CS 31: Introduction to Computer Systems

03: Binary Arithmetic and Introduction to C

01-28-2025

SWARTHMORE COLLEGE

Announcements

- Register your clicker! https://forms.gle/YBFvNWPTXgiySMHx5
- Reading quizzes count from this week!
- 👀 Keep an eye out for the CS Department Mentoring Program Email!
- Edstem: Turn on notifications so you get email when we post
- HW 1 is out! – New Due Date is Monday Feb 3rd
- Please give me your accommodation forms this week

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

<u>Check your frequency:</u>

- 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

What we will learn this week

1. Operations on binary data (continued from last week)
   - Addition and Subtraction on integer types. (e.g.: 6 + 12 15 – 5 -9 + 12)
   - Some other operations on bits
   - Bit shifting, bit-wise OR, AND and NOT

2. Introduction to C
   - Comparison of C vs. Python
   - Basics of C programming
   - Data organization and strings

What is a computer system?

Hardware (HW) & Special Systems Software (OS) that work together to run application programs

What are the goals of our system? Correctness

- Is x^2 >= 0?
  - Floating point values: Yes!
  - Integers
    - 40000 * 40000 = 1600000000
    - 50000 * 50000 = ??

example: courtesy Eleanor Birrell

Two's Complement Representation (for four bit values)

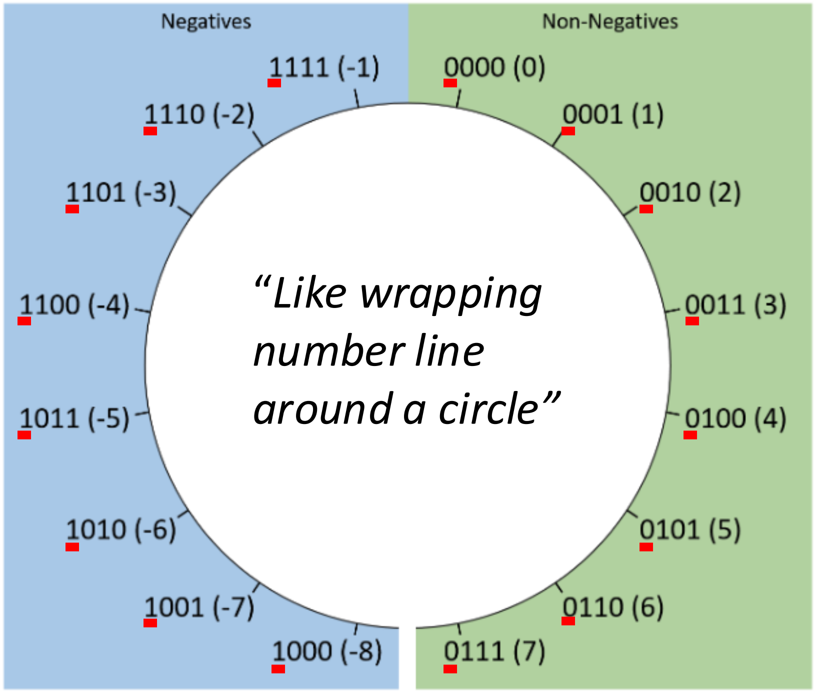
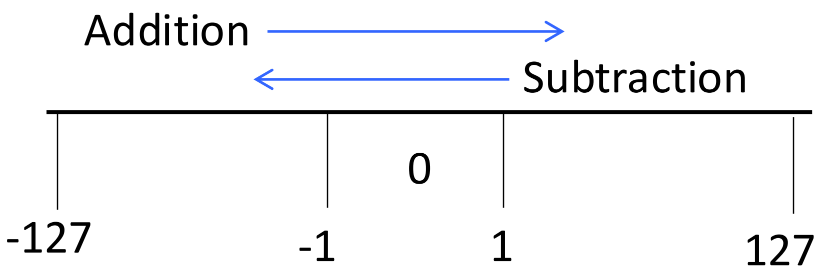


Figure 2. A logical layout of two's complement values for bit sequences of length four.

