Data III & Integers I

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub

http://xkcd.com/257/

Relevant Course Information

- HW2 due tonight, Wednesday (10/02) @ 11:59 pm
- HW3 due Friday (10/04) @ 11:59 pm
- HW4 due Monday (10/07) @ 11:59 pm (out after class)
- Lab 1a due Tuesday (10/08) @11:59pm
- From here on out, at 11am on day of lecture:
  - Reading for that lecture is DUE at 11am
  - Lecture activities from the previous lecture are DUE at 11am

Lab 1a released!

- Labs can be found linked on our course home page:
  - https://courses.cs.washington.edu/courses/cse351/24au/labs/lab1a.html
- 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 Tuesday 10/08, will overlap a bit with Lab 1b
  - Submit in Gradescope - we grade just your *last* submission
  - Don't wait until the last minute to submit! 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 understanding of what you should have learned from the lab
  - Also great practice for short answer questions on the exams

Memory, Data, and Addressing

- Representing information as bits and bytes
  - Binary, hexadecimal, fixed-widths
- Organizing and addressing data in memory
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
  - Assignment
  - Pointers, pointer arithmetic, and arrays
- **Boolean algebra and bit-level manipulations**

Reading Review

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

Review Questions

- Compute the result of the following expressions for `char c = 0x81;` // 0b 1000 0001

bitwise
  - `c ^ c` → 0x00

    1000 0001
    ^ 1000 0001
    0000 0000

  - `~c & 0xA9` → 0x28

    0111 1110
    & 1010 1001
    0010 1000

logical
  - `c || 0x80` (True, True) → 0x1
  - `!(!c)` (False) → True

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

  Decimal

  1111 0000

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

  0000 1111
  + 1
  0001 0000

  $-sc = 16$

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

input bit   bitmask   input bit

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

"set to zero"

$b \,\&\, 0 =$ <u>0</u> $0 \& 0 \rightarrow 0$, $1 \& 0 \rightarrow 0$

$b \mid 0 =$ <u>b</u> $0 \mid 0 \rightarrow 0$, $1 \mid 0 \rightarrow 1$

$b \wedge 0 =$ <u>b</u> $0 \wedge 0 \rightarrow 0$, $1 \wedge 0 \rightarrow 1$

$b \,\&\, 1 =$ <u>b</u> $0 \& 1 \rightarrow 0$, $1 \& 1 \rightarrow 1$

$b \mid 1 =$ <u>1</u> $0 \mid 1 \rightarrow 1$, $1 \mid 1 \rightarrow 1$

"set to one"

$b \wedge 1 =$ <u>$\bar{b}$</u> $0 \wedge 1 \rightarrow 1$, $1 \wedge 1 \rightarrow 0$

