CS 61c: Great Ideas in Computer Architecture

Floating Point Numbers, Measuring Performance

Instructor: Alan Christopher

July 8, 2014

Review

- **Compiler** converts a single HLL file into a single assembly file
- **Assembler** removes pseudo-instructions, converts what it can into machine language, and creates a checklist for linker (relocation table)
  - Resolves addresses by making 2 passes (for forward references)
- **Linker** combines several object files and resolves absolute addresses
  - Enable separate compilation and use of libraries
- **Loader** loads executable into memory and begins execution

Review/MT Practice

Discuss with Neighbors:(previous midterm question)
In one word each, name the most common producer and consumer of the following items. Choose from *Linker*, *Loader*, *Compiler*, *Assembler*, *Programmer*

(item)	This is the output of:	This is the input to:
`bne $t0, $s0, done`	Compiler	Assembler
`char *s = "hello world"`		
`app.o string.o`		
`firefox`		

Review/MT Practice

Discuss with Neighbors:(previous midterm question)
In one word each, name the most common producer and consumer of the following items. Choose from *Linker*, *Loader*, *Compiler*, *Assembler*, *Programmer*

(item)	This is the output of:	This is the input to:
`bne $t0, $s0, done`	Compiler	Assembler
`char *s = "hello world"`	Programmer	Compiler
`app.o string.o`	Assembler	Linker
`firefox`	Linker	Loader

Outline

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and unsigned integers

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and unsigned integers
  - Characters (ASCII)

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and unsigned integers
  - Characters (ASCII)
  - Instructions & Addresses

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and unsigned integers
  - Characters (ASCII)
  - Instructions & Addresses
- How do we encode?
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^{23})
  - Very small numbers (e.g. 7.21×10^{-34})
  - "Special" numbers (e.g. ∞)

Number Rep Revisited

- Given one word (32 bits), what can we represent so far?
  - Signed and unsigned integers
  - Characters (ASCII)
  - Instructions & Addresses
- How do we encode?
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^{23})
  - Very small numbers (e.g. 7.21×10^{-34})
  - "Special" numbers (e.g. ∞)
- Floating Point!

Goals of Floating Point

- Support a wide range of values
  - Both very small and very large

Goals of Floating Point

- Support a wide range of values
  - Both very small and very large
- Keep as much *precision* as possible
  - Not equivalent to accuracy

Goals of Floating Point

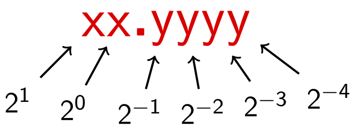
- Support a wide range of values
  - Both very small and very large
- Keep as much *precision* as possible
  - Not equivalent to accuracy
- Help programmer with errors in real arithmetic
  - Support $\pm\infty$, Not-a-Number (`NaN`), exponent overflow and underflow

Goals of Floating Point

- Support a wide range of values
  - Both very small and very large
- Keep as much *precision* as possible
  - Not equivalent to accuracy
- Help programmer with errors in real arithmetic
  - Support $\pm\infty$, Not-a-Number (`NaN`), exponent overflow and underflow
- Keep encoding that is somewhat compatible with integer representations
  - e.g. 0 in FP is the same as 0 in two's complement
  - Can use the same comparator operator for floats as for signed integers (sign and magnitude, not two's complement)

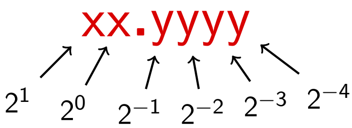
Fractions in Base 2

- **"Binary Point" like decimal point signifies boundary between integer and fractional parts:**
- Example 6-bit representation:



Fractions in Base 2

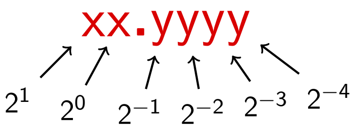
- **"Binary Point" like decimal point signifies boundary between integer and fractional parts:**
- Example 6-bit representation:



- Example: $10.1010_2 = 1 \times 2 + 1 \times \frac{1}{2} + 1 \times \frac{1}{8} = 2.625_{10}$

