# Data III & Integers I

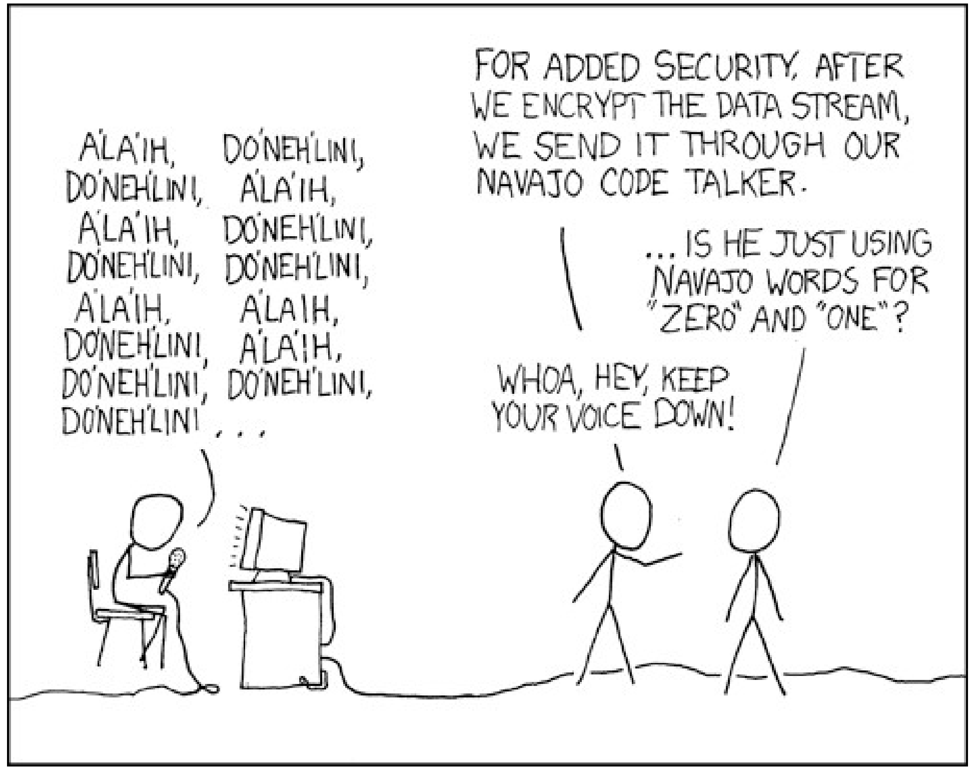
CSE 351 Spring 2019


**Instructor:**

Ruth Anderson

**Teaching Assistants:**

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh


http://xkcd.com/257/

# Administrivia

- Lab 0 due TODAY @ 11:59 pm
  - *You will be revisiting this program throughout this class!*
- Homework 1 due Wednesday
  - Reminder: autograded, 20 tries, no late submissions
- Lab 1a released
  - 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 4/15, will overlap a bit with Lab 1b
    - 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 Canvas
  - 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

# Memory, Data, and Addressing

- Hardware - High Level Overview
- Representing information as bits and bytes
  - Memory is a byte-addressable array
  - Machine "word" size = address size = register size
- Organizing and addressing data in memory
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
- **Boolean algebra and bit-level manipulations**

# Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic (True → 1, False → 0)
  - AND: `A&B=1` when both A is 1 and B is 1
  - OR: `A|B=1` when either A is 1 or B is 1
  - XOR: `A^B=1` when either A is 1 or B is 1, but not both
  - NOT: `~A=1` when A is 0 and vice-versa
  - DeMorgan's Law: `~(A|B) = ~A & ~B`
    `~(A&B) = ~A | ~B`

AND

| & | 0 | 1 |
|---|---|---|
| 0 | **0** | **0** |
| 1 | **0** | **1** |

OR

| \| | 0 | 1 |
|---|---|---|
| 0 | **0** | **1** |
| 1 | **1** | **1** |

XOR

| ^ | 0 | 1 |
|---|---|---|
| 0 | **0** | **1** |
| 1 | **1** | **0** |

NOT

| ~ | |
|---|---|
| 0 | **1** |
| 1 | **0** |

# General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
  - All of the properties of Boolean algebra apply

```
    01101001      01101001      01101001
&   01010101   |  01010101   ^  01010101   ~  01010101
    01000001      01111101      00111100      10101010
```

- Examples of useful operations:

$x \wedge x = 0$

"sets to 1"

$x \mid 1 = 1,$

0 | 1 = 1
1 | 1 = 1

"leaves as is"

$x \mid 0 = x$

0 | 0 = 0
1 | 0 = 1

```
  01010101
^ 01010101
  00000000   ← creates 0

  01010101   ← data of interest
| 11110000   ← bit mask (specifically chosen)
  11110101
  set  left as is
```

