# Data III & Integers I

CSE 351 Autumn 2020


**Instructor:**

Justin Hsia

**Teaching Assistants:**

Aman Mohammed
Ami Oka
Callum Walker
Cosmo Wang
Hang Do
Jim Limprasert
Joy Dang
Julia Wang
Kaelin Laundry
Kyrie Dowling
Mariam Mayanja
Shawn Stanley
Yan Zhe Ong

http://xkcd.com/257/

# Administrivia

- hw3 due Friday, hw4 due Monday

- 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 ptest (`./ptest`) and check for correct behavior
    4) Run rule/syntax checker (`python3 dlc.py`) and check output
  - Due Monday 10/12, 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 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 (`&`, `|`, `^`, `~`)
  - 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;`
  - `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 $\bar{b}$):

$b \;\&\; 0 = ___$ $\qquad$ $b \;\&\; 1 = ___$

$b \mid 0 = ___$ $\qquad$ $b \mid 1 = ___$

$b \;\wedge\; 0 = ___$ $\qquad$ $b \;\wedge\; 1 = ___$

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

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

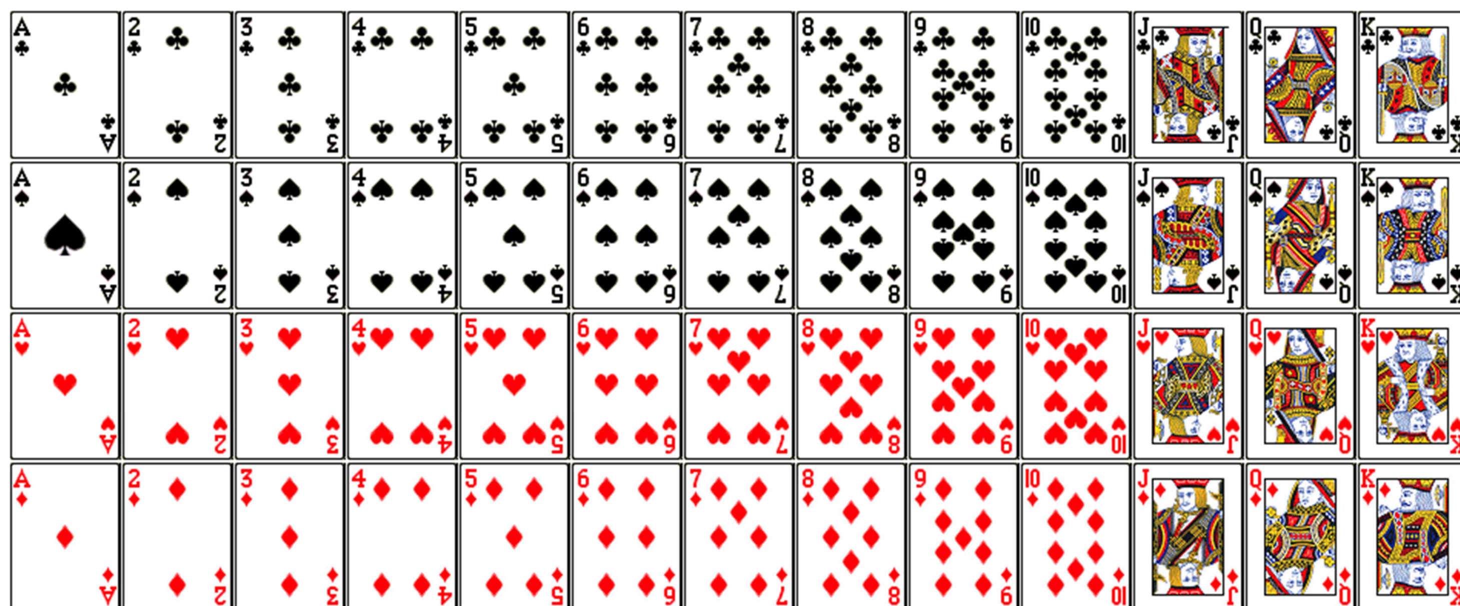
# Short-Circuit Evaluation

- 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

# 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?



# Representations and Fields

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

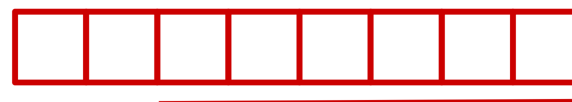
- "One-hot" encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

