CSE251: System Programming

3. Bits, Bytes, Integers, and Floats

Seongil Wi

Department of Computer Science and Engineering

Motivation: Great Reality!

- **Question: Is $x^2 \geq 0$?**
  - In an ideal mathematical setting: Yes!
  - But in real computing systems? Let's see.

Motivation: Great Reality!

- In case of int’s
  - 40000 * 40000 → 1600000000
  - 50000 * 50000 → ?

Motivation: Great Reality!

- In case of int's
  - 40000 * 40000 → 1600000000
  - 50000 * 50000 → ?

It is important to understand how modern computer systems **store , represent, and manipulate data**

Today's topic

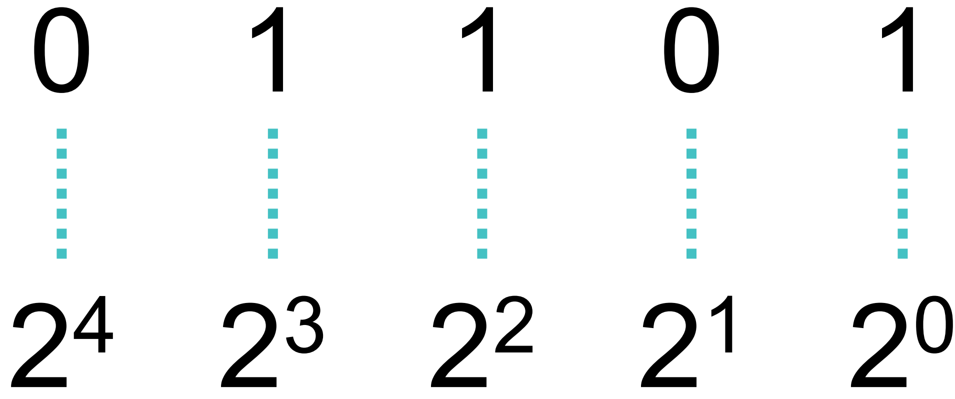
Bits

*Modern computers store and process information represented as **two-valued signals**. These lowly binary digits (i.e., **bits**) form the basis of the digital revolution.*

Binary Representation

- Leads to a simple binary, i.e., base-2 numbering system
  - 0 represents 0
  - 1 represents 1
  - Each "place" represents a power of two

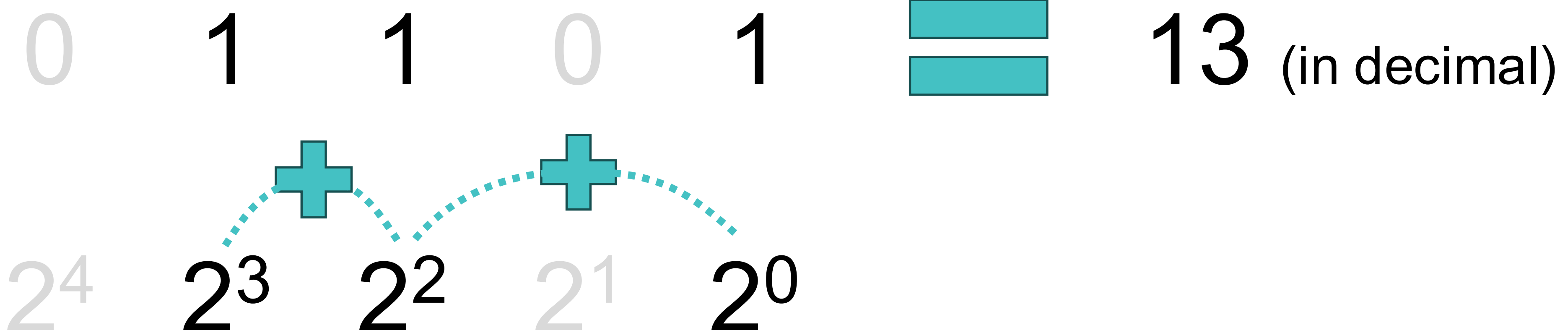
- By encoding/interpreting sets of bits in various ways, we can represent different things:
  - Numbers, characters, operations, …



Binary Representation

- Leads to a simple binary, i.e., base-2 numbering system
  - 0 represents 0
  - 1 represents 1
  - Each “place” represents a power of two

- By encoding/interpreting sets of bits in various ways, we can represent different things:
  - **Numbers**, characters, operations, …



Binary Representation: Simple Numbers

- For example, we can count in binary, a base-2 numbering system (e.g., 000, 001, 010, 011, 100, 101, 110, 111, …)
  - 000 = $0*2^2 + 0*2^1 + 0*2^0 = 0$ (in decimal)
  - 001 = $0*2^2 + 0*2^1 + 1*2^0 = 1$ (in decimal)
  - 010 = $0*2^2 + 1*2^1 + 0*2^0 = 2$ (in decimal)
  - 011 = $0*2^2 + 1*21 + 1*2^0 = 3$ (in decimal)
  - Etc.

- (FYI) Consider some base-10 examples:
  - 000 = $0*10^2 + 0*10^1 + 0*10^0$
  - 001 = $0*10^2 + 0*10^1 + 1*10^0$
  - 357 = $3*10^2 + 5*10^1 + 7*10^0$

Hexadecimal (Hex, Base 16 Number System)

- Writing out numbers in binary takes too many digits ☹
- We want a way to represent numbers more densely such that fewer digits are required
  - Any power-of-two base provides this property ☺
    - E.g, base-8, base-10, base-16
  - Each has been used by "computer people" over time
  - Hexadecimal (i.e., base-16) is often preferred because it is denser

Hexadecimal (Hex, Base 16 Number System)

- Hexadecimal 00_{16} to FF_{16}
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
- Consider 1A2B in Hexadecimal:

  $1*16^3 + A*16^2 + 2*16^1 + B*16^0$

  $= 1*16^3 + 10*16^2 + 2*16^1 + 11*16^0 = 6699$ (in decimal)
- The C language prefixes hex with "`0x`" so they aren't confused with decimal numbers
  - `0xFA1D37B`
  - `0Xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Boolean Algebra

- Developed by George Boole in 19th century
  - Algebraic representation of logic
  - 1: “True” / 2: “False”

Boolean Algebra: AND (&), OR (|)

And (&)

- A & B = 1

  when both A=1 and B=1

Input		Output
a	b	
0	0	0
0	1	0
1	0	0
1	1	1

Or (|)

- A | B = 1

  when either A=1 or B=1

Input		Output
a	b	
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Algebra: NOT (~), XOR (^)

Not (~)

- ~A = 1

  when A=0

Input	Output
0	1
1	0

eXclusive-OR (^)

