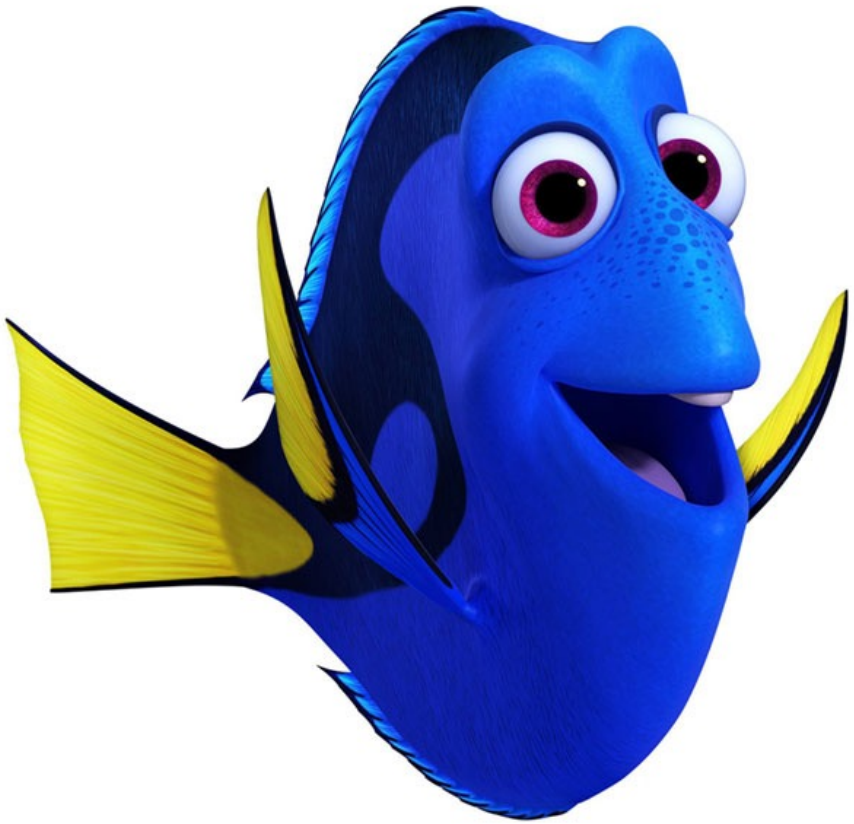
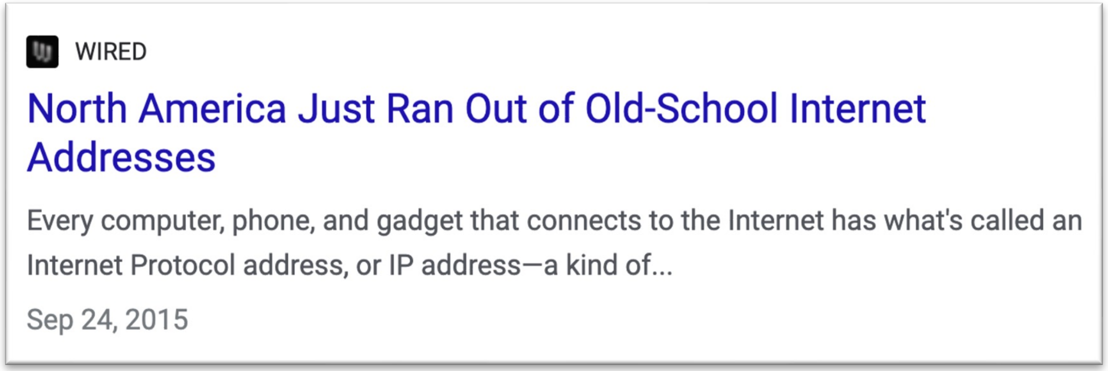
Welcome! *Discuss now with your neighbor*:



1. What is 192.168.1.1?

2. What is 2001:db8:3333:4444:5555:6666:7777:8888?



WIRED

North America Just Ran Out of Old-School Internet Addresses

Every computer, phone, and gadget that connects to the Internet has what's called an Internet Protocol address, or IP address—a kind of...

Sep 24, 2015

IPv4: $2^{32} \sim= 4.2 \times 10^{9}$ addresses

IPv6: $2^{128} \sim= 3.4 \times 10^{38}$ addresses

CS31: Introduction to Computer Systems

Week 2, Class 2

Binary Arithmetic

02/01/24

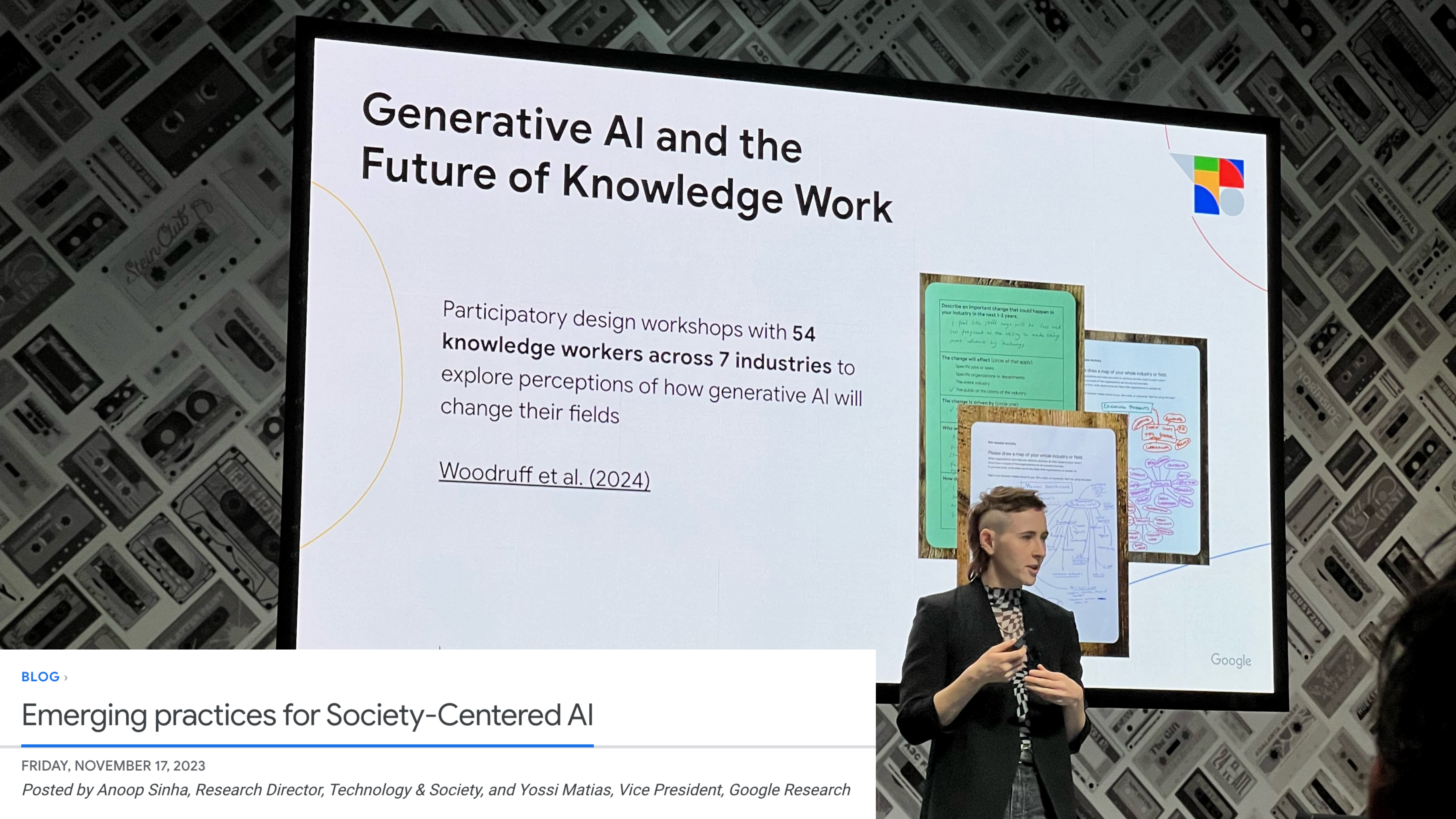
Dr. Sukrit Venkatagiri
Swarthmore College

