CS 31: Introduction to Computer Systems

03: Binary Arithmetic

January 28

SWARTHMORE COLLEGE

Clicker Question:
Have you registered your clicker?

A. Yes
B. No
C. What’s a clicker?

Check your frequency:

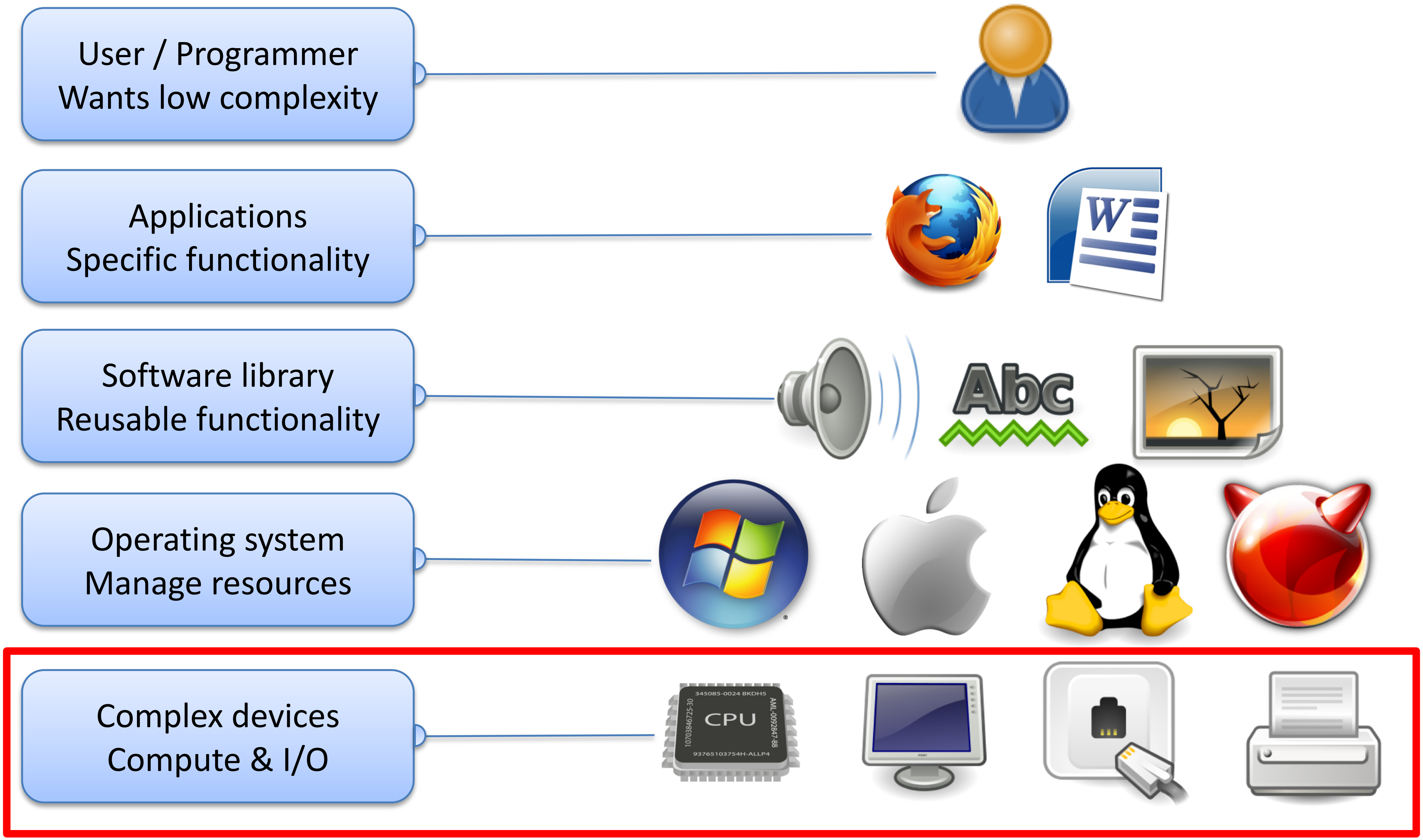
- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

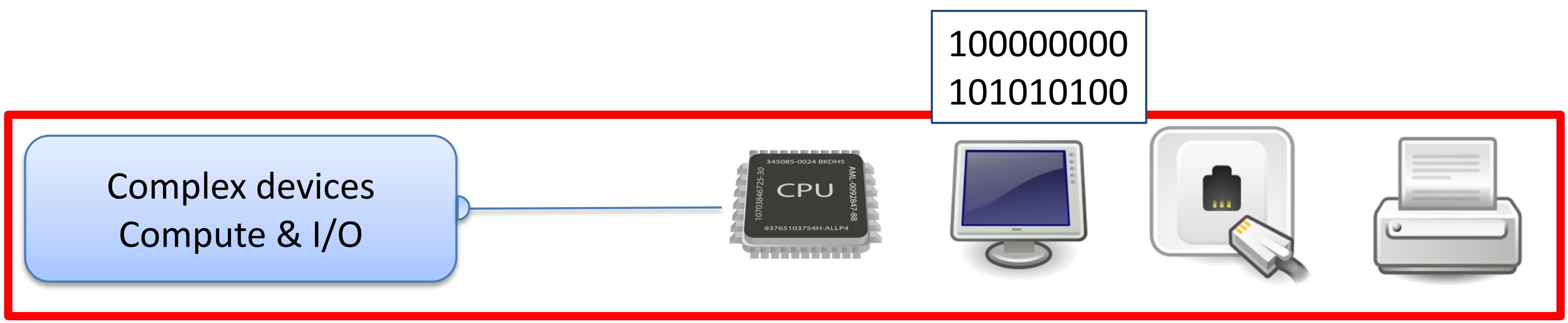
Reset:
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success