# Bit-Level Operations in C

- & (AND), | (OR), ^ (XOR), ~ (NOT)
  - View arguments as bit vectors, apply operations bitwise
  - Apply to any "integral" data type
    - (8 bytes) (4 bytes) (2 bytes) (1 byte)
    - long, int, short, char, unsigned

bit vector will be width of datatype

- Examples with `char a, b, c;`

C code — Internally — Result

```
a = (char) 0x41;    // 0x41->0b 0100 0001
b = ~a;             //        0b 1011 1110->0x BE
```

```
a = (char) 0x69;    // 0x69->0b 0110 1001
b = (char) 0x55;    // 0x55->0b 0101 0101
c = a & b;          //        0b 0100 0001->0x 41
```

```
a = (char) 0x41;    // 0x41->0b 0100 0001
b = a;              //        0b 0100 0001
c = a ^ b;          //        0b 0000 0000->0x 00
```

# Contrast: Logic Operations

- Logical operators in C: `&&` (AND), `||` (OR), `!` (NOT)
  - **0** is False, **anything nonzero** is True
  - **Always** return 0 or 1
  - Early termination (a.k.a. short-circuit evaluation) of `&&`, `||`

0xCC = 0b 1100 1100
0x33 = 0b 0011 0011

- Examples (`char` data type)

0xCC & 0x33 -> 0x00

  - `!0x41 -> 0x00`
  - `!0x00 -> 0x01`
  - `!!0x41 -> 0x01`
  - `0xCC && 0x33 -> 0x01`
  - `0x00 || 0x33 -> 0x01`
  - `p && *p`
    - If `p` is the null pointer (0x0), then `p` is never dereferenced!