Dive into Systems by Matthews, Newhall, and Webb

Stable Diffusion + Artists

Announcements

- Clickers will count for credit from Tuesday, Feb 6th
- Lab 1 due today 11:59pm
  - For Lab 1: Feedback re: using style guide
  - From Lab 2: points will be deducted if you don’t ~follow style guide
- HW1 due tomorrow 11:59pm
- **Office hours today: 3-4pm**
- Accommodations -> please meet with me



Generative AI and the Future of Knowledge Work

Participatory design workshops with **54 knowledge workers across 7 industries** to explore perceptions of how generative AI will change their fields

Woodruff et al. (2024)

BLOG ›

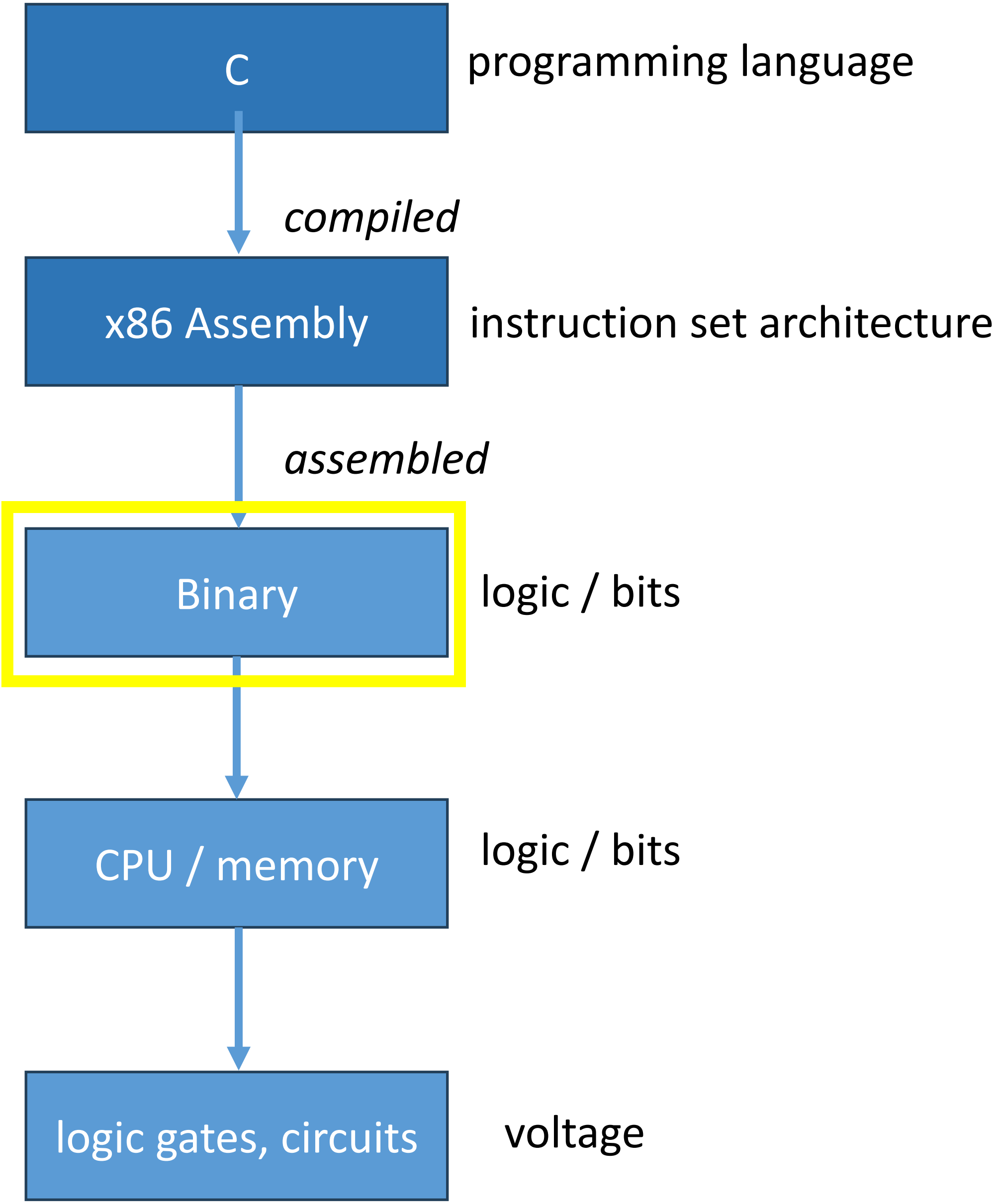
Emerging practices for Society-Centered AI

FRIDAY, NOVEMBER 17, 2023

Posted by Anoop Sinha, Research Director, Technology & Society, and Yossi Matias, Vice President, Google Research

Where are we?

Wk	Lecture	Lab
1	Intro to C	C Arrays, Sorting
2	Binary Representation, Arithmetic	Data Rep. & Conversion
3	Digital Circuits	Circuit Design
4	ISAs & Assembly Language	”
5	Pointers and Memory	Pointers and Assembly
6	Functions and the Stack	Binary Maze
7	Arrays, Structures & Pointers	”
Spring Break		
8	Storage and Memory Hierarchy	Game of Life
9	Caching	”
10	Operating System, Processing	Strings
11	Virtual Memory	Unix Shell
12	Parallel Applications, Threading	”
13	Threading	pthreads Game of Life
14	Threading	”



C — programming language

compiled

x86 Assembly — instruction set architecture

assembled

Binary — logic / bits

CPU / memory — logic / bits

logic gates, circuits — voltage

Last class…

- Chars, strings, structs, and functions
- Bits, bytes
- Binary and hexadecimal representation
- Converting from binary to decimal

On a computer, data is stored in the following format(s)…

A. Hexadecimal

B. Binary

C. It depends on the operating system

D. It depends on the programming language

On a computer, data is stored in the following format(s)…

A. Hexadecimal

B. Binary

C. It depends on the operating system

D. It depends on the programming language

Today: Data Representation & Binary Arithmetic

- Number systems + conversion
- Sizes, representation
- Signed-ness
- Binary arithmetic
- Overflow rules

Other (common) number systems

- Base 2: How data is stored in hardware
- Base 8: Used to represent file permissions
- Base 10: Preferred by people
- Base 16: Convenient for representing memory addresses
- Base 64: Commonly used on the Internet (e.g. email attachments)

It’s all stored as **binary** in the computer

Different representations (or visualizations) of the same information!

Hexadecimal: Base 16

- Fewer digits to represent same value
  - Same amount of information!

- Like binary, the base is power of 2

- Each digit is a “nibble”, or half a byte.