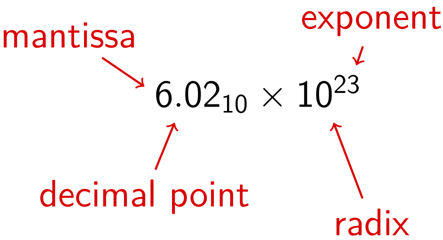
Fractions in Base 2

- **"Binary Point" like decimal point signifies boundary between integer and fractional parts:**
- Example 6-bit representation:



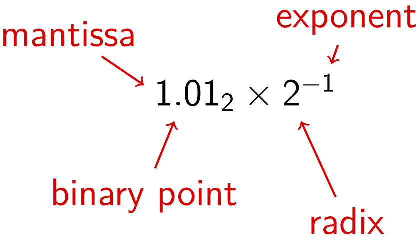
- Example: $10.1010_2 = 1 \times 2 + 1 \times \frac{1}{2} + 1 \times \frac{1}{8} = 2.625_{10}$
- This 6-bit binary point format can represent numbers between 0 (00.0000_2) and 3.9375 (11.1111_2)

Scientific Notation (Decimal)



- *Normalized form*: exactly one non-zero digit to the left of decimal point
- Multiple ways of representing 10^{-9} if we don't insist of normalizing, e.g.
    - Normalized: 1.0×10^{-9}
    - Not normalized: 10.0×10^{-10}, 0.1×10^{-8}

Scientific Notation (Binary)



- Computer arithmetic that supports this format is called *floating point*, due to the "floating" nature of the binary point
  - `float` and `double` types in C

Translating to and from Scientific Notation

- Consider the number $1.011_2 \times 2^4$

Translating to and from Scientific Notation

- Consider the number $1.011_2 \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_2 = 22_{10}$

Translating to and from Scientific Notation

- Consider the number $1.011_2 \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_2 = 22_{10}$
- For negative exponents, shift decimal to the left
  - $1.011_2 \times 2^{-2} \rightarrow 0.01011_2 = 0.34375_{10}$

Translating to and from Scientific Notation

- Consider the number $1.011_2 \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_2 = 22_{10}$
- For negative exponents, shift decimal to the left
  - $1.011_2 \times 2^{-2} \rightarrow 0.01011_2 = 0.34375_{10}$
- Go from ordinary number to scientific notation by shifting until normalized
  - $1101.001_2 \rightarrow 1.101001_2 \times 2^3$

Translating to and from Scientific Notation

- Consider the number $1.011_2 \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_2 = 22_{10}$
- For negative exponents, shift decimal to the left
  - $1.011_2 \times 2^{-2} \rightarrow 0.01011_2 = 0.34375_{10}$
- Go from ordinary number to scientific notation by shifting until normalized
  - $1101.001_2 \rightarrow 1.101001_2 \times 2^3$
- Just like base 10 (if you're short a few fingers)

Floating Point Encoding I

- Use normalized, base 2 scientific notation:

$$\pm 1.xxx\ldots x_2 \times 2^{yyy\ldots y_2}$$

Floating Point Encoding I

- Use normalized, base 2 scientific notation:

$$\pm 1.xxx\ldots x_2 \times 2^{yyy\ldots y_2}$$

- Split 32-bit word into 3 fields:

  - S represents sign (1 if negative, 0 otherwise)
  - Exponent field represents the base's exponent
  - Mantissa field represents the scientific notation's mantissa *except* for the leading 1.

The Exponent Field

- Use biased notation
  - Read exponent as unsigned, but with bias of -127
  - Defines -127 through 128 as 0b00000000 through 0b11111111
  - Exponent 0 is represented as 0b01111111 $= 127_{10}$

The Exponent Field

- Use biased notation
  - Read exponent as unsigned, but with bias of -127
  - Defines -127 through 128 as 0b00000000 through 0b11111111
  - Exponent 0 is represented as 0b01111111 $= 127_{10}$
- To encode in biased notation, subtract the bias (add 127), then encode in unsigned:
  - $1 \rightarrow 128 \rightarrow$ 0b10000000
  - $127 \rightarrow 254 \rightarrow$ 0b11111110

