Data III & Integers I

CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/257/

Relevant Course Information

- hw3 due Friday, hw4 due Monday
- Lab 1a released
  - Some later functions require *bit shifting*, covered in Lec 5
  - 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 10/10, will overlap a bit with Lab 1b
    - We grade just your *last* 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 (&, |, ^, ~)
  - 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;` // 0b 1000 0001
  - `c ^ c` = 0x00
    1000 0001
    1000 0001
    0000 0000
  - `~c & 0xA9` = 0x28
    "true" "true" "true"
    0b 0111 1110
    0b 1010 1001
    0010 1000
  - `c || 0x80` = 0x01
    "true"
  - `!(!c)` = 0x01
    "false"
    0x0
- Compute the value of `signed char sc = 0xF0;` (Two's Complement)
  = 0b 1111 0000
  $= -2^7+2^6+2^5+2^4$
  = -16
  -sc = ~sc + 1 = 0b 0000 1111 + 1
  0b 0001 0000 = +16
  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

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

0/1

b & 0 = 0 "set to zero" | b & 1 = b "keep as is"

b | 0 = b "keep as is" | b | 1 = 1 "set to one"

b ^ 0 = b "keep as is" | b ^ 1 = $\bar{b}$ "flip"

(0 → 0, 1 → 0) (0 → 0, 1 → 1)

(0 → 0, 1 → 1) (0 → 1, 1 → 1)

(0 → 0, 1 → 1) (0 → 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

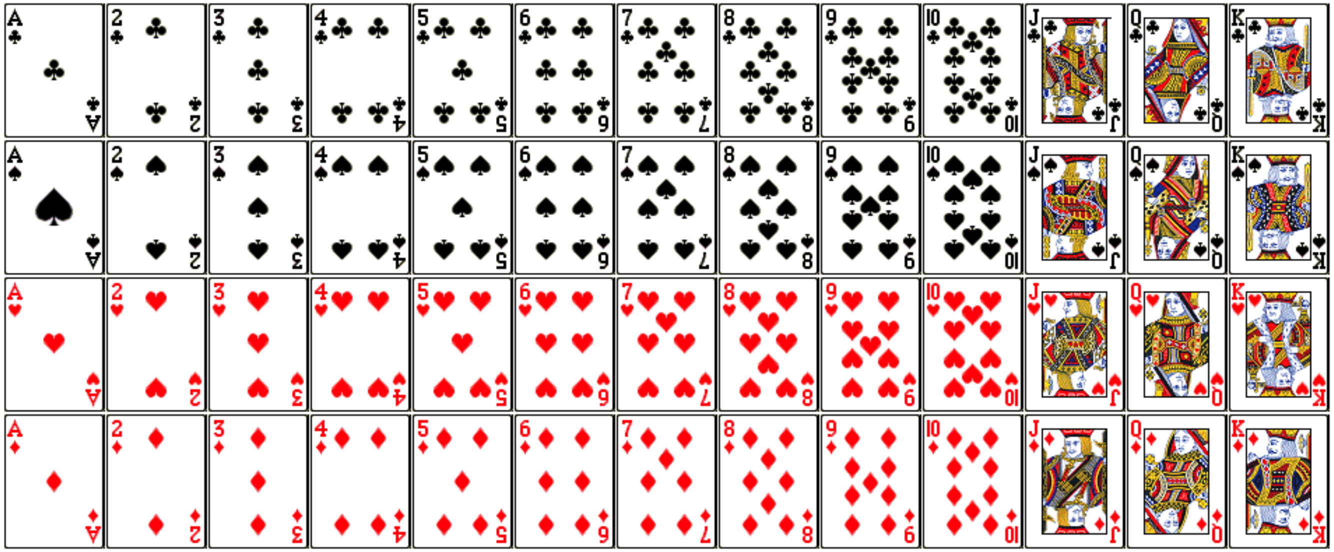
- Example: $b|0 = b,\ b|1 = 1$

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

"set to one" "keep as is"

Numerical Encoding Design Example

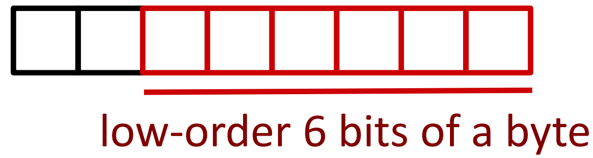
- Encode a standard deck of playing cards
  - 52 cards in 4 suits
- Operations to implement:
  - Which is the higher value card?
  - Are they the same suit?