- A ^ B = 1

  when either A=1 or B=1, but not both

Input		Output
a	b	
0	0	0
0	1	1
1	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001
& 01010101
----------
  01000001
```

```
  01101001
| 01010101
----------
  01111101
```

```
~ 01010101
----------
  10101010
```

Usage Example: Set Operations

- Representation: $a_j = 1$ if $j \in A$

Set A:	**Set B:**
01101001	01010101
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
→ { 0, 3, 5, 6 }	→ { 0, 2, 4, 6 }

- Operations

A & B: Intersection	01000001	{ 0, 6 }
A | B: Union	01111101	{ 0, 2, 3, 4, 5, 6 }
A ^ B: Symmetric difference	00111100	{ 2, 3, 4, 5 }
~B : Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations & , | , ~ , ^ available in C
  - Apply to any "integral" data type
    - `char` (1 byte)
    - `short` (2 bytes)
    - `int` (4 bytes)
    - …

(Background) Data Type in C

- Each variable in C has an associated *data type*
- It specifies ***<u>the type of data that the variable can store</u>*** like integer, character, floating, double, etc

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit machine
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

(Background) Data Type in C

```
int age = 20;             // integer
float height = 5.7;       // floating-point
double pi = 3.14159;      // double-precision floating-point
char grade = 'A';         // character

printf("Age: %d\n", age);
printf("Height: %.1f\n", height);
printf("Pi: %.5lf\n", pi);
printf("Grade: %c\n", grade);
```

Print “Grade: ”

Format string

(Background) Format String in C

- An argument right before "..." (variable-length arguments) that is used to convert C data types into a string (e.g., `printf`, `sprintf`, `sscanf`, `syslog`, ...)

```
int printf(const char *format, ...);
```

(Background) Format String in C

- An argument right before "..." (variable-length arguments) that is used to convert C data types into a string (e.g., `printf`, `sprintf`, `sscanf`, `syslog`, ...)

Format string

```
int printf(const char *format, ...);
```

(Background) Format String in C

Format	Meaning
%c	Character output
%d	Decimal output
%x	Hexadecimal output
%u	Unsigned decimal output
%s	String (terminated by \0) output
%f	Float/double output
…	…

(Background) Data Type in C

```c
int age = 20;              // integer
float height = 5.7;        // floating-point
double pi = 3.14159;       // double-precision floating-point
char grade = 'A';          // character

printf("Age: %d\n", age);
printf("Height: %.1f\n", height);
printf("Pi: %.5lf\n", pi);
printf("Grade: %c\n", grade);
```

Print "Grade: "

Format string

New line

Output

```
Age: 20
Height: 5.7
Pi: 3.14159
Grade: A
```

Bit-Level Operations in C (Example)

```
char a = 0x69;
char b = 0x55;
printf("0x%x\n", a&b);
```

Bit-Level Operations in C (Example)

```
char a = 0x69;
char b = 0x55;
printf("0x%x\n", a&b);
```

0x69 & 0x55 → 01101001_2 & 01010101_2

→ 01000001_2

→ 0x41

Output:

```
0x41
```

Logic Operations

- Logic operations: && , || , !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return "True" or "False"
  - Example:
    - `!0x41          →   0x00`
    - `!0x00          →   0x01`
    - `0x69 && 0x55   →   0x01`

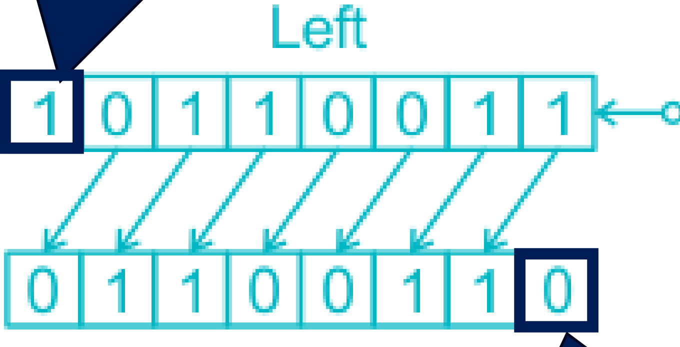
- Bit-level Operations: & , | , ~ , ^

Different from && , || , …

Shift Operations in C

- Left shift: x << y
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
  - Fill with 0’s on right

- Right shift: x >> y
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on left
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit (MSB) on left



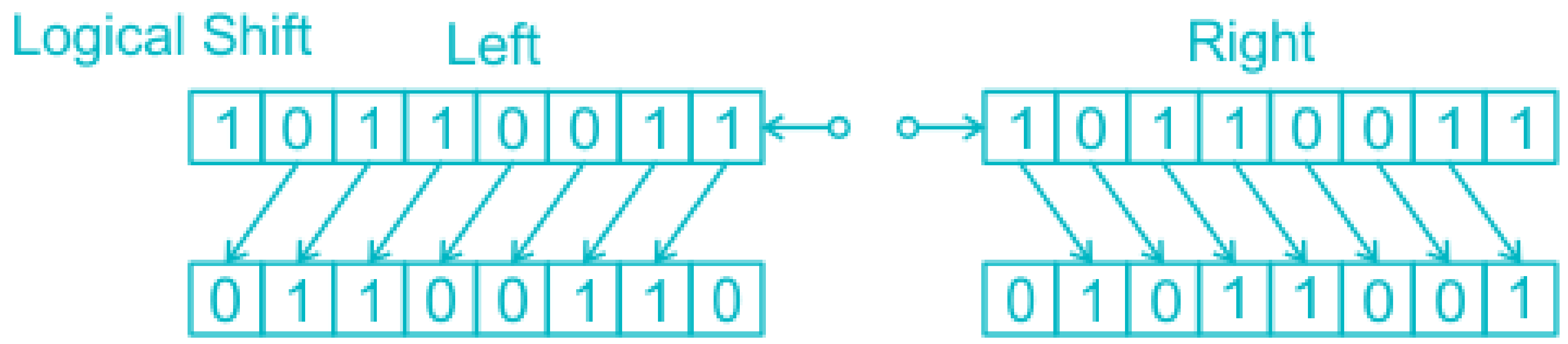
FYI: LSB and MSB

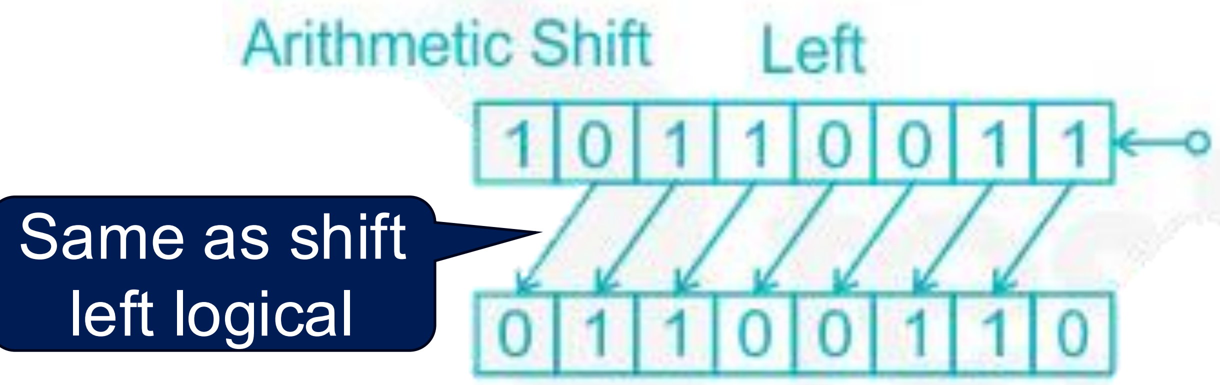
Most Significant Bit (MSB)

1	0	1	1	0	1	1	0	1	0	0	1

Least Significant Bit (LSB)

Two Types of Shift Operations





Same as shift left logical

- Shift left by 1 bit → multiply by 2
- Shift left by 2 bits → multiply by 2^2
- …

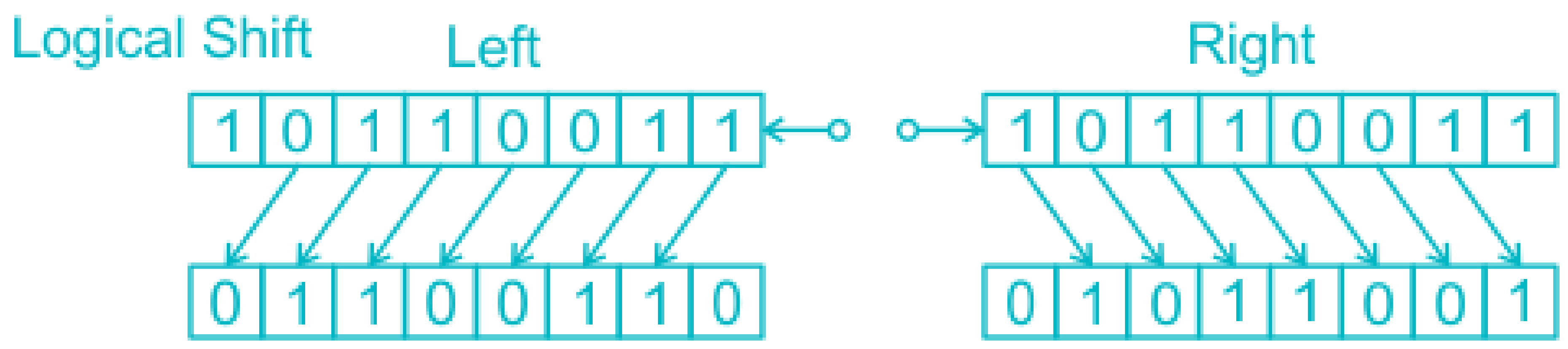
$00001011_2 = 11_{10}$

Arithmetic shift left by 1 bit

$00010110_2 = 22_{10}$

Image from https://testbook.com/question-answer/which-of-the-following-isare-the-types-of-shifts--5f9c078f4112f193f7d991c2

FYI: Logical Shift and Arithmetic Shift



- Shift left by 1 bit → multiply by 2
- Shift left by 2 bits → multiply by 2^2
- …

- Shift right by 1 bit → divide by 2
- Shift right by 2 bits → divide by 2^2
- …

Image from https://testbook.com/question-answer/which-of-the-following-isare-the-types-of-shifts--5f9c078f4112f193f7d991c2

Arithmetic Right Shifts vs. Logical Right Shifts

- The C standards do not precisely define which type of right shift should be used for the >> operator
- However, almost all compiler/machine behaves as follows:
  - For signed data, an arithmetic right shift is performed
  - For unsigned data, a logical right shift is performed

Shift Operations in C

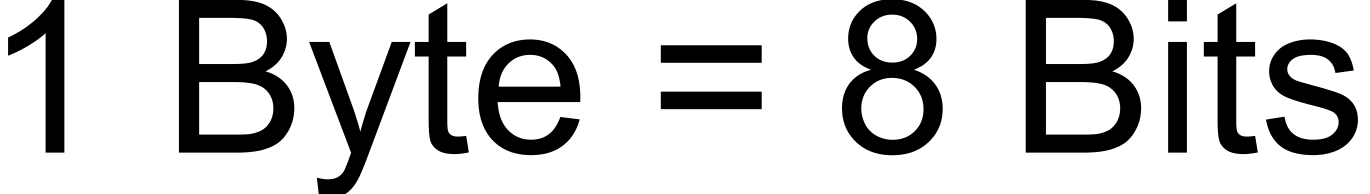

```c
int a = 0xdeadbeef;
printf("0x%x\n", a << 4);
printf("0x%x\n", a >> 4);
```

Output:

```
0xeadbeef0
0xfdeadbee
```

Bytes

Bytes



Show Bytes in C

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}
```

(Example) Show Bytes in C

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```

Address	Memory
0x10e	
0x10d	\0 (0x00)
0x10c	o (0x6f)
0x10b	l (0x6c)
0x10a	l (0x6c)
0x109	e (0x65)
0x108	H (0x48)
0x107	
0x106	

(byte-addressable) Memory

(FYI) ASCII Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

(Example) Show Bytes in C

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```

start 0x107

b 0x107

Output

48

Address	(byte-addressable) Memory
0x10e	
0x10d	\0 (0x00)
0x10c	o (0x6f)
0x10b	l (0x6c)
0x10a	l (0x6c)
0x109	e (0x65)
0x108	H (0x48)
0x107	
0x106	

(Example) Show Bytes in C


```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```

start 0x108

b 0x107

Output

48 65

Address	Memory
0x10e	
0x10d	\0 (0x00)
0x10c	o (0x6f)
0x10b	l (0x6c)
0x10a	l (0x6c)
0x109	e (0x65)
0x108	H (0x48)
0x107	
0x106	

(byte-addressable) Memory

(Example) Show Bytes in C

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```


Output

```
48 65 6c
```

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```

Output

```
48 65 6c 6c
```

Address	Memory
0x10e	
0x10d	\0 (0x00)
0x10c	o (0x6f)
0x10b	l (0x6c)
0x10a	l (0x6c)
0x109	e (0x65)
0x108	H (0x48)
0x107	
0x106	

start: 0x10a

b: 0x107

(byte-addressable) Memory

(Example) Show Bytes in C


```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    char b[] = "Hello";
    show_bytes(b, 5);
    return 0;
}
```

Output

```
48 65 6c 6c 6f
```

start: 0x10b

b: 0x107

Address	Memory
0x10e	
0x10d	\0 (0x00)
0x10c	o (0x6f)
0x10b	l (0x6c)
0x10a	l (0x6c)
0x109	e (0x65)
0x108	H (0x48)
0x107	
0x106	

(byte-addressable) Memory

How About This One?


