

hello!



# Administrivia

- ❖ Lab 0 due tonight, HW 1 on Wednesday
- ❖ Lab 1a released
  - Workflow:
    - 1) Edit `pointer.c`
    - 2) Run the Makefile (`make`) and check for compiler errors & warnings
    - 3) Run `pctest (./pctest)` and check for correct behavior
    - 4) Run rule/syntax checker (`python dlc.py`) and check output = “[]”
  - Due Wed 1/23, will overlap 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 Canvas
  - 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

# Memory, Data, and Addressing

- ❖ Representing information as bits and bytes
- ❖ Organizing and addressing data in memory
- ❖ Manipulating data in memory using C
- ❖ **Strings**
- ❖ Boolean algebra and bit-level manipulations

# Representing strings

- ❖ C-style string stored as a sequence of bytes (**char\***)
  - Elements are one-byte **ASCII codes** for each character
  - No “String” keyword, unlike Java

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

**ASCII:** American Standard Code for Information Interchange

# Null-Terminated Strings

- ❖ **Example:** "Life is good" stored as a 13-byte array

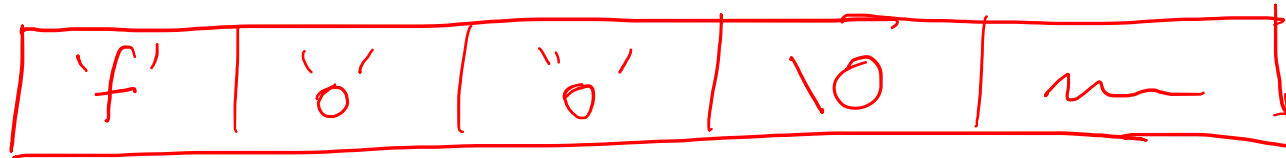
Decimal:	76	105	102	101	32	105	115	32	103	111	111	100	0
Hex:	0x4c	0x69	0x66	0x65	0x20	0x69	0x73	0x20	0x67	0x6f	0x6f	0x64	0x00
Text:	L	i	f	e	.	i	s		g	o	o	d	\0

- ❖ Last character followed by a 0 byte ( ' \0 ' )  
(a.k.a. "null terminator")
  - Must take into account when allocating space in memory
  - Note that ' 0 '  $\neq$  ' \0 ' (i.e. character 0 has non-zero value)
- ❖ How do we compute the length of a string?
  - Traverse array until null terminator encountered

# strlen()

*char \*s = "foo";*

```
char s[5] = "foo";  
int n = strlen(s);
```



- ❖ A. Not sure
- ❖ B. 3
- ❖ C. 4
- ❖ D. 5

# Examining Data Representations

## ❖ Code to print byte representation of data

- Any data type can be treated as a *byte array* by **casting** it to **char\***
- C has **unchecked** casts **!! DANGER !!**

```
void show_bytes(char* start, int len) {  
    int i;  
    for (i = 0; i < len; i++)  
        printf("%p\t0x%.2x\n", start+i, *(start+i));  
    printf("\n");  
}
```

### printf directives:

%p	Print pointer
\t	Tab
%x	Print value as hex
\n	New line



# Boolean Algebra

## ❖ Developed by George Boole in 19th Century

- Algebraic representation of logic (True  $\rightarrow$  1, False  $\rightarrow$  0)
- AND:  $A \& B = 1$  when both A is 1 and B is 1
- OR:  $A | B = 1$  when either A is 1 or B is 1
- XOR:  $A \wedge B = 1$  when either A is 1 or B is 1, but not both
- NOT:  $\sim A = 1$  when A is 0 and vice-versa
- DeMorgan's Law:
 
$$\sim(A | B) = \sim A \& \sim B$$

$$\sim(A \& B) = \sim A | \sim B$$

AND			OR			XOR			NOT		
$\&$	0	1		0	1	$\wedge$	0	1	$\sim$		
0	0	0	0	0	1	0	0	1	0	1	
1	0	1	1	1	1	1	1	0	1	0	

# General Boolean Algebras

- ❖ Operate on bit vectors
  - Operations applied bitwise
  - All of the properties of Boolean algebra apply

$$\begin{array}{cccc}
 01101001 & 01101001 & 01101001 & 01101001 \\
 \& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101 \\
 \hline
 000001 & 111111 & 111111 & 111111 & 111111
 \end{array}$$

- ❖ Examples of useful operations:

$$x \wedge x = 0$$

$$\begin{array}{r}
 01010101 \\
 \wedge 01010101 \\
 \hline
 00000000
 \end{array}$$

$$\underline{x \mid 1 = 1}, \quad \underline{x \mid 0 = x}$$

$$\begin{array}{r}
 01010101 \\
 | 11110000 \\
 \hline
 11110101
 \end{array}$$

0xf0

# Bit-Level Operations in C

- ❖  $\&$  (AND),  $|$  (OR),  $\wedge$  (XOR),  $\sim$  (NOT)
  - View arguments as bit vectors, apply operations bitwise
  - Apply to any “integral” data type
    - long, int, short, char, unsigned