Each hex digit is a “nibble”

- One hex digit: 16 possible values (0-9, A-F)
- 16 = 2^4, so each hex digit has exactly four bits worth of information.
- We can map each hex digit to a four-bit binary value (helps for converting between bases)

Each hex digit is a “nibble”

Example value: 0x1B7

Four-bit value: 1

Four-bit value: B (decimal 11)

Four-bit value: 7

In binary: 0001 1011 0111

1 B 7

Hexadecimal ↔ Binary Conversion

- Bit patterns as base-16 numbers
- Convert binary to hexadecimal: by splitting into groups of 4 bits each.

Example:

0b0011 1100 1010 1101 1011 0011 = 0x3CADB3

Bin	0011	1100	1010	1101	1011	0011
Hex	3	C	A	D	B	3

Converting Decimal -> Binary

- Two methods:
  - division by two remainder
  - powers of two and subtraction

Method 1: decimal value D, binary result b (b_i is ith digit):

```
i = 0
while (D > 0)
   if D is odd
          set $b_i$ to 1
   if D is even
          set $b_i$ to 0
   i++
   D = D/2
```

Example: Converting 105

idea:

example: D = 105 b_0 = 1

Method 1: decimal value D, binary result b (b_i is ith digit):

```
i = 0
while (D > 0)
   if D is odd
          set $b_i$ to 1
   if D is even
          set $b_i$ to 0
   i++
   D = D/2
```

Example: Converting 105

```
idea:     D           example: D = 105        $b_0$ = 1
          D = D/2              D =  52        $b_1$ = 0
```

Method 1: decimal value D, binary result b (b_i is ith digit):

```
i = 0
while (D > 0)
   if D is odd
          set $b_i$ to 1
   if D is even
          set $b_i$ to 0
   i++
   D = D/2
```

Example: Converting 105

idea:	D	example:	D = 105	$b_0 = 1$
	D = D/2		D = 52	$b_1 = 0$
	D = D/2		D = 26	$b_2 = 0$
	D = D/2		D = 13	$b_3 = 1$
	D = D/2		D = 6	$b_4 = 0$
	D = D/2		D = 3	$b_5 = 1$
	D = D/2		D = 1	$b_6 = 1$
	D = 0 (done)		D = 0	$b_7 = 0$

105 = 01101001

Method 2

- $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$
-

  To convert 105:
  - Find largest power of two that's less than 105 (64)
  - Subtract 64 (105 – 64 = 41), put a 1 in d_6
  - Subtract 32 (41 – 32 = 9), put a 1 in d_5
  - Skip 16, it's larger than 9, put a 0 in d_4
  - Subtract 8 (9 – 8 = 1), put a 1 in d_3
  - Skip 4 and 2, put a 0 in d_2 and d_1
  - Subtract 1 (1 – 1 = 0), put a 1 in d_0 (Done)

1	1	0	1	0	0	1
d_6	d_5	d_4	d_3	d_2	d_1	d_0

What is the value of 357 in binary?

8 7654 3210

→ digit position

A. 1 0110 0011
B. 1 0110 0101
C. 1 0110 1001
D. 1 0111 0101
E. 1 1010 0101

$2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$,

$2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$

What is the value of 357 in binary?

8 7654 3210

→ digit position

A. 1 0110 0011
B. 1 0110 0101
C. 1 0110 1001
D. 1 0111 0101
E. 1 1010 0101

357 – 256 = 101
101 – 64 = 37
37 – 32 = 5
5 – 4 = 1

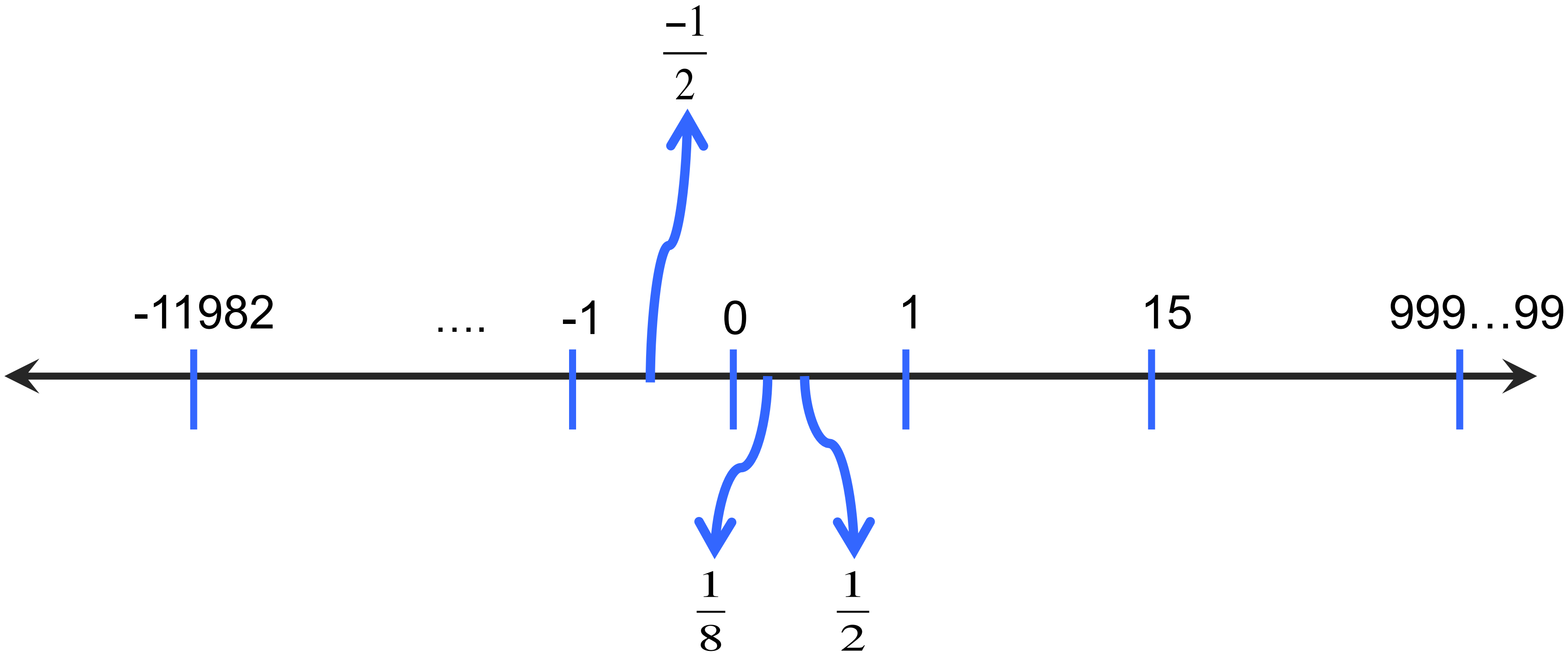
1	0	1	1	0	0	1	0	1
d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0

$2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$,

$2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$

Additional Info: Fractional binary numbers

How do we represent fractions in binary?



Additional Info: Floating Point Representation

1 bit for sign

8 bits for exponent

23 bits for precision (fraction)

sign | exponent | fraction |

value = $(-1)^{\text{sign}}$ * 1.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`

<u>I don't expect you to memorize this</u>

The integer representation we use in nearly every number system today is...

A. Signed magnitude

B. One’s complement

C. Two’s complement

D. Unsigned binary only

E. (There’s no name for this)

The integer representation we use in nearly every number system today is...

