CS 211: Intro to Computer Architecture

3.2: *Fixed and Floating Point Representations*

Minesh Patel

Spring 2025 – Thursday 6 February

Announcements

- **Due**
  - **PA1 tonight** (Gradescope)
  - **WA1 tomorrow** (Gradescope)
- **WA2 and PA2 to be assigned tomorrow (or Saturday, please be patient)**
  - WA2: 1 week
  - PA2: 1.5 weeks

Recap: Interpreting Binary Representations

Object in Memory

`0x22c`

`0xbaad_c0de`

`0x230`

← **What does this represent?**

Text

ASCII	*<error>*
Latin-1	ºÀÞ
UTF-8	*<error>*
UTF-16 BE	몭샞
UTF-16 LE	궺�
Other: ???	???

Number

Unsigned (BE):	`3131949278`
Unsigned (LE):	`3737169338`
Two's Complement (BE):	`-1163018018`
Two's Complement (LE):	`-557797958`
Sign Magnitude (BE):	`-984465630`
Sign Magnitude (LE):	`-1589685690`
Other: ???	???

None of the Above?

Random number generator output
Encrypted ciphertext
Junk (?)
Proprietary format
Metadata

- Every object is **just a collection of bits**
- Can't know what it is supposed to be without:
  - Knowing how to interpret it (image, text, number, code, etc.)
  - Knowing its intended representation (encoding, number format, endianness, etc.)

Can You Find the Unsigned Integer Representations?

Source Code

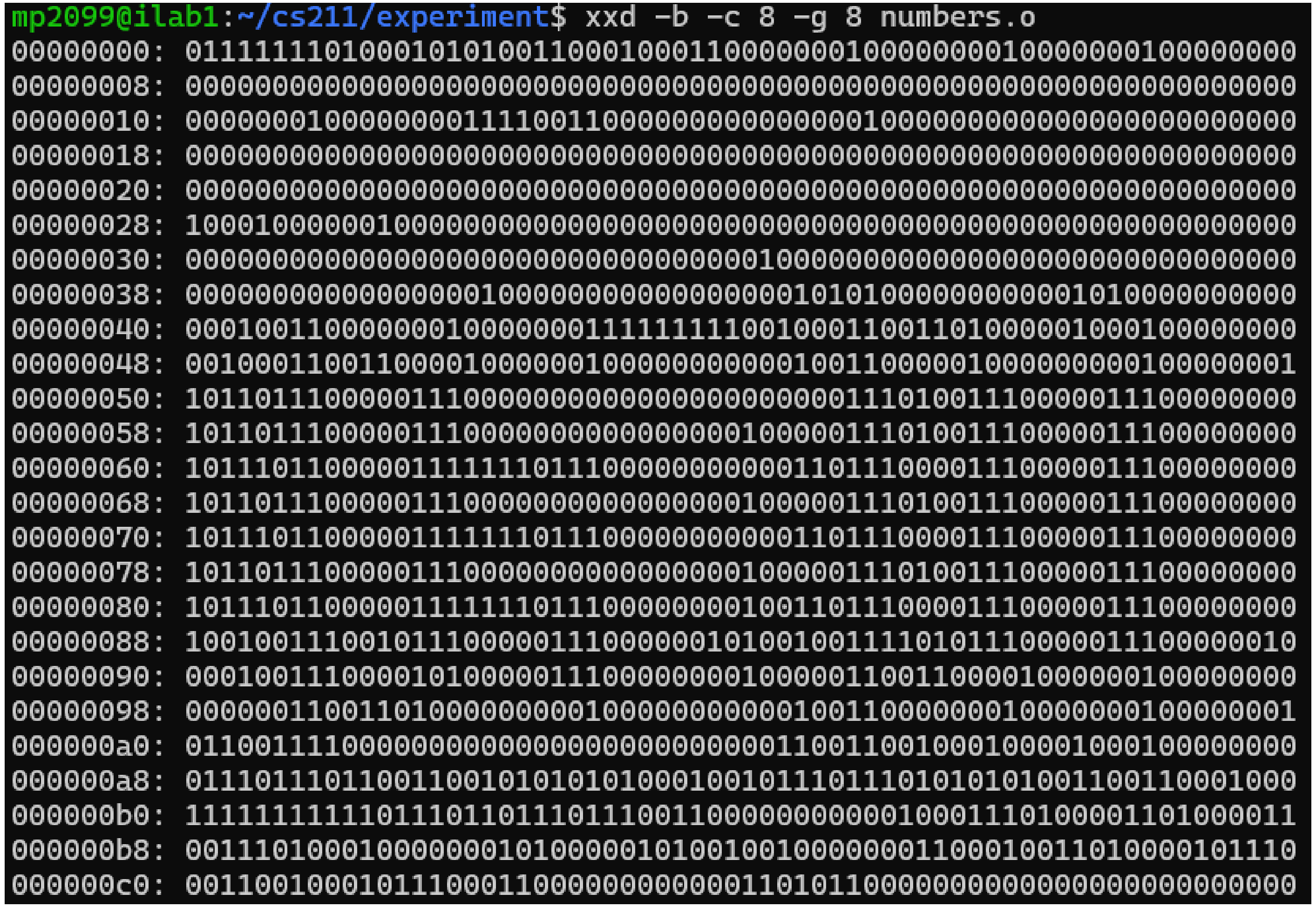
```c
#include <stdint.h>

uint32_t a = 0x00112233;
uint32_t b = 0x44556677;
uint32_t c = 0x8899aabb;
uint32_t d = 0xccddeeff;

uint64_t add(void)
{
   return a + b + c + d;
}
```

Compiler

Machine Code



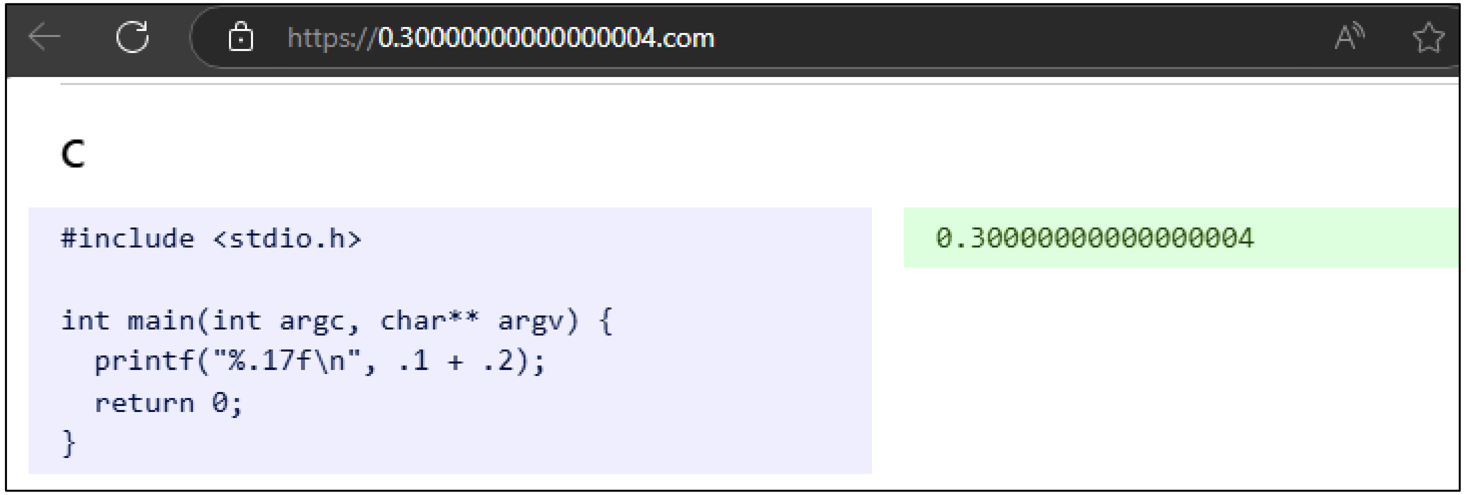
```
mp2099@ilab1:~/cs211/experiment$ xxd -b -c 8 -g 8 numbers.o
00000000: 0111111101000101010011000100011000000010000000010000000100000000
00000008: 0000000000000000000000000000000000000000000000000000000000000000
00000010: 0000000100000000111100110000000000000001000000000000000000000000
00000018: 0000000000000000000000000000000000000000000000000000000000000000
00000020: 0000000000000000000000000000000000000000000000000000000000000000
00000028: 1000100000010000000000000000000000000000000000000000000000000000
00000030: 0000000000000000000000000000000001000000000000000000000000000000
00000038: 0000000000000000010000000000000000010101000000000001010000000000
00000040: 0001001100000001000000011111111100100011001101000001000100000000
00000048: 0010001100110000100000010000000000010011000001000000000100000001
00000050: 1011011100000111000000000000000000000011101001110000011100000000
00000058: 1011011100000111000000000000000010000011101001110000011100000000
00000060: 1011101100000111111101110000000000011011100001110000011100000000
00000068: 1011011100000111000000000000000010000011101001110000011100000000
00000070: 1011101100000111111101110000000000011011100001110000011100000000
00000078: 1011011100000111000000000000000010000011101001110000011100000000
00000080: 1011101100000111111101110000000010011011100001110000011100000000
00000088: 1001001110010111000001110000001010010011110101110000011100000010
00000090: 0001001110000101000001110000000010000011001100001000000100000000
00000098: 0000001100110100000000010000000000010011000000010000000100000001
000000a0: 0110011110000000000000000000000000110011001000100001000100000000
000000a8: 0111011101100110010101010100010010111011101010101001100110001000
000000b0: 1111111111101110110111011100110000000000010001110100001101000011
000000b8: 0011101000100000001010000010100100100000001100010011010000101110
000000c0: 0011001000101110001100000000000011010110000000000000000000000000
```

Floating Point

- **Today:** Approximating **all real numbers** using 32 bits

Floating Point Trickery (Approximation Error)

- **Today:** Approximating **all real numbers** using 32 bits
  - Clearly: 2^{32} `< infinity`
  - Most numbers are **approximately represented**