## ❖ Examples with char a, b, c;

- a = (char) 0x41; // 0x41  $\rightarrow$  0b 0100 0001  
 b = ~a; // 0b 1011 1111  $\rightarrow$  0x
- a = (char) 0x69; // 0x69  $\rightarrow$  0b 0110 1001  
 b = (char) 0x55; // 0x55  $\rightarrow$  0b 0101 0101  
 c = a & b; // 0b 0100 0001  $\rightarrow$  0x
- a = (char) 0x41; // 0x41  $\rightarrow$  0b 0100 0001  
 b = a; // 0b 0100 0001  
 c = a ^ b; // 0b 0000 0000  $\rightarrow$  0x

# Contrast: Logic Operations

## ❖ Logical operators in C: `&&` (AND), `||` (OR), `!` (NOT)

■ 0 is False, anything nonzero is True

■ Always return 0 or 1

■ **Early termination** (a.k.a. short-circuit evaluation) of `&&`, `||`

• int x = (42 <sup>0</sup> == 6) <sup>1</sup> || boom();

• int y = (42 == 6) && boom();

!!(16)

⊗

X

int z = p || (\*p)

## ❖ Examples (char data type)

■ !0x41 -> 0x00

■ 0xCC && 0x33 -> 0x01

■ !0x00 -> 0x01

■ 0x00 || 0x33 -> 0x01

■ !!0x41 -> 0x01

# 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();
```

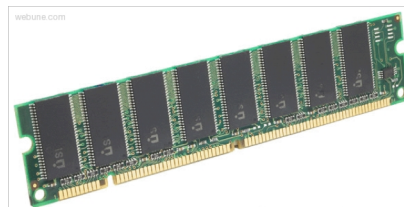
Assembly language:

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

Machine code:

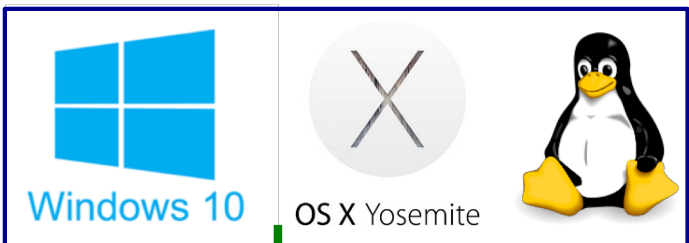
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