Representations and Fields

1) Binary encoding of all 52 cards – only 6 bits needed
   - $2^6 = 64 \geq 52$

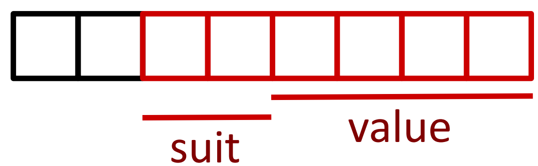
     $2^5 = 32 < 52$



   - Fits in one byte
   - How can we make value and suit comparisons easier?

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

   4 suits   13 values



   - 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 ( same_suit(card1, card2) ) { ... }
```

text substitution

```
#define SUIT_MASK  0x30

int same_suit(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)

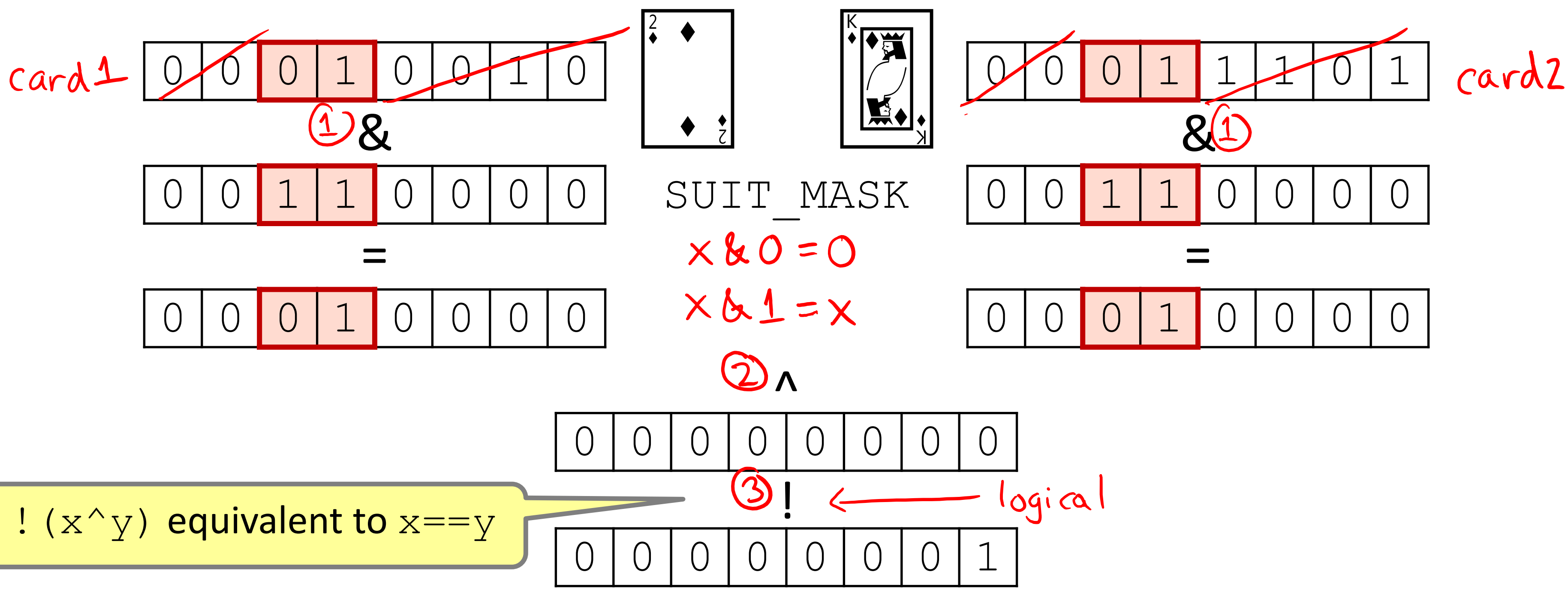
x & 0 = 0

x & 1 = x

Compare Card **Suits**

```
#define SUIT_MASK  0x30

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



! (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 ( greater_value(card1, card2) ) { ... }
```

```
#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 (discard) value (keep)

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));