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

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

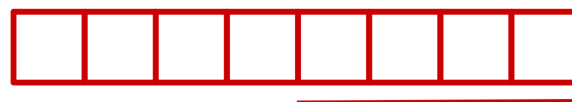
# Representations and Fields

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) ) { ... }
```

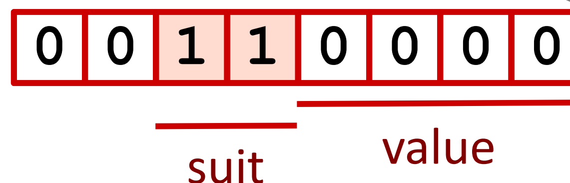
```
#define SUIT_MASK  0x30

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

returns `int`

equivalent

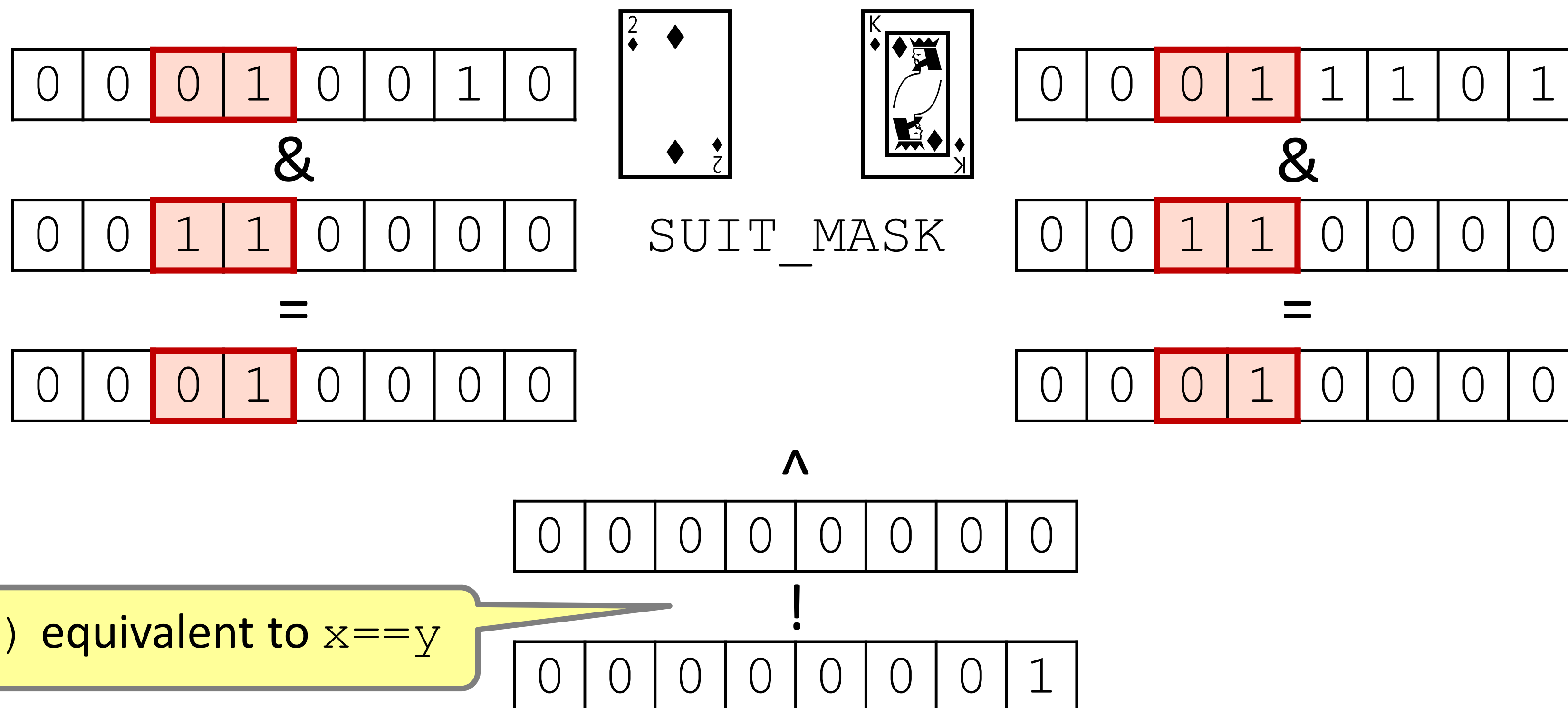
SUIT_MASK = 0x30 =