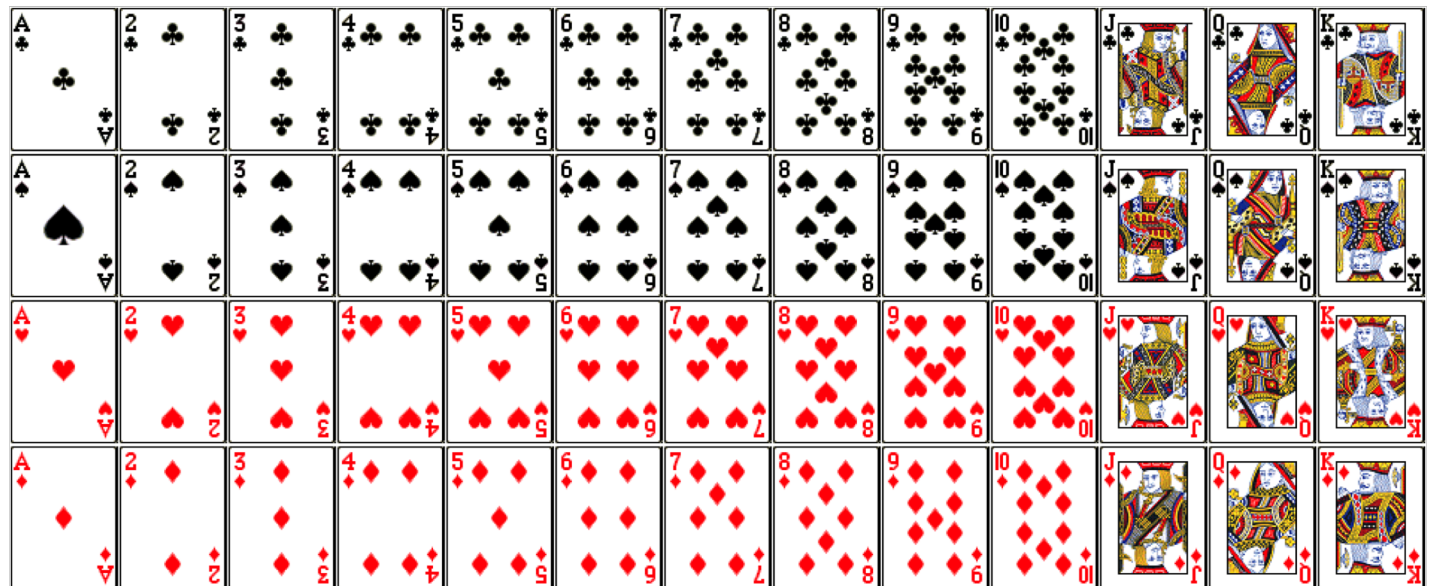
Memory & data  
**Integers & floats**  
 x86 assembly  
 Procedures & stacks  
 Executables  
 Arrays & structs  
 Memory & caches  
 Processes  
 Virtual memory  
 Memory allocation  
 Java vs. C

OS:



# But before we get to integers....

- ❖ Encode a standard deck of playing cards
  - How do we encode suits, face cards?
- ❖ 52 cards in 4 suits
  - Which is the higher value card?
  - Are they the same suit?



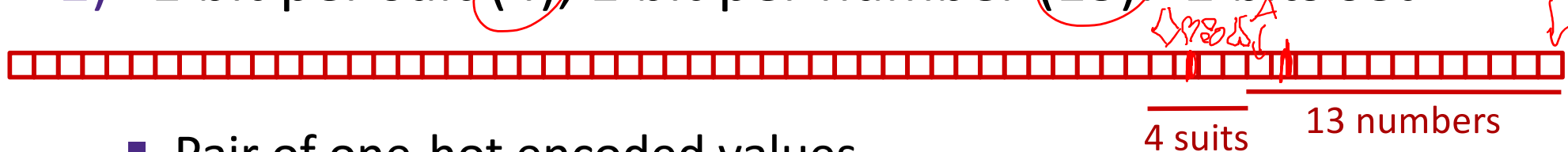
# Two possible representations

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

# Two better representations

## 3) Binary encoding of all 52 cards – only 6 bits needed

■  $2^6 = 64 \geq 52$

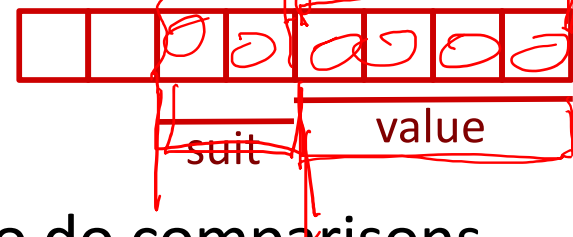
$2^5 = 32$



- 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)

$16 \Rightarrow 13$



- 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 turns all *but* the bits of interest in  $v$  to 0.

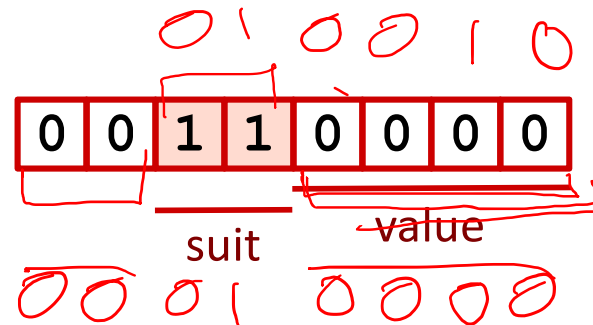
```
char hand[5];           // represents a 5-card hand
char card1, card2;      // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( isSameSuit(card1, card2) ) { ... }
```