}
```

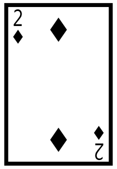
0	0	1	0	0	0	1	0

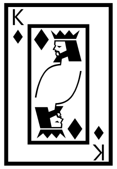
&

0	0	0	0	1	1	1	1

=

0	0	0	0	0	0	1	0





VALUE_MASK

0	0	1	0	1	1	0	1

&

0	0	0	0	1	1	1	1

=

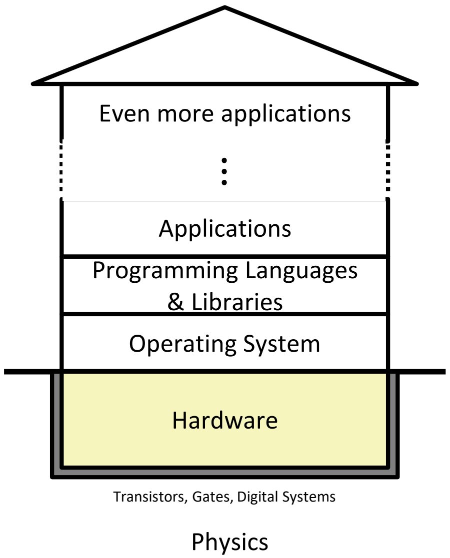
0	0	0	0	1	1	0	1

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

0 (false)

The Hardware/Software Interface

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

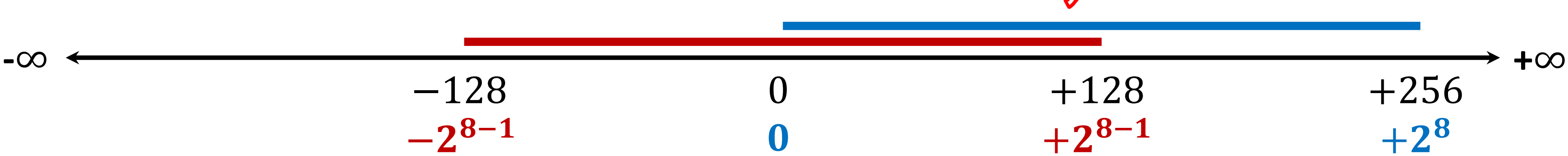


- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?

Encoding Integers

- The hardware (and C) supports two flavors of integers
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives
- 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`)

same widths, just shifted



Unsigned Integers (Review)

- Unsigned values follow the standard base 2 system
  - $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$
- 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 ← x, 6 1's in a row

