CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 6: Endianness, Floating Point Numbers

Instructors: Dan Garcia, Justin Yokota

Agenda

- Endianness
- Floating Point
- Summary

What will happen when this code is run?

- A compiler error/warning
- The code will crash
- Almost all outputs are three characters long; "hi" followed by a random third character
- Almost all outputs are three characters or longer; "hi" followed by random nonsense characters
- Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters
- All outputs are "hi"

What will happen when this code is run?

A compiler error/warning

The code will crash

Almost all outputs are three characters long; "hi" followed by a random third character

Almost all outputs are three characters or longer; "hi" followed by random nonsense characters

Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters

All outputs are "hi"

What will happen when this code is run?

- A compiler error/warning
- The code will crash
- Almost all outputs are three characters long; "hi" followed by a random third character
- Almost all outputs are three characters or longer; "hi" followed by random nonsense characters
- Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters
- All outputs are "hi"

Agenda

- **Endianness**
- Floating Point
- Summary

Endianness

- So far, we've discussed how we store values in binary
    - Ex. We write **`int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}.`**
    - If we assume **`&i == 0xF000 0000`**, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			

- If we do **`i[1]`**, the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address as an integer. This corresponds to 0x73206E65, so it works.
- What happens if we do **`((char*) i)[2]`**?

Endianness

- So far, we've discussed how we store values in binary
  - Ex. We write `int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}`.
  - If we assume `&i == 0xF000 0000`, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	?	?	?	?	?	?	?	?	?	?	?	?

- If we do `((char*) i)[2]`, the compiler adds 0xF000 0000 + 2 * sizeof(char) = 0xF000 0002, then takes the one byte starting from that address as a char. This yields something (since there is data there), but what it yields depends on how the subbytes of our 4-byte int get stored in memory. The way things get stored is known as the endianness of the system.

Analogy: How do we write dates?

- You want to communicate three dates to your friend:
  - October 7, 1999 (your birthday)
  - June 30, 2022 (today)
  - February 3, 2022 (When the US Formatting and Localization team will meet, per xkcd)
- Challenge: You can only write down nine numbers (no slashes or dashes).
- Option 1: Write the date in year-month-day:
  - 99 10 07 22 06 30 22 02 03
- Option 2: Write the date in day-month-year:
  - 07 10 99 30 06 22 03 02 22
- Option 3: Write the date in month-day-year:
  - 10 07 99 06 30 22 02 03 22
- Does it matter which one you use?
  - No, as long as your friend knows which one you're using

Analogy: How do we write dates?

- 99 10 07 22 06 30 22 02 03 (year-month-day) goes from largest-size to smallest-size. This is known as **big-endian** order.
- 07 10 99 30 06 22 03 02 22 (day-month-year) goes from smallest-size to largest-size. This is known as **little-endian** order.
- 10 07 99 06 30 22 02 03 22 (month-day-year) is neither big-endian nor little-endian. We generally call all such orderings **middle-endian**.
- In both versions, each group of three numbers is a date. As long as you read groups of 3 numbers together as a date (and read it the same way you wrote it), you'll extract the correct date.
- The only difference is if I ask something like "What's the third number written down"

Big-Endian

- In a big-endian system, we write the Most Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F

- If we do `((char*) i)[2]`, the compiler adds `0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`, then takes the one byte starting from that address as a char. This yields 0x61.
- If we do `printf((char*) i);` we would interpret this block of memory as a character array, so we’d get 0x64 0x72 0x61 …, which when converted to ASCII yields “drags net”

Little-Endian

- In a little-endian system, we write the Least Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- If we do **`((char*) i)[2]`**, the compiler adds **`0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`**, then takes the one byte starting from that address as a char. This yields 0x72.
- If we do **`printf((char*) i);`** we would interpret this block of memory as a character array, so we’d get 0x67 0x61 0x72 …, which when converted to ASCII yields “garden sod”