```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    int a = 98303; // 0x00017fff
    show_bytes(a, 4);
    return 0;
}
```

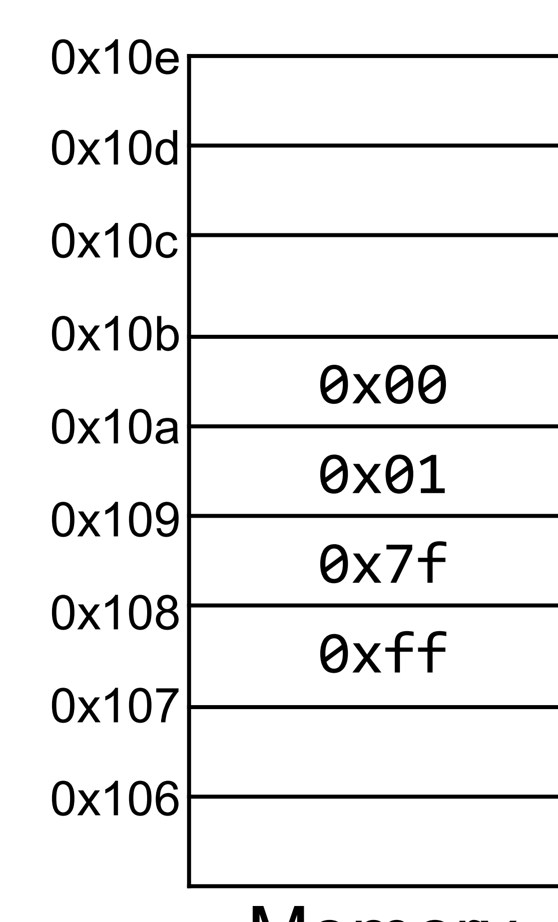
How About This One?

```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", start[i]);
    printf("\n");
}

int main() {
    int a = 98303; // 0x00017fff
    show_bytes(a, 4);
    return 0;
}
```

Output

```
ff 7f 01 00
```



How About This One?


```
void show_bytes(unsigned char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
```

Why is 98303, which is `0x17fff`, represented by `ff 7f 01 00`?
The answer is endianness.

```
int main() {
    int a = 98303; // 0x00017fff
    show_bytes(a, 4);
    return 0;
}
```

Output

```
ff 7f 01 00
```

Address	Value
0x10e	
	0x7f
0x108	0xff
0x107	
0x106	

(byte-addressable) Memory

Endianness

- The order in which a sequence of bytes are stored in memory

- Big Endian = The MSB goes to the lowest address
- Little Endian = The LSB goes to the lowest address

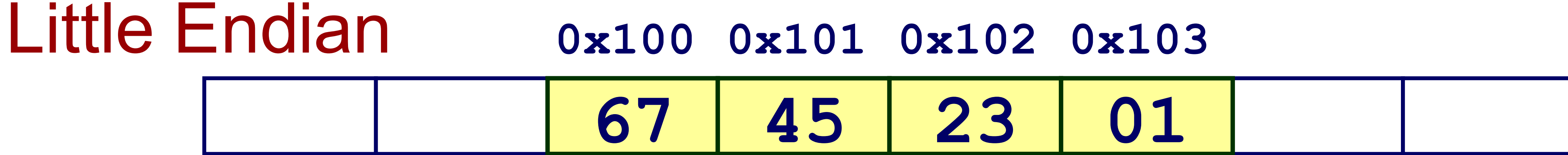
X86 (and X86_64) uses Little Endian

Byte Ordering Example

- Variable x has 4-byte value of `0x01234567`
- Address given by &x is `0x100`

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		



Byte Ordering

- Big Endian: Sun (Oracle SPARC), PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86, ARM processors running Android, iOS, and Linux
  - Least significant byte has lowest address
- Important notes:
  - Bits are not reversed, as the low order bit is the reference point.
  - Doesn’t affect chars, or strings (arrays of chars), as chars are only one byte

Integers

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $2^n - 1$
- Example

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

  $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

How to represent negative numbers?

- Range of 32 bits (n=32): 0 to +4,294,967,295
- Range of 64 bits (n=64): 0 to +18,446,744,073,709,551,615

Approach #1: Sign Magnitude

001 = 1

101 = -1

Approach #1: Sign Magnitude

0001 = 1

1101 = -1

MSB represents the sign
- 0: +
- 1: -

Approach #1: Sign Magnitude

Problems?

MSB represents the sign
- 0: +
- 1: -

The rest of the bits represent the magnitude

Problems of Sign Magnitude

$$
\begin{array}{r}
001 = 1 \\
+\ 110 = -2 \\
\hline
111 \neq -1
\end{array}
$$

Inconsistencies in operations

Approach #2: One's Complement

001 = 1

110 = -1

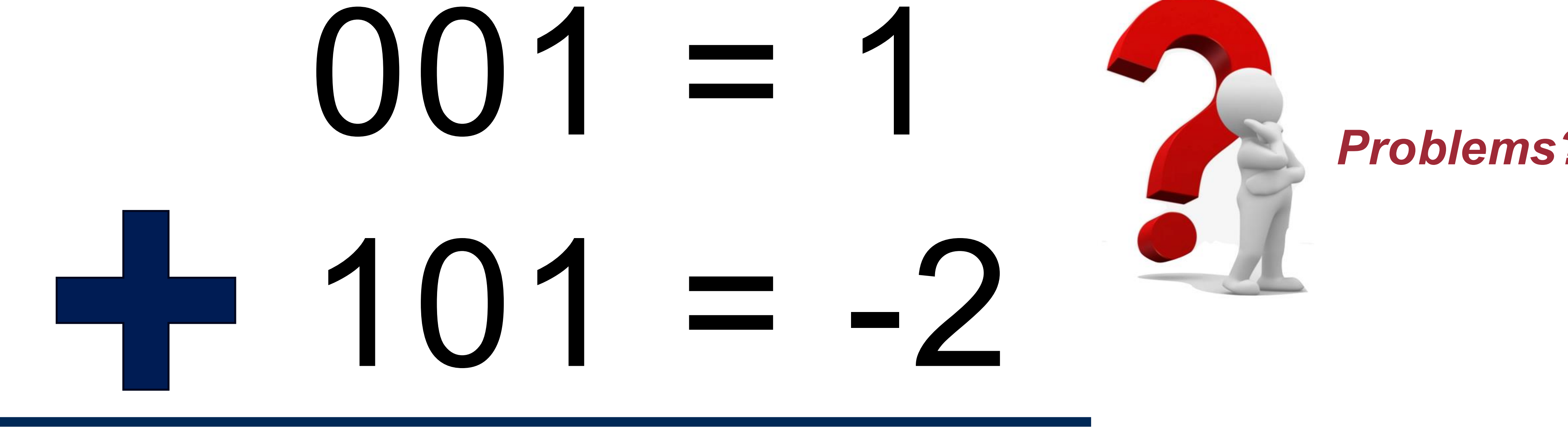
To represent a negative number, all the bits of the corresponding positive number are inverted

Approach #2: One's Complement

$$
\begin{array}{lr}
 & 001 = 1 \\
+ & 101 = -2 \\
\hline
 & 110 = -1
\end{array}
$$

Provides ease of operations in terms of computing

Approach #2: One's Complement



110 = -1

Provides ease of operations in terms of computing

Problems of One’s Complement (+ Sign Magnitude)

Sign Magnitude	One’s Complement
000 = +0	000 = +0
001 = +1	001 = +1
010 = +2	010 = +2
011 = +3	011 = +3
100 = -0	100 = -3
101 = -1	101 = -2
110 = -2	110 = -1
111 = -3	111 = -0

Problems of One's Complement (+ Sign Magnitude)

Sign Magnitude	One's Complement
000 = +0	000 = +0
001 = +1	001 = +1
010 = +2	010 = +2
011 = +3	011 = +3
100 = -0	100 = -3
101 = -1	101 = -2
110 = -2	110 = -1
111 = -3	111 = -0

Two representations for zero

Approach #3: Two's Complement

001 = 1

111 = -1

= One's Complement + 1

Approach #3: Two's Complement

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Approach #3: Two's Complement

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1
Range: -3 ~ +3	Range: -3 ~ +3	Range: -4 ~ +3

Approach #3: Two's Complement

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Provides ease of operations and
a unique representation for zero

2’s complement code is the most widely used representation of signed numbers in computer systems ☺

Negative Number in C