x+1 = 0b1 000 000
= 2^6

$x = 2^6 - 1$

Sign and Magnitude

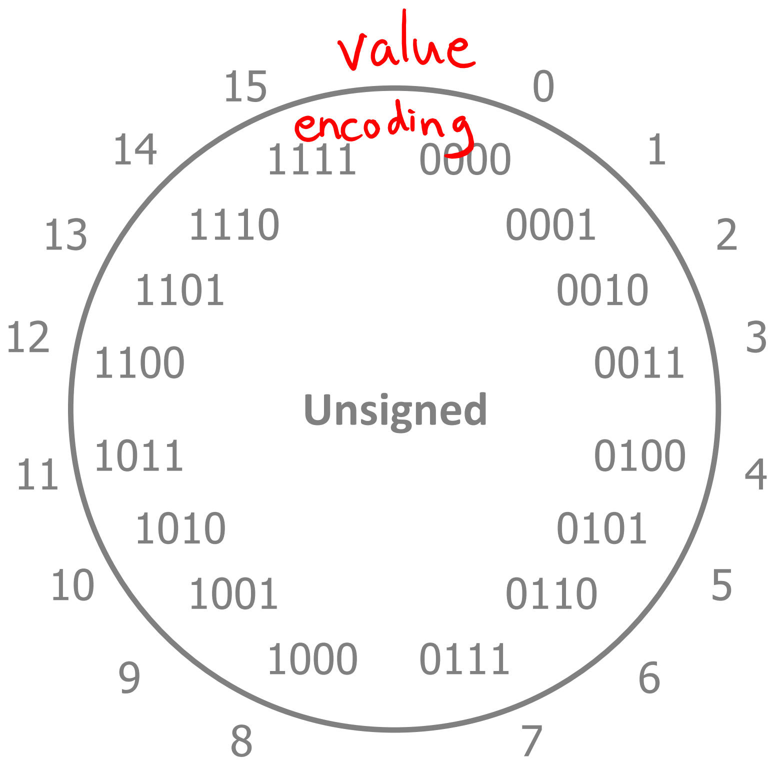
Not used in practice for integers!

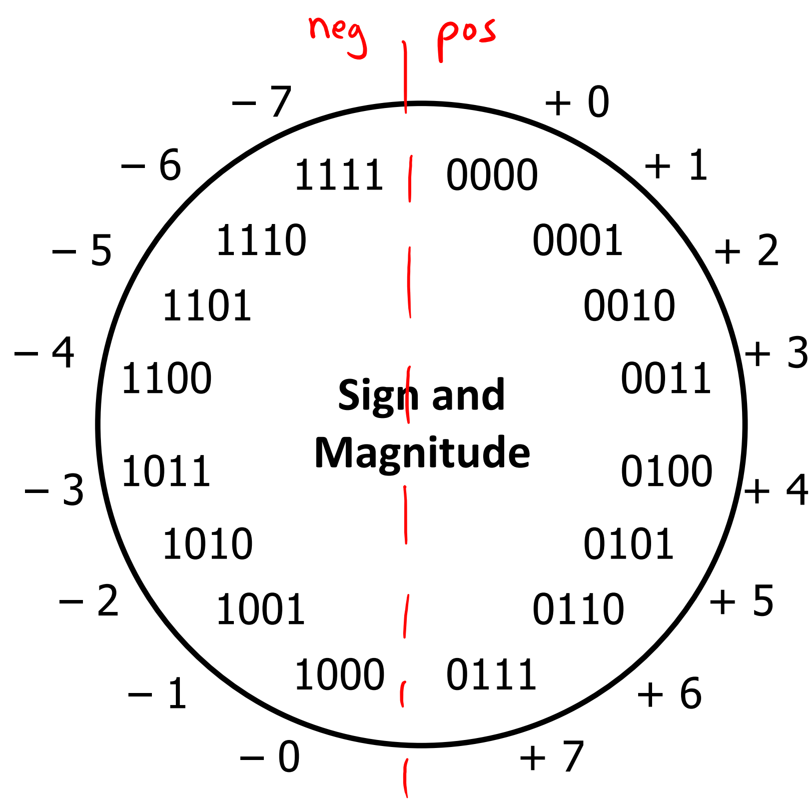
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive numbers; `sign=1`: negative numbers
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned   unsigned: 0b 0010 = 2^1 = 2 ; sign+mag: 0b 0010 = $+2^1$ = 2 ☑
  - All zeros encoding is still = 0
- Examples (8 bits):
  - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
  - 0x7F = 01111111_2 is non-negative ($+127_{10}$) 2^7-1
  - 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?