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 ← You choose
  11110101  ← output
```

You choose

"set to one" "keep as is"

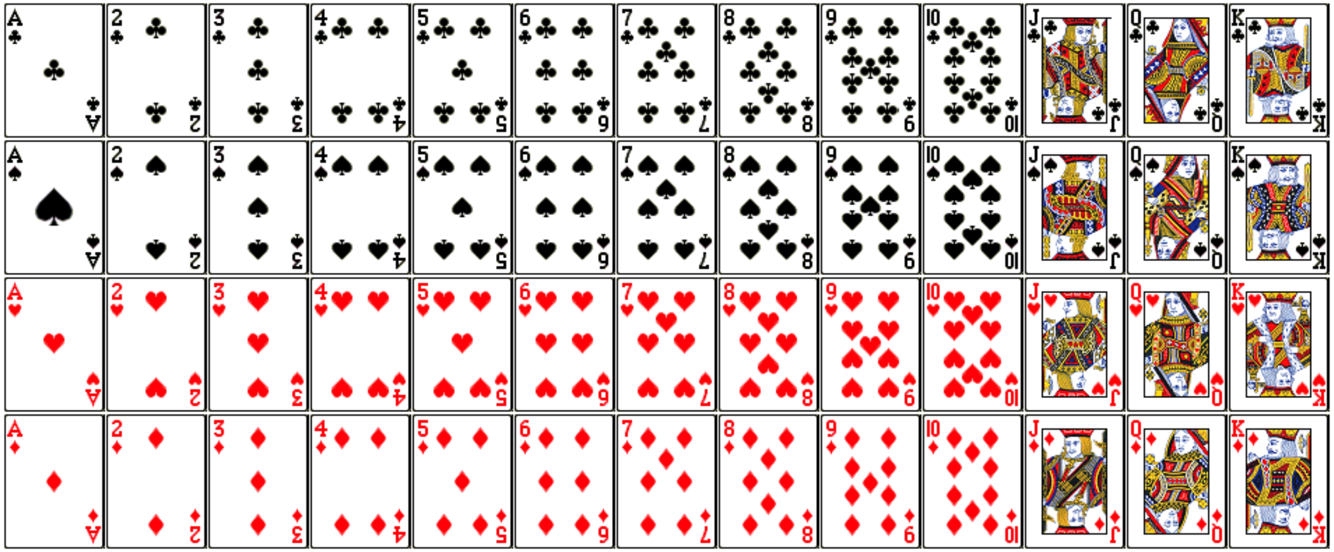
Short-Circuit Evaluation

- If the result of a binary logical operator (`&&`, `||`) can be determined by its first operand, then the second operand is never evaluated
  - Also known as *early termination*

- Example: `(p && *p)` for a pointer `p` to "protect" the dereference
  - Dereferencing `NULL` (0) results in a segfault

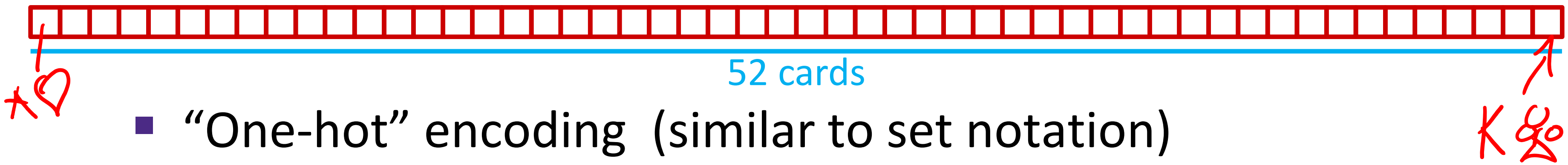
Numerical Encoding Design Example

- Encode a standard deck of playing cards
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?



Two possible representations

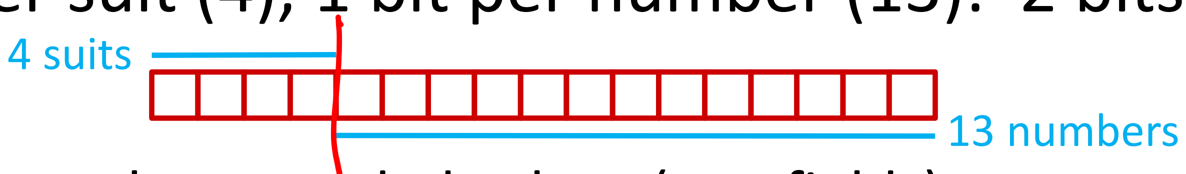
1) 1 bit per card (52): bit corresponding to card set to 1



- "One-hot" encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

52 bits per card

2) 1 bit per suit (4), 1 bit per number (13): 2 bits set



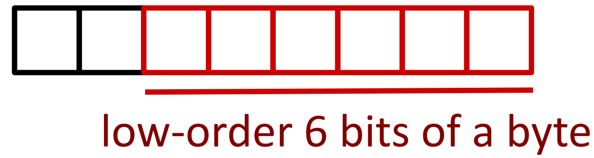
- **Pair** of one-hot encoded values (two fields)
- Easier to compare suits and values, but still lots of bits used

17 bits per card

Two better representations

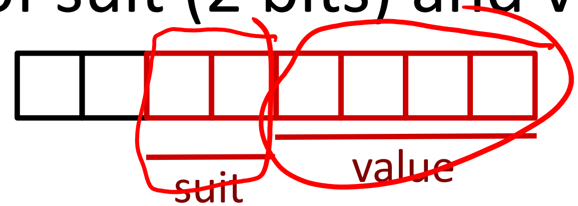
3) Binary encoding of all 52 cards – only 6 bits needed

- $2^6 = 64 \geq 52$



- 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)



- 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**

equivalent

SUIT_MASK = 0x30 = | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

suit value

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);
}
```

