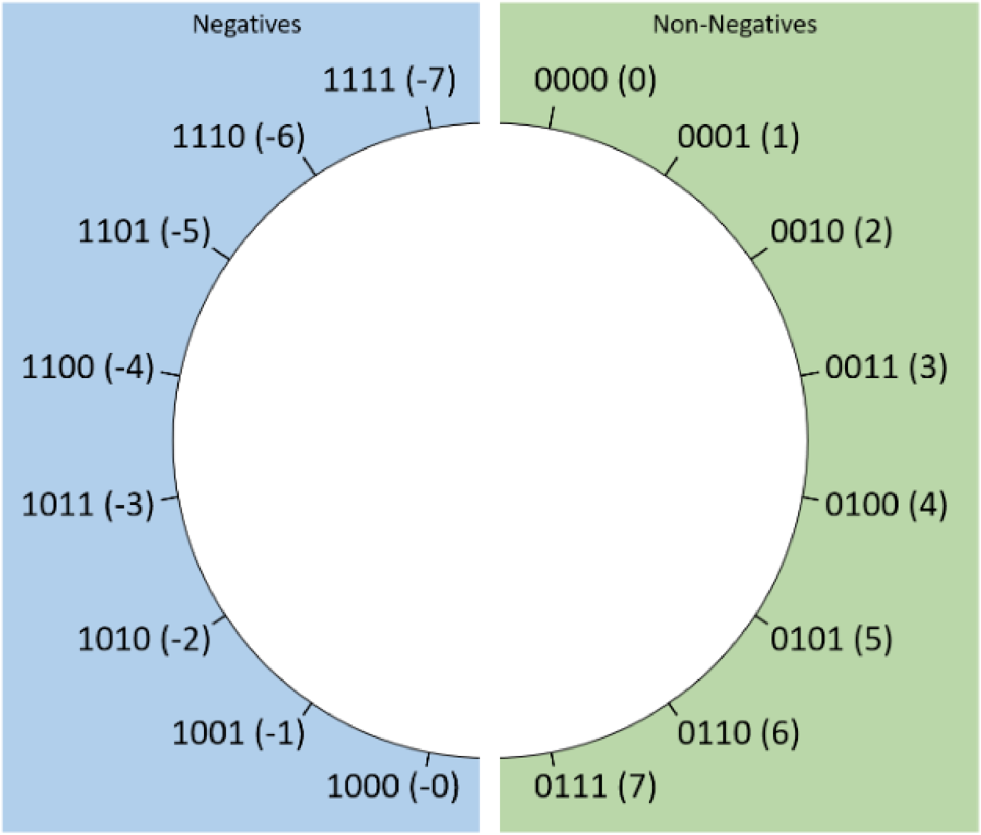


Figure 1. A logical layout of signed magnitude values for bit sequences of length four.

Not Used: Signed Magnitude

- One bit (usually left-most) signals:
  - 0 for positive
  - 1 for negative

For one byte:

0 = 00000000

-0? = 10000000

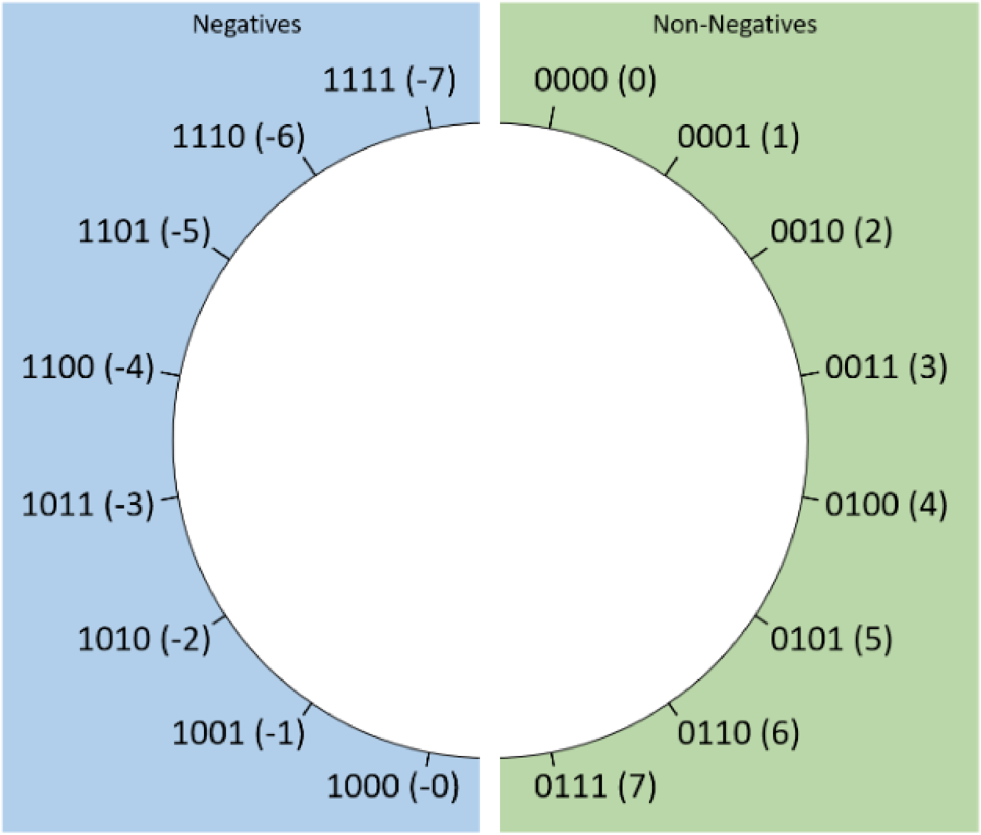
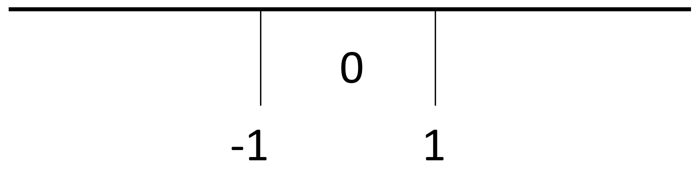


Figure 1. A logical layout of signed magnitude values for bit sequences of length four.

Major con: Two ways to represent zero.

Used Today: Two's Complement

- Borrow nice property from number line:



Only one instance of zero!
Implies: -1 and 1 on either side of it.

- For an 8 bit range we can express 256 unique values:
  - 128 non-negative values (0 to 127)
  - 128 negative values (-1 to -128)

