Data III & Integers I

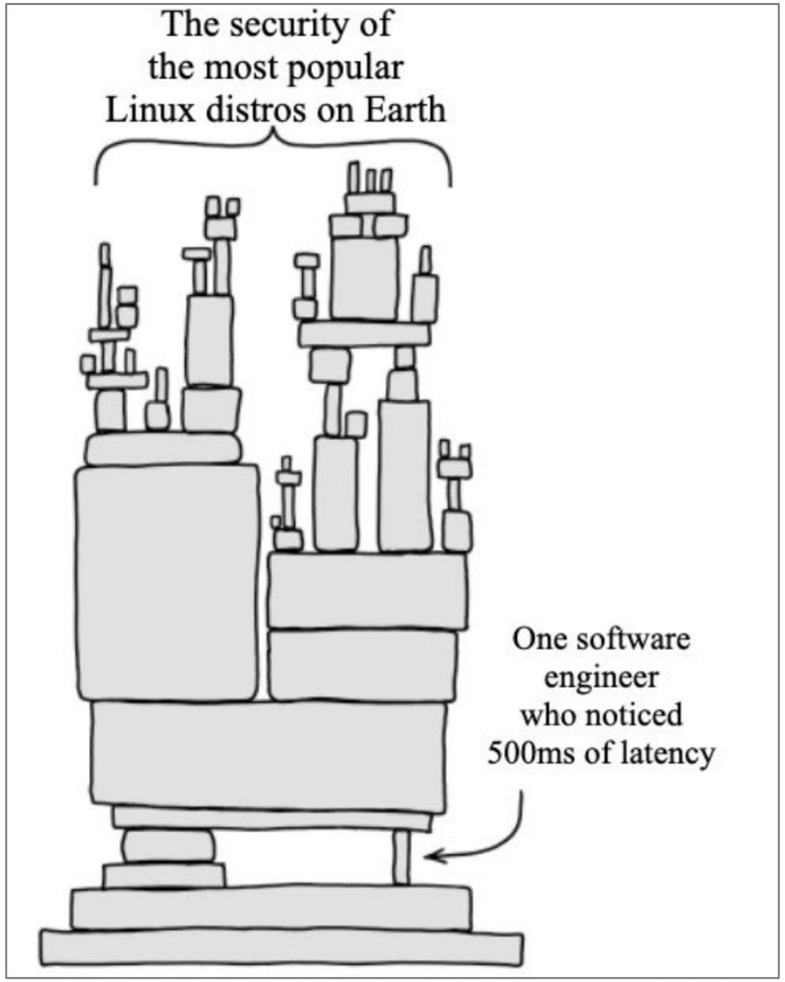
CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson



https://www.openwall.com/lists/oss-security/2024/03/29/4

Announcements, Reminders

- HW2 and Lab 0 due Today! (As was RD4 and LC3, btw.)
- HW3 due Wednesday (03 Apr), HW4 due Friday (05 Apr)
- Elba’s Office Hours in CSE 438
  - Tuesdays 11 AM – 12 PM
  - Wednesdays 4:30 PM – 5:30 PM
- Lab 1a released
  - Some later functions require **bit shifting**, covered in L05

Announcements, Reminders

- Lab 1a released
  - New Workflow:
    1) Edit `pointer.c`
    2) Run the Makefile (**`make clean`** followed by **`make`**) and check for compiler errors & warnings
    3) Run ptest (**`./ptest`**) and check for correct behavior
    4) Run rule/syntax checker (**`python3 dlc.py`**) and check output
  - Due Monday (08 Apr) via Gradescope, will overlap a bit with Lab 1b
    - We grade just your <u>last</u> submission
    - Don't wait until the last minute to submit – need to check autograder output

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 understand of what you should have learned from the lab
  - Also great practice for short answer questions on the exams

Reading Review

- Terminology:
  - Bitwise operators: AND (&), OR (|), XOR (^), NOT(~)
  - Logical operators: AND (&&), OR (||) , NOT (!)
  - Short-circuit evaluation
  - Unsigned integers
  - Signed integers (Two's Complement)

Review Questions

- Compute the result of the following expressions for `char c = 0x81;`
  - `c ^ c` 0x00
  - `~c & 0xA9`
  - `c || 0x80` (true || true) 0x01
  - `!(!c)` 0x01

1000 0001
^ 1000 0001
0000 0000

0111 1110
& 1010 1001
0010 1000

- Compute the value of `signed char sc = 0xF0;` *(using Two's Complement)*

① "flip plus one"

~sc = 0000 1111
\+ 1
0001 0000

⇒ sc = −16

② 1111 0000 (neg!)