# 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 =

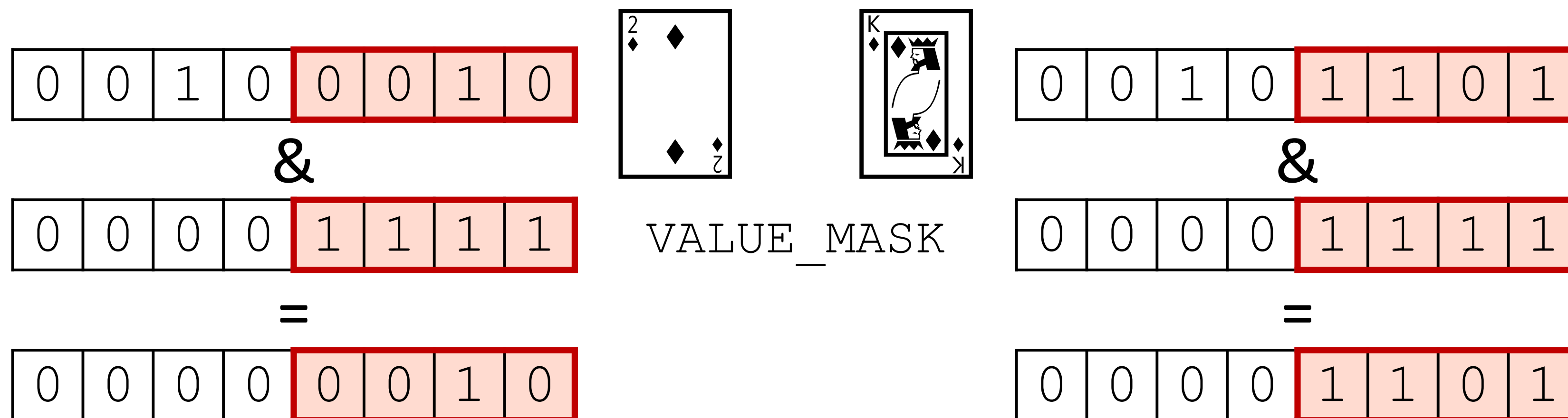
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

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)

# 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

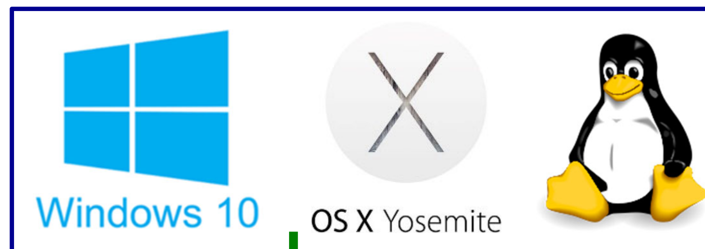
Assembly language:

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

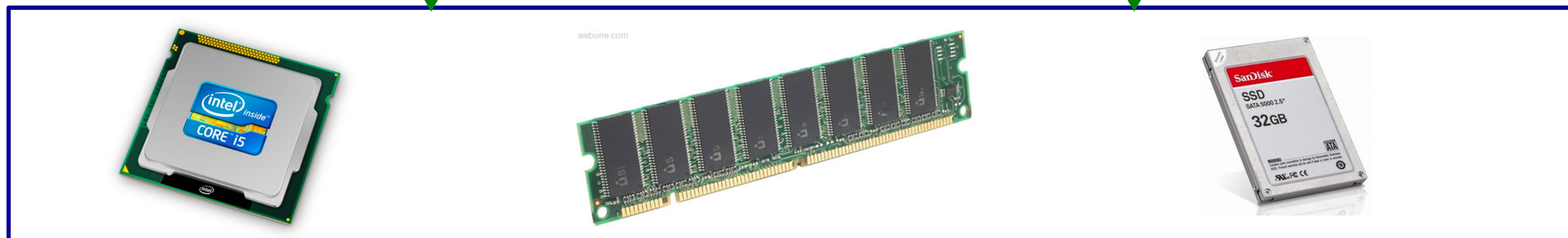
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

