

Data III & Integers I

CSE 351 Winter 2023

Instructor:

Sam Wolfson

Teaching Assistants:

Aman Mohammed

Angela Xu

Armin Magness

Clare Edmonds

David Dai

Jenny Peng

Maggie Jiang

Mara Kirdani-Ryan

Nayha Auradkar

Yoonseo Song



Kyle 
@KylePlantEmoji

Me: I'm so sorry, my dog ate my homework

Comp Sci Professor: your dog ate your coding assignment?

Me:

Prof:

Me: it took him a couple bytes

10:38 PM · 4/8/19 · [Twitter for Android](#)

6,433 Retweets 65.1K Likes



Tweet your reply



Relevant Course Information

- ❖ Lab 0 due tonight
- ❖ hw2 due Wednesday, hw3 due Friday
- ❖ Lab 1a released
 - Workflow:
 - 1) Edit `pointer.c`
 - 2) Run the Makefile (`make clean` followed by `make`) and check for compiler errors & warnings
 - 3) Run `pctest (. /pctest)` and check for correct behavior
 - 4) Run rule/syntax checker (`python3 dlc.py`) and check output
 - Due Wednesday 1/18, will overlap a bit with Lab 1b
 - We grade just your *last* submission
 - Don't wait until the last minute to submit – the auto-grader may take a few minutes to run, and will identify basic issues

Lab Synthesis Questions

- ❖ All subsequent labs (after Lab 0) have a “synthesis question” portion
 - Can be found on the lab specs and are intended to be done *after* you finish the lab
 - You will type up your responses in a `.txt` file for submission on Gradescope
 - These will be graded “by hand” (read by TAs)
- ❖ Intended to check your understanding of what you should have learned from the lab
 - Also great practice for short answer questions

Reading Review

❖ Terminology:

- Bitwise operators (&, |, ^, ~)
- Logical operators (&&, ||, !)
- Short-circuit evaluation
- Unsigned integers
- Signed integers (Two's Complement)

❖ Questions from the Reading?

Review Questions

❖ Compute the result of the following expressions for
char c = 0x81; = 0b1000 0001

■ c ^ c = 0x0

■ ~c & 0xA9 0b0111 1110 ⇒ 0x28

■ c || 0x80 & 0b1010 1001
1000

■ !!c
↓ 0x1 ↓ 0x1

❖ Compute the value of signed char sc = 0xF0;
(Two's Complement)

0b1111 0000
2⁷ 2⁰

$$-(2^7) + 2^6 + 2^5 + 2^4 = -16$$

Bitmasks

- Typically, binary bitwise operators ($\&$, $|$, \wedge) are used with one operand being the “input” and other operand being a specially-chosen **bitmask** (or *mask*) that performs a desired operation

- Operations for a bit b (answer with 0, 1, b , or \bar{b}):

$$b \& 0 = \underline{0}$$

$$b \& 1 = \underline{b}$$

$$b | 0 = \underline{b}$$

$$b | 1 = \underline{1}$$

$$b \wedge 0 = \underline{b}$$

$$b \wedge 1 = \underline{\bar{b}}$$

$$\begin{array}{r|l} 0 & 0 \\ 1 & 1 \end{array}$$

↙ "not b"

Bitmasks

- ❖ Typically, binary bitwise operators ($\&$, $|$, \wedge) are used with one operand being the “input” and other operand being a specially-chosen **bitmask** (or *mask*) that performs a desired operation
- ❖ Example: $b|0 = b$, $b|1 = 1$

\rightarrow	01010101	\leftarrow input
	11110000	\leftarrow bitmask
<hr/>		
	11110101	