$\Leftarrow -2^7 + 2^6 + 2^5 + 2^4 + 0 + \ldots + 0$

Bitmasks

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

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

$b \mathbin{\&} 0 =$ 0 "zero out!"
0 & 0 = 0
1 & 0 = 0

$b \mid 0 =$ b
0 | 0 = 0
1 | 1 = 1

$b \wedge 0 =$ b
0 ^ 0 = 0
1 ^ 0 = 1

$b \mathbin{\&} 1 =$ b
0 & 1 = 0
1 & 1 = 1

$b \mid 1 =$ 1 "all 1's!"
0 | 1 = 1
1 | 1 = 1

$b \wedge 1 =$ $\bar{b}$ "flip!"
0 ^ 1 = 1
1 ^ 1 = 0

Bitmasks

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

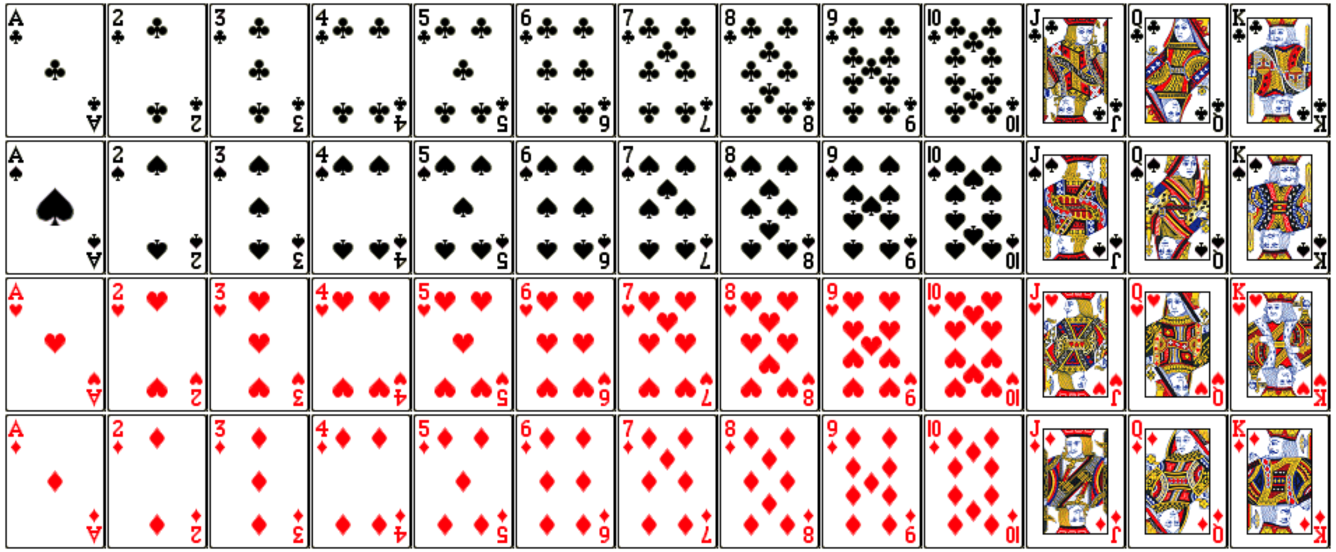
- Example: $b|0 = b, \; b|1 = 1$

```
   01010101  ← input
|  11110000  ← bitmask
   --------
   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?

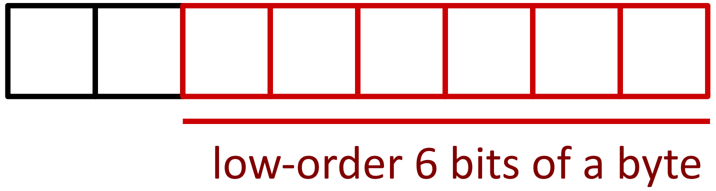
First: How to represent?