Big Endian: Example

- Note: In either case, if we read **`i[1]`**, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Big Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0x73 0x20 0x6E 0x65. Since this is big-endian, we combine them with the first address being the MSB, so we get 0x73206E65, which is the correct result

Little Endian: Example

- Note: In either case, if we read **`i[1]`**, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Little Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0x65 0x6E 0x20 0x73. Since this is little-endian, we combine them with the first address being the LSB, so we get 0x73206E65, which is the correct result

Endianness

- All endiannesses work as long as you’re consistent. It only affects cases where you either transfer to a new endianness (different systems can use different endiannesses), or when you split/merge a block of data into smaller/larger chunks.
  - This comes up a lot when working with unions, since a union is designed to interpret the same block of data in different ways.
- Officially, C defines this to be undefined behavior; you’re not supposed to do this.
  - With unions, you’re supposed to use that block as only one of the components
- In practice, undefined behavior is a great way to hack someone’s program, so it ends up popping up in CS 161. It also ends up showing up when you do memory dumps (see homework).

Endianness Use Cases

- Big Endian is commonly used in networks (e.g. communicating data between computers).
- Little Endian is commonly used within the computer
    - The default endianness for C, RISC-V, x86, etc. In general, you'll be working with little-endian systems when programming

Endianness

- Some systems are bi-endian (allows you to switch endianness if you want)
- Others use a completely different endianness (ex. PDP-11 does some weird order)
- There are mild benefits for either endianness (one might let you read important data faster than the other), but ultimately, we use different endiannesses because computer programmers are bad at coordinating standards, and once a standard is made, backwards compatibility is an issue.
- The **`hton`** and **`ntoh`** function sets convert from host endianness to network endianness and vice versa.

Agenda

- Endianness
- **Floating Point**
- Summary

Recall: Goals for memory storage systems

- Any system/convention to store data ideally has the following properties:
- 1: The values that can be represented are relevant to the subject at hand
    - Different subject matters need different value sets, so we often have several different options with various trade-offs
- 2: The system is efficient, in terms of:
    - 2a: Memory use: As many possible bitstrings correspond to valid data values as possible
        - Ideal goal: Make it so that every bitstring corresponds to a different, valid data value
    - 2b: Operator cost: The operators we define are simple to make in circuitry, and require few transistors
        - Ideal goal: If we can reuse an existing circuit, then we don't need to add any additional circuitry!
    - Other aspects depending on the system; for now, we'll focus on these two
- 3: The system is intuitive for a human to understand
    - Often goes together with small operator cost, as well as adaptability to other systems
    - If a system's too complicated, no one will use it. Though if you manage to abstract away a lot of the complexity, you can get away with a less-intuitive system…

IEEE-754 Floating Point Numbers

- An encoding scheme to store arbitrary real numbers
- It starts off with a simple structure
  - Able to store relevant values
  - Relatively simple circuitry
  - Analogous to something we already use
- After that, the standard gets messy, with minor “optimizations” to improve relevance and useability at the cost of “intuition”
  - Denorms, Infinities, NaNs

Goals: What numbers do we want to store?

- IEEE-754 was designed as a “universal” number system, in order to encourage consistency (don’t want to deal with translating between different conventions)
- As such, we want to handle:
  - Really big numbers (ex. Avogadro’s number = $6.022*10^{23}$)
  - Really small numbers (ex. Planck’s constant = $6.626*10^{-34}$)
- Notable observation: When dealing with large numbers, we generally don’t care about small absolute differences
  - Ex. If you’re dealing with a 10 kg bag of rice, you don’t care about the extra weight of a few grains of rice
  - If you’re measuring the weight of atoms, you probably care a lot about the extra weight of a few grains of rice

Goals: What numbers do we want to store?