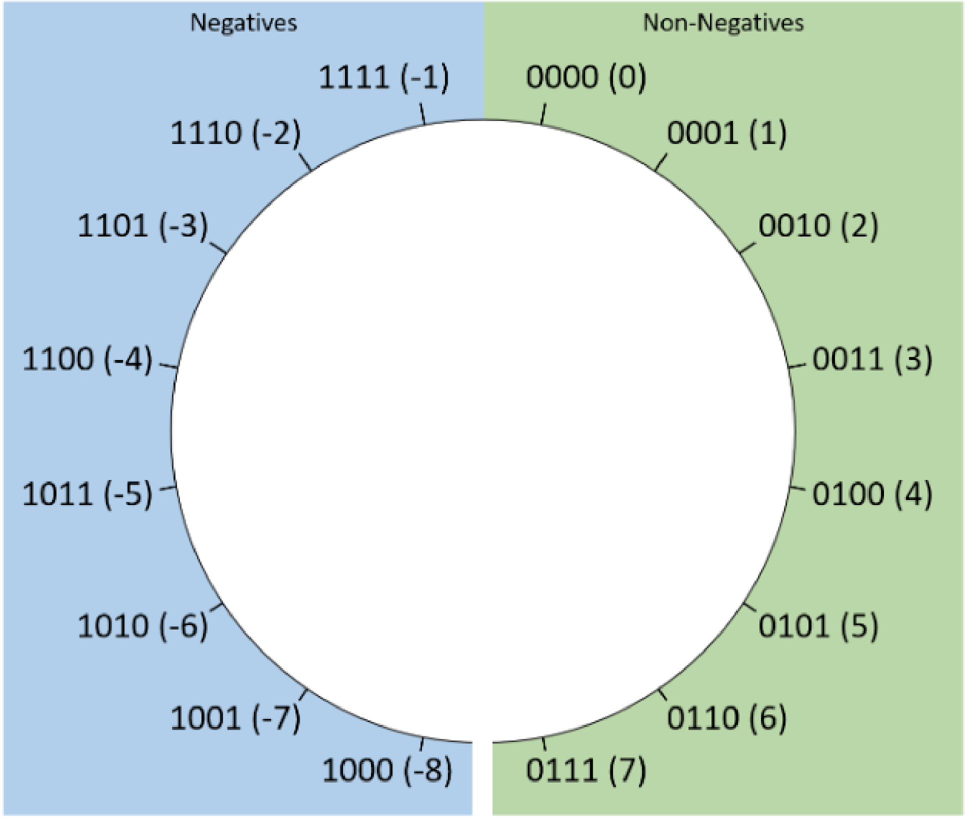


Figure 2. A logical layout of two's complement values for bit sequences of length four.

Two's Complement

- Only one value for zero

- With N bits, can represent the range:
  - -2^{N-1} to $2^{N-1} - 1$

- <u>Most significant</u> (first) bit still designates positive (0) /negative (1)

- Negating a value is slightly more complicated:

  1 = 00000001, -1 = 11111111

From now on, unless we explicitly say otherwise, we'll assume all integers are stored using two's complement! This is the standard!

Two's Compliment

- Each two's compliment number is now:

$$[-2^{n-1} * d_{n-1}] \quad + \quad [2^{n-2} * d_{n-2}] \quad + \ldots + \quad [2^{1} * d_{1}] \quad + \quad [2^{0} * d_{0}]$$

Note the <u>negative sign</u> on just the first digit.
This is why first digit tells us negative vs. positive.

(The other digits are unchanged and carry the same meaning as unsigned.)

If we interpret 11001 as a two's complement number, what is the value in decimal?

- Each two's compliment number is now:

$$[-2^{n-1} * d_{n-1}] \quad + \quad [2^{n-2} * d_{n-2}] \quad + \ldots + \quad [2^1 * d_1] \quad + \quad [2^0 * d_0]$$

A. -2

B. -7

C. -9

D. -25

“If we interpret…”

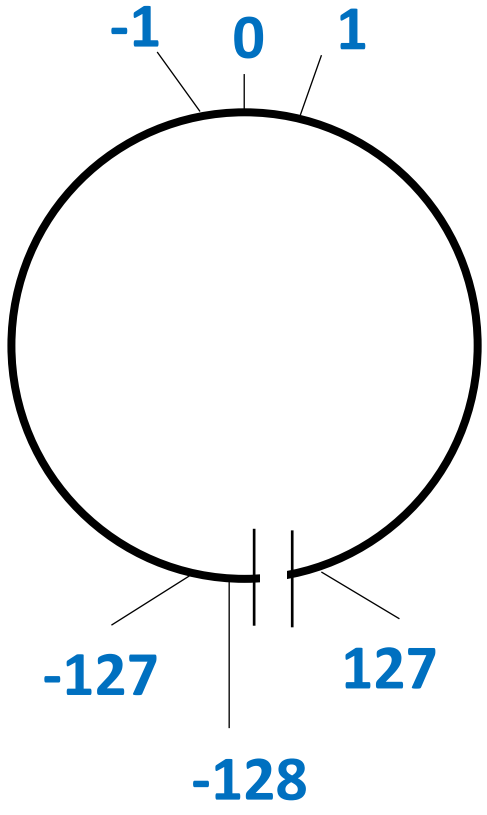
- What is the decimal value of 1100?

