Floating point

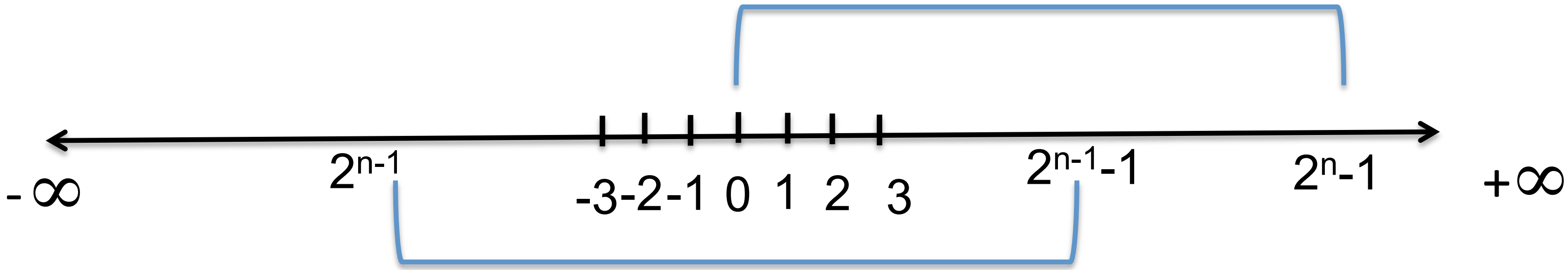
Jinyang Li

Some are based on Tiger Wang's slides

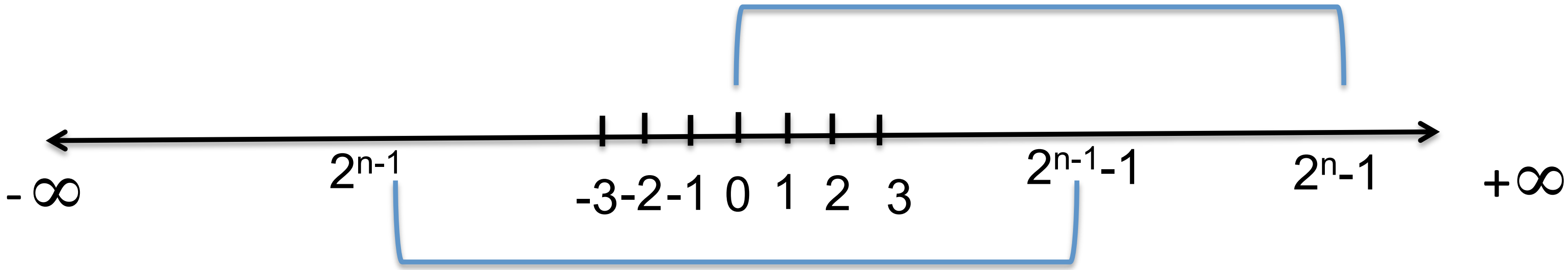
Representing Real Numbers using bits

Representing Numbers in bits

What we have studied



Representing Numbers in bits



Today: How to represent fractional numbers?

Representing real numbers: decimal

Real Numbers	Decimal Representation (Expansion)
11 / 2	$(5.5)_{10}$
1 / 3	$(0.3333333...)_{10}$
$\sqrt{2}$	$(1.4128...)_{10}$

Representing real numbers: decimal

Real Numbers	Decimal Representation (Expansion)
$11 / 2$	$(5.5)_{10}$
$1 / 3$	$(0.3333333...)_{10}$
$\sqrt{2}$	$(1.4128...)_{10}$

$(5.5)_{10} = 5 * 10^{0} + 5 * 10^{-1}$

$(0.333333...)_{10} = 3 * 10^{-1} + 3 * 10^{-2} + 3 * 10^{-3} + ...$

$(1.4128...)_{10} = 1 * 10^{0} + 4 * 10^{-1} + 1 * 10^{-2} + 2 * 10^{-3} + ...$

Representing real numbers: decimal

Real Numbers	Decimal Representation (Expansion)
11 / 2	$(5.5)_{10}$
1 / 3	$(0.3333333...)_{10}$
√2	$(1.4128...)_{10}$

$(5.5)_{10} = 5 * 10^0 + 5 * 10^{-1}$

$(0.333333...)_{10} = 3 * 10^{-1} + 3 * 10^{-2} + 3 * 10^{-3} + ...$

$(1.4128...)_{10} = 1 * 10^0 + 4 * 10^{-1} + 1 * 10^{-2} + 2 * 10^{-3} + ...$

$$r_{10} = (d_m d_{m-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots d_{-n})_{10}$$

$$= \sum_{i=-n}^{m} 10^i \times d_i$$

Binary Representation

$(5.5)_{10} = 4 + 1 + 1 / 2$

$= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1}$

Binary Representation

$$(5.5)_{10} = 4 + 1 + 1 / 2$$

$$= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1}$$

$$= (101.1)_2$$

Binary Representation

$$(5.5)_{10} = 4 + 1 + 1 / 2$$
$$= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1}$$
$$= (101.1)_2$$

$$(0.333333...)_{10} = 1 / 4 + 1 / 16 + 1 / 64 + ...$$
$$= (0.01010101...)_2$$

Binary Representation

$$r_{10} = (d_m d_{m-1} d_1 d_0 \cdot d_{-1} d_{-2} \ldots d_{-n})_{10}$$

$$= (b_p b_{p-1} b_1 b_0 \cdot b_{-1} b_{-2} \ldots b_{-q})_2$$

2^p

2^{p-1}

2

1

$$b_p b_{p-1} \cdots b_1 b_0 \cdot b_{-1} b_{-2} \cdots b_{-q} = \sum_{i=-q}^{p} 2^i \times b_i$$

