

Lecture 06

Smaller Floating Point, FP Arithmetic, and Intro to Assembly Languages

Comp_Sci 213

Introduction to Computer Systems

Monday, Oct. 12, 2026

Kate Smith – Fall 2026

Slides adapted from:

Ghena, St-Amour, Hardavellas, Bustamante (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Notes

- Pack Lab due Tuesday, Oct. 13
- HW 1 due Thursday, Oct. 15
- Midterm on Monday, Oct. 26
 - More info about this next time

Topics

- Floating Point
 - Fractional binary numbers
 - Representing floating point
 - Smaller floating point
- Floating Point Arithmetic
- Intro to Assembly Languages

***If you are following along in the textbook, FP gets us to the end of Chapter 2. Assembly overview gets us started on Chapter 3!*

Topics

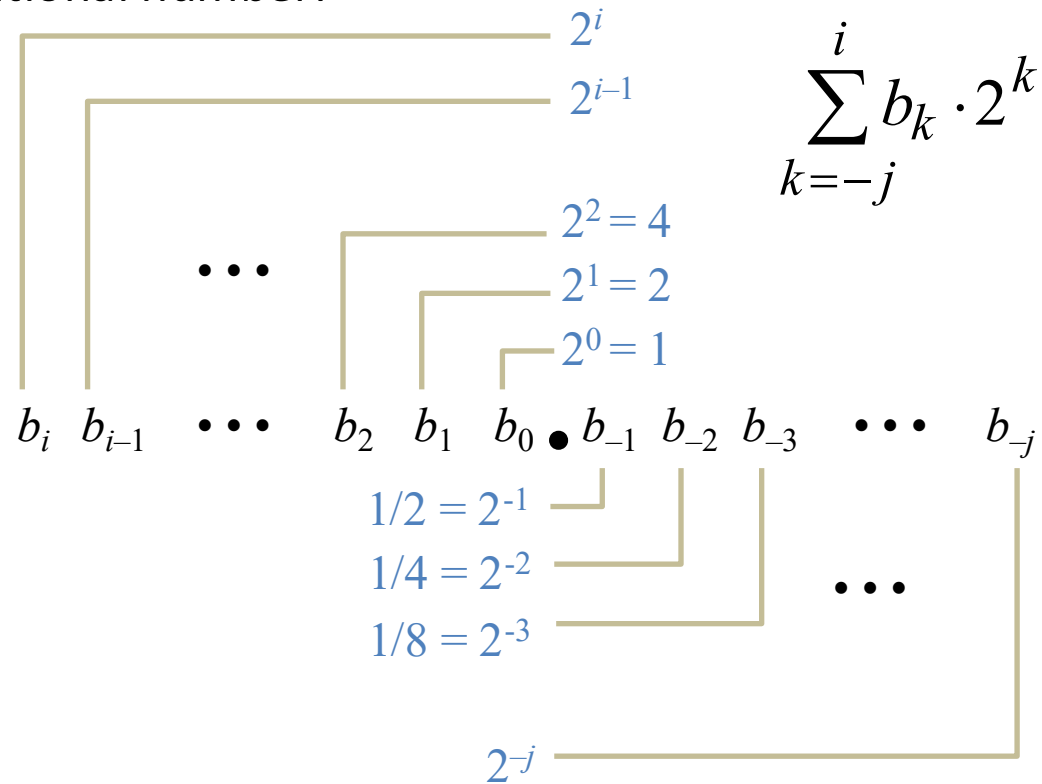
- **Floating Point**
 - **Fractional binary numbers**
 - Representing floating point
 - Smaller floating point
- Floating Point Arithmetic
- Intro to Assembly Languages

Floating point numbers

- In decimal:
 - 123450_{10}
 - 123.450_{10}
 - 1.23450_{10}
- We can use this same system in binary as well:
 - 1010110_2 (86_{10})
 - 1010.110_2 ($10.75_{10} = \frac{86}{2^3}$)
 - 1.010110_2 ($1.34375_{10} = \frac{86}{2^6}$)

Fractional Binary Numbers

- Representation
 - Bits to right of “binary point” represent fractional powers of 2
 - Represents rational number:



Example binary conversion

1010.110

Before the binary point:

$$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 1*2^3 + 1*2^1 = 8+2 = 10$$

After the binary point:

$$1*2^{-1} + 1*2^{-2} + 0*2^{-3} = 1*2^{-1} + 1*2^{-2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4} = 0.75$$

Fractional Binary Number Examples

- $5 + 3/4$ $= 0b101.\boxed{11}$ Note:
This the number 3!
- $2 + 7/8$ $= 0b10.\boxed{111}$ This is the number 7!
- $63/64$ $= 0b0.\boxed{111111}$ This is the number 63!

Scientific Notation Binary

- Scientific notation works in binary just like in decimal
 - Decimal: $134 = 1.34 * 10^2$
 - Binary: $101 = 1.01 * 2^2$
- Positive and negative exponents change the direction the point moves
 - Positive powers get bigger: $1.00101 * 2^4 \rightarrow 10010.1$
 - Negative powers get smaller: $1.00101 * 2^{-3} \rightarrow 0.00100101$

Binary point is part of the solution, but not an entire encoding

- Some problems remain:
 1. Computers are finite, but real numbers are not
 - Need to choose how many bits to use
 - Many decimal numbers would take infinite binary bits to represent perfectly
 - $3.14_{10} = 11.0010001111010111_2$ (we could keep going)
 2. We also need to represent where the “binary point” is located
 - We’ll use scientific notation and some of our bits to do so
 3. Should include signed numbers while we’re at it

Topics

- **Floating Point**
 - Fractional binary numbers
 - **Representing floating point**
 - Smaller floating point
- Floating Point Arithmetic

Floating Point Standard – IEEE754

https://en.wikipedia.org/wiki/IEEE_754

- Floating point representations
 - Encodes rational numbers of the form $V = m \times 2^e$
 - Base 2 scientific notation!
- IEEE Standard 754 (IEEE floating point)
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many various formats with different rules
 - Headed by William Kahan, CS prof. at UC Berkeley (later won Turing Award for it)
 - Supported by all major CPUs
- Driven by numerical concerns and numerical analysts
 - Nice standards for rounding, overflow, underflow
 - Had to be implementable in fast hardware as well and support many languages

Floating Point Representation

Numerical form:

$$V = (-1)^S * M * 2^E$$

Sign bit Significand (Mantissa) Exponent

- Sign bit **S** determines whether number is negative (1) or positive (0)
- Significand **M** normally a fractional value in range [1.0,2.0) or [0.0,1.0)
 - Called ***mantissa*** or ***significand***
- Exponent **E** weights value by power of two
- Heads up: Most of this equation is decimal, except Significand, which is floating point binary