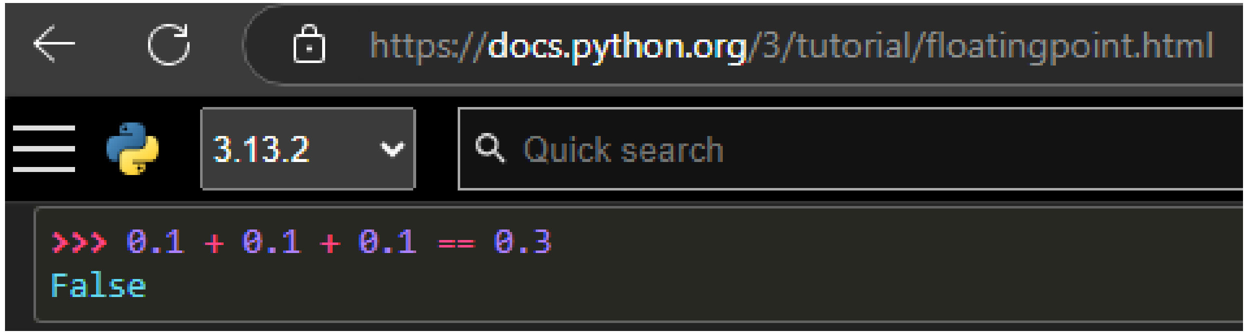


C

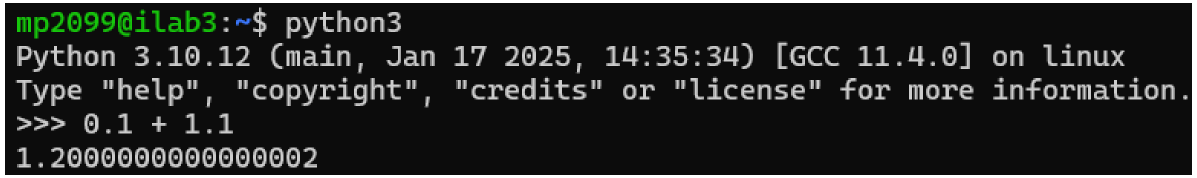
```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("%.17f\n", .1 + .2);
  return 0;
}
```

```
0.30000000000000004
```



```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```



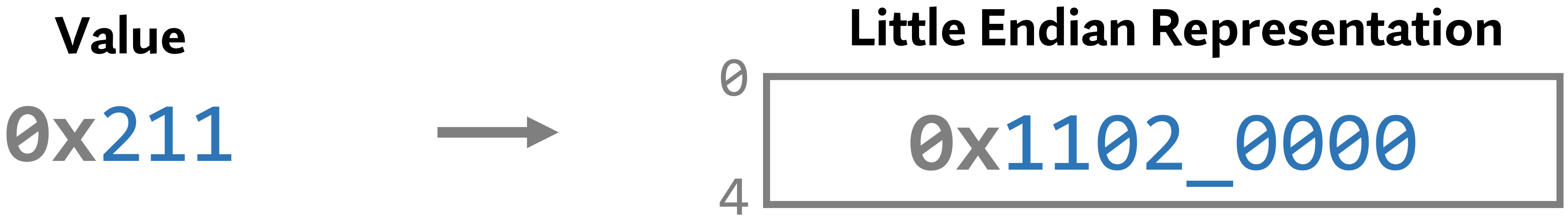
```
mp2099@ilab3:~$ python3
Python 3.10.12 (main, Jan 17 2025, 14:35:34) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0.1 + 1.1
1.2000000000000002
```

Agenda

- **Fixed Point**
- Floating Point
    - IEEE Standard 754
- Special Floating Point Numbers

Fractions

- So far, we've dealt with only whole numbers



Value

`0x211` →

Little Endian Representation

0

4

`0x1102_0000`

- What about fractions?

Fraction

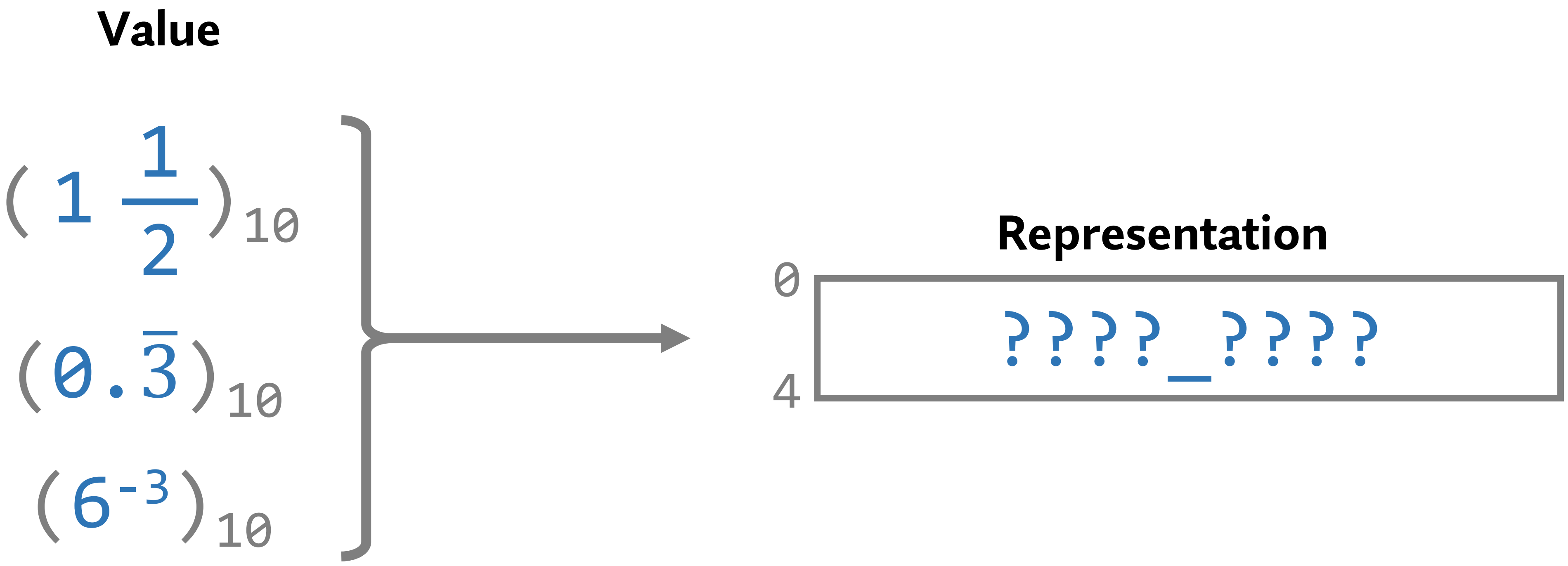
$$\left(2\frac{1}{2}\right)_{10}$$

Decimal

$$(2.5)_{10}$$

Decimal point

Representing Fractions



Value

$(1\frac{1}{2})_{10}$

$(0.\overline{3})_{10}$

$(6^{-3})_{10}$

Representation

`????_????`

- First, let's consider fractions in **other bases**

Fractions in Other Bases

Number	Base 2	Base 10	Base 12	Base 16
$(2\frac{1}{2})_{10}$	$(10.1)_2$	$(2.5)_{10}$	$(2.6)_{12}$	$(2.8)_{16}$
	Binary point	Decimal point	Duodecimal point	Hexadecimal point

- **Nothing's changed:** this is just another positional code
  - The "point" indicates **the position** of the **$(\mathrm{base})^0$** term
  - We now have **negative powers** of the base

$$(10.1)_2$$

$$1\cdot2^1 + 0\cdot2^0 + 1\cdot2^{-1}$$

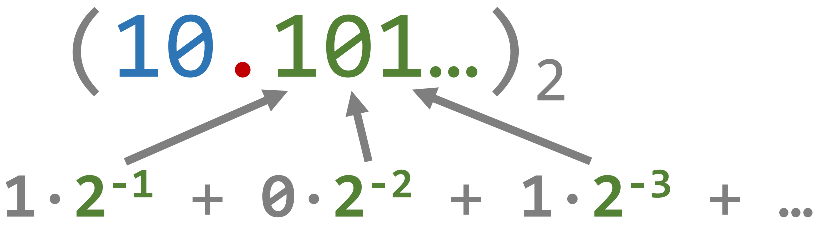
$$(2.5)_{10}$$

$$2\cdot10^0 + 5\cdot10^{-1}$$

$$(2.8)_{16}$$

$$2\cdot16^0 + 8\cdot16^{-1}$$

Negative Power Weights



$$(10.101\ldots)_2$$

$$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + \ldots$$

$(i)_{16}$	2^{-i}
0	1
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
a	0.0009765625
b	0.00048828125
c	0.000244140625
d	0.0001220703125
e	0.00006103515625
f	0.000030517578125

Base Conversion: Other to Decimal

$(802.11)_9$

$(101.1)_2$

$(\mathtt{2fac.e})_{16}$

Base Conversion: Decimal to Other

Integer Part

Algorithm: Divide by the base

Fraction Part

Algorithm: Multiply by the base

$$\begin{tikzcd}
 & (3.75)_{10} \arrow[dl] \arrow[dr] & \\
(3)_{10} \arrow[d] & + & (0.75)_{10} \arrow[d] \\
(11)_{2} \arrow[dr] & + & (0.11)_{2} \arrow[dl] \\
 & (11.11)_{2} &
\end{tikzcd}$$

Base Conversion: Decimal to Other

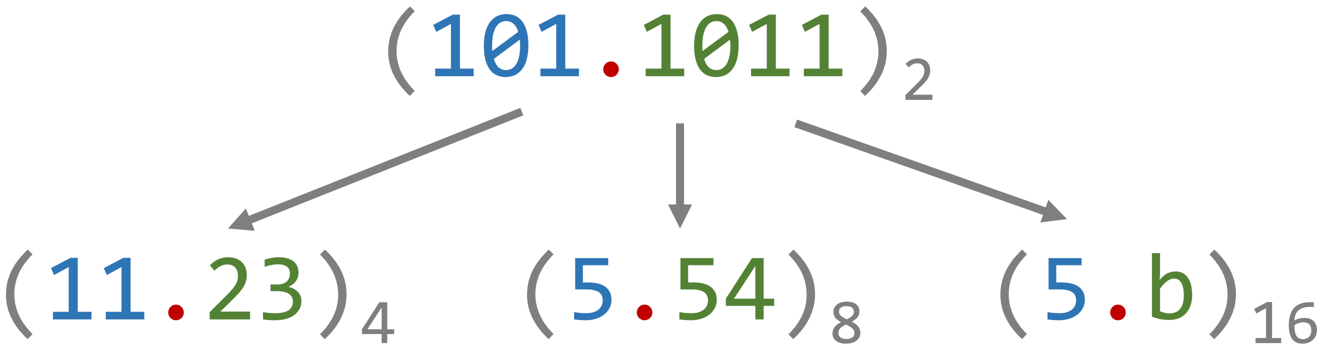
$(10.1)_{10}$

Base Conversion: Other to Other

- Two step conversion

1. Convert to base 10
2. Convert to final base

- **Shortcut**: powers of two



$(101.1011)_2$

