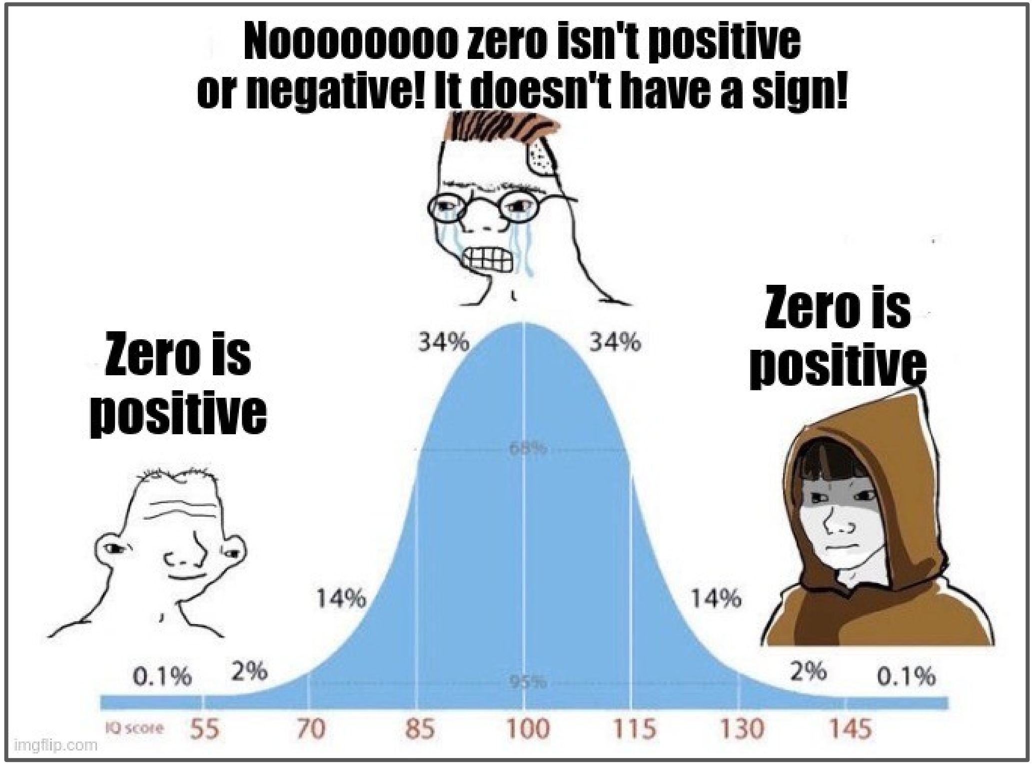
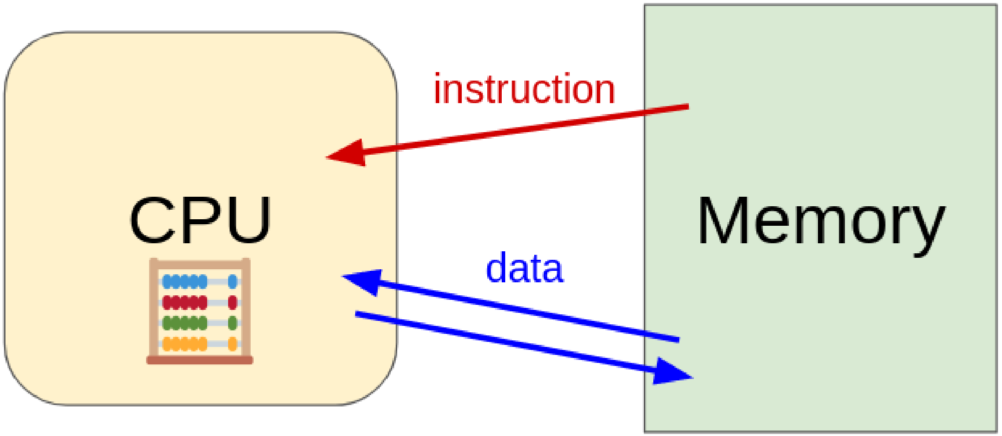
# Data III, Integers I



Recap: CPU and Memory




a. How does the CPU find its data in memory?