Representations and Fields

Binary encoding of all 52 cards – only 6 bits needed

- $2^6 = 64 \geq 52$
- Fits in one byte
- How can we make value and suit comparisons *easier*?

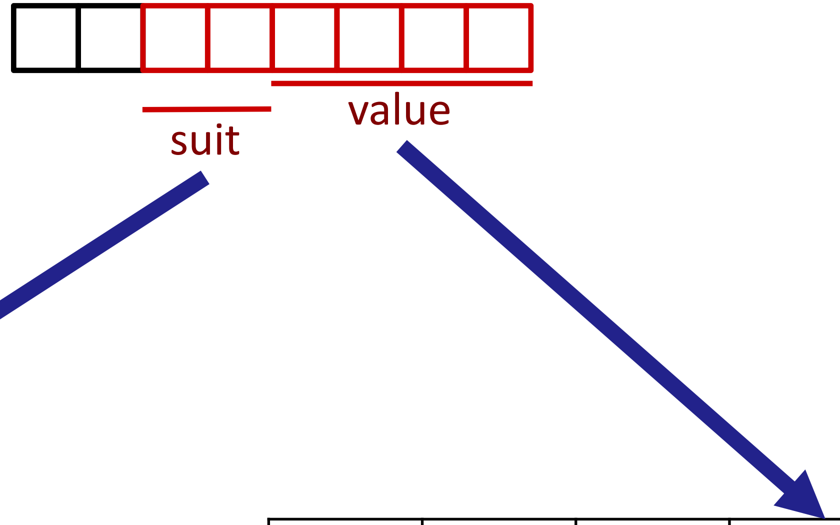


Binary	Suit & Value
000000	Ace of Clubs
000001	Ace of Diamonds
000010	Ace of Hearts
000011	Ace of Spades
...	...
110001	King of Diamonds
110010	King of Hearts
110011	King of Spades

Representations and Fields

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

- **Still** fits in one byte, and easier to do comparisons! 😎



♣	00
♦	01
♥	10
♠	11

K	Q	J	. . .	3	2	A
1101	1011	1010	. . .	0011	0010	0001

Compare Card Suits

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

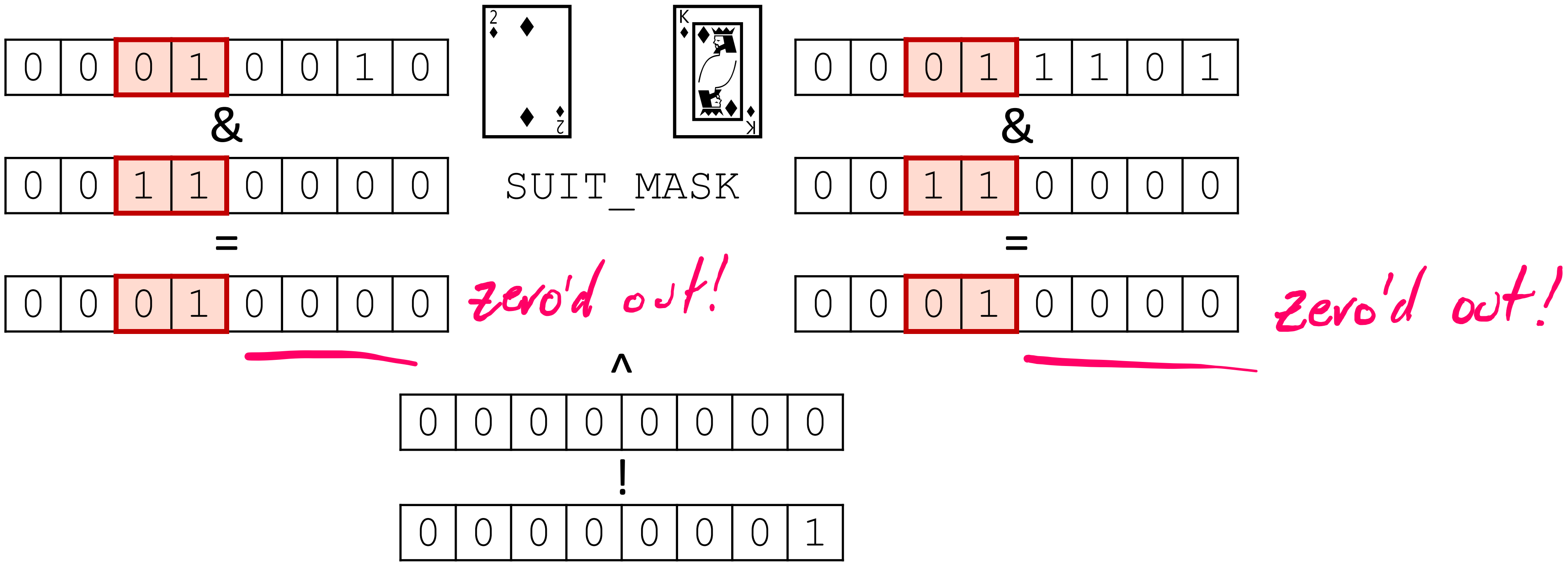
```
#define SUIT_MASK  0x30 // in binary: 0b00110000