$(11.23)_4$ $(5.54)_8$ $(5.b)_{16}$

Binary Representation of Fractions

- **Key idea**: put the point in a "fixed" location (**"fixed point"** representation)
  - Point location is **implied**: hardware does not know about it (treated as an integer)
  - Programmer must remember the point's location

Unsigned Representation

$b_7b_6b_5b_4.b_3b_2b_1b_0$

$$\left(2\frac{1}{2}\right)_{10} \rightarrow (10.1)_2 \rightarrow \boxed{0010\_1000}$$

0

1

Two's Complement Representation

$b_7b_6b_5b_4.b_3b_2b_1b_0$

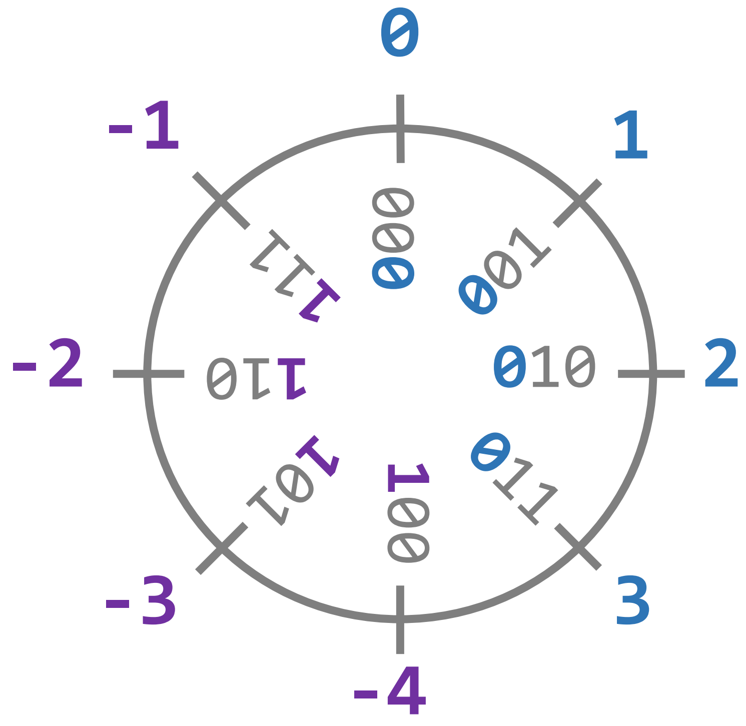
$$\left(-2\frac{1}{2}\right)_{10} \rightarrow (-10.1)_2 \rightarrow \boxed{1101\_1000}$$

0

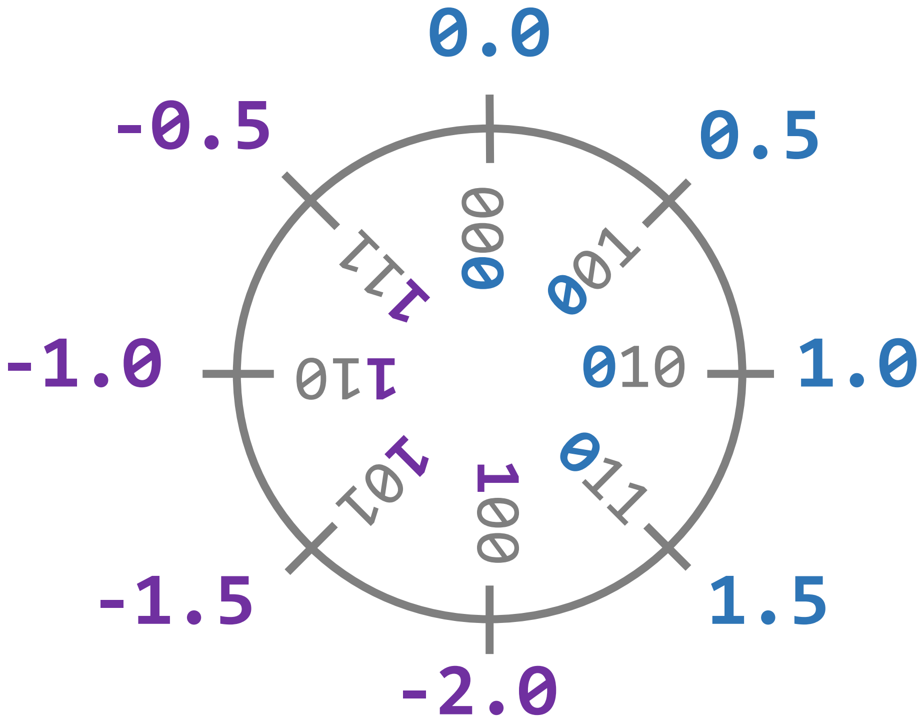
1

Integer vs. Fixed Point

Two's Complement Integer

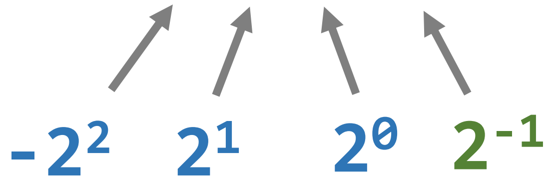


Two's Complement Fixed Point ($b_2b_1.b_0$)

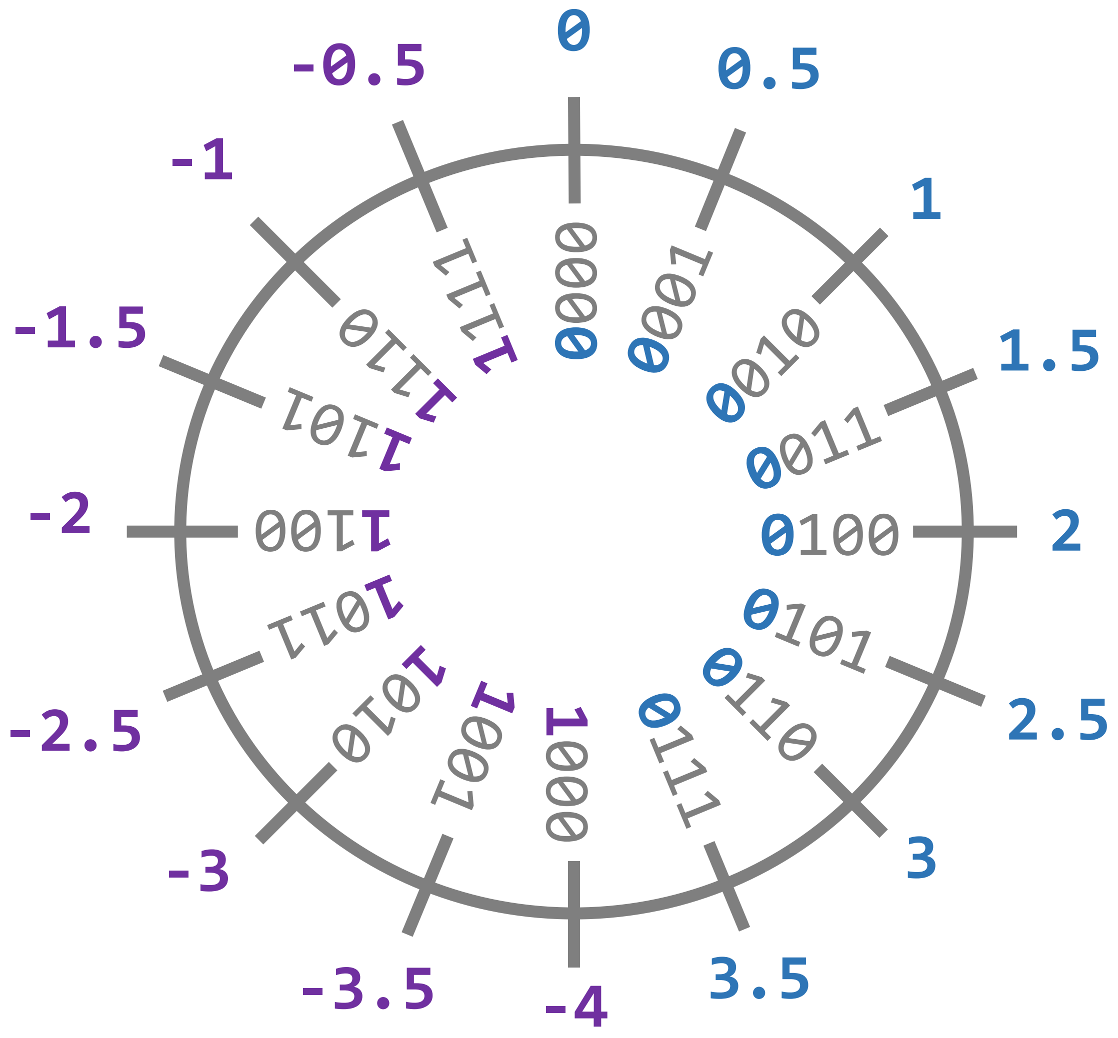


The Fixed Point Number Wheel

- Consider two’s complement fixed point of the form $b_3b_2b_1.b_0$

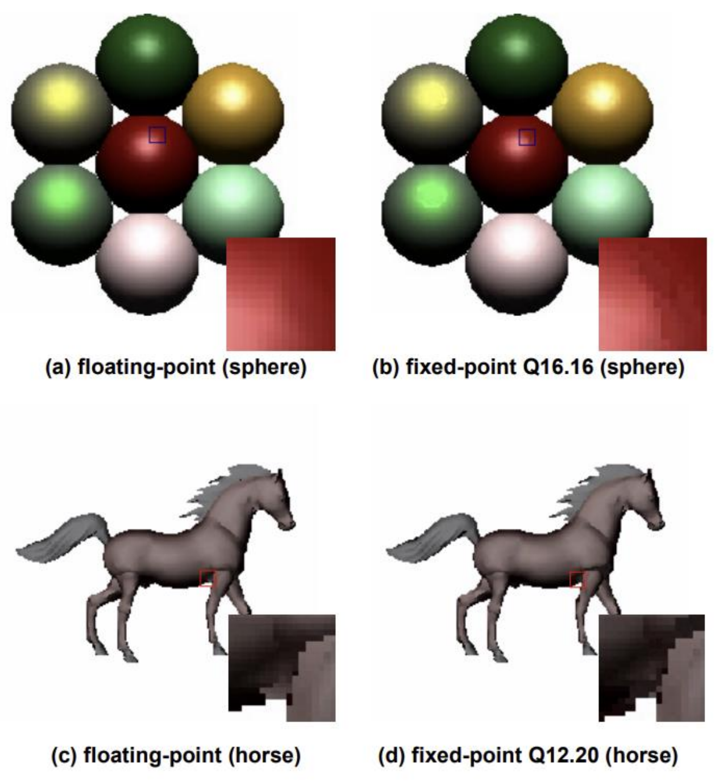


Positional code weights



Fixed Point Uses

- Commonly-used in **signal processing**, **graphics**, and **machine learning**
  - Fast and low-power (identical to just using integers)



(a) floating-point (sphere) (b) fixed-point Q16.16 (sphere)

(c) floating-point (horse) (d) fixed-point Q12.20 (horse)

Figure 8: Fixed-point vertex shading of 3D objects (spheres: 5068 vertices, horse: 6798 vertices)

Ju-Ho Sohn+, "A Programmable Vertex Shader with Fixed-Point SIMD Datapath for Low Power Wireless Applications," HWWS, 2004.

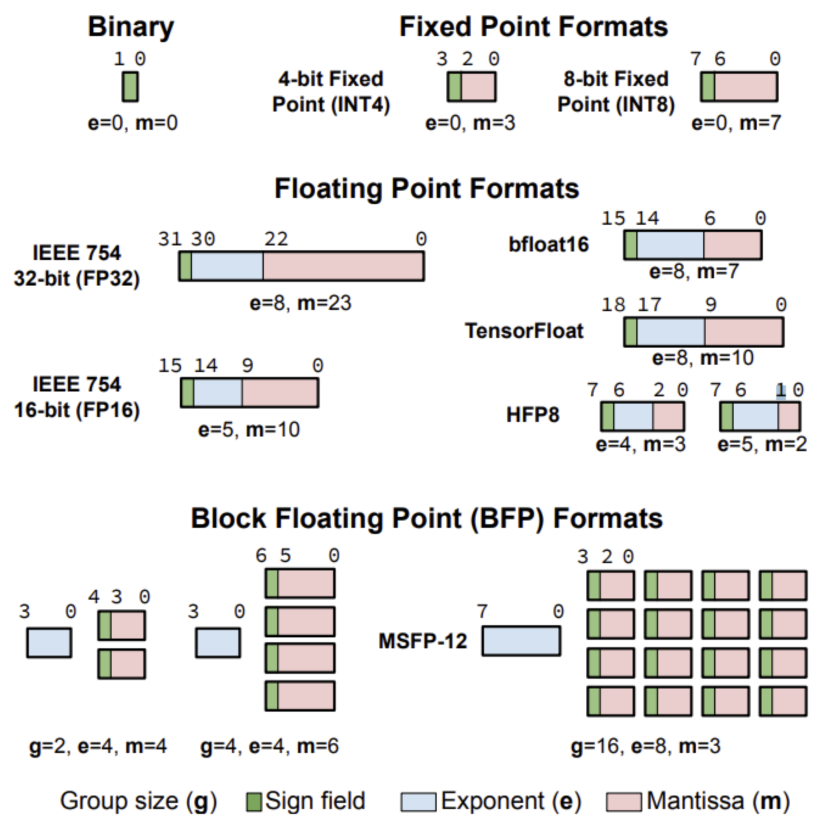


Fig. 2. Number formats commonly used for DNN training and inference.

Zhang+, "FAST: DNN Training under Variable Precision Block Floating Point with Stochastic Rounding," HPCA, 2022.

Problem: Limited Range of Representation

	Value	32-Bit Fixed Point Representation b[31:16].b[15:0]
Not Representable!	$(2face.0)_{16}$ →	0xface_0000
	$(2fac.e)_{16}$ →	0x2fac_e000
	$(2fa.ce)_{16}$ →	0x02fa_ce00
	$(2f.ace)_{16}$ →	0x002f_ace0
	$(2.face)_{16}$ →	0x0002_face
Not Representable!	$(0.2face)_{16}$ →	0x0000_2fac

Problem: Multiplication and Division

- **Add/subtract are the same:** just like with unsigned/two’s complement
  - Hardware just treats the representations as integers
  - Programmer must keep track of the “point”’s position
- Multiplication/division require **moving the “point”**

Addition

$$\begin{array}{r} 1.1 \\ +\ 0.1 \\ \hline \end{array}$$

Multiplication

$$\begin{array}{r} 1.1 \\ \times\ 0.1 \\ \hline \end{array}$$

Agenda

- Fixed Point
- **Floating Point**