A. Signed magnitude

B. One’s complement

C. Two’s complement

D. Unsigned binary only

E. (There’s no name for this)

In two’s complement, we can store…

A. A larger negative value than positive (-128 to 127)

B. The same range (e.g., -128 to 128)

C. A larger positive value than negative (-127 to 128)

In two’s complement, we can store…

A. **A larger negative value than positive (-128 to 127)**

B. The same range (e.g., -128 to 128)

C. A larger positive value than negative (-127 to 128)

If we add two positive operands and get a result that's smaller than either of the operations, ____ has occurred:

A. Numeric inflation
B. Overflow
C. Integer expansion
D. Bidenomics
E. There’s no name for this; it can’t happen

If we add two positive operands and get a result that's smaller than either of the operations, _____ has occurred:

A. Numeric inflation
B. Overflow
C. Integer expansion
D. Bidenomics
E. There’s no name for this; it can’t happen

So far: Unsigned Integers

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

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

$10110 = \underline{0}10110 = \underline{000}10110 = \underline{00000}10110$

So far: Unsigned Integers

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

- `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

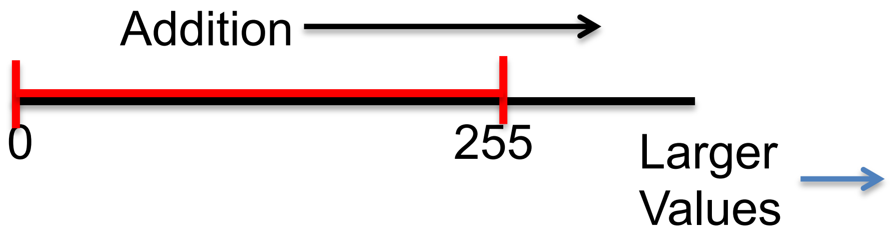
252 = 11111100

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

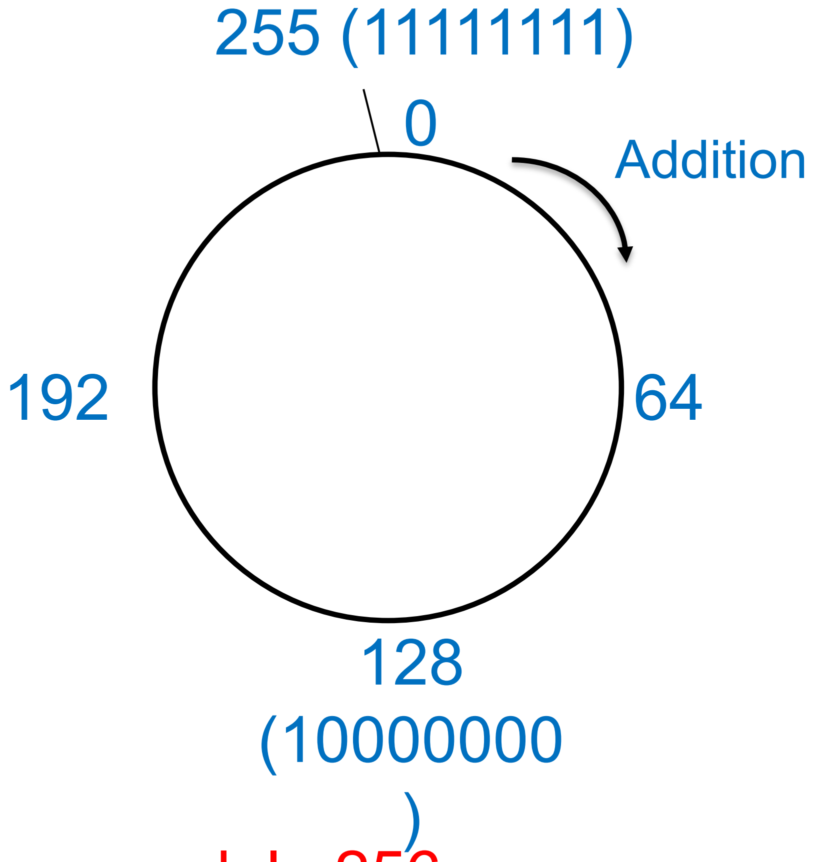
  252 = 11111100

  253 = 11111101

  254 = 11111110

  255 = 11111111

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



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:

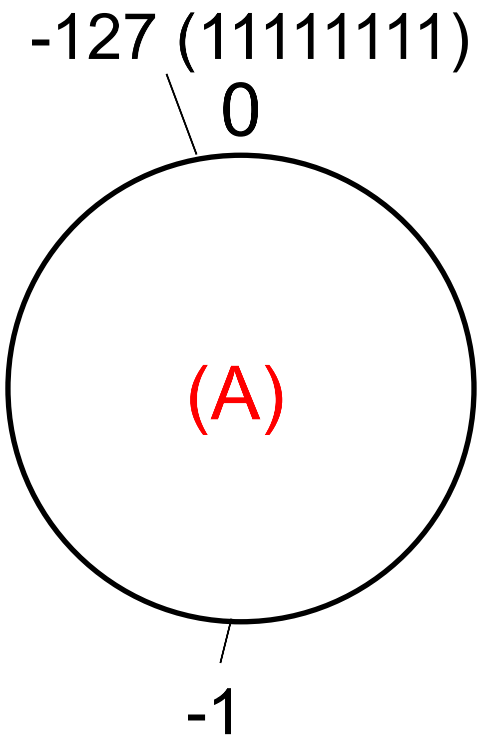
```
  1
  0110    6       1100    12
+ 0100  + 4     + 1010   +10
------  ---     ------   ---
  1010   10     1 0110     6
^no carry out   ^carry out
```

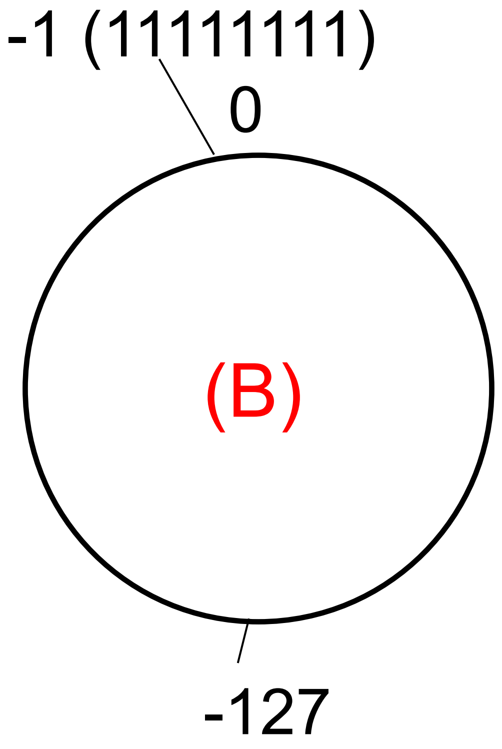
Four bits give us range: 0 - 15

Overflow!

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

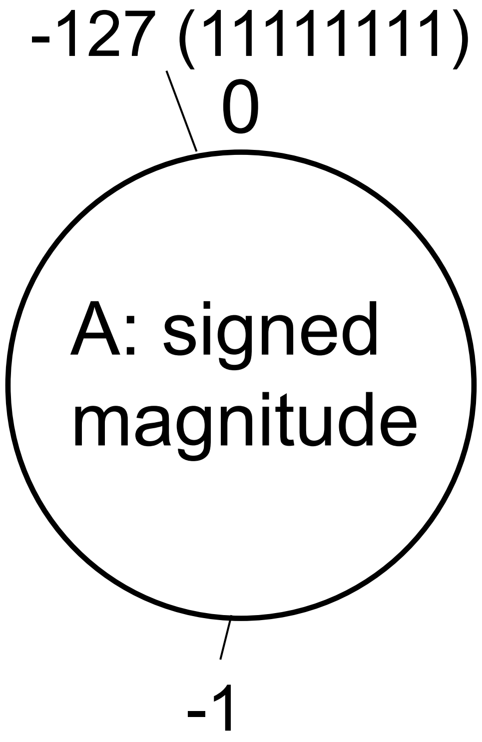
Suppose we want to support signed values (positive **and** negative) in 8 bits, where should we put -1 and -127 on the circle? Why?





