CS 33:
Introduction to Computer Organization

Week 5

Admin

- Homework 4 “due” Friday, November 6th
  - Hopefully assigned sometime before then.
- Lab 3: Smashing lab “due” Friday, November 13th.

Agenda

- Floating Point
- Floating Point
- Floating Point
- Floating Point
- Optimization

So far

- We have one way of interpreting binary. Bit 'n' contributes 0 or 2^n to the total value of the number. This means a 32-bit number can represent 2^32 values.
- As you can probably guess, this alone is not sufficient.
- For example, without some modification, this won't allow for the representation of non-integer numbers.

So far

- With this in mind, let's try extending our existing binary representation by adding a decimal point. Consider a 5-bit representation.
- Instead of 00000 to 11111, we'll have 000.00 to 111.11.
- The bit immediately to the left of the point contributes 0 or 2^0 while the bit immediately to the right contributes 0 or 2^{-1}. The next bit to the right contributes 0 or 2^{-2}
- This is known as “fixed point” binary.

Fixed Point Binary

- Consider an 8-bit fixed point binary value where the point is between bits 3 and 4 (0000.0000 to 1111.1111)
- 0101.0110
- = $2^2 + 2^0 + 2^{-2} + 2^{-3} = 5.175$
- All right. That seems marginally effective.
- Because we have 4 fractional bits, we can express a granularity of 2^{-4} or .0625.
- What's the cost of doing things this way?

Fixed Point Binary

- Originally, an 8-bit signed number represents values from -128 to 127
- Now our 8-bit signed number represents the following values:
- 1000.0000 to 0111.1111 or:
- -8 to 7.9375
- With this 8 bit signed representation, in order to represent up to fractional digits, we've reduced our overall range from 256 to approximately 16.

Fixed Point Binary

- Well, that was a dummy example anyway. What if we tried this in a real system that uses 32 bit values?
- For each bit we use for a fractional bit, we reduce the approximate range of values by 2.
- This means that if we use 5 bits to achieve a granularity of 2^{-5} (.03125), the range of values drops to about 2^{27} (approximately 134 million)

Fixed Point Binary

- We paid a huge cost to in order to represent only 5 fractional digits.
- ...and we still can't even represent .1 precisely!
- .00011 = .09375
- .00100 = .125
- The fact is, regardless of what we do, there will be “simple” decimal fractional numbers that we can't represent in binary.

Fixed Point Binary

- With this finite, discrete binary representation, we will always have limited precision and limited range. However, perhaps there is a representation that loses precision in an acceptable way.
- Consider an unsigned 32-bit fixed point where 8 bits are used for the fractional bits. This means we can represent an unsigned number (nearly) as large as 2^{24} with a granularity of 2^{-8}.

Fixed Point Binary

- However, if I'm representing a number in the range of 2^{23}, how interested are we in a value of 2^{-8}?
- Given $8.398 * 10^6$, what is $8.398 * 10^6 + 1$?
- It's just going to be $8.398 * 10^6$ simply because 1 is entirely overshadowed by the other number.
- That's the beauty of scientific notation right?

Fixed Point Binary

- The goal for floating point is essentially to be able to represent base 2 scientific notation
- Generic:
  - F * B^E
- Base 10:
  - 4.15 * 10^3
  - F = 4.15 and E = 3
- Base 2:
  - 1.011_2 * 2^3
  - F = 1.011_2, E = 3

Floating Point

- Warning: Floating point is bizarre and upsetting.
- Each binary configuration is split into three components:
  - s: Sign bit
  - e: Exponential part
  - f: Fraction part
- [s][exponent][fractional]
- 0 1101 100
- Seems normal so far.

Floating Point: Normalized

- There are four different “forms” of how to interpret the numbers. The main form is “normalized”
- A number is “normalized” if the exponent part is non-zero and not all 1s.
- [s][exponent][fractional]
- 0 1101 100
- $V = (-1)^{s} * 2^{e\text{-Bias}} * 1.f_{2}$

Floating Point: Normalized

- $V = (-1)^{s} * 2^{e\text{-Bias}} * 1.f_2$
- First of all, what is “1.f”?
- f is the fractional field, in our example, it was:
  - 100
- $1.f = 1.100_2$
- This value is interpreted as fixed point.
- $1.100 = 2^0 + 2^{-1} = 1.5$

Floating Point: Normalized

- $V = (-1)^{s} * 2^{e\text{-Bias}} * 1.f_{2}$
- Secondly, what is the “Bias”? Why not just have it be 2^{e}.
- Our exponent field ranges from 0000 and 1111. If we just did 2^{e}, we wouldn't be able to represent negative exponents. We need a bias.
- According to the ancient texts, if k = the number of bits that are used to represent the exponent field:
  - $\text{Bias} = 2^{k-1} - 1$

Floating Point: Normalized

- Finally, the one simple part, s is a sign bit.
- Thus, what is this value?
- [s][exponent][fractional]
- 0 1101 100
- $V = (-1)^{s} * 2^{e\text{-Bias}} * 1.f_{2}$
- $V = (-1)^{0} * 2^{13-7} * 1.100$
- $= 2^{6} * 1.5 = 96$

Floating Point: Normalized

- This is “normalized”, because the fractional field always begins with a 1
  - If f = 1010, the fractional field is 1.1010
  - If f = 0000, the fractional field is 1.0000
- The reason for this is so that there is only one way to represent a single value. We don't want:
  - V = 2^4 * .11 where e = 11 and f = 11
  - V = 2^5 * .011 where e = 12 and f =011

Floating Point: Normalized

- However, what's the minimum value we can represent like this?
- The minimum value is limited by the smallest exponent that e can be. The f will always contribute a value that is >= 1 and < 2.
- Maybe we want to represent smaller numbers somehow?