  - IEEE Standard 754
- Special Floating Point Numbers

Limitations of Fixed Point

- Programmer needs to micromanage the “point” location
- Range of representable values is limited

$$\left.\begin{array}{l} 6.626 \times 10^{-34} \\ 6.022 \times 10^{23} \end{array}\right\} \text{Difference is} \sim 10^{57}\ (2^{189.35})$$

Representing both with **the same fixed point representation** would take over 57 decimal digits (**190 bits**)

Toward a General-Purpose Representation

- We want **one** number representation for **all real numbers**

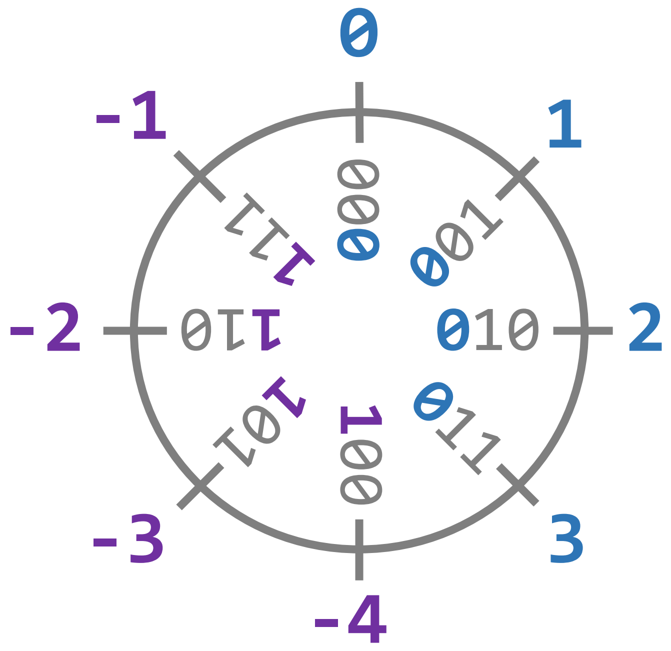
Regular integers	{0, -30, 62}
Tiny numbers	6.626 x 10^{-34}
Huge numbers	6.022 x 10^{23}
Irrational Numbers	π
Extreme numbers	-infinity
Invalid numbers	<uninitialized>

- **Constraint**: we have N bits to work with (only 2^N unique numbers)

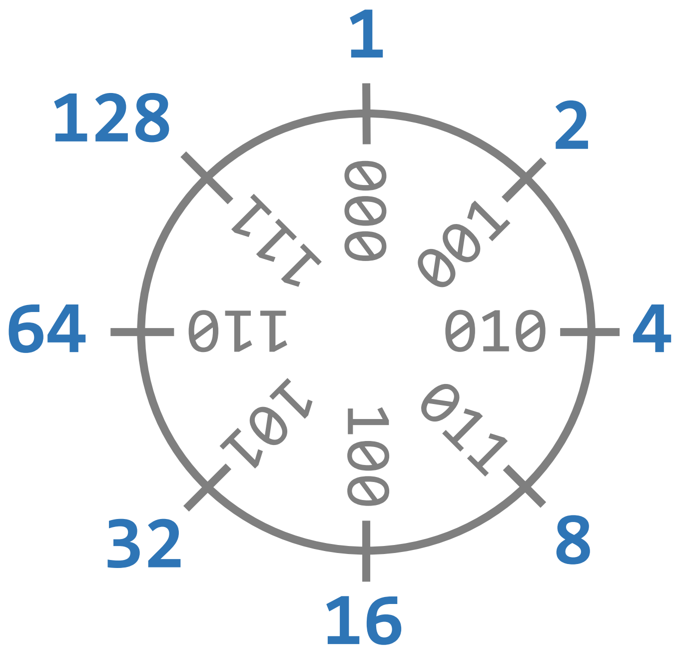
Limits of Number Representation

- N bits represent only 2^N values
  - So far, we've stuck to **linearly-spaced values**
  - But nobody said these need to be linear (or contiguous, monotonic, etc.)

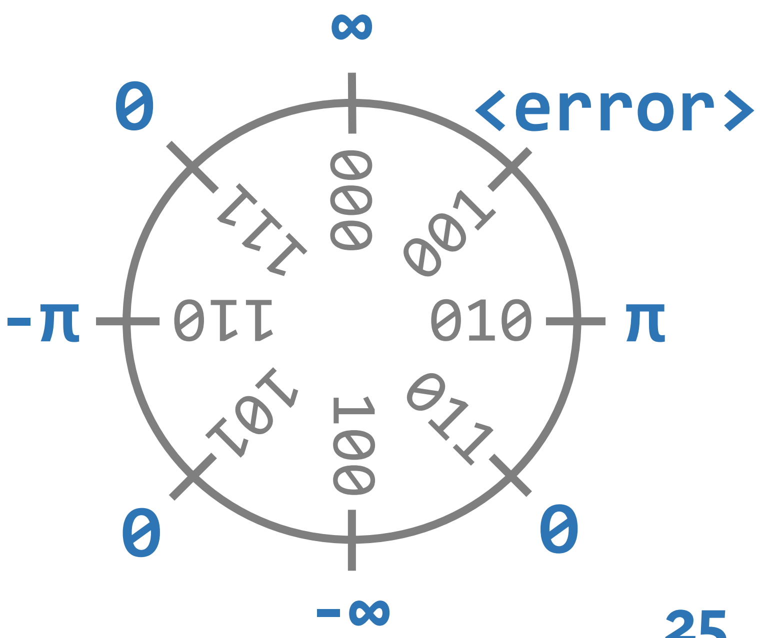
3-Bit Two's Complement (linear spacing)



A Valid 3-Bit Representation (exponential spacing)



Another Valid Representation (??? probably useless)



Floating Point

- **Key idea:** Use part of the representation to store **the point's position**
  - Let the hardware handle all the point micromanagement
- Turns out, this is just the **scientific notation** we already know and love

$$\underbrace{-}_{\textbf{sign}}\underbrace{(6.022)}_{\textbf{significand}}{}_{10} \times 10^{(\underbrace{23}_{\substack{\textbf{exponent}\\ \text{position of the point}}})_{10}}$$

⇩ *some floating-point representation*

Sign	Exponent	Significand
-	$(23)_{10}$	$(6.022)_{10}$

Different Floating Point Standards

- Many different floating point standards exist

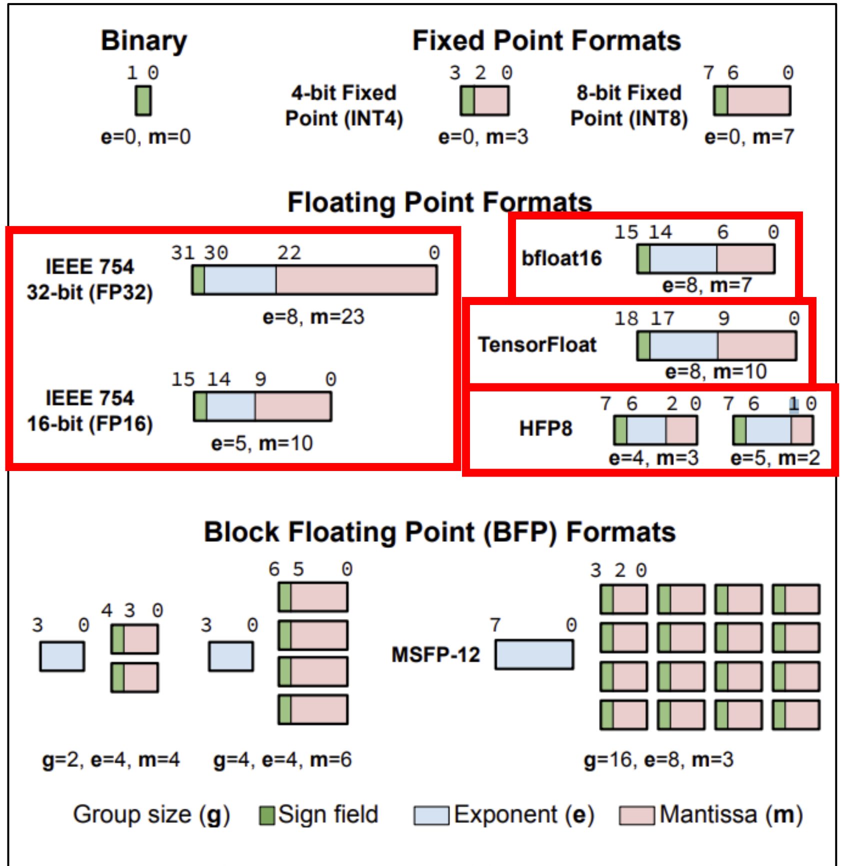


Fig. 2. Number formats commonly used for DNN training and inference.

IEEE

Google

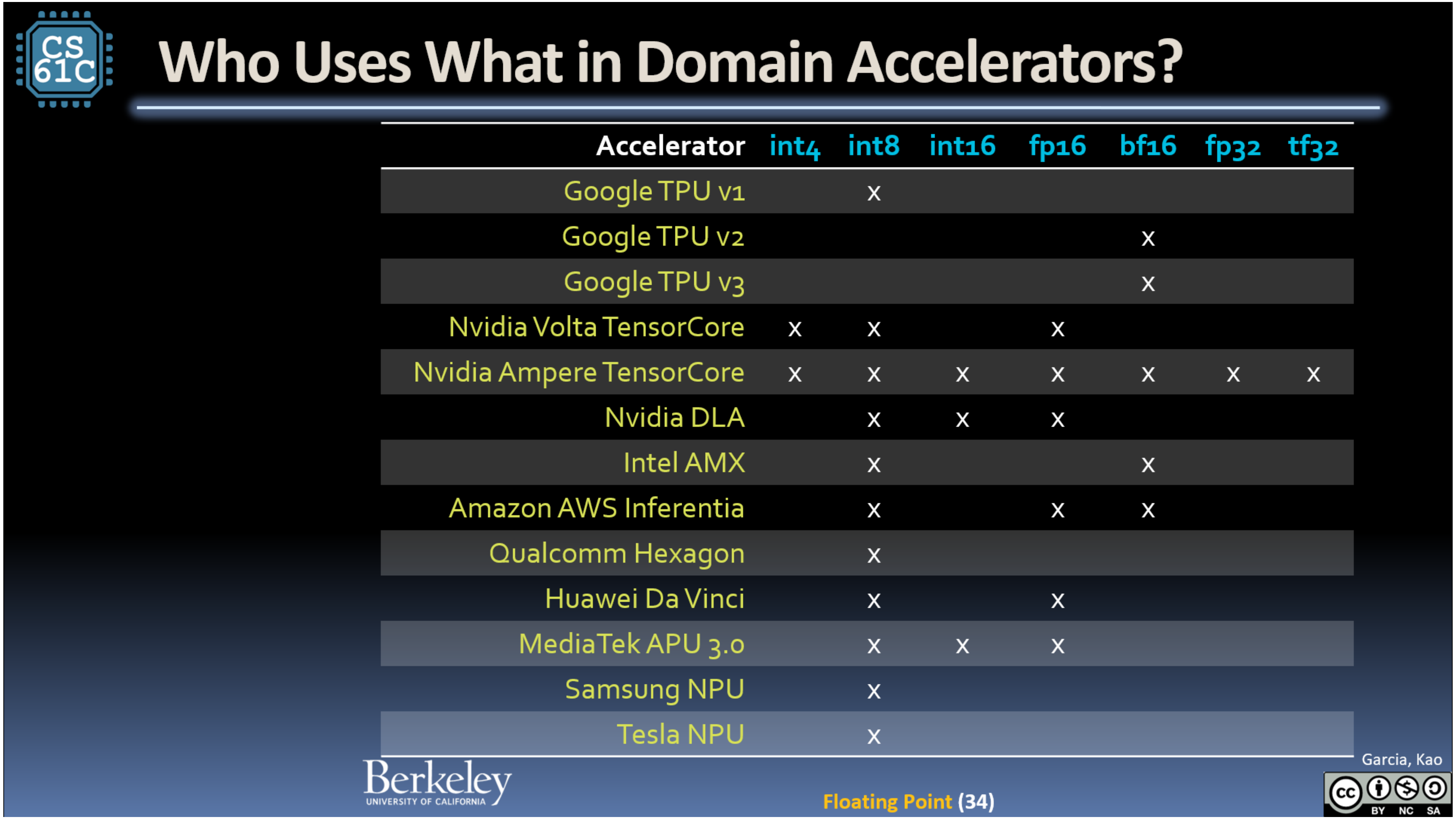
NVIDIA

NeurIPS'19

...and many others across industry and academia

Zhang+, "FAST: DNN Training under Variable Precision Block Floating Point with Stochastic Rounding," HPCA, 2022.

Number Representations in the Wild



Who Uses What in Domain Accelerators?

Accelerator	int4	int8	int16	fp16	bf16	fp32	tf32
Google TPU v1		x					
Google TPU v2					x		
Google TPU v3					x		
Nvidia Volta TensorCore	x	x		x			
Nvidia Ampere TensorCore	x	x	x	x	x	x	x
Nvidia DLA		x	x	x			
Intel AMX		x			x		
Amazon AWS Inferentia		x		x	x		
Qualcomm Hexagon		x					