Numerical Encoding Design Example

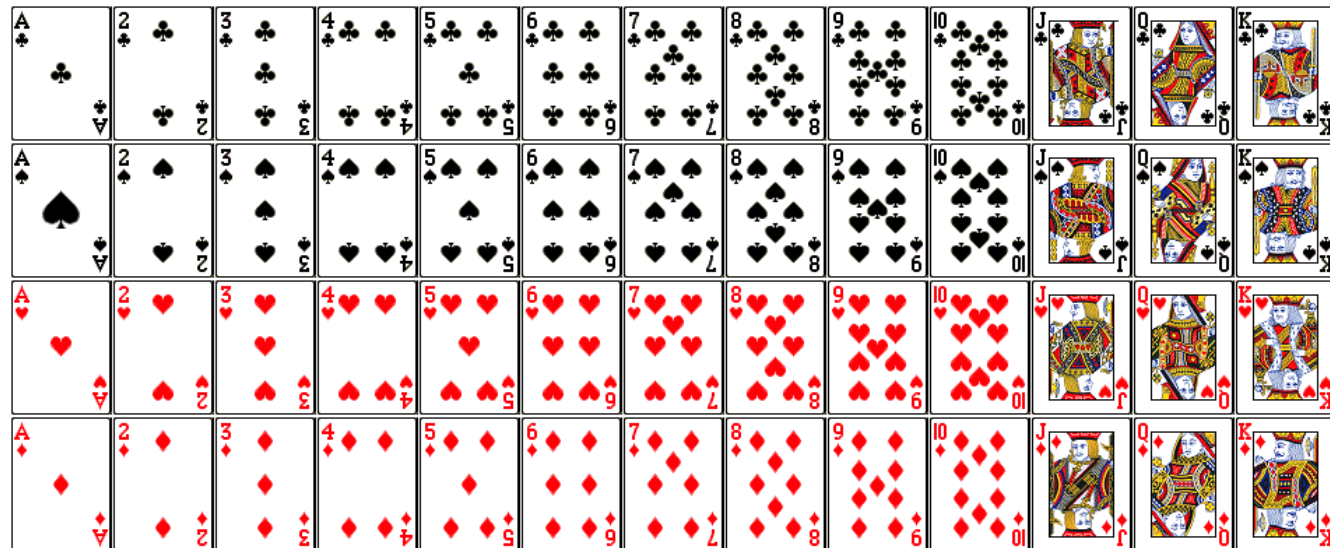
- ❖ Encode a standard deck of playing cards

- 52 cards in 4 suits

- ❖ Operations to implement:

- Which is the higher value card?
 - Are they the same suit?

13 cards / suit



Representations and Fields

1) Binary encoding of all 52 cards – only 6 bits needed

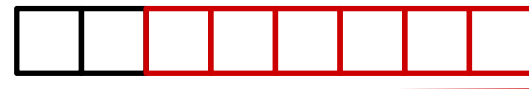
- $2^6 = 64 \geq 52$



low-order 6 bits of a byte

- Fits in one byte (smaller than one-hot encodings)
- How can we make value and suit comparisons easier?





4) Separate binary encodings of suit (2 bits) and value (4 bits)



suit value

- Also fits in one byte, and easy to do comparisons

K	Q	J	...	3	2	A
1101	1100	1011	...	0011	0010	0001

	00
	01
	10
	11

Compare Card Suits

mask: a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector v .
Here we turn all *but* the bits of interest in v to 0.

```
char hand[5];           // represents a 5-card hand
char card1, card2;      // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuitP(card1, card2) ) { ... }
```

```
#define SUIT_MASK 0x30
```

```
int sameSuitP(char card1, char card2) {
    return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

returns **int**

SUIT_MASK = 0x30 =

0	0	1	1	0	0	0	0
---	---	---	---	---	---	---	---

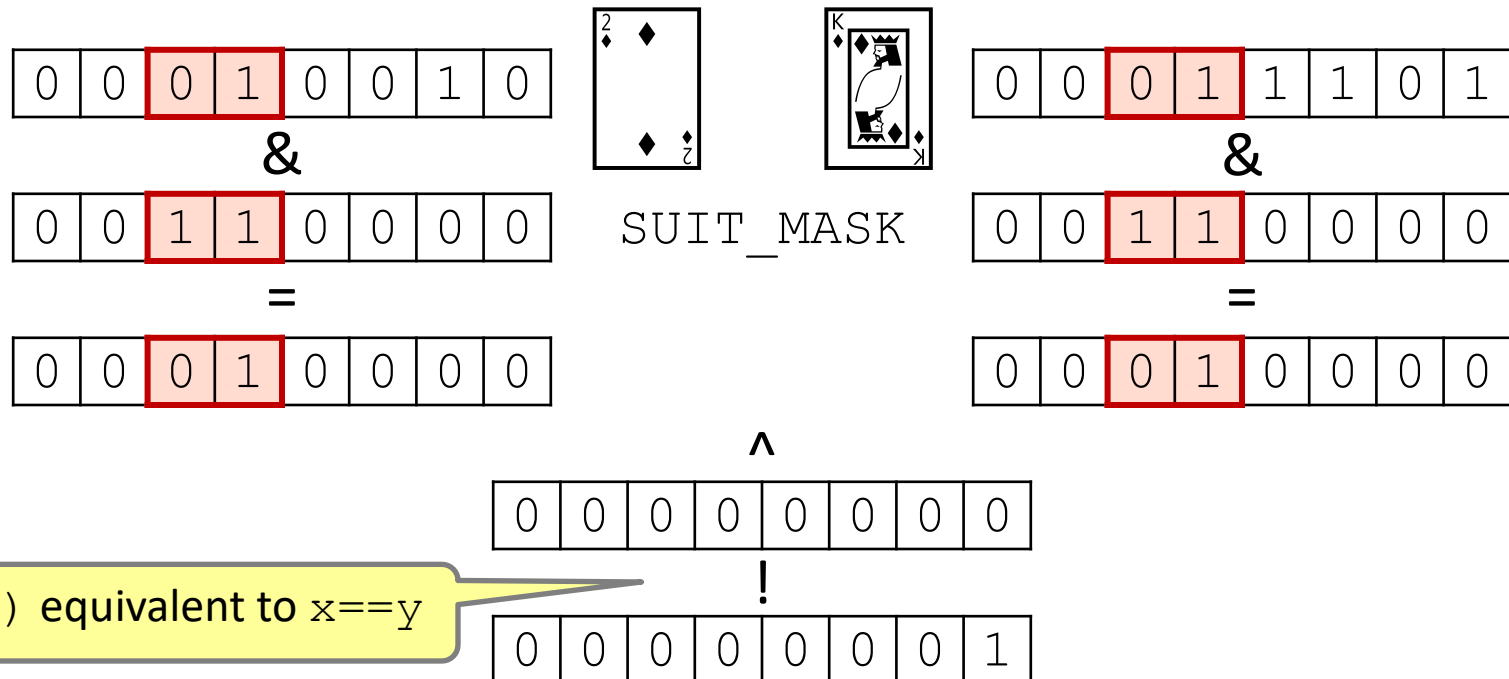
suit value

equivalent

Compare Card Suits