```
#define SUIT_MASK 0x30
```

```
int isSameSuit(char card1, char card2) {
    return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

returns int

SUIT\_MASK = 0x30 =



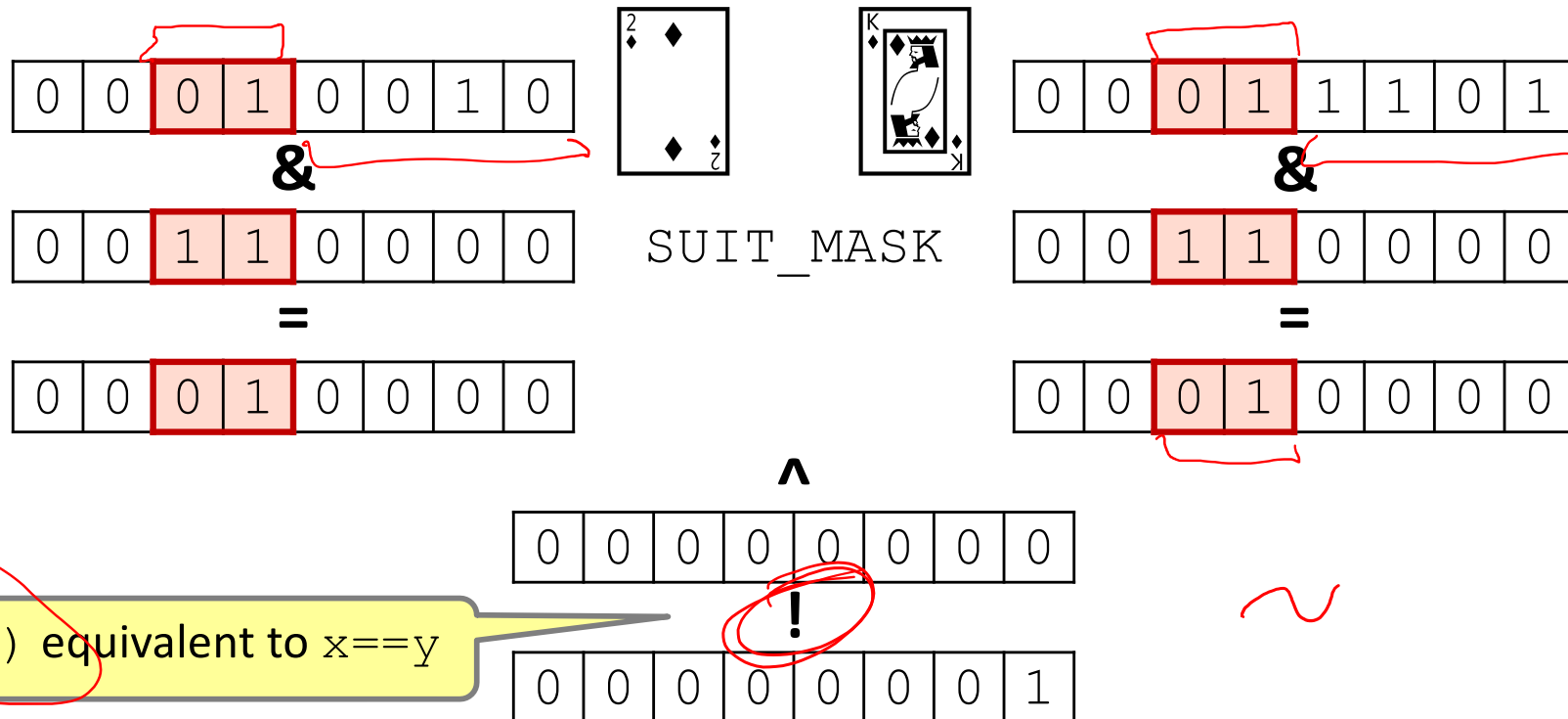
# Compare Card Suits

**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector  $v$ .

Here we turns all *but* the bits of interest in  $v$  to 0.

```
#define SUIT_MASK 0x30
```

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



**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector  $v$ .

# 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 char) (card1 & VALUE_MASK) >
            (unsigned char) (card2 & VALUE_MASK));
}
```

VALUE\_MASK = 0x0F = 

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

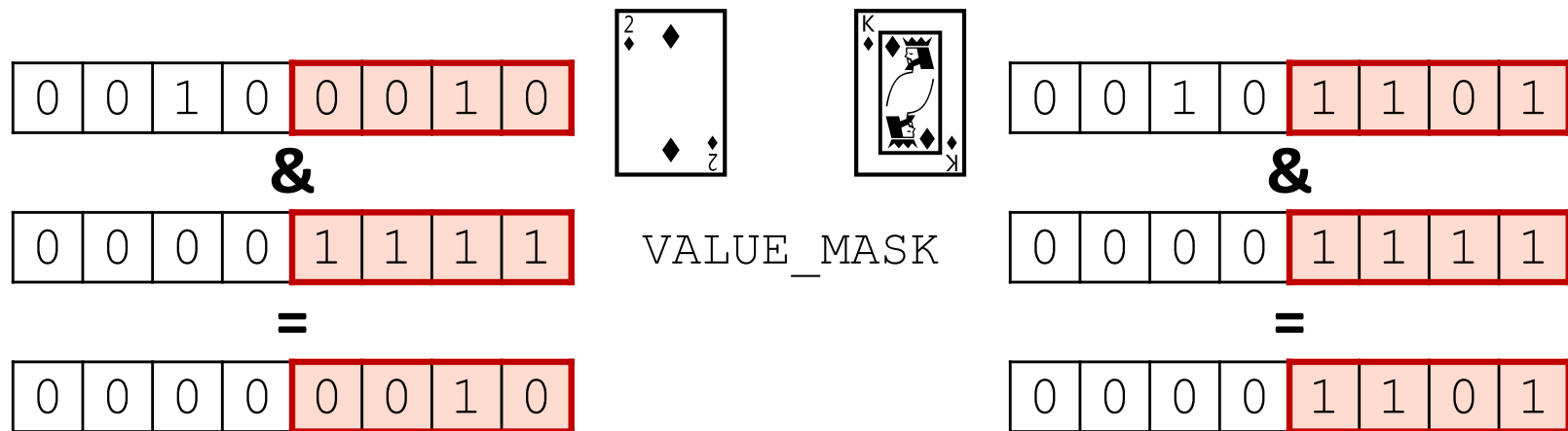


**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector  $v$ .

# Compare Card Values

```
#define VALUE_MASK 0x0F

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



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

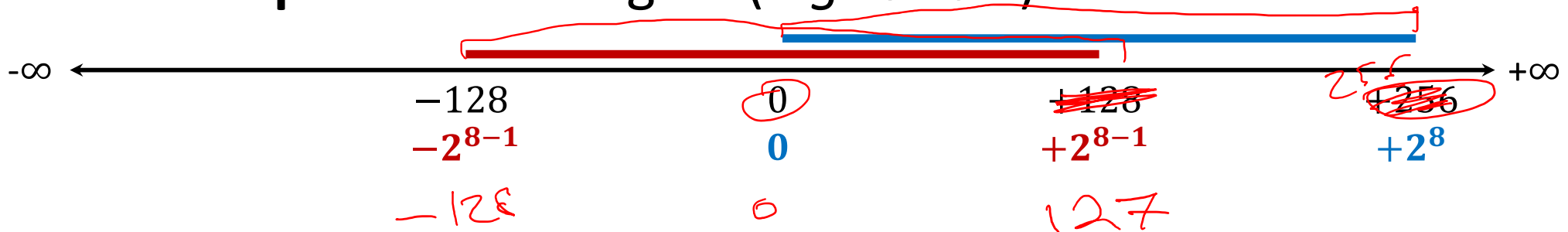
0 (false)