1/2

1/4

$1/2^q$

Exercise

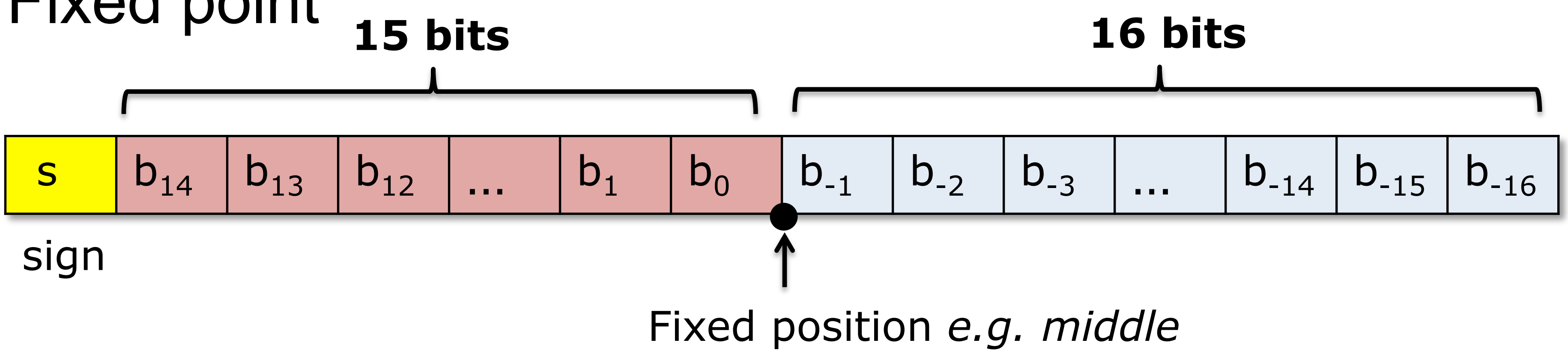
Binary Expansion	Formula	Decimal
10.011_2		
	$2^{-3} + 2^{-4} + 2^{-6}$	
	$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$	

Exercise

Binary Expansion	Formula	Decimal
10.011_2	$2^1 + 2^{-2} + 2^{-3}$	2.375_{10}
0.001101_2	$2^{-3} + 2^{-4} + 2^{-6}$	0.203125_{10}
0.1111_2	$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$	0.9375_{10}

How to represent real numbers in fixed # of bits?

Fixed point



Naive idea: Fixed point

Fixed point

15 bits **16 bits**

s	b_{14}	b_{13}	b_{12}	...	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	...	b_{-14}	b_{-15}	b_{-16}

sign

Fixed position *e.g. middle*

$(10.011)_2$

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

Problems of Fixed Point

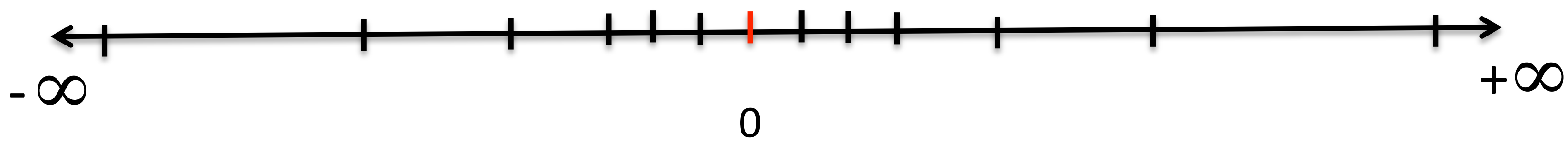
Limited range and precision: e.g., 32 bits

- Largest number: 2^{15} $(011\ldots111)_2$
- Highest precision: 2^{-16}

→Rarely used (No built-in hardware support)

The idea

- Limitation of fixed point notation:
  - Represents evenly spaced fractional numbers
    - ➔ hard tradeoff between high precision and high magnitude
- How about un-even spacing between numbers?



Floating Point: decimal

Based on the normalized scientific notation

$r_{10} = \pm M * 10^{E}$, where $1 <= M < 10$

M: significant (mantissa), E: exponent

Normalized form cannot represent 0!

Floating Point: decimal

Example:

$365.25 = 3.6525 * 10^{2}$

$0.0123 = 1.23 * 10^{-2}$

Decimal point **floats** to the position immediately after the first nonzero digit.