```
#define SUIT_MASK 0x30
```

```
int sameSuitP(char card1, char card2) {
    return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```



Compare Card Values

```
char hand[5];           // represents a 5-card hand
char card1, card2;      // two cards to compare
card1 = hand[0];
card2 = hand[1];

...

if ( greaterValue(card1, card2) ) { ... }
```

```
#define VALUE_MASK  0x0F
```

```
int greaterValue(char card1, char card2) {
    return ((unsigned int) (card1 & VALUE_MASK) >
            (unsigned int) (card2 & VALUE_MASK));
}
```

VALUE_MASK = 0x0F =

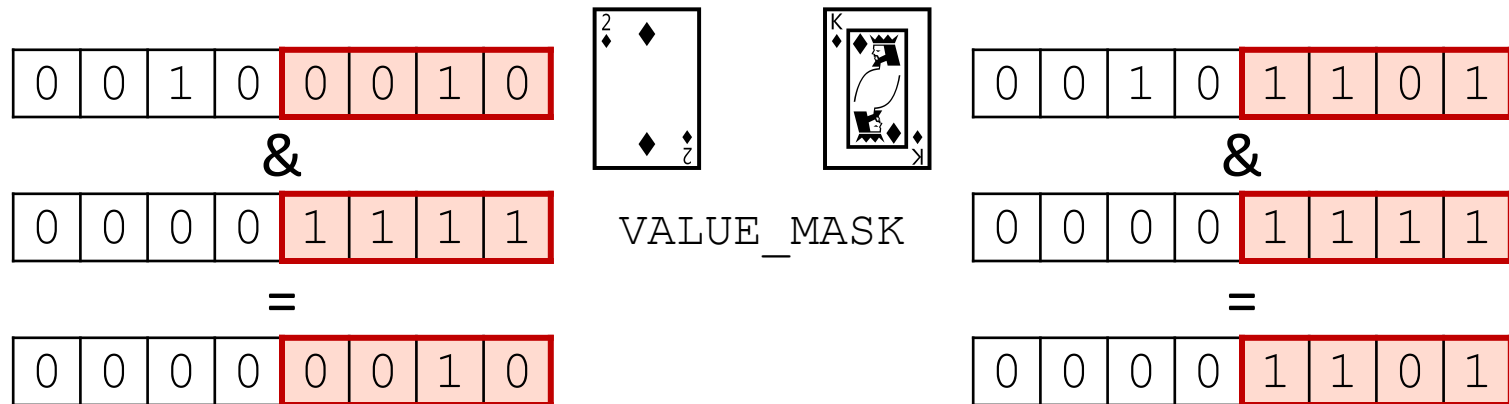
0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

suit value

Compare Card Values

```
#define VALUE_MASK 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
            (unsigned int)(card2 & VALUE_MASK));
}
```