If ① determines output of logical operator, then ② is never evaluated

# 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
```

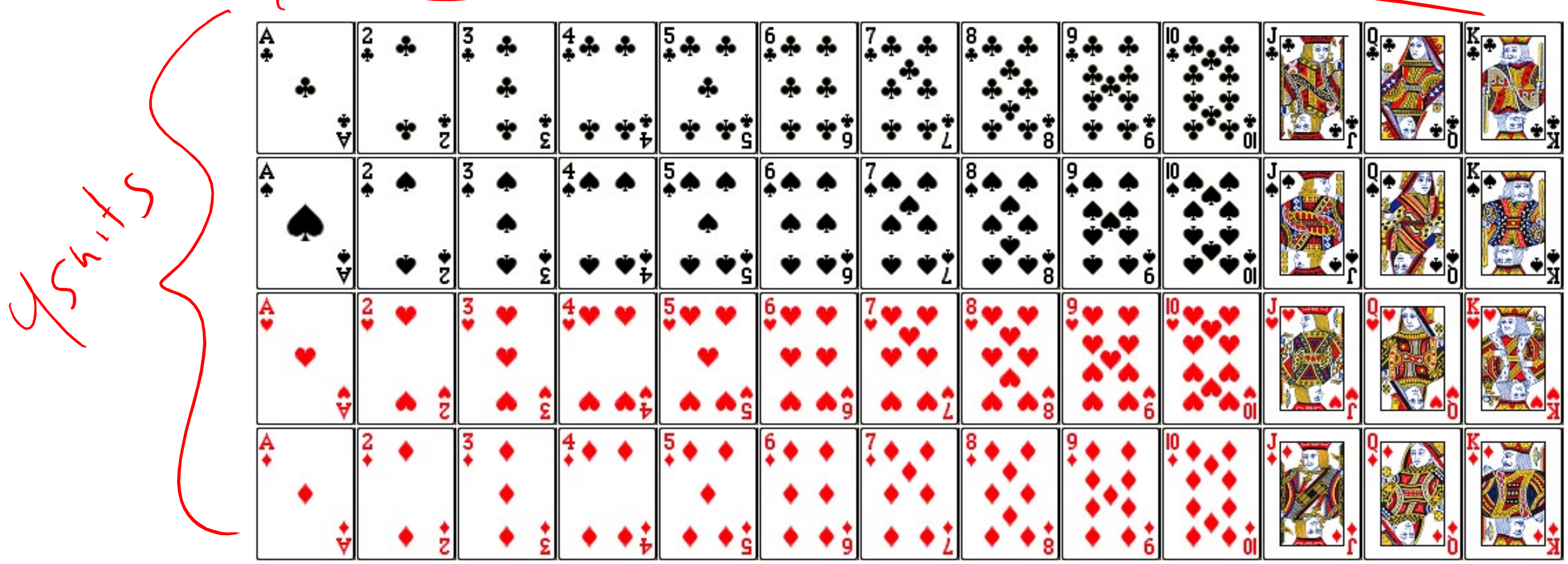
OS:

Windows 10
OS X Yosemite

Computer system:

# But before we get to integers….

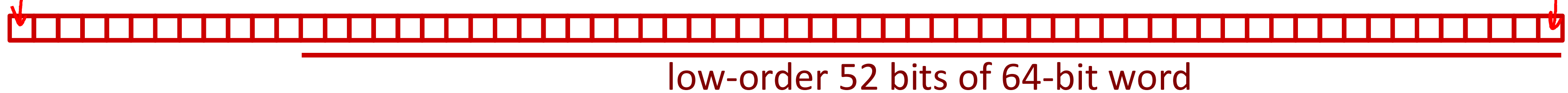
- Encode a standard deck of playing cards
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?



# Two possible representations

K♤ A♧

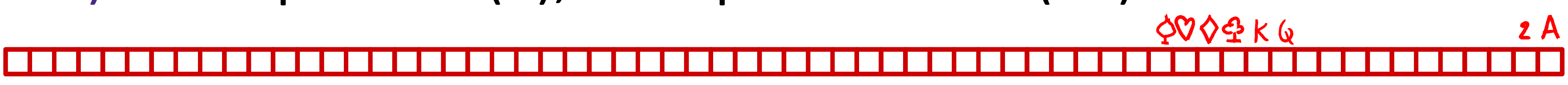
1) 1 bit per card (52): bit corresponding to card set to 1




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

52 bits fits in → 7 bytes (56 bits)

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



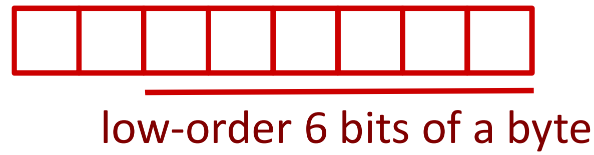

4 suits 13 numbers
17 bits → 3 bytes

- Pair of one-hot encoded values
- Easier to compare suits and values, but still lots of bits used

# Two better representations

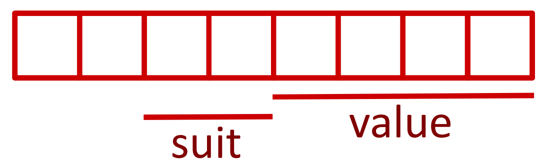
3) Binary encoding of all 52 cards – only 6 bits needed

- $2^6 = 64 \geq 52$
  $2^5 = 32 < 52$




- 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

| | | |
|---|---|---|
| C | ♣ | 00 |
| D | ♦ | 01 |
| H | ♥ | 10 |
| S | ♠ | 11 |

| K | Q | J | . . . | 3 | 2 | A |
|---|---|---|---|---|---|---|
| 1101 | 1100 | 1011 | . . . | 0011 | 0010 | 0001 |
| 13 | | | . . . | | | 1 |

# 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) ) { ... }
```

text substitution

```
#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 (keep) value (discard)

x & 0 = 0

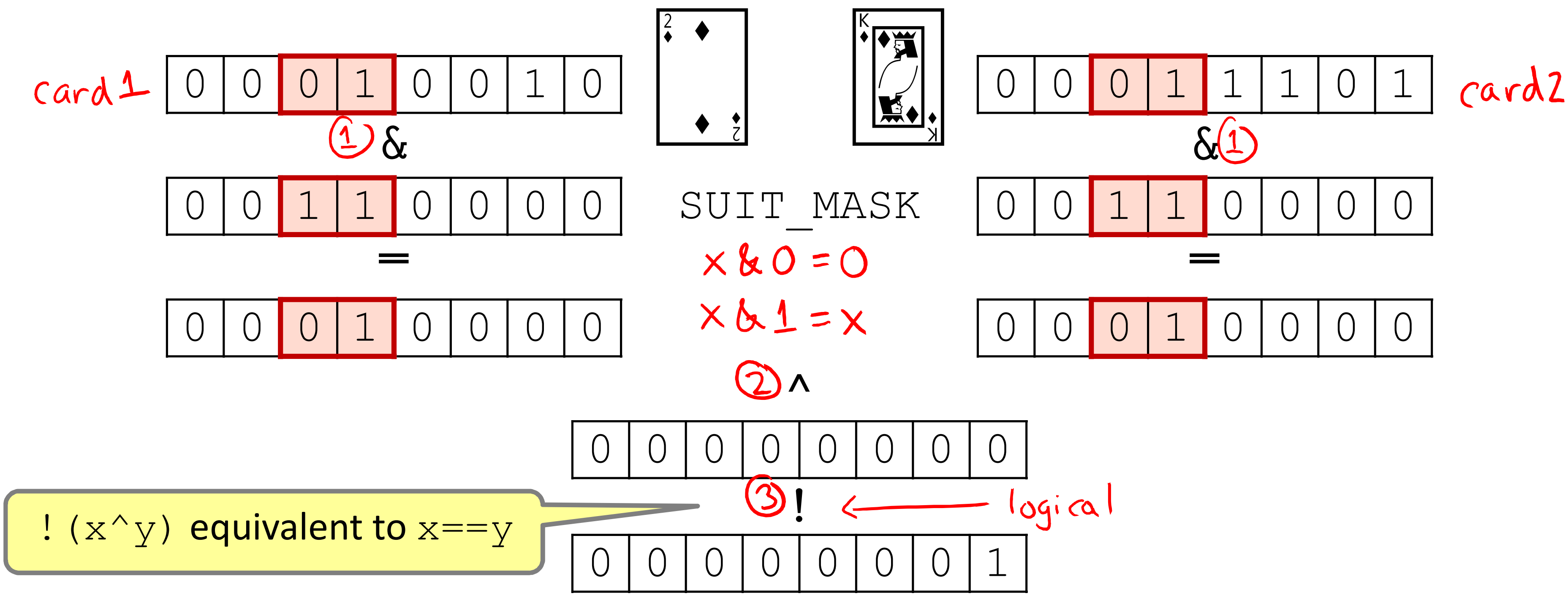
x & 1 = x

# 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.