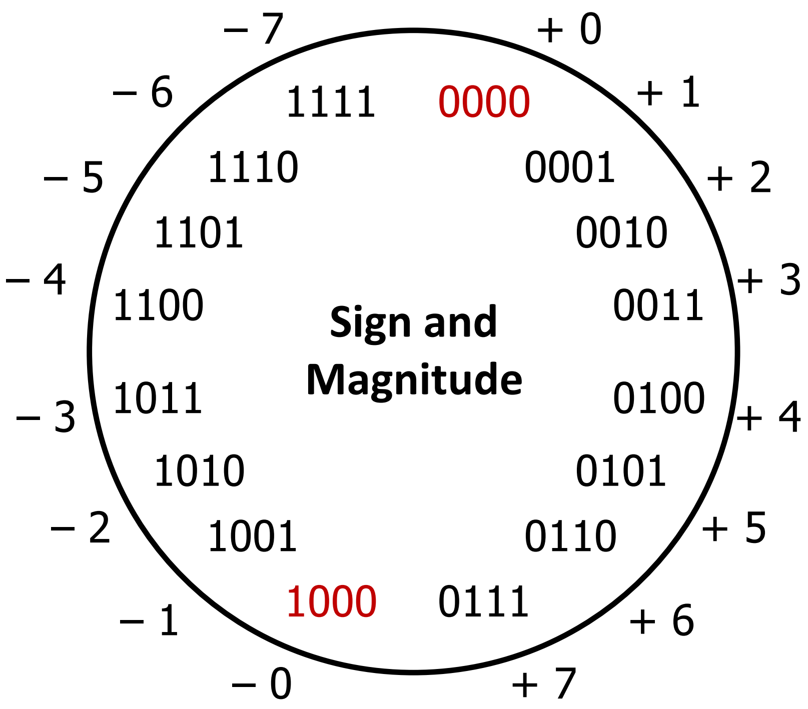




Sign and Magnitude

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)`

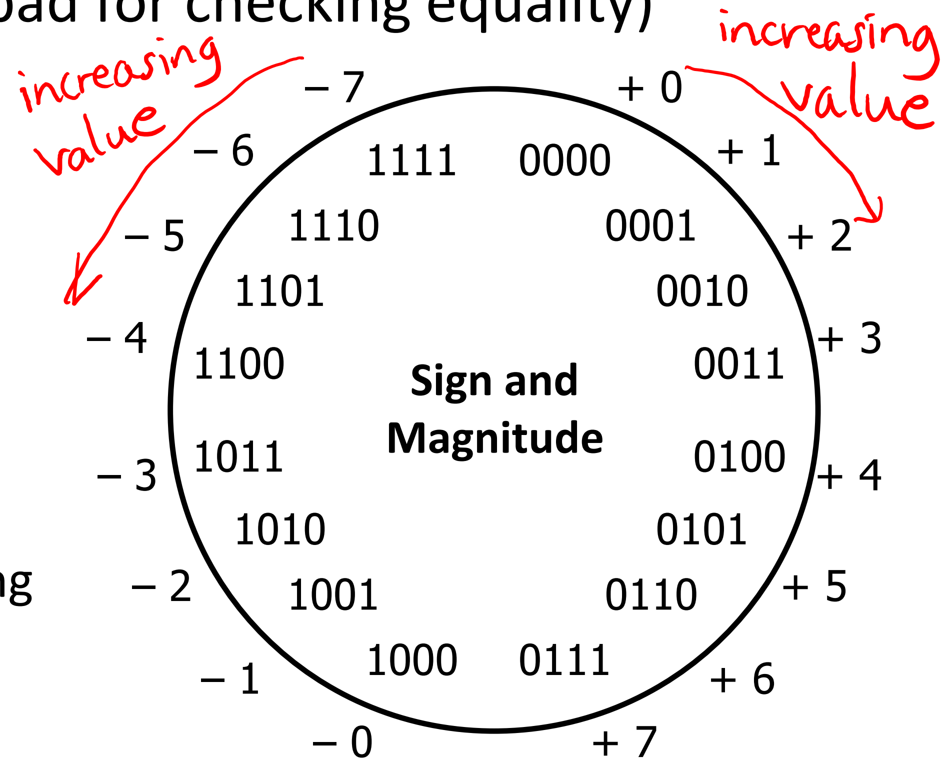
