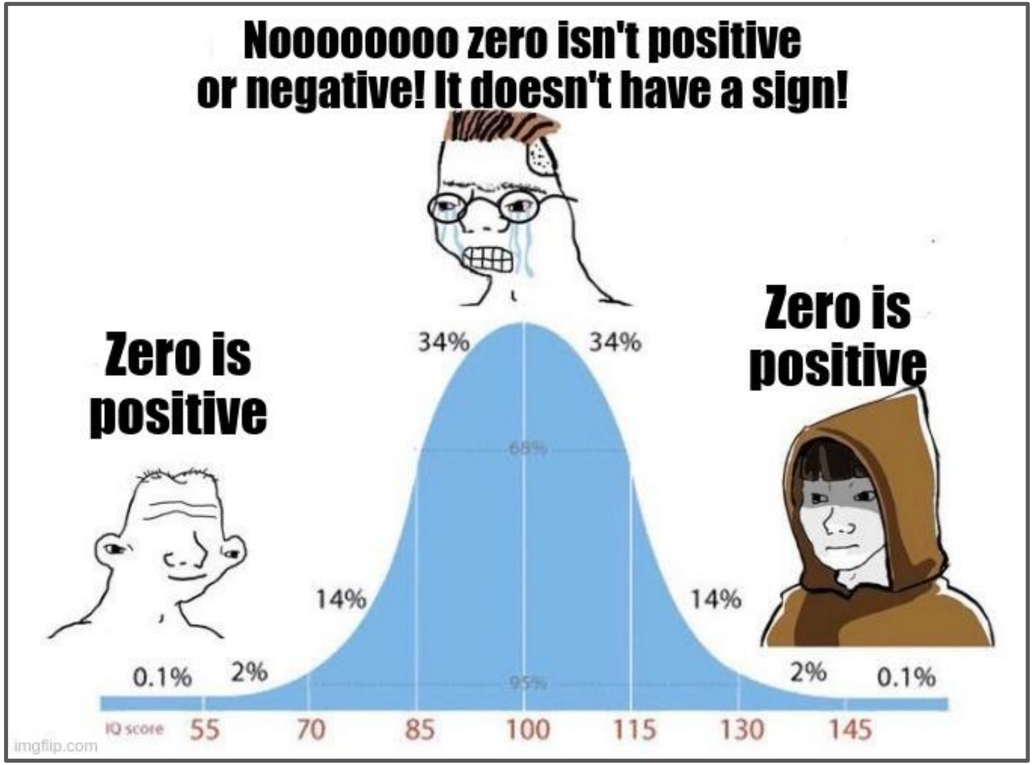
Data III, Integers I

CSE 351 Summer 2024

Instructor:
Ellis Haker

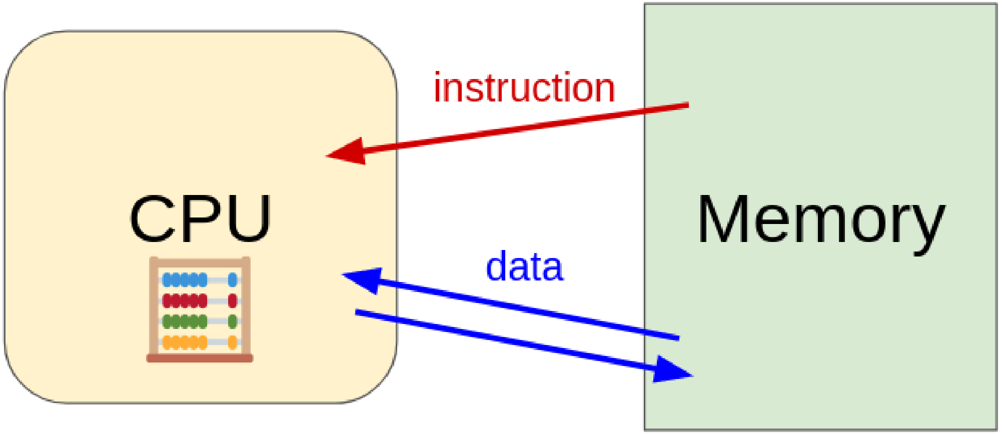
Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- HW 2 due today (11:59pm)
- Due Friday:
  - RD 5 (1pm)
  - HW 3 (11:59pm)
- Due Monday:
  - RD 6 (1pm)
  - HW 4 (11:59pm)