For an 8 bit range we can express 256 unique values:

- 128 non-negative values (0 to 127)
- 128 negative values (-1 to -128)

Borrow nice property from number line:



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

- Addition moves to the right
- Subtraction moves to the left.

Used Today

Let's try some more examples

High order bit is the sign bit, otherwise just like unsigned conversion. 4-bit and 8-bit examples:

4 bit numbers (4th bit is the sign bit)

0110

1110

8 bit numbers (8th bit is the sign bit)

00001010

11111111

Let's try some more examples

High order bit is the sign bit, otherwise just like unsigned conversion. 4-bit and 8-bit examples:

4 bit numbers (4th bit is the sign bit)

0110 = $-2^3 \times 0 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 0 = 6$

1110 = $-2^3 \times 1 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 0 = -2$

8 bit numbers (8th bit is the sign bit)

00001010= $-2^7 \times 0 + 2^6 \times 0 + 2^5 \times 0 + 2^4 \times 0$

$+2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0 = 10$

11111111= $-2^7 \times 1 + 2^6 \times 1 + 2^5 \times 1 + 2^4 \times 1$

$+2^3 \times 1 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 1 = -1$

“If we interpret…”

- What is the decimal value of 1100?

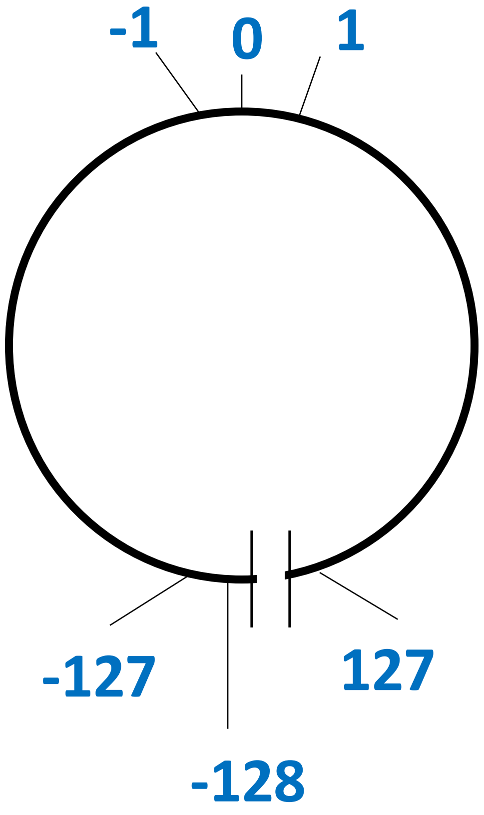
- …as unsigned, 4-bit value: 12 (%u)
- …as signed (two’s complement), 4-bit value: -4 (%d)