- …as unsigned, 4-bit value: 12 (%u)
- …as signed (two’s complement), 4-bit value: -4 (%d)

- …as an 8-bit value: 12
  (i.e., **0000**1100)

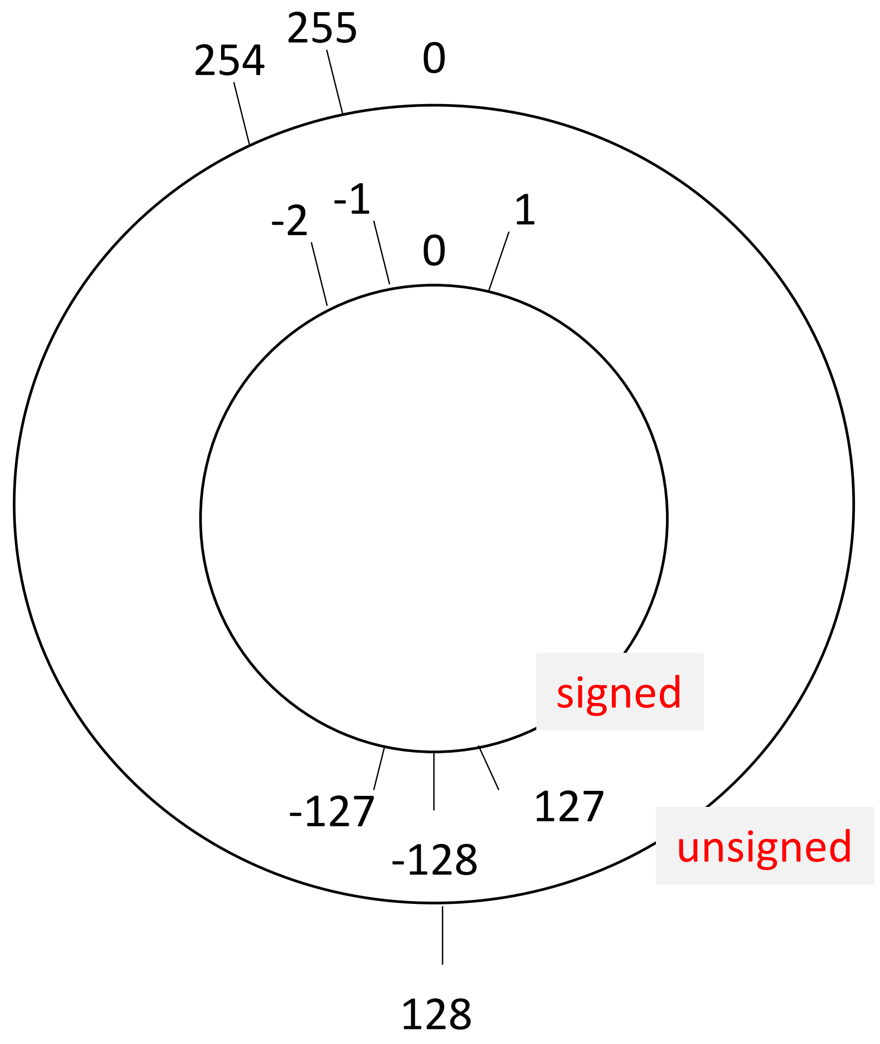
Two's Complement Negation

- To negate a value x, we want to find y such that $x + y = 0$.
- For N bits, $y = 2^N - x$



Negation Example (8 bits)

- For N bits, $y = 2^N - x$
- Negate 00000010 (2)
  - $2^8 - 2 = 256 - 2 = 254$
- Our wheel only goes to 127!
  - Put -2 where 254 would be if wheel was unsigned.
  - 254 in binary is 11111110



Given 11111110, it's 254 if interpreted as <u>unsigned</u> and -2 interpreted as <u>signed</u>.

Negation Shortcut

- A much easier, faster way to negate:
  - Flip the bits (0's become 1's, 1's become 0's)
  - Add 1

- Negate 00101110 (46)
  - 2^8 - 46 = 256 - 46 = 210
  - 210 in binary is 11010010

46: 00101110

Flip the bits: 11010001

Add 1

+ 1

-46: 11010010

Decimal to Two's Complement with 8-bit values

(high-order bit is the sign bit)

For positive values, use same algorithm as unsigned

```
For example, 6:       6 - 4 = 2 (4:2^2)
                      2 - 2 = 0 (2:2^1):  00000110
```

For negative values:

1. convert the equivalent positive value to binary
2. then negate binary to get the negative representation

```
For example, -3:
                      3: 00000011
                 negate: 11111100+1 = 11111101 = -3
```

What is the 8-bit, two’s complement representation for -7?

