CS 31: Introduction to Computer Systems

03: Binary Arithmetic

January 29

SWARTHMORE COLLEGE

WiCS!

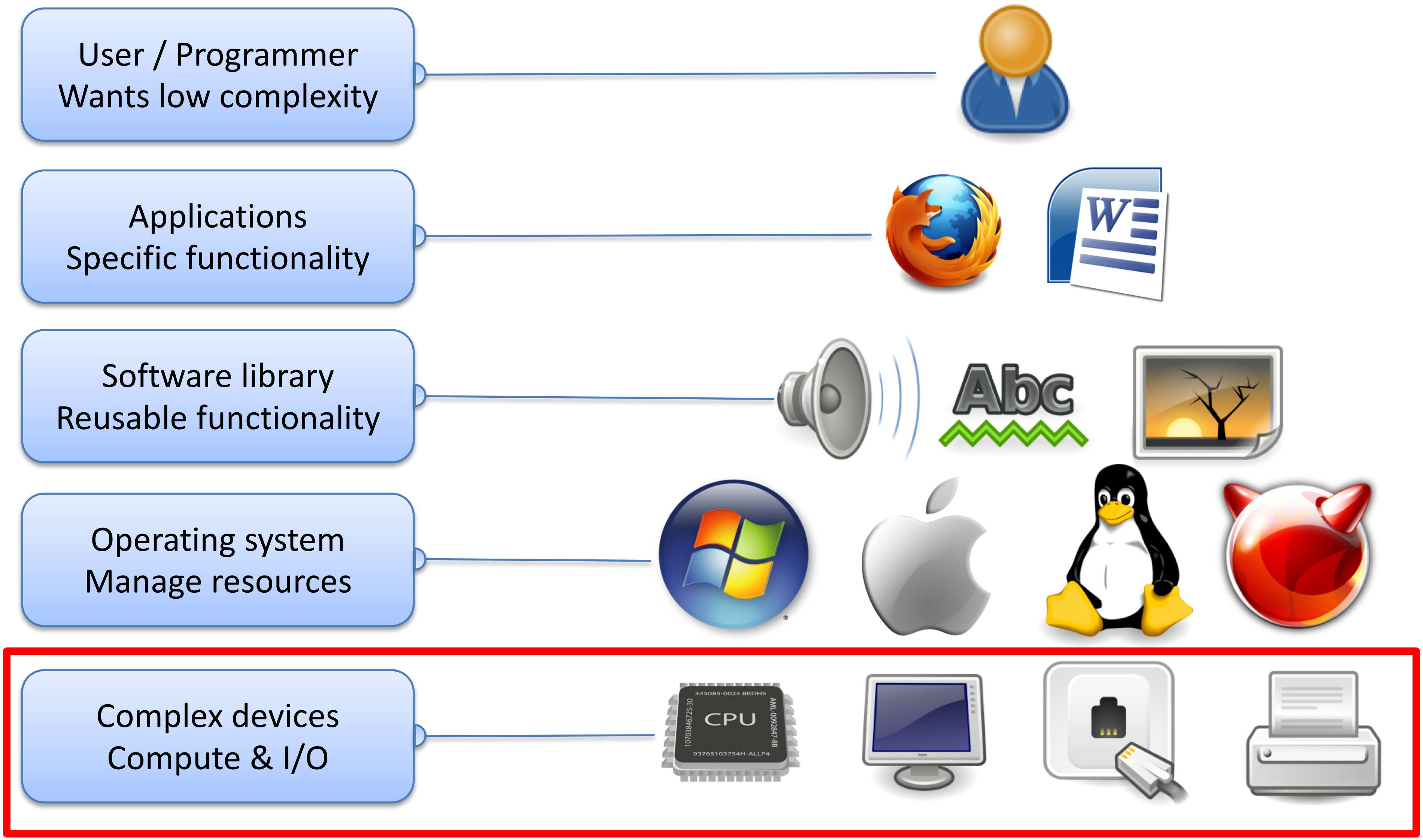
Swarthmore Women in Computer Science

Today

- Binary Arithmetic
  - Unsigned addition
  - Subtraction
- Representation
  - Signed magnitude
  - Two’s complement
  - Signed overflow
- Bit operations

Reading Quiz

Abstraction



User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O

Last Class: Binary Digits: (BITS)

7 6 5 4 3 2 1 0

Most significant bit → $\underline{1}000111\underline{1}$ ← Least significant bit

Representation: $1 \times 2^7 + 0 \times 2^6 \ldots\ldots + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

10001111 = 143

one byte is the smallest addressable unit - contains 8 bits

Last Class: Unsigned Integers

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

C types and their (typical!) sizes

- 1 byte (8 bits = 2^8 unique values):
  - **`char, unsigned char`**
- 2 bytes (16 bits = 2^{16} unique values):
  - **`short, unsigned short`**
- 4 bytes (32 bits = 2^{32} unique values):
  - **`int, unsigned int, float`**
- 8 bytes (64 bits = 2^{64} unique values):
  - **`long long, unsigned long long, double`**
- 4 or 8 bytes: **`long, unsigned long`**

```
unsigned long v1;
short s1;
long long ll;
```