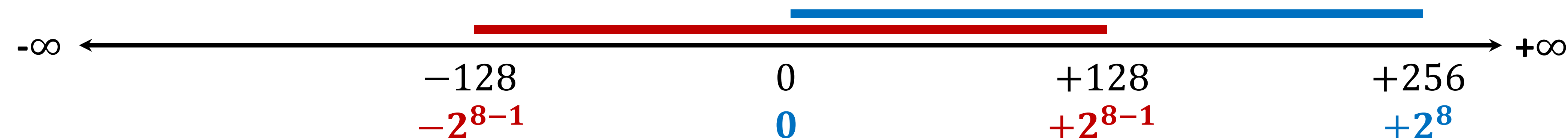



Computer system:



# 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
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$
- Useful formula: $2^{N-1} + 2^{N-2} + \ldots + 2 + 1 = 2^N - 1$
  - *i.e.*, N ones in a row = $2^N - 1$
  - *e.g.*, 0b111111 = 63
- How would you make *signed* integers?

# Sign and Magnitude

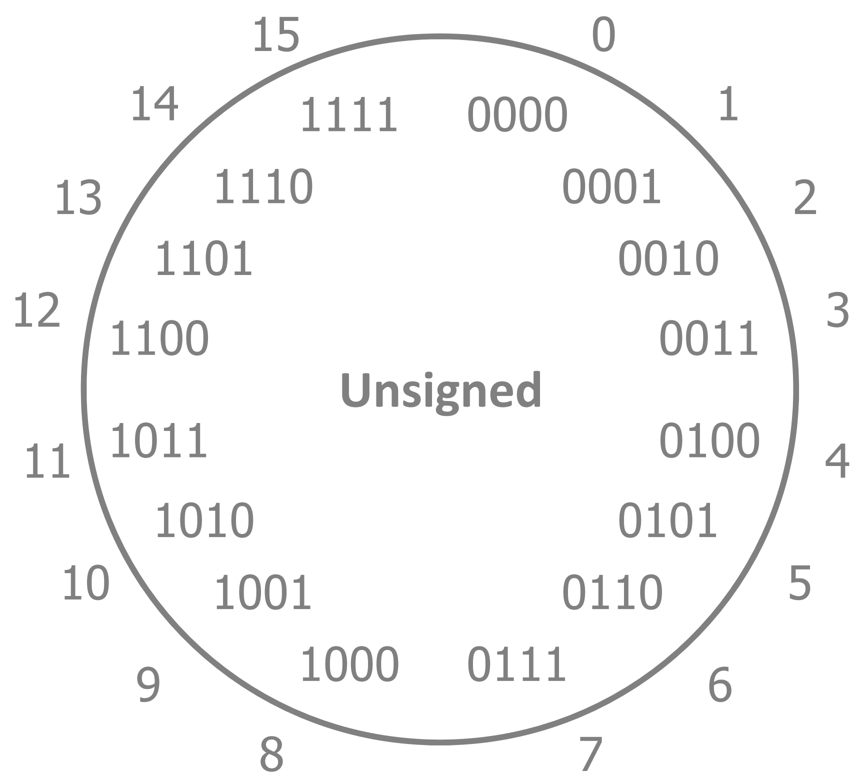
Not used in practice!