- …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, $y = 2^N - x$

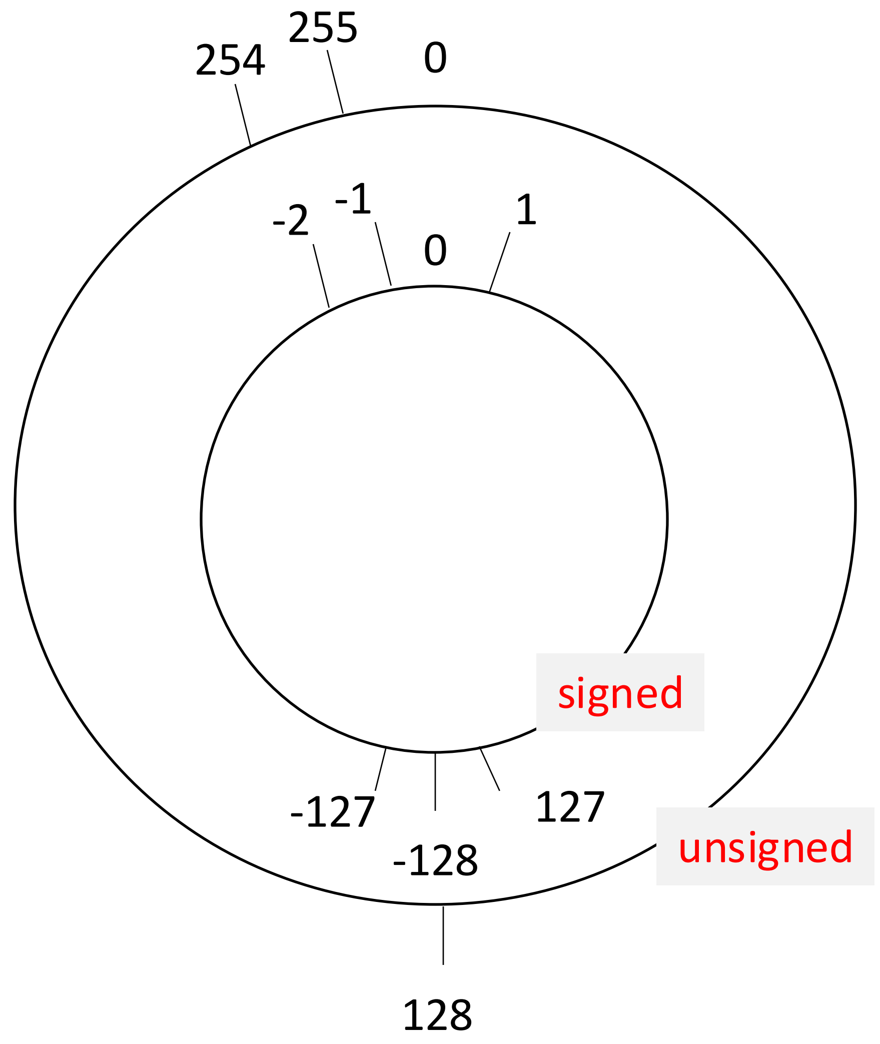


Negation Example (8 bits)

- 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 unsigned binary is 11010010 = -46

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

Negation Summary: 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 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

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

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

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

<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)