Last Class: Unsigned Integers

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

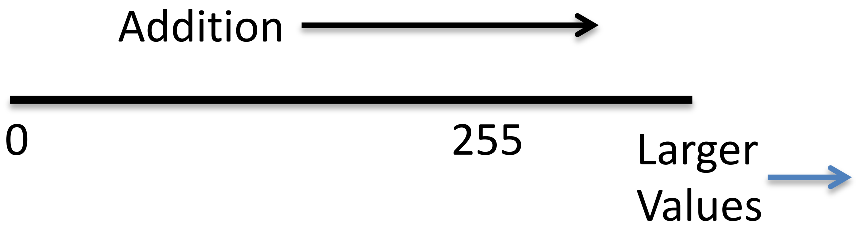
252 = 11111100

253 = 11111101

254 = 11111110

255 = 11111111

Traditional number line:



Last Class: 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!

Last Class: Unsigned Integers

Suppose we had one byte

- Can represent 2^8 (256) values
- If unsigned (strictly non-negative):

0 – 255

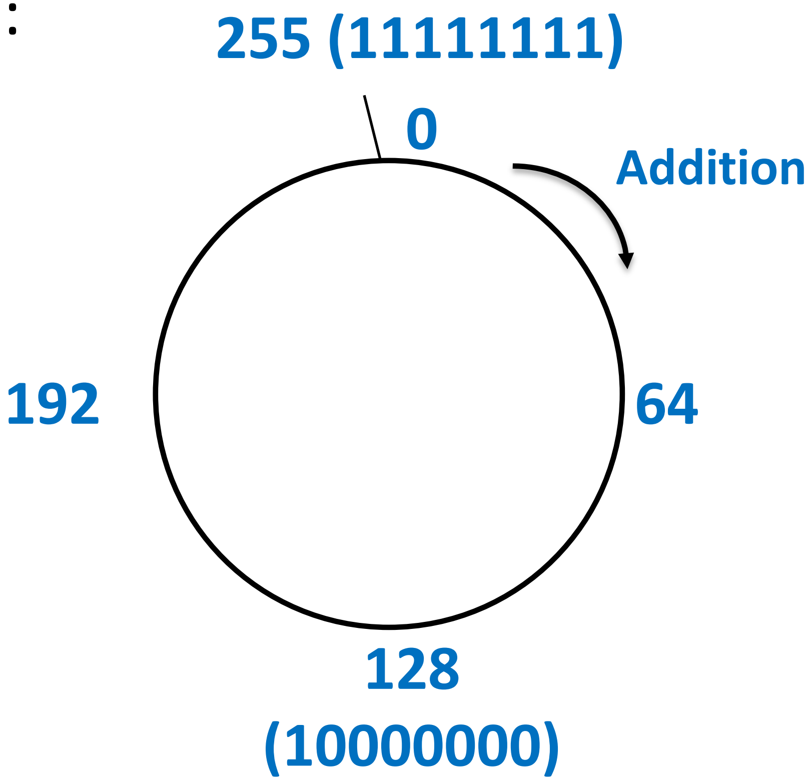
252 = 11111100

253 = 11111101

254 = 11111110

255 = 11111111

<u>What if we add one more?</u>



Modular arithmetic: Here, all values are modulo 256.

Last Class: Unsigned Addition (4-bit)

Addition works like grade school addition:

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

Four bits give us range: 0 - 15

Last Class: Unsigned Addition (4-bit)

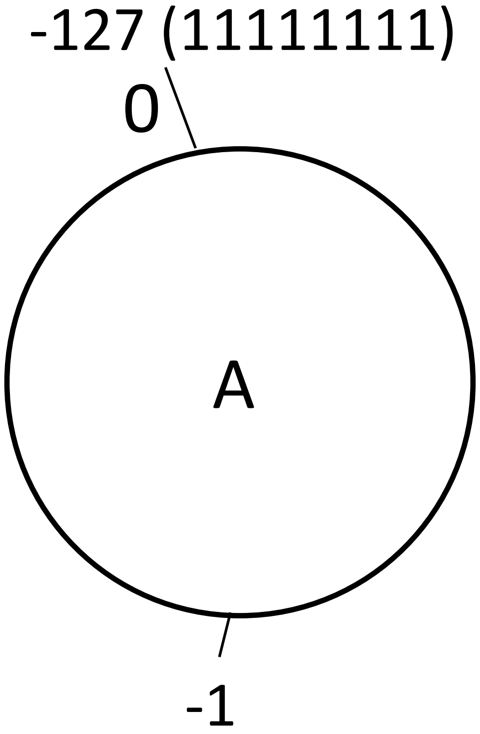
- Addition works like grade school addition:

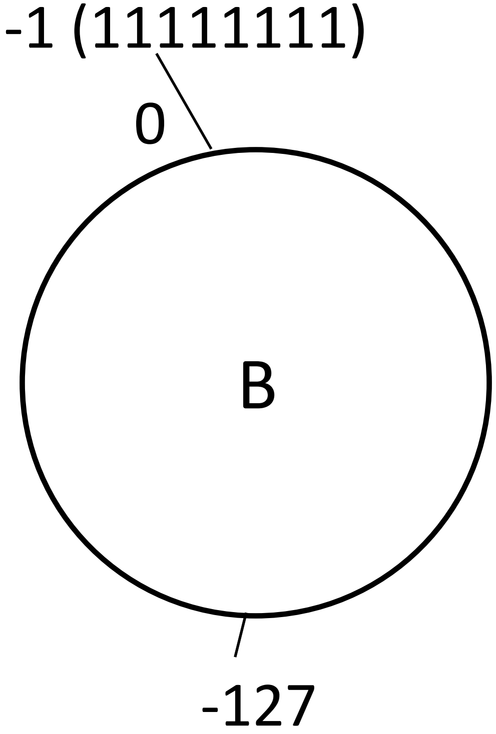
```
    1
    0110      6          1100     12
+  0100    +  4      +  1010     +10
  ------   -----     --------    ---
    1010     10      1  0110       6
                     ^carry out
```

Four bits give us range: 0 - 15

Overflow!

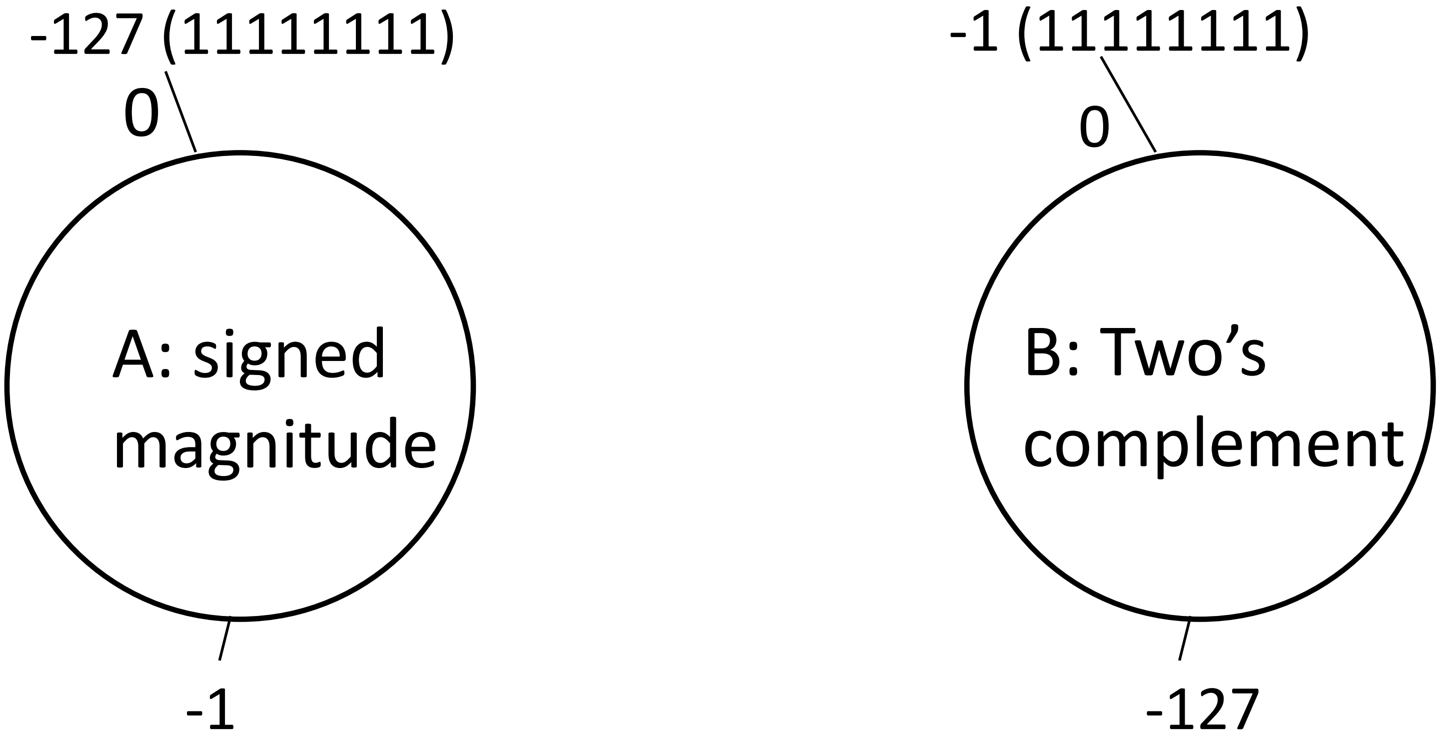
Suppose we want to support signed values (positive and negative) in 8 bits, where should we put -1 and -127 on the circle? Why?