Reading Quiz

Abstraction



Abstraction



Today

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

Last Class: Binary Digits: (BITS)

	7 6 5 4 3 2 1 0	
Most significant bit →	10001111	← 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

Last Class: Unsigned Arithmetic (one byte)

- one byte
  - 2^8 (256) values
  - unsigned : 0 – 255

0 = 00000000

+1

1 = 00000001

2 = 00000010

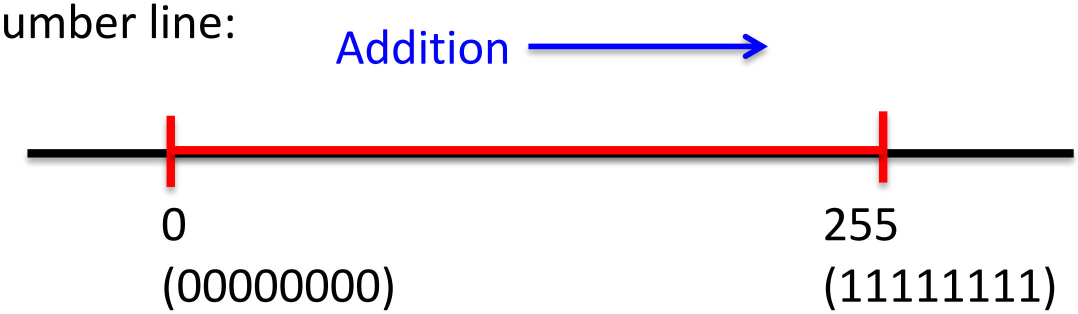
...

254 = 11111110

+1

255 = 11111111

Number line:



255 =
$1*2^7 + \ldots + 1*2^0$

Last Class: Unsigned Arithmetic (one byte)

- one byte
  - 2^8 (256) values
  - unsigned : 0 – 255

we cannot represent an infinite number of values in a finite storage space

252 = 11111100

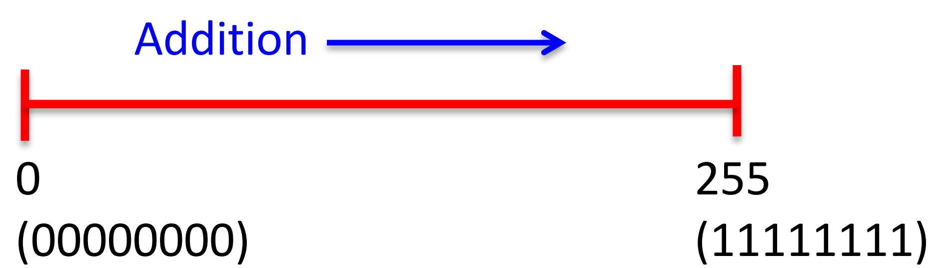
+1

253 = 11111101

254 = 11111110

255 = 11111111

+1



What if we add one more?

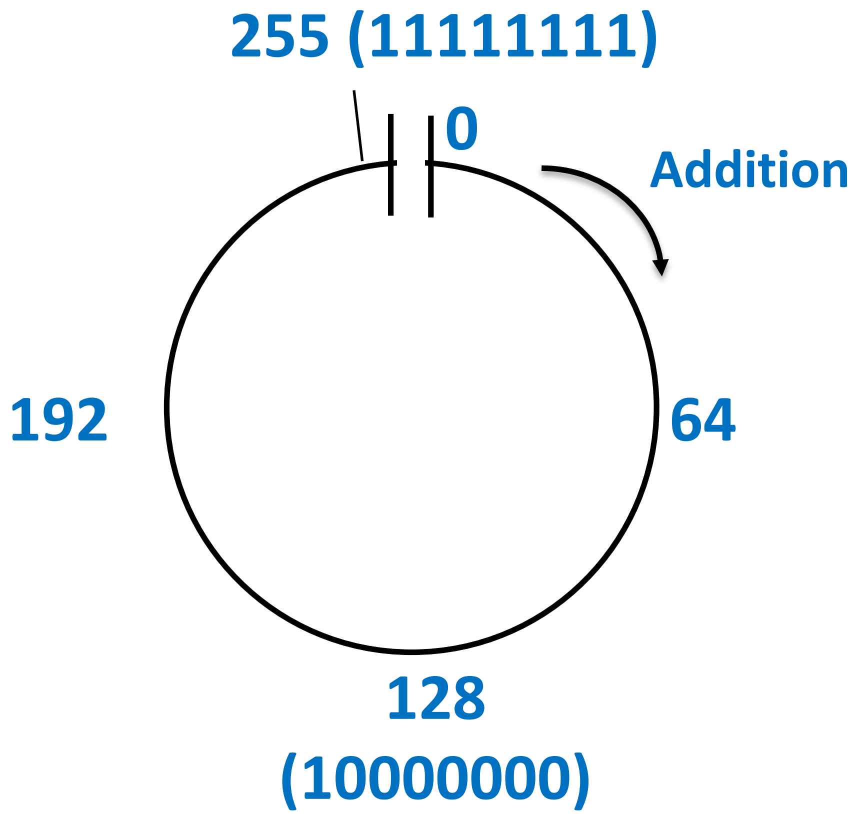
255 + 1 is ?