IEEE754 Floating Point Encoding

Numerical form:

$$V = (-1)^s * M * 2^E$$

Diagram illustrating the numerical form of a floating-point number V :

- s : Sign bit (indicated by a blue arrow labeled "Sign bit")
- M : Significand (Mantissa) (indicated by a blue arrow labeled "Significand (Mantissa)")
- E : Exponent (indicated by a blue arrow labeled "Exponent")

- IEEE754 Encoding
 - MSb is sign bit (can still look at most-significant bit alone to determine sign!)
 - **exp** field encodes E , k -bits (note: "*encodes E*" != "*is E*")
 - **frac** field encodes M , n -bits



IEEE754 Floating Point Precision

- Sizes

- Single precision: $k = 8$ exp bits, $n = 23$ frac bits (32b total). **float** in C



- Double precision: $k = 11$ exp bits, $n = 52$ frac bits (64b total). **double** in C



Categories for IEEE754 Encoded Values

Value encoded – three cases, depending on value of **exp**

1. Normalized, the most common



2. Denormalized (very small values)



3. Special values – infinity and NaN



Categories for IEEE754 Encoded Values

Value encoded – three cases, depending on value of **exp**

1. Normalized, the most common



2. Denormalized (very small values)



3. Special values – infinity and NaN



Normalized, Significand

$$V = (-1)^s * \boxed{M} * 2^E$$

- Condition: not a special exponent (all zeros or ones)
- Significand (M) is encoded with implied leading 1
 - $M = 1.xxx...x_2$ (implied leading 1 representation)
 - xxx...x: bits of **frac** used directly (i.e., 1.**frac**)
 - **frac** ranges from [0,1)
- Idea: every normalized number is 1.xxxx
 - So we're not going to include the leading 1 in the frac
 - We'll just know it's there when we convert to decimal
 - Saves one extra bit in the encoding!



$$V = (-1)^s * M * 2^E$$

Normalized, Exponent

- Condition: not a special exponent (all zeros or ones)
- Exponent coded as a biased value
 - $E = \text{Exp} - \text{Bias}$
 - Exp : unsigned value denoted by **exp**
 - Bias : Bias value = $2^{k-1} - 1$, k is number of exponent bits
 - Single precision (8-bit exp): 127 (Exp: 1...254, E: -126...127)
 - Double precision (11-bit exp): 1023 (Exp: 1...2046, E: -1022...1023)
- Exponent really just pushes the binary point around
 - $1.11 * 2^2 = 11.1 * 2^1 = 111.0 * 2^0 = 111$
 - $111 * 2^{-2} = 11.1 * 2^{-1} = 1.11 * 2^0 = 1.11$



Decoding example for normalized floating point (32-bit)

- $0xC1900000 = 0b11000001100100000000000000000000$
 - Group bits **s**: 1 **exp**: 10000011 **frac**: 001000000000000000000000
 - **exp** is not all zeros or all ones $\Rightarrow$ not a special case
- $M = 1.001000000000000000000000 = 1.001$
- $\text{exp} = 10000011 = 131$
 - $E = \text{exp} - \text{bias} = 131 - 127 = 4$
 - $\text{bias} = 2^{k-1} - 1, k=8 \rightarrow 2^7 - 1 = 127$
- $\text{Result} = (-1)^1 * 1.001_2 * 2^4 = -10.01_2 * 2^3 = -10010.2 = -18$

$V = (-1)^s * M * 2^E$	<table border="1"><tr><td>s</td><td>exp</td><td>frac</td></tr></table>	s	exp	frac
s	exp	frac		

Normalized Encoding Example

- **Value**

- `float F = 15213.0; // single precision: 8 exp bits, 23 frac bits`
- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

- **Significand**

- $M = 1.\underline{1101101101101}_2$
- $\text{frac} = \underline{1101101101101}0000000000$ pad with 0s *on the right*. (example: 1.5 = 1.500)

- **Exponent**

- $E = 13$
- Bias = 127
- $\text{exp} = E + \text{Bias} = 140 = 10001100_2$

Floating Point Representation:

Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
exp:	100	0110	0					
frac:				110	1101	1011	0100	0000 0000

Normalized Numbers: Why These Choices?

- Significand coded with implied leading 1
 - Any non-zero integer will start with a 1 bit somewhere
 - Leading 1 carries no information, so don't need to store it!
 - Can express mantissas between:
 - 1.0 when frac is all 0s
 - 2.0 (nearly) when frac is all 1s
 - Want smaller? Use a smaller exponent!
- Exponent coded as biased value
 - $E = Exp - Bias$
 - Alternative to using two's complement to represent signed integers
 - Reasons are a bit tricky
 - Floating point binary values increase in the same order as unsigned, which means they can be compared just like integer encodings
 - Bias also provides a more useful range (when considering denormalized)

Example

- $0x3F800000 = 0b00111111100000000000000000000000$
 - Group bits **s**: 0 **exp**: 01111111 **frac**: 00000000000000000000000000000000
 - **exp** is not 0...0 or 1...1 $\Rightarrow$ not a special case
- $M =$
- $E = \mathbf{exp} - \mathbf{bias} =$
 - $\mathbf{bias} = 2^{k-1} - 1, k=8 \rightarrow 2^7 - 1 = 127$

$$V = (-1)^s * M * 2^E$$



Example

- $0x3F800000 = 0b00111111100000000000000000000000$
 - Group bits **s**: 0 **exp**: 01111111 **frac**: 00000000000000000000000000000000
 - **exp** is not 0...0 or 1...1 $\Rightarrow$ not a special case
- $M = 1.00000000000000000000000000000000 = 1.0$
- $E = \mathbf{exp} - \mathbf{bias} = 127 - 127 = 0$
 - $\text{bias} = 2^{k-1} - 1, k=8 \rightarrow 2^7 - 1 = 127$
- $\text{Result} = (-1)^0 * 1.0_2 * 2^0 = 1$