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:

```
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:

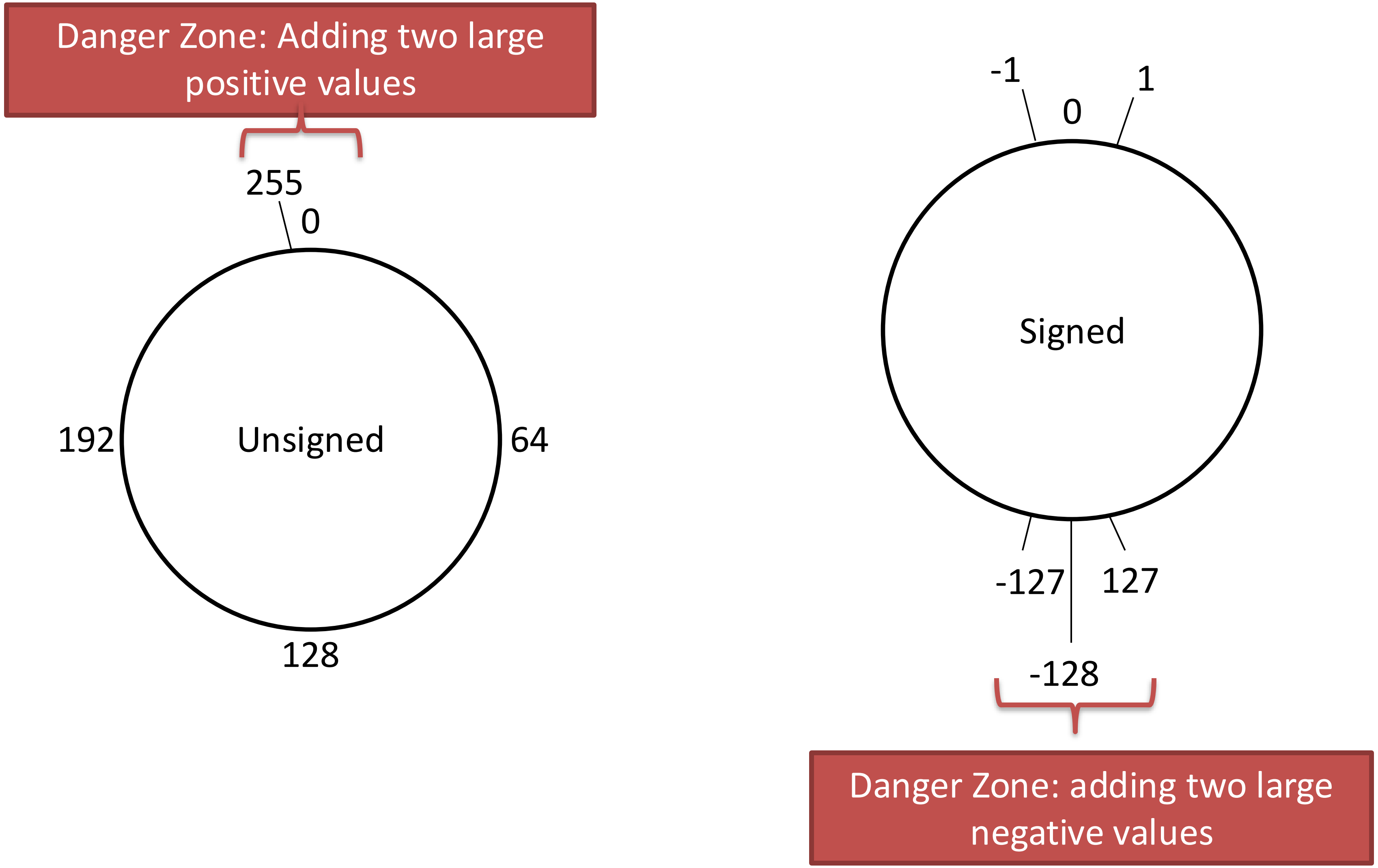
Unsigned subtraction: flip bits and add 1