Recap: CPU and Memory



a. How does the CPU find its data in memory?
b. How are common C types encoded?
c. How can we use C to manipulate data in memory?

Review Questions

bitwise XOR

0b10000001

1. Compute the result of the following expressions for `char c = 0x81;`
   - `c ^ c` - anything XOR itself = $\boxed{0}$
   - `~c & 0xA9` ~c = 0b01111110 → & w/ 0b10101001 = 0b00101000 = $\boxed{0x28}$
   - `c || 0x80` c and 0x80 are both True, True OR True = True = $\boxed{1}$
   - `!!c` c = True, so NOT c = False, NOT(NOT c) = True = $\boxed{1}$

bitwise AND

bitwise NOT

logical OR

logical NOT

2. Compute the decimal value of `signed char sc = 0xF0;` (using 2's complement)

2 ways:

a) 0xF0 = 0b11110000 = $-2^7+2^6+2^5+2^4 = \boxed{-16}$

negative in 2's comp

b) ~0xF0 +1 = 0b00001111+1
= 0b00010000 = 16

-sc = 16, so sc = $\boxed{-16}$

Logical Operators (Review)

- No boolean type in C by default
  - All non-zero values are treated as “true,” zero is “false”
  - Result is always a 1 or 0
- AND (&&), OR (||), NOT (!)

&& (AND)	F	T
F	F	F
T	F	T

|| (OR)	F	T
F	F	T
T	T	T

! (NOT)	
F	T
T	F

Bitwise Operators (Review)

same as !=

- Apply the given operation (AND, OR, NOT, XOR) to *each bit* of a value separately
  - Ex: `0xA | 0x3` = `0b1010 | 0b0011` = `0b1011 = 0xB`

& (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

Bitmasks

- We can use binary bitwise operators (&, |, ^) along with a specially chosen **bitmask** in order to read or write to particular bits in a piece of data

Useful operations - for any bit *b* (answer with 0, 1, *b*, or ~*b*):

$b \,\&\, 0 =$ 0

$b \,\&\, 1 =$ b

set some bits to 0, keep the rest the same

$b \wedge 0 =$ b

$b \wedge 1 =$ ~b

negate some bits, keep the rest the same

$b \mid 0 =$ b

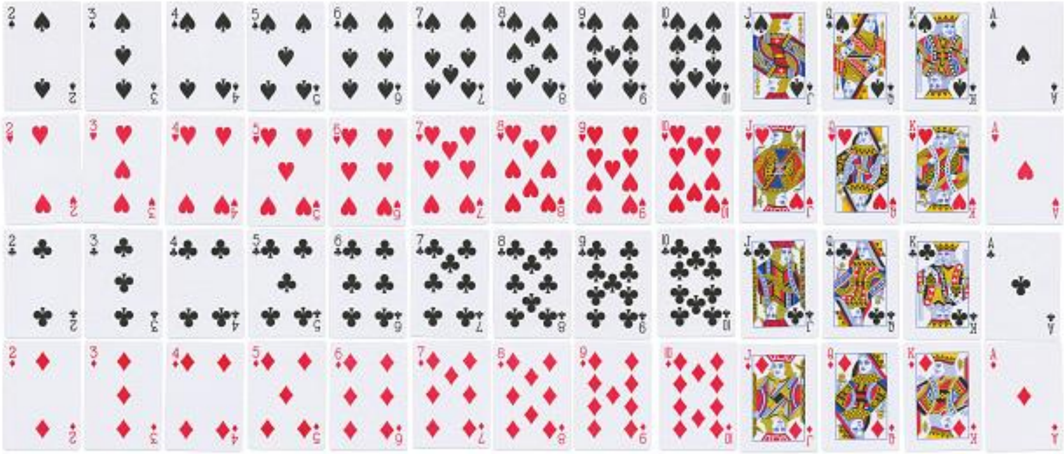
$b \mid 1 =$ 1

set some bits to 1, keep the rest the same

Numerical Encoding Design Example

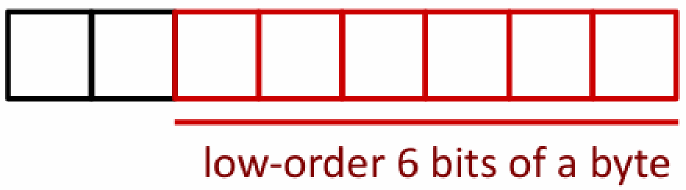
want to keep our representation ≤ 1 byte

- Encode a standard deck of playing cards
  - 4 suits, 13 cards each = 52 total
- Operations to implement:
  - Which card is of higher value?
  - Are they the same suit?
- **First: how to represent?**



Naive Approach

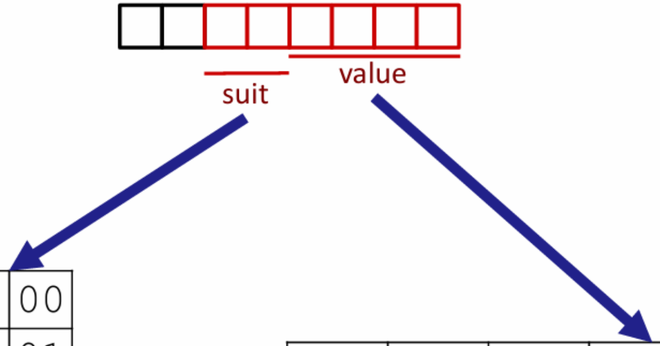
- Binary encoding of 52 cards - only 6 bits needed
    - $2^6 = 64 >= 52$
    - Fits in one byte
- Just count cards in binary
- **Problem:** hard to compare value & suit



Binary	Suit & Value
000000	Ace of Clubs
000001	Ace of Diamonds
000010	Ace of Hearts
000011	Ace of Spades
…	…
110010	King of Hearts
110011	King of Spades

Better Approach: Fields

- Separate binary encodings of suit (2 bits) and value (4 btis)
  - Still fits in one byte, easier to do comparisons 😎



♣	00
♦	01
♥	10
♠	11

encoding = numerical value (if we treat A as a low card)

K	Q	J	. . .	3	2	A
1101	1011	1010	. . .	0011	0010	0001

Compare Card Suits

b&1 = b   b&0 = 0

```c
#define SUIT_MASK = 0x30            // 0b00110000

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

negate the ^ to get ==

preserve suit, 0 out the rest

same as !=

SUIT_MASK = 0x30 =

0	0	1	1	0	0	0	0

suit: bits 1 1; value: bits 0 0 0 0

Compare Card Suits (pt 2)

```
return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
```

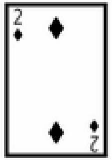
0	0	0	1	0	0	1	0

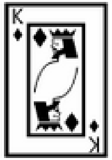
&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0





0	0	0	1	1	1	0	1

&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0

^

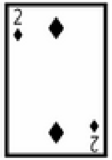
0	0	0	0	0	0	0	0

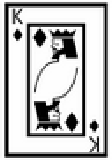
!

0	0	0	0	0	0	0	1

Compare Card Suits: Equivalent Technique

```
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

0	0	0	1	1	1	0	1

&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0

==

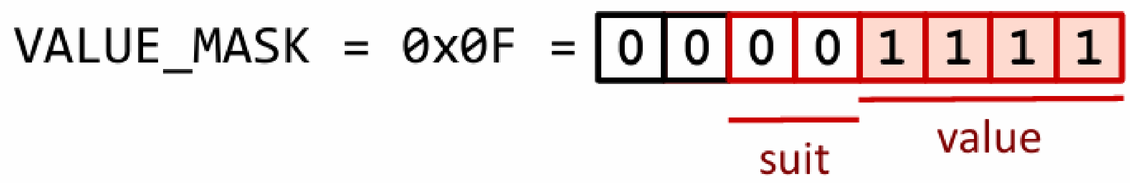
0	0	0	0	0	0	0	1

Compare Card Values

```
#define VALUE_MASK = 0x0F        // 0b00001111

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

preserve value, 0 out the rest



Compare Card Values

```
return ((unsigned int)(card1 & VALUE_MASK) >
    (unsigned int)(card2 & VALUE_MASK));
```

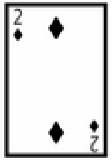
0	0	0	1	0	0	1	0

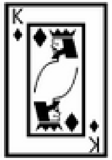
&

0	0	0	0	1	1	1	1

=

0	0	0	0	0	0	1	0





0	0	0	1	1	1	0	1

&

0	0	0	0	1	1	1	1

=

0	0	0	0	1	1	0	1

>

0	0	0	0	0	0	0	0

Note: CPU doesn't keep track of what type your variables are!

ex:

signed interpretation

```
printf("%d", -1)
```

prints -1

unsigned interpretation

```
printf("%u", -1)
```

reads the binary for -1, but interprets as unsigned, prints out a large positive number.

Integers

Encoding Integers (Review)

- The hardware (and C) supports two flavors of integers
  - **Unsigned** - only non-negative numbers
  - **Signed** - positive and negative numbers
- By default, C ints are signed → can specify (ex: unsigned int, signed long, etc.)
  - Java *only* supports signed
- Reminder: we cannot represent all integers in a finite number of bits!
  - If our data type is *w* bits wide, we have 2^w different encodings
  - Unsigned values: $0 \ldots 2^w - 1$
  - Signed values: $-2^{w-1} \ldots 2^{w-1} - 1$

Unsigned Integers (Review)

- Just like the binary->base 10 conversion from day 1

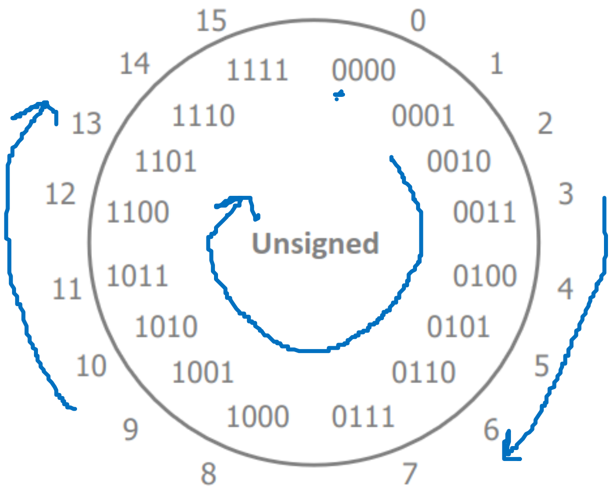
$$b_7b_6b_5b_4b_3b_2b_1b_0 = b_7{*}2^7 + b_6{*}2^6 + \ldots + b_1{*}2^1 + b_0{*}2^0$$

diagram shows binary increasing clockwise

- Arithmetic: just add like "normal"
  - If sum exceeds 1 bit, carry over to the next

<u>Ex:</u> 4+5 = 9

```
      1   <- "carry"
   0b0100
 + 0b0101
 ---------
 = 0b1001
```



How do we represent signed integers?

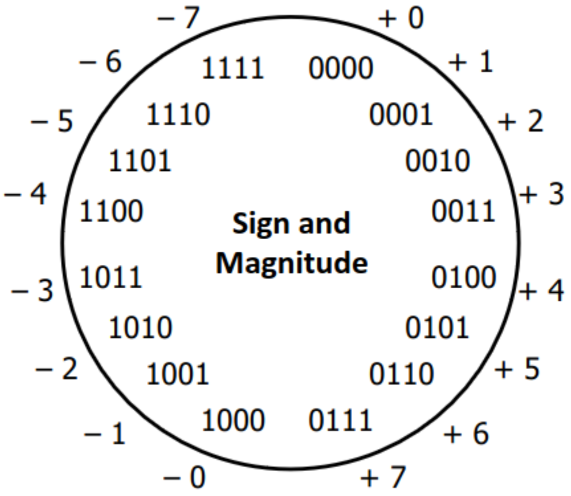
- Historically, different machines did this different ways
  - Sign and magnitude
  - 1's complement
  - 2's complement

what's currently used

Sign and Magnitude (Review)

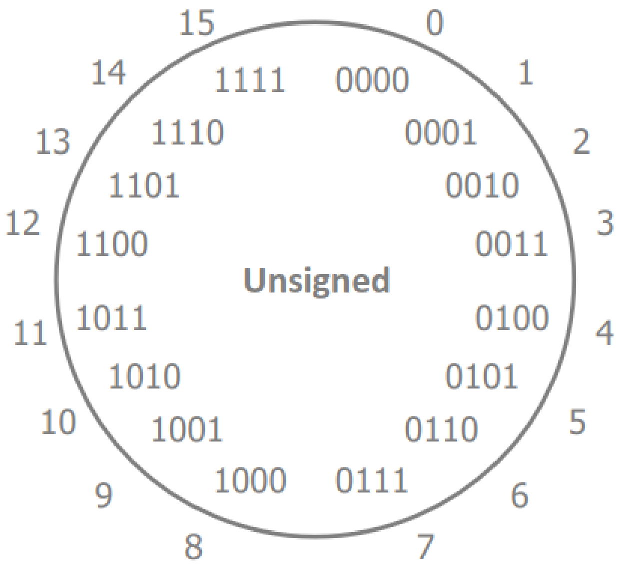
Not used in practice for integers!

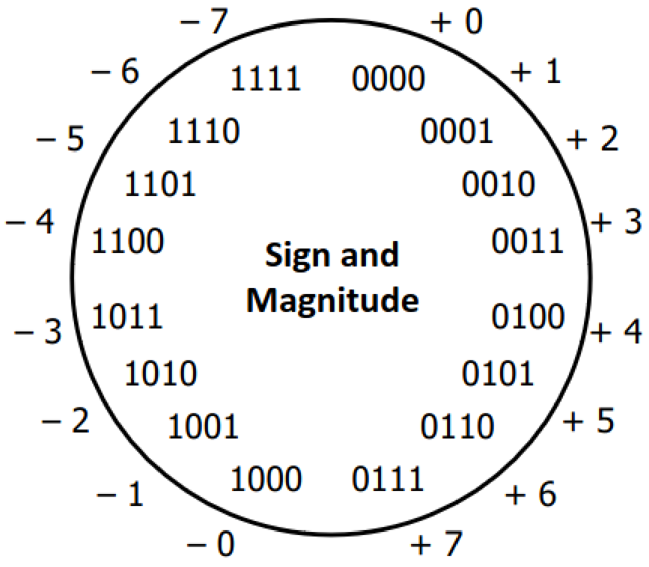
- Designate highest-order (most-significant) bit to represent sign
  - Sign = 0: positive number
    - 0x7F = 0b01111111 = positive 0b1111111 = 127
  - Sign = 1: negative number
    - 0xFF = 0b11111111 = negative 0b1111111 = -127
- Benefits:
  - Positive numbers have the same encoding as their unsigned equivalents
  - 0x00 = 0
  - Easy to tell the sign of a number



Sign and Magnitude (pt 2)

- Drawbacks?





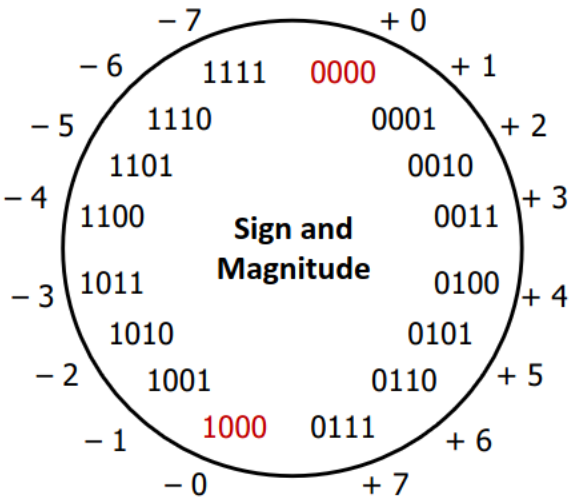
Sign and Magnitude (pt 2)

Not used in practice for integers!

- Drawbacks:
  - Two representations of 0 (bad for checking equality)

0x00 = 0b<u>0</u>000000 = positive 0b0000000 = “positive” 0

0x80 = 0b<u>1</u>000000 = positive 0b0000000 = “negative” 0

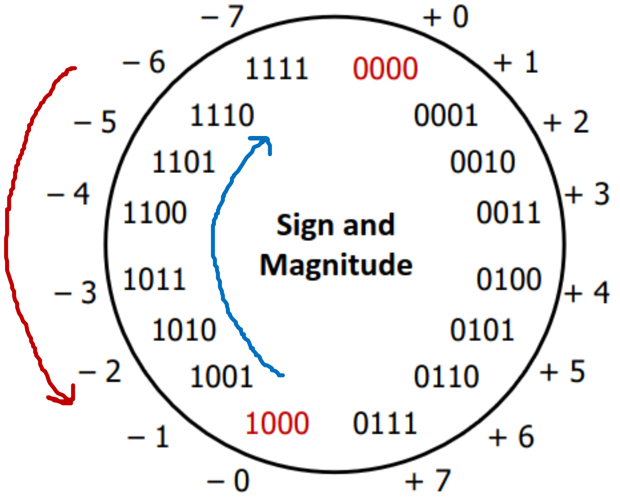


Sign and Magnitude (pt 3)

Not used in practice for integers!

- Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - Arithmetic is cumbersome
    - Negative numbers increment in the wrong direction

as binary increases, negative #s decrease



<u>Ex:</u> 4-3 != 4+(-3)

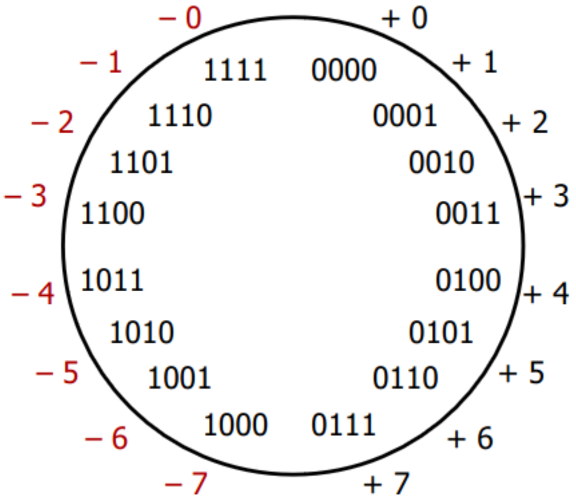
```
  0100     4
- 0011     3
----------------
= 0001     1
```