## Logical Operators

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

- 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

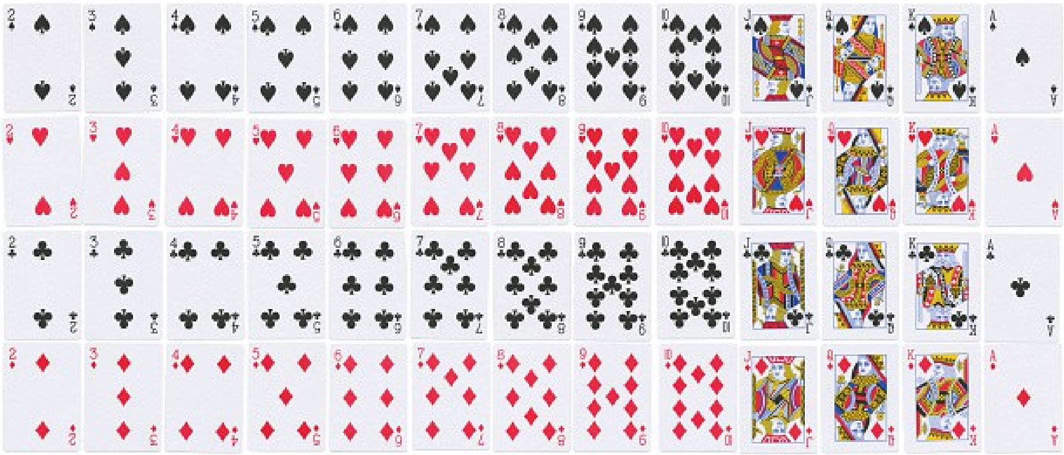
- We can use binary bitwise operators (&, |, ^) along with a specially chosen **bitmask** in order to read or write to particular bits in a piece of data

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

| | | |
|---|---|---|
| *b* & 0 = ___ | *b* ^ 0 = ___ | *b* \| 0 = ___ |
| *b* & 1 = ___ | *b* ^ 1 = ___ | *b* \| 1 = ___ |

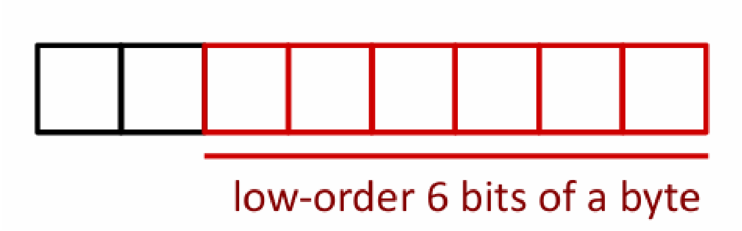
Numerical Encoding Design Example

- 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

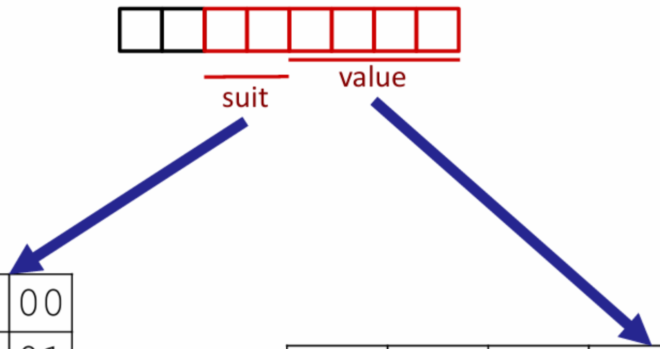
- Binary encoding of 52 cards - only 6 bits needed
  - $2^6 = 64 >= 52$
  - Fits in one byte
- Just count cards in binary
- **Problem:** hard to compare value & suit




| 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

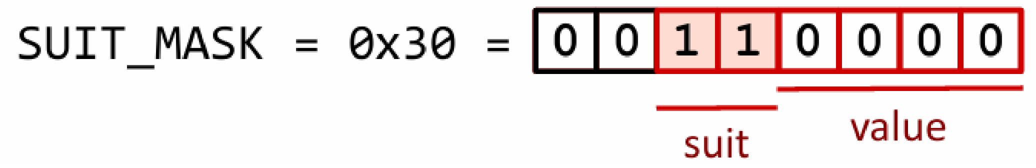



| | |
|---|---|
| ♣ | 00 |
| ♦ | 01 |
| ♥ | 10 |
| ♠ | 11 |

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

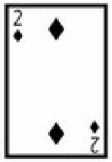
## Compare Card Suits

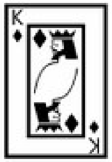
```
#define SUIT_MASK = 0x30      // 0b00110000

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



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);
```