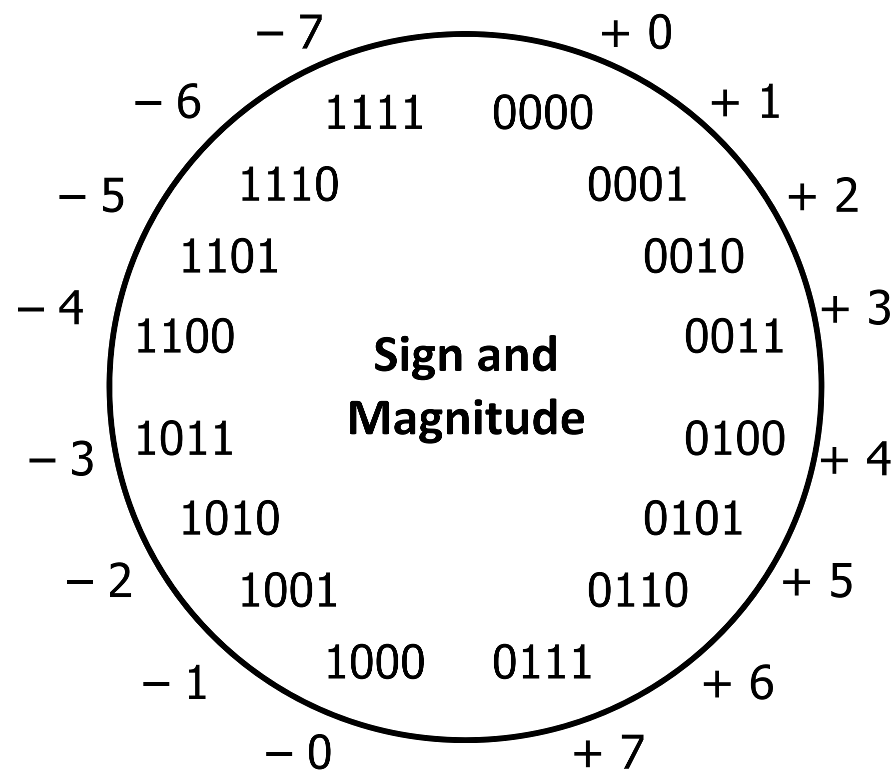
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive numbers; `sign=1`: negative numbers
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0
- Examples (8 bits):
  - 0x00 = $00000000_2$ is non-negative, because the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