For negative values:

1. convert the equivalent positive value to binary
2. then negate binary to get the negative representation

A. 11111001
B. 00000111
C. 11111000
D. 11110011

Addition & Subtraction

- Addition is the same as for unsigned
  - One exception: different rules for overflow
  - Can use the same hardware for both

- Subtraction is the same operation as addition
  - Just need to negate the second operand…

- $6 - 7 = 6 + (-7) = 6 + (\sim 7 + 1)$
  - ~7 is shorthand for "flip the bits of 7"

Subtraction Hardware

<u>Negate and add 1 to second operand:</u>

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1

input 1 -------------------------------->

input 2 --> $\boxed{\text{possible bit flipper}}$ --> ADD CIRCUIT ---> result

possible +1 input-------->

Let's call this possible +1 input: "Carry in"

(0: on add, 1: on subtract)

4-Bit Subtraction Example

Subtraction via addition: a - b is same as a + ~b + 1

Subtraction: flip bits and add 1

```
3 -  6 =    0011
            1001      (6: 0110  ~6: 1001)
         +     1
            ----
            1101 = -3
```

Equivalent addition: don't flip bits or add 1

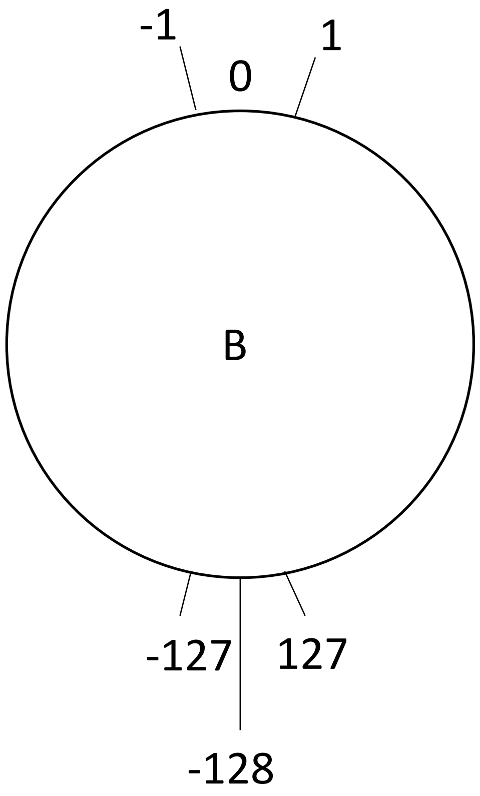
```
3 + -6 =    0011
         +  1010
            ----
            1101 = -3
```

By switching to two's complement, have we solved this value "rolling over" (overflow) problem?