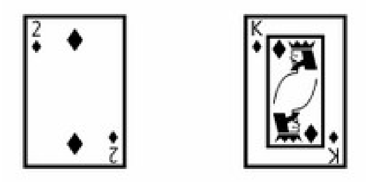
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

&

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

=

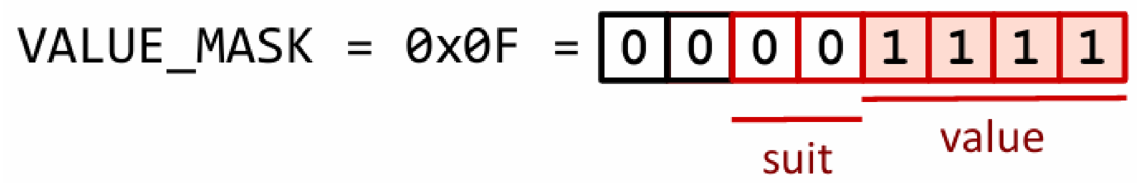
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

==

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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));
}
```



Compare Card Values

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

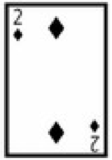
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

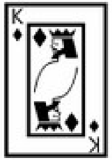
&

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

Integers

## Encoding Integers

- The hardware supports two flavors of integers
  - **Unsigned** - only non-negative numbers
  - **Signed** - positive and negative numbers
- By default, C ints are signed
  - 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: $0 \ldots 2^w - 1$ ex: w=4, $2^4$ -> 16 possible values for unsigned from 0 to 15
  - Signed values: $-2^{w-1} \ldots 2^{w-1} - 1$ ex: w=4, $2^4$ -> 16 possible values for signed, from -8 to +7

# Unsigned Integers

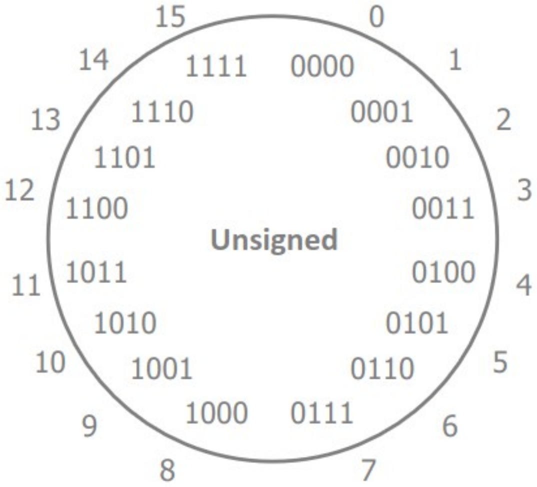
- Just like the binary->base 10 conversion from day 1

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

- 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
```



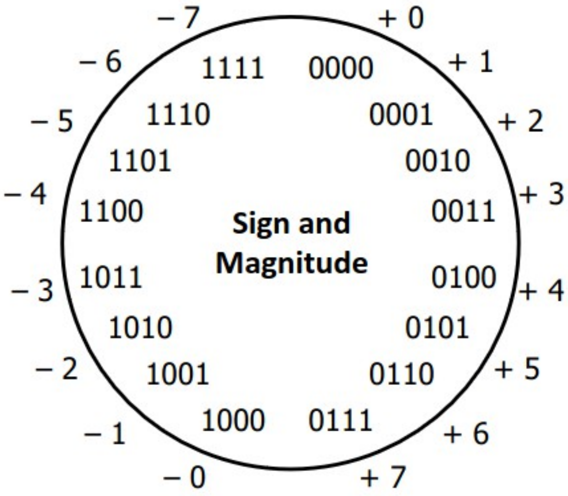
How do we represent signed integers?