# Integers

- ❖ **Binary representation of integers**
  - Unsigned and signed
  - Casting in C
- ❖ Consequences of finite width representation
  - Overflow, sign extension
- ❖ Shifting and arithmetic operations

# 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:  $0 \dots 2^w - 1$
  - Signed values:  $-2^{(w-1)} \dots 0 \dots 2^{(w-1)} - 1$
- ❖ **Example:** 8-bit integers (e.g. char)



# Unsigned Integers

- ❖ Unsigned values follow the standard base 2 system
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + \dots + b_12^1 + b_02^0$
- ❖ Add and subtract using the normal “carry” and “borrow” rules, just in binary

$$\begin{array}{r} 63 \\ + 8 \\ \hline 71 \end{array}$$

$$\begin{array}{r} 00111111 \\ + 00001000 \\ \hline 01000111 \end{array}$$

- ❖ Useful formula: N ones in a row =  $2^N - 1$

- ❖ How would you make *signed* integers?

8-bits  



# Sign and Magnitude

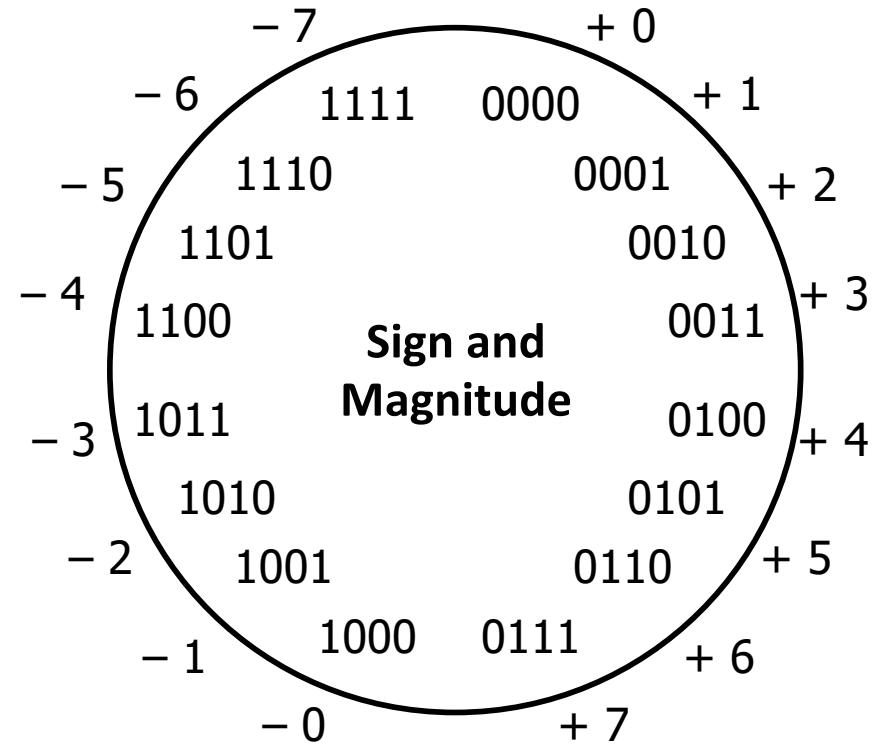
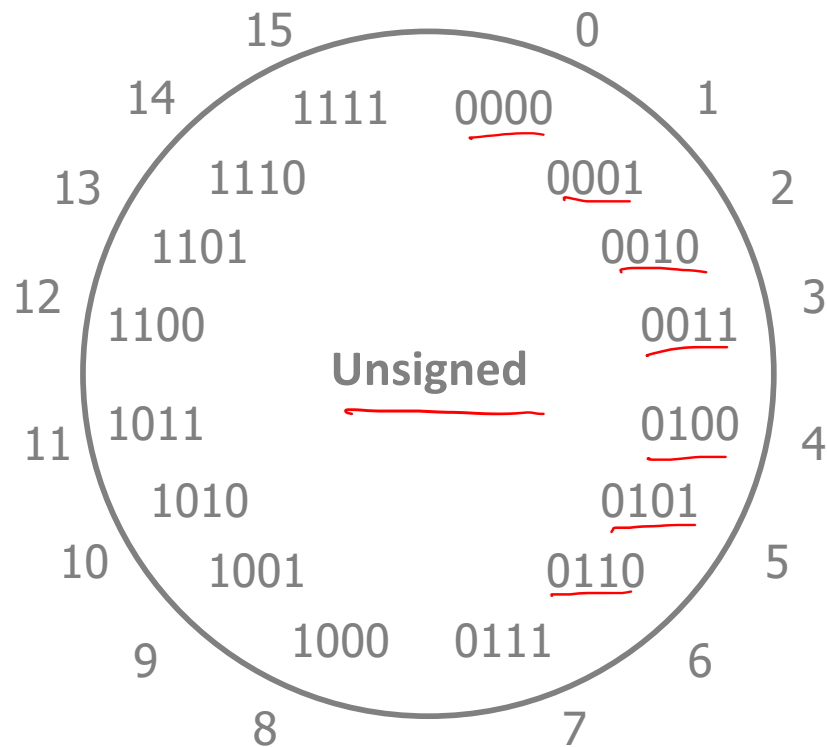
Most Significant Bit

- ❖ 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):
  - $0x00 = \underline{0}0000000_2$  is non-negative, because the sign bit is 0
  - $0x7F = \underline{0}1111111_2$  is non-negative (+127<sub>10</sub>)
  - $0x85 = \underline{1}0000101_2$  is negative (-5<sub>10</sub>)
  - $0x80 = \underline{1}0000000_2$  is negative... zero???