```
#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

**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector *v*.

```
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 |

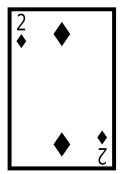
suit (discard) — value (keep)

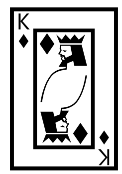
# Compare Card Values

**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector *v*.

```
#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 |

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

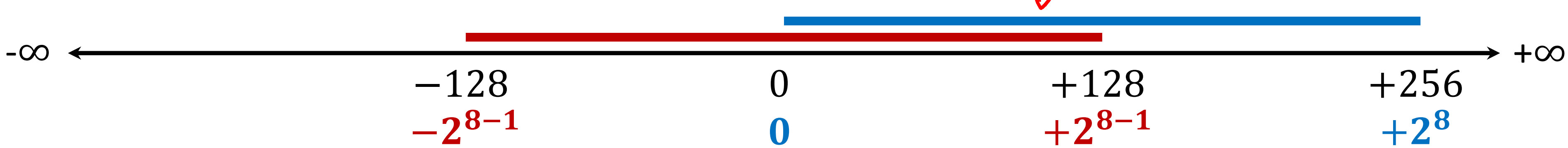
0 (false)

# Integers

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

# 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   w → 8 bits
  - Unsigned values: $0 \ldots 2^w - 1$   0 . . . 255
  - Signed values: $-2^{w-1} \ldots 2^{w-1} - 1$   -128 . . . 127
  
  same widths, just shifted
- **Example:** 8-bit integers (*e.g.* `char`)



# Unsigned Integers

- Unsigned values follow the standard base 2 system
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$
- Add and subtract using the normal “carry” and “borrow” rules, just in binary

```
  63        00111111
+  8      + 00001000
  71        01000111
```

6 1's in a row

- Useful formula: $2^{N-1} + 2^{N-2} + \ldots + 2 + 1 = 2^N - 1$
  - *i.e.* N ones in a row = $2^N - 1$
- How would you make *signed* integers?