Car odometer "rolls over".

Last Class: Arithmetic and Fixed Storage

- Fixed Storage: finite set of values
  - 1 byte: 2^8 (256) values
  - unsigned values: 0 – 255
- Yields Modular Arithmetic
  - All operations are % 256

(eg) 255 + 4 = 259 % 256 = 3



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:

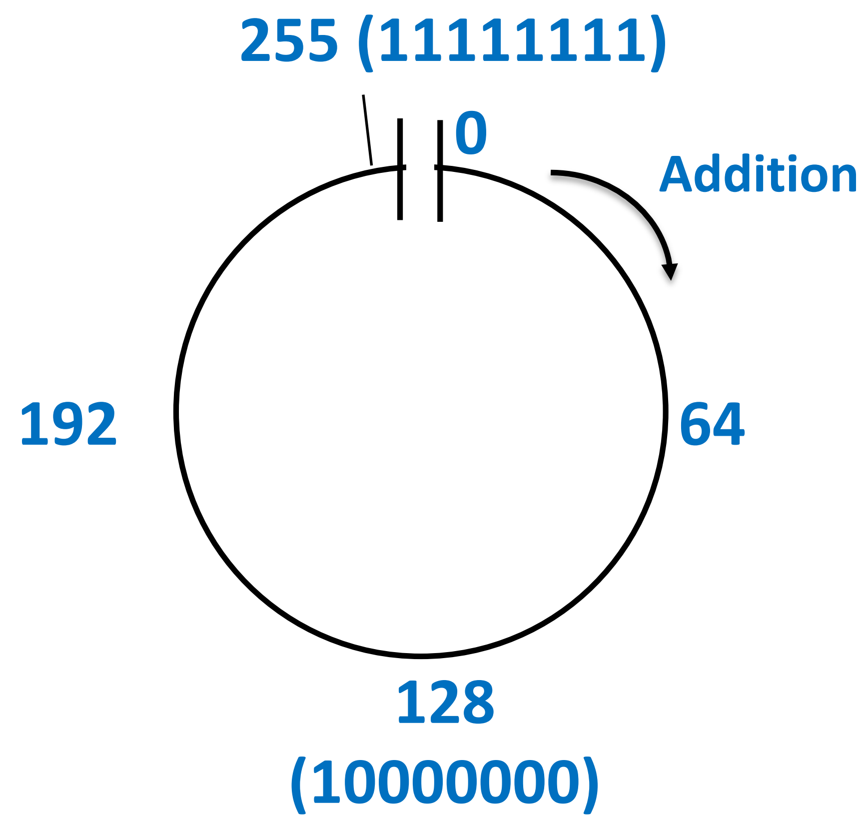
```
   1
   0110     6           1100     12
+  0100   + 4        +  1010    +10
---------------      ----------------
   1010    10        1  0110      6
                     ^carry out
```

Four bits give us range: 0 - 15

Overflow!

Last Class: Arithmetic and Fixed Storage

- Fixed Storage: finite set of values
  - 1 byte: 2^8 (256) values
  - unsigned values: 0 – 255



Not Used: Signed Magnitude Representation (for 4 bit values)

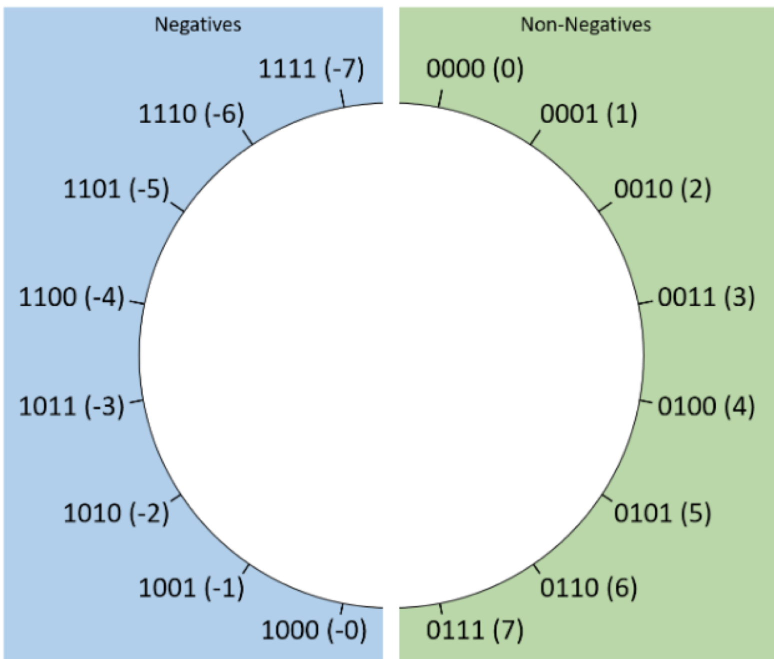


Figure 1. A logical layout of signed magnitude values for bit sequences of length four.

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!

For one byte:

0 = 00000000

-0? = 10000000

Major con: Two ways to represent zero.