(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?

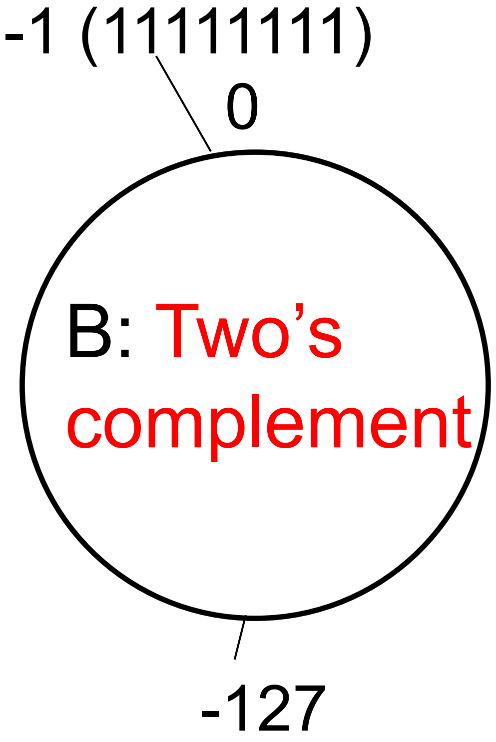


-127 (11111111)

0

A: signed magnitude

-1



-1 (11111111)

0

B: Two's complement

-127

C: Put them somewhere else.

NOT USED: Signed Magnitude Representation (for 4 bit values)

This is not what
we do in present
day systems

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

For one byte:

1 = 00000001
-1 = 10000001

Pros: Negation is very simple!

For one byte:

0 = 00000000
-0 = 10000000 ????

Major con: Two ways to represent zero!

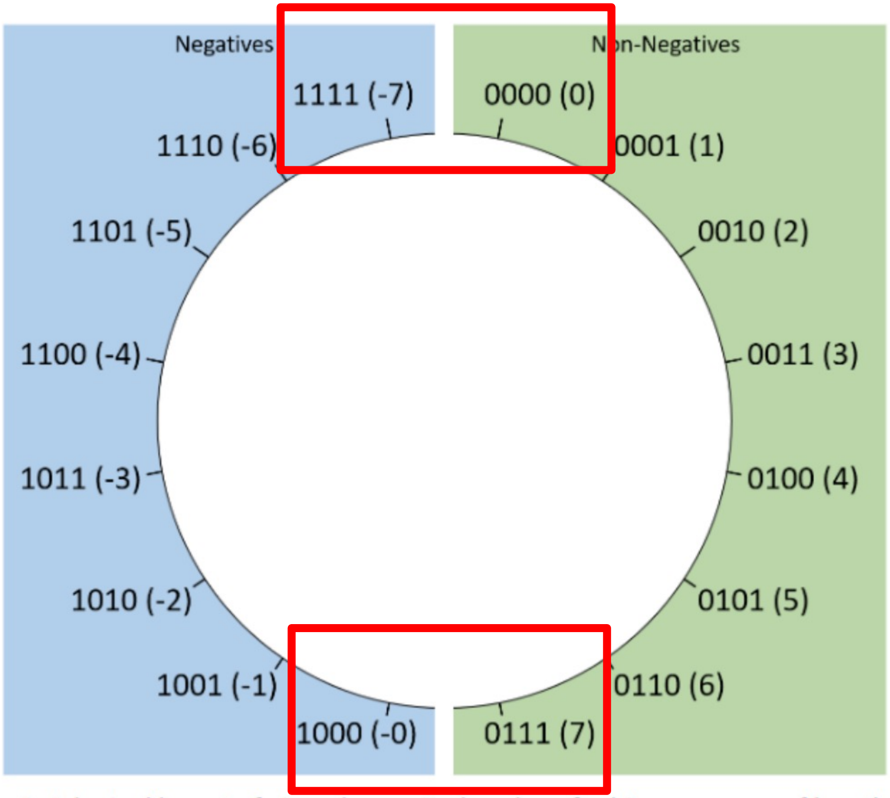


Figure 1. A logical layout of signed magnitude values for bit sequences of length four.

Two’s Complement Representation (for four bit values)

Used
Today

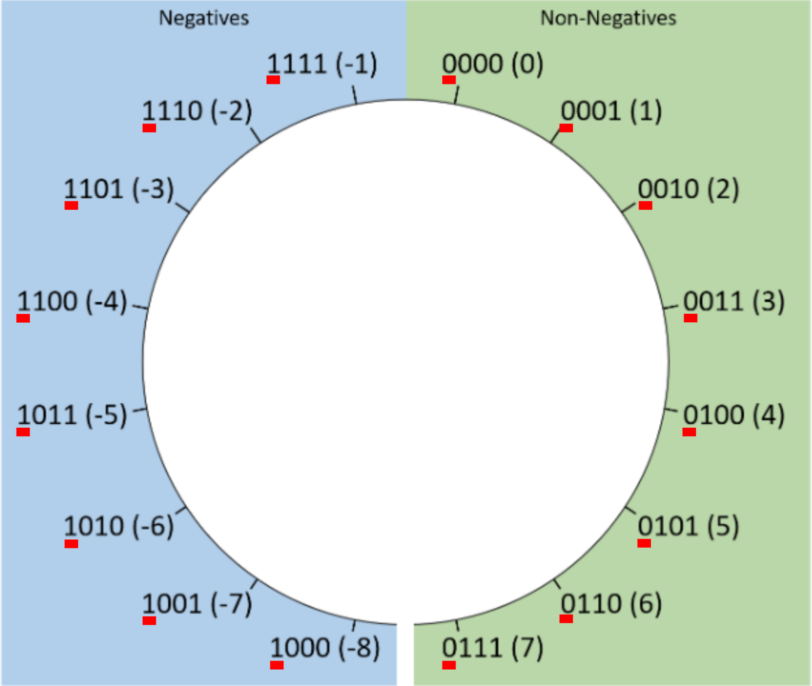
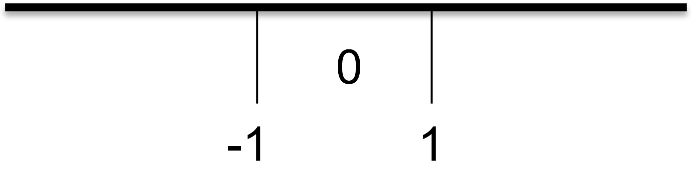


Figure 2. A logical layout of two’s complement values for bit sequences of length four.

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.