Floating Point: Denormalized

- We can do this by interpreting the bits differently depending on the configuration.
- A number is considered denormalized when e = 0.
- [s][exponent][fractional]
- 0 0000 100
- $V = (-1)^{s} * 2^{1\text{-Bias}} * .f_2$

Floating Point: Denormalized

- [s][exponent][fractional]
- 0 0000 100
- $V = (-1)^{s} * 2^{1\text{-Bias}} * .f_{2}$
- Consider the two key differences:
  - The exponent is now $2^{1\text{-Bias}}$ instead of $2^{e\text{-Bias}}$.
  - The fractional part is now .f instead of 1.f (hence “denormalized”)
- We fix the exponent part to allow for a denormalized fractional part.

Floating Point: Denormalized

- 0 0000 100
- $V = (-1)^{s} * 2^{1-\text{Bias}} * .f_2$
- Bias = 7
- $V = 2^{1-7} * .100 = 2^{-6} * .5 = .0078125$
- Note that even though e = 0000, the exponent part is not 0 – Bias, it's 1-Bias. Thus, the minimum exponent part is technically $- 2^{k-1} + 2$.

Floating Point: Denormalized

- 0 0000 000
- In this representation, what is the maximum positive denormalized value?

Floating Point: Denormalized

- 0 0000 000
- In this representation, what is the minimum positive normalized value?
- 0 0001 000
- $V = 2^{1\text{-Bias}} * 1.000 = 2^{-6} * (1)$
- $= 2^{-6}$

Floating Point: Denormalized

- 0 0000 000
- In this representation, what is the maximum positive denormalized value?
- 0 0000 111
- $V = 2^{1\text{-Bias}} * .111 = 2^{-6} * (2^{-1} + 2^{-2} + 2^{-3})$
- $= 2^{-6} * (1 – 2^{-3})$
- Note: Denormalized numbers in this range have a granularity of 2^{-9}. Thus, We don't use $2^{e\text{-Bias}}$ where e = 0 in order to maintain this gradual transition from normalized to denormalized (ie MIN_NORM – MAX_DENORM = 2^{-9}).

Floating Point: Normalized

- Useful tip:
- $f(n, k) = 2^n + 2^{n+1} + 2^{n+2} + \ldots + 2^{n+k} = \Sigma\ 2^i$ where i is from n to n+k
- $= 2^{n+k+1} – 2^n$
- For example: $2^0 + 2^1 + 2^2 + 2^3 + \ldots\ 2^n = 2^{n+1} - 1$

Floating Point: Normalized

- $f(n,k) = 2^{n} + 2^{n+1} + 2^{n+2} + \ldots + 2^{n+k} = ?$
- $2*f(n, k) = 2^{n+1} + 2^{n+2} + 2^{n+3} + \ldots + 2^{n+k+1}$
- $2*f(n, k) – f(n,k) = -2^{n} + 2^{n+1} – 2^{n+1} + 2^{n+2} – 2^{n+2} + \ldots 2^{n+k} – 2^{n+k} + 2^{n+k+1}$
- $f(n, k) = 2^{n+k+1} – 2^{n}$

Floating Point: Segue

- In class the rationale for why we have floating point values was overflow.
- To be clear, regardless of how we represent numbers, overflow is unavoidable in a representation with limited bits.
- However, what happens when an integer overflows? Consider a 4-bit signed with wrap semantics:
  - 0111 + 1 = 1000
  - 7 + 1 = -8

Floating Point: Segue

- With integer overflow, you get an incredibly wrong answer that looks sort of legitimate.
- Maybe with floating point, we can handle overflow by producing a less wrong answer that's easy to identify.
- One way of doing this, is to allow for the representation of infinity. If a value overflows, it becomes infinity.

Floating Point: Infinty

- Here's how floating point does it.
- If e = 111...111 and f = 0, then the number is either + or – infinity:
- 0 1111 000 = +Inf
- 1 1111 000 = -Inf
- Now when you have an overflow, the answer is still sort of correct.
  - ex. 7 + 1 = infinity
  - Eh, good enough.

Floating Point: NaN

- While we're at it, let's allow floating point to represent numbers that aren't numbers. We'll cleverly call them not-a-numbers (NaN)?
- If e= 111...111 and f != 0, then the value is NaN
- 0 111 1000 = NaN
- 0 111 0101 = NaN
- These numbers are not a numbers.
- Up is down. Left is right.