```
int i = -1;
printf("0x%x\n", i);
```

Output:

```
0xffffffff
```

2's-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n} - 1$
- Example

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$$

$$= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$$

$$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$$

- Range of 32 bits (n=32): –2,147,483,648 to +2,147,483,647
- Range of 64 bits (n=64): -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

2's-Complement: Number Range

- 32-bit signed numbers:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0002_{two} = +\ 2_{ten}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

2's-Complement: Number Range

- 32-bit signed numbers:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0002_{two} = +\ 2_{ten}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2,147,483,646_{ten}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2,147,483,647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2,147,483,648_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2,147,483,647_{ten}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

maxint

minint

Integer Overflow

Example:

maxint + 1 = *minint* (≠ *maxint* + 1 in real world)

maxint

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$

Integer Overflow in C


```
int i = 2147483647;

printf("%d\n", i);
printf("%d\n", i + 1);
```

Output:

```
2147483647
-2147483648
```

Zero-Day Alert: Google Chrome Under Active Attack, Exploiting New Vulnerability

Nov 29, 2023 Ravie Lakshmanan

Zero-Day / Web Browser

— Trending News

Google has rolled out security updates to fix seven security issues in its Chrome browser, including a zero-day that has come under active exploitation in the wild.

Tracked as CVE-2023-6345, the high-severity vulnerability has been described as an ***integer overflow*** bug in Skia, an open source 2D graphics library.

Another Benefit of 2's-Complement: Simplifies Overflow Detection

Basic idea: If …

Positive Number + Positive Number == Negative Number

Or…

Negative Number + Negative Number == Positive Number

Overflow is occurred!

Another Benefit of 2's-Complement: Simplifies Overflow Detection

From a logic circuit perspective:

$$
\begin{array}{rl}
 & 010 = 2 \\
+ & 011 = 3 \\
\hline
 & 101 = -3\ (\neq 5)
\end{array}
$$

Another Benefit of 2's-Complement: Simplifies Overflow Detection

From a logic circuit [perspective]:

Carry in (C_{in})	Carry out (C_{out})	Overflow occurred
0	0	X
0	1	O
1	0	O
1	1	X

$$
\begin{array}{r}
010 = 2 \\
+\ 011 = 3 \\
\hline
101 = -3\ (\neq 5)
\end{array}
$$

Another Benefit of 2's-Complement: Simplifies Overflow Detection

From a logic circuit … ective:

Carry in (C_{in})	Carry out (C_{out})	Overflow occurred
0	0	X
0	1	O
1	0	O
1	1	X

Exception:
Overflow has occurred

C_{out} C_{in}

0 1

$$
\begin{array}{r}
010 = 2 \\
+\ 011 = 3 \\
\hline
101 = -3\ (\neq 5)
\end{array}
$$

2's-Complement Signed Integers

- 31st-bit (MSB) is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation

- Some specific numbers
  - 0: `0000 0000 … 0000`
  - –1: `1111 1111 … 1111`
  - Most-negative: `1000 0000 … 0000`
  - Most-positive: `0111 1111 … 1111`

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2

$+2 = 0000\ 0000\ \ldots\ 0010_2$

$-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

$= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value

- **Replicate the sign bit to the left**
  - c.f. unsigned extension: extend with 0s

- Examples: 8-bit to 16-bit

```
+2: 0000 0010 => 0000 0000 0000 0010
–2: 1111 1110 => 1111 1111 1111 1110
```

Sign Extension in C

```
char c = 0x80;      // equivalent to -128
int i = c;          // sign extension
printf("0x%x\n", i);
```

Output:

```
0xffffff80
```

Float / Double

Floating-point Number: Motivation

We need a way to represent ...

- *Infinite decimal (e.g., 3.1415926535…)*
- *Very small numbers*
- *Very large numbers*

from computer!

Solution: Floating-point Number Representation

Floating-point Number: Motivation

We need a way to represent ...

- *Infinite decimal (e.g., 3.1415926535…)*

3.1415

- *Very small numbers*

0.001×10^{-20}

- *Very large numbers*

3.15576×10^{19}

Can be represented with a limited number of bits!

Solution: Floating-point Number Representation

Floating-point Number: Notations

- **Scientific notation**: renders numbers with a single digit to the *left* of the point
  - Example: 7.15576×10^4, 0.314×10^1

Normalized

- **Normalized scientific notation**: scientific notation that has *no leading 0s*
  - Example: 7.15576×10^4, 3.14×10^0

Floating-point Number: Notations

- **Scientific notation**: renders numbers with a single digit to the *left* of the point

1 ≤ Significand < 10

- **Normalized scientific notation**: scientific notation that has *no leading 0s*
  - Example: 7.15576×10^4, 3.14×10^0

Significand

Base

Exponent

(sign) x significand x base^{exponent}

Floating-point Number: Binary

- **Scientific notation**: renders numbers with a single digit to the *left* of the point

  $1 \le$ Significand < 2

- **Normalized scientific notation**: scientific notation that has *no leading 0s* ***(= Always 1 to the left of the point)***
  - Example: **1.1×2^{-1}** ($0.75_{ten} \rightarrow 0.5+0.25 \rightarrow 0.11_{two} \rightarrow 1.1 \times 2^{-1}$)

2

(sign) x significand x $\text{base}^{\text{exponent}}$

Floating-point Number: Binary

- **Scientific notation**: renders numbers with a single digit to the *left* of the point

1 ≤ Significand < 2

How are floating-point numbers represented in computers?

→ 0.11_{two} → 1.1×2^{-1})

2

(sign) x significand x base^{exponent}

IEEE 754 Floating-point Standard

- Developed in response to *divergence of representations*

Divergence of representations

Normalized representation

IEEE 756 representation

$0.11_{two} \equiv 1.1_{two} \times 2^{-1} \equiv 11_{two} \times 2^{-2}$

$1.1_{two} \times 2^{-1}$

S	Exponent	Fraction

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Sign