Floating Point: binary

Binary (normalized) scientific notation:

$r_{10} = \pm M * 2^{E}$, where $1 <= M < 2$

$M = (1.b_1 b_2 b_3 \ldots b_n)_2$

M: significant, E: exponent

$(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$

Exercises

The scientific notation of $(10.25)_{10}$ is ?

Exercises

The scientific notation of $(10.25)_{10}$ is ?

$$(10.25)_{10} = (1010.01)_2 = (1.01001)_2 * 2^3$$

How to represent a binary scientific notation in fixed # of bits?

$r_{10} = \pm M * 2^E$, where $1 <= M < 2$

$M = (1.b_1b_2b_3…b_n)_2$

M: significant, E: exponent

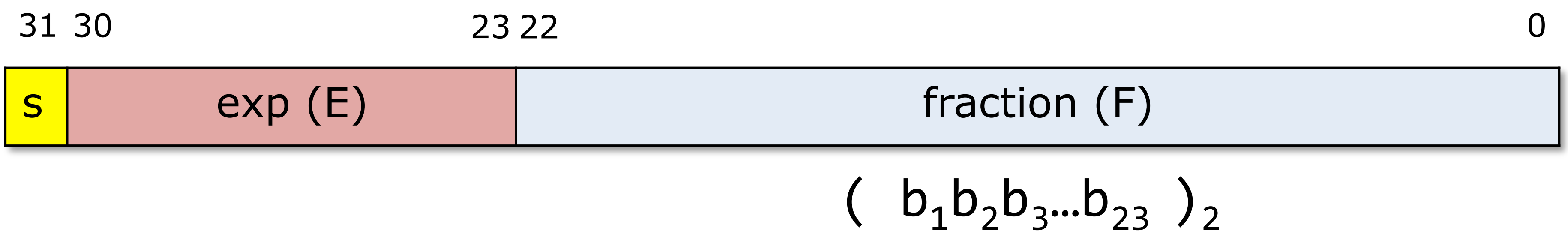
$(1.b_1b_2b_3…b_n)_2$

How to represent a binary scientific notation in fixed # of bits?

$r_{10} = \pm M * 2^E$, where $1 <= M < 2$

$M = (1.b_1b_2b_3 \ldots b_{23})_2$

M: significant, E: exponent



How to represent a binary scientific notation in fixed # of bits?

$r_{10} = \pm M * 2^{E}$, where $1 <= M < 2$

$M = (1.b_1 b_2 b_3 \ldots b_{23})_2$

M: significant, E: exponent

31	30 … 23	22 … 0
0	0000 0010	0110 0000 0000 0000 0000 000

$(b_1 b_2 b_3 \ldots b_{23})_2$

$(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$

Exercise

What's the normalized representation of $(71)_{10}$ and $(10.25)_{10}$

Exercise

What's the normalized representation of $(71)_{10}$ and $(10.25)_{10}$

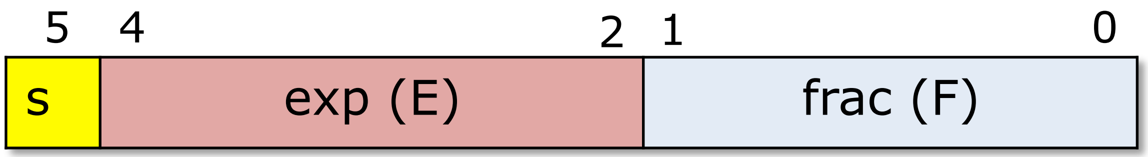
$(10.25)_{10} = (1010.01)_2 = (1.01001)_2 * 2^3$

31	30 … 23	22 … 0
0	0000 0011	0100 1000 0000 0000 0000 000

$(71)_{10} = (1000111)_2 = (1.000111)_2 * 2^6$

31	30 … 23	22 … 0
0	0000 0110	0001 1100 0000 0000 0000 000

Toy Number System



6-bit floating point representation

- exponent: 3 bits
- fraction: 2 bits

Largest positive number ?

Toy Number System

5	4 2	1 0
s	exp (E)	frac (F)

6-bit floating point representation

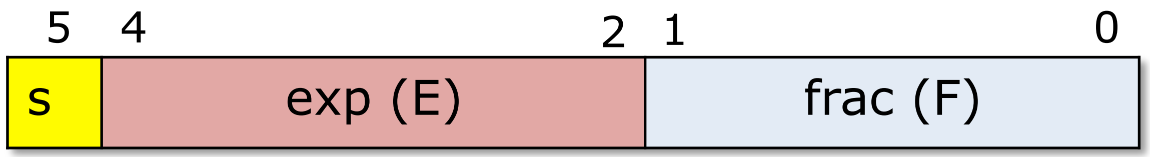
- exponent: 3 bits
- fraction: 2 bits

Largest positive number ?

0	1 1 1	1 1

$(1.11)_2 * 2^7 = 224$

Toy Number System



6-bit floating point representation

- exponent: 3 bits
- fraction: 2 bits

Largest positive number: 224

Smallest positive number ?