Huawei Da Vinci		x		x			
MediaTek APU 3.0		x	x	x			
Samsung NPU		x					
Tesla NPU		x					

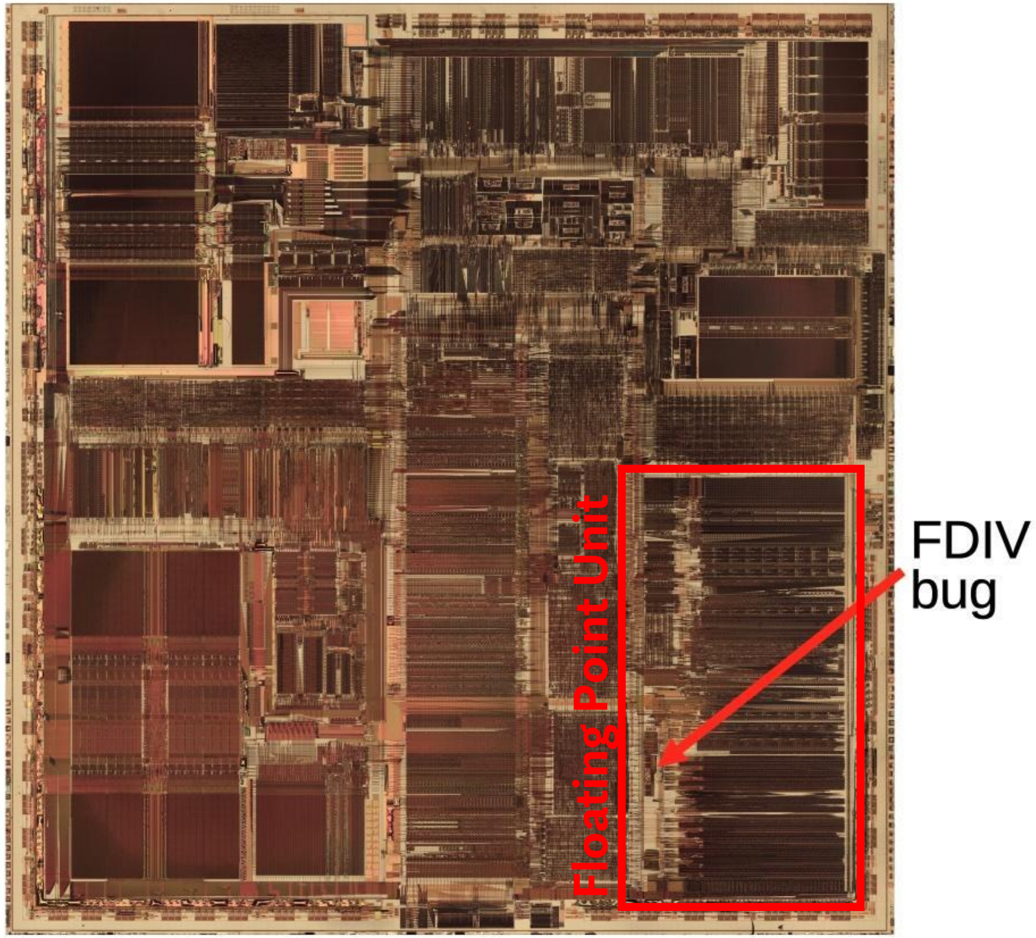
Garcia, Kao

Floating Point (34)

Floating Point Hardware

- Requires very complex hardware to operate correctly

Intel's $475 million error: the silicon behind the Pentium division bug



December 22, 1994

To owners of Pentium® processor-based computers and the PC community.

We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw.

The Intel Inside® symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect.

What Intel continues to believe is that an extremely minor technical problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns.

We want to resolve these concerns.

Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer. Just call +44 1793 696776, between 9am-7pm (Central European Time), on normal working days.

Sincerely,

Andy Grove
Andrew S. Grove
President and
Chief Executive Officer

Craig R. Barrett
Executive Vice President and
Chief Operating Officer

Gordon E. Moore
Chairman of the Board

intel.

https://www.righto.com/2024/12/this-die-photo-of-pentium-shows.html

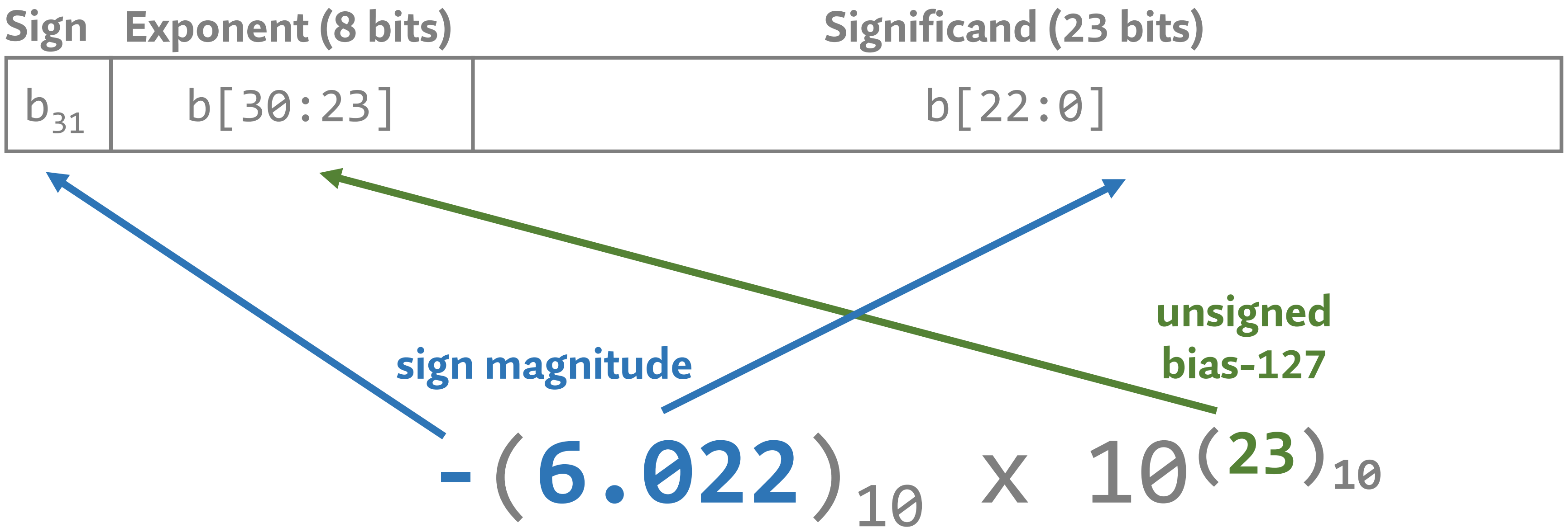
Agenda

- Fixed Point
- Floating Point
  - **IEEE Standard 754**
- Special Floating Point Numbers

IEEE Standard 754 (est. 1985)

- An industry-standardized **implementation of floating point**
  - Used in almost every processor that supports floating point

“Single Precision” Floating Point (32 Bits)



Sign	Exponent (8 bits)	Significand (23 bits)
b_{31}	b[30:23]	b[22:0]

sign magnitude

unsigned
bias-127

$$-(6.022)_{10} \times 10^{(23)_{10}}$$

Significand: Normalized Form

- A number has infinite valid forms in scientific notation

$$-(0.0111)_2 \times 2^{2}$$

$$-(0.111)_2 \times 2^{1}$$

$$\boxed{-(1.11)_2 \times 2^{0}}$$ “Normalized form”

$$-(11.10)_2 \times 2^{-1}$$

$$-(111.0)_2 \times 2^{-2}$$

- **Normalized form**: one digit to the left with no leading zeroes
  - Only one unique normalized form for every number ☺

Significand: Normalized Form

Normalized Number	Significand Representation (23 bits)
$-(1.11)_2 \times 2^{0}$	`110_0000_0000_0000_0000_0000`
$(1.000001)_2 \times 2^{14}$	`000_0010_0000_0000_0000_0000`
$(1.1010101010101)_2 \times 2^{-124215325}$	`101_0101_0101_0100_0000_0000`

Every normalized number starts with ‘`1.`’

- We can **imply** the leading ‘`1.`’ without wasting a bit to store it

Exponent: Bias-127

- 8 exponent bits yield 256 unique representations
  - **`0x00`** and **`0xff`** reserved for “special numbers” (e.g., infinity, zero, NaN)