Floating Point: Special Values

- Being able to represent infinity means that if you overflow/underflow, rather than wrapping around to the lowest/highest value which is extremely incorrect behavior, it becomes + or – infinity, which more accurately describes the outcome that you were looking for.
- NaN IS useful for representing imaginary numbers and other expressions that result in values that aren't quite meaningful as a normal number.

Floating Point: Special Values

- Notes on NaN:
  1. NaN's are infectious: any operation that deals with NaN result in NaN. Ex. 1 + NaN = NaN.
  2. Despite the fact that two NaN's can have the same binary representation, (NaN == NaN) = false.
     - Ex:
     - float nan = 0.0/0.0;
     - nan == nan will evaluate to false.

Floating Point: Special Values

- Didn't I just say that any any operation that deals with NaN results in NaN? Shouldn't NaN == NaN = NaN? It should. But it doesn't. Logical operations are the exception.
- Last year, Professor Eggert suggested that maybe there should be a NaB (Not-a-boolean)

Floating Point: Special Values

- We managed to avoid this in integer representation but now we have +0 and -0.
- If there are f fractional bits, we have 2^{f+1} -2 ways of representing NaN. Seems a little excessive.

IEEE 754 Floating Point

- For float:
  - e is 8-bits
  - Bias is $2^7 - 1 = 127$
  - Range of exponent field is [$2^8 - 2 - 127$, $1 - 127$ [127, -126].
  - Notice the asymmetry around 0, except unlike two's complement there are more positive values (for maximum confusion and torment).
  - f is 23 bits

IEEE 754 Floating Point

- For double:
  - e is 11-bits
  - Bias is $2^{10} – 1 = 1023$
  - Range of exponent field is [-1022, 1023]
  - f is 52 bits.

Floating Point: Segue

- How do you do floating point arithmetic?
- Frankly, it's complicated and not really the point of going over floating point.
- The point of this section is to teach you of the overall behavior and interesting properties so that you understand what can happen when you use floating point (inexact representations, underflow, etc).

Homework Problem 2.88

- *And now… a selection from:*

  Practice Problem 2.88
- Consider two 9-bit floating-point representations based on the IEEE floating point format.
- Format A:
  - 1 sign bit.
  - k = 5 exponent bits.
  - n = 3 fractional bits
  - [0][00000][000] to [1][11111][111]
- Format B:
  - 1 sign bit.
  - k = 4 exponent bits
  - n = 4 fractional bits
  - [0][0000][0000] to [1][1111][1111]

Homework Problem 2.88

- Below you are given some bit patterns in Format A and your task is to convert them to the closest value in Format B. If necessary, round up (ie. to + infinity).

Format A	[0][00000][000]
Bits	Value
0 10110 011	
1 00111 010	
1 11100 000	
0 10111 100	

Format A	[0][0000][0000]
Bits	Value

Homework Problem 2.88

- The general strategy in converting between formats is as follows. Say we're converting from notation A to B.
- Express the number in scientific notation:
- $V = 2^E * F_A$
- $E = e_A – Bias_A$
- Attempt to find a way to represent E in format B.
- $E = e_A – Bias_A = e_B – Bias_B$
- $e_B = E + Bias_B$

Homework Problem 2.88

- e_B = E + $Bias_B$
- Then, attempt to match the the bit configuration of F_A with the number of fraction bits in B.
- Yes. “attempt”
- Let's “do” an example.

Homework Problem 2.88

- Convert 0 10110 011 (Format A) to Format B (4 exponent bits, 4 fractional bits).
- First, what are the biases?

Homework Problem 2.88

- Convert 0 10110 011 (Format A) to Format B (4 exponent bits, 4 fractional bits).
- First, what are the biases?
- $\text{Bias}_A = 2^{5-1} – 1 = 15$
- $\text{Bias}_B = 2^{4-1} – 1 = 7$
- $V = 2^{e\text{-BiasA}} * 1.011 = 2^{22-15} * (1 + 2^{-2} + 2^{-3}) = 176$
- $E = 22-15 = 7$
- $7 = e_B – \text{Bias}_B$
- $7 = e_B – 7$
- $e_B = 14 = 1110$

Homework Problem 2.88

- Convert 0 10110 011 (Format A) to Format B (4 exponent bits, 4 fractional bits).
- e_B = 14 = 1110
- F_A = 1.011
- F_B = 1.0110
- 176 can be represented exactly in format B:
  - 0 1110 0110

Homework Problem 2.88

- Below you are given some bit patterns in Format A and your task is to convert them to the closest value in Format B. If necessary, round up (ie. to + infinity).

Format A	[0][00000][000]
Bits	Value
0 10110 011	176
1 00111 010	
1 11100 000	
0 10111 100	

Format A	[0][0000][0000]
Bits	Value
0 1110 0110	176

Homework Problem 2.88

- Convert 1 00111 010 (Format A, Bias_A = 15) to Format B (4 exponent bits, 4 fractional bits, Bias_B = 7).
- V = $-2^{e-\text{BiasA}} * 1.010 = -2^{7-15} * (1 + 2^{-2}) = -2^{-8} * (5/4)$

  $= -5/1024$
- E = - 8
- $-8 = e^B - \text{Bias}^B$

  $= e^B - 7$
- $e_B = -1 = ?$
- We can't represent -1 as the exponent.
- Panic.