Toy Number System

5	4 – 2	1 – 0
s	exp (E)	frac (F)

6-bit floating point representation

- exponent: 3 bits
- fraction: 2 bits

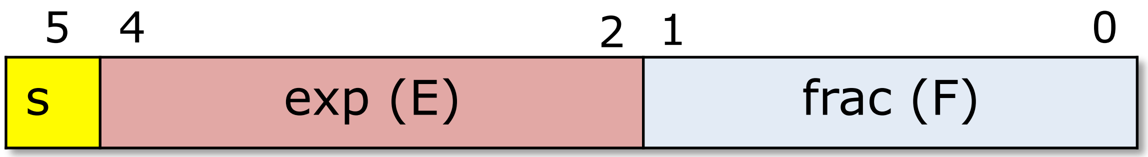
Largest positive number: 224

Smallest positive number: 1

0	0 0 0	0 0

$(1.00)_2 * 2^0 = 1$

Toy Number System



6-bit floating point representation

– exponent: 3 bits
– fraction: 2 bits

Positive number: 1 to 224

Negative number: -224 to -1

No more bit patterns left to represent numbers (-1, 1)

Questions

How to represent

1. numbers close or equal to 0?
2. special cases:
   - the result of dividing by 0, e.g. 1/0 ?
   - $$\infty * 0$$

Lots of different implementations around 1950s!

IEEE Floating Point Standard

IEEE p754
A standard for binary
floating point representation

Prof. William Kahan
University of California at Berkeley
Turing Award (1989)

The Only Book Focuses On IEEE Floating Point Standard

Numerical Computing with IEEE Floating Point Arithmetic

Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises

Michael L. Overton
Courant Institute of Mathematical Sciences
New York University
New York, New York

hardware. This degree of altruism was so astonishing that MATLAB's creator Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754."

https://cs.nyu.edu/overton/NumericalComputing/protected/NumericalComputingSIAM.pdf
With you nyu netid/password. You can also search the pdf with google.

What we have learnt so far

- normalized representation of floating point

$$r_{10} = \pm M * 2^{E} \qquad M = (1.b_1b_2b_3\ldots b_{23})_2$$

31	30 … 23	22 … 0
0	0000 0010	0110 0000 0000 0000 0000 000

$$(5.5)_{10} = (101.1)_2 = (1.011)_2 * 2^2$$

- how to represent numbers in range (-1,1)
- how to represent special cases? e.g. ∞

Goals of IEEE Standard

- Consistent representation of floating point numbers
- Correctly rounded floating point operations, using several rounding modes.
- Consistent treatment of exceptional situations such as division by zero

Restrictions on Normalized Representation

$$r_{10} = \underline{+}M * 2^E \quad M = (\ 1.b_0b_1b_2b_3...b_n\ )_2$$

31	30 … 23	22 … 0
s	exp (E)	fraction (F)

$(\ b_0b_1b_2b_3...b_n\ )_2$

E can not be $(1111\ 1111)_2$ or $(0000\ 0000)_0$

E_{max} = ? 254, $(1111\ 1110)_2$

E_{min} = ? 1, $(0000\ 0001)_2$

Exponential Bias

$$r_{10} = \underline{+}M * 2^{E},\ M = (\ 1.b_0b_1b_2b_3\ldots b_n\ )_2$$

To represent (-1,1),
we must allow
negative exponent.

- How to represent negative E?
  - ~~2's complement~~
  - use bias

31	30 … 23	22 … 0
s	exp (E) + 127	fraction (F)

Bias: 127

$(\ b_0b_1b_2b_3\ldots b_n\ )_2$

IEEE normalized representation

$$r_{10} = \pm M * 2^{E}, \quad M = (\ 1.b_0 b_1 b_2 b_3 \ldots b_n\ )_2$$

31	30 … 23	22 … 0
s	exp (E) + 127	fraction (F)

$$(\ b_0 b_1 b_2 b_3 \ldots b_n\ )_2$$

Bias: 127

$E_{max} = 254 - 127 = 127$

$E_{min} = 1 - 127 = -126$

Smallest positive number 2^{-126}

Negative number with smallest absolute value: -2^{-126}

Questions

Q1. Why using **bias**?

Q2. Why is **bias** 127?

Questions

Q1. Why using **bias** instead of 2’s complement?

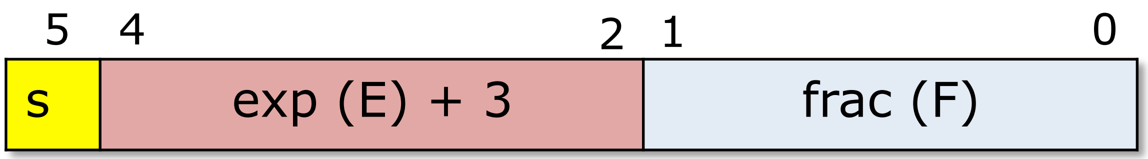
Answer: easier circuitry for comparison.