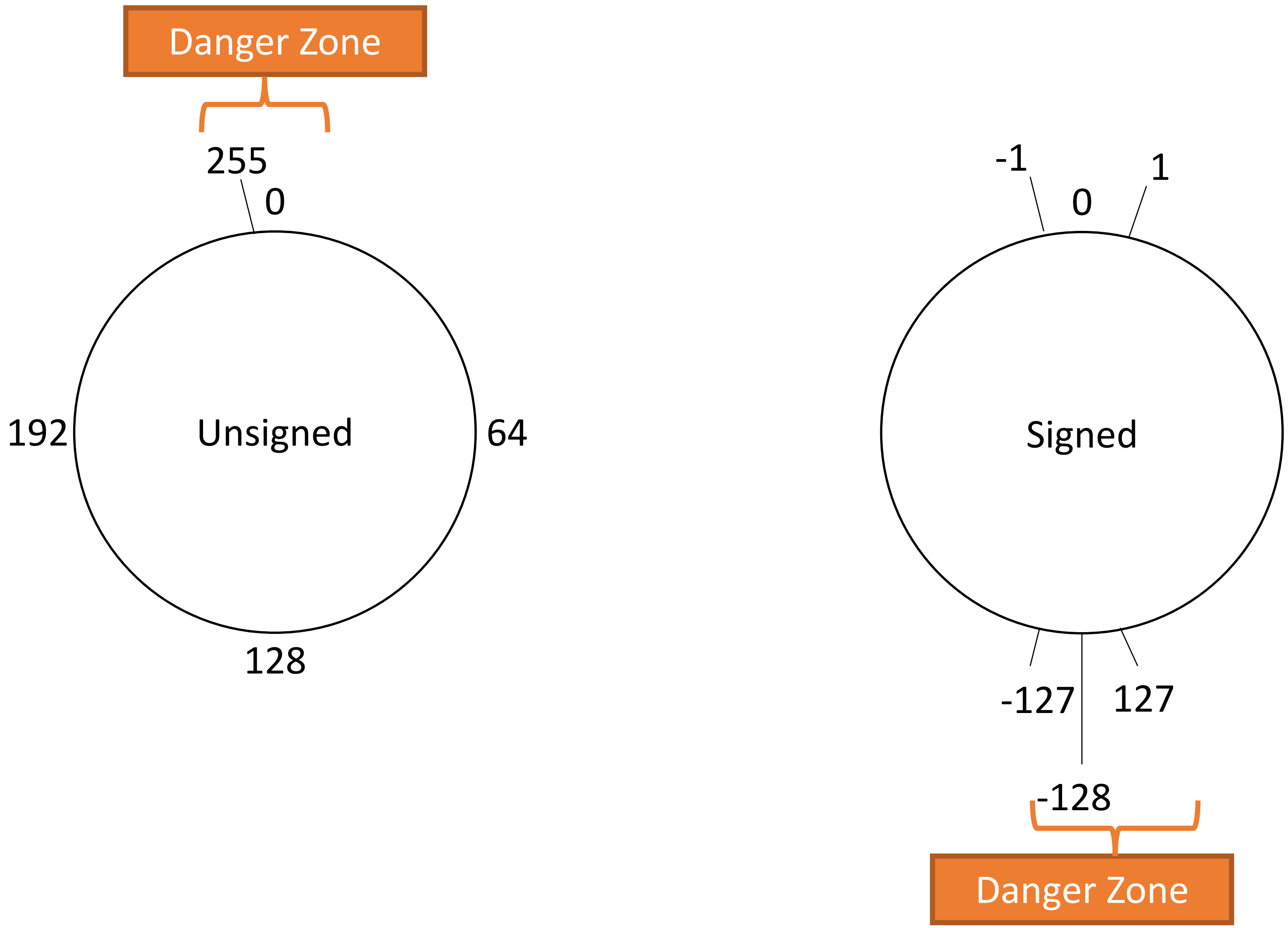
A. Yes, it's gone.