int same_suit(char card1, char card2) {
  return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
```

SUIT_MASK = 0x30 =

0	0	1	1	0	0	0	0
		suit	suit	value	value	value	value

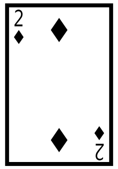
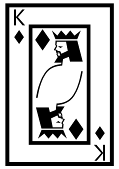
Compare Card Suits

```c
#define SUIT_MASK  0x30

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



Compare Card Suits: Equivalent Technique

```
#define SUIT_MASK  0x30

int same_suit(char card1, char card2) {
  // Equivalent computation
  return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

card1		SUIT_MASK		card2
0 0 0 1 0 0 1 0	&	0 0 1 1 0 0 0 0	&	0 0 0 1 1 1 0 1
= 0 0 0 1 0 0 0 0				= 0 0 0 1 0 0 0 0

==

0 0 0 0 0 0 0 1

x==y equivalent to !(x^y)

Compare Card **Values**

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

```c
#define VALUE_MASK  0x0F

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

0	0	1	0	0	0	1	0

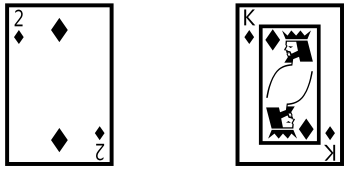
&

0	0	0	0	1	1	1	1

=

0	0	0	0	0	0	1	0

zero'd out!