- Single: 1 bit
- Double: 1 bit

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

$$+1.1 \times 2^{-1}{}_{two}$$

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

$+1.1 \times 2^{-1}{}_{two}$

Leading 1 bit is implicit

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

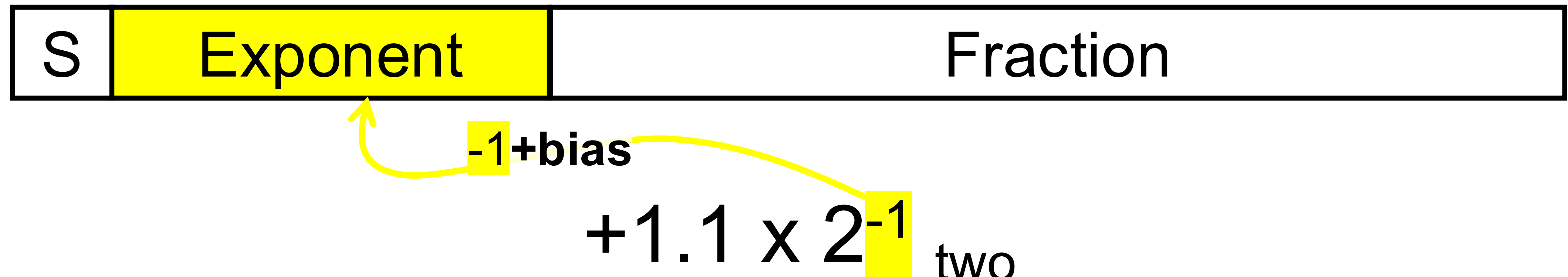
-1+bias

$+1.1 \times 2^{-1}_{two}$

Leading 1 bit is implicit

Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*



Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*

Exponent (8 bits)	Decimal (Unsigned)	Biased by 127
00000000	0	0-127 = -127
00000001	1	1-127 = -126
00000010	2	2-127 = -125
…	…	…
11111111	255	255-127 = 128

S	Exponent	

-1+bias

$+1.1 \times 2^{-1}_{two}$

Exponent: Why Biased?

- To make sorting easier
- **Bias**: *127 for single precision* and *1023 for double precision*

Exponent (8 bits)	Decimal (Unsigned)	Biased by 127
00000000	0	0-127 = -127
00000001	1	1-127 = -126
00000010	2	2-127 = -125
…	…	…
11111111	255	255-127 = 128

S | Exponent

-1+bias

$+1.1 \times 2^{-1}_{two}$

Easy sorting

Tradeoff between Precision and Range

- Increasing *the size of the fraction* enhances ***the precision***
  - Shorter length: 1.1×2^{-1}
  - Longer length: 1.10110×2^{-1}
- Increasing *the size of the exponent* increases ***the range***
  - Shorter length: 1.1×2^{2}
  - Longer length : 1.1×2^{23}

S	Exponent	Fraction

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

-1+bias

$+1.1 \times 2^{-1}_{\text{two}}$

Leading 1 bit is implicit

IEEE 754 Floating-point Standard

- Two representations
  - **Single precision (32-bit)**: type `float` in C
  - **Double precision (64-bit)**: type `double` in C

Sign
- Single: 1 bit
- Double: 1 bit

Exponent
- Single: 8 bits
- Double: 11 bits

Fractional part of significand
- Single: 23 bits
- Double: 52 bits

S	Exponent	Fraction

- 0: positive (+)
- 1: negative (-)

-1+bias

$+1.1 \times 2^{-1}_{two}$

Leading 1 bit is implicit

IEEE 754 Floating-point Standard: Summary

$$(-1)^{sign} \times (1+fraction) \times base^{exponent-bias}$$

S	Exponent	Fraction

Single-precision Range

- Exponents 00000000 and 11111111 reserved

- **Smallest value**
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
  - Exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

