信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture

Everything is a Number

Instructors:

Chundong Wang, Siting Liu & Yuan Xiao

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2025/2/20

Administrative

- HW1 is available, Due **Feb. 25th**!
- Lab1 will be available today, keep an eye on Piazza/course website. It will be checked in the lab sessions next week.
- Discussion 2 next week on Linux installation, git, gdb, etc.

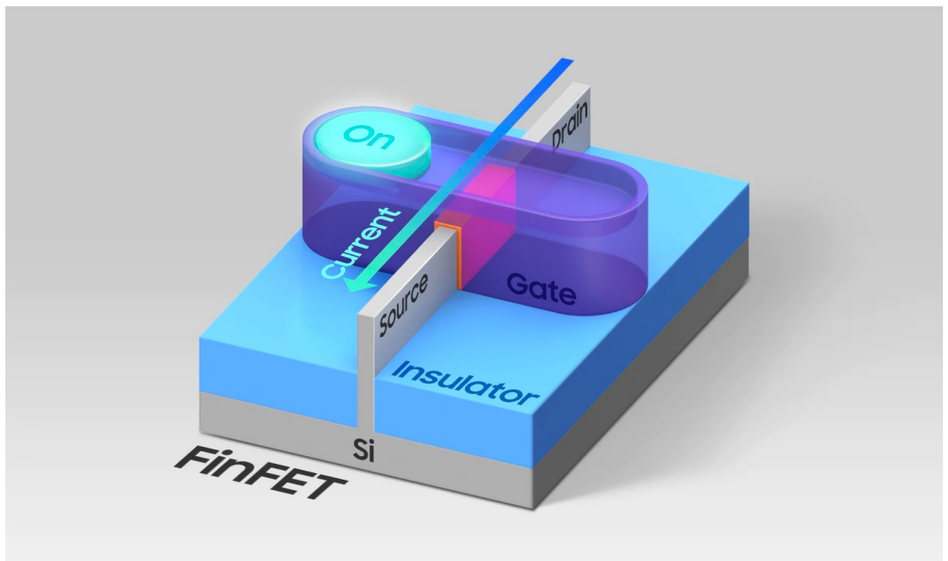
  Mon. or Fri. 19:50-21:30, teaching center 301 by TA Guanghui Hu.

Outline

- Binary system
- Everything is a number
- Signed and Unsigned integers
  - 2’s-complement representation
- Floating-point numbers

Binary System

- 0 and 1 (binary digit or bit, unit of information entropy)
- Decided by the characteristic of semiconductor devices (bi-stable states)
- Resilient to noise (threshold)
- Supported by Boolean algebra theory (George Boole, 1854)
- Basic operations: ^, |, ~



Binary System

- Represent a number in binary system
- Analogy to decimal (to represent values)
- Positionally weighted coding
- Binary-decimal conversion
- Extend to Hexadecimal (Base 16)/Octal (Base 8)

Binary Arithmetic

```
  0011
+ 0010
------
```

```
  1100
- 1010
------
```

```
  1101
X 1110
------
```

