Data III & Integers I

CSE 351 Spring 2021

Instructor:

Ruth Anderson

Teaching Assistants:

Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Neil Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu

http://xkcd.com/257/

Administrivia

- Lab 0 and hw2 due tonight (4/05) @ 11:59 pm
- hw3 due Wednesday (4/07) @ 11:59 pm
- hw4 due Friday (4/09) @ 11:59 pm
- 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
- **Questions Doc**: You must log on with your @uw google account to access!!
  - https://tinyurl.com/CSE351-21sp-Questions
  - Then open the "TODAY's Lecture Questions" doc for 11:30/2:30
  - See https://edstem.org/us/courses/4805/discussion/339425 for info about turning on your UW G Suite account for your @uw address

Lab 1a released

- Labs can be found linked on our course home page:
  - https://courses.cs.washington.edu/courses/cse351/21sp/labs/lab1a.php
- Workflow:
  1) Edit `pointer.c`
  2) Run the Makefile (`make`) and check for compiler errors & warnings
  3) Run ptest (`./ptest`) and check for correct behavior
  4) Run rule/syntax checker (`python dlc.py`) and check output
- Due Monday 4/12, will overlap a bit with Lab 1b
  - Submit in Gradescope
  - We grade just your *last* submission

Lab Reflections

- All subsequent labs (after Lab 0) have a “reflection” portion
  - The Reflection questions 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

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;`
  - `c ^ c`
  - `~c & 0xA9`
  - `c || 0x80`
  - `!!c`

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

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

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

$b\ \&\ 0 = \_\_\_$ $b\ \&\ 1 = \_\_\_$

$b\ |\ 0 = \_\_\_$ $b\ |\ 1 = \_\_\_$

$b\ \wedge\ 0 = \_\_\_$ $b\ \wedge\ 1 = \_\_\_$

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

- <u>Example</u>: $b|0 = b,\ b|1 = 1$

```
    01010101  ← input
 |  11110000  ← bitmask
 -----------
    11110101
```

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

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

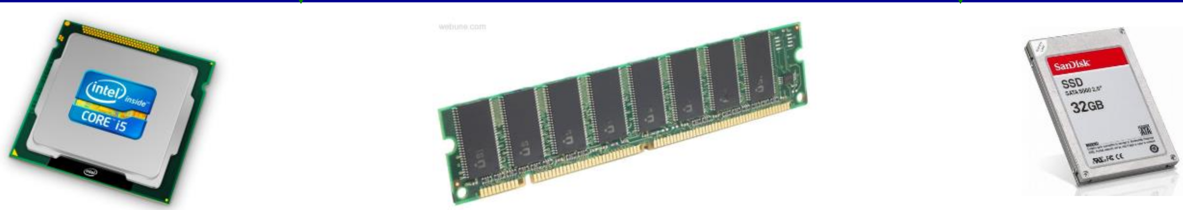
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