4	0100
− 3	− 0011
1	0001

✓

4	0100
+ −3	+ 1011
−7	1111

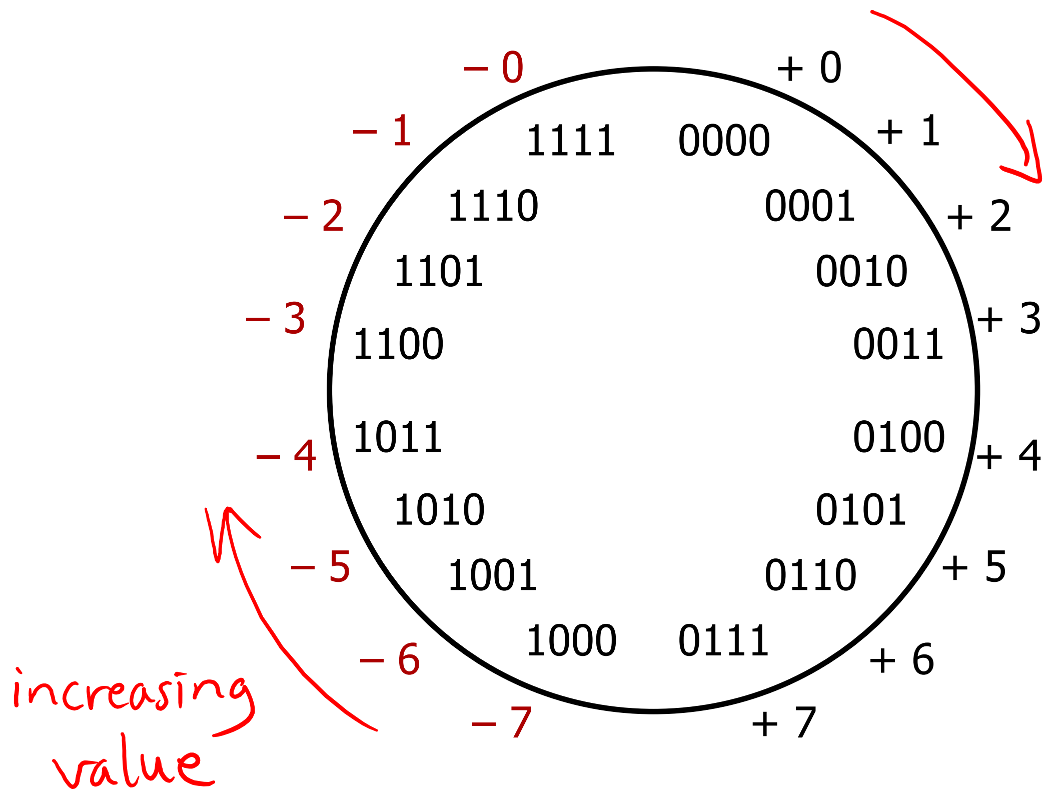
✗

    - Negatives "increment" in wrong direction!