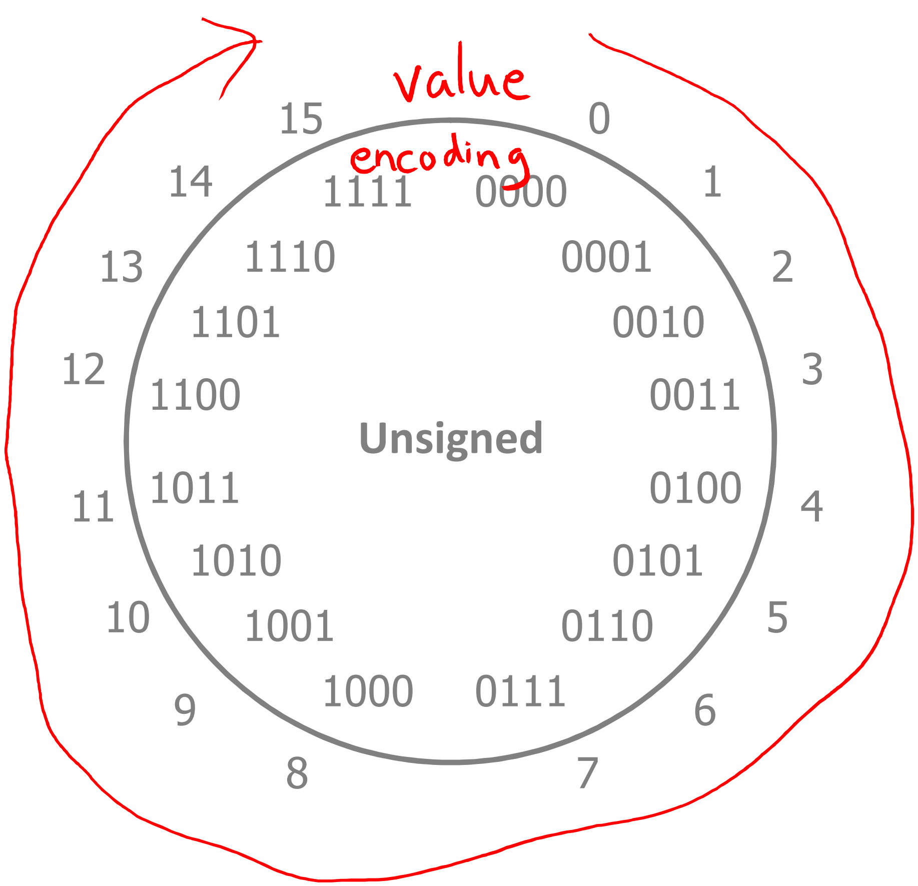
# Sign and Magnitude

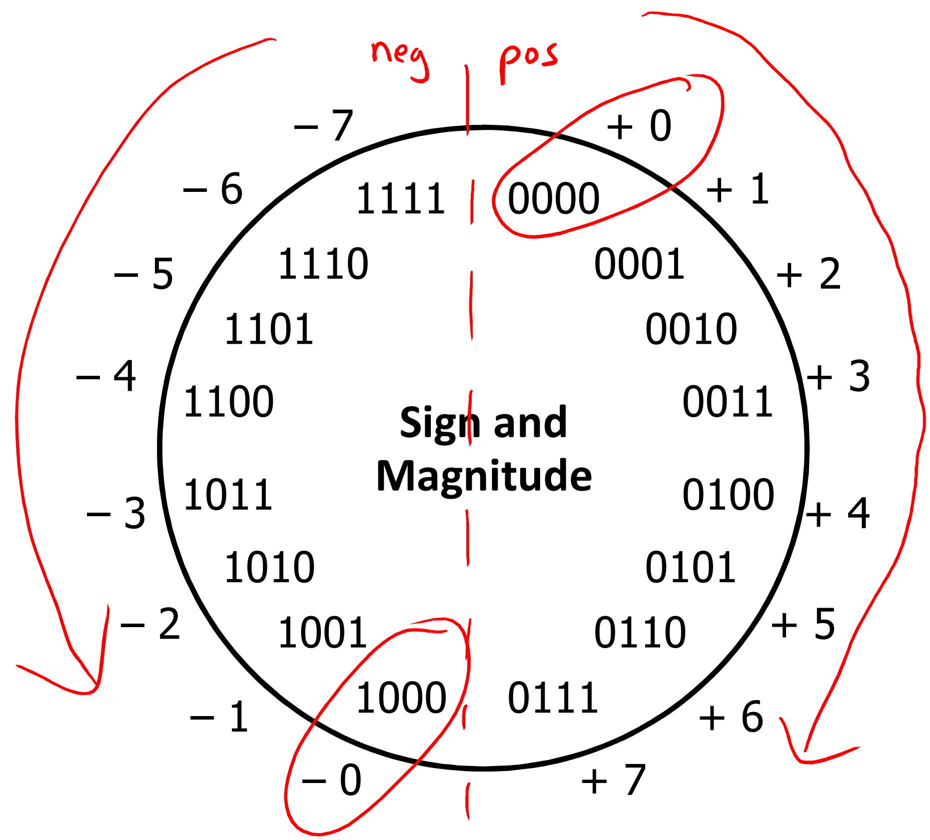
Most Significant Bit

- 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 unsigned: 0b 0010 = $2^1$ = 2 ; sign + mag: 0b 0010 = $+2^1$ = 2 ☑
  - 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