Questions

Q2. Why is bias 127?

A2. Balance positive exponents (magnitude) and negative exponents (precision)

Toy 6-bit Floating Point



6-bit floating point representation

- exponent: 3 bits
- fraction: 2 bits
- **bias: 3**

Smallest positive number ?

Toy 6-bit Floating Point

5	4 … 2	1 … 0
s	exp (E) + 3	frac (F)

6-bit floating point representation

- exponent: 3 bits
- fraction: 2 bits
- **bias: 3**

Smallest positive number: 0.25

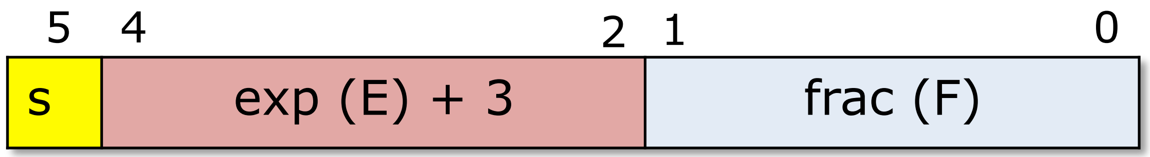
0	0 0 1	0 0

$(1.00)_2 * 2^{-2} = 0.25$

Smallest number >0.25?

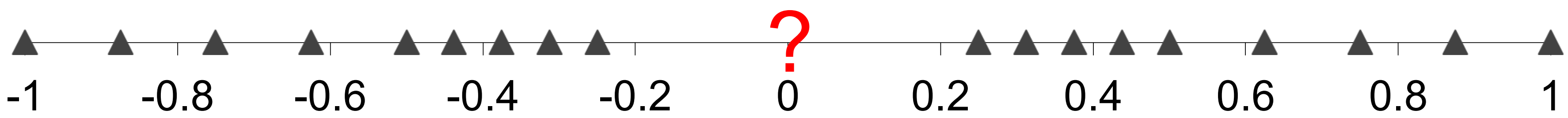
$(1.01)_2 * 2^{-2} = 0.25+0.0625$

Toy 6-bit Floating Point



6-bit floating point representation

– exponent: 3 bits
– fraction: 2 bits
– **bias: 3**



represent values which are close and equal to 0

IEEE denormalized representation

$$r_{10} = \underline{+}M * 2^{E}$$

Normalized Encoding:

31	30 … 23	22 … 0
s	exp (E) + 127	fraction (F)

$1 <= M < 2, M = (1.F)_2$

Denormalized Encoding:

31	30 … 23	22 … 0
s	0000 0000	fraction (F)

$E = 1 – Bias = -126$ $0 <= M < 1, M = (0.F)_2$

Zeros

+0.0

0	0000 0000	0000 0000 0000 0000 0000 000

-0.0

1	0000 0000	0000 0000 0000 0000 0000 000

Denormalized representation examples

$(0.1)_2 * 2^{-126}$

0	0000 0000	1000 0000 0000 0000 0000 000

$-(0.010101)_2 * 2^{-126}$

1	0000 0000	0101 0100 0000 0000 0000 000

Special Values

Special Value's Encoding:

31	30 … 23	22 … 0
s	1111 1111	fraction (F)

values	sign	frac
$+\infty$	0	all zeros
$-\infty$	1	all zeros
NaN	any	non-zero

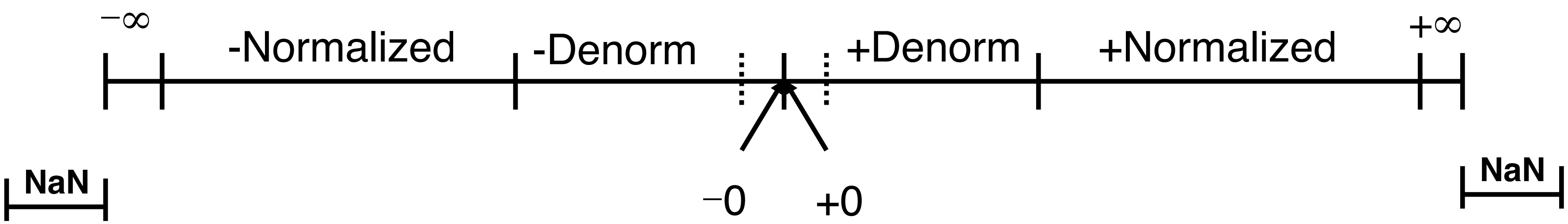
Exercises

representation	**E**	**M**	**V**
0100 1001 0101 0000 0000 0000 0000 0000			
			$2.5 * 2^{-127}$
			$-1.25 * 2^{-111}$
1111 1111 1111 1111 0000 0000 0000 0000			
1111 1111 1000 0000 0000 0000 0000 0000			
			$1.5 * 2^{-127}$