Two's Complement

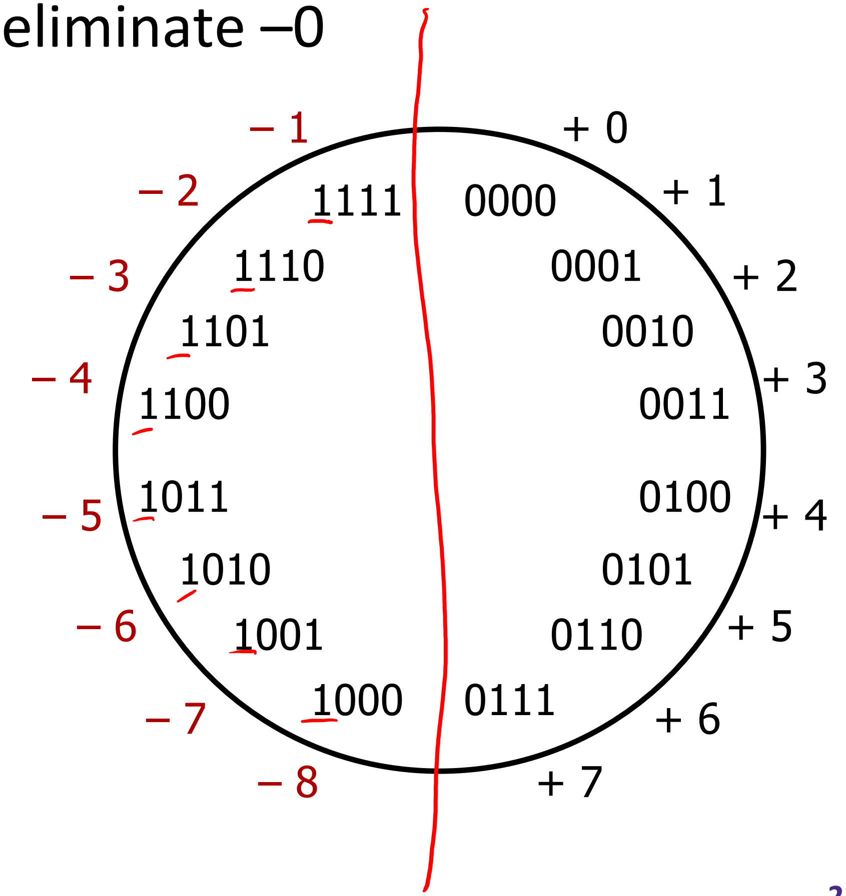
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



Two's Complement

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

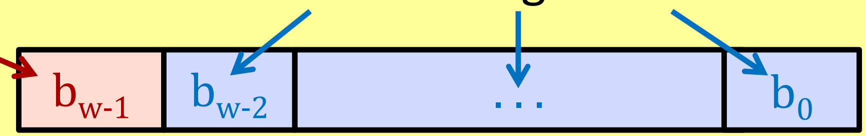
- MSB *still* indicates sign!
  - This is why we represent one more negative than positive number (-2^{N-1} to 2^{N-1} –1)



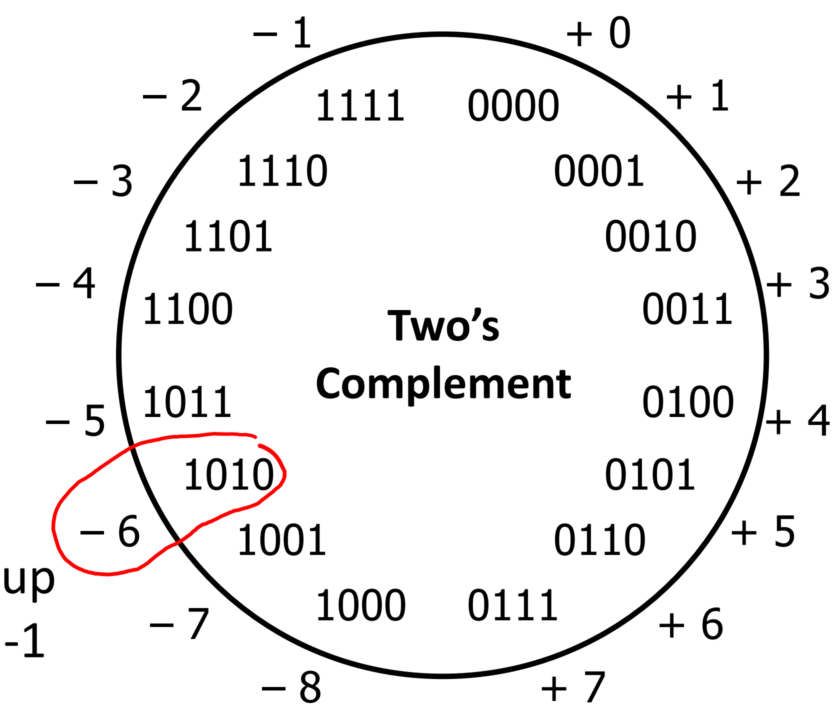
Two's Complement Negatives (Review)

- Accomplished with one neat mathematical trick!