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 (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:

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

Note the negative sign 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 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…”

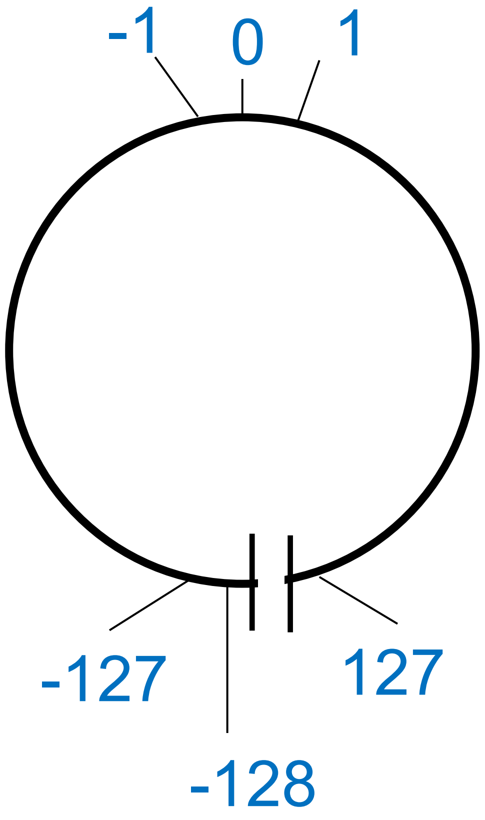
- What is the decimal value of 1100?

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

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

Two’s Complement Negation

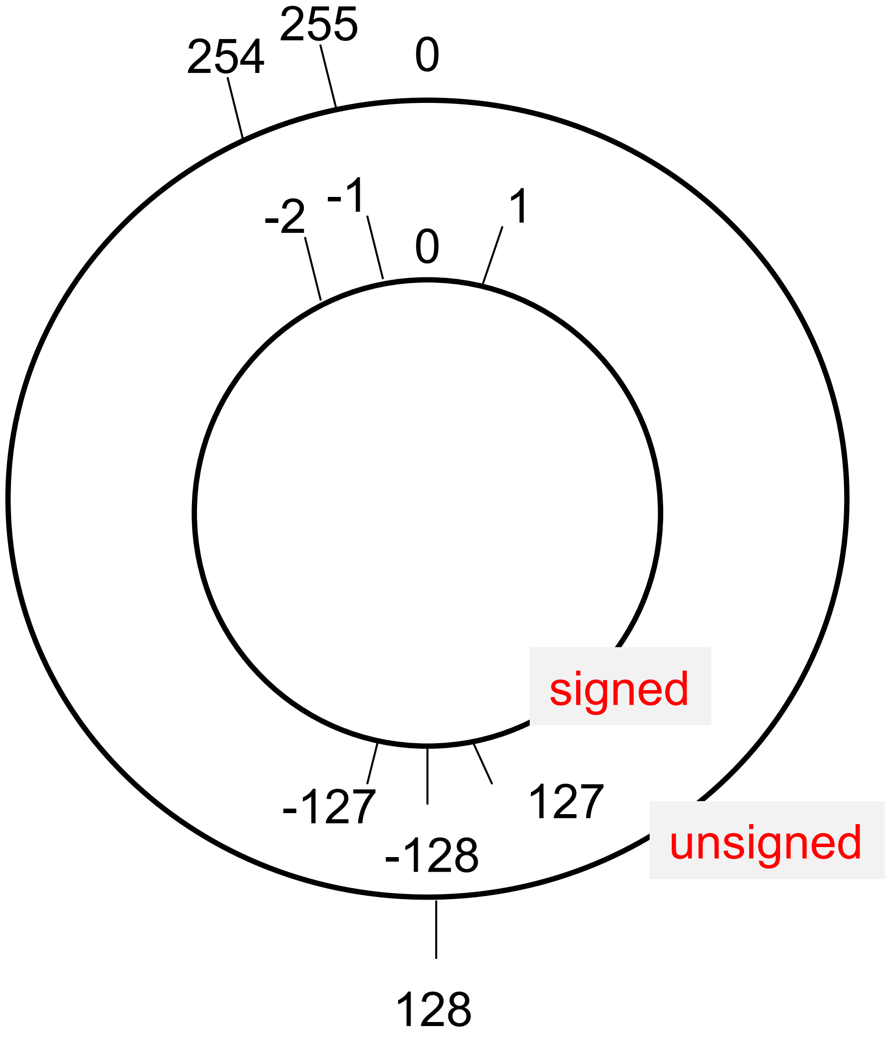
- To negate a value x, we want to find y such that x + y = 0.
- For N bits, $y = 2^N - x$



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

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, add 1, 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…

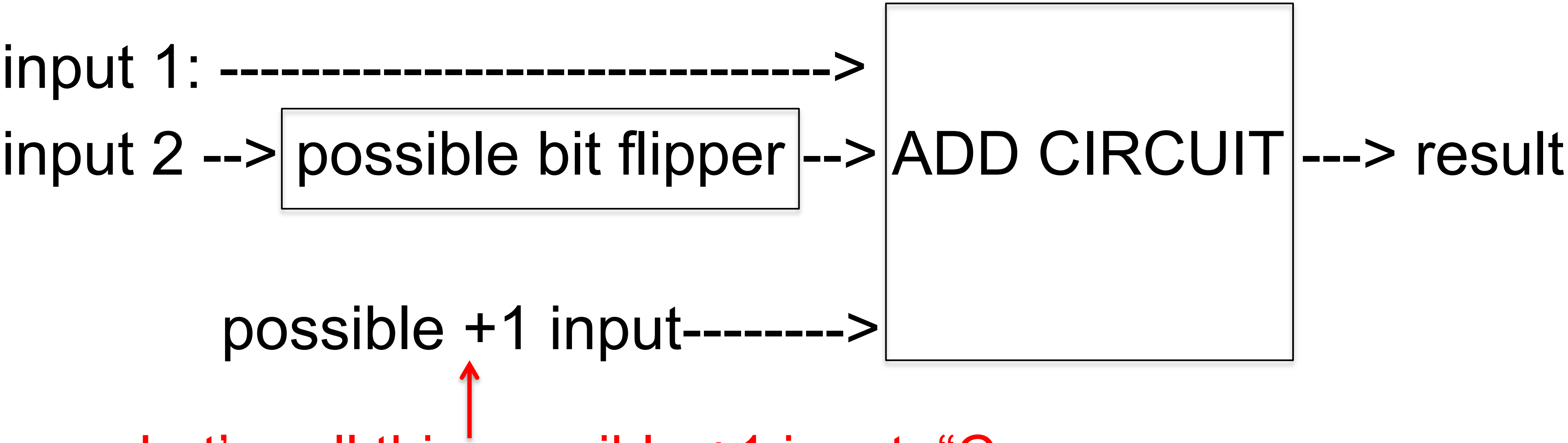
- 6 - 7 = 6 + (-7) = 6 + (~7 + 1)
  - ~7 is shorthand for "flip the bits of 7"