Homework Problem 2.88

- This method attempts to convert the number to normalized Format B. However, this cannot occur because e_B cannot be a negative number.
- Thus, we have to attempt to use denormalized.
- $V= -2^{-8} * (1 + 2^{-2})$
- In denormalized Format B, the exponent is 1-Bias_B or -6. Let's rewrite V so that it matches this form:
- $V = -2^{-8} * (1 + 2^{-2}) = -2^{-6} * (2^{-2} + 2^{-4})$

Homework Problem 2.88

- $V = -2^{-6} * (2^{-2} + 2^{-4})$
- $V = -2^{1-\text{BiasB}} * 1.0101$
- Format B:
- 1 0000 0101

Homework Problem 2.88

- Below you are given some bit patterns in Format A and your task is to convert them to the closest value in Format B. If necessary, round up (ie. to + infinity).

Format A	[0][00000][000]
Bits	Value
0 10110 011	176
1 00111 010	-5/1024
1 11100 000	
0 10111 100	

Format A	[0][0000][0000]
Bits	Value
0 1110 0110	176
1 0000 0101	-5/1024

Homework Problem 2.88

- Convert 1 11100 000 (Format A, Bias_A = 15) to Format B (4 exponent bits, 4 fractional bits, Bias_B = 7).
- $V = -2^{e\text{-BiasA}} = -2^{28-15} = -2^{13}$
- E = 13
- $13 = e_B – \text{Bias}_B$

  $= e_B -7$
- e_B = 20 = [1]0100 (?)
- We can't represent 20 with 4 bits.
- Panic.

Homework Problem 2.88

- Because Format A has 5 exponent bits, its range exceeds that of Format B. This proposed number of -2^{13} is too negative to be represented in Format B.
- If we round down, this means we round to -infinity.
- However, our goal is to round up, which means we have to round to the most negative number that can be represented in format B.
- Format B: 1 1110 1111 = -248

Homework Problem 2.88

- Below you are given some bit patterns in Format A and your task is to convert them to the closest value in Format B. If necessary, round up (ie. to + infinity).

Format A	[0][00000][000]
Bits	Value
0 10110 011	176
1 00111 010	-5/1024
1 11100 000	-8192
0 10111 100	

Format A	[0][0000][0000]
Bits	Value
0 1110 0110	176
1 0000 0101	-5/1024
1 1110 1111	-248

Homework Problem 2.88

- Convert 0 10111 100 (Format A, Bias_A = 15) to Format B (4 exponent bits, 4 fractional bits, Bias_B = 7).
- $V = 2^{e\text{-BiasA}} = 2^{23-15} * 1.1 = 2^{8} * (3/2)$
- E = 8
- $8 = e_B – \text{Bias}_B$

  $= e_B -7$
- e_B = 15 = 1111 (wait, that's NaN or Inf)
- We can't represent 15 with 4 bits.
- Panic.

Homework Problem 2.88

- Same deal as before, except this time, now the number is too great to represent in Format B.
- If we round down, this means we round to the highest positive number that can be represented by Format B.
- However, our goal is to round up so we round to infinity.
- Format B: 0 1111 0000 = inf

Homework Problem 2.88

- Below you are given some bit patterns in Format A and your task is to convert them to the closest value in Format B. If necessary, round up (ie. to + infinity).

Format A	[0][00000][000]
Bits	Value
0 10110 011	176
1 00111 010	-5/1024
1 11100 000	-8192
0 10111 100	384

Format A	[0][0000][0000]
Bits	Value
0 1110 0110	176
1 0000 0101	-5/1024
1 1110 1111	-248
0 1111 0000	infinity

Floating Point: Rounding Error

- When we convert an int to a float, is it possible to lose precision (ie, an integer value cannot be represented exactly)?
- The book says yes.
- The cow says moo.
- Intuitively, we can see that float has 23 fractional bits while ints have 32 bits. Therefore, floats cannot provide the precision required by ints, suggesting that there are some ints that cannot be represented. However, how can we prove it?

Floating Point: Rounding Error

- One way is to find the point at which we can no longer represent all integers in a float.
- In other words, what's the smallest integer that cannot be represented exactly in floating point?
- What's a simple way of approaching this?

Floating Point: Rounding Error

- You can no longer represent all integers when the least significant bit of the fractional field contributes more than 1 to the total.
- Consider a case where you have you have 5 fractional bits.
- If the current exponent E is 6, then the least significant bit will contribute 2
  - $V1 = 2^6 * 1 = 64$
  - $V2 = 2^6 * (1 + 2^{-5}) = 66$
- Thus, when e – Bias > [No. of fractional bits], you cannot represent all integers.