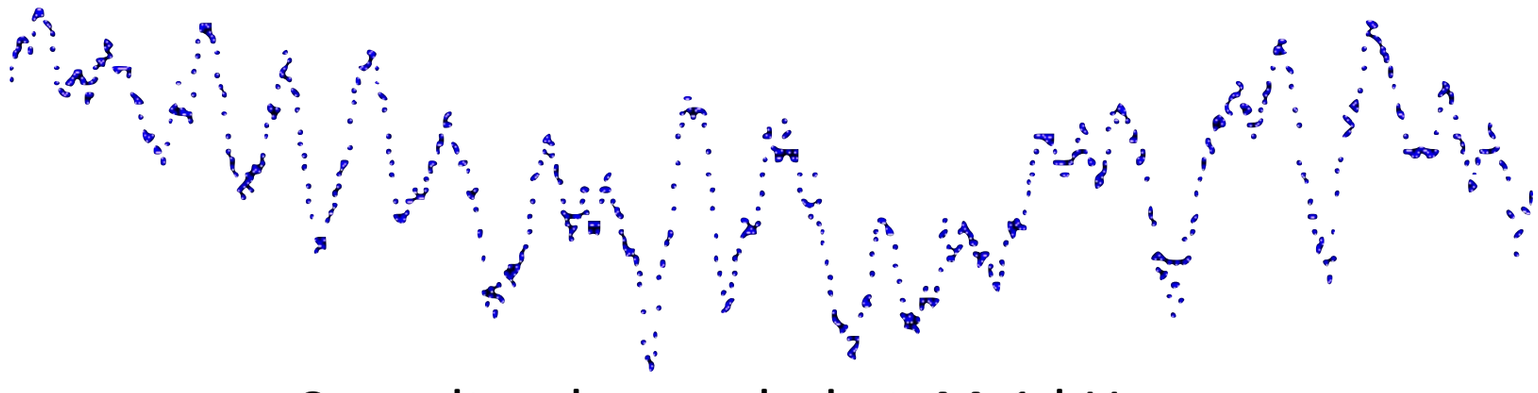
```
110 ) 10101111
```

Can be implemented by logic operations

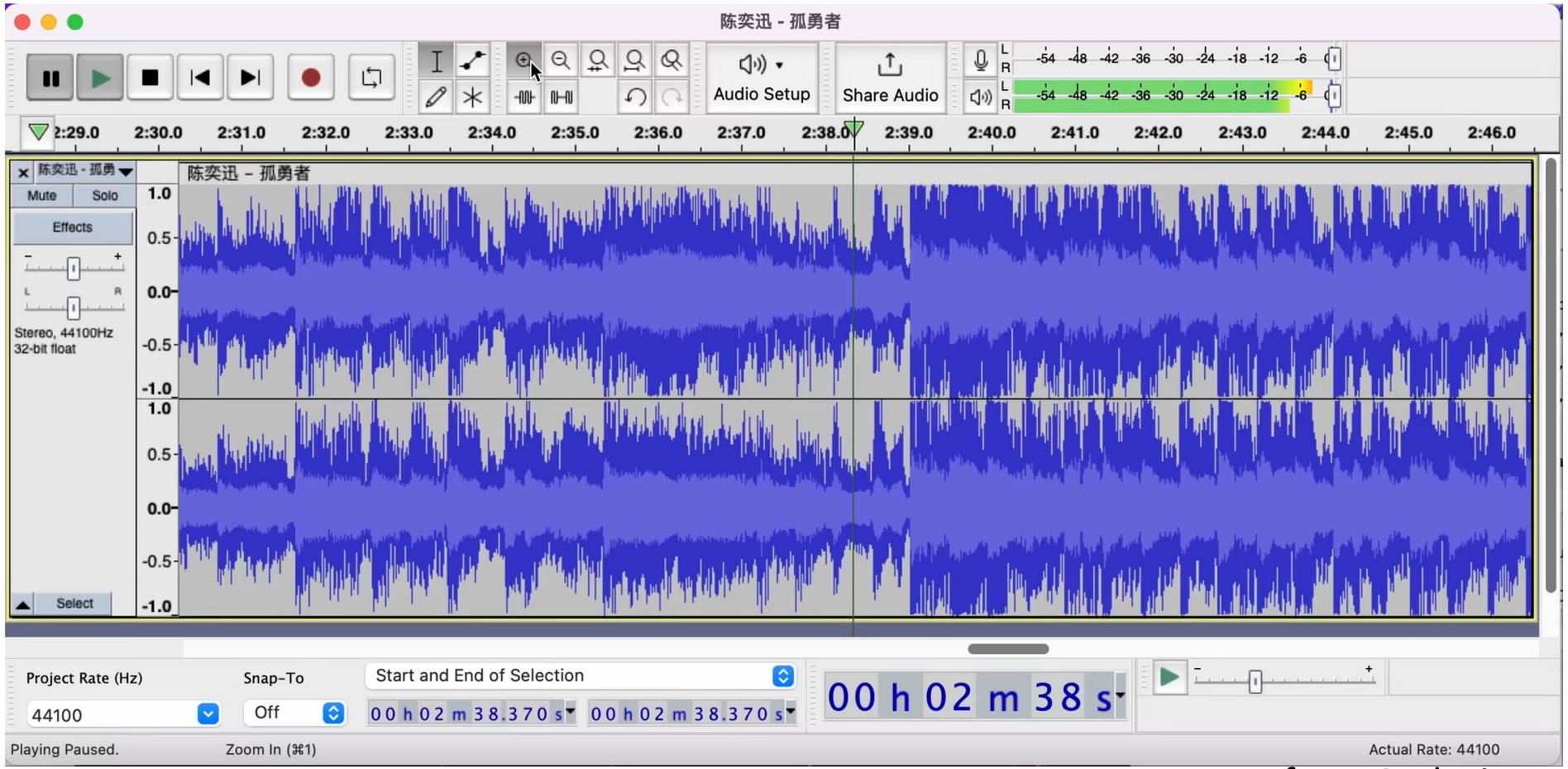
Everything is a Number

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 4-bit nibbles (rarely used), 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …

Everything is a Number



Soundtrack sampled at 44.1 kHz

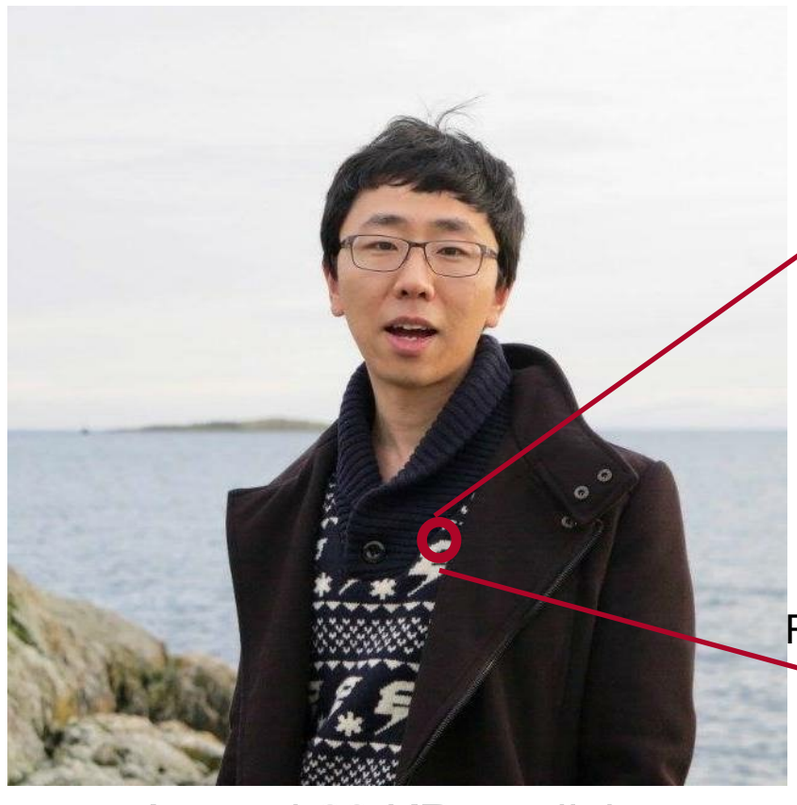


from Audacity

Everything is a Number

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 4-bit nibbles (rarely used), 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …

A 640*640 pixel color image



Around 60 KB on disk

https://www.graphicsmill.com/blog/2014/11/06/Compression-ratio-for-different-JPEG-quality-values

A 20*20 slice of the color image

HEX: #292023
RGB(41, 32, 35)

HEX: #c6c3ba
RGB(198, 195, 186)