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

# Sign and Magnitude

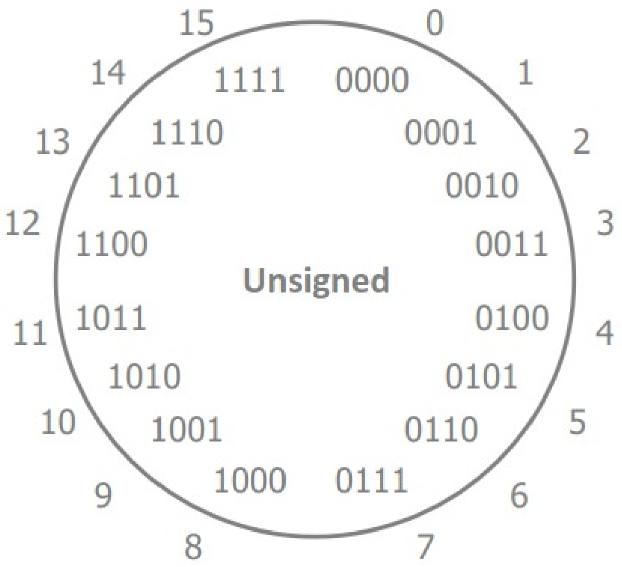
Not used in practice for integers!

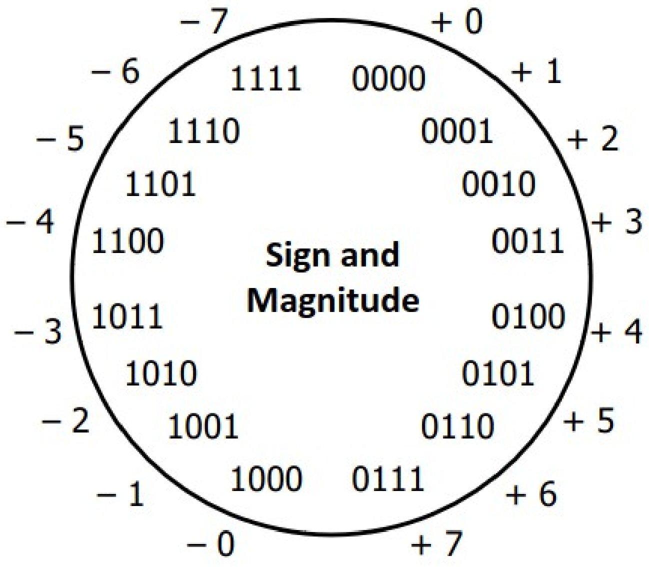
- Designate highest-order (most-significant) bit to represent sign
  - Sign = 0: positive number
    - 0x7F = 0b01111111 = positive 0b1111111 = 127
  - Sign = 1: negative number
    - 0xFF = 0b11111111 = negative 0b1111111 = -127
- Benefits:
  - Positive numbers have the same encoding as their unsigned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



# Sign and Magnitude (pt 2)

- Drawbacks?



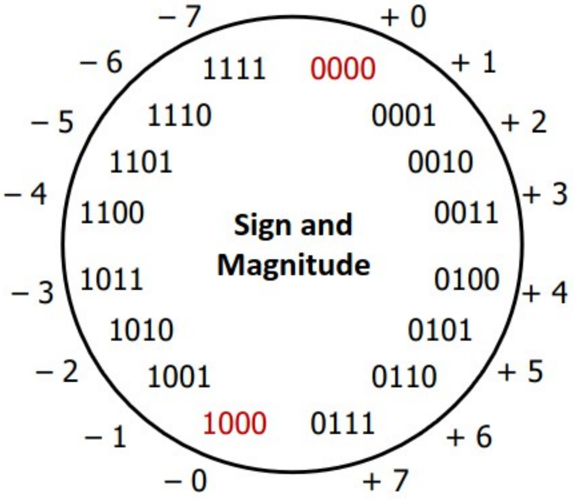
Sign and Magnitude (pt 2)

Not used in practice for integers!