Floating Point: Rounding Error

- For real IEEE floating point you have:
  - 8 fractional bits
  - 23 exponent bits
  - Bias = $2^7 – 1 = 127$
- Thus (N is the number of fractional bits):
- $2^{e\text{-Bias}} > 2^N$
- $= 2^{e-127} > 2^{23}$
- $= e-127 > 23$
- $e > 150$

Floating Point: Rounding Error

- Let's take e = 151, our numbers are of the form:
- $V = 2^{151-127} * 1.f = 2^{24} * 1.f$
- $2^{24} = 16777216$
- The next representable number is:
- $V = 2^{24} * (1 + 2^{-23}) = 2^{24} + 2$
- = 16777218
- We can't represent 16777217.
- Signed ints range from -2147483648 to 2147483647. 16777218 is well within this range. Thus, we can't represent all “ints” with a float.

Floating Point: Rounding

- Rounding modes:
  - Round up
  - Round down
  - Round to zero
  - Round to even
- Round to even is something of a misnomer. It means the following:
  - Round to the representable number whose value is closest.
  - If directly between two representable numbers, round to the value whose least significant bit is 0.

Floating Point: Rounding

- Consider a floating point with the following representation:
- [s][exponent][fractional]
- 0 0000 0000
- Consider the decimal value 71 and -71 This would have to be represented by:
- 71: 0 1101 0001 [11]
- -71: 1 1101 0001 [11]

Floating Point: Rounding

- Round up:
  - 71: 0 1101 0001 [11] => 0 1101 0010 = 72
  - -71: 1 1101 0001 [11] => 1 1101 0001 = -68
- Round down:
  - 71: 0 1101 0001 [11] => 0 1101 0001 = 68
  - -71: 1 1101 0001 [11] => 1 1101 0010 = -72
- Round to zero:
  - 71: 0 1101 0001 [11] => 0 1101 0001 = 68
  - -71: 1 1101 0001 [11] => 1 1101 0001 = -68

Floating Point: Rounding

- Round to even:
- 71: 0 1101 0001 [11]
  - The actual value is 71. The closest representable values are:
  - 0 1101 0001 = 68
  - 0 1101 0010 = 72
  - Thus, round to 72.
- This is an example of round to closest. How can we change this example so that it will have to follow the round to even convention (ie equidistant from the nearest representable values)?

Floating Point: Rounding

- Round to even:
- 70: 0 1101 0001 [1]
  - The actual value is 70. The closest representable values are:
  - 0 1101 0001 = 68
  - 0 1101 0010 = 72
  - Since there is a tie, we round to the value whose least significant bit (or LSB as the kids are saying) is 0. In this instance, we round up to 72.

Floating Point: Rounding

- Round to even:
- 66: 0 1101 0000 [1]
  - The actual value is 66. The closest representable values are:
  - 0 1101 0001 = 68
  - 0 1101 0000 = 64
  - Since there is a tie, we round to the value whose least significant bit is 0. In this instance, we round down to 64.
- Round to even tends to be the default rounding system.

Floating Point: Operations

- This general strategy of:
  1. Convert to scientific notation
  2. Coerce the number into the desired form
  3. Convert back to float
- ...applies to other operations dealing with floating point.

Floating Point: Operations

- As mentioned before, the point of teaching floating point is not so that you can be an expert at implementing floating point operations
- It is so that you understand what can and will happen when you use floating points in order to write more informed code.
- The better you understand what happens to your code before a machine executes it, the better decisions you'll be able to make when writing it.
- That said, here are some ways of doing basic operations.

Floating Point: Convert Integer to Floating Point

- Say you have the floating point representation of:
- s e f
- 0 0000 000
- How do you convert a decimal integers into a floating point?
- Ex: 37 to floating point.

Floating Point: Convert Integer to Floating Point

- 0 0000 000

1. Convert 37 to integer binary/sum of powers of 2.
   - $37 = 100101 = 2^5 + 2^2 + 2^0$
2. Factor out the most significant power of 2.
   - $37 = 2^5 (1 + 2^{-3} + 2^{-5})$
3. Determine e and f.
   - 5 = e – Bias, e = 12 = 1100
   - f = 001 [01] (recall the most significant 1 is implicit)
4. Apply rounding:
   - Ex. Round to even: 0 1100 001 = 36

Floating Point: Adding two floats

- Say we have the representation:
- s e f
- 0 000 0000
- V1 = 0 101 1011 = 6.75
- V2 = 0 010 0100 = .625
- V1 + V2 in floating point?

Floating Point: Adding two floats

- 0 000 0000
- V1 = 0 101 1011 = 6.75
- V2 = 0 010 0100 = .625
- Convert to scientific notation:
  - $V1 = 2^{2} * (1 + 2^{-1} + 2^{-3} + 2^{-4})$
  - $V2 = 2^{-1} * (1 + 2^{-2})$
- Multiply to make the exponents equal
  - $V2 = 2^{-1} * (1 + 2^{-2}) = (2^{3} / 2^{3}) * 2^{-1} * (1 + 2^{-2})$
  - $= 2^{2} * (2^{-3} + 2^{-5})$