Used Today: Two's Complement Representation (for four bit values)

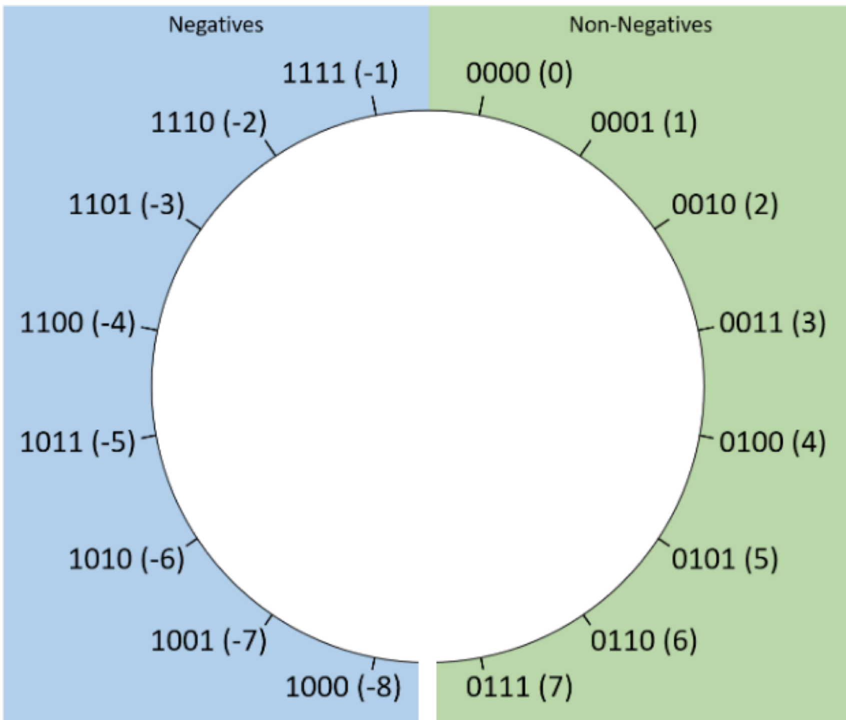
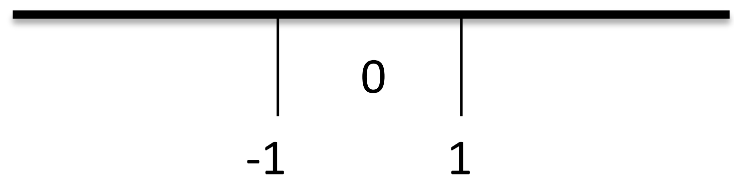


Figure 2. A logical layout of two's complement values for bit sequences of length four.

- 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
  - 0: positive
  - 1: negative

- 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 Complement

- Each two's complement 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 complement 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 complement 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 $\qquad -16 + 8 + 1 = -7$

C. -9

D. -25

"If we interpret…"

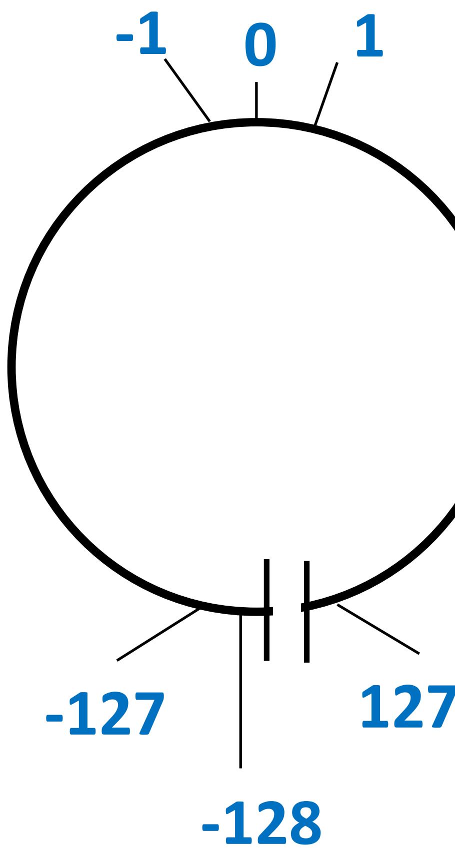
What is the decimal value of 1100?