Exercises

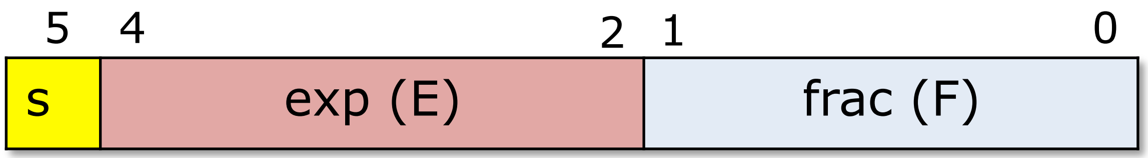
representation	E	M	V
0100 1001 0101 0000 0000 0000 0000 0000	146 – 127 = 19	$(1.101)_2$ = 1.625	$1.625 * 2^{19}$
0000 0000 1010 0000 0000 0000 0000 0000	1 – 127 = -126	$(1.01)_2$ = 1.25	$2.5 * 2^{-127}$ $= (1.01)_2 * 2^{-126}$
1000 1000 0010 0000 0000 0000 0000 0000	16 – 127 = -111	$(1.01)_2$ = 1.125	$-1.25 * 2^{-111}$
1111 1111 1111 1111 0000 0000 0000 0000	-	-	Nan
1111 1111 1000 0000 0000 0000 0000 0000	-	-	- ∞
0000 0000 0110 0000 0000 0000 0000 0000	-126	$(0.11)_2$	$(0.11)_2 * 2^{-126}$ $= 1.5 * 2^{-127}$

Distribution of Representable Values

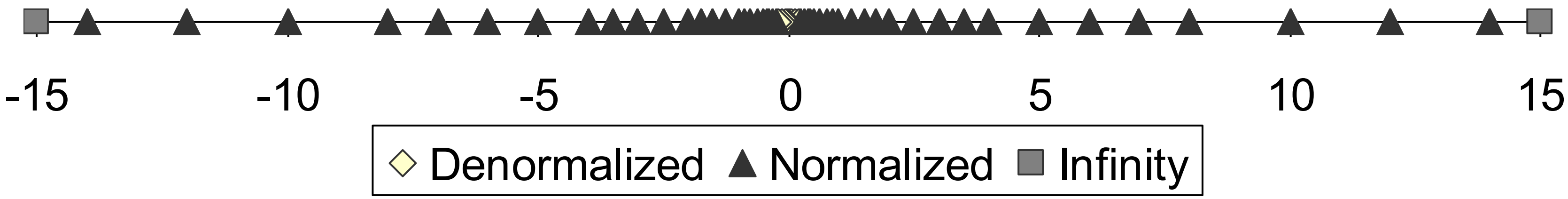


Adjacent floats have adjacent bit representation

Distribution of Representable Values: toy 6-bit FP



- exponent: 3 bits
- fraction: 2 bits
- bias: 3



Smallest number greater than 0?

Largest number?

Single, double precision

s	exp (bias:127)	frac
1	8-bits	23-bits

float f = 0.1;
double d = 0.1;