VALUE_MASK

0	0	1	0	1	1	0	1

&

0	0	0	0	1	1	1	1

=

0	0	0	0	1	1	0	1

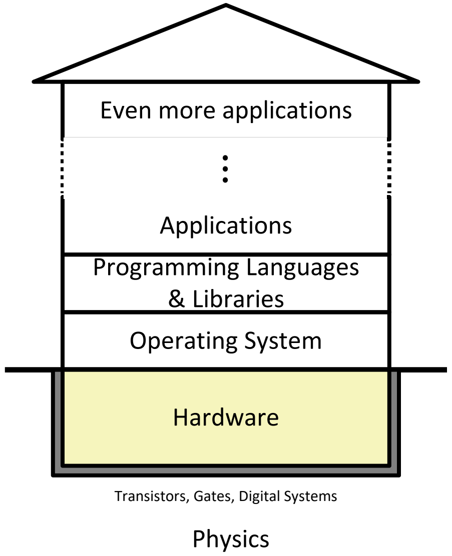
zero'd out!

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

0 (false)

The Hardware/Software Interface

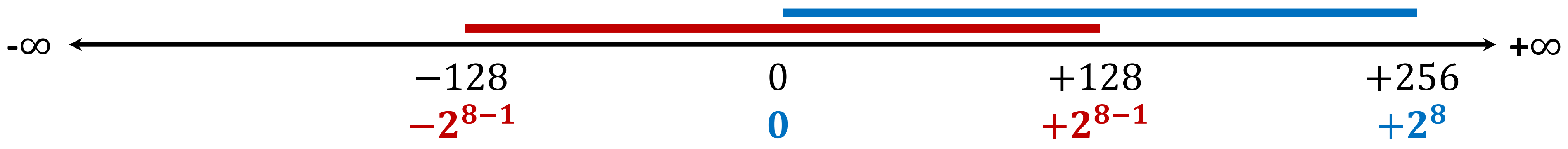
- Topic Group 1: **Data**
  - Memory, Data, **Integers**, Floating Point, Arrays, Structs



- 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
- We cannot represent all integers with w bits!
  - Only 2^w distinct bit patterns
  - Unsigned values: $0 \ldots 2^w - 1$
  - Signed values (2's C): $-2^{w-1} \ldots 2^{w-1} - 1$
- **Example:** 8-bit integers (*e.g.*, `char` in C)



Unsigned Integers (Review)

- Unsigned values follow the standard base 2 system:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$$

Y'all know this well :)

Sign and Magnitude

Not used in practice for integers!