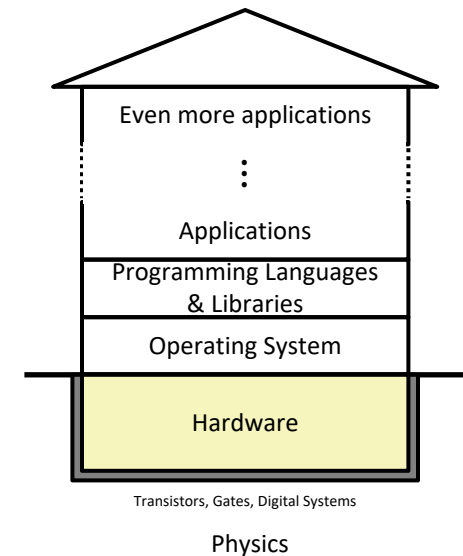
$2_{10} > 13_{10}$

0 (false)

The Hardware/Software Interface

❖ Topic Group 1: **Data**

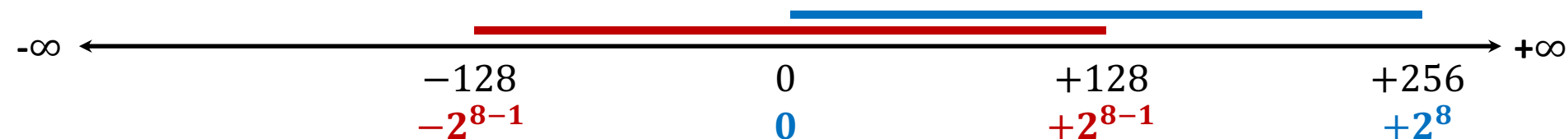
- Memory, Data, **Integers**, Floating Point, Arrays, Structs




- ❖ How do we store information for other parts of the house of computing to access?
 - How do we represent data and what limitations exist?
 - What design decisions and priorities went into these encodings?

Encoding Integers

- ❖ The hardware (and C) supports two flavors of integers
 - *unsigned* – only the non-negatives
 - *signed* – both negatives and non-negatives
- ❖ Cannot represent all integers with w bits
 - Only 2^w distinct bit patterns
 - Unsigned values: $0 \dots 2^w - 1$
 - Signed values: $-2^{w-1} \dots 2^{w-1} - 1$
- ❖ **Example:** 8-bit integers (*e.g.*, `char`)



Unsigned Integers (Review)

- ❖ Unsigned values follow the standard base 2 system
 - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + \dots + b_12^1 + b_02^0$
- ❖ Useful formula: $2^{N-1} + 2^{N-2} + \dots + 2 + 1 = 2^N - 1$
 - *i.e.*, N ones in a row = $2^N - 1$
 - *e.g.*, 0b00111111 = 63

- ❖ But what if I want to store negative numbers?

Sign and Magnitude



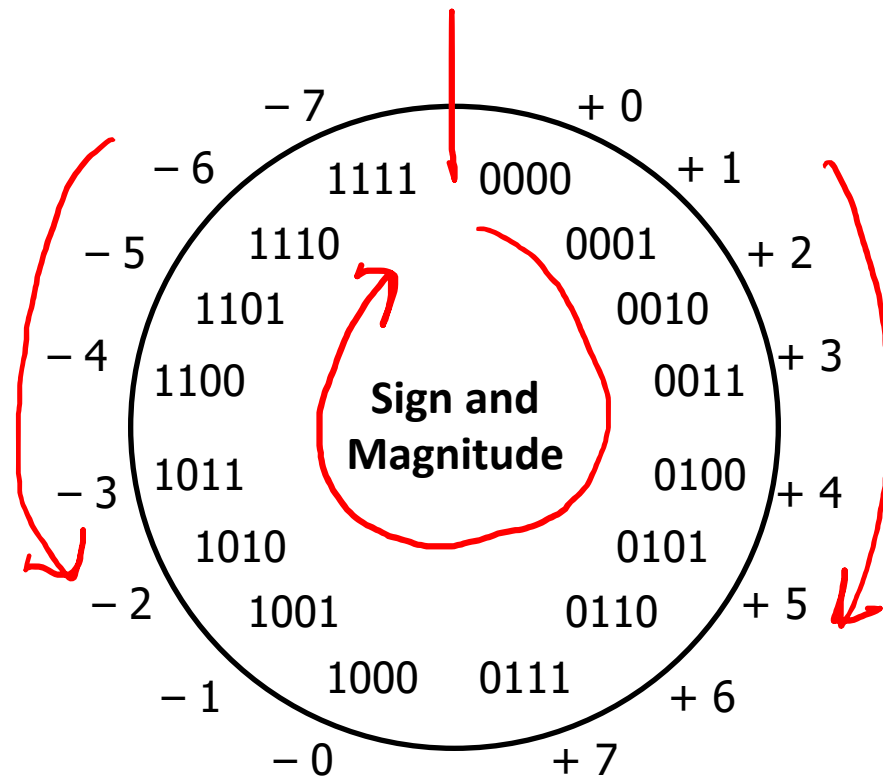
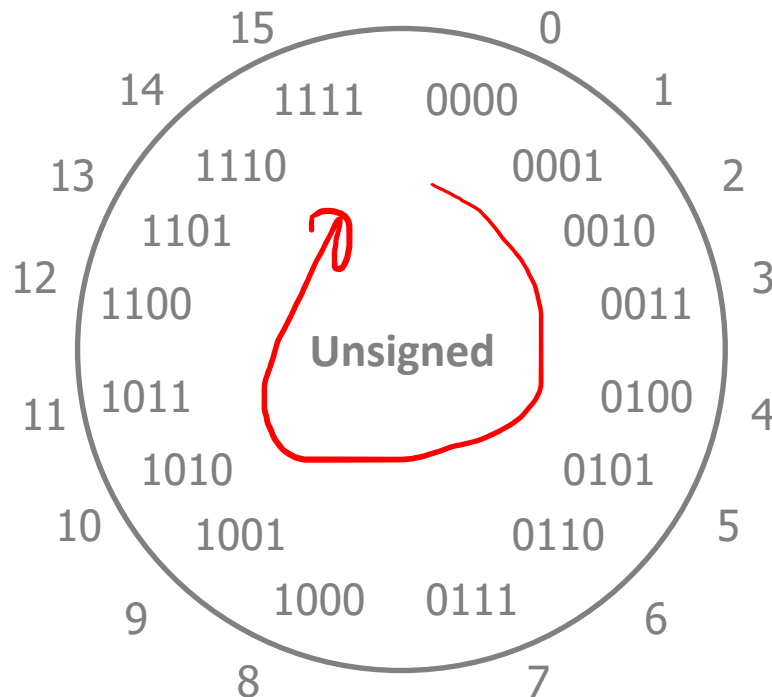
Not used in practice
for integers!