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

0x00 = 0b<u>0</u>0000000 = positive 0b0000000 = “positive” 0

0x80 = 0b<u>1</u>0000000 = positive 0b0000000 = “negative” 0



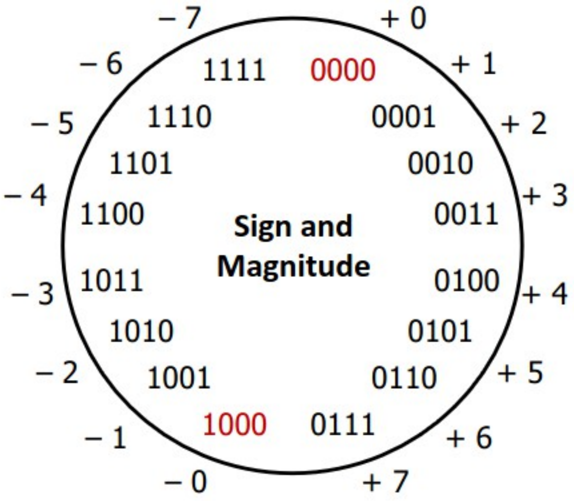
# Sign and Magnitude (pt 3)

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

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

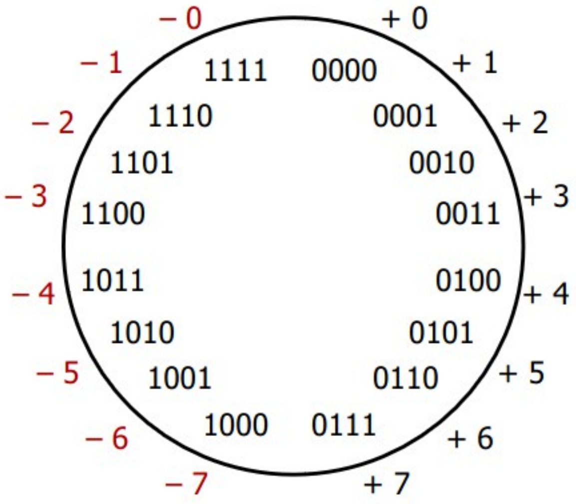
```
   1011  4
+  0100 -3
----------
=  1111 -7
```



# One’s Complement

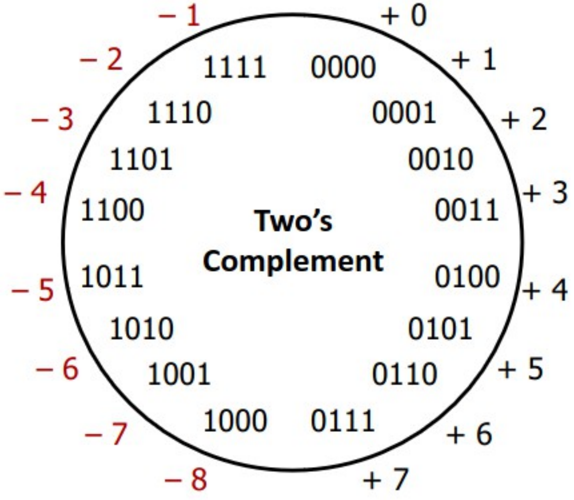
Flip bits:

- To get the 1's complement of a binary number, you invert all the bits (turn 1's into 0's and 0's into 1's).
- Example: The 1's complement of 0101 (which is +5) is 1010 (representing -5 in 1's complement).
- One challenge with 1's complement is that it has two representations for zero: positive zero (0000) and negative zero (1111)



Two’s Complement

- To get the 2's complement of a binary number, you first find the 1's complement (invert all the bits) and then add 1 to the result.
- Example:
  - Start with the binary number 0101 (+5).
  - Invert all the bits to get 1010 (1's complement of 5).
  - Add 1: 1010 + 1 = 1011 (which is -5 in 2's complement).
  - So the 2's complement of 0101 is 1011




Advantages of 2's Complement:

- There’s only one representation for zero.
- It simplifies arithmetic operations, especially addition and subtraction, since both can use the same circuitry.
- It allows for easy detection of overflow during operations.

# Two’s Complement Negatives

- 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!

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)

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

# 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