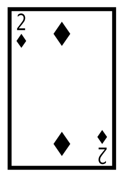
0	0	0	1	0	0	1	0

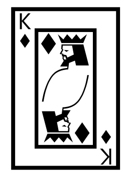
&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0





`SUIT_MASK`

0	0	0	1	1	1	0	1

&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0

^

0	0	0	0	0	0	0	0

!

0	0	0	0	0	0	0	1

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

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));
}
```

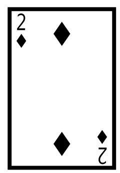
VALUE_MASK = 0x0F =

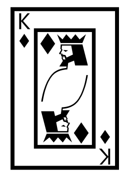
0	0	0	0	1	1	1	1
		suit		value			

Compare Card **Values**

```
#define VALUE_MASK  0x0F

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





0	0	1	0	0	0	1	0

&

0	0	0	0	1	1	1	1

=

0	0	0	0	0	0	1	0

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

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

0 (false)

Integers

- **Binary representation of integers**
  - **Unsigned and signed**
- Shifting and arithmetic operations
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension

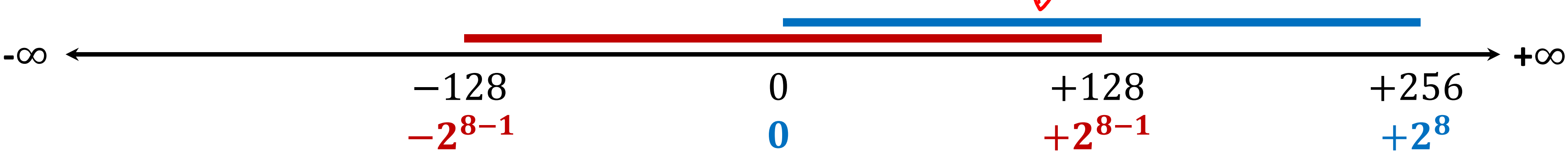
Encoding Integers

- 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: $0 \ldots 2^w - 1$
  - Signed values: $-2^{w-1} \ldots 2^{w-1} - 1$

8 bits
unsigned 0 . . . 255
signed -128 127

same widths, just shifted

- **Example:** 8-bit integers (*e.g.* `char`)



Unsigned Integers (Review)

- Unsigned values follow the standard base 2 system
  - $\mathrm{b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0} = \mathrm{b_7} 2^7 + \mathrm{b_6} 2^6 + \cdots + \mathrm{b_1} 2^1 + \mathrm{b_0} 2^0$
- Useful formula: $2^{\mathrm{N}-1} + 2^{\mathrm{N}-2} + \ldots + 2 + 1 = 2^{\mathrm{N}} - 1$
  - *i.e.*, N ones in a row = $2^{\mathrm{N}} - 1$
  - *e.g.*, 0b111111 = 63