C: Put them somewhere else.

Suppose we want to support signed values (positive and negative) in 8 bits, where should we put -1 and -127 on the circle? Why?



-127 (11111111)

0

A: signed magnitude

-1

-1 (11111111)

0

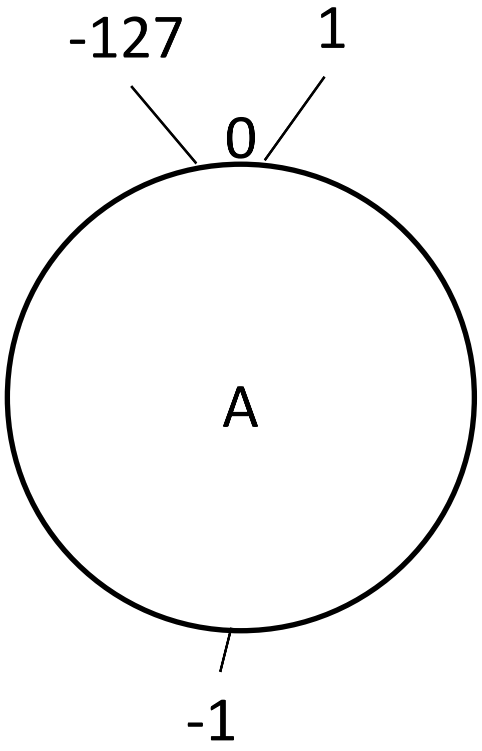
B: Two's complement

-127

C: Put them somewhere else.

Signed Magnitude

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



For one byte:

1 = 00000001,   -1 = 10000001

Pros: Negation (negative value of a number) is very simple!

Signed Magnitude

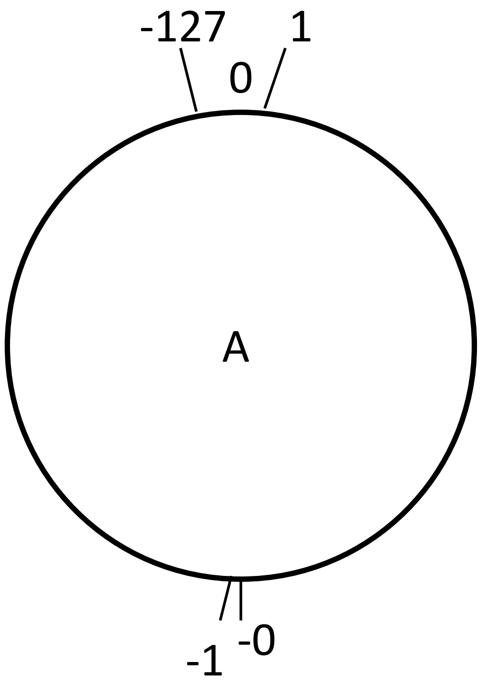
- One bit (usually left-most) signals:
  - 0 for positive
  - 1 for negative

For one byte:

0 = 00000000

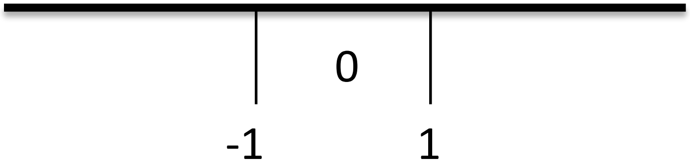
What about 10000000?

Major con: Two ways to represent zero.



Two's Complement (signed)

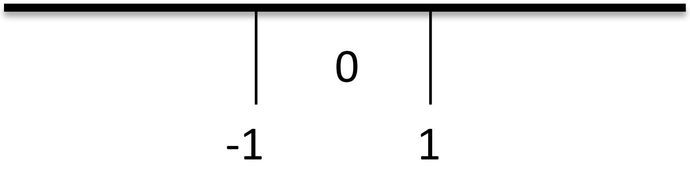
- Borrow nice property from number line:



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

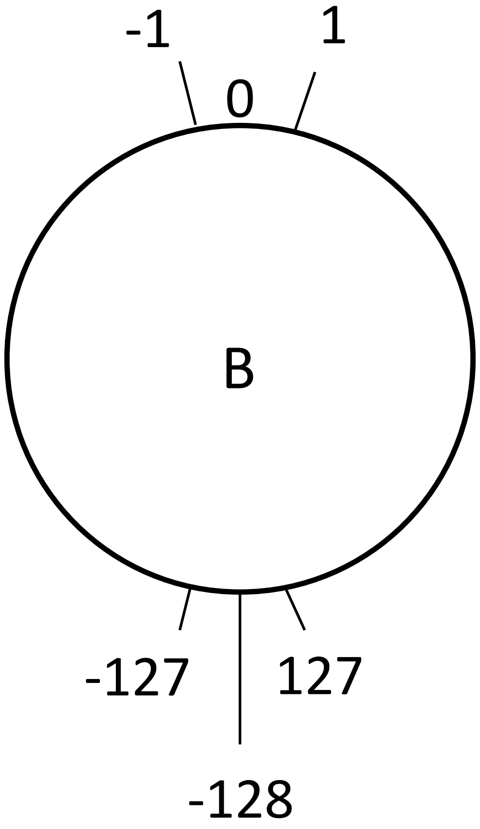
Two's Complement

- 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)



Two's Complement

- Only one value for zero
- With N bits, can represent the range:
  - -2^{N-1} to $2^{N-1} - 1$
- Most significant 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 negative sign on just the most significant bit. This is why first bit tells us whether the value is negative vs. positive.

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 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 $-16 + 8 + 1 = -7$

C. -9

D. -25

“If we interpret…”

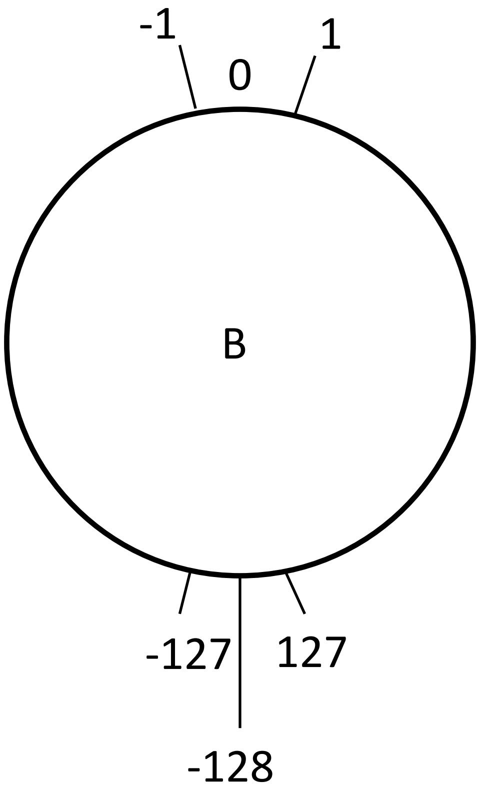
What is the decimal value of 1100?

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

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

Two's Complement Negation

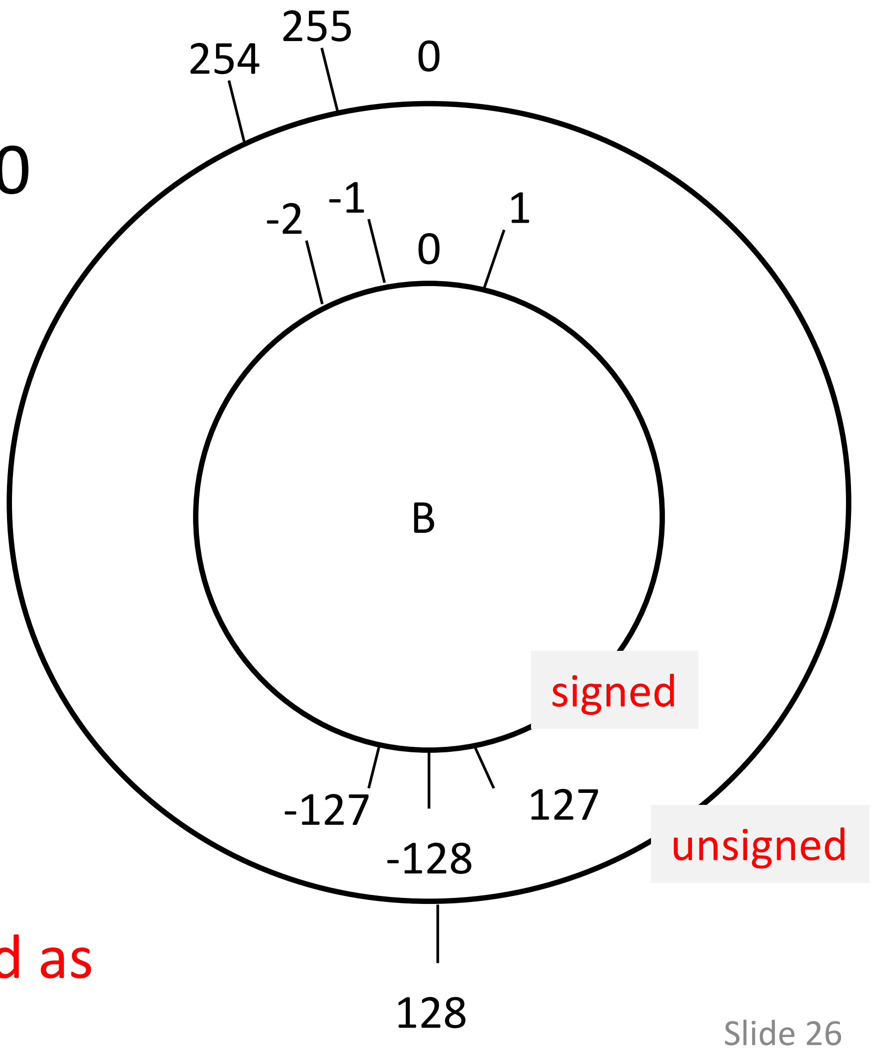
- To negate a value x, we want to find y such that x + y = 0.
- For N bits, $y = 2^N - x$