$$V = (-1)^s * M * 2^E$$



Categories for IEEE754 Encoded Values

- Value encoded – three cases, depending on value of **exp**

1. Normalized, the most common



2. Denormalized (very small values)

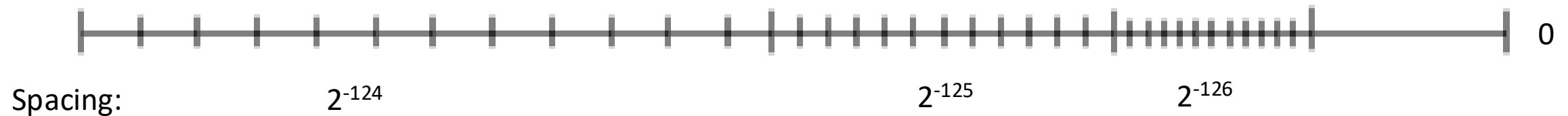


3. Special values – infinity and NaN

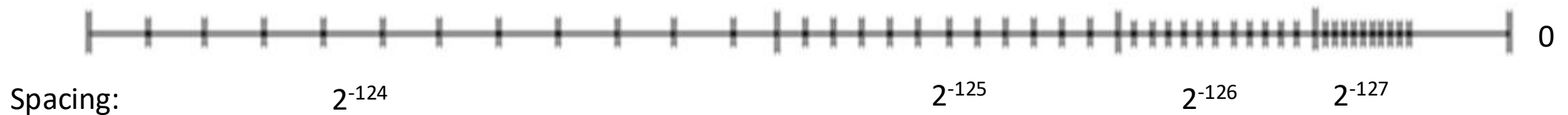


Normalized floating point leaves a gap around zero

- Gap is the size of $1.0000 * 2^{\text{Min Exponent}}$ (due to leading 1 bit)
 - How do we encode “zero?”

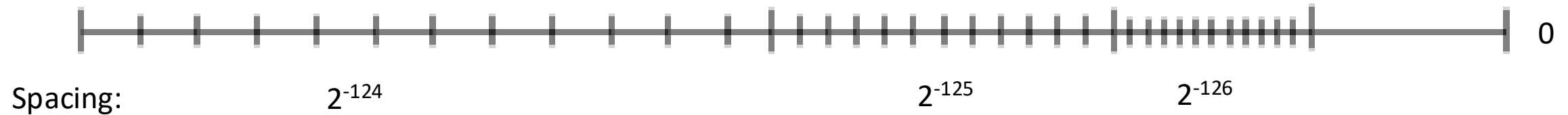


- Problem: if we just kept doing what we're doing we never reach 0
 - We keep getting half-way there

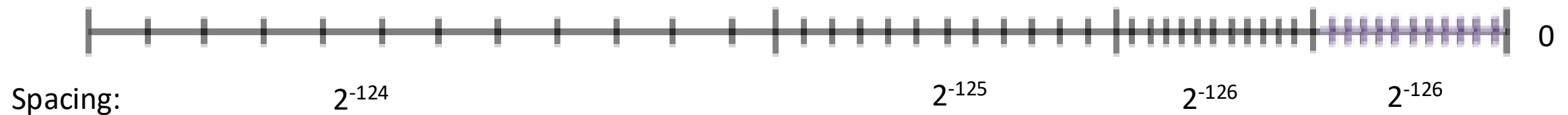


Solution is to do something different for the smallest numbers

- Problem: always starting with 1.XXXXX leaves a gap



- Solution: fill in numbers between 0 and $1 * 2^{\text{Min Exponent}}$
 - Using same spacing as the previous range, in the form 0.XXXXX
 - **Gradual underflow** – possible numeric values are spaced evenly near 0.0



$$V = (-1)^s * M * 2^E$$

Denormalized Values

- Purpose: gracefully represent numbers approaching ± 0
- Condition: $\text{exp} = 000\dots 0_2$
- Value
 - Exponent value $E = \underline{1 - \text{Bias}}$
 - Note: NOT $E = 0 - \text{Bias}$ as it would be if we followed the previous rules
 - This means we're instead re-using the spacing from smallest normalized numbers
 - Significand value $M = \underline{0}.\text{xxx}\dots\text{x}_2$ ($0.\text{frac}$)
 - $\text{xxx}\dots\text{x}$: bits of frac . Leading 0 instead of leading 1
- Cases
 - $\text{exp} = 000\dots 0$, $\text{frac} = 000\dots 0 \Rightarrow$ Represents value 0
 - Note that we have distinct values $+0$ and -0 , which depends on if the sign bit is 0 or 1, respectively
 - $\text{exp} = 000\dots 0$, $\text{frac} \neq 000\dots 0 \Rightarrow$ Numbers very close to 0.0

Categories for IEEE754 Encoded Values

- Value encoded – three cases, depending on value of **exp**

1. Normalized, the most common



2. Denormalized (very small values)



3. Special values – infinity and NaN



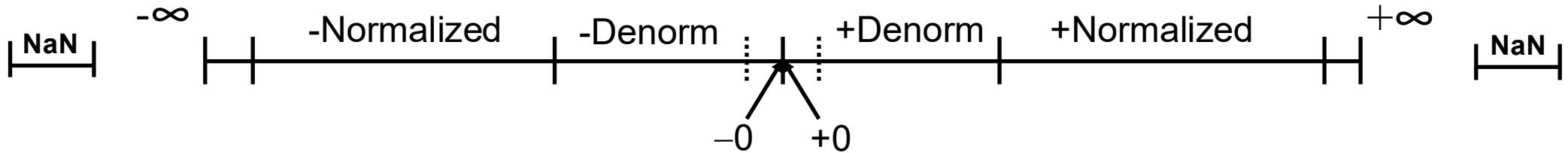
Special Values

- Purpose: represent quantities that $(-1)^s * M * 2^E$ cannot
- Condition: $\text{exp} = 111\dots 1_2$
- Cases
 - $\text{exp} = 111\dots 1_2, \text{frac} = 000\dots 0_2$
 - Represents value ∞ (infinity)
 - Both positive and negative infinity (sign bit to tell apart, $s=0$ is positive and $s=1$ is negative)
 - Operation that overflows: nicer mathematical behavior than truncation/modulo when numbers get too big or small!
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $-1.0/0.0 = -\infty$
 - IEEE 754 has a divide by zero exception that prevents programs from crashing since default behavior is to produce an infinity value and set a floating-point status flag
 - $\text{exp} = 111\dots 1_2, \text{frac} \neq 000\dots 0_2$
 - Not-a-Number (NaN)
 - Represents case when no numeric value can be determined
 - Fraction could be used to distinguish sources (rarely used in practice)
 - E.g., $\sqrt{-1}$, $\infty - \infty$, $\infty * 0$

Floating Point in C

- C guarantees two levels
 - **float** single precision
 - **double** double precision
- Conversions
 - **int** → **float**
 - maybe rounded
 - less bits for actual value (32 → 23)
 - **int** or **float** → **double**
 - exact value preserved
 - double has greater range and higher precision (52 bits for **frac**)
 - **double** → **float**
 - may overflow, underflow (too small to represent), or be rounded (IEEE 754)
 - C99 standard says **undefined** if value out of range
 - **double** or **float** → **int**
 - rounded toward zero (-1.999 → -1)
 - C99 standard says **undefined** if value out of range