Sign and Magnitude

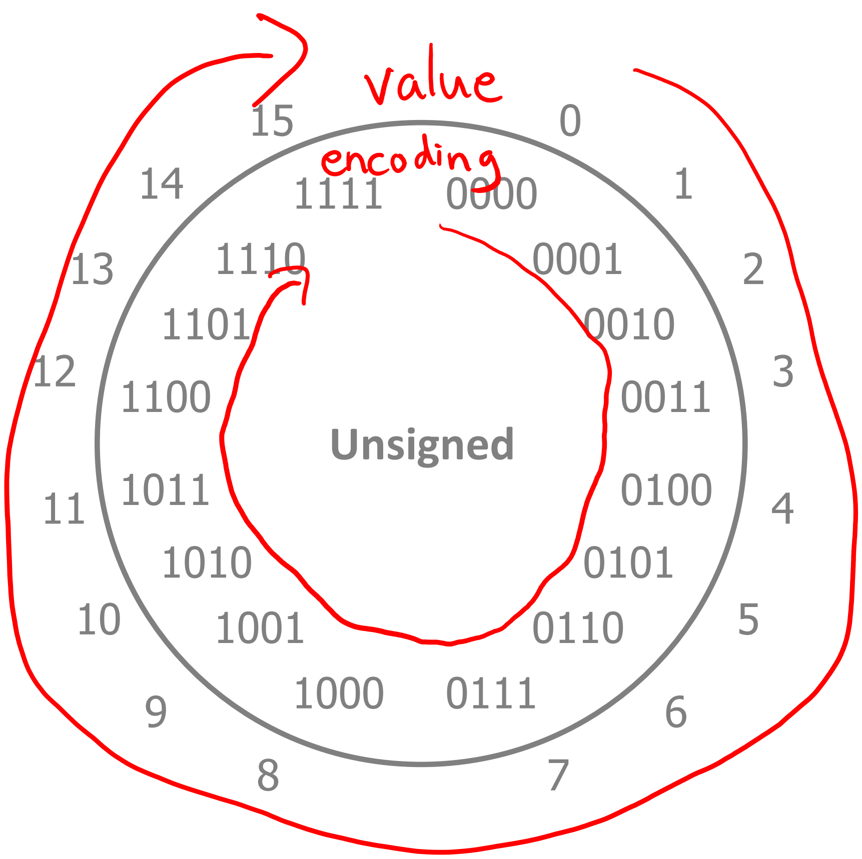
Not used in practice for integers!

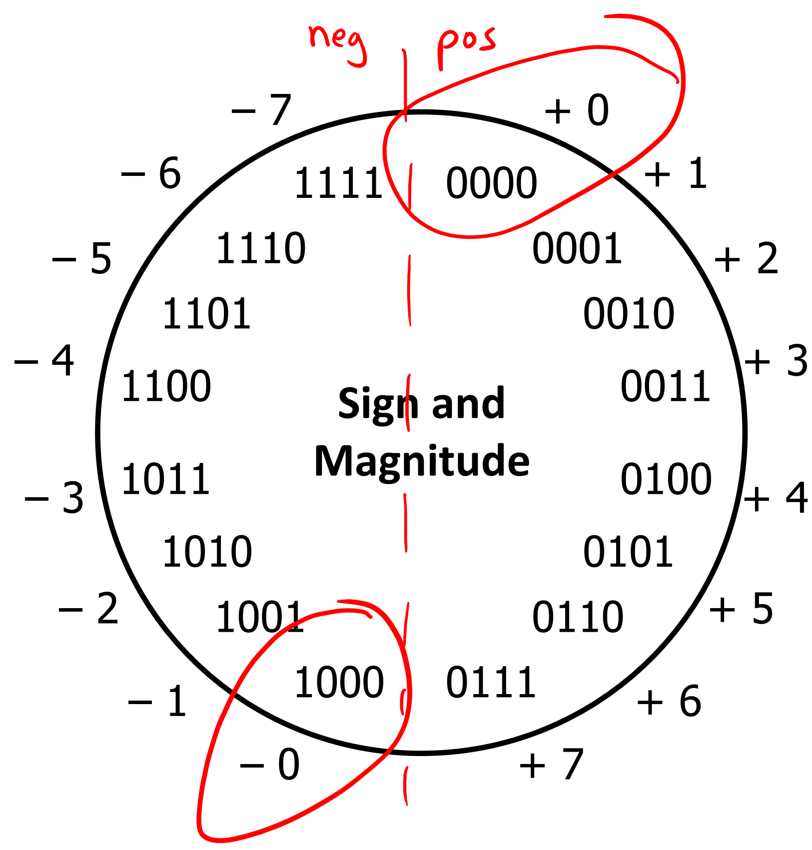
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive numbers; `sign=1`: negative numbers
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0
- Examples (8 bits):
  - 0x00 = 00000000_2 is non-negative, because the sign 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?