- ❖ Designate the high-order bit (MSB) as the “sign bit”
 - $\text{sign}=0$: positive numbers; $\text{sign}=1$: negative numbers
- ❖ Benefits:
 - Using MSB as sign bit matches positive numbers with unsigned
 - All zeros encoding is still = 0
- ❖ Examples (8 bits):
 - $0x00 = 00000000_2$ is non-negative, because the sign bit is 0
 - $0x7F = 01111111_2$ is non-negative ($+127_{10}$)
 - $0x85 = 10000101_2$ is negative (-5_{10})
 - $0x80 = 10000000_2$ is negative... zero???

Sign and Magnitude

Not used in practice
for integers!

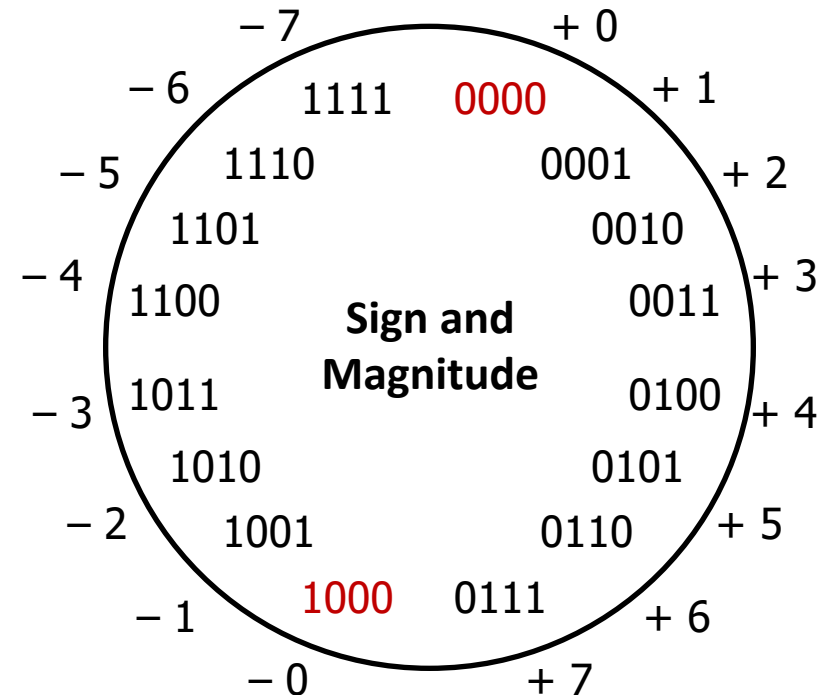
- ❖ MSB is the sign bit, rest of the bits are magnitude
- ❖ Drawbacks?



Sign and Magnitude

Not used in practice
for integers!

- ❖ MSB is the sign bit, rest of the bits are magnitude
- ❖ Drawbacks:
 - **Two representations of 0** (bad for checking equality)



Sign and Magnitude

Not used in practice
for integers!