- IEEE-754 was designed as a “universal” number system, in order to encourage consistency (don’t want to deal with translating between different conventions)
- As such, we want to handle:
    - Really big numbers (ex. Avogadro’s number = $6.022*10^{23}$)
    - Really small numbers (ex. Planck’s constant = $6.626*10^{-34}$)
- Goal: Attempt to store numbers to a high relative precision
    - If we can’t store a number exactly, make sure that we can store a number that’s accurate to within 0.001%

Analogous system: Scientific Notation

- Goal: Attempt to store numbers to a high relative precision
  - If we can't store a number exactly, make sure that we can store a number that's accurate to within 0.001%
- Scientific notation already does this!
- Ex: $6.022*10^{23}$ stores Avogadro's number using:
  - 1 bit (+/-)
  - 4 decimal digits (6022) (Called the mantissa or the significand)
  - An exponent (23)
- 4 decimal digits means we get precision up to 10^{-3} (numbers we store are accurate to within 0.1%)
- If we set limits on the exponent to +/- 500, we can store very small AND very large numbers using 2 numbers and a sign bit

Floating Point System: V0

- To store a floating point number, we need to save:
  - Sign bit: + or -
  - Mantissa: Positive integer with no leading zeros
  - Exponent: Positive or negative integer
- Since we're working with binary, we'll use scientific notation with a base of 2
  - Ex. We'd save something like 0b1.011 * 2^5
    - Note: In binary, keep in mind that numbers after the binary point are smaller powers of 2. So 0b1.011 = $2^0+2^{-2}+2^{-3}$=1+0.25+0.125 = 1.375
- How to store mantissa?
  - Mantissa is just a positive integer, so we can save it as an unsigned number
- How to store exponent?

Floating Point System V0: Exponent storage options

- Ultimately, our storage scheme depends on how the exponent gets used
  - Goal: Minimize the amount of complexity needed to implement floating point numbers on a system that already has unsigned and signed numbers.
- Three major types of operators that are relevant to both floats and ints
  - Add/Subtract (Ex. $5*10^4+6*10^8$, $1.234*10^4-1.233*10^4$)
    - Float addition affects exponent in complicated ways; often around the max of the two exponents, but can be different.
  - Multiply/Divide (Ex. $5*10^4 x 6*10^8$, $1.234*10^4/1.233*10^4$)
    - Affects exponent in complicated ways; often around the sum/difference of the two exponents, but can be different.
  - Comparisons (Ex. $5*10^4<6*10^8$, $1.234*10^4>=1.233*10^4$)
    - Compare the exponent first, and if the two are equal, compare the mantissa
    - Similar to how we compare integers (compare the top digit, if equal, compare the next digit)
- Conclusion: When dealing with exponent: The most common operator run on them directly is comparators

Floating Point System: V0

- To store a floating point number, we need to save:
  - Sign bit: + or -
  - Mantissa: Positive integer with no leading zeros
  - Exponent: Positive or negative integer
- Since we're working with binary, we'll use scientific notation with a base of 2
  - Ex. We'd save something like 0b1.011 * 2^5
    - Note: In binary, keep in mind that numbers after the binary point are smaller powers of 2. So 1.011 = $2^0+2^{-2}+2^{-3}$=1+0.25+0.125 = 1.375
- How to store mantissa?
  - Mantissa is just a positive integer, so we can save it as an unsigned number
- How to store exponent?
  - Exponent gets stored with a biased encoding. Further, we store this data in the order sign-exponent-mantissa so that we can reuse the comparator circuit from sign-magnitude numbers.
  - Bias is often set to $-(2^{n-1}-1)$ for an n-bit exponent, to have about equal positive/negative exponents.