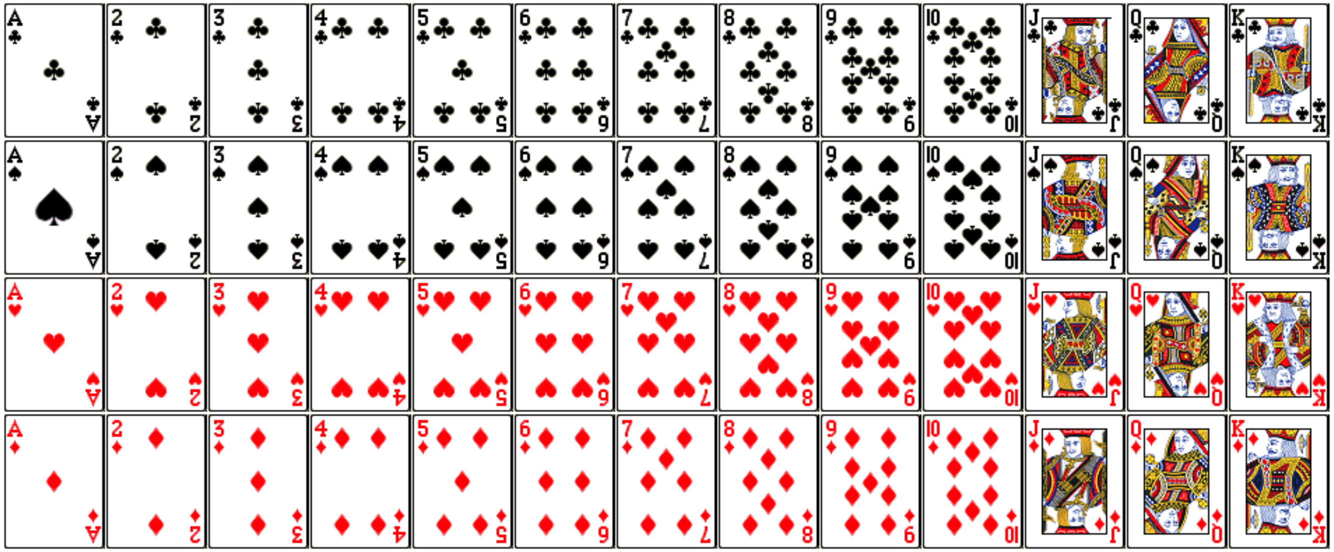
Computer system:



- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

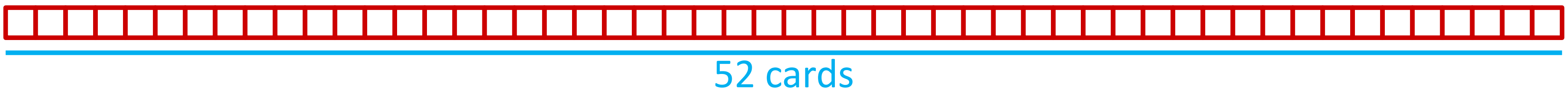
But before we get to integers….

- 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?



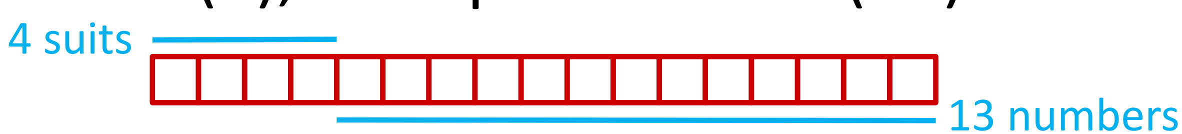
Two possible representations

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

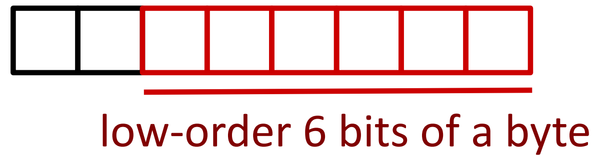
2) 1 bit per suit (4), 1 bit per number (13): 2 bits set



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

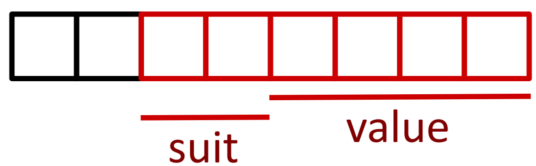
Two better representations

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)



♣	00
♦	01
♥	10
♠	11

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

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

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 =

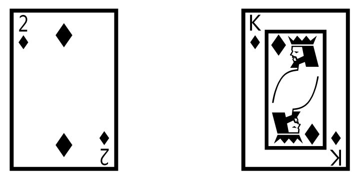
0	0	1	1	0	0	0	0

suit (bits 1 1), value (last four bits 0 0 0 0)