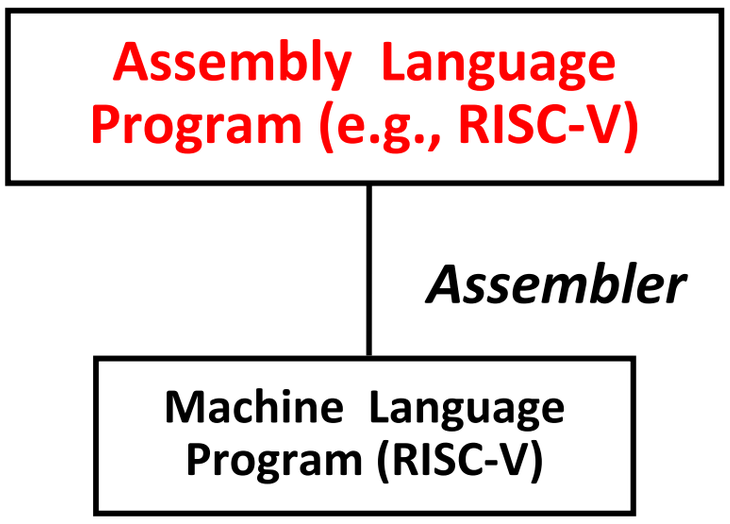
HEX: #d5d2c9
RGB(213, 210, 201)

Everything is a Number

- Inside computers, everything is a number
- BUT NOT NECESSARY THE BINARY VALUE
- Identity, bank account, profile, …
  - ID number, DoB (date of birth), criminal record, mobile, etc.
  - Bank account numbers, balance, loan, transaction records, etc.
  - Game account, coins, equipments, …