Subtraction Hardware

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 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
```

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

A. Yes, it’s gone

B. Nope, still here

C. It’s even worse 🙀

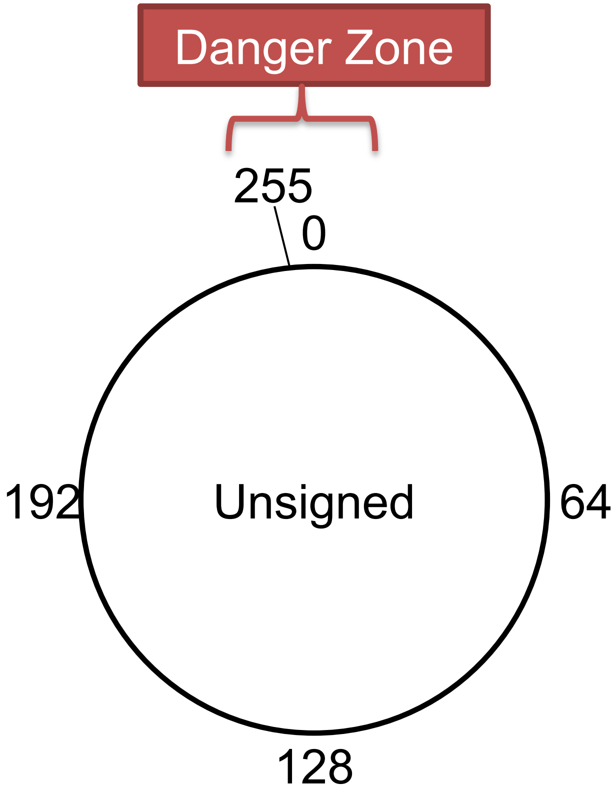
By switching to two's complement, have we solved this value "rolling over" (overflow) problem?

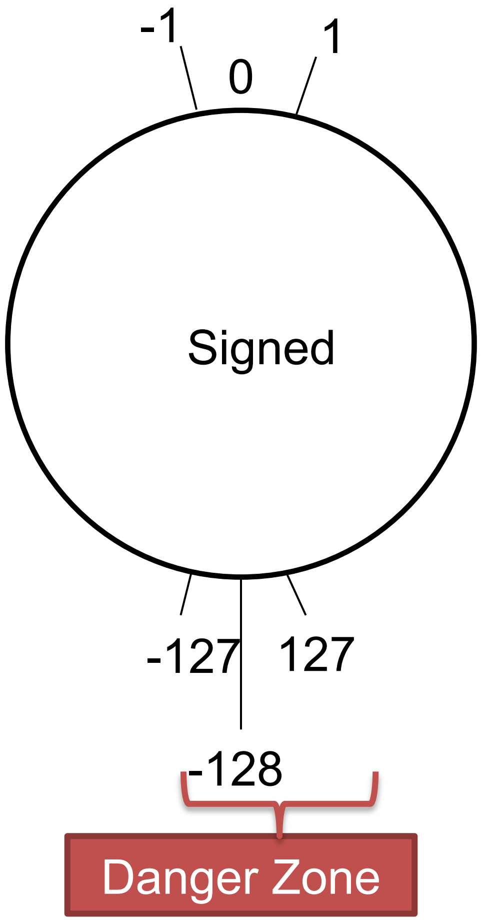
A. Yes, it’s gone

B. Nope, still here

C. It’s even worse 🙀

Overflow, Revisited



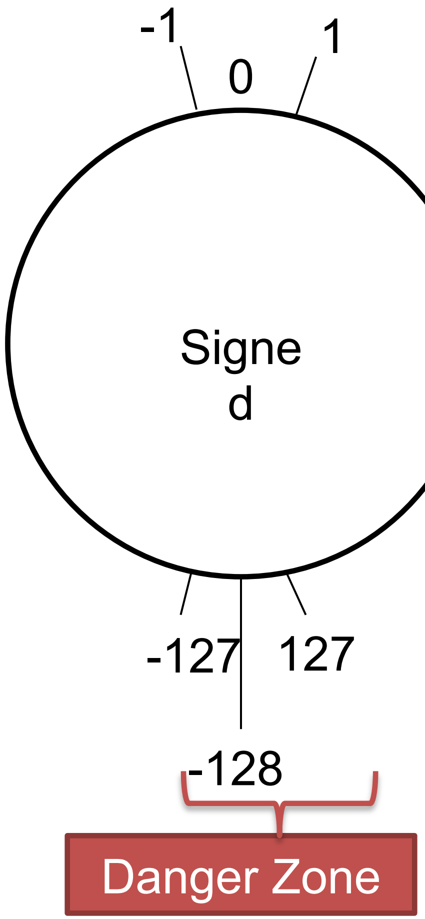


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never

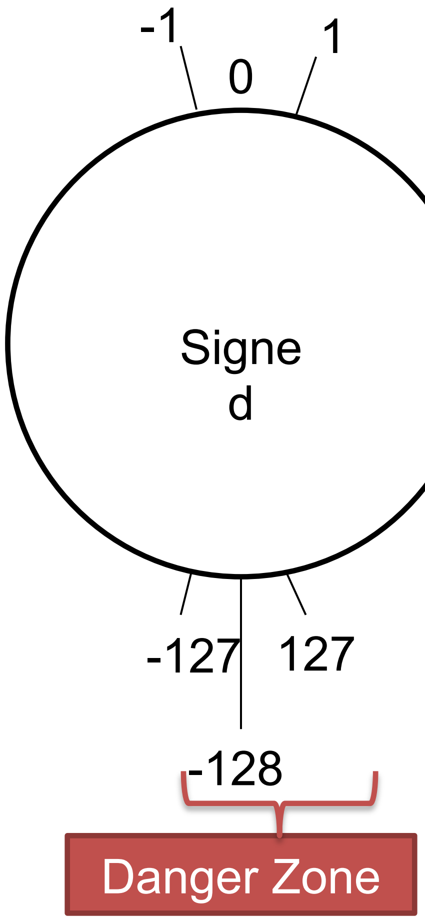


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never



So what did we just talk about…?

Your TODO List

- **HW1, Lab 1**
- **The next 12 weeks:** Read the readings before class

Two's Complement Overflow For Addition

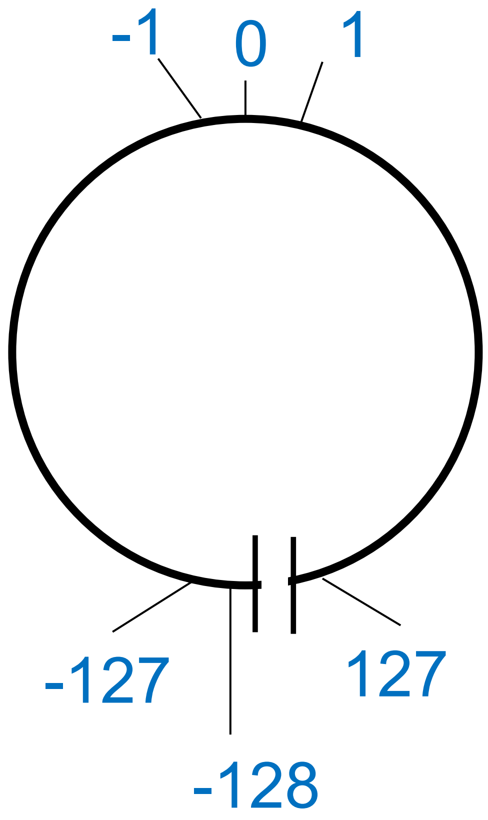
- **Addition Overflow**: IFF the sign bits of operands are the same, but the sign bit of result is different.
- Not enough bits to store result!

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

of result

sign of operands ≠ sign

no overflow

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

overflow

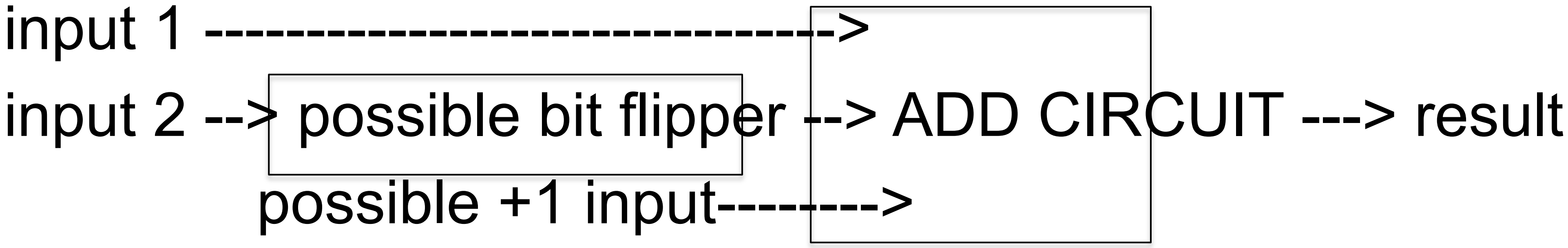
```
 4+7=11    -6-8=-14
 0100         1010
+0111        +1000
 1011       1  0010
```

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)

$$9 + 11 = 1001 + 1011 + \overset{\substack{\text{carry-in} \\ \downarrow}}{0} = \overset{\substack{\text{carry-out} \\ \downarrow}}{1} \quad 0100$$

$$9 + 6 = 1001 + 0110 + 0 = 0 \quad 1111$$

$$3 + 6 = 0011 + 0110 + 0 = 0 \quad 1001$$

Subtraction (carry-in = 1)

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

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

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)

$$\begin{array}{cccccccccccccc}
 & & & & & & & & \overset{\text{carry-in}}{\downarrow} & & \overset{\text{carry-out}}{\downarrow} & & & \\
9 & + & 11 & = & 1001 & + & 1011 & + & 0 & = & 1 & 0100 & = & 4 \\
9 & + & 6 & = & 1001 & + & 0110 & + & 0 & = & 0 & 1111 & = & 15 \\
3 & + & 6 & = & 0011 & + & 0110 & + & 0 & = & 0 & 1001 & = & 9
\end{array}$$

Subtraction (carry-in = 1)

$$\begin{array}{cccccccccccccc}
6 & - & 3 & = & 0110 & + & \overbrace{1100}^{(-3)} & + & 1 & = & 1 & 0011 & = & 3 \\
3 & - & 6 & = & 0011 & + & \underbrace{1001}_{(-6)} & + & 1 & = & 0 & 1101 & = & 13
\end{array}$$

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

3 − 6 = 0011 + $\underbrace{1001}_{(-6)}$ + 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)

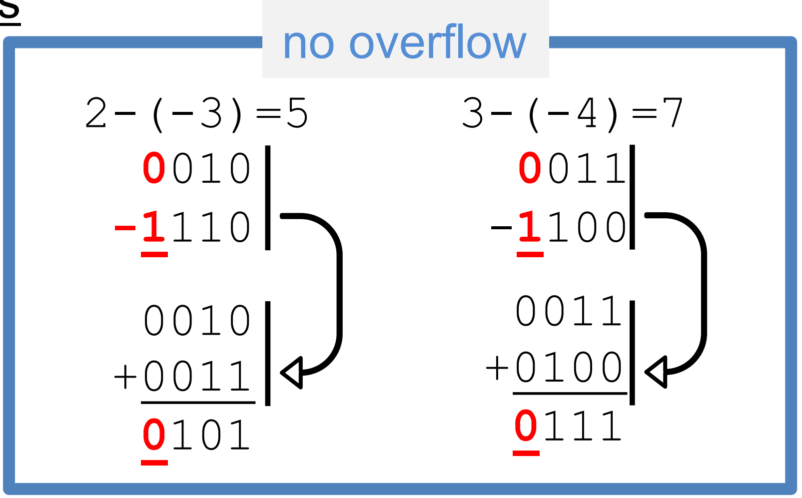
Two’s Complement Overflow For Subtraction

- **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

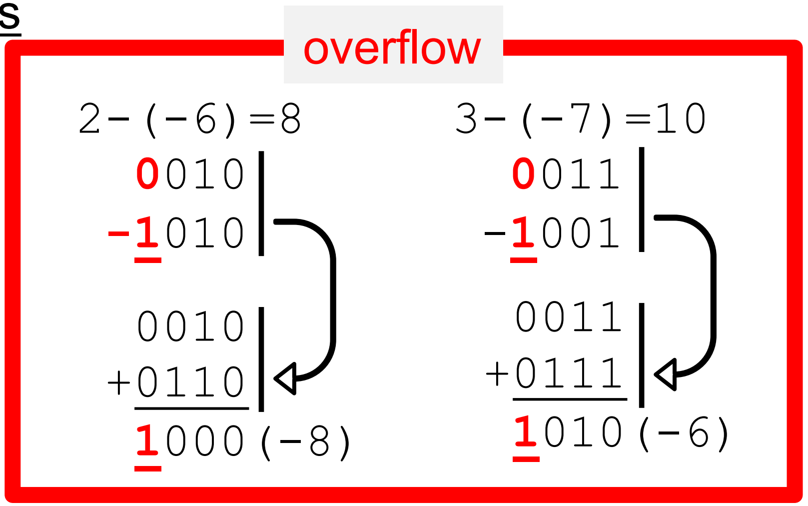


no overflow