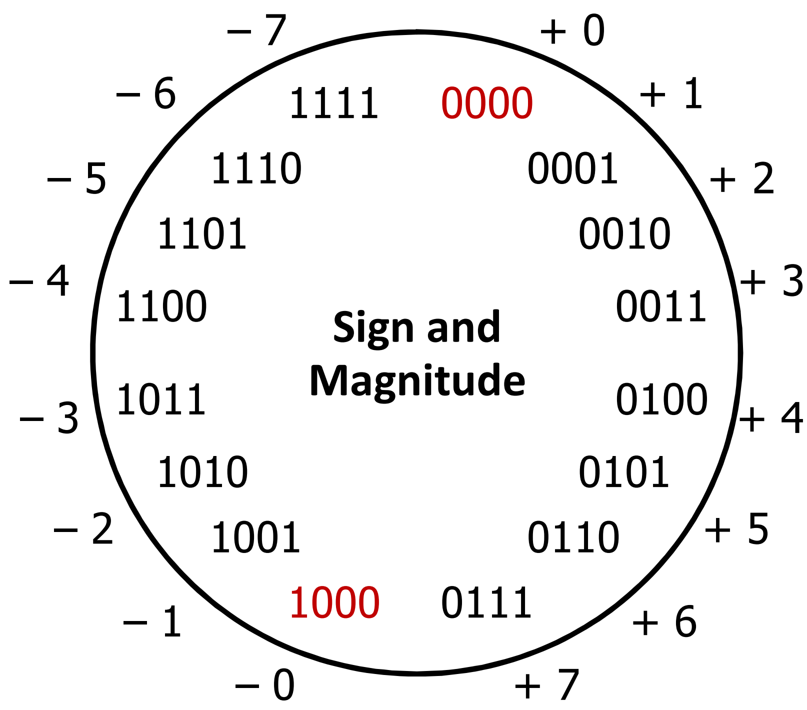
- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?



# Sign and Magnitude

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



# Sign and Magnitude

- 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)`

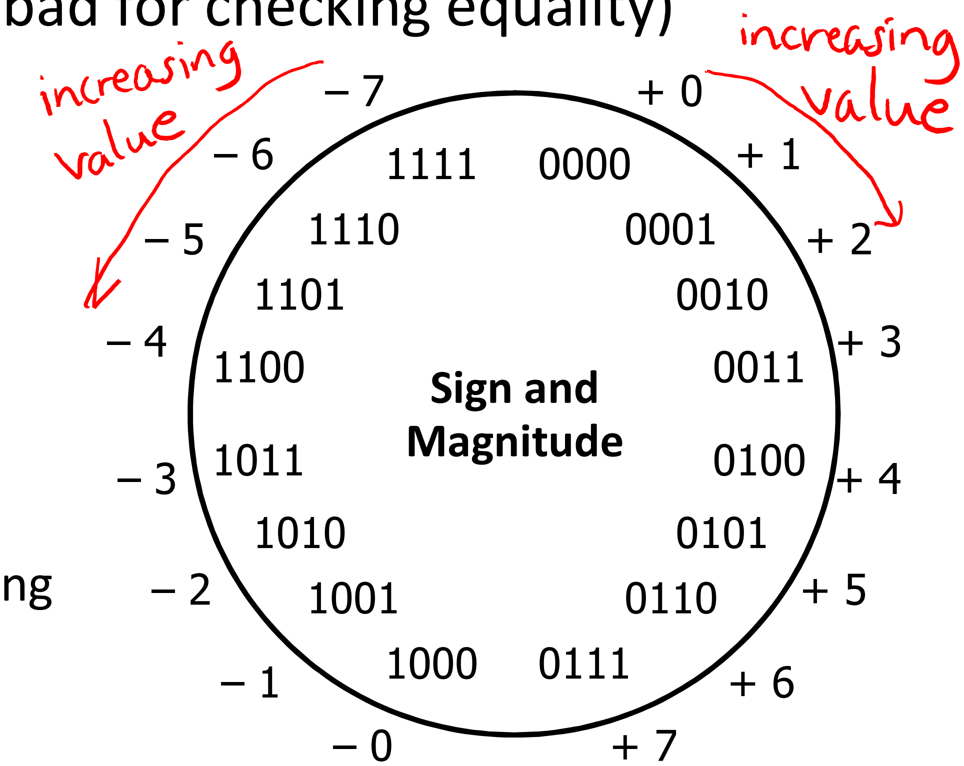
| | |
|---|---|
| 4 | 0100 |
| − 3 | − 0011 |
| 1 | 0001 |