- …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

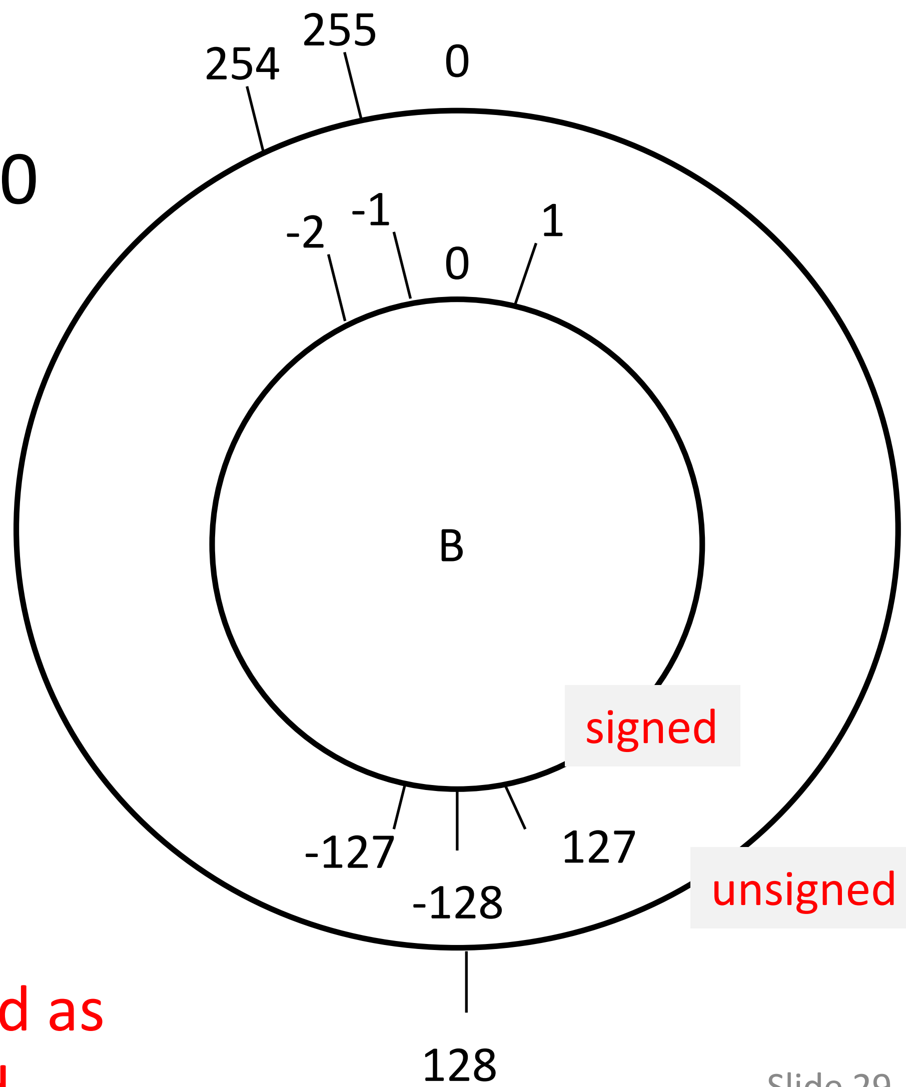
- To negate a value x, we want to find y such that x + y = 0.
- For N bits, $y = 2^N - x$



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

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

Negation Two Ways

4 bit Examples			
x	-x	2^4 - x	Bit flip + 1
0000	0000	10000 – 0000 = 0000	1111 + 1 = 0000
0001	1111	10000 – 0001 = 1111	1110 + 1 = 1111
0010	110	10000 – 0010 = 1110	1101 + 1 = 1110
0111	1001	10000 – 0111 = 1001	1000 + 1 = 1001

Decimal to Two's Complement with 8 bit values (high-order bit is the sign bit)

for positive values, use same algorithm as for unsigned

- `(E.g.)6   6 - 4 = 2` $(4{:}2^2)$
- `2 - 2 = 0` $(2{:}2^1)$`:   00000110`

for negative values:

- convert negation (positive) to binary
- then negate binary to get negative

`E.g.: -3`

- `3: 00000011`
- `negate: 11111100+1 = 11111101 = -3`

Decimal to Two's Complement with 8 bit values (high-order bit is the sign bit)

for negative values:

- convert negation (positive) to binary
- then negate binary to get negative

Try converting -7 to Two's Complement representation

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

Decimal to Two's Complement with 8 bit values (high-order bit is the sign bit)

for negative values:

- convert negation (positive) to binary
- then negate binary to get negative

Try converting -7 to Two's Complement representation

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

-7 = (1) 7: 00000111
(2) negate: 11111000 + 1 = 11111001

Addition & Subtraction for Integers

- 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

<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 signed Examples:

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
```

Addition: don't flip bits or add 1

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

Signed & Unsigned 4-bit Subtraction:

Unsigned subtraction: flip bits and add 1

```
13 -  1 =
```

Signed subtraction: flip bits and add 1

```
-3 - 1 =
```

A. 1100 & 1100
B. 1100 & 1010
C. 1010 & 1010
D. 1001 & 1100

Signed & Unsigned 4-bit Subtraction:

Unsigned subtraction: flip bits and add 1