Negation Example (8 bits)

- For N bits, y = 2^N - x
- Negate the value (2) 00000010
- 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 unsigned and -2 interpreted as signed.

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)

46: 00101110
Flip the bits: 11010001
Add 1 + 1
-46: 11010010

- Formally:
  - $2^8 - 46 = 256 - 46 = 210$
  - 210 in binary is 11010010

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…

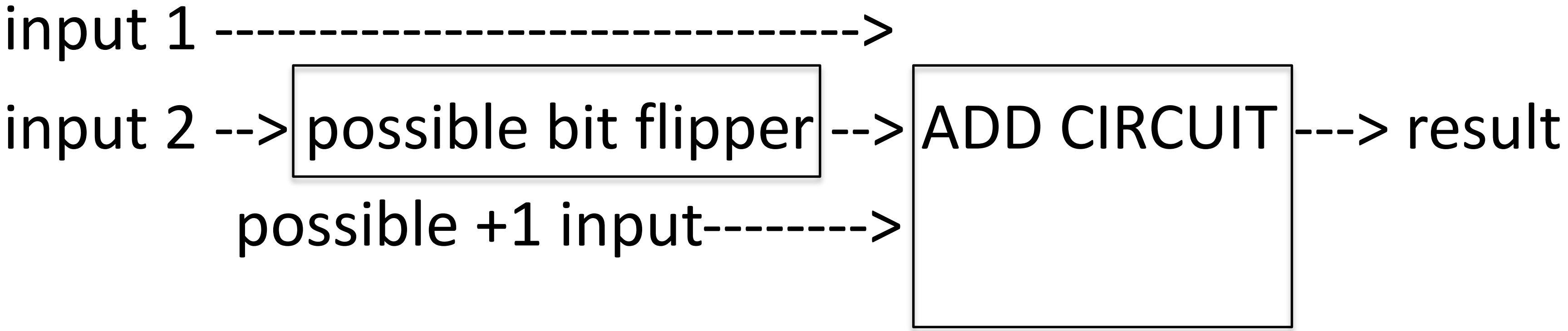
- 6 - 7 = 6 + (-7) = 6 + (~7 + 1)
  - ~7 is shorthand for “flip the bits of 7”

Subtraction Hardware

Negate and add 1 to second operand:

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1



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

input 2 --> possible bit flipper --> ADD CIRCUIT ---> result

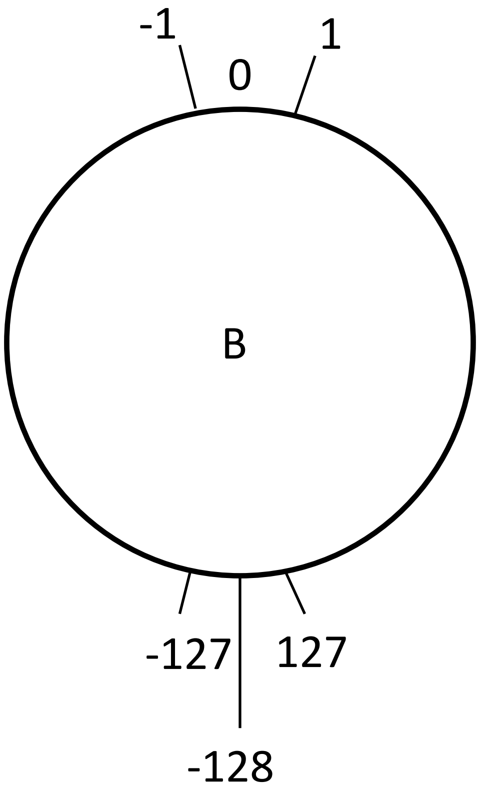
possible +1 input-------->

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

A. Yes, it's gone.

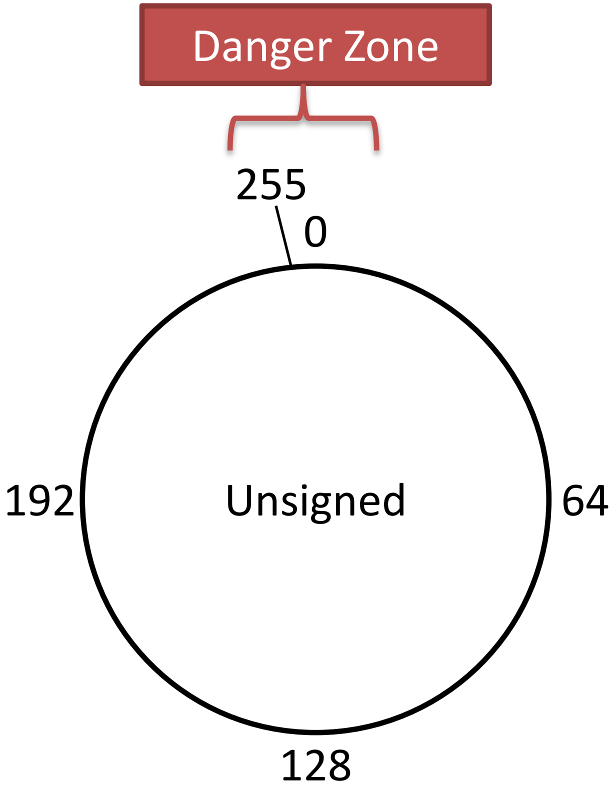
B. Nope, it's still there.

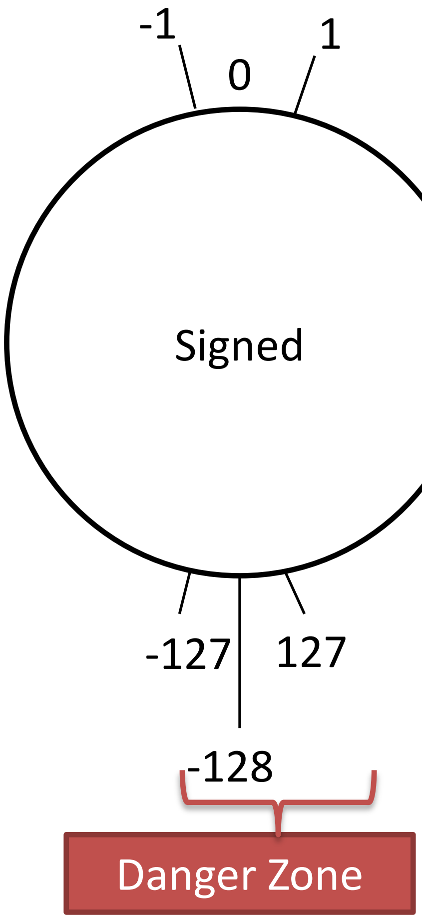
C. It's even worse now.



This is an issue we need to be aware of when adding and subtracting!