Floating Point Encoding II

$$(-1)^{S} \times (1.\text{Mantissa}) \times 2^{(\text{Exponent} - 127)}$$

- Note the implicit 1 in front of the significand
  - Ex: 0b0 01111111 10000000000000000000000 is read as $1.1_2 = 1.5$, NOT $0.1_2 = 1.5$
  - Gives us some extra precision by avoid duplicate representations

Exponent Comparison

- Which is smaller (closer to $-\infty$)?
  - 0 or 10^{-127}
  - 10^{-126} or 10^{-127}
  - -10^{-127} or 0
  - -10^{-126} or -10^{-127}

Exponent Comparison

- Which is smaller (closer to $-\infty$)?
  - 0 or 10^{-127}
  - 10^{-126} or 10^{-127}
  - -10^{-127} or 0
  - -10^{-126} or -10^{-127}
- Notice: When positive, a smaller exponent takes us closer to $-\infty$, but when negative, the opposite happens
  - Just like with sign and magnitude
  - Can use sign+magnitude comparisons to sort floating point numbers
  - This is a big reason why we prefer bias to two's complement inside of floats

Outline

Administrivia

- Reminder: You are in (almost) complete control of how you implement project 1.
  - If you don't like how the skeleton does something, feel free to throw it out
  - Don't ask questions about what something in the skeleton is "supposed" to do. It's supposed to do whatever you want it to.
  - Do, however, feel free to ask if a given approach is sane or not
- The rest of this week's lectures are particularly difficult for students (historically).
  - Get an extra shot of espresso in your morning coffee
  - Don't be afraid to ask questions, everyone else will be confused with you

Outline

Representing Very Small Numbers

- So, uhhh, what about zero?

Representing Very Small Numbers

- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$

Representing Very Small Numbers

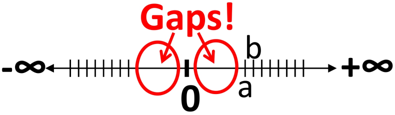
- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$
  - *Special Case*: Exponent and mantissa all zero $\Rightarrow 0$

Representing Very Small Numbers

- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$
  - *Special Case*: Exponent and mantissa all zero $\Rightarrow 0$
  - Two zeros! But at least 0x0 $==$ 0 like in integers

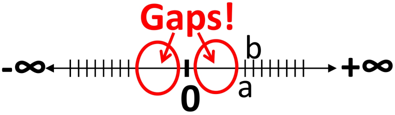
Representing Very Small Numbers

- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$
  - *Special Case*: Exponent and mantissa all zero $\Rightarrow 0$
  - Two zeros! But at least 0x0 $==$ 0 like in integers
- Numbers closest to 0:
  - $a = 1.0...0 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...1 \times 2^{-126} = 2^{-126} + 2^{-149}$



Representing Very Small Numbers

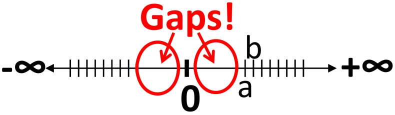
- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$
  - *Special Case*: Exponent and mantissa all zero $\Rightarrow 0$
  - Two zeros! But at least 0x0 $==$ 0 like in integers
- Numbers closest to 0:
  - $a = 1.0...0 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...1 \times 2^{-126} = 2^{-126} + 2^{-149}$



  - Normalization and implicit 1 are to blame

Representing Very Small Numbers

- So, uhhh, what about zero?
  - Using standard encoding 0x0 is $1.0 \times 2^{-127} \neq 0$
  - *Special Case*: Exponent and mantissa all zero $\Rightarrow 0$
  - Two zeros! But at least 0x0 == 0 like in integers
- Numbers closest to 0:
  - $a = 1.0...0 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...1 \times 2^{-126} = 2^{-126} + 2^{-149}$



  - Normalization and implicit 1 are to blame
  - *Special case:* Exponent $= 0 \Rightarrow$ *denormalized number*

Denorms

- Short for "denormalized numbers"
  - No leading 1
  - Implicit exponent is -126, NOT -127

Denorms

- Short for "denormalized numbers"
  - No leading 1
  - Implicit exponent is -126, NOT -127
- Now what do the gaps look like?
  - Smallest norm: $1.0...0 \times 2^{-126} = 2^{-126}$
  - Largest denorm: $0.1...1 \times 2^{-126} = 2^{-126} - 2^{-149}$
  - Smallest (pos) denorm: $1.0...1 \times 2^{-126} = 2^{-149}$