B. Nope, it's still there.

C. It's even worse now.



Overflow, Revisited



If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never

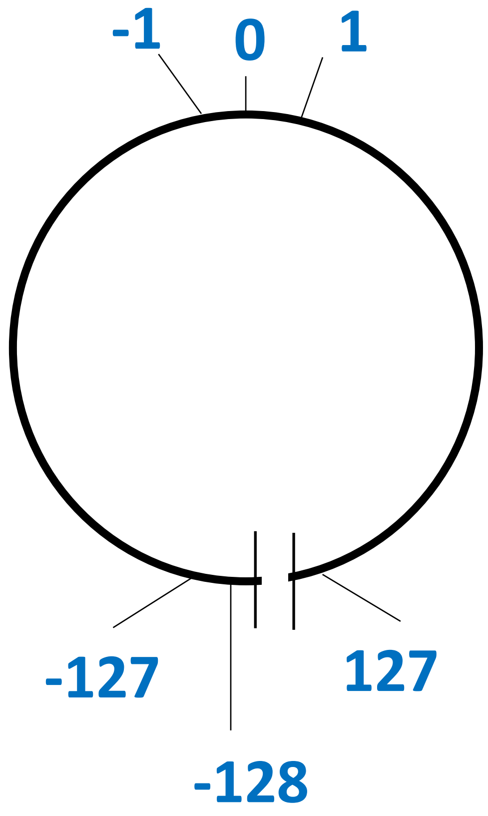
Two's Complement Overflow For Addition

- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

sign of operands = sign of result

no overflow

3+4=7	-2+-3=-5
0011	1110
+0100	+1101
0111	1 1011



Two's Complement Overflow For Addition

- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

sign of operands = sign of result

no overflow

3+4=7	-2+-3=-5
0011	1110
+0100	+1101
0111	1 1011

sign of operands ≠ sign of result

overflow

4+7=11	-6-8=-14
0100	1010
+0111	+1000
1011	1 0010
(-5)	(2)

Two's Complement Overflow For Subtraction

- **Rule 1:**

  Minuend Subtrahend Result

  - Positive operand - Negative operand = Positive Result: No Overflow
  - Positive operand - Negative operand = Negative Result: Overflow
  - **Intuition:** We know a positive – negative is equivalent to a positive + positive. If this sum does not result in a positive value we have an overflow

- **Rule 2:**

  Minuend Subtrahend Result

  - Negative operand - Positive operand = Negative Result: No Overflow
  - Negative operand - Positive operand = Positive Result: Overflow
  - **Intuition:** We know a negative – positive number is equivalent to a negative + negative number. If this sum does not result in a negative value we have an overflow

Two's Complement Overflow For Subtraction

- **Rule 1:**

  $\overbrace{\text{Positive operand}}^{\text{Minuend}}$ - $\overbrace{\text{Negative operand}}^{\text{Subtrahend}}$ = $\overbrace{\text{Positive Result}}^{\text{Result}}$: No Overflow

  - Positive operand - Negative operand = Positive Result: No Overflow
  - Positive operand - Negative operand = Negative Result: Overflow
  - **Intuition:** We know a positive – negative is equivalent to a positive + positive. If this sum does not result in a positive value we have an overflow

Subtrahend and Result have **different sign bits**

no overflow

```
2-(-3)=5        3-(-4)=7
 0010            0011
-1110           -1100

 0010            0011
+0011           +0100
 0101            0111
```

Subtrahend and Result have the **same sign bits**

overflow

```
2-(-6)=8             3-(-7)=10
 0010                 0011
-1010                -1001

 0010                 0011
+0110                +0111
 1000(-8)             1010(-6)
```

Two’s Complement Overflow For Subtraction

- **Rule 2:**

  $\overbrace{\text{Negative operand}}^{\text{Minuend}} - \overbrace{\text{Positive operand}}^{\text{Subtrahend}} = \overbrace{\text{Negative Result}}^{\text{Result}}$: No Overflow

  - Negative operand - Positive operand = Negative Result: No Overflow
  - Negative operand - Positive operand = Positive Result: Overflow
  - **Intuition:** We know a negative – positive number is equivalent to a negative + negative number. If this sum does not result in a negative value we have an overflow

Subtrahend and Result have **different sign bits**

no overflow

```
-2-(3)=-5          -3-(4)=-7
  1110               1101
 -0011              -0100

  1110               1101
 +1101              +1100
1 1011(-5)         1 1001(-7)
```

Subtrahend and Result have the **same sign bits**

overflow

```
-2-(7)=-9          -4-(7)=-11
  1110               1100
 -0111              -0111

  1110               1100
 +1001              +0111
1 0111(7)          1 0011(-6)
```

Two’s Complement Overflow For Subtraction

Subtraction Overflow Rules Summarized:

- Overflow occurs IFF the sign bits of the subtraction operands are different, and the sign bit of the Result and Subtrahend are the same as shown below:
  - Minuend - Subtrahend = Result
  - If positive – negative = negative (overflow)
  - If negative – positive = positive (overflow)

- Now that we have rules for two’s complement, let’s revisit unsigned numbers and formalize the overflow rules there!

Recall: Subtraction Hardware

<u>Negate and add 1 to second operand:</u>

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1

```
input 1 -------------------------------->
input 2 --> | possible bit flipper | --> | ADD CIRCUIT | ---> result
          possible +1 input-------->
```

Let’s call this possible +1 input: “Carry in”

(0: on add, 1: on subtract)

How many of these unsigned operations have overflowed?