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);
}
```



Card 1		Card 2
0 0 0 1 0 0 1 0		0 0 0 1 1 1 0 1
&		&
0 0 1 1 0 0 0 0	SUIT_MASK	0 0 1 1 0 0 0 0
=		=
0 0 0 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));
}
```

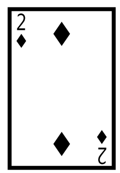
VALUE_MASK = 0x0F =

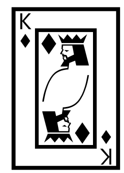
0	0	0	0	1	1	1	1
		suit		value			

Compare Card Values

```
#define VALUE_MASK  0x0F

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





card1		card2
0 0 1 0 0 0 1 0		0 0 1 0 1 1 0 1
&		&
0 0 0 0 1 1 1 1	VALUE_MASK	0 0 0 0 1 1 1 1
=		=
0 0 0 0 0 0 1 0		0 0 0 0 1 1 0 1

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

0 (false)

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- **Integers & floats**
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

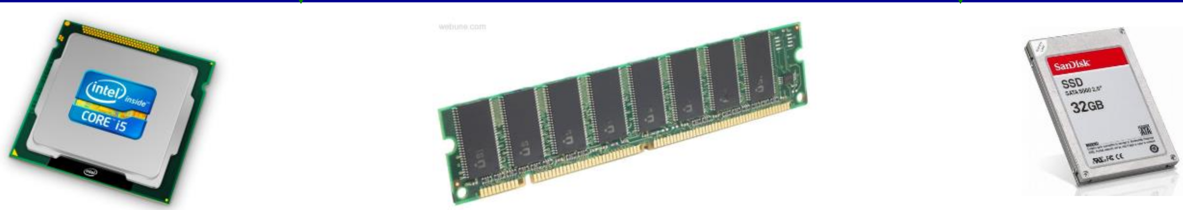
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10 | OS X Yosemite

Computer system:

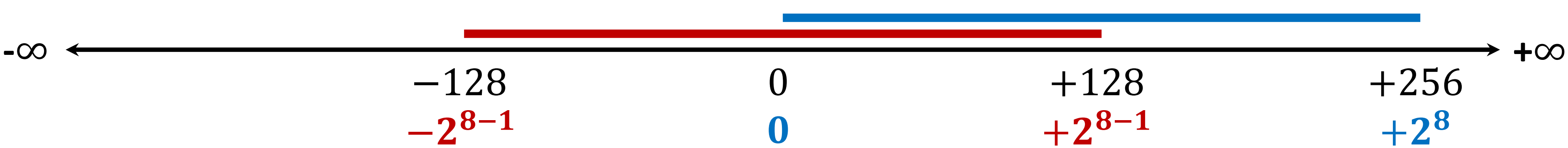


Integers

- **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: $-2^{w-1} \ldots 2^{w-1} - 1$
- **Example:** 8-bit integers (*e.g.* `char`)



Unsigned Integers