  - **`0x01-0xfe`** are valid exponents (bias-127)

Normalized Form	**Exponent Value**	**Biased Value**	**Exponent Representation (8 bits)**
$(1.0)_2$ x 2^{0}	0	0 + 127	0111_1111
$(1.0)_2$ x 2^{127}	127	127 + 127	1111_1110
$(1.0)_2$ x 2^{-126}	-126	-126 + 127	0000_0001

Floating Point Example

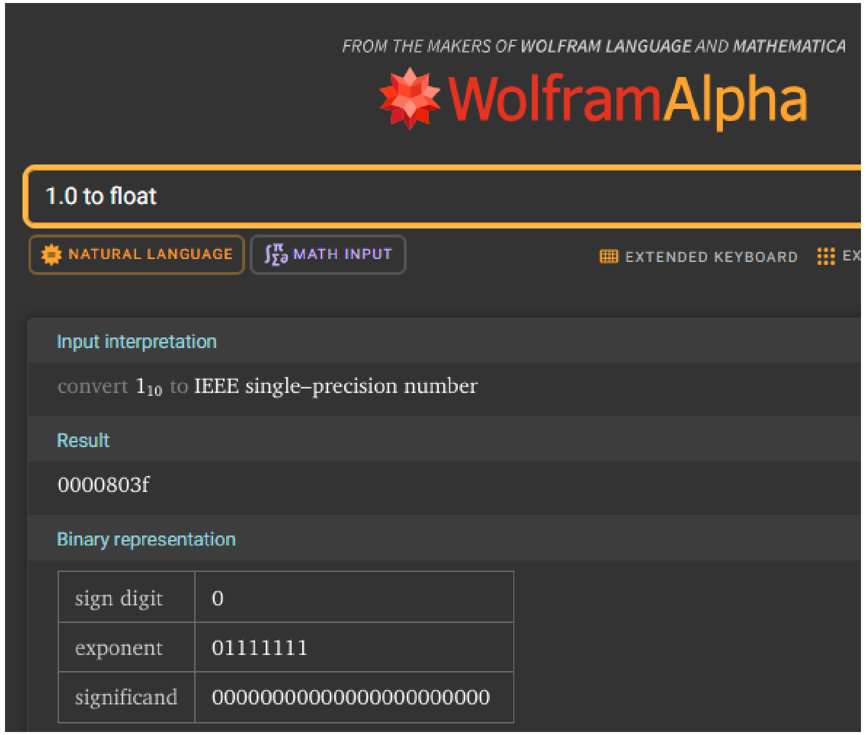
$(1.0)_{10}$

Floating Point Example

$$(1.0)_{10} = (1.0)_2 = (1.0)_2 \times 2^0$$

Sign	Exponent (8 bits)	Significand (23 bits)
0	0111_1111	000_0000_0000_0000_0000_0000

biased value: 127



1.0 to float

Input interpretation

convert 1_{10} to IEEE single–precision number

Result

0000803f

Binary representation

sign digit	0
exponent	01111111
significand	00000000000000000000000

Floating Point Example

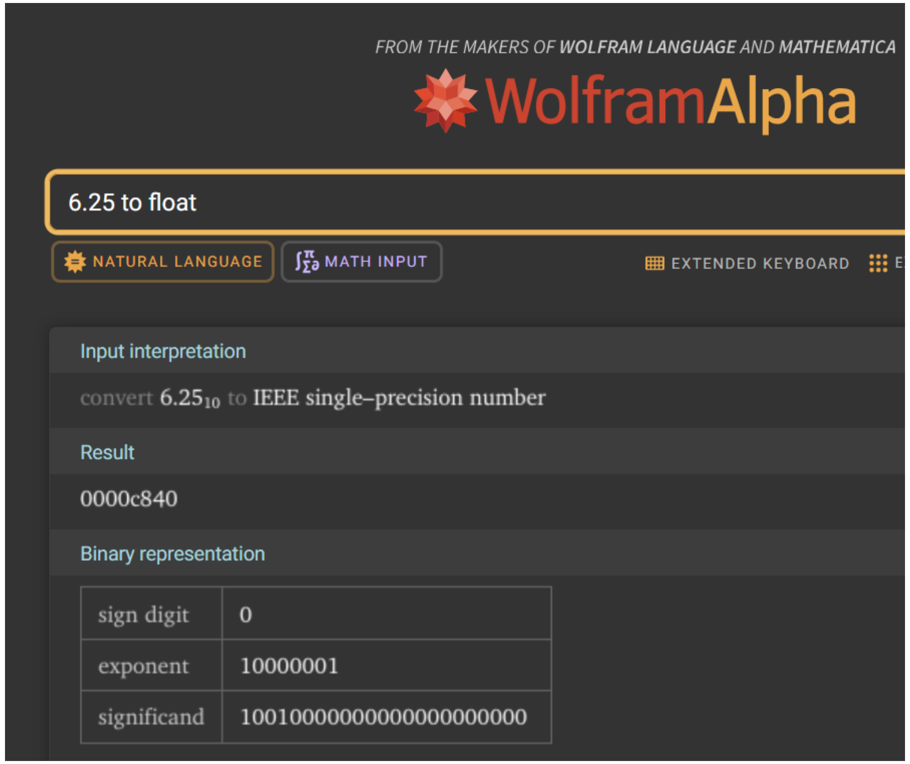
$$(6.25)_{10}$$

Sign	Exponent (8 bits)	Significand (23 bits)

Floating Point Example

$$(6.25)_{10} = (110.01)_2 = (1.1001)_2 \times 2^2$$

Sign	Exponent (8 bits)	Significand (23 bits)
0	1000_0001	100_1000_0000_0000_0000_0000
	biased value: 129	



6.25 to float

Input interpretation

convert 6.25_{10} to IEEE single–precision number

Result

0000c840

Binary representation

sign digit	0
exponent	10000001
significand	10010000000000000000000

Floating Point Example

Sign	Exponent (8 bits)	Significand (23 bits)
`1`	`0001_1000`	`011_0000_0000_0000_0000_0000`

Floating Point Example

Sign	Exponent (8 bits)	Significand (23 bits)
1	0001_1000	011_0000_0000_0000_0000_0000

Unbiased exponent = $(11000)_2 - (127)_{10} = (-103)_{10}$

Significand = $(.011)_2$

Sign = -

$$-(1.011)_2 \times 2^{-103} = (1.375)_{10} \times 2^{-103}$$

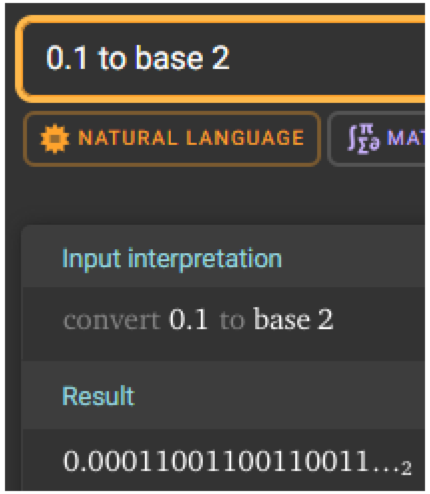
More Examples

$$(0.1)_{10} = (1.\overline{1001})_2 \times 2^{-4}$$

biased exponent: -4 + 127

$$(0.2)_{10} = (1.\overline{1001})_2 \times 2^{-3}$$

biased exponent: -3 + 127



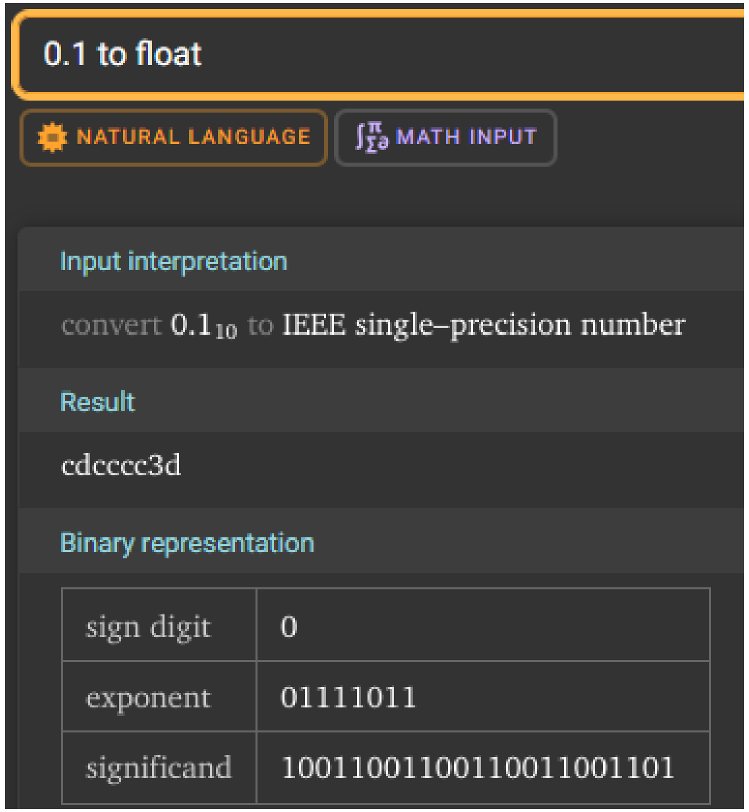
0.1 to base 2

Input interpretation

convert 0.1 to base 2

Result

$0.00011001100110011\ldots_2$



0.1 to float

Input interpretation

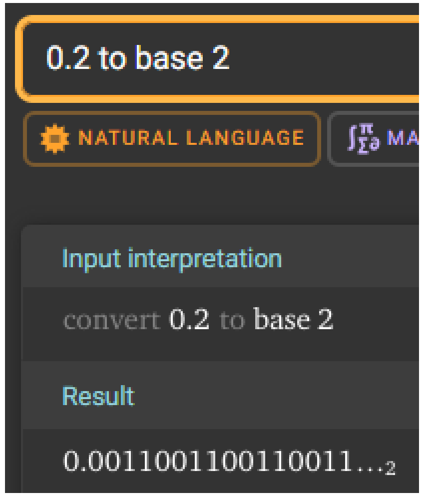
convert 0.1_{10} to IEEE single–precision number

Result

cdcccc3d

Binary representation

sign digit	0
exponent	01111011
significand	10011001100110011001101



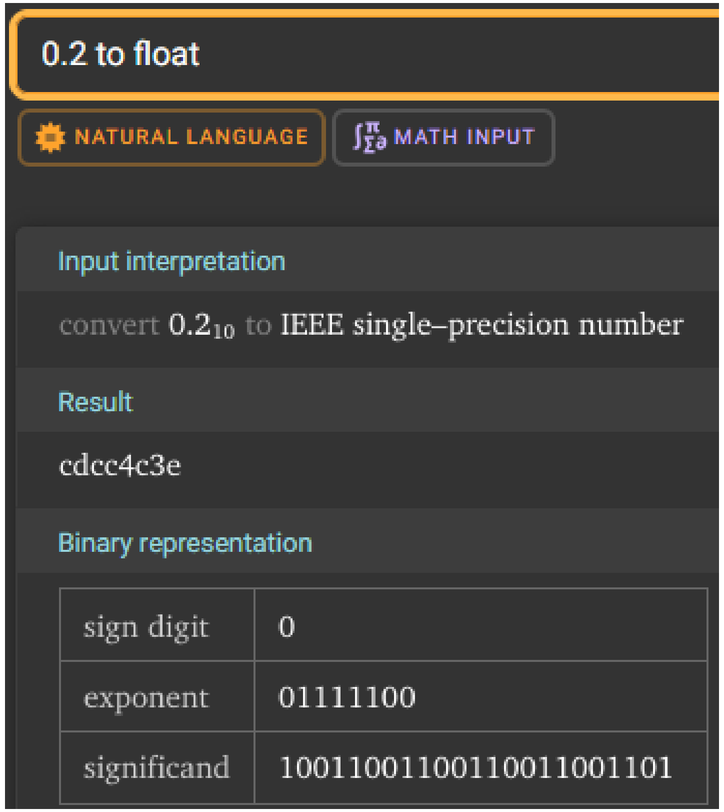
0.2 to base 2

Input interpretation

convert 0.2 to base 2

Result

$0.0011001100110011\ldots_2$



0.2 to float

Input interpretation

convert 0.2_{10} to IEEE single–precision number

Result

cdcc4c3e

Binary representation

sign digit	0
exponent	01111100
significand	10011001100110011001101