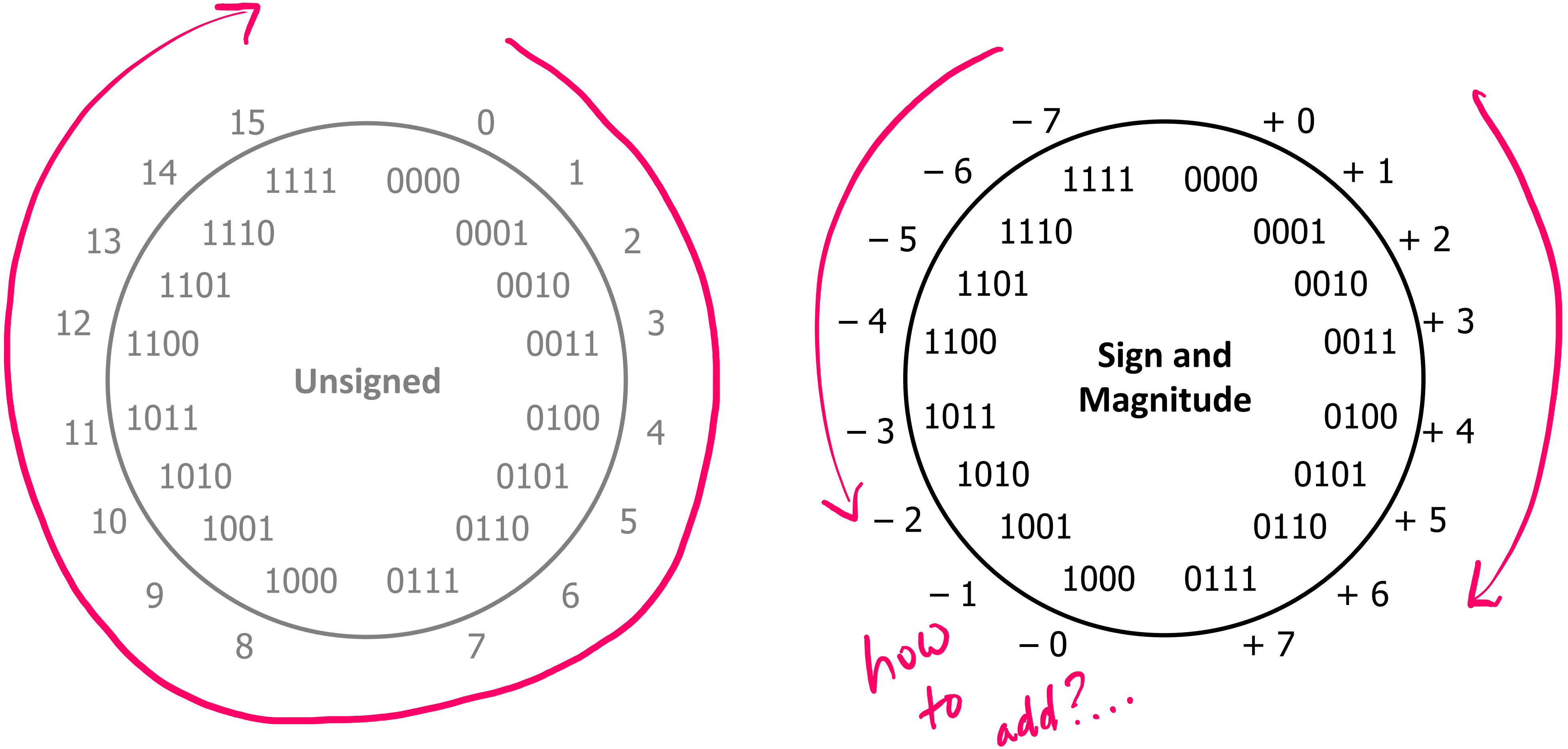
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive numbers; because the sign bit is 0
    - e.g. `0x7F` = `01111111`$_2$ is non-negative (+127$_{10}$)
  - `sign=1`: negative numbers
    - e.g. `0x85` = `10000101`$_2$ is negative (-5$_{10}$)
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned!
  - All zeros encoding is still = 0
- Some Examples (8 bits):
  - `0x00` = `00000000`$_2$ is positive!
  - `0x80` = `10000000`$_2$ is negative!

Oh dear...

Sign and Magnitude

Not used in practice for integers!

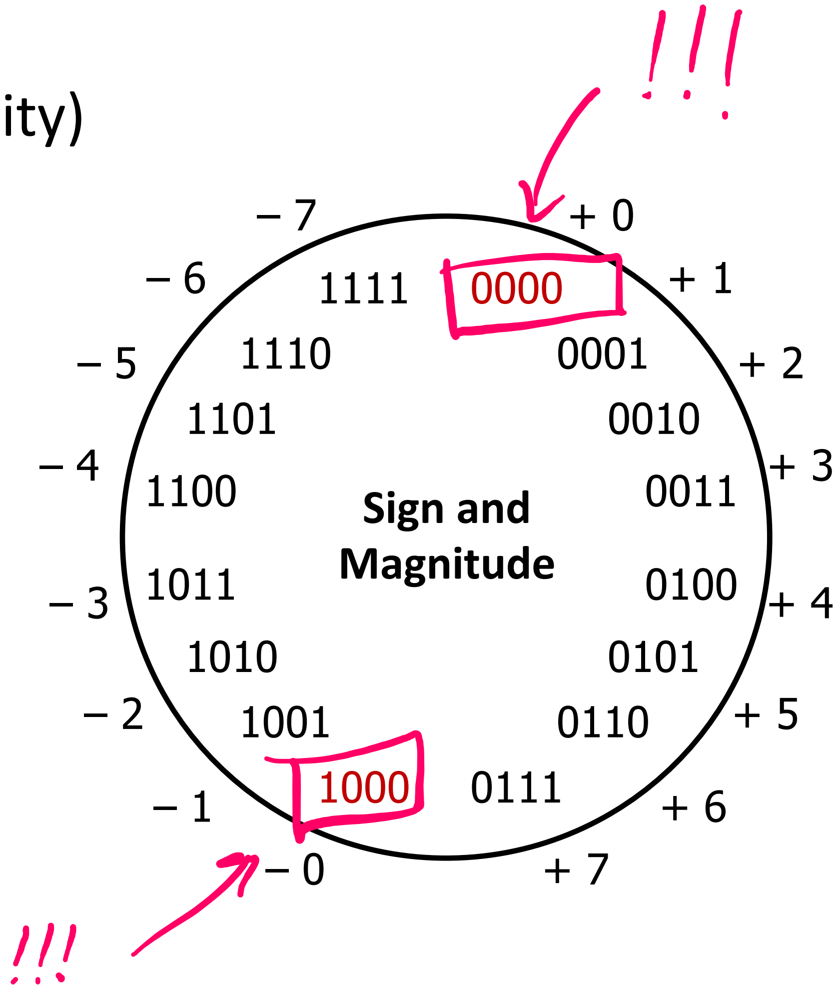
- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?