Summary of FP Real Number Encodings



$$V = (-1)^s * M * 2^E$$

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	$\text{exp} \neq 000\dots 0_2$ and $\text{exp} \neq 111\dots 1_2$	$\text{exp} = 000\dots 0_2$
frac	$x_1x_2x_3\dots x_j$	$x_1x_2x_3\dots x_j$
Bias=	$2^{(k-1)} - 1$, for k exponent bits	$2^{(k-1)} - 1$, for k exponent bits
E=	$\text{exp} - \text{Bias}$	$1 - \text{Bias}$
M=	$1.x_1x_2x_3\dots x_j$ a.k.a. 1.frac	$0.x_1x_2x_3\dots x_j$ a.k.a. 0.frac
V=	$(-1)^s \times (1.\text{frac}) \times 2^{(\text{exp} - \text{Bias})}$	$(-1)^s \times (0.\text{frac}) \times 2^{(1 - \text{Bias})}$

Topics

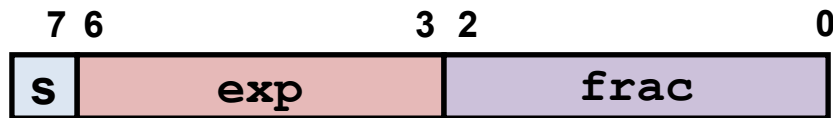
- **Floating Point**
 - Fractional binary numbers
 - Representing floating point
 - **Smaller floating point**
- Floating Point Arithmetic
- Intro to Assembly Languages

Floating point examples

- We'll often do floating point in custom bit widths
 - Rather than 32-bit (float) or 64-bit (double)
- Reasons
 1. 64 is just too many bits to write out and think about
 - And the math gets hard to do without a calculator
 2. Make sure you understand the concepts of floating point
 - Smaller versions still demonstrate concepts! (e.g., 8-bit)

Example: Tiny Floating Point

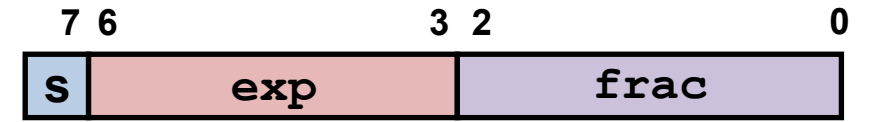
- 8-bit Floating Point Representation
 - Sign bit is in the most significant bit.
 - Next four (k) bits are exp, with a bias of 7 ($2^{k-1}-1$)
 - Last three (n) bits are frac
- Same general form as IEEE 754 format
 - normalized, denormalized numbers
 - representation of 0, NaN, infinity



Sidebar: increasingly useful for Machine Learning use!

- Models often don't need 32-bits of precision

Denormalized encoding example

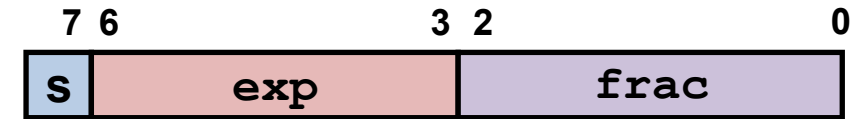


- Convert 5/512 to 8-bit tiny float
 - $5/512 = 0b101 * 2^{-9} = 10.1 * 2^{-8} = 1.01 * 2^{-7}$
- $E = \text{exp} - \text{bias} \rightarrow -7 = \text{exp} - (2^{(4-1)} - 1) \rightarrow -7 = \text{exp} - 7$
 - $\text{exp} = 0 ???$
 - But exp can't be less than 1 (or we're denormalized)
 - So, the answer must be a denormalized number. Reset the problem!

Denormalized encoding example

- Convert 5/512 to 8-bit tiny float

- $5/512 = 0b101 * 2^{-9} = 10.1 * 2^{-8} = 1.01 * 2^{-7}$



- $E = 1 - \text{bias} = 1 - 7 = -6$
- $0.xxx * 2^{-6} \rightarrow 1.01 * 2^{-7} = 0.101 * 2^{-6}$
- S: 0 (positive) exp: 0000(denorm) frac: 101
- 0b0 0000 101 $\rightarrow$ 0x05

Exponents for 8-bit tiny floats

$$\text{Bias} = 2^{4-1} - 1 = 7$$

(4-bit exp)

Denormalized
 $E = 1 - \text{Bias}$

Normalized
 $E = \text{exp} - \text{Bias}$

Special

exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, NaN)

Dynamic Range of 8-bit tiny float

Bias = 7

$$V = (-1)^s$$

$$\times (0.\text{frac})$$

$$\times 2^{(1 - \text{Bias})}$$

Denormalized numbers

Normalized numbers

$$V = (-1)^s$$

$$\times (1.\text{frac})$$

$$\times 2^{(\text{exp} - \text{Bias})}$$

Special values

s exp frac

E

Value

Notes of Interest

0 0000 000

-6

0

0 0000 001

-6

$1/8 * 1/64 (2^{-6}) = 1/512$

closest to zero

0 0000 010

-6

$2/8 * 1/64 = 2/512$

...

0 0000 110

-6

$6/8 * 1/64 = 6/512$

0 0000 111

-6

$7/8 * 1/64 = 7/512$

largest denorm

0 0001 000

-6

$8/8 * 1/64 = 8/512$

smallest norm

0 0001 001

-6

$9/8 * 1/64 = 9/512$

...

0 0110 110

-1

$14/8 * 1/2 = 14/16$

0 0110 111

-1

$15/8 * 1/2 = 15/16$

closest to 1 below

0 0111 000

0

$8/8 * 1 = 1$

0 0111 001

0

$9/8 * 1 = 9/8$

closest to 1 above

0 0111 010

0

$10/8 * 1 = 10/8$

...

0 1110 110

7

$14/8 * 128 = 224$

0 1110 111

7

$15/8 * 128 = 240$

largest norm

0 1111 000

n/a

inf

0 1111 001

n/a

NaN

...