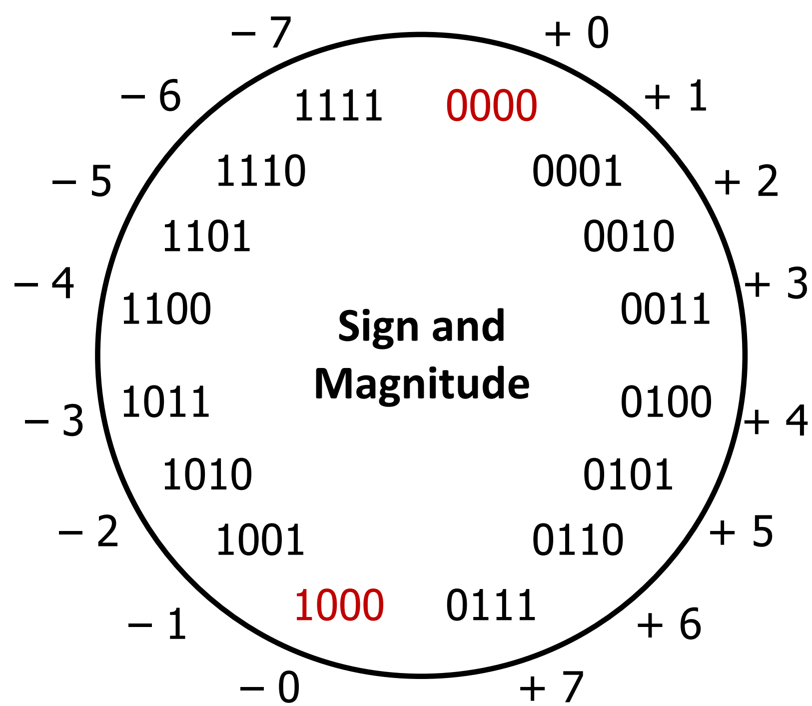




Sign and Magnitude

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 != 4+(-3)`

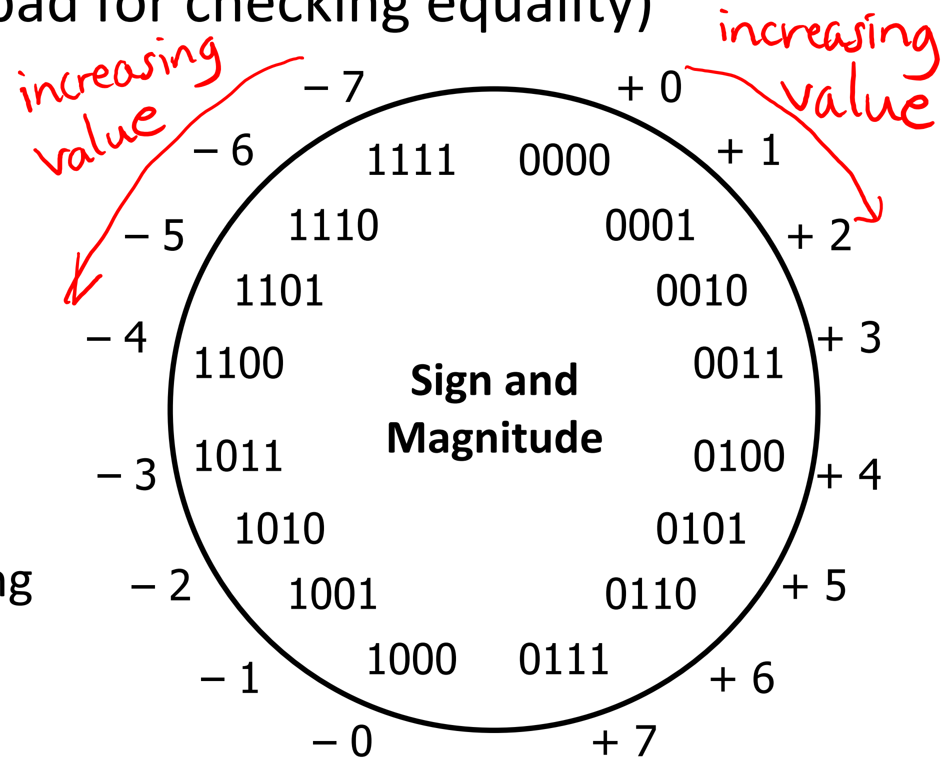