Floating Point: Adding two floats

- 0 000 0000
- V1 = 0 101 1011 = 6.75 = $2^2 * (1 + 2^{-1} + 2^{-3} + 2^{-4})$
- V2 = 0 010 0100 = .625 = $2^2 * (2^{-3} + 2^{-5})$
- Add:
  - $2^2 * (1 + 2^{-1} + 2^{-3} + 2^{-4}) + 2^2 * (2^{-3} + 2^{-5})$
  - $= 2^2 * (1 + 2^{-1} + 2^{-3} + 2^{-3} + 2^{-4} + 2^{-5})$
  - $= 2^2 * (1 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5})$
  - 2 = e – Bias, e = 5 = 101
  - f = 1101 [1]
- Round:
  - V1 + V2 = 0 101 1101 [1] =RtE=> 0 101 1110 = 7.5

Floating Point: Multiplying two floats

- 0 000 0000
- V1 = 0 101 1011 = 6.75
- V2 = 0 010 0100 = .625
- V1 * V2 in floating point?
- Convert to scientific notation:
  - V1 = $2^{2} * (1 + 2^{-1} + 2^{-3} + 2^{-4})$
  - V2 = $2^{-1} * (1 + 2^{-2})$

Floating Point: Multiplying two floats

  - $V1 = 2^{2} * (1 + 2^{-1} + 2^{-3} + 2^{-4})$
  - $V2 = 2^{-1} * (1 + 2^{-2})$
- Multiply, keeping the powers of 2:
  - $V1*V2 = 2^{1} * (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6})$
  - $= 2^{1} * (2 + 2^{-4} + 2^{-5} + 2^{-6})$
- Normalize:
  - $= 2^{2} * (1 + 2^{-5} + 2^{-6} + 2^{-7})$
  - 2 = e – Bias, e = 5
  - f = 0000[111]
- Round:
  - V1*V2 = 0 101 0000 [1111] =RtE=> 0 101 0001 = 4.25

Floating Point: Properties

- For +/-/*:
  - Commutative: x + y == y + x
  - NOT Associative: (x + y) + z != x + (y + z)
- Why isn't it associative? Consider float x, float y, and float z.
  - x =16777216.0
  - y = 1
  - z = -16777216.0

Floating Point: Properties

- x =167777216.0, y = 1, z = -167777216.0
- (x + y) + z
  - (x + y) = 167777217. However, recall, 167777217 is not representable in float. As a result, we round to even.
  - 167777217 : 0 10010111 000...000 [1]. Thus, round to 0 10010111 000...000 or 167777216.
- (x + y) + z = 167777216 + -167777216 = 0

Floating Point: Properties

- x =167777216.0, y = 1, z = -167777216.0
- x + (y + z)
  - (y + z) = -167777215. Since integers between 16777216 and -16777216 are not representable, this number is representable without any rounding.
- x + (y + z) = 16777216 + -16777215 = 1

Floating Point: Why denormalized?

- In class, we presented the following rationale for the “hugely” “controversial” denormalized numbers:
- In particular, we don't want this:
  - a – b = 0 && a != b
- ...to ever be true.
- Why could this be true if we only had normalized numbers?

Floating Point: Why denormalized?

- Consider:
- s e f
- 0 000 0000
- NOTE: The following is a thought experiment. This is NOT how floating point is interpreted.
- Assume we're using a denormalized-less representation. Thus the formula: $V = (-1)^{s} * 2^{(e-Bias)} * 1.f$ is true even if e = 0.
- In order for us to be able to represent 0, let 0 000 0000 be a special case to mean 0.

Floating Point: Why denormalized?

- 0 000 0000
  - Bias = 3
- 0 000 0000 = 0
- 0 000 0001 = 2^{-3} * $(1 + 2^{-4})$ = $2^{-3} + 2^{-7}$
- 0 000 0010 = 2^{-3} * $(1 + 2^{-3})$ = $2^{-3} + 2^{-6}$
- Note, $2^{-3} + 2^{-7}$ is absolutely the smallest positive value we can represent.
- 0 000 0010 – 0 000 0001 = $2^{-3} + 2^{-6} - 2^{-3} - 2^{-7}$
  - = 2^{-6}

Floating Point: Why denormalized?

- 0 000 0000
  - Bias = 3
- Because 2^{-6} is closer to 0 than it is to $2^{-3} + 2^{-7}$, 2^{-6} or 0 000 0010 – 0 000 0001 will be rounded to zero.
- 0 000 0010 – 0 000 0001 == 0 && 0 000 0010 != 0 000 0001
- Whoops.

(Compiler) Optimization

- Consider the optimization example mentioned in class. Each of these are defined in the same module, foo.c.

```
int S_len(struct S *s)
{
  return s->len;
}

void addvec(struct S *s, int* sum)
{
  *sum = 0;
  for(int i = 0; i < S_len(s); i++)
  {
    *sum += s->vec[i];
  }
}
```

```
struct S
{
  int len;
  int vec[];
};
```

(Compiler) Optimization

- Consider the optimization example mentioned in class.
- There were two types of basic “hoistings” that were mentioned as helping out with this code.

```
void addvec(struct S *s, int* sum)
{
  *sum = 0;
  for(int i = 0; i < S_len(s); i++)
  {
    *sum += s->vec[i];
  }
}
```

(Compiler) Optimization

- 1. Move S_len(s) out of the loop. The length of the vector isn't going to change over the course of the loop.
- 2. Replace *sum in the loop with a temporary variable. Dereferencing is pointer is (more) expensive because there is no “add MEM, MEM” instruction.

```
void addvec(struct S *s, int* sum)
{
  int temp = 0;
  int len = S_len(s);
  for(int i = 0; i < len; i++)
  {
    temp += s->vec[i];
  }
  *sum = temp
}
```

(Compiler) Optimization

- However as mentioned in class, despite how smart the compiler can be, it is only willing to make one of these hoistings.
- Which is the compiler willing to hoist?
- Why is the compiler unwilling to hoist the other one?