Floating Point System: V1

- A floating point number uses x bits for the exponent, and y bits for the mantissa (with x,y specified as parameters). For this slide, we'll use as an example a system with 3 exponent bits (bias of -3) and 4 mantissa bits

S XXX MMMM

- This represents the number $(-1)^{S} * 0bM.MMM * 2^{(0bXXX+(-3))}$

Floating Point V1: Example

- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 10.875 to a V1 float
- Step 1: Write the number in binary
  - 0b1010.111 = 0b1.010111000… * 2^3
- Step 2: Determine Sign/Exponent/Mantissa
  - Sign: Positive → 0
  - Exponent: 3-(-127) = 130 → 0b1000 0010
  - Mantissa: 1010 1110 0000 0000 0000 000
- Step 3: Concatenate
  - 0b0100 0001 0101 0111 0000 0000 0000 0000
  - 0x41570000

Floating Point V1: Example

- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 0xC3CC0000 as a V1 float to decimal
  - Sign: 1 -> Negative
  - Exponent: 0b1000 0111 → 135-127 = 8
  - Mantissa: 0b1001 1000… → 0b1.001 1000 = $1+2^{-3}+2^{-4}$
  - $(1+2^{-3}+2^{-4})*2^8 = 2^8+2^5+2^4=304 \rightarrow -304$
- Convert 0x00000000 as a V1 float to decimal
  - Sign: 0 -> Positive
  - Exponent: 0b0000 0000 → 0-127 = -127
  - Mantissa: 0b0000 0000 0000 0000 0000 000 → 0
  - $0*2^{-127}=0$

Optimization 1: The implicit 1

- Note that our mantissa is guaranteed to not have any leading zeros
    - If we wanted to write $0.234*10^5$, we'd instead write it as $2.340*10^4$
- In binary, every digit is only either 1 or 0. Since the MSB can't be 0, it must therefore be 1.
    - If the first bit will always be 1, we don't need to store it!
- Therefore, we can save 1 bit (or alternatively add another bit of precision) to the mantissa by not including the MSB of our mantissa
    - This is known as the implicit 1, and the resulting mantissa is a "normalized" number.

Floating Point System: V2

- A floating point number uses x bits for the exponent, and y bits for the mantissa (with x,y specified as parameters). For this slide, we'll use as an example a system with 3 exponent bits (bias of -3) and 4 mantissa bits

S XXX MMMM

- This represents the number $(-1)^{S} * 0b1.MMMM * 2^{(0bXXX+(-3))}$

Floating Point V2: Example

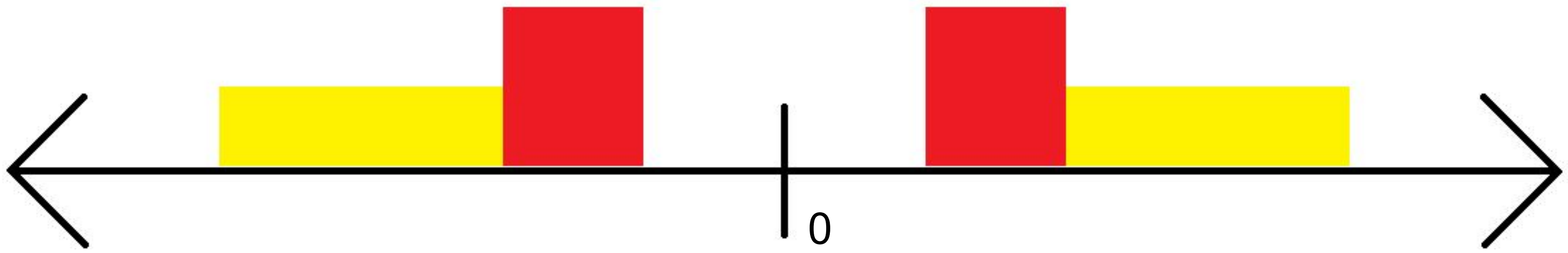
- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 10.875 to a V2 float
- Step 1: Write the number in binary
  - 0b1010.111 = 0b1.010 1110 00… * 2^3
- Step 2: Determine Sign/Exponent/Mantissa
  - Sign: Positive → 0
  - Exponent: 3-(-127) = 130 → 0b1000 0010
  - Mantissa: 0101 1100 0000 0000 0000 000
- Step 3: Concatenate
  - 0b0100 0001 0010 1110 0000 0000 0000 0000
  - 0x412E0000