4	0100
– 3	– 0011
1	0001

✓

4	0100
+ -3	+ 1011
-7	1111

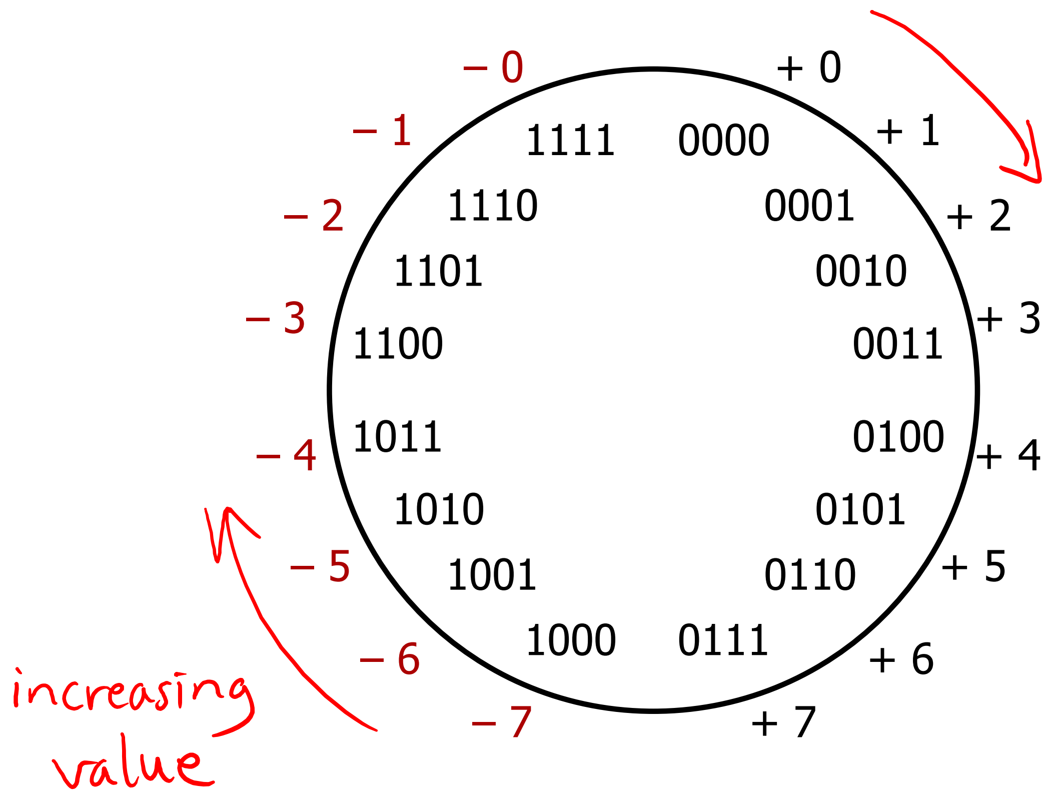
✗

    - Negatives "increment" in wrong direction!