0 1111 111

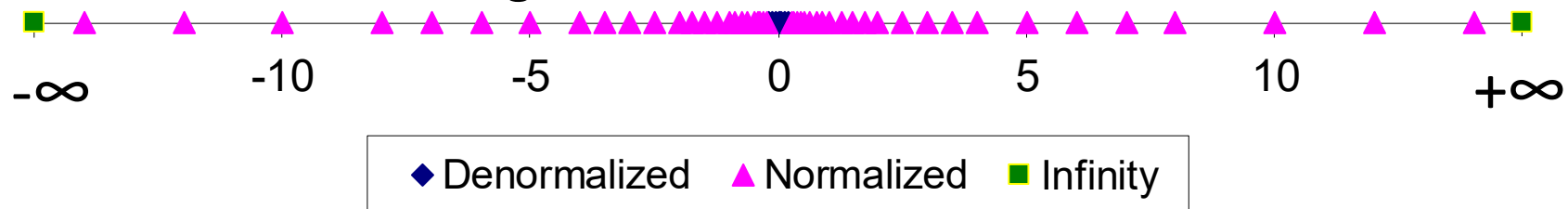
n/a

NaN

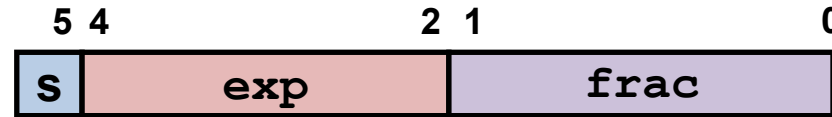
Distribution of Values

- 6-bit IEEE-like format
 - exp = 3 exponent bits
 - frac = 2 fraction bits
 - Bias is 3 ($2^{3-1}-1$)

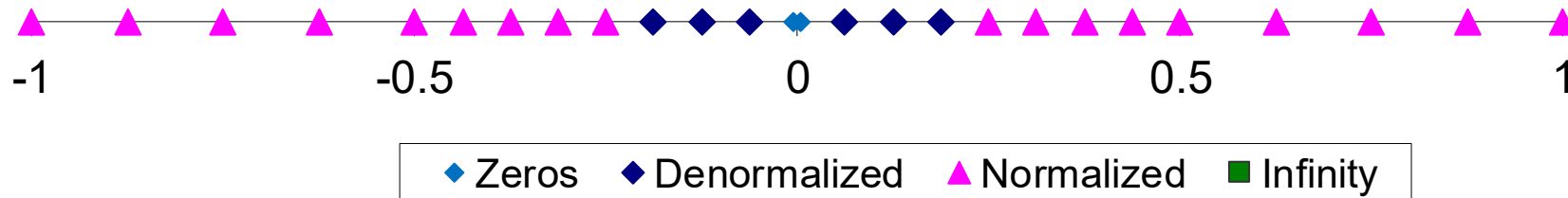
- Notice how the distribution gets denser toward zero.



Distribution of Values (Close-up View)



- 6-bit IEEE-like format
 - exp = 3 exponent bits
 - frac = 2 fraction bits
 - Bias is 3 ($2^{3-1}-1$)
- Smooth transition between normalized and de-normalized numbers due to definition $E = 1 - \text{Bias}$ for denormalized values
 - Zeros are denormalized numbers too! (+0 and -0)



Topics

- Floating Point
 - Fractional binary numbers
 - Representing floating point
 - Smaller floating point
- **Floating Point Arithmetic**
- Intro to Assembly Languages

Floating Point Operations

- Conceptual view
 - $x +_{\text{float}} y = \text{Fit}(x +_{\text{math}} y)$
 - $x *_{\text{float}} y = \text{Fit}(x *_{\text{math}} y)$
- First compute exact, mathematical result
 - As a human: convert to decimal first, do math in decimal
- Then make it fit into desired precision
 - **Step 1:** Determine frac, exp (decimal $\rightarrow$ float conversion we already practiced)
 - Frac must be of the form 1.xxxx (0.xxx if denormalized)
 - Change exp if needed to get frac to that form (e.g., if result is 101.xxx)
 - **Step 2:** Possibly overflow if exponent too is large
 - Unlike integer overflow, result is mathematically reasonable: set to infinity
 - **Step 3:** Possibly round to fit into frac if we have too many mantissa bits

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
- Round-to-even example
 - Illustrated with rounding of money

	\$1.40	\$1.60
Rounded	\$1	\$2

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
- Round-to-even example
 - Illustrated with rounding of money

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
Rounded	\$1	\$2			

Rounding

- Default rounding mode for IEEE floating point is Round-to-even
 - Other methods are statistically biased (round up, round down, round-to-zero)
 - Sum of set of positive numbers will consistently be over- or under- estimated
 - Round to nearest number
 - If **exactly** in between, round to nearest **even** number
 - Round-to-even example
 - Illustrated with rounding of money
- | | | | | | |
|---------|--------|--------|--------|--------|---------|
| | \$1.40 | \$1.60 | \$1.50 | \$2.50 | -\$1.50 |
| Rounded | \$1 | \$2 | \$2 | \$2 | -\$2 |
- Both round to \$2 because this produces an even least significant digit*

Closer Look at Round-to-even

- Rounding to other decimal places than the decimal point
 - When exactly halfway between two possible values
 - Round so that least significant digit is even
 - E.g., round to nearest hundredth (i.e., 2 decimal digits in fractional part)
 - $1.23\underline{49999}$ $\Rightarrow 1.23$ (Less than half way)
 - $1.23\underline{50001}$ $\Rightarrow 1.24$ (Greater than half way)
 - $1.23\underline{50000}$ $\Rightarrow 1.24$ (Half way—round to even)
 - $1.24\underline{50000}$ $\Rightarrow 1.24$ (Half way—round to even)

Rules for IEEE754 rounding

1. Always truncate the bits that don't fit
 2. If bits were truncated, round. Only two options for rounding:
 - Add 0 to least-significant remaining bit (round down)
 - Add 1 to least-significant remaining bit (round up)
- To decide which, look at the bits that are being truncated:
 - If they are less than 100..., round down (add 0)
 - If they are more than 100..., round up (add 1)
 - If they exactly equal 100...,
 - Either add 0 or 1, whichever makes the least-significant remaining bit 0
 - 0 is even

Rounding Binary Numbers

- Rules reminder:
 - If they are less than **100...**, round down (add 0)
 - If they are more than **100...**, round up (add 1)
 - If they exactly equal **100...**,
 - Either add 0 or 1, whichever makes the least-significant remaining bit 0
- Examples
 - Round to nearest 1/4 (keep 2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
$2+3/32$	$10.00\textcolor{red}{11}_2$	10.00_2	(<1/2—down)	2
$2+3/16$	$10.00\textcolor{red}{110}_2$	10.01_2	(>1/2—up)	$2+1/4$
$2+3/8$	$10.01\textcolor{red}{100}_2$	10.10_2	(1/2—up to even)	$2+1/2$
$2+5/8$	$10.10\textcolor{red}{100}_2$	10.10_2	(1/2—down to even)	$2+1/2$
$2+7/8$	$10.11\textcolor{red}{100}_2$	11.00_2	(1/2—up to even)	3

Important: remember how rounding works

- Only two options when rounding
 - Leave the number alone
 - Or add one to the number
- 1010.000010010000
 - Part to remove is 10...0, so we need to round
 - Only options are:
 - 1010.0000100 (leave it alone)
 - 1010.0000101 (add one)
 - Pick the one that ends in zero: 1010.0000100