smallest positive: $(0.00....1)_2 * 2^{-126} \approx 1.4*10^{-45}$

largest positive: $(1.1....1)_2 * 2^{127} \approx 3.4*10^{38}$

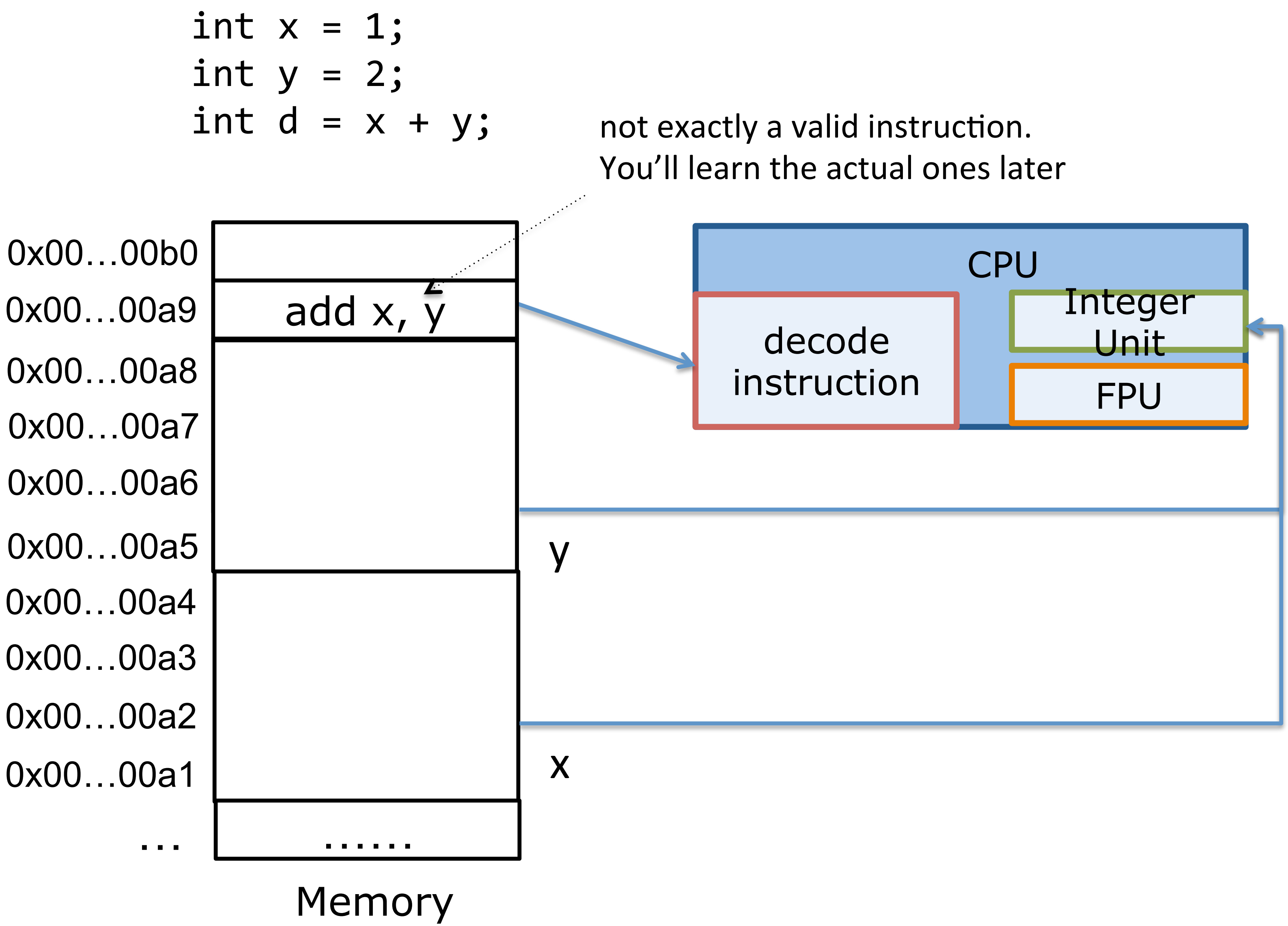
s	exp (bias:1023)	frac
1	11-bits	52-bits