Everything is a Number

- Inside computers, everything is a number (but not necessary the value)
- Instructions: e.g., move directions: forward, backward, left, right; use $(00)_2$, $(01)_2$, $(10)_2$ and $(11)_2$




```
lw      t0, 0(s2)
lw      t1, 4(s2)
swt1, 0(s2)
swt0, 4(s2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Stored program

- It is how you interpret the numbers decides the meaning

Anything can be represented as a *number*, i.e., data or instructions
ISA, defined by human, decides the meaning

Everything is a Number

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 4-bit nibbles (rarely used), 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating-point operations can lead to results too big/small to store within their representations: overflow
- To avoid overflow, use more bits or extend the range by interpreting a number differently

Signed and Unsigned Integers

- Commonly used in computers to represent integers
- C, C++ have signed integers, e.g., 7, -255:
  - `int x, y, z;`
  - `x = 7;`
- C, C++ also have unsigned integers, e.g. for addresses
- Unsigned integers use their values to represent numbers directly
- Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4,294,967,295) (4 Gibi)

Unsigned Integers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
… …
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 2{,}147{,}483{,}649_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2{,}147{,}483{,}650_{ten}$
… …
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

$$(a_n a_{n-1} \dots a_1 a_0)_2 = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

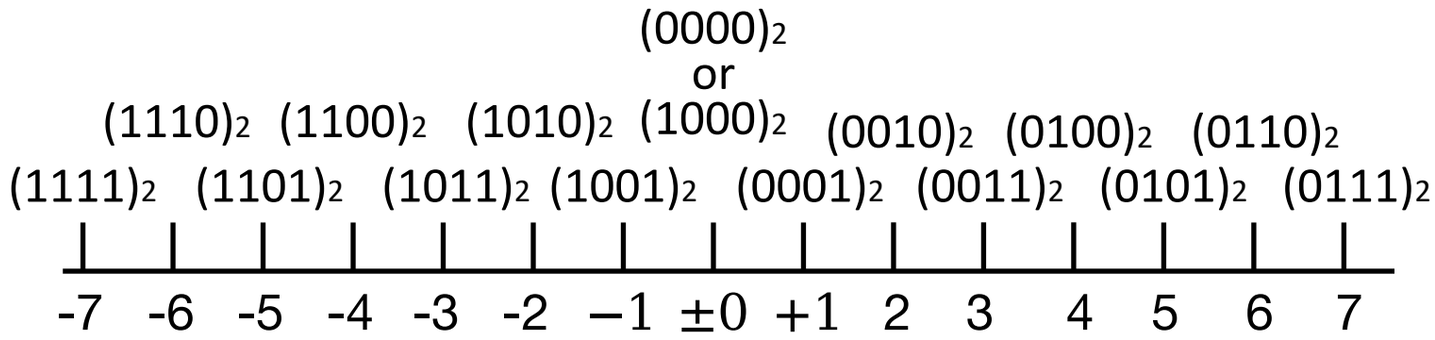
Signed Integers

- A straight-forward method: add a sign bit (sign-magnitude)
- Most-significant bit (MSB, leftmost) is the sign bit, 0 means positive, 1 means negative; the other bits remain unchanged

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two} = 3_{ten}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two} = -3_{ten}$

Sign bit



- Range:
  - Positive: $0 \sim 2^{(n-1)}-1$
  - Negative: $-0 \sim -(2^{(n-1)}-1)$
  - Arithmetically unfriendly

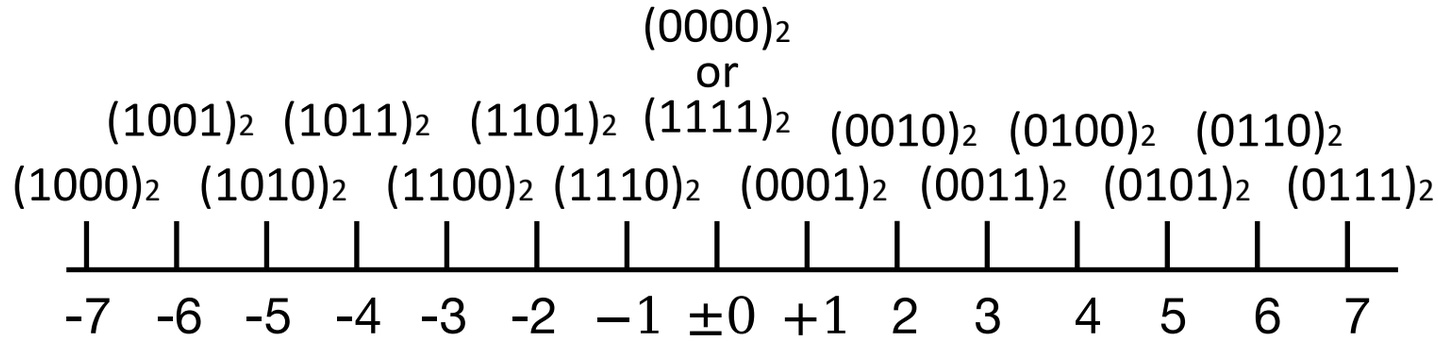
Signed Integers
One's- & Two's-Complement Representation