Two's Complement

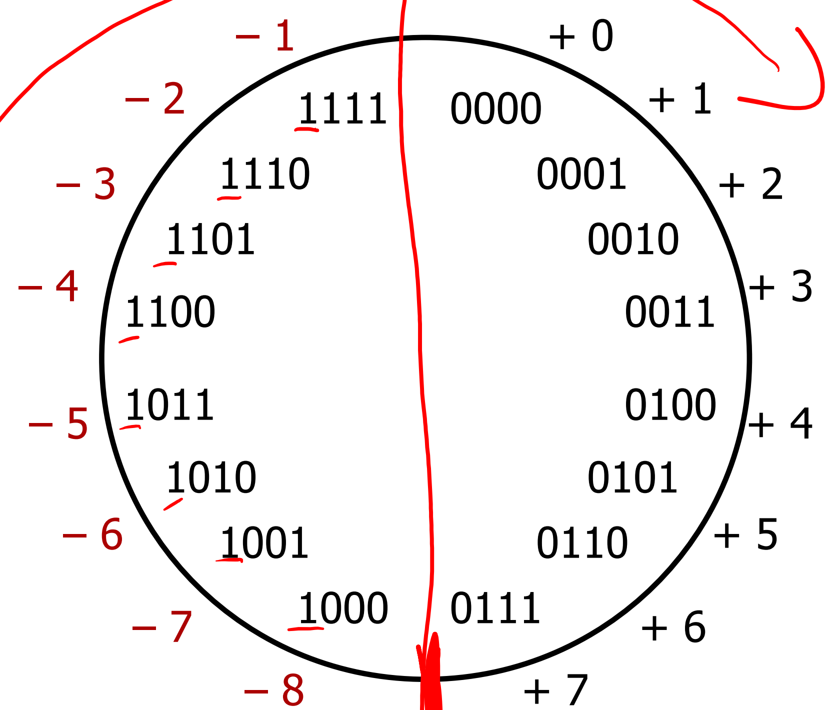
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



Two’s Complement

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

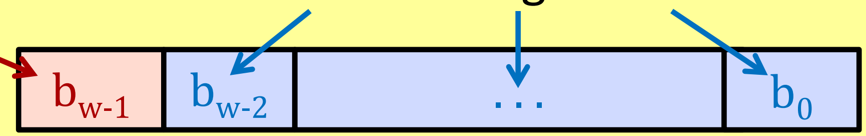
- MSB *still* indicates sign!
  - This is why we represent one more negative than positive number (-2^{N-1} to $2^{N-1}-1$)