smallest positive: $(0.00....1)_2 * 2^{-1022}$

largest positive: $(1.1...1)_2 * 2^{1023}$

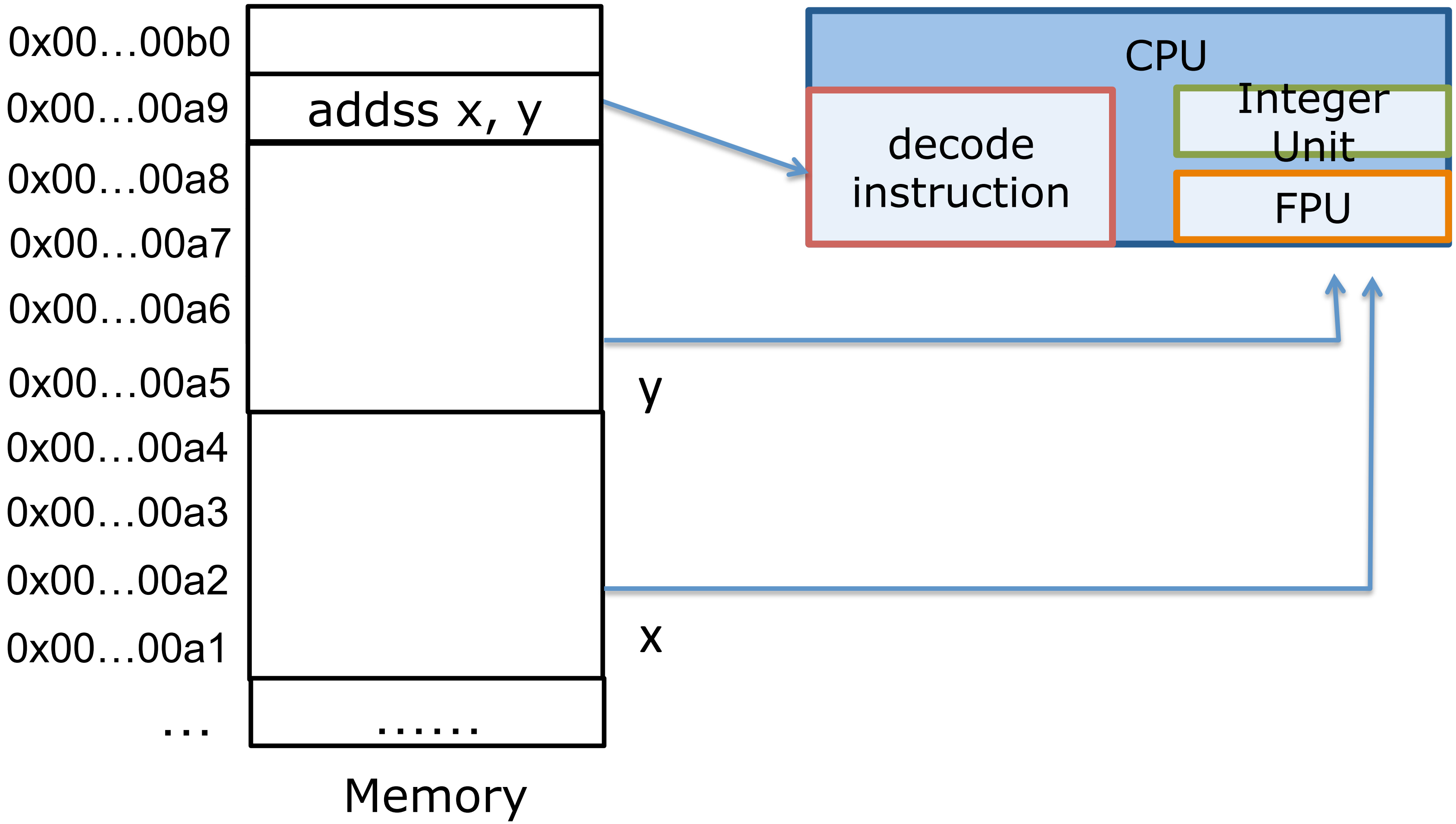
Floating point operations

- Addition, subtraction, multiplication, division etc.
- How does CPU know if a bit pattern is to be interpreted as IEEE float point or integer?



int x = 1;
int y = 2;
int d = x + y;
not exactly a valid instruction.
You'll learn the actual ones later
0x00…00b0
0x00…00a9
add x, y
0x00…00a8
0x00…00a7
0x00…00a6
0x00…00a5
y
0x00…00a4
0x00…00a3
0x00…00a2
0x00…00a1
x
…
……
Memory
CPU
decode instruction
Integer Unit
FPU

```
float x = 1e2;
float y = 2.0;
float d = x + y;
```

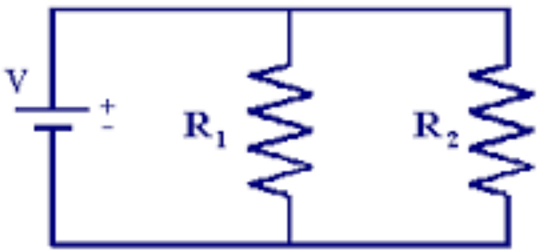


Floating point caveats

- Invalid operation: 0/0, sqrt(-1), $\infty+\infty$
- Divide by zero: $x/0 \rightarrow \infty$
- Overflows: result too big to fit
- Underflows: 0 < result < smallest denormalized value
- Inexact: round it!

Why divide by zero = ∞?

- Allow a calculation to continue and produce a valid result
- Example:



$$Parallel\ resistance = \frac{1}{\frac{1}{R1} + \frac{1}{R2}}$$

If R1 or R2 is 0, overall resistance should be 0

Floating point addition

- Commutative? yes. x+y == y+x
- Associative? (x+y)+z = x + (y+z)?
  - Overflow:

```
(2e38+2e38)-1e38 = inf
 2e38+(2e38-1e38) = 3e38
```

  - Rounding

```
(3.14+1e38)-1e38 = 0
 3.14+(1e38-1e38) = 3.14
```

- Monoticity? a ≥ b ⇒ a+c ≥ b+c?

Floating point multiplication

- Commutative? yes. x* y == y*x
- Associative? (x*y)*z = x * (y*z)?
  - Overflow:
    - `(1e20*1e20)*1e-20= inf,`
    - `1e20*(1e20*1e-20)= 1e20`
  - Rounding:
- Distributive? (x+y)*z = x*z + y*z?
    - `1e20*(1e01-1e20)= 0.0,`
    - `1e20*1e20 - 1e20*1e20 =NaN`
- Monoticity? $a \geq b \Rightarrow a*c \geq b*c$?

Floating point in the real world

- Using floating point to measure distances in games
- Precision diminishes as distance gets larger

FP value	Length	FP precision	Precision size
1	1 meter	1.19E-07	Virus
100	100 meter	7.63E-06	red blood cell
10 000	10 km	0.000977	toenail thickness
1000 000	.16x earth radius	0.0625	credit card width

Table source: Random ASCII

Floating point trouble

- Comparing floats for equality is a bad idea!

```
float f = 0.1;
while (f != 1.0) {
        f += 0.1;
}
```

```
f is 0.2000000030
f is 0.3000000119
f is 0.4000000060
f is 0.5000000000
f is 0.6000000238
f is 0.7000000477
f is 0.8000000715
f is 0.9000000954
f is 1.0000001192
f is 1.1000001431
f is 1.2000001669
...
```

Floating point trouble

- Never count using floating points

```
count = 0;
for (float f = 0.0; f < 1.0; f += 0.1) {
    count++;
}
```

Floating point summary

- Floating points are tricky
  - Precision diminishes as magnitude grows
  - overflow, rounding error
- Many real world disasters due to FP trickiness
  - Patriot Missile failed to intercept due to rounding error (1991)
  - Ariane 5 explosion due to overflow in converting from double to int (1996)