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



- 4-bit Examples:
  - 8 2
  - 1010_2 unsigned:
    $1*2^3+0*2^2+1*2^1+0*2^0$ = **10**
  - -8
  - 1010_2 two's complement:
    $-1*2^3+0*2^2+1*2^1+0*2^0$ = **–6**
- -1 represented as:
  $1111_2 = -2^3+(2^3-1)$
  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

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 in Ed Lessons

A. -4

~~**B. -5**~~

~~**C. 11**~~

~~**D. -3**~~

E. We're lost…

unsigned: $8 + 2 + 1 = 11$

sign + mag: $1011 \rightarrow -(2+1) = -3$

two's: $-8 + 2 + 1 = -5$

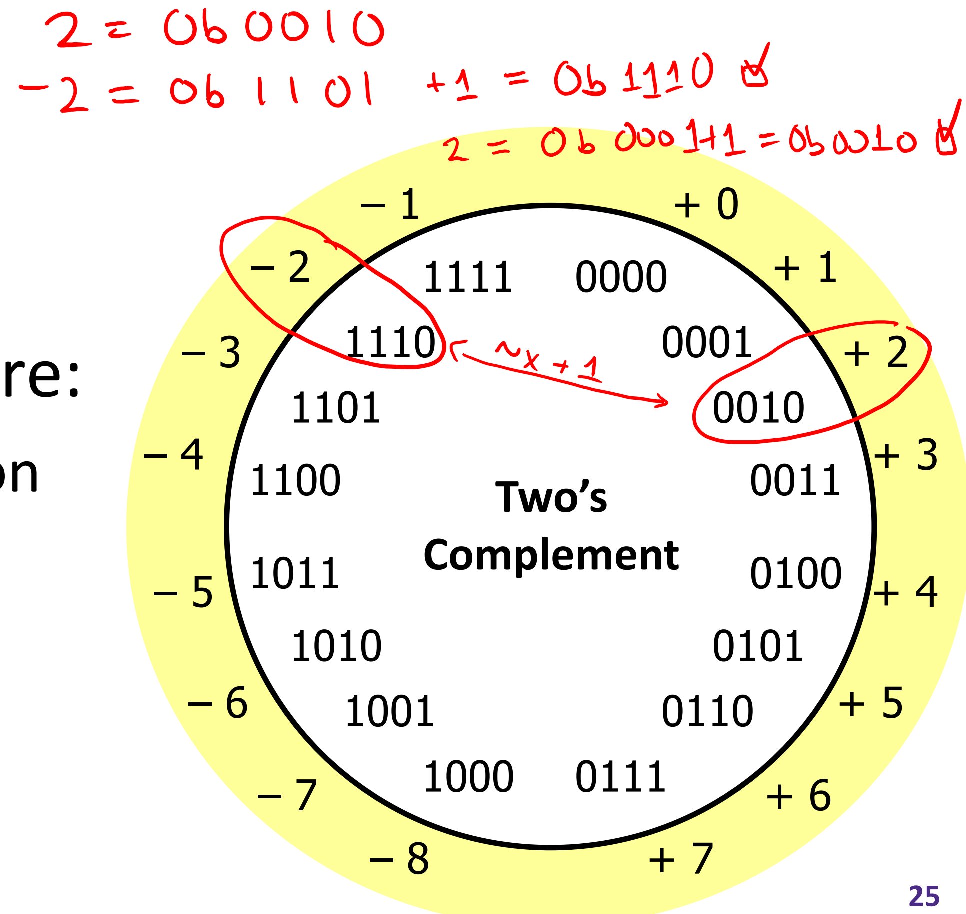
$-x = 0b\ 0100 + 1 = 5 \rightarrow x = -5$

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