Denorms

- Short for “denormalized numbers”
  - No leading 1
  - Implicit exponent is -126, NOT -127
- Now what do the gaps look like?
  - Smallest norm: $1.0...0 \times 2^{-126} = 2^{-126}$
  - Largest denorm: $0.1...1 \times 2^{-126} = 2^{-126} - 2^{-149}$
  - Smallest (pos) denorm: $1.0...1 \times 2^{-126} = 2^{-149}$
- Notice: gap between smallest norm and largest denorm is small
  - So is the gap between 0 and the smallest denorm

Other Special Cases

- $\pm\infty$
  - Exponent = 0xFF, Mantissa = 0x0
  - e.g. division by 0
  - can be used in comparisons

Other Special Cases

- $\pm\infty$
  - Exponent = 0xFF, Mantissa = 0x0
  - e.g. division by 0
  - can be used in comparisons
- **NaN** (Not a Number)
  - Exponent = 0xFF, Mantissa $\neq$ 0
  - e.g. square root of negative number
  - NaN "contaminates" computations
  - Value of Mantissa can (theoretically be useful for debugging)
    - In practice a NaN is usually just a NaN

Other Special Cases

- $\pm\infty$
  - Exponent $=$ 0xFF, Mantissa $=$ 0x0
  - e.g. division by 0
  - can be used in comparisons
- **NaN** (Not a Number)
  - Exponent $=$ 0xFF, Mantissa $\neq 0$
  - e.g. square root of negative number
  - NaN "contaminates" computations
  - Value of Mantissa can (theoretically be useful for debugging)
    - In practice a NaN is usually just a NaN
- Largest finite value?
  - Exponent $=$ 0xFF is taken, 0xFE now has largest: $1.1...1_2 \times 2^{127} = 2^{128} - 2^{104}$

Float Encoding Summary

Exponent	Mantissa	Meaning
0	0	± 0
0	non-zero	$\pm$Denorm
1-254	anything	$\pm$Normalized
255	0	$\pm\infty$
255	non-zero	NaN

On the Topic of Free Lunches

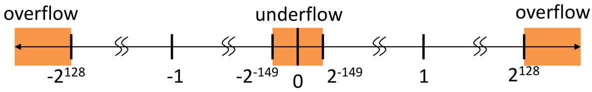
- ***There is no such thing***
  - All design decisions have tradeoffs
  - FP is no different
- Single precision IEEE floats only have 32 bits, same as a 32-bit signed int
  - Cannot represent more things
  - Can only change which things we decide to represent

Floating Point Limitations I

- What if result `x` is too large? (`abs(x)` $> 2^{128}$)
  - *Overflow*: Exponent is larger than can be represented
  - saturate to $\pm\infty$

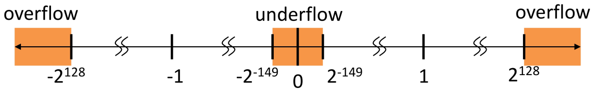
Floating Point Limitations I

- What if result `x` is too large? ($\texttt{abs(x)} > 2^{128}$)
    - *Overflow*: Exponent is larger than can be represented
    - saturate to $\pm\infty$
- What if result `x` is too small? ($\texttt{abs(x)} < 2^{-149}$)
    - *Underflow*: Negative exponent is larger than can be represented
    - saturate to 0



Floating Point Limitations I

- What if result `x` is too large? ($\texttt{abs(x)} > 2^{128}$)
  - *Overflow*: Exponent is larger than can be represented
  - saturate to $\pm\infty$
- What if result `x` is too small? ($\texttt{abs(x)} < 2^{-149}$)
  - *Underflow*: Negative exponent is larger than can be represented
  - saturate to 0



- What if the result runs off the end of the mantissa?
  - *Rounding* occurs and can lead to unexpected results
  - FP has different *rounding modes*. Most common is round-to-nearest.