Mathematical Properties of FP Arithmetic

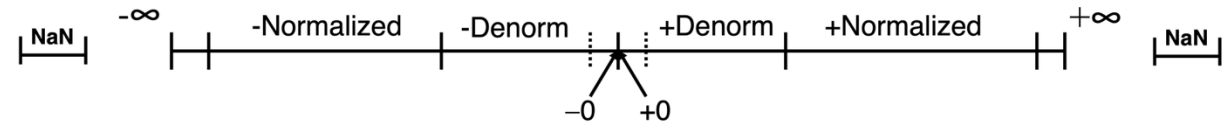
- Mathematical properties of FP Addition
 - Addition is Associative? **NO**
 - $(x + y) + z = x + (y + z)$
 - Possibility of overflow and inexactness of rounding
 - $(3.14 + 1e10) - 1e10 = 0$ (rounding)
 - $3.14 + (1e10 - 1e10) = 3.14$
- Mathematical properties of FP Multiplication
 - Multiplication is Associative? **NO**
 - $(x \times y) \times z = x \times (y \times z)$
 - Possibility of overflow, inexactness of rounding
 - Multiplication distributes over addition? **NO**
 - $x \times (y + z) = (x \times y) + (x \times z)$
 - Possibility of overflow, inexactness of rounding

Floating Point Summary

- IEEE Floating point (IEEE 754) has clear mathematical properties
 - But not always the ones you may expect!
- Represents numbers of form $V = (-1)^S \times M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not quite the same as arithmetic on real numbers
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

Flow for floating point

- Translating between decimal and IEEE754 floating point has a clear set of steps that you always take



- Steps for IEEE754 to Decimal:

1. IEEE754 encoding
2. Scientific notation binary
3. Floating point binary
4. Decimal

$$V = (-1)^s * M * 2^E$$

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	$\text{exp} \neq 000\dots 0_2$ and $\text{exp} \neq 111\dots 1_2$	$\text{exp} = 000\dots 0_2$
frac	$x_1x_2x_3\dots x_j$	$x_1x_2x_3\dots x_j$
Bias=	$2^{(k-1)} - 1$, for k exponent bits	$2^{(k-1)} - 1$, for k exponent bits
E=	$\text{exp} - \text{Bias}$	$1 - \text{Bias}$
M=	$1. x_1x_2x_3\dots x_j$ a.k.a. 1.frac	$0. x_1x_2x_3\dots x_j$ a.k.a. 0.frac
V=	$(-1)^s \times (1.\text{frac}) \times 2^{(\text{exp} - \text{Bias})}$	$(-1)^s \times (0.\text{frac}) \times 2^{(1 - \text{Bias})}$

- Steps from Decimal to IEEE754:

1. Decimal
2. Floating point binary
3. Scientific notation binary
4. IEEE754 encoding

Special numbers:

- Exp all 1s, frac all 0s $\rightarrow \infty$
- Exp all 1s, frac $\neq 0 \rightarrow \text{NaN}$

Example: Decimal to FP

Consider the integer 3,510,593

Example: Decimal to FP

Consider the integer 3,510,593

Step 1: Determine binary representation

- $3,510,593_{10} \rightarrow 0x00359141$
- $0x00359141 \rightarrow 0b0000\ 0000\ 0011\ 0101\ 1001\ 0001\ 0100\ 0001$
- *Truncate leading 0s*

Example: Decimal to FP

Step 2: Scientific notation in binary

- 11 0101 1001 0001 0100 0001
- Shift right by 21 places
- $1.1010110010001010000001 * 2^{21}$

Example:

Decimal to FP

Single precision: $k = 8$ exp bits, $n = 23$ frac bits (32b total). `float` in C



Step 3: Form the fraction field for $1.1010110010001010000001 * 2^{21}$

- Drop the leading 1 and add 2 zeros (we need 23 bits in frac for single precision)
- 101011001000101000000100

Step 4: For `exp` field for $1.1010110010001010000001 * 2^{21}$

- The exponent is formed by adding bias to 127 to 21, giving 148
- $148_{10} \rightarrow 0b10010100$

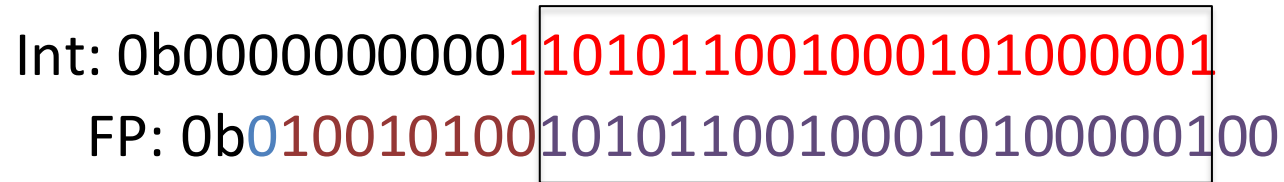
Step 5: Combine `exp` and `frac` with a sign bit (s) of 0

- 0b01001010010101100100010100000100

Example: Decimal to FP

Interesting note: we see that the matching bits in the two representations (integer and FP) correspond to the low-order bits of the integer, up to the MSB equal to 1 matching the high-order 21 bits of the fraction in the FP value

```
Int: 0b0000000000001101011001000101000001
FP:  0b01001010010101100100010100000100
```

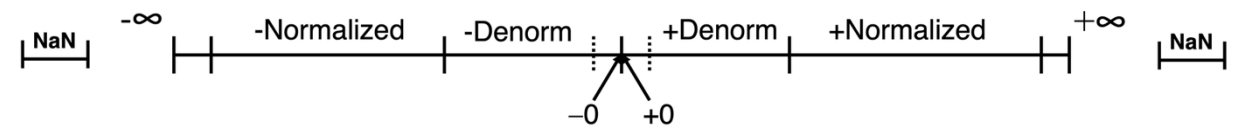


Example 2

Let's convert -10.75 to single-precision FP:

- Value is negative, so sign bit is 1
- Step 1: Convert magnitude to binary
 - $10.75_{10} \rightarrow 0b1010.11$
- Step 2: Normalize so we have a single 1 to the left of binary point
 - $1010.11 \rightarrow 1.01011 * 2^3$
- Step 3: Exponent (single precision uses a bias of 127)
 - Biased exponent = $3 + 127 = 130 \rightarrow 0b10000010$
- Step 4: Determine fraction field / Mantissa.
 - Drop implied leading 1, keep all the bits after the binary point, and pad with zeros to get 23 bits
 - $0b01011000000000000000000$
- Step 5: Put it all together!
 - $0b110000010010110000000000000000$
 - $0x0xC12C0000$
- Bonus: Check your work
 - $(-1)^1 \times 1.01011_2 \times 2^{(130-127)} = -1.34375 \times 8 = -10.75$