```
13 -  1 =  1101
           1110      (1: 0001  ~1: 1110)
        +  ____1
       1   1100 = 12
```

Signed subtraction: flip bits and add 1

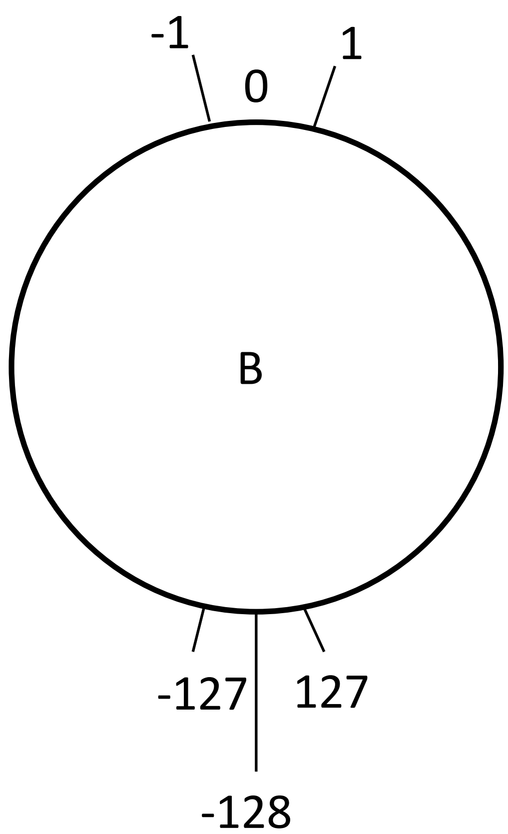
```
-3 - 1 =   1101
           1110
        +  ____1
       1   1100 = -4
```

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.

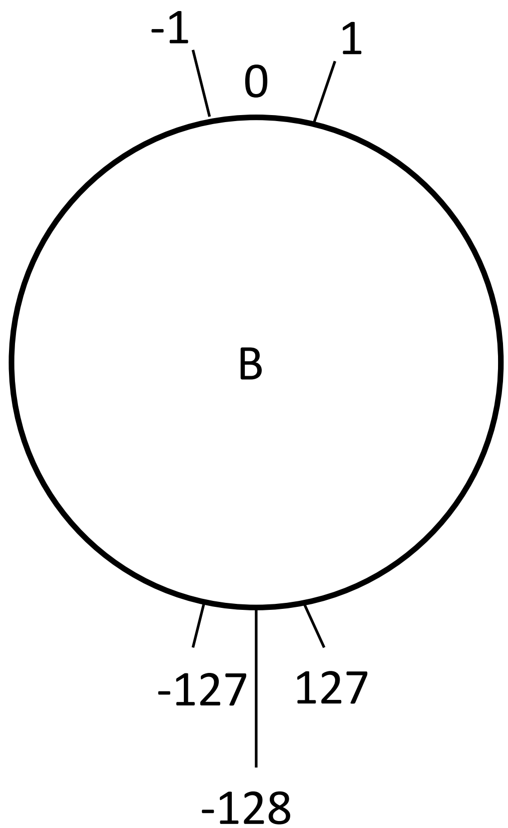


By switching to two's complement, have we solved this value "rolling over" (overflow) problem?

A. Yes, it's gone.

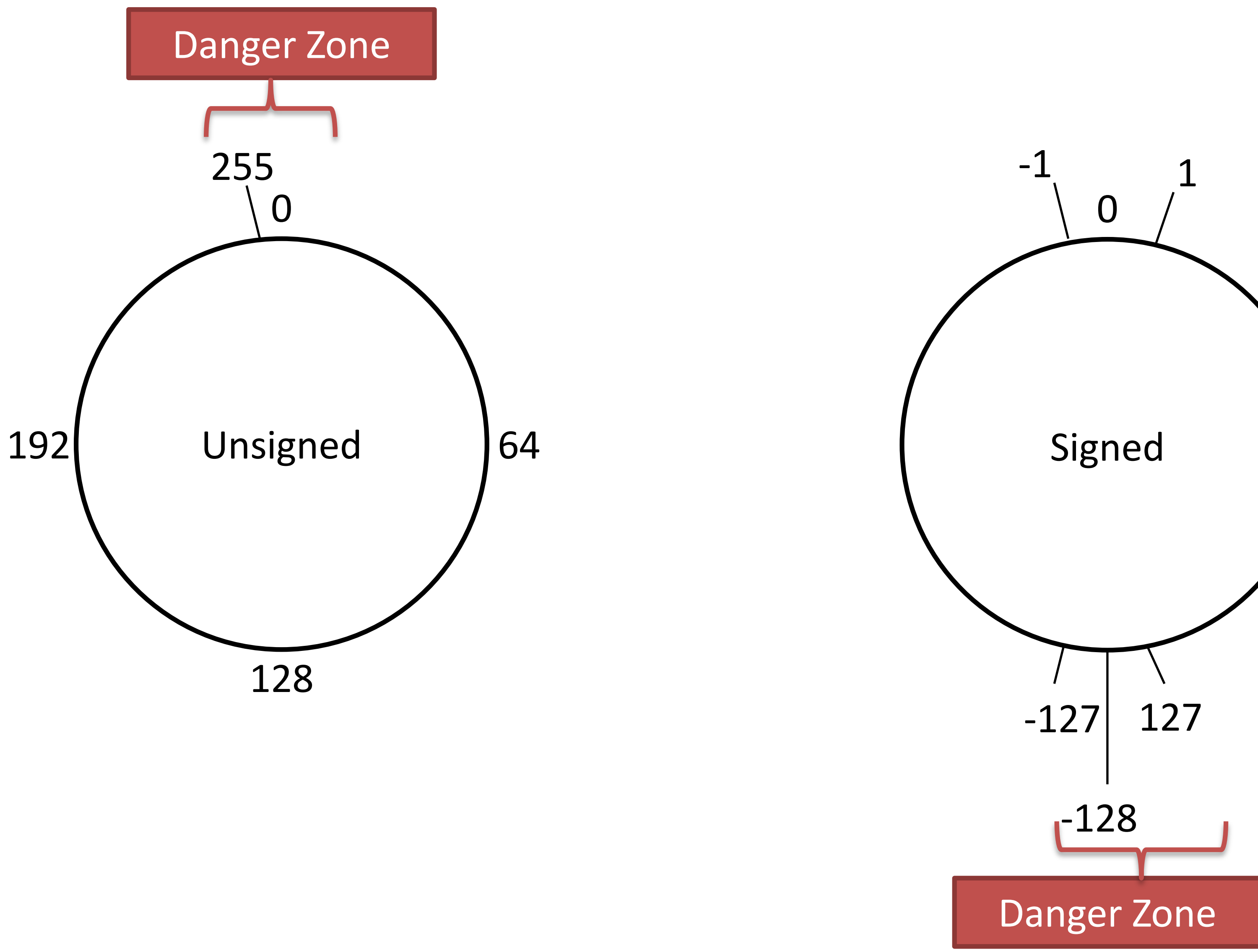
B. Nope, it's still there.

C. It's even worse now.



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

Overflow, Revisited

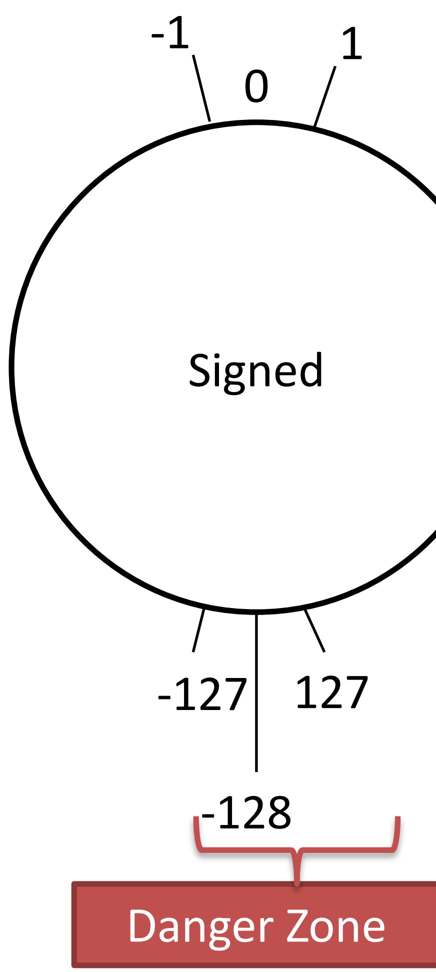


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never

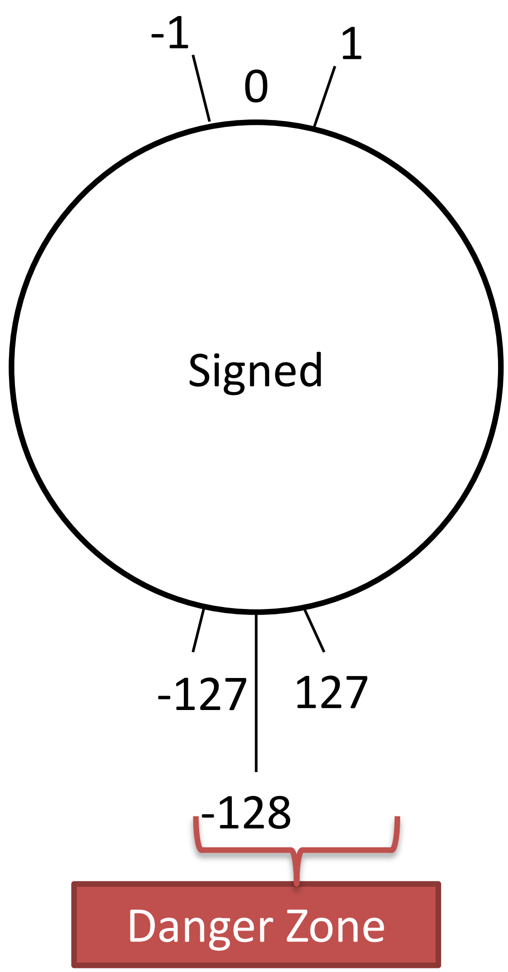


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never



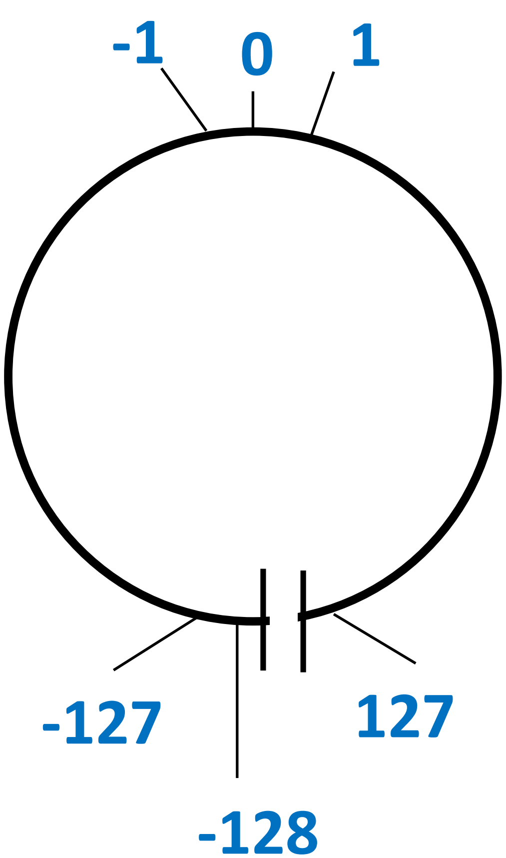
Signed (Two's Complement) Overflow For Addition

- **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
```