Overflow, Revisited



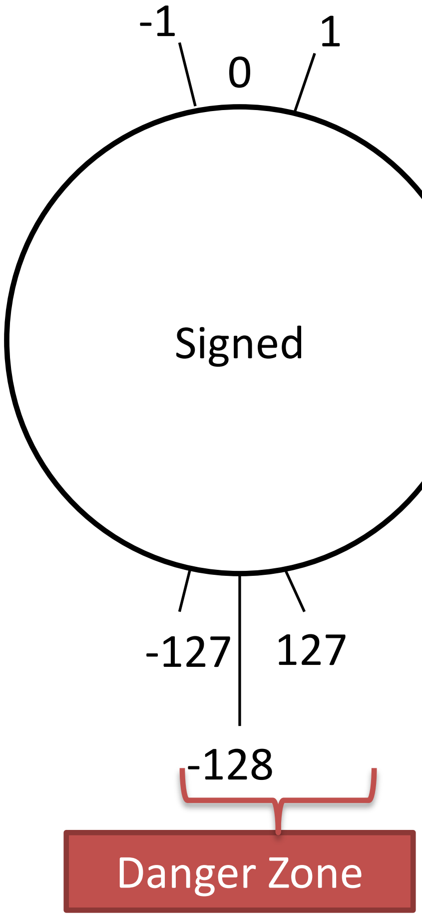


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

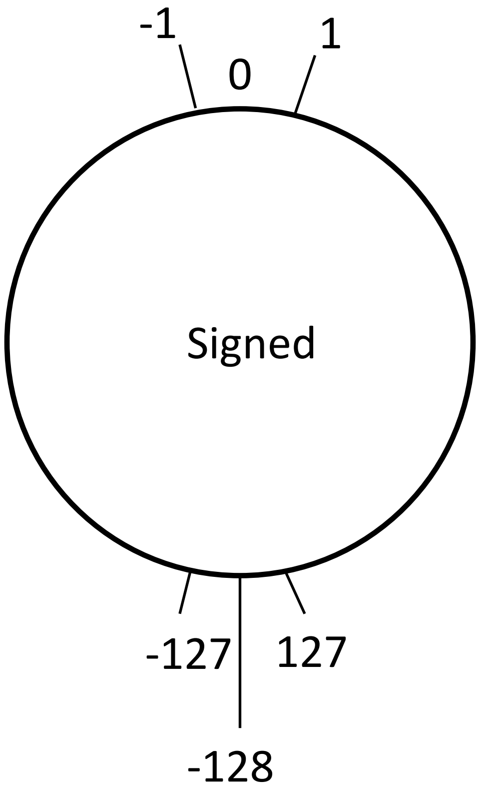
B. Sometimes

C. Never



Signed Overflow

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



<u>Signed addition (and subtraction)</u>:

```
   2+-1=1      2+-2=0      2+-4=-2
  0010        0010        0010
 +1111       +1110       +1100
 -----       -----       -----
1 0001      1 0000        1110
```

No chance of overflow here - signs of operands are different!

Signed Overflow

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

Signed addition (and subtraction):

2+-1=1	2+-2=0	2+-4=-2	2+7=-7	-2+-7=7
0010	0010	0010	**0**010	**1**110
+1111	+1110	+1100	+**0**111	+**1**001
1 0001	1 0000	1110	**1**001	1 **0**111

Overflow here! Operand signs are the same, and they don't match output sign!

Overflow Rules

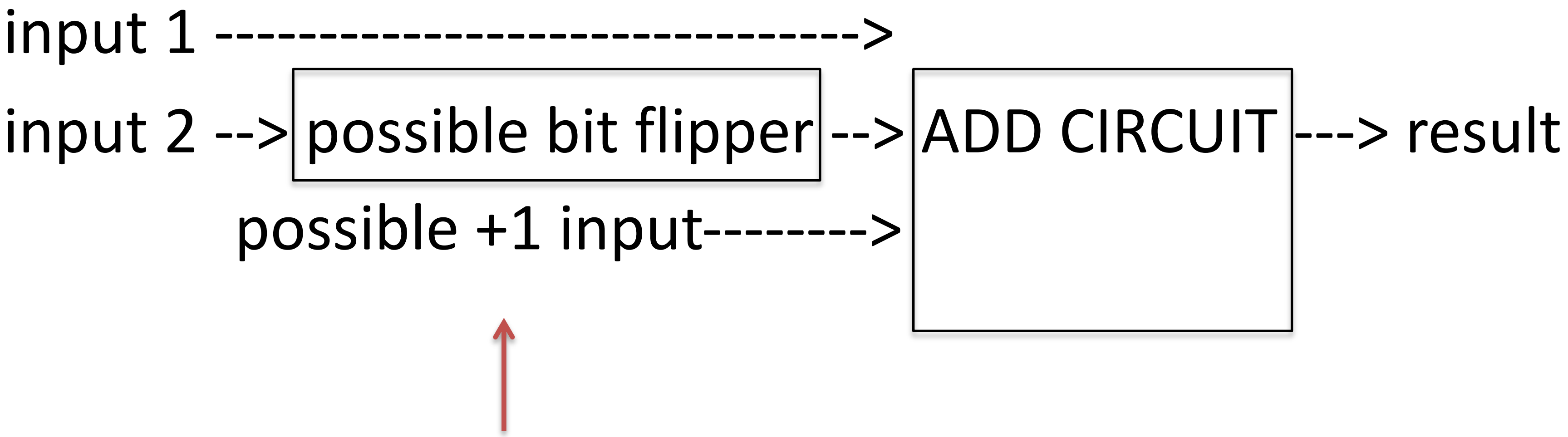
- Signed:
  - The sign bits of operands are the same, but the sign bit of result is different.
- Can we formalize unsigned overflow?
  - Need to include subtraction too, skipped it before.

Recall Subtraction Hardware

Negate and add 1 to second operand:

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 +1 input: “Carry in”

How many of these unsigned operations have overflowed?

4 bit unsigned values (range 0 to 15):

Addition (carry-in = 0)

```
                                     carry-in   carry-out
                                         ↓          ↓
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)

```
                       (-3)
6 - 3  =   0110 + 1100 + 1  =  1   0011
3 - 6  =   0011 + 1010 + 1  =  0   1101
                       (-6)
```

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 (range 0 to 15):

```
                                     carry-in    carry-out
                                         ↓           ↓
Addition (carry-in = 0)
   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)
                      (-3)
   6 -  3  =   0110 + 1100 + 1  =  1   0011 =  3
   3 -  6  =   0011 + 1010 + 1  =  0   1101 = 13
                      (-6)
```

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 to equivalent larger size:

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

```
   z = z + y;                        z = z + x;

0000000000001010                  0000000000000101
+       00000010                  +       11110011
0000000000000010                  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, 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 you please)

& (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

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

More Operations on Bits

- 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

Up Next!

- C programming