Single precision: $k = 8$ exp bits, $n = 23$ frac bits (32b total). `float` in C



$$V = (-1)^s * M * 2^E$$

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	$\text{exp} \neq 000...0_2$ and $\text{exp} \neq 111...1_2$	$\text{exp} = 000...0_2$
frac	$x_1x_2x_3...x_j$	$x_1x_2x_3...x_j$
Bias=	$2^{(k-1)} - 1$, for k exponent bits	$2^{(k-1)} - 1$, for k exponent bits
E=	$\text{exp} - \text{Bias}$	$1 - \text{Bias}$
M=	$1. x_1x_2x_3...x_j$ a.k.a. 1.frac	$0. x_1x_2x_3...x_j$ a.k.a. 0.frac
V=	$(-1)^s \times (1.\text{frac}) \times 2^{(\text{exp} - \text{Bias})}$	$(-1)^s \times (0.\text{frac}) \times 2^{(1 - \text{Bias})}$

Example 3

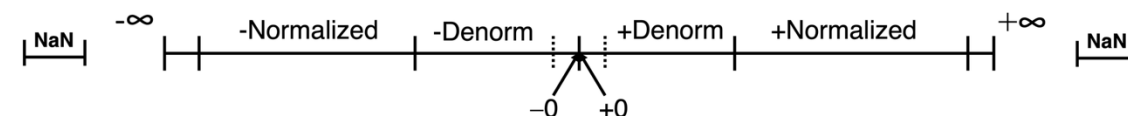
(denormalized)

Single precision: $k = 8$ exp bits, $n = 23$ frac bits (32b total). `float` in C



Let's convert -0.375×2^{-126} . This is approximately -4.408×10^{-39}

- Step 1: Sign bit is 1
- Step 2: Exponent field is all 0, which indicates a denormalized number
 - Exponent, E , is $1 - \text{Bias} = 1 - 127 = -126$
 - Field `exp` = 00000000
- Step 3: Fraction / Mantissa (no implicit leading 1)
 - $0.375 = 0.25 + 0.125 = 0b0.011$
 - Pad to 23 bits
 - Field `frac` = 01100000000000000000000
- Step 4: Assemble the FP
 - 0b1 00000000 01100000000000000000000
 - 0x80300000
- Bonus: Check our work
 - $(-1)^1 \times 0.011_2 \times 2^{-126} = -0.375 \times 1.1755 \times 10^{-38} \approx -4.408 \times 10^{-39}$



$$V = (-1)^s * M * 2^E$$

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	$\text{exp} \neq 000...0_2$ and $\text{exp} \neq 111...1_2$	$\text{exp} = 000...0_2$
frac	$x_1x_2x_3...x_j$	$x_1x_2x_3...x_j$
Bias=	$2^{(k-1)} - 1$, for k exponent bits	$2^{(k-1)} - 1$, for k exponent bits
E=	$\text{exp} - \text{Bias}$	$1 - \text{Bias}$
M=	$1.x_1x_2x_3...x_j$ a.k.a. 1.frac	$0.x_1x_2x_3...x_j$ a.k.a. 0.frac
V=	$(-1)^s \times (1.\text{frac}) \times 2^{(\text{exp} - \text{Bias})}$	$(-1)^s \times (0.\text{frac}) \times 2^{(1 - \text{Bias})}$

Topics

- Floating Point
 - Fractional binary numbers
 - Representing floating point
 - Smaller floating point
- Floating Point Arithmetic
- **Intro to Assembly Languages**

Assembly (Also known as: Assembly Language, ASM)

- Purpose of a CPU: execute instructions
- High-level programs (like in C) are split into many small instructions
- Assembly is a low-level programming language where the program instructions match a particular architecture's operations
 - Assembly is a human-readable text representation of machine code
 - Each assembly instruction is one machine instruction (usually)

Programs can be written in assembly or machine instructions

C Program (source code)

```
a = (b+c) - (d+e) ;
```

Assembly Program

```
addq %rdi, %rsi  
addq %rdx, %rcx  
subq %rcx, %rsi  
movq %rsi, %rax
```

Machine Instructions

```
0x4889D3  
0x488903  
0x53  
0x5B
```

There are many assembly languages

- Instruction Set Architecture: All programmer-visible components of a processor needed to write software for it
 - Operations the processor can execute
 - The system's state (registers, memory, program counter)
 - The effect operations have on system state
- Each assembly language has instructions that match a particular processor's Instruction Set Architecture (ISA)
- Assembly is not portable to other architectures (like C is)

Which instructions should an assembly include?

Each assembly language has its own operations

There are some obviously useful instructions:

- Add, subtract, and bit shift
- Read and write memory

But what about:

- Only run the next instruction if these two values are equal
- Perform four pairwise multiplications simultaneously
- Add two ascii numbers together ('2' + '3' = 5)

Instruction Set Philosophies

Early trend: add more instructions to do elaborate operations

Complex Instruction Set Computing (CISC)

- Handle many different types of operations
- More options for the compiler
- Complicated hardware runs more slowly

Opposite philosophy later began to dominate:

Reduced Instruction Set Computing (RISC)

- Simpler (and smaller) instruction set makes it easier to build fast hardware
- Let software do the complicated operations by composing simpler ones

Modern reality is somewhere between these two



Mainstream Instruction Set Architectures



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

PCs & Servers
(Intel & AMD processors)
[x86 Instruction Set](#)



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility ^[1]
Endianness	Bi (little as default)

Smartphones (iPhone, Android),
Apple Hardware, Raspberry Pi,
Embedded systems
[ARM Instruction Set](#)