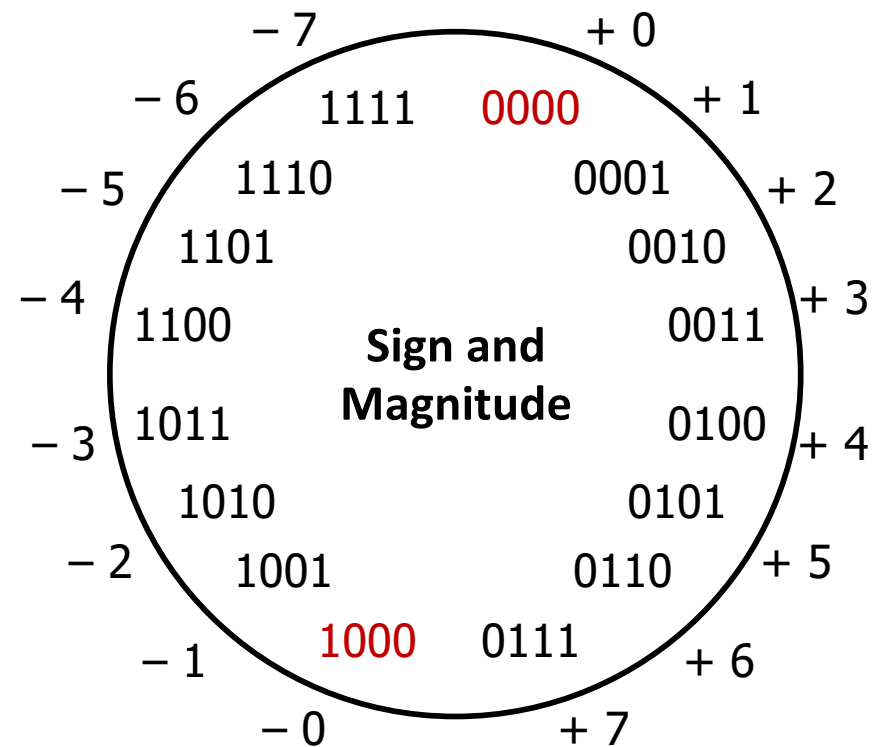
# Sign and Magnitude

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



# Sign and Magnitude

- ❖ MSB is the sign bit, rest of the bits are magnitude
- ❖ Drawbacks:
  - Two representations of 0 (bad for checking equality)



# Sign and Magnitude

- ❖ MSB is the sign bit, rest of the bits are magnitude
- ❖ Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - **Arithmetic is cumbersome**
    - Example:  $4 - 3 \neq 4 + (-3)$