```
  1011   4
+ 0100  -3
----------------
= 1111  -7
```

Two's Complement

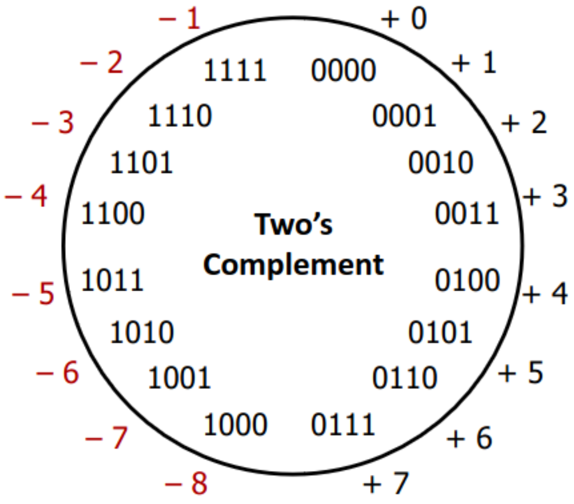
- Let's fix these problems:
  1. Flip negative encodings so incrementing works
     a. This is called "one's complement"

you don't need to know this, just a fun fact :)



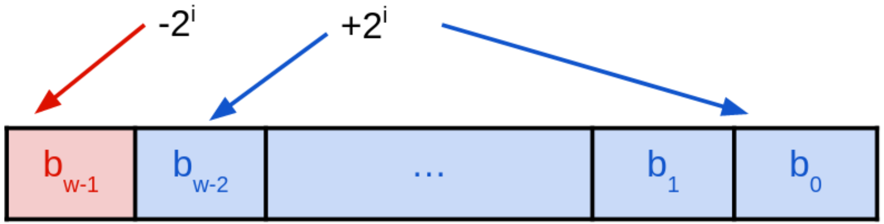
Two's Complement (pt 2)

- Let's fix these problems:
    1. Flip negative encodings so incrementing works
    2. Shift negative encodings over by 1 to eliminate double-0
- Still has a lot of the same benefits as sign-magnitude
    - Positive values still the same as unsigned
    - MSB still indicates sign!
        - 0 is treated as "positive", so we can represent one more negative number than positive



Two's Complement Negatives (Review)

- Accomplished with one neat mathematical trick!
  - Most-significant bit has negative weight
- 4-bit example:
  - 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 is represented as $11..11_2$
  - MSB makes it "super negative," need to add as much positive value as possible to get to -1
- Easy trick to negate: just flip the bits and add 1!



ex: 1= 0b0001
~1= 0b1110
~1+1= 0b1111 = -1 :)

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, or 2's Complement)

unsigned: $2^3 + 2^1 + 2^0 = 11$

sign-mag: $-(2^1 + 2^0) = -3$

2's comp: $-2^3 + 2^1 + 2^0 = -5$

A) -4
B) -5 ✓
C) 11 ✓
D) -3 ✓
E) We're lost…

Discussion

- Discuss these questions in groups of 2-4
  - We'll discuss as a class afterwards, so be prepared to share out
  - Please be respectful of others' opinions and experiences
- Java was designed to only support <u>signed</u> ints
  - Why might the designers of Java chosen this?
  - What are some benefits and drawbacks of this decision?
  - What does this tell you about the implicit *values* embedded in C vs Java?

pros of Java:
- easy to learn
- less error-prone

cons of Java:
- less efficient
- less programmer control
- can't represent as many positive values

Java's priorities:
- ease of use
- portability
- beginner-friendly

C's priorities:
- efficiency
- programmer freedom
- close to hardware

Summary

- **Bitwise operators** allow for fine-grained manipulations of data
  - Bitwise AND (**&**), OR (**|**), and NOT (**~**) are *different* than logical AND (&&), OR (||), and NOT (!)
  - Useful for **bitmasks**
- Choice of *encoding scheme* is important
  - Tradeoffs based on size requirements and desired operations
- Integers are represented using **unsigned** and **two's complement** representations
  - Sign and Magnitude no longer used for integers
  - Limited by fixed bit width