✓

| | |
|---|---|
| 4 | 0100 |
| + −3 | + 1011 |
| −7 | 1111 |

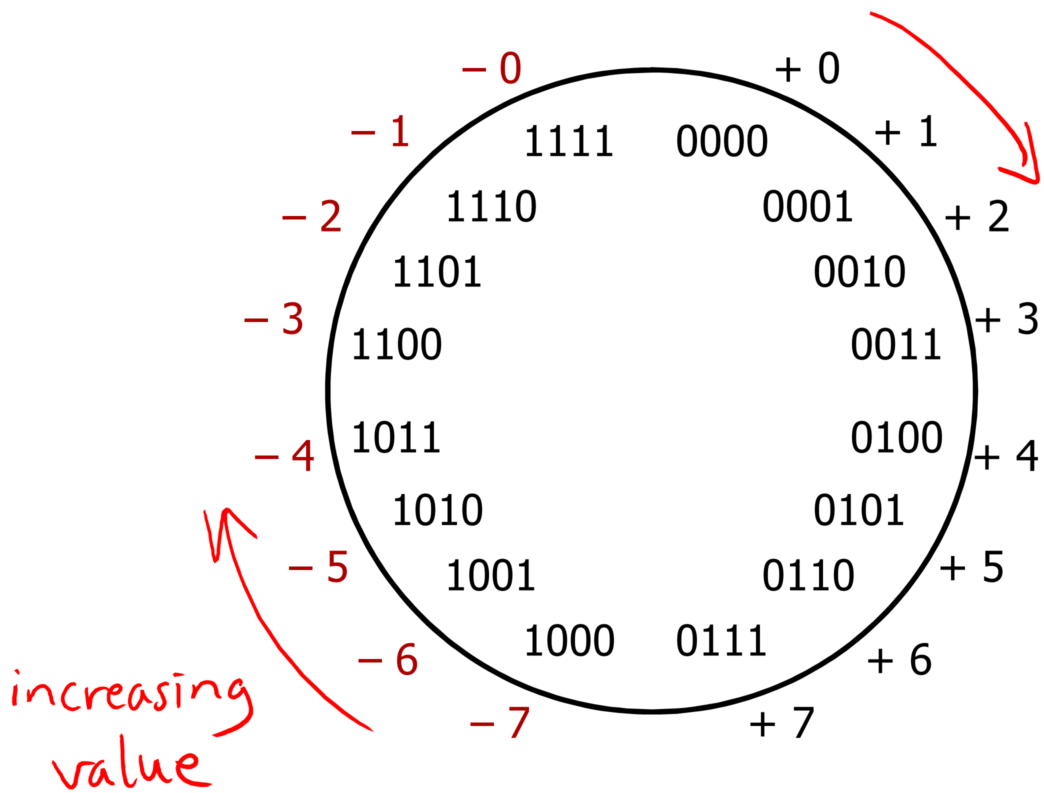
✗

    - Negatives "increment" in wrong direction!



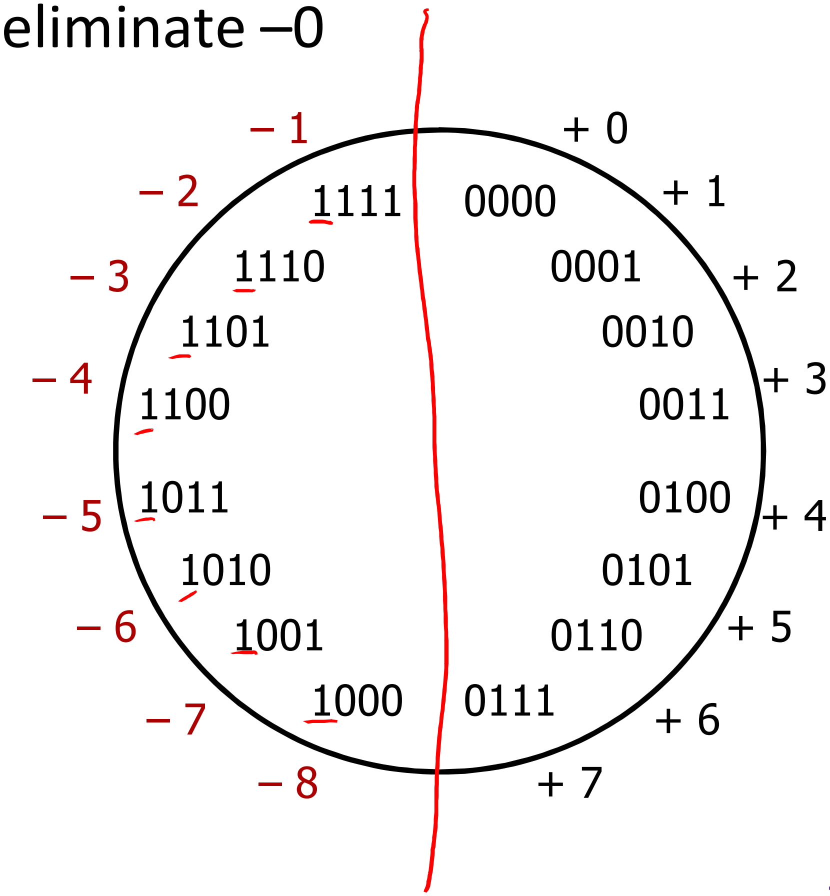
# Two's Complement

- 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 $2^{N-1}$ $-1$)