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 != 4+(-3)`

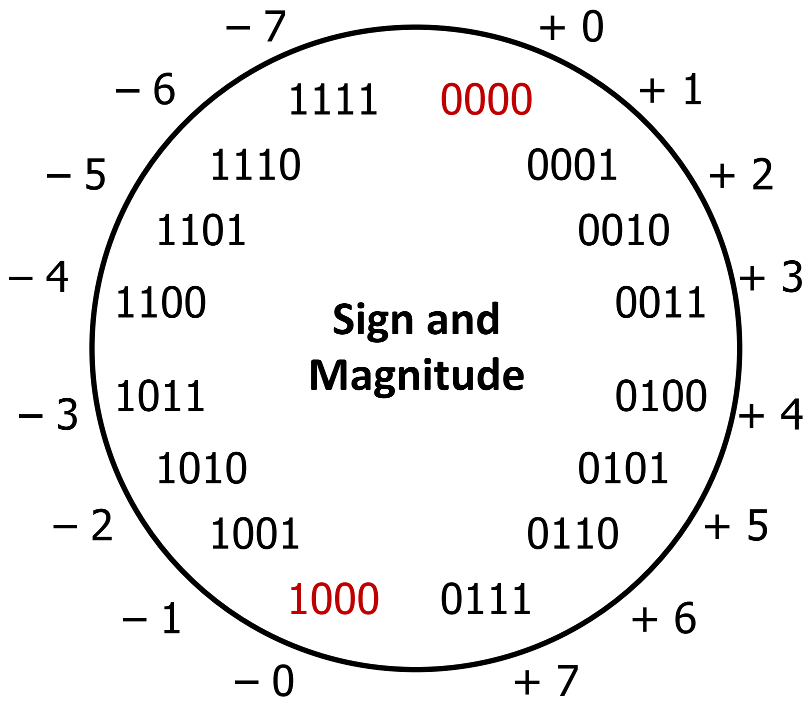
4	0100
– 3	– 0011
1	0001

✓

4	0100
+ -3	+ 1011
-7	1111

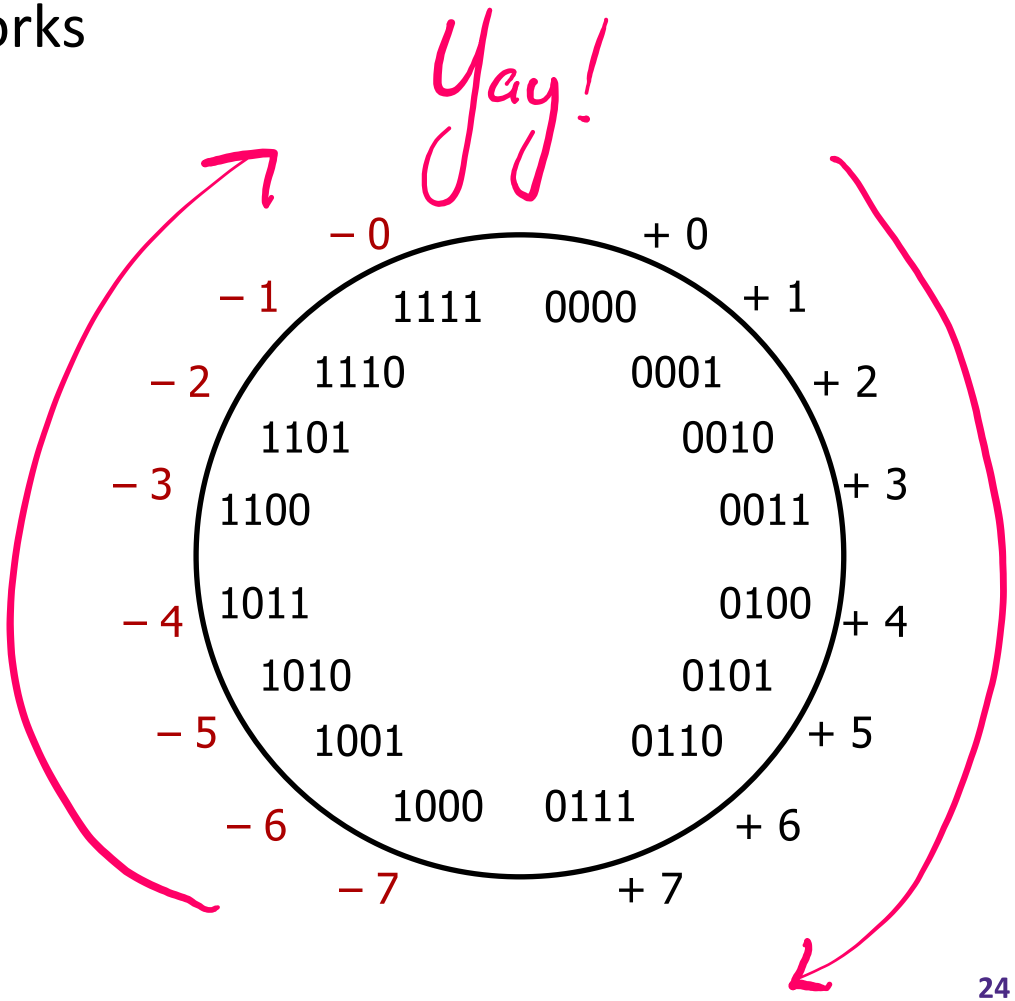
✗

Negatives "increment" in wrong direction!



Two's Complement

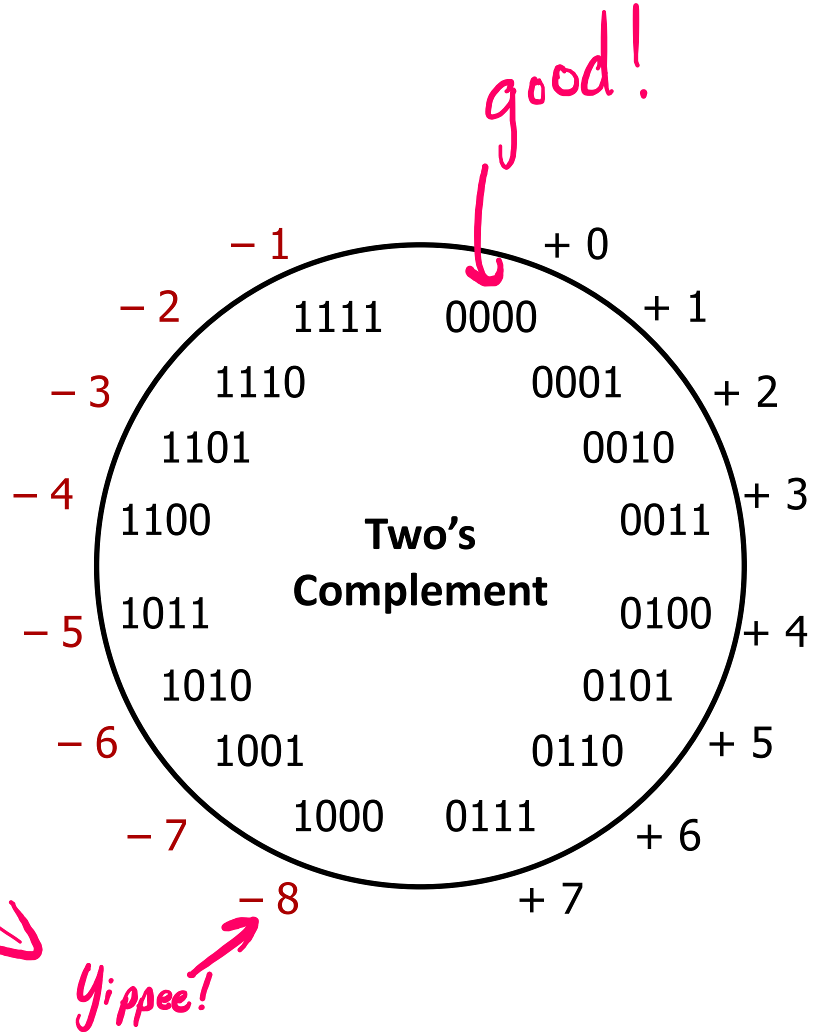
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



Two's Complement

- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works
  2) "Shift" negative numbers to eliminate –0

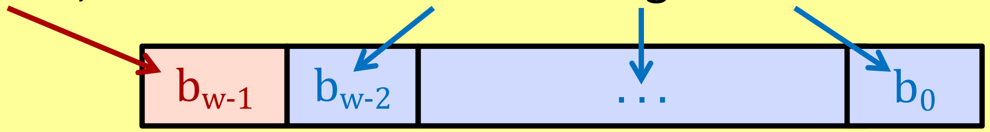
- MSB *still* indicates sign!
  - This is why we represent one more negative than positive number (-2^{N-1} to $2^{N-1}-1$)



Two's Complement Negatives (Review)