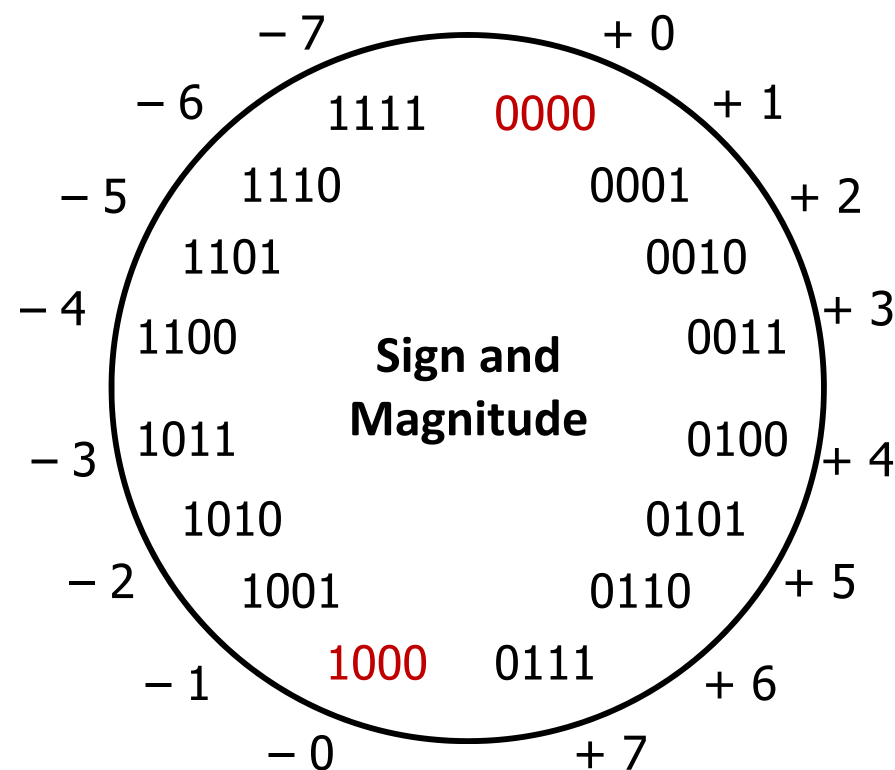


# Sign and Magnitude

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

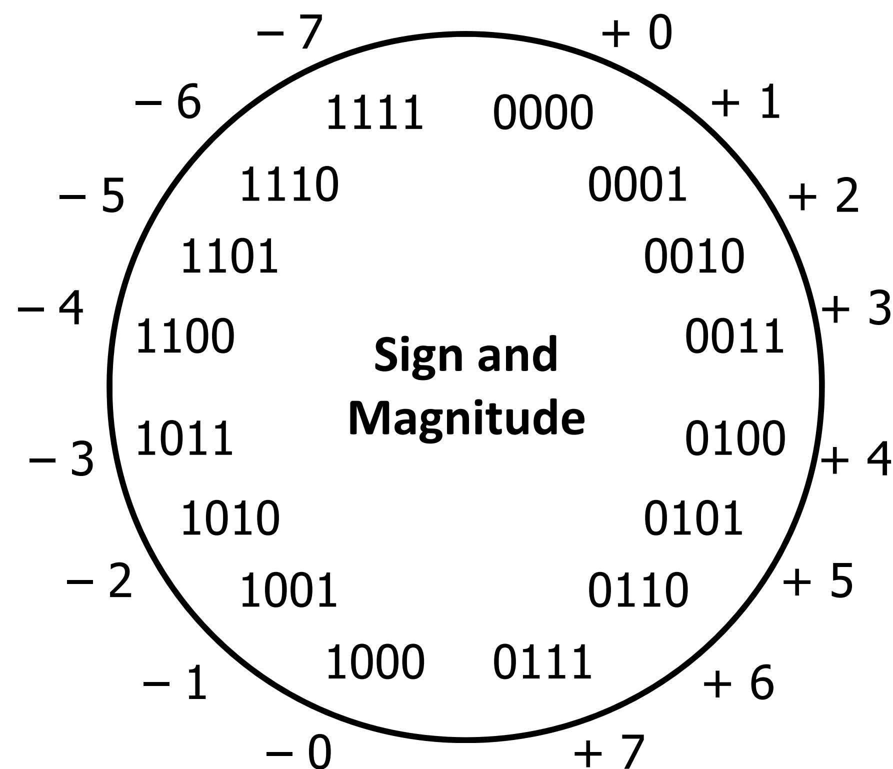
Not used in practice!

- 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 |

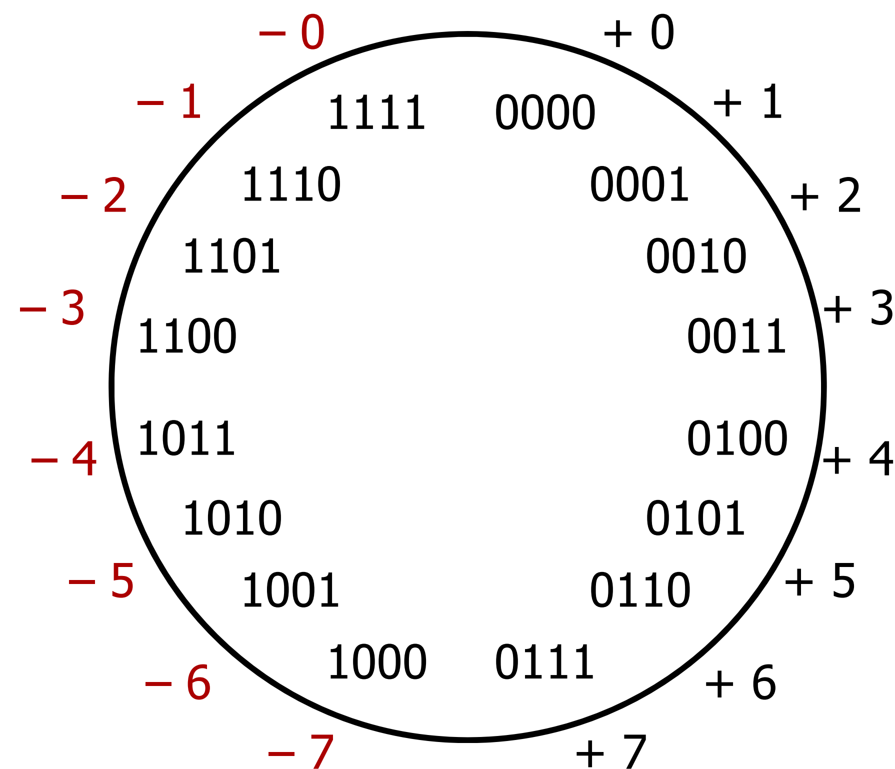
| | |
|---|---|
| 4 | 0100 |
| + −3 | + 1011 |
| −7 | 1111 |