```
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)
```

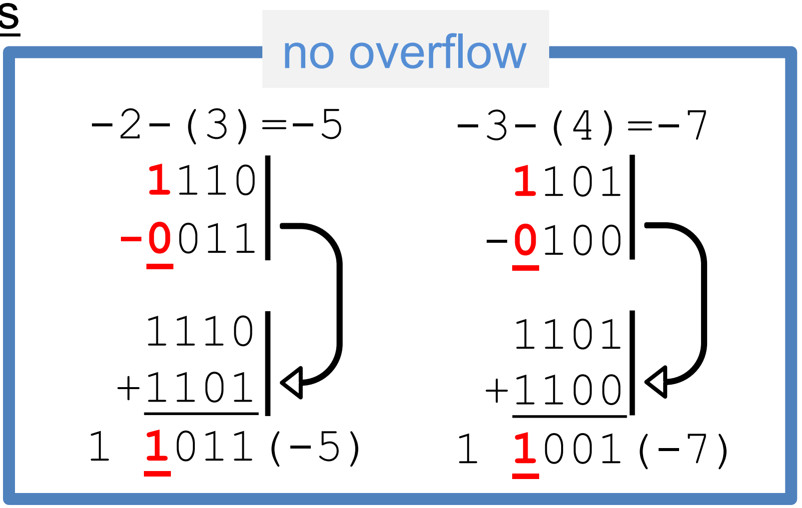
Two’s Complement Overflow For Subtraction

- **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

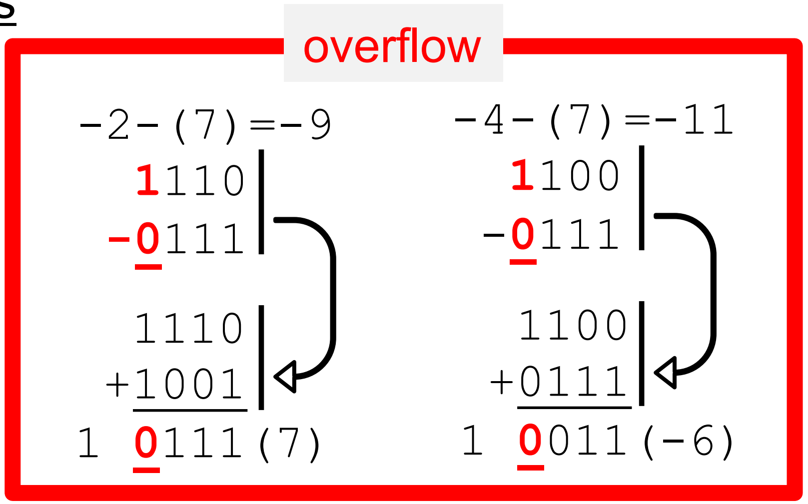


no overflow

```
-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

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).