- One's-complement representation
  - Positive numbers, stay unchanged; Negative numbers, toggle all bits

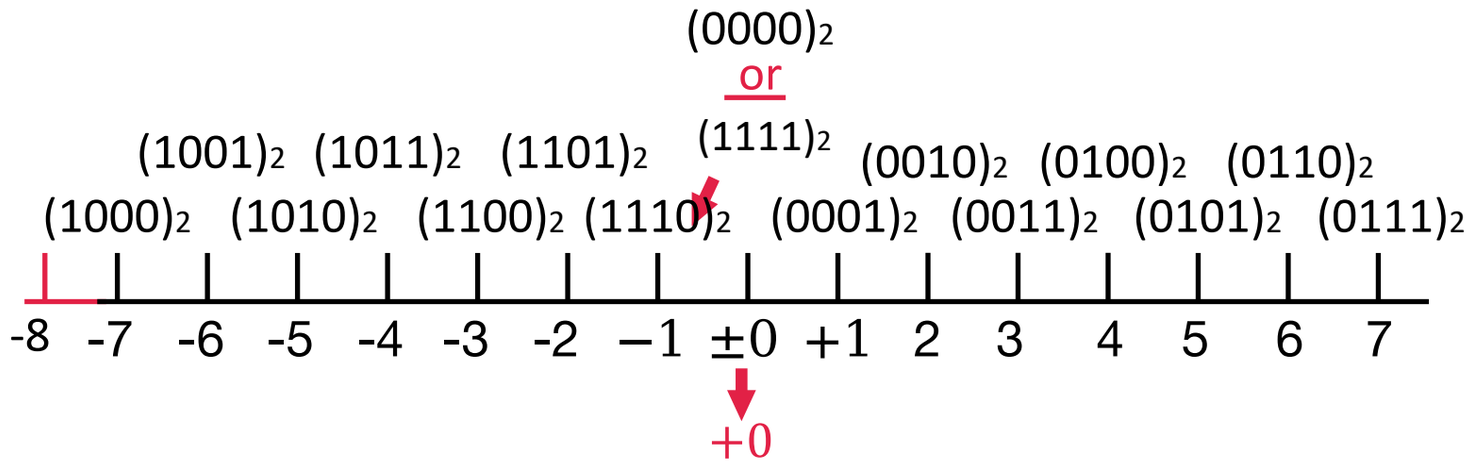
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two} = 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two} = -3_{ten}$

Sign bit

- Range:
  - Positive: $0 \sim 2^{(n-1)}-1$
  - Negative: $-0 \sim -(2^{(n-1)}-1)$
  - Arithmetically unfriendly



Two's-Complement Representation (Signed Integer)

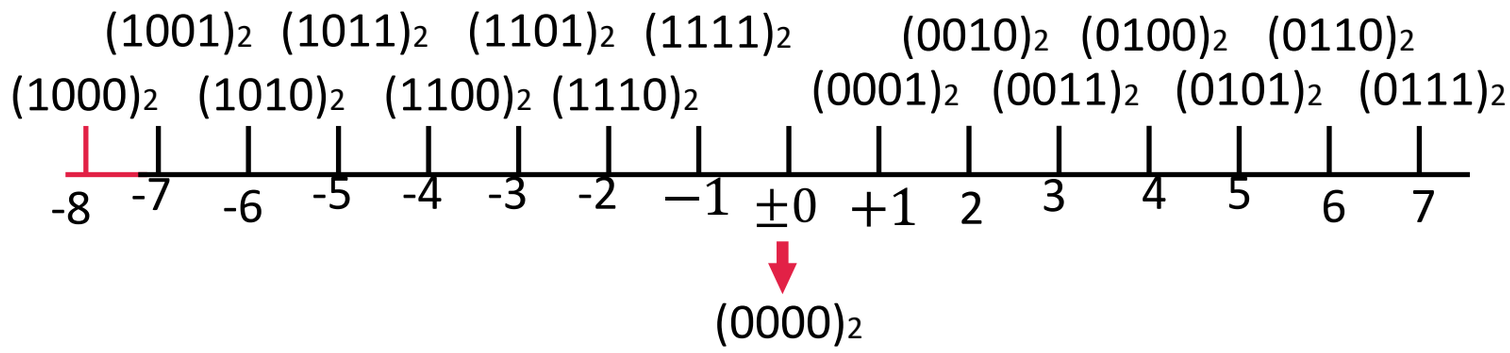


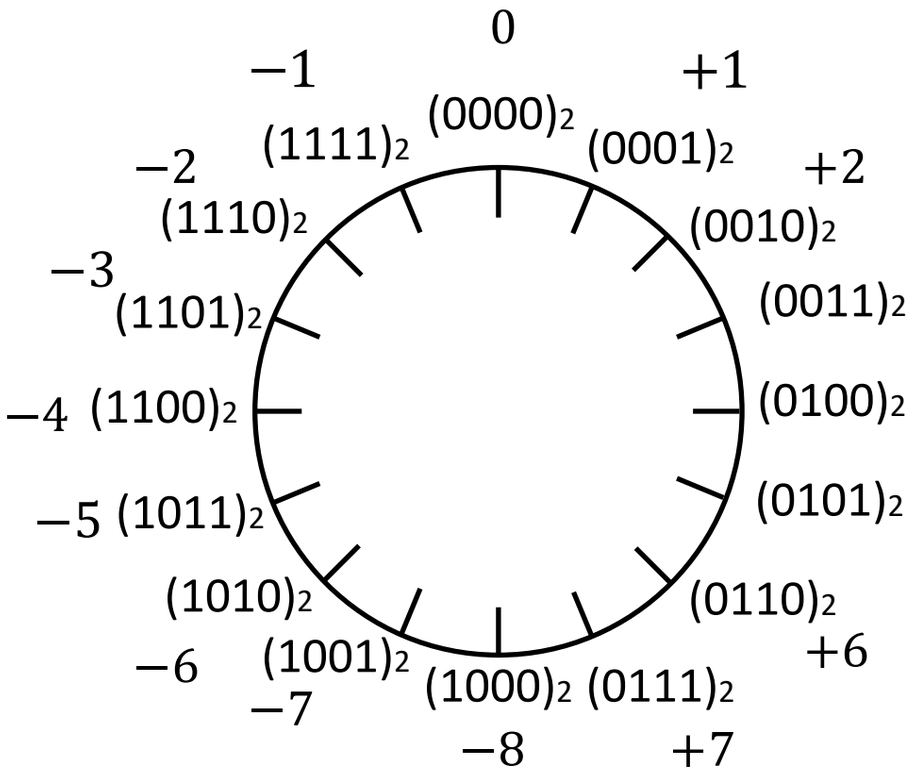
- Two's-complement representation:
  - Positive numbers, stay unchanged; Negative numbers, apply two's complement (for an n-bit number A, complement to 2^n is 2^n-A, or toggling all bits and adding 1)
- Easy for arithmetics

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two} = 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$

Sign bit

Two's-Complement Representation (Signed Integer)





- 2's complement number $(a_n a_{n-1} \dots a_1 a_0)_2$ represents

$$(a_n a_{n-1} \dots a_1 a_0)_2 = -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

- Sign extension

Two's-Complement Arithmetic (Addition & Subtraction)

3	0011	3	0011	-3	1101
+2	0010	+ (-2)	1110	+ (-2)	1110

7	0111
+1	0001

-8	1000
+ (-1)	1111

- Overflow check!

Comparison

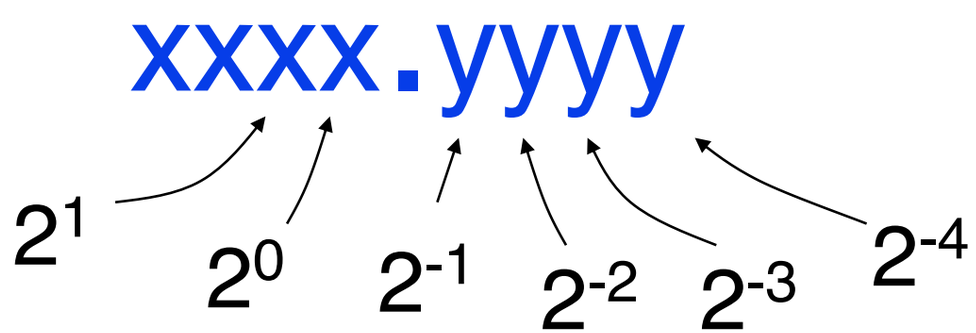
Sign-magnitude	One's-complement	Two's-complement
5 0101	5 0101	5 0101
+ (-3) 1011	+ (-3) 1100	+ (-3) 1101
0000	0001	0010
✗	✗	✓

Two's-Complement Representation (Signed Integer)

- Two's complement treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to $2^{31}-1$ (2,147,483,647)
  - Note: one negative number with no positive version
  - Every computer uses two's complement today
- Most-significant bit (MSB) (leftmost) is the sign bit, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant (MSB), bit 0 is least significant (LSB)

Fractional

- "Binary Point" like decimal point signifies boundary between integer and fractional parts:



$$0010.1010_{two} = 1x2^{1} + 1x2^{-1} + 1x2^{-3} = 2.625_{ten}$$

- Fixed-point: 1234.5678; 1234.0

Fixed-Point Numbers

Addition is straightforward

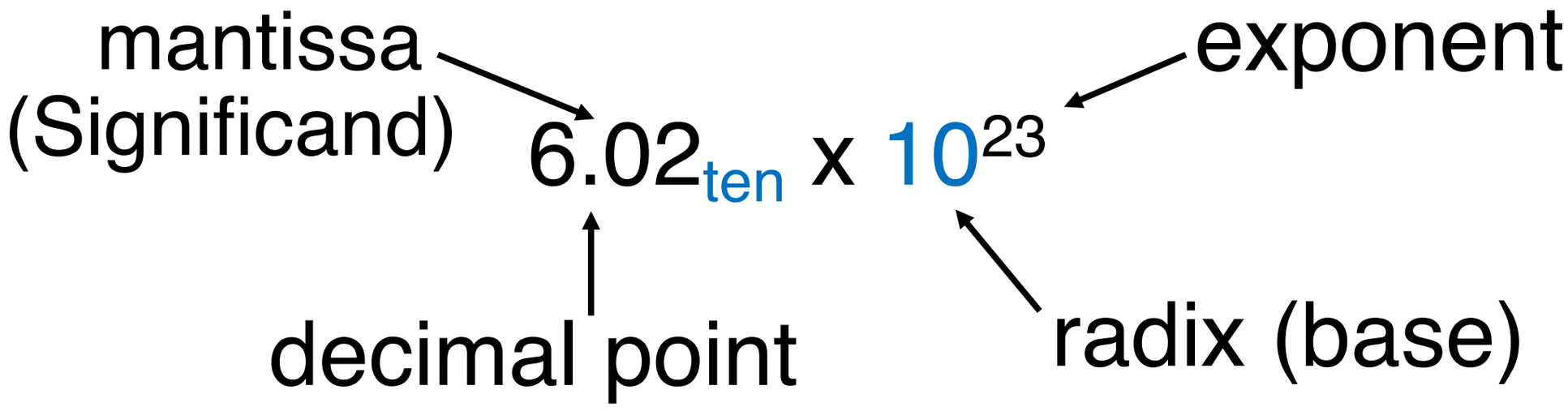
```
  01.100   1.5ten
+ 00.100   0.5ten
  10.000   2.0ten
```