- 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

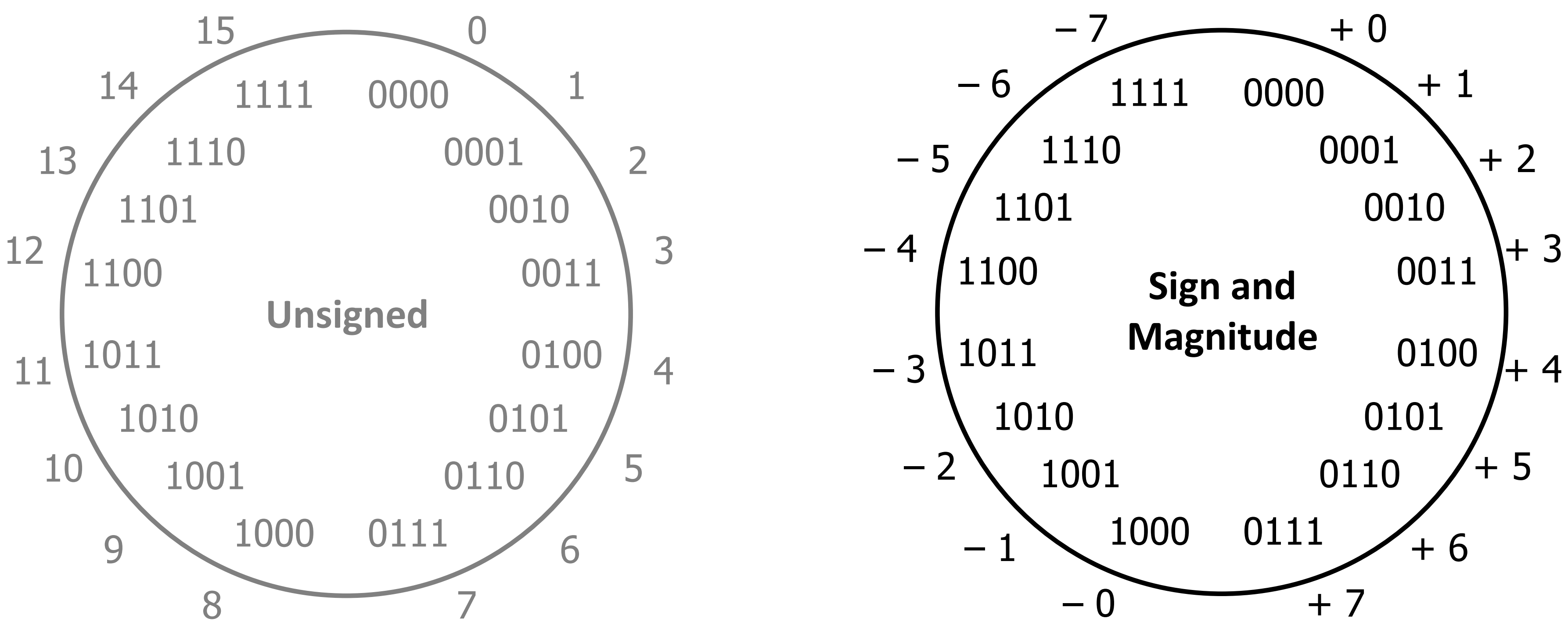
Not used in practice!

- 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!

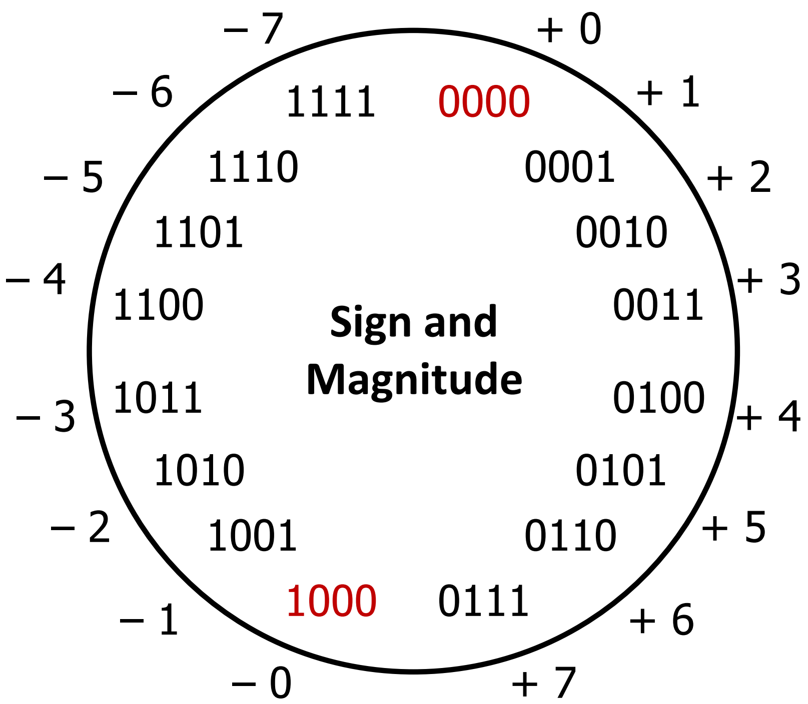
- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?



Sign and Magnitude

Not used in practice!

- 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!

- 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)`

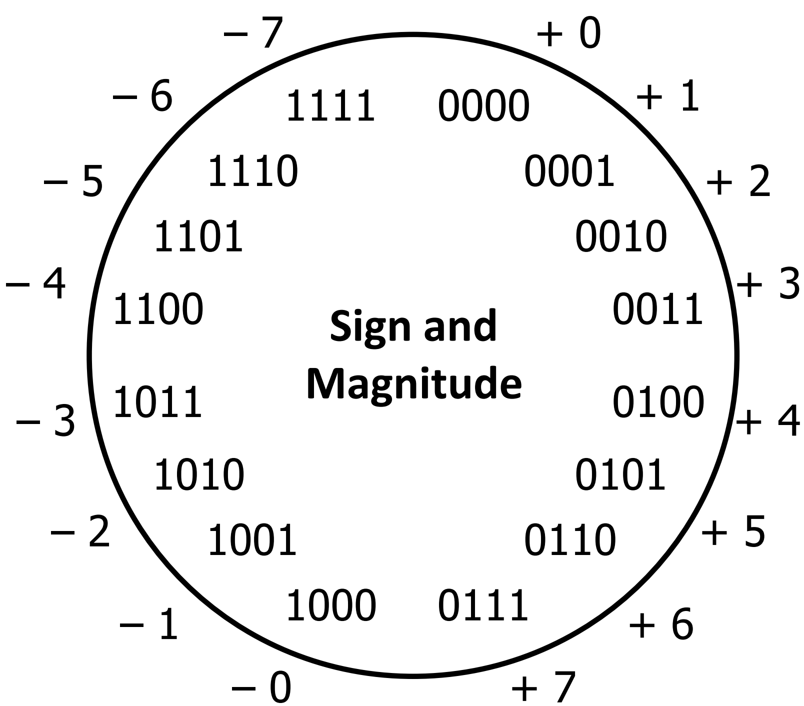
4	0100
− 3	− 0011
1	0001

✓

4	0100
+ −3	+ 1011
−7	1111

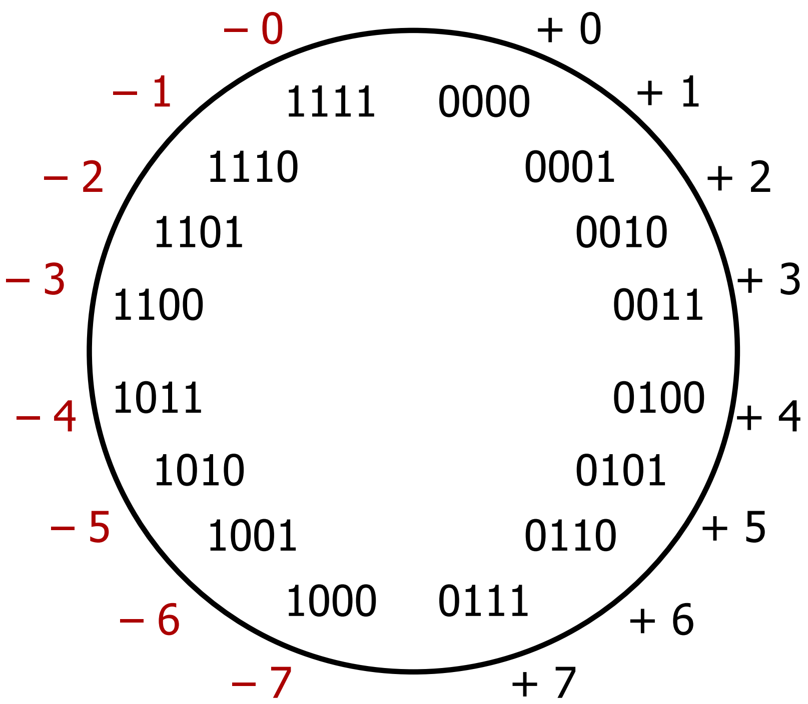