    - Negatives "increment" in wrong direction!



# Two's Complement

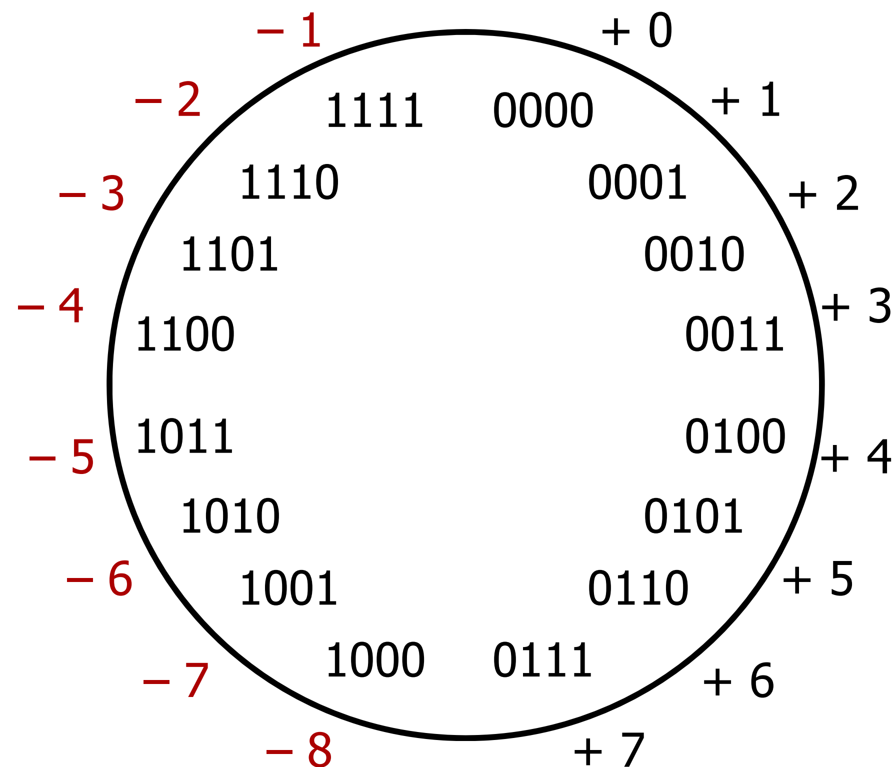
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