Floating Point V2: Example

- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 0x00000000 as a V2 float to decimal
  - Sign: 0 -> Positive
  - Exponent: 0b0000 0000 -> 0-127 = -127
  - Mantissa: 0b0000… → 0b1.000…
  - $1*2^{-127}$
- Convert 0x00000001 as a V2 float to decimal
  - Sign: 0 → Positive
  - Exponent: 0b0000 0000 → 0-127 = -127
  - Mantissa: 0b0000…1 → 0b1.000…1 = $1+2^{-23}$
  - $(1+2^{-23})*2^{-127}=2^{-127}+2^{-150}$

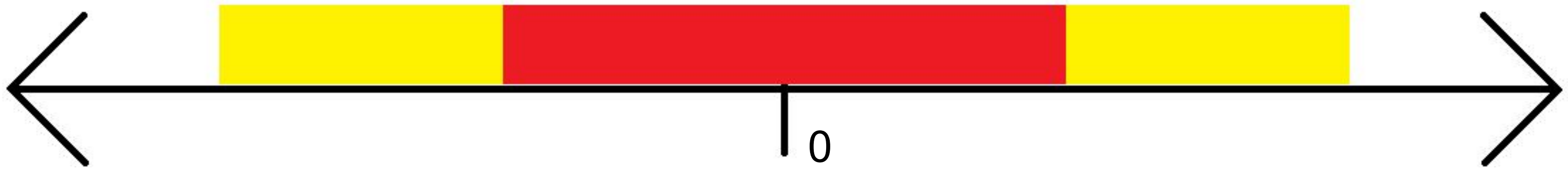
Problems with the implicit 1: Underflow

- The smallest number (in absolute value) we can represent is 2^{-127}
  - We can't represent 0
- The second smallest number is $2^{-127}+2^{-150}$
  - There's a big gap in the set of representable numbers (10 million times bigger than the gaps between consecutive numbers)
- This is known as an underflow; the result of computation gets too small to be represented.
- In the below diagram, the red and yellow box represents the range (width of the box) and density (height of the box) of exponent 0b000…0 and 0b000…1, respectively. Note that each exponent "halves" the distance to 0.



Solution: Denormalized numbers

- Solution: Change the behavior of exponent 0b000…0 so that its range extends to 0
- How to do that?
  - If we use the same step size as exponent 0b000…1, then we can get exactly to 0
  - When dealing with these numbers, use an implicit 0 instead of an implicit 1, so it doesn't overlap with exponent 0b000…1
- Ends up losing precision at small numbers (so there's still underflow), but at least it's not a sudden cliff drop.



Floating Point System: V3

- A floating point number uses x bits for the exponent, and y bits for the mantissa (with x,y specified as parameters). For this slide, we'll use as an example a system with 3 exponent bits (bias of -3) and 4 mantissa bits

S XXX MMMM

- If the exponent bits are nonzero, then it represents the number $(-1)^S * 0b1.MMMM * 2^{(0bXXX+(-3))}$
- If the exponent bits are all zero, then it instead represents $(-1)^S * 0b0.MMMM * 2^{(0b000+(-3)+1)}$ (also known as a denormalized number, or "denorm")

Floating Point V3: Example

- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 0x00000000 as a V3 float to decimal
  - Sign: 0 -> Positive
  - Exponent: 0b0000 0000. All zeros, so exponent meaning is 0-127+1 = -126
  - Mantissa: 0b0000… -> 0b0.000…
  - $0*2^{-126}$ = 0 (We can represent 0!)
- Convert 0x00000001 as a V3 float to decimal
  - Sign: 0 -> Positive
  - Exponent: 0b0000 0000 -> 0-127+1 = -126
  - Mantissa: 0b0000…1 -> 0b0.000…1 = $0+2^{-23}$
  - $(0+2^{-23})*2^{-126}=2^{-149}$ (Much closer to 0)