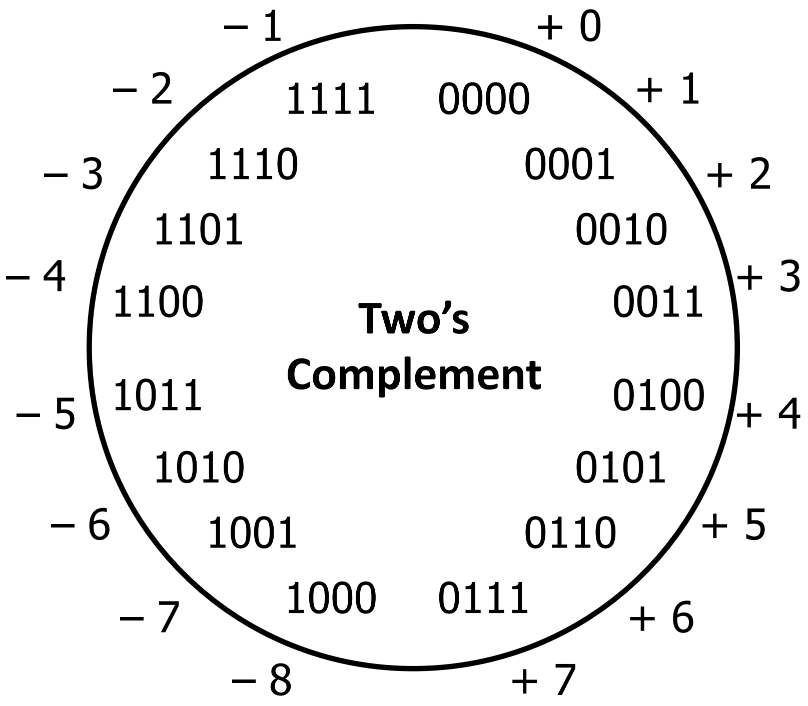
- Accomplished with one neat mathematical trick!

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



- 4-bit Examples:
  - 1010_2 unsigned:
    $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 =$ **10**
  - 1010_2 two's complement:
    $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 =$ **–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
  - Vote in Ed Lessons

A. -4

B. -5 ← two's complement

C. 11 ← unsigned

D. -3 ← sign & magnitude

E. We're lost...

Never!

- Two's Complement:

  0b1011

  $-2^3+0+2^1+2^0=5$

- Unsigned:

  $2^3+0+2^1+2^0=11$

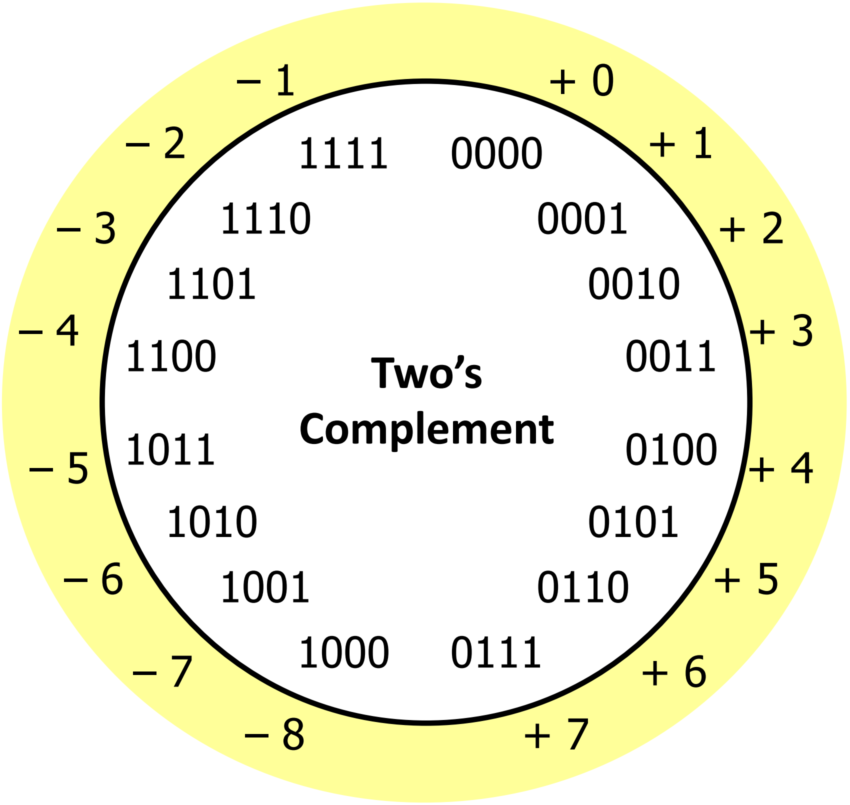
- Sign + Magnitude:

  0b1¦011 → sign neg 3 → -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!
    `( ~x + 1 == -x )`



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