✗

    - Negatives "increment" in wrong direction!



Two's Complement

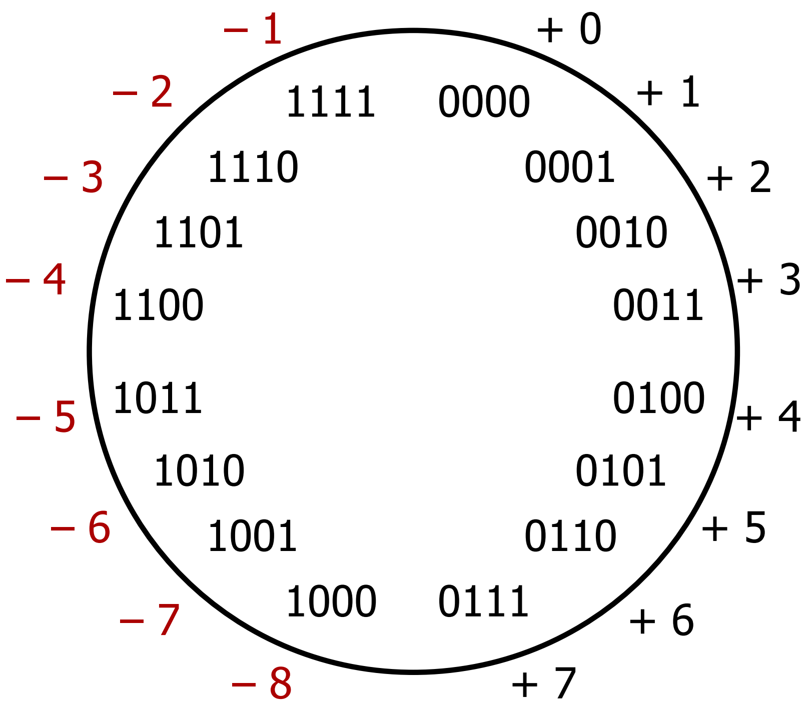
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



Two’s Complement

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

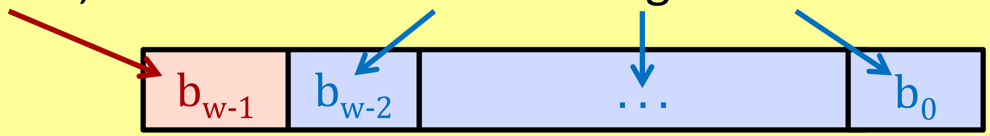
- MSB *still* indicates sign!
  - This is why we represent one more negative than positive number (-2^{N-1} to $2^{N-1}-1$)



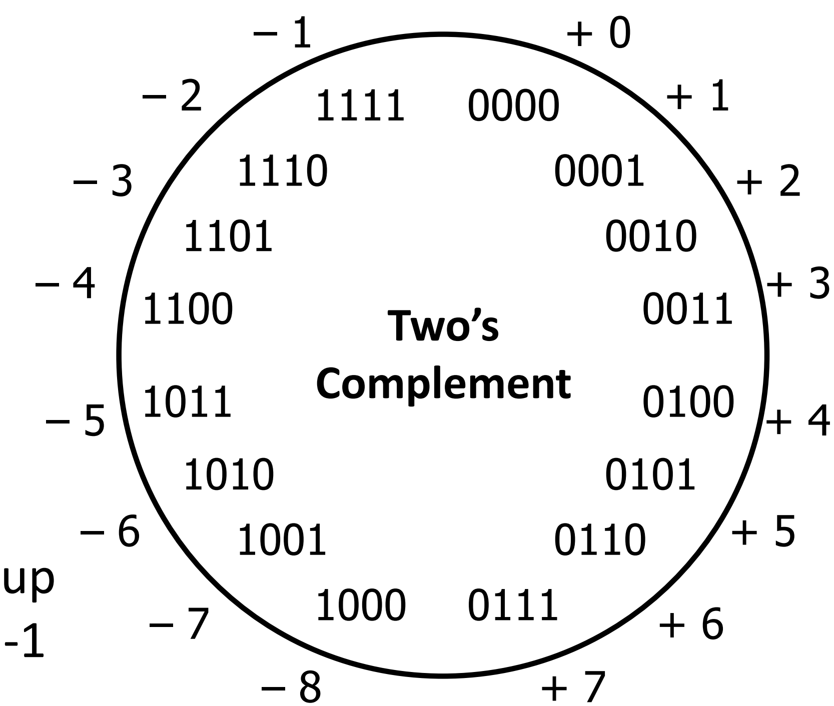
Two's Complement Negatives

- Accomplished with one neat mathematical trick!

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