(Compiler) Optimization

- The compiler is generally willing to hoist S_len(s) out of the loop.
- However, the compiler is unwilling to hoist the pointer out of the loop into a temporary variable because of aliasing. What if for some i, &s->vec[i] is an alias for int * sum?
- In this case, if you're at *sum += s->vec[0] and sum is really an alias for &s->vec[1], s->vec[0] must be added to *sum to be added to s->vec[1].
- Lets compile this code to see what it looks like.

(Compiler) Optimization

- addvec (compiled with “gcc -O2 -fno-inline foo.c”):

```
0x4004e0 <addvec>:              mov    %rdi,%rcx
0x4004e3 <addvec+3>:            movl   $0x0,(%rsi)
0x4004e9 <addvec+9>:            xor    %edx,%edx
0x4004eb <addvec+11>:           jmp    0x4004fc <addvec+28>
0x4004ed <addvec+13>:           nopl   (%rax)
0x4004f0 <addvec+16>:           movslq %edx,%rax
0x4004f3 <addvec+19>:           add    $0x1,%edx
0x4004f6 <addvec+22>:           mov    0x4(%rcx,%rax,4),%eax
0x4004fa <addvec+26>:           add    %eax,(%rsi)
0x4004fc <addvec+28>:           mov    %rcx,%rdi
0x4004ff <addvec+31>:           callq  0x4004d0 <S_len>
0x400504 <addvec+36>:           cmp    %eax,%edx
0x400506 <addvec+38>:           jl     0x4004f0 <addvec+16>
0x400508 <addvec+40>:           repz retq
```

(Compiler) Optimization

- addvec (compiled with “gcc -O2 -fno-inline foo.c”):

```
0x4004e0 <addvec>:                mov    %rdi,%rcx
0x4004e3 <addvec+3>:              movl   $0x0,(%rsi)
0x4004e9 <addvec+9>:              xor    %edx,%edx
0x4004eb <addvec+11>:             jmp    0x4004fc <addvec+28>
0x4004ed <addvec+13>:             nopl   (%rax)
0x4004f0 <addvec+16>:             movslq %edx,%rax
0x4004f3 <addvec+19>:             add    $0x1,%edx
0x4004f6 <addvec+22>:             mov    0x4(%rcx,%rax,4),%eax
0x4004fa <addvec+26>:             add    %eax,(%rsi)
0x4004fc <addvec+28>:             mov    %rcx,%rdi
0x4004ff <addvec+31>:             callq  0x4004d0 <S_len> ?
0x400504 <addvec+36>:             cmp    %eax,%edx
0x400506 <addvec+38>:             jl     0x4004f0 <addvec+16>
0x400508 <addvec+40>:             repz retq
```

- Is that right?

(Compiler) Optimization

- Why didn't the compiler hoist S_len(s)?

(Compiler) Optimization

- Why didn't the compiler hoist S_len(s)?
- int *sum could also be an alias for something that S_len does.
- In particular, the compiler knows that S_len(s) fetches s->len and for all it knows, int * sum is an alias for &s->len.
  - addvec(s, &s->len)
- Lets define a new alias-free version of addvec.

(Compiler) Optimization

```
int addvec2(struct S *s)
{
  int sum = 0;
  for(int i = 0; i < S_len(s); i++)
  {
    sum += s->vec[i];
  }
  return sum;
}
```

- What does the assembly look like now?

(Compiler) Optimization

```
addvec2 (gcc -O2 -fno-inline test.c)

0x400510 <addvec2>:          mov    %rdi,%rsi
0x400513 <addvec2+3>:        callq  0x4004d0 <S_len>
0x400518 <addvec2+8>:        xor    %edx,%edx
0x40051a <addvec2+10>:       xor    %ecx,%ecx
0x40051c <addvec2+12>:       jmp    0x400528 <addvec2+24>
0x40051e <addvec2+14>:       xchg   %ax,%ax
0x400520 <addvec2+16>:       add    0x4(%rsi,%rdx,4),%ecx
0x400524 <addvec2+20>:       add    $0x1,%rdx
0x400528 <addvec2+24>:       cmp    %edx,%eax
0x40052a <addvec2+26>:       jg     0x400520 <addvec2+16>
0x40052c <addvec2+28>:       mov    %ecx,%eax
0x40052e <addvec2+30>:       retq
```

- That's more like it.