```
      01.100  1.5ten
00.100  0.5ten
00 000
000 00
0110 0
 00000
 00000
0000110000
```

Multiplication a bit more complex

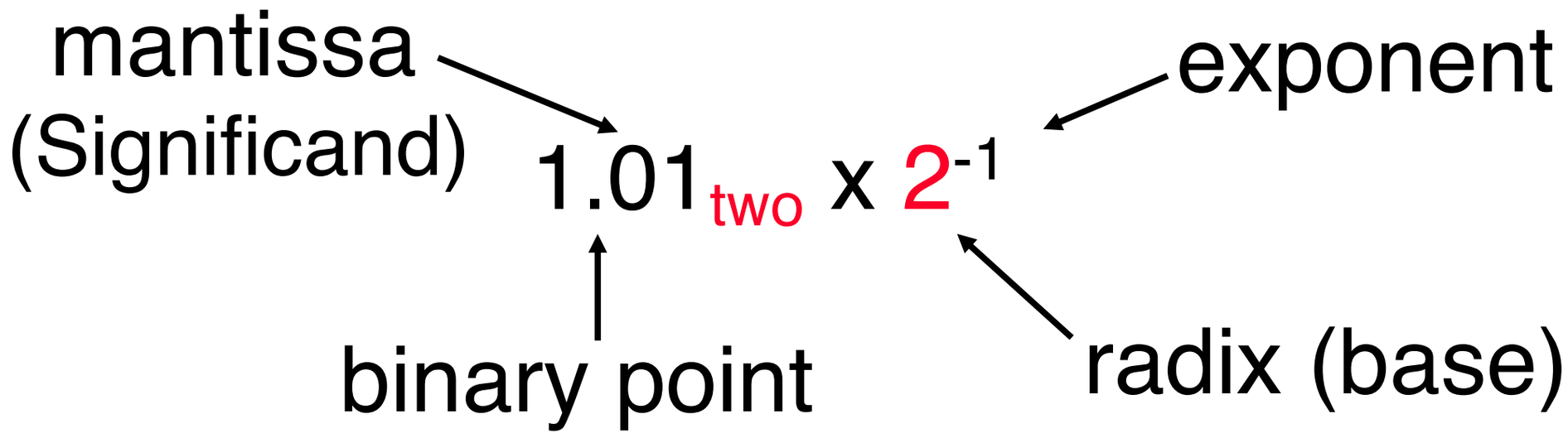
(Need to remember where point is)

Scientific Notation (in Decimal)



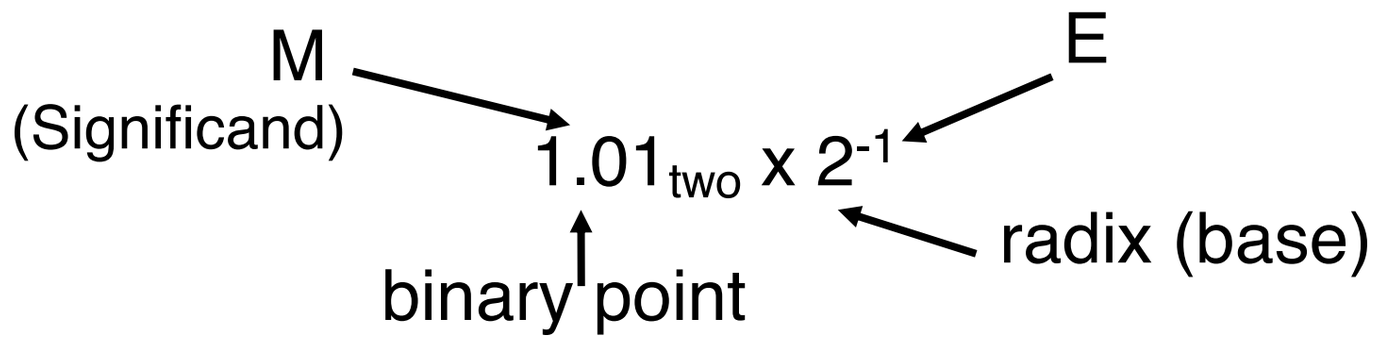
- Normalized form: no leadings 0s
  (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8},10.0×10^{-10}

Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed
  - Declare such variables in C as float (32b); double for double precision (64b).
  - How to represent in computer? Everything is a number.

Single-Precision 32-bit floating point (IEEE 754)



$$S = (-1)^{Sign} \quad E = Exponent_2 - 127_{10} \quad M = (1.Mantissa)_2$$

$$Value = S \times M \times 2^E$$

https://ieeexplore.ieee.org/document/8766229

- Biased exponent: It can represent numbers in [-127, 128], and allows comparing two floating point number easier (bit by bit) than the other representations.

Play with: https://www.h-schmidt.net/FloatConverter/IEEE754.html

How do we read a FP32 number?

Example

`0 01111011 11000000000000000000000`

- Step 1: determine the sign
  - $S = (-1)^{\text{Sign}} = (-1)^0 = 1$ (positive)
- Step 2: determine the unbiased exponent
  - $E = \text{Exponent}_2 - 127_{10} = 01111011_2 - 127_{10} = 123 - 127 = -4$
- Step 3: determine the Mantissa
  - $M = 1.\text{Mantissa}_2 = 1.11_2 = 1.75_{10}$
- Step 4: determine the converted decimal by using S, E and M
  - decimal $= S \times M \times 2^E = 1 \times 1.75 \times 2^{-4} = 0.109375$

Conversion—Store a FP32 number

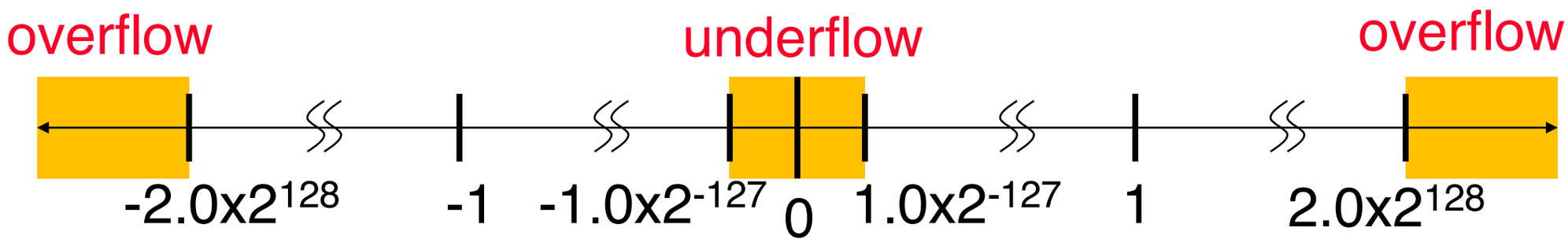
Example: 0.09375 into single precision floating point

- Step 1: determine the sign
  - Positive => Sign bit = 0
- Step 2: Convert the magnitude 0.09375 to binary
  - $0.09375_{10} = 0.00011_2$
- Step 3: Convert to scientific/normalized notation to obtain mantissa and unbiased exponent
  - $0.00011_2 = 1.1_2 \times 2^{-4}$
- Step 4: determine the biased exponent and remove the leading 1 from 1.1