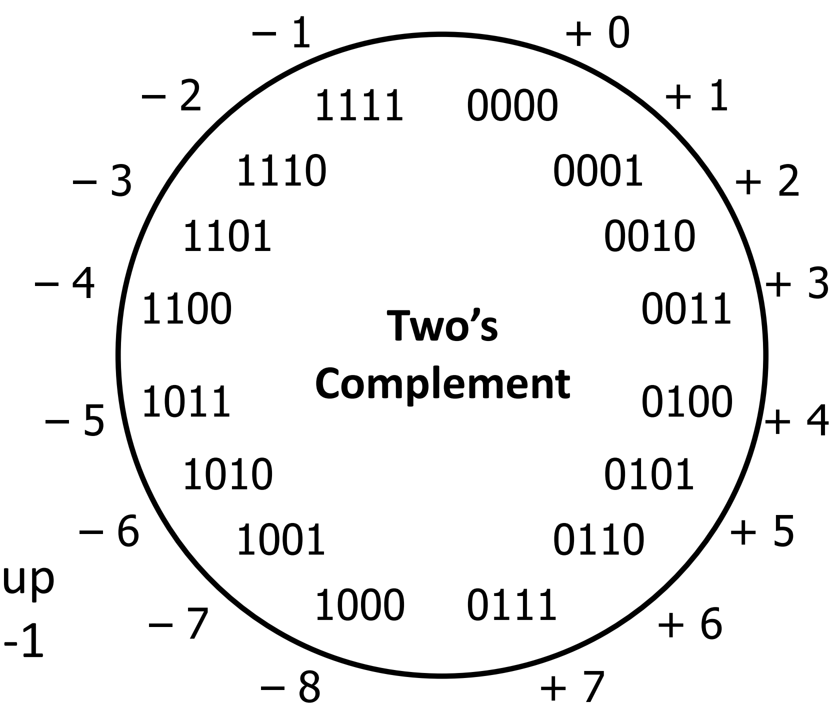
Two’s Complement Negatives (Review)

- 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:
    $1*2^3+0*2^2+1*2^1+0*2^0 = \mathbf{10}$
  - 1010_2 two’s complement:
    $-1*2^3+0*2^2+1*2^1+0*2^0 = \mathbf{-6}$



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

  3 one's in a row

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

Polling Question

2^3 2^1 2^0

- 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

C. 11

D. -3

E. We're lost…

unsigned = 8 + 2 + 1 = 11

(two's complent) signed = -8 + 2 + 1 = -5

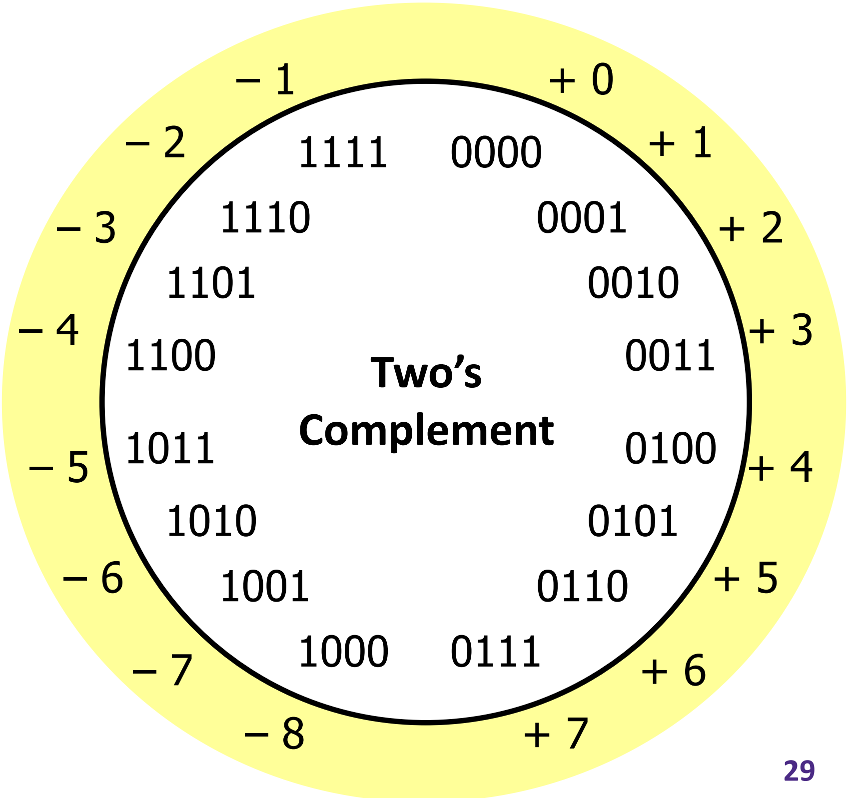
sign-magnitude = 2+1 → -3

1st bit says "this is a negative #"

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