Optimization 2: Dealing with infinity, division by zero

- We get a fairly large range using this (10^{38} with 8 exponent bits), but we can still exceed the maximum float.
- We also should deal with “1 / 0”
- Ideally, we’d like to include an “infinity” value to handle these cases
- While we’re at it, let’s add some values for error handling, that don’t correspond to any actual number
- What bitstrings do we use for this
    - Since we want an infinity, let’s use the largest exponent for this

Floating Point System: Final Version

- An IEEE-754 standard binary floating point number uses x bits for the exponent, and y bits for the mantissa (with x,y specified as parameters). For this slide, we'll use as an example a system with 3 exponent bits (bias of -3) and 4 mantissa bits

S XXX MMMM

- If the exponent bits are nonzero and not all ones, then it represents the number $(-1)^{S}$*0b1.MMMM * $2^{(0bXXX+(-3))}$
- If the exponent bits are all zero, then it instead represents $(-1)^{S}$*0b0.MMMM * $2^{(0b000+(-3)+1)}$
- If the exponent bits are all ones, then:
    - If the mantissa is all zeros, it's either ∞ or -∞ (depending on the sign bit)
    - If the mantissa isn't all zeros, then it's a NaN (which means "Not a Number")

Floating Point System: Final Version

- An IEEE-754 standard binary floating point number uses x bits for the exponent, and y bits for the mantissa (with x,y specified as parameters). For this slide, we'll use as an example a system with 8 exponent bits (bias of -127) and 23 mantissa bits

SXXX XXXX XMMM MMMM MMMM MMMM MMMM MMMM

- If the exponent bits are nonzero and not all ones, then it represents the number $(-1)^{S}$ * 0b1.MMMM… * $2^{(\text{0bXXXX XXXX}+(-127))}$
- If the exponent bits are all zero, then it instead represents $(-1)^{S}$*0b0.MMMM… * $2^{(\text{0b0}+(-127)+1)}$
- If the exponent bits are all ones, then:
    - If the mantissa is all zeros, it's either ∞ or -∞ (depending on the sign bit)
    - If the mantissa isn't all zeros, then it's a NaN (which means "Not a Number")

Floating Point Final: Example

- For this example, let's assume we're working with a system with 8 exponent bits (bias of -127) and 23 mantissa bits.
- Convert 0xFF80 0000 as an IEEE-754 float to decimal
  - Sign: 1 -> Negative
  - Exponent: 0b1111 1111. All ones, so we're dealing with a special case
  - Mantissa: 0b0000… -> All zeros
  - $-\infty$
- Convert 0xFF80 0001 as an IEEE-754 float to decimal
  - Sign: 1 -> Negative
  - Exponent: 0b1111 1111. All ones, so we're dealing with a special case
  - Mantissa: 0b0000…1 -> Not all zeros
  - NaN

IEEE-754: More notes on IEEE-754

- The IEEE-754 standard specifies five aspects:
- Arithmetic formats
  - The binary format, as discussed here. NaNs are supposed to use their mantissa to help with debugging purposes, but officially, you only need two types of NaNs (for quiet failures and signalling failures, respectively)
  - A decimal format which uses similar rules, but with decimal floating point instead. Due to the difficulty of writing decimal numbers in a binary encoding, the storage system of this format is *significantly* more complicated, and out of scope.
- Interchange formats

IEEE-754: Interchange Formats

- IEEE-754 specifies certain combinations of exponent and significand bits with special names:
- Single precision (also known as float):
  - 8 exponent bits, -127 bias, 23 mantissa bits
- Double precision (also known as double):
  - 11 exponent bits, -1023 bias, 52 mantissa bits
- Quad precision:
  - 15 exponent bits, 112 mantissa bits
- Octuple precision:
  - 19 exponent bits, 236 mantissa bits
- Half precision:
  - 5 exponent bits, 10 mantissa bits