  - Exponent $= -4_{10} + 127_{10} = 123_{10} = 01111011_2$, Mantissa $= 1$
- Step 5: padding 0s to the end of mantissa to make up to 23 bits/truncate if more than 23 bits

0 01111011 10000000000000000000000

Overflow vs. Underflow

- What if number too large/small? (> 2.0×2^{128} , < -2.0×2^{128})
  - Overflow! => Exponent larger than represented in 8-bit Exponent field
- What would help reduce chances of overflow?
- Double-precision (FP64): 1 (sign)- 11(exponent, bias 1023)-52(mantissa) in IEEE 754 standard

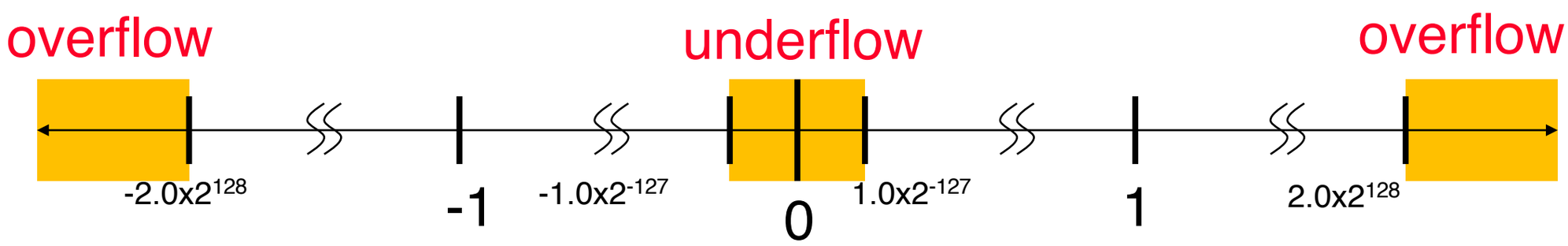


$\pm\infty$

- All exponent = 1s & All Mantissa = 0s
- Sign defined by the sign bit
- Valid Arithmetic
  - (+ - /) with finite numbers
  - Multiplication with finite/infinite non-zero numbers
  - Square root of $+\infty$
  - Conversion (e.g. fp32 ∞ to fp64 ∞)
  - remainder(x,∞) for finite normal x
- Can be produced by
  - division (x/0, x≠0), log(0), etc. along with divideByZero exception
  - Overflow with overflow exception

Overflow vs. Underflow

- What if result too small? (>0 & < $1.0x2^{-127}$, <0 & > $-1.0x2^{-127}$)
  - Underflow! => Negative exponent larger than represented in 8-bit Exponent field (have method solving it partially later)
- What would help reduce chances of underflow?
- Double-precision (FP64): 1 (sign)- 11(exponent, bias 1023)- 52(mantissa) in IEEE 754 standard

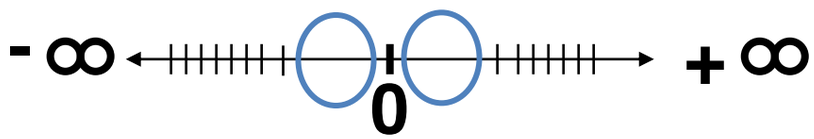


Denorms.

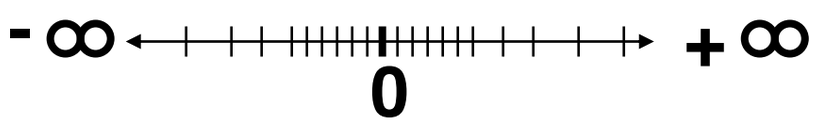
- Denormalized numbers
- Gap between smallest positive/largest negative FP numbers and 0

Normalization and hidden 1 is to blame!

Gaps!



- Use exponent all 0s, mantissa ≠ 0
- No implicit leading 1, implicit exponent = -126
- Extend smallest pos./largest neg. single-precision FP to $\pm 2^{-149}$ (non-zero)
- Followed by $\pm 2^{-148}$, $\pm 1.5 \times 2^{-148}$, $\pm 2^{-147}$, $\pm 2^{-147}$, $\pm 1.25 \times 2^{-147}$, … …
- Underflow still exists



NaN (Not-a-Number)