Floating Point Range (Normalized)

- Sign bit makes floats symmetric across positive/negative numbers
- Makes sense to talk about **smallest/largest** representable values instead

Smallest Normalized Value

$$(1.000\_0000\_0000\_0000\_0000\_0000)_2 \text{ x } 2^{-126} \approx (1.18)_{10} \text{ x } 2^{-38}$$

Sign	Exponent (8 bits)	Significand (23 bits)
b_{31}	0000_0001	000_0000_0000_0000_0000_0000

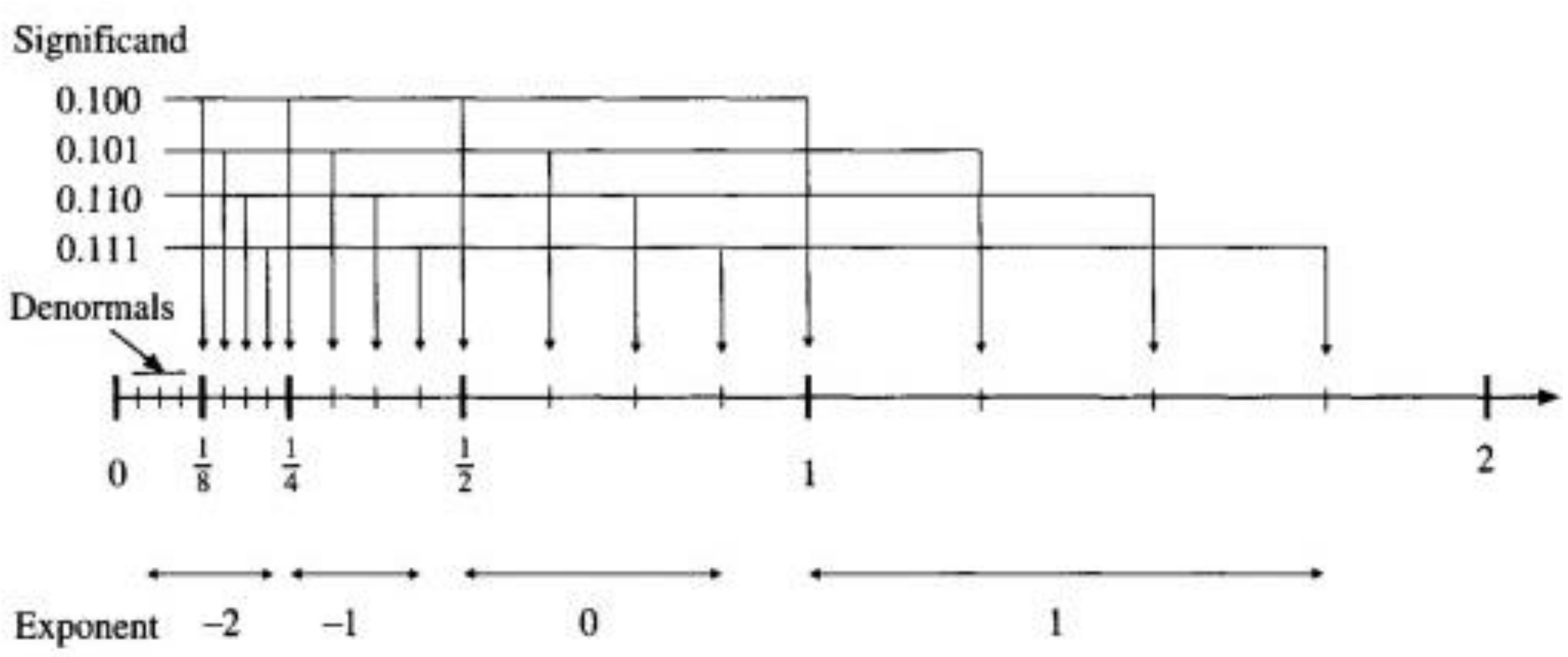
Largest Normalized Value

$$(1.111\_1111\_1111\_1111\_1111\_1111)_2 \text{ x } 2^{127} \approx (3.40)_{10} \text{ x } 2^{38}$$

Sign	Exponent (8 bits)	Significand (23 bits)
b_{31}	1111_1110	111_1111_1111_1111_1111_1111

Floating Point Value Distribution

- $2^{N=32}$ values are **nonuniformly distributed**



Milos Ercegovac, Tomas Lang, "Digital Arithmetic", Morgan Kaufman, 2004

Agenda

- Fixed Point
- Floating Point
  - IEEE Standard 754
- **Special Floating Point Numbers**

Going Even Smaller: Denormalization

- What if we allowed a leading zero for **really small numbers**?
  - Special exponent value = `0x00`
  - Called a "**denormalized value**"

Smallest Normalized Value

Sign	Exponent (8 bits)	Significand (23 bits)
b_{31}	0000_0001	000_0000_0000_0000_0000_0000

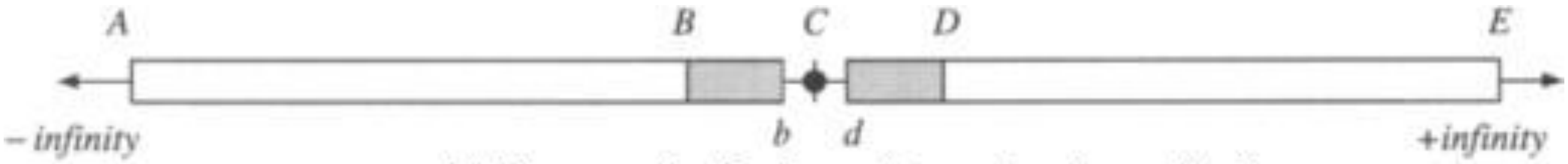
$$(1.000\_0000\_0000\_0000\_0000\_0000)_2 \text{ x } 2^{-126} \approx (1.18)_{10} \text{ x } 2^{-38}$$

Smallest Denormalized Value

Sign	Exponent (8 bits)	Significand (23 bits)
b_{31}	0000_0000	000_0000_0000_0000_0000_0001

$$(0.000\_0000\_0000\_0000\_0000\_0001)_2 \text{ x } 2^{-126} \approx (1.4)_{10} \text{ x } 2^{-45}$$

Floating Point Range (Normal + Denormal)



$[A, B]$ — negative floating-point numbers (normalized)

$[D, E]$ — positive floating-point numbers (normalized)

$(B, b]$ & $[d, D)$ — denormals

C — zero

$> E$ — positive overflow

$< A$ — negative overflow

(B, C) — negative underflow (normalized)

(C, D) — positive underflow (normalized)

Milos Ercegovac, Tomas Lang, "Digital Arithmetic", Morgan Kaufman, 2004

Special Numbers in IEEE 754

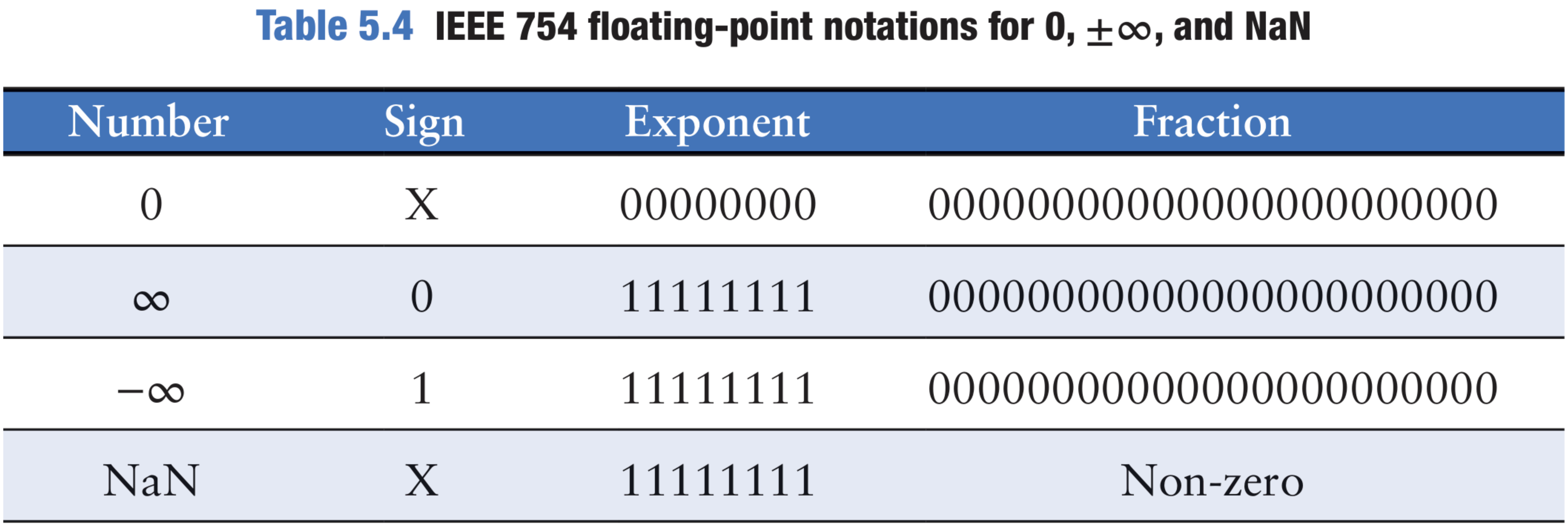


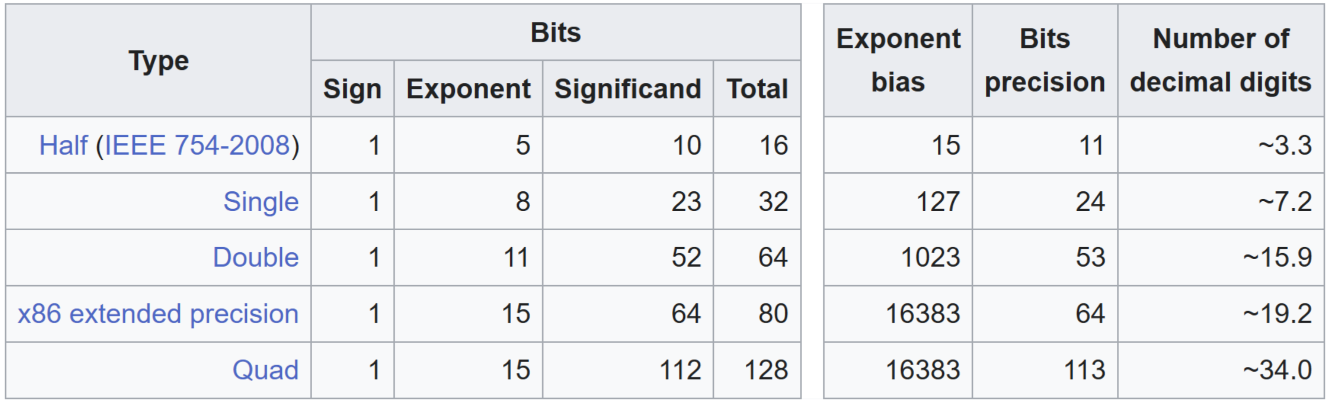
Table 5.4 **IEEE 754 floating-point notations for 0, ±∞, and NaN**

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Harris + Harris, Chapter 5.3

Other Floating Point Formats

- IEEE Standard 754 specifies other types of float representations



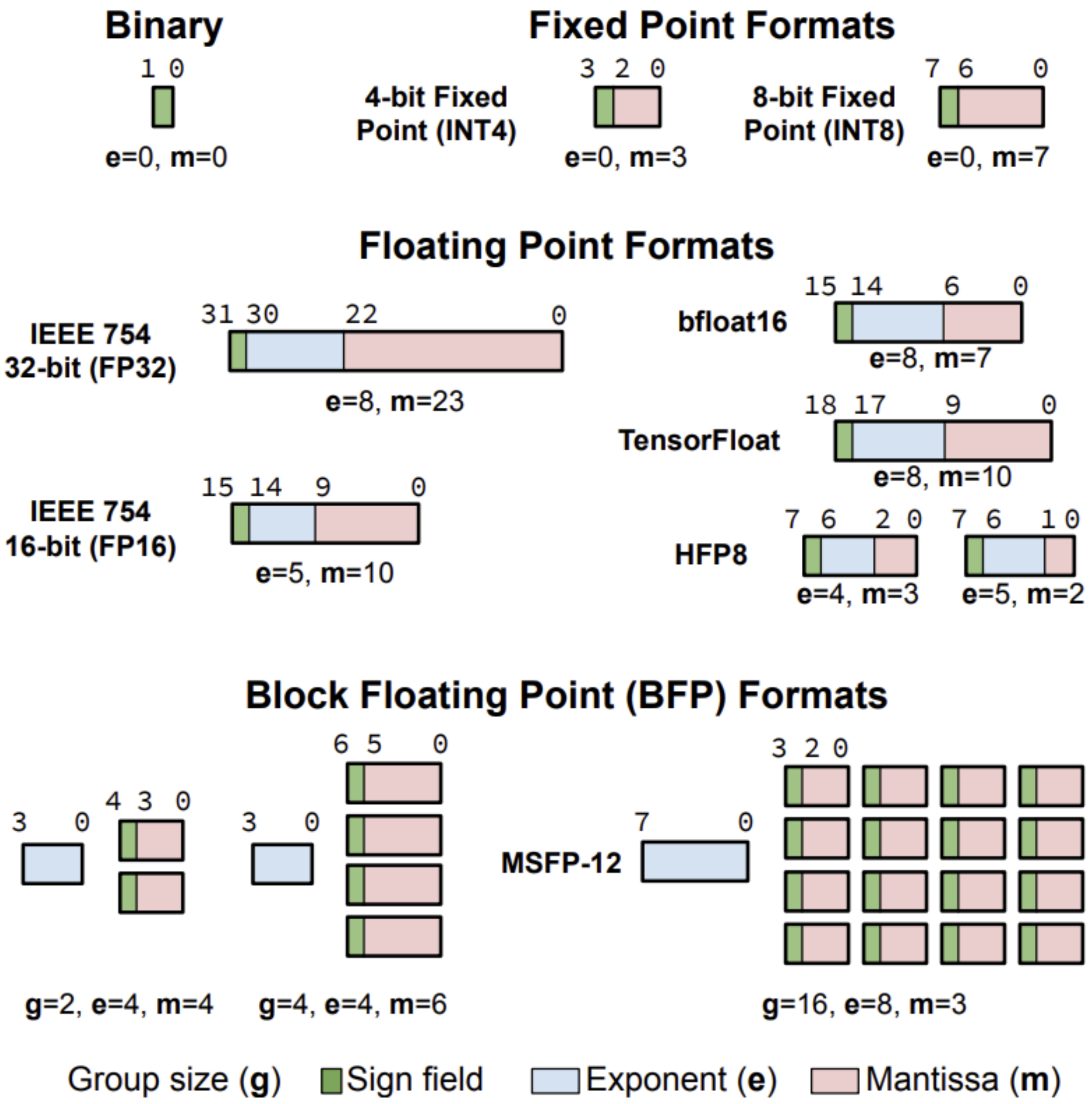
Type	Bits				Exponent bias	Bits precision	Number of decimal digits
	Sign	Exponent	Significand	Total			
Half (IEEE 754-2008)	1	5	10	16	15	11	~3.3
Single	1	8	23	32	127	24	~7.2