Floating Point Limitations II

- Floating point arithmetic is NOT associative
  - You can find Big and Small numbers such that: Small + Big + Small $\neq$ Small + Small + Big
  - This is due to rounding errors: FP must *approximate* results because it only has 23 bits of mantissa
- Despite being seemingly "more accurate", FP cannot represent all integers
  - Must be careful when casting between `int` and `float`

Double Precision

- Encodes a floating point number in 64 bits

- Corresponds to the C type `double`
- Exponent bias of 1023
- Otherwise like single precision floats
- Much greater precision due to larger mantissa – generally preferred to `float`s in real computations for that reason

Question:
Let FP(1,2) = # of floats between 1 and 2
Let FP(2,3) = # of floats between 2 and 3

Which of the following statements is true?

(blue) FP(1,2) > FP(2,3)
(green) FP(1,2) = FP(2,3)
(purple) FP(1,2) < FP(2,3)
(yellow) It depends

Question:

Let FP(1,2) = # of floats between 1 and 2
Let FP(2,3) = # of floats between 2 and 3

Which of the following statements is true?

(blue) FP(1,2) > FP(2,3)
(green) FP(1,2) = FP(2,3)
(purple) FP(1,2) < FP(2,3)
(yellow) It depends

Question: Suppose we have the following floats in C:

$$\text{Big} = 2^{60},\ \text{Tiny} = 2^{-15},\ \text{BigNeg} = -2^{60},$$

What will the following conditionals evaluate to?

1. `(Big * Tiny) * BigNeg == (Big * BigNeg) * Tiny`
2. `(Big + Tiny) + BigNeg == (Big + BigNeg) + Tiny`

	1	2
(blue)	F	F
(green)	F	T
(purple)	T	F
(yellow)	T	T

Question: Suppose we have the following floats in C:

Big = 2^{60}, Tiny = 2^{-15}, BigNeg = -2^{60},

What will the following conditionals evaluate to?

1. `(Big * Tiny) * BigNeg == (Big * BigNeg) * Tiny`
2. `(Big + Tiny) + BigNeg == (Big + BigNeg) + Tiny`

	1	2
(blue)	F	F
(green)	F	T
(purple)	T	F
(yellow)	T	T

Technology Break

Outline

Defining CPU Performance

- What does it mean to say that X is faster than Y?
- Ferrari vs. School bus

  - 2009 Ferrari 599 GTB
    - 2 passengers, 11.1 second quarter mile
  - 2009 Type D school bus
    - 54 passengers, abysmal quarter mile time? `http://www.youtube.com/watch?v=KwyCoQuhUNA`

Defining CPU Performance

- What does it mean to say that X is faster than Y?
- Ferrari vs. School bus

  - 2009 Ferrari 599 GTB
    - 2 passengers, 11.1 second quarter mile
  - 2009 Type D school bus
    - 54 passengers, abysmal quarter mile time? http://www.youtube.com/watch?v=KwyCoQuhUNA
- Depends on whether we care about throughput or latency

Measurements of Performance

There are two metrics which are generally considered when measuring performance

- *Latency* (also *response time* or *execution time*)
  - Time to complete one task
- *Bandwidth* (or *throughput*)
  - Tasks completed per unit time

Cloud Performance: Why Latency Matters

Server Delay (ms)	Increased time to next click (ms)	Queries/ user	Any clicks/ user	User satisfac- tion	Revenue/ User
50	--	--	--	--	--
200	500	--	-0.3%	-0.4%	--
500	1200	--	-1.0%	-0.9%	-1.2%
1000	1900	-0.7%	-1.9%	-1.6%	-2.8%
2000	3100	-1.8%	-4.4%	-3.8%	-4.3%

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].

- Key figure of merit: application responsiveness
  - The longer the delay, the fewer the user clicks, the lower the user happiness, and the lower the revenue per user

Defining Relative Performance

- Compare performance of X vs. Y
  - Latency in this case

Defining Relative Performance

- Compare performance of X vs. Y
  - Latency in this case
- $\text{Perf}_X = \frac{1}{\text{Program Execution Time}_X}$

Defining Relative Performance

- Compare performance of X vs. Y
  - Latency in this case
- $\text{Perf}_X = \frac{1}{\text{Program Execution Time}_X}$
- $\text{Perf}_X > \text{Perf}_Y \Rightarrow \text{Execution Time}_X < \text{Execution Time}_Y$
- "Computer X is N times faster than Y"