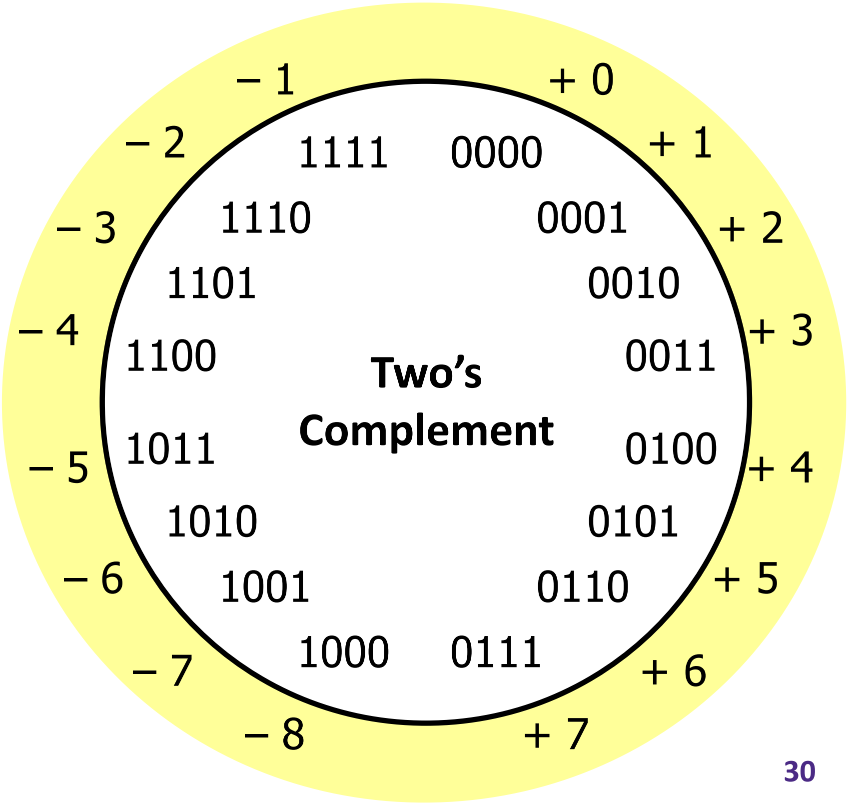


- 4-bit Examples:
  - 1010_2 unsigned:
    $1*2^3+0*2^2+1*2^1+0*2^0$ = **10**
  - 1010_2 two's complement:
    $-1*2^3+0*2^2+1*2^1+0*2^0$ = **–6**
- -1 represented as:
  $1111_2 = -2^3+(2^3 - 1)$
  - MSB makes it super negative, add up all the other bits to get back up to -1



Why Two's Complement is So Great

- 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 )`



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

C. 11

D. -3

E. We're lost…

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