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

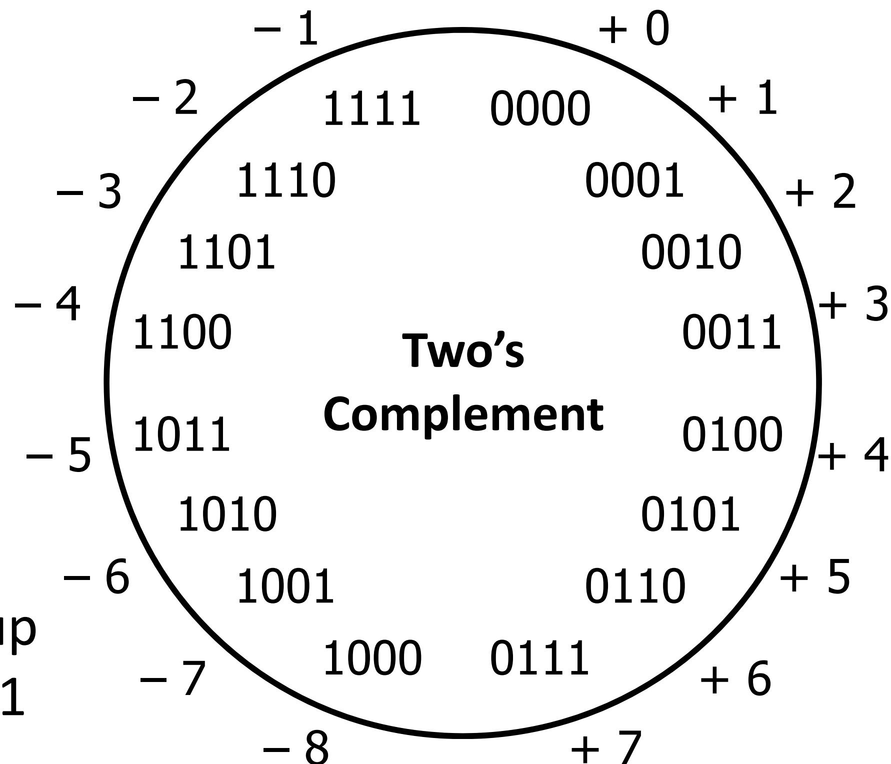


# Two's Complement Negatives

- Accomplished with one neat mathematical trick!

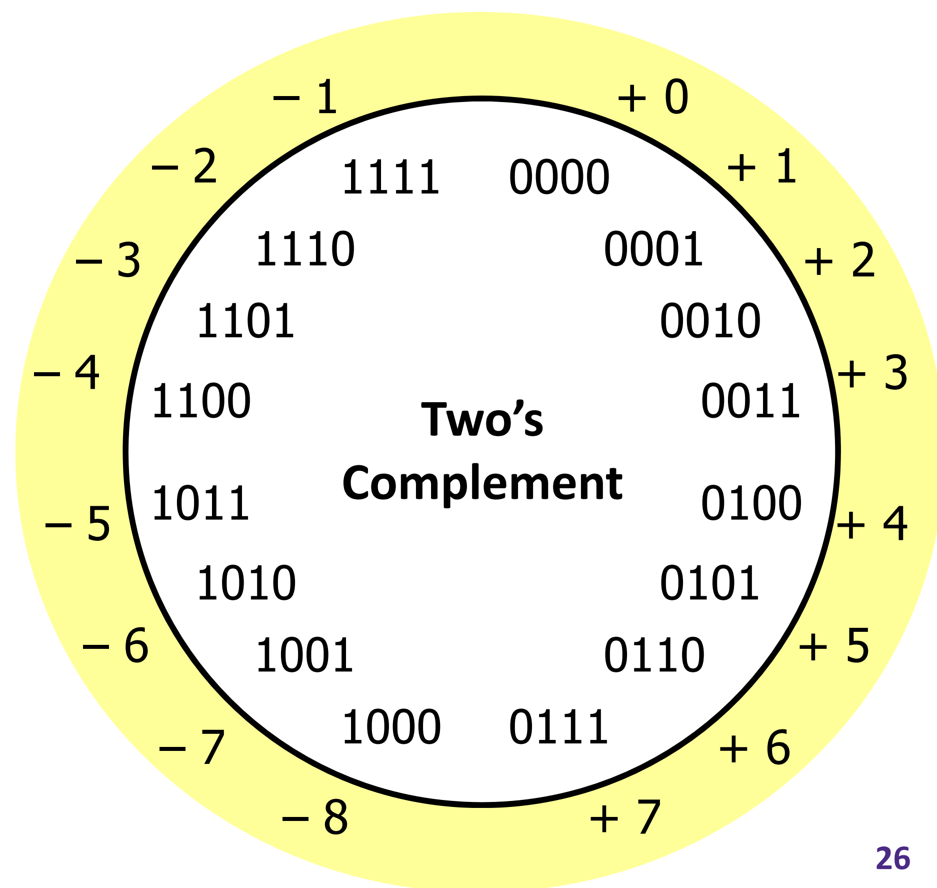
$b_{w-1}$ has weight $-2^{w-1}$, other bits have usual weights $+2^i$

- 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