Interpret these as 4-bit unsigned values (valid range 0 to 15):

```
                                            carry-in  carry-out
                                                ↓        ↓
Addition (carry-in = 0)
  9 + 11  =   1001 + 1011 + 0 =   1   0100
  9 +  6  =   1001 + 0110 + 0 =   0   1111
  3 +  6  =   0011 + 0110 + 0 =   0   1001
```

Subtraction (carry-in = 1)

$$6 - 3 = 0110 + \overbrace{1100 + 1}^{(-3)} = 1\ \ 0011$$

$$3 - 6 = 0011 + \underbrace{1001 + 1}_{(-6)} = 0\ \ 1101$$

A. 1

B. 2

C. 3

D. 4

E. 5

How many of these unsigned operations have overflowed?

Interpret these as 4-bit unsigned values (valid range 0 to 15):

Addition (carry-in = 0)

					carry-in		carry-out		
9 + 11	=	1001	+	1011	+ 0	=	1	0100	= 4
9 + 6	=	1001	+	0110	+ 0	=	0	1111	= 15
3 + 6	=	0011	+	0110	+ 0	=	0	1001	= 9

Subtraction (carry-in = 1)

$$6 - 3 = 0110 + \overbrace{1100 + 1}^{(-3)} = 1 \quad 0011 = 3$$

$$3 - 6 = 0011 + \underbrace{1001 + 1}_{(-6)} = 0 \quad 1101 = 13$$

A. 1
B. 2
C. 3
D. 4
E. 5

Pattern?

Overflow Rule Summary

- Signed overflow:
  - The sign bits of operands are the same, but the sign bit of result is different.

- Unsigned: overflow
  - The carry-in bit is different from the carry-out.

C_{in}	C_{out}	C_{in} XOR C_{out}
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

So far, all arithmetic on values that were the same size. What if they're different?

Sign Extension

- When combining signed values of different sizes, expand the smaller value to equivalent larger size:

```
char y = 2, x = -13;
short z = 10;
```

```
z = z + y;

0000000000001010
+        00000010
0000000000000010
```

```
z = z + x;

0000000000000101
+        11110011
1111111111110011
```

Fill in high-order bits with sign-bit value to get same numeric value in larger number of bytes.

Let’s verify that this works

4-bit signed value, sign extend to 8-bits, is it the same value?

`0111  --->  0000 0111` obviously still 7

`1010  --->  1111 1010` is this still -6?

-128 + 64 + 32 + 16 + 8 + 0 + 2 + 0 = -6 yes!

Operations on Bits

- For these, it doesn't matter how the bits are interpreted (signed vs. unsigned)
- Bit-wise operators (AND, OR, NOT, XOR)
- Bit shifting

Bit-wise Operators

- Bit operands, Bit result (interpret as appropriate for the context)

& (AND) | (OR)   ~(NOT)   ^(XOR)

A	B	A & B	A | B	~A	A ^ B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

```
  01101010      01010101      10101010    ~10101111
& 10111011    | 00100001    ^ 01101001     01010000
----------    ----------    ----------
  00101010      01110101      11000011
```

More Operations on Bits (Shifting)

- Bit-shift operators: << left shift, >> right shift

```
01010101 << 2  is 01010100
                  2 high-order bits shifted out
                  2 low-order bits filled with 0
01101010 << 4  is 10100000
01010101 >> 2  is 00010101
01101010 >> 4  is 00000110

10101100 >> 2  is 00101011 (logical shift)
               or 11101011 (arithmetic shift)
```

Arithmetic right shift: fills high-order bits w/sign bit

C automatically decides which to use based on type: signed: arithmetic, unsigned: logical

Try some 4-bit examples:

bit-wise operations:

- 0101 & 1101
- 0101 | 1101

Logical (unsigned) bit shift:

- 1010 << 2
- 1010 >> 2

Arithmetic (signed) bit shift:

- 1010 << 2
- 1010 >> 2

Up Next

- Circuits
  - How can we build hardware to perform all these operations on bits?