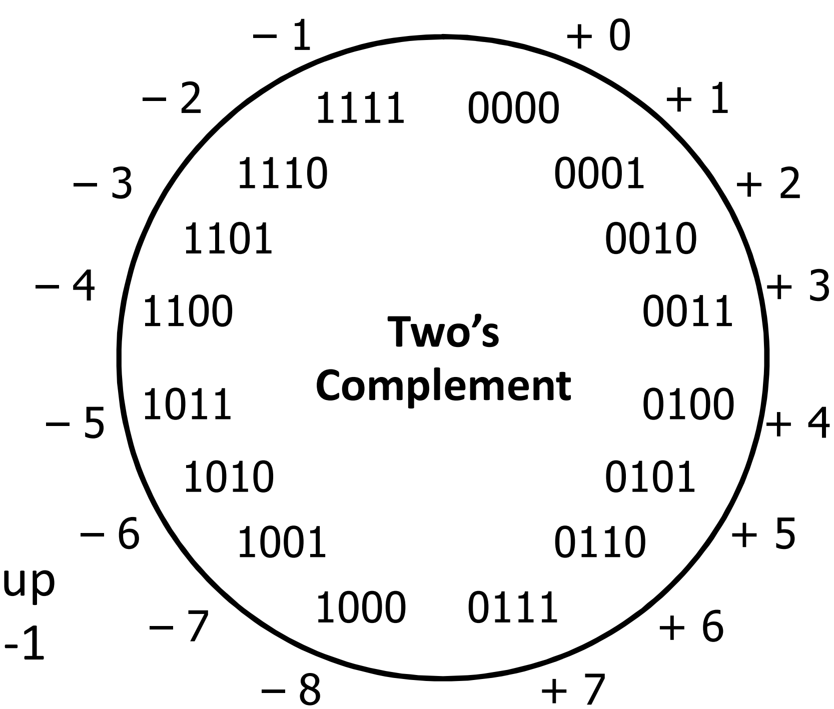
# Two's Complement Negatives

- 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

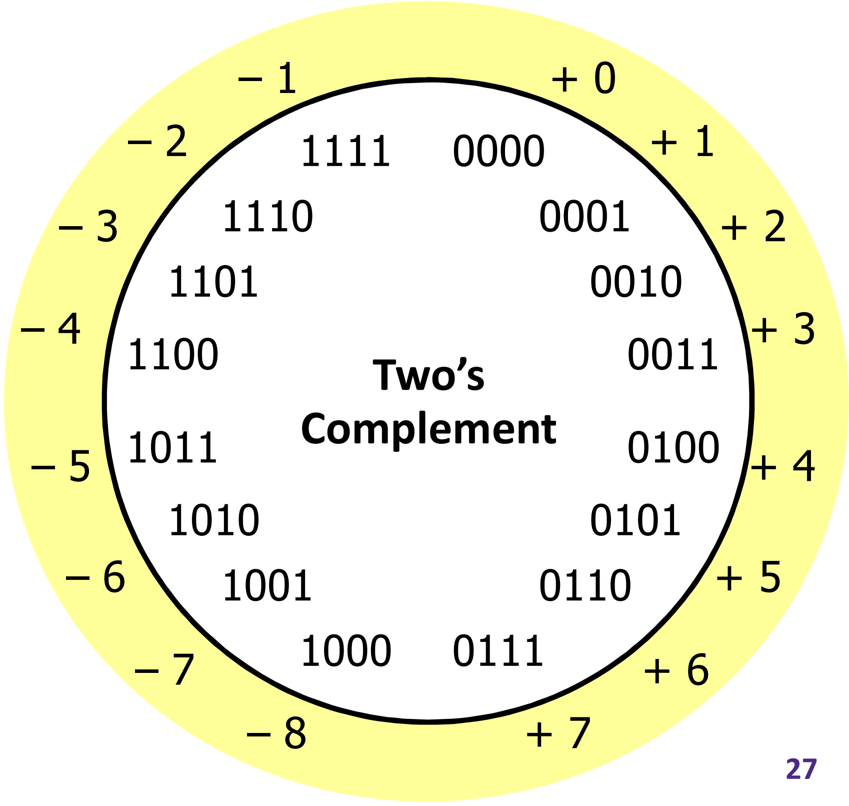
3 one's in a row



# 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 )`



# Peer Instruction Question

MSB

- 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 at http://PollEv.com/rea

A. -4

B. ~~-5~~

C. ~~11~~

D. ~~-3~~

E. We're lost…

unsigned: 8 + 2 + 1 = 11

sign + mag: 1011 → −(2+1) = −3

two's: −8 + 2 + 1 = −5

−x = 0b 0100+1 = 5 → x = −5

# 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