- Resulting from invalid operations (neither overflow nor underflow)
  - e.g. operations with NaN (quiet invalid operations, generally)
  - $0\times\infty$, $\sqrt{n}(n<0)$, $0/0$, ∞/∞, magnitude subtraction of infinities (signaling invalid operation exception)
- Exponent all 1s, mantissa non-zero, sign don't-care
- Why NaN?
  - Represent missing values
  - Find sources of NaNs using signaling NaNs

Representations for Special Cases

Exponent	Mantissa	Represented value
All ones	All zeros	± Inf
All ones	Not all zeros	Not a number (NaN)
All zeros	All zeros	± Zero
All zeros	Not all zeros	Sub/denormal

- https://ieeexplore.ieee.org/document/8766229 See sections 3.4, 6&7 for more details.

Normal numbers: Exponent 1-254, -126-127 after biasing

$$Value = S \times M \times 2^{E}$$

FP Arithmetic (normal numbers)

- Floating-point addition
  1. Alignment shift/preshift
  2. Add/subtract the significands
  3. Normalization shift/postshift
  4. Round
  5. Repeat 3,4 if not normalized
- (Detect inf and generate exception if necessary)

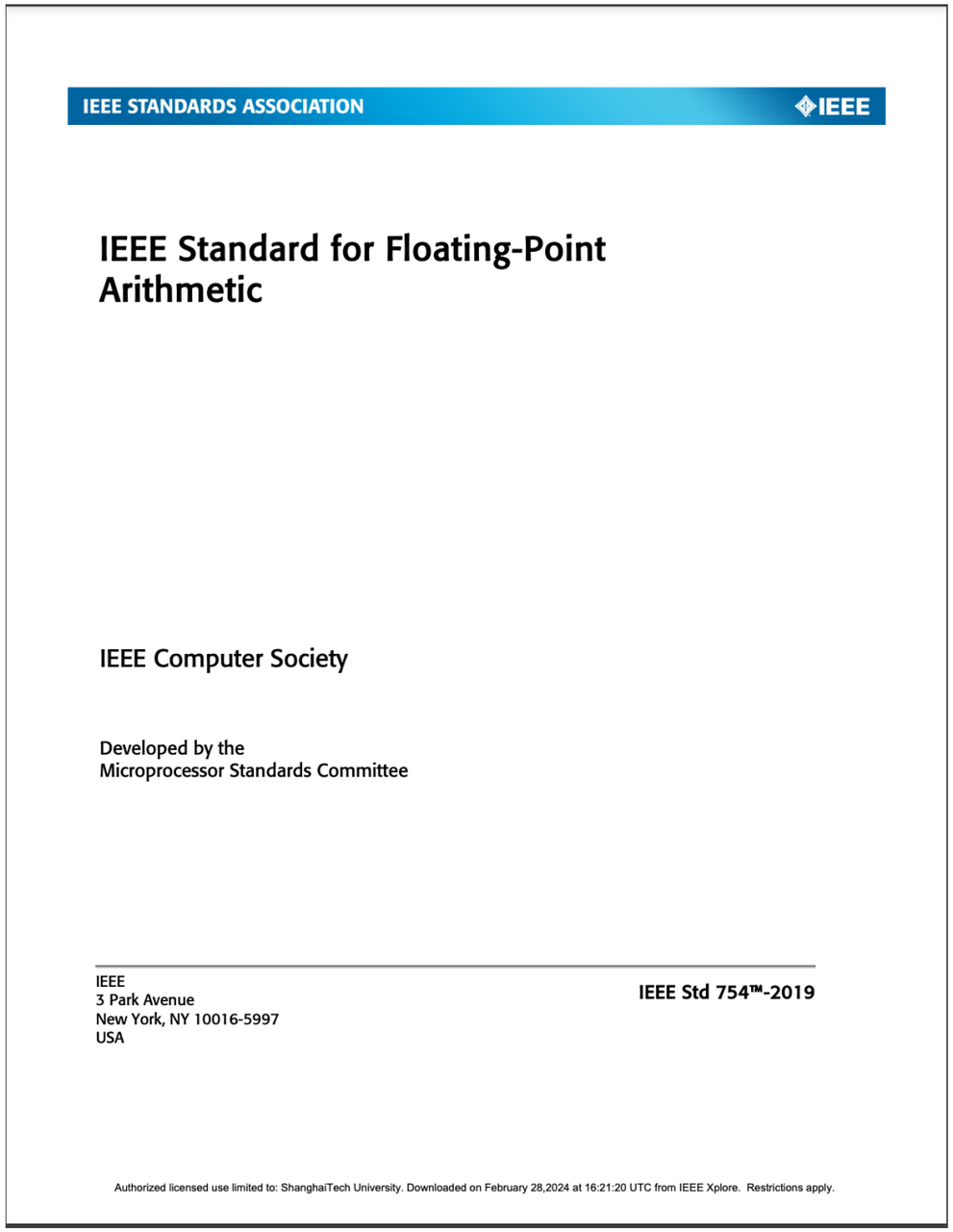
Rounding Modes

- Default to “round-to-nearest-ties-to-even” (avoid systematic biases)
- Other modes:
  - Round-to-−+∞ (up)
  - Round-to-−−∞ (down)
  - Round-towards-0
  - Round-to-nearest-ties-to-max-magnitude (roundTiesToAway, similar to SiSheWuRu)
- Used for FP arithmetics and FP-integer conversions

FP Arithmetic (normal numbers)

- Floating-point multiplication
  1. Calculate the actual exponents (if considering the bias)
  2. Add the actual exponents and add the bias (-127)
  3. Multiply the significands
  4. Normalize if necessary
  5. Round
  6. Repeat 4,5 if not normalized
  7. Calculate the sign bit
- (Detect inf and generate exception if necessary)

IEEE 754 standard



IEEE STANDARDS ASSOCIATION IEEE

IEEE Standard for Floating-Point Arithmetic

IEEE Computer Society

Developed by the
Microprocessor Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 754™-2019

https://ieeexplore.ieee.org/document/8766229