Signed (Two's Complement) Overflow For <u>Addition</u>

- **<u>Addition Overflow</u>**: IFF the sign bits of <u>operands are the same,</u> but the sign bit of <u>result is different</u>.
- 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

Signed (Two's Complement) Overflow For Subtraction

Subtraction Overflow Two Rules:

- **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

Signed (Two's Complement) Overflow For Subtraction

Subtraction Overflow Rules Summarized:

- 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)

Signed (Two's Complement) Overflow For Subtraction

Subtraction Overflow Two Rules:

– **Rule 1:**

- Positive operand - Negative operand = Positive Result: No Overflow
- Positive operand - Negative operand = Negative Result: 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)
```

Signed (Two's Complement) Overflow For Subtraction

Subtraction Overflow Two Rules:

– **Rule 2:**

- Negative operand - Positive operand = Negative Result: No Overflow
- Negative operand - Positive operand = Positive Result: 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)
```

Overflow Rules

- Signed (Two's Complement):
  - Addition:
    - The sign bits of operands are the same, but the sign bit of result is different.
  - Subtraction:
    - First compute the following: if the sign bits of the subtraction operands are different, and the sign bit of the result and subtrahend are the same. (minuend-subtrahend – result)
    - then, turn into an Addition operation

- Can we formalize unsigned overflow?
  - Need to include subtraction too, skipped it before.