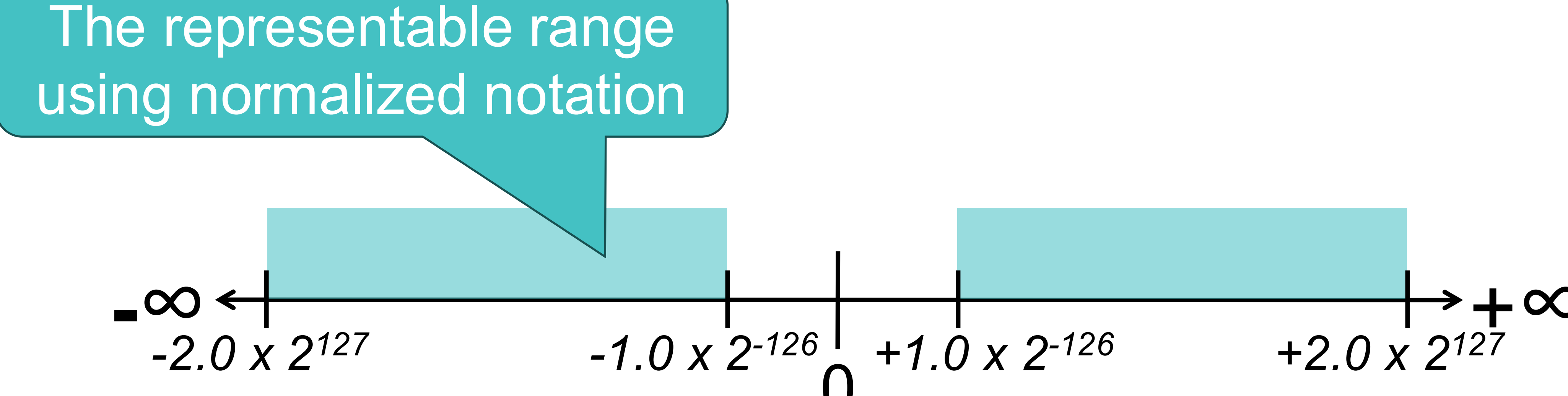
Double-precision Range

- Exponents 0000…00 and 1111…11 reserved

Used for special cases

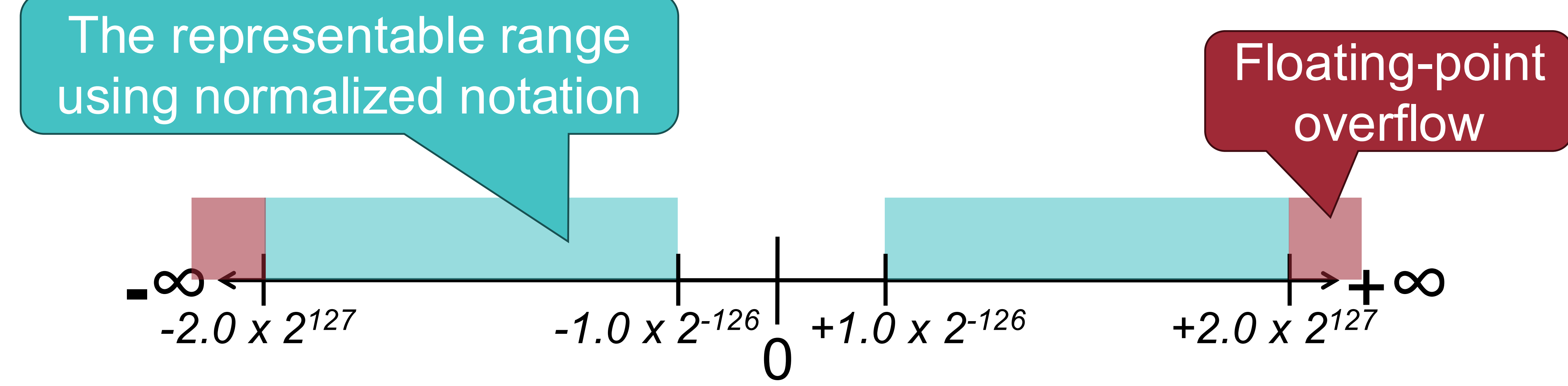
- **Smallest value**
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Overflow and Underflow



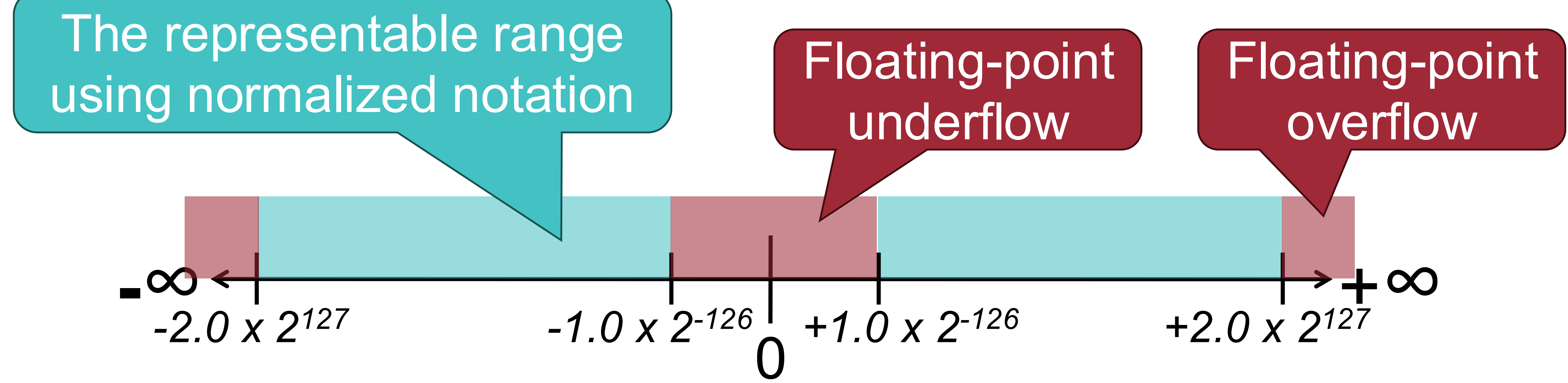
Overflow and Underflow

- **Overflow**: occurs when a result has a magnitude ***too big*** to be represented

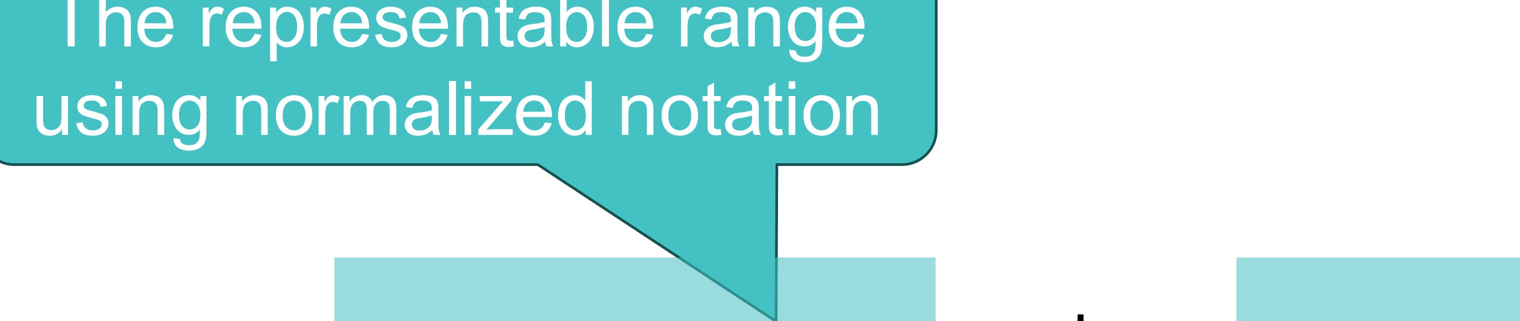


Overflow and Underflow

- **Overflow**: occurs when a result has a magnitude ***too big*** to be represented
- **Underflow**: occurs when a result has a magnitude ***too small*** to be represented

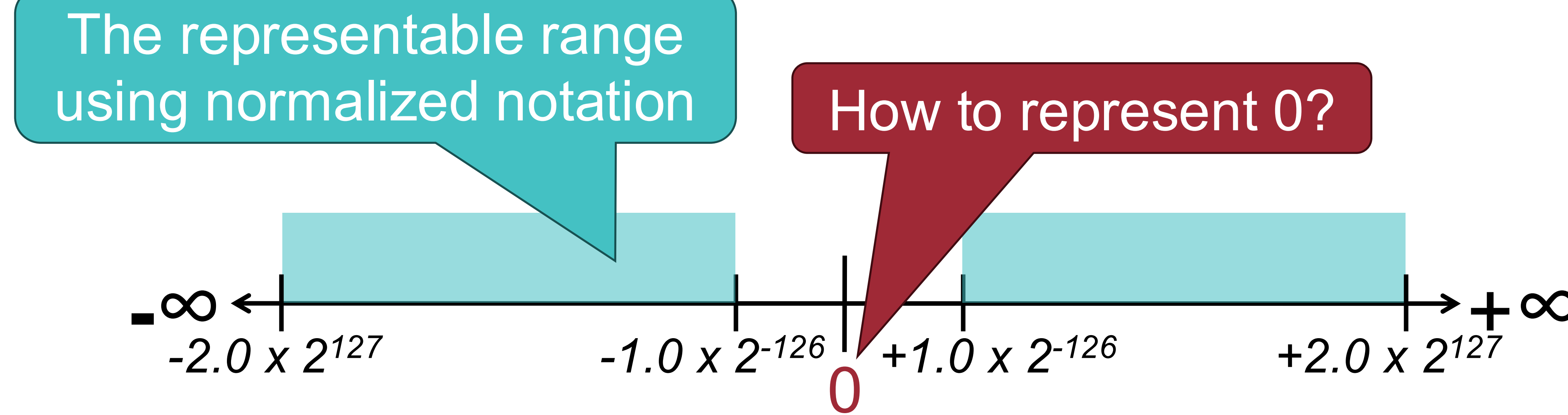


IEEE 754: Special Cases



IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  → Not 1.0×2^{-127} but **0**



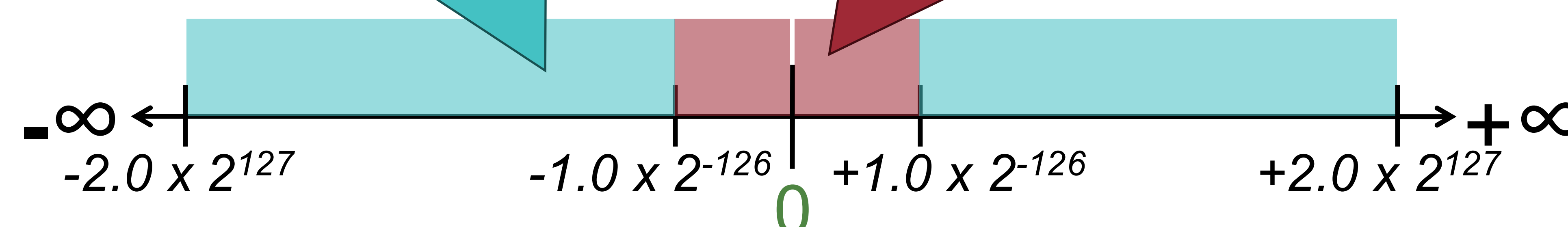
IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  → Not 1.0 x 2^{-127} but **0**

- Exponent = 00…0, Fraction ≠ 00…0
  → Not (1 + fraction) x 2^{-127} but **(0 + fraction) x 2^{-126}**
  → Denormalized real numbers (to represent very small numbers)

The representable range using normalized notation

How to represent very small numbers?
→ ***Denormalized numbers***



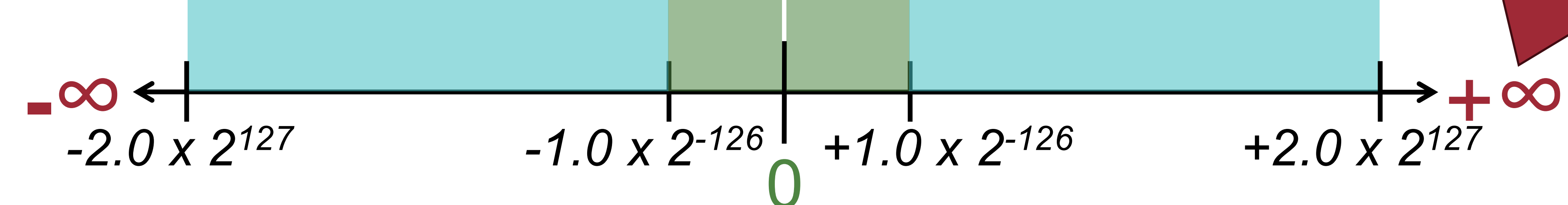
IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  → Not 1.0×2^{-127} but **0**

- Exponent = 00…0, Fraction ≠ 00…0
  → Not $(1 + \text{fraction}) \times 2^{-127}$ but **$(0 + \text{fraction}) \times 2^{-126}$**
  → Denormalized real numbers (to represent very small numbers)

- Exponent = 11…1, Fraction = 00…0
  → **±infinity**

How to represent infinity?



IEEE 754: Special Cases

- Exponent = 00…0, Fraction = 00…0
  - → Not 1.0 x 2^{-127} but **0**
- Exponent = 00…0, Fraction ≠ 00…0
  - → Not (1 + fraction) x 2^{-127} but **(0 + fraction) x 2^{-126}**
  - → Denormalized real numbers (to represent very small numbers)
- Exponent = 11…1, Fraction = 00…0
  - → **±infinity**
- Exponent = 11…1, Fraction ≠ 00…0
  - → **Not-a-Number (*NaN*)**
  - → Indicates illegal or undefined result

How to represent the result of invalid operations (e.g., 0/0)?

IEEE 754 Encoding of Floating-point Numbers

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Question?