❖ MSB is the sign bit, rest of the bits are magnitude

❖ Drawbacks:

■ Two representations of 0 (bad for checking equality)

■ **Arithmetic is cumbersome**

• Example: $4 - 3 \neq 4 + (-3)$

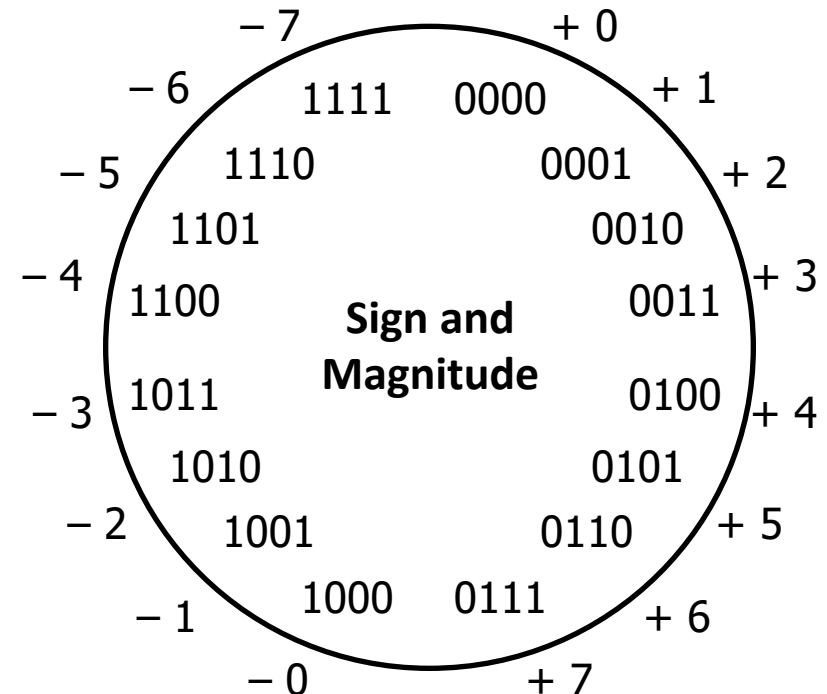
4	0100
- 3	- 0011
<hr/>	
1	0001



4	0100
+ -3	+ 1011
<hr/>	
-7	1111



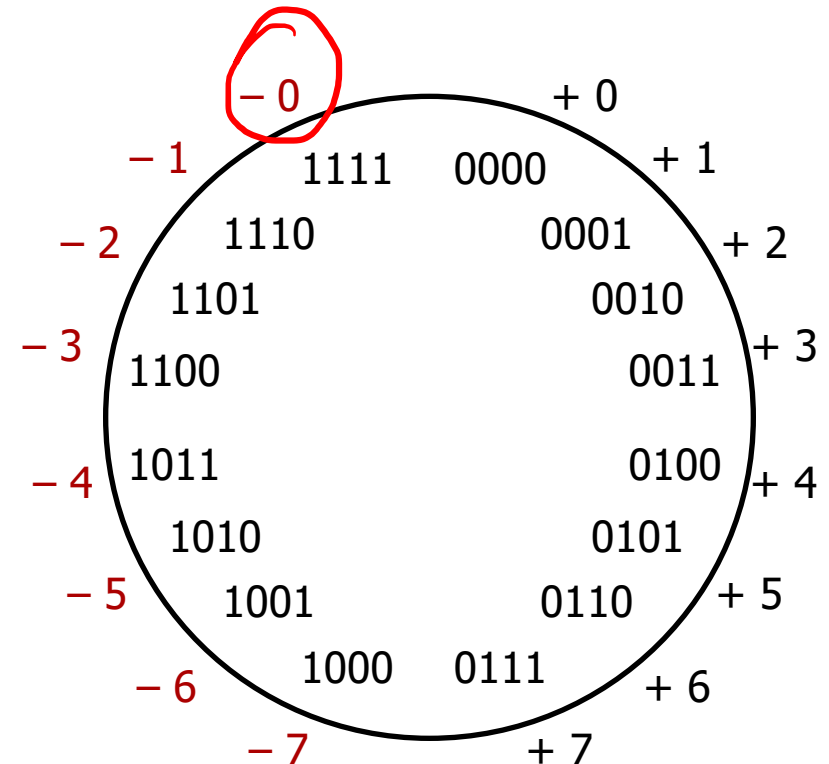
• Negatives “increment” in wrong direction!