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = N$$

Measuring CPU Performance

- Computers use a clock to determine when events take place within hardware
- *Clock cycles*: discrete quanta of computer execution
  - a.k.a. clocks, cycles, clock periods, clock ticks
- *Clock rate* or *clock frequency*: clock cycles per second
- Example: 3 GHz clock rate means a clock cycle time of $1/(3 \cdot 10^9)$ seconds $=$ 333 picoseconds

CPU Performance Factors

- Distinguish between time spent by the processor, and time waiting for I/O
  - *CPU time* is the time spent in the processor

$$\frac{\text{CPU Time}}{\text{Program}} = \frac{\text{Clock Cycles}}{\text{Program}} \times \text{Clock Cycle Time}$$
$$= \frac{\text{Clock Cycles}}{\text{Program}} \times \frac{1}{\text{Clock Rate}}$$

CPU Performance Factors

- But programs execute instruction!
  - Accounting for that we have

$$\frac{\text{CPU Time}}{\text{Program}} = \frac{\text{Clock Cycles}}{\text{Program}} \times \text{Clock Cycle Time}$$
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{1}{\text{Clock Rate}}$$

- Generally call $\frac{\text{Clock Cycles}}{\text{Instruction}}$ the *CPI* (Cycles Per Instruction) of a program

Components that Affect Performance

Component (HW/SW)	Factors Affected
Algorithm	Instruction Count, (CPI)
Programming Language	Instruction Count, CPI
Compiler	Instruction Count, CPI
Instruction Set Architecture	Instruction Count, CPI, Clock Rate

Question: Which statement is **TRUE**, given the following?

- Computer A clock cycle time 250ps, CPI = 2
- Computer B clock cycle time 500ps, CPI = 1.2
- Assume A and B have the same ISA

(blue) Computer A is ≈1.2 times faster than B
(green) Computer A is ≈4.0 times faster than B
(purple) Computer B is ≈1.7 times faster than A
(yellow) Computer B is ≈3.4 times faster than A

Question: Which statement is **TRUE**, given the following?

- Computer A clock cycle time 250ps, CPI = 2
- Computer B clock cycle time 500ps, CPI = 1.2
- Assume A and B have the same ISA

$$\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Period}$$

(blue) Computer A is ≈1.2 times faster than B
(green) Computer A is ≈4.0 times faster than B
(purple) Computer B is ≈1.7 times faster than A
(yellow) Computer B is ≈3.4 times faster than A

And In Conclusion

- Floating point approximates real numbers

  - Very high precision when representing small numbers
  - Very large range when representing large numbers
  - Encodings for 0, $\pm\infty$, NaN as well
- Performance measured in *latency* or *bandwidth*
- Latency measurement:
  - CPU Time $=$ Instructions $\times$ CPI $\times$ Clock Period
  - Affected by different components of the computer

Outline

Bonus Slides

We will likely not have time to cover these slides in lecture, but you are still responsible for the material presented within them. They have been put together in such a way as to be easily readable even without a live lecturer presenting them.

Casting floats to ints and vice versa

`(int) floating_point_expression`

Coerces and coverts it to the *nearest* integer, rounded toward zero (i.e. it truncates)

```
i = (int) (3.14159 * f);
```

`(float) integer_expression`

Converts integer to *nearest* floating point

```
f = f + (float) i;
```

float → int → float

```
if (i == (float)((int) i)) {
    printf("true");
}
```

- Will not always print “true”
  - Small floating point numbers (< 1) don’t have integer representations
- For other numbers, often will be rounding errors

int → float → int

```
if (f == (int)((float) f)) {
    printf("true");
}
```

- Will not always print "true"
  - Many large valued integers don't have exact floating point representations (recall: free lunches, and the ain't thereof)
- What about `double`?

int → float → int

```
if (f == (int)((float) f)) {
    printf("true");
}
```

- Will not always print "true"
  - Many large valued integers don't have exact floating point representations (recall: free lunches, and the ain't thereof)
- What about `double`?
  - Significand is now 52 bits, which can hold all of a 32-bit integer, so will always print "true" (assuming 32 bit `int`s)