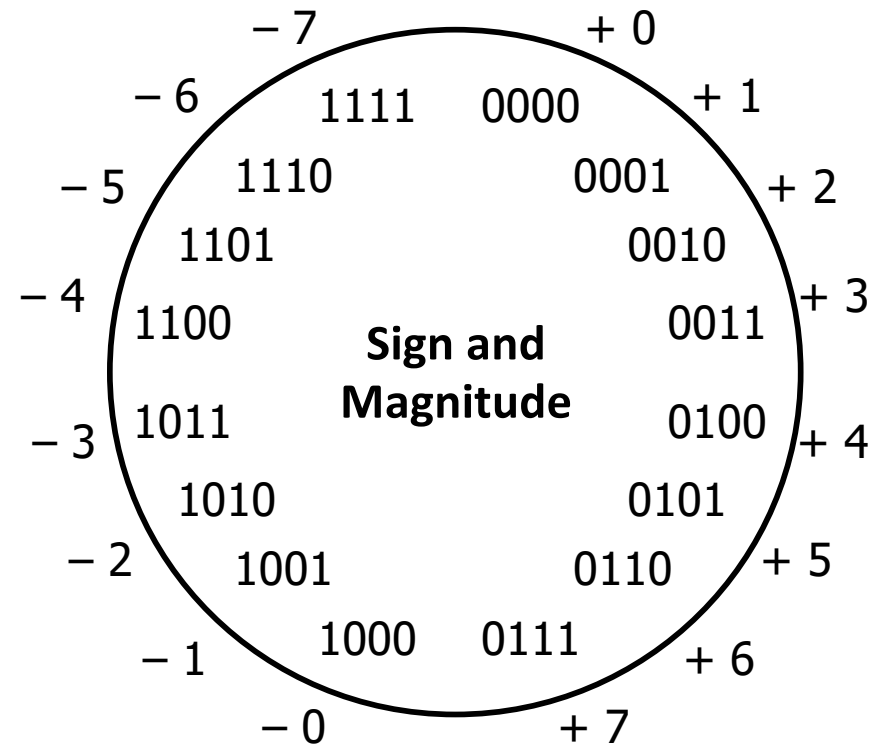
4	0100
- 3	- 0011
1	0001



4	0100
+ -3	+ 1011
-7	1111



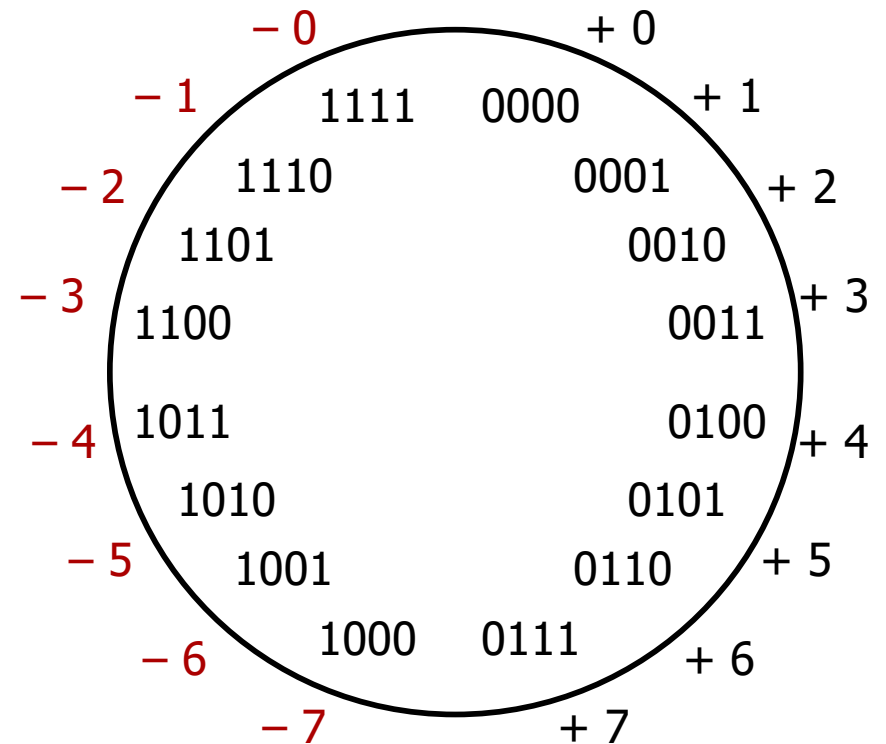
- Negatives “increment” in wrong direction!



# Two's Complement

❖ Let's fix these problems:

1) "Flip" negative encodings so incrementing works



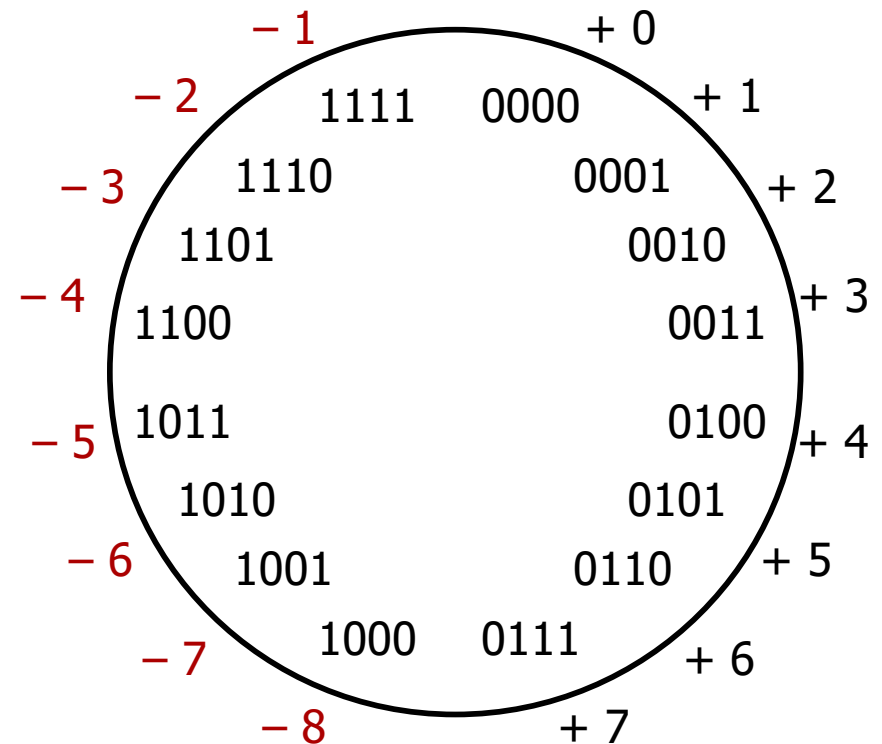
# Two's Complement

❖ 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  $+2^i$



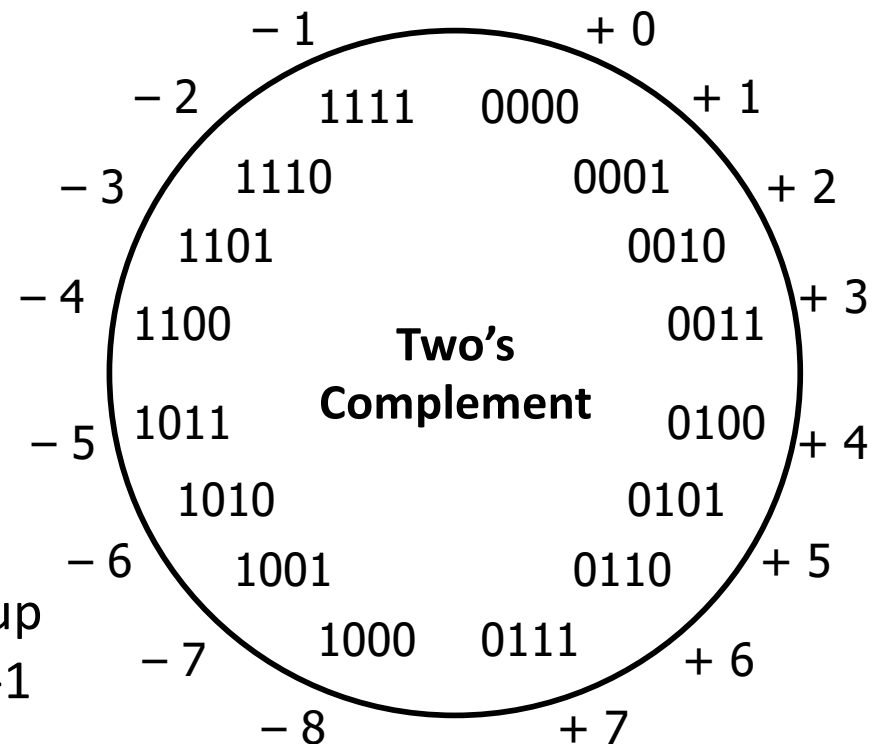
- 4-bit Examples:

- $1010_2$  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:

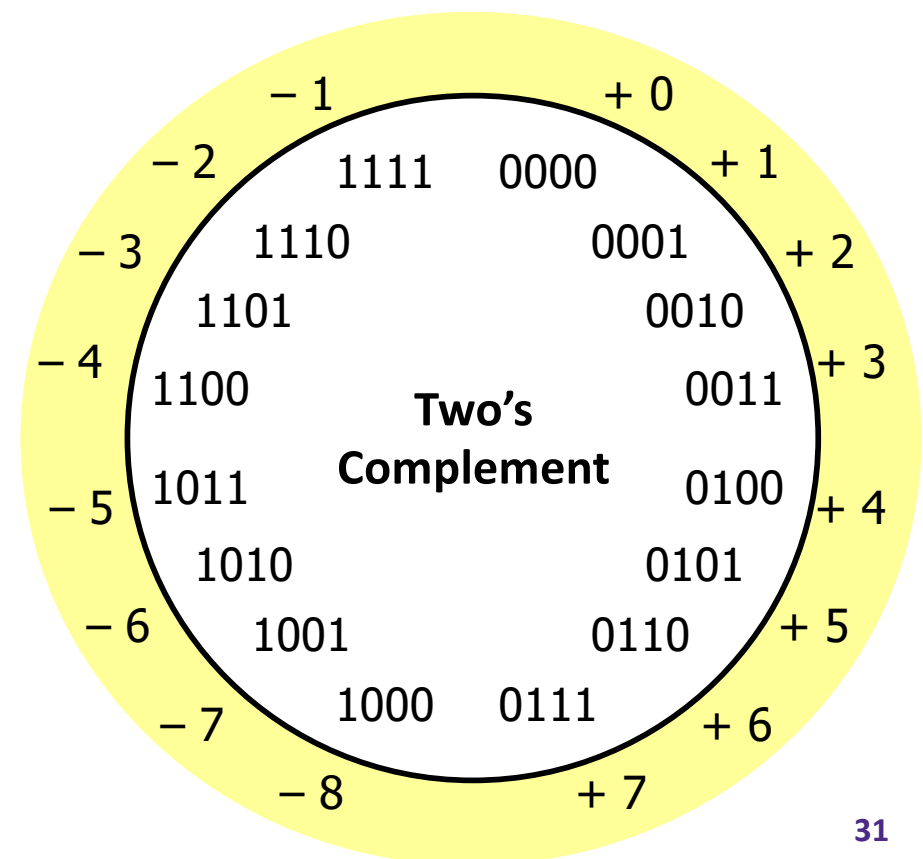
$$1111_2 = -2^3 + (2^3 - 1)$$

- 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
- ❖ Simple arithmetic ( $x + -y = x - y$ )
- ❖ Single zero
- ❖ All zeros encoding = 0
- ❖ Simple negation procedure:
  - Get negative representation of any integer by taking bitwise complement and then adding one!  
 $( \sim x + 1 == -x )$



# Peer Instruction Question

- ❖ Take the 4-bit number encoding **x = 0b1011**
- ❖ What's it mean as an **unsigned** 4-bit integer?
- ❖ What about **signed**?

- 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 ( $\sim$ ) **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