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?

4 bit unsigned values (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)

```
                    (-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?

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?

How many of these unsigned operations have overflowed?

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

                             (-3)
Subtraction (carry-in = 1)
   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;

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, 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

Try out 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

Try out some 4-bit examples:

bit-wise operations:

- 0101 & 1101 = 0101
- 0101 | 1101 = 1101

Logical (unsigned) bit shift:

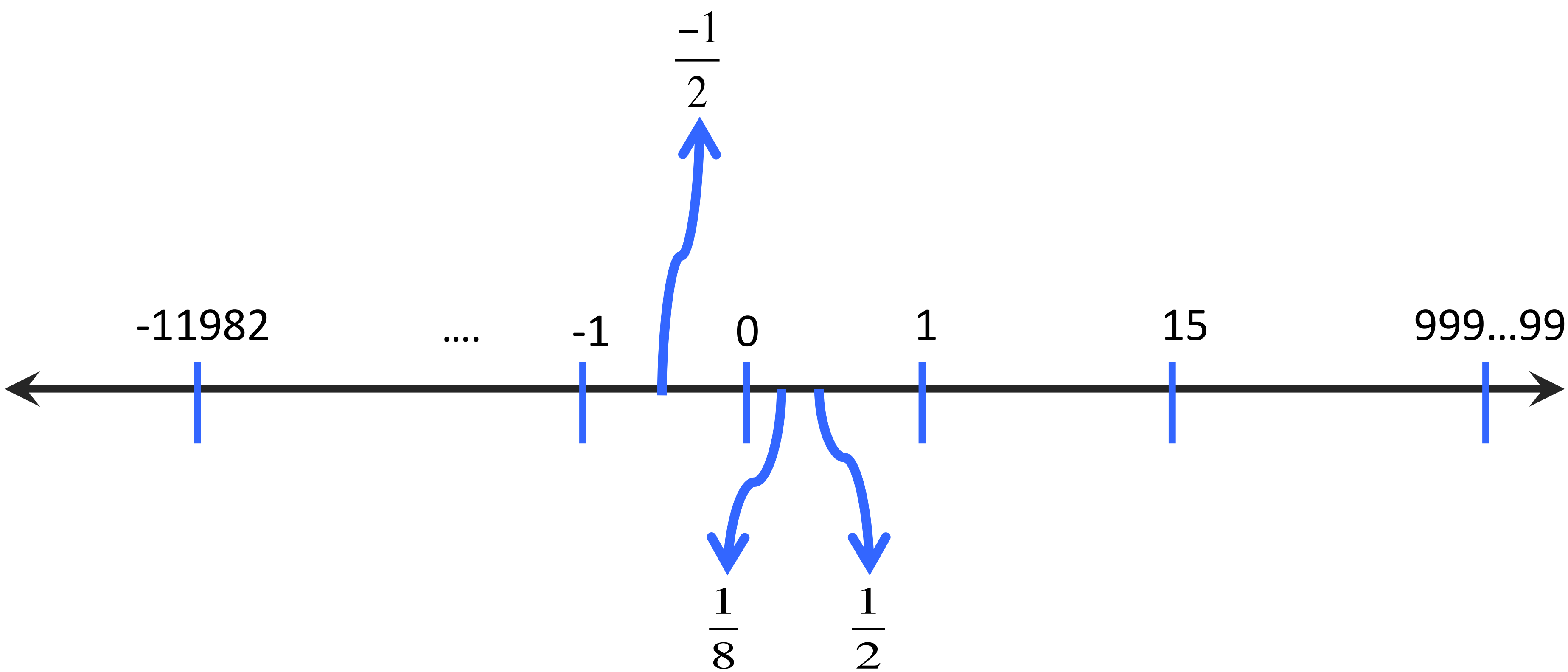
- 1010 << 2 = 1000
- 1010 >> 2 = 0010

Arithmetic (signed) bit shift:

- 1010 << 2 = 1000
- 1010 >> 2 = 1110

Additional Info: Fractional binary numbers

How do we represent fractions in binary?



Additional Info: Representing Signed Float Values

- One option (used for floats, NOT integers)
  - Let the first bit represent the sign
  - 0 means positive
  - 1 means negative

- For example:
  - 0101 -> 5
  - 1101 -> -5

- Problem with this scheme?

Additional Info: Floating Point Representation

1 bit for sign
8 bits for exponent
23 bits for precision

sign | exponent | fraction |

$$\text{value} = (-1)^{\text{sign}} * 1.\text{fraction} * 2^{(\text{exponent}-127)}$$

let's just plug in some values and try it out

```
0x40ac49ba: 0 10000001   01011000100100110111010
      sign = 0 exp = 129   fraction = 2902458
```

$= 1*1.2902458*2^{2} = 5.16098$

I don't expect you to memorize this

Up Next

- C programming