```
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
```

Overflow, Revisited

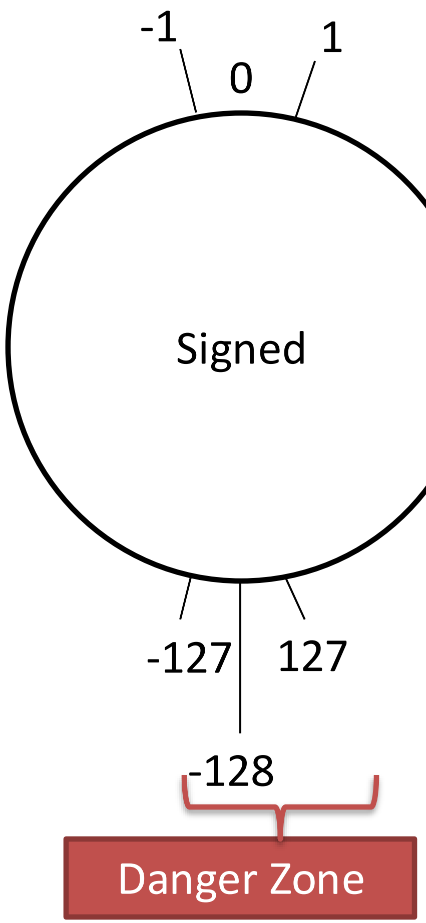


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never

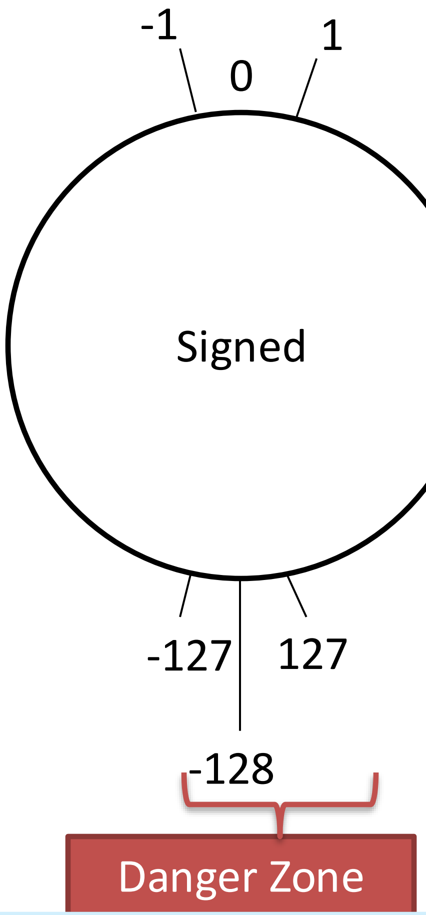


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never



Two's Complement Overflow For Addition

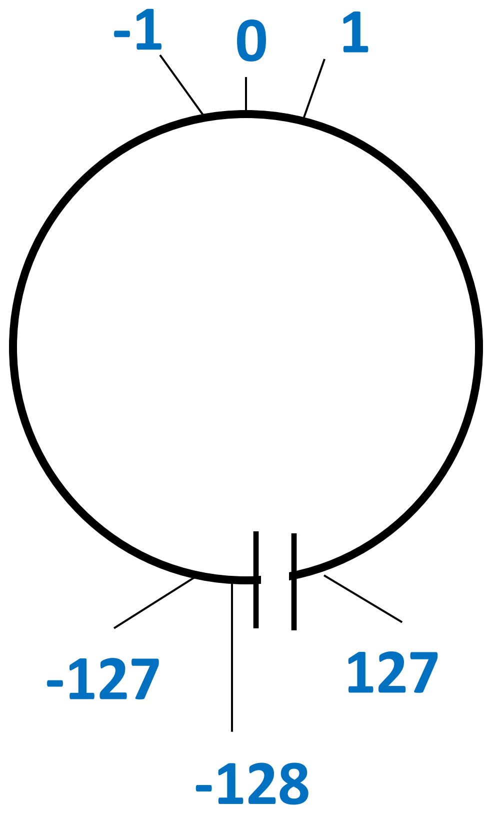
- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

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
```

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):

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)

6 - 3 =	0110 +	1100 (-3) +	1 =	1	0011
3 - 6 =	0011 +	1001 (-6) +	1 =	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):

Notice a Pattern?

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	+ 1100 (-3)	+ 1	=	1	0011	=	3
3 - 6	=	0011	+ 1001 (-6)	+ 1	=	0	1101	=	13

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

How many of these <u>unsigned</u> operations have overflowed?

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

Notice a Pattern?

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	+ 1100 (-3)	+ 1	=	1	0011	= 3
3 - 6	=	0011	+ 1001 (-6)	+ 1	=	0	1101	= 13

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

Overflow Rule Summary

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

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)

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*

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:**

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*

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

– **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*

⭐ 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;                            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, 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

Try some 4-bit examples:

bit-wise operations:

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

Logical (unsigned) bit shift:

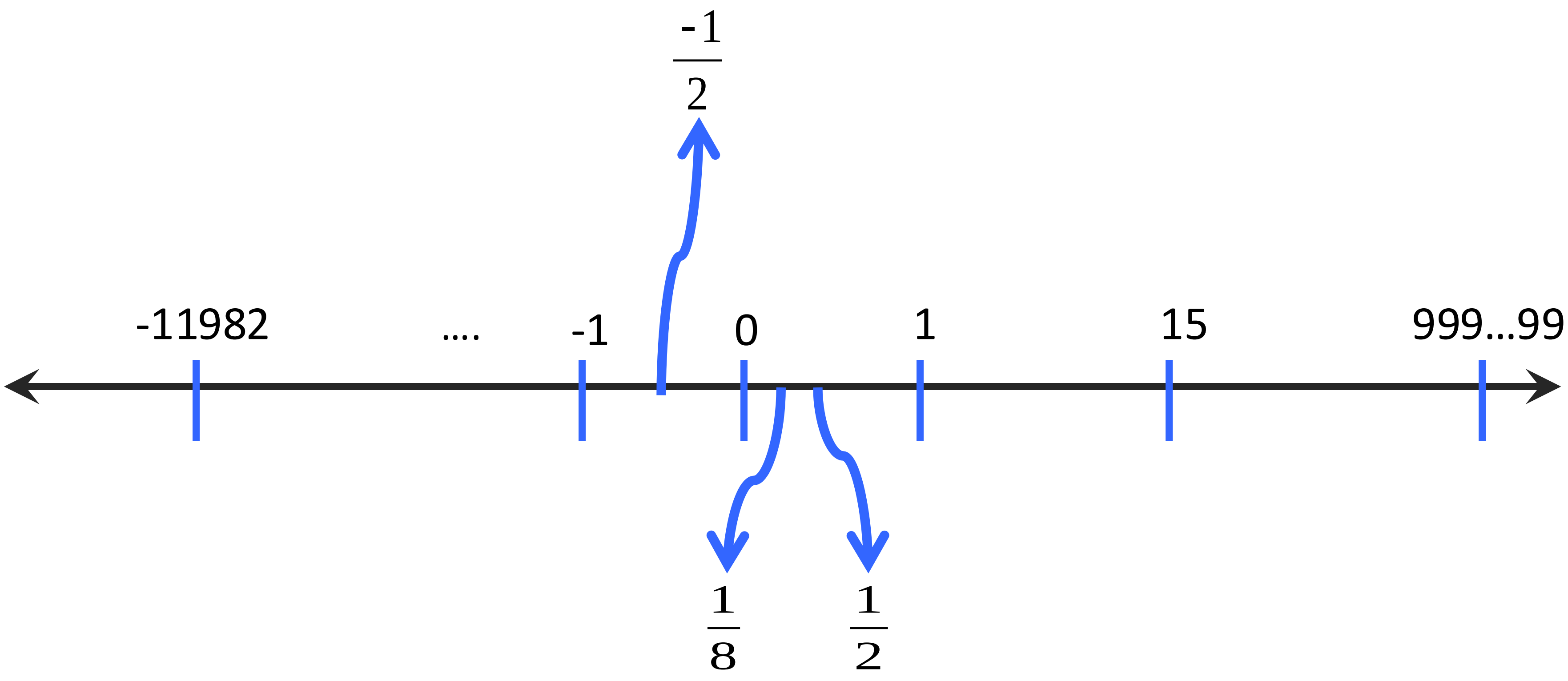
- 1010 << 2 = 1000
- 1010 >> 2 = 0010

Arithmetic (signed) bit shift:

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

Additional Info: (not assessable) Fractional binary numbers

How do we represent fractions in binary?



Additional Info: (not assessable) Floating Point Representation

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

sign | exponent | fraction |

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

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
```

<u>You’re not expected to memorize this</u>

Summary

- Images, Word Documents, Code, and Video can represented in bits.
- Byte or 8 bits is the smallest addressable unit
- N bits can represent 2^N unique values
- A number is written as a sequence of digits: in the decimal base system
  - [dn * 10 ^ n] + [dn-1 * 10 ^ n-1] + ... + [d2 * 10 ^ 2] + [d1 * 10 ^ 1] + [d0 * 10 ^ 0]
  - For any base system:
  - [dn * b ^ n] + [dn-1 * b ^ n-1] + ... + [d2 * b ^ 2] + [d1 * b ^ 1] + [d0 * b ^ 0]
- Hexadecimal values (represent 16 values): {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
  - Each hexadecimal value can be represented by 4 bits. (2^4=16)
- A finite storage space we cannot represent an infinite number of values. For e.g., the max unsigned 8 bit value is 255.
  - Trying to represent a value >255 will result in an overflow.
- Two's Complement Representation: 128 non-negative values (0 to 127), and 128 negative values (-1 to -128).