(Compiler) Optimization

- The compiler is going to do its best to compile code based on what it knows.
- This means it has to prepare for corner cases.
- However, now that addvec2 only deals with local variables to save and accumulate the sum, the compiler can reasonably assume that there is no aliasing between the variables in struct *s and the local variables in addvec2.

(Compiler) Optimization

- Is it possible that the compiler is... wrong?
- int sum is a variable local to addvec2, which means it is/should be stored on the stack.
- struct* s is a pointer to a struct that was previously allocated.
- Without relying on undefined behavior, it does not make sense for the location of the struct to overlap with that of int sum.

(Compiler) Optimization

- In a real system, it is likely that your program will be split into modules. Let's split test.c up now we have test.c, helper.c, and struct_s.h

test.c:

```
#include "struct_s.h"

int S_len(struct S * s);

int addvec3(struct S *s)
{
  int sum = 0;
  for(int i = 0; i < S_len(s); i++)
  {
    sum += s->vec[i];
  }
  return sum;
}
```

helper.c:

```
#include "struct_s.h"

int S_len(struct S *s)
{
  return *s->len;
}
```

struct_s.h:

```
struct S
{
  int* len;
  int vec[];
};
```

(Compiler) Optimization

- Let's compile this with “gcc -c -O2 -fno-inline test.c” and examine addvec3.

(Compiler) Optimization

```
addvec3:
  0x0 <addvec2>:         push   %r12
  0x2 <addvec2+2>:       mov    %rdi,%r12
  0x5 <addvec2+5>:       push   %rbp
  0x6 <addvec2+6>:       xor    %ebp,%ebp
  0x8 <addvec2+8>:       push   %rbx
  0x9 <addvec2+9>:       xor    %ebx,%ebx
  0xb <addvec2+11>:      jmp    0x1b <addvec2+27>
  0xd <addvec2+13>:      nopl   (%rax)
  0x10 <addvec2+16>:     movslq %ebx,%rax
  0x13 <addvec2+19>:     add    $0x1,%ebx
  0x16 <addvec2+22>:     add    0x8(%r12,%rax,4),%ebp
  0x1b <addvec2+27>:     mov    %r12,%rdi
  0x1e <addvec2+30>:     callq  0x23 <addvec2+35>
  0x23 <addvec2+35>:     cmp    %eax,%ebx
  0x25 <addvec2+37>:     jl     0x10 <addvec2+16>
  0x27 <addvec2+39>:     mov    %ebp,%eax
  0x29 <addvec2+41>:     pop    %rbx
  0x2a <addvec2+42>:     pop    %rbp
```

(Compiler) Optimization

```
addvec3:
  0x0 <addvec2>:          push   %r12
  0x2 <addvec2+2>:        mov    %rdi,%r12
  0x5 <addvec2+5>:        push   %rbp
  0x6 <addvec2+6>:        xor    %ebp,%ebp
  0x8 <addvec2+8>:        push   %rbx
  0x9 <addvec2+9>:        xor    %ebx,%ebx
  0xb <addvec2+11>:       jmp    0x1b <addvec2+27>
  0xd <addvec2+13>:       nopl   (%rax)
  0x10 <addvec2+16>:      movslq %ebx,%rax
  0x13 <addvec2+19>:      add    $0x1,%ebx
  0x16 <addvec2+22>:      add    0x8(%r12,%rax,4),%ebp
  0x1b <addvec2+27>:      mov    %r12,%rdi
  0x1e <addvec2+30>:      callq  0x23 <addvec2+35>
  0x23 <addvec2+35>:      cmp    %eax,%ebx
  0x25 <addvec2+37>:      jl     0x10 <addvec2+16>
  ...
```

- Sonuva...

(Compiler) Optimization

- Why is the compiler back to not being able to hoist S_len(s)?

(Compiler) Optimization

- Why is the compiler back to not being able to hoist S_len(s)?
- Now, the compiler knows that it will be calling S_len but unlike before, test.c doesn't contain the source code for S_len. Therefore, the compiler doesn't know what S_len does.
- The compiler has to assume that S_len might have side effects. Ex:

```
int s_len(struct S * s)
{
  return s->len--;
}
```

(Compiler) Optimization

- Ex:

```
int S_len(struct S * s)
{
  return s->len--;
}
```

- If this were true, for each iteration of the loop, S_len(struct S *s) is modifying the length.
- As a result, it would be incorrect for the compiler to generate code that computes the “length” once.
- Also, what was up with all of those extra push/pops?

(Compiler) Optimization

- In some ways, “optimizing” means writing code that the compiler can optimize.
- Knowing what the compiler can optimize is a function of knowing what the compiler knows when it is going to compile your code.
- The compiler is pessimistic. If it doesn't explicitly know something, it can't make any assumptions about it, lest it generates incorrect code.
- Of course, the exception being...

(Compiler) Optimization

- ...aliasing
- Compilers are often allowed to assuming that aliases of different types won't be an issue.
- What we're hoping with this section is for you to get an intuition as to what the compiler is and isn't going to do.
- If you can predict this, you can write better code based on your understanding of how the compiler is going to treat your creation/child.

End of

The Fifth Week

-Five Weeks Remain-