Two's Complement

❖ Let's fix these problems:

1) “Flip” negative encodings so incrementing works



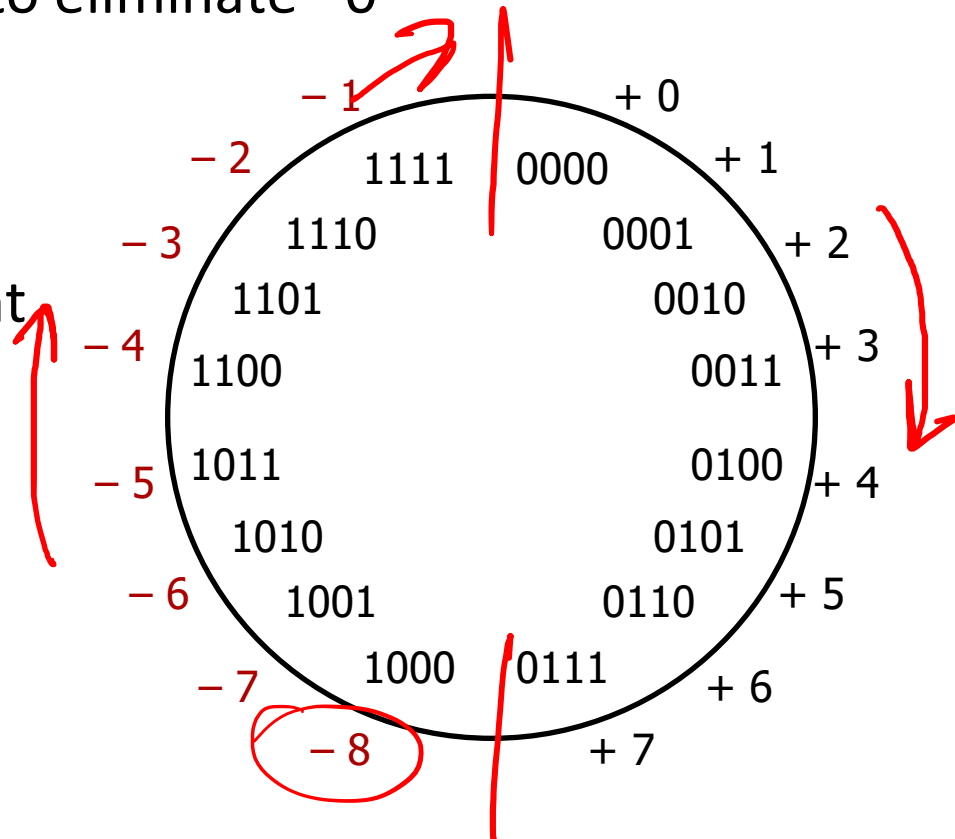
Two's Complement

❖ Let's fix these problems:

- 1) "Flip" negative encodings so incrementing works
- 2) "Shift" negative numbers to eliminate -0

❖ MSB *still* indicates sign!

- This is why we can represent one more negative than positive number (-2^{N-1} to $2^{N-1} - 1$)

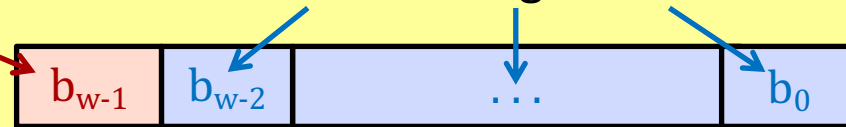


Two's Complement Negatives (Review)

Mathematicians
hate him!

❖ Accomplished with This One Weird Trick!

- b_{w-1} has weight -2^{w-1} , other bits have usual weights $+2^i$



■ 4-bit Examples:

- 1010_2 unsigned:

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

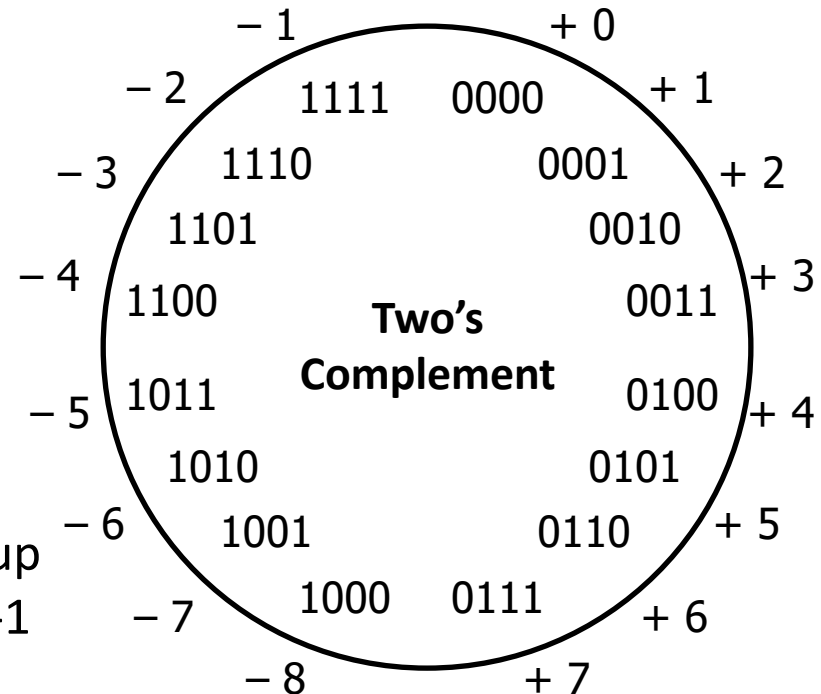
- 1010_2 two's complement:

$$-1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -6$$

■ -1 represented as:

$$1111_2 = -2^3 + (2^3 - 1)$$

- MSB makes it super negative, add up all the other bits to get back up to -1



Polling Question

- ❖ Take the 4-bit number encoding $x = 0b1011$
- ❖ Which of the following numbers is NOT a valid interpretation of x using any of the number representation schemes discussed today?
 - Unsigned, Sign and Magnitude, Two's Complement

A. -4

B. -5 ✓

C. 11 ✓

D. -3 ✓

E. We're lost...

unsigned: $2^3 + 2^1 + 2^0 = 11$
2's comp: $-(2^3) + 2^1 + 2^0 = -5$
sign + mag: $-(2^1 + 2^0) = -3$

Two's Complement is Great (Review)

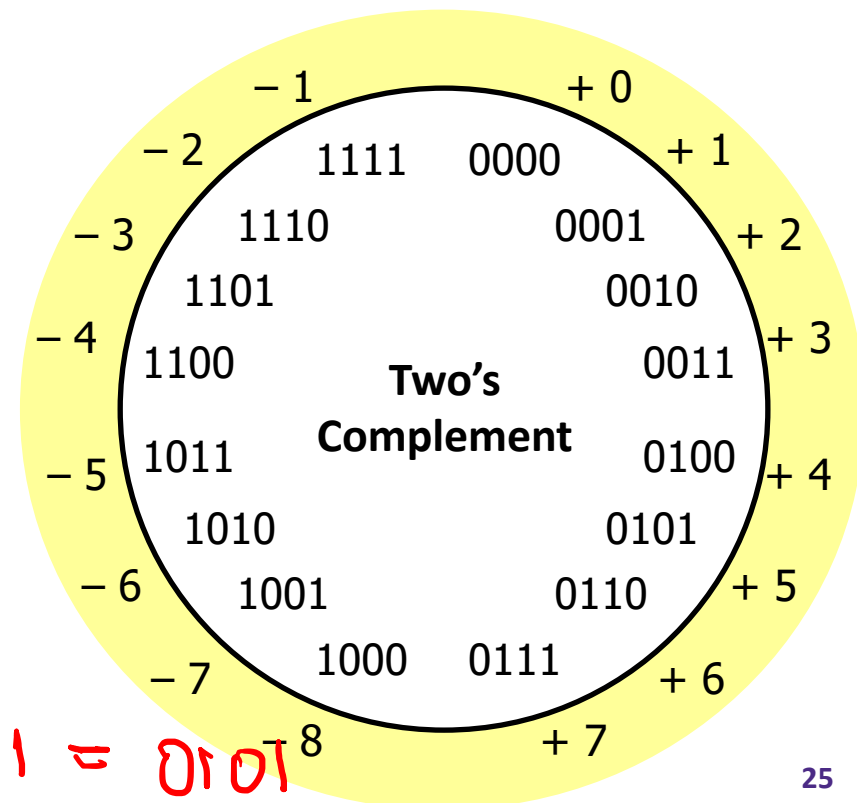
- ✓ Roughly same number of (+) and (−) numbers
- ✓ Positive number encodings match unsigned
- ✓ Single zero
- ✓ All zeros encoding = 0

❖ Simple negation procedure:

- Get negative representation of any integer by taking bitwise complement and then adding one!

$$(\sim x + 1 == -x)$$

$$0b1011 \sim \Rightarrow 0100 + 1 = 0101$$



Summary

- ❖ Bit-level operators allow for fine-grained manipulations of data
 - Bitwise AND (`&`), OR (`|`), and NOT (`~`) different than logical AND (`&&`), OR (`||`), and NOT (`!`)
 - Especially useful with bit masks
- ❖ Choice of *encoding scheme* is important
 - Tradeoffs based on size requirements and desired operations
- ❖ Integers represented using unsigned and two's complement representations
 - Limited by fixed bit width
 - We'll examine arithmetic operations next lecture