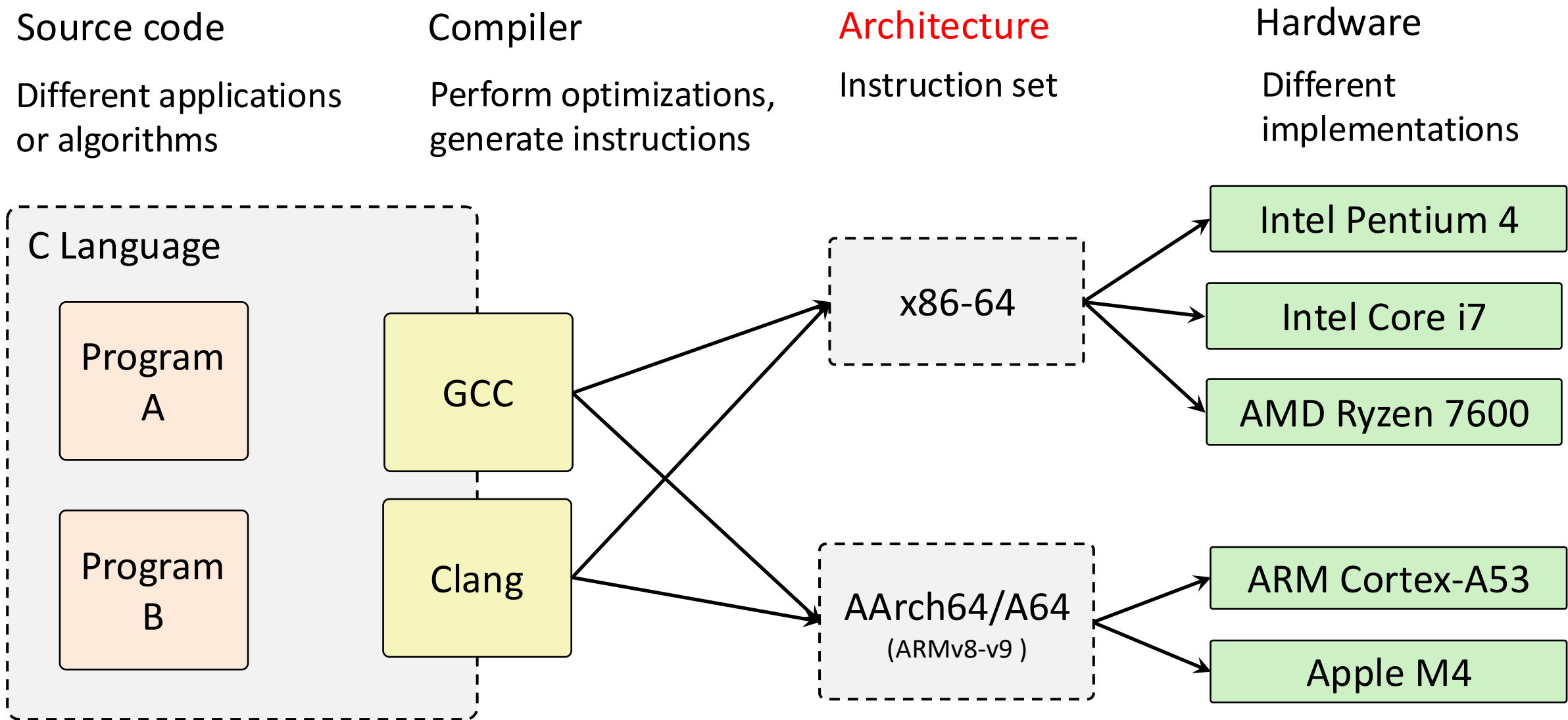


RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	6 August 2014 ^[1]
Version	unprivileged ISA 20250508, ^[2] privileged ISA 20250508 ^[2]
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little ^{[3]:9[a]}

Open-source, designed for
cloud computing, embedded
systems, academic use
[RISCV Instruction Set](#)

Instruction Set Architecture sits at software/hardware interface



Intel x86 Processors

- Dominate desktop and server market, and Windows laptops
 - No longer completely dominant in laptops though (MacOS moved to ARM)
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But only small subset encountered by normal programs
- Design evolved over time
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
 - Historical legacy has **large** impact on architecture

Transistor count



Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

How small? Today: 3nm!
< ½ the size of most viruses!

Evolution of x86 ISA

Name	Date	Transistors	Comments
8086	1978	29k	16b processor, basis for IBM PC & DOS; 1MB address space
80286	1982	134K	Elaborate (!useful) addressing; basis for IBM PC and Windows
386	1985	275K	Extended to 32b , added “flat addressing” that Linux/gcc uses
486	1989	1.9M	Improved performance; integrated FP unit into chip
Pentium	1993	3.1M	Improved performance
PentiumPro	1995	6.5M	Conditional move instructions ; big change in microarch. (P6)
Pentium II	1997	7M	Merged Pentium/MMZ + PentiumPro, MMX instructions within P6
Pentium III	1999	8.2M	Integer and floating point vector instructions (SSE); Level2 cache
Pentium 4	2001	42M	8B ints and floating point formats to vector instructions
Pentium 4E	2004	125M	Hyperthreading (able to run 2 programs simultaneously), 64b
Core 2	2006	291M	P6-like, multicore , no hyperthreading
Core i7 (Nehalem)	2008	781M	Hyperthreading + multicore, TurboBoost (run fewer cores faster)
Core i3 (Nehalem)	2010	383M+177M	GPU on second silicon die within package (at 2010 version)
Core i3, i5, i7 (Sandy Bridge)	2011	997M (i7 – 4 cores)	Cores and GPU within the same processor die
Core i3, i5, i7 (Ivy Bridge)	2012	1400M (i7 – 4 cores)	Tri-gate transistors, much lower power consumption
Xeon E7 8800 V4 (Broadwell-EX)	2016	>5690M (22 cores)	14nm technology

Backwards Compatibility

- Programs that worked on one x86 processor should keep working on the next one
 - Old programs work on new processors, which makes upgrading possible
 - Even today's x86-64 processors boot thinking they are 8086s!
- Adding powerful new features while keeping backwards compatibility is a careful balancing act
 - Backwards compatibility introduces a lot of constraints
 - May rule out “cleaner” designs that would break existing programs
 - The cause of some “surprising” aspects of the design of x86-64
 - “The x86 really isn't all that complex—it just doesn't make a lot of sense.”
— Mike Johnson (AMD's x86 architect), 1994
- Not just a hardware thing!

Machine-Level Code: Highlights

We will focus on x86-64: the current standard (also known as x64)

- The **ISA** defines the format and behavior of a machine-level program
 - Most ISAs describe the behavior of a program as if each instruction is executed in sequence, with one instruction completing before the next one begins
- Memory addresses used by a machine-level program are ***virtual addresses***
 - This provides a memory model that appears to be a very large byte array
 - Actual implementation of memory systems involves a combination of multiple hardware memories and OS software – more about this later!
- Compiler transforms abstract programs (C) into elementary instructions that the processor executes (binary format)
 - Assembly code representation is very close to machine code, but it's more readable

Machine-Level Code: Highlights

Machine code for x86-64 differs greatly from the original C code! Parts of the processor state are exposed that are normally hidden from the C programmer:

- The ***Program Counter (PC)*** indicates the address in memory of the next instruction to be executed
 - Called `%rip`
- The integer ***register file*** contains 16 named location storing 64-bit values
 - These registers hold addresses (corresponding to C pointers) or integer data
 - Some registers are used to hold temporary data (arguments, local variables, values returned by a function, etc.) while others keep track of critical parts of program state
- The ***condition code registers*** hold status information about the most recently executed arithmetic or logical instruction
 - Used to implement conditional changes in the control or data flow (needed for `if` and `while` statements)
- A set of ***vector registers*** can hold one or more integer or floating-point values