IEEE-754: Interchange Formats

- C’s float and double correspond to single-precision and double-precision floating point numbers, respectively
- Float:
  - Has enough precision for 6-7 decimal digits
  - Max value around 10^{38}
- Double:
  - Has enough precision for ~13 decimal digits
  - Max value around 10^{308}
  - Despite its name, it is generally considered the “default” floating point representation

IEEE-754: More notes on IEEE-754

- The IEEE-754 standard specifies five aspects:
- Arithmetic formats
  - The binary format, as discussed here. NaNs are supposed to use their mantissa to help with debugging purposes, but officially, you only need two types of NaNs (for quiet failures and signalling failures, respectively)
  - A decimal format which uses similar rules, but with decimal floating point instead. Due to the difficulty of writing decimal numbers in a binary encoding, the storage system of this format is *significantly* more complicated, and out of scope.
- Interchange formats
- Rounding Rules

IEEE-754: Rounding Rules

- Several different options, but the most common one is “round to the nearest value, and break ties to the even number (last bit 0)
  - Ex. To round 14.5 to the nearest 2, go to 14, since 14 is closer to 14.5 than 16.
  - Ex. To round 15 to the nearest 2, go to 16, since 16 ends in more 0 bits than 14
- This is intended to be deterministic, but in practice, it’s rather unpredictable
- Generally a good idea not to rely too much on rounding behavior.
- Note that rounding happens after every operator in a string of operations, so we get slightly less precise results every step
  - This is why we use so many bits for the mantissa in doubles, and why doubles are the standard float option; we generally need more precision than range/memory
  - Often different orders of doing operations yields different results even if they should be mathematically equivalent. The rate at which precision gets lost (and how to order operations to minimize precision loss) is covered in Math 128A

IEEE-754: More notes on IEEE-754

- The IEEE-754 standard specifies five aspects:
- Arithmetic formats
  - The binary format, as discussed here. NaNs are supposed to use their mantissa to help with debugging purposes, but officially, you only need two types of NaNs (for quiet failures and signalling failures, respectively)
  - A decimal format which uses similar rules, but with decimal floating point instead. Due to the difficulty of writing decimal numbers in a binary encoding, the storage system of this format is *significantly* more complicated, and out of scope.
- Interchange formats
- Rounding Rules
- Operations

IEEE-754: Operations

- Requires conversion to/from integer, arithmetic, comparisons, abs/negate, floor/ceiling, etc.
  - Two new operators are: square root and fused multiply-add (a*b+c done in one step)
- Additionally recommends native support for other math operations (e^x, 2^x, 10^x, log x, distance formula, trig functions, hypertrig functions, arctrig functions)
- Most of these functions are available through the <math.h> library

IEEE-754: More notes on IEEE-754

- The IEEE-754 standard specifies five aspects:
- Arithmetic formats
  - The binary format, as discussed here. NaNs are supposed to use their mantissa to help with debugging purposes, but officially, you only need two types of NaNs (for quiet failures and signalling failures, respectively)
  - A decimal format which uses similar rules, but with decimal floating point instead. Due to the difficulty of writing decimal numbers in a binary encoding, the storage system of this format is *significantly* more complicated, and out of scope.
- Interchange formats
- Rounding Rules
- Operations
- Exception Handling

IEEE-754: Exception Handling

- Defines five classes of exceptions, along with recommended behavior:
- Invalid operation
    - Ex. sqrt(-1.0)
    - By default, returns a NaN (quiet)
- Division by zero
    - By default, returns infinity
- Overflow
    - By default, returns infinity
- Underflow
    - Returns a denorm. Note that we consider any result using a denorm to suffer from underflow (due to precision loss)
- Inexact value
    - Any math that yields a number that can't be exactly represented, like 1.0/3
    - Rounds to a representable number according to the rounding rule.