Double	1	11	52	64	1023	53	~15.9
x86 extended precision	1	15	64	80	16383	64	~19.2
Quad	1	15	112	128	16383	113	~34.0

- Many other formats exist out in the wild

Type	Sign	Exponent	Trailing significand field	Total bits
FP8 (E4M3)	1	4	3	8
FP8 (E5M2)	1	5	2	8
Half-precision	1	5	10	16
Bfloat16	1	8	7	16
TensorFloat-32	1	8	10	19
Single-precision	1	8	23	32

https://en.wikipedia.org/wiki/Floating-point_arithmetic

Common DNN Training Number Formats



Zhang+, "Fast: Dnn training under variable precision block floating point with stochastic rounding," HPCA, 2022.

Summary of Float and Double Representations

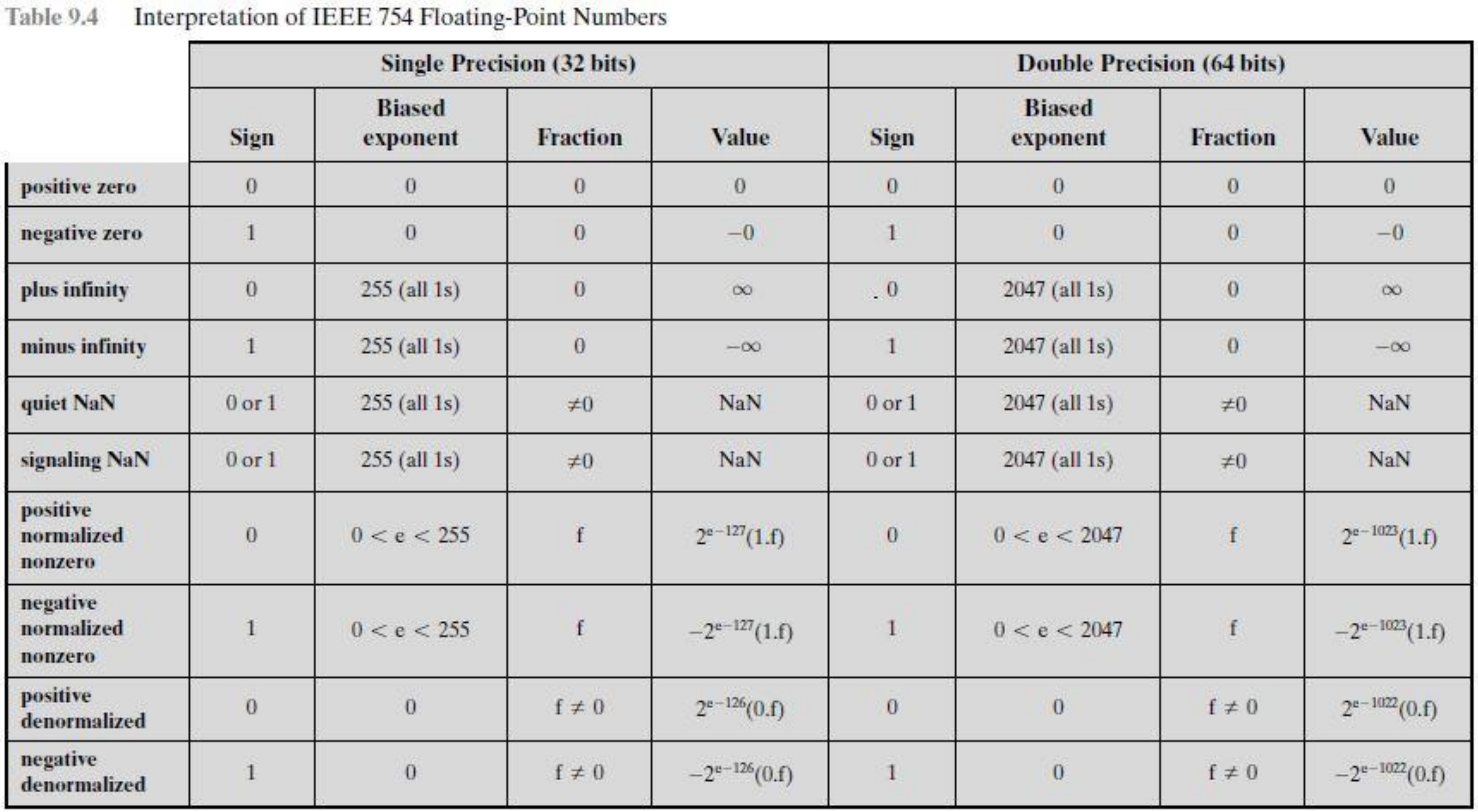


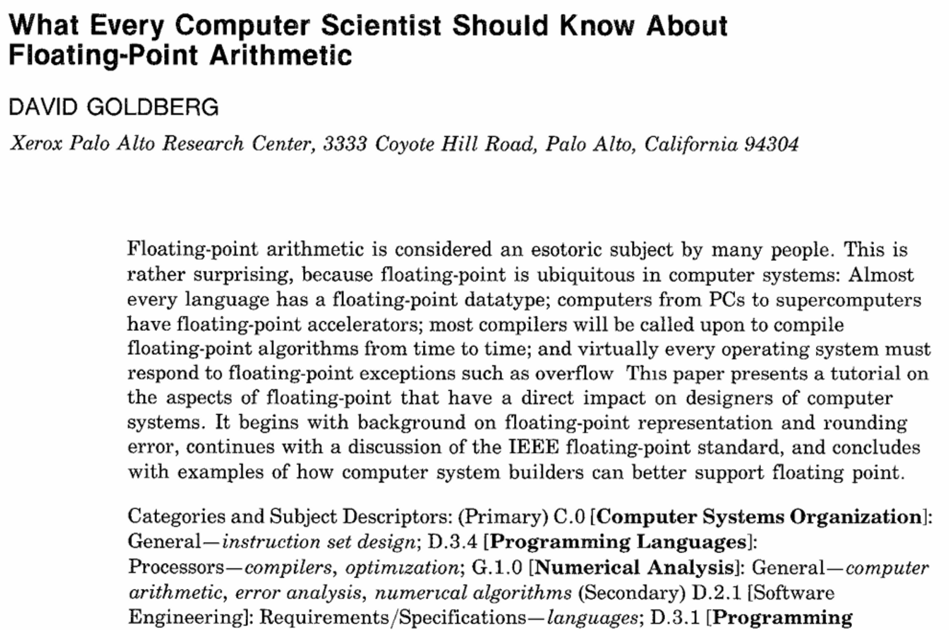
Table 9.4 Interpretation of IEEE 754 Floating-Point Numbers

	Single Precision (32 bits)				Double Precision (64 bits)			
	Sign	Biased exponent	Fraction	Value	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0	0	0	0	0
negative zero	1	0	0	−0	1	0	0	−0
plus infinity	0	255 (all 1s)	0	∞	0	2047 (all 1s)	0	∞
minus infinity	1	255 (all 1s)	0	−∞	1	2047 (all 1s)	0	−∞
quiet NaN	0 or 1	255 (all 1s)	≠0	NaN	0 or 1	2047 (all 1s)	≠0	NaN
signaling NaN	0 or 1	255 (all 1s)	≠0	NaN	0 or 1	2047 (all 1s)	≠0	NaN
positive normalized nonzero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
negative normalized nonzero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
positive denormalized	0	0	$f \neq 0$	$2^{e-126}(0.f)$	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
negative denormalized	1	0	$f \neq 0$	$-2^{e-126}(0.f)$	1	0	$f \neq 0$	$-2^{e-1022}(0.f)$

Stallings, “Computer Organization and Architecture,” Pearson Education, 2008.

More on Floats

- There’s a semester’s worth of floating point material to consider
    - Precision, machine epsilon, and ulp
    - Normalized and denormalized numbers
    - Specialized floating-point types (e.g., 8-bit, 16-bit, etc.)
    - Representation error, rounding error, error propagation and bounding



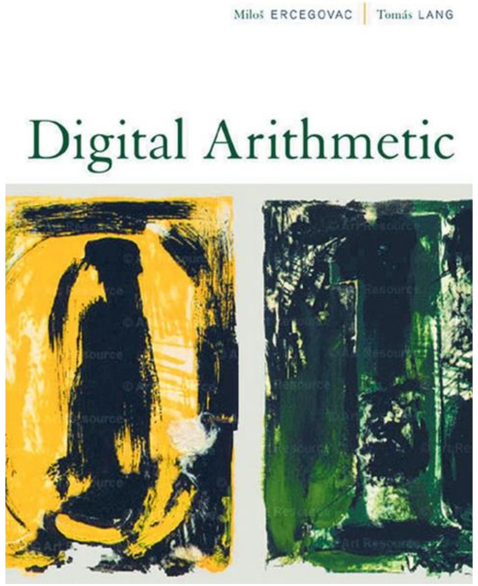
What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esotoric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [**Computer Systems Organization**]: General—*instruction set design*; D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*, *error analysis*, *numerical algorithms* (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—*languages*; D.3.1 [**Programming**



Milos Ercegovac, Tomas Lang, "Digital Arithmetic", Morgan Kaufman, 2004

CS 211: Intro to Computer Architecture

3.2: Fixed and Floating Point Representations

Minesh Patel

Spring 2025 – Thursday 6 February