The Hardware/Software Interface

Data III & Integers I

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus, Divya Ramu, Grace Zhou, Jessie Sun, Jiuyang Lyu, Kanishka Singh, Kurt Gu, Liander Rainbolt, Mendel Carroll, Ming Yan, Naama Amiel, Pollux Chen, Rose Maresh, Soham Bhosale, Violet Monserate

http://xkcd.com/257/

Relevant Course Information

- HW2 due tonight, HW3 due Friday, HW4 due next Wednesday

- Lab 1a released
  - Some later functions require *bit shifting*, covered in Reading/Lecture 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 (`./dlc.py`) and check output
  - Due Monday 10/6, 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 understanding of what you should have learned from the lab
  - Also, great practice for short answer questions on the exams
  - Some are reflective questions – we expect a *personal* (*i.e.*, not generic) response

Lecture Outline (1/3)

Data III:

- **Bitwise and Logical Operators**

Numerical Representation:

- Numerical Encoding Design Example
- Encoding Integers

Boolean Algebra and Bitwise Operators (Review)

- Developed by George Boole in 19th Century
  - Algebraic representation of logic (True → 1, False → 0)

- Bitwise operators apply Boolean operations to bit vectors of matching length
  - Apply to any "integral" data type
    - `char`, `short`, `int`, `long`, `unsigned`
  - Examples:

```
  01101001      01101001
& 01010101    | 01010101

  01101001
^ 01010101    ~ 01010101
```

AND

Outputs 1 only when both input bits are 1:

&	0	1
0	**0**	**0**
1	**0**	**1**

OR

Outputs 1 when either input bit is 1:

|	0	1
0	**0**	**1**
1	**1**	**1**

XOR

Outputs 1 when either input is *exclusively* 1:

^	0	1
0	**0**	**1**
1	**1**	**0**

NOT

Outputs the opposite of its input:

~	
0	**1**
1	**0**

Logical Operators (Review)

- Logical operators: && (AND), || (OR), ! (NOT)
  - In C: **0** is False, **anything nonzero** is True; **always** return 0 or 1

- Examples (char data type)
  - `0xCC && 0x33 -> 0x01`
  - `0x00 || 0x33 -> 0x01`
  - `!0x33 -> 0x00`
  - `!0x00 -> 0x01`

Polling Questions (1/2)

- Compute the result of the following expressions for **char** `c = 0x81;`
  - `c ^ c`
  - `~c & 0xA9`
  - `c || 0x80`
  - `!!c`

Short-Circuit Evaluation (Review)

- 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

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 $\bar{b}$):

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

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

$b \mid 0 = \_\_\_$

$b \mid 1 = \_\_\_$

$b \,\hat{}\, 0 = \_\_\_$

$b \,\hat{}\, 1 = \_\_\_$

Bitmasks Example

- 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

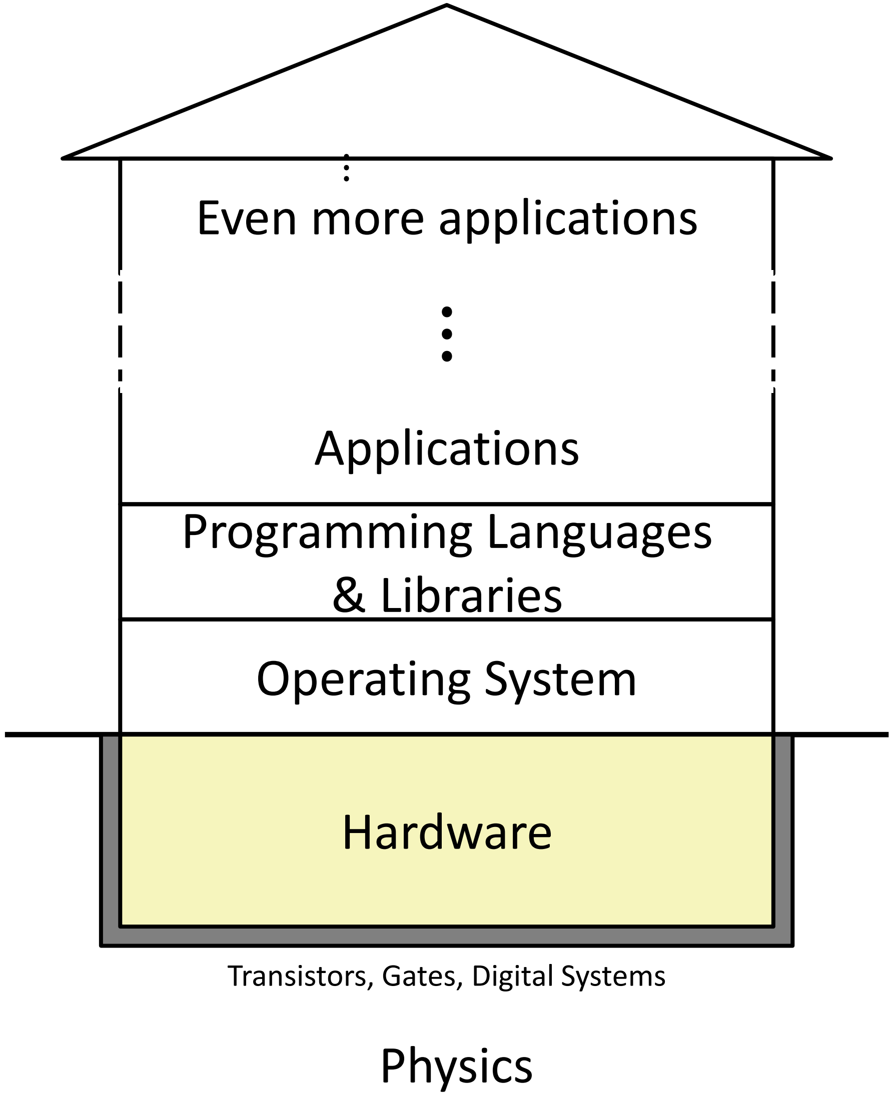
- Example: $b|0 = b,\ b|1 = 1$

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

House of Computing Check-In

- 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?



Lecture Outline (2/3)

Data III:

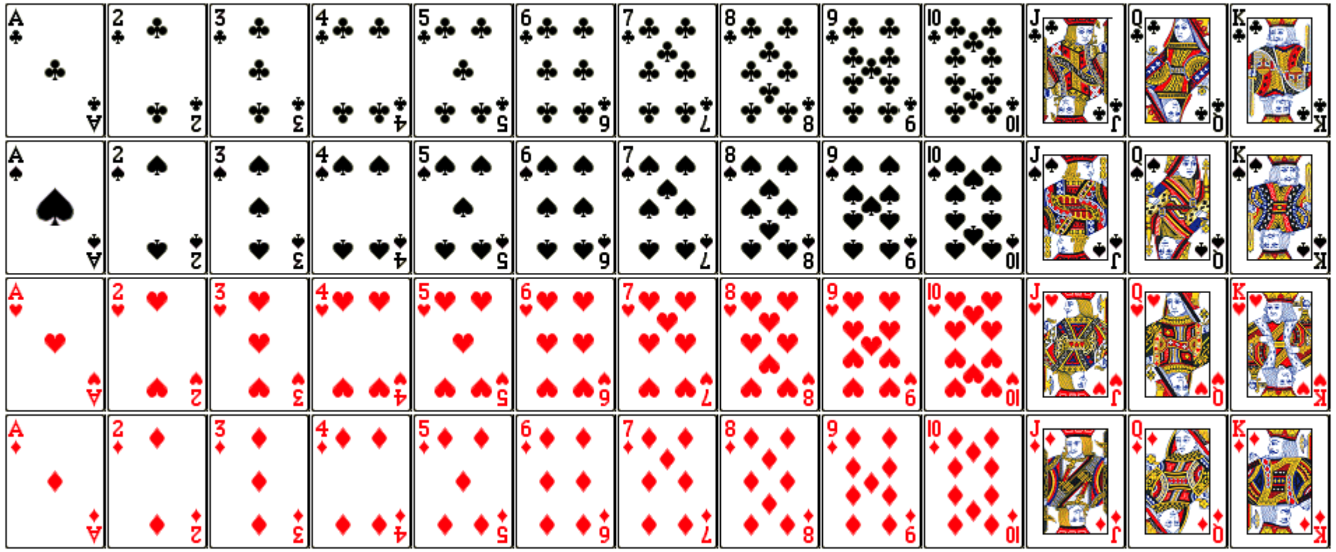
- Bitwise and Logical Operators

Numerical Representation:

- **Numerical Encoding Design Example**
- Encoding Integers

Numerical Encoding Design Example

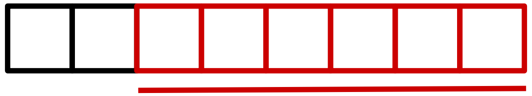
- Encode a standard 52-card deck of French-suited (4 suits) playing cards



- Operations to implement:
  - Which is the higher value card?
  - Are they the same suit?

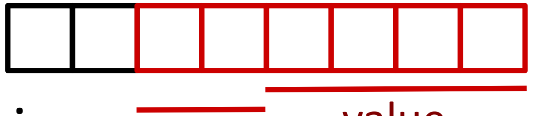
Representations and Fields

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



   - 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)



   - 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

```
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) ) { ... }
```

```
#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);
}
```

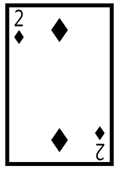
SUIT_MASK = 0x30 = | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

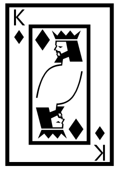
suit (bits 1 1) value (bits 0 0 0 0)

Compare Card Suits Example

```
#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);
}
```





0	0	0	1	0	0	1	0

&

0	0	1	1	0	0	0	0

=

0	0	0	1	0	0	0	0

SUIT_MASK

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

`!(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		value			

Compare Card Values Example

```
#define VALUE_MASK  0x0F

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

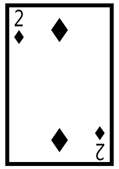
0	0	1	0	0	0	1	0

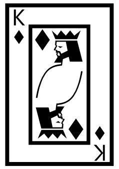
&

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)

Takeaways

- Custom encodings may need to be created when dealing with custom data types or if you're trying to be very space efficient

- There may be many valid encodings but your choices matter
  - *e.g.*, space efficiency, ease of implementation
  - Can separate encoding into multiple fields

- Bitwise and logical operators can be useful for manipulating data

Lecture Outline (3/3)

Data III:

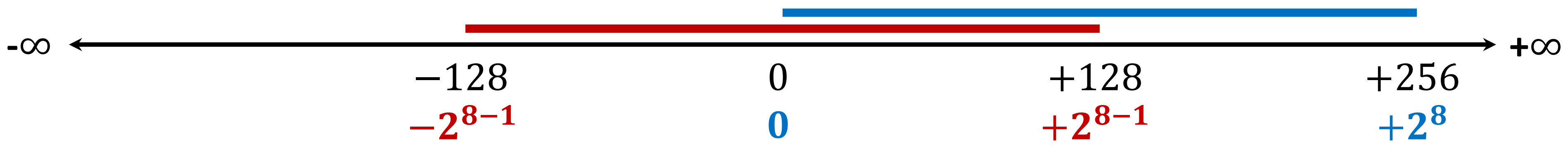
- Bitwise and Logical Operators

Numerical Representation:

- Numerical Encoding Design Example
- **Encoding Integers**

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: $-2^{w-1} \ldots 2^{w-1} - 1$
- <u>Example</u>: 8-bit integers (*e.g.*, `char`)



Unsigned Integers (Review)

- 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:

$$\begin{array}{r} 63 \\ +\underline{\ \ 8} \end{array} \Leftrightarrow \begin{array}{r} 00111111_2 \\ +\underline{00001000}_2 \end{array} \qquad \begin{array}{r} 55 \\ -\underline{\ \ 8\ } \end{array} \Leftrightarrow \begin{array}{r} 00110111_2 \\ -\underline{00001000}_2 \end{array}$$

- In C, add "`unsigned`" keyword in front of any integral type
  - *e.g.*, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`

Sign and Magnitude

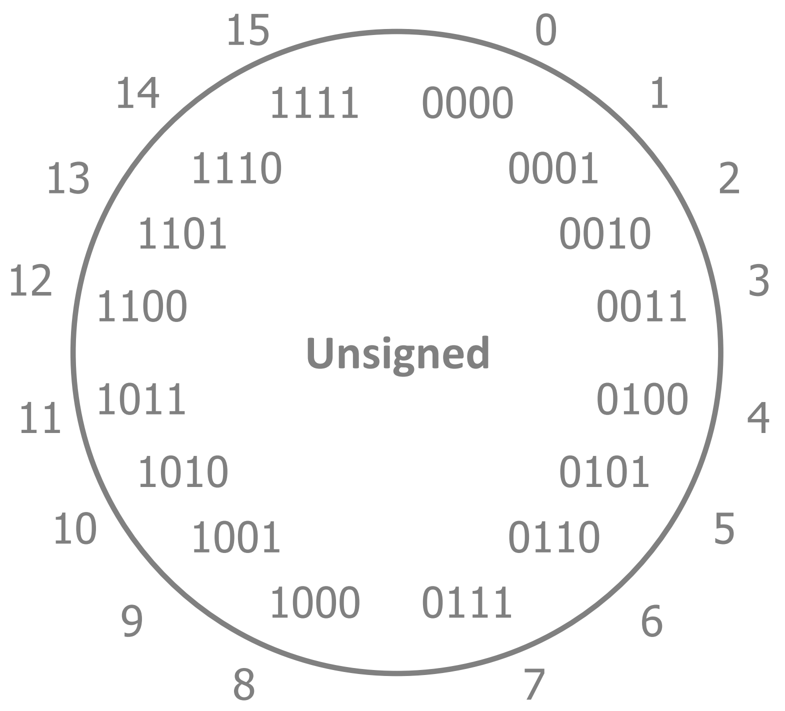
Not used in practice for integers!

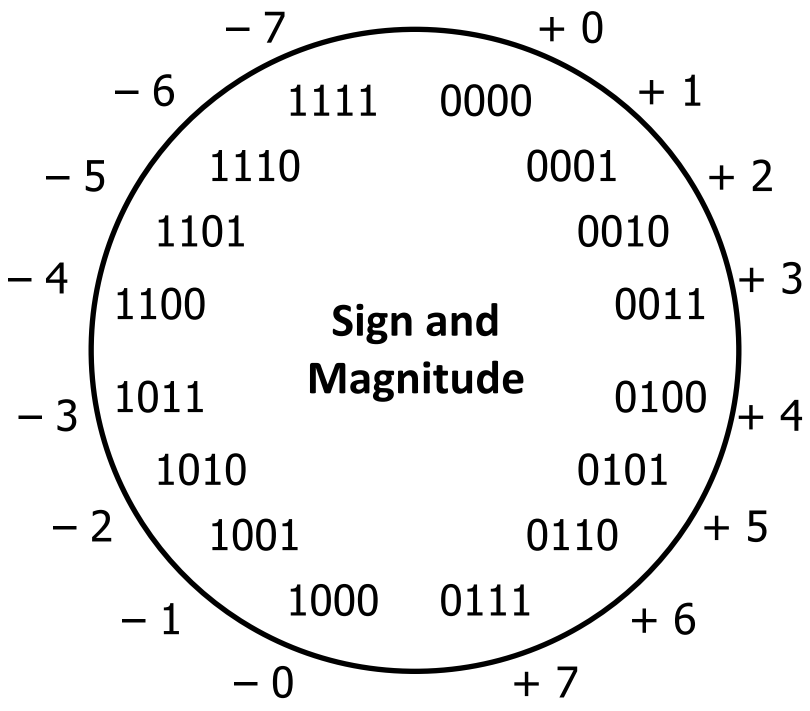
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive number; `sign=1`: negative number
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0
- Examples (8 bits):
  - 0x00 = 00000000_2 is non-negative, because the sign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

Not used in practice for integers!

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

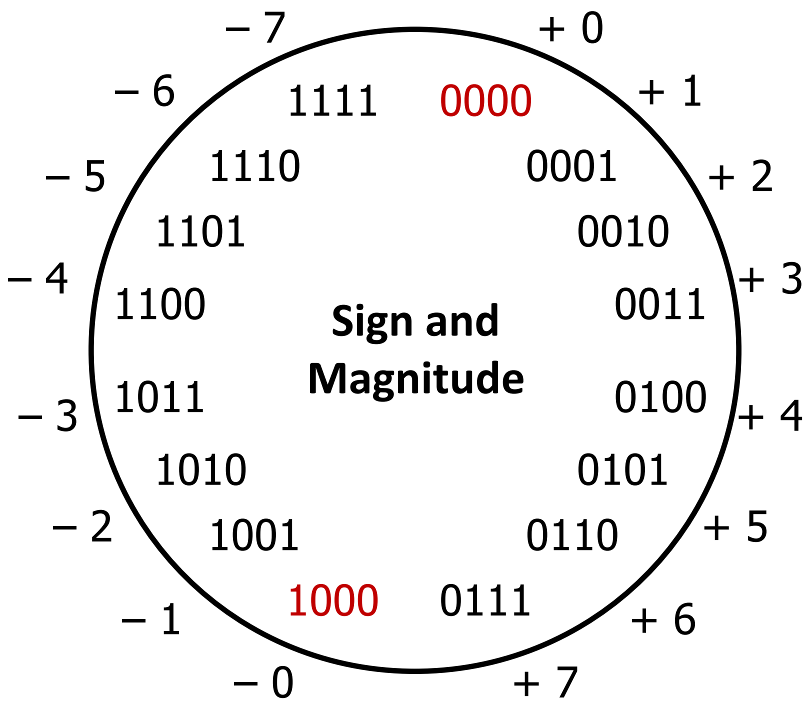




Sign and Magnitude Drawbacks (1/2)

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 Drawbacks (2/2)

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)`

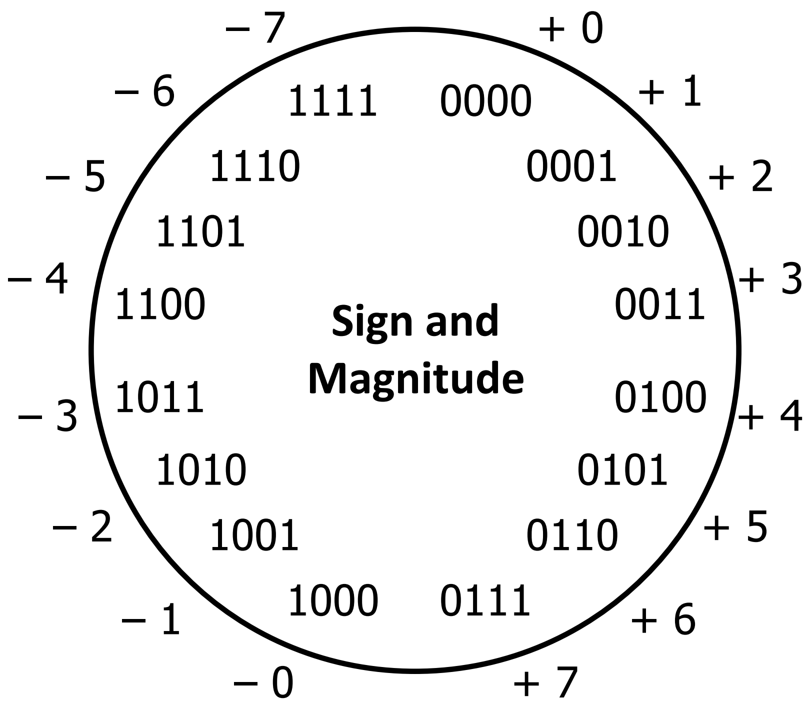
4	0100
- 3	- 0011
1	0001

✓

4	0100
+ -3	+ 1011
-7	1111

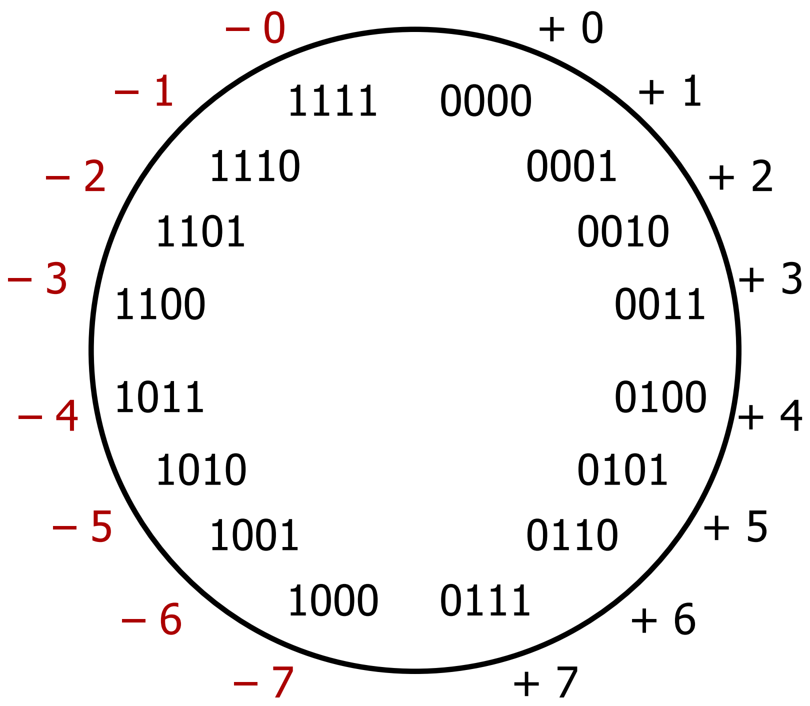
✗

    - Negatives "increment" in wrong direction!



Two’s Complement Development (1/2)

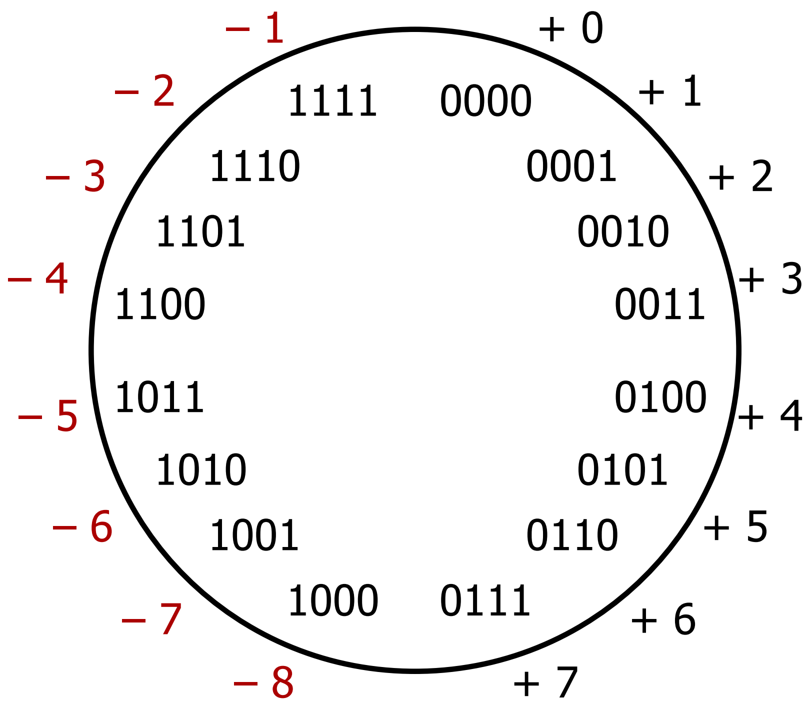
- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
  2) “Shift” negative numbers to eliminate –0



Two's Complement Development (2/2)

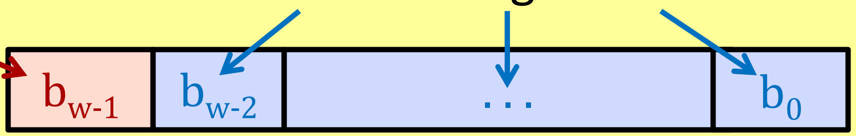
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works
  2) "Shift" negative numbers to eliminate –0

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

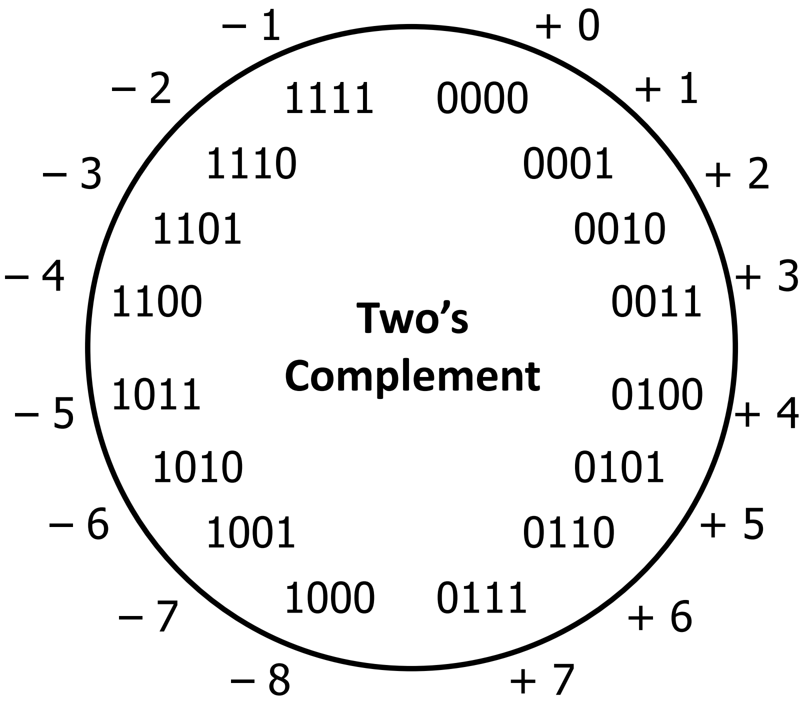


Two's Complement Negatives (Review)

- Accomplished with one neat mathematical trick!



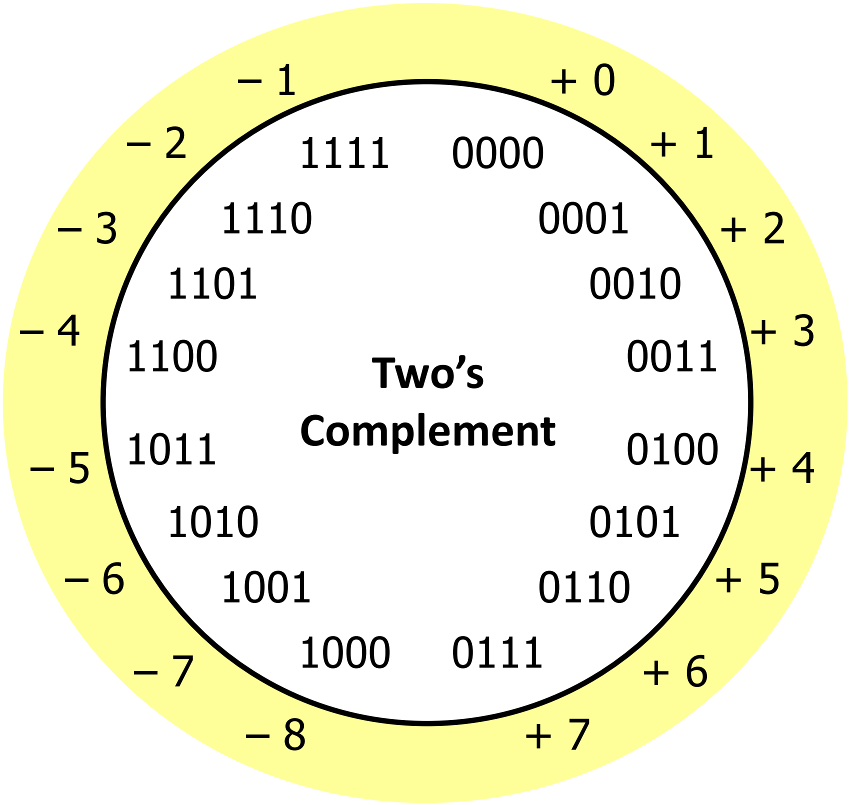
- 4-bit Example:
  - 1010_2 unsigned:
    $1*2^3+0*2^2+1*2^1+0*2^0$ = **10**
  - 1010_2 two's complement:
    $-1*2^3+0*2^2+1*2^1+0*2^0$ = **–6**



Two's Complement is Great (Review)

- Roughly same number of (+) and (–) numbers
- Positive number encodings match unsigned
- Single zero (with all `0`'s encoding)

- Simple negation procedure:
  - Get negative representation of any integer by taking bitwise complement and then adding one!
    `( ~x + 1 == -x )`



Polling Questions (2/2)

- 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

A. -4

B. -5

C. 11

D. -3

E. We're lost…

Integer Hardware

- In practice, all modern system use **unsigned** and **two's complement** encoding schemes for integers
  - Sign and magnitude for integers is a historical artifact, but useful context for design decision and for floating point (next unit)
  - Much of the same hardware can be used for both encoding schemes (*e.g.*, $+$, $-$)

- Fun fact: Java was designed to only support <u>signed</u> data types
  - *Assumed easier for beginners to understand* than having unsigned as well (*i.e.*, eliminate potential sources of error)
  - Unsigned operation support provided with Unsigned Integer API (starting with Java SE 8 in 2014)

Summary (1/2)

- Bit-level operators allow for fine-grained manipulation
  - Bitwise AND (&), OR (|), XOR (^) and NOT (~) operate on the individual bits of the data
  - Especially useful with bitmasks, chosen bit vectors used with &, |, or ^
    - `b & 0 = 0, b & 1 = b` (set to zero or keep as-is)
    - `b | 0 = b, b | 1 = 1` (keep as-is or set to one)
    - `b ^ 0 = b, b & 1 = ~b` (keep as-is or flip the bit)

AND

Outputs 1 only when both input bits are 1:

&	0	1
0	**0**	**0**
1	**0**	**1**

OR

Outputs 1 when either input bit is 1:

|	0	1
0	**0**	**1**
1	**1**	**1**

XOR

Outputs 1 when either input is *exclusively* 1:

^	0	1
0	**0**	**1**
1	**1**	**0**

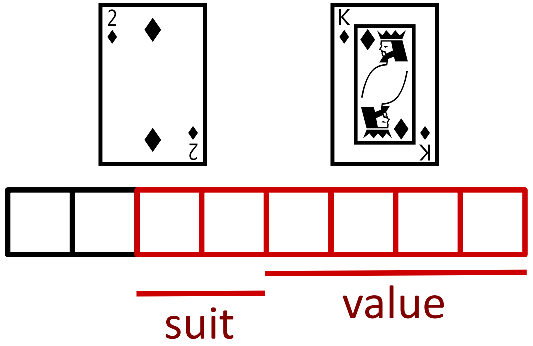
NOT

Outputs the opposite of its input:

~	
0	**1**
1	**0**

- Logical operators work on "truthiness" of data
  - 0 = False, anything else = True
  - Logical AND (&&), OR (||), and NOT (!) → always evaluate to 1 for True

Summary (2/2)



- Choice of *encoding scheme* is important
  - Tradeoffs based on size requirements and desired operations
- Integers represented using unsigned and two's complement representations (sign and magnitude not used in practice)
  - Limited